Accelerating GNN-based SAR Automatic Target Recognition on HBM-enabled FPGA

Bingyi Zhang*, Rajgopal Kannan†, Viktor Prasanna*, Carl Busart†
*University of Southern California †DEVCOM US Army Research Lab
*{bingyizh, prasanna}@usc.edu †{rajgopal.kannan.civ, carl.e.busart.civ}@army.mil

***Abstract*—Synthetic Aperture Radar (SAR) automatic target recognition (ATR) is a key technique for remote-sensing image recognition. In real-world applications, massive SAR images are captured by airplanes or satellites, requiring high-throughput and low-latency processing. Recently, Graph Neural Networks (GNNs) have shown superior performance for SAR ATR in terms of accuracy and computational complexity. In this paper, we accelerate GNN-based SAR ATR on an FPGA. In the proposed design, we develop a customized data path and memory organization to execute various computation kernels of GNNs, including feature aggregation and feature transformation. We exploit the high bandwidth memory (HBM) of the FPGA to speed up data loading and store intermediate results. We employ the splitting kernel technique to improve the routability and frequency of the design on FPGA. We implement the proposed design using High-level Synthesis (HLS) on a state-of-the-art data-center FPGA board, the AMD/Xilinx Alveo U280. Compared with implementations on state-of-the-art CPUs (GPUs), our FPGA implementation achieves a $5.2\times$ ($1.57\times$) lower latency, a $10\times$ ($3.3\times$) higher throughput, and is $36.2\times$ ($7.35\times$) more energy efficient.**

***Index Terms*—Synthetic Aperture Radar, Automatic Target Recognition, Graph Neural Network, High Bandwidth Memory**

I. Introduction

Synthetic Aperture Radar (SAR) is capable of acquiring remote-sensing images in all-weather conditions, allowing for observations of targets on the Earth's surface. SAR has found many applications in various fields, such as agriculture [1], [2] and civilization [3], [4]. SAR automatic target recognition (ATR) is a key technique for classifying targets in SAR images. Figure 1 illustrates the end-to-end workflow of SAR ATR in real-world applications. It comprises four steps: (1) *data acquisition*: SAR raw data is acquired from satellite-based or airborne SAR sensors, which emit microwave signals towards the target area and record the backscattered signals. (2) *preprocessing*: The acquired SAR raw data is preprocessed to generate SAR images and reduce noise. This step may involve radiometric calibration, geometric correction, speckle filtering, and more. (3) *target recognition*: SAR images are processed by machine learning (ML) models, such as Convolutional Neural Networks (CNNs) or Graph Neural Networks (GNNs), for target recognition. (4) *decision making*: The classification results are sent to the decision maker for analysis. Among the four steps, the preprocessing and target recognition steps are typically executed on a data center. Additionally, the target recognition step can become a performance bottleneck due to the high computational complexity of ML models. Many applications, including defense and military scenarios, require high-throughput and low-latency SAR ATR for real-time decision making. Therefore, achieving high performance SAR ATR on the data center is crucial.

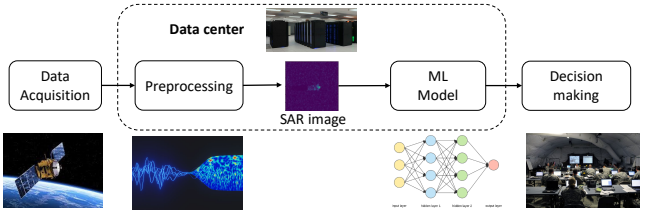

Fig. 1: SAR automatic target recognition (ATR) in data center

Graph Neural Networks (GNNs) [5], [6] have recently been developed for SAR automatic target recognition (ATR). When compared with state-of-the-art CNNs [7]–[11] for SAR ATR, GNN-based approaches [5], [6] achieve comparable classification accuracy while exhibiting lower computational complexity. This reduced complexity of GNN-based approaches holds the potential for high-performance and energy-efficient SAR ATR.

In this paper, we focus on accelerating GNN-based SAR ATR in a data center environment. However, GNN-based approaches involve graph message passing, which exhibits irregular computation and memory access patterns. Executing these approaches on general-purpose processors (e.g., CPUs, GPGPUs) in data centers, which typically have complex cache hierarchies, is not efficient. We use a data-center FPGA (e.g., AMD/Xilinx Alveo U280) as our target platform. Utilizing HBM-enabled FPGAs offers several benefits: (1) Data-center FPGAs provide ample programmable computation resources and on-chip resources, enabling significant computation parallelism. (2) The programmability of FPGAs allows us to design customized data paths and on-chip memory organizations that efficiently handle the irregular computation pattern and memory access pattern of GNNs. (3) HBM-enabled FPGAs are equipped with High Bandwidth Memory (HBM), which provides substantial external memory bandwidth. This is particularly advantageous for GNNs, which often exhibit poor data locality in this computations. Our contributions can be summarized as follows:

- We accelerate GNN-based SAR ATR on an HBM-enabled data-center FPGA.

