Cross-Modality Program Representation Learning for Electronic Design Automation with High-Level Synthesis

Zongyue Qin*, Yunsheng Bai*
Atefeh Sohrabizadeh, Zijian Ding, Ziniu Hu
Yizhou Sun, Jason Cong
University of California, Los Angeles
USA
{qinzongyue,yba,atefehsz,bradyd,bull,yzsun,cong}@cs.ucla.edu

Abstract

In recent years, domain-specific accelerators (DSAs) have gained popularity for applications such as deep learning and autonomous driving. To facilitate DSA designs, programmers use high-level synthesis (HLS) to compile a high-level description written in C/C++ into a design with low-level hardware description languages that eventually synthesize DSAs on circuits. However, creating a high-quality HLS design still demands significant domain knowledge, particularly in microarchitecture decisions expressed as *pragmas*. Thus, it is desirable to automate such decisions with the help of machine learning for predicting the quality of HLS designs, requiring a deeper understanding of the program that consists of original code and pragmas. Naturally, these programs can be considered as sequence data. In addition, these programs can be compiled and converted into a control data flow graph (CDFG). But existing works either fail to leverage both modalities or combine the two in shallow or coarse ways. We propose ProgSG, a model that allows interaction between the source code sequence modality and the graph modality in a deep and fine-grained way. To alleviate the scarcity of labeled designs, a pre-training method is proposed based on a suite of compiler's data flow analysis tasks. Experimental results show that ProgSG reduces the RMSE of design performance predictions by up to 22%, and identifies designs with an average of 1.10× and 1.26× (up to 8.17× and 13.31×) performance improvement in design space exploration (DSE) task compared to HARP and AutoDSE, respectively.

CCS Concepts

• **Hardware → High-level and register-transfer level synthesis**; • **Computing methodologies → Neural networks**.

Keywords

GNN, Language Model, HLS, FPGA

*Both authors contributed equally to this research.

MLCAD '24, September 9–11, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0699-8/24/09
https://doi.org/10.1145/3670474.3685952

ACM Reference Format:
Zongyue Qin[1], Yunsheng Bai, Atefeh Sohrabizadeh, Zijian Ding, Ziniu Hu, and Yizhou Sun, Jason Cong. 2024. Cross-Modality Program Representation Learning for Electronic Design Automation with High-Level Synthesis. In *2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD '24), September 9–11, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3670474.3685952

1 Introduction

Over the past decades, the need for specialized computing systems to accelerate specific applications has grown, leading to the emergence of domain-specific accelerators (DSAs) like application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs). Designing DSAs is challenging because it involves using hardware description languages (HDLs) at the register-transfer level (RTL) with Verilog and VHDL, which are mainly familiar to circuit designers. High-level synthesis (HLS) was introduced to address this by raising the level of abstraction to C/C++/OpenCL/SystemC, allowing designers to describe high-level behavioral representations of their designs. Despite this, HLS tools still require significant hardware design knowledge through synthesis directives in the form of pragmas, which specify computation parallelization, data caching, memory buffer partitioning, etc. These optimizations are typically done by hardware programmers and are beyond the reach of average software programmers. Our objective is to automate and accelerate the optimization of integrated circuit (IC) design, making it more accessible to software programmers.

There is a growing trend to apply machine learning to IC design automation [16]. For example, researchers have developed learning-based methods to predict the quality of HLS designs [39, 40], to explore the HLS design space intelligently for optimal resource allocation [48], etc. These methods fundamentally rely on an informative representation of an input design for high-quality performance prediction. We, therefore, focus on the representation learning for IC designs defined with HLS C/C++ (in short, we call them HLS designs) which are annotated with compiler directives/pragmas. Specifically, we aim to design an encoder-decoder framework where the encoder provides powerful representations for the input HLS designs so that the designs' quality can be predicted accurately.

One limitation of the existing representation learning methods for programs and HLS designs is that they usually restrict the model to only using either the source code or the compiler-derived representation, but not both. For example, previous works [39, 40, 49] compile the HLS code into LLVM intermediate representation,

which is then further transformed into a graph representation before a graph neural network (GNN) is used to encode it. Meanwhile, [10, 15, 19, 47] directly apply a large language model (LLM) to the source code to obtain the representations that catch the semantics of general computer programs.

However, we argue that only utilizing either one of the modalities is not good enough to obtain a comprehensive program representation. On the one hand, the graph modality tends to ignore the semantic information in the source code which is helpful to understand a program's behavior. For example, in CDFG, it is difficult for GNN to understand the functionality of a call site, particularly to ones such as standard libraries (e.g., `glibc`). What is worse, a statement such as "`A[i][j] *= beta;`" would be converted to a relatively large and complex subgraph in the CDFG making it difficult for the model to understand the semantic meaning. On the other hand, two source code programs with similar semantics and functionalities could have significantly different latency and communication requirements. This is where the lower-level control-flow structure of the programs can help. Therefore, a novel model that effectively utilize information from both modalities could be the key to generating powerful representations of HLS designs and general programs.

In this paper, we propose ProgSG (Program representation learning combining the source Sequence and the control data flow Graph) for a unified representation learning that leverages both the source code modality and an enriched CDFG graph modality, with pre-training performed on both modalities. To handle the interaction between source code and CDFG graph, we propose two innovative designs in the architecture: (1) An attention-summary architecture for coarse interaction between the two modalities; (2) A fine-grained node-to-token message passing mechanism to enable further collaboration between the two modalities. We also propose a novel pre-training method based on predicting node-node relationships for compiler analysis tasks which helps the GNN encoder to address the label scarcity issue. Experiment results show the proposed ProgSG achieves a state-of-art performance on design quality prediction and design space exploration.

2 Preliminaries

2.1 HLS Design and Optimization Pragmas

The goal of this paper is to train a model to effectively predicts the quality of the HLS design, which is a C/C++ program with inserted pragmas serving as design specification. The quality of a design is measured by its latency in cycle counts (perf), the utilization rate of block RAM (util-BRAM), digital signal processors (util-DSP), flip-flop (util-FF), and lookup-tables (util-LUT) [39, 41].

We specifically consider the optimization pragmas of the Merlin Compiler, an open-source tool widely used for HLS designs[1]. The Merlin Compiler provides three types of optimization pragmas, namely `PIPELINE`, `PARALLEL`, and `TILE` to define the desired microarchitecture [41]. As shown in Code 1 in the Appendix, these pragmas can be applied at the loop level and offer control over the type of pipelining, the parallelization factor, and the amount of data caching. Table 1 summarizes the parameter space of these pragmas.

[1]https://github.com/Xilinx/merlin-compiler

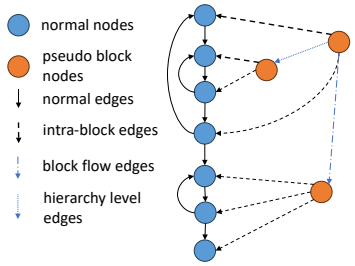


Figure 1: An Illustration of HARP control data flow graph. Compared with a normal CDFG, it has additional block nodes and three types of edges: intra-block edges, block-flow edges, and hierarchy-level edges.

For a given program P, any change in the option of any of the pragmas results in a different design D with a unique microarchitecture. For example, the "fg" option in pipelining refers to the case where all the inner loops are unrolled (parallelized with separate logic) and each parallel unit is pipelined. The "cg" option, on the other hand, results in coarse-grained processing elements (PEs) that are pipelined together. For example, it can create pipelined load-compute-store units. The `PARALLEL` and `TILE` pragmas take numeric values that determine the degree of parallelization and loop tiling, respectively.

Table 1: Target pragmas with their options.

