Accurate, Efficient and Scalable Power Modeling for FPGA HLS

Sen Yan, *Student Member, IEEE,* Kuangxun Huang, *Student Member, IEEE,* Kang Zhao, *Member, IEEE,* and Zhe Lin, *Member, IEEE*

***Abstract*—Power estimation serves as the foundation for a series of hardware optimization strategies. However, it is challenging to deliver accurate power estimation at an early design stage like high-level synthesis (HLS). In this paper, we propose *PowerGear*, a graph-learning-assisted power estimation approach for FPGA HLS, which features high accuracy, efficiency, scalability, transferability, and applicability. PowerGear comprises two main components: a graph construction flow and a customized graph neural network (GNN) model. In the graph construction flow, PowerGear introduces buffer insertion, datapath merging, graph trimming and feature annotation techniques to transform HLS designs into graph-structured data, which encode both intra-operation micro-architectures and inter-operation interconnects annotated with switching activities. Additionally, PowerGear incorporates cross-function graph integration, dataflow adaptation, and design scaling strategies that make it compatible with multi-kernel, computation-intensive and large-scale applications. Furthermore, we propose a novel power-aware heterogeneous edge-centric GNN model, which effectively learns heterogeneous edge semantics and structural properties of the constructed graphs via edge-centric neighborhood aggregation, and eventually fits the formulation of dynamic power. Compared with on-board measurement, PowerGear shows average errors of 3.60% and 8.81% in total and dynamic power estimation, respectively. It also supports large-scale and dataflow-optimized neural networks with an error of only 3.91%, demonstrating broad applicability. Moreover, PowerGear achieves a 9.1× speedup over direct measurement, and up to 41.9× when combined with the design scaling strategy. Finally, PowerGear facilitates design space exploration for FPGA HLS, yielding a performance gain of up to 11.2% compared with state-of-the-art predictive models.**

***Index Terms*—High-level synthesis (HLS), field-programmable gate array (FPGA), graph neural network (GNN), power modeling**

I. Introduction

Power efficiency has emerged as one of the first-order constraints for hardware systems such as field-programmable gate arrays (FPGAs), and the design optimization with regard to power efficiency usually necessitates the knowledge of power consumption [1], [2]. However, the power evaluation flow for FPGA designs induces large overheads of design turnaround time. In general, accurate FPGA power estimation requires obtaining the signal activities of critical components and I/O ports via vector-based gate-level simulation, and a set of physical component measurements obtained through the register-transfer level (RTL)-based FPGA implementation flow, including synthesis, placement and routing, etc. With the low-level hardware details disclosed after conducting the above steps, analytical models [3] can be used to deduce signal activities for internal components and then infer power consumption. Therein, the cycle-accurate gate-level simulation and the FPGA implementation flow with NP-complete problems gives rise to a large amount of runtime. Overall, in a power-oriented hardware optimization loop, designers have to repeatedly perform the above power evaluation steps while refining the design architecture until power closure is achieved, which incurs long development time and high labor cost for every single design.

In the research field, great efforts have been made to speed up hardware power estimation. As for RTL level, some works [4], [5] propose to collect a small set of signal activities through RTL simulation and then use them as input features to construct learning-based models for power inference, including decision trees, ensemble models, and convolutional neural networks. These works expedite the power estimation process by replacing the inefficient FPGA implementation flow with efficient modeling strategies. Other works such as [6] seek to build models to predict gate-level internal signal activities using I/O and register activities from RTL simulation, and leverage the commercial power analysis tools to conduct power estimation after physical implementation, avoiding the tedious vector-based gate-level simulation. These works showcase gains in runtime efficiency by skipping either the RTL-based implementation flow or gate-level simulation, while the remaining steps are still time-consuming.

Another branch of studies [7]–[9] aims at achieving total power estimation at a higher abstraction level, i.e., high-level synthesis (HLS) [10], which greatly accelerates power estimation by dispensing with the low-level simulation and implementation steps in power inference. Some works [7], [8] extract activity features during C-level program execution, and either develop analytical power models for specific types of functions [7] or utilize machine learning models [8] to learn power characteristics of downstream FPGA implementation. These works are design-specific and the built models are not applicable to new designs of interest. A recent work HL-Pow [9] adopts histograms as a way of feature alignment over

This work was supported in part by the National Natural Science Foundation of China (Grant No. 62404257); in part by the Shenzhen Science and Technology Program (Grant No. RCBS20231211090600003); and in part by the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2023A1515110769, 2024A1515013155). *(Corresponding author: Zhe Lin.)*

Sen Yan, Kuangxun Huang, and Zhe Lin are with the School of Integrated Circuits, Sun Yat-sen University, Shenzhen Campus, China (e-mail: yans6@mail2.sysu.edu.cn; kaungxun55@mail2.sysu.edu.cn; linzh235@mail.sysu.edu.cn).

Kang Zhao is with the School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China (e-mail: zhaokang@bupt.edu.cn).

different designs, which allows power inference for unseen cases. This is accomplished by encoding the activities of each type of HLS operations into a histogram individually, concatenating histograms as overall design features, and then training models to infer power.

Nevertheless, these prior arts on HLS power estimation mainly carry out feature engineering on individual operations, neglecting the impact of interconnects between different operations and the switching activities associated with interconnects, even though interconnects have been proven to significantly contribute to power dissipation [11], especially dynamic power. In order to improve early-stage power estimation for FPGA HLS, we argue that it is vital to take into account the impact of interconnects in power modeling. To this end, we leverage graph neural networks (GNNs) in an effort to jointly learn micro-architectures of HLS operations and interconnects with switching activities.

Specifically, we present a graph construction flow to transform the HLS-based hardware designs into graph-structured data, wherein operations are cast as nodes with features indicative of micro-architectures, interconnects are projected as edges with relation types, and netlist activities are encoded as edge features. On this basis, we devise a novel heterogeneous edge-centric GNN model for power modeling, called *HEC-GNN*. Different from general-purpose GNNs that primarily work on node features, HEC-GNN is enhanced with the capability to exploit informative heterogeneous edge semantics and structural properties via the novel edge-centric aggregation scheme, and is aware of power via adaptively approximating the formation of dynamic power.

Putting it all together, we propose a graph-learning-assisted power modeling approach for FPGA HLS, named *PowerGear*, which distinguishes itself from the prior arts with the following key features: (1) **accuracy**: PowerGear demonstrates high accuracy for estimating dynamic power besides total power, which has not been jointly studied in prior related works [7]–[9] on HLS; (2) **efficiency**: compared with RTL-based works [4]–[6] and Vivado integrated power estimator [12], PowerGear gains considerable improvement in the turnaround time of power modeling; (3) **scalability**: PowerGear streamlines power prediction for designs of different scales by adapting the graph from one scale to another, thus supporting light-weight power estimation across multiple variants of the same application; (4) **transferability**: in contrast to the task-dependent modeling frameworks [4], [5], [7], [8], PowerGear can generalize to previously unseen designs without requiring model retraining; and (5) **applicability**: PowerGear is capable of modeling compound, large-scale, and dataflow-optimized designs that incorporate function calls and hierarchical structures, broadening its applicability to realistic scenarios.

To the best of our knowledge, this paper presents the first attempt to apply customized GNNs for early-stage power estimation in HLS with both total and dynamic power predicted accurately. In all, our major contributions are listed as follows:

- We introduce a graph construction flow to convert the FPGA HLS designs into graph data that preserve both intra-operation and inter-operation power-related features.
- We propose cross-function graph integration and adaptation to dataflow optimization to support multi-kernel and computation-intensive HLS designs, enhancing the modeling capability to accommodate a wide range of scenarios.
- We propose a design scaling strategy that directly derives large-scale graph representations from their corresponding small-scale counterparts without rerunning electronic design automation (EDA) tools, thereby achieving high scalability.
- We develop a novel power-aware GNN model, HEC-GNN, which fully mines rich heterogeneous edge semantics and structural properties via the edge-centric neighborhood aggregation, and subtly fits the dynamic power formulation.
- We demonstrate how our power estimation approach benefits power-efficient design space exploration in HLS.

II. Related Work

Fast evaluation of the hardware designs' quality of results (QoR) at an early design stage can tremendously streamline the convergence process in the hardware optimization loop. As a result, the research topic of early-stage QoR estimation has attracted considerable research interests in recent years [13]–[16]. Among various QoR metrics, performance, power and area (PPA) are the three most influential design metrics for hardware designs. Regarding PPA, reliable estimation of power consumption is the most challenging due to its dependence on post-routing netlist information and signal toggling traces. Generally speaking, pre-RTL power estimation approaches can be broadly classified into two categories, i.e., traditional methods and machine learning (ML)-based methods.

A. Traditional Method for Pre-RTL Power Estimation

Traditional methods for pre-RTL power estimation tend to perform power characterization under the guidance of human expertise, while taking into account the high-level design micro-architectures and signal switching activities observed from system-level simulations. LOPASS [17] delves into the detailed power consumption of various FPGA logic components and interconnects, and applies architectural knowledge to guide the development of analytical models for overall power. McPAT [18] adopts a similar idea but it constructs a power model purely for the multicore and manycore systems. Sumit Ahuja *et al.* [19] conduct simulation with the SystemC models, extracts activities of SystemC variables, and generates the mapping of variable activities to RTL signals, after which existing RTL-based power estimators can be exploited to profile power. Aladdin [20] integrates an offline power library consisting of the power characteristics of various individual functional units, which is built by applying a series of microbenchmarks to exercise different input patterns, and finally performs functional unit decomposition, power lookup and aggregation online. Wei Zuo *et al.* [7] specialize in affine functions that can be decomposed into multiple identical tiles. Instead of collecting power in the complete design level, this work reduces the modeling complexity by only characterizing

the power behaviors of a single tile per design, and infers overall power consumption with an analytical model that accumulates power from tiles. FlexCL [21] introduces a power estimation approach by taking into account the mechanisms of operation scheduling, work-items and work-groups specific to the OpenCL programming model. In general, the above traditional methods are dedicated to certain types of micro-architectures, programming models or design instances, which indicates limited applicability.

B. Machine Learning Method for Pre-RTL Power Estimation

The rapid advancements in the field of artificial intelligence have opened up new avenues for pre-RTL power estimation, which leverages large-scale datasets to feed into sophisticated algorithms in an effort to establish the relationships between design architectures, signal activities and power consumption.