- We develop a customized data path and memory organization to efficiently execute various computation kernels of GNN for low-latency inference.
- We exploit the High Bandwidth Memory (HBM) of the FPGA to speed up data loading and storing intermediate results.
- We utilize the splitting kernel technique to improve the routability and frequency of the design.
- We implement our design on a state-of-the-art FPGA board, AMD/Xilinx Alveo U280. Compared with implementations on state-of-the-art CPUs (GPUs), our FPGA implementation achieves 5.2× (1.57×) lower latency, 10× (3.3×) higher throughput, and is 36.2× (7.35×) more energy efficient.

II. Background

A. GNN-based SAR Automatic Target Recognition

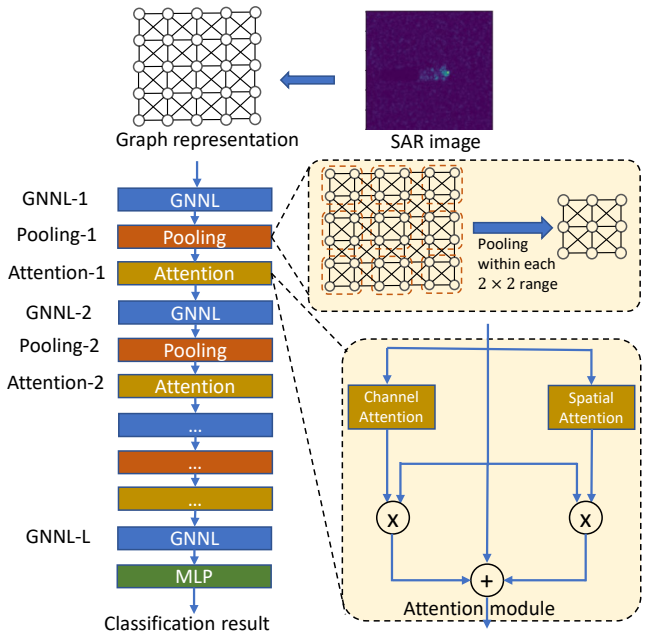

Fig. 2: The graph construction approach and GNN model used in [6]

Recently, Graph Neural Networks (GNNs) [5], [6] have demonstrated superior performance in SAR automatic target recognition, in terms of accuracy and computational complexity. In [6], the authors propose constructing the input graph from the SAR image based on the order of the pixel grayscale values (See Figure 7 and Figure 8 of [6] for details). The constructed graph is then fed into a graph neural network for classification. However, the graph construction approach in [6] disregards the structure of the target and is sensitive to variations in pixel values. In contrast, in [5], the authors propose transforming the input SAR image into a 2-D mesh graph, as shown in Figure 2. Each pixel in the original SAR image is represented as a vertex, and edges are formed by connecting each pixel to its eight neighboring pixels. The resulting graph is then input into the GNN model for classification. Compared with [6], the graph construction approach in [5] preserves the structure of the target and exhibits greater robustness to variations in pixel values. Therefore, we choose to accelerate the GNN model proposed in [5].

The GNN model in [5] consists of three types of layers: the GNN layer (GNNL), the Graph pooling layer (Pooling), and the Attention layer (Attention). We represent the input graph as $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{H}^0)$, where $\mathcal{V}$ denotes the set of vertices, $\mathcal{E}$ denotes the set of edges, and $\boldsymbol{H}^0$ denotes the input feature matrix. The i^{th} row of $\boldsymbol{H}^0$ (denoted as $\boldsymbol{h}_i^0$) represents the input feature vector of vertex i ($i \in \mathcal{V}$).

GNN layer: The GNN layer is the GraphSAGE layer [12] that follows the aggregate-update paradigm:

$$\begin{aligned} &\text{aggregate:} \boldsymbol{z}_i^l = \text{Mean}\left(\boldsymbol{h}_j^{l-1} : j \in \mathcal{N}(i) \cup \{i\}\right) \\ &\text{update:} \boldsymbol{h}_i^l = \text{ReLU}\left(\boldsymbol{z}_i^l \boldsymbol{W}_{\text{neighbor}}^l + \boldsymbol{h}_i^{l-1} \boldsymbol{W}_{\text{self}}^l\right) \end{aligned} \tag{1}$$

where $\boldsymbol{h}_j^{l-1}$ denotes the feature vector of vertex j at layer $l-1$, $\mathcal{N}(i)$ denotes the set of neighbors of vertex i, $\boldsymbol{W}_{\text{neighbor}}^l$ and $\boldsymbol{W}_{\text{self}}^l$ are the weight matrices. In *aggregate* phase, each vertex aggregates vertex features from the neighbors, and the aggregated feature vectors are reduced using the element-wise Mean() function. In *update* phase, the feature vectors ($\boldsymbol{z}_i^l$,) of each vertex are transformed by the weight matrices ($\boldsymbol{W}_{\text{neighbor}}^l$ and $\boldsymbol{W}_{\text{self}}^l$) to generate the updated feature vector $\boldsymbol{h}_i^l$. The update phase has two parts – neighbor update $\boldsymbol{z}_i^l \boldsymbol{W}_{\text{neighbor}}^l$ and self update $\boldsymbol{h}_i^{l-1} \boldsymbol{W}_{\text{self}}^l$.