Pragma	Parameter Name	Parameter Space
`PARALLEL`	factor	integer
`PIPELINE`	mode	"cg", "fg", off
`TILE`	factor	integer

2.2 Hierarchical Graph Representation of HLS Designs

We leverage HARP's approach [40] to generate the hierarchical graph representation of an HLS design, which is an enriched CDFG with extra block nodes and their connections. Figure 1 depicts an illustration of a *HARP graph*. Specifically, given the source code $C = (c_1, \ldots, c_I)$ (c_i, $i = 1, \ldots, I$ denotes the i-th token of the source code), it is first transformed into an LLVM [20] intermediate representation (IR), and further converted into a CDFG[2]. Then to insert hierarchical information into the graph, auxiliary nodes are added into the graph where each auxiliary node represents a distinct LLVM IR block. Each of these blocks is a sequence of instructions that has a single entry point and a single exit point. Each auxiliary node has three types of edges: the edges to all instruction and data nodes within that block (intra-block edges), the edges to the previous and next block (block flow edges), and the edges building connections based on the hierarchy level of the "for" loops in

[2]Strictly speaking, it is a modified ProGraML graph with additional call relations between instructions and explicit nodes for operands with additional pragma nodes, but for convenience and without loss of generality, we use the term "CDFG" in this paper.

the C/C++ code (hierarchy level edges). HARP [40] shows that the hierarchical graph representation helps propagate the long-range dependency information in the graph, which helps it learn a better graph representation.

3 Proposed Method: ProgSG

In this section, we first describe the overall encoder-decoder architecture of ProgSG. Then, we focus on our novel encoder with a graph summary augmented sequence representation, and a fine-grained node-to-token alignment for the unification of the two modalities. Finally, we introduce a novel pre-training framework for program graphs.

3.1 Overall Architecture for Design Quality Prediction

Given a design D with source code C and HARP graph G, the overall model $f(D) = f(C, G)$ first encodes designs into a set of embeddings, and then generates predictions $\hat{\boldsymbol{y}}$ with a multilayer perceptron (MLP) based decoder. Figure 2 depicts the overall diagrams of our model. Let $\boldsymbol{y}$ indicate the ground-truth targets (i.e., perf, util-BRAM, util-DSP, util-FF, and util-LUT). Our objective is to minimize the loss function that measures the mean squared error (MSE) between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$, i.e., $\mathcal{L}_{\text{task}} = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||^2$.

Since one modality is the source code sequence, and the other is the HARP graph, it is natural to adopt a transformer model on C and a GNN model on G, which produce token representations $\{\boldsymbol{h}_j \in \mathbb{R}^d | j \in \{1, \ldots, I\}\}$ via the transformer's self-attention mechanism, and node representations $\{\boldsymbol{h}_k \in \mathbb{R}^d | k \in \{1, \ldots, |V|\}\}$ via the message passing mechanism, respectively. d denotes the embedding dimension. The starting token c_1's embedding is then taken as the source code summary, $\boldsymbol{h}_{\text{src}} \in \mathbb{R}^d$, and a graph-level aggregation can be performed on the node embeddings serving as the graph summary, $\boldsymbol{h}_{\text{graph}} \in \mathbb{R}^d$. The encoder outputs the concatenation of the two modalities summaries, concat$(\boldsymbol{h}_{\text{src}}, \boldsymbol{h}_{\text{graph}})$, and lets the MLP-based decoder generate predictions.

This model serves as the foundation of our architecture. However, it solely relies on the MLP-based decoder to manage the interaction between the two modalities. We denote this simplified version of our model as ProgSG-ca.

3.2 ProgSG-si: Graph-Summary-Augmented Sequence Representation

One limitation of the ProgSG-ca encoder is the shallow and ineffective modeling of the interaction between C and G. We propose a novel yet simple way to address the issue, by making the following observation: The transformer operates on the sequence of tokens $C = (c_1, \ldots, c_I)$ by enabling every token to pay attention to every other token. That is,

$$\boldsymbol{h}_{\text{src}} = AGG(g_{\text{att}}(\boldsymbol{h}_{c_1}^{(0)}, \ldots, \boldsymbol{h}_{c_I}^{(0)})) \tag{1}$$

where AGG can be any aggregation function, and g_{att} denotes the multi-layer self-attention encoder of a transformer model, capturing the interaction between pairwise source code tokens, $\boldsymbol{h}_{c_j}^{(0)}$ stands for the j-th token's initial embedding[3], $\boldsymbol{h}_{\text{src}}$ denotes the final program-level source code embedding.

Based on the above observation, we propose to insert the graph summary $\boldsymbol{h}_{\text{graph}}$ to the beginning of the sequence, forming an augmented sequence representation $C^{(\text{aug})} = (\boldsymbol{h}_{\text{graph}}, c_1, \ldots, c_I)$ as input to the transformer[4]. Overall,

$$\boldsymbol{h}_{\text{src}} = AGG(g_{\text{att}}(\boldsymbol{h}_{\text{graph}}, \boldsymbol{h}_{c_1}^{(0)}, \ldots, \boldsymbol{h}_{c_I}^{(0)})). \tag{2}$$

We name such an encoder as ProgSG-si (Summary Interaction), since it first performs GNN with L_1 layers on G to obtain a summary, and let the expressive transformer of L_2 layers handle the pairwise attention between tokens and that summary embedding, which efficiently allows cross-modality interaction. In other words, the graph is treated as a derivative of the source code whose summary embedding is used to augment the source code sequence. During training, the gradients back-propagate through $\boldsymbol{h}_{\text{graph}}$ to the GNN, updating both the GNN and the transformer.

3.3 Full Model ProgSG: Leveraging Fine-grained Node Token Interaction

While ProgSG-si enables interaction between the graph and tokens, the graph-level summary is too coarse for the model to fully exploit the information from both modalities. Intuitively, the information exchange between two modalities would be more effective if the interaction happens in node/token level. A straightforward way is to utilize a cross attention module to all node embeddings $\boldsymbol{h}_{v_1}, \ldots, \boldsymbol{h}_{v_{|V|}}$ and token embeddings $\boldsymbol{h}_{c_1}, \ldots, \boldsymbol{h}_{c_I}$. However, since there could be thousands of nodes and tokens for an HLS design, the computation overhead is too expensive. So a more efficient way to leverage fine-grained node token interactions is needed.

Recall that there are auxiliary nodes in the HARP graph that stand for the LLVM-IR blocks (see Sec 2.2 for more details). Meanwhile, the source code is segmented into multiple chunks so that the length of each chunk is within the input length limit of the transformer. Let $\boldsymbol{h}_{v_{a_1}}, \ldots, \boldsymbol{h}_{v_{a_N}}$ denote the embeddings of auxiliary block nodes and let $\boldsymbol{h}_{c_{s_1}}, \ldots, \boldsymbol{h}_{c_{s_M}}$ indicate the embeddings of the summary tokens in source code chunks. Since the auxiliary block nodes in the graph modality and the chunks of source codes provide an intermediate granularity between graph/program and node/token level, we propose to utilize them to conduct a hierarchical node/token interaction, which is illustrated in Figure 3. The information between two modalities are first exchanged between the block nodes and the summary tokens via the following cross-modality message passing mechanism inspired by message passing GNNs:

$$\begin{aligned} \boldsymbol{h}'_{v_{a_k}} &= \boldsymbol{h}_{v_{a_k}} + \text{MLP}_2\Big(\textstyle\sum_j \alpha_{k,j} \text{MLP}_1(\boldsymbol{h}_{c_{s_j}})\Big), \\ \boldsymbol{h}'_{c_{s_j}} &= \boldsymbol{h}_{c_{s_j}} + \text{MLP}_4\Big(\textstyle\sum_k \alpha_{j,k} \text{MLP}_3(\boldsymbol{h}_{v_{a_k}})\Big), \end{aligned} \tag{3}$$

where the attention coefficients are computed via a dot product attention with learnable weight matrices $\boldsymbol{W}_1 \in \mathbb{R}^{d \times d}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{d \times d}$, $\alpha_{k,j} = \text{Softmax}\left(\frac{(\boldsymbol{W}_1 \boldsymbol{h}_{v_k})^\top (\boldsymbol{W}_2 \boldsymbol{h}_{c_j})}{\sqrt{d}}\right)$.

[3] This is usually implemented by looking it up in a dictionary that maps each token ID into a d-dimensional learnable vector representing the initial embeddings.
[4] This is equivalent to augmenting the initial embedding lookup dictionary with a special token initialized as the output of a GNN.

