DSPlacer: DSP Placement for FPGA-based CNN Accelerator

Baohui Xie[1†], Xinrui Zhu[1†], Zhiyuan Lu[1], Yuan Pu[2], Tongkai Wu[1], Xiaofeng Zou[3], Bei Yu[2], Tinghuan Chen[1*]
[1]Chinese University of Hong Kong, Shenzhen [2]Chinese University of Hong Kong
[3]Shandong Yunhai Guochuang Cloud Computing Equipment Industry Innovation Co., Ltd.

***Abstract*—Deploying convolutional neural networks (CNNs) on hardware platforms like Field Programmable Gate Arrays (FPGAs) has garnered significant attention due to their inherent flexibility and parallelism. Achieving optimal timing closure remains a critical challenge, as placement directly impacts clock frequency and throughput. Existing approaches often face scalability issues with large designs or fail to formalize placement rules into automated algorithms. In this paper, we propose DSPlacer, a novel DSP placement framework designed for diverse CNN accelerator architectures in the context of FPGA design. The proposed approach iteratively optimizes the placement of datapath DSPs to enhance timing performance. To achieve this, DSPlacer integrates several advanced techniques, including graph convolutional network-based datapath DSP identification, DSP graph construction, min-cost-flow DSP assignment, and integer linear programming (ILP)-based cascade constraint legalization. These techniques collectively address two key requirements for datapath DSP placement: (1) cascading datapath DSPs to achieve a compact layout, and (2) preserving direct datapath information between the processing system and programmable logic. The framework has been evaluated on multiple academic benchmarks and compared against AMD Xilinx Vivado 2020.2 and AMF-Placer 2.0. Experimental results demonstrate that DSPlacer improves Worst Negative Slack (WNS) by 32% and 65%, respectively, highlighting its efficacy and superiority.**

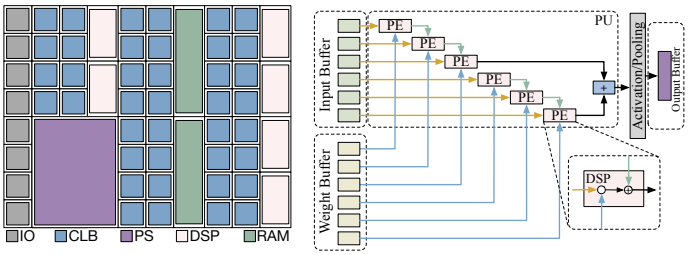
I. Introduction

Convolutional Neural Networks (CNNs) have succeeded in various computer vision applications. Field Programmable Gate Array (FPGA) is a promising hardware platform to deploy the CNN model due to its reconfigurability [1]–[3]. Besides, in modern FPGA, processing systems (PSs) such as CPU and programmable logics (PLs) are integrated into a device to further facilitate CNN deployment. In order to deploy the CNN model on FPGA, logic synthesis, placement, and routing need to be performed by taking a design described by a hardware description language (HDL) as an input. Logic synthesis converts an HDL of design into an optimized gate-level netlist. After logic synthesis, the netlist consists of heterogeneous components, such as lookup tables (LUTs), flip-flops (FFs), digital signal processors (DSPs), random access memories (RAMs), and I/O pads. Placement maps all heterogeneous components onto the FPGA and optimizes some metrics, such as wirelength. Finally, routing determines the precise wiring layout needed to connect the placed components.

Placement plays an important role since the location mapping of components highly affects the final timing performance. In FPGA design, the fixed positions of logic resources, as shown in Fig. 1(a), constrain the mapping process, making it a highly complex combinatorial optimization problem. Solving this problem is exactly NP-hard due to the exponential number of possible mappings required to meet timing and other design constraints.

Typical FPGA placement algorithms can be categorized into simulated annealing and analytical methods. The simulated annealing-based FPGA placement algorithms might lead to long placement runtime when the input netlist is large [4]. In contrast, analytical-based methods can achieve high scalability and quality [5]–[11]. However, these analytical placement methods often result in suboptimal placements, leading to detours routing, especially when applied to CNN accelerators with wider bit widths.

†Equal contribution. *Corresponding author.



(a) (b)

Fig. 1 (a) Xilinx UltraScale+ FPGA layout; (b) CNN architecture.

An intuitive method is using the datapath information in the placement stage to alleviate detours [12]–[18]. Different from ASIC, FPGA has a column-wise heterogeneous resource distribution, which brings a challenge to datapath-driven placement. Previously, Koch *et al* cast the placement of all components on regular datapath as a 0-1 integer linear program (ILP) problem [19], [20]. However, this method has an unacceptable running time when placing all heterogeneous datapath components of the CNN accelerator on the modern FPGA. To reduce running time, Kong *et al* propose to place processing engine (PE) arrays instead of all datapath components by the ILP formulation. Despite this effort, the empirically hierarchical design partitioning causes timing degradation, compared with the FPGA commercial tool [21]. Moreover, while these approaches incorporate regularity into the placement process, they fail to account for datapath-specific information, which is crucial for optimizing CNN accelerators.

In the CNN accelerator, the datapath primarily consists of regular processing units (PU) responsible for computations, as shown in Fig. 1(b). Each PU is composed of multiple PE arrays, which are primarily implemented using DSPs [22], [23]. Therefore, leveraging the DSP interconnection topology as a datapath provides a natural avenue for integrating it into FPGA placement. In [24], graph clustering forms graph cliques, minimizing the width and height of the smallest rectangle and bounding all DSPs. However, this approach struggles to scale for large designs, such as CNN accelerators. Similarly, in [25], empirical rules are proposed for DSP placement, such as assigning PEs to DSP columns within a region. However, these rules are not formalized into an automated algorithm. As computational demands increase and design scales expand, traditional computational architectures are insufficient. To overcome these challenges, advanced architecture, such as systolic array, have gained attention, especially in the CNN accelerator domain. The state-of-the-art systolic array placement method, R-SAD [26], introduces a dedicated analytical algorithm to exploit systolic array regularity fully, achieving significant wirelength reduction. However, its specialized nature limits its applicability to CNN accelerators with more diverse architectures.

