Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks

Ecenur Ustun*, Chenhui Deng*, Debjit Pal, Zhijing Li, Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{eu49,cd574,debjit.pal,zl679,zhiruz}@cornell.edu

ABSTRACT

Modern heterogeneous FPGA architectures incorporate a variety of hardened blocks for boosting the performance of arithmetic-intensive designs, such as DSP blocks and carry blocks. Since hardened blocks can be configured in different ways, a variety of datapath patterns can be mapped into these blocks. We observe that existing high-level synthesis (HLS) tools often fail to capture some of the operation mapping patterns, leading to limited estimation accuracy in terms of resource usage and delay. To address this deficiency, we propose to exploit graph neural networks (GNN) to automatically learn operation mapping patterns. We apply GNN models that are trained on microbenchmarks directly to realistic designs through inductive learning. Experimental results show that our approach can effectively infer various valid mapping patterns on both microbenchmarks and realistic designs. Furthermore, the proposed framework is exploited to improve the accuracy of delay estimation in HLS.

CCS CONCEPTS

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

KEYWORDS

FPGAs, Graph Neural Networks, High-Level Synthesis

ACM Reference Format:

Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, Zhiru Zhang. 2020. Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415657

1 INTRODUCTION

Operation delay estimation is a crucial task in high-level synthesis (HLS) [4], as it directly impacts where the clock cycle boundaries are introduced during scheduling and pipelining [5–7, 24]. Currently, most academic and commercial HLS tools rely on simple delay models where each operation is pre-characterized in isolation and the overall delay is calculated additively over individual operations [2, 4, 35]. However, during technology mapping, operations are clustered into device resources, invalidating the additive approach. It is a challenging task for HLS tools to accurately predict operation mappings as this requires knowledge of the target device architecture (including the architecture of the hardened blocks), application requirements (e.g., target speed), and better heuristics to identify how the given application requirements are best met with the given design structure on the given device architecture. Introducing LUT-based mapping-awareness in HLS has been shown to improve area and timing by producing better scheduling solutions [28, 39]. However, these prior efforts mainly consider optimizing the mapping on soft logic and can only improve logic-intensive designs, despite the fact that modern FPGA architectures include a variety of hardened blocks in addition to the array of programmable LUTs.

Digital signal processing (DSP) blocks are designed for high-speed arithmetic operations such as multiplication and accumulation which are fundamental in various applications, such as machine learning (ML), image processing, and high performance computing. DSP blocks can be configured in various ways depending on the application requirements and design constraints, each configuration corresponding to a unique datapath pattern. Modern FPGAs also embed dedicated carry blocks consisting of dedicated carry-lookahead logic, muxes, and fast routing that are independent of the LUT-based soft logic. They further improve efficiency of arithmetic operations such as adders and counters. Existing commercial HLS tools typically follow hard-coded rules to infer mapping of operations onto DSP or carry blocks through subgraph matching. These rules need to be re-generated by HLS tool developers when new FPGA architectures are available, which in practice is a labor-intensive process. Moreover, these manual rules often fail to predict some of the more complex mapping behaviors of logic synthesis.

There has been an increasing use of ML to improve quality of results estimations in HLS [8, 21, 23, 38]. Among these, QuickEst [8] leverages ML inference to narrow the gap between resource estimates in the HLS report and actual post-implementation results. While QuickEst helps to reduce error in area estimation, it cannot be easily extended to delay estimation because the underlying ML model does not use any structural features from the input design.

In order to increase productivity and portability in HLS development, we argue that it is essential to automate the inference of mapping patterns. To this end, we propose to leverage the emerging graph learning techniques, specifically graph neural networks (GNNs), to automatically learn mapping and clustering of arithmetic operations in HLS. Since mapping and clustering patterns are local structures around arithmetic operations and GNN captures

*Equal contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '20, November 2–5, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415657

local neighborhood information in graph-structured data, we find GNN to be an effective way to learn these patterns. We then show improvements in the accuracy of delay characterization in HLS with the introduction of mapping-awareness, which accounts for the optimizations in logic synthesis. Our major technical contributions are as follows:

- To the best of our knowledge, this is the first work to predict technology mapping patterns of operations in HLS using graph learning. Particularly, we propose GNN-based inference on the clustering of arithmetic operations in HLS dataflow graphs.
- We propose D-SAGE, an enhanced GNN model targeting at learning high-quality node representations on directed graphs. We apply D-SAGE to learning operation clustering on dataflow graphs and achieve 17.3% accuracy improvement compared to a widely-used baseline GNN model called GraphSAGE [12].
- We develop parameterized microbenchmark generation of arithmetic operations in order to generate data with various mapping patterns for building generalizable GNN models. We then extract the mapping between HLS intermediate representation (IR) and post-mapping netlist at the arithmetic operation level to formulate our GNN model in HLS and learn optimization decisions in logic synthesis.
- We perform mapping-aware delay characterization in HLS based on the patterns learned by our GNN-based technique. Our approach leads to a 72% reduction in root-mean-square error compared to operation delay estimation in Vivado HLS.

The rest of this paper is organized as follows: Section 2 provides background on delay characterization in HLS, mapping of arithmetic operations onto FPGA resources, and GNNs. Section 3 presents our overall approach to learn operation mapping patterns using GNNs. Section 4 reports experimental results. Section 5 surveys related work, followed by conclusions in Section 6.

2 PRELIMINARIES

Delay Characterization in HLS – Academic and commercial HLS tools provide delay estimations to guide implementation decisions and design space exploration, and meet certain post-implementation criteria such as timing closure. These tools typically build simple additive delay models based on pre-characterization of individual operations [2, 4, 35]. For a given device family, operations of various types and bitwidths in HLS IR are pre-characterized to extract operation delays. With the pre-characterized library of target device families, HLS tools can provide delay estimations of designs by looking up operation delays from the library and calculating overall delays by summing up delays of operations along the paths in the dataflow graphs. Such an approach leads to inaccurate logic delay estimations due to the lack of awareness of downstream optimizations in logic synthesis and technology mapping.