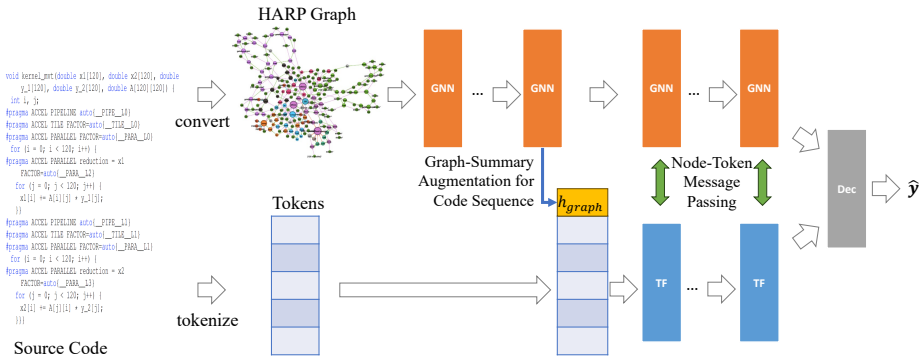


Figure 2: The overall diagrams of ProgSG. "GNN", "TF", and "Dec" refer to Graph Neural Network Layer, Transformer Layer, and Decoder, respectively.

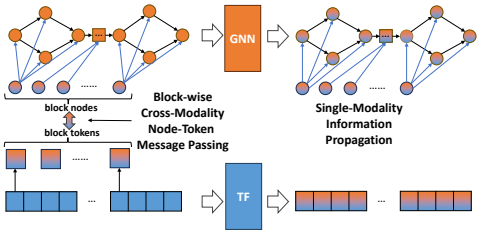


Figure 3: Illustration of the node-token message passing mechanism. The cross-modality information is first exchanged via block nodes and block tokens. Then the information is propagated to normal nodes and tokens through the GNN and transformer layers, respectively.

Then, the exchanged information is propagated to each node and token via a GNN and a transformer layer, respectively. Specifically, for a node v_i (token c_i) that is not a block node (summary token), let $\boldsymbol{h}'_{v_i} = \boldsymbol{h}_{v_i}$ ($\boldsymbol{h}'_{c_i} = \boldsymbol{h}_{c_i}$), the second step can be written as

$$\begin{aligned} \boldsymbol{h}''_{v_i} &= \boldsymbol{h}'_{v_i} + \mathrm{MLP}_6\Big(\textstyle\sum_j \alpha_{i,j}\mathrm{MLP}_5(\boldsymbol{h}'_{v_j})\Big) \\ \boldsymbol{h}''_{c_1}, \ldots, \boldsymbol{h}''_{c_I} &= Attention(\boldsymbol{h}'_{c_1}, \ldots, \boldsymbol{h}'_{c_I}) \end{aligned} \tag{4}$$

Such cross-modality interaction enables ***fine-grained*** interaction between the two modalities so that more informative embeddings for the final prediction task can be generated. As an additional benefit, the interaction step is significantly more efficient than the full cross-attention because the number of auxiliary nodes and summary tokens is usually small. To allow ***deep*** cross-modality interaction, we perform the above node-token message passing L_2 times where L_2 is the number of transformer layers, e.g., 6 for the pre-trained CodeT5 model used in our experiments. In each of the L_2 layers, ProgSG performs the self-attention encoder on $C^{(aug)}$, and executes GNN on G, followed by the node-token interaction.

3.4 Pretraining GNNs for Graph Modality

Generating ground-truth targets with an HLS simulator is slow, resulting in a scarcity of labeled data. To mitigate this issue, we propose utilizing pre-training tasks. While there is extensive work on pre-training transformer models with code [10, 19, 47], our focus is on pre-training GNNs for graph modality. Existing self-supervised tasks for GNNs are for general graphs instead of CDFG; thus, we propose employing data flow analyses as self-supervised learning tasks. Data flow analysis is fundamental to modern compiler technology [7] and necessitates that GNNs extract crucial information from a program's structure. Furthermore, these tasks can be effectively addressed by non-ML techniques, allowing us to easily obtain a substantial set of labeled data for pre-training.

In particular, we select four data analyses tasks: (1) reachability: if a node can be reached from another node, (2) dominators: if every control-flow path to an instruction node passes through another node, (3) data dependencies: if a variable is defined in an instruction and used in another instruction, and (4) liveness: if a variable is live-out of a statement n. More detailed definitions of these tasks can be found in [7]. These tasks cover a full range of forward and backward analyses, and control and data analyses. In addition, these tasks focus on predicting the relationship between two nodes in a CDFG. Such node-level tasks help the GNN to learn meaningful node embeddings, which is the foundation of generating good graph embeddings. Each task can be viewed as a binary classification problem. Given a pair of nodes v_i, v_j and a label y_{ij} which is a binary label indicating if the nodes have a particular relationship, we employ the cross entropy loss for pre-training loss.

Normally after pre-training, we would directly fine-tune the pre-trained GNN for the downstream task. However, the pre-training dataset does not contain any pragma nodes, which is important for predicting the quality of the HLS design. Therefore, we propose to use the pre-trained node embeddings as guidance to train a new (target) GNN for the downstream task. Specifically, given a graph with pragma nodes, denoted as G, we would generate a corresponding graph without pragma nodes, designated as G'. Then for a node v that appears in both G and G', we would compute its embedding in G' with the pre-trained GNN and compute its embedding in G with the GNN to be trained. Then, we would maximize the cosine similarity between the two embeddings with the following loss $\mathcal{L}_{\text{guide}} = 1 - \cos\langle g_{\text{cont}}(\boldsymbol{h}_{v,G}), \boldsymbol{h}_{v,G'}\rangle$ where g_{cont} is a continuous function (e.g., MLP, identity function). In this way, the target GNN would learn how to extract useful node-level information from the pre-trained GNN, which would in turn improve the quality of graph-level embeddings.

4 Experiments

Here we present the main experiment results. Additional experiments and settings are provided in the Appendix C.

4.1 Dataset

We assembled a database of medium-complexity kernels that function as fundamental building blocks for larger applications. We selected a total of 42 kernels from two well-known benchmark suites, namely, the MachSuite benchmark [35] and the Polyhedral benchmark suite (Polybench) [56]. The kernels in the database were chosen to have a broad range of computation intensities, including linear algebra operations on matrices and vectors (e.g., BLAS kernels), data mining kernels (e.g., CORRELATION and COVARIANCE), stencil operations, encryption, and a dynamic programming application. The database is a new version of datasets released in [4], generated by the AMD/Xilinx HLS tool version 2021 to implement the design, with the AMD/Xilinx Alveo U200 as the target FPGA and a working frequency of 250MHz. For each kernel, we perform a random split with the training, validation, and testing ratio being 70:15:15. For each design point, we recorded the `latency` in terms of cycle counts, as well as the resource utilization for DSP, BRAM, LUT, and FF. These targets are normalized following the same procedure in [4, 39]. The statistics of the dataset are presented in Table 3 in the Appendix. The dataset are released publicly[5].

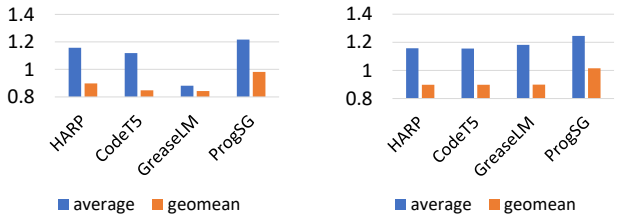
4.2 Performance Prediction Results

We compare the accuracy of performance prediction of PROGSG against three categories of baselines: (1) models of source code modality, CODE2VEC [2] and CODET5 [47]; (2) models of graph modality, HARP [40]; and (3) models of two modalities, GREASELM [57]. We also include PROGSG-CA, which is a simple concatenation of the summary representations described (Section 3.1), and PROGSG-SI which combines the two modalities without fine-grained interaction (Section 3.2).

