Dynasparse: Accelerating GNN Inference through Dynamic Sparsity Exploitation

Bingyi Zhang, Viktor Prasanna
University of Southern California, Los Angeles, California, USA
bingyizh@usc.edu, prasanna@usc.edu

***Abstract*—Graph Neural Network (GNN) inference is used in many real-world applications. Data sparsity in GNN inference, including sparsity in the input graph and the GNN model, offer opportunities to further speed up inference. Also, many pruning techniques have been proposed for model compression that increase the data sparsity of GNNs.**

We propose Dynasparse, a comprehensive hardware-software codesign on FPGA to accelerate GNN inference through dynamic sparsity exploitation. For this, we decouple the GNN computation *kernels* from the basic computation *primitives*, and explore hardware-software codesign as follows: 1) *Hardware design*: We propose a novel unified accelerator design on FPGA to efficiently execute various computation primitives. We develop a customized soft processor that is tightly coupled with the accelerator to execute a runtime system. Moreover, we develop efficient hardware mechanisms to profile the data sparsity and perform on-the-fly data format transformation to prepare the input data for various computation primitives; 2) *Software design*: We develop a runtime system that works synergistically with the accelerator to perform dynamic kernel-to-primitive mapping based on data sparsity. We implement Dynasparse on a state-of-the-art FPGA platform, Xilinx Alveo U250, and evaluate the design using widely used GNN models (GCN, GraphSAGE, GIN and SGC). For the above GNN models and various input graphs, the proposed accelerator and dynamic kernel-to-primitive mapping reduces the inference latency by $3.73\times$ on the average compared with the static mapping strategies employed in the state-of-the-art GNN accelerators. Compared with state-of-the-art CPU (GPU) implementations, Dynasparse achieves up to $56.9\times$ ($2.37\times$) speedup in end-to-end latency. Compared with state-of-the-art FPGA implementations, Dynasparse achieves $2.7\times$ speedup in accelerator execution latency.

***Index Terms*—Graph neural network, hardware-software codesign, hardware architecture, runtime system**

I. Introduction

Graph Neural Networks (GNNs) have achieved great success in many real-world applications, such as recommendation systems, social media, etc. *Low-latency GNN inference* is needed in many real-world applications, such as traffic prediction [1], scientific simulation [2], etc.

While many techniques [3], [4], [5], [6], [7], [8], [9] have been proposed to accelerate GNN inference, no work has systematically studied the data sparsity in GNNs to reduce the inference latency. GNNs ([10], [11]) involve various computation kernels, where there are three types of data sparsity: (1) *Sparsity of graph structure*: The graphs in the real-world applications are usually sparse, as most vertices have a small number of neighbors, (2) *Sparsity of vertex features*: The vertex features have various sparsity depending on the property of the graphs, activation function, etc., and (3) *Sparsity of GNN model*: The weight matrices in GNN models can also have data sparsity due to model pruning, etc. Moreover, the data sparsity can vary significantly based on the input graphs and GNN models (See Section II-B). Prior works directly map the GNN kernels to computation primitives (See Section II-B), and do not consider data sparsity, leading to potentially suboptimal performance.

To efficiently utilize the data sparsity in GNN inference, we propose to decouple the GNN *kernels* (feature aggregation and feature transformation) from the basic *primitives* (dense-dense matrix multiplication (GEMM), sparse-dense matrix multiplication (SpDMM), sparse-sparse matrix multiplication (SPMM)). A GNN kernel can be dynamically mapped to a primitive according to the sparsity of the data. However, there are several challenges: (1) While the sparsity of the graph structure and GNN model is known before the execution of inference (runtime), the sparsity of vertex features in the intermediate layers is known only at runtime. Therefore, static (compile time) kernel-to-primitive mapping may not be optimal. (2) While the GNN kernels can be mapped to various primitives, these primitives have different data formats and layouts. Switching the data format and the data layout can incur large overhead during execution. (3) Different primitives have different computation patterns and memory access patterns. While general purpose processors are efficient for dense primitives (GEMM), their data path and cache organization are inefficient for sparse primitives (SpDMM, SPMM).

To address the above challenges, we propose Dynasparse, a hardware-software codesign, which can efficiently exploit the data sparsity in GNN inference. For the hardware design, we use Field Programmable Gate Array (FPGA) as the target hardware platform. The programmability of FPGA allows us to (1) develop a customized data path and memory organization to support various computation primitives, (2) develop efficient hardware mechanism for sparsity profiling and transformation of data format and data layout (Section V-A), and (3) implement a lightweight and customized soft processor to perform dynamic kernel-to-primitive mapping at runtime. We summarize our main contributions as follows:

- We develop a complete system on FPGA with the following innovations in hardware design:
  - a novel hardware architecture, named Agile Computation Module, consisting of multiple Computation

Cores with flexible data path and memory organization that can execute various computation primitives, including GEMM, SpDMM and SPMM.
  - an efficient hardware mechanism that supports fast sparsity profiling and data format/layout transformation.
- We propose a soft processor and develop a runtime system on the soft processor to enable dynamic sparsity exploitation, including:
  - dynamic kernel-to-primitive (K2P) mapping strategy that automatically selects the optimal computation primitive for a given kernel based on an analytical performance model.
  - task scheduling strategy that manages the execution of the computation primitives on the accelerator to achieve load balance across multiple Computation Cores in the FPGA accelerator.
- We implement the proposed codesign on a state-of-the-art FPGA, Xilinx Alveo U250. For various GNN models and input graphs, the proposed accelerator and the dynamic kernel-to-primitive mapping reduce the inference latency by 3.73× on the average compared with the static mapping strategies employed in the state-of-the-art GNN accelerators. Compared with state-of-the-art CPU (GPU) implementations, Dynasparse achieves up to 56.9× (2.37×) speedup in end-to-end latency. Compared with state-of-the-art FPGA implementations, Dynasparse achieves 2.7× speedup in accelerator execution latency.

II. Background

A. Graph Neural Network

GNNs [10], [11] are proposed for representation learning on graphs $\mathcal{G}(\mathcal{V},\mathcal{E})$, and follow the message-passing paradigm (Algorithm 1) in which the vertices recursively aggregate information from the neighbors. $\boldsymbol{h}_v^L$ denotes the last-layer embedding of the target vertex v. The Update() is usually a Multi-Layer Perceptron that transforms the vertex features. An element-wise activation function is applied to the feature vectors after the Aggregate() and Update() in each layer. The output embedding $\boldsymbol{h}_v^L$ can be used for many downstream tasks, such as node classification ([11], [10]), link prediction, etc. GCN [10], GraphSAGE [11], GIN [12], and SGC [13] are some representative GNN models. Table I summarizes the notations used in this paper.

TABLE I: Notations

Notation	Description	Notation	Description
$\mathcal{G}(\mathcal{V},\mathcal{E})$	input graph	v_i	i^{th} vertex
$\mathcal{V}$	set of vertices	e_{ij}	edge from v_i to v_j
$\mathcal{E}$	set of edges	L	number of GNN layers
$\boldsymbol{A}$	graph adjacency matrix	$\mathcal{N}(i)$	the set of neighbors of v_i
$\boldsymbol{h}_i^{l-1}$	input feature vector of v_i at layer l	$\boldsymbol{W}^l$	weight matrix of layer l
$\boldsymbol{H}^{l-1}$	input feature matrix to layer l	$\sigma()$	activation function

Algorithm 1 GNN Computation Abstraction

Input: Input graph: $\mathcal{G}(\mathcal{V},\mathcal{E})$; vertex features: $\{\boldsymbol{h}_1^0,\boldsymbol{h}_2^0,\boldsymbol{h}_3^0,...,\boldsymbol{h}_{|\mathcal{V}|}^0\}$;
Output: Output vertex features $\{\boldsymbol{h}_1^L,\boldsymbol{h}_2^L,\boldsymbol{h}_3^L,...,\boldsymbol{h}_{|\mathcal{V}|}^L\}$;
1: **for** $l = 1...L$ **do**
2: **for** each vertex $v \in \mathcal{V}$ **do**
3: $\boldsymbol{a}_v^l$ = Aggregate$(\boldsymbol{h}_u^{l-1} : u \in \mathcal{N}(v))$
4: $\boldsymbol{z}_v^l$ = Update$(\boldsymbol{a}_v^l, \boldsymbol{W}^l)$, $\boldsymbol{h}_v^l = \sigma(\boldsymbol{z}_v^l)$

B. Data Sparsity in GNN inference

The *density* of a matrix is defined as the total number of non-zero elements divided by the total number of elements. Note that, the *sparsity* is given by $(1 - density)$. The computation kernels in GNNs involve three types of matrices: graph adjacency matrix $\boldsymbol{A}$, vertex feature matrix $\boldsymbol{H}$, and weight matrix $\boldsymbol{W}$. The adjacency matrix $\boldsymbol{A}$ of different graph datasets [14] can have different densities. For a given adjacency matrix, different parts of the matrix have different densities. Figure 2 shows the densities of feature matrices in GCN [10]. For different graphs, the input feature matrices have different densities. The feature matrices of different layers also have different densities. For the weight matrices, prior works ([15], [16]) have proposed various pruning techniques to reduce the density of the weight matrices.

