electronics

MDPI

Article

Accelerated Graph Neural Networks on an SoC FPGA for Onboard LEO Satellite Network Routing

Jinhyung Park [1], Heoncheol Lee [1,2,*], Sungryul Kim [3,*], Bongsoo Roh [4] and Myonghun Han [4]

[1] School of Electronic Engineering, Kumoh National Institute of Technology, Gumi-si 39177, Republic of Korea; wnsqja_0903@kumoh.ac.kr
[2] Department of IT Convergence Engineering, Kumoh National Institute of Technology, Gumi-si 39177, Republic of Korea
[3] Department of Software, Kumoh National Institute of Technology, Gumi-si 39177, Republic of Korea
[4] Agency for Defense Development, Daejeon 34186, Republic of Korea; saintroh@add.re.kr (B.R.); mengddor@add.re.kr (M.H.)
* Correspondence: hclee@kumoh.ac.kr (H.L.); sungryul@kumoh.ac.kr (S.K.)

Abstract

This paper presents a system-on-chip field-programmable gate array (SoC FPGA) acceleration architecture for graph-neural-network- and deep-reinforcement-learning (GNN–DRL)-based routing inference in low-Earth-orbit (LEO) satellite networks. Because LEO satellites move at high orbital speeds, the network topology changes continuously, and routing decisions must track the current link state rather than rely only on static rules. GNN-based DRL routing can represent the graph structure of the network when selecting paths, but its message-passing and readout stages are computationally expensive for resource-constrained onboard platforms. To address this limitation, the trained GNN routing model is ported to an SoC FPGA and implemented with a collaborative processing-system (PS) and programmable-logic (PL) architecture. The PS handles candidate-path generation, environment setup, path selection, and network-state updates, whereas the PL executes the computationally dominant message-passing neural network (MPNN) and readout layers. Post-training INT8 quantization, nonlinear-function approximation, vector-level parallelization, and a parallel multiply–accumulate structure are applied to reduce memory pressure and execution time. Experiments on a ZCU104 board using a PYNQ-controlled PS–PL implementation and an NSFNET-based routing environment show that the proposed PS–PL structure reduces the evaluation time from 94.08 s to 12.63 s compared with the PS-only implementation while maintaining an evaluation score close to that of the original model.

Keywords: low-Earth-orbit satellite network; graph neural network; deep reinforcement learning; GNN-based routing; SoC FPGA; onboard computing; INT8 quantization; parallelization

Academic Editor: Tee Hui Teo

Received: 26 May 2026
Revised: 11 June 2026
Accepted: 12 June 2026
Published: 16 June 2026

1. Introduction

Terrestrial communication infrastructure has expanded worldwide, yet coverage gaps still remain. Network connectivity can also degrade or fail when earthquakes, tsunamis, or other disasters damage ground base stations, or when traffic demand rises sharply in a local area [1,2]. Satellite networks have therefore regained attention as a complementary communication infrastructure [2]. Conventional satellite communication has relied largely on geostationary Earth orbit (GEO) satellites. A single GEO satellite can cover a large

area, but GEO systems suffer from high propagation delay, higher link loss, and additional communication-equipment requirements, which have limited their use to relatively specialized services [2,3]. Low-Earth-orbit (LEO) satellite systems provide an alternative. As launch and maintenance costs have decreased, LEO satellite constellations have become more practical for broadband and low-latency services [2,4]. LEO systems have lower delay and lower path loss than GEO systems, although their smaller footprint requires a large constellation to sustain continuous coverage [2–4]. These characteristics make LEO networks attractive for latency-sensitive applications such as Internet-of-Things services and autonomous mobility [4].

Routing in satellite networks has traditionally been based on predefined rules, heuristics, or shortest-path algorithms [5–7]. While this approach has the advantage of being simple to implement and easy to interpret, it has the limitation that various factors such as changes in network status, traffic load, link quality, and latency must be reflected in predefined cost functions or rules [5,8]. In particular, LEO satellite networks face challenges in directly applying existing terrestrial or static satellite network routing algorithms because their topology constantly changes due to the high-speed movement of satellites, and link states and traffic distribution between satellites also vary over time [5,7,9].

Reinforcement-learning-based routing is a promising approach for such dynamic environments because the agent can observe the network state and learn a routing policy from interaction with the environment [8–10]. In particular, deep reinforcement learning (DRL) allows routing problems to be modeled as sequential decision-making problems, making it well-suited for learning path selection policies that improve long-term network performance based on information such as link delay, congestion, bandwidth, and queue states [9–12]. Therefore, DRL-based routing provides a suitable framework for adaptive path selection in dynamically changing LEO satellite networks [9–12]. It should be noted that learning-based routing does not provide the formal delivery guarantees of classical shortest-path protocols. However, prior studies have shown that DRL-based routing can maintain stable path-selection performance under the dynamic topology and time-varying link states of LEO networks [9–13]. In this work, DRL is therefore used as an adaptive path-selection policy rather than a hard-guaranteed routing mechanism.

However, when DRL-based routing is combined with graph neural networks (GNNs) to represent graph-structured network states, the inference process becomes computationally intensive. In particular, message passing and readout operations require repeated aggregation and feature transformation over nodes and links, which can increase inference latency on a CPU-only onboard computer (OBC). Although GPUs can accelerate these operations, their power consumption and thermal requirements are not well suited to resource-constrained onboard platforms. In this paper, an SoC FPGA refers to a system-on-chip field-programmable gate array that integrates a processing system (PS) and programmable logic (PL) in a single device. This platform is suitable for the proposed routing-inference workload because control-oriented routing procedures can be handled by the PS, whereas repetitive GNN operations can be parallelized in the PL [14–19].

The key problem addressed in this paper is the high inference latency of GNN–DRL-based routing inference when executed on a processing system alone, which limits its applicability to time-sensitive onboard routing decisions in dynamic LEO satellite networks. Therefore, the goal of this study is to design and evaluate a hardware-accelerated routing-inference architecture that reduces execution time while preserving the routing behavior of the original GNN–DRL model. The main idea is to partition the routing process between the PS and PL: control-oriented tasks such as candidate-path generation and environment-state updates are executed in the PS, while the computationally intensive GNN inference

layers are offloaded to the PL using INT8 post-training quantization, nonlinear-function approximation, and vector-level parallelism.

The main contributions of this paper are summarized as follows:

1. A GNN–DRL-based routing inference algorithm is ported to an SoC FPGA and implemented using a PS–PL collaborative structure.
2. Post-training INT8 quantization is applied to reduce FPGA resource usage, thereby enabling greater PL parallelism for accelerating the routing inference algorithm.
3. The proposed structure is implemented on a ZCU104 SoC FPGA board and evaluated using an NSFNET-based routing environment to verify the feasibility and acceleration effect of the proposed PS–PL implementation. The results show that the proposed implementation substantially reduces evaluation time while maintaining a routing score close to that of the original model.