In this paper, we present a datapath-driven DSP placement framework designed to support various FPGA-based CNN accelerator architectures. To enhance timing performance, the framework focuses on datapath DSP extraction and datapath-driven DSP placement. The

key contributions are summarized as follows:

- We introduce DSPlacer, a framework that iteratively enhances datapath DSP placement by cooperating with state-of-the-art placers.
- We propose a novel placement methodology incorporating graph convolution networks (GCNs) for datapath DSP identification, datapath graph construction, min-cost-flow DSP assignment, and ILP-based cascade constraint legalization. This approach cascades datapath DSPs to achieve a compact layout while preserving direct datapath connections between the processing system and programmable logic.
- We evaluate DSPlacer on multiple academic benchmarks, comparing it with AMD Xilinx Vivado 2020.2 and AMF-Placer 2.0. Experimental results show that DSPlacer improves Worst Negative Slack (WNS) by 32% and 65%, respectively, demonstrating its effectiveness and superiority.

II. Preliminaries

A. FPGA Architecture

This paper focuses on the prominent Xilinx UltraScale+ architecture [27] for heterogeneous FPGAs, having a column-wise heterogeneous resource distribution. As shown in Fig. 1(a), the configurable logic blocks (CLBs), digital signal processors (DSPs), and block random-access-memories (RAMs) and IO components are variably arranged across the FPGA fabric. Besides, PS, such as the CPU, is fixed and placed at the bottom left of the device.

B. Problem Formulation

Our DSPlacer takes a pre-implementation netlist and DSP specifications of a target FPGA as inputs. A pre-implementation netlist defines the connections among various logic components, while the DSP specifications indicate the available DSP locations on the FPGA. Using our datapath DSP placement results as constraints, off-the-shelf FPGA PnR tools are used to perform other component placement and routing. We aim to maximize timing performance while ensuring all component placements meet design rules. Our problem formulation is formally defined as follows:

Problem 1 (DSP placement for FPGA-based CNN accelerator)**.** *Given a pre-implementation netlist of the CNN accelerator and DSP specifications, place all datapath DSPs in legal locations to maximize timing performance after placing all other components and routing all nets.*

C. Overview

To address Problem 1, we propose DSPlacer, a DSP placement framework tailored for FPGA-based CNN accelerators. Our DSPlacer overall flow is shown in Fig. 2 and comprises two main stages: datapath DSP extraction and datapath DSP placement, with our key contributions highlighted in purple and green.

We take a pre-implementation netlist of the CNN accelerator and DSP specifications as inputs. Firstly, we initialize the placement of all components using the off-the-shelf FPGA placement tool to generate an initial placement solution. Meanwhile, in the datapath DSP extraction stage, a pre-implementation netlist is represented as a graph, and a GCN are used to distinguish datapath DSPs from all DSP components by embedding features for each node. Then, a graph construction procedure is developed to transfer a pre-implementation netlist to a datapath DSP graph, which only involves DSP components with their potential datapath. In the datapath-driven DSP placement stage, datapath DSPs and other components are placed alternatively and iteratively. In particular, the placement of datapath

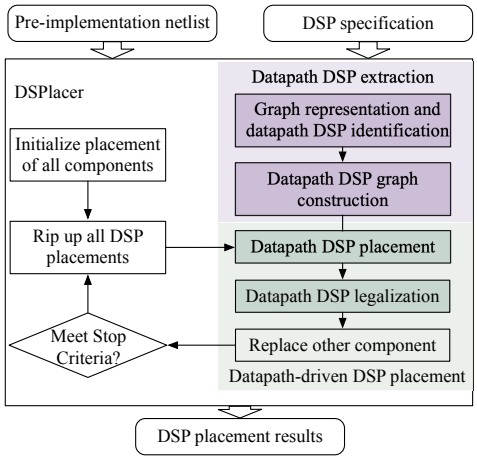
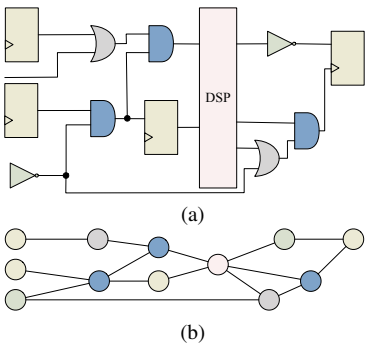


Fig. 2 Our DSPlacer overall flow.



Architecture	
Input Layer	[x,32]
Hidden Layer	[32,32]
MLP Layer	[32,32]
Output Layer	[32,2]
Activation	ReLU
Classification	Softmax

Training Setting	
Optimizer	Adam
Learning Rate	0.001
Dropout	0.35

(c)

Fig. 3 (a) Netlist; (b) Graph representation; (c) GCN configurations.

DSPs is first formulated as a 0-1 quadratic programming problem. To reduce computational complexity, this formulation is further refined into an MCF problem, where the unimodular property of the MCF model guarantees integer solutions. Following this, to strictly enforce cascade constraints, DSPs are legalized both inter-column and intra-column using ILP. Once the DSP placement is finalized, DSPlacer outputs the DSP placement results. Using our output DSP placement results as constraints, the off-the-shelf FPGA PnR tool iteratively places other components and performs routing to generate the final layout. According to this flow as shown in Fig. 2, our DSPlacer can account for datapath-specific information, resulting in improved timing performance for FPGA-based CNN accelerators.