Starting with the RTL design, APPOLO [22] and Simmani [23] propose to collect a limited subset of RTL signals that can reflect the power changes of the complete design, and then utilize the activities of these selected signals to build linear or polynomial regression models in order to infer single-cycle or multi-cycle power consumption. Similarly, PRIMAL [24] captures signal activities from registers, maps the registers to two-dimensional pixels to indicate spatial relationships between signals, and finally applies convolutional neural networks (CNNs) for power learning. While these methods can achieve up to 92% power prediction accuracy, they require building a dedicated regressor per RTL design from scratch, incurring significant timing overhead during the modeling process, i.e., from sample collection, power model construction, signal selection, to tracer instrumentation, and they also suffer from low generalization ability. To overcome these challenges, MasterRTL [25] transforms the RTL code into a customized bit-level representation consisting of only five types of operators, and predicts module-level power consumption using tree-based models. Although the developed model is transferrable, it still suffers from limited prediction accuracy, inducing an error of up to 38%.

To go one step further, some recent works [8], [9] raise the abstraction level from RTL to C/C++ when performing power prediction. Dongwook Lee *et al.* [8] extracts signal activities according to HLS scheduling and binding, which are used as features to develop linear regression model for power synthesis. While this method exhibits high accuracy, the pre-built power models are application-specific and can not be generalized to unseen designs. HL-Pow [9] directly operates on HLS metadata, based on which it develops transferrable power estimators using ML techniques without the need for digging into RTL details. Specifically, HL-Pow extracts signal activities from the intermediate representation (IR) of the HLS front-end compiler, constructs activity histograms for each type of operators to capture their activity distribution, and learns power characteristics using either linear regression, tree-based ensemble or CNN models. HL-Pow can effectively predict the total power consumption, whereas it is still not able to distinguish static and dynamic power consumption, the two fundamental sources of power consumption.

C. Summary

Early-stage power estimation methodologies for hardware designs encompass a wide range of techniques, from traditional analytical methods to advanced ML-based approaches. Traditional methods [7], [17]–[21] require human expertise to guide the benchmark and stimulus generation, power library establishment or analytical model formulation. ML-based approaches [8], [9], [22]–[25] are mostly dedicated to a specific design or are not precise enough to perform static and dynamic power decomposition. For the first time, in this paper, we present a pre-RTL power modeling method based on HLS, which is able to accurately predict both static and dynamic power consumption.

Compared to our preliminary work [26], this extended paper makes substantial enhancements in accuracy, efficiency, scalability, and applicability. First, we introduce a design scaling mechanism that enables direct conversion of graphs for designs at a specified scale into graphs at other scales, yielding a 6.6× speedup over the work [26] at the cost of negligible degradation in accuracy. Second, we develop an adaptive strategy for dataflow optimization that fully reflects the concurrent nature of dataflow, reducing the dynamic power estimation error for dataflow designs from 11.24% to 3.91%. Third, we augment the graph construction flow with a cross-function integration scheme, allowing PowerGear to model multi-kernel designs. Finally, we implement a complicated CNN design as a case study to showcase PowerGear's comprehensive capabilities in supporting compound, dataflow-optimized and large-scale application scenarios.

III. Background

A. FPGA Power Decomposition

The FPGA power consumption can be decomposed into two key components: static power and dynamic power. The static power mainly stems from the leakage currents passing through the transistors whenever the devices are powered up, independently of the workloads running in real time. As for earlier FPGA series, the static power is constant regardless of the implemented designs. However, recent FPGA products, such as Xilinx Ultrascale+ FPGA, have enabled automatic power gating [27] on unused hard blocks such as logic units and memories, making static power dependent on design scale, resource utilization, and the types of resources used.

The dynamic power consumption, on the other hand, is caused by signal switching activities, i.e., transistor switches or register value changes that repeatedly charge and discharge interconnect capacitance. In contrast to static power, dynamic power reflects the runtime workloads driven by specific data patterns and data characteristics. The overall dynamic power consumption is the sum of power consumption for each single netlist component triggering signal toggling, which can be formulated as

$$P_{dyn} = \sum_{i \in I} \alpha_i C_i V_{dd}^2 f, \tag{1}$$

where α is the signal switching activity, C is the interconnect capacitance, V_{dd} is the supply voltage, f is the operating frequency and i is an interconnect of the whole set I. For a

specific FPGA, the supply voltage V_{dd} is fixed, and C for each i is decided by the FPGA placement and routing algorithms. Hence, given a target design on a specific FPGA, dynamic power is largely determined by runtime workloads that give rise to different signal switching activities and operating frequencies.

B. Graph-Structured Data and Graph Neural Network

Graph-structured data, such as social media networks and molecules, are a form of general data representation that are widely found in real life. Graphs have two types of fundamental components, namely, nodes and edges. A node represents an object or entity while an edge connecting two nodes indicates a type of relationship between these nodes. A graph G can be defined as $G = (V, E)$, where V denotes the set of nodes in G, and likewise, E represents the set of edges in G. Unlike images that organize pixels into regular grids, graphs have variable node sizes, and most importantly, each node in a graph can have a different number of connected edges. Graph data analysis is of paramount significance as it makes it possible to profile intricate graph patterns, discover hidden structures, and understand the dynamics of graphs.

GNNs have recently emerged as a cornerstone in analyzing graph data. The core idea behind GNN is to project each node to a d-dimensional vector space (so-called embedding), let each node aggregate the information from neighbor nodes via the *message passing mechanism* and then update itself. This flow is iterated as the GNN is deepened by adding more layers. GNNs can perform different types of graph analysis tasks, including node-level or graph-level classification or regression, and link prediction.

IV. PowerGear Methodology

In PowerGear, we investigate an automated design flow to capture the power-related information in the HLS designs, generate graph-structured data samples encapsulating both individual and contextual information of various types of design components, and develop a GNN-based power estimation model. The overview of PowerGear is shown in Fig. 1, which comprises three design flows.

In the power inference flow, we construct the power graphs solely based on HLS results, i.e., the intermediate representation (IR) from HLS front-end compilation, and the finite state machine with datapath (FSMD) from HLS back-end optimization. These graph data are used for both model training and inference. To conduct power inference, new designs of interest only need to pass through the HLS flow, be converted into power graphs, and be fed into the trained GNN model to infer power directly, skipping the time-consuming RTL implementation process.

In the power training flow, we collect ground truth power values of each design point via real measurement on FPGA after performing the RTL implementation flow. The power values and the graph-structured features constructed by the power inference flow are bundled into samples for modeling training. Besides this, we develop a novel directed, heterogeneous and edge-centric GNN model to effectively learn the principles behind FPGA power consumption.

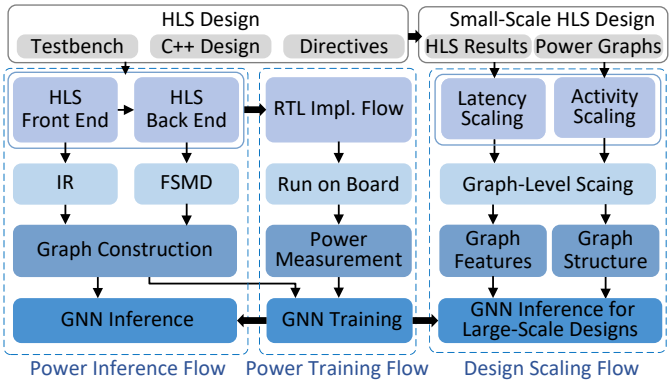


Fig. 1: PowerGear overview.

In the design scaling flow, we enable fast power graph construction for designs of varying scales by extrapolating from small-scale HLS designs. Specifically, when scaling up the design size of a reference design in the database, we propose to inherit the pre-built graph structure of the reference design, but update the node and edge features via adaptively scaling latency and switching activities to align with the target design scale. Furthermore, this design scaling mechanism can also benefit new designs of interest by first generating power graphs for their smaller-scale counterparts and then converting these graphs to represent the corresponding large-scale designs. In all, this scaling flow enables us to quickly generate the power graphs for designs under various design scales without repeatedly invoking EDA tools, thereby achieving efficient power estimation for large-scale designs.

V. Graph Construction Flow

In the graph construction flow, we propose to generate graph-structured samples for FPGA HLS designs in order to make use of GNNs for power learning. HLS tool flows usually provide IR and FSMD information to recover dataflow graphs (DFGs) [28]. Indeed, the DFGs have already presented graph-like data: each DFG node is associated with an IR operation which defines its micro-architecture, and the interconnect between two operations forms a DFG edge. Directly using the HLS DFGs for power modeling, however, is problematic, because primitive DFGs tend to be large-scale but convey limited power-related details, harming both model efficiency and efficacy. To tackle this problem, we introduce five optimization strategies on the HLS DFGs to retrieve and retain from the DFG nodes the hardware components that have significant influence on power consumption, and augment the interconnects, i.e., DFG edges, with power information. Note that these graph transformations are designed to capture and reflect power behaviors, rather than to synthesize actual circuits. Specifically, the function-level graph construction techniques, including buffer insertion, datapath merging, graph trimming and feature annotation, are described in Fig. 2 (a) and (b), whereas the cross-function integration technique is explained in Fig. 2 (c).

Buffer insertion. In the DFGs, buffer components, including internal and I/O buffers, have not been declared explicitly.

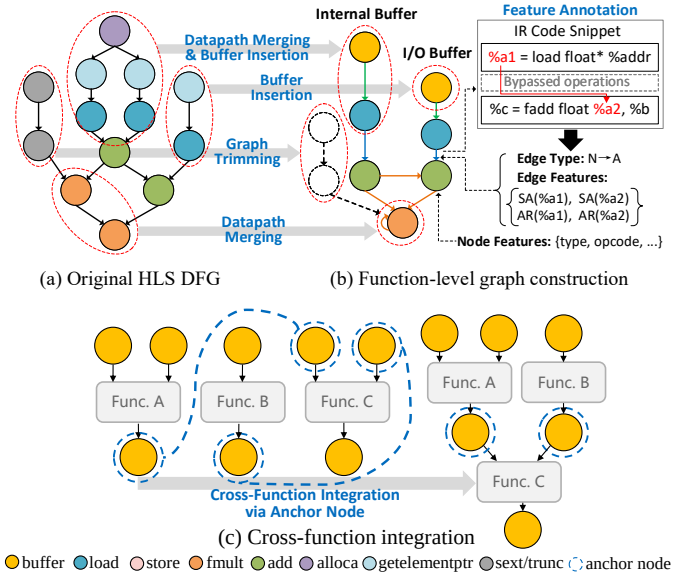


Fig. 2: Graph construction flow with HLS DFG optimization.