Graph pooling layer: It downscales the input graph $\mathcal{V}_{l-1}$ into a smaller output graph $\mathcal{V}_l$. The pooling operaton is similar to the pooling in the 2-D images:

$$\begin{aligned} \boldsymbol{h}_{p(i,j)}^l = \max(&\boldsymbol{h}_{p(2i,2j)}^{l-1}, \boldsymbol{h}_{p(2i+1,2j)}^{l-1}, \\ &\boldsymbol{h}_{p(2i,2j+1)}^{l-1}, \boldsymbol{h}_{p(2i+1,2j+1)}^{l-1}) \end{aligned} \tag{2}$$

where $v_{i,j}^l \in \mathcal{V}_l$, $v_{p(2i,2j)}^{l-1}, v_{p(2i+1,2j)}^{l-1}, v_{p(2i,2j+1)}^{l-1}, v_{p(2i+1,2j+1)}^{l-1} \in \mathcal{V}_{l-1}$, and $p(i,j)$ denotes the index of the vertex locating at the coordinate (i,j) in the 2-D mesh graph.

Attention layer: The attention layer consists of *feature attention* that calculates the attention scores for each vertex feature, and *vertex attention* that calculates the attention scores for each vertex. The feature attention is calculated by:

$$\begin{aligned} \boldsymbol{F}_{\text{fa}} = \text{sigmoid}(&\text{mean}(\{\boldsymbol{h}_{i,j} : v_{i,j} \in \mathcal{V}\}) \boldsymbol{W}_{\text{fa}}^{\text{mean}} + \\ &\text{sum}(\{\boldsymbol{h}_{i,j} : v_{i,j} \in \mathcal{V}\}) \boldsymbol{W}_{\text{fa}}^{\text{sum}}) \end{aligned} \tag{3}$$

where $\boldsymbol{h}_{i,j}, \boldsymbol{F}_{\text{fa}} \in \mathbb{R}^c$, $\boldsymbol{W}_{\text{fa}}^{\text{mean}}, \boldsymbol{W}_{\text{fa}}^{\text{sum}} \in \mathbb{R}^{c \times c}$, and c denotes the length of feature vector. fa$[i]$ is the attention score for i^{th} feature. The vertex attention score is calculated using a GNN layer:

$$\{\alpha_{i,j} : v_{i,j} \in \mathcal{V}_l\} = \text{sigmoid}(\text{GNNL}(\{\boldsymbol{h}_{i,j} : v_{i,j} \in \mathcal{V}_{l-1}\})), \tag{4}$$

Where $\alpha_{i,j}$ is the attention score for vertex $v_{i,j}$. Then, the output of the attention layer is calculated by:

$$\{\boldsymbol{h}_{i,j}^{\text{out}} : \boldsymbol{h}_{i,j}^{\text{out}} = (1 + \alpha_{i,j}) \boldsymbol{h}_{i,j}^{\text{in}} + \boldsymbol{h}_{i,j}^{\text{in}} \otimes \boldsymbol{F}_{\text{fa}}\} \tag{5}$$

where $\otimes$ is element-wise multiplication.

B. HBM-enabled FPGA

Data-center Field Programmable Gate Arrays (FPGAs) have emerged as promising platforms widely used in data centers, such as Amazon AWS [13] and Intel Developer Cloud [14]. Data-center FPGAs offer a plethora of programmable resources, including Look-up Tables (LUTs), Digital Signal Processing (DSP) units, Block RAMs (BRAMs), and more. These abundant computational resources enable massive parallelism for computationally intensive workloads, such as machine learning inference [15]–[35]. Additionally, FPGA vendors, such as AMD/Xilinx and Intel, have integrated High Bandwidth Memory (HBM) into their FPGA chips, such as Xilinx Alveo U280 [36] and Intel Stratix 10 MX FPGA [37]. The diagram of the HBM in Xilinx Alveo U280 [36] is shown in Figure 3. HBM comprises multiple parallel and independent Pseudo Channels (PCs), with memory controllers (MCs) directly interacting with these Pseudo Channels for data read/write operations. For every four PCs, there is a crossbar switch network (SW) of size 4×4, enabling all-to-all data communication between the four AXI channels [38] and four PCs. The parallel PCs of HBM provide massive memory bandwidth (up to 460 GB/s with 32 HBM PCs) and substantial memory capacity (up to 8GB with 32 HBM PCs). Consequently, HBM holds great potential for accelerating memory-bound applications, such as GNN inference.

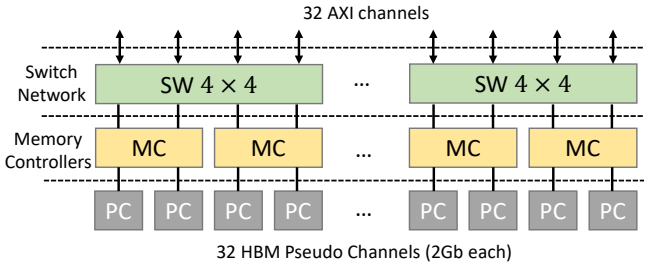

Fig. 3: High Bandwidth Memroy (HBM) in AMD/Xilinx Alveo U280.