III. Datapath DSP Extraction

This section details our datapath DSP extraction process, which consists of two main steps: (1) graph representation and datapath DSP identification and (2) datapath DSP graph construction. Given a pre-implementation netlist, a well-trained GCN model is adopted to identify datapath DSP, and a DSP graph guides the assignment step.

A. Graph representation and datapath DSP identification

A pre-implementation netlist can be represented as a graph $\mathbb{G} = (\mathcal{V}, \mathcal{E})$, as shown in Figs. 3(a) and 3(b). Let the node set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ represents n components in the netlist, and the edge set $\mathcal{E} = \{e_1, e_2, \ldots, e_m\}$ represent m connections. Typically, there are heterogeneous components in a pre-implementation netlist.

To identify the datapath DSP in a netlist, we formulate this task as a node classification problem handled by the GCN framework. Starting

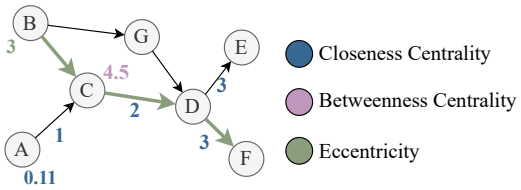


Fig. 4 1. Closeness centrality of A is the reciprocal of the sum of 1, 2, 3, and 3. 2. Betweenness centrality of C is calculated as $1+1+1+0.5+0.5+0.5=4.5$. 3. Eccentricity of B equals the distance from B to F.

from the pre-implementation netlist, we conduct network analysis to extract node features, including centrality metrics and connectivity properties. The dataset is then split into training and test sets, and the trained model is employed to identify datapath DSP nodes in unseen netlists. Specifically, the model consists of two graph convolutional layers, each with 32 hidden units, followed by three fully connected layers with softmax activation. Dropout regularization is applied to mitigate overfitting. The detailed configuration of the GNN model and training setup is presented in Fig. 3(c). To address the class imbalance in the dataset, a weighted loss function is utilized, assigning higher penalties to minority class misclassifications based on class ratios.

Node features play a critical role in classification accuracy. In PADE [28], datapath regularity is exploited by extracting graph automorphism features to classify datapaths. However, while this method identifies local regularities, it struggles to capture global graph properties, which are critical for understanding datapath connectivity and overall functionality. To address this limitation, our approach assigns each node a feature vector $\boldsymbol{x}$ that captures both local and global graph properties. Specifically, the feature vector includes: (a) closeness centrality, (b) feedback loops, (c) eccentricity, (d) indegree, (e) outdegree, (f) betweenness centrality, and (g) the average shortest path distances to other DSP nodes. While most features apply to all nodes, the average shortest path distance is unique to DSP nodes, representing the mean distance to other DSPs. Indegree and outdegree quantify the number of incoming and outgoing neighbors, while feedback loops commonly represent control path feedback. By incorporating global and local metrics, our approach offers a broader evaluation of graph properties. Definitions and Fig. 4 for betweenness centrality, eccentricity, and closeness centrality are provided.

Definition 1 (Betweenness Centrality)**.** *The betweenness centrality of a node is the sum of the fraction of all pairs of shortest paths that pass through this node.*

It can be calculated as follows.

$$\mathrm{C_B}(v) = \sum_{u_i, u_j \in \mathcal{V}} \frac{\sigma(u_i, u_j) \| v}{\sigma(u_i, u_j)}, \tag{1}$$

where $\sigma(u_i, u_j)$ is the number of shortest paths between nodes u_i and u_j. $\sigma(u_i, u_j) \| v$ is the number of those paths passing through the node v, with $v \neq u_i$ and $v \neq u_j$.

Definition 2 (Closeness Centrality)**.** *The closeness centrality of a node is the reciprocal of the sum of the shortest path distances from this node to all other nodes in the network.*

It can be calculated as follows.

$$\mathrm{C_C}(v) = \frac{1}{\sum_{u \neq v} d(v, u)}, \tag{2}$$

where $d(v, u)$ is the shortest path distance from node v to node u, with $v \neq u$.

Definition 3 (Eccentricity)**.** *The eccentricity of a node is the maximum shortest path distance from this node to any other node in the network.*

It can be calculated as follows.

$$\mathrm{ecc}(v) = \max_{u \in \mathcal{V}} d(v, u), \tag{3}$$

where $d(v, u)$ is the shortest path distance from the node v to the node u, with $v \neq u$.

Betweenness centrality measures how frequently a node acts as an intermediary along the shortest path between any two other nodes in the network, highlighting its role in information flow. As observed in [29], control logic DSPs, which direct signals to manage datapath blocks, typically have higher betweenness centrality than datapath DSPs. Closeness centrality, inversely proportional to the average distance to all other nodes, is generally higher for control path DSPs due to the extensive signals they receive. In contrast, eccentricity tends to have values for datapath DSPs that are more distributed towards either the higher or lower extremes of the overall range, reflecting the distributed placement for localized computations.

B. Datapath DSP graph construction

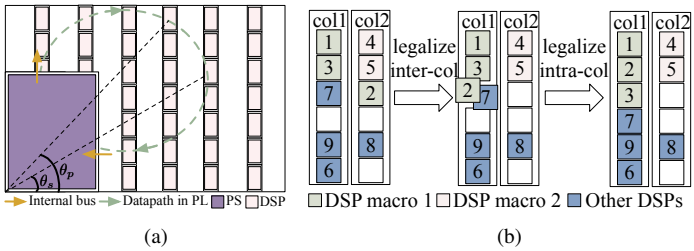
To construct the DSP graph, we perform a search algorithm directly on the netlist to capture the datapath of DSPs and compute their average shortest path distances. Common search algorithms such as Depth-First Search (DFS) and Breadth-First Search (BFS) face limitations in this context: DFS may fail to find the shortest paths, while BFS has high space complexity, making it impractical for large-scale netlists. To address these issues, we adopt an Iterative Deepening Depth-First Search (IDDFS) method, which combines the space efficiency of DFS with the ability to identify shortest paths. This algorithm ensures efficient traversal and supports the accurate construction of the DSP graph.