Nevertheless, memory activity is a critical source of power consumption. Hence, we seek to instrument buffers in the DFGs. We observe that memory elements can be inferred from DFG nodes with IR opcodes *alloca* and *getelementptr* followed by *load* or *store*. Specifically, we perform pattern matching of the above memory-related DFG nodes, insert buffer nodes in DFGs, connect buffer nodes to downstream nodes, and annotate buffer nodes with memory resource utilization.

Datapath merging. The DFGs may contain plenty of identical node chains from a source node to a sink node. For instance, loading data from a specific buffer and storing back to another one in different loop execution may cause different IR operations to be instantiated, even though these operations are highly similar. This leads to multiple duplicate DFG node chains generated between two DFG nodes, which deviates from the exact hardware realization. To restore the real hardware implementation from the DFGs, we perform datapath merging to fuse identical DFG node chains between two nodes. Furthermore, we try to account for resource sharing between different FSM stages in RTL, and to this end, we merge the DFG nodes utilizing the same set of hardware resources.

Graph trimming. The DFGs encompass paths that are not computationally intensive and thus contribute little to power consumption. Accordingly, we remove DFG nodes whose operation types fall into the following two categories: (1) control-flow nodes that have no direct correspondence to hardware units, including *br*, *switch*, and *call*; and (2) light-weight datapath nodes that have negligible impact on architecture, including *bitconcatenate*, *zext*, *sext*, *getelementptr*, *partselect*, *bitcast*, *bitselect*, *trunc*, and *urem*. This helps to improve the modeling efficiency by reducing the scale of the generated graph samples, and suppress noise by making the model focus more on crucial arithmetic-intensive datapaths.

Feature annotation. The DFGs do not reflect any signal toggling intrinsically. As the signal activity is the trigger of dynamic power consumption, we propose to integrate signal activities in interconnects, i.e., edges of graph samples for power learning. First, we annotate the signals across each edge of interest. That is, we identify the dataflow variables (operands or results of IR operations) transferred through the edges from the source nodes to the sink nodes. Then, we instrument detection probes to trace the values of variables of interest at the IR level. Next, we link together the testbenches with input stimuli, the instrumented IR and the function entities of detection probes following the method of [9]. Subsequently, we compile them into a single executable, and run the executable to acquire traced variable values in different execution cycles. Finally, we compute edge switching activities SA_{edge} by considering the injected and outgoing data on each edge individually. This gives

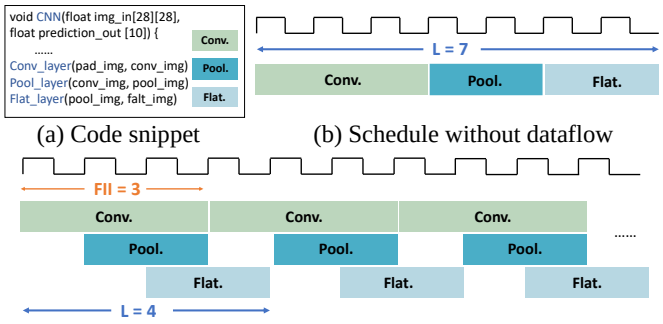
$$SA_{edge} = \left\{ \frac{\sum_{i=1}^{N_{\boldsymbol{v}}^{dir}} \mathrm{HD}\left(\boldsymbol{v}_{dir}(i), \boldsymbol{v}_{dir}(i-1)\right)}{L} \;\middle|\; dir \in \{src, snk\} \right\}, \tag{2}$$

where $\boldsymbol{v}$ is the set of bit vectors of variable values that pass through the edge and change at different execution cycles, dir is the dataflow direction that separates $\boldsymbol{v}$ produced by source nodes (src) from $\boldsymbol{v}$ utilized by sink nodes (snk) of the edge, $N_{\boldsymbol{v}}^{dir}$ is the number of execution cycles that cause the change of $\boldsymbol{v}$ with dir, L is the latency of the design, and $\mathrm{HD}(\cdot)$ is the Hamming distance computation which counts the number of variable bits that are different in two execution cycles. Besides SA_{edge}, we also extract the activation rate for each edge, which is defined as

$$AR_{edge} = \left\{ \frac{N_{\boldsymbol{v}}^{dir}}{L} \;\middle|\; dir \in \{src, snk\} \right\}. \tag{3}$$

In this way, we construct four-dimensional *edge features* comprising the switching activities and the activation rates of both the source and sink variables through the edge. We further classify the DFG nodes into two categories: arithmetic (A) and non-arithmetic (N) nodes. Correspondingly, the edges are annotated with four types of source-to-sink relations: A→N, A→A, N→A, and N→N. As for *node features*, we use the IR operation type, opcode, latency, resource utilization (LUT, DSP, BRAM), in-degree, out-degree, activation rate, input, output and total switching activities. The operation type and opcode are categorical features with one-hot encoding while the others are numeric features.

Cross-function integration. The aforementioned strategies are designed to generate a power graph that represents a single computational kernel encapsulated as the top function within HLS. However, as the design complexity increases, an entire design typically consists of multiple computational kernels. To accommodate our graph construction flow for the growing needs of large-scale and sophisticated scenarios, we introduce a non-trivial technique called cross-function integration to fuse multiple individual function-level graphs into an overall graph structure while maintaining the topological correctness. We notice that communication between functions typically involves data transmission via intermediate buffers such as random access memory (RAM) and first in first out (FIFO) memory, with control signals sent via handshake protocols to orchestrate the overall execution timeline. Hence, the key



(a) Code snippet (b) Schedule without dataflow

(c) Schedule with dataflow

Fig. 3: Dataflow optimization in HLS.

idea for function integration is to identify intermediate nodes shared between functions as *anchor nodes*. Note that we have already inserted the buffer entities for each function after buffer insertion. In this stage, we further extract the input and output buffers of each function as anchor nodes. Subsequently, we merge power graphs of different functions by linking the same anchor nodes among them, as illustrated in Fig. 2 (c). In addition, it is worth noting that cross-function resource sharing can occur. To address this, we capture resource reuse relationships across functions by matching identical hardware entities. Building on this foundation, we enhance the datapath merging strategy to fuse the shared nodes among functions. This strategy enables us to construct an integrated graph for the entire design while maintaining function-level dependencies, facilitating comprehensive power modeling for complex and large-scale designs.

VI. Adaptation to Dataflow Application

Dataflow optimization is one of the most important and widely used strategies in HLS. It is effective for applications that contain multiple functions with sequential execution characteristics, as shown in Fig. 3 (a) with an example of CNN for illustration. When the design is fed into HLS without applying dataflow, the three functions in Fig. 3 (a) execute in serial according to the readiness of their input data, as shown in Fig. 3 (b). This design paradigm may incur large overall execution latency (L) because each function only runs in a small portion of time over the whole invocation period, rather than being fully utilized throughout the entire execution. In contrast, Fig. 3 (c) explains the mechanism of dataflow optimization, in which the three kernel functions can overlap in their execution and finally lead to the reduction of L. To be more specific, applying dataflow allows multiple functions or loops with dependencies to run in parallel, with each of them starts processing a different set of input data after an interval marked as *FII* in Fig. 3 (c). In this way, the idle time of each functional unit is greatly reduced since new data can be streamed into the hardware without the need to wait for the previous rounds of data processing to complete.

The improvement of dataflow optimization mainly stems from the concurrent processing of different sets of input data. Note that in the feature annotation stage as illustrated in Section V, we use L to normalize the switching activities, as shown in Eq. 2. This is feasible in non-dataflow situations. Nevertheless, when dataflow is deployed, it is no longer suitable to use L to standardize activities due to the concurrency property of tasks: multiple rounds of task invocation overlap in certain timing periods, causing L to be imprecise when describing a single invocation. In light of this issue, we propose the concept of *effective latency (EL)* to characterize the amortized behaviors of a task with dataflow optimization. Formally, *EL* is defined as the average execution cycles of a task invocation over N invocations, which is given as

$$EL = \frac{L + FII \times (N-1)}{N}, \tag{4}$$

where *FII* denotes the function-level initiation interval between the start of two invocations. From Eq. 4, we can see that *EL* is related to the number of invocations. In a situation that the tasks are steadily processed, the number of invocations tends to be infinite, we discover that *EL* can be reduced to *FII*, as shown in

$$EL = \lim_{N\to\infty} \frac{L + FII \times (N-1)}{N} = \lim_{N\to\infty} \left(\frac{L}{N} + FII \times \left(1 - \frac{1}{N}\right)\right) = FII. \tag{5}$$

Following this, the switching activity of the dataflow design, SA_{edge_df}, can be reformulated as

$$SA_{edge\_df} = \left\{ \frac{\sum_{i=1}^{N_{\boldsymbol{v}}^{dir}} \mathrm{HD}\left(\boldsymbol{v}_{dir}(i), \boldsymbol{v}_{dir}(i-1)\right)}{EL} \;\middle|\; dir \in \{src, snk\} \right\}. \tag{6}$$

As a result, by employing *EL* as the exact latency for activity normalization, a more precise representation of the switching activities in dataflow-optimized designs is derived. Without loss of generality, Eq. 6 is applicable to non-dataflow designs by defining *FII* as L in non-dataflow scenarios.

VII. Enhancement on Scalability

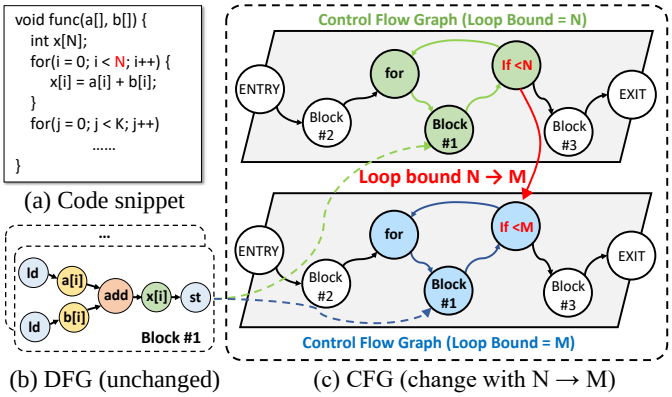
A. Problem and Observation

Problem formulation. Noting that HLS runtime increases significantly with design scale, we introduce a scalability-augmented power prediction mechanism, as illustrated by the design scaling flow of Fig. 1. This mechanism offers two key advantages: (1) it utilizes the pre-built HLS database during training to facilitate graph construction for existing designs at larger scales; and (2) for a new large-scale design of interest, it reduces HLS runtime overhead by performing HLS only once on a smaller-scale counterpart, and directly extrapolating the results of this design to the original large-scale configuration. To start with, we present the definition of design scaling and describe the problem we try to tackle.

