Graph Neural Networks for High-Level Synthesis Design Space Exploration

LORENZO FERRETTI, University of California, Los Angeles, USA
ANDREA CINI, IDSIA, Università della Svizzera italiana, Switzerland
GEORGIOS ZACHAROPOULOS, Computing Systems Lab, Huawei Zurich Research Center, Switzerland
CESARE ALIPPI, IDSIA, Università della Svizzera italiana, Switzerland
LAURA POZZI, Università della Svizzera italiana, Switzerland

High-level Synthesis (HLS) Design-Space Exploration (DSE) aims at identifying Pareto-optimal synthesis configurations whose exhaustive search is unfeasible due to the design-space dimensionality and the prohibitive computational cost of the synthesis process. Within this framework, we address the design automation problem by proposing graph neural networks that jointly predict acceleration performance and hardware costs of a synthesized behavioral specification given optimization directives. Learned models can be used to rapidly approach the Pareto curve by guiding the DSE, taking into account performance and cost estimates. The proposed method outperforms traditional HLS-driven DSE approaches, by accounting for the arbitrary length of computer programs and the invariant properties of the input. We propose a novel hybrid control and dataflow graph representation that enables training the graph neural network on specifications of different hardware accelerators. Our approach achieves prediction accuracy comparable with that of state-of-the-art simulators without having access to analytical models of the HLS compiler. Finally, the learned representation can be exploited for DSE in unexplored configuration spaces by fine-tuning on a small number of samples from the new target domain. The outcome of the empirical evaluation of this transfer learning shows strong results against state-of-the-art baselines in relevant benchmarks.

CCS Concepts: • **Hardware → Hardware description languages and compilation;**

Additional Key Words and Phrases: Design space exploration, high-level synthesis, graph neural networks

ACM Reference format:
Lorenzo Ferretti, Andrea Cini, Georgios Zacharopoulos, Cesare Alippi, and Laura Pozzi. 2022. Graph Neural Networks for High-Level Synthesis Design Space Exploration. *ACM Trans. Des. Autom. Electron. Syst.* 28, 2, Article 25 (December 2022), 20 pages.
https://doi.org/10.1145/3570925

Lorenzo Ferretti and Andrea Cini contributed equally to this research.

Authors' addresses: L. Ferretti, Engineering VI, 404 Westwood Plaza, Los Angeles, CA 90095; email: ferrelo@cs.ucla.edu; A. Cini, C. Alippi, and L. Pozzi, Università della Svizzera italiana, Via la Santa 1, 6962 Lugano, Svizzera; emails: andrea.cini@usi.ch, cesare.alippi@usi.ch, laura.pozzi@usi.ch; G. Zacharopoulos, Huawei Technologies, Zurich Research Center, Thurgauerstrasse 40, 8050 Zürich, Switzerland; email: giorgiozacharo@gmail.com.

1084-4309/2022/12-ART25 $15.00
https://doi.org/10.1145/3570925

1 INTRODUCTION

Traditional *Integrated Circuits (ICs)* design methodologies rely on *Hardware Description Languages (HDLs)* to describe logical components and their interaction at a *Register Transfer Level (RTL)*. This approach requires designers to manually define the concurrent description of millions of transistors, working in parallel to carry out the desired computations. To reduce the burden of this task, *High-Level Synthesis (HLS)* comes into play. HLS tools enable one to start the design process from high-level behavioral specifications in C/C++/SystemC, hence dispensing designers from the error-prone task of implementing the functionality at RTL level. Besides specifying the desired behavior, as shown in Figure 1, designers can guide the synthesis process by applying directives able to tune the resulting RTL implementation according to target performance and cost requirements. Synthesis directives specify how to implement, in hardware, specific software constructs such as loops, arrays, and functions. While HLS enables the exploration of a vast design space of architectural variations by using different directives, the resulting performance and resource utilization of each implementation cannot be determined *a priori*. In fact, exhaustive exploration involves time-consuming syntheses, whose number grows exponentially w.r.t. the number of applied directives. Moreover, among all the possible configurations (i.e., combinations of directives), only a few are Pareto-optimal from a performance and cost perspective. As designers are interested in effective methodologies to automate the *Design-Space Exploration (DSE)*, the HLS-driven DSE problem consists in identifying as accurately as possible the set of Pareto implementations while minimizing the number of syntheses.

Recent works demonstrated the possibility to guide DSEs by exploiting the notion of function similarity and the knowledge acquired from past explorations performed on different functions [14, 17, 45] and have shown promising results despite the small number of source domains. In this work, we introduce a methodological framework, based on a data-driven approach, which we believe can significantly open new research directions.

We propose both a data representation able to capture the critical elements of the HLS process and the data-processing tools to profit from such representation. At first, we introduce a novel graph representation of computer programs, based on an augmented hybrid *Control and Dataflow Graph (CDFG)*, capturing invariant properties and relevant information from the HLS perspective. Then, we exploit this representation to train a novel graph neural network model in a supervised fashion, by fitting the model on a dataset of previously synthesized configurations (behavioral specifications plus optimization directives) to predict the latency and resource utilization corresponding to each design. Our main contribution is the introduction of a novel methodology to use graph representation learning from software specifications to perform HLS-driven design space exploration. We show that the learned model can be used for effective DSE after fine-tuning on a small set of samples and that our method compares favorably against *State-of-the-Art (SoA)* baselines on relevant benchmarks. We refer to our framework as `gnn4hls`. We believe that the results achieved here constitute a strong signal for the community that calls for a general effort in collecting large datasets of syntheses, to unlock the full potential of graph deep learning solutions to the HLS-driven DSE problem. In light of this, *we claim that data-driven approaches are the way forward for HLS-driven DSE.*

The rest of the article is organized as follows. In Section 2, we define the problem by introducing the main concepts and terminology for HLS-driven DSE and graph neural networks. Then, in Section 3 we lay out the details of our approach. In Section 4, we evaluate the proposed method on relevant benchmarks. Finally, we discuss the related works in Section 5, draw our conclusions, and discuss future works in Section 6.

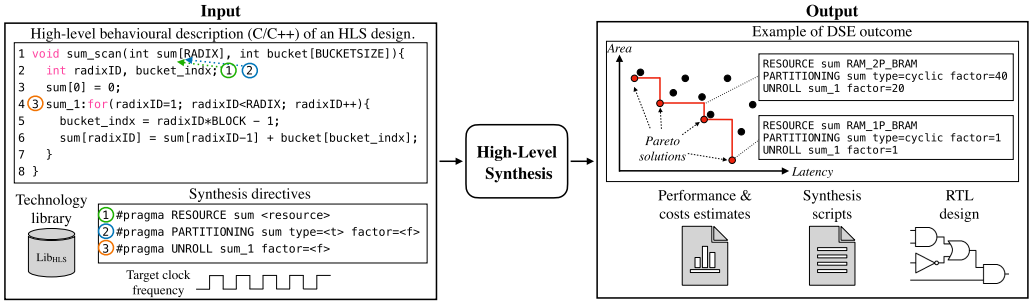

Fig. 1. Example of HLS design flow. A behavioral description, synthesis directives, technology library, and a target frequency are given to the HLS tool, which generates as a result an RTL design, performance and cost reports, as well as the synthesis scripts.

2 BACKGROUND

2.1 HLS-Driven Design-Space Exploration

Given a software functionality, e.g., the `sum_scan` function from the Radix Sort algorithm in Machsuite [29] (used as a running example in the rest of the document), we define as *HLS design*, or simply *design*, the functionality to be realized in hardware, and as *specifications* the behavioral description of the design in a high-level programming language such as C/C++. The specification (SW) is given in input to the HLS tool together with the target technology library (Lib_{HLS}) and a target frequency (F). The result of the synthesis process is named *implementation*, and it is an automatically generated RTL code, usually in VHDL or Verilog. The resulting RTL is coupled to a performance metric (e.g., latency or throughput), and a cost metric (e.g., area or energy costs).

An implementation is generated by applying a set of *directives*—specified using compiler pragmas—to SW, affecting the resulting performance and costs. The set of directives affecting an implementation is named *configuration*. Each directive is associated with a *target* in SW, which can be either a label or a code construct (e.g., labeled statements, function names, variables). In addition, a directive is characterized by its *type* and an associated *value*. The directive value forces a given directive type to a specific value. As an example, a `loop unroll` directive can be assigned an unrolling factor of 2 that doubles the logic required to implement in hardware the loop body, enabling parallel hardware execution of two iterations. The HLS design flow and the different elements characterizing it are shown in Figure 1.