The construction procedure begins with a netlist traversal to extract all DSP nodes. IDDFS is then applied to each DSP node to compute the shortest paths to other DSPs, recording the paths, the cell type, and the number of cells along each path. This information is then used to construct a DSP graph, retaining only DSP nodes and their connectivity. The resulting graph effectively captures the topology and shortest path distances among DSPs, embedding critical dataflow information for downstream tasks.

However, after constructing an initial DSP graph, a refinement step is necessary. Initial DSP graphs include both datapath DSPs and others, such as control path DSPs. Incorporating control path DSPs into the DSP graph placement can result in a less compact datapath layout, potentially degrading the improvements in timing performance. Our observations reveal that control path DSPs are typically associated with more storage elements, such as flip-flops and RAMs, which facilitate signal holding and allow for greater placement flexibility and less compactness. In contrast, datapath DSPs, characterized by fewer storage elements, require tighter placement to maintain performance. Based on this insight, DSPlacer retains only the datapath DSPs identified by the GCN and removes the others from the DSP graph. This ensures that datapath DSP placement is prioritized for a more compact and efficient layout. After the datapath DSPs are placed, the placement of control path DSPs is handled by standard PnR tools, ensuring a balanced and effective placement strategy.

IV. Datapath-driven DSP placement

In this section, we introduce a datapath-driven DSP placement strategy. Our approach primarily encompasses (1) datapath DSP placement



(a) (b)

Fig. 5 (a) DSP datapath and (b) Datapath DSP legalization.

and (2) datapath DSP legalization. Utilizing the datapath DSP graph, an optimization technique is employed to allocate DSP locations. Then, a subsequent legalization process is applied to ensure strict compliance with the cascade constraints.

A. Datapath DSP placement

In an FPGA device, we assume that there are M DSP locations and each with coordinate $(p_{x,j}, p_{y,j})$, where $j \in \{1, 2, \cdots, M\}$ represents the index of each DSP location. Unlike ASIC placement, DSP placement is an assignment problem. In this context, all DSP components in the design netlist must be mapped to specific DSP locations within the device to optimize timing performance. We introduce binary variables $x_{i,j} \in \{0, 1\}$ to represent the mapping of each DSP component to a location. Specifically, if $x_{i,j} = 1$, the i-th DSP component is mapped to the j-th DSP location in the device, where $i \in \{1, 2, \cdots, N\}$, and $j \in \{1, 2, \cdots, M\}$. N is the number of datapath DSP components in the netlist identified in Section III and M is the number of available DSP locations in the device. $N \leq M$ ensures that all DSP components can be placed within the device. Therefore, we intuitively identify two constraints, as follows.

$$\sum_{j=1}^{M} x_{i,j} = 1, \qquad \sum_{i=1}^{N} x_{i,j} \leq 1, \tag{4}$$

where the first constraint ensures that each DSP component must be mapped to a DSP location, and the second states that each DSP location can be occupied by at most one DSP component.

In addition to constraint (4), DSP macros must also be considered. DSP components within the same macro must be assigned to adjacent positions within the same column. Assume that the DSP location list is sorted in ascending order of coordinates, such that adjacent locations within the same column have consecutive indices. This enables us to express the cascade constraint as follows.

$$x_{c_p,j} = x_{c_s,j+1}, \quad \forall c \in C, \tag{5}$$

where C denotes the set of cascaded DSP component pairs within DSP macros, with c_p and c_s as the predecessor and successor indices of each pair, respectively. The constraint (5) ensures that each pair of cascaded DSP components and their corresponding locations are either simultaneously placed or not placed.

In addition to the constraints above, we incorporate datapath information as observed in Fig. 5(a). In modern FPGA devices (e.g., Xilinx ZCU104), the PS is fixed at the bottom-left corner. Data buses transferring from PS to PL are located above the PS, while those from PL to PS are on the right. This layout forms an ideal datapath from the top to the right of the PS. To facilitate smooth routing, we encourage the angle of the predecessor DSP location to be larger than that of the successor for each DSP graph edge. Using the cos function, the soft datapath constraint is formulated as follows.

$$\cos\theta_{e_{Dp}} \leq \cos\theta_{e_{Ds}}, \tag{6}$$

where $\cos\theta_{e_{Dp}} = \sum_{j=1}^{M} x_{e_{Dp},j} p_{x,j} / \sqrt{p_{x,j}^2 + p_{y,j}^2}$ is the cosine angle between the predecessor and horizontal line, and $\cos\theta_{e_{Ds}} = \sum_{j=1}^{M} x_{e_{Ds},j} p_{x,j} / \sqrt{p_{x,j}^2 + p_{y,j}^2}$ is the cosine angle between the successor and horizontal line. However, different from constraints (4) and (5), we take the datapath constraint as a penalty term in the objective to encourage datapath DSP placement to satisfy our desired datapath.