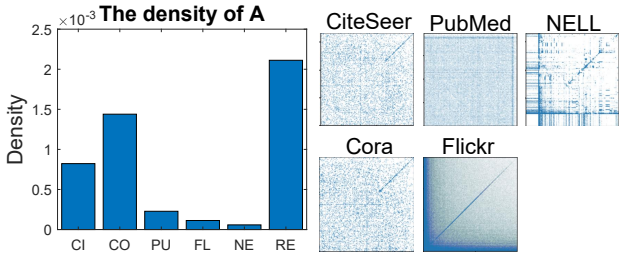


Fig. 1: The density and the visualization of graph adjacency matrix $\boldsymbol{A}$ of various graphs [14]

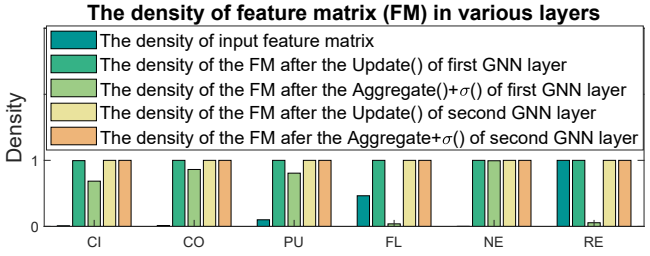


Fig. 2: Density of the feature matrices in the GCN model [10]

C. GNN Acceleration based on Data Sparsity

Although there are various data sparsities in GNNs, no prior work has systematically studied exploiting the data sparsity for GNN inference acceleration. HyGCN [3] and BoostGCN [4] map Aggregate() to SpDMM and map update() to GEMM, ignoring the data sparsity in feature matrices and weight matrices. AWB-GCN [17] maps both Aggregate() and update() to SpDMM. Then, they propose an accelerator to efficiently

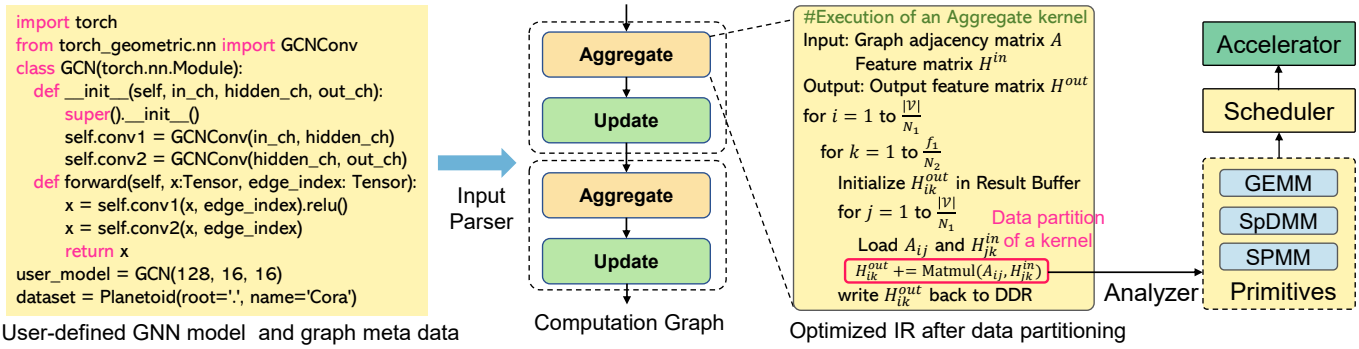


Fig. 3: Proposed workflow

execute SpDMM. However, they do not exploit the data sparsity in weight matrices. DeepBurning-GL [18] is a design automation framework that generates the optimized hardware accelerator given the information of the input graph and the GNN model. However, their framework needs to regenerate the optimized accelerator if the sparsity of the data is changed. To summarize, prior GNN accelerators do not fully exploit the data sparsity in GNNs, or are not flexible to exploit data sparsity in GNN inference.

III. Overview

A. Problem Definition

The computation **kernels** in GNN inference are feature aggregation and feature transformation which correspond to Aggregate() and Update() in the message-passing paradigm of GNN (Algorithm 1).

- Aggregate(): The input is graph adjacency matrix $\boldsymbol{A}$ and feature matrix $\boldsymbol{H}_{\text{in}}$. The output is $\boldsymbol{H}_{\text{out}} = \boldsymbol{A} \times \boldsymbol{H}_{\text{in}}$.
- Update(): The input is vertex feature matrix $\boldsymbol{H}_{\text{in}}$ and weight matrix $\boldsymbol{W}$. The output is $\boldsymbol{H}_{\text{out}} = \boldsymbol{H}_{\text{in}} \times \boldsymbol{W}$.

The computation **primitives** are GEMM, SpDMM and SPMM. While all the primitives perform multiplication of two input matrices to produce an output matrix, they have different ways of dealing with the zero elements: (1) GEMM views the two input matrices as dense matrices, and performs multiply-accumulate for all the matrix elements no matter whether an element is non-zero or not. (2) SpDMM views one input matrix as sparse matrix and skips the computation operations for all the zero elements in this input matrix. (3) SPMM takes two input sparse matrices and skips the computation operations for all the zero elements in the two input matrices.

This work targets full-graph inference: given a GNN model and an input graph, we perform the message-passing paradigm (Algorithm 1) in the full input graph to obtain the embeddings of all the vertices. Full-graph inference has been widely studied in the literature [3], [17], [4]. Our objective is to exploit the data sparsity of GNN kernels to further accelerate the inference process. We assume that the sparsity of the data is unknown before the accelerator design or hardware execution. Our intent is to develop a single hardware-software codesign on FPGA that is efficient and flexible to support various graphs and GNN models of various data sparsity. Therefore, the proposed work does not require regenerating the FPGA accelerator if data sparsity changes.

B. System Overview

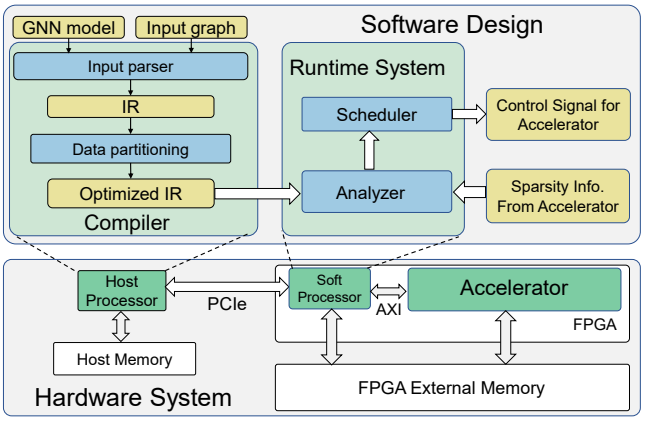


Fig. 4: Overview of the proposed system

Figure 4 depicts the proposed system design. The software comprises of a *compiler* and a *runtime system*. The hardware system has three components:

- **Host processor**: The compiler is executed on the host processor to perform compilation (preprocessing) for the input GNN model and the input graph to generate the intermediate representation (IR). The IR is sent to the soft processor for execution.
- **Soft Processor on FPGA**: The runtime system is executed on the soft processor. It takes the IR as input, and dynamically schedules the computation tasks on the accelerator by sending control signals to the accelerator.
- **Accelerator on FPGA**: It executes the three computation primitives (GEMM, SpDMM, SPMM), profiles data sparsity, and performs data layout/format transformation. It receives the control signals from the soft processor to execute the computation tasks, and also sends the data sparsity information to the soft processor at runtime.

The workflow is illustrated in Figure 3. The execution of GNN inference consists of two steps:

Step 1. Compilation/Preprocessing: The compiler (See Section IV-A) performs the following preprocessing: ❶ **Generating intermediate representation (IR):** It takes the specifications of the user-defined GNN model and the graph meta data as input, and generates the IR for the GNN computation graph (See Figure 3). ❷ **Data partitioning**: The compiler performs data partitioning for each kernel. Data partitioning is required since (1) in real-world applications, the input graph can be very large and the FPGA accelerator has limited on-chip memory, (2) within a matrix, different parts of the matrix can have different data sparsity. Data partitioning enables fine-grained kernel-to-primitive mapping (See Section VI-B), leading to more efficient sparsity exploitation. ❸ **Preprocessing of data sparsity**: When the compiler performs data partitioning, it uses counters to profile the sparsity information of graph adjacency matrix $\boldsymbol{A}$, weight matrix $\boldsymbol{W}$, and input feature matrix $\boldsymbol{H}^0$. Note that the sparsity information of the feature matrices in the intermediate layers $\{\boldsymbol{H}^1, ..., \boldsymbol{H}^L\}$ is unknown at compile time and is profiled by the accelerator at runtime.

Step 2. Runtime execution: At runtime, the soft processor and the accelerator collaborate to perform GNN inference. The runtime system on the soft processor consists of *an Analyzer* and *a Scheduler*. The accelerator contains multiple Computation Cores. The Analyzer takes the optimized IR from the compiler and the data sparsity information from the compiler and the accelerator to dynamically map a kernel to a primitive based on a performance model. Then, the Scheduler schedules the execution of the primitives on the accelerator (Section VI-C). The runtime system performs dynamic kernel-to-primitive (K2P) mapping. Note that the mapping must be performed dynamically at runtime: (1) The densities of the feature matrices in the intermediate layers $\{\boldsymbol{H}^1, ..., \boldsymbol{H}^L\}$ are unknown before runtime; (2) The Computation Core has various execution modes (Section V-B) with each mode executing a specific primitive. These execution modes have different computation efficiency (See Section VI-A) with respect to the density of data. As a result, for a computation kernel of high density, executing it using GEMM primitive on the Computation Core will be more efficient. For a GNN kernel of low density, executing it using SpDMM or SPMM primitive on the Computation Core will be more efficient. To handle this scenario, we build an analytical performance model (Section VI-B) to estimate the execution latency of a given primitive on the Computation Core with respect to the data sparsity.

The rest of the paper is organized as follows: Section IV covers the details of the compiler; Section V introduces the proposed accelerator design; Section VI introduces the proposed runtime system; Section VII and VIII describe the implementation details and evaluation results, respectively.

IV. Compiler

A. Intermediate Representation (IR)

We define the meta data in the IR in Table II, including the meta data of the kernel and the meta data of the execution scheme. The execution scheme of a kernel is the plan for executing the kernel. The IR defines two types of kernels – *Aggregate* and *Update*, corresponding to Aggregate() and Update() in the GNN abstraction (See Algorithm 1).

