The Graph's Apprentice: Teaching an LLM Low-Level Knowledge for Circuit Quality Estimation

Reza Moravej , Saurabh Bodhe, Zhanguang Zhang, Didier Chételat, Dimitrios Tsaras, Yingxue Zhang, Hui-Ling Zhen, Jianye Hao, Mingxuan Yuan

Huawei Noah's Ark Lab

reza.moravej@huawei.com

Abstract

Logic synthesis is a crucial phase in the circuit design process, responsible for transforming hardware description language (HDL) designs into optimized netlists. However, traditional logic synthesis methods are computationally intensive, restricting their iterative use in refining chip designs. Recent advancements in large language models (LLMs), particularly those fine-tuned on programming languages, present a promising alternative. This work proposes augmenting LLMs with predictor networks trained to estimate circuit quality directly from HDL code. To enhance performance, the model is regularized using embeddings from graph neural networks (GNNs) trained on Look-Up Table (LUT) graphs, thereby incorporating lower-level circuit insights. The proposed method demonstrates superior performance compared to existing graph-based RTL-level estimation techniques on the established benchmark OpenABCD, while providing instant feedback on HDL code quality.

1 Introduction

Rapid technological advancements in computing power has taken an increasingly important role in the past decades in driving scientific research in biology , chemistry , physics and especially artificial intelligence, where it has been estimated that at least half of all performance gains in the past ten years have stemmed from hardware improvements alone [Dorner, 2021; Karpathy, 2022; Erdil and Besiroglu, 2022]. This ever-rising demand for compute power means that efficient and effective electronic chip design has become increasingly critical.

Modern electronic chip design is a complex, multi-stage endeavor that begins with a chip architect specifying the digital circuit's functionality in a Hardware Description Language (HDL), such as Verilog [Thomas and Moorby, 2008] or VHDL [Coelho, 2012]. This HDL code is then subjected to a series of transformations and optimizations, ultimately yielding a physical circuit design that can be manufactured [LaMeres, 2023]. The quality of the resulting circuits are usually measured using physical characteristics only available in the later stages, such as circuit area or delay [Brayton
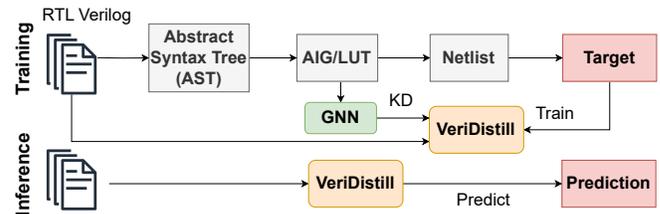

Figure 1: Overview of the training and inference pipeline. During training, LUT graphs, and the area/delay labels are used to train the model. During inference, only the source Verilog is required to generate the post-synthesis area/delay prediction.

and Mishchenko, 2010]. However, the computational cost of logic synthesis makes the iterative improvement according to the resulting circuit quality metrics prohibitively expensive. Optimizations are best made early on in this pipeline, ideally at the RTL level, so as to leave maximal flexibility in circuit design. Thus, efficient feedback methods that can estimate the quality of results of the HDL, can improve the resulting circuit and reduce the overall chip design time.

This discrepancy has led to interest in using artificial intelligence methods in the circuit design process [Huang *et al.*, 2021]. In this literature, machine learning models are trained to provide feedback on HDL code without running the actual logic synthesis process. This is done using supervised learning on a training set of circuits for which logic synthesis has been run and from which quality-of-result (QoR) metrics such as circuit area and delay have been computed. Although this approach seems straightforward, finding an representation of the RTL code appropriate for machine learning models has proven a challenge. The few works that have approached this topic did so by extracting graphical information about the code and using hand-designed statistics of those graphs as features [Zhou *et al.*, 2019; Sengupta *et al.*, 2022; Fang *et al.*, 2023]. Despite encouraging results, the performance of these methods has ultimately been limited by the relatively shallow understanding of the semantics of the code that these statistics can provide.

Recently, Large Language Models fine-tuned on code, such as Code-T5 [Wang *et al.*, 2021], Codex [Chen *et al.*, 2021], CodeGen [Nijkamp *et al.*, 2023], CodeLlama [Roziere *et al.*, 2023] and DeepSeek-Coder [Guo *et al.*, 2024], have proven

to be remarkably successful on a wide range of tasks [Zheng *et al.*, 2023], most notably as code assistants such as Github Copilot[1]. This raises the question as to whether their internal representations could be used as inputs to machine learning to predict circuit quality estimates.

In this work, we propose to feed Verilog code to the state-of-the-art Large Language Models, and train an inexpensive decoder neural network that uses the LLM's hidden states as features to predict area and delay. In addition, and critically, we regularize this decoder to encourage its embeddings to resemble those of a graph neural network model trained on Look-Up Table (LUT) graph, an intermediate representation used during the logic synthesis process. The resulting decoder is shown to strongly outperform state-of-the-art baselines in RTL-level circuit quality estimation, while keeping training and inference costs practical.

Our work makes the following main contributions:

1. We develop the first truly end-to-end machine learning model in the literature, named VeriDistill, which can take raw Verilog code, without any preprocessing, and produce accurate estimates of circuit area/delay metrics.

2. Moreover, we apply during training a novel knowledge distillation method which allows to transfer low-level insights about the circuit, in the form of LUT graphs, back into the machine learning predictor model.