DSP Mapping – Modern heterogeneous FPGA devices include specialized DSP blocks to support high-performance implementations of arithmetic operations, such as FIR filters and FFT. DSP blocks are highly configurable to realize a wide range of functionalities and system requirements. They have pre-fabricated datapaths which can be configured by various control bits. For example, datapath of the DSP48E2 primitive in Xilinx Ultrascale devices [34] includes a 27-bit pre-adder unit, a 27×18-bit multiplier, a 48-bit ALU, registers, wide logic operations, and multiplexers controlled by configuration bits to be able to implement various circuit subgraphs. Depending on the optimizations in logic synthesis and technology mapping stages, subgraphs in a design can map to DSP blocks in various ways. HLS tool proposed in [27] targets an older DSP block primitive, i.e., DSP48E1, which supports smaller number of configurations compared to the DSP48E2 primitive. More importantly, they enumerate all subgraph templates in a DSP block which means mapping rules need to be regenerated for newer DSP architectures. Due to the increasing complexity of specialized blocks on newer FPGA device families, it becomes more challenging to enumerate all possible templates. We propose to automatically learn DSP mapping patterns by formulating GNNs on arithmetic-intensive dataflow graphs (DFGs).

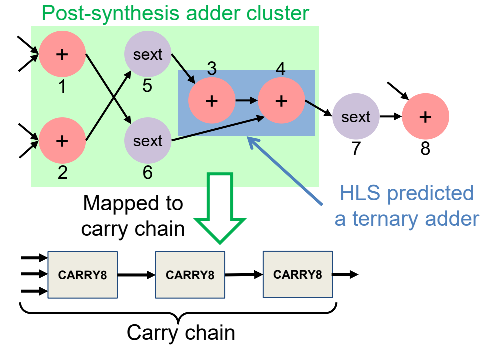


Figure 1: Adder-cluster for an input DFG – + represents an add operation; sext represents a sign extension operation. HLS clusters add operations 3 and 4 only (blue rectangular box), whereas technology mapping clusters add operations 1-4 along with sign extension operations 5-6 (green rectangular box).

Adder Cluster Mapping – In addition to the specialized DSP blocks, modern heterogeneous FPGA devices also include dedicated carry logic to speed up arithmetic operations (e.g., CARRY8 blocks in Xilinx Virtex UltraScale+ devices [33]). RTL synthesis can take advantage of the carry blocks by mapping two or more adjacent addition/subtraction operations on a single carry chain along with other logic operations (e.g., sign extension, bit truncation, zero extension). We define such patterns of two or more adjacent addition/subtraction operations along with other logic operations as an adder cluster. We have shown one such adder cluster in Figure 1, where all the add and sign-extension operations within the green block are mapped to a single carry chain. On the other hand, HLS usually can only predict the ternary adder patterns where an addition/subtraction operation directly feeds a following addition/subtraction operation; this is illustrated in the the inner blue block of Figure 1. In this work, we also propose a learning-assisted methodology to automatically identify adder cluster that is mapped to a carry chain, thereby improving delay estimation.

Graph Neural Networks – Recent years have seen a surge of interest in deep learning on graphs, also known as graph neural network, which aims to encode nodes into low dimensional vectors that maximally preserve graph structural information. Specifically, given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ represent node and

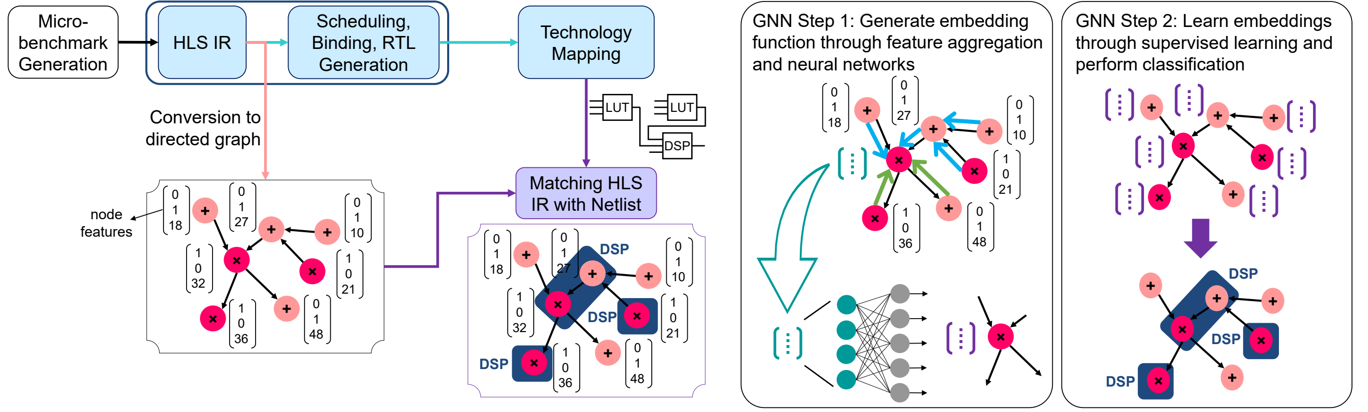


Figure 2: Overall flow – GNN-based learning of operation mapping and clustering in arithmetic-intensive designs

edge set, respectively, we have the adjacency matrix $A \in \mathbb{R}^{n \times n}$ that captures the topology information of the graph with n nodes. Let $X \in \mathbb{R}^{n \times k}$ be the node attribute matrix whose i-th row represents the attribute information of the i-th node in a k-dimension vector. A graph neural network (GNN) model is essentially a trainable mapping function f such that $H = f(A, X)$, where $H \in \mathbb{R}^{n \times d}$ is called embedding matrix and $H_{i,:}$ represents the d-dimension embedding vector which preserves the (local) structural information of node i.

A powerful GNN model should encode nodes with similar local structures into similar embedding vectors in the embedding space (e.g., Euclidean space). GNN techniques have shown promising results for various applications such as node classification, link prediction, and community detection [1, 11, 40]. Although there exist various GNN models, they can broadly be divided into the following two categories:

- **Transductive model**: To obtain the embedding vector of each node, a transductive GNN model requires to see the whole graph structure during training, which means that the model needs to be retrained once graph structure has changed (e.g., adding a new node to the graph).
- **Inductive model**: Instead of knowing the whole graph structure for producing the embedding vector per node, the inductive GNN model learns the aggregation function that aggregates node attribute vectors from neighbors to generate node embeddings. The trainable aggregation function is shared across the graph, which enables the trained model to be applied on a different graph without retraining. In this work, we use inductive GNN model to learn operation clustering across different DFGs.