TABLE II: Meta data of a kernel in the IR

Layer Type	Aggregate(0), Update(1)
Layer ID	1,2,3,...
Input Dimension	f_{in}
Output Dimension	f_{out}
# of vertices	$\lvert\mathcal{V}\rvert$
# of edges	$\lvert\mathcal{E}\rvert$
Aggregation operator	Max, Sum, Min, Mean
Activation type	ReLU, PReLU
Activation enabled	True, False
Meta data of execution scheme	{...} (See Algorithm 2 and 3)

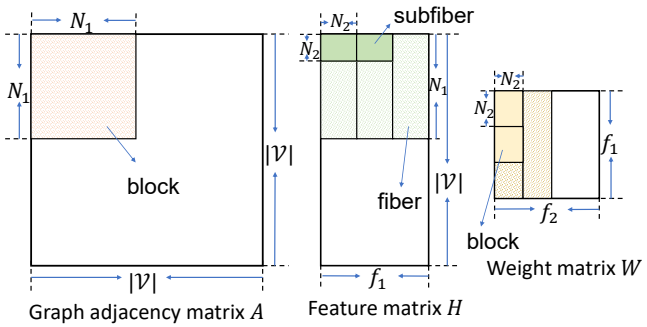


Fig. 5: Illustration of data and model partitioning

B. Compilation Process

The compilation process has two steps (See Figure 4):

- **Step 1 (parsing the input)**: The compiler takes the specification of the GNN model (Defined using Pytorch Geometric Library [14]) and the graph meta data as input, and generates the computation graph for GNN inference (See the example in Figure 3). The computation graph has $\sum_{l=1}^{L} k_l$ nodes, where L denotes the number of GNN layers in the GNN model and k_l denotes the number of kernels in layer l ($1 \leqslant l \leqslant L$). In the computation graph, each node represents the IR of a kernel. An edge denotes the data dependency between two kernels.
- **Step 2 (data partitioning and execution scheme generation)**: The compiler performs data partitioning for each kernel and generates the execution scheme for the kernel. Then, the meta data of the execution scheme is stored in the IR to produce the optimized IR (See Figure 3) that is sent to the runtime system.

C. Data Partitioning

Figure 5 depicts the proposed data partition scheme. The graph adjacency matrix $\boldsymbol{A}$ has the dimension $|\mathcal{V}| \times |\mathcal{V}|$. $\boldsymbol{A}$ is partitioned into blocks with each block having dimension of $N_1 \times N_1$. We use $\boldsymbol{A}_{ij}$ to denote a block where $\boldsymbol{A}_{ij} = \boldsymbol{A}[i*N_1 : (i+1)*N_1][j*N_1 : (j+1)*N_1]$. The feature matrix $\boldsymbol{H}$ of dimension $|\mathcal{V}| \times f_1$ is partitioned into fibers. Each fiber has dimension $N_1 \times N_2$ and $\boldsymbol{H}_{ij} = \boldsymbol{H}[i*N_1 : (i+1)*N_1][j*N_2 :$

$(j+1) * N_2]$. We further partition each fiber into subfibers where each subfiber has size $N_2 \times N_2$. $\boldsymbol{H}_{ij-k}$ denotes the k^{th} subfiber of $\boldsymbol{H}_{ij}$. We use $\boldsymbol{H}_{i-k}$ to denote the concatenation of $\{\boldsymbol{H}_{i1-k}, \boldsymbol{H}_{i2-k}, ..., \boldsymbol{H}_{i\frac{N_1}{N_2}-k}\}$. The weight matrix $\boldsymbol{W}$ is partitioned into blocks with each block having size of $N_2 \times N_2$. $\boldsymbol{W}_{ij} = \boldsymbol{W}[i * N_2 : (i+1) * N_2][j * N_2 : (j+1) * N_2]$.

D. Execution Scheme

Based on the data partition scheme, the compiler generates the execution plan for each computation kernel, shown in Algorithm 2 and 3. The execution of a computation *kernel* is decomposed into a set of independent computation *tasks*. Each task performs the execution of an output data partition and there is no data dependency among the tasks within a kernel. Each task performs the multiplication of data partitions to obtain an output data partition, and the computation primitive to execute the matrix multiplication `Matmul()` is determined by the Runtime System. We generalize the representation of a task in Algorithm 4.

Algorithm 2 Execution scheme of an Aggregate kernel

Input: Graph adjacency matrix $\boldsymbol{A}$; Input feature matrix $\boldsymbol{H}^{\text{in}}$;
Output: Output feature matrix $\boldsymbol{H}^{\text{out}}$;

Execute the Aggregate kernel — Kernel
for i = 1 to $\frac{|\mathcal{V}|}{N_1}$ **do**
  for k = 1 to $\frac{f_1}{N_2}$ **do**
    Initialize $\boldsymbol{H}^{\text{out}}_{ik}$ in the Result Buffer — Task
    for j = 1 to $\frac{|\mathcal{V}|}{N_1}$ **do**
      Load $\boldsymbol{A}_{ij}$ and $\boldsymbol{H}^{\text{in}}_{jk}$
      $\boldsymbol{H}^{\text{out}}_{ik}$ += Matmul($\boldsymbol{A}_{ij}, \boldsymbol{H}^{\text{in}}_{jk}$)
    Write $\boldsymbol{H}^{\text{out}}_{ik}$ back to DDR memory

Algorithm 3 Execution scheme of an Update Kernel

Input: Input feature matrix $\boldsymbol{H}^{\text{in}}$; Weight matrix $\boldsymbol{W}$;
Output: Output feature matrix $\boldsymbol{H}^{\text{out}}$;

Execute the Update kernel — Kernel
for i = 1 to $\frac{|\mathcal{V}|}{N_2}$ **do**
  for k = 1 to $\frac{f_2}{N_2}$ **do**
    $g = \lfloor \frac{i \times N_2}{N_1} \rfloor$, $f = i\%(\frac{N_1}{N_2})$ — Task
    Initialize $\boldsymbol{H}^{\text{out}}_{gk-f}$ in the Result Buffer
    for j = 1 to $\frac{f_1}{N_1}$ **do**
      Load $\boldsymbol{H}^{\text{in}}_{gj-f}$ and $\boldsymbol{W}_{jk}$
      $\boldsymbol{H}^{\text{out}}_{gk-f}$ += Matmul($\boldsymbol{H}^{\text{in}}_{gj-f}, \boldsymbol{W}_{jk}$)
    Write $\boldsymbol{H}^{\text{out}}_{gk-f}$ back to DDR memory

Algorithm 4 A computation task

Input: $\{\boldsymbol{X}_{i1}, \boldsymbol{X}_{i2}, ..., \boldsymbol{X}_{iK}\}$ and $\{\boldsymbol{Y}_{1j}, \boldsymbol{Y}_{2j}, ..., \boldsymbol{X}_{Kj}\}$;
Output: Output matrix: $\boldsymbol{Z}_{ij}$;

Initialize $\boldsymbol{Z}_{ij}$ in the Result Buffer
for k = 1 to K **do**
  Load $\boldsymbol{X}_{it}$ and $\boldsymbol{Y}_{tj}$ onto the on-chip buffer
  $\boldsymbol{Z}_{ij}$ += Matmul($\boldsymbol{X}_{it}, \boldsymbol{Y}_{tj}$)
Write $\boldsymbol{Z}_{ij}$ back to DDR memory

V. Accelerator Design

In Section V-A, we introduce the data layout and data format that are used by Dynasparse. In Section V-B1, we introduce the Agile Computation Module which can execute three primitives (GEMM, SpDMM, and SPMM). In Section V-B2, we describe the hardware mechanism for sparsity profiling, and data format/layout transformation.

A. Data format and data layout

Data format: We store the matrices using *sparse* format or *dense* format. We use Coordinate (COO) format to represent a sparse matrix where an nonzero element is represented using a three-tuple $(col, row, value)$ denoting the column index, row index, and value, respectively. COO format is the standard data format used in the state-of-the-art GNN libraries [14].

Data layout: It defines the order of storing the matrix elements. For a sparse matrix in the row-major order, the elements within the same row are stored in contiguous locations. Otherwise, it is *column-major* order. Similarly, row-major and column-major order for a dense matrix can be derived.

Notations: For a matrix $\boldsymbol{B}$, we use $\boldsymbol{B}[i]$ to denote the i^{th} row of $\boldsymbol{B}$ and use $\boldsymbol{B}[i:j]$ to denote the submatrix of $\boldsymbol{B}$ from i^{th} row to $(j-1)^{\text{th}}$ row. We use $\boldsymbol{B}[i][j]$ to denote the element of $\boldsymbol{B}$ at the i^{th} row and the j^{th} column. An element $(j, i, value)$ in sparse $\boldsymbol{B}$ will also be denoted as $\boldsymbol{B}[i][j] = value$.

B. Microarchitecture

Each Computation Core (Figure 6) has an Agile Computation Module (ACM) and an Auxiliary Hardware Module (AHM). The ACM has an ALU (Arithmetic Logic Unit) array of dimension $p_{sys} \times p_{sys}$ and the interconnection among the ALUs are shown in Figure 7. AHM performs sparsity profiling, data layout and format transformation (Section V-B2).

1) Agile Computation Module (ACM): It has four data buffers – BufferU, BufferO, BufferP and Result Buffer (RB). Buffer[U/O/P] store the input matrices and RB stores the output matrix. Each Buffer has p_{sys} memory banks (denoted bank 0 to bank $p_{sys} - 1$) for parallel on-chip memory access. Each ALU can execute various arithmetic operations, including multiplication, max, addition, etc. There are two interconnection networks – Index Shuffle Network (ISN) and Data Shuffle Network (DSN) – for data communication. The ACM has three execution modes – *GEMM mode*, *SpDMM mode* and *SPMM mode*. The required data format and layout for various execution modes are summarized in Table III.