The rest of this paper is organized as follows. Section 2 reviews related studies on learning-based routing and hardware acceleration for LEO satellite networks. Section 3 describes the characteristics of LEO satellite networks and the implementation challenges of SoC FPGA-based GNN acceleration. Section 4 presents the proposed GNN–DRL routing-inference structure, including PS–PL task partitioning, INT8 quantization, hardware-oriented nonlinear-function approximation, and parallelization. Section 5 reports the experimental setup, training results, porting and quantization results, execution-time comparison, and FPGA resource utilization. Section 6 discusses the limitations of the current evaluation and future research directions. Section 7 concludes the paper.

2. Related Work

Several recent studies have investigated learning-based routing for LEO satellite networks. Kim et al. proposed a Dueling-DQN-based routing algorithm for LEO satellite networks and accelerated part of the convolutional-neural-network (CNN) computation on an FPGA, achieving the same routing results with an approximately 3.10-fold speedup [20]. That work, however, targets a CNN-based structure and therefore does not directly exploit the graph structure of the satellite network. Jo et al. offloaded CNN computation to the PL in a PS–PL FPGA architecture and applied INT8 quantization, achieving an approximately 14.7× speedup and demonstrating the effectiveness of quantization for computationally intensive neural-network inference [21]. Xu et al. modeled inter-satellite routing as a graph, extracted satellite relationships using a GNN, and selected routing paths through DRL [9]. These studies demonstrate that DRL can serve as an alternative to conventional rule-based routing for LEO satellite networks.

As shown in Table 1, prior studies have addressed different aspects of learning-based LEO satellite network routing. Kim et al. applied DRL-based routing and FPGA acceleration, but did not explicitly model the satellite network as a graph. Jo et al. demonstrated efficient integer-only neural-network inference on an SoC FPGA, but did not focus on graph-based topology modeling or DRL-based routing decisions. Xu et al. considered graph-based topology modeling and DRL-based routing using GNNs, but did not investigate FPGA-based acceleration. Therefore, existing studies do not simultaneously address graph-aware routing intelligence and hardware-efficient implementation.

Table 1. Comparison of related studies on LEO satellite-network routing.

Feature	Kim et al. [20]	Jo et al. [21]	Xu et al. [9]	Proposed
Graph-based topology modeling	No	No	Yes	Yes
DRL-based routing decision	Yes	No	Yes	Yes
FPGA-based acceleration	Yes	Yes	No	Yes
INT8 Quantization	No	Yes	No	Yes

Among these approaches, GNN-based routing is particularly attractive because it can naturally represent the graph structure of a LEO satellite network [9,13]. However, no prior study combines graph-aware routing intelligence with a hardware-efficient onboard implementation. The hardware-level challenges that arise when GNN inference is executed on a resource-constrained onboard platform are analyzed in Section 3.

3. System Configuration and Problem Statement

3.1. LEO Satellite Network and Routing Structure

LEO satellites operate at altitudes between approximately 160 km and 2000 km above the Earth and orbit at high speeds of about 6.9–7.8 km/s. This orbital regime provides low propagation delay and low path loss, but it also creates routing challenges. Because an individual LEO satellite covers a relatively small footprint, global coverage requires a constellation composed of hundreds or thousands of satellites. Inter-satellite links (ISLs) enable multi-hop data forwarding between satellites, and the routing path is therefore determined by the time-varying connectivity among satellites [22]. The high orbital speed of LEO satellites also means that the connected ground stations, user terminals, and neighboring satellites change over time. A LEO satellite network is thus a dynamic-topology network, and routing methods designed for static networks are not suitable without modification [22,23]. Figure 1 illustrates the motivation and overall concept of the proposed SoC FPGA-based GNN acceleration approach for onboard LEO satellite routing.

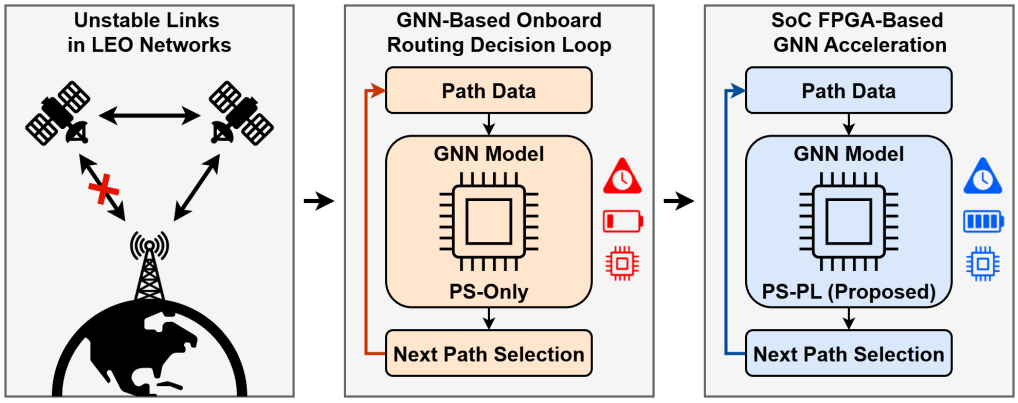

Figure 1. Conceptual overview of LEO satellite routing and SoC FPGA-based GNN acceleration. Frequent link disruptions in dynamic LEO networks require repeated onboard routing decisions, while PS-only GNN inference can suffer from high inference time, high power demand, and inefficient use of onboard computing resources. The proposed PS–PL structure accelerates GNN inference for efficient next-path selection.

3.2. Challenges in SoC FPGA-Based GNN Acceleration

Implementing and accelerating a GNN-based routing algorithm on an SoC FPGA raises several hardware-level challenges. The GNN model used in this work follows the standard message-passing neural network (MPNN) framework, in which node or link representations are updated by aggregating information from neighboring elements [24]. The routing-inference structure is based on the GNN–DRL routing model in [25], where graph-level representations are used to evaluate candidate routing paths.