3 APPROACH

We propose to automatically learn mapping and clustering patterns of arithmetic operations, i.e., how various arithmetic subgraphs map to configurable datapaths of hardened blocks on FPGAs, using graph learning techniques. We formulate GNNs on arithmetic-intensive DFGs to learn mapping and clustering patterns based on dataflow structures and operation features. To be able to carry out our learning-assisted methodology, we automatically extract the mapping between HLS operations and netlist objects, which is a nontrivial task due to the difference in the level of abstractions and difficult-to-predict optimization decisions in RTL synthesis.

Our overall flow is illustrated in Figure 2. From a set of microbenchmarks, our tool extracts dataflow graphs from the HLS IR and matches DFG operations with netlist objects after the technology mapping stage. Based on these correlations, actual operation mapping and clustering patterns are extracted automatically. Dataflow graph structure, node features, and the actual mapping patterns (i.e., ground truth) are used to train our GNN model. Once the model is trained, we can infer mapping patterns from given dataflow graphs using the previously trained GNN model. Our technique can successfully learn operation mappings and their clustering into device resources, and consequently avoid the manual and tedious process for matching HLS subgraphs to FPGA resources.

3.1 Microbenchmark Generation

To learn mapping patterns from various local structures and generalize to unseen designs, we generate microbenchmarks composed of arithmetic operations. These microbenchmarks are used to train our GNN model in an inductive manner such that the trained model can directly be used to predict mapping patterns of unseen designs. Therefore, microbenchmarks should contain a wide spectrum of subgraph structures that can be mapped to device resources after technology mapping. We first generate directed, weakly connected, simple graphs of addition and multiplication operations with different connectivities as shown in Figure 3a. Total number of operations in a graph is set to n, whereas the number of multiplication operations and the number of inputs are varied to generate various data-sharing patterns, contributing to the variance in training set. Generated graphs are then converted to C programs to be input to the HLS flow as shown in Figure 3b.

3.2 Feature and Ground Truth Extraction

The input to our flow is HLS IR, which in our case is the dataflow graph. Operations in the HLS IR are represented as nodes and data dependencies are represented as edges in dataflow graphs. Learning-assisted approaches build models based on the features and ground truth of the underlying data. Unlike regular Euclidean data types such as images, graphs are highly irregular which makes manual

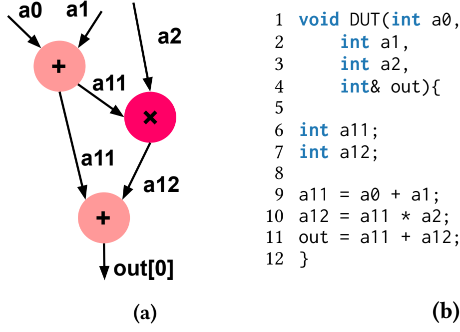


Figure 3: Microbench generation procedure – (a) Weakly connected directed graph comprising a total of $n = 3$ operations of which two operations are addition and one operation is multiplication. (b) C program generated from the graph of (a).

feature engineering inefficient. Therefore, there has been an increasing interest in learning representations in graph-structured data, where the main goal is learning low-dimensional representations of graph objects [32].

In the context of arithmetic operation mapping and clustering, embeddings need to capture both structural and contextual information. Structural information stands for nodes in a graph and their connectivity patterns, illustrated as the directed dataflow graph extracted from HLS IR in Figure 2. This sort of information is necessary to identify the neighborhood of an arithmetic operation and consequently their mapping patterns. In our formulation, structural information is encoded as adjacency matrices, from which node connectivities can be inferred. Contextual information, i.e., node features, are supplementary information we have about the operations, which are operation type and bitwidth. Node features, illustrated as the vectors on each node in Figure 2, are essential for two reasons. First, mapping patterns are local structures around arithmetic operations, bringing about the need to distinguish different operation types. Secondly, hardened blocks (e.g., DSPs) support functionalities of certain bitwidths, making operation bitwidths an important factor in identification of operation clustering.

To guide embedding generation in GNNs in a supervised manner such that embeddings preserve actual operation mapping and clustering information, we extract ground truth (i.e., node and edge labels) of the underlying data. Extraction of ground truth requires matching operations in HLS IR to device resources in post-mapping netlist. To the best of our knowledge, this is the first work to realize automatic matching of HLS IR to post-mapping netlist at the arithmetic operations level. Automation involves the analysis of configuration bits and input signal bits of each hardened block along with connectivities of operations in dataflow graphs and netlist objects in post-mapping netlists.

3.3 Graph Neural Network Model

As indicated in [27], DSP mapping patterns are essentially local structure around multiply operation. As a consequence, whether an operation is mapped into a DSP block is fully determined by its local structure and attribute information of its neighbors (e.g., operation type, bitwidth). Similarly whether an add operation is

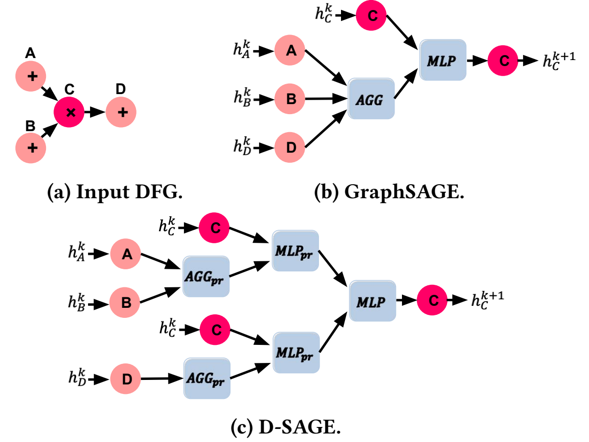


Figure 4: Comparison of GraphSAGE and D-SAGE at the k-th layer – (a) Input dataflow graph. (b) Compute embedding vector of node C (i.e., multiply operation) with GraphSAGE. (c) Compute embedding vector of node C with D-SAGE.

clustered with other nearby adjacent add and logic operations (if any) is determined by its local structure. Since GNN can produce node embeddings that incorporate both local structure and node attributes, we exploit GNN to learn operation mapping and clustering. Specifically, given the data flow graph, for DSP mapping we label nodes mapped into a DSP block as 1, and 0 otherwise, and for adder cluster mapping we label nodes mapped into a carry chain as 1, and 0 otherwise. Then we can apply the GNN model to this binary node classification task. Similarly, we can also perform binary edge classification to predict whether an edge is mapped into a DSP or a carry block. The embedding vector of an edge can be obtained by averaging the embeddings of source and target node.