III. Overview

We accelerate the GNN-based SAR ATR approach proposed in [5]. The structure of the GNN model is introduced in Section II-A.

Figure 4 provides an overview of the proposed FPGA accelerator. The accelerator consists of multiple parallel processing elements (PEs). Each PE includes a Feature Aggregation (FA) Module, a Feature Update (FU) Module, and a Multi-layer Perceptron (MLP) Module. Each PE is capable of executing the inference of a SAR image. During runtime, the SAR images are transferred from the host processor to the High Bandwidth Memory (HBM) via PCIe interconnection. Subsequently, the PEs perform the inference of the SAR images, and the inference results are sent back to the host processor through PCIe. The algorithm for executing SAR ATR using multiple PEs is shown in Algorithm 1. When a PE is idle, a SAR image is assigned to this PE for inference until the predicted label for this SAR image generated. It enables parallel processing of multiple SAR images.

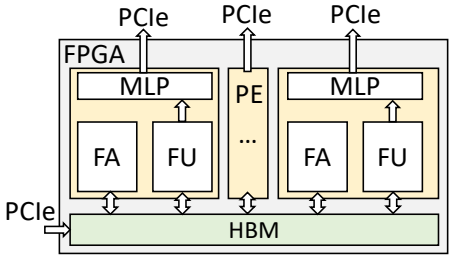

Fig. 4: Proposed accelerator architecture.

Algorithm 1 Inference using multiple PEs

Input: A set of input SAR images $\mathcal{X}$;
Output: The predicted labels for each SAR image in $\mathcal{X}$;
1: **for** each SAR image $\boldsymbol{X} \in \mathcal{X}$ **Parallel do**
2: **if** there is an idle PE: PE_i **then**
3: PE_i executes the inference for image $\boldsymbol{X}$

IV. Architecture Design

Figure 5 illustrates the components of a Processing Element (PE), which includes a Feature Aggregation (FA) module, a Feature Update module, and an MLP module.

A. Feature Aggregation Module

The Feature Aggregation (FA) module executes the aggregate phase (Section II-A) of the GNN layer and the graph pooling layer. The FA module comprises a Feature Buffer, an Edge Buffer, a Result Buffer, a Shuffling Network, and multiple computation pipelines. The Feature Buffer and the Edge Buffer are responsible for storing the input feature matrix and edges, respectively.

Algorithm 2 Scatter-Gather paradigm

1: **while** not done **do**
2: ======= Shuffling Network ========
3: **for** each edge $e\langle src, dst, weight\rangle$ **do**
4: Fetch $src.fvector$ from Edge Buffer
5: Send $(src.vector, e.weight)$ to pipeline $dst\%p$
6: ======= Pipelines ========
7: **for** each $(src.vector, e.weight)$ **do**
8: Produce update $u \leftarrow$Scatter$(src.vector, e.weight)$
9: Update vertex $v_{dst} \leftarrow$ Gather$(u.vector)$

Executing the aggregate phase: The FA module executes the aggregate phase using the Scatter-Gather paradigm [39], as shown in Algorithm 2. Suppose there are p parallel pipelines. Each pipeline has q multipliers for executing the Scatter() function and q accumulators for executing the Gather() function. The execution of the aggregate phase is edge-centric [39]. For each edge denoted as a three-tuple $e\langle src, dst, weight\rangle$, the src index is routed to the Feature Buffer to fetch the