Table 2 provides a detailed breakdown of the prediction accuracy across different target variables. Notably, our results consistently reveal that the cross-modality model outperforms the single-modality model in terms of rooted mean square error (RMSE). This finding strongly supports our argument for the benefits of integrating multiple modalities within our model architecture. Furthermore, the comparison between the error rates of PROGSG-SI and PROGSG-CA highlights the effectiveness of our graph-summary-augmented sequence representation. Moreover, PROGSG surpasses PROGSG-CA, PROGSG-SI, and GREASELM. This outcome underscores the superiority of our fine-grained node token interaction module, enabling more accurate predictions across a diverse range of target variables. In summary, our experimental results validate the effectiveness of our novel cross-modality program encoder.

4.3 Design Space Exploration Results

In addition, we evaluate how our method performs in finding the best design of a given kernel, i.e., design space exploration. Following previous studies [39, 40], for each kernel we have each model to verify as many designs points as possible in an hour following a heuristic order. The design points with the top 10 predicted performance is recorded. Then we run an HLS simulation to get the ground-truth performance of the selected design points and compare them with the best design point found by running AutoDSE [41] for 25 hours. We use the average speedup between design points found by the model and those found by AutoDSE as the metric to evaluate the performance of each model on DSE task. Figure 4a shows the average and geomean of the DSE performance of HARP, CODET5, GREASELM, and PROGSG. PROGSG outperforms all the baselines, revealing that our cross-modality model is superior. In addition, GREASELM and CODET5 are worse than HARP in the DSE task, though it has a smaller RMSE in the regression task. We think it is because running CODET5 and GREASELM for inference is about $1-2\times$ slower than running HARP, as the GNN is much smaller. As a result, the number of designs verified by CODET5 is smaller. Meanwhile, although PROGSG also suffers from the slow inference, it still manages to find better design points due to its better prediction accuracy.



(a) Running each model for one hour. **(b) Running each model on 1K candidates returned by HARP.**

Figure 4: Relative performance improvement of best design found by our model compared to running AutoDSE for twenty-five hours.

One way to handle the slow inference speed of CODET5 and PROGSG is to do a two-level design space exploration. That is, HARP is first run for an hour to find 1,000 candidate designs with the best predicted performance, then the larger model (CODET5 or PROGSG) is used to select the top 10 designs from them. This two-level approach can simultaneously utilize the efficiency of the GNN model and the effectiveness of larger cross-modality model. Figure 4b shows the DSE performance of this two-level approach. It is clear that the performance of CODET5 and PROGSG are significantly improved, showing the advantage of the two-level design space exploration. However, CODET5 still cannot outperform HARP, demonstrating that LLM itself might not be powerful enough for our task.

4.4 Training with Multiple Versions of Data

In addition, HARP [40] revealed that training the model with data obtained through multiple versions of HLS tools can improve the performance of the model. In their experiments, HARP is first trained with data of one version, then fine-tuned with data of another version. To investigate if the conclusion is true for PROGSG, we conduct a similar experiment with data of three different versions using HARP and PROGSG. Each model is first trained with

[5] https://github.com/ZongyueQin/ProgSG

Table 2: Rooted mean square error (RMSE) of different methods in predicting target values.

		perf	util-LUT	util-FF	util-DSP	util-BRAM	total
Single Modalities Model	Code2vec	1.0641	0.5462	0.3103	0.9989	0.1555	3.6150
	HARP	0.2671	0.1043	0.0565	0.1584	0.0611	0.6474
	CodeT5	0.2077	0.0985	0.0619	0.1881	0.0597	0.6159
Cross Modalities Model	GreaseLM	0.2033	0.0805	0.0499	0.1349	0.0459	0.5146
	ProgSG-ca	0.2181	0.1232	0.0532	0.1381	0.0334	0.5660
	ProgSG-si	0.1591	0.1630	0.0514	0.1558	0.0335	0.5628
	ProgSG	**0.1481**	**0.0709**	**0.0406**	**0.1084**	**0.0242**	**0.3923**

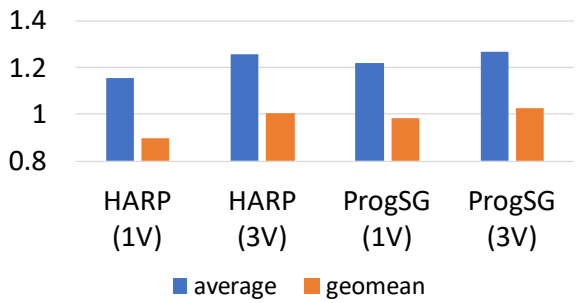


Figure 5: DSE results of HARP and ProgSG trained with 1 version (v21, denoted as 1V) and three versions (v18, v20, and v21, designated as 3V) of HLS tools.

data of the first version (HLS v18) for 1,000 epochs, then fine-tuned with the data of the second version (HLS v20) for 200 epochs, and finally fine-tuned with the data of third versions (HLS v21) for 400 epochs. Figure 5 illustrates the DSE performance of HARP and ProgSG trained with 1 version and 3 versions of data. It is clear that training ProgSG with multiple versions of data significantly increases its performance.

5 Related Work

Machine Learning for Electronic Design Automation Machine learning (ML) for electronic design automation (EDA) is a rising research area [16] with applications at various stages of hardware design, such as design verification [25, 44, 53], high-level synthesis (HLS) [5, 12, 39, 40, 43, 48], circuit design [36, 45, 46, 54], etc. This work focuses on obtaining representations of HLS designs using information from both the source code and the CDFG graph for FPGA design quality regression. Many works depict the input design/circuit as graphs [36, 39, 43]. Recently, large language models (LLMs) are used to directly generate EDA scripts [27, 28]. However, their results show that LLMs can only generate a few lines of scripts without considering the quality of the design. This work is among the first to combine both the source code and the graph modalities.

Representation Learning for Programs Based on the modality of data, current methods can be divided into source-code-based methods and data-structure-based methods. Source-code-based methods [10, 19, 42, 47] employ language models [9, 11, 14, 23, 32, 33, 37, 59] on source code to perform various types of tasks. However, it has not been demonstrated that these language models can predict the program's runtime, let alone predicting the corresponding hardware design performance. The data-structure-based methods [2, 39, 40] obtain the program embeddings from the data structure that represents a program. But the sizes of the models are usually small, restricting their prediction ability.

Multi-modal Learning with Transformers Modality-wise, transformers have been employed in cross-modality tasks spanning across vision [17, 22, 50], language [21, 58], source code [8], knowledge graphs [34, 55], audio [3, 13], point clouds [1], etc. In fact, multi-modal learning using transformers has recently been considered possible for achieving generalist artificial intelligence [30, 31]. More thorough surveys on graphs and transformers can be found at Jin et al. [18], Li et al. [24], Liu et al. [26]. GreaseLM [57] combines GNN and transformer for knowledge-graph augmented QA tasks, focusing on integrating graph and text modalities. However, our task presents distinct challenges, such as the need for fine-grained interactions due to subtle differences in program structures, the importance of efficiency given larger program-derived graphs, and the hierarchical nature of programs. To address these, we propose a novel model that interacts modalities at both global and block levels, balancing effectiveness and efficiency, making us the first to explore cross-modality models for program representation learning.

6 Conclusion

We propose ProgSG, a novel two-modality program representation learning method for IC design (defined with HLS C/C++) optimization. The key assumption is that there is critical information in both the source code modality and the assembly code modality, which must be captured jointly. To achieve that, we propose a graph-summary-augmented sequence representation for the source code transformer, a fine-grained alignment utilization method, and a novel pre-training method for the GNN encoder for the CDFG. Experiments confirm the superiority of the proposed ProgSG over baselines. We believe the core idea of using both modalities together with their alignment is general and can be adapted for other tasks.

7 Acknowledgement

This work was partially supported by NSF grants 2211557, 1937599, 2119643, and 2303037, SRC JUMP 2.0 PRISM Center, NASA, Okawa Foundation, Amazon Research, Cisco, Picsart, Snapchat, and the CDSC industrial partners (https://cdsc.ucla.edu/partners/). The authors would also like to thank Maria Brbic (EPFL) for early discussions on integrating GNN and LLM models, AMD/Xilinx for HACC equipment donation, and Marci Baun for editing the paper. J. Cong has a financial interest in AMD.