Inductive learning. We extend the popular GNN model called GraphSAGE [12] to learn the node embeddings in an inductive manner. More specifically, GraphSAGE learns an aggregation function (AGG) that collects the information from neighbors and produces node embeddings at the k-th layer as follows:

$$h_i^{(k+1)} = MLP^{(k)}(h_i^{(k)} \,||\, AGG^{(k)}(\{h_j^{(k)}, j \in \mathcal{N}(i)\})) \qquad (1)$$

where $\mathcal{N}(i)$ is the neighbor set of node i, $\cdot||\cdot$ is featurewise concatenation, $AGG^{(k)}$ and $MLP^{(k)}$ represent the aggregation function and multilayer perceptron at the k-th layer, respectively. Note that $h_i^{(0)} = x_i$, where x_i is the initial attribute vector of node i.

As shown in Eq. (1), GraphSAGE learns the embedding vectors in two steps: the aggregation function AGG first collects embedding vectors of neighbors to get the aggregated vector, which is then fed into MLP to update the embedding vector from previous layer. Since the trainable weights inside MLP module do not depend on the whole structure (i.e., adjacency matrix), the trained GraphSAGE model can be directly applied to generate embedding vectors of new graphs without retraining.

Enhanced GNN model for directed graph. Although GraphSAGE inductively learn node embeddings, it fails to support directed graphs. Specifically, the aggregation function of GraphSAGE treats all neighbors uniformly when gathering their information,

which makes it unable to distinguish predecessors and successors. Figure 4a shows a simple dataflow graph, where nodes A, B, and D denote addition operations and node C represents multiplication operation. Since nodes A and B are the predecessors of node C (i.e., pre-adders), while node D is the successor of C (i.e., post-adder), as shown in Figure 4b, the AGG of GraphSAGE fails to capture such information, which limits its power to learn high-quality node embeddings on directed graphs. To tackle this issue, we derive a variant of GraphSAGE model called D-SAGE in the following.

$$h_{i,pr}^{(k+1)} = MLP_{pr}^{(k)}(h_i^{(k)} \,||\, AGG_{pr}^{(k)}(\{h_j^{(k)}, j \in \mathcal{PR}(i)\})) \tag{2}$$

$$h_{i,su}^{(k+1)} = MLP_{su}^{(k)}(h_i^{(k)} \,||\, AGG_{su}^{(k)}(\{h_j^{(k)}, j \in \mathcal{SU}(i)\})) \tag{3}$$

$$h_i^{(k+1)} = MLP^{(k)}(h_{i,pr}^{(k+1)} \,||\, h_{i,su}^{(k+1)}) \tag{4}$$

where $\mathcal{PR}(i)$ and $\mathcal{SU}(i)$ denote the predecessors and successors of node i, respectively. Intuitively, D-SAGE works on directed graphs via separately collecting the information from predecessors and successors of each node, and then combines the information from both sides to update node embeddings at each layer. It is worth noting that the concatenate operation in Eq. (2) and (3) can be viewed as "skip connection" between layers, which leads to significant gains in performance [13, 14]. As indicated by Eq. (5), we implement AGG_{pr}, AGG_{su}, and AGG of D-SAGE via the "MEAN" aggregator, which is the most efficient yet effective one compared to other options, such as LSTM and Max-Pooling aggregators [12].

$$AGG^{(k)}(\{h_j^{(k)}, j \in \mathcal{N}(i)\}) = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} h_j^k \tag{5}$$

As shown in Figure 4c, our D-SAGE model exploits AGG_{pr} to aggregate the information of predecessors per node and then feeds it into MLP_{pr} to generate the aggregated embedding vector of the predecessor set. D-SAGE then produces the aggregated embedding vector of the successor set in a similar way. Lastly, D-SAGE combines both aggregated vectors and feeds them into another MLP module to update the node embedding, as indicated in Eq. (4). In this way, D-SAGE can effectively incorporate edge direction information, which enables D-SAGE capture both pre-adder (nodes A and B) and post-adder (node D) when learning DSP mapping. While D-SAGE adopts the idea of learning aggregation function from GraphSAGE, the way that D-SAGE works on directed graphs and the addition of "skip connection" make a substantial difference for learning operation clustering. Our experimental results in Section 4.2 show that D-SAGE converges faster and considerably improves the accuracy of operation clustering compared to the vanilla GraphSAGE.

3.4 Learning Operation Mapping

The first step to learning mapping schemes is identifying the type of device resource each arithmetic operation in a dataflow graph maps onto, i.e., learning operation mapping. For example, an addition operation can map to a LUT network or DSP blocks. Here, the problem is binary node classification where nodes correspond to operations in DFGs and these nodes are labeled depending on the type of resource that they map onto after technology mapping, i.e., 1 if DSP, 0 if LUT. For adder-cluster mapping, we label all add/sub operations in a cluster along with other logic operations (e.g., sext, zext etc., if any) as 1 if they are mapped to a carry-chain; any single add/sub operation that is mapped to a carry-chain is labeled as 0. In Figure 1, add operations 1-4 and sign extension operations 5-6 are mapped into a single carry chain, hence all those nodes are labeled 1. On the other hand, add operation 8 is not clustered with any other add/sub operations and hence is labeled 0.

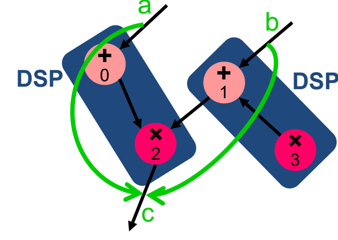


Figure 5: An example to illustrate the impact of operation clustering on timing.