GEMM Mode: The ALU array is organized as a two-dimensional systolic array (See Figure 7) to execute GEMM using output stationary dataflow. The systolic array can execute p_{sys}^2 multiply-accumulate (MAC) operations per clock cycle.
SpDMM Mode: The ALU array is divided into $p_{sys}/2$ Update Units and $p_{sys}/2$ Reduce Units. Each Update or Reduce Unit has an ALU array of size $p_{sys}/2 \times 2$. Multiplication of a sparse matrix with a dense matrix is executed using the Scatter-Gather Paradigm shown in Algorithm 5. The sparse matrix denoted as $\boldsymbol{X}$ (in BufferU) is stored in row-major order using

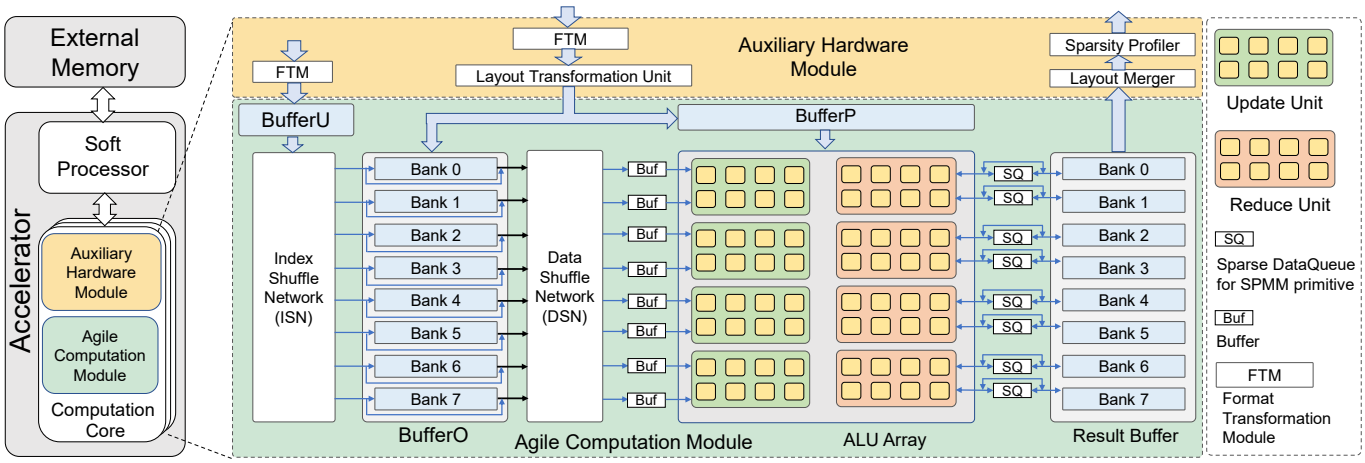


Fig. 6: Diagram of a Computation Core

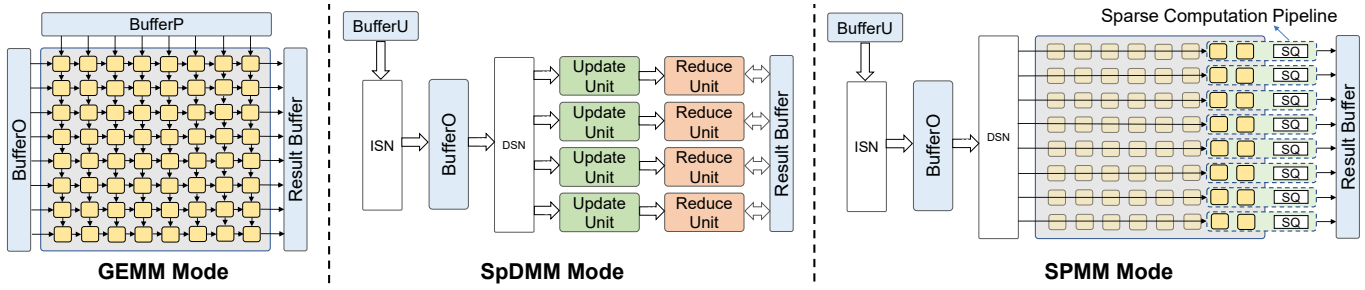


Fig. 7: Various execution modes of a Computation Core

TABLE III: Buffer (data format) [data layout] requirement to store the input/output matrices for executing $\boldsymbol{Z} = \boldsymbol{X} \times \boldsymbol{Y}$ in the three execution modes

	$\boldsymbol{X}$	$\boldsymbol{Y}$	$\boldsymbol{Z}$
GEMM	BufferO (dense) [row major]	BufferP (dense) [column major]	Result Buffer (dense) [row major]
SpDMM	BufferU (sparse) [row or column major]	BufferO (dense) [row major]	Result Buffer (dense) [row major]
SPMM	BufferU (sparse) [row major]	BufferO (sparse) [row major]	Result Buffer (dense) [row major]

Algorithm 5 SpDMM using Scatter-Gather Paradigm

Input: Sparse matrix (BufferU): $\boldsymbol{X}$; Dense matrix (BufferO): $\boldsymbol{Y}$;
Output: Output matrix (Result Buffer): $\boldsymbol{Z}$ ($\boldsymbol{Z} = \boldsymbol{X} \times \boldsymbol{Y}$);

```
while not done do
    for each e(i, j, value) in X Parallel do        ▷ Scatter Phase
        Fetch Y[i] from BufferO                     ▷ ISN routes e to BufferO
        Form input pair (Y[i], e)
        # DSN routes input pair to Update Units
    for each input pair Parallel do                 ▷ Gather Phase
        u ←Update(Y[i], e.value)                    ▷ Update Unit
        Fetch Z[j] from Result Buffer
        Z[j] ← Reduce(u)                            ▷ Reduce Unit
```

COO format. The dense matrix denoted as $\boldsymbol{Y}$ (in BufferO) is stored in row-major order using dense format, and $\boldsymbol{Y}[i]$ is stored in bank $(i \mod p_{\text{sys}})$ of BufferO. Each non-zero element $e(i, j, weight)$ in $\boldsymbol{X}$ is fetched from the BufferU ($p_{sys}/2$ elements can be fetched from BufferU per cycle) and sent to the ISN. Then e is routed to bank $(i \mod p_{\text{sys}})$ for fetching $\boldsymbol{Y}[i]$, which forms the input data pair ($\boldsymbol{Y}[i]$, e). The input pair is routed to the $(j \mod p_{\text{sys}}/2)^{\text{th}}$ Update Unit. The Update Unit performs the multiplication of $e.value$ and $\boldsymbol{Y}[i]$ to produce the intermediate result u. Then the corresponding Reduce Unit adds u to $\boldsymbol{Z}[j]$. SpDMM Mode can efficiently skip zero elements in the sparse matrix $\boldsymbol{X}$. The SpDMM Mode can execute $p_{sys}^2/2$ MAC operations per clock cycle.

SPMM Mode: The ALU array is organized as p_{sys} parallel Sparse Computation Pipelines (SCP) as shown in Figure 7. Each SCP has two ALUs to perform multiplication of two non-zero elements and the merging of intermediate results. Each SCP also has a Sparse Data Queue (SQ) to store the intermediate results in sparse format. The multiplication of two input sparse matrices is executed using the Row-wise Product with Scatter-Gather paradigm as shown in Algorithm 6. For Row-wise Product, an row $\boldsymbol{Z}[j]$ of output matrix $\boldsymbol{Z}$ is calculated through:

$$\boldsymbol{Z}[j] = \sum_i \boldsymbol{X}[j][i] * \boldsymbol{Y}[i] \tag{1}$$

For calculating the output matrix $\boldsymbol{Z}$, a SCP is assigned the

Algorithm 6 SPMM using Row-wise Product with Scatter-Gather Paradigm

Input: Sparse matrix (BufferU): $\boldsymbol{X}$; Sparse matrix (BufferO): $\boldsymbol{Y}$;
Output: Output matrix (In Result Buffer): $\boldsymbol{Z} = \boldsymbol{X} \times \boldsymbol{Y}$;

for each row $\boldsymbol{Z}[j]$ in $\boldsymbol{Z}$ **Parallel do**
  Assign the workload of $\boldsymbol{Z}[j]$ to SCP[$j\%p_{sys}$]
  load $\boldsymbol{Z}[j]$ to the Sparse Data Queue from Results Buffer
  for each $e(i, j, value)$ in $\boldsymbol{X}[j]$ **do** ▷ Scatter Phase
    Fetch $\boldsymbol{Y}[i]$ from BufferO ▷ ISN routes e to BufferO
    Form input pair ($\boldsymbol{Y}[i]$, e) ▷ DSN routes input to SCPs
  for each input pair ($\boldsymbol{Y}[i]$, e) **do** ▷ Gather Phase
    for each non-zero $\boldsymbol{Y}[i][k]$ in $\boldsymbol{Y}[i]$ **do** ▷ SCP
      Produce $u \leftarrow$ Update($e.value \times \boldsymbol{Y}[i][k]$)
      Merge $\boldsymbol{Z}[j][k] \leftarrow$ Reduce(u)
  Store $\boldsymbol{Z}[j]$ to the Result Buffer ▷ Obtain $\boldsymbol{Z}[j]$

workload of an row of output matrix (Equation 1). p_{sys} SCPs can calculate p_{sys} output rows in parallel until all the rows of the output matrices are calculated. To efficiently execute Row-wise Product, all input sparse matrices ($\boldsymbol{X}$, $\boldsymbol{Y}$) and output matrix are stored using COO format in row-major order (See Section V-A). Using SPMM Mode, we can skip the zero elements in both the input matrices. SPMM Mode can execute p_{sys} multiply-accumulate (MAC) operations per clock cycle.

Mode switching: The execution mode is set by the control bits of the hardware multiplexers in ACM. The overhead of switching execution modes is just one clock cycle.

Trade-off: The three execution modes have different ways of dealing with non-zero elements in the two input matrices (Section III-A). Therefore, their execution time of multiplying two input matrices depends on the data sparsity. We analyze the trade-off of the three execution modes w.r.t. data sparsity in Section VI-A.