Before introducing the equations, we define $h_v^{(l)}$ as the feature representation of node or link v at layer l, $\mathcal{N}(v)$ as the set of neighboring elements of v, e_{uv} as the edge feature between u and v, $\phi_m(\cdot)$ as the message function, and $\phi_u(\cdot)$ as the update function. The message aggregation and update operations are expressed as

$$m_v^{(l)} = \sum_{u \in \mathcal{N}(v)} \phi_m\left(h_v^{(l)}, h_u^{(l)}, e_{uv}\right), \tag{1}$$

$$h_v^{(l+1)} = \phi_u\left(h_v^{(l)}, m_v^{(l)}\right), \tag{2}$$

After several message-passing iterations, a graph-level representation is generated by a readout operation. Here, h_G denotes the graph-level representation obtained after L message-passing layers, $\mathcal{V}$ is the set of nodes or links included in the graph, and $f_{\text{readout}}(\cdot)$ maps the graph representation to the Q-value used for candidate-path evaluation. The readout operation is expressed as

$$h_G = \sum_{v \in \mathcal{V}} h_v^{(L)}, \quad Q = f_{\text{readout}}(h_G). \tag{3}$$

These equations show that GNN inference consists of repeated neighbor-dependent aggregation, feature transformation, and graph-level reduction. Unlike CNNs, where the data access pattern is regular, GNN operations depend on the graph connectivity. As a result, memory accesses for gathering neighbor features and scattering messages are irregular, and the amount of computation varies depending on the number of neighbors assigned to each node or link. This makes it difficult to fully utilize parallel hardware resources, even when multiple processing elements are implemented in the PL [14–17].

In addition, the feature-transformation and readout functions are implemented using standard affine transformations. Let x, W, b, and y denote the input feature vector, weight matrix, bias vector, and output feature vector, respectively. The linear operation is expressed as

$$y = Wx + b, \tag{4}$$

These affine transformations require a large number of multiply–accumulate operations. Mapping these operations directly to the PL without optimization can significantly increase the usage of DSPs, LUTs, BRAMs, and on-chip memory bandwidth. Furthermore, if intermediate features, graph indices, and Q-values are repeatedly transferred between the PS and PL, the communication overhead can reduce the benefit of hardware acceleration.

These challenges are critical in onboard LEO satellite routing because routing decisions must be generated within a limited time under continuously changing network topology and link states [5,7,9]. However, onboard computing platforms have strict constraints on power consumption, thermal dissipation, memory capacity, and hardware resources [18,26]. Therefore, efficient SoC FPGA-based acceleration of GNN routing requires not only parallelizing arithmetic operations, but also reducing resource usage and data movement. In this work, quantization, nonlinear function approximation, and HLS-based parallelization are applied to implement the MPNN and Readout operations efficiently on the PL while maintaining the routing decision capability of the original GNN model [25].

4. Proposed Method

4.1. Overall Architecture

The GNN-based DRL routing model used in this work is based on the open-source implementation associated with [25]. Specifically, this work adopts the graph-based network-state representation, candidate-path evaluation using Q-values, and the MPNN-and-Readout-based inference structure from [25]. In this framework, the GNN model evaluates each candidate path based on the current network state, and the path with the highest routing score is selected. The model was modified to satisfy FPGA resource constraints, and its computationally intensive layers were restructured for parallel acceleration

in the PL. To keep the evaluation environment consistent with the open-source implementation and to support a fair comparison, the NSFNET topology was used. Although NSFNET does not directly represent a real LEO satellite topology, it provides a graph-based routing environment with nodes and links similar in structure to those used in LEO routing studies; therefore, it is suitable for functional verification. Model training was performed on a PC, and the trained model was then used to run routing inference on the FPGA. Figure 2 summarizes the overall PS–PL cooperative routing-inference flow.

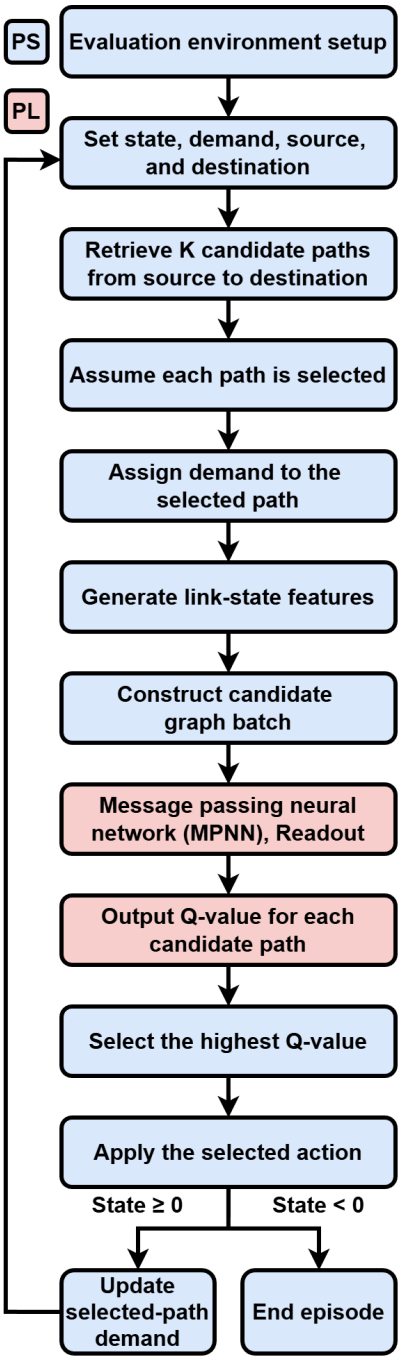

Figure 2. Flowchart of the proposed routing-inference algorithm. Blue blocks represent PS tasks for environment setup, candidate-path generation, path selection, and network-state updates, whereas red blocks represent PL tasks for MPNN and Readout-based Q-value computation.

The flowchart was constructed according to the PS–PL task partitioning principle used in this work. The proposed routing-inference flow was selected because it separates control-

dominant routing procedures from computation-dominant GNN inference operations. Candidate-path generation, environment setup, path selection, and network-state updates involve irregular control flow and dynamic graph-state management; therefore, they are executed in the PS. In contrast, the MPNN and readout layers consist mainly of repeated aggregation, feature transformation, and multiply–accumulate operations, which are suitable for parallel execution in the PL. Alternative implementations, such as PS-only inference, full-PL implementation, or GPU-based inference, were considered less suitable for the target onboard scenario because they either increase inference latency, complicate dynamic control and memory management, or impose higher power and thermal requirements.

4.2. GNN-Based LEO Satellite Routing Inference

As shown in Figure 2, the routing-inference procedure begins with environment configuration on the PS through PYNQ. Each link is assigned a capacity of 200, and each scenario starts with a random source node, destination node, and traffic demand selected from 8, 32, and 64. The PS then obtains four candidate paths from the source to the destination using a depth-first-search-based K-shortest-path (DFS-KSP) procedure. For each candidate path, the algorithm temporarily assumes that the path is selected, marks the demand on the links belonging to that path, and constructs the GNN input link-state vectors using link capacity, betweenness, and a one-hot demand indicator. These link-state vectors are batched and stored in memory. The PL then executes the MPNN and readout layers and outputs a Q-value for each candidate path. The PS selects the path with the largest Q-value and applies the demand to the links in that path. If the capacity of any link becomes negative, the environment is reset and the next episode begins; otherwise, a new routing request with a random source, destination, and demand is generated.