Given a *design* D, the designer limits the set of configurations to explore during a DSE by defining a *configuration space*. The *configuration space* X_D is defined as the Cartesian product among the set of *directive values* V_i associated to each i-th *directive*, i.e., $X_D = V_1 \times V_2 \times \cdots \times V_N$, where N is the number of considered *directives*. The size of the *configuration space* is given by its cardinality $|X_D|$. Given a configuration space X_D, a *design space* Y_D can be defined as the set of *implementations* resulting from the synthesis of the configuration in X_D.

Task formulation. The DSE problem is a *Multi-Objective Optimization Problem (MOOP)* having costs and merit as objective functions. In the context of hardware design, common performance measure and cost are *latency*, and *area* or *power*, respectively. In this work, we target *Field Programmable Gate Array (FPGA)* and we use as performance the *effective latency* (LAT), i.e., the number of clock cycles required by the hardware implementation to execute its functionality multiplied by the estimated clock of the system. For costs, we consider the percentage of resource utilization required to implement the design on the FPGA resources, expressed in terms of

the number of *Flip-Flops (FFs)*, *Look-Up Tables (LUTs)*, *Digital Signal Processor (DSP)*, and *Block RAM (BRAM)*.[1] In particular, the objective is to identify a subset P_D of the configuration space X_D such that $P_D = \{x | x \in X_D$ and x is Pareto$\}$. A *Pareto configuration* (p) of a design D is defined as $p \in P \Leftrightarrow \nexists\, x \in X_D, x \neq p \mid a_x \leq a_p \wedge l_x \leq l_p$, with a_p, a_x, and l_p, l_x being the cost (a) and merit (l) associated to the implementation of p and x. respectively.

2.2 Graph Neural Networks

Input behavioral specifications (programs) significantly differ in terms of size, structure, constructs, and optimization directives. This variability is hardly captured by vector representations that have been, we argue, one of the main limiting factors of previous works in attempting to learn predictive models of the HLS process [17, 45]. Differently, when modeling the problem in the space of graphs, we can use methods that naturally exploit existing functional dependencies, account for the variability in the structure of different specifications, and seamlessly transfer learned representations across different configuration spaces. Furthermore, graph-processing techniques permit the exploitation of properties of graph representations (e.g., permutation invariance) inducing positive inductive biases that restrict the hypothesis space explored by the learning system to plausible models.

We consider attributed directed graphs, with attributes associated with both nodes and edges. In particular, a graph $\mathcal{G}$ is a tuple (V, E, u), where $V = \{1, \ldots, N\}$ is a set of nodes, $E = \{(i, j) | i, j \in V\}$ is a set of edges, and $u \in R^u$ is a global attribute vector. We denote by $v_i \in R^v$ the raw features associated with node $i \in V$ and with $e_{i,j} \in R^e$ the attribute vector associated with edge $(i, j) \in E$ connecting nodes i and j.

Different works propose general frameworks to design *Graph Neural Networks (GNNs)*: inspired by Gilmer et al. [15] and Battaglia et al. [2], we consider a very general class of *Message-Passing Neural Networks (MPNNs)* with global attributes where the t-th propagation layer (or step) can be written as

$$v_i^t = \tau_v^t \left(v_i^{t-1}, \mathtt{AGGR}_v \left\{ \psi_v^t \left(v_j^{t-1}, v_i^{t-1}, e_{j,i} \right); (j, i) \in E \right\}, u^{t-1} \right), \tag{1}$$

$$u^t = \tau_u^t \left(u^{t-1}, \mathtt{AGGR}_u \left\{ \psi_u^t \left(v_i^t, u^{t-1} \right); i \in V \right\} \right), \tag{2}$$

where the update (τ_v^t,τ_u^t) and message (ψ_v^t,ψ_u^t) functions can be implemented by any differentiable function, e.g., *Multi-Layer Perceptrons (MLPs)*. Aggregation functions $\mathtt{AGGR}_u\{\cdot\}$ and $\mathtt{AGGR}_v\{\cdot\}$ can be any permutation invariant operation. Multi-layer GNNs are built by stacking several propagation layers, which allow for aggregating, at each node, messages from different neighborhoods.

3 METHODS

In this section, we discuss in detail the components of the proposed approach. We start by describing and motivating in detail the proposed graph representation of the input programs; then, we present a graph neural architecture to process it and, finally, provide a straightforward DSE strategy to exploit the introduced components.

3.1 Data Representation

Control Flow Graphs (CFGs) represent the possible execution paths in a program. A CFG is a directed graph $\mathcal{G}_{\mathtt{CFG}}$ defined as the tuple $(V_{\mathtt{CFG}}, E_{\mathtt{CFG}})$, where $V_{\mathtt{CFG}}$ is the set of nodes corresponding to the basic blocks of the program, and $E_{\mathtt{CFG}}$ is the set of edges representing the possible control

[1]For the HLS designs considered in this work, the DSEs have not affected the number of BRAM; therefore, we have not included BRAM estimation in our experiments.

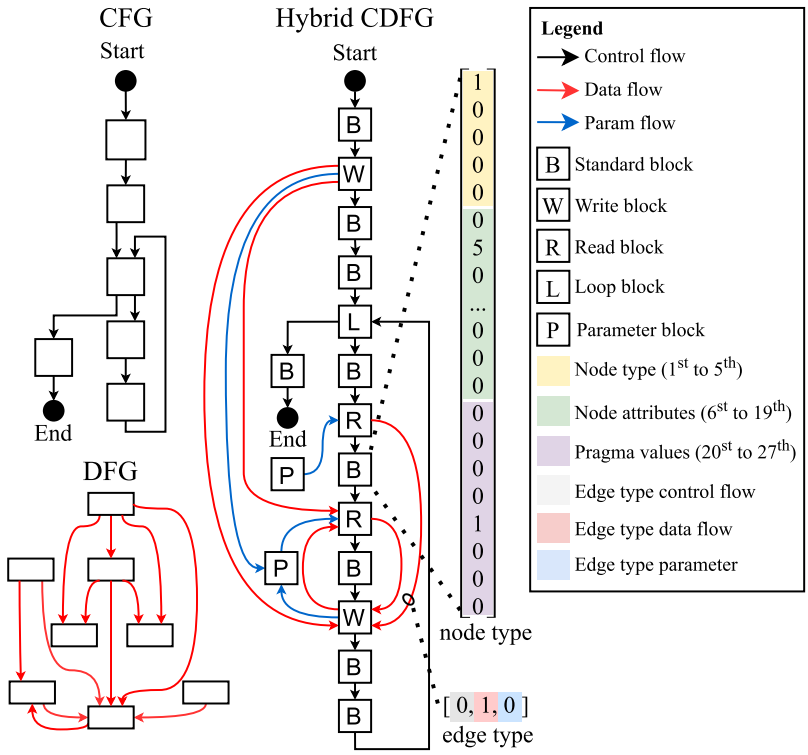

Fig. 2. Data representation. Control Flow Graph, Dataflow graph, and Hybrid Control Dataflow Graph representations of the behavioral specification of the function in Figure 1. CFG and DFG information are merged to obtain the Hybrid CDFG. Node attributes are obtained from LLVM IR, clang AST, and the original source code.

flows among basic blocks. An example of CFG for the `sum_scan` function is shown in Figure 2 (right). Traditionally, a basic block is defined as a consecutive sequence of instructions without incoming and outgoing branches except for the first and last instructions of the block, respectively. In our proposed formulation, we differentiate basic blocks according to the type of instructions they perform. We discriminate between the following types of blocks: *loop block* identifying basic blocks including loop instructions, *read block* including a single load instruction from main memory, *write block* including a single store instruction to main memory, *function block* including a function invocation instruction, and lastly *standard block* being basic blocks including instructions performing computations that do not belong to any of the above-mentioned categories. The effect of this representation and taxonomy affects the granularity of the CFG representation, increasing the number of blocks w.r.t.the traditional one. Figure 2 shows the differences of the proposed CFG representation—with a higher block granularity—w.r.t. the traditional one. This choice aims at avoiding the limitation of approaches relying on a vector-based representation of the program and directives [11, 12, 17], while, at the same time, focusing only on the information that is more relevant from the HLS and DSE perspectives. Notably, recent works highlight the effectiveness of adopting a similar taxonomy of basic blocks to capture similarities among designs in different configuration spaces [14].

In addition to the node type, attribute vectors associated with each node include block-type related information, such as the number of instructions in a block, number of iterations of a loop

block, presence of loop carried dependencies, loop stride, loop depth, loop trip count, number of function invocations, number of function parameters (including their types and bit-width), and function instructions are extracted through static and dynamic code analysis performed using custom LLVM [18] and Clang compiler passes.