2) Auxiliary Hardware Module (AHM): While the ACM can execute various primitives, the data format and layout should meet the requirement of the execution modes (Table III). Moreover, the soft processor needs the data sparsity information at runtime for dynamic K2P mapping. To this end, the AHM has the following hardware modules: (1) a Layout Transformation Unit and a Layout Merger to transform the data layout, (2) a Sparsity Profiler (SP) to obtain the density of the intermediate results, (3) Format Transformation Module (FTM), which contains a Sparse-to-Dense Module and a Dense-to-Sparse Module.

Layout Transformation Unit (LTU): Transformation of the data layout between row-major order and column-major order is transposing a matrix. LTU is implemented using a streaming permutation network [19] (See [19] for details) for efficient layout transformation. Since most of the on-chip data are stored using row-major order, we store all the data partitions of ($\boldsymbol{A}$, $\boldsymbol{H}$, $\boldsymbol{W}$) in the external memory using row-major order to minimize the effort for data layout transformation.

Layout Merger: When the accelerator executes a task (See algorithm 4), the results $\boldsymbol{Z}$ can be in row-major or column-major order. Therefore, in Results Buffer, we store two partial results of $\boldsymbol{Z}$ in row-major and column-major order, respectively. The two partial results of $\boldsymbol{Z}$ are merged by Layout Merger into row-major order when $\boldsymbol{Z}$ is sent back to the external memory. Note that the LTU is also used by BufferO to transform the data layout for $\boldsymbol{X}_2^\intercal$ (column-major order of $\boldsymbol{X}_2$).

Sparsity Profiler: To profile the density of sparse matrix or dense matrix, we use the adder tree based design for the Sparsity Profiler. At the output port of the Result Buffer, we implement a comparator array with an adder tree to count the total number of non-zero elements. After obtaining the data sparsity of the current output matrix, the sparsity information is sent to the soft processor.

Dense-to-Sparse (D2S) Module: It transforms an array from dense format to sparse format. Suppose the D2S Module can read n elements per clock cycle. Then, the D2S Module has $\log(n)$ pipeline stages. For an n-element array, we use the value of Prefix-Sum to indicate the number of zeros before an element in this array. An example is shown in Figure 8. In Stage i ($1 \leqslant i \leqslant \log(n)$), an array element will be shifted left by 2^{i-1} positions if the $(i-1)^{\text{th}}$ bit of Prefix Sum value is equal to 1. The throughput of D2S Module is n elements per cycle. For example, a DDR4 channel of the FPGA board can output 16 32-bit data per cycle. A D2S Module of $n = 16$ is sufficient to match the data rate of a DDR4 channel. The architecture of S2D is similar to D2S, but in the reverse direction.

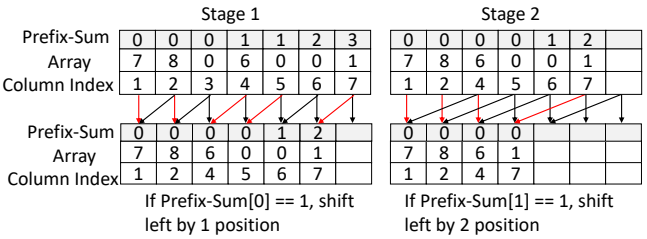


Fig. 8: Transforming dense format to sparse format

3) Double Buffering: We exploit double buffering technique for Buffer[U/O/P] and Results Buffer. Therefore, when the Computation Core is executing the current task, the Buffers can load the input data of the next task. The data sparsity profiling, data layout and format transformation are streaming processes that can be executed during the data loading/storing process. Double Buffering not only overlaps the computation and data communication, but also hides the overhead of sparsity profiling and data layout/format transformation.

VI. Runtime System

A. Performance Model

The performance model predicts the execution time of the primitives for a given data sparsity. For analysis, we denote the two input matrices to a Computation Core as $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{Y} \in \mathbb{R}^{n \times d}$ where $\boldsymbol{X}$ has the density $\alpha_{\boldsymbol{X}}$ ($0 \leqslant \alpha_{\boldsymbol{X}} \leqslant 1$) and $\boldsymbol{Y}$ has the density $\alpha_{\boldsymbol{Y}}$ ($0 \leqslant \alpha_{\boldsymbol{Y}} \leqslant 1$).

In the GEMM mode, the two input matrices are viewed as dense matrices and the Computation Core can execute

TABLE IV: Performance model

	GEMM	SpDMM	SPMM
MACs per cycle	p_{sys}^2	$p_{sys}^2/2$	p_{sys}
Execution time (cycles)	$\frac{mnd}{p_{sys}^2}$	$\alpha_{\min}\frac{2mnd}{p_{sys}^2}$, where $\alpha_{\min} = \text{Min}(\alpha_{\boldsymbol{X}}, \alpha_{\boldsymbol{Y}})$	$\alpha_{\boldsymbol{X}}\alpha_{\boldsymbol{Y}}\frac{mnd}{p_{sys}}$

p_{sys}^2 MACs per cycle. Therefore, the total execution time is $\frac{mnd}{p_{sys}^2}$ cycles. In the SpDMM mode, the Computation Core can skip the zero elements in one input matrix and can execute $p_{sys}^2/2$ MACs per cycle. We view the input matrix with lower density as a sparse matrix and view another input matrix as a dense matrix. Therefore, the total execution time is $\alpha_{\min}\frac{2mnd}{p_{sys}^2}$ cycles where $\alpha_{\min} = \text{Min}(\alpha_{\boldsymbol{X}}, \alpha_{\boldsymbol{Y}})$. In the SPMM mode, the Computation Core can skip the zero elements in both two input matrices and can execute p_{sys} MACs per cycle. Therefore, the total execution time is $\alpha_{\boldsymbol{X}}\alpha_{\boldsymbol{Y}}\frac{mnd}{p_{sys}}$ cycles. In the state-of-the-art FPGA such as Xilinx Alveo U250, the dimension of a Computation Core p_{sys} can be chosen to be $\geqslant 8$. We denote $\alpha_{max} = \text{Max}(\alpha_{\boldsymbol{X}}, \alpha_{\boldsymbol{Y}})$. To summarize, for executing $\boldsymbol{Z} = \boldsymbol{X} \times \boldsymbol{Y}$ on a Computation Core, when $\alpha_{\min} \geqslant \frac{1}{2}$, GEMM Mode has the least execution time; When $\alpha_{\min} < \frac{1}{2}$ and $\alpha_{max} \geqslant \frac{2}{p_{sys}}$, SpDMM Mode has the least execution time; When $\alpha_{\min} < \frac{1}{2}$ and $\alpha_{max} < \frac{2}{p_{sys}}$, SPMM Mode has the least execution time. The three cases are non-overlapping and cover all the points in the domain $0 \leqslant \alpha_{min} \leqslant \alpha_{max} \leqslant 1$.

B. Dynamic Kernel-to-primitive Mapping

Algorithm 7 Dynamic kernel-to-primitive (K2P) mapping Algorithm for a computation task

Input: $\{\boldsymbol{X}_{i1}, \boldsymbol{X}_{i2}, ..., \boldsymbol{X}_{iK}\}$ and $\{\boldsymbol{Y}_{1j}, \boldsymbol{Y}_{2j}, ..., \boldsymbol{X}_{Kj}\}$;
1: **for** $t = 1$ to K **do**
2: TargetPrimitive($\boldsymbol{X}_{it}$, $\boldsymbol{Y}_{tj}$) ← NULL
3: The buffers to store $\boldsymbol{X}_{it}$ and $\boldsymbol{Y}_{tj}$: $B_{\boldsymbol{X}_{it}}$, $B_{\boldsymbol{Y}_{tj}}$
4: $\alpha_{\min} = \text{Min}(\alpha_{\boldsymbol{X}_{it}}, \alpha_{\boldsymbol{Y}_{tj}})$ ▷ $\alpha_{\boldsymbol{X}_{it}}$: The density of $\boldsymbol{X}_{it}$
5: $\alpha_{\max} = \text{Max}(\alpha_{\boldsymbol{X}_{it}}, \alpha_{\boldsymbol{Y}_{tj}})$ ▷ $\alpha_{\boldsymbol{Y}_{tj}}$: The density of $\boldsymbol{Y}_{tj}$
6: **if** $\alpha_{\min} = 0$ **then** ▷ Skip empty input matrix
7: Skip the multiplication of $\boldsymbol{X}_{it}$ and $\boldsymbol{Y}_{tj}$
8: **if** $\alpha_{\min} \geqslant \frac{1}{2}$ **then**
9: TargetPrimitive($\boldsymbol{X}_{it}$, $\boldsymbol{Y}_{tj}$) ← GEMM
10: $B_{\boldsymbol{X}_{it}}$ ← BufferO and $B_{\boldsymbol{Y}_{tj}}$ ← BufferP
11: **else**
12: **if** $\alpha_{\max} \geqslant \frac{2}{p_{sys}}$ **then**
13: TargetPrimitive($\boldsymbol{X}_{it}$, $\boldsymbol{Y}_{tj}$) ← SpDMM
14: $B_{\text{argmin}(\alpha_M)}$ ← BufferU, ($M \in \{\boldsymbol{X}_{it}, \boldsymbol{Y}_{tj}\}$)
15: $B_{\text{argmax}(\alpha_M)}$ ← BufferO, ($M \in \{\boldsymbol{X}_{it}, \boldsymbol{Y}_{tj}\}$)
16: **else**
17: TargetPrimitive($\boldsymbol{X}_{it}$, $\boldsymbol{Y}_{tj}$) ← SPMM
18: $B_{\boldsymbol{X}_{it}}$ ← BufferU and $B_{\boldsymbol{Y}_{tj}}$ ← BufferO

The Analyzer performs dynamic kernel-to-primitive (K2P) mapping for each computation task shown in Algorithm 7. For each pair of input matrices ($\boldsymbol{X}_{it}$, $\boldsymbol{Y}_{tj}$), the runtime system fetches their densities $\alpha_{\boldsymbol{X}_{it}}$ and $\alpha_{\boldsymbol{Y}_{tj}}$. Then, the Analyzer determines the target primitive for multiplying $\boldsymbol{X}_{it}$ and $\boldsymbol{Y}_{tj}$, and also determines which buffers to store $\boldsymbol{X}_{it}$ and $\boldsymbol{Y}_{tj}$. The proposed dynamic K2P algorithm has the computation complexity $\mathcal{O}(K) = \mathcal{O}(\frac{|\mathcal{V}|}{N_1} + \frac{f_1}{N_2})$ for a computation task, which has small overhead compared with total computation complexity of a task $\mathcal{O}(|\mathcal{V}| * N_2 + f_1 * N_2^2)$. See evaluation results in Section VIII-C. There are several benefits: (1) the proposed dynamic K2P mapping is fine-grained that for different data partitions, we can use different primitives to efficiently exploit the data sparsity in the input. (2) When the accelerator is executing kernel l, the runtime system can perform K2P mapping for kernel $l+1$. Therefore, the overhead of the runtime system can be hidden.