Definition 1 (Design Scaling): Given a reference HLS design, design scaling refers to the practice of increasing the amount of data to process in a single invocation of the reference design through scaling up the loop bound of loops and the size of arrays used within the loops.

Problem 1 (Graph Construction with Design Scaling): Given a HLS design with a specified design scale, construct the graph for power prediction when the design scale is enlarged.

Observation. We propose an efficient solution to address *Problem 1*, based on two key observations from HLS compilation. **Our first observation is that the DFG topology remains intact for the same application at different scales**. The front-end process of HLS first decomposes the complete



(a) Code snippet (b) DFG (unchanged) (c) CFG (change with N → M)

Fig. 4: Illustration of DFG and CFG at different design scales.

program into multiple basic blocks (BBs), then constructs data and control flow [29] of the target design, and finally converts them into DFG and control flow graph (CFG), respectively. As shown in Fig. 4 (b), a DFG describes the arithmetic and logic operations within BBs and the dependencies between operations. The designs at different scales share the same computational patterns and data dependencies within loop bodies, which are well captured and described by the DFG at an arbitrary design scale.

Our second observation is that the invocations and activities of each operation are dependent on the CFG, which change as the design scale varies. Different from a DFG, a CFG depicts the execution paths and branch conditions (such as loop entry or exit) among BBs, as illustrated in Fig. 4 (c). When the design scale changes, the criteria for entering and exiting BBs in the CFG are modified accordingly. This affects the number of times each BB in DFG is executed, subsequently impacting the activity features of each node and edge in the constructed graphs.

Based on the above problem formulation and observations, it is feasible to preserve the pre-built graph topology from one design scale to another, while only modifying node and edge activities to account for the changes in CFG after design scaling. To this end, we propose two novel techniques to efficiently transform the constructed graph for a reference design at a specified scale to another scale without going through the entire graph construction flow and any EDA flow. These two techniques are (1) *analytical latency scaling*, and (2) *adaptive activity scaling*.

B. Analytical Latency Scaling

The latency is used as a key factor to normalize the switching activities, as shown in Eq. 6. In light of this, we develop an analytical model that uses a reference design as the starting point to infer execution latency at different design scales. We place our emphasis on loop-level latency modeling. This is because loops in C/C++ programs usually carry out a large number of computation-intensive operations throughout multiple iterations, which makes them the primary source of latency. Using HLS, loops can be transformed with HLS directives such as *loop pipelining* and *loop unrolling*. It is important to note that each of these directives exerts a distinct influence on execution latency. Although some research works [16], [30] have proposed prediction models to generalize the mechanisms by which different directives affect latency, few of them have investigated the scenarios under design scaling.

Without loss of generality, we elaborate our analytical model based on a two-level nested loop comprising an *inner loop* and an *outer loop*, while more complicated loops can be hierarchically decomposed into outer-loop and inner-loop structures that fit into our case. Under this context, we investigate the latency models responsible for the inner loop and outer loop with design scaling, respectively.

Scaling the inner loop. We start with developing the latency model for the inner loop by taking into account the effect of design scaling. Indeed, an inner loop is just a single loop structure. For such a case, the loop latency can be modeled based on whether loop pipelining is exerted. Specifically, if the loop is non-pipelined, all the loop iterations are executed in serial, and hence, the total latency of the loop is the sum of the single iteration latency of all iterations. In contrast, loop pipelining allows different loop iterations to operate simultaneously with a small interval between two consecutive iterations. In general, the latency after design scaling, L_I^S, can be formulated as

$$L_I^S = \begin{cases} IL_I + II_I \times (T_I^S - 1) & \text{if pipelined;} \\ IL_I \times T_I^S & \text{if not pipelined;} \end{cases} \tag{7}$$

where IL_I is the iteration latency of the reference loop, II_I is the initiation interval (II) of the reference loop, i.e., the number of cycles elapsed between the issuing of two consecutive iterations, and T_I^S is the trip count after design scaling. Based on the above observations, it is worth noting that the iteration latency IL_I and initiation interval II_I are unaffected by design scaling while the trip count T_I^S should be updated as follows:

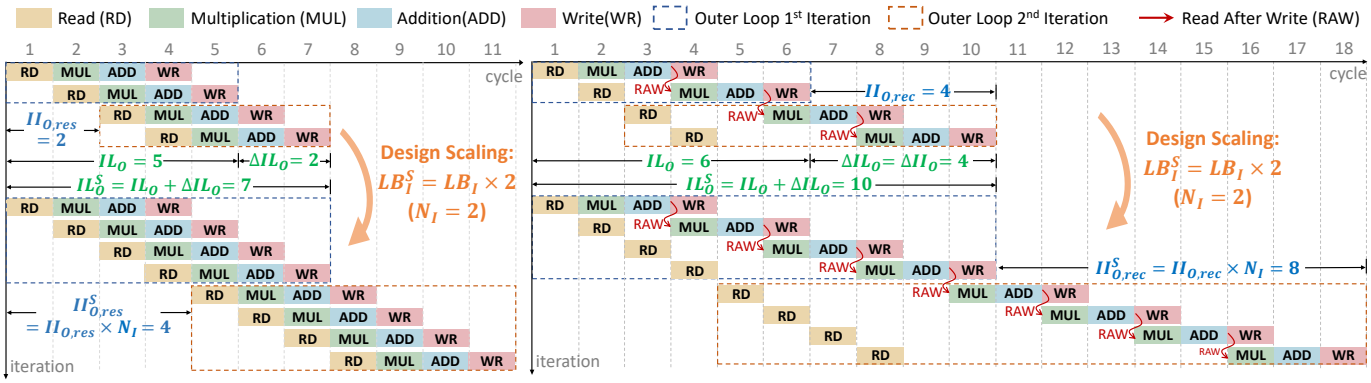
$$T_I^S = \frac{LB_I}{U_I} \times N_I, \tag{8}$$

where LB_I is the loop bound of the reference loop, U_I is the factor of *loop unrolling* applied to the inner loop, and N_I is the scaling factor that indicates the ratio by which the design size of the inner loop is amplified.

Scaling the outer loop. C/C++ programs may incorporate complicated nested loop structures. In such cases, we can start by examining the characteristics of the two innermost levels of the loop, and then regard these two levels to be a flattened loop structure within a deeper nested loop. Hence, we describe the approach to model the behavior of the outer loop of a two-level nested loop. In the meanwhile, we note that when loop pipelining is applied to the outer loop, the inner loop is completely unrolled [30]. Hence, we decompose the analytical model to account for non-pipelining and pipelining separately.

1) Non-pipelining: A non-pipelined outer loop executes the outer iterations serially. The latency per outer iteration comes from three sources: (1) the execution of the entire inner loop after scaling, L_I^S, as shown in Eq. 7; (2) the additional operations outside the inner loop but within the outer loop, L_{OP}; and (3) the extra overhead to move from the outer loop to the inner loop, L_M. Note that L_{OP} and L_M are consistent across different scales.

As the design size of the outer loop increases by a scaling factor of N_O, the number of iterations to execute in the outer



(a) Resource-constrained outer-loop pipeline

(b) Recurrence-constrained outer-loop pipeline

Fig. 5: Illustration of changes in latency under design scaling in both resource-constrained and recurrence-constrained pipelined outer loops.

loop will also increase proportionally. Putting it all together, the latency of the non-pipelined outer loop with design scaling can be represented as

$$L_O^S = (L_I^S + L_{OP} + L_M) \times T_O^S, \tag{9}$$

where T_O^S denotes the trip count of the outer loop after scaling. T_O^S can be deduced from Eq. 8, by substituting LB_I, U_I and N_I with the loop bound of the reference outer loop, LB_O, the unrolling factor applied to the outer loop, U_O, and the scaling factor of the outer loop, N_O, respectively, which can be formulated as

$$T_O^S = \frac{LB_O}{U_O} \times N_O. \tag{10}$$

2) Pipelining: When the outer loop is pipelined, the inner loop is fully unrolled [30], thereby flattening the hierarchies between the outer loop and the inner loop. In other words, all operations in the inner loop are scheduled concurrently into the pipeline of the outer loop. In this context, the execution latency may change dramatically, as opportunities for operation-level parallelism arise. In general, the overall latency of the pipelined outer loop with design scaling can be formulated as

$$L_O^S = IL_O^S + II_O^S \times (T_O^S - 1). \tag{11}$$

Herein, IL_O^S denotes the latency of a single iteration of the outer loop after scaling (also known as the pipeline fill time), II_O^S is the scaled initiation interval, and T_O^S is the scaled total number of outer loop iterations, as shown in Eq. 10. We now detail the derivation of II_O^S and IL_O^S, respectively.

