LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators

Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, Tulika Mitra
School of Computing, National University of Singapore
{*zhaoying, danwu20, dmd, tulika*}*@comp.nus.edu.sg*

***Abstract*—Spatial accelerators, such as Coarse-Grained Reconfigurable Arrays (CGRA), provide a promising pathway to scale the performance and power efficiency of computing systems. These accelerators depend on effective compilers to take advantage of the parallelism offered by the underlying architecture. Currently, the compilers are handcrafted for spatial accelerators, which is challenging from time to market perspective, especially with the rapid increase of diverse accelerators. In this paper, we present a portable compilation framework, called LISA, that can be tuned automatically to generate quality mapping for varied spatial accelerators. Our key contribution is to automatically identify the impact of the dataflow graph (DFG) structure characteristics (representing an application) on the mapping for a new accelerator. Towards this end, we abstract the DFG structure in graph attributes, use Graph Neural Network (GNN) to analyze the graph attributes, and identify the mapping impact for an accelerator architecture with an all-encompassing global view. Finally, we augment a simulated annealing-based mapping approach to take into account the impact of DFG structure in guiding the placement of the dataflow graph nodes and the routing of the dependencies on the accelerator. Our experimental evaluation concretely demonstrates the substantial benefit of our approach compared to the state-of-the-art solutions.**

***Keywords*-Spatial Accelerators; CGRA; Compiler**

I. Introduction

The spatial accelerators provide a promising way to continue scaling the performance and efficiency of computing hardware [1]–[3]. Recently, we have witnessed a rapid increase of spatial accelerators, such as domain-specific deep learning accelerators [4]–[7] and domain-agnostic CGRAs (Coarse-Grained Reconfigurable Array) [8]–[19]. Meanwhile, several accelerator generation frameworks [20]–[24] have been proposed to automatically generate spatial accelerators for a specific set of applications. But we need effective compiler support to fully realize the potential of the emerging diverse spatial accelerators.

The spatial accelerators are programmable, consisting of Processing Elements (PE) such as ALU and MAC, on-chip memory, and network-on-chip. Fig. 1 shows a 4×4 CGRA, a representative example of the spatial accelerator. The compiler first generates a Dataflow Graph (DFG) representing the application program and then maps the DFG on the spatial accelerator by respecting the data dependencies. The details of the architecture are exposed to the compiler for generating configurations for the programmable units. These details create an enormous search space for the compiler to place the operations onto the PEs and route the data from the producer PE to the consumer PEs. This compilation process is often referred to as *mapping*. Routing is the most time-consuming step in the mapping, and the placement of the operations on the PEs informs the routing decisions. Placement and routing are thus interrelated, affecting the compiler efficiency to find the quality mapping.

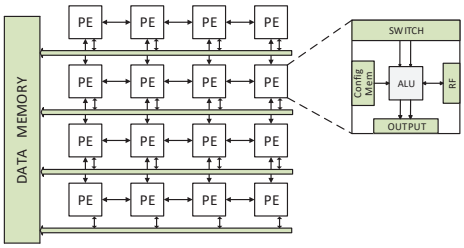


Figure 1: Spatial accelerator example: 4×4 CGRA

We classify the compilers for spatial accelerators into three categories: meta-heuristic approaches [25] [26] [22] such as simulated annealing, mathematical optimization (Polyhedral model [27], Integer Linear Programming (ILP) [28]) to solve a mathematical formulation of the mapping problem, and hybrid heuristic [29]–[34] that combines several scheduling methods and optimizes mapping by leveraging the architectural characteristics. Due to the accelerator architecture diversity and the large search spaces, the different classes of compilers trade off mapping quality, compilation time, scalability, and generality. Hybrid heuristics usually provide quality mapping for some specific architectures as they make full use of target architecture characteristics, while meta-heuristic methods work for diverse accelerators but cannot guarantee mapping quality. Mathematical optimization-based approaches can generate optimal solutions but usually need a long compilation time for a large search space and are hence not scalable. In summary, none of the compilation approaches can satisfy all the desired criteria of quality mapping, scalability, and easy adaptability to diverse accelerators.

With the rapid increase of varied accelerators, handcrafting quality compiler for each accelerator is challenging from a time-to-market perspective. Hence we need portable compilers to generate quality mapping for different accelerator and application combinations without manual effort. In other words, for a new accelerator architecture, the portable compiler should be able to automatically tune its parameters and the mapping decisions to generate superior quality mapping within reasonable compilation time.

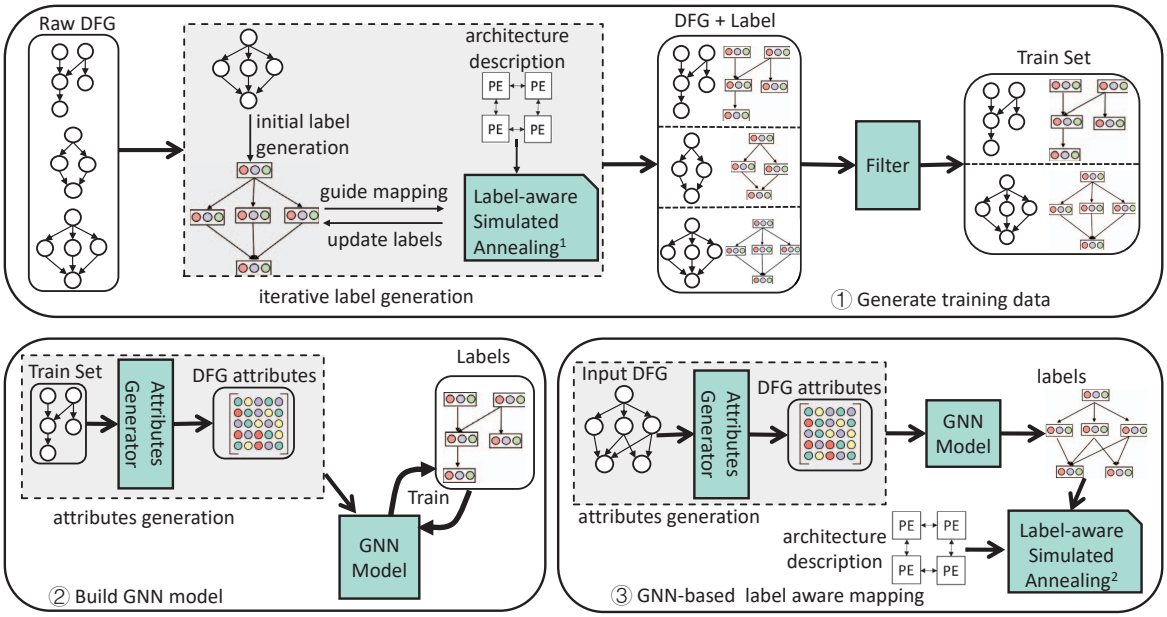


Figure 2: Overview of GNN-based portable mapping framework - LISA

While handcrafted compilers strive to optimize mapping by leveraging specific accelerator characteristics, portable compilers need to provide a unified way to generate quality mapping for varied accelerators. Towards this end, portable compilers need to understand the characteristics of the accelerator and how the DFG structure affects mapping on that accelerator. For example, an accelerator with sufficient routing resources can handle the dense communication among the DFG nodes. In contrast, an accelerator with fewer routing resources often needs more effort to map the dense parts of the DFG. Meanwhile, as the mapping of an operation influences the mapping of some other operations, the portable compiler needs to identify the strength of the association among the operations. Therefore, the compiler needs an all-encompassing global view of the DFG structure and the accelerator resources even when making local placement/routing decisions.

To map a DFG onto the spatial accelerator, our goal is to identify the impact of the DFG structure on mapping for a particular spatial accelerator and apply that knowledge to guide the mapping decisions. For example, as some DFG nodes that have complex dependencies with the predecessor and successor nodes are more difficult to route, we can set higher scheduling priority for these nodes. If we identify the impact of the DFG structure, we can avoid getting stuck in the local minima in our quest to find the quality mapping. In addition, different accelerators can have different mapping characteristics for the same DFG. For example, a small accelerator has a limited number of PEs and needs to map the DFG nodes along the temporal dimension in addition to the spatial dimension. In contrast, a big accelerator should take advantage of the spatial dimension as much as possible for parallelism. Therefore, we propose to derive the DFG mapping aspects considering both the DFG structure and the accelerator characteristic. Such aspects describe the predicted mapping information of DFG nodes and edges, e.g., the estimated routing resources required by DFG edges and the predicted mapping distance between DFG nodes. We name such special aspects *labels* and labels are used to inform compilers about placement, routing decisions that lead to quality mapping.

Automatic generation of labels is the key to tuning the portable compiler for different accelerators. We employ the Graph Neural Network (GNN) [35] to learn how to derive the labels. A GNN can aggregate the graph attributes, propagate information to neighboring nodes, and predict the desired aspects. In our case, the desired aspects are the labels, and input graph attributes of the GNN model are the inherent properties of the DFG. Moreover, the GNN is retrained to learn how to generate the labels for a specific accelerator. For a new accelerator, we generate a set of synthetic DFGs and use an iterative method to generate the labels to create a training dataset. It is worth noting that the iterative mapping method to generate the labels for training is extremely time-consuming and can only be used as a one-off technique. In comparison, once the GNN model has been re-trained for a new accelerator, it can generate the labels for a new DFG very fast.