C. Task Scheduling

The scheduler performs scheduling of computation tasks (See Section IV-A) on the parallel Computation Cores as shown in Algorithm 8. The proposed task scheduling is a *dynamic task scheduling strategy*. Each Computation Core maintains an interrupt interface to trigger the interrupt handling in the soft processor when the Computation Core is idle. Then, the soft processor assigns a task to the Computation Core.

Algorithm 8 Task scheduling

Input: Intermediate Representation of the GNN model: IR; The number of computation kernels in the IR: L;
Output: Output of the GNN model;
1: **for** $l = 1$ to L **do**
2: **for** each *Task* in kernel l of IR **parallel do**
3: **if** there is an idle CC: CC_i **then**
4: Assign this *Task* to CC_i
5: CC_i executes this computation *Task*
6: Wait until all the *Tasks* in kernel l are executed

Partition size (N_1, N_2): The objectives of the data partitioning are to (1) enable fine-grained data sparsity exploitation, (2) exploit data locality, and (3) maximize resource utilization during dynamic task scheduling (Algorithm 8). Specifically, to maximize resource utilization that keeps all the Computation Cores busy, the compiler selects the partition configuration (N_1, N_2) such that there will be at least $\eta * N_{CC}$ ($\eta \geq 1$) tasks in each computation kernel assigned to N_{CC} Computation Cores. η is a factor that is determined empirically. Since different partitions can have different data sparsity leading to the different workloads of the tasks, small η (e.g., $\eta = 1$) can potentially lead to long idle time for the Computation Cores with small workloads. Therefore, we set $\eta = 4$ following state-of-the-art graph processing frameworks [20].

To meet the above three objectives, we use a heuristic approach to determine the partition size as shown in Algorithm 9. As shown in Algorithm 2 (line 2-3), the number of tasks of an Aggregate kernel is $\mathcal{T}_a = \frac{|\mathcal{V}| * f_1}{N_1 * N_2}$. Also, as shown in Algorithm 3 (line 2-3), the number of tasks of an Update

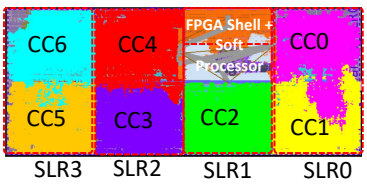
kernel is $\mathcal{T}_u = \frac{|\mathcal{V}|*f_2}{N_2*N_2}$. For simplicity, we use Q to denote the workload of a kernel (e.g., $Q = |\mathcal{V}| * f_1$ or $Q = |\mathcal{V}| * f_2$), and use $Q[k]$ to denote the workload of k^{th} ($1 \leqslant k \leqslant L$) kernel. We use $p()$ to denote the function that determines the number of tasks of a kernel based on Q, N_1, and N_2. For example, $\mathcal{T}_a = p(Q, N_1, N_2) = \frac{Q}{N_1*N_2}$ and $\mathcal{T}_u = p(Q, N_2) = \frac{Q}{N_2*N_2}$. In line 9 and line 15 of Algorithm 9, the partition size of each kernel is constrained by $N_{it} = \min(N_{it}, N_{\max})$, where $\min(N', N_{\max})$ is the largest partition size such that $N_{it} \leqslant N'$ and $N_{it} \leqslant N_{\max}$. $N \leqslant N'$ ensures that there will be at least $\eta * N_{CC}$ tasks of a kernel for load balance. $N_{it} \leqslant N_{\max}$ ensures that the data partition does not exceed the size of on-chip memory. Lines 10 and 16 find a partition size N_1 and N_2 that can be used for all the kernels.

Algorithm 9 Data partitioning algorithm

Input: On-chip memory size S_o; Computation workload of each kernel: $\{Q[k] : 1 \leqslant k \leqslant L\}$; $p()$: function that determines the number of tasks of a kernel based on Q, N_1 and N_2; $g()$: function that determines the maximum partition size based on the on-chip memory size S_o; η: factor for load balance.
Output: Partition size N_1, N_2;
1: $N_{\max} \leftarrow g(S_o)$ ▷ Maximum partition size
2: //Objective: Maximize N_1 and N_2 to improve data locality
3: //Constraint 1 (Maximize utilization): $\mathcal{T}_a$, $\mathcal{T}_u \geq \eta * N_{CC}$
4: //Constraint 2 (Memory capacity): N_1, $N_2 \leqslant N_{\max}$
5: ======= Step 1: determine N_2 ========
6: $N_2 \leftarrow N_{\max}$
7: **for** each Update kernel: k^{th} kernel **do**
8: Choose largest N' such that $\mathcal{T}_u[k] = p(Q[k], N') = \eta * N_{CC}$
9: $N_{it} \leftarrow \min(N', N_{\max})$
10: $N_2 \leftarrow \min(N_{it}, N_2)$
11: ======= Step 2: determine N_1 ========
12: $N_1 \leftarrow N_{\max}$
13: **for** each Aggregate kernel: k^{th} kernel **do**
14: Choose largest N' such that $\mathcal{T}_a[k] = p(Q[k], N', N_2) = \eta * N_{CC}$
15: $N_{it} \leftarrow \min(N', N_{\max})$
16: $N_1 \leftarrow \min(N_{it}, N_1)$

VII. Implementation Details

We implement the proposed accelerator on a state-of-the-art FPGA board – Xilinx Alveo U250, which has four Super Logic Regions (SLR) [21]. As shown in Figure 9, we implement two Computation Cores (CC) in each SLR except for SLR1, because the FPGA shell (which handles the CPU-FPGA communication) and soft processor is placed in SLR1. For each CC, $p_{sys} = 16$. We develop the CC using Verilog HDL, and implement the soft processor using Xilinx Microblaze Soft IP core [22]. Each CC is connected to the soft processor through the AXI4-Stream interface [22], through which the soft processor sends the control signals to CC and the CC sends the sparsity information to the soft processor. We develop the compiler using Python. The IR of a kernel is implemented as a Python object that stores the meta data of a kernel and its execution scheme. We develop the Runtime system on the soft processor using C in Xilinx Vitis Unified Software Platform (version 2020.1). The Index Shuffle Network and Data Shuffle Network are implemented using a butterfly network with buffering to handle the routing congestion. We perform synthesis and Place&Route using Vivado 2020.1. The resource utilization is shown in Figure 9. The CCs run at 250 MHz.



	LUTs	DSPs	BRAMs	URAMs
Soft Processor	5.5K	6	26	0
One CC	118K	1024	96	120
FPGA Shell	181K	13	447	0
Total	1011K	7187	1145	840
Available	1728K	12288	2688	960
Utilization	58.6%	58.4%	42.6%	87.5%

Fig. 9: The layout (FPGA chip) and resource utilization of the proposed design on Xilinx Alveo U250. The Computation Cores (CC0-CC6) are represented using different colors.

Soft processor: Our implementation achieves 370 MHz and around 500 Million Instructions Per Second [22] performance. It has two caches – an Instruction Cache (I-Cache) and a Data Cache (D-Cache). I-Cache has size 32 KB which is sufficient to hold the binary code of the runtime system after a warm-up execution. D-Cache has the size 64 KB which stores the sparsity of the data partitions. For large graphs that D-Cache is not enough to hold the sparsity information of all the data partitions, we store it in the external memory and prefetch the sparsity information to the D-Cache. The soft processor reads/writes the data from/to the AXI-stream interface through the `get` and `put` instructions [22], which have one or two clock cycles latency.

VIII. Evaluation Results

This Section is organized as follows: In Section VIII-B, we measure of the impact of the dynamic K2P mapping strategy. In Section VIII-C, we analyze the overhead of compilation and runtime system. In Section VIII-D, we compare our work with the state-of-the-art implementations.

A. Benchmarks and Baselines

Benchmarks: We evaluate Dynasparse on four widely used GNN models – GCN [10], GraphSAGE (SAGE) [11], GIN [12], and SGC [13]. Figure 10 shows the IR of various GNN layers. We evaluate the design on six widely used graph datasets – Cora (CO) [10], CiteSeer (CI) [10], PubMed (PU) [10], Flickr (FL) [23], NELL (NE) [24], Reddit (RE) [11]. We evaluate the 2-layer GNN models used in [10], [17], [3], [4], where the hidden dimension for CO, CI and PU is set as 16, and the hidden dimension for FL, NE and RE is set as 128.

Baselines: We compare our work with the state-of-the-art CPU (AMD Ryzen 3990x), GPU (Nvidia RTX3090) and GNN accelerators HyGCN [3], BoostGCN [4]. The details of the platforms are shown in Table V.