3. We demonstrate through experiments that the combination of those two elements outperforms previous state-of-the-art baselines in a large-scale Verilog dataset and enhances the model's ability to transfer to out-of-distribution data.

4. Finally, we also demonstrate that both using LLM representations and the knowledge distillation are essential, in that removing any one of these components brings the performance back below the previous baselines.

The remainder of this paper is structured as follows. Section 2 provides an overview of the relevant literature and background information. In Section 3, we present a detailed description of our proposed methodology, including its key components and underlying assumptions. The efficacy of our approach is then demonstrated through a series of experiments, which are reported in Section 4. Finally, Section 5 summarizes our main findings, discusses their implications, and outlines potential avenues for future research.

2 Related Work

2.1 Quality-of-Result Prediction from HDL Code

Closest to ours is the work of [Sengupta *et al.*, 2022]. Their approach consists in computing the Abstract Syntax Tree (AST) induced by Verilog code, and extracting from this free vector- and graph-based features. They then train several machine learning models to predict from these features the total negative slack and dynamic power of the circuit. Among all the models evaluated, the XGBoost Regressor performs best and achieves 95% R2-score. The analysis was however limited to different runs of a single circuit and it is not clear how

the performance would generalize to different circuits. Since the Abstract Syntax Tree is essentially the raw Verilog code with extra syntactic information, which can be obtained at little cost at inference time by a grammar parser, we include it (along with variants) as baselines in our experimental section.

Further related is the work of [Fang *et al.*, 2023] and [Fang *et al.*, 2024b]. They propose to process Verilog code into a new representation called Simple Operator Graph (SOG), and test several machine learning models (Transformers, Random Forests, Graph Neural Networks and XGBoost regressors) to predict path delay, module-level power and combinatorial area. Although achieving promising results, computing the SOG requires expensive conversion of linguistic data into bit-level operators using logic synthesis tool Yosys [Wolf *et al.*, 2013], and a Verilog-to-graph parser, which is outside the scope of this work.

Finally, some works take a step further and try to assist circuit design by annotating which parts of HDL is most critical to achieve quality-of-result metrics. For example, [Sengupta *et al.*, 2023] attempts to identify timing critical components based on path delay prediction. The AST of each Verilog design is extracted and converted into a graph, with nodes representing IO ports, registers or behavior logic. Behavioral paths are extracted from the graph and used for path-level feature generation. Delay labels of timing paths are generated using commercial synthesis tools, and are assigned to corresponding behavior paths with the same start and end points. By training an XGBoost model on the resulting features, the authors achieve an average classification accuracy of 91%. Also similar is RTL-Timer [Fang *et al.*, 2024a], which ensembles four bit-level circuit representations to predict the post-logic synthesis endpoint arrival time. Such predictions can then be mapped to registers in HDL code to identify critical code paths. Just as in the work of [Fang *et al.*, 2023], however, these representations are bit-level rather than word-level and require pre-processing by logic synthesis tools like Yosys.

2.2 LLMs for Verilog

Large language models (LLMs) such as GPT [Ouyang *et al.*, 2022] and Llama [Touvron *et al.*, 2023] have achieved exceptional success in various natural language tasks and have expanded their success to programming languages as well. Although excellent on generalist programming languages like Python or C++, these models have been trained on the relatively small amount of HDL code that is publicly available on the internet, and therefore have performed poorly on Verilog benchmarks like VerilogEval [Liu *et al.*, 2023b] and RTLLM [Lu *et al.*, 2024]. This has motivated further work to build LLMs with a higher-degree of knowledge of hardware description languages. Both CodeGen-Verilog [Thakur *et al.*, 2023] and VeriGen [Thakur *et al.*, 2024] used a combination of customized Verilog datasets from code repository website GitHub[2] and various textbooks to fine-tune code LLMs. Finally, RTLCoder [Liu *et al.*, 2023c] used the GPT 3.5 language model [Brown *et al.*, 2020] to generate further Verilog data, in a form of data augmentation, while CodeV [Zhao *et al.*, 2024] used the same model to generate natural lan-

---

[1]https://github.com/features/copilot

[2]www.github.com

guage description of real world Verilog code through multi-level summarization.

Besides Verilog code generation from natural language description, LLMs were also explored for other EDA-related tasks. RTLFixer [Tsai *et al.*, 2023] employed Retrieval-Augmented Generation (RAG) and ReAct prompting techniques to interactively debug syntax errors in Verilog code, and achieved remarkable improvement in success rates in the VerilogEval benchmark. ChipNemo [Liu *et al.*, 2023a] explored the application of LLMs in chip design process and adopted several domain adaptation techniques to train an LLM for various applications including assistant chatbots, EDA script generation, and bug summarization and analysis. Finally, ChatEDA [Wu *et al.*, 2024] used code LLMs as an agent to autonomously complete the entire chip design flow from HDL code to the Graphic Data System Version II (GDSII) by managing task planning, script generation and task execution. We refer the reader to the extensive survey of [Zhong *et al.*, 2023] for more details on the application of LLMs in electronic design automation and future research directions in this field.

2.3 Alignment of LLM and GNN Embeddings

The multimodal alignment regularizer we propose during training also relates to the broader literature on tuning large language models to align with a pre-trained graph neural network, to incorporate its capabilities.