Characterization of the device resource types in a DFG is not sufficient to learn delay of a given design, which is illustrated in Figure 5. Both node 0 and node 1 are labeled as 1, and whether node 2 is clustered with node 0 or node 1 into a DSP block has a direct impact on delay calculation, as the path from a to c passes through a single DSP block whereas the path from b to c passes through two DSP blocks, or vice versa. To calculate delay accurately, we should also know which operation is clustered with which other operation(s) into a DSP block. To this end, we perform the second stage of our learning formulation, i.e., learning operation clustering. Here we have a binary edge classification problem, where each edge with label 1 represents clustering of its two nodes in the same device resource. In this setting, edges are abstract in the sense that they represent clustering schemes.

4 EVALUATION

We evaluate our operation mapping, clustering, and delay characterization flows with microbenchmarks we generated as described in Section 3.1 and real designs from MachSuite [25]. We collect data and run experiments using Xilinx Vivado and Vivado HLS 2019.1 targeting a Virtex UltraScale+ device. The corresponding DSP architecture is the DSP48E2 slice, which consists of a 27-bit pre-adder unit, a 27×18-bit multiplier, and a 48-bit ALU [34].

For the classification tasks, metrics of interest are (1) precision, which is the percentage of the correctly classified samples among all samples with the predicted label 1, (2) recall, which is the percentage of the correctly classified samples among all samples with the actual label 1, and (3) F1, which represents a balance between precision and recall, and is our main metric for evaluating classification accuracy. Precision, recall, and F1 scores take values from 0 to 1, while values closer to 1 mean higher accuracy.

For the regression tasks, metrics of interest are (1) root-mean-square error (RMSE), which is a standard measure of the spread of the actual values around those predicted by the model, and (2) coefficient of determination (R2), which is another standard measure of how well a model approximates actual data. R2 metric can take any value smaller than or equal to 1, while 1 corresponds

to the best possible accuracy. RMSE carries the same unit as the ground truth and smaller values correspond to higher accuracy.

In regard to hyperparameters of machine learning models in our experiments, we choose three layers MLP with a hidden layer dimension of 256, and 400 epochs. Moreover, we use two layers in both D-SAGE and GraphSAGE, with the hidden dimension of 60. We use a learning rate of 0.001, a weight decay of 0.0005, and rectified linear unit (ReLU) as activation function for all models. For node classification task, we use an early stopping strategy on the observed training loss, with a patience of 20 epochs. For edge classification task, we train all models for 100 epochs with the batch size of 512. Since our tasks are binary classification, we use binary cross entropy (BCELoss) with positive label weight as 6 to undermine the dataset imbalance issue. We further leverage stochastic gradient descent for optimization.

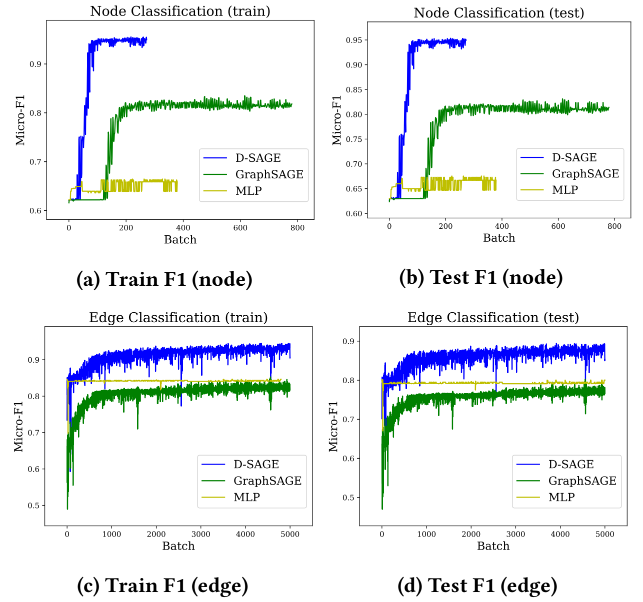
4.1 Data Collection

We set the total number of operations for each microbenchmark to 20 while the number of multiplication operations is varied between 4 to 8. We further vary the number of inputs of each design from 8 to 12 in order to generate various data-sharing patterns across DFGs. Each microbenchmark is synthesized as combinational logic on Virtex UltraScale+ device xcvu11p. In our dataset generated from 2000 microbenchmarks, HLS-estimated LUT usage ranges from 23 to 449, DSP usage ranges from 3 to 12, and estimated delay ranges from 7.2 ns to 24.3 ns; whereas post-routing LUT usage ranges from 15 to 356, DSP usage ranges from 3 to 11, and delay ranges from 5.2 ns to 17.3 ns. To evaluate generalizability of our learning model to real designs, we perform additional testing on FIR filter and five MachSuite [25] benchmarks with DSP usage (i.e., FFT, GEMM, MD, SPMV, and STENCIL). Non-optimized versions of these real benchmarks (e.g., non-pipelined) are synthesized on the same device as the microbenchmarks, targeting 250 MHz clock frequency.

4.2 Comparison of Learning Models

To show our model is indeed useful for learning operation mapping and clustering, we compare D-SAGE with vanilla GraphSAGE as well as multilayer perceptron (MLP) in Figure 6. We train all models on mul4,5,7,8 and test on mul6. In regard to node classification task, as shown in Figure 6a and 6b, GNN-based models (D-SAGE and GraphSAGE) significantly improve the F1 score compared to MLP, which indicates that learning graph structural information plays a crucial role in operation mapping. Moreover, D-SAGE further improves GraphSAGE by a relative gain of $(0.95 - 0.81)/0.81 \times 100\% = 17.3\%$, which means that learning the edge direction information does make a difference. It is worth noting that D-SAGE also converges much faster than both baselines.

As for edge classification, as shown in Figure 6c and 6d, D-SAGE still achieves the best performance and improves GraphSAGE by a margin of $(0.89 - 0.76)/0.76 \times 100\% = 17.1\%$. Our insight is that D-SAGE can effectively learn the roles of pre-adder and post-adder respectively, while GraphSAGE fails to capture such information and learns all adders in the same way.



(a) Train F1 (node) **(b) Test F1 (node)**

(c) Train F1 (edge) **(d) Test F1 (edge)**

Figure 6: Comparison of various learning-assisted classification tasks – (a) Training F1 score for node classification. (b) Test F1 score for node classification. (c) Training F1 score for edge classification. (d) Test F1 score for edge classification.