We propose a framework LISA, Learning Induced mapping for Spatial Accelerators, to provide a portable compiler for different accelerator and application combinations. The main insight is that the GNN cannot directly generate the

mapping for a given DFG but the labels can bridge the gap between the GNN and the mapping. The GNN aggregates the DFG attributes to derive the labels; the labels inform the compiler about proper placement and routing decisions that lead to quality mapping. For a new accelerator, the portable compiler re-trains the GNN model to adapt the labels according to the accelerator characteristics.

Fig. 2 provides an overview of the LISA framework, which consists of three parts: training data generation for the GNN model, building the GNN model, and GNN-based label-aware mapping. For each accelerator, we generate a set of unlabelled DFGs and propose an iterative mapping method to assign labels for the nodes of these DFGs. We design a metric to filter the labels from the iterative mapping method to generate the training set for the GNN model. Then we design an Attributes Generator, which uses traditional graph algorithms, to collect the DFG structure information to generate the DFG attributes. The GNN then can process these attributes to derive the labels. We design a network for each label and train these networks on the dataset. Once a GNN has been trained for a specific accelerator, given a new DFG corresponding to a real application program, the trained GNN model can automatically derive the labels. Finally, we design a label-aware simulated annealing approach to map any new DFG on the accelerator with an all-encompassing global view.

Our concrete contributions are as follows:

- We propose a portable compilation framework LISA that works across different spatial accelerators. The framework is open-source and available from https://github.com/ecolab-nus/lisa
- We quantify the impact of the DFG structure towards mapping on a specific accelerator and provide a global view in the mapping process. This contrasts with existing works that only focus on the local view in the placement and routing process to select hardware resources.
- To the best of our knowledge, this is the first work that uses GNN for mapping on spatial accelerators. The GNN provides a flexible mechanism to automatically collect holistic graph attributes corresponding to the DFG. However, the original DFG has only a few attributes such as operation type and data dependency. To bridge this gap, we generate richer attributes representing the DFG structure information and train the GNN to derive the labels from these attributes effectively.
- We demonstrate that LISA can achieve substantial improvement in mapping quality and compilation time compared to Integer Linear Programming (ILP) and vanilla simulated annealing approach on multiple diverse spatial accelerators and applications.

Paper Organization: In Section II, we introduce the background of spatial accelerators mapping, the impact of DFG structure on mapping, and a primer on GNN. Then we introduce the design of labels and label-aware mapping in Section III. Section IV describes the use of GNN to automatically derive the labels from DFG, and Section V explains the generation of training data for the GNN model. We evaluate LISA in Section VI and discuss related work in Section VII.

II. Background and Motivation

We first introduce the background on spatial accelerators. Then we show the impact of the DFG structure on mapping that motivates this work. Next, we present the background on GNN and discuss the use of GNN in DFG analysis. Lastly, we discuss the challenges of creating GNN models that can automatically mine information from a DFG and create appropriate labels corresponding to an accelerator.

A. Spatial Accelerators

There exist many spatial accelerators for different purposes [4]–[17]. They generally provide configurability for the PE or the network-on-chip. Overall, the spatial accelerators differ in the network-on-chip, PE functions, PE heterogeneity among others.

One representative example of spatial accelerators is Coarse-Grained Reconfigurable Architecture (CGRA) [8]–[17]. CGRA allows configuration of the network and PE functionality on a per-cycle basis. Fig. 1 shows a 4×4 CGRA in a 2D mesh architecture where each PE can communicate with the neighboring PEs. Each PE contains a Register File (RF), ALU, network switch, and configuration memory. The configuration memory contains the instructions for the ALU and the configuration bits for the network switch and ports.

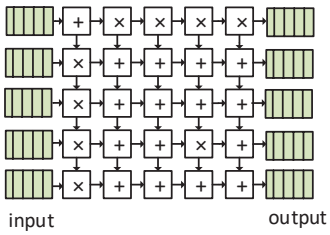


Figure 3: Spatial accelerator example: Systolic Array

The systolic array is another representative spatial accelerator architecture [36]–[40]. Fig. 3 shows a systolic array where the left-most PEs load the input data and the right-most PEs store the output. The configuration of the computing unit is similar to the basic unit of Revel [40]. The PEs can execute either multiply or add operations. Each PE in the systolic array executes a fixed operation throughout the execution process.

B. Mapping and DFG Structure Characteristic

The compilers for spatial accelerators accept as input compute-intensive loop kernels from the application. The compiler then generates the dataflow graph (DFG) corresponding to the loop kernel. Fig. 4 shows a DFG example

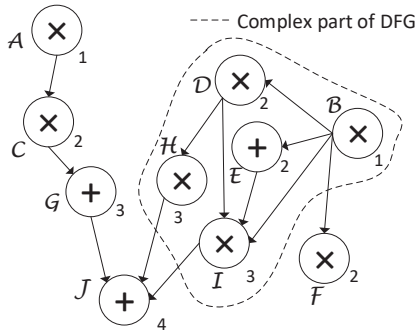


Figure 4: An example dataflow graph (DFG)

corresponding to the loop body of a kernel. A DFG node corresponds to an operation, and an edge corresponds to a data dependency between the operations. The DFG nodes within the dotted area have more complex data dependencies than the other nodes. We use As Soon As Possible (ASAP) value as the scheduling order. The numbers to the right of the nodes represent the scheduling order. The smaller the number, the earlier the node is scheduled (i.e., placed on a PE). If two nodes have the same ASAP value, we first schedule the node on the left side of Fig. 4 (we can get such order using topological order or graph drawing algorithm).

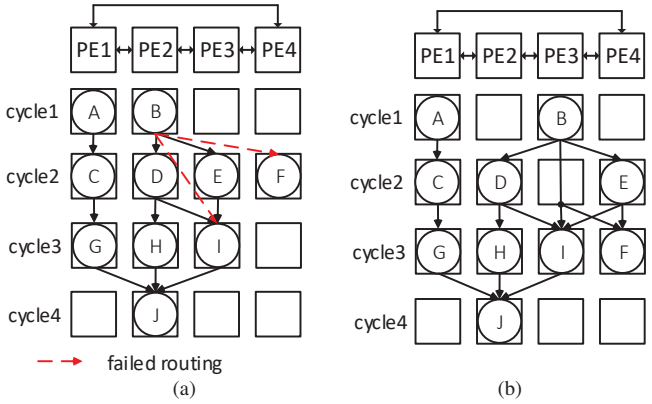


Figure 5: Mapping examples for a time-extended 2x2 CGRA. (a) A failed mapping without any DFG structure knowledge. (b) A successful mapping with DFG structure knowledge.

Fig. 5 provides two mapping examples of the DFG on a 2×2 CGRA. For our illustrative example, let us assume that any DFG node can be mapped to any PE of the CGRAs. Both examples use spatio-temporal mapping and are presented in the form of a time-extended CGRA where the CGRA resources have been replicated along the time dimension. Each PE can do either compute or routing per cycle. In Fig. 5a, we schedule the nodes A and B first. As there is no dependence constraint yet, we can place *A* and *B* in any PE in cycle 1. *C* is placed on PE1 in cycle 2 because of the minimum (zero) routing cost from *A*. Similarly, *D* and *E* are placed on PE2 and PE3 in cycle 2. After placing *D* and *E*, however, we cannot place node *F* as we cannot route data from *B* to *F*. If *F* is placed on PE4 in cycle 2 by ignoring this routing failure, we will continue by placing *G* and *H* on PE1 and PE2 in cycle 3. Then *I* cannot be placed as we cannot route data from *B* to *I*. In this mapping example, the compiler does not have a global view of the complex data dependencies and cannot create a valid mapping. Any algorithm without the whole DFG structure knowledge will end up generating such mappings.

In contrast, Fig. 5b shows a successful mapping that understands the DFG structure and places the nodes with a global view. After placing *A*, we know *B* is the parent of four nodes and has far more connections than the other nodes. Hence we should place *B* in a PE that has sufficient routing resources, e.g., PE3. We also know *D* and *E* are children of *B* and should map them onto PEs without blocking all the routing options from *B*. In this way, we get a successful mapping with the knowledge of the whole DFG structure.

This motivating example demonstrates the need for a global perspective during mapping. Indeed the impact of the DFG structure on mapping extends far beyond this simple example. If there is an edge from *A* to *J* in the DFG of Fig. 4, this edge is more difficult to route than the other edges as it needs a temporal connection with a length equal to the length of $A \rightarrow C \rightarrow G \rightarrow J$. We need to prioritize such edges in routing. Therefore, we need an all-encompassing view to analyze the impact of DFG structure on an accelerator.