The work closest to ours is that of [Mavromatis *et al.*, 2023], who train a language model to perform a node classification task while adding a regularizer that encourages the predictive distributions to match a pre-trained graph neural network model. The language model makes predictions by passing the graph as input, and extracting the representation corresponding to a final [CLS] classification token. Also similar is [Zou *et al.*, 2023], which jointly trains a language model and a graph neural network on a common "context graph prediction" task which encourage alignment of their representations. They then discard the graph neural network and only keep the language model, so that topological characteristics best captured by graph convolutions can be said to have been incorporated in the language model.

More generally, there is a large literature on integrating pretrained graph neural networks with language models by training an adaptive module [Liu *et al.*, 2024; Liu *et al.*, 2023d; Chai *et al.*, 2023; Tang *et al.*, 2024; Cao *et al.*, 2023], allowing the language model to receive inputs from the graph neural network. Alternatively, multiple works have interlaced graph neural network layers and language model layers [Yasunaga *et al.*, 2021; Zhang *et al.*, 2022; Jin *et al.*, 2023]. In either case, some kind of training is necessary to allow for interactions between the graph neural network and the language model, although the result is not distillation of the graph neural network's perspective into the language model per se.

3 Methodology

We now present our VeriDistill approach in detail. As described in the introduction, turning a high-level description of a circuit in a Hardware Description Language like Verilog into a physical description ready for manufacturing is a computationally expensive process involving several steps, each with an associated intermediate representation describing progressively lower-level elements of the circuit. Our goal is to predict low-level quality-of-result metrics, like area and delay, from the high-level representation, namely the HDL code to allow for fast iterative improvement of the RTL design. Figure 1 provides and overview of logic synthesis as well as the training and inference pipelines.

3.1 Model

Our model takes as input Verilog code, which is fed to a Large Language Model (LLM). This LLM has been specifically fine-tuned on Verilog code generation. The code is first split into a sequence of tokens, which are then fed in parallel in the LLM. As an output, the LLM produces a sequence of high-dimensional "hidden state" vectors, one for each token that is inputted to the LLM. We average these hidden states, producing a single vector. This vector is then fed to a feedforward neural network, composed of several linear layers with nonlinear activations, which finally outputs the QoR estimate.

3.2 Training

We produce a training set of circuits with Verilog code for which the expensive logic synthesis process has been performed, so we know their QoR metric (such as area or delay). In addition, as an intermediate product of the logic synthesis process, an LUT graph is produced immediately following the logic optimization phase, which we save. This yields a collection of training triples $\mathcal{D} = \{(X_{\text{Verilog}}, X_{\text{LUT}}, y_{\text{QoR}})\}$.

Supervised learning: Given such a dataset, we treat our problem by supervised machine learning. The LLM, which has been pretrained on Verilog code, is kept frozen, so that only the FNN gets updated. In a training step, the Verilog code X_{Verilog} is fed to the VeriDistill model to produce a prediction $\hat{y}_{\text{QoR}}$. This prediction is compared in mean-squared error loss with the true QoR metric y_{QoR} as a supervised learning loss

$$\mathcal{L}_{\text{SL}} = \left(\hat{y}_{\text{QoR}} - y_{\text{QoR}}\right)^2. \quad (1)$$

Low-level knowledge distillation: In practice, training only with the supervised learning loss leads to limited performance. One potential explanation is that there is too much of a gap between a high-level circuit description like Verilog and the low-level metrics we purport to predict. Intuitively, to perform high-quality predictions, we would want the model to possess some degree of understanding of lower-level circuit design while still only taking Verilog code as input.

We propose the following approach to address this problem. Prior to training, we pretrain a Graph Neural Network (GNN) to predict the same QoR metric as VeriDistill, but from the Look-Up-Table (LUT) graph X_{LUT} of the circuit obtained after optimization using Yosys [Wolf *et al.*, 2013]. This graph, which can be seen as an alternative to the more popular And-Inverter Graph (AIG) format, is particularly suitable for GNN training as it is compact with rich node information. Moreover, as a circuit representation, it sits intermediate between a high-level description of the circuit encoded in the

Verilog code, and a physical circuit description. Prediction from LUT graphs is thus easier than prediction from Verilog code, but not completely trivial either.

The GNN architecture we adopt is composed of a sequence of graph convolutions, followed by joint mean and max pooling, and a sequence of linear layers. We pretrain it using the supervised learning loss (1) until good predictive performance is achieved. Then, during the VeriDistill training, we keep the GNN weights frozen and we propose to encourage the last-layer activations of the VeriDistill model $z_{\text{VeriDistill}}^{(-1)}$ to resemble those of the GNN model $z_{\text{GNN}}^{(-1)}$, despite these models operating on different inputs. We perform this simply by adding a mean-square error loss

$$\mathcal{L}_{\text{KD}} = \left\| z_{\text{VeriDistill}}^{(-1)} - z_{\text{GNN}}^{(-1)} \right\|_2^2 \tag{2}$$

in the total loss. As the weights of the GNN are pretrained and kept frozen while the VeriDistill model is being trained, this is effectively a form of knowledge distillation from the GNN to the VeriDistill model.

Total loss: We balance the importance given to the knowledge distillation compared to the supervised learning objective using a hyperparameter factor α, yielding the final loss

$$\mathcal{L} = \alpha \mathcal{L}_{\text{SL}} + (1 - \alpha) \mathcal{L}_{\text{KD}}.$$

A diagram describing the VeriDistill training process is provided as Figure 2.