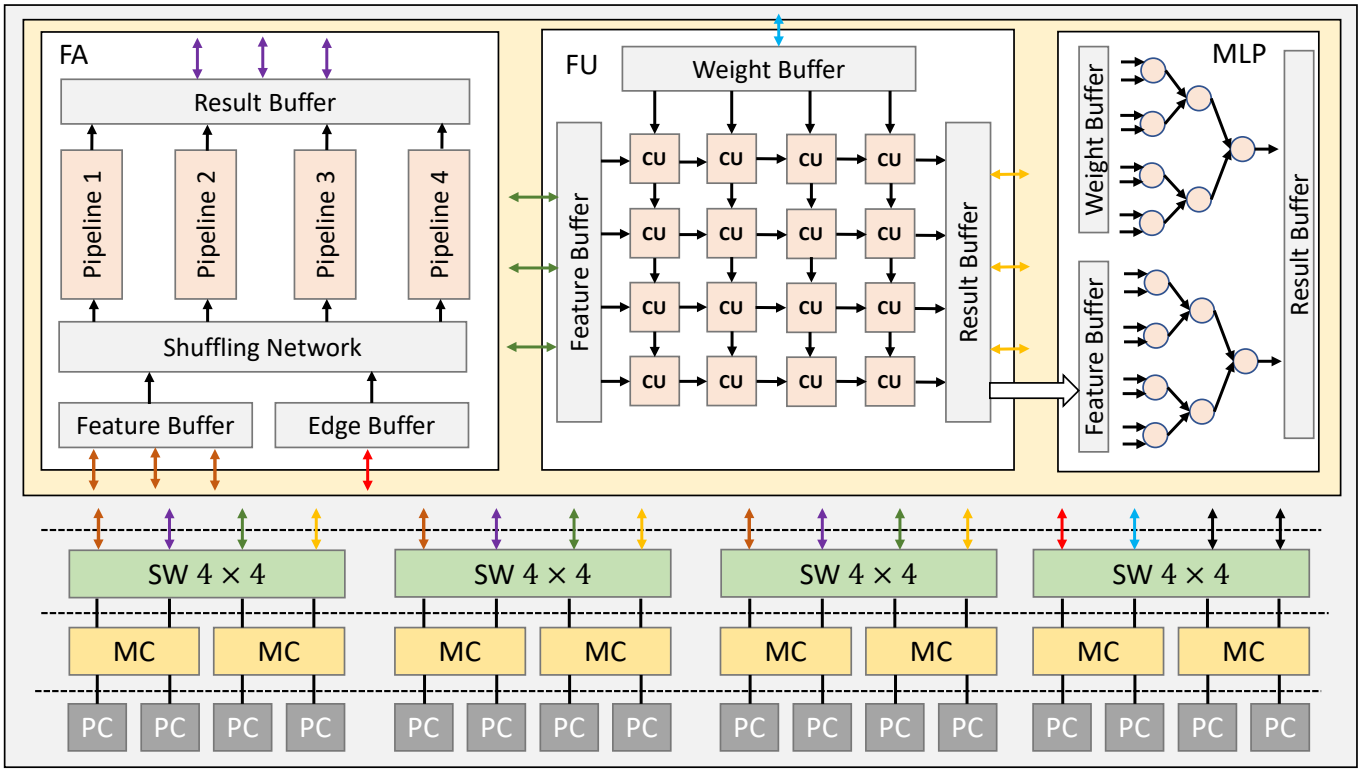

Fig. 5: Architecture of the Processing Element (PE).

feature vector of vertex src denoted as $src.vector$. Then, the input pair $(src.vector, e.weight)$ is sent to pipeline $dst\%p$ for processing. The Shuffling Network can handle the irregular memory access of the aggregate phase. p pipelines can execute pq multiply-accumulate (MAC) operations per clock cycle.

Executing the graph pooling layer: The Graph pooling layer can also be executed using the Scatter-Gather paradigm. The calculation of Equation 2 can be represented as a message-passing process where vertices $v_{p(2i+1,2j)}$, $v_{p(2i,2j+1)}$, and $v_{p(2i+1,2j+1)}$ propagate their feature vectors to $v_{p(2i,2j)}$. Then, $v_{p(2i,2j)}$ reduces the feature vectors using element-wise Max() as the Gather() function (See Algorithm 2).

B. Feature Update Module

The Feature Update (FU) module executes the *update* phase of the GNN layer. FU consists of a Feature Buffer, a Weight Buffer, a Result Buffer, and a 2-D array of Computation Units (CUs). Each CU serves as a multiply-accumulate unit. The 2-D array is organized as a 2-D systolic array to perform the multiplication of the feature matrix $\boldsymbol{H}$ and the weight matrix $\boldsymbol{W}$. Suppose the 2-D CU array has a size of $m \times m$. The 2-D CU array can execute m^2 multiply-accumulate operations per clock cycle.

C. MLP Module

The Multi-layer Perceptron module executes the final MLP of the model (See Figure 2). The primary computation in the last MLP involves multiplying a weight matrix $\boldsymbol{W}$ with a vector. To perform this operation efficiently, we employ an adder tree-based design. Suppose there are s_1 $(s_1 > 0)$ adder trees, and each adder tree has s_2 $(s_2 > 0)$ input ports. Therefore, each adder tree can perform s_2 multiply-accumulate operations per cycle, and with s_1 adder trees, a total of $s_1 \times s_2$ multiply-accumulate operations can be executed per cycle.

V. Optimizations

We introduce optimizations for increasing the scalability of the design and optimizing the inference latency.

A. Exploiting High-Bandwidth Memory

In the proposed design, High-bandwidth Memory (HBM) is utilized to store the input SAR image and the intermediate results of each layer. HBM is chosen for storing the intermediate results due to the following reasons: (1) FPGA on-chip memory has limited capacity (e.g., AMD/Xilinx Alveo U280 FPGA board only provides 41 MB on-chip memory). Utilizing on-chip memory to store all the intermediate results imposes restrictions on the size of SAR image that the accelerator can process. On the other hand, HBM offers a significantly larger capacity (e.g., up to 8 GB) compared to on-chip memory. (2) HBM provides a considerably higher memory bandwidth (460 GB/s) compared to DDR memory (19-77 GB/s) , which is sufficient to supply input data to the computation units within the hardware modules. (3) All the data loading operations from HBM to the hardware modules follow a sequential data access pattern. This sequential data

access effectively utilizes the bandwidth of HBM. (4) FPGA vendors (e.g., AMD/Xilinx) offer built-in crossbar switches that can be employed for efficient data communication among hardware modules through HBM.