Under the constraints defined in (4) (5) and penalty term in (6), our objective in datapath DSP placement is to minimize the distance between any two connected datapath DSP components and between a datapath DSP component and any fixed-location component. Thus, our mathematical formulation is defined as follows.

$$\begin{aligned} \min_{x_{i,j}} & \sum_{e \in \mathcal{E}} (\boldsymbol{x}_{e_p} - \boldsymbol{x}_{e_s})^\top (\boldsymbol{p}_x \boldsymbol{p}_x^\top + \boldsymbol{p}_y \boldsymbol{p}_y^\top)(\boldsymbol{x}_{e_p} - \boldsymbol{x}_{e_s}) \\ & + \lambda \sum_{e_D \in \mathcal{E}_D} (\cos\theta_{e_{Dp}} - \cos\theta_{e_{Ds}}) \\ & + \eta \sum_{c \in C} \sum_{j=1}^{M-1} (x_{c_p,j} - x_{c_s,j+1})^2, \\ \text{s.t.} \quad & (4), x_{i,j} \in \{0, 1\}. \end{aligned} \tag{7}$$

where $\mathcal{E}$ denotes the edge set in the netlist graph and $\mathcal{E}_D$ denotes the edge set in the datapath DSP graph. $\boldsymbol{x}_{e_p}$ and $\boldsymbol{x}_{e_s}$ represent the assignment variables for any two connected components via an edge in the netlist graph. Note that $\boldsymbol{x}_{e_p}$ or $\boldsymbol{x}_{e_s}$ is constant if the corresponding component is not a datapath DSP. These variables will be determined by the off-the-shelf FPGA placement tool. $\boldsymbol{p}_x$ ($\boldsymbol{p}_y$) denotes the x (y) coordinate value vector for DSP locations. $\theta_{e_{Dp}}$ ($\theta_{e_{Ds}}$) represents the angle between the horizontal line and the predecessor (successor) of the datapath DSP for the edge e in the datapath DSP graph. $x_{i,j}$ represents the scalars in $\boldsymbol{x}_{e_i}$ and $\boldsymbol{x}_{e_j}$. λ is a hyperparameter that controls the trade-off between distance and datapath effort. $\eta \sum_{c \in C} \sum_{j=1}^{M-1} (x_{c_p,j} - x_{c_s,j+1})^2$ indicates that constraint (5) has been relaxed to penalty, where η is the penalty factor. We rewrite Formulation (7) as follows.

$$\begin{aligned} \min_{x_{i,j}} & \sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{p=1}^{N} \sum_{q=1}^{M} a_{i,j,p,q} x_{i,j} x_{p,q} + \lambda \sum_{i=1}^{N} \sum_{j=1}^{M} c_j x_{i,j}, \\ \text{s.t.} \quad & (4), x_{i,j} \in \{0, 1\}. \end{aligned} \tag{8}$$

where $a_{i,j,p,q}$ and c_j are quadratic and linear coefficients, respectively.

An ILP solver can solve Formulation (8) by replacing each quadratic term with a binary variable and adding artificial constraints. However, as the design scale increases, the variables grow significantly, leading to prohibitive runtimes. Inspired by [30], we adopt the following heuristic to linearize term $x_{i,j} \cdot x_{p,q}$ and solve iteratively.

$$a_{i,j,p,q} x_{i,j} x_{p,q} \approx \frac{1}{2} a_{i,j,p,q} (x_{i,j}^{'} x_{p,q} + x_{i,j} x_{p,q}^{'}), \tag{9}$$

where $x_{i,j}^{'}$ and $x_{p,q}^{'}$ are the values of $x_{i,j}$ and $x_{p,q}$ from the previous iteration. Using the linearization technique in Equation (9), the objective function in Formulation (8) is a weighted sum of all the $x_{i,j}$, which can be solved through a min-cost network flow model [30]. The weighted sum of $x_{i,j}$ variables can be viewed as several assignments from DSP components to DSP locations, with constraint (4) integrated into the flow model by specifying edge capacities.

B. Datapath DSP legalization.

In Section IV-A, the cascade constraint (5) is introduced and relaxed into a penalty term within the MCF formulation as a soft constrain. However, while constraint (5) can be satisfied by finding the optimal solution to formulation (7), the linearization method does not guarantee optimality, and violations of the constraint may therefore occur. To

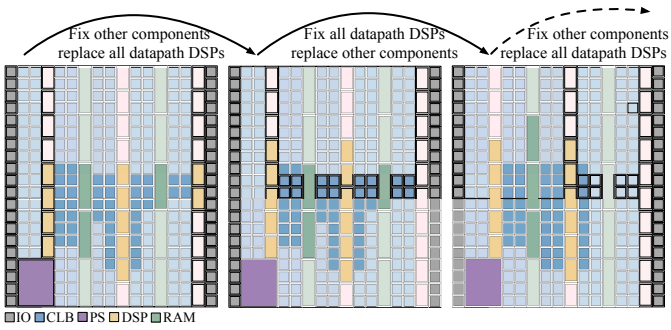


Fig. 6 Incremental Datapath DSP Placement.

address these violations, we introduce a legalization process as a post-processing stage to refine the results obtained from the MCF model. The goal is to enforce cascade constraints while minimizing wire length overhead and preserving the datapath, as guided by the penalty term in (6). To do that, DSP components are adjusted to remain close to their original positions. As shown in Fig. 5(b), the legalization process involves two sequential steps: inter-column legalization to adjust placements across columns, followed by intra-column legalization to refine placements within each column.

For inter-column legalization, we minimize the horizontal displacement by solving the following ILP problem.

$$\min_{t_{i,j}} \sum_{i=1}^{N} \sum_{j=1}^{N_{col}} D_{col}(i,j) t_{i,j}, \tag{10}$$

$$\text{s.t.} \quad \sum_{j=1}^{N_{col}} t_{i,j} = 1, \quad \sum_{i=1}^{N} t_{i,j} \le M_j, \tag{10a}$$

$$t_{c_p,j} = t_{c_s,j}, \quad \forall c \in C, \tag{10b}$$

$$t_{i,j} \in \{0,1\},$$

where $t_{i,j}$ is a binary variable indicating the column to which the DSP component is assigned. N_{col} denotes the number of columns on the device. $D_{col}(i,j)$ denotes the horizontal distance between the newly assigned column and the original column if the i-th DSP component is assigned to the j-th column. $t_{c_p,j}$ ($t_{c_s,j}$) denotes the variable for the predecessor (successor) of the cascaded DSP component pairs, and M_j denotes the number of DSP locations in the j-th column. Constraints (10a) (10b) ensure the solution satisfies constraints (4) (5) at the column level. Since the number of DSP columns is much smaller than the number of DSP locations, the ILP solver remains efficient.