While CFGs contain information about the execution flow of a program, they do not model the flow of data and information. *Dataflow Graphs (DFGs)* address this aspect. DFGs are used to represent dependencies between program instructions. In particular, they represent the use-def chains among variables in the program execution. The DFG is a directed graph $\mathcal{G}_{\texttt{DFG}}$ defined as a tuple $(V_{\texttt{DFG}}, E_{\texttt{DFG}})$, where $V_{\texttt{DFG}}$ is the a set of nodes corresponding to instructions in the source code, and $E_{\texttt{DFG}}$ is the set of edges representing the dataflow among instructions. The DFG representation for the running example function is shown in Figure 2 (left).

Hybrid Control Data Flow Graph. HLS tools use instrumented CFGs and DFGs as program representations to implement the design functionality in hardware. In our approach, we aim at using a similar representation as input of a learning method. We propose a tool-agnostic graph representation of the software description including both the CFG and the DFG information. In particular, we augment the CFG representation by adding dataflow edges and nodes representing the input and output parameters of the function. Dataflow edges are added among the nodes involving operations affecting the input and output parameters. These edges are identified by tracking the def-use chains among parameter variables in the DFG and embedding them in the CFG. In addition, edges among the parameter nodes and the CFG read and write blocks are added to the set of edges (*param flow*). We indicate this *Hybrid Control Dataflow Graph* as $\mathcal{G}_{\texttt{hls}}$ and we use it as input representation in our methodology. $\mathcal{G}_{\texttt{hls}}$ is a tuple $(V_{\texttt{hls}}, E_{\texttt{hls}}, u)$, where $V_{\texttt{hls}}$ is the a set of nodes corresponding to basic blocks and function parameters, and $E_{\texttt{hls}}$ is the set of attributed edges representing the control, data, and parameter flows. Each edge attribute vector $e_{i,j}$ is a three-dimensional feature vector with a one-hot encoding representation of the edge type. Lastly, $v_i \in V_{\texttt{hls}}$ is the feature vector associated with each node with a one-hot-encoding representation of the nodes type and their attributes, plus the value of optimization directives. Figure 2 (right) shows the Hybrid Control Dataflow Graph representations for the running example function.

$\mathcal{G}_{\texttt{hls}}$ nodes include block-specific information. This information is used to populate the node attribute vector. Each node has 27 attributes describing node type (5 elements), node attributes (14 elements), and pragma values (7 elements). Node-level information is extracted from the CFG and Clang parsing. To retrieve such information, we perform different analyses of the LLVM IR, the Clang *Abstract Syntax Tree (AST)*, and the source code. An LLVM pass generates the IR, which is used to identify basic blocks in the code and the traditional CFG representation. Then, for each instruction in the basic blocks, we check the type of operation performed and we discriminate among the different types of blocks. Basic blocks including store instructions are split separating the store instruction from the predecessor and successor instructions. The newly created store block is then connected to the new basic blocks resulting from the split. Read blocks are generated in the same way. Loop blocks are identified from the IR. Loop information is extracted in order to model loop carried dependencies, loop stride, and loop trip count. Similarly, call blocks are identified from the IR. In this case, information about the number of parameters, the number of function invocations, and the number of LLVM instructions of the invoked function is extracted and included in the attribute vector. From the remaining uncategorized standard blocks, we extract and add to the attribute vector information about the number of LLVM instructions. Parameter blocks instead are generated by combining information from the LLVM IR and the AST analysis. In particular, we extract information associated with the parameter type (pointer or value), parameter data type (the size of the data type), and the number of pointer parameters. This last information is required by the HLS tools in order to evaluate the memory required by the hardware accelerator. However,

since this information is not preserved in the LLVM IR, we extract it from a joint AST and source code analysis. Node pragma attributes include resource type (one or dual port BRAM associated), array partitioning pragmas (partitioning type and factor), loop unrolling pragmas (unrolling factor), and function inlining pragmas (enable, disabled). The list of the pragmas available for each benchmark is listed in Table 1. Node types are categorical attributes encoded by using a one-hot representation of the different block categories described above. Conversely, node attributes on the other hand are scalars and are as such included in the vector representation. Lastly, pragma values can be either numerical (e.g., unrolling factor) or categorical (partitioning type). Numerical values are used directly in the vector representation while categorical ones are represented by using one-hot encoding. To avoid large numerical differences among attributes, we use a logarithmic scale for the number of LLVM instructions, loop trip counts, array partitioning factors, data types, and their related directive values. Similarly, edge attributes have categorical attributes describing the edge type and these are expressed using one-hot encoding.

Figure 2 (right) shows the Hybrid Control Dataflow Graph representations for the running example function. The resulting attributed graph representation ($\mathcal{G}_{\texttt{hls}}$) allows for representing any program by exploiting graphs with an arbitrary number of nodes and topology where node attributes (of fixed length) encode the available knobs of the HLS implementation.

Global attribute. To provide further function-level information to the model, w.r.t. the target configuration space, we add a global (i.e., graph-level) attribute to our representation. Given a design and its associated configuration space, we define $\boldsymbol{s} \in \mathbb{R}^5$ as the vector having for each component the average among the directive values set associated with each pragma type. As an example, given a configuration space having directive value sets with two resource types (one-hot encoded), two partitioning types (one-hot encoded), partitioning factors of 1, 2, 4, 8, unrolling factors of 10, 20, 30, and two options for function inlining (one-hot encoded), the resulting configuration space vector will be $\boldsymbol{s} = [0.5, 0.5, 3.75, 20, 0.5]$. Then, given a specific configuration, we analogously generate the vector $\boldsymbol{c} \in \mathbb{R}^5$ as the average vector among directive values of the same type in that particular configuration. Similarly, we generate vectors $\boldsymbol{s'}$ and $\boldsymbol{c'}$ using the median among the directive set values instead of the mean. Finally, given the number of instructions l and the number of input parameters p, we define the global attribute vector $\boldsymbol{u}$ w.r.t. the considered configuration as

$$\boldsymbol{u} = l \parallel p \parallel (\boldsymbol{s} - \boldsymbol{c}) \parallel (\boldsymbol{s'} - \boldsymbol{c'}), \tag{3}$$

where $\parallel$ is the vector concatenation operation. Intuitively, this vector representation captures how much a configuration leans toward a region of the design space. In practice, the vectors $(\boldsymbol{s} - \boldsymbol{c})$ and $(\boldsymbol{s'} - \boldsymbol{c'})$ are normalized so that each element has unitary variance across the configuration space. The encoding $\boldsymbol{u}$ of an HLS design and its associated configurations allows for a unique representation of each implementation.

3.2 GNNs for HLS-Driven DSE

Given a graph $\mathcal{G}_{\texttt{hls}}$ representing a program annotated with optimization directives, we process it with message-passing neural networks by parametrizing the computation of messages exchanged between neighboring nodes of the graph with MLPs. In particular, we designed the architecture so that the global attribute characterizing each configuration is updated at each layer by aggregating information from the representations of the CDFG nodes through an attention mechanism. This approach avoids information bottlenecks caused by squashing the representations of each CDFG node in a single vector with a standard global pooling operator (as usually done in GNN architectures for graph-level regression). A schematic of the model is shown in Figure 3: we describe at first each computational block in detail and then discuss the training procedure.

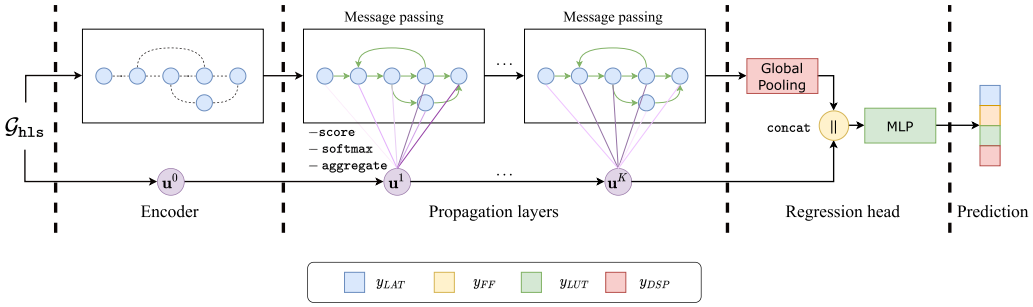

Fig. 3. gnn4hls. An encoder block maps the pre-process input representation with standard MLPs. Propagation layers perform message passing and update the global representation with multi-head attention layers. The final block maps the latent representation learned by the network into the regression targets.

Encoder. The Encoder block maps node, edge, and global graph features into a first hidden representation without performing any message-passing operation. The Encoder is implemented by using standard MLPs as update functions

$$v_i^0 = \texttt{MLP}_v^{\texttt{enc}}(v_i), \quad u^0 = \texttt{MLP}_u^{\texttt{enc}}(u), \quad e_{i,j}^{\texttt{enc}} = \texttt{MLP}_e^{\texttt{enc}}(e_{i,j}), \tag{4}$$

as using feed-forward layers before message passing has shown to be beneficial to final performance in GNNs [48]. The Encoder block is followed by a stack of propagation layers.