Algorithms 1 and 2 summarize the routing inference procedure and the PS–PL task split. In the algorithms, E denotes the number of evaluation episodes, R denotes the number of routing requests per episode, and K denotes the number of candidate paths. S denotes the current network state, and each routing request is represented by demand d, source node s, and destination node t. $\mathcal{P}$ denotes the set of candidate paths between s and t. The tensors X, g, f_1, and f_2 represent the link-state vector tensor, graph identifier tensor, source index tensor for message passing, and destination index tensor for message passing, respectively. N denotes the total number of links in the batched graph, and n_{valid} denotes the number of valid candidate paths. Q denotes the Q-values of candidate paths, and a^* is the selected action index.

Algorithm 1 GNN–DRL Routing Evaluation Flow

Require: Number of episodes E, requests per episode R, candidate path number $K = 4$
Ensure: Average routing score
for $e = 1$ to E **do**
  Initialize network state S
  for $r = 1$ to R **do**
    Generate or load routing request (d, s, t)
    $\mathcal{P} \leftarrow \text{LookupCandidatePaths}(s, t, K)$
    $(X, g, f_1, f_2, N, n_{\text{valid}}) \leftarrow \text{BuildGraphBatch}(S, d, \mathcal{P}, K)$
    $Q \leftarrow \text{GNNInference(PL)}(X, g, f_1, f_2, N)$
    $a^* \leftarrow \arg\max_{0 \le i < n_{\text{valid}}} Q_i$
    $S \leftarrow \text{ApplyPath}(S, \mathcal{P}_{a^*}, d)$
    if any link capacity in $S < 0$ **then**
      Terminate current episode
      break
    end if
    Accumulate reward
  end for
end for
return average score over E episodes

Algorithm 2 GNNInference (PL)

Require: X, g, f_1, f_2, N
Ensure: Q-values Q
$H \leftarrow X$
for $i = 1$ to 4 **do**
  $M \leftarrow \text{Message}(H, f_1, f_2)$
  $A \leftarrow \text{Scatter}(M, f_2, N)$
  $H \leftarrow \text{GRUUpdate}(A, H)$
end for
$Z \leftarrow \text{GraphSum}(H, g)$
$Q \leftarrow \text{Readout}(Z)$
return Q

4.3. INT8 Quantization

The original open-source GNN routing model stores model parameters and intermediate values in FP32 format. Because FP32 uses 32 bits per value, it requires about four times as much storage as INT8 for the same number of values. This becomes a significant burden in an FPGA implementation with limited BRAM and external-memory bandwidth. FP32 arithmetic also consumes more hardware resources than INT8 arithmetic, reducing the amount of parallelism that can be implemented in the PL. To reduce this cost, the trained GNN model parameters were quantized to INT8 using post-training quantization (PTQ).

INT8 quantization was selected as a practical balance between implementation efficiency and inference accuracy. While higher-precision formats such as FP16 and INT16 can better preserve numerical precision, they offer more limited reductions in memory footprint and hardware cost. Conversely, lower-precision formats such as INT4 or binary quantization can further reduce resource usage but often require additional retraining or calibration and may introduce larger accuracy degradation. Considering these trade-offs, INT8 provides an effective compromise that maintains routing-performance quality while enabling efficient FPGA implementation. Moreover, low-precision integer inference, particularly INT8 inference, is widely used as a practical target in contemporary AI accelerators and adaptive SoC platforms, which further supports INT8 as a suitable precision format for hardware-oriented inference.

In this work, symmetric INT8 post-training quantization is applied following the standard uniform quantization formulation used for efficient integer inference [27,28]. For a given FP32 tensor x, the scale factor s is computed from the maximum absolute value of the tensor. The lower bound of 10^{-8} is included as an implementation safeguard to avoid division by zero:

$$s = \max\left(\frac{\max(|x|)}{127}, 10^{-8}\right). \tag{5}$$

The FP32 value is then converted to an INT8 value by rounding and clipping:

$$q = \text{clip}\left(\text{round}\left(\frac{x}{s}\right), -127, 127\right). \tag{6}$$

When the quantized value is used for inference, it can be reconstructed approximately as

$$\hat{x} = s \cdot q. \tag{7}$$

For linear operations, the multiplication between the quantized input and weight is performed using INT8 values, and the result is rescaled using the corresponding scale factors:

$$y = s_x s_W \sum_i x_{q,i} W_{q,i} + s_b b_q, \tag{8}$$

where s_x and s_W denote the scale factors of the input feature and weight, respectively, and b_q denotes the quantized bias with scale factor s_b. This formulation reduces the storage size of model parameters and enables the use of lower-cost integer arithmetic in the PL.

Operations that are sensitive to numerical precision, such as accumulation and selected rescaling operations, are retained in FP32 to reduce numerical degradation. Because quantization reduces representation precision, the evaluation score and final path choices can differ from those of the original FP32 model. Therefore, the influence of PTQ is evaluated by comparing the average evaluation score and final path-selection consistency before and after quantization.

4.4. Hardware-Oriented Nonlinear Function Approximation

The MPNN and readout layers contain nonlinear functions that are expensive to implement directly in the PL. In the MPNN, SELU is used after message generation, and sigmoid and hyperbolic tangent functions are used in the gated recurrent unit (GRU)-based update. In the Readout layer, SELU is used after each linear layer. Repeated exponential, tanh, and sigmoid operations increase both resource use and execution time if implemented directly on the FPGA. Therefore, in the proposed hardware implementation, these nonlinear functions are replaced with implementation-specific approximation structures. The approximation equations in this subsection are not general GNN model equations, but hardware-oriented formulations used in this work to reduce PL resource usage and execution latency.

For the exponential term used in the negative region of SELU, values of $\exp(x)$ for $x \in [-8, 0]$ are stored at intervals of 0.5. If the input is smaller than -8, the value is clamped to $\exp(-8)$. Otherwise, the output is computed by linear interpolation between the two nearest stored values. For example, $\exp(-1.3)$ is approximated by linearly interpolating between the stored values of $\exp(-1.5)$ and $\exp(-1.0)$. In the proposed implementation, the tanh function is approximated using the following clipped rational function:

$$\tanh_{\text{approx}}(x) = \begin{cases} -1, & x < -3, \\ \dfrac{x(27 + x^2)}{27 + 9x^2}, & -3 \le x \le 3, \\ 1, & x > 3. \end{cases} \tag{9}$$

The sigmoid function is approximated using the standard relationship between sigmoid and tanh. In this implementation, $\tanh(\cdot)$ is replaced with the approximated tanh function in the following expression:

$$\sigma_{\text{approx}}(x) = \frac{1 + \tanh_{\text{approx}}(x/2)}{2}. \tag{10}$$

These approximations reduce the cost of nonlinear computation while preserving numerical fidelity for routing inference.