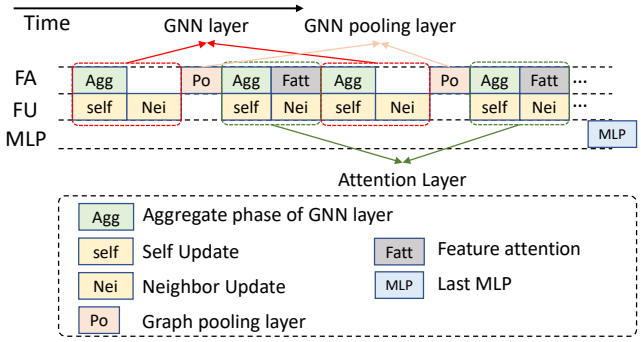

Fig. 6: The diagram of Task scheduling.

Figure 5 illustrates the connectivity details between HBM and the hardware modules. As loading feature vectors demands the majority of the memory bandwidth, each Feature Buffer (in FA and FU) and each Result Buffer (in FA and FU) is connected to three Pseudo Channels (PCs) within HBM. With the assistance of the built-in crossbar, the intermediate results stored by one hardware module can be directly loaded by another hardware module (FA or FU or MLP) without any additional overhead. Consequently, the use of HBM simplifies the task scheduling process, as depicted in Figure 6.

B. Splitting Kernel Technique

The data-center FPGA commonly incorporates multiple Super Logic Regions (SLRs) with limited interconnection capabilities across SLRs. Furthermore, the HBM interfaces are typically connected to a single SLR. These factors can result in challenges during the Place & Route process and can potentially lead to routing failures. Additionally, utilizing cross-SLR interconnections may introduce long wire connections, which can negatively impact the design's frequency. To overcome these issues, we employ the splitting kernel technique [40]. In our illustration using the AMD/Xilinx Alveo U280 platform, we take the following steps: (1) We split a single Processing Element (PE) into multiple hardware modules, including FA, FU, and MLP, and distribute these modules across different SLRs. (2) To address the long wire connections across SLRs, we insert registers into these connections, thereby improving the design's frequency. (3) For the input data to the MLP, we establish a direct connection from the FU to the MLP without the need to load the data from the HBM. This approach eliminates the requirement for long wire connections across three SLRs. The approach outlined above is depicted in Figure 7.

VI. Evaluation

A. Implementation details

We implement the proposed design on AMD/Xilinx Alveo U280, which is a state-of-the-art HBM-enabled data-center

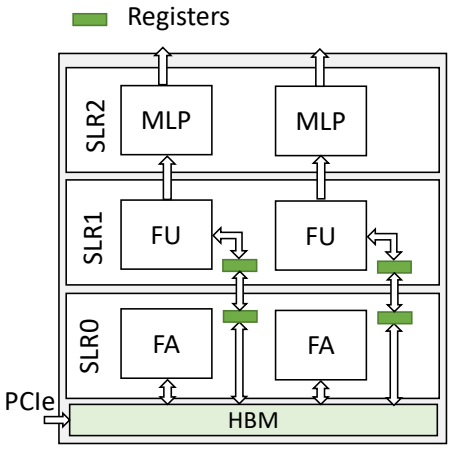

Fig. 7: Kernel placement using splitting kernel technique on AMD/Xilinx Alveo U280.

FPGA board. Alveo U280 offers 8 GB HBM with total memory bandwidth of 460 GB/s. The HBM has 32 pseudo channels (PCs). Alveo U280 has three Super Logic Regions (SLRs) – SLR0, SLR1, and SLR2. The HBM is directly connected to SLR0. We implement the design using Xilinx High-level Synthesis (HLS). We implement two Processing Elements (PEs) on Alveo U280 as shown in Figure 8. Each PE utilizes 11 PCs as shown in Figure 5. For the Feature Aggregation module, $p = 4$ and $q = 16$. For the Feature Update module, $m = 16$. For the MLP module, $s_1 = 4$ and $s_2 = 16$. The resource utilization of the proposed design on AMD/Xilinx Alveo U280.

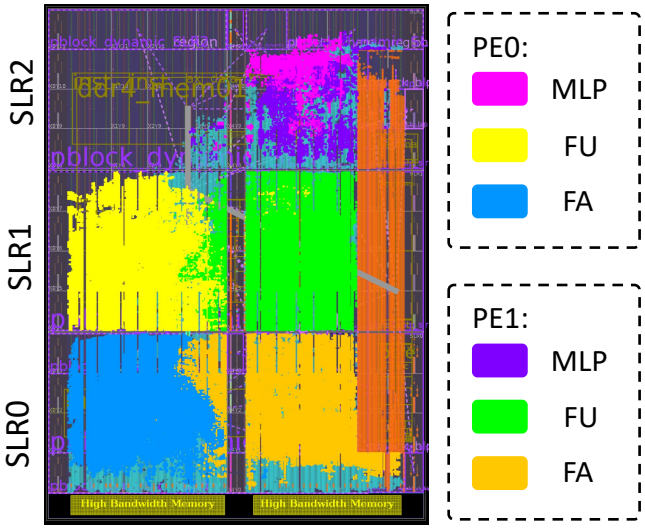

Fig. 8: Device map on AMD/Xilinx Alveo U280. Different hardware modules are highlighted using different colors.