References

[1] Mohamed Afham, Isuru Dissanayake, Dinithi Dissanayake, Amaya Dharmasiri, Kanchana Thilakarathna, and Ranga Rodrigo. 2022. Crosspoint: Self-supervised cross-modal contrastive learning for 3d point cloud understanding. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 9902–9912.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3, POPL (2019), 1–29.

[3] Relja Arandjelovic and Andrew Zisserman. 2017. Look, listen and learn. In Proceedings of the IEEE international conference on computer vision. 609–617.

[4] Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, and Jason Cong. 2023. Towards a Comprehensive Benchmark for High-Level Synthesis Targeted to FPGAs. In Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track.

[5] Yunsheng Bai, Atefeh Sohrabizadeh, Yizhou Sun, and Jason Cong. 2022. Improving GNN-based accelerator design automation with meta learning. In Proceedings of the 59th ACM/IEEE Design Automation Conference. 1347–1350.

[6] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2015. Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289 (2015).

[7] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. 2021. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In International Conference on Machine Learning. PMLR, 2244–2253.

[8] Yong Dai, Duyu Tang, Liangxin Liu, Minghuan Tan, Cong Zhou, Jingquan Wang, Zhangyin Feng, Fan Zhang, Xueyu Hu, and Shuming Shi. 2022. One model, multiple modalities: A sparsely activated approach for text, sound, image, video and code. arXiv preprint arXiv:2205.06126 (2022).

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020).

[11] Weimin Fu, Shijie Li, Yifang Zhao, Haocheng Ma, Raj Dutta, Xuan Zhang, Kaichen Yang, Yier Jin, and Xiaolong Guo. 2024. Hardware Phi-1.5 B: A Large Language Model Encodes Hardware Domain Specific Knowledge. arXiv preprint arXiv:2402.01728 (2024).

[12] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. 2023. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD). IEEE, 1–9.

[13] Chuang Gan, Deng Huang, Hang Zhao, Joshua B Tenenbaum, and Antonio Torralba. 2020. Music gesture for visual sound separation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 10478–10487.

[14] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks Are All You Need. arXiv preprint arXiv:2306.11644 (2023).

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366 (2020).

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

[17] Zhenyu Huang, Guocheng Niu, Xiao Liu, Wenbiao Ding, Xinyan Xiao, Hua Wu, and Xi Peng. 2021. Learning with noisy correspondence for cross-modal matching. NeurIPS 34 (2021), 29406–29419.

[18] Bowen Jin, Gang Liu, Chi Han, Meng Jiang, Heng Ji, and Jiawei Han. 2023. Large Language Models on Graphs: A Comprehensive Survey. arXiv preprint arXiv:2312.02783 (2023).

[19] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119), Hal Daumé III and Aarti Singh (Eds.). PMLR, 5110–5121. https://proceedings.mlr.press/v119/kanade20a.html

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In International Symposium on CGO.

[21] Kenton Lee, Mandar Joshi, Iulia Turc, Hexiang Hu, Fangyu Liu, Julian Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. 2022. Pix2Struct: Screenshot parsing as pretraining for visual language understanding. arXiv preprint arXiv:2210.03347 (2022).

[22] Junnan Li, Ramprasaath Selvaraju, Akhilesh Gotmare, Shafiq Joty, Caiming Xiong, and Steven Chu Hong Hoi. 2021. Align before fuse: Vision and language representation learning with momentum distillation. NeurIPS 34 (2021), 9694–9705.

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! arXiv preprint arXiv:2305.06161 (2023).

[24] Yuhan Li, Zhixun Li, Peisong Wang, Jia Li, Xiangguo Sun, Hong Cheng, and Jeffrey Xu Yu. 2023. A survey of graph meets large language model: Progress and future directions. arXiv preprint arXiv:2311.12399 (2023).

[25] Rongjian Liang, Nathaniel Pinckney, Yuji Chai, Haoxin Ren, and Brucek Khailany. 2023. Late Breaking Results: Test Selection For RTL Coverage By Unsupervised Learning From Fast Functional Simulation. In 2023 60th ACM/IEEE Design Automation Conference (DAC). IEEE, 1–2.

[26] Jiawei Liu, Cheng Yang, Zhiyuan Lu, Junze Chen, Yibo Li, Mengmei Zhang, Ting Bai, Yuan Fang, Lichao Sun, Philip S Yu, et al. 2023. Towards graph foundation models: A survey and beyond. arXiv preprint arXiv:2310.11829 (2023).

[27] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. Chipnemo: Domain-adapted llms for chip design. arXiv preprint arXiv:2311.00176 (2023).

[28] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD). IEEE, 1–8.

[29] Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. ICLR (2019).

[30] Gengchen Mai, Weiming Huang, Jin Sun, Suhang Song, Deepak Mishra, Ninghao Liu, Song Gao, Tianming Liu, Gao Cong, Yingjie Hu, et al. 2023. On the opportunities and challenges of foundation models for geospatial artificial intelligence. arXiv preprint arXiv:2304.06798 (2023).

[31] Michael Moor, Oishi Banerjee, Zahra Shakeri Hossein Abad, Harlan M Krumholz, Jure Leskovec, Eric J Topol, and Pranav Rajpurkar. 2023. Foundation models for generalist medical artificial intelligence. Nature 616, 7956 (2023), 259–265.

[32] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. OpenAI blog 1, 8 (2019), 9.

[33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. The Journal of Machine Learning Research 21, 1 (2020), 5485–5551.

[34] Jiahua Rao, Zifei Shan, Longpo Liu, Yao Zhou, and Yuedong Yang. 2023. Retrieval-based Knowledge Augmented Vision Language Pre-training. arXiv preprint arXiv:2304.13923 (2023).

[35] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In IISWC.

[36] Haoxing Ren, George F Kokai, Walker J Turner, and Ting-Sheng Ku. 2020. ParaGraph: Layout parasitics and device parameter prediction using graph neural networks. In 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 1–6.

[37] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023).

[38] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. 2021. Masked label prediction: Unified message passing model for semi-supervised classification. IJCAI (2021).

[39] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated accelerator optimization aided by graph neural networks. In Proceedings of the 59th ACM/IEEE Design Automation Conference. 55–60.

[40] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2023. Robust GNN-Based Representation Learning for HLS. In 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD). IEEE, 1–9.

[41] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. ACM Transactions on Design Automation of Electronic Systems (TODAES) 27, 4 (2022), 1–27.

[42] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 1433–1443.

[43] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. 2020. Accurate operation delay prediction for FPGA HLS using graph neural networks. In Proceedings of the 39th International Conference on Computer-Aided Design. 1–9.

[44] Shobha Vasudevan, Wenjie Joe Jiang, David Bieber, Rishabh Singh, C Richard Ho, Charles Sutton, et al. 2021. Learning semantic representations to verify hardware

designs. NeurIPS 34 (2021), 23491–23504.
[45] Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. 2020. GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 1–6.
[46] Haoyu Peter Wang, Nan Wu, Hang Yang, Cong Hao, and Pan Li. 2022. Unsupervised Learning for Combinatorial Optimization with Principled Objective Relaxation. In NeurIPS.
[47] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. EMNLP (2021).
[48] Nan Wu, Yuan Xie, and Cong Hao. 2022. Ironman-pro: Multi-objective design space exploration in hls via reinforcement learning and graph neural network based modeling. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2022).
[49] Nan Wu, Hang Yang, Yuan Xie, Pan Li, and Cong Hao. 2022. High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing. In Proceedings of the 59th ACM/IEEE Design Automation Conference. 49–54.
[50] Peng Wu, Xiangteng He, Mingqian Tang, Yiliang Lv, and Jing Liu. 2021. Hanet: Hierarchical alignment networks for video-text retrieval. In Proceedings of the 29th ACM international conference on Multimedia. 3518–3527.
[51] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. arXiv preprint arXiv:2309.17453 (2023).
[52] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. ICML (2018).
[53] Peng Xu, Alejandro Salado, and Guangrui Xie. 2020. A reinforcement learning approach to design verification strategies of engineered systems. In 2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, 3543–3550.
[54] Tai Yang, Guoqing He, and Peng Cao. 2022. Pre-routing path delay estimation based on transformer and residual framework. In 2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 184–189.
[55] Michihiro Yasunaga, Antoine Bosselut, Hongyu Ren, Xikun Zhang, Christopher D Manning, Percy S Liang, and Jure Leskovec. 2022. Deep bidirectional language-knowledge graph pretraining. Advances in Neural Information Processing Systems 35 (2022), 37309–37323.
[56] Tomofumi Yuki and Louis-Noël Pouchet. [n. d.]. PolyBench/C. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[57] Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D Manning, and Jure Leskovec. 2022. Greaselm: Graph reasoning enhanced language models. In International conference on learning representations.
[58] Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. 2019. ERNIE: Enhanced language representation with informative entities. ACL (2019).
[59] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. arXiv preprint arXiv:2303.17568 (2023).