Graph Neural Network [35], a neural network on graphs, has become popular in recent years. A GNN can aggregate information from far-flung nodes and edges, and generate collective information for nodes and edges. A graph is represented as $G = (V, E)$, where V and E represent node and edge set, respectively. We have $n \times n$ adjacency matrix, which represents the edge information for a graph with n nodes. Let $X^v \in R^{n \times d}$ be the node attribute matrix whose *i-th* row represents the attribute information of the *i-th* node in a d-dimension vector. Meanwhile, let X^e represent edge attributes, where $X^e \in R^{m \times c}$ is an edge attributes matrix whose *j-th* row represents the attribute information of the *j-th* edge in a c-dimension vector.

GNN provides flexible ways to traverse a graph and propagate information. GNN has three main analytical tasks: node-level, edge-level, and graph-level. We introduce the first two tasks here. Node-level analysis aggregates attribute from the neighboring nodes and generates new attributes on the nodes. Then it can aggregate new attributes from the neighbors and process them again in an iterative fashion. The edge-level GNN model processes information from connected vertices to predict edge information, using similar operations as the node-level model. A GNN model is defined by the attributes and process functions. The final graph information are the labels for nodes and edges.

C. Graph Neural Network (GNN)

Fig. 6 shows an example of a node-level GNN. The matrix below the DFG is the node attributes matrix. This GNN contains two Graph Convolution layers and two ReLU layers. It is worth noting that graph convolution is different

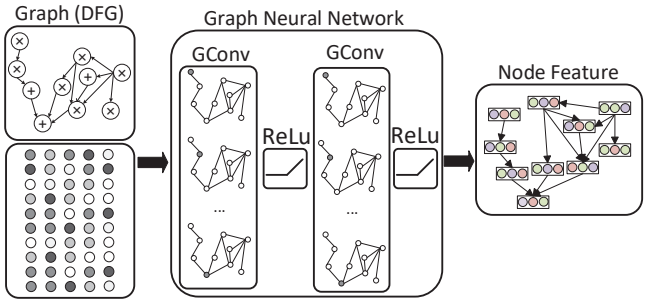


Figure 6: Node-level GNN example

from the traditional convolution operations in Convolutional Neural Network (CNN) [41]. The GNN uses weights (usually in vector and matrix) to directly multiply the graph attributes. The GNN uses pooling functions such as min and max to aggregate neighboring node attributes, and then uses convolutions to process the aggregated information, and updates the attributes with ReLu. Edge-level GNN also performs similar operations on the edge attributes. Hence, GNN has the potential to analyze dataflow graphs and generate labels on DFG nodes and edges.

D. Challenges in using GNN for DFG analysis

So far, we have discussed the impact of the DFG structure on mapping and the potential of using GNN to analyze DFG and generate labels. However, it is challenging to realize this potential. First, we need to design proper labels to reflect the DFG structure and accelerator characteristics, and need to use these labels to guide the placement and routing choices. Meanwhile, we need to ensure that the GNN and the mapping process are consistent, i.e., the GNN can generate the labels, and the labels can be used for mapping.

Another challenge is that current GNN models cannot be used directly to derive the labels for the DFG. One reason is that the existing GNN models usually rely on sufficient number of graph attributes to generate the desired information, such as (age, gender, location, etc.) in a social network graph, while our DFG only has a few straightforward attributes such as operation type and data dependency. Another reason is that the impact of the DFG structure on mapping is not straightforward; the placement and routing of one node is interlinked with the placement and routing of other nodes, including nodes without direct dependency. We need to design effective GNN models to find the connection between DFG structure characteristics and mapping impact. The following sections elaborate on how we overcome these challenges.

III. DFG Label Design and Mapping

We now introduce the label types and how the labels are used in the mapping. Labels describe how nodes and edges should be mapped onto the accelerator. We design a label-aware compiler to decide placement and routing and generate quality mapping on spatial accelerators.

Table I: Types and purposes of labels

label id	label name	placement	routing
1	schedule order	✔	✘
2	same-level nodes association	✔	✘
3	spatial mapping distance	✔	✔
4	temporal mapping distance	✘	✔

A. Label

Table I summarizes the four labels guiding the compiler to make proper placement and routing decisions.

- **Schedule order** represents the execution order of the DFG nodes. It captures more information than topological or similar orders. For example, a node with more children should be scheduled earlier to route to multiple destinations, while topological order is decided by node dependency only. Our schedule order captures the holistic DFG structure information and enables such nodes to be scheduled earlier.

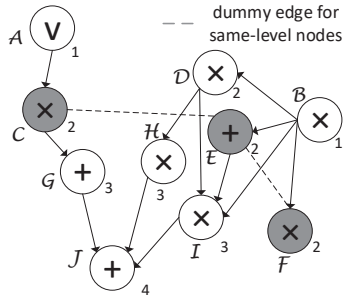


Figure 7: Example of same-level nodes association

- **Same-level nodes association** defines the spatial distance between the same level nodes that do not have data dependency. If DFG nodes with the same ASAP value have the same ancestor or descendant node, they are defined as same-level nodes. As these nodes are typically scheduled closer in time, this label represents how far away the same-level nodes can be placed. Fig. 7 provides an example for the DFG in Fig. 4. We show the association for the same-level nodes *C*, *E*, and *F* with dummy edges. Note that there is no dummy edge between *C* and *F* as they do not have any common ancestor or descendant.
- **Spatial mapping distance** indicates the spatial distance of a DFG edge in a mapping. We cannot always place two dependent nodes closely, as other children of the parent and other parents of the child also influence the placement choice. We need a label to describe the spatial mapping distance of a DFG edge to figure out how the DFG structure characteristics affect the placement of parent and child nodes. Moreover, both the same-level node association and spatial mapping distance describe the spatial distance between two nodes; the former between non-dependent nodes, while the latter between dependent nodes.

Algorithm 1: Label-aware Simulated Annealing

Input: DFG, labels of DFG, spatial accelerator;
Output: Mapping

```
1  while not find a valid mapping or not exceed time limitation do
2      unmap one or more DFG nodes;
3      sort unmapped DFG node by schedule order (label 1);
4      foreach unmapped DFG node do
5          foreach PE candidate do
6              calculate placement cost according to label 2, 3, 4;
7          calculate deviation σ = max{1, α × T − Acc};
8          use normal distribution to decide the cost to select PE;
9      sort unrouted data by routing priority using label 4;
10     foreach unrouted data do
11         Route using Dijkstra's algorithm ;
```

- **Temporal mapping distance** indicates the temporal distance of a DFG edge in a mapping. Child node cannot always be executed immediately after one parent node as this child may have other parents. We estimate the routing resources required by the edge using temporal distance.

The last three labels describe the spatial and temporal distance between nodes. The way to calculate the spatial distance depends on the accelerator. For the 2D mesh accelerators in Fig. 1, we can use Manhattan distance. The temporal distance is calculated by the distance along the temporal dimension (number of cycles) in the mapping.

Let us use two examples to demonstrate how to use labels to guide placement and routing. Each placement or routing implies the use of resources. Obviously, mapping the first few nodes is always easier than mapping the last few nodes, as the latter has fewer resources. We can use temporal mapping distance to find the edges that have long temporal distances and need more routing resources. We then route these "long" edges first to avoid routing them with insufficient resources.

Another example is to use same-level nodes association to guide placement. Though such nodes do not have any direct dependency, they have connections due to the same ancestor or descendant. If they have the same child, they should be placed closer to ensure the data can be routed easily. If there are many nodes between the same-level nodes and their common descendant/ancestor, then the same-level nodes can be placed in two distant PEs.

B. Label-aware Mapping

Compilers can use combinations of labels to decide placement and routing. Note that the compiler can perform placement and routing in parallel or place the DFG nodes first and then route the data. Both types of compilers can use the labels. The compilers have the flexibility to customize the label combinations to decide placement and routing. The labels also have the generality for accelerator mapping as all the spatial accelerator mapping can use the information (albeit different values for different accelerators). As these labels are associated with the nodes and the edges, we can employ GNN to derive the labels.

A variety of mapping methods can use these labels. We demonstrate the power of the labels in simulated annealing (SA), a popular method for spatial accelerator mapping [42] [43] [26] [22]. SA-based approaches usually create an initial random mapping that may oversubscribe the resources. SA movement uses a cost function to estimate the quality of the current mapping. It then attempts to remove some nodes from the current placement and remove related routing. This process is referred to as *unmap*. Simulated annealing places these unmapped nodes and routes them again. The unmap and remap processes are called movements in SA. If the movement creates a better mapping, it is accepted. Even if the new mapping after movement is worse quality according to the cost function, SA can still accept it with a small probability to overcome the local minima.