Firstly, the initiation interval after design scaling, II_O^S, is determined by two constraints: (1) *resource constraint*, which arises when the number of memory ports fail to support the number of simultaneous data accesses; and (2) *recurrence constraint*, which occurs when operations in different iterations have read-after-write (RAW) dependencies. Using the single port memory as an example, the scaled resource-constrained initiation interval, $II_{O,res}^S$, can be represented as

$$II_{O,res}^S = \max_m \left(\left\lceil \frac{\#Access_m^S}{\#Port_m} \right\rceil \right) \approx II_{O,res} \times N_I, \tag{12}$$

where $\#Access_m^S$ and $\#Port_m$ denote the number of memory access operations in an iteration of the scaled outer loop, and the number of memory ports for array m which remains unchanged after scaling. With design scaling, the number of memory accesses similarly scales by a factor of N_I, causing $II_{O,res}$ to scale proportionally, as illustrated in Fig. 5 (a). Likewise, if the loop is dominated by recurrence constraint, as depicted in Fig. 5 (b), we can deduce the scaled recurrence-constrained initiation interval $II_{O,rec}^S$ as

$$II_{O,rec}^S = II_{O,rec} \times N_I. \tag{13}$$

Based on the above discussion, we can find that if the inner loop is scaled by a factor of N_I, both $II_{O,res}^S$ and $II_{O,rec}^S$ increase by a factor of N_I simultaneously. Eventually, the II_O^S can be formulated as

$$\begin{aligned} II_O^S &= \max\left(II_{O,res}^S, II_{O,rec}^S\right) \\ &= N_I \times \max\left(II_{O,res}, II_{O,rec}\right) \\ &= N_I \times II_O. \end{aligned} \tag{14}$$

Secondly, the scaled iteration latency of the outer loop, IL_O^S, is mainly determined by the total execution latency of the entire inner loop after fully unrolled, plus the latency of intermediate operations between the inner and the outer loops. The original IL_O has already accounted for the effect of inner-loop unrolling and intermediate operations. However, we also need to add the extra execution latency induced by scaling the inner loop, which is shown in Fig. 5 (a) and (b). This leads to the formulation of IL_O^S as follows:

$$\begin{aligned} IL_O^S &= IL_O + \Delta II_O \\ &= IL_O + (II_O^S - II_O). \end{aligned} \tag{15}$$

C. Adaptive Activity Scaling

We detail how we deduce the edge activity adaptively to account for design scaling. As shown in Eq. 6, the edge activity is determined by the overall effective latency of the design, *EL*, the number of invocations for the operations connected via this edge, N_v^{dir}, and the Hamming distance $\mathrm{HD}(\cdot)$ of the signal associated with this edge across the whole execution period. It can be reasonably assumed that, for the same application, the average Hamming distance HD_{svg} is statistically consistent because the characteristics of the application remain unchanged. In addition, if we fix *dir* to *src* for simplicity of explanation (without loss of generality, *dst* can be computed in the same way), N_v^{dir} can be represented

as $N_{\boldsymbol{v}}$. On this basis, Eq. 6 for both dataflow and non-dataflow designs can be reduced to

$$SA_{edge} = \frac{N_{\boldsymbol{v}}}{EL} \times HD_{svg}. \tag{16}$$

To take one step further, we can formulate the scaled activity of an edge, SA^S_{edge}, as follows

$$SA^S_{edge} = \frac{N^S_{\boldsymbol{v}}}{EL^S} \times HD_{svg} = \frac{N^S_{\boldsymbol{v}}}{N_{\boldsymbol{v}}} \times \frac{EL}{EL^S} \times SA_{edge}, \tag{17}$$

where EL^S and $N^S_{\boldsymbol{v}}$ denote the scaled effective latency and the scaled number of invocations of the operation connected via this edge, respectively. Since SA_{edge}, $N_{\boldsymbol{v}}$, and EL can be extracted from the reference design, we next discuss how we deduce $N^S_{\boldsymbol{v}}$ and EL^S, after which SA^S_{edge} can be calculated.

Firstly, the scaled number of operational invocations, $N^S_{\boldsymbol{v}}$, depends on the loop hierarchy in which the operation resides. For example, if an operation is within a two-level nested loop, the scaling factors of both the loop levels affect the number of invocations simultaneously. As a result, we develop LLVM [31] passes to profile the loop hierarchies of the design, and we also extract the loop bounds associated with each loop level. On this foundation, we can compute $N^S_{\boldsymbol{v}}$ as

$$N^S_{\boldsymbol{v}} = \left(\prod_{h \in \mathcal{H}} \frac{LB^S_h}{LB_h} \right) \cdot N_{\boldsymbol{v}}, \tag{18}$$

where $\mathcal{H}$ is the set of loop hierarchies that determines the execution of the specified operation, and LB^S_h denotes the scaled loop bound of the loop level h.

Secondly, the scaled effective latency of the design, EL^S, is the sum of all the latency of loops within the design, plus the latency of the standalone inter-loop operations, which can be represented as

$$EL^S = L_{inter} + \sum_{lp \in \mathcal{LP}} L^S_{lp}, \tag{19}$$

where $\mathcal{LP}$ denotes the set of loops in the design, L^S_{lp} denotes the scaled loop latency of the loop *lp*, which is described in Section VII-B, and L_{inter} denotes the latency of the inter-loop operations, which remains constant under different scales and can be extracted from the HLS reports of the reference design.

VIII. Heterogeneous Edge-Centric GNN Model

The graphs generated by our proposed construction flow exhibit the following characteristics: (1) **heterogeneity**: the graph edges have different relation types, indicating whether the corresponding datapaths comprise arithmetic operations; (2) **edge expressivity**: signal switching activities, the major contributors to dynamic power as shown in Eq. 1, are intactly encoded in the edges, which demonstrates that the edge features have higher expressive capability than node features in terms of power consumption; (3) **directionality**: the edges are directed, meaning that data can only pass through an edge from the source node to the sink node, while the opposite is infeasible.

Mainstream GNN models [32], [33] mostly pay attention to node feature aggregation. Some recent works also support using edge features as a supplement to node features [34], [35], or explicitly account for different relation types [36], while node features still play a dominant role in message passing. However, these existing GNN models are ineffective in capturing heterogeneous edge semantics that are essential in the context of power estimation. In light of this, we develop a power-aware heterogeneous edge-centric GNN model, named *HEC-GNN*, which makes full use of edges in neighborhood aggregation to benefit power modeling.

HEC-GNN convolutional layer. Each graph sample built with our construction flow can be represented as $G = (\mathcal{V}, \mathcal{E}, \mathcal{R})$, where $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{R}$ denote the set of graph nodes, edges and relation types, respectively. Formally, our proposed HEC-GNN collects information from neighbors and updates node embeddings at the k-th convolutional layer as

$$\boldsymbol{h}_v^{(k)} = \sigma\left(\boldsymbol{W}_{\mathcal{V}}^{(k)} \boldsymbol{h}_v^{(k-1)} + \mathrm{AGG}^{(k)}(\{\boldsymbol{e}_{u,v,r} | r \in \mathcal{R}, u \in \mathcal{N}_v^r\}) \right), \tag{20}$$

where $\boldsymbol{h}_v^{(k)}$ is the embedding vector of node $v \in \mathcal{V}$, $\boldsymbol{e}_{u,v,r}$ is the edge feature vector of the directed edge from node u to node v with relation type $r \in \mathcal{R}$ and $(u, v, r) \in \mathcal{E}$, $\mathcal{N}_v^r$ is the set of nodes with the successor being node v and the edge relation type being r, $\boldsymbol{W}_{\mathcal{V}}^{(k)}$ is the learnable weight matrix for updating node embeddings from the last layer, σ is the ReLU activation function, and $\mathrm{AGG}^{(k)}$ represents the aggregation function in the k-th layer. HEC-GNN distinguishes itself from existing GNNs by mainly aggregating information from edge feature vectors. Moreover, HEC-GNN retains heterogeneity by modeling the interconnects between nodes with different relation types and separately gathering information of each relation type in the aggregation scheme.

Our HEC-GNN aggregation mechanism $\mathrm{AGG}^{(k)}$ is

$$\mathrm{AGG}^{(k)}(\{\boldsymbol{e}_{u,v,r} | r \in \mathcal{R}, u \in \mathcal{N}_v^r\}) = \sum_{r \in \mathcal{R}} \sum_{u \in \mathcal{N}_v^r} \boldsymbol{W}_r^{(k)} \boldsymbol{W}_{\mathcal{E}}^{(k)} \boldsymbol{e}_{u,v,r}, \tag{21}$$

where $\boldsymbol{W}_{\mathcal{E}}^{(k)}$ and $\boldsymbol{W}_r^{(k)}$ are learnable weight matrices for all edges and the relation type $r \in \mathcal{R}$, respectively. Recall that in Eq. 1, the dynamic power can be viewed as weighted aggregation of interconnect activity α_i with weight being $C_i V^2 f$. The heuristic behind Eq. 21 is to simulate the formation of dynamic power consumption via the HEC-GNN aggregation and update process. Note that the activity α_i has already been encoded in the edge feature vector $\boldsymbol{e}_{u,v,r}$ with interconnect $i = (u, v, r)$ and our goal is to adaptively fit the weight term $C_i V^2 f$ via learnable weight matrices. To achieve this, we first use a global weight matrix $\boldsymbol{W}_{\mathcal{E}}^{(k)}$ to learn common knowledge from all types of edges, which corresponds to the equation term $V^2 f$. Next, we simplify the intricate interconnects using multiple relation types. This enables us to approximate the interconnect capacitance C_i using relation-specific interconnect capacitance C_r, which is produced by the weight matrix $\boldsymbol{W}_r^{(k)}$. Putting it all together, the edge feature vector $\boldsymbol{e}_{u,v,r}$ reflects the activity term α_i, the global weight matrix $\boldsymbol{W}_{\mathcal{E}}^{(k)}$ accounts for $V^2 f$, while the relation-specific weight matrices $\boldsymbol{W}_r^{(k)}, \forall r \in \mathcal{R}$, match with the interconnect capacitance C_i. Hence, the HEC-GNN aggregation mechanism perceives dynamic power consumption by subtly fitting the dynamic power formula.

HEC-GNN overall architecture. Fig. 6 depicts the overall architecture of our HEC-GNN model for power learning. First, the graph data are fed into multiple HEC-GNN convolutional layers to learn and produce node embeddings. After that, the

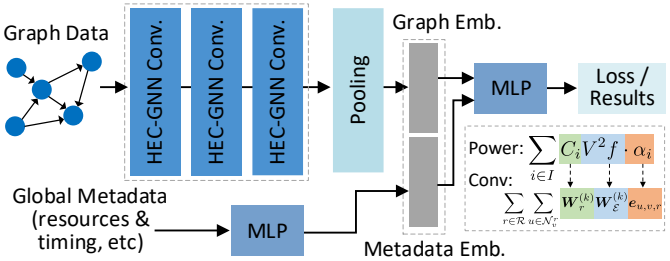


Fig. 6: Overall architecture of HEC-GNN for power learning.

graph-level embedding $\boldsymbol{h}_G$ is obtained via sum pooling:

$$\boldsymbol{h}_G = \sum_{k\in\mathcal{K}} \sum_{v\in\mathcal{V}} \boldsymbol{h}_v^{(k)}, \tag{22}$$

where $\mathcal{K}$ is the set of indexes of graph convolutional layers. Instead of simply pooling the node embeddings from the last convolutional layer, we sum node embeddings obtained from different graph convolutional layers, which can be viewed as a form of skip connection to enhance generalization ability.