4.3 Operation Mapping Prediction

We label each node in DFG with respect to the type of device resource they map onto after technology mapping. Operations are labeled as 1 if they are implemented with DSP blocks, or labeled as 0 if they are implemented with LUT networks. We first measure the accuracy of the HLS tool for this node classification task, and then show our GNN-based classification results. Our model is trained on a set of designs and tested on designs unseen during training. To account for the potential impact of network structure on the learning process, we reserve our test set to designs with a certain number of multiplication operations and hide these designs from training. For example, when the test set is mul4, our model is trained on designs with 5, 6, 7, and 8 multiplication operations and tested on those with 4 multiplication operations. In Table 1, we report the classification metrics of HLS and our GNN-based formulation for different training-testing splits. Our model is able reduce false positive cases by 75% on average, reflected in the increase in precision and F1 scores. Furthermore, we test our model that is trained on microbenchmarks directly on real designs from MachSuite [25]. Testing results on real designs are shown in Table 2. Compared to HLS, our model D-SAGE achieves 46% improvement in the operation mapping task.

We label each edge in DFG with respect to the clustering of operations into device resources after technology mapping. Edges are labeled as 1 if its nodes are clustered into the same DSP block after technology mapping, and 0 otherwise. At this second stage of the learning formulation, node labels predicted at the first stage are used as extra node attributes, because whether operations are mapped into DSP or not effect the clustering schemes. For example, the edge between a node with label 1 and another node with label 0

Table 1: Node classification accuracy on microbenchmarks.

Method	Training Set	Test Set	Precision	Recall	F1
HLS	-	mul4	0.69	0.99	0.82
	-	mul5	0.72	0.99	0.84
	-	mul6	0.74	1.00	0.85
	-	mul7	0.77	1.00	0.87
	-	mul8	0.77	1.00	0.87
D-SAGE	mul5,6,7,8	mul4	0.88	0.95	0.91
	mul4,6,7,8	mul5	0.90	0.97	0.93
	mul4,5,7,8	mul6	0.92	0.98	0.95
	mul4,5,6,8	mul7	0.93	0.95	0.94
	mul4,5,6,7	mul8	0.87	0.99	0.92

Table 2: Node classification accuracy on real designs.

Method	Training Set	Test Set	Precision	Recall	F1
HLS	-	MachSuite	0.33	0.60	0.43
D-SAGE	mul4,5,6,7,8	MachSuite	0.45	1.00	0.63

Table 3: Edge classification accuracy on microbenchmarks.

Method	Training Set	Test Set	Precision	Recall	F1
HLS	-	mul4	0.61	0.87	0.71
	-	mul5	0.62	0.85	0.72
	-	mul6	0.60	0.81	0.69
	-	mul7	0.59	0.77	0.67
	-	mul8	0.57	0.74	0.64
D-SAGE	mul5,6,7,8	mul4	0.83	0.95	0.89
	mul4,6,7,8	mul5	0.82	0.97	0.89
	mul4,5,7,8	mul6	0.80	0.97	0.88
	mul4,5,6,8	mul7	0.78	0.95	0.86
	mul4,5,6,7	mul8	0.78	0.96	0.86

Table 4: Edge classification accuracy on real designs.

Method	Training Set	Test Set	Precision	Recall	F1
HLS	-	MachSuite	0.25	0.60	0.35
D-SAGE	mul4,5,6,7,8	MachSuite	0.42	1.00	0.59

has to be labeled 0 since only one of the nodes is mapped to a DSP block. We first measure the accuracy of the HLS tool for this edge classification task. We then show our GNN-based classification results, which are later used to improve the accuracy of delay prediction in HLS. We follow the same training-testing split strategy as in the node classification task to ensure our model is generalizable to unseen structures. In Table 3, we report the classification metrics of HLS and our GNN-based formulation. Compared to the previous node classification task, our improvement over the HLS tool in the edge classification task is more significant. Our GNN-based formulation reduces false positive cases by 62% and false negative cases by 67% on average, reflected in the significant increase in the F1 score. HLS tool fails to capture many clustering schemes of operations into DSP blocks, which has a direct impact on timing. Furthermore, we test our model that is trained on microbenchmarks directly on real designs from MachSuite [25]. Testing results on real designs are shown in Table 4. Compared to HLS, our model D-SAGE achieves significant improvement in the operation clustering task.

As shown in Table 5, we further evaluate our approach on our microbenchmarks for carry chain clustering task. Specifically, we

Table 5: Edge classification accuracy of carry chain.

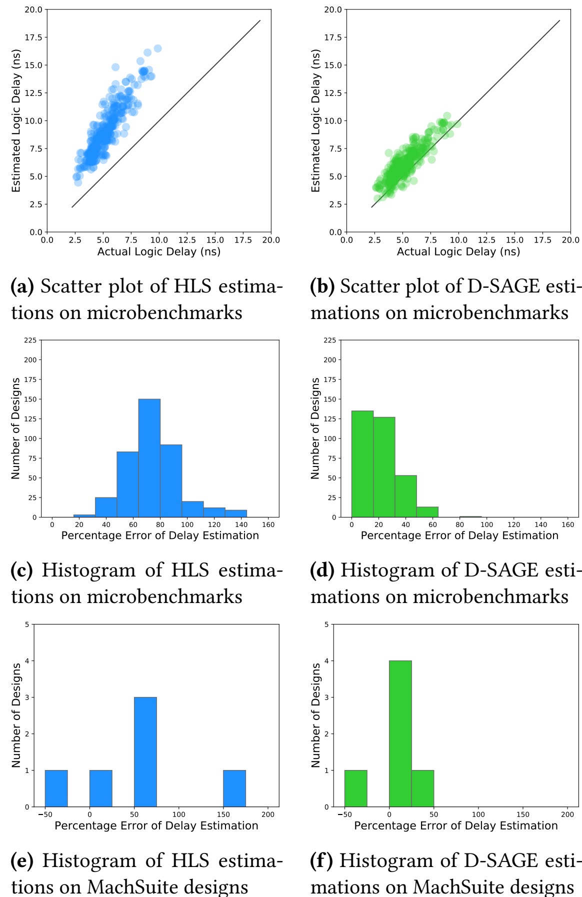
Method	Training Set	Test Set	Precision	Recall	F1
HLS	-	mul6	0.13	0.24	0.17
MLP	mul4,5,7,8	mul6	0.61	0.92	0.73
GraphSAGE	mul4,5,7,8	mul6	0.63	0.94	0.75
D-SAGE	mul4,5,7,8	mul6	0.73	0.94	0.82

formulate the carry chain clustering as an edge classification task, where edge is labeled 1 if it is mapped into carry chain, and 0 otherwise. Similar to the settings of DSP mapping, we train on designs with 4, 5, 7, and 8 multiply operations, and test on designs with 6 multiply operations. Our results indicate that D-SAGE achieves the best F1 score compared to other baselines, confirming its efficacy on directed graphs.