Algorithm 1 presents our proposed label-aware simulated annealing (SA) approach. We place the unmapped nodes first and then route data. This algorithm generates the initial mapping in the first iteration as all the nodes are unmapped at first. If all the nodes are mapped, the algorithm randomly selects some nodes to unmap (Line 2). The schedule order label decides the placement order (Line 3-4). The other three labels decide routing priority and placement choice. When selecting the PE candidate to place a node, we use a cost model that combines three labels (Line 6). The cost for each PE candidate is the sum of the differences between the actual mapping distance and the labels' expected distance. As a minor difference in placement can lead to a completely different mapping, the PE candidate with the minimal cost might not be the best one. Hence we use a normal distribution to select the candidates based on costs (Line 7-8), ensuring the candidate with lower cost has higher chance of getting selected. The deviation parameter σ of normal distribution decides the variance of the distribution. We use the function $\sigma = max\{1, \alpha \times T - Acc\}$ to decide the deviation parameter, where T represents the number of attempted movements, Acc represents the number of accepted movements, and α is the customized factor. When the accept rate Acc is low, this function will have a bigger deviation parameter σ, randomly selecting PE candidates to eliminate the current invalid mapping. To decide the routing order of the node, we use the sum of temporal mapping distance label, which represents the routing resource that a DFG node needs (Line 9). Note that the nodes that need more routing resources are routed first. The label-aware approach only changes the placement choice and routing priority compared to the original SA.

IV. GNN Model for DFG Label Generation

The key challenges in using GNN to derive the labels are (a) providing effective graph attributes for the GNN model, and (b) using these attributes to derive the labels to describe how DFGs should be mapped. However, the connection between DFG structure and mapping impact is not easy to

capture; thus, label values are difficult to calculate. Hence deriving labels using GNN is different from traditional GNN analysis. We need insights on how the DFG structure impacts the mapping. We next describe an Attributes Generator to represent the DFG structure characteristics with graph attributes, and introduce how we use GNN models to derive the labels from the attributes.

A. Attributes Generator

To derive the labels from DFG, we first need to provide the graph attributes to the GNN. Deriving one label requires multiple attributes. Traditional GNN analyses [44] on graphs (e.g., social network) usually have natural attributes to analyze. However, the DFGs do not have such attributes, as nodes usually only have operation type attribute.

Our approach is to design an Attributes Generator on DFG to generate the relevant attributes. This Attributes Generator uses traditional graph algorithms to represent the DFG structure characteristics in terms of graph attributes. These graph attributes affect placement and routing decisions. For example, a DFG node with many child nodes needs enough routing resources. Hence, the node needs a higher schedule order. Meanwhile, to leave enough space for routing dependencies to child nodes, the node should not be placed too close to its parent nodes. We intuitively select graph attributes which we think can affect placement and routing decisions. We generate the following node attributes: (1) ASAP, (2) in-degree, (3) out-degree, (4) the number of ancestor nodes, (5) the number of descendant nodes, and (6) operation type. We have the following edge attributes: (1) ASAP value difference between child and parent node, (2) the number of nodes between two nodes; a node is between two nodes if its ASAP value is in between the two nodes' ASAP values, (3) the number of nodes with the same ASAP value as the parent or the child, (4) the number of ancestors of the parent node, and (5) the number of descendants of the child node.

Same-level nodes forms the dummy edges in the DFG. Thus, we also need to provide attributes for these dummy edges so that the edge-level GNN model can predict same-level nodes association labels. We generate the following dummy edge attributes: (1) the distance from same-level nodes to the closest common ancestor node and (2) the distance to the closet common descendant node, (3) the number of nodes whose ASAP value is bigger than the ancestor node and smaller than the same-level nodes, (4) the number of nodes whose ASAP value is smaller than the descendant node and bigger than the same-level nodes, (5) the number of nodes whose ASAP value equals the ASAP value of ancestor node, descendant node, or same-level nodes, (6) the number of nodes on the path from the same-level nodes to the ancestor node, and (7) the number of nodes on the path from the same-level nodes to the descendant node.

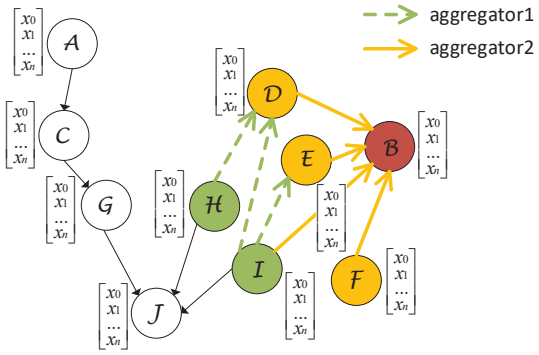


Figure 8: GNN label derivation example.

B. GNN Model

We design a GNN for each label. The GNN model parameters can be inferred from the equations in each network. Deriving one label does not need all the attributes, and we intuitively select the relevant attributes for each GNN.

Schedule order: This label is relevant to the complexity of data dependency and the scheduling order of neighbors (parents and children). For example, a node with many children needs to be scheduled earlier to route multiple dependencies. Subsequently, the node's parent(s) should also be scheduled earlier. Hence, after processing the attributes of the current node, we need to propagate the changes to its neighbors. We design a network consisting of four layers, and all the layers use the following two equations:

$$\mathbf{m}_v^{(t+1)} = \mathbf{W_1}\left[\mathbf{mean}(\mathbf{m}_u^{(t)}), \mathbf{max}(\mathbf{m}_u^{(t)}), \mathbf{min}(\mathbf{m}_u^{(t)})\right] \quad (1)$$

$$\mathbf{h}_v^{(t+1)} = \mathbf{W_2}\left(\mathbf{W_3}\mathbf{h}_v^{(t)} + \mathbf{m}_v^{(t+1)}\right) \quad (2)$$

In each layer, the network aggregates the changes of the neighbors and updates the scheduling order. Let us use $\mathcal{N}(v)$ to represent the neighbors of a node v, t to represent the layer index, m to represent the schedule order changes of a node, and h to represent the node attributes including the input attributes and schedule order label. Equation 1 aggregates the change of neighbors $\forall u \in \mathcal{N}(v)$ through a combination of three pooling functions: min, max, and mean. Then the aggregated changes are processed by a weight matrix. Next, Equation 2 updates the scheduling order of each node by combining neighbors changes. In the first layer, the schedule order $\mathbf{h}_v^{(t)}$ is the ASAP value and $\mathbf{m}_v^{(t+1)}$ is calculated by $W_1 \times Attributes(v)$, where $Attributes(v)$ is all the attributes of node v generated by the Attributes Generator.

Fig. 8 illustrates how the information is propagated from node *H* and *I* to *B*. First, *H* and *I* update their attributes through a graph convolution on their attributes $W_1 \times Attributes(v)$, where attributes are represented by a vector. The attributes describe the degree of a node, the number of ancestor and descendant nodes, and the operation type etc. Through these attributes, we know the communication density of the node and thus if it needs a higher schedule

priority. Then *D* and *E* aggregate $(min, max, mean)$ neighbors schedule order changes from *H* and *I*, and update their schedule orders (*D* also aggregates the change of its neighbor *B*, but we do not show this for the sake of convenience). Through aggregation, *D* and *E* know whether children nodes need a higher schedule priority or not. Finally, *B* aggregates schedule order changes from *D*, *E*, *I*, and *F*, and update its own schedule order.

Same-level nodes association: This label describes the spatial distance between two same-level nodes. Same-level nodes do not have any direct dependency but have a common ancestor/descendant node. The spatial mapping distance between two same-level nodes is affected by their shortest distance to their common ancestor/descendant. Meanwhile, the number of nodes between them and the common ancestor/descendant also affects the distance, as more intermediate nodes need more PEs to schedule. The Attributes Generator traverses the graph to collect these information as edge attributes. We predict the feature of dummy edges between same-level nodes through the following network:

$$\mathbf{h}_e = \mathbf{MLP}\left(edge\_attributes(e)\right) \tag{3}$$

We use *multilayer perceptron* (MLP), consisting of two convolution layers and one activation layer, to process the edge attributes. The MLP . Hidden channels represent the output channels of the first layer and the input channels of the second layer. We set the number of hidden channels equal to the number of edge attributes. We use ReLu as the activation layer.

Spatial mapping distance: This label describes the spatial mapping distance of an edge. The distance can be affected by other dependent nodes. For example, if the source node of an edge has many neighbors, it needs more routing resources because the neighbors need to placed on "distant" PEs to leave enough space for routing.

$$\mathbf{h}_e^1 = \mathbf{W_1}\left(edge\_attributes(e)\right) \tag{4}$$

$$\nu = \left[\frac{1}{\mathbf{mean}(\mathbf{e}(v))}, \frac{1}{\mathbf{sum}(\mathbf{e}(v))} \frac{1}{\mathbf{max}(\mathbf{e}(v))}, \frac{1}{\mathbf{min}(\mathbf{e}(v))}\right] \tag{5}$$

$$\mathbf{h}_e^2 = \mathbf{W_2}\mathbf{h}_e^1 + \nu \cdot \mathbf{W_3}\mathbf{h}_e^1 \tag{6}$$

First, as shown in Equation 4, we use a convolution layer to process the edge attributes from the Attributes Generator. Similar to same-level nodes association, these attributes reflect the DFG structure complexity around parent and child nodes. We get an initial spatial distance for each edge through the convolution layer on these attributes. We use $e(v)$ to represent the attributes of connected edges of parent and child nodes. Next Equation 5 uses four aggregators to generate a normalization vector ν. If the value of a denominator is zero, the corresponding normalization factor is set to one. As mentioned earlier, the sibling nodes of parent and child nodes can also affect the distance. Hence we use a combination of multiple normalization factors to aggregate, which collects comprehensive information from neighboring nodes. Finally, we generate the label value with Equation 6, the sum of weighted and normalized features.