A Artifact Appendix

A.1 Abstract

The artifact contains the trained ProgSG model, the source code to run training and inference, the dataset used in our experiment, and the instruction to reproduce the experiment in Section 4.2.

A.2 Artifact check-list (meta-information)

- **Algorithm: ProgSG algorithm.**
- **Model: The trained ProgSG model is available for download.**
- **Data set: The dataset is released in the artifact**
- **Run-time environment: Linux, Python, Pytorch, transformers, etc. See README.md in github for details of installing necessary packages in python environment.**
- **Hardware: CPU, Nvidia-GPU**
- **Metrics: Rooted Mean Squared Error (RMSE)**
- **Output: command line output**
- **Experiments: See README for details**
- **How much disk space required (approximately)?: 10G**
- **How much time is needed to prepare workflow (approximately)?: 2 hours**
- **How much time is needed to complete experiments (approximately)?: less than 1 hour, depends on the GPU.**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: Apache-2.0**
- **Data licenses (if publicly available)?: Apache-2.0**
- **Archived (provide DOI)?: No**

A.3 Description

The artifact is available on github (https://github.com/ZongyueQin/ProgSG), the trained model can be downloaded from google drive (see github README for download link).

You should have a GPU with enough GPU memory to run our model. However, our model can also run in CPU-only environment with much longer running time.

You need to install python with packages including Pytorch, transformers, and others. See README in our github repo for details of installing the environment.

A.4 Installation

First, download our code from github. Second, download our model from google drive and decompress it in the `src/logs` folder. Third, install necessary packages in your python environment.

A.5 Experiment Workflow

Simply run `python main.py -force_regen True` to reproduce our experiments.

A.6 Evaluation and Expected Results

The training and test RMSE will be printed out in command line. The RMSE result should be close or lower than the number reported in Table 2.

B Additional Background in HLS Design

```
void kernel_mvt(double x1[120], double x2[120], double y_1[120], double
      y_2[120], double A[120][120]) {
  int i, j;
#pragma ACCEL PIPELINE auto{__PIPE__L0}
#pragma ACCEL TILE FACTOR=auto{__TILE__L0}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L0}
  for (i = 0; i < 120; i++) {
#pragma ACCEL PARALLEL reduction = x1 FACTOR=auto{__PARA__L2}
    for (j = 0; j < 120; j++) {
      x1[i] += A[i][j] * y_1[j];
    }}
#pragma ACCEL PIPELINE auto{__PIPE__L1}
#pragma ACCEL TILE FACTOR=auto{__TILE__L1}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L1}
  for (i = 0; i < 120; i++) {
#pragma ACCEL PARALLEL reduction = x2 FACTOR=auto{__PARA__L3}
    for (j = 0; j < 120; j++) {
      x2[i] += A[j][i] * y_2[j];
    }}}
```

Code 1: Code snippet of the MVT kernel (Matrix Vector Product and Transpose) with its 8 pragmas starting with "`#pragma`".

C Supplementary Experiment Details

C.1 Model Hyperparameters and Training Details

During training, we combine the proposed loss functions including $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}} + \gamma_1 \mathcal{L}_{\text{fineAlign}} + \gamma_2 \mathcal{L}_{\text{coarseAlign}} + \gamma_3 \mathcal{L}_{\text{guide}}$, where γs are hyperparameters controlling the weight for the different loss terms. During inference, we apply the encoder-decoder architecture to obtain $\hat{\boldsymbol{Y}}$.

We set the maximum number of tokens to 64 for the tokenizer, and chunk each source code into multiple subsequences to handle the long input source code sequence. We leave the exploration using more advanced modeling for long sequences such as [51] as future work. Since the task is on the whole program level, for each subsequence, we use the final embedding of the initial token ("`[cls]`") as the summary of each subsequence (for ProgSG-si and ProgSG, an additional MLP is applied to project $\boldsymbol{h}_{\text{src}} = \boldsymbol{H}_{\text{src}}[0:1]$ from dimension 1024 to 512), and aggregate all summaries into a final sequence-level embedding (denoted as $\boldsymbol{h}_{\text{src}}$ in the main paper) which is fed into the decoder. For the two-modality models, the decoder receives the concatenation of $\boldsymbol{h}_{\text{src}}$ and $\boldsymbol{h}_{\text{CDFG}}$ as described in the main paper.

The decoder consists of 6 sequentially stacked layers that project the input to a scalar. If the model is of a single modality, the MLP decoder has hidden dimensions 512-256-128-64-32-16-1. If the model is of two modalities, the MLP decoder has hidden dimensions 1024-768-512-256-128-61-1. The above scheme is administered consistently to all the methods for a fair comparison. Since we have 5 target metrics to predict as mentioned in Section 4.1 of the main paper, we use 5 MLPs applied on the input embeddings to transform them into the final $\hat{\boldsymbol{y}} \in \mathbb{R}^5$. We use the Exponential Linear Unit (ELU) function [6].

Our framework is implemented with PyTorch, PyTorch Geometric, Transformers, etc[6]. Training is performed on a server with NVIDIA Tesla V100 GPUs. We employ the AdamW optimizer [29] with the initial learning rate tuned for each model using a validation set. We perform training with $\gamma_1 = \gamma_2 = \gamma_3 = 1$ over 1000 epochs with the best model selected based on a validation set for final adaptation and testing.

For the pre-trained GNN, we use utilize GNN with 5 transformer convolutional layers [38] as encoders and a 2-layer MLP as the decoder for each data analysis task. We use a training set with 276,197 graphs. The β in focal loss is set to 2. We employ a validation set with 500 graphs to select the best pre-trained GNN. The β in focal loss is set to 2.

C.2 Model Setup and Hyperparameters

We follow [40] to generate the HARP graphs. We adopt $L_1 = 8$ layers of TransformerConv [38] with a jumping knowledge network [52] as the final node embedding aggregation method. The embedding dimension $d = 512$. For the source code, we use CodeT5 [47] with $L_2 = 6$ layers to embed the source code[7]. AutoDSE defines a variable for each pragma, as shown in Code 2, that is a placeholder for the option of the pragma. Since the pragmas ζ must be reflected in the input source code, for each design, we add the pragma options to their respective variables, e.g., we change "`__PARA__L0__`" to "`__PARA__L0=1`", "`__PIPE__L2`" to "`__PIPE__L2=flatten`", etc. We set the maximum number of tokens to 64 for the tokenizer, and chunk each source code into multiple subsequences to handle the long input source code sequence. The summaries of all subsequences are aggregated into the final representation for the decoder. We report the full hyperparameters in the appendix.

[6] https://github.com/ZongyueQin/ProgSG

[7] Specifically, we use CodeT5-small from https://huggingface.co/Salesforce/codet5-small to initialize the transformer encoder for source code, and fine-tune the whole model.

Table 3: Dataset statistics. "#D", "#P", "A#P", "A#T", "A#N", "A#E", and "A#MP" denote "# designs", "# programs", "avg # pragmas per design", "avg # tokens per program", "avg # nodes per program's CDFG", and "avg # edges per program's CDFG", respectively.

Dataset	#D	#P	A#P	A#T	A#N	A#E
Vitis 2021	10,868	40	8.1	1286.3	354.7	1246.4

Table 4: Effects of pre-training to the prediction RMSE of our model.