4.4 Delay Prediction

We introduce mapping-awareness in HLS delay prediction by calculating logic delays based on the results from learning operation mapping. Improvement in the accuracy of delay prediction is shown in Figure 7. We first show scatter plots of all microbenchmarks with respect to logic delays, where each marker corresponds to a microbenchmark, x-axis represents actual logic delays from post-mapping netlists, and y-axis represents logic delays estimated in HLS. As shown in Figure 7a, HLS deviates significantly from the actual delay values, especially for longer datapaths where downstream optimizations may become harder to predict. Our framework, on the other hand, captures the timing behavior across all datapath ranges as shown in Figure 7b. To better understand the source of improvement, we also show histogram of percentage error in logic delay prediction across all microbenchmarks. In Figure 7c, where we show delay results from the HLS tool, majority of the designs have more than 60% error in delay estimation and the maximum error is 143%. In Figure 7d, where we show D-SAGE-based results, almost all designs have less than 60% error in delay estimation and the maximum error is 88%. We further incorporate operation mapping results of D-SAGE in improving delay estimations of MachSuite designs. Percentage error of delay estimation in HLS for the realistic designs are shown in Figure 7e, whereas the results from D-SAGE are shown in Figure 7f. D-SAGE reduces maximum observed error from 164% down to 40%. Our GNN-based model is considerably more accurate compared to the HLS tool due to the introduction of mapping-awareness in delay calculations.

To further show learning structural information is critical to HLS delay estimation, we compare our approach against both HLS report and QuickEst [8], which is the state-of-the-art method of estimating resource usage for HLS. Specifically, we append the corrected estimated delay from D-SAGE to the original HLS 87 features (e.g., estimated number of DSPs), and then feed the new features to QuickEst. We evaluate all the methods on the MachSuite designs. Since QuickEst uses root relative squared error (RRSE) as the performance metric, we choose the same metric for a fair comparison. Lower RRSE score denotes better delay estimation results. Table 6 shows that applying D-SAGE to incorporate structure information significantly reduces the delay estimation error of both HLS report and QuickEst, which confirms that learning structural information via D-SAGE does make a difference for HLS delay estimation.



(a) Scatter plot of HLS estimations on microbenchmarks **(b)** Scatter plot of D-SAGE estimations on microbenchmarks

(c) Histogram of HLS estimations on microbenchmarks **(d)** Histogram of D-SAGE estimations on microbenchmarks

(e) Histogram of HLS estimations on MachSuite designs **(f)** Histogram of D-SAGE estimations on MachSuite designs

Figure 7: Operation delay prediction in HLS.

Table 6: Delay estimation error (RRSE) on realistic designs.

Method	Logic delay	Logic+Interconnect delay
HLS	2.19	2.22
QuickEst [8]	1.92	2.44
D-SAGE+HLS	0.82	0.83
D-SAGE+QuickEst	0.89	1.20

5 RELATED WORK

With increasing complexity of FPGA architectures, performance optimization has become more challenging than ever. In recent years, ML techniques are increasingly used to address that challenge by automatically exploring optimal FPGA tool configurations [15, 29, 36]. In spite of such extensive research efforts, the discrepancy between QoR estimations in HLS and post-implementation results hinders fast design closure.

To efficiently and effectively narrow the gap between HLS estimation and actual QoR, several prior efforts attempt to leverage ML approaches. For example, Zhao et al. use gradient boosting model to improve estimation accuracy of routing congestion in HLS [38]. In [8, 20, 21], various ML models are employed to achieve fast and accurate area estimation in HLS. Recently, Lin et al. propose an ML-based power estimation model for HLS [18]. However, none of the existing approaches consider operation clustering in their formulations.

Considering clustering of logic operations into LUTs in logic-intensive designs to improve scheduling solutions has been addressed in HLS [28, 39]. In [26, 27], Ronak et al. propose a novel HLS flow to fully exploit DSP block capabilities considering other design structures such as arithmetic-intensive designs and mapping those arithmetic operations onto DSP blocks. Several prior work have leveraged the performance boost that DSP delivers to FPGAs for accelerating various algorithms (e.g., color space conversion [3], filters [22], and cryptography [9]). However, to the best of our knowledge, there has not been an effort to automatically identify operation mapping patterns by formulating learning-assisted methodologies on graph-structured data.

Gori et al. first outline the notion of GNNs that aims to extract key graph structural information [10]. Inspired by convolutional neural networks (CNNs) that achieve impressive results on image-based tasks, Bruna et al. introduce convolutional operation on graphs, followed by Kipf and Welling, who simplify the graph convolution operation and propose graph convolutional network (GCN)[16]. Later, Hamilton et al. propose GraphSAGE that aims to learn nodes embeddings inductively [12]. Moreover, Velickovic et al. derive graph attention network (GAT) by exploiting attention mechanism to further improve the expressiveness of GNNs, with the cost of computing the attention score per edge [30]. Nonetheless, the above GNN models either fail to work on directed graphs or are unable to support inductive learning. Our model D-SAGE tackles both issues and show promising results when learning operation clustering.

In recent years, GNN models have been applied to a plethora of EDA problems that are traditionally known to be extremely difficult. Kirby et al. introduced a novel GNN-based solution for congestion estimation without requiring placement information by learning frequent cell connectivity patterns that cause congestion [17]. Circuit-GNN, a GNN-based model for designing distributed circuits, was proposed in [37] where the model learns to simulate electromagnetic properties of distributed circuits. Ma et al. have proposed a GNN-based classifier to insert observation points in design netlist to enhance design testability [19]. Most recently, Wang et al. have proposed GCN-RL circuit designer which combines GNN and reinforcement learning to transfer the knowledge of transistor sizing from one circuit to another to reduce re-design overhead [31].