Propagation layer. Propagation layers are instantiated in the message-passing framework shown in Equation (1). In particular, the node update and message functions are implemented as

$$v_i^t = \texttt{MLP}_{\tau_v}^t\left(v_i^{t-1} \| \texttt{MEAN}\left\{\texttt{MLP}_{\psi_v}^t\left(v_j^{t-1} \| e_{j,i}^{\texttt{enc}}\right); (j,i) \in E_{\texttt{hls}}\right\}\right), \tag{5}$$

where $\|$ is the vector concatenation operator and $\texttt{MEAN}\{\cdot\}$ indicates that we aggregate incoming messages by averaging them out. To update the global representation u^t, we use an MLP as an update function that takes in input the concatenation of u at the previous propagation step and node attributes aggregated by exploiting the attention mechanism [41, 42]. In particular, we use a MLP ($\texttt{MLP}_\alpha(\cdot)$) to compute a raw attention score for each node attribute vector v_i^t given the global features u^{t-1}; raw scores (i.e., logits) are then normalized over the nodes of the graph with a `softmax` function. Normalized node scores are used to aggregate node features processed by a third MLP. Putting all together, each propagation layer updates the global representation as follows:

$$r_i^t = \texttt{MLP}_\alpha\left(u^{t-1}, v_i^t\right), \tag{6}$$

$$\alpha_i^t = \texttt{softmax}\left\{r_i^t | r_1^t, \ldots, r_n^t \in V_{\texttt{hls}}\right\} = \frac{e^{r_i^t}}{\sum_{j=1}^N e^{r_j^t}}, \tag{7}$$

$$u^t = \texttt{MLP}_{\tau_u}^t\left(u^{t-1} \| \texttt{SUM}\left\{\alpha_i^t \odot \texttt{MLP}_{\psi_u}^t\left(v_i^{t-1}\right); i \in V_{\texttt{hls}}\right\}\right), \tag{8}$$

where $\odot$ is the elementwise multiplication operator and $\texttt{SUM}\{\cdot\}$ indicates aggregation by graph-wise summation of node features. In practice, multiple attention heads can be used in parallel for increased model capacity.

Regression head. After T message-passing blocks, node representations are pooled in a single vector using a permutation invariant aggregation function (e.g., by taking the sum or the average of node representations, optionally weighted by learned attention scores). The pooled representation is then concatenated to the global attributes u^T leading to a vector representation of the input graph. This feature vector is fed through an MLP that maps it to a prediction of latency and resources as shown in Figure 3.

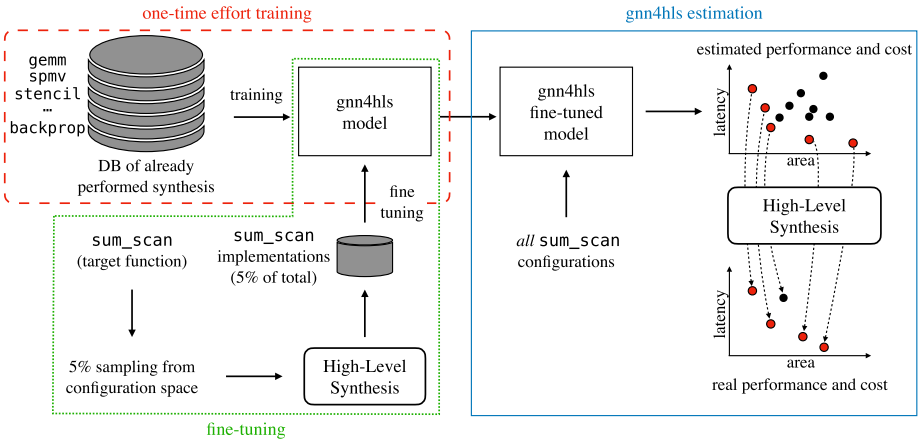

Fig. 4. gnn4hls. Overview of the DSE flow. gnn4hls is trained using the previously synthesized design (red-dashed rectangle). Then, the model is updated by synthesizing 5% of randomly sampled configurations from the target function (in the example radix sort) configuration space (green-dotted box). Lastly, the fine-tuned model is used to estimate the performance and cost of the remaining target function configurations and synthesize the estimated Pareto-optimal ones (blue box).

3.3 Training Procedure and Transfer Learning

We train the GNN by supervised learning to predict the outcome of the HLS procedure. We use data from several synthesized designs, with program specifications relevant to different domains (we refer to Section 4 for more details). While learning to predict the outcome of the synthesis process for configuration spaces already partially explored can be interesting on its own, we are particularly interested in assessing the possibility of exploiting the model for DSE. In particular, we aim at assessing if the learned representation can support transfer to different domains (designs) when only a few samples, or none, from the target configuration space are available. We comment that our method differs from previous approaches, which usually are domain-specific and tied to the characteristics of the target design space. Instead, our methodology is general and can easily incorporate knowledge from different design spaces by simply including synthesized points in the training dataset.

Design Space Exploration. Once a model of performance and costs is available, there are several exploration strategies that could be exploited (ranging from simple heuristics to more advanced methods, e.g., reinforcement learning). To provide a lower bound for the performance achievable with our model, we adopt a straightforward policy. In particular, we rely on gnn4hls to estimate the performance and cost of each implementation of the target function configuration space before deciding on the configurations to synthesize. We consider the setting where the designer performs an initial naïve random sampling of the configuration space, uses the synthesized points to fine-tune the model, and then uses the model's estimates over the configuration space to approximate the Pareto curve. Specifically, we select only the configurations expected to be Pareto-optimal by considering as cost the weighted sum of the utilized resources. The aggregated cost function C is defined as follows:

$$C = \frac{1}{4} \cdot \left(\frac{FF_{used}}{FF_{available}} + \frac{LUT_{used}}{LUT_{available}} + \frac{DSP_{used}}{DSP_{available}} + \frac{BRAM_{used}}{BRAM_{available}} \right). \tag{9}$$

Figure 4 shows the DSE scenario and its different elements. At first, the gnn4hls model is trained using the previously synthesized designs available to the designer (red box). Then, the model is

fine-tuned using a small number of syntheses from the target function (green box). Lastly, the remaining non-synthesized configurations' performance and cost are estimated and the estimated Pareto-optimal designs are synthesized (blue box).

4 EXPERIMENTAL EVALUATION

The graph representation of HLS designs and the associated `pragma` values are generated by combining LLVM [18] compiler passes, Clang AST analysis, Frama-C [10] internal representation of program dependencies, and HLS synthesis information from a recently published database of HLS-driven DSEs [13]. The custom LLVM pass generates the CFG representation from the compiler *Intermediate Representation (IR)* and performs static program analysis to identify the block proprieties. In order to account for the information lost in the LLVM IR representation, an AST visitor extracts and maps the SW information to the CFG blocks. Dataflow information is extracted from the Frama-C program dependency analysis [10], and used to generate the dataflow edges of $\mathcal{G}_{\texttt{hls}}$. We generated graph representations from 23 different functions in MachSuite [29]. For each design, we used configurations and synthesis results available from `db4hls` [13], an open source database of HLS DSEs. The total number of configurations considered in this work is 103, 093. Configuration spaces contain from several hundreds to several thousands of design points. Designs have been synthesized by using VivadoHLS [43] version 2018.2, targeting the ZynqMP Ultrascale+ (xczu9eg) FPGA chip, with a target clock of 10 ns. The list of considered functions, the associated design space, and the function lines of code are shown in Table 1.[2]

For the GNN, we use as global graph attributes the number of LLVM instructions, the number of input parameters, and the average value of each directive set within the configuration minus the mean value of the directive sets computed over the entire configuration space. To increase robustness w.r.t. outliers, we also concatenate to the representation the same values minus the median (as described in Section 3). In all experiments, we use GNNs with four propagation blocks; all MLPs have a single layer and `ELU` activation function with 128 and 256 units for node and global representations, respectively. In each propagation block, we use two global attention heads, implemented as in [42]. Models are trained with Adam for 800 epochs, with a learning rate of 0.001 and a batch size of 128. We use as regression targets the logarithm of the target values to account for the different scales. We also clip the gradient norm to a maximum value of 3 to avoid learning instabilities. We did not use any form of regularization since we did not observe any sign of overfitting. We do not perform any additional scaling to the input features w.r.t. the preprocessing steps described in Section 3.