4.5. Parallelization and Acceleration

The quantization and approximation steps reduce the hardware resource pressure of the GNN model, and the remaining resources are used to increase the parallelism of the PL accelerator. In this work, two levels of parallelization are applied: link-state vector parallelization for repeated link-state operations and tiled multiply–accumulate (MAC) parallelization for linear layers.

First, link-state vector parallelization is applied to the operations repeatedly used in the MPNN and readout layers. The link-state vector has a dimension of $F = 20$. If all elements are processed sequentially, link-state vector operations such as message generation, GRU

gate computation, activation, and graph-sum require repeated loop execution over the 20-dimensional vector. A general element-wise operation can be expressed as

$$y_i = f(x_i), \quad i = 0, 1, \ldots, F-1. \tag{11}$$

In the proposed design, a vector parallel factor of $P_{\text{vec}} = 4$ is used. Therefore, four elements $(x_i, x_{i+1}, x_{i+2}, x_{i+3})$ are processed at the same time. To support simultaneous access to these elements, the on-chip buffers that store the link-state vectors are organized as multiple independent memory banks. Each bank supplies one link-state vector element to a dedicated arithmetic lane, and the same operation is replicated across four parallel lanes. As a result, the 20-dimensional link-state vector can be processed in five parallel steps instead of 20 sequential steps. In the HLS implementation, the memory-banked buffer structure and replicated arithmetic lanes are implemented using `ARRAY_PARTITION` and `UNROLL`, respectively. The resulting link-state vector parallelization structure is illustrated in Figure 3.

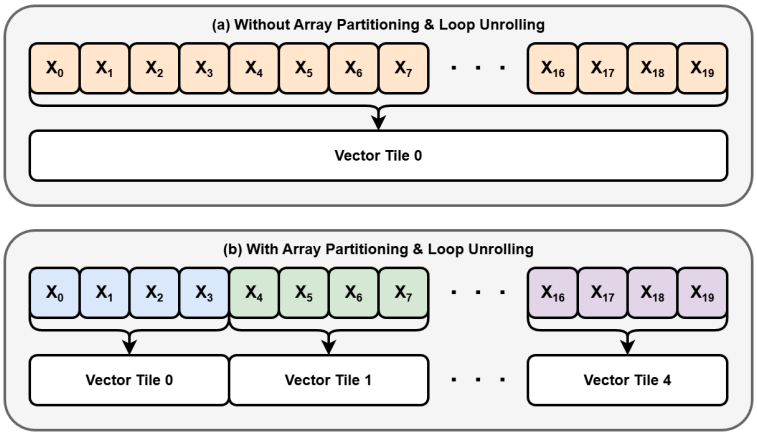

Figure 3. Link-state vector parallelization using memory banking and replicated arithmetic lanes. In (**a**), without array partitioning and loop unrolling, a single vector tile processes the 20 elements of a link-state vector sequentially. Assuming that processing one element requires time T, the total latency becomes $20T$. In (**b**), with array partitioning and loop unrolling, four elements are processed in parallel in each step. As a result, the 20-element link-state vector is divided into five 4-element tiles, and the total latency is reduced to $5T$.

Second, the linear layers in the Message MLP, GRU update, and Readout MLP are accelerated using a tiled MAC structure. For output channel o, the linear operation is expressed as

$$y_o = \sum_{i=0}^{D_{\text{in}}-1} W_{o,i} x_i + b_o. \tag{12}$$

Here, D_{in} denotes the input dimension of the linear layer. In the hardware implementation, a weight tile is arranged according to the input-parallel and output-parallel dimensions. Therefore, the 4×5 notation in Figure 4 denotes $P_{\text{in}} \times P_{\text{out}}$, not the row–column shape of the mathematical weight matrix. In the proposed implementation, the input-vector parallel factor and output-channel parallel factor are set to $P_{\text{in}} = 4$ and $P_{\text{out}} = 5$, respectively. This means that four input elements are multiplied with the corresponding weights for five output channels in one computation tile.

$$P_{\text{in}} \times P_{\text{out}} = 4 \times 5 = 20. \tag{13}$$

Therefore, up to 20 multiply–accumulate operations can be executed in parallel within a single tile. The partial sums for the five output channels are accumulated across multiple

input tiles until the full linear operation is completed. Figure 4 illustrates the tiled MAC structure used for parallel linear-layer computation.

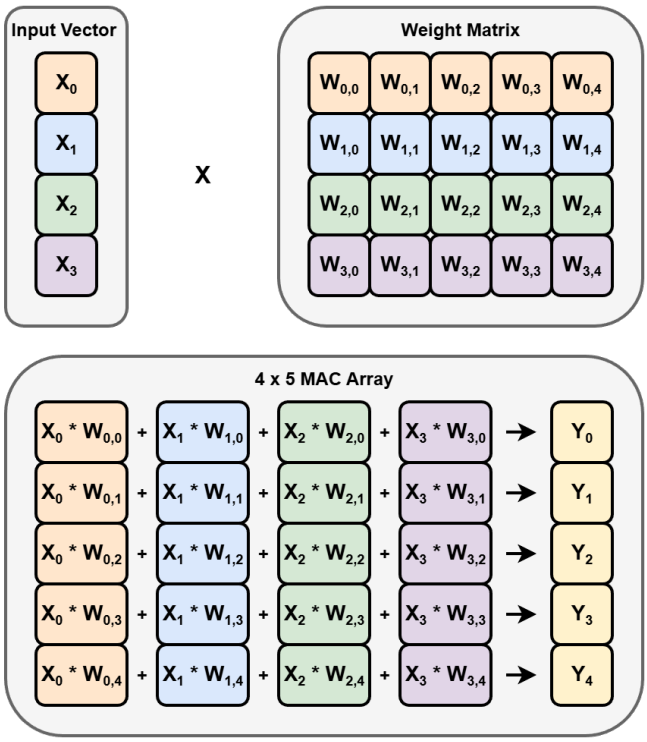

Figure 4. Tiled MAC structure with 4 input-parallel lanes and 5 output-parallel channels for linear-layer acceleration. A 4-element input tile is combined with weights for five output channels, enabling 20 parallel MAC operations and partial-sum accumulation for the Message MLP, GRU update, and Readout MLP.