Performance metric: Following the convention in [17], [3], [4], we use *latency* (accelerator execution latency) as the metric which is the duration from the time when the accelerator starts to execute the optimized IR to the time all the inference results are obtained. The preprocessing time by the compiler

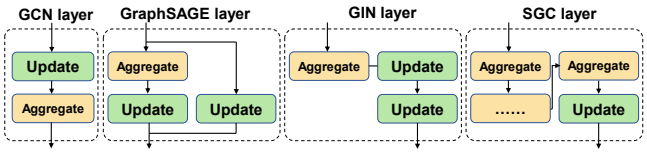


Fig. 10: The IR of various GNN layers

TABLE V: Specifications of platforms

	CPU	GPU	[3]	[4]	Dynasparse
Platform	Ryzen 3990x	Nvidia RTX3090	ASIC	Stratix 10 GX	Alveo U250
Technology	TSMC 7 nm	TSMC 7nm	TSMC 12 nm	Intel 14 nm	TSMC 16 nm
Frequency	2.90 GHz	1.7 GHz	1 GHz	250 MHz	250 MHz
Peak Performance (TFLOPS)	3.7	36	4.608	0.64	0.512
On-chip Memory	256 MB	6 MB	35.8 MB	32 MB	45 MB
Memory Bandwidth	107 GB/s	936.2 GB/s	256 GB/s	77 GB/s	77 GB/s

is not included in the latency, because (1) the overhead of generating the optimized IR is usually small (See Section VIII-C) and the optimized IR can be stored and reused if the sparsity of the input graph and GNN model changes, (2) we follow the same convention in [17], [3], [4] for a fair comparison.

B. Impact of Dynamic K2P Mapping Strategy

To demonstrate the impact of the proposed dynamic K2P mapping strategy, we execute the following three K2P mapping strategies on our proposed accelerator:

- `Static-1` (`S1`): It is used in [3], [4] that Aggregate() is mapped to SpMM and Update() is mapped to GEMM.
- `Static-2` (`S2`): It is used in [17] that both the Aggregate() and Update() are mapped to SpDMM. For Aggregate($\boldsymbol{A}$, $\boldsymbol{H}$), it views $\boldsymbol{A}$ as sparse matrix and views $\boldsymbol{H}$ as dense matrix. For Update($\boldsymbol{H}$, $\boldsymbol{W}$), it views $\boldsymbol{H}$ as sparse matrix and views $\boldsymbol{W}$ as dense matrix.
- `Dynamic`: It is our proposed dynamic K2P mapping strategy (Algorithm 7).

We use SO-S1 to denote the speedup of `Dynamic` over `S1`. We use SO-S2 to denote the speedup of `Dynamic` over `S2`.

Evaluation on unpruned GNN models: We evaluate the above three strategies using unpruned GNN models where all the weight matrices have density 100%. The results are shown in Table VII. Compared with `S1` and `S2`, `Dynamic` achieves 2.13× and 1.59× speedup on the average (geometric mean), respectively. `Dynamic` achieves limited speedup over `S2` on GCN because (1) for the first Update($\boldsymbol{H}^0$, $\boldsymbol{W}^1$) kernel of GCN, there is high data sparsity in $\boldsymbol{H}^0$ of CI, CO, PU and NE (See Table VI), (2) both `Dynamic` and `S2` can exploit the sparsity of feature matrix $\boldsymbol{H}^0$ while `S1` does not exploit the sparsity of $\boldsymbol{H}^0$. As the first Update($\boldsymbol{H}^0$, $\boldsymbol{W}^1$) kernel of GCN consumes majority of the execution time, `Dynamic` achieves very large speedup over `S1` on GCN. Since the weight matrices have density 100%, both `Dynamic` and `S2` map Update($\boldsymbol{H}^0$, $\boldsymbol{W}^1$) to SpDMM (for CI, CO, PU and NE), leading to similar performance of `Dynamic` and `S2` on GCN.

TABLE VI: Dataset Statistics

Dataset	Vertices	Edges	Features	Classes	Density of $\boldsymbol{A}$	Density of $\boldsymbol{H}^0$
CI	3327	4732	3703	6	0.08%	0.85%
CO	2708	5429	1433	7	0.14%	1.27%
PU	19717	44338	500	3	0.02%	10.0%
FL	89,250	899,756	500	7	0.01%	46.4%
NE	65,755	251,550	61,278	186	0.0058%	0.01%
RE	232,965	11×10^7	602	41	0.21%	100.0%

TABLE VII: The latency (ms) on the unpruned GNN models

		CI	CO	PU	FL	NE	RE
GCN [10]	`S1`	31E-1	9.6E-1	2.7E-1	10E0	83E2	9.3E1
	`S2`	8.9E-3	5.6E-3	7.1E-3	9.9E0	5.4E0	12E1
	`Dynamic`	7.7E-3	4.7E-3	6.3E-2	8.8E0	2.9E0	8.4E1
	SO-S1	41.3×	21.5×	4.29×	1.13×	278×	1.10×
	SO-S2	1.15×	1.19×	1.12×	1.11×	1.82×	1.42×
SAGE [11]	`S1`	74E-2	25E-2	65E-2	20E0	17E2	334E0
	`S2`	75E-2	25E-2	69E-2	28E0	17E2	389E0
	`Dynamic`	33E-2	11E-2	42E-2	19E0	83E1	331E0
	SO-S1	1.93×	1.72×	1.56×	1.02×	2.05×	1.01×
	SO-S2	1.94×	1.73×	1.65×	1.41×	2.05×	1.17×
GIN [12]	`S1`	4.3E-1	1.5E-1	4.1E-1	1.3E1	8.8E2	3.1E2
	`S2`	7.4E-1	2.4E-1	6.5E-1	2.0E1	1.7E3	3.4E2
	`Dynamic`	3.3E-1	1.1E-1	3.7E-1	1.2E1	8.3E2	2.7E2
	SO-S1	1.30×	1.40×	1.11×	1.13×	1.06×	1.15×
	SO-S2	2.26×	2.31×	1.76×	1.73×	2.05×	1.25×
SGC [13]	`S1`	5.3E-1	2.0E-1	5.5E-1	1.29E-1	9.33E2	5.7E2
	`S2`	8.5E-1	3.0E-1	7.9E-1	2.18E-1	1.77E3	6.0E2
	`Dynamic`	4.3E-1	1.5E-1	5.1E-1	1.27E-1	8.83E2	5.0E2
	SO-S1	1.23×	1.27×	1.08×	1.02×	1.06×	1.13×
	SO-S2	1.95×	1.91×	1.55×	1.72×	1.99×	1.19×

Evaluation on pruned GNN models: We evaluate the three strategies using the pruned GNN models [15] where the weight matrices are pruned to have various sparsity. Figures 11 and 12 show the speedup of `Dynamic` over `S1&2`. For evaluation, all the weight matrices in a GNN model are pruned to have the same sparsity, and the sparsity of weights in Figures 11&12 means the average sparsity of all the weight matrices in a GNN model. Table VIII summarizes the average (geometric mean) speedup under various sparsity of weight matrices. The achieved speedup over `S1` is because `S1` cannot exploit the data sparsity in feature matrices and weight matrices. The achieved speedup over `S2` is due to (1) when there is limited data sparsity (density < 50%) in Update(), executing Update() using SpDMM primitive is not efficient. (2) In Aggregate(), `S2` does not exploit data sparsity in feature matrix $\boldsymbol{H}$ since `S2` views $\boldsymbol{H}$ as a dense matrix.

TABLE VIII: Average speedup (geometric mean)

Sparsity of weight matrices	< 50%	50% – 70%	70% – 90%	> 90%
SO-S1	2.16×	4.36×	10.77×	15.96×
SO-S2	1.38×	1.64×	2.11×	5.03×

In conclusion, the proposed dynamic K2P mapping strategy leads to lower accelerator execution latency compared with the static mapping strategies. Using dynamic K2P mapping strategy, the execution latency reduces as the data sparsity increases.

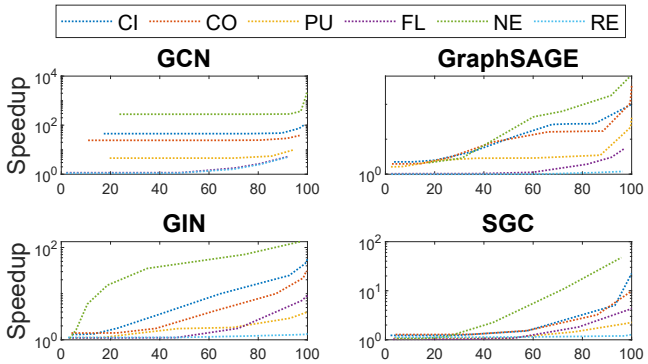


Fig. 11: Speedup of `Dynamic` over `S1` when there are various sparsity (%) in the GNN weight matrices (X-axis)

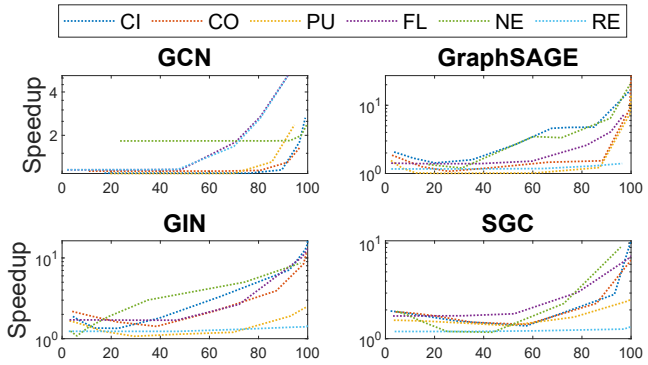


Fig. 12: Speedup of `Dynamic` over `S2` when there are various sparsity (%) in the GNN weight matrices (X-axis)

C. Analysis of Compiler and Runtime System

TABLE IX: The preprocessing time of the compiler (ms)

	CI	CO	PU	FL	NE	RE
GCN	2.5E-1	2.2E-2	5.7E-1	2.68E0	1.70E0	5.1E1
GraphSAGE	2.3E-1	2.6E-1	5.9E-1	2.58E0	1.65E0	4.9E1
GIN	2.4E-1	2.6E-3	5.8E-1	2.69E0	1.71E0	5.0E1
SGC	2.3E-1	2.4E-3	6.1E-1	2.74E0	1.73E0	5.2E1

Overhead of the compilation/preprocessing: Table IX shows the overhead of the compiler on the host processor (Intel Xeon 5120). The processing time includes the overheads of generating IR, data partitioning, and preprocessing of data sparsity. Compared with design automation framework [18] which needs to regenerate FPGA accelerator if the graph or GNN model changes, the overhead of the compiler in our design is small.