Temporal mapping distance: This label is relevant to the DFG structure complexity around the edge. For example, for the temporal distance of edge (B, I) in Fig. 8, *E* is the child of *B* and parent of *I*; thus *E* makes the temporal distance longer. We use all the edge attributes from the Attributes Generator and use MLP to process these edge attributes as shown in Equation 7.

$$\mathbf{h}_e = \mathbf{MLP}\left(edge\_attributes(e)\right) \tag{7}$$

The number of hidden channels is equal to the number of edge attributes. We use ReLu as the activation layer.

V. GNN Training Data Generation

This section describes the generation of training data for GNN models. First, we generate a random set of DFGs and initialize their labels. Then we use an iterative label-aware simulated annealing approach to update the labels. Finally, we use a metric to evaluate and filter these labels.

A. Raw DFG Generation

We need sufficient number of DFGs to generate a training data set. However, current publicly available benchmarks are not enough for training. Hence we generate a set of random DFGs with wide spectrum of structures. We first generate random directed and weakly connected graphs. The number of DFG nodes are set from *n* to *m*, which is based on the real applications. The number of connected edges for each node is also set to a range. This ensures that we generate diverse DFGs and a robust training set. Then according to the supported operations, we randomly assign operations to guarantee the validity of the DFGs.

B. Label Generation

Label quality in the training dataset is vital to build an effective GNN model. To generate labels for training, we first initialize labels for the raw DFGs. Then we use these labels in a label-aware simulated annealing (SA) approach. After getting the mapping, we extract the labels from the mapping result and update the DFG labels. We use updated labels to map again and repeat this process. We do not update the labels if the labels either do not generate a mapping or do not lead to better mapping. In this case, we use previous labels to map again. As SA makes random choices, the mapping result is different for each run and can improve via remapping.

To initialize the labels, we assign a scheduling order with the ASAP value. For same-level nodes association, we use the average value of the shortest distances between nodes and common ancestor/descendant. We initialize the spatial mapping distance as zero and temporal mapping distance as one.

The label-aware SA in training data generation is different from the one in Section III. The latter uses labels to provide guidance throughout the mapping process. The training only uses the labels for initial mapping and does not use labels in the later random mapping movement. Hence, this is a partial label-aware SA.

We extract label values from the mapping result. The DFG execution time can be an arbitrary positive number. Hence, we normalize the execution time to the range from zero to the length of the longest path to get the schedule order. For the other three labels, we calculate the distance according to the mapping distance. As accelerators can have diverse architectures and layouts, the spatial distance depends on the accelerator definition. For the 2D mesh accelerators in Fig. 1, we can use Manhattan distance. The temporal distance is calculated by the distance along the temporal dimension (number of cycles) in the mapping.

From an iterative method, we can get multiple labels for each DFG. We need to evaluate the quality of these labels. As the labels are extracted from mapping, the mapping quality reflects the quality of the label. For the mapping quality, a straightforward metric is the execution time or Initial Interval (II) for CGRA [9] which is the interval between successive iterations of a loop kernel. Besides, we use the routing cost for labels with the same execution time or II, representing how many routing resources are used in the mapping.

We have two rounds to select candidate labels from multiple labels, and we combine the candidate labels to generate the final one. In the first round, we select the labels with the lowest execution time or II value. For the second round, we set the label with the lowest routing cost as the standard candidate, and select labels whose corresponding mappings have similar routing costs as other candidates. In practice, if the routing cost is less than 1.15x of the routing cost of the standard one, the label is a candidate. Then we use the average value of candidate labels (including the standard one) to generate the final label for a DFG.

C. Label Filter

The above iterative method cannot always generate a valid mapping for a DFG or enough label candidates. We need to decide which DFG and its corresponding label can be included in the training set. We use a metric e to filter these labels: $e = O + \sigma \times N$, where O represents how close the execution time of label-corresponding mapping is to the theoretical minimal execution time, N represents the number of candidate labels, and σ is a customized factor. The closer it is to the theoretical performance, the higher is the O value. There are different ways of calculating the theoretical performance for different accelerators. For example, in CGRA, the theoretical lowest execution time is resource Minimal II, calculated as the number of DFG nodes divided by the number of PEs. If there are not sufficient labels for a DFG and O is far from the theoretical optimal performance of the corresponding DFG, the DFG and the corresponding label are not included in the training set. As long as we get the minimum II for a DFG, only one candidate label is sufficient to be used as training data.

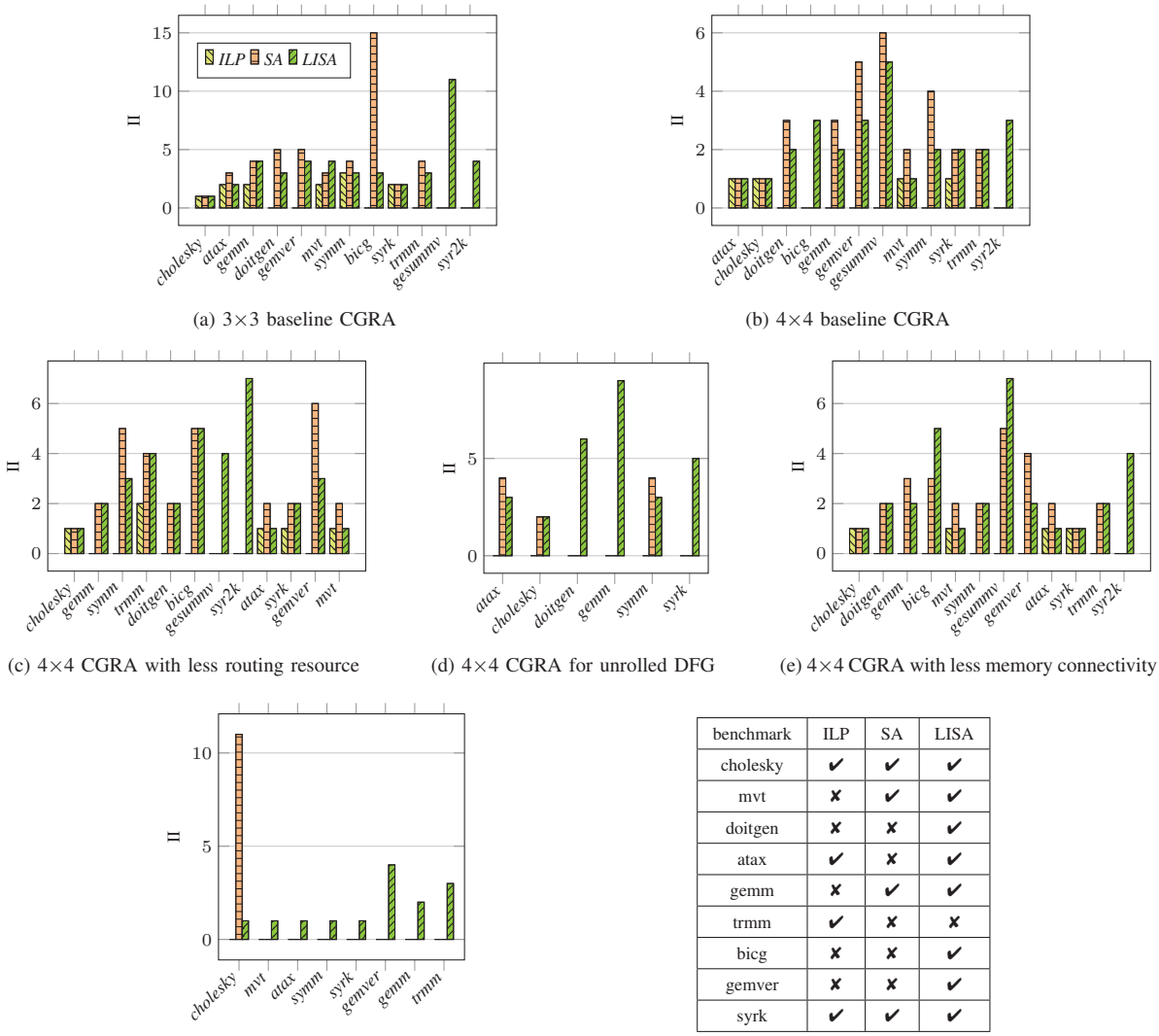
VI. Experimental Evaluation

We evaluate LISA along multiple dimensions: quality of mapping, compilation time, portability, and the effectiveness of different components in our mapper and GNN modeling.