Performance and cost estimation. We split the synthesized configuration space of each function in three folds, keeping 70% of the available points for training, 10% for validation, and 20% for testing. Selecting a proper baseline for comparison is not easy since none of the approaches existing in the literature can easily be extended to our settings. The most similar approach is the one introduced by Kwon and Carloni [17], where an MLP is trained in a multi-domain setting. However, their approach, based on multi-task learning [3], relies on training different input and output layers for each domain, hence limiting flexibility. Furthermore, they use only optimization directives as input and predict normalized scores for performance and costs instead of the actual latency and resources. For these reasons, we consider a DeepSets [50] model to be a more appropriate baseline. For the baseline, we use a node-level MLP with five hidden layers with 512 units each and `ReLU` activation function, followed by $\texttt{SUM}\{\cdot\}$ aggregation and a second global MLP with a single hidden layer with the same activation and number of neurons. In practice, we use a node-level MLP followed by a permutation invariant aggregation and a second MLP to process the aggregated

[2]Function names have been shortened to improve readability.

Table 1. List of Functions Considered in This Work from MachSuite [29]

Bench.	HLS design	SLOC	Pragma	| CS |	ADRS	S_{rel}^{max}
gemm	ncubed	41	▼■♦	2744	0.043	0.98x
	bbgemm	45	■♦	1600	0.016	0.96x
spmv	ellpack	28	■♦	1600	0.032	0.99x
radix sort	hist	34	▼■♦	4704	0.107	0.86x
	init	29	▲▼■♦	484	1.3	0.74x
	last_step_scan	32	▼■♦	800	0.211	0.98x
	sum_scan	31	▲▼■♦	1280	0.030	0.97x
	local_scan	32	▼■♦	704	0.167	0.60x
	update	36	▼■♦	2400	0.002	0.99x
	ss_sort	50	■♦★	1792	0.042	0.93x
Stencil	stencil2d	39	▼■♦	1344	0.193	0.83x
	stencil3d	62	▼■♦	1536	0.115	0.97x
md	knn	71	■♦	1152	0.006	0.99x
backprop	get_matrix_weights1	45	▼■♦	21952	0.087	0.97x
	get_matrix_weights2	45	▼■♦	31213	0.054	0.99x
	get_matrix_weights3	45	▼■♦	21952	0.087	0.92x
	bias_input_layer	48	■♦★	1372	0.005	0.99x
	bias_second_layer	48	■♦★	686	0.0001	1x
	bias_output_layer	48	■♦★	392	0.003	0.99x
	take_difference	44	▼■♦	512	0.002	0.99x
	oracle_activations1	48	■♦	2401	0.048	0.99x
	oracle_activations2	48	■♦	1372	0.013	0.95x
	update_weights	127	■♦	1024	0.0002	0.99x

The table shows benchmark (Bench.), target HLS design, source lines of code (SLOC), pragmas considered for the DSE, configuration space size ($|CS|$), average ADRS obtained at the end of the DSE, and average relative speedup with respect to the max achievable one (S_{rel}^{max}).
Pragma types: resource type ▲, partitioning type ▼, partitioning factor ■, loop unrolling ♦, function inlining ★.

features. The network architecture was chosen to have a similar number of parameters w.r.t. the GNN. For both models, we run the experiments using five different seeds.

The boxplot in Figure 5 shows, for each figure of merit, the median, first and third quartiles, and the interquartile range of the *Mean Absolute Percentage Error (MAPE)* of the two models over the 23 functions in the dataset. Results, averaged overfive independent runs, show that our model drastically outperforms the baseline by achieving an average MAPE, averaged over performance and cost estimates, of 2.7% against 15.4%—an improvement of over 82%. Furthermore, the performance obtained by our model is qualitatively similar to that of the SoA simulator, which exploits an analytical model of the HLS process, HLScope+ [7] for the latency, and MPSeeker [54] for resources.[3]

In particular, qualitatively, latency estimation performance is comparable to the best from the SoA (1.1% MAPE of HLScope+ vs. 2.1% of `gnn4hls`), and our area performance estimation outperforms that from existing models. `gnn4hls` achieves 4.8%, 2.6%, and 1.3% estimation error for FFs, LUTs, and DSPs compared to the 14.7%, 13.2%, and 12.7%, respectively, from MPSeeker.

[3]A direct comparison with these methodologies was not possible since these were not available as open source software. Thus, we qualitatively compare to the performance reported in the original papers.

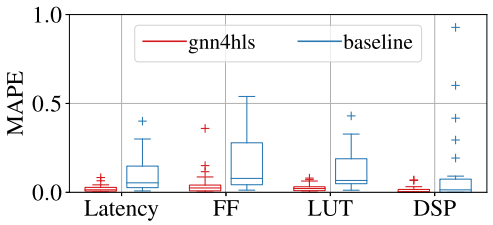

Fig. 5. Comparison among the gnn4hls approach proposed in this work and a MLP ones adopted as a baseline.

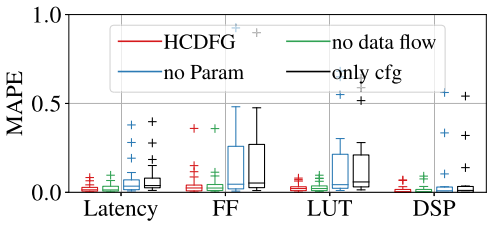

Fig. 6. Ablation study of gnn4hls graph structure.

In addition, inference time is greatly reduced from seconds to tens of milliseconds w.r.t. HLScope+ [7] and MPSeeker [54]. In fact, the network used here requires ≈10ms to process a single point from the `get_delta_matrix_weights2` function on an Intel(R) Xeon(R) Silver CPU @ 2.10 GHz, and only ≈3.5 s to provide performance and cost estimates for the *entire* design space on a Nvidia Titan V GPU (≈0.11ms per design).

Ablation study. To evaluate the effectiveness of the proposed graph representation, we have performed an ablation study on the graph edges. Figure 6 shows results obtained given three different ablations on the graph structure: no dataflow edges, no param edges, and both dataflow and param edges removed. For all the representations, the same type of nodes and attributes have been considered. Results show that the proposed Hybrid CDFG representation leads to the best results. The ablation study shows the large impact of the *param edges* on model performance. One of the reasons is the fact that *param edges* let additional important information flow through the graph nodes. Parameter nodes are in fact connected to the network *only* through *param edges*. Moreover, it is also possible to observe that part of the *data edges* can largely be inferred from *param edges*, thus the similar performance of the hybrid CDFG setting and the one missing only the DFG ones. For example, in Figure 2, the parameter block P in the bottom is connected with an outgoing edge to a write node W, and with an incoming one to a read node R with *param edges*. At the same time, the W and R nodes are connected by a *data edge*, and *control edges* connect the same two nodes through a third block. Thus, it is possible to observe the lack of *data edge* information can often be compensated by the presence of *param* and *control edges*.

Design-space exploration. The second set of experiments aims at addressing DSE. Herein, we focus on approximating the Pareto-frontier of a target HLS design, given the gnn4hls model trained by using the synthesis outcomes of all the considered functions but the target one (i.e., we perform a leave-one-out evaluation w.r.t. the available functions). This scenario aims at imitating a real case where a designer, given a new target function to implement in hardware, relies on an existing gnn4hls model—previously trained on a dataset of synthesis results from other functions—to perform the DSE. We further emphasize that training the model is a one-time effort performed by using the already synthesized implementations available to a designer—in our case all the considered target functions but the one object of our DSE. Given a base model, we use the exploration strategy specified in Section 3. In particular, we consider the case where the designer randomly synthesizes a small percentage of the design space—in our case 5% capped to 150 designs—before the fine-tuning stage. The time required by this phase depends entirely on the synthesis time of the selected configurations. Then, gnn4hls is fine-tuned using the newly generated syntheses. Because of the small number of sampled designs, the fine-tuning procedure requires less than 5 minutes on an Nvidia Titan V GPU in the considered scenarios. After the fine-tuning, the remaining configurations—all but the ones selected for the fine-tuning—are given in input to the model, and performance and cost are estimated. As already mentioned, gnn4hls performance

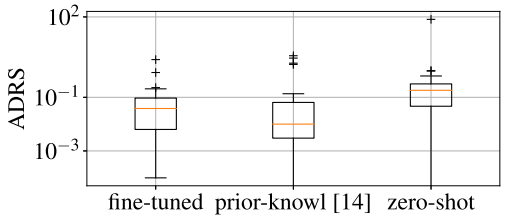

Fig. 7. Comparison between fine-tuned, prior-knowledge [14], and zero-shot approaches.

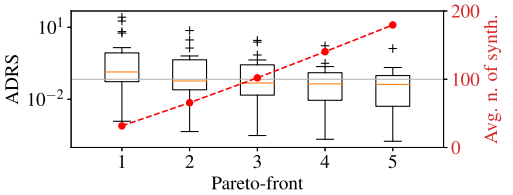

Fig. 8. Effect of the inference from multiple Pareto-frontier on the ADRS and the number of syntheses.

and cost estimation take ≈10 ms per configuration on a 2.10 GHz CPU, while exploiting GPUs and parallel computing drastically reduces these timings to ≈0.1ms per configuration. Lastly, the estimated Pareto-optimal implementations are synthesized and the actual performance and cost can be retrieved.