4 Experiments

This section is organized as follows: We begin by presenting the implementation details of our experimental setup in Section 4.1, including hardware, model, and training hyperparameters. Next, we describe the dataset used and the data preprocessing steps for training and evaluation in Section 4.2. We then introduce the baseline methods and their implementation details in Section 4.3. Finally, we present the results on the main datasets and a study on unseen out-of-distribution circuits in Sections 4.4 and 4.5.

4.1 Experimental Setup

We use the following implementation of the model. We employ DeepSeek-Coder-V2-Lite-Base [DeepSeek-AI *et al.*, 2024] and CodeV-7B (based on CodeLlama) [Zhao *et al.*, 2024] as the Verilog LLM, and three layers with ReLU activations in the feedforward neural network. The model takes as input strings, which are broken into a sequence of tokens in the LLM's vocabulary. The language model processes these inputs into a sequence of the same length, made up of 512-dimensional vectors. After mean pooling, the resulting vector is passed to the feedforward neural network, which uses 512-dimensional activations, before making the final prediction. In particular, this architecture means that the last-layer activations $z_{\text{VeriDistill}}^{(-1)}$ are 512-dimensional. Results for other variants of CodeV-7B [Zhao *et al.*, 2024] as the Verilog LLM are included in the Appendix.

The auxiliary GNN teacher model takes a LUT graph with 16-dimensional node attributes, and passes it through three 64-dimensional graph convolutional layers interleaved
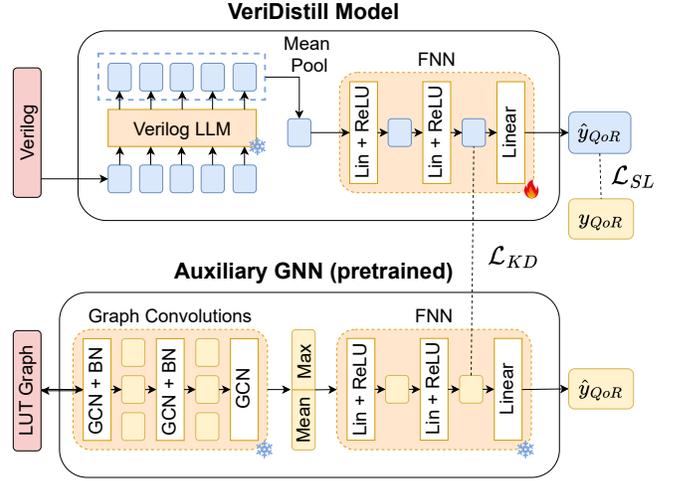

Figure 2: The training procedure. The Verilog training examples are passed to the VeriDistill model, which produces predictions of the QoR metric. These predictions are scored against the true QoR values by a mean-squared error supervised learning loss. In addition, the LUT graph representation resulting from logic optimization is fed to an auxiliary GNN model, pretrained to perform the same QoR prediction task. The hidden representations at the last layer of both the VeriDistill and GNN models is extracted, and a mean-square error knowledge distillation loss encourages these two representations to be similar, despite having different inputs. Both the pretrained GNN and LLMs modules are kept frozen during training.

with batch normalization layers. After concatenation of the mean and max pooling outputs, the 128-dimensional vector is passed through three 512-dimensional linear layers with ReLU activations before the final prediction. Thus, in particular, the last-layer activations $z_{\text{GNN}}^{(-1)}$ are 512-dimensional, matching with those of the VeriDistill model.

We implement VeriDistill and the baselines using the `PyTorch` and `PyG` libraries. Models which do not use our knowledge distillation procedure are trained using the ReduceLROnPlateau scheduler with initial learning rate 1e-3, patience set to 30 epochs and factor set to 0.5. In contrast, models involving our knowledge distillation procedure are trained using the CosineAnnealingLR [Loshchilov and Hutter, 2017] scheduler, with an initial learning rate of 1e-3 and number of iterations set to 50. We start the training process with $\alpha = 0.5$, and increase α to 0.75 and 1 at epochs 150 and 250. The idea is put less emphasis on knowledge distillation at every warm re-start. We find that this approach results in marginal gain compared to other optimization methods.

All models are trained until full convergence. Since the LLM is kept frozen during training, it was possible to save training time by extracting the forward pass through the LLM only once and saving it. We performed this phase on a machine with 8 Nvidia V100 GPUs with 32GB of memory and 32 Intel(R) Xeon(R) Gold 6140 CPUs. Once the hidden states are saved, we then trained each model following the procedure detailed in the paper on the same machine using a single V100 GPU with 1024 minibatch sizes. The training times for each model are summarized in the Appendix.

method	Area				Delay			
	MAE ↓	R2 ↑	MAPE ↓	RSE ↓	MAE ↓	R2 ↑	MAPE ↓	RSE ↓
LUT-GNN (Teacher)	0.251	0.955	0.309	0.045	0.109	0.948	0.023	0.052
AST-XGBoost	1.497	0.255	71.205	2.899	0.480	0.280	0.108	2.564
AST-GNN	0.893	0.661	1.435	0.339	0.317	0.604	0.071	0.396
AST-GNN w/ KD	0.892	0.666	1.647	0.334	0.315	0.619	0.071	0.381
DeepSeek + Decoder	1.119	0.548	2.004	0.452	0.401	0.478	0.094	0.522
CodeV + Decoder	0.991	0.629	1.69	0.371	0.367	0.533	0.086	0.467
VeriDistill (DeepSeek)	0.497	0.867	0.863	0.133	0.23	0.793	0.053	0.207
VeriDistill (CodeV)	0.482	0.872	0.784	0.128	0.236	0.781	0.054	0.219

Table 1: The performance of different Verilog models on the test dataset, where the best result for each metric is bolded. In addition, we report the performance of the teacher model trained on the LUT graphs, which serves as an upper-bound.