In addition to leveraging the graph embeddings that encapsulate localized information mainly contributing to dynamic power, we also construct *global features* from HLS reports that are complementary to the graph information and are indicative of static power. These features include timing information (latency and achieved clock period), global resource utilization (LUT, DSP and BRAM), and the scaling factors, i.e., the ratio of the above design metrics over those of the unoptimized baseline. As shown in Fig. 6, we use a multi-layer perceptron (MLP) with one fully connected layer followed by ReLU activation to embed the above global features. Thereafter, the two sets of embeddings, i.e., the graph embedding $\boldsymbol{h}_G$ and the global feature embedding $\boldsymbol{h}_M$, are concatenated to form a holistic embedding. Finally, the holistic embedding is fed into another MLP with two fully connected layers and ReLU activation in between. The output of this MLP is the total or dynamic power estimation $P_{est.}$:

$$P_{est.} = \mathrm{MLP}(\boldsymbol{h}_G \parallel \boldsymbol{h}_M). \tag{23}$$

The above model components are cascaded to constitute our end-to-end supervised model, HEC-GNN, which is trained via regression to minimize the mean average percentage error loss. Furthermore, we adopt an ensemble learning strategy, in which we perform 10-fold cross-validation together with three different random seeds to generate different training and validation sets for model generation, and average all the output of trained models to get the final prediction results. Overall, HEC-GNN is heterogeneous, edge-centric and directed.

IX. Experimental Results

A. Experimental Setup

The graph construction flow of PowerGear is implemented with Python for HLS data extraction and graph generation, and C++ for IR modification and detection probe implementation. The proposed HEC-GNN model and the baseline GNNs [32]–[36] are developed with PyTorch Geometric [37] and Scikit-learn [38] toolkits. Regarding the hyperparameters of GNNs, we employ three layers of graph convolution with a hidden dimension of 128, a batch size of 128, a dropout rate of 0.2 and a learning rate of 0.0005. We train the GNN models with 1,200 and 2,400 epochs for total and dynamic power estimation, respectively. 20% of the training data are used as the validation set. Software design flows run on 80-core Intel Xeon CPU at 2.4 GHz with one Nvidia Tesla V100 GPU.

We evaluate our approach using Polybench [39] datasets with different dataset properties shown in Table I and a large-scale CNN instance [40] as a case study. The hardware designs are developed with Vivado HLS and Vivado 2018.2 and implemented on Xilinx Ultrascale+ ZCU102 FPGA board at the frequency of 100 MHz. HLS design samples are generated by applying loop pipelining, loop unrolling and array partitioning on multiple hierarchies. Ground truth power values are obtained via power measurement with Power Advantage Tool [41]. Overall, this produces a sample set with substantial variation: dynamic power ranges from 1.61 to 5,434 mW; latency varies from 7.8×10^2 to 5.36×10^7 cycles; and resource utilization spans 0.2%–81.6% for LUT, 0.3%–43.2% for DSP, 0.3%–49.1% for BRAM.

B. Estimation Accuracy

In this experiment, we evaluate both total and dynamic power estimation accuracy for HLS-based FPGA designs using PowerGear. To evaluate the method's transferability, we adopt a *leave-one-out* training scheme: we iteratively leave one benchmark out of the nine benchmarks from Polybench [39] as the test set, and use all the others for training. This setup ensures strict separation between training and testing applications, preventing any bias toward the test set. We compare PowerGear with the Vivado power estimator [12], the state-of-the-art work HL-Pow [9], and mainstream GNNs [32]–[36]. As for Vivado power estimation, we import the gate-level netlist after physical implementation and provide *.saif* activity files via vector-based simulation, which ensures a high confidence level of estimation precision. Moreover, we observe through experiments that the Vivado power estimator neglects the impact of power gating [27] on unused hard blocks, leading to a severe deviation from real power consumption. Hence, we further calibrate the results with a linear regression model. As for HL-Pow [9], we follow its design flow and implement the gradient boosting decision tree (GBDT) models. Similar to GNNs, we use 20% of training data for validation, based on which we tune the hyperparameters with tree size in [10, 500], tree depth in [5, 10], minimum samples per leaf in [2, 8], and learning rate in {0.005, 0.01, 0.05}.

As shown in Table I, regarding total power estimation, Vivado power estimation diverges considerably from the real measurement, leading to an average error of 21.82%, whereas the HL-Pow and our proposed PowerGear achieve good prediction accuracy with average errors of 3.79% and 3.60%, respectively. In general, PowerGear achieves the best average accuracy for total power estimation. Regarding dynamic power estimation, we compare PowerGear with HL-Pow and commonly used GNN models [32]–[36]. Overall, PowerGear outperforms all baseline methods on all the evaluated datasets, significantly reducing the estimation error of dynamic power from 12.67% down to 8.81% compared with HL-Pow. Results

TABLE I: Dataset properties, results of total and dynamic power estimation.

Dataset	Dataset Properties		Error of Total Power (%)			Error of Dynamic Power (%)						
	#Samples	Avg. #Nodes	Vivado	HL-Pow	PowerGear	GCN	R-GCN	GraphSage	*k*-GNN	GINE	HL-Pow	PowerGear
Atax	521	141	15.04	**5.99**	6.15	13.96	11.91	14.59	11.93	15.45	14.92	**11.18**
Bicg	489	137	13.61	**4.42**	4.97	11.96	11.65	11.35	11.06	10.44	12.21	**9.65**
Gemm	531	341	25.59	3.22	**2.75**	11.02	9.51	9.66	9.53	8.92	10.35	**8.32**
Gesummv	529	221	28.27	2.74	**2.67**	16.59	12.35	14.33	12.14	11.89	13.52	**9.35**
2mm	483	443	24.34	**3.52**	4.47	8.51	9.13	8.87	10.32	8.93	9.14	**6.81**
3mm	483	447	16.93	**2.19**	2.63	12.55	12.60	12.06	11.55	10.92	12.60	**8.62**
Mvt	531	165	19.03	2.98	**2.77**	13.94	13.04	12.09	9.94	10.04	12.38	**8.77**
Syrk	530	320	26.24	5.37	**3.76**	14.53	15.24	14.51	13.83	11.70	14.50	**8.64**
Syr2k	483	444	27.36	3.70	**2.26**	13.42	12.56	9.75	8.82	12.24	14.45	**7.98**
Average	–	295	21.82	3.79	**3.60**	12.94	11.99	11.91	11.01	11.17	12.67	**8.81**

TABLE II: Error (%) of dynamic power estimation using different HEC-GNN variants.

Dataset	W/o opt.	W/o e.f.	W/o dir.	W/o hetr.	W/o md.	Sgl.	Prop.
Atax	13.32	11.40	11.84	11.39	12.37	11.68	**11.18**
Bicg	10.63	10.56	10.19	9.98	11.08	10.06	**9.65**
Gemm	10.04	10.39	8.53	9.48	8.66	8.79	**8.32**
Gesummv	13.83	11.14	9.99	10.31	9.74	9.68	**9.35**
2mm	10.25	8.67	6.97	7.64	7.47	6.93	**6.81**
3mm	11.77	9.62	9.29	9.15	11.93	8.96	**8.62**
Mvt	12.54	9.40	8.70	9.71	8.81	**8.47**	8.77
Syrk	12.64	10.92	8.95	9.51	10.75	9.04	**8.64**
Syr2k	9.62	9.66	8.51	8.97	**7.16**	8.10	7.98
Average	11.74	10.20	9.22	9.57	9.77	9.08	**8.81**

verify that PowerGear's graph construction flow successfully retains interconnect and activity features that have important implication for power consumption, and moreover, the HEC-GNN model proposed in PowerGear effectively mines rich heterogeneous edge semantics and structural properties via the edge-centric neighborhood aggregation mechanism. In comparison to the GNN baselines, including GCN [32] and GraphSage [33] that simply focus on node features, R-GCN [36] used to model relational graphs, and *k*-GNN [35] and GINE [34] which use both node and edge features, PowerGear still reaps notable accuracy improvement. Our insight is that PowerGear intelligently fits the dynamic power formulation in HEC-GNN aggregation and additionally accounts for overall power characteristics using metadata embeddings, leading to high efficacy and generalization ability for power modeling.

C. Ablation Study

To gain a better understanding of various optimization strategies adopted by HEC-GNN in PowerGear, we conduct an ablation study which evaluates the accuracy of dynamic power estimation using six variants of HEC-GNN in addition to the proposed HEC-GNN denoted as *prop.* for short. These models include: (1) *w/o opt.*: single unoptimized HEC-GNN without considering edge features, directionality, heterogeneity, and global metadata embeddings; (2) *w/o e.f.*: single HEC-GNN without using edge features; (3) *w/o dir.*: single HEC-GNN without directionality; (4) *w/o hetr.*: single HEC-GNN without using multiple relation types; (5) *w/o md.*: single HEC-GNN without using the global metadata embeddings; and (6) *sgl.*: single optimized HEC-GNN without considering ensemble.

As shown in Table II, our proposed HEC-GNN model is consistently superior to its six variants in seven out of nine datasets for dynamic power estimation with FPGA HLS, and finally yields the best accuracy on average. It can be inferred from the results that the adoption of edge features, directionality, heterogeneity, global metadata embeddings and ensemble all contribute to the enhancement of model accuracy, which explains the effectiveness of HEC-GNN that jointly makes use of all these optimization strategies.

TABLE III: Latency and power estimation with design scaling.

Dataset	Error (%) of Latency	Error (%) of Dynamic Power	
		Exact Method	Scaling Method
Atax	1.68	11.94	11.71
Bicg	0.12	9.03	9.81
Gemm	0.29	7.18	8.56
Gesummv	0.77	11.53	15.25
2mm	0.14	6.88	8.61
3mm	0.49	13.44	13.22
Mvt	0.33	8.13	7.97
Syr2k	0.11	8.96	9.56
Syrk	1.64	7.91	8.56
Average	0.62	9.44	10.36

D. Accuracy and Speedup of Design Scaling

In this experiment, we examine the effectiveness of the proposed design scaling strategy. In the previous experiments, the design sizes of the benchmarks are configured to be 4,096. That is, the maximum array size in the applications is 4,096 and the arrays are processed in several multi-level nested loops. Under design scaling, we scale up the design size to 16,384, and keep the loop structures intact while enlarging the loop bounds to coordinate the manipulation on arrays. We use the same set of benchmarks in Section IX-B and correspondingly scale up their design points, which yields a total of 9,227 training samples. Among them, the scaled designs exhibit an average dynamic power increase of 40–60%. We then evaluate the benefits from our design scaling strategy as illustrated in Section VII. Specifically, we focus our evaluation on three aspects: the efficacy of latency modeling, the accuracy of power prediction, and the runtime speedup, all of which are compared with the primitive version of PowerGear [26] without considering design scaling.