We assessed the quality of the DSEs measuring the *Average Distance from Reference Set (ADRS)* [14, 17, 34] metric among the real Pareto-solutions and the one estimated to be Pareto-optimal. The ADRS for two objective functions is defined as

$$ADRS(\bar{P}, P) = \left[\frac{1}{P}\sum_{p\in P}\min_{\bar{p}\in\bar{P}}(d(\bar{p}, p))\right], \tag{10}$$

$$d(\bar{p}, p) = \max\left\{0, \frac{A_{\bar{p}} - A_p}{A_p}, \frac{L_{\bar{p}} - L_p}{L_p}\right\}. \tag{11}$$

The ADRS score is an adimensional value used to quantify the distance between two curves, or, in our case, Pareto-frontiers. A low value of ADRS implies an accurate approximation of the real Pareto-frontier, and an ADRS value of 0 implies that the two Pareto-fronts are the same.

To increase robustness w.r.t. prediction errors, we iteratively select candidate Pareto-optimal points by removing from the configuration space the already selected configurations and recomputing the Pareto curve up to five times. Results are shown in Figure 7. In particular, we compare the performance of the fine-tuning approach (averaged over 40 independent runs) against the current SoA one on the considered dataset, namely, the *prior-knowl.* approach [14] (see Section 5). For the fine-tuning procedure, we use a maximum of 128 points from the target domain, capped at 5% of the configuration space dimension; we fix the number of SGD updates to 150, with a batch size of 32.

We considered only the portion of the configuration space that is actually synthesizable (i.e., we considered only configurations present in the database). Results show the distributions of the ADRS across the 23 considered functions. In particular, the *fine-tuned* model obtains Pareto-frontier approximations comparable to the SoA, while reducing the number of outliers compared to *prior knowl.* and obtaining an average ADRS of 0.20 versus 0.45—a 55% improvement. This result is particularly appealing when observing that our approach neither uses any heuristic to perform the initial sampling nor does it rely on domain knowledge provided by the designer. Furthermore, our method provides the user with performance and cost estimates of the candidate configurations, which could be instrumental in further reducing the number of syntheses required to obtain the desired performance and satisfy hardware constraints. Table 1 shows for each explored design the obtained ADRS and the relative speedup (S_{rel}^{max}) obtained w.r.t. to the best achievable one, over the 40 runs. The relative speedup is obtained by dividing the lowest-latency implementation available by the ground truth with respect to the lowest-latency implementation identified by the DSE. Results show that our methodology achieves an average S_{rel}^{max} of 0.94x, with 1x being the max value.

Lastly, Figure 8 shows how the ADRS scores change w.r.t. the number of iteratively estimated Pareto-frontiers during the DSE. This setting imitates the scenario of a designer using gnn4hls to synthesize the estimated Pareto solutions and not being satisfied by the first result obtained. The designer can then remove the synthesized configuration from the design space, and recalculate the Pareto-frontier to identify the Pareto-neighbor of the originally estimated ones. We can observe, how by exploring the neighborhood of estimated Pareto solutions we can improve the quality of the retrieved Pareto-frontier. Compared to *prior knowl.*, our approach requires a higher number of syntheses; however, our method, in addition to providing synthesis estimates, reduces the ADRS value exponentially (plot in logarithmic scale along the y axis), w.r.t. a linear growth in the number of required syntheses. The higher number of syntheses can be explained by the fact that *prior knowl.* attempts to map similar configurations across different design spaces, without having any knowledge. Thus, when the mapping is performed from a larger design space to a smaller one, it may happen that multiple configurations from a larger space will be mapped to the same configuration in the smaller target space. While, in the opposite scenario, from a smaller design space to a larger target one, not all of the Pareto-solution will be retrieved. In our case, gnn4hls can estimate all the possible configurations and each solution estimated to be Pareto-optimal can be synthesized. Thus, causing the number of syntheses to be higher w.r.t. *prior knowl.* We argue that we might expect this gap to reduce significantly when considering a larger dataset or a more advanced exploration heuristic.

4.1 Domain Transfer

Sensitivity to clock frequency. Herein, we analyze the ability of gnn4hls to generalize with respect to unseen designs generated by using different clock frequencies from the one used to train the model. The designs used to train the gnn4hls, from db4hls by Ferretti et al. [13], have been synthesized using a target clock period of 10 ns, equivalent to a 100 MHz frequency. We then repeat the synthesis process of the design points of local_scan from the radix sort benchmark, using three different clock periods of 5 ns, 20 ns, and 50 ns, equivalent to 200 MHz, 50 MHz, and 200 MHz. Then, we run gnn4hls in the DSE scenario to evaluate the exploration outcome in the different scenarios. We use the experimental setting previously described for DSE, i.e., the gnn4hls model is trained by using all the available data in the database (with clock period 100 MHz) except for the target function (i.e., local_scan), then fine-tuned on a few points from the target (with a potentially different clock period) before starting the exploration.

Figure 9 shows the result in terms of ADRS on logarithmic scale. The plot shows how gnn4hls is able to perform reasonably well when considering different clock periods. In particular, the average ADRS is 0.058, 0.162, 0.076, and 0.098 with 94, 89, 92, and 91 synthesized designs w.r.t. clock periods of 5 ns, 10 ns, 20 ns, and 50 ns, respectively.

Transfer to different application domains. Lastly, we evaluate the performance of gnn4hls when targeting designs from a completely different domain with respect to those used during training. In our experiments on performance and cost estimation and design-space exploration, we considered functions from benchmarks belonging to domains such as *Linear Algebra (LA)*—gemm, spmv; *Machine Learning (ML)*—backprop, md; *Sorting (SORT)*—radix; and *Numerical Data Processing (NDP)*—stencil. Usually, applications belonging to similar designs share some patterns, e.g., ML designs are often characterized by a large number of loops implementing matrix operations, and similarly, LA ones rely on computational-intensive loops.

To address transfer across different application domains, we additionally target designs from the aes benchmark from the *encryption (ENC)* domain, whose synthesized designs are available in the db4hls [13] database. Differently from ML or LA workloads, ENC applications are characterized by a larger number of bitwise operations, rather than computationally expensive loops (often

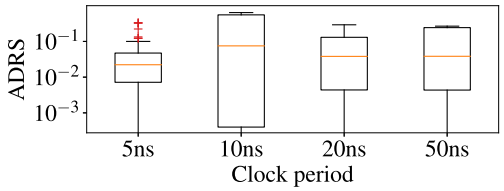

Fig. 9. Comparison among the DSE performance obtained by gnn4hls for the same design (*local_scan*) using different target frequencies.

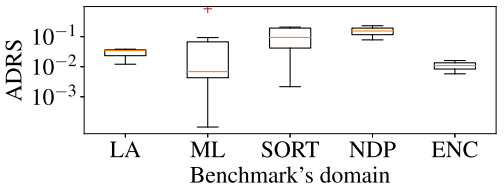

Fig. 10. Performance of gnn4hls on different application domains.

implemented in hardware with DSPs). Similarly to the choice made for the other benchmarks, we have selected designs having at least 100 different configurations and with data structures handled by the gnn4hls front-end.[4]

We have then performed a DSE for the aes_addRoundKey and aes_addRoundKey_cpy target functions repeating the same experimental setting used for the other DSE experiments. In the experiments, aes_addRoundKey and aes_addRoundKey_cpy obtain an average ADRS of 0.016 and 0.006 requiring 37 and 29 synthesis, respectively, with an aggregated average ADRS value of 0.011. Figure 10 shows the ADRS values (on a logarithmic scale) obtained after DSE for the various application domains considered in our experiment. The figure shows how gnn4hls can effectively guide DSE, independently from the target application domain.

5 RELATED WORKS

A few recent surveys from Schafer and Wang [34], Reyes Fernandez de Bulnes et al. [30], and Shathanaa and Ramasubramanian [36] have summarized the effort from the hardware design community to address the HLS-driven DSE problem. Among the works presented in the surveys, we can identify two main categories: *model-based* approaches and *refinement-based* ones.