Targets	wo pretrain	with pretrain	relative impr.
perf	0.1387	0.1481	-6.8%
util-LUT	0.0830	0.0709	14.6%
util-FF	0.0461	0.0406	11.9%
util-DSP	0.1084	0.1022	9.97%
util-BRAM	0.0281	0.0242	13.9%
total	0.4163	0.3923	5.77%

C.3 Effects of Pre-training

In addition to our main analysis, we conducted an ablation study to delve deeper into the impact of our Graph Neural Network (GNN) pre-training strategy on model prediction accuracy. The results, as presented in Table 4, provide insightful observations. Notably, while there is a slight decrease in the accuracy of performance prediction, the prediction accuracy for the other four targets shows a notable improvement ranging from 10% to 15%. The drop of accuracy in performance is because we train the model to predict multiple targets simultaneously, and the accuracy of one target might drop while the overall prediction effectiveness improves. Furthermore, when considering the overall prediction accuracy, we observe an improvement of 5.57%. This substantial boost reaffirms the efficacy of our pre-training approach in refining the model's effectiveness across diverse prediction tasks.

C.4 Attention Visualization

To better understand if the transformer model learns to attend tokens that are relevant to HLS pragma configurations, we visualize the average attention scores of some of the pragma-related tokens in the Gemm-n kernel (shown in Code 2) for the transformer before and after training for our regression task (illustrated in Figure 6). We can see that 11 out of 15 tokens have higher attention scores after fine-tuning. For the 4 tokens that have lower attention

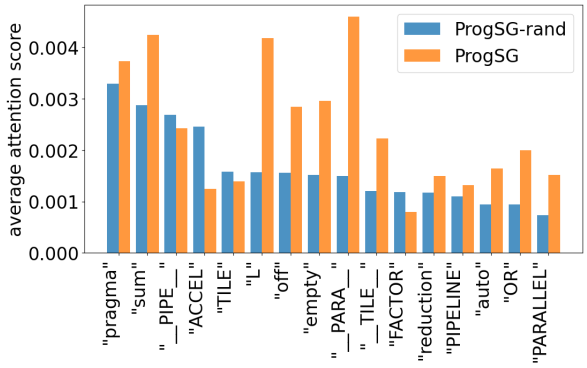


Figure 6: Bar plots of the average attention scores of pragma-related tokens before (ProgSG-rand) and after (ProgSG) being fine-tuned.

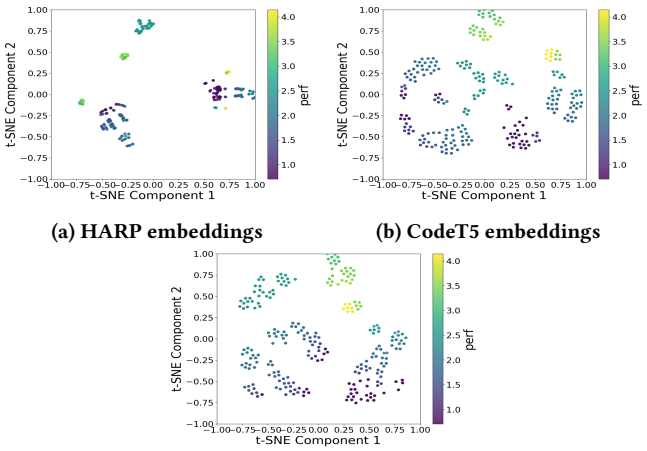
scores (i.e., "__PIPE__", "ACCEL", "TILE", and "FACTOR"), we can see that they often appear simultaneously with other keywords such as "PIPELINE", "__TILE__", and "PARALLEL", which makes them somewhat redundant. If we compute the summation of their attention scores with the attention scores of tokens that simultaneously appear with them (e.g., "__PIPE__" and "PIPELINE"), we find that the summed attention score increases after training. So the changes in attention score suggest that the transformer model does learn to attend to the pragma-related tokens, which are important to predicting the quality of an HLS pragma configuration, even though these tokens are not included in its pre-training stage.

```
void gemm_N(double m1[4096],double m2[4096],double prod[4096])
{
  int i,j,k,k_col,i_col;
  double mult;
#pragma ACCEL PIPELINE auto{__PIPE__L0}
#pragma ACCEL TILE FACTOR=auto{__TILE__L0}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L0}
for (i = 0; i < 64; i++) {
#pragma ACCEL PIPELINE auto{__PIPE__L1}
#pragma ACCEL TILE FACTOR=auto{__TILE__L1}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L1}
    for (j = 0; j < 64; j++) {
      i_col = i * 64;
      double sum = (double )0;
#pragma ACCEL PARALLEL reduction=sum FACTOR=auto{__PARA__L2}
      for (k = 0; k < 64; k++) {
        k_col = k * 64;
        mult = m1[i_col + k] * m2[k_col + j];
        sum += mult;
      }
      prod[i_col + j] = sum;
    }}}
```

Code 2: Code snippet of the Gemm-Ncubed kernel with its pragmas starting with "#pragma".

C.5 Embedding Visualization

To gain further insight into why ProgSG outperforms CodeT5 and HARP, we visualize the embeddings of valid "*correlation*" kernel designs in Figure 7. The colors represent the ground-truth performance targets. All methods form distinctive clusters with similar performance within each cluster. However, HARP's clusters are more crowded, likely due to the larger sizes of CodeT5 and ProgSG, which can better differentiate designs. Additionally, ProgSG's clusters align more closely with performance targets, as evidenced by the closer proximity of the purple points in ProgSG's embeddings compared to those in CodeT5's embeddings. This suggests that



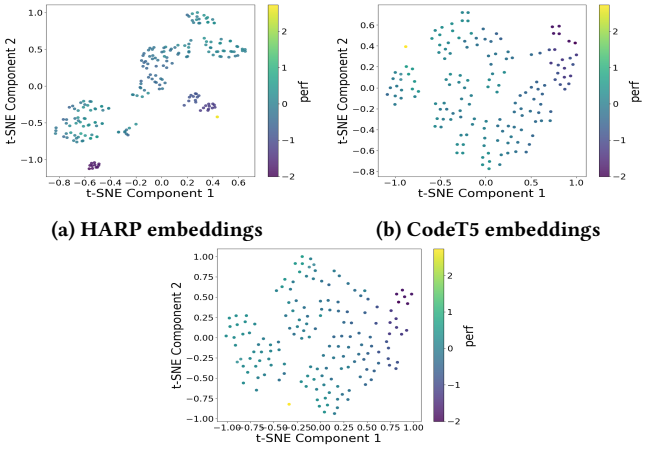
(a) HARP embeddings **(b) CodeT5 embeddings**

(c) ProgSG embeddings

Figure 7: Embedding visualizations with different methods for "*Correlation*" kernel using t-SNE. The color indicates the value of "perf" target.

ProgSG's embeddings more accurately reflect design performance, thereby explaining its superior DSE and prediction results.

In Figure 8 we visualize the embeddings of different models for "*symm-opt-medium*" kernel, where ProgSG achieves more than eight times speed up compared to HARP. Similar to the embeddings of "Correlation" kernel, the embeddings of HARP are more crowded, suggesting weaker generalization ability. Moreover, comparing the embeddings of CodeT5 and ProgSG, we can see that the yellow point (which represents the design point with the best performance) in ProgSG's embeddings are further away from other points than in CodeT5's embeddings. It suggests ProgSG can better distinguish the good design points.



(a) HARP embeddings **(b) CodeT5 embeddings**

(c) ProgSG embeddings

Figure 8: Embedding visualizations with different methods for "*Symm-Opt-Medium kernel*" using t-SNE. The color indicates the value of "perf" target.

C.6 Case Studies of Best Design Points in DSE experiments

In this section, we show the design points returned by AutoDSE (25 hours), HARP (1 hour), and ProgSG (1 hour) in the DSE experiments for some kernels ("Correlation", "Symm-opt-medium", "Gemver-medium") where ProgSG outperforms HARP significantly. We show the source code of these kernels in Code 3, Code 4, and Code 5. And we show the design points in Table 5, Table 6, and Table 7.