First, we evaluate the efficacy of the proposed analytical latency scaling approach. Since latency serves as a key factor in normalizing the switching activity, as shown in Eq. 6, its prediction accuracy directly impacts the fidelity of scaled graph features. We compare the latency predicted by our scaling approach with the ground-truth latency obtained from Vivado HLS for every scaled design point. The results are presented in Table III. Our latency scaling approach achieves the prediction errors within 2% across all evaluated benchmarks,

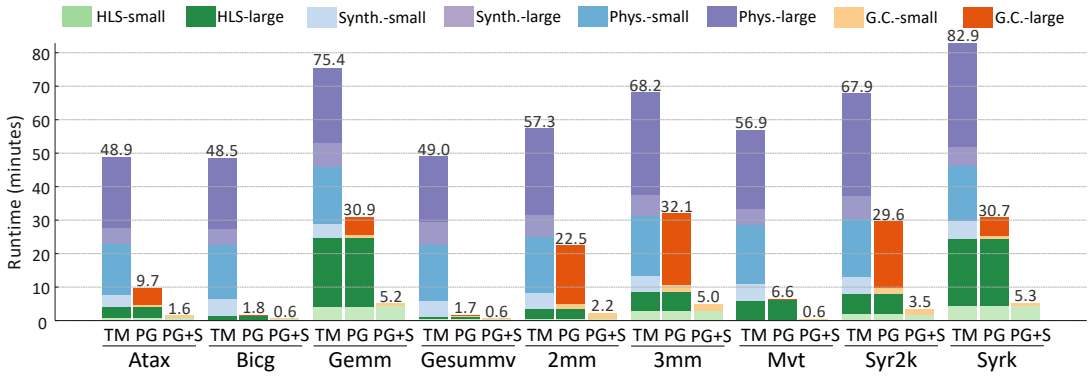


Fig. 7: Runtime breakdown per design point for the traditional measurement method (TM), primitive PowerGear (PG) and PowerGear with scaling (PG+S), in which '-small' and '-large' represent the design scale of 4,096 and 16,384, respectively.

with an average error of 0.62%. These results confirm that the proposed method can reliably estimate the effective execution latency by generalizing from the reference design rather than relying on the invocation of EDA flows or simulation, laying a solid foundation for efficient power modeling.

Second, we evaluate the accuracy of dynamic power estimation using our scaling-based modeling approach. In this context, we construct the graphs for large-scale designs from existing small-scale reference designs using our scaling strategy. These scaled graphs are used together with those of the small-scale reference designs to train power models under the leave-one-out scheme. We compare the dynamic power estimation accuracy of the scaled design between our scaled-graph-based model and the exact model that is developed from precise graphs generated through a complete graph construction process. As presented in Table III, the scaling-based model achieves an average error of 10.36% across all benchmarks, while the exact model delivers an average error of 9.44%. The scaling-based model shows competitive performance, with a 0.92% degradation in accuracy compared to the precise modeling method. In all, these results fully justify that the scaling mechanism preserves key structural and semantic properties essential to power prediction, and can serve as a light-weight alternative when large-scale synthesis is not feasible or has excessive runtime costs.

Third, we evaluate the runtime speedup of PowerGear. To start with, we analyze the time composition of the following three methods: traditional method with on-board measurement (TM), primitive PowerGear without design scaling (PG) [26], and the proposed PowerGear with design scaling (PG+S). The corresponding results are depicted in Fig. 7. From Fig. 7, we can see that the most time-consuming portion is the physical design, which consumes more than 67.8% of the total runtime for TM. In contrast, PG replaces the slow physical design process with fast graph construction and inference flow (marked as G.C.), notably reducing the runtime by 2.1×–28.8×, with an average speedup of 9.1×. Furthermore, we notice that the runtime of graph construction still increases with the design scales, especially for complex designs such as *3mm* and *Syr2k*. PG+S only requires the execution of HLS and graph construction flow for small-scale designs, and can directly infer the graphs for large-scale designs. This scaling method only involves light-weight update on the latency and switching activities using existing small-scale designs, which can be completed in seconds. As a result, PG+S improves the speedup by 2.8×–11× (6.6× on average) over PG. Eventually, this leads to an average overall speedup of 41.9× for PowerGear with design scaling compared to the traditional measurement approach.

E. Case Study: Large-Scale Dataflow CNN

We validate the effectiveness of our adaptation to dataflow optimization strategy, as described in Section VI, through a case study of large-scale dataflow designs. We utilize the CNN [40] as the benchmark, which processes 28×28 grayscale handwritten digit images and outputs the corresponding classification results. We construct an entire family of CNN instances with different levels of architectural complexities: (1) each design point comprises 2–4 convolutional layers (each followed by a ReLU activation), one max-pooling layer, one flattening layer, two fully connected layers, and a softmax function; and (2) each convolutional layer is equipped with 4–8 convolution kernels of size 3×3, and inter-function communication is supported by FIFOs. To introduce diversity from HLS, we iteratively add loop pipelining to loops in all layers, and we also build dataflow and non-dataflow variants by selectively applying dataflow directive to the top-level function. This yields a total number of 10,080 distinct design points. Among these designs, the maximum resource utilization reaches 42.5% for BRAM, 33.4% for DSP, 27.6% for FF, and 80.7% for LUT on the target FPGA ZCU102, while the maximum power consumption reaches 1,550 mW and the maximum latency is 1.62 million cycles, which clearly demonstrates the large-scale nature of this application.

We construct two types of graph variants, either using the proposed *EL* (adaptation to dataflow) or *L* (original method in [26]) reported by Vivado HLS as normalization factors in Eq. 6, respectively. For each type of graph variants, we split the design points into training, validation, and testing sets with a ratio of 70%, 15%, and 15%, respectively. The design points in all the three sets are consistent across both the graph types, ensuring fairness in comparison. Finally, we train the GNNs specifically for each type of graphs to predict dynamic power.

The experimental results are shown in Table IV. The estimation error of dataflow designs decreases significantly from 11.24% to 3.91% when switching from the original method

TABLE IV: Error (%) of dynamic power estimation for large-scale dataflow CNN application.

Graph Variant	Test Case	Error (%)
Original Method in [26] (Normalization with *L*)	Non-Dataflow	10.91
	Dataflow	11.24
Adaptation to Dataflow (ours) (Normalization with *EL*)	Non-Dataflow	4.86
	Dataflow	3.91

to our adaptive method for dataflow, indicating that *EL* can well capture the concurrency of dataflow designs. This further calibrates the switching activities, ultimately improving the generalization ability of power modeling. Interestingly, for non-dataflow designs, our method also reduces the error from 10.91% to 4.86%, even though the graphs for non-dataflow designs are identical to those of the original method. This improvement reveals that more accurate graph construction for dataflow designs also benefits the learning of non-dataflow cases. Furthermore, the success in modeling large-scale CNN designs also validates the effectiveness of the proposed cross-function integration technique.

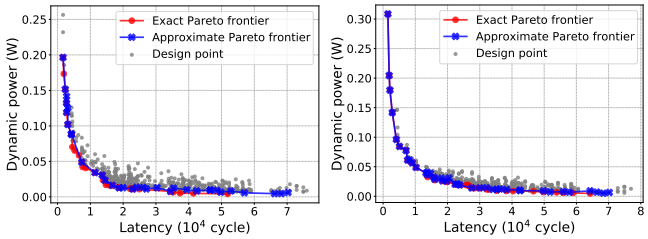
F. Case Study: Design Space Exploration

We demonstrate a case study in which PowerGear is exploited as the prediction model to guide fast HLS-based hardware design space exploration (DSE) to trade off between latency and dynamic power. Given a dataset for DSE, we first sample a small subset of design points for HLS and then utilize PowerGear to estimate dynamic power. Together with the set of latency derived from HLS, we compute the dynamic power-latency Pareto frontier using existing sampling points, based on which a sampling algorithm [9] is applied to select promising design points that are most likely to be Pareto-optimal for further evaluation. The above steps are conducted iteratively to calibrate the approximate Pareto frontier until the total sampling budget is met. Experiments are performed with an initial sampling budget of 2% and total sampling budgets of 20%, 30% and 40%, respectively. Vivado power estimator and HL-Pow are used as alternative power prediction models for comparison. A commonly used evaluation metric *average distance from reference set (ADRS)* is employed to quantify the difference between the approximate and exact Pareto frontiers:

$$\mathrm{ADRS}(\Gamma, \Omega) = \frac{1}{|\Gamma|} \sum_{\gamma \in \Gamma} \min_{\omega \in \Omega} f(\gamma, \omega), \tag{24}$$

where Ω is the approximate Pareto-optimal set, Γ is the exact Pareto-optimal set, and $f(\cdot)$ computes the distance between two design points $\omega \in \Omega$ and $\gamma \in \Gamma$. A lower ADRS means that the approximate Pareto set is closer to the exact one.

Fig. 8 shows the DSE results of two datasets under a total sampling budget of 40%, which indicate that PowerGear can effectively aid in DSE algorithms to realize close approximation of Pareto-optimal sets with moderate runtime overhead. Moreover, as shown in Table V, the ADRS of Pareto search using PowerGear as the prediction model is the best for all three sampling budgets. Compared with the methods using Vivado and HL-Pow, PowerGear-assisted DSE yields relative performance gains of 39.2–52% and 6.9–11.2%, respectively. Moreover, PowerGear achieves a 27–35× runtime



(a) Atax (b) Mvt

Fig. 8: Pareto frontiers of Atax and Mvt with PowerGear.

TABLE V: ADRS and runtime of design space exploration using Vivado (V), HL-Pow (HP) and PowerGear.