For intra-column legalization, we process each column in parallel and minimize the vertical displacement. For the j-th column, assume that the indices of the DSP components in the column are consistent with the order after sorting by vertical locations. DSP components within a macro are sorted by the average vertical location of the corresponding macro, while other DSP components are sorted by their individual vertical locations. We then solve the formulation below:

$$\min_{r_i} \sum_{i=1}^{N_j} |r_i - R_{col}(i)|, \tag{11}$$

$$\text{s.t.} \quad x_i \in \{1,2,...,M_j\},$$

$$r_{i+1} - r_i = 1, \forall (i,i+1) \in C_j, \tag{11a}$$

$$r_{i+1} - r_i \ge 1, \forall (i,i+1) \notin C_j, \tag{11b}$$

where r_i is an integer variable that indicates the row index of the location to which the i-th DSP component in the column will be assigned. N_j denotes the number of DSP components in the j-th column, $R_{col}(i)$ denotes the original row index of the i-th DSP component in the j-th column and C_j represents cascaded DSP component pairs within the j-th column. Constraint (11a) ensures that the successor is placed immediately after the predecessor in the cascaded pairs, and constraint (11b) allows space between non-cascaded DSP components while preventing overlap.

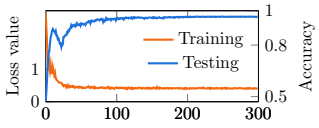
In the datapath DSP placement stage, as shown in Fig. 6, our placement is performed incrementally, alternating between placing datapath DSPs and other components in each iteration. During each iteration, we first fix the positions of the other components and place all the datapath DSPs, then fix the positions of the datapath DSPs and place the remaining components. This method incorporates datapath information while alleviating detours caused by the datapath-driven approach.

TABLE I Benchmarks detail.

Design	#LUT	#LUTRAM	#FF	#BRAM	#DSP	DSP%	freq.(MHz)
iSmartDNN	53503	2919	55767	122	197	11%	130.0
SkyNet	43146	2748	51410	192	346	20%	150.0
SkrSkr-1	35743	3611	53887	196	642	37%	195.0
SkrSkr-2	70558	3815	64007	196	1180	68%	175.0
SkrSkr-3	70382	3791	67257	196	1431	83%	175.0

	SVM [28]	GCN
iSmartDNN	80%	88.1%
SkyNet	85%	90%
SkrSkr-1	96%	96%
SkrSkr-2	69%	95.5%
SkrSkr-3	78.7%	96.6%

(a)



(b)

Fig. 7 (a) Datapath DSP identification comparison; (b) Training and Testing.

V. Experimental Results

A. Target Device, Benchmark and Environment

In the DSPlacer framework, the datapath DSP extraction and DSP assignment tasks are relatively independent. To facilitate implementation, the datapath DSP extraction was developed in Python, utilizing the NetworkX [31] library for graph construction and the PyTorch [32] Geometric library for the graph learning model. For the DSP assignment, we implemented the MCF in C++ and used the Lemon solver [33] for assignment tasks. The cascade legalization after MCF was solved using Gurobi [34]. For placement, routing, and timing analysis of other components, we employed AMF-Placer 2.0 [11], [35] and Xilinx Vivado 2020.2. Since Vivado lacks a built-in wirelength calculation function, RapidWright [36] was used to compute the wirelength. All experiments were conducted on a Linux machine with 10 cores, targeting the Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA. It is worth mentioning that AMF-Placer 2.0, as an academic open-source placer, originally only supported the Xilinx VCU108 platform. Significant effort was dedicated to adapting it for ZCU104. The benchmarks include several designs from the DAC System Design Contest: iSmartDNN [37], SkyNet [38], and SkrSkr [39].

B. GCN model training and performance

Our classification methodology leverages global graph properties to capture structural importance and node connectivity within DSPs, forming a robust basis for classification. In contrast, PADE [28] only employs automorphism-based features to extract datapath regularity. To evaluate the model, we adopt a leave-one-out strategy across five benchmarks: four benchmarks are used for training, and the resulting

TABLE II Experiment Result.

Benchmark	Vivado				AMF				DSPlacer			
	WNS (ns)	TNS (ns)	HPWL (um)	Runtime (s)	WNS (ns)	TNS (ns)	HPWL (um)	Runtime (s)	WNS	TNS	HPWL (um)	Runtime (s)
iSmartDNN	-0.131	-0.391	2965232	771	-0.830	-649.995	5152227	1552	**0.151**	**0**	3107487	865
SkyNet	-0.164	-7.765	3482155	868	-0.154	-3.557	3633692	970	**0.189**	**0**	3878462	940
SkrSkr-1	-0.587	**-97.799**	2488788	775	-0.883	-4043.358	3384970	691	**-0.164**	-109.042	4473110	701
SkrSkr-2	-0.597	-826.739	3700116	1091	-1.865	-16672.268	8809141	4486	**0.009**	**0**	4489697	2462
SkrSkr-3	-0.216	-21.417	4034489	1232	-0.786	-804.950	6104589	3329	**0.007**	**0**	4912782	1755
Normalize	1.325×	1.042×	0.550×	0.485×	1.658×	1.103×	1.446×	2.145×	**1.000**×	**1.000**×	1.000×	1.000×

model is tested on the remaining benchmark. This process is repeated for all benchmarks to ensure comprehensive evaluation. Our GCN-based approach achieves an average accuracy of 96%, significantly outperforming PADE's SVM-based model, which achieves around 81% accuracy on average, as shown in Fig. 7(a). Additionally, an accuracy curve is presented in Fig. 7(b). These results underscore the advantages of global centrality features over local automorphism-based methods, showcasing superior classification performance and adaptability for node classification in netlists.