4.2 Datasets

We train and evaluate on two separate datasets. The first dataset is used for training, validation, and testing of all the methods, while OpenABCD contains out-of-distribution circuits aiming to challenge VeriDistill and determine its ability to generalize.

Customized Dataset To train and evaluate our proposed method, we collect 18.4k Verilog examples provided by [Pei *et al.*, 2024] and 5.8k from [Thakur *et al.*, 2022]. These Verilog examples are obtained from open-source GitHub repositories and textbooks and have been verified for syntax correctness. We use an open-sourced EDA platform Open-ROAD [Ajayi *et al.*, 2019] with 7nm technology PDK provided to conduct logic synthesis and record post-synthesis labels of area and delay. We convert the AIG graphs obtained after logic optimization into LUT graphs and save them for training the auxiliary GNN model.

Note that a substantial fraction of the code snippets end up being functionally incorrect and failing some stage of the logic synthesis pipeline. Since we require functionally correct examples for their QoR metric to be well-defined, we removed such examples during the preprocessing. In addition, although not strictly a problem for our method, one of the competing baselines requires extracting the Abstract Syntax Tree (AST) of the Verilog, which is obtained by running a parser on the code. The parser was unable to produce AST representations for a small fraction of the instances (FRACTION%), which we removed from consideration. The resulting dataset, after filtering bad examples, ended up having 16k examples, which we split into training, validation, and test sets with a ratio of 0.75/0.1/0.15, respectively. Details about the dataset and label distributions can be found in the Appendix.

We note that OpenROAD provides two optimization recipes for the logic synthesis process: "ABC_AREA=1" for area optimization and "ABC_SPEED=1" for timing optimization. The results reported under Section 4.4 are produced under the speed optimization. We report the results under area optimization in the Appendix. We find that our approach works as well under different recipe optimization settings.

OpenABCD Additionally, we consider data provided by [Chowdhury *et al.*, 2021] to evaluate the transferability of

our method to unseen circuits. The OpenABCD dataset consists of functionally diverse designs such as bus communication protocols, computing processors, digital signal processing cores, cryptographic accelerators and system controllers.

4.3 Baselines

While many prior works have attempted to predict post-synthesis circuit quality at the RTL-stage, none of them perform prediction directly from source Verilog files. Several works rely on lower-level circuit representation that requires extra processing using logic synthesis tools [Zhou *et al.*, 2019; Fang *et al.*, 2023]. Utilizing low-level circuit representations as input is advantageous for the circuit quality prediction. However, in some cases, obtaining the low-level description can be prohibitively expensive. In addition, obtaining the low-level description relies on external processing tools which are prone to errors. As such, we compare our method to approaches which take the raw Verilog or the AST representation of the circuit.

We adopt the method proposed by [Sengupta *et al.*, 2022] as our baseline. It relies on AST representations that can be easily converted from Verilog source files. We implement the method based on description in [Sengupta *et al.*, 2022]. Verilator [Snyder, 2004] is used to convert each source Verilog into its respective AST representation, which can be represented as a graph. The nodes in the graph represent one of the following five semantic categories from the source Verilog (`root`, `variable`, `operation`, `constant`, `edge`), while edges are created between nodes with connections. We implement three variants of the AST-based method:

AST-XGBoost We compute the following features: (i) the total number of input bits, (ii) the total number of output bits, (iii) the longest path in the AST, (iv) the frequency of each node type in the graph and (v) the frequency of each logic type in the graph. The features are concatenated to form a feature vector with 108 features [3]. We perform a thorough hyper-parameter selection using grid search and employ early stopping to prevent over-fitting.

AST-GNN w/o KD The AST-GNN model takes in the following features per node: (i) the total number of input bits,

---

[3] $108 = 1 + 1 + 1 + 5 + 100$ features coming from feature categories $(i)....,(v)$
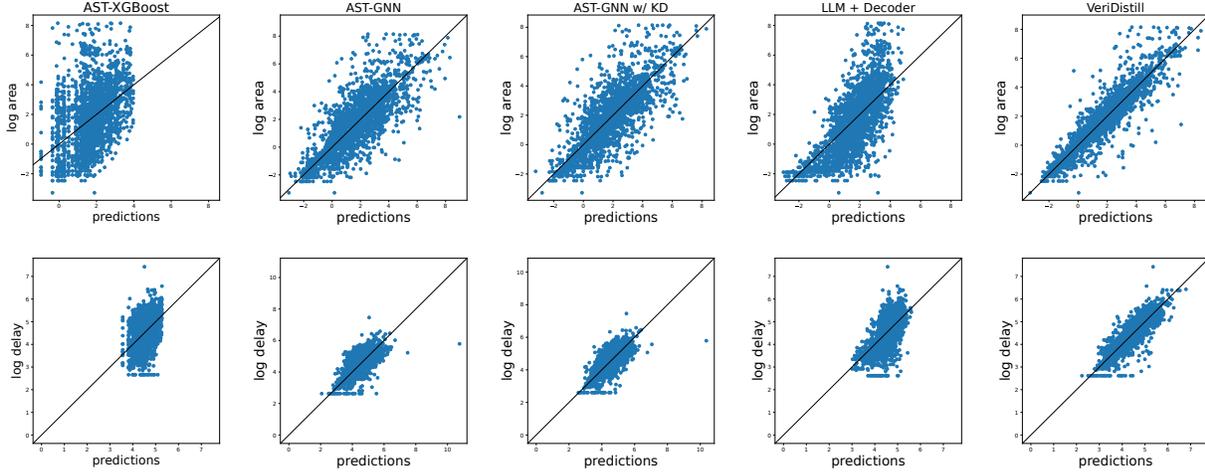
Figure 3: Prediction vs. target on test data, where DeepSeek-V2-Lite is utilized as the LLM. The predicted values using different methods are plotted against the targets. (Top) Area prediction. (Bottom) Delay prediction.