Modelled Spatial Accelerators: As LISA is a portable compiler framework that works across a range of spatial accelerators, we evaluate it on diverse architectures. (1) 4×4 baseline CGRA; (2) 3×3 baseline CGRA; (3) 8×8 baseline CGRA; (4) 4×4 CGRA with less routing resources: Registers are vital for buffering data during routing especially when producer and consumer are far apart along spatial or temporal dimension. The baseline CGRAs have four registers per PE. For the CGRA with less routing resources, we set one register per PE; (5) 4×4 CGRA with less memory connectivity: While the above CGRAs allow all the PEs to access the on-chip memory, we only allow the left-most PEs to access on-chip memory in this version; (6) 5×5 systolic accelerator (Fig. 3) with computing unit similar to Revel basic unit [40].

We implement the CGRA architectures in Verilog HDL and synthesize on a 22 nm process using Synopsys Design Compiler to obtain the power consumption. We set the frequency to 100 MHz similar to other low-power spatial accelerators [45]. For CGRA, the schedule of a loop kernel is repeated after Initiation Interval (II) cycles. The lower the II, the higher the performance. Each PE has 24 configuration entries in all the architectures except the systolic array, which means the maximum possible II is 24. CGRA execution cycles are entirely deterministic as it follows a compiler-generated static schedule and has software-controller scratchpad memory. Hence, execution time is calculated by the II value of the mapping and the clock frequency.

CGRA-ME state-of-the-art compiler: To compare the mapping quality, compilation time, and portability with state-of-the-art compilers, we use CGRA-ME [26], a popular open-source CGRA compilation framework. We select CGRA-ME because it allows flexible modeling of different spatial accelerators. LISA is compared with two mapping approaches from CGRA-ME: Integer Linear Programming (ILP) and Simulated Annealing (SA). SA is the most popular algorithm in modern spatial accelerator compilation frameworks (CGRA-ME [26], DSAGEN [22], AURORA [23]). As SA makes random choices, the mapping is different for each run and is not stable. Thus we run SA three times for each architecture and benchmark combination and use the median performance. The original routing algorithm in CGRA-ME cannot always find the shortest path; we modify



(a) 3×3 baseline CGRA

(b) 4×4 baseline CGRA

(c) 4×4 CGRA with less routing resource

(d) 4×4 CGRA for unrolled DFG

(e) 4×4 CGRA with less memory connectivity

(f) 8×8 CGRA for unrolled DFG

benchmark	ILP	SA	LISA
cholesky	✔	✔	✔
mvt	✘	✔	✔
doitgen	✘	✘	✔
atax	✔	✘	✔
gemm	✘	✔	✔
trmm	✔	✘	✘
bicg	✘	✘	✔
gemver	✘	✘	✔
syrk	✔	✔	✔

(g) Systolic Accelerator

Figure 9: Performance comparison of LISA with ILP and SA on multiple CGRAs and one systolic accelerator. For CGRAs, the lower the II, the better the performance. If II is zero, it implies that the benchmark cannot be mapped by the corresponding method. For the systolic accelerator, ✔ implies that the corresponding method can map the benchmark and ✘ means the method cannot map the benchmark.

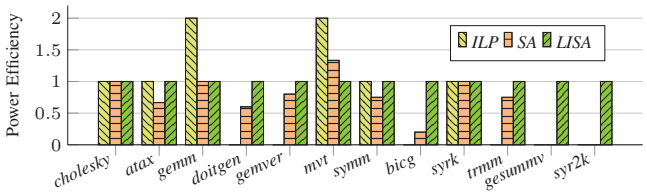
to use Dijkstra's algorithm to find the shortest path. We implement the GNN models with PyTorch Geometric [46].

Benchmarks: We use the PolyBench benchmark suite [47] as well as an unrolled version (unrolling factor is 2) of kernels. Some of the benchmarks in PolyBench are not supported by CGRA-ME.

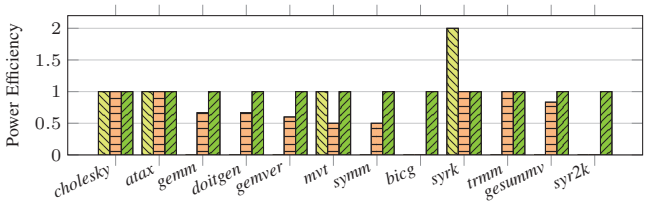
A. *Quality of the Mapping*

Performance: Fig. 9 shows the quality of the mapping with II values (lower the II value, the better the performance) for the different architectures and benchmarks. We use 12 DFGs supported by CGRA-ME from PolyBench [47], excluding combinations that cannot be mapped by any method. In total, Fig. 9 has 71 combinations of benchmarks and accelerators. LISA can map almost all the combinations except for *trmm* on the systolic accelerator. But ILP and SA can only map 23 and 49 combinations out of 71. Hence, LISA can map 48 combinations that ILP cannot map and 21 combinations that SA cannot map.

LISA can also achieve obvious improvement in II value over SA for the mapped benchmarks. For 71 accelerator-application combinations, ILP and SA can generate better mappings than LISA for only 6 and 3 combinations. As LISA uses labels to guide mapping, it cannot always find the optimal mappings. ILP can find optimal solutions for some combinations but cannot map 48 combinations even with a generous time limit (two hours for each target II). The random movements of SA may sometimes find better solutions (3 out of 71 combinations). The main reasons for LISA's success are: 1) LISA maps the DFG with an all-encompassing global view, which makes it easier to handle

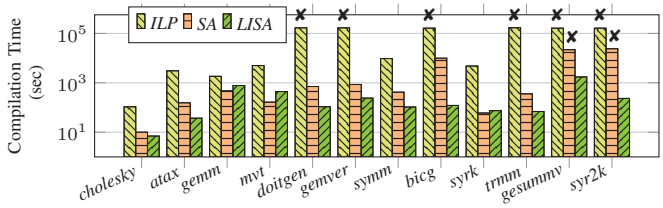


(a) 3×3 baseline CGRA

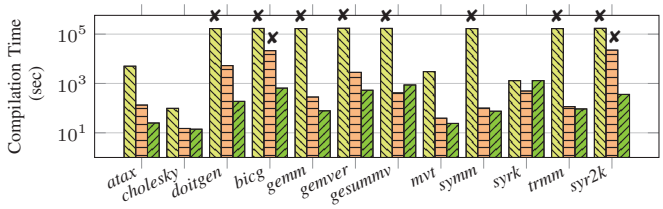


(b) 4×4 baseline CGRA

Figure 10: Power efficiency comparison on 3×3 and 4×4 baseline CGRAs, where value 0 indicates that the DFG cannot be mapped by the corresponding method.



(a) 3×3 baseline CGRA



(b) 4×4 baseline CGRA

Figure 11: Compilation time comparison on 3×3 CGRA and 4×4 baseline CGRA. ✘ means the corresponding method cannot map the benchmark and we use termination time as the compilation time.

complex DFGs without getting stuck in local minima, and 2) LISA uses GNN to generate labels for varied accelerators and is better aware of accelerator characteristics and resources.

LISA clearly demonstrates the ability to handle complex DFGs and varied accelerators well. For example, in Fig. 9b and 9c, SA cannot map two benchmarks (*bicg*, *syr2k*) for 4×4 CGRA, and two benchmarks (*gesummv*, *syr2k*) for a CGRA with less routing resources. But LISA can map all the benchmarks on both architectures. Interestingly, SA can map *bicg* on the CGRA with less routing resources but cannot map on baseline 4×4 CGRA. The former CGRA has less routing resources and hence a smaller search space, making it easier for SA to navigate. Fig. 9d shows the II comparison for unrolled DFGs (unrolling factor 2) on 4×4 CGRA. Unrolling increases the complexity of the DFG. LISA can achieve superior performance on unrolled DFGs than ILP and SA. Among the six benchmarks, ILP cannot map any, SA can only map three, while LISA can map all. Out of three benchmarks mapped by both LISA and SA, LISA still outperforms on *atax* and *symm*.

LISA is also scalable. In Fig. 9f, for 8×8 CGRA and 8 unrolled DFGs, LISA can map all of them and the performance scales compared to 4×4 CGRA, while ILP cannot map any DFG, and SA maps only one. With more hardware resources and relatively complex DFGs, ILP requires more variables, constraints, and cannot scale. SA has difficulty in finding valid solutions using random movement. Benefiting from the global view, LISA can construct valid and quality mappings.

Power Efficiency: Fig. 10 shows the power efficiency comparison on 3×3 and 4×4 baseline CGRAs. LISA can map all the benchmarks for both CGRAs, while ILP and SA cannot map 14 and 4 combinations, respectively. We normalize performance per Watt (MOPS/W) values w.r.t. LISA. As LISA maps most benchmarks with lower II, it can achieve an average of 1.58x and 1.4x power efficiency compared to SA on 3×3 and 4×4 CGRA with mapped DFGs. Compared to LISA, ILP achieves better power efficiency for *gemm* and *mvt* on 3×3 CGRA and *syrk* on 4×4 CGRA. However, ILP cannot map more than half of the combinations, making it a unrealistic choice.