C. Placement setting

To evaluate the performance of DSPlacer across varying DSP counts, we extended the SkrSkr benchmark suite by incorporating a broader spectrum of DSP configurations, resulting in three distinct benchmarks: SkrSkr-1, SkrSkr-2, and SkrSkr-3. Our experiments utilize five benchmarks with DSP counts ranging from 197 to 1431, as detailed in TABLE I. To demonstrate the advantage of DSPlacer, we first use Vivado for placement while progressively increasing the clock frequency for each benchmark until a negative WNS is observed. At the same frequency, DSPlacer is then employed for placement. If DSPlacer avoids negative WNS under the same conditions, its advantages are validated. Furthermore, the number of iterations of DSPlacer's internal MCF algorithm is set to 50. The hyperparameter λ, which governs the trade-off between minimizing distance and preserving datapath integrity, is set to 100 based on the experiment.

D. Placement performance comparison

To evaluate the timing improvements achieved by DSPlacer, we report post-route PPA metrics, including routed wirelength, setup WNS, TNS, and total runtime, as summarized in TABLE II. While AMF is tailored for the VCU108 platform, its performance, and adaptability to other platforms, such as ZCU104, is limited. For WNS and TNS, DSPlacer achieves improvements of 65% and 10%, respectively, with reduced wirelength and runtime. Compared to Vivado, DSPlacer improves setup WNS by an average of 32.5% and up to 60% in the benchmark case SkrSkr-2, at the cost of an additional half runtime and HPWL. These results demonstrate that DSPlacer delivers superior WNS and TNS performance compared to Vivado and AMF-Placer while offering greater universality and robustness than AMF-Placer.

E. Analysis, Profiling, and Visualization

The timing improvement achieved by DSPlacer derives from satisfying two critical datapath placement requirements: (1) cascading datapath DSPs to create a more compact layout and (2) preserving the datapath information between PS and PL. These optimizations significantly reduce total WNS. However, the trade-off for the compactness is a slightly increased routing time due to a medium congestion level.

Fig. 8 presents the runtime breakdown of DSPlacer on iSmartDNN and SkyNet. The prototype placement solution and other component placements dominate the total runtime (90.61% for iSmartDNN and 88.31% for SkyNet), whereas datapath DSP extraction and datapath-driven DSP placement account for only about 2% of the total runtime.

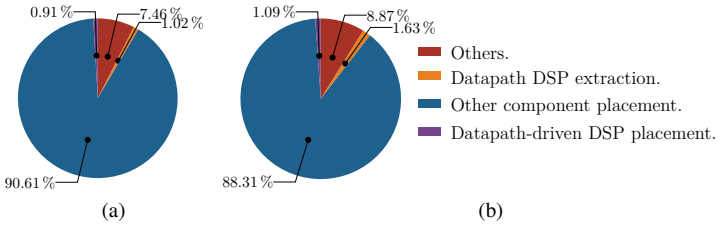


Fig. 8 Runtime profiling: (a) iSmartDNN and (b) SkyNet.

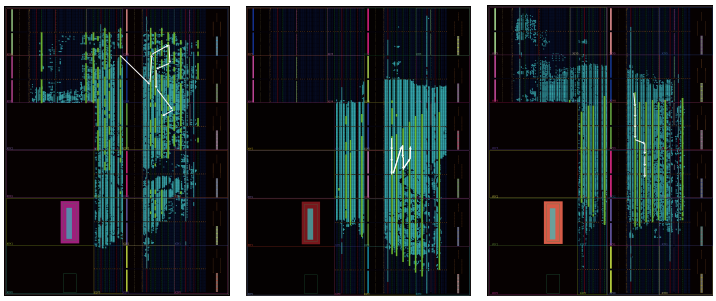


Fig. 9 Datapath visualizations of the SkrSkr-1 placement layout within different design tools: (a) Vivado, (b) AMF, and (c) DSPlacer.

Fig. 9 visualizes the post-route layouts of SkrSkr using different placement tools. Compared to Vivado, DSPlacer produces a more compact and regular datapath layout. While AMF achieves a compact layout similar to DSPlacer, it fails to maintain the datapath information between PS and PL, resulting in a disordered datapath.

VI. Conclusion

In this paper, we propose DSPlacer, a DSP placement framework for FPGA-based CNN accelerators, designed to enhance timing performance. DSPlacer combines GCN techniques with optimization methods to address the challenges of DSP placement. Specifically, the framework extracts datapath DSPs and their shortest path information to enable precise identification of critical DSP nodes. The placement optimization process is formulated as a 0-1 integer programming problem, further simplified into a min-cost-flow model to improve computational efficiency. By integrating datapath-specific information into the placement stage, DSPlacer complied with cascading constraints while ensuring timing closure. Experimental results show a 32% improvement in WNS compared to Vivado and a 65% improvement over AMF. These results highlight the framework's ability to achieve substantial timing performance improvements.

Acknowledgement

This work is supported in part by the National Key Research and Development Program of China (No. 2023YFB4402900), the National Natural Science Foundation of China (No. 62304197) and Pangomicro.

REFERENCES

[1] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management." in *Proc. DAC*, 2019, pp. 125–1.

[2] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN co-design: An efficient design methodology for 1ot intelligence on the edge," in *Proc. DAC*. IEEE, 2019, pp. 1–6.

[3] Y. Zhang, X. Zhang, P. Xu, Y. Zhao, C. Hao, D. Chen, and Y. Lin, "AutoAI2C: An automated hardware generator for DNN acceleration on both FPGA and ASIC," *IEEE TCAD*, vol. 43, no. 10, pp. 3143–3156, 2024.

[4] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker *et al.*, "VTR 8: High-performance CAD and customizable FPGA architecture modelling," *ACM TRETS*, vol. 13, no. 2, pp. 1–55, 2020.