Baseline: We compare our design with state-of-the-art CPU (Intel® Core™ i9-13900K Processor) and GPU (Nvidia RTX A5000). The platform specifications are shown in Table II.
Dataset: The evaluated GNN model in [6] performs inference on the widely used MSTAR [41] dataset.

TABLE I: The resource utilization of the proposed design on AMD/Xilinx Alveo U280

Resource	Used	Available	Utilization (%)
LUT	512520	1303680	39.31%
LUTRAM	76160	600960	12.67%
FF	817142	2607360	31.34%
BRAM	760.5	2016	37.72%
URAM	416	960	43.33%
DPS	3919	9024	43.43%

Performance metrics: We consider the following performance metrics:

- *Latency*: The hardware execution latency of performing inference for a SAR image using the GNN model in [6].
- *Throughput*: The number of SAR images that can be processed per Second.
- *Energy Efficiency*: The energy consumption of performing inference for a SAR image. It includes the energy consumption of entire FPGA board, including the energy consumption of FPGA chip and the energy consumption of FPGA DDR memory.

TABLE II: Specifications of platforms

	CPU	GPU	Our Design
Platform	Intel® Core™ i9-13900K	Nvidia A5000	Alveo U280
Technology	Intel 7 nm	Samsung 8 nm	TSMC 16 nm
Frequency	5.80 GHz	1.7 GHz	260 MHz
Peak Performance (TFLOPS)	0.78	27.7	0.4
On-chip Memory	32 MB	6 MB	45 MB
Memory Bandwidth	107 GB/s	768 GB/s	460 GB/s

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

TABLE III: Comparison of performance

	CPU	GPU	Our Design
Latency (ms)	14.1 (1×)	4.2 (3.3×)	2.7 (5.2×)
Throughput (image/second)	76 (1×)	248 (3.3×)	759 (10×)
Average power (W)	138	61	38
Energy Efficiency (J/image)	1.81 (1×)	0.246 (7.35×)	0.05 (36.2×)

Table III shows the comparison with the state-of-the-art CPU (Intel® Core™ i9-13900K Processor) and GPU (Nvidia RTX A5000). Our implementation on FPGA achieves $5.2\times$ and $1.57\times$ lower latency compared with the implementations on CPU and GPU. In terms of throughput, our implementation on FPGA achieves $10\times$ and $3.3\times$ speedup compared with the implementation on CPU and GPU. Moreover, our implementation on FPGA is $36.2\times$ and $4.9\times$ more energy-efficient that the implementation CPU and GPU. The speedup over CPU and GPU platforms is due to: The computation kernels in GNN have irregular computation and memory access patterns and low data reuse. We optimizes the data path and memory organization for various GNN computation kernels. The processors in CPU or GPU have limited cache sizes (e.g., 32 KB L1 cache and 1 MB L2 cache). The data exchange (due to low data reuse) among L1, L2, and L3 caches becomes the performance bottleneck and results in reduced sustained performance. On the CPU platform, loading data from the L3 cache incurs latency of 32 ns, and loading data from L2 cache incurs latency of $5-12$ ns. Compared with the CPU-only/CPU-GPU, the computation units can access data in one clock cycle from the on-chip Feature/Edge/Weigth/Result buffers. Therefore, although the baseline CPU and GPU platforms have higher $(1.95\times)$ peak performance than the state-of-the-art FPGAs, our implementation on FPGA still outperforms the baselines.

VII. Conclusion and Future Work

In this work, we accelerated the GNN-based SAR automatic target recognition on HBM-enabled FPGA. We developed the customized data path and memory organization to efficiently execute various computation kernels of GNN. We exploited the High Bandwidth Memory (HBM) of FPGA to speedup the data loading and store the intermediate results. We utilized the splitting kernel technique to improve the routability and frequency of the design. Our implementation on a state-of-the-art HBM-enabled FPGA achieves $5.2\times$ and $1.57\times$ lower latency compared with the implementations on CPU and GPU, achieves $10\times$ and $3.3\times$ higher throughput compared with the implementation on CPU and GPU, and are $36.2\times$ and $4.9\times$ more energy-efficient that the implementation CPU and GPU. In the future, we plan to develop a design automation framework to automatically generate the hardware design given a FPGA platform.

Acknowledgement

This work is supported by the DEVCOM Army Research Lab (ARL) under grant W911NF2220159. Equipment and support by AMD AECG are greatly appreciated.

References

[1] L. Landuyt, A. Van Wesemael, G. J.-P. Schumann, R. Hostache, N. E. Verhoest, and F. M. Van Coillie, "Flood mapping based on synthetic aperture radar: An assessment of established approaches," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 57, no. 2, pp. 722–739, 2018.

[2] P. Zhan, W. Zhu, and N. Li, "An automated rice mapping method based on flooding signals in synthetic aperture radar time series," *Remote Sensing of Environment*, vol. 252, p. 112112, 2021.