Sampling Budget	ADRS of Evaluation Methods			Performance Gains		Runtime Gains	
	Vivado	HL-Pow	PowerGear	v.s. V	v.s. HP	v.s. V	v.s. HP
20%	0.1657	0.1050	0.0981	39.2%	6.9%	35.84×	-1.22%
30%	0.1520	0.0841	0.0774	45.8%	10.4%	27.57×	2.06%
40%	0.1423	0.0691	0.0626	52.0%	11.2%	35.47×	1.76%

speedup over Vivado and maintains comparable runtime to HL-Pow, demonstrating its superior efficiency. Through this case study, we show that PowerGear's capability to provide accurate and efficient dynamic power estimation opens up more opportunities for power optimization in FPGA HLS.

X. Conclusion

In this paper, we propose PowerGear, a graph-learning-assisted framework for HLS-based early-stage power estimation of FPGA designs. PowerGear introduces a comprehensive graph construction flow that encapsulates micro-architectures and interconnects with switching activities, and proposes a novel heterogeneous edge-centric GNN that adapts to dynamic power modeling. To support realistic applications, PowerGear integrates cross-function graph construction and dataflow-specific normalization using effective latency, enabling robust modeling of compound and computation-intensive designs with concurrency. To improve modeling scalability, PowerGear is equipped with an analytical latency scaling model and an adaptive activity scaling strategy that allow accurate yet efficient graph inference under varying design scales without the need for graph reconstruction.

Extensive experiments demonstrate that PowerGear outperforms state-of-the-art commercial and learning-based estimators in terms of total and dynamic power estimation accuracy, generalization ability, and runtime efficiency. Furthermore, the design scaling mechanism enables over 6.6× speedup with negligible accuracy loss, and the adaptation to dataflow significantly reduces dynamic power estimation error of dataflow-optimized designs from 11.24% to 3.91%. Finally, we showcase PowerGear's practical usage through a case study of design space exploration, where it enables high-quality Pareto frontier approximation with small sampling costs. Altogether, these innovations make PowerGear an accurate, efficient and scalable power modeling solution for modern FPGA HLS applications.

References

[1] J. M. Levine, E. Stott, and P. Y. Cheung, "Dynamic voltage & frequency scaling with online slack measurement," in *Proc. of FPGA*, 2014.

[2] G. Akgün, M. Ali, and D. Göhringer, "Power-aware computing systems on fpgas: A survey," in *Proc. of FPL*, 2021, pp. 45–51.
[3] D. Liu and C. Svensson, "Power consumption estimation in CMOS VLSI chips," *IEEE Journal of Solid-State Circuits (JSSC)*, 1994.
[4] Z. Lin, S. Sinha, and W. Zhang, "An ensemble learning approach for in-situ monitoring of FPGA dynamic power," *TCAD*, 2018.
[5] Y. Zhou, H. Ren, Y. Zhang, B. Keller, Z. Zhang *et al.*, "PRIMAL: Power inference using machine learning," in *Proc. of DAC*, 2019.
[6] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph neural network inference for transferable power estimation," in *Proc. of DAC*, 2020.
[7] W. Zuo, W. Kemmerer, J. B. Lim, L.-N. Pouchet *et al.*, "A polyhedral-based SystemC modeling and generation framework for effective low-power design space exploration," in *Proc. of ICCAD*, 2015.
[8] D. Lee, L. K. John, and A. Gerstlauer, "Dynamic power and performance back-annotation for fast and accurate functional hardware simulation," in *Proc. of DATE*, 2015.
[9] Z. Lin, T. Liang, J. Zhao, S. Sinha, and W. Zhang, "HL-Pow: Learning-assisted pre-RTL power modeling and optimization for FPGA HLS," *TCAD*, vol. 42, no. 11, pp. 3925–3938, 2023.
[10] P. Coussy, D. D. Gajski, M. Meredith *et al.*, "An introduction to high-level synthesis," *IEEE Design Test of Computers*, 2009.
[11] L. Zhong and N. K. Jha, "Interconnect-aware high-level synthesis for low power," in *Proc. of ICCAD*, 2002.
[12] Xilinx Ltd, "Vivado design suite tutorial: Power Analysis and Optimization," *Xilinx White Paper*, October 2019.
[13] N. Wu, H. Yang, Y. Xie, P. Li, and C. Hao, "High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing," in *Proc. of DAC*, 2022.
[14] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *Proc. of FCCM*. IEEE, 2018, pp. 129–132.
[15] E. Murphy and L. Josipović, "Balor: Hls source code evaluator based on custom graphs and hierarchical gnns," in *Proc. of ICCAD*, 2024.
[16] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Robust gnn-based representation learning for hls," in *Proc. of ICCAD*, 2023, pp. 1–9.
[17] D. Chen, J. Cong, Y. Fan, and L. Wan, "Lopass: A low-power architectural synthesis system for fpgas with interconnect estimation and optimization," *TVLSI*, vol. 18, no. 4, pp. 564–577, 2009.
[18] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of MICRO*, 2009, pp. 469–480.
[19] S. Ahuja, D. A. Mathaikutty, G. Singh, J. Stetzer, S. K. Shukla, and A. Dingankar, "Power estimation methodology for a high-level synthesis framework," in *Proc. of ISQED*, 2009, pp. 541–546.
[20] Y. 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," in *Proc. of ISCA*, 2014.
[21] Y. Liang, S. Wang, and W. Zhang, "Flexcl: A model of performance and power for opencl workloads on fpgas," *IEEE Transactions on Computers*, 2018.
[22] Z. Xie, X. Xu, M. Walker, J. Knebel, K. Palaniswamy, N. Hebert, J. Hu, H. Yang, Y. Chen, and S. Das, "Apollo: An automated power modeling framework for runtime power introspection in high-volume commercial microprocessors," in *Proc. of MICRO*, 2021, pp. 1–14.
[23] D. Kim, J. Zhao, J. Bachrach, and K. Asanović, "Simmani: Runtime power modeling for arbitrary rtl with automatic signal selection," in *Proc. of MICRO*, 2019, pp. 1050–1062.
[24] Y. Zhou, H. Ren, Y. Zhang, B. Keller, B. Khailany, and Z. Zhang, "Primal: Power inference using machine learning," in *Proc. of DAC*, 2019, pp. 1–6.
[25] W. Fang, Y. Lu, S. Liu, Q. Zhang, C. Xu, L. W. Wills, H. Zhang, and Z. Xie, "Masterrtl: A pre-synthesis ppa estimation framework for any rtl design," in *Proc. of ICCAD*. IEEE, 2023, pp. 1–9.
[26] Z. Lin, Z. Yuan, J. Zhao, W. Zhang, H. Wang, and Y. Tian, "Powergear: Early-stage power estimation in fpga hls via heterogeneous edge-centric gnns," in *Proc. of DATE*, 2022, pp. 1341–1346.
[27] S. Kolluri, "UltraScale Architecture Low Power Technology Overview," *Xilinx White Paper*, October 2015.
[28] E. Ustun, C. Deng, D. Pal *et al.*, "Accurate operation delay prediction for fpga hls using graph neural networks," in *Proc. of ICCAD*, 2020.
[29] R. Namballa, N. Ranganathan, and A. Ejnioui, "Control and data flow graph extraction for high-level synthesis," in *Proc. of ISVLSI*, 2004, pp. 187–192.
[30] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance modeling and directives optimization for high-level synthesis on fpga," *TCAD*, vol. 39, no. 7, pp. 1428–1441, 2020.
[31] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," in *Proc. of CGO*, 2004, pp. 75–86.
[32] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. of ICLR*, 2017.
[33] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. of NeurIPS*, 2017.
[34] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang *et al.*, "Strategies for pre-training graph neural networks," in *Proc. of ICLR*, 2020.
[35] C. Morris, M. Ritzert, M. Fey *et al.*, "Weisfeiler and leman go neural: Higher-order graph neural networks," in *Proc. of AAAI*, 2019.
[36] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *Proc. of ESWC*, 2018.
[37] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *Proc. of ICLR Workshop*, 2019.
[38] F. Pedregosa, G. Varoquaux *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, 2011.
[39] L.-N. Pouchet. (2012) Polybench: The polyhedral benchmark suite. [Online]. Available: http://www.cs.ucla.edu/%7Epouchet/software/polybench
[40] F. Serafini, "High level synthesis of a trained cnn for andwritten digit recognition," *Embedded Systems Course for Universita degli Studi di Parma*, July 2022. [Online]. Available: https://github.com/FedericoSerafini/HLS-CNN
[41] Xilinx Ltd, "Zynq UltraScale+ MPSoC Power Advantage Tool 2018.1," *Xilinx Wiki*, 2018.

Sen Yan (Student Member, IEEE) received the B.Eng. degree in Microelectronics Science and Engineering from the School of Integrated Circuits, Sun Yat-sen University, China, in 2025. Currently, he is pursuing the M.S. degree at the School of Integrated Circuits, Sun Yat-sen University. His research interests are efficient power modeling methods for emerging computing platforms.

Kuangxun Huang (Student Member, IEEE) received the B.Eng. degree in Optoelectronic Information Science and Engineering from the School of Physics and Optoelectronic Engineering, Guangdong University of Technology, China, in 2022. He is currently pursuing the M.S. degree in Integrated Circuit Engineering at the School of Integrated Circuits, Sun Yat-sen University, China. His research interests include power modeling for FPGA, and algorithm-hardware co-design for communication systems.

Kang Zhao (Member, IEEE) received the Ph.D. degree in the Department of Computer Science and Technology from Tsinghua University, China in 2009.

Dr. Zhao is a Professor with Beijing University of Posts and Telecommunications (BUPT), where he leads the EDA research team. Prior to join BUPT, he worked in Tsinghua University, Intel, Xilinx, and AMD, successively. When working in Xilinx and AMD, he served as the senior staff manager and the leader of the Vitis HLS product team. His research interests include FPGA and EDA, especially on high-level synthesis, logic synthesis, compiler techniques, placement and floorplan, and other VLSI CAD algorithms.

Zhe Lin (Member, IEEE) received the B.S. degree from the School of Electronic Science and Engineering, Southeast University, China, in 2014, and the Ph.D. degree from the Department of Electronic and Computer Engineering at Hong Kong University of Science and Technology, Hong Kong, China, in 2020.

Dr. Lin is an Assistant Professor with the School of Integrated Circuits, Sun Yat-sen University, China. Previously, he was an Associate Research Fellow with Peng Cheng Laboratory, China. Dr. Lin's research interests cover front-end design automation for FPGA, hardware-software co-design, and heterogeneous computing systems. Dr. Lin was a recipient of the Best Paper Award Nomination at ICCD 2024, DATE 2022 and FCCM 2019.