[5] C.-W. Pui, G. Chen, W.-K. Chow, K.-C. Lam, J. Kuang, P. Tu, H. Zhang, E. F. Young, and B. Yu, "RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs," in *Proc. ICCAD*. IEEE, 2016, pp. 1–8.

[6] R. Pattison, Z. Abuowaimer, S. Areibi, G. Gréwal, and A. Vannelli, "GPlace: A congestion-aware placement tool for ultrascale FPGAs," in *Proc. ICCAD*. IEEE, 2016, pp. 1–7.

[7] Y.-C. Kuo, C.-C. Huang, S.-C. Chen, C.-H. Chiang, Y.-W. Chang, and S.-Y. Kuo, "Clock-aware placement for large-scale heterogeneous FPGAs," in *Proc. ICCAD*. IEEE, 2017, pp. 519–526.

[8] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," *IEEE TCAD*, vol. 37, no. 4, pp. 869–882, 2018.

[9] W. Li, Y. Lin, M. Li, S. Dhar, and D. Z. Pan, "UTPlaceF 2.0: A high-performance clock-aware FPGA placement engine," *ACM TODAES*, vol. 23, no. 4, pp. 1–23, 2018.

[10] T. Liang, G. Chen, J. Zhao, S. Sinha, and W. Zhang, "AMF-Placer: High-performance analytical mixed-size placer for FPGA," in *Proc. ICCAD*, 2021, pp. 1–9.

[11] ——, "AMF-Placer 2.0: Open-source timing-driven analytical mixed-size placer for large-scale heterogeneous FPGA," *IEEE TCAD*, vol. 43, no. 9, pp. 2769–2782, 2024.

[12] T. T. Ye and G. De Micheli, "Data path placement with regularity," in *Proc. ICCAD*. IEEE, 2000, pp. 264–270.

[13] S. I. Ward, M.-C. Kim, N. Viswanathan, Z. Li, C. J. Alpert, E. E. Swartzlander, and D. Z. Pan, "Structure-aware placement techniques for designs with datapaths," *IEEE TCAD*, vol. 32, no. 2, pp. 228–241, 2013.

[14] S. Chou, M.-K. Hsu, and Y.-W. Chang, "Structure-aware placement for datapath-intensive circuit designs," in *Proc. DAC*, 2012, pp. 762–767.

[15] Z. He, P. Liao, S. Liu, Y. Ma, Y. Lin, and B. Yu, "Physical synthesis for advanced neural network processors," in *Proc. ASPDAC*, 2021, pp. 833–840.

[16] J.-M. Lin, W.-F. Huang, Y.-C. Chen, Y.-T. Wang, and P.-W. Wang, "DAPA: A dataflow-aware analytical placement algorithm for modern mixed-size circuit designs," in *Proc. ICCAD*. IEEE, 2021, pp. 1–8.

[17] D. Fang, B. Zhang, H. Hu, W. Li, B. Yuan, and J. Hu, "Global placement exploiting soft 2D regularity," in *Proc. ISPD*, 2022, pp. 203–210.

[18] A. B. Kahng and Z. Wang, "DG-RePlAce: A dataflow-driven GPU-accelerated analytical global placement framework for machine learning accelerators," *arXiv preprint arXiv:2404.13049*, 2024.

[19] A. Koch, "Module compaction in FPGA-based regular datapaths," in *Proc. DAC*, 1996, pp. 471–476.

[20] ——, "Structured design implementation: a strategy for implementing regular datapaths on FPGAs," in *Proc. FPGA*, 1996, pp. 151–157.

[21] H. Kong, L. Feng, C. Deng, B. Yuan, and J. Hu, "How much does regularity help FPGA placement?" in *Proc. FPT*, 2020, pp. 76–84.

[22] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. ICCAD*, 2016, pp. 1–8.

[23] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. FPGA*, 2017, pp. 45–54.

[24] S. Gowda, A. Parsons, R. Jarnot, and D. Werthimer, "Automated placement for parallelized FPGA FFTs," in *Proc. FCCM*, 2011, pp. 206–209.

[25] J. Zhang, W. Zhang, G. Luo, X. Wei, Y. Liang, and J. Cong, "Frequency improvement of systolic array-based CNNs on FPGAs," in *Proc. ISCAS*, 2019, pp. 1–4.

[26] H. Hu, D. Fang, W. Li, B. Yuan, and J. Hu, "Systolic array placement on FPGAs," in *Proc. ICCAD*. IEEE, Oct. 2023, p. 1–9.

[27] "AMD Zynq UltraScale+ MPSoCs: Heterogeneous multiprocessing platform for broad range of embedded applications."

[28] S. Ward, D. Ding, and D. Z. Pan, "PADE: a high-performance placer with automatic datapath extraction and evaluation through high dimensional data learning," in *Proc. DAC*, 2012, pp. 756–761.

[29] S. D. Chowdhury, K. Yang, and P. Nuzzo, "ReIGNN: State register identification using graph neural networks for circuit reverse engineering," in *Proc. ICCAD*. IEEE, 2021, p. 1–9.

[30] B. Yu, D. Liu, S. Chowdhury, and D. Z. Pan, "TILA: Timing-driven incremental layer assignment," in *Proc. ICCAD*, 2015, pp. 110–117.

[31] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[32] *Pytorch*, https://pytorch.org/.

[33] *LEMON*, https://lemon.cs.elte.hu/trac/lemon.

[34] *gurobi*, https://www.gurobi.com/.

[35] *AMF-Placer 2.0*, https://github.com/zslwyuan/AMF-Placer.

[36] *rapidwright*, https://www.rapidwright.io/.

[37] *iSmartDNN*, https://github.com/onioncc/iSmartDNN.

[38] *SkyNet*, https://github.com/ZhuangzhuangWu/SkyNet.

[39] *SkrSkr*, https://github.com/jiangwx/SkrSkr.