(ii) the total number of output bits, (iii) the node semantic type and (iv) the node operation type. Each feature is represented via a one-hot vector and is projected to a 4-dimensional space via a linear layer. The final node features consist of a (4×4) = 16-dimensional vector. We cap the number of input/output bits to 200, since 99.9 percent of the nodes in the dataset have less than 200 input/outputs. The AST-GNN model utilizes the same hyperparameters and architecture as the auxiliary GNN model used for the knowledge distillation objective in VeriDistill.

AST-GNN w/ KD   We propose a third baseline, where the AST-GNN model is guided by the LUT GNN model. The baseline utilizes the same student-teacher knowledge distillation as our method. We introduce this baseline to demonstrate the effectiveness of utilizing an LLM in the student network.

4.4   Main results

We first summarize the results of our main experiment, where we train and test the model on the large Customized Dataset (see Section 4.2). Table 1 outlines the performance of different models on the test set. As can be seen, our proposed method, utilizing both the LLM as an encoder and knowledge distillation, outperforms other baselines across all the metrics, especially with area prediction. Interestingly, simply using a decoder on the LLM representation performs worse than the previous state-of-the-art, while knowledge distillation on the AST-GNN model has almost no effect. Only when both are used together is there substantial impact on performance, which suggests our knowledge distillation procedure is crucial in fully exploiting the richness of the LLM representations.

The CodeV + Decoder model greatly outperforms the DeepSeek-V2-Lite + Decoder model. We hypothesize this performance gap is due to the fact that CodeV representations are more aligned with Verilog semantics. While DeepSeek-V2-Lite is pre-trained on a variety of programming languages, the CodeV is specially fine-tuned on Verilog data after being pre-trained on various programming languages.

However, the performance gap between the two VeriDistill models is much smaller. This results hints that using knowledge distillation can, to a high degree, mitigate the lack of additional fine-tuning of the base LLM on Verilog. The result hints that VeriDistill can achieve similar results when applied on top of various code LLMs without the need for further fine-tuning the base LLM on Verilog data.

IP	IO	Nodes	Edges	Lines	Tokens
aes	1212	28925	58379	1406	21305
aes_secwork	5691	40778	84160	2443	28630
aes_xcrypt	3780	45840	93485	985	16308
des3_area	367	4971	10006	2545	44650
dft	75014	245046	527509	4637	59292
dyna_node	5283	18094	38763	6251	63011
ethernet	21153	67164	144750	10841	131015
fir	761	4558	9467	307	2892
fpu	1041	29623	59655	1910	27060
i2c	305	1169	2466	1246	13690
idft	75022	241552	520523	4638	59356
iir	935	6978	14397	395	3870
mem_ctrl	2149	16307	37146	5880	70632
pci	6586	19547	42251	22692	306937
sasc	260	613	1351	597	5783
sha256	2985	15816	32647	1054	10551
simple_spi	296	930	1992	463	5010
spi	492	4219	8676	794	10348
ss_pcm	194	462	896	223	2173
tv80	997	11328	23017	4736	56461
usb_phy	222	487	1064	1102	10317
vga_lcd	34385	105334	227731	5078	54555
wb_conmax	4197	47840	97755	7108	108718

Table 2: OpenABCD circuit statistics. IO, Node and Edges are the number of primary inputs/outputs, AIG nodes and AIG edges of the circuits. Lines and Tokens refer to the number of lines and tokens in the Verilog RTL file.

	Area			Delay		
IP	GT	AE (wo/ KD)	AE (w/ KD)	GT	AE (wo/ KD)	AE (w/ KD)
aes	7.3	2.909	**2.746**	5.886	0.442	**0.212**
aes_secworks	7.629	3.763	**0.631**	6.33	1.089	**0.494**
aes_xcrypt	8.013	3.742	**3.319**	6.363	0.949	**0.713**
des3_area	5.772	1.971	**1.068**	6.05	0.639	**0.22**
dft	9.671	5.813	**4.004**	5.557	0.267	**0.167**
dynamic_node	7.046	**3.379**	4.965	5.986	**0.937**	1.245
ethernet	8.464	5.134	**4.395**	5.883	0.801	**0.525**
fir	5.04	2.12	**0.829**	5.765	0.83	**0.527**
fpu	7.092	4.077	**0.977**	7.714	2.682	**1.574**
i2c	4.193	2.038	**1.006**	5.412	0.692	**0.157**
idft	9.668	5.731	**4.002**	5.537	0.208	**0.14**
iir	5.508	2.297	**0.983**	5.768	0.736	**0.572**
mem_ctrl	6.366	2.597	**0.748**	6.155	1.067	**0.585**
pci	7.124	3.179	**3.162**	5.765	**0.574**	0.594
sasc	3.795	0.971	**0.583**	5.17	**0.229**	0.427
sha256	6.594	2.259	**0.043**	5.587	0.305	**0.194**
simple_spi	4.07	2.151	**0.985**	5.263	0.636	**0.173**
spi	5.363	2.321	**1.968**	5.82	0.896	**0.734**
ss_pcm	3.309	0.78	**0.192**	4.89	**0.106**	0.234
tv80	6.235	3.126	**1.259**	6.178	1.097	**0.793**
usb_phy	3.412	0.359	**0.172**	4.92	**0.05**	0.14
vga_lcd	8.883	5.7	**4.984**	5.778	0.811	**0.724**
wb_conmax	7.689	**3.253**	4.665	5.866	**0.832**	1.521
Mean Value		3.029	**2.073**		0.734	**0.551**