Model-based methodologies [7, 51, 53] aim at modeling the behavior of HLS tools and estimate the effect of HLS directives given *a priori* knowledge of the HLS tool employed and of the application structure. For these methodologies, the model is usually defined as an analytical model able to estimate the tool performances. The generation of the analytical model requires a characterization of HLS directives and of the specific HLS tool. However, estimating the effect of HLS directive combinations is an extremely challenging task requiring reverse-engineering of the tool for the selected pragmas. Thus, *model-based* methodologies are often constrained to only a few directives. Nonetheless, such models are tailored for specific HLS tools and do not have robustness guarantees w.r.t. new versions of the same tool. Examples of *model-based* approaches are the works from Cong et al. [8, 9], Choi and Cong [6], and Chi et al. [4]. Choi and Cong [6] proposed a tool estimating HLS resources and performance of HLS designs subject to array partitioning, loop unrolling, and pipelining directives. In particular, Choi and Cong [6] build an analytical model to perform resource estimation. Similarly, analytical models are employed by Chi et al. [4], and Cong et al. [8, 9], to estimate performance and resource consumption ahead of the actual synthesis. In these works, the exploration space and the model complexity are limited to a specific class of applications (stencils in the case of SODA [4]), architectures (systolic arrays in PolySA [8]), or to a reduced configuration space obtained by transforming the code into fixed design templates with source-to-source transformations [9]. In a similar fashion, Liu et al. [20] and Tan et al. [39] adopt

[4]The aes benchmark uses struct construct and globally defined arrays to share the parameters across functions. These implementation choices are not widely diffused in HLS benchmarks and are the only cases in the whole MachSuite [29] benchmark suite. Our front-end does not currently handle C struct and globally defined arrays.

architectural templates to limit the design space and reduce the complexity of the problem before building an analytical model for performance prediction. More recently, the approaches introduced by Zhao et al. [52], Wang et al. [44], and Zhong et al. [55], have exploited static graph analysis. These methodologies rely on compiler passes to generate a graph representation of the design starting from the original code. The generated graph representations (mainly DFGs) are then used to analyze the included processing elements and their interaction. Among these, COMBA, from Zhao et al. [52], focuses on the operation chaining aspect of the designs. The DFG and operation chaining information are used to build analytical models of iteration latency and loop latency, which are then used to model the effect of unrolling and pipeline directives. Thus, the array partitioning factor is estimated in order to maximize the effect of the chosen unrolling or pipelining factors. Shao et al. [35] propose Aladdin, an RTL simulator for fixed-function accelerators. Aladdin estimates power, performance, and area of a design starting from their software implementation and a set of HLS-like directives. Similarly, Choi et al. [5] have released a rapid HLS simulation flow estimating performance and costs. However, despite the good estimation capabilities and the reduced evaluation costs (from tens of minutes to minutes to perform the estimations) of both approaches, the time required to assess the design quality remains a significant limitation while addressing vast design spaces. All the *model-based* methodologies described share some commonalities: they all are effective in addressing the DSE problem but they achieve that by limiting the design space, either constraining its size(handling a limited number of directives) or targeting a specific class of applications. The main drawback of such approaches is the lack of transferability and the need for designer and researcher expertise to revise the model in order to include additional directives or to update the model to the last version of the HLS tool.

On the other hand, *black-box-based* approaches rely on the outcome of a few heuristically sampled synthesis runs as a starting point for DSE. These methodologies are usually agnostic to the HLS tool and to the HLS directives considered. While these approaches do not rely on a specific HLS tool and set of directives, they need to build their knowledge during the DSE process, while HLS synthesis is performed. This phase can be performed offline (i.e., before the exploration), online (i.e., during the exploration), or it can be a combination of both. While model-based methods do not have to go through this phase and are usually more efficient, black-box approaches do not require an analytical model and in-depth characterization of the HLS tools. *Black-box-based* methodologies include *learning-based* methodologies, which rely on an initial learning phase to produce an approximate model of the HLS process needed to guide DSE, and *refinement-based* approaches which build and refine their knowledge during the DSE.

Works belonging to the *learning-based* category often exploit a vector-based representation of the input encoding the design characteristics. The work from Ozisikyilmaz et al. [25] was the first to use neural networks to build a predictive model to perform DSE for computer architectures. In the context of HLS-driven DSE exploration, the scarcity of large datasets to train deep models, and the difficulties to encode programs in a vector representation have limited the adoption of neural-network-based solutions. Zuluaga et al. [57] proposed an area and latency regression model relying on a Gaussian Process to predict cost and performance given an initial (small) training set. Alternatively, Schafer et al. exploited pattern-matching techniques to perform DSE [33]. From a different perspective, Meng et al. [24] proposed an adaptive threshold non-Pareto elimination technique focusing on the risk analysis of losing high-quality designs due to inaccurate models. Lastly, Zacharopoulos et al. [49] combined a compiler pass analysis and a random forest classifier to predict the optimal unrolling factor of loops.

Widely adopted approaches for *refinement-based* methodologies are instead meta-heuristic and ad-hoc heuristics. These approaches have been shown to be effective while dealing with multi-objective optimization problems (i.e., estimating the whole Pareto-frontier instead of

only low-latency or low-area implementations). Several works have explored population-based heuristics [1, 11, 16, 27] to investigate the use of Genetic Algorithms to address DSE problems and navigate the design space across the directive combinations. These approaches, while able to effectively generate good candidate solutions, require many iterations, and, therefore, a high number of syntheses before reaching a good Pareto approximation. Response Surface Model approaches have also been widely adopted [23, 26, 38, 47] to refine the simulation-based exploration and generate a model of the synthesis engine. Simulated annealing approaches have been adopted by Mahapatra and Schafer [22], and Schafer et al. [32] to iteratively generate designs able to efficiently navigate the design space. Schafer [31] propose a probabilistic model to predict Pareto-dominant solutions. Divide and conquer approaches exploring different loops separately and merging the exploration results together have been explored in [33]. Random forests have been used by Liu and Carloni [21] to infer a model of the HLS tool and refine the model at each new synthesis. Piccolboni et al. [28] have instead proposed an exploration heuristic that concurrently coordinates both HLS directives and memory optimization exploration. Lastly, Ferretti et al. [12] have proposed local search techniques based on PCA to navigate the DSE-based correlations among directives and their Pareto-dominance property.

Taking an orthogonal direction, Ferretti et al. [14], and Wang et al. [45] have proposed DSE strategies leveraging prior knowledge to address the DSE of unseen designs. The proposed approaches search for similarity in the source code of already explored designs, and, based on the result of past DSEs, decides how to optimize the target one. We compared to *prior-knowl.* [14] due to the possibility to use `db4hls` [13] for direct comparison and the high-perfomance results achieved by the methodology across a large variety of designs. Similarly, a recent work [17] proposes a neural network model for mixed-sharing multi-domain transfer learning to transfer the knowledge obtained from previously explored design spaces in exploring a new target one; however, it relies on a vector-based representation that limits the flexibility and effectiveness of the method.

In recent years, graph deep learning has surged in popularity and several architectural improvements and GNN variants have found widespread adoption by the community. Among the many applications, GNNs have been widely used in software engineering to automate program analysis and code optimization [19, 37, 56]. Recently, GNNs have also been exploited to predict the performance of the physical implementations of HLS designs addressing delay estimation [40] and constrained resource optimization [46]. However, none of these works aim at assessing the behavior of HLS pragmas without constraining the exploration. In particular, the IronMan [46] model predicts performance and resources (LUT and DSP) of implementations given directives targeting DSP usage (enable/not-enable, reuse factor). Furthermore, IronMan relies on a DFG representation of the input and uses different learnable models for each regression target.

Differently from the above-mentioned works, `gnn4hls`, exploits a novel graph representation adopting a coarser grain granularity of the design behavioral description. In our representation, pragmas are associated with each graph node instead of being independent nodes of the graph. Moreover, `gnn4hls` handles pragmas targeting different software constructs (array, loops, functions) and estimates multiple resources without constraints, using a unique model for all the regression targets. Lastly, our DSE focuses on multiple objectives, i.e., searches for the entire Pareto-frontier of solutions instead of focusing on a single performance metric (latency) as done in IronMan [46].

6 CONCLUSION AND FUTURE DIRECTIONS

In this work, we presented `gnn4hls`, a graph-based learning framework for HLS-driven DSE. Compared to the state of the art, our method offers tools that are general and that can easily be applied

to any configuration space. A key aspect of `gnn4hls` is its simplicity w.r.t. its effectiveness. We show that our method compares favorably against state-of-the-art solutions w.r.t. both quantitative and qualitative metrics. Furthermore, `gnn4hls` explores the entire Pareto-frontier of possible solutions and provides an accurate estimation of all the FPGA resources. In the future, we plan to investigate more advanced solutions from the few-shot learning literature to improve transfer to unseen domains and to consider finer-grained representations of basic blocks. By investigating transfer learning approaches, we can address the problem of updating the model across different versions of the HLS tools to keep the model updated with the most recent releases. Similarly, we can investigate the integration of unseen pragmas and adapt the learned model across different HLS tools. Then, we argue for the possibility of replacing existing heuristics to perform DSE with an exploration policy learned with a (model-based) reinforcement learning approach. To support breakthroughs in this direction, we renew our invitation to the community in participating in a common effort for the collection of datasets fully enabling the application of deep learning in the context of HLS-driven DSE. We believe that this work represents a milestone for the application of graph deep learning toward a fully automated design of HW accelerators.