6 CONCLUSIONS

We present a new approach based on graph learning to automatically learn operation mapping patterns in HLS. Local structure around arithmetic operations impact the mapping between operations and FPGA resources, and consequently timing. We leverage automatically learned operation mapping behaviors to perform mapping-aware timing characterization of arithmetic-intensive designs, which are more accurate compared to the conventional timing models in the academic and commercial HLS tools.

ACKNOWLEDGMENTS

This work was supported in part by NSF Awards #1512937, #1723715, and Intel Strategic Research Alliance (ISRA) Program. We would like to thank Bain Syrowik and Dr. Aravind Dasu from Intel Corporation, and the anonymous reviewers for their invaluable feedback.

REFERENCES

[1] H. Cai, V. W. Zheng, and K. C. Chang. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Transactions in Embedded Computing Systems (TECS)*, 2013.

[3] Z. Chun, Z. Yongjun, C. Xin, and G. Xiaoguang. Research on Technology of Color Space Conversion Based on DSP48E. *Int'l Conf. on Measuring Technology and Mechatronics Automation (ICMTMA)*, 2011.

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[5] J. Cong and Z. Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. *Design Automation Conf. (DAC)*, 2006.

[6] S. Dai, G. Liu, and Z. Zhang. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[7] S. Dai and Z. Zhang. Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning. *Design Automation Conf. (DAC)*, 2019.

[8] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Y. Young, and Z. Zhang. Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[9] A. de la Piedra, A. Braeken, and A. Touhafi. Leveraging the DSP48E1 Block in Lightweight Cryptographic Implementations. *Int'l Conf. on e-Health Networking, Applications and Services (Healthcom)*, 2013.

[10] M. Gori, G. Monfardini, and F. Scarselli. A New Model for Learning in Graph Domains. *IEEE Int'l Joint Conference on Neural Networks*, 2005.

[11] P. Goyal and E. Ferrara. Graph Embedding Techniques, Applications, and Performance: A Survey. *Knowledge-Based Systems (KBS)*, 2018.

[12] W. Hamilton, Z. Ying, and J. Leskovec. Inductive Representation Learning on Large Graphs. *Advances in Neural Information Processing Systems*, 2017.

[13] W. L. Hamilton, R. Ying, and J. Leskovec. Representation Learning on Graphs: Methods and Applications. *arXiv preprint arXiv:1709.05584*, 2017.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Identity Mappings in Deep Residual Networks. *European Conf. on Computer Vision (ECCV)*, 2016.

[15] N. Kapre, H. Ng, K. Teo, and J. Naude. InTime: A Machine Learning Approach for Efficient Selection of FPGA CAD Tool Parameters. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2015.

[16] T. N. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907*, 2016.

[17] R. Kirby, S. Godil, R. Roy, and B. Catanzaro. CongestionNet: Routing Congestion Prediction Using Deep Graph Neural Networks. *Int'l Conf. on Very Large Scale Integration (VLSI-SoC)*, 2019.

[18] Z. Lin, J. Zhao, S. Sinha, and W. Zhang. HL-Pow: A Learning-Based Power Modeling Framework for High-Level Synthesis. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2020.

[19] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu. High Performance Graph Convolutional Networks with Applications in Testability Analysis. *Design Automation Conf. (DAC)*, 2019.

[20] M. Makni, M. Baklouti, S. Niar, and M. Abid. Hardware Resource Estimation for Heterogeneous FPGA-based SoCs. *Symp. on Applied Computing (SAC)*, 2017.

[21] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. Pudukotai Dinakarrao, H. Homayoun, and S. Rafatirad. Pyramid: Machine Learning Framework to Estimate the Optimal Timing and Resource Usage of a High-Level Synthesis Design. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2019.

[22] R. Mehra and S. Devi. FPGA Implementation of High Speed Pulse Shaping Filter for SDR Applications. *Recent Trends in Networks and Communications*, 2010.

[23] V. Mrazek, M. A. Hanif, Z. Vasicek, L. Sekanina, and M. Shafique. autoAx: An Automatic Design Space Exploration and Circuit Building Methodology utilizing Libraries of Approximate Components. *Design Automation Conf. (DAC)*, 2019.

[24] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2015.

[25] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. *Int'l Symp. on Workload Characterization (IISWC)*, 2014.

[26] B. Ronak and S. A. Fahmy. Efficient Mapping of Mathematical Expressions into DSP Blocks. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2014.

[27] B. Ronak and S. A. Fahmy. Mapping for Maximum Performance on FPGA DSP Blocks. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[28] M. Tan, S. Dai, U. Gupta, and Z. Zhang. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2015.

[29] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang. LAMDA: Learning-Assisted Multi-stage Autotuning for FPGA Design Closure. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph Attention Networks. *arXiv preprint arXiv:1710.10903*, 2017.

[31] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H. S. Lee, and S. Han. GCN-RL Circuit Designer: Transferable Transistor Sizing with Graph Neural Networks and Reinforcement Learning. *arXiv preprint arXiv:2005.00406*, 2020.

[32] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[33] Xilinx Inc. UltraScale Architecture Configurable Logic Block. 2017.

[34] Xilinx Inc. UltraScale Architecture DSP Slice User Guide. 2019.

[35] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. 2020.

[36] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang. A Parallel Bandit-Based Approach for Autotuning FPGA Compilation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[37] G. Zhang, H. He, and D. Katabi. Circuit-GNN: Graph Neural Networks for Distributed Circuit Design. *Int'l Conf. on Machine Learning (ICML)*, 2019.

[38] J. Zhao, T. Liang, S. Sinha, and W. Zhang. Machine Learning Based Routing Congestion Prediction in FPGA High-Level Synthesis. *Design, Automation, and Test in Europe (DATE)*, 2019.

[39] R. Zhao, M. Tan, S. Dai, and Z. Zhang. Area-Efficient Pipelining for FPGA-Targeted High-Level Synthesis. *Design Automation Conf. (DAC)*, 2015.

[40] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph Neural Networks: A Review of Methods and Applications. *arXiv preprint arXiv:1812.08434*, 2018.