Compilation Time: Fig. 11 shows the compilation time for the three methods on 3×3 and 4×4 baseline CGRA. We use the termination time as the compilation time for the benchmarks that ILP and SA cannot map. On 3×3 CGRA, LISA achieves 594x and 17x compilation time reduction compared to ILP and SA, respectively. On 4×4 CGRA, LISA achieves 724x and 12x compilation time reduction compared to ILP and SA, respectively. This is because 1) The compiler starts with target II equal to MII and increments by one if it cannot map, until the target II exceeds the maximum II. LISA achieves feasible mapping at lower II and thus can complete early, and 2) LISA uses labels to map with a global view, steering the search in the right direction.

In summary, for almost all application, accelerator combinations, LISA substantially outperforms ILP, SA in mapping quality (performance, power efficiency) and compilation time. Benefiting from an all-encompassing global view, LISA scales with spatial accelerators, complex DFGs and handles limited routing resources. As the GNN model can generate adaptive label values for various accelerators via retraining, LISA generates quality mapping on different architectures, creating a truly portable compiler framework for spatial accelerators.

B. GNN Model Accuracy

For each accelerator, we randomly generate 1,000 DFGs and use the iterative mapping method to generate training data with labels. As each DFG has tens of nodes and edges, the training data is sufficient for our GNN models. GNN

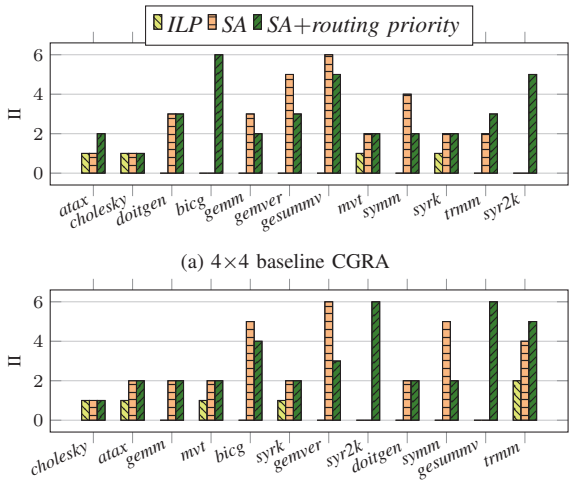
Table II: GNN label prediction accuracy.

Spatial accelerator architecture	Prediction accuracy			
	label1	label2	label3	label4
4×4 baseline	0.788	0.856	0.932	0.992
3×3 baseline	0.648	0.939	0.992	0.938
4×4 with less routing resource	0.758	0.885	0.951	0.977
4×4 with less memory connectivity	0.738	0.852	0.941	0.988
8×8 baseline	0.685	0.716	0.914	0.990
systolic accelerator	0.759	0.768	0.907	1.000

model parameters can be directly inferred from the equations in Section IV-B. We use a learning rate of 0.001, a weight decay of 0.0005, and train for 500 epochs.

Different from existing node classification GNNs, the GNN models in LISA aggregate values. The label values are arbitrary positive numbers and hard to predict. Fortunately, the labels guide the mapping process, and label-aware SA can handle some inaccuracy in the label values. The prediction is accurate for same-level nodes association and spatial mapping distance if the difference between prediction and ground truth is not more than one. For temporal mapping distance, the prediction is accurate if the difference is not more than two, as the temporal mapping distance can be any positive number. For scheduler order, the prediction is accurate if prediction and ground truth values are the same.

Table II shows the high GNN model accuracy for different spatial accelerators and label IDs from Table I, which has label schedule order, same-level nodes association, spatial mapping distance, and temporal mapping distance. The scheduler order (label 1) has relatively low accuracy compared to the other labels because the scheduling order of one node is affected by other nodes, which is hard to estimate accurately. Compared to 4×4 CGRA, the GNN model on 3×3 CGRA has higher accuracy for label 2, 3, and lower accuracy for label 1, 4. As labels 2, 3 describe spatial distance, a smaller accelerator makes the prediction easier for the GNN model. A smaller CGRA needs more cycles (temporal length) to map the same DFG compared to a larger CGRA with more resources. Also, on a smaller accelerator, a slight difference in the DFG can cause a completely different mapping as one node's placement can be easily affected by other nodes with limited resources. Hence the labels related to temporal distance (1, 4) have lower accuracy in smaller CGRA. 8×8 CGRA has lower accuracy for the first three labels compared to 4×4 CGRA. For the schedule order label, we normalize the execution cycle to the length of the longest path in the DFG to get the scheduling order; the execution cycles on 8×8 CGRA is much less than 4×4 CGRA. Thus it is hard to predict schedule order on 8×8 CGRA. As the other two labels are related to spatial distance, corresponding GNN models have relatively lower accuracy on 8×8 CGRA. Nevertheless, these labels are accurate enough to let LISA achieve relatively lower II on with 8×8 CGRA and make it scalable.



(a) 4×4 baseline CGRA

(b) 4×4 CGRA with less routing resource

Figure 12: Effectiveness evaluation for temporal mapping distance (label 4) on 4×4 CGRA. The routing priority is decided by temporal mapping distance (label 4).

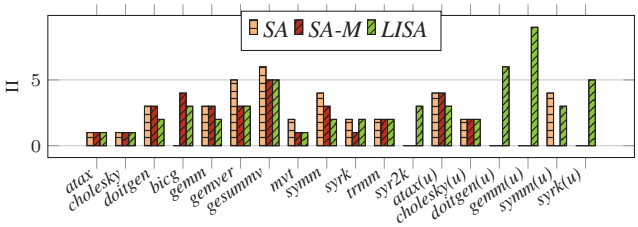


Figure 13: Performance evaluation on 4×4 CGRA for DFGs and unrolled versions, where *(u)* represents unrolled DFGs. SA-M has 10× random movements at each temperature compared to SA.

C. Effectiveness of Label-aware Mapping

LISA uses multiple labels for mapping and temporal mapping distance (label 4) to decide routing priority (Algorithm 1). To confirm the power of individual labels in assisting the mapping process, we add routing priority to the original SA in Fig. 12. SA with routing priority can map four benchmark, accelerator combinations where the original SA failed: *bicg*, *syr2k* for baseline CGRA, and *gesummv*, *syr2k* for CGRA with less routing resources. Besides, SA with routing priority achieves better II than the original SA on seven combinations. Still, SA with routing priority cannot match the performance of LISA, which uses multiple labels with a global perspective.

To further evaluate LISA's efficacy, we maintain 50 movements at each temperature for SA and LISA, but create a new version of SA with 10× movements, called SA-M. In Fig.13, SA-M maps one more DFG (*bicg*) and achieves better II on four DFGs compared to SA. LISA still achieves lower II for four original DFGs and maps one more DFG (*syr2k*) compared to SA-M, though SA-M has lower II on *syrk*. For unrolled DFGs, SA-M does not provide any benefit; it cannot

map unrolled *symm* that SA maps. More movements may create lower acceptance rate, resulting in early termination of SA. This is the limitation of random movement compared to LISA's global perspective.

VII. Related Work

We delve into related work from three perspectives: traditional mapping methods, machine learning and mapping, and Graph Neural Network.

Traditional mapping methodology. Many related works apply basic DFG structure information for spatial accelerator mapping. One classic example is the topological order used by many DFG mapping techniques. Park et al. [48] calculate affinity between nodes with common consumers to check whether to place them close or not. Similarly, Park et al. [49] clusters low out-degree nodes to consume data as soon as possible. However, the DFG information in these existing works is only a part of the graph, which does not comprehensively analyze the DFG. LISA emphasizes global DFG analysis and the mapping impact of these DFG characteristics. Moreover, to use the above DFG information for different accelerators, these existing works need to set the parameters in mapping manually. By contrast, LISA can automatically derive the impact of DFG structure on mapping, generating high-quality mapping for diverse accelerators.

Machine Learning and Mapping: Using machine learning to map for spatial accelerator has been explored. Park et al. [50] use Reinforcement Learning to map on CGRA. This method can provide more valid solutions for relatively big DFGs. However, [50] only supports DFGs where the number of nodes is less than or equal to the number of PEs. Cummins et al. [51] design a tool, DeepTune, to optimize heuristics in the schedule. Through neural networks, DeepTune can automatically find the proper execution model on heterogeneous devices. We are the first to use GNN for spatial accelerator mapping.

Graph Neural Network: GNN has been used in many domains to analyze graphs [52] including analysis of DFGs. To provide a transferable representation of computing graph, Google designs an inductive graph embedding model that encodes operation features and dependencies [53]. [54] uses GNN to analyze the vulnerability of DFGs, while [55] uses GNN to predict operation delay latency in HLS. Though [55] achieves high accuracy, it only supports micro-benchmarks. Also, the prediction task is simpler than the label prediction task in LISA that requires a comprehensive DFG analysis.

VIII. Conclusion

Tremendous manual effort is needed to design a quality compiler for a spatial accelerator. We presented a portable compiler framework *LISA* that can automatically provide quality mapping for varied accelerators. We use the abstraction of *labels* to represent the impact of the DFG structure on accelerator mapping and provide a global view for mapping. We employ GNN models to automatically derive the labels for various accelerators. This is the first work to use GNN for spatial accelerator mapping. Experiments show LISA substantially outperforms ILP and Simulated Annealing basd mapping on multiple accelerators.