REFERENCES

[1] I. Ahmad, M. K. Dhodhi, and F. H. Hielscher. 1994. Design-space exploration for high-level synthesis. In *Proceedings of the 13th IEEE Annual International Phoenix Conference on Computers and Communications*.

[2] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261.

[3] R. Caruana. 1997. Multitask learning. *Machine Learning* 28, 1 (1997), 41–75.

[4] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. IEEE, 1–8.

[5] Young-kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. 2020. FLASH: Fast, parallel, and accurate simulator for HLS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4828–4841.

[6] Young-kyu Choi and Jason Cong. 2018. HLS-based optimization and design space exploration for applications with variable loop bounds. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. IEEE, 1–8.

[7] Y.-K. Choi, P. Zhang, P. Li, and J. Cong. 2017. HLScope+: Fast and accurate performance estimation for FPGA HLS. In *International Conference on Computer-Aided Design*. IEEE, 691–698.

[8] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. IEEE, 1–8.

[9] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC'18)*. IEEE, 1–6.

[10] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. 2012. Frama-C: A software analysis perspective *(SEFM'12)*. 9, 1 (2012), 35–43.

[11] L. Ferretti, G. Ansaloni, and L. Pozzi. 2018. Cluster-based heuristic for high level synthesis design space exploration. *IEEE Transactions on Emerging Topics in Computing* 99 (2018), 1–9.

[12] L. Ferretti, G. Ansaloni, and L. Pozzi. 2018. Lattice-traversing design space exploration for high level synthesis. In *International Conference on Computer Design*. 210–217.

[13] L. Ferretti, J. Kwon, G. Ansaloni, G. Di Guglielmo, L. Carloni, and L. Pozzi. 2021. DB4HLS: A database of high-level synthesis design space explorations. *IEEE Embedded Systems Letters* 13, 4 (2021), 194–197.

[14] L. Ferretti, J. Kwon, G. Ansaloni, G. Di Guglielmo, L. P. Carloni, and L. Pozzi. 2020. Leveraging prior knowledge for effective design-space exploration in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3736–3747.

[15] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. 2017. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*. PMLR, 1263–1272.

[16] Martin Holzer, Bastian Knerr, and Markus Rupp. 2007. Design space exploration with evolutionary multi-objective optimisation. In *International Symposium on Industrial Embedded Systems*. IEEE, 126–133.

[17] J. Kwon and L. P. Carloni. 2020. Transfer learning for design-space exploration with high-level synthesis. In *ACM/IEEE Workshop on Machine Learning for CAD (MLCAD'20)*.

[18] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization*. 75.

[19] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *International Conference on Machine Learning*. PMLR, 3835–3845.

[20] Gai Liu, Mingxing Tan, Steve Dai, Ritchie Zhao, and Zhiru Zhang. 2017. Architecture and synthesis for area-efficient pipelining of irregular loop nests. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 11 (2017), 1817–1830.

[21] Hung-Yi Liu and Luca P. Carloni. 2013. On learning-based methods for design-space exploration with high-level synthesis. In *Proceedings of the 50th Design Automation Conference*. 1–6.

[22] A. Mahapatra and B. C. Schafer. 2014. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *Electronic System Level Synthesis Conference*. 1–6.

[23] Giovanni Mariani, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. 2012. OSCAR: An optimization methodology exploiting spatial correlation in multicore design spaces. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 5 (May 2012), 740–753.

[24] Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner. 2016. Adaptive threshold non-Pareto elimination: Re-thinking machine learning for system level design space exploration on FPGAs. In *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*. IEEE, 918–923.

[25] Berkin Ozisikyilmaz, Gokhan Memik, and Alok Choudhary. 2008. Efficient system design space exploration using machine learning techniques. In *Proceedings of the 45th Design Automation Conference*. ACM, 966–969.

[26] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 2009. ReSPIR: A response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 12 (Nov. 2009), 1816–1829.

[27] Maurizio Palesi and Tony Givargis. 2002. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign*. 67–72.

[28] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2017. Cosmos: Coordination of high-level synthesis and memory optimization for hardware accelerators. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–22.

[29] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization*. 110–119.

[30] Darian Reyes Fernandez de Bulnes, Yazmin Maldonado, and Leonardo Trujillo. 2020. Development of multiobjective high-level synthesis for FPGAs. *Scientific Programming* 2020 (2020).

[31] Benjamin Carrion Schafer. 2015. Probabilistic multiknob high-level synthesis design space exploration acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 3 (2015), 394–406.

[32] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. 2009. Adaptive simulated annealer for high level synthesis design space exploration. In *International Symposium on VLSI Design, Automation and Test*. IEEE, 106–109.

[33] Benjamin Carrion Schafer and Kazutoshi Wakabayashi. 2012. Divide and conquer high-level synthesis design space exploration. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17, 3 (June 2012), 29.

[34] B. C. Schafer and Z. Wang. 2020. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2628–2639.

[35] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. IEEE, 97–108.

[36] R. Shathanaa and N. Ramasubramanian. 2018. Design space exploration for architectural synthesis—a survey. In *Recent Findings in Intelligent Computing Techniques*, Pankaj Kumar Sa, Sambit Bakshi, Ioannis K. Hatzilygeroudis, and Manmath Narayan Sahoo (Eds.). 519–527.

[37] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song. 2018. Learning loop invariants for program verification. In *Neural Information Processing Systems*.

[38] C. Silvano, W. Fornaciari, G. Palermo, V. Zaccaria, F. Castro, M. Martinez, S. Bocchio, R. Zafalon, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, M. Wouters, C. Kavka, L. Onesti, A. Turco, U. Bondik, G. Mariani, H. Posadas, E. Villar, C. Wu, F. Dongrui, Z. Hao, and T. Shibin. 2010. MULTICUBE: Multi-objective design space exploration of multi-core architectures. In *IEEE Computer Society Annual Symposium on VLSI*. 488–493.

[39] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. 2015. Elasticflow: A complexity-effective approach for pipelining irregular loop nests. In *Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'15)*. IEEE, 78–85.

[40] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang. 2020. Accurate operation delay prediction for FPGA HLS using graph neural networks. In *International Conference on Computer-Aided Design*.

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention is all you need. In *Neural Information Processing Systems*. 6000–6010.
[42] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. 2018. Graph attention networks. *In International Conference on Learning Representations*.
[43] VivadoHLS. 2018. Vivado High-Level Synthesis. Retrieved December 1, 2022 from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
[44] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In *Proceedings of the 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. 1–6.
[45] Z. Wang, J. Chen, and B. C. Schafer. 2020. Efficient and robust high-level synthesis design space exploration through offline micro-kernels pre-characterization. In *2020 Design, Automation Test in Europe Conference Exhibition*. 145–150.
[46] N. Wu, Y. Xie, and C. Hao. 2021. *IRONMAN: GNN-Assisted Design Space ExploRAtiOn in High-Level SyNThesis via ReinforceMEnt LeARNIng*. 39–44.
[47] Sotirios Xydis, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. 2015. SPIRIT: Spectral-aware Pareto iterative refinement optimization for supervised high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 1 (Oct. 2015), 155–159.
[48] J. You, Z. Ying, and J. Leskovec. 2020. Design space for graph neural networks. *Neural Information Processing Systems* 33 (2020).
[49] Georgios Zacharopoulos, Andrea Barbon, Giovanni Ansaloni, and Laura Pozzi. 2018. Machine learning approach for loop unrolling factor prediction in high level synthesis. In *Proceedings of the 2018 International Conference on High Performance Computing & Simulation (HPCS'18)*. IEEE, 91–97.
[50] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola. 2017. Deep sets. In *Neural Information Processing Systems*, Vol. 30.
[51] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proceedings of the International Conference on Computer Aided Design*. 430–437.
[52] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. IEEE, 430–437.
[53] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. 2016. Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *Proceedings of the 53rd Design Automation Conference*. 136:1–136:6.
[54] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar. 2017. Design Space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation & Test in Europe Conference & Exhibition, 2017*. IEEE.
[55] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar. 2014. Design space exploration of multiple loops on FPGAs using high level synthesis. In *Proceedings of the International Conference on Computer Design*. 456–463.
[56] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, P. Phothilimtha, S. Wang, A. Goldie, A. Mirhoseini, and J. Laudon. 2020. Transferable graph optimizers for ML compilers. In *Neural Information Processing Systems*.
[57] Marcela Zuluaga, Andreas Krause, Peter Milder, and Markus Püschel. 2012. Smart design space sampling to predict Pareto-optimal solutions. In *ACM SIGPLAN Notices*. 119–128.

Received 16 February 2022; revised 22 October 2022; accepted 31 October 2022