Table 3: The Absolute Error (AE) on OpenABCD circuits. VeriDistill with or without KD have been trained on customized datasets and used to predict post-synthesis area and delay of OpenABCD circuits without any finetuning. Due to the large length of the Verilogs, we utilize DeepSeek-V2-Lite with a context window of 128k tokens.

We gain further insight on the benefits of our approach by analyzing scatter plots of the predictions against the targets. As can be seen in Figure 3, the baseline models perform well primarily on circuits with small delay and area but struggle with larger circuits, likely due to their lower representation in the training set. In contrast, our model achieves consistently strong performance across circuits of all sizes. This contrast is particularly pronounced when comparing against the same model without knowledge distillation (LLM+Decoder), which indicates that our knowledge distillation procedure is crucial in allowing our model to perform well across the whole range of circuit sizes. The impact of knowledge distillation on the AST model is relatively minimal compared to its effect on the LLM-based model. This can be attributed to the enhanced alignment between the teacher representations and the LLM when used as the encoder. A visualization of the t-SNE projection of the final hidden space representations is provided in the Appendix to verify the above claim.

4.5 Additional Out-of-Distribution Results

Finally, we evaluate how our knowledge-distillation procedure can impact the ability of the trained model to generalize to new out-of-distribution circuits. For this, we take our model, trained with and without knowledge distillation on our Customized Dataset, and apply it to instances in the Open-ABCD benchmark (see Section 4.2). We outline the Open-ABCD circuit statistics in the Table 2. Due to the scarcity of

large circuit Verilog data, the circuits from the OpenABCD benchmark are larger than the majority of the circuits present in our dataset (that is, the Verilog files contains more lines and the circuits have generally larger area and delay). As can be seen in Table 2, our knowledge distillation procedure systematically improves the LLM-based model's ability to transfer prediction performance on out-of-distribution instances, which differ significantly from those seen during training. A comparison between the performance of VeriDistill with the AST based approaches on the OpenABCD benchmark can be found in the Appendix. where VeriDistill outperforms the baselines on the well majority of the circuits.

5 Conclusion

In this work, we propose a novel procedure to predict quality-of-result electronic circuit metrics from Verilog code, by training a small neural network model on Verilog LLM representations with a knowledge distillation regularizer which align its internal activations with those of a low-level GNN model. We show that this new model outperforms previous approaches in predicting the post synthesis QoR labels of a circuit. Beyond the potential of our method for future practical applications, our results underscore the value of the information encoded in the LLM's representations for predicting circuit quality. Additionally, they highlight the crucial role of our knowledge distillation procedure in enabling downstream models to effectively leverage this information.

References

[Ajayi *et al.*, 2019] T Ajayi, D Blaauw, TB Chan, CK Cheng, VA Chhabria, DK Choo, M Coltella, S Dobre, R Dreslinski, M Fogaça, et al. Openroad: Toward a self-driving, open-source digital layout implementation tool chain. In *GOMACTECH*, 2019.

[Brayton and Mishchenko, 2010] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pages 24–40. Springer, 2010.

[Brown *et al.*, 2020] Tom B. Brown, Benjamin Mann, and etc. Language models are few-shot learners. In *NeurIPS*, 2020.

[Cao *et al.*, 2023] He Cao, Zijing Liu, Xingyu Lu, Yuan Yao, and Yu Li. Instructmol: Multi-modal integration for building a versatile and reliable molecular assistant in drug discovery. *arXiv*, 2023.

[Chai *et al.*, 2023] Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiao Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. Graphllm: Boosting graph reasoning ability of large language model. *arXiv*, 2023.

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv*, 2021.

[Chowdhury *et al.*, 2021] Animesh Basak Chowdhury, Benjamin Tan, Ramesh Karri, and Siddharth Garg. Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis. *arXiv*, 2021.

[Coelho, 2012] David R Coelho. *The VHDL handbook*. Springer Science & Business Media, 2012.

[DeepSeek-AI *et al.*, 2024] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024.

[Dorner, 2021] Florian E Dorner. Measuring progress in deep reinforcement learning sample efficiency. *arXiv*, 2021.

[Erdil and Besiroglu, 2022] Ege Erdil and Tamay Besiroglu. Algorithmic progress in computer vision. *arXiv*, 2022.

[Fang *et al.*, 2023] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. Masterrtl: A pre-synthesis ppa estimation framework for any rtl design. In *ICCAD*, 2023.

[Fang *et al.*, 2024a] Wenji Fang, Shang Liu, Hongce Zhang, and Zhiyao Xie. Annotating slack directly on your verilog: Fine-grained rtl timing evaluation for early optimization. In *DAC*, 2024.