IX. Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments. We also thank Jun Zeng, Peng Chen, and Kaihang Ji for discussion and feedback on earlier drafts of this paper. This research is partially supported by the National Research Foundation, Singapore under its Competitive Research Programme Award NRF-CRP23-2019-0003.

References

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[2] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.

[3] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, no. 6495, 2020.

[4] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[5] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.

[6] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.

[7] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.

[8] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[9] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[10] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[11] D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra, "Cascade: High throughput data streaming via decoupled access-execute cgra," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–26, 2019.

[12] Y. Zhang, A. Rucker, M. Vilim, R. Prabhakar, W. Hwang, and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 615–628.

[13] B. Wang, M. Karunarathne, A. Kulkarni, T. Mitra, and L.-S. Peh, "Hycube: A 0.9 v 26.4 mops/mw, 290 pj/op, power efficient accelerator for iot applications," in *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2019, pp. 133–136.

[14] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4d-cgra: Introducing branch dimension to spatio-temporal application mapping on cgras," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[15] A. Nayak, K. Zhang, R. Setaluri, A. Carsello, M. Mann, S. Richardson, R. Bahr, P. Hanrahan, M. Horowitz, and P. Raina, "A framework for adding low-overhead, fine-grained power domains to cgras," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 846–851.

[16] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic cgras for irregular loop specialization," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 412–425.

[17] M. Pellauer, A. Parashar, M. Adler, B. Ahsan, R. Allmon, N. Crago, K. Fleming, M. Gambhir, A. Jaleel, T. Krishna *et al.*, "Efficient control and communication paradigms for coarse-grained spatial architectures," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–32, 2015.

[18] C. Tan, N. B. Agostini, J. Zhang, M. Minutoli, V. G. Castellana, C. Xie, T. Geng, A. Li, K. Barker, and A. Tumeo, "Opencgra: Democratizing coarse-grained reconfigurable arrays," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2021, pp. 149–155.

[19] C. Tan, C. Xie, T. Geng, A. Marquez, A. Tumeo, K. Barker, and A. Li, "Arena: Asynchronous reconfigurable accelerator ring to enable data-centric parallel computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2880–2892, 2021.

[20] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina *et al.*, "Magnet: A modular accelerator generator for neural networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[21] R. Bahr, C. Barrett, N. Bhagdikar, A. Carsello, R. Daly, C. Donovick, D. Durst, K. Fatahalian, K. Feng, P. Hanrahan *et al.*, "Creating an agile hardware design flow," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[22] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 268–281.

[23] T. Cheng, X. Chenhao, L. Ang, B. Kevin, and T. Antonino, "Aurora: Automated refinement of coarse-grained reconfigurable accelerators," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1–5.

[24] K. B. Thilini, D. Wijerathne, T. Mitra, and L.-S. Peh, "Revamp: A systematic framework for heterogeneous cgra realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[25] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," in *2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings.* IEEE, 2002, pp. 166–173.

[26] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017, pp. 184–189.

[27] D. Liu, S. Yin, L. Liu, and S. Wei, "Polyhedral model based mapping optimization of loop nests for cgras," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–8.

[28] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[29] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "Spr: an architecture-adaptive cgra mapping tool," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 191–200.

[30] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–10.

[31] L. Chen and T. Mitra, "Graph minor approach for application mapping on cgras," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–25, 2014.

[32] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–15.

[33] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1192–1197.

[34] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "Chordmap: Automated mapping of streaming applications onto cgra," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[35] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2. IEEE, 2005, pp. 729–734.

[36] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[37] J. Cong and J. Wang, "Polysa: Polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[38] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 821–834.

[39] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister *et al.*, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," *arXiv preprint arXiv:1911.09925*, 2019.

[40] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 703–716.

[41] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997.

[42] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proceedings of the 2003 Conference on Design, Automation and Test in Europe*, ser. DATE'03. IEEE, 2003, pp. 296–301.

[43] B. De Sutter, P. Coene, T. Vander Aa, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers and Tools for Embedded System*, ser. LCTES'08. ACM, 2008, pp. 151–160.

[44] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[45] S. Das, K. J. Martin, D. Rossi, P. Coussy, and L. Benini, "An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 6, pp. 1095–1108, 2018.

[46] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[47] J. Karimov, T. Rabl, and V. Markl, "Polybench: The first benchmark for polystores," in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2018, pp. 24–41.

[48] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, 2006, pp. 136–146.

[49] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 166–176.

[50] D. Liu, S. Yin, G. Luo, J. Shang, L. Liu, S. Wei, Y. Feng, and S. Zhou, "Data-flow graph mapping optimization for cgra with deep reinforcement learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 12, pp. 2271–2283, 2018.

[51] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 219–232.

[52] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[53] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, P. M. Phothilimthana, S. Wang, A. Goldie *et al.*, "Transferable graph optimizers for ml compilers," *arXiv preprint arXiv:2010.12438*, 2020.

[54] J. Jiao, D. Pal, C. Deng, and Z. Zhang, "Glaive: Graph learning assisted instruction vulnerability estimation," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 82–87.

[55] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for fpga hls using graph neural networks," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

APPENDIX

A. Abstract

Our artifact provides the source code of LISA, scripts to run LISA and collect data, performance data samples, and visualize data. The results reported in this paper were obtained on a CPU server with 14 cores. We provide a docker image including all the source code to allow the readers to quickly set up the environment.

B. Artifact check-list (meta-information)

- **Algorithm: LISA.**
- **Program: The benchmarks have been included.**
- **Run-time environment: Linux. We provide a docker image.**
- **Metrics: Mapping II value, compilation time, and GNN model accuracy.**
- **Output: Numerical results in text file with a script to visualize.**
- **Experiments: Python and shell scripts**
- **How much disk space required (approximately)? 10 GB.**
- **How much time is needed to prepare workflow (approximately)? 15 minutes.**
- **How much time is needed to complete experiments (approximately)? 6 hours.**
- **Archived (provide DOI)? doi.org/10.5281/zenodo.5788011 We are publicly releasing the source code of the GNN model that generates the labels for the dataflow graph (DFG). Our label-aware mapper is built on top of the CGRA-ME framework, which does not allow redistributing the software. Hence, we created our label-aware mapper as a patch to the original CGRA-ME code.**

C. Description

1) How to access: https://github.com/ecolab-nus/lisa

2) Hardware dependencies: To run all the tasks in parallel and reproduce the results, you need a CPU machine with more than 13 cores. It is possible to use fewer cores for the following experiments but the compilation time might be far longer.

3) Software dependencies: Docker.

4) Data sets: We have included PolyBench in the docker image.

D. Installation

Please follow the instructions to build the docker image and and create a container (20 minutes).

E. Experiment workflow

Start the container

```
$ docker start lisa_ae
$ docker exec -it lisa_ae /bin/bash
```

Map PolyBench DFGs on 3×3 CGRA (one hour)

```
$ cd /home/lzy/lisa/cgra_me
$ ./cgrame_env
$ conda activate lisa
$ python3 run_exper.py gnn_lisa 1 0 13
```

Note: you may use *nohup* to run all the tasks in the background

```
$ nohup python3 run_exper.py gnn_lisa 1
0 13 &
```

Then use *top* to check whether all the gnn_lisa_cgra tasks are finished or not.

Map PolyBench DFGs on 4×4 CGRA (one hour)

```
$ nohup python3 run_exper.py gnn_lisa 0
0 13 &
```

Note: These tasks take around one hour to finish.

Train GNN model for all the accelerators (20 minutes)

```
$ cd /home/lzy/lisa/lisa_gnn/lisa_gnn_model
$ conda activate lisa
$ bash run_all_training.sh
```

F. Evaluation and expected results

All the mapping results are in *lisa/cgra_me/result*. We use a script to extract the II value and compilation time from the result, and then visualize the results. Due to the randomness in simulated annealing, the results might be different from the reported ones. For most applications, the difference between reproduced II and reported one should not be bigger than one. For a few applications, the difference might be much bigger. We have included the ILP and SA mapping results in the corresponding data files. If you want to reproduce other evaluations, please find the *Experimentation customization* for more details.

```
$ cd /home/lzy/lisa/cgra_me/
&& python3 get_stat.py
result/cgra_me_3_3_result.txt
```

The above command will generate a figure *3_3.png* and a text file *3_3.txt* in *lisa/cgra_me/figs*.

To get 4×4 CGRA results, run:

```
$ python3 get_stat.py
result/cgra_me_4_4_result.txt
```

All the accuracy results can be found in *lisa/lisa_gnn/lisa_gnn_model/accuracy_log.txt*. The accuracy can have a maximum 6% difference from the reported one.

G. Experiment customization

We provide README.md in *lisa* and *lisa/cgra_me*. Through the instructions, users can evaluate different accelerators. Moreover, users can generate the training data set for a new accelerator, train the GNN model, and map DFG on the accelerator.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html