Overhead of the Runtime System: We measure the overhead of the runtime system, which is the execution time of dynamic K2P mapping on the soft processor. See Figure 13. on the average, the Runtime System takes 6.8% of the total execution time and is hidden by the task scheduling (Section VI-C). For the pruned GNN models, as the densities of weight matrices decrease, the overhead of the Runtime System will decrease since there will be more empty data partitions skipped by the runtime system (Algorithm 7).

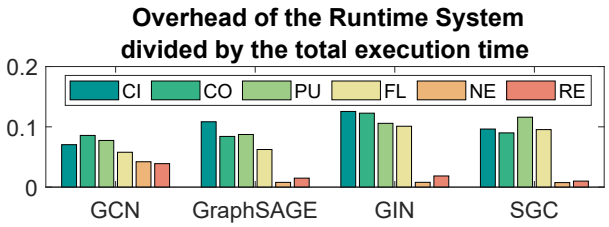


Fig. 13: Overhead of runtime system on unpruned GNNs

D. Comparison with the State-of-the-art

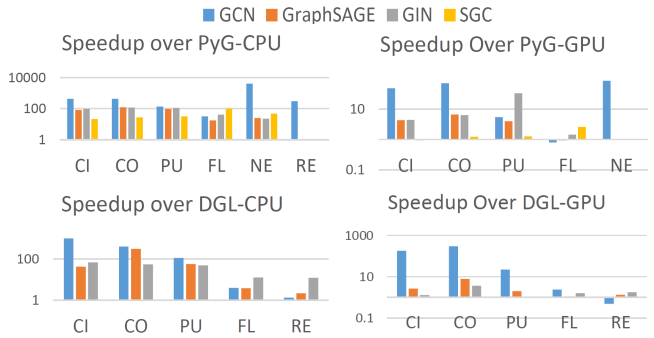


Fig. 14: Speedup over the CPU and GPU platforms (Some results are not shown due to out of memory on CPU/GPU)

Comparison with CPU/GPU: We execute the state-of-the-art GNN frameworks – Pytorch Geometric (PyG, version 1.11.0) and Deep Graph library (DGL, version 0.8.0post2) on CPU and GPU platforms (Table V). The evaluation results are shown in Figure 14. We execute the same unpruned GNN models on CPU, GPU and Dynasparse for a fair comparison. Dynasparse achieves 306×, 16.4×, 141.9× and 35× speedup compared with PyG-CPU, PyG-GPU, DGL-CPU and DGL-GPU, respectively. Note the CPU and GPU have 7.2× and 70× higher peak performance than Dynasparse. The achieved speedup is because Dynasparse can efficiently exploit the data sparsity in graph structure, vertex features and weight matrices. In contrast, PyG and DGL on CPU and GPU only exploit the sparsity in the graph structure. Moreover, Dynasparse exploits FPGA-specific optimizations: (1) customized data-path with Index/Data Shuffle Networks to handle the irregular memory access pattern of GNNs, (2) customized on-chip memory management for exploiting data locality, (3) dedicated hardware modules for sparsity profiling and layout/format transformation; The proposed double buffering hides their overheads, and (4) lightweight soft processor interacting with Computation Cores with extreme low latency for dynamic kernel-to-primitive mapping.

Comparison with GNN accelerators: Table X shows the comparison of the latency with the state-of-the-art GNN accelerators, which do not require regenerating accelerator if the data sparsity changes. All the accelerators execute the

TABLE X: Comparison of latency with the state-of-the-art GNN accelerators (using GCN model)

	CI	CO	PU	FL	NE	RE	Peak Perf. (TFLOPS)
BoostGCN [4]	1.9E-2	2.5E-2	1.6E-1	4.0E1	N/A⋆	1.9E2	1.35
HyGCN [3]	2.1E-2	3E-1	6.4E1	N/A	N/A	2.9E2	4.6
Dynasparse	7.7E-3	4.7E-3	6.3E-2	8.8E0	2.9E0	1.0E2	0.512

⋆ N/A: not available.

same unpruned GCN models and graph datasets. Dynasparse achieves 2.7×, 171× speedup on the average than BoostGCN and HyGCN, respectively. The platforms used in BoostGCN and HyGCN have 1.25×, 9× higher peak performance than Dynasparse. The achieved speedup is because Dynasparse can efficiently exploit data sparsity in vertex features. We expect to achieve higher speedup when executing the same pruned GNN models, since [4], [3] do not exploit the sparsity in weights.

Discussion of preprocessing and data communication overheads: We define the *end-to-end latency* as the sum of **(1)** the overhead of compilation/preprocessing (Section VIII-C), **(2)** the overhead of CPU-FPGA data movement (moving the processed input graph, processed GNN model, and optimized IR from the host memory to FPGA external memory), and **(3)** execution latency of the accelerator. With respect to end-to-end latency, Dynasparse still achieves 56.9×, 2.37×, 16.3×, 1.37× speedup on the unpruned GNN models compared with PyG-CPU, PyG-GPU, DGL-CPU, and DGL-GPU, respectively. The preprocessing overhead, data movement overhead, and execution latency contribute to 43.1%, 27.2%, 27.6% of the total end-to-end latency on the average. The major overhead in preprocessing is data partitioning that reorganizes the input data into data partitions. It can be reduced by multi-threading and increasing the host memory bandwidth.

Note that the CPU-FPGA data movement overhead depends on the PCIe bandwidth. The sustained PCIe bandwidth of the Alveo U250 FPGA board is around 11.2 GB/s while the baseline GPU (Nvidia RTX3090) has PCIe bandwidth of 31.5 GB/s. The overhead of CPU-FPGA data movement can be reduced by exploiting state-of-the-art CPU-FPGA interconnection techniques (offered by FPGA vendors), such as PCIe 5.0. Since prior GNN accelerators [4], [3] do not include their preprocessing overheads (data partitioning and CPU-FPGA data movement), in Table X, we only compare the accelerator execution latency with [4], [3] for a fair comparison.

IX. Conclusion and Future Work

In this paper, we proposed a hardware-software codesign for dynamic sparsity exploitation in GNN inference. The proposed dynamic K2P mapping reduces the inference latency by 3.73× on the average compared with the static mapping strategies. Compared with state-of-the-art CPU (GPU) implementations, Dynasparse achieves up to 56.9× (2.37×) speedup in end-to-end latency. Compared with state-of-the-art FPGA implementations, Dynasparse achieves 2.7× speedup in accelerator execution latency. In the future, we plan to extend Dynasparse on heterogeneous platforms that consist of CPU, GPU and FPGA, where GPU is effective for dense primitives, FPGA is effective for sparse primitives and the CPU can execute complex control flow (e.g., dynamic K2P mapping).

Acknowledgment

This work is supported by the National Science Foundation (NSF) under grants CCF-1919289 and OAC-2209563. Equipment and support by Xilinx are greatly appreciated.

References

[1] W. Zhong and P. Balaprakash, "Explainable graph pyramid autoformer for long-term traffic forecasting," *arXiv preprint arXiv:2209.13123*.
[2] J. Hewes, "Graph neural network for object reconstruction in liquid argon time projection chambers," in *EPJ Web of Conferences*.
[3] M. Yan and L. Deng, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020.
[4] B. Zhang, R. Kannan, and V. Prasanna, "Boostgcn: A framework for optimizing gcn inference on fpga," in *2021 FCCM*. IEEE.
[5] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale gcn inference," in *2020 IEEE ASAP*, pp. 61–68.
[6] Y.-C. Lin, B. Zhang, and V. Prasanna, "Hp-gnn: generating high throughput gnn training implementation on cpu-fpga heterogeneous platform," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 123–133.
[7] Y. Meng, S. Kuppannagari, R. Kannan, and V. Prasanna, "Dynamap: Dynamic algorithm mapping framework for low latency cnn inference," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 183–193.
[8] B. Zhang, H. Zeng, and V. Prasanna, "Graphagile: An fpga-based overlay accelerator for low-latency gnn inference," *arXiv preprint arXiv:2302.01769*, 2023.
[9] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "Flowgnn: A dataflow architecture for universal graph neural network inference via multi-queue streaming," *arXiv preprint arXiv:2204.13103*, 2022.
[10] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
[11] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st NeurIPS*.
[12] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.
[13] F. Wu and A. Souza, "Simplifying graph convolutional networks," in *International conference on machine learning*. PMLR, 2019.
[14] "graph datasets." [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html
[15] M. Rahman, A. Azad *et al.*, "Triple sparsification of graph convolutional networks without sacrificing the accuracy."
[16] T. Chen and Y. Sui, "A unified lottery ticket hypothesis for graph neural networks," in *ICML*, 2021.
[17] T. Geng and A. Li, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 IEEE/ACM MICRO*, 2020.
[18] S. Liang, "Deepburning-gl: an automated framework for generating graph neural network accelerators," in *2020 IEEE/ACM ICCAD*.
[19] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *2015 ACM/SIGDA FPGA*.
[20] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "Gpop: A scalable cache-and memory-efficient framework for graph processing over parts," *ACM Transactions on Parallel Computing (TOPC)*, 2020.
[21] "Xilinx alveo u250." [Online]. Available: https://docs.xilinx.com/r/en-US/ds962-u200-u250/FPGA-Resource-Information
[22] "Microblaze." [Online]. Available: https://docs.xilinx.com/v/u/2021.1-English/ug984-vivado-microblaze-ref
[23] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *International Conference on Learning Representations*.
[24] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting semi-supervised learning with graph embeddings," in *International conference on machine learning*. PMLR, 2016, pp. 40–48.