For *correlation* kernel, the values of "__PARA__L5" in AutoDSE and ProgSG's design points are much larger than the value in HARP's design point. So the design point returned by HARP leads to sub-optimal data loading procedures, which takes up 96,000 cycles of the total latency. While the design point returned by ProgSG does not have this issue. For *symm-opt-medium* kernel, the design point returned by HARP has smaller parallelization factor than the design points returned by AutoDSE and ProgSG, leading to worse efficiency. For *gemver-medium* kernel, the parameter "__PARA__L4" is 64 in the design point returned by HARP, which can not divide 400, which is the total number of the for-loop. As the result, the loop takes up 120,000 cycles of the total latency, leading to worse performance. On the other hand, The "__PARA__L4" is 25 in the design point returned by ProgSG, which can divide 400, thus avoiding the problem.

```
void kernel_correlation(double float_n,double data[100][80],double
      corr[80][80],double mean[80],double stddev[80])
{
  int i;
  int j;
  int k;
  double eps = 0.1;

#pragma ACCEL PIPELINE auto{__PIPE__L0}
#pragma ACCEL TILE FACTOR=auto{__TILE__L0}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L0}
  for (j = 0; j < 80; j++) {
    mean[j] = 0.0;
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L4}
    for (i = 0; i < 100; i++) {
      mean[j] += data[i][j];
    }
    mean[j] /= float_n;
  }
#pragma ACCEL PIPELINE auto{__PIPE__L1}
#pragma ACCEL TILE FACTOR=auto{__TILE__L1}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L1}
  for (j = 0; j < 80; j++) {
    stddev[j] = 0.0;
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L5}
    for (i = 0; i < 100; i++) {
      stddev[j] += pow(data[i][j] - mean[j],(double )2);
    }
    stddev[j] /= float_n;
    stddev[j] = sqrt(stddev[j]);
    stddev[j] = (stddev[j] <= eps?1.0 : stddev[j]);
  }
#pragma ACCEL PIPELINE auto{__PIPE__L2}
#pragma ACCEL TILE FACTOR=auto{__TILE__L2}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L2}
  for (i = 0; i < 100; i++) {
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L6}
    for (j = 0; j < 80; j++) {
      data[i][j] -= mean[j];
      data[i][j] /= sqrt(float_n) * stddev[j];
    }
  }
#pragma ACCEL PIPELINE auto{__PIPE__L3}
#pragma ACCEL TILE FACTOR=auto{__TILE__L3}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L3}
  for (i = 0; i < 80 - 1; i++) {
    corr[i][i] = 1.0;
#pragma ACCEL PIPELINE auto{__PIPE__L7}
    for (j = i + 1; j < 80; j++) {
      corr[i][j] = 0.0;
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L7_0}
      for (k = 0; k < 100; k++) {
        corr[i][j] += data[k][i] * data[k][j];
      }
      corr[j][i] = corr[i][j];
    }
  }
  corr[80 - 1][80 - 1] = 1.0;
}
```

Code 3: Code snippet of the Correlation kernel with its pragmas starting with "#pragma".

	AutoDSE	HARP	ProgSG
__PARA__L0	1	1	1
__PARA__L1	1	1	1
__PARA__L2	1	1	1
__PARA__L3	1	1	1
__PARA__L4	1	5	4
__PARA__L5	32	5	25
__PARA__L6	1	10	4
__PARA__L7_0	1	1	1
__PIPE__L0	fg	off	off
__PIPE__L1	off	off	off
__PIPE__L2	off	off	off
__PIPE__L3	off	off	off
__PIPE__L7	fg	fg	fg
__TILE__L0	1	1	1
__TILE__L1	1	1	1
__TILE__L2	1	1	1
__TILE__L3	1	1	1
perf	60,237	165,135	61,287

Table 5: Best design points returned by AutoDSE, HARP, and ProgSG on "*Correlation*" kernel.

```
void kernel_symm(double alpha,double beta,double C[200][240],double
      A[200][200],double B[200][240])
{
  int i,j,k;
#pragma ACCEL PIPELINE auto{__PIPE__L0}
#pragma ACCEL TILE FACTOR=auto{__TILE__L0}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L0}
  for (i = 0; i < 200; i++) {
#pragma ACCEL PIPELINE auto{__PIPE__L1}
#pragma ACCEL TILE FACTOR=auto{__TILE__L1}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L1}
    for (j = 0; j < 240; j++) {
      double tmp = B[i][j];
#pragma ACCEL PARALLEL reduction=C FACTOR=auto{__PARA__L2}
      for (k = 0; k < 200; k++) {
        if (k < i) {
          C[k][j] += alpha * tmp * A[i][k];
        }
      }
      double temp2 = (double )0;
#pragma ACCEL PARALLEL reduction=temp2 FACTOR=auto{__PARA__L3}
      for (k = 0; k < 200; k++) {
        if (k < i) {
          temp2 += B[k][j] * A[i][k];
        }
      }
      C[i][j] = beta * C[i][j] + alpha * B[i][j] * A[i][i] + alpha * temp2;
    }
  }
}
```

Code 4: Code snippet of the Symm-opt-medium kernel with its pragmas starting with "#pragma".

	AutoDSE	HARP	ProgSG
__PARA__L0	1	1	1
__PARA__L1	1	1	1
__PARA__L2	1	1	1
__PARA__L3	1	8	8
__PARA__L4	2	64	25
__PARA__L5	1	10	10
__PARA__L6	25	20	25
__PIPE__L0	off	off	off
__PIPE__L1	fg	off	cg
__PIPE__L3	off	cg	off
__TILE__L0	1	1	1
__TILE__L1	1	1	1
__TILE__L3	8	1	1
perf	210,335	265,686	167,270

Table 7: Best design points returned by AutoDSE, HARP, and ProgSG on "*Gemver-medium*" kernel.

	AutoDSE	HARP	ProgSG
__PARA__L0	1	1	1
__PARA__L1	1	1	1
__PARA__L2	25	25	25
__PARA__L3	200	32	200
__PIPE__L0	cg	off	cg
__PIPE__L1	off	cg	off
__TILE__L0	1	1	1
__TILE__L1	1	8	1
perf	4,345,927	35,536,546	4,345,927

Table 6: Best design points returned by AutoDSE, HARP, and ProgSG on "*Symm-OPT-Medium*" kernel.

```
void kernel_gemver(int n,double alpha,double beta,double A[400][400],double
      u1[400],double v1[400],double u2[400],double v2[400],double
      w[400],double x[400],double y[400],double z[400])
{
  int i,j;
#pragma ACCEL PIPELINE auto{__PIPE__L0}
#pragma ACCEL TILE FACTOR=auto{__TILE__L0}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L0}
  for (i = 0; i < 400; i++) {
#pragma ACCEL PARALLEL reduction=A FACTOR=auto{__PARA__L4}
    for (j = 0; j < 400; j++) {
      A[i][j] += + u1[i] * v1[j] + u2[i] * v2[j];
    }
  }
#pragma ACCEL PIPELINE auto{__PIPE__L1}
#pragma ACCEL TILE FACTOR=auto{__TILE__L1}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L1}
  for (i = 0; i < 400; i++) {
#pragma ACCEL PARALLEL reduction=x FACTOR=auto{__PARA__L5}
    for (j = 0; j < 400; j++) {
      x[i] += beta * A[j][i] * y[j];
    }
  }
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L2}
  for (i = 0; i < 400; i++) {
    x[i] = x[i] + z[i];
  }
#pragma ACCEL PIPELINE auto{__PIPE__L3}
#pragma ACCEL TILE FACTOR=auto{__TILE__L3}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L3}
  for (i = 0; i < 400; i++) {
#pragma ACCEL PARALLEL reduction=w FACTOR=auto{__PARA__L6}
    for (j = 0; j < 400; j++) {
      w[i] += alpha * A[i][j] * x[j];
    }}}
```

Code 5: Code snippet of the GEMVER-MEDIUM kernel with its pragmas starting with "`#pragma`".

Received 29 July 2024; revised 29 July 2024; accepted 14 Aug 2024