[Fang *et al.*, 2024b] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. Transferable pre-synthesis ppa estimation for rtl designs with data augmentation techniques. *TCAD*, 2024.

[Guo *et al.*, 2024] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv*, 2024.

[Huang *et al.*, 2021] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. Machine learning for electronic design automation: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(5):1–46, 2021.

[Jin *et al.*, 2023] Bowen Jin, Wentao Zhang, Yu Zhang, Yu Meng, Xinyang Zhang, Qi Zhu, and Jiawei Han. Patton: Language model pretraining on text-rich networks. In *Proceedings of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023.

[Karpathy, 2022] Andrej Karpathy. Deep neural nets: 33 years ago and 33 years from now, 2022.

[LaMeres, 2023] Brock J LaMeres. *Introduction to logic circuits & logic design with VHDL*. Springer Nature, 2023.

[Liu *et al.*, 2023a] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. Chipnemo: Domain-adapted llms for chip design. *arXiv*, 2023.

[Liu *et al.*, 2023b] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *ICCAD*, 2023.

[Liu *et al.*, 2023c] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. RTLCoder: Outperforming RTL-3.5 in design RTL generation with our open-source dataset and lightweight solution. *arXiv*, 2023.

[Liu *et al.*, 2023d] Zhiyuan Liu, Sihang Li, Yanchen Luo, Hao Fei, Yixin Cao, Kenji Kawaguchi, Xiang Wang, and Tat-Seng Chua. MolCA: Molecular graph-language modeling with cross-modal projector and uni-modal adapter. In *EMNLP*, 2023.

[Liu *et al.*, 2024] Pengfei Liu, Yiming Ren, Jun Tao, and Zhixiang Ren. Git-mol: A multi-modal large language model for molecular science with graph, image, and text. *Computers in biology and medicine*, 2024.

[Loshchilov and Hutter, 2017] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. In *ICLR*, 2017.

[Lu *et al.*, 2024] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *ASP-DAC*, 2024.

[Mavromatis *et al.*, 2023] Costas Mavromatis, Vassilis N Ioannidis, Shen Wang, Da Zheng, Soji Adeshina, Jun Ma, Han Zhao, Christos Faloutsos, and George Karypis. Train your own gnn teacher: Graph-aware distillation on textual graphs. In *ECML PKDD*, 2023.

[Nijkamp *et al.*, 2023] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *ICLR*, 2023.

[Ouyang *et al.*, 2022] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *NeurIPS*, 2022.

[Pei *et al.*, 2024] Zehua Pei, Huiling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. BetterV: Controlled verilog generation with discriminative guidance. In *ICLR*, 2024.

[Roziere *et al.*, 2023] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv*, 2023.

[Sengupta *et al.*, 2022] Prianka Sengupta, Aakash Tyagi, Yiran Chen, and Jiang Hu. How good is your verilog rtl code? a quick answer from machine learning. In *ICCAD*, 2022.

[Sengupta *et al.*, 2023] Prianka Sengupta, Aakash Tyagi, Yiran Chen, and Jiang Hu. Early identification of timing critical rtl components using ml based path delay prediction. In *MLCAD*, 2023.

[Snyder, 2004] Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, volume 79, 2004.

[Tang *et al.*, 2024] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. Graphgpt: Graph instruction tuning for large language models. In *SIGIR*, 2024.

[Thakur *et al.*, 2022] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond A. Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. *DATE*, 2022.

[Thakur *et al.*, 2023] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *DATE*, 2023.

[Thakur *et al.*, 2024] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *TCAD*, 2024.

[Thomas and Moorby, 2008] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[Touvron *et al.*, 2023] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv*, 2023.

[Tsai *et al.*, 2023] YunDa Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv*, 2023.

[Wang *et al.*, 2021] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021.

[Wolf *et al.*, 2013] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Austrochip*, 2013.

[Wu *et al.*, 2024] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. Chateda: A large language model powered autonomous agent for eda. *TCAD*, 2024.

[Yasunaga *et al.*, 2021] Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. QA-GNN: Reasoning with language models and knowledge graphs for question answering. In *NAACL*, 2021.

[Zhang *et al.*, 2022] X Zhang, A Bosselut, M Yasunaga, H Ren, P Liang, C Manning, and J Leskovec. Greaselm: Graph reasoning enhanced language models for question answering. In *ICLR*, 2022.

[Zhao *et al.*, 2024] Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, et al. Codev: Empowering llms for verilog generation through multi-level summarization. *arXiv*, 2024.

[Zheng *et al.*, 2023] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv*, 2023.

[Zhong *et al.*, 2023] Ruizhe Zhong, Xingbo Du, Shixiong Kai, Zhentao Tang, Siyuan Xu, Hui-Ling Zhen, Jianye Hao, Qiang Xu, Mingxuan Yuan, and Junchi Yan. c. *arXiv*, 2023.

[Zhou *et al.*, 2019] Yuan Zhou, Haoxing Ren, Yanqing Zhang, Ben Keller, Brucek Khailany, and Zhiru Zhang. Primal: Power inference using machine learning. In *DAC*, 2019.

[Zou *et al.*, 2023] Tao Zou, Le Yu, Yifei Huang, Leilei Sun, and Bowen Du. Pretraining language models with text-attributed heterogeneous graphs. In *EMNLP*, 2023.