[3] N. Li, Z. Guo, J. Zhao, L. Wu, and Z. Guo, "Characterizing ancient channel of the yellow river from spaceborne sar: Case study of chinese gaofen-3 satellite," *IEEE Geoscience and Remote Sensing Letters*, vol. 19, pp. 1–5, 2021.

[4] T. Zhang, X. Zhang, J. Shi, and S. Wei, "Hyperli-net: A hyper-light deep learning network for high-accurate and high-speed ship detection from synthetic aperture radar imagery," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 167, pp. 123–153, 2020.

[5] B. Zhang, S. Wijeratne, R. Kannan, V. Prasanna, and C. Busart, "Graph neural network for accurate and low-complexity sar atr," *arXiv preprint arXiv:2305.07119*, 2023.

[6] H. Zhu, N. Lin, H. Leung, R. Leung, and S. Theodoidis, "Target classification from sar imagery based on the pixel grayscale decline by graph convolutional neural network," *IEEE Sensors Letters*, vol. 4, no. 6, pp. 1–4, 2020.

[7] M. Zhang, J. An, L. D. Yang, L. Wu, X. Q. Lu *et al.*, "Convolutional neural network with attention mechanism for sar automatic target recognition," *IEEE Geoscience and Remote Sensing Letters*, vol. 19, pp. 1–5, 2020.

[8] D. A. Morgan, "Deep convolutional neural networks for atr from sar imagery," in *Algorithms for Synthetic Aperture Radar Imagery XXII*, vol. 9475. SPIE, 2015, pp. 116–128.

[9] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[10] J. Pei, Y. Huang, W. Huo, Y. Zhang, J. Yang, and T.-S. Yeo, "Sar automatic target recognition based on multiview deep learning framework," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 4, pp. 2196–2210, 2017.

[11] Z. Ying, C. Xuan, Y. Zhai, B. Sun, J. Li, W. Deng, C. Mai, F. Wang *et al.*, "Tai-sarnet: Deep transferred atrous-inception cnn for small samples sar atr," *Sensors*, vol. 20, no. 6, p. 1724, 2020.

[12] 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.

[13] "Amazon ec2 f1 instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[14] "Intel® developer cloud." [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html

[15] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin *et al.*, "I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO-54: 54th annual IEEE/ACM international symposium on microarchitecture*, 2021, pp. 1051–1063.

[16] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 922–936.

[17] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan *et al.*, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[18] B. Zhang, R. Kannan, and V. Prasanna, "Boostgcn: A framework for optimizing gcn inference on fpga," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 29–39.

[19] S. Liang, "Deepburning-gl: an automated framework for generating graph neural network accelerators," in *2020 IEEE/ACM ICCAD*.

[20] 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.

[21] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale gcn inference," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 61–68.

[22] B. Zhang, H. Zeng, and V. Prasanna, "Graphagile: An fpga-based overlay accelerator for low-latency gnn inference," *arXiv preprint arXiv:2302.01769*, 2023.

[23] B. Zhang, H. Zeng, and V. Prasanna, "Low-latency mini-batch gnn inference on cpu-fpga heterogeneous platform," *arXiv preprint arXiv:2206.08536*, 2022.

[24] B. Zhang and V. Prasanna, "Dynasparse: Accelerating gnn inference through dynamic sparsity exploitation," *arXiv preprint arXiv:2303.12901*, 2023.

[25] 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.

[26] B. Zhang, S. R. Kuppannagari, R. Kannan, and V. Prasanna, "Efficient neighbor-sampling-based gnn training on cpu-fpga heterogeneous platform," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[27] Y. C. Lin, B. Zhang, and V. Prasanna, "Gcn inference acceleration using high-level synthesis," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–6.

[28] H. Zhou, B. Zhang, R. Kannan, V. Prasanna, and C. Busart, "Model-architecture co-design for high performance temporal gnn inference on fpga," *arXiv preprint arXiv:2203.05095*, 2022.

[29] B. Zhang, H. Zeng, and V. Prasanna, "Accelerating large scale gcn inference on fpga," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 241–241.

[30] S. Abi-Karam and C. Hao, "Gnnbuilder: An automated framework for generic graph neural network accelerator generation, simulation, and optimization," *arXiv preprint arXiv:2303.16459*, 2023.

[31] 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.

[32] Y. Meng, H. Men, and V. Prasanna, "Accelerator design and exploration for deformable convolution networks," in *2022 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2022, pp. 1–6.

[33] Y. Meng, R. Kannan, and V. Prasanna, "A framework for monte-carlo tree search on cpu-fpga heterogeneous platform via on-chip dynamic tree management," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2023, pp. 235–245.

[34] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "Hybriddnn: A framework for high-performance hybrid dnn accelerator design and implementation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[35] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnexplorer: a framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[36] "Amd/xilinx alveo u280." [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html

[37] "Intel® stratix® 10 mx fpga." [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10/mx.html

[38] "Advanced extensible interface." [Online]. Available: https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface

[39] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "Hitgraph: High-throughput graph processing framework on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[40] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 54–64.

[41] "Mstar dataset." [Online]. Available: https://www.sdms.afrl.af.mil/index.php?collection=public-data&page=public-data-list