The parallel factors are selected conservatively rather than fully unrolling all operations. Fully parallelizing the GNN computation would require excessive DSPs, LUTs, BRAMs, and routing resources, which can exceed the available resources of the target FPGA. Therefore, the proposed design uses limited parallel factors that fit within the ZCU104 resource budget.

5. Experimental Results

5.1. Experimental Setup

The experiments were conducted on a Zynq UltraScale+ MPSoC ZCU104 evaluation board (Xilinx, San Jose, CA, USA). The ZCU104 board was selected as a prototyping platform because it provides both an ARM-based PS and FPGA PL on a single device, allowing the proposed PS–PL task partitioning and PL-based GNN acceleration to be evaluated under resource-constrained conditions. After model training on a PC, PTQ was applied to the trained model. After PTQ was applied, the MPNN and readout layers were ported to the C++14 standard, and the corresponding IP core was generated using Vitis HLS 2022.2 (Xilinx, San Jose, CA, USA). The overall system was then designed in Vivado 2022.2 (Xilinx, San Jose, CA, USA) and deployed through PYNQ 3.1.1 (Xilinx, San Jose, CA, USA) using the generated hardware information. The software environment used Python 3.10.4 (Python Software Foundation, Wilmington, DE, USA) and PyTorch 2.0.1 (Meta AI, Menlo Park, CA, USA). The default clock and implementation settings of Vitis HLS 2022.2 and Vivado 2022.2 were used, and the synthesized IP core met timing under these settings.

Figure 5 shows the experimental setup used for the proposed SoC FPGA-based routing-inference evaluation.

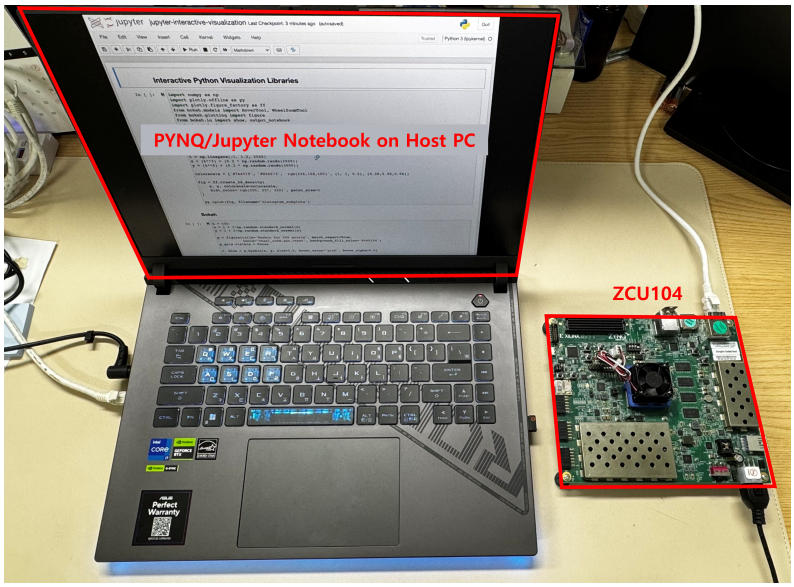

Figure 5. Experimental setup for SoC FPGA-based GNN routing inference. The ZCU104 board executes the proposed PS–PL routing-inference implementation, while the host PC accesses the PYNQ 3.1.1 environment to deploy the generated hardware design, control the evaluation procedure, and collect the execution results.

5.2. Training Results

Among the environments provided by the open-source work, NSFNET was selected for training and evaluation, as shown in Figure 6. NSFNET is a widely used benchmark topology derived from the U.S. National Science Foundation Network. In this work, it is used as a fixed graph-based routing benchmark with 14 nodes and bidirectional links, as shown in Figure 6. Although it does not capture the dynamic orbital behavior of LEO satellite constellations, it provides a compact and reproducible environment for evaluating the proposed GNN–DRL routing and FPGA acceleration framework. To simplify later PYNQ execution and PL porting, the original TensorFlow-based code was ported to PyTorch. All training results reported in this section are from the PyTorch implementation.

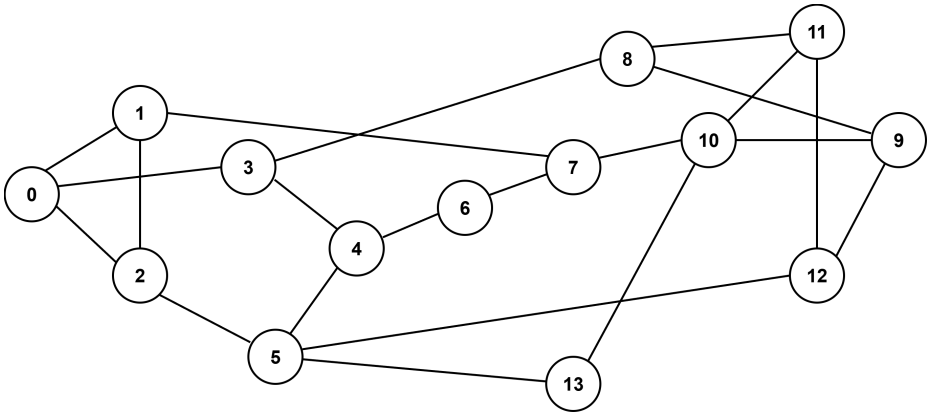

Figure 6. NSFNET benchmark topology used for training and evaluation. The topology is inherited from the open-source baseline and provides a graph-based routing environment for evaluating the proposed FPGA-based GNN routing-inference structure.

The training loss of the PyTorch-ported model is shown in Figure 7.

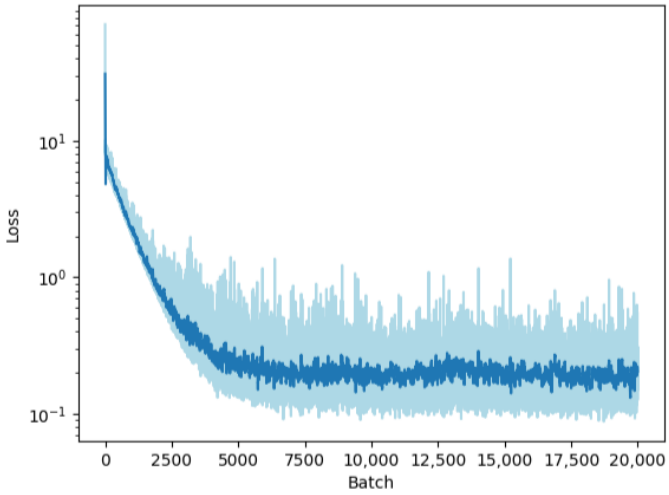

Figure 7. Training loss of the PyTorch-ported GNN–DRL routing model. The light-blue curve represents the raw average loss recorded at each batch, while the dark-blue curve shows the smoothed loss trend. The loss is initially large because Q-value prediction is unstable, but it decreases to approximately 0.17 as training proceeds, indicating that the model learns the relationship between the network state and candidate paths.

The average test score measured during training is shown in Figure 8.

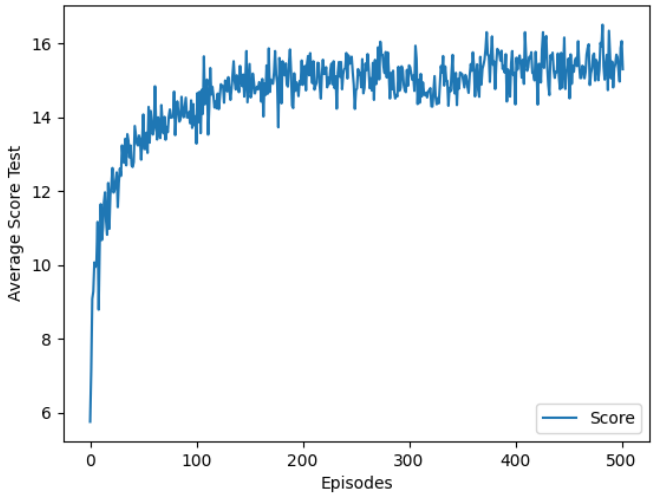

Figure 8. Average test score of the PyTorch-ported GNN–DRL routing model during training. The score increases from approximately 6.81 to a maximum of 16.74, indicating that the learned policy gradually improves its ability to accommodate traffic by considering the residual network capacity.

5.3. Porting and Quantization Results

The trained model was evaluated before and after PTQ. Each evaluation was conducted for 50 episodes, and the seed was fixed so that all evaluations used the same environment. After porting to PyTorch, the average evaluation score before PTQ was 16.01. After INT8 PTQ, the average score was 15.66, corresponding to a decrease of approximately 0.35. The quantized model nevertheless followed the score trend of the original open-source model and remained above the non-learning routing baselines reported in that study. When the quantized inference was executed on the PS through PYNQ, the average evaluation score was also 15.66, matching the post-PTQ score result.

The MPNN and readout layers were then executed in the PL under the PS–PL collaborative structure. This implementation uses the PTQ-applied model and the nonlinear approximations described above. Over 50 evaluation episodes, the average score of the PS–PL implementation was 15.85, which is 0.19 higher than the PTQ-only score of 15.66. When path selections were compared with those of the pre-PTQ model in each scenario, the path agreement rate was 87.45%. The average-score comparison is shown in Figure 9.

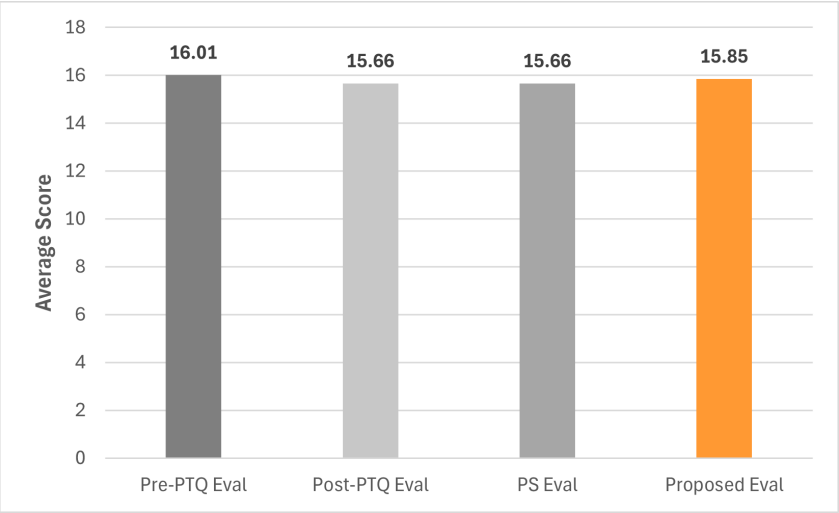

Figure 9. Average evaluation score comparison before and after quantization and FPGA execution. The pre-PTQ model achieves an average score of 16.01, while the post-PTQ and PS-only implementations achieve 15.66. The proposed PS–PL implementation achieves 15.85, showing that the quantized and approximated FPGA implementation maintains a score close to the original model.

The slight increase from 15.66 to 15.85 after PS–PL execution is attributed to implementation-dependent numerical differences introduced during C++/HLS porting, nonlinear-function approximation, rescaling, and the PL execution flow. These differences can slightly alter Q-values and, when candidate-path Q-values are close, may lead to different path selections. As routing decisions affect subsequent network states, such variations can result in a slightly different average score. Therefore, the observed increase reflects minor numerical deviations in the implementation rather than an improvement in the learned policy.

5.4. Execution-Time Comparison

The execution time was compared for three implementations: a PS-only implementation without approximation or parallelization, a basic PS–PL collaborative implementation without approximation or parallelization, and the proposed PS–PL implementation with approximation and parallelization. The measured interval excludes environment setup and corresponds to the full routing-evaluation flow after the "Evaluation environment setup" block in Figure 2. The PS-only implementation required 94.08 s, the basic PS–PL implementation required 22.90 s, and the proposed PS–PL implementation required 12.63 s. Thus, the proposed structure reduces the evaluation time by approximately 7.45 times compared with the PS-only implementation and by approximately 1.81 times compared with the basic PS–PL implementation, as summarized in Figure 10.

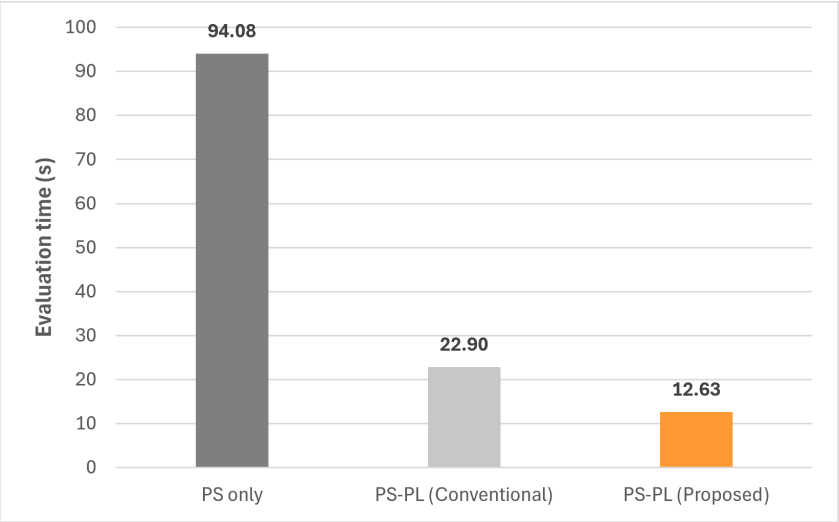

Figure 10. Execution-time comparison of PS-only, basic PS–PL, and proposed PS–PL routing-inference implementations. The proposed PS–PL implementation reduces the execution time from 94.08 s to 12.63 s compared with the PS-only implementation and from 22.90 s to 12.63 s compared with the basic PS–PL implementation.

5.5. FPGA Resource Utilization

The proposed implementation increases the use of LUTs, FFs, DSPs, and BUFGs because additional parallelism is introduced. However, quantization and nonlinear approximation limit the resource increase and allow the GNN–DRL inference algorithm to run within the available resources of the ZCU104 board. The proposed implementation also reduces BRAM use from 211 blocks to 176 blocks, reflecting the benefit of the quantized parameter representation and the revised memory structure.

Table 2 compares the FPGA resource utilization of the basic and proposed PS–PL implementations on the same ZCU104 board. This table is intended as an implementation-level comparison under the same model, device, and evaluation flow, rather than a direct resource comparison with prior studies. A direct resource comparison with other known solutions is not included because previous works use different routing models, neural-network structures, FPGA devices, and evaluation workloads.

Table 2. FPGA resource utilization of the basic and proposed PS–PL implementations on the ZCU104 board. The proposed design increases LUT, FF, and DSP usage due to additional parallelism, while BRAM usage is reduced through quantization and revised memory organization.

Resource	Available	PS–PL (Conventional)	PS–PL (Proposed)
LUT	230,400	41,622 (18.07%)	111,597 (48.44%)
LUTRAM	101,760	3177 (3.12%)	4027 (3.96%)
FF	460,800	32,092 (6.96%)	96,105 (20.86%)
BRAM	312	211 (67.63%)	176 (56.41%)
DSP	1728	228 (13.19%)	1183 (68.46%)
BUFG	544	2 (0.37%)	5 (0.92%)

The improvement in execution time is obtained at the cost of increased PL resource utilization. Compared with the basic PS–PL implementation, the proposed design increases LUT, FF, and DSP usage because additional parallel arithmetic lanes and tiled MAC units are introduced for GNN inference acceleration. In particular, DSP usage increases because the proposed design executes multiple multiply–accumulate operations in parallel. Therefore, the proposed implementation trades additional PL resources and reduced resource headroom for lower routing-inference latency. Nevertheless, the design still fits within

the available resources of the ZCU104 board, and BRAM usage is reduced through INT8 quantization and revised memory organization.

6. Discussion and Future Work

This study used the NSFNET benchmark topology to maintain consistency with the open-source baseline and to verify the feasibility of SoC FPGA-based GNN routing inference under a graph-based routing environment. However, NSFNET does not fully capture the time-varying orbital dynamics of realistic LEO satellite constellations. Therefore, the results should be interpreted as a functional verification and acceleration evaluation of the proposed PS–PL implementation rather than a complete validation under operational LEO satellite-network conditions. Future work will extend the evaluation to realistic LEO constellation scenarios generated from orbital parameters or satellite ephemeris data.

The present evaluation focuses on routing score, execution time, and FPGA resource utilization. Direct board-level power measurement and energy-per-routing-decision analysis were not included in this study. Therefore, this paper does not claim a quantitatively measured energy-efficiency advantage over other hardware platforms. Accordingly, the contribution of this work is limited to demonstrating that the proposed GNN–DRL routing inference can be implemented within the resource budget of an SoC FPGA while reducing execution time.

Scalability and PS–PL communication overhead also require further investigation. The current implementation was evaluated using a relatively small graph topology, whereas realistic LEO satellite constellations can contain a much larger number of satellites, inter-satellite links, and candidate paths. As the graph size increases, memory usage, graph-batch construction time, PS–PL data movement, and external-memory bandwidth demand may become important bottlenecks. Future work will therefore evaluate larger graph structures and analyze the effect of graph size and PS–PL data transfer on end-to-end routing-inference latency.

7. Conclusions

This paper has proposed an SoC FPGA-based acceleration architecture for GNN–DRL routing inference targeting LEO satellite-network applications. In the proposed structure, candidate-path generation, environment setup, path selection, and network-state updates are executed in the PS, while the computationally intensive MPNN and readout layers are executed in the PL. PTQ-based INT8 quantization, hardware-oriented nonlinear function approximation, vector-level parallelization, and a parallel MAC structure were applied to fit the inference algorithm into the limited FPGA resource budget and to reduce execution time. Experimental results showed that the average score of the pre-PTQ model was 16.01 and that the PS–PL implementation achieved 15.85, maintaining performance close to that of the original model. The path agreement rate between the pre-PTQ model and the PS–PL implementation was 87.45%. The execution time was 94.08 s for the PS-only implementation, 22.90 s for the basic PS–PL implementation, and 12.63 s for the proposed PS–PL implementation. These results correspond to a 7.45-fold reduction relative to PS-only execution and a 1.81-fold reduction relative to the basic PS–PL structure. The results confirm that GNN–DRL routing inference can be executed efficiently within the resource constraints of an SoC FPGA.

Author Contributions: Methodology, J.P. and H.L.; Software, J.P. and S.K.; Validation, J.P., H.L. and S.K.; Formal analysis, B.R. and M.H.; Writing—original draft, J.P. and H.L.; Writing—review and editing, S.K., B.R. and M.H. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Agency for Defense Development, funded by the Korean government (Defense Acquisition Program Administration) (UI247034TF).

Data Availability Statement: The original data is not available due to the policy of the funder.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
ARM	Advanced RISC Machine
BRAM	Block Random Access Memory
BUFG	Global Buffer
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DFS	Depth-First Search
DRL	Deep Reinforcement Learning
DQN	Deep Q-Network
DSP	Digital Signal Processor
FF	Flip-Flop
FP16	16-bit Floating Point
FP32	32-bit Floating Point
FPGA	Field-Programmable Gate Array
GEO	Geostationary Earth Orbit
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HLS	High-Level Synthesis
INT4	4-bit Integer
INT8	8-bit Integer
INT16	16-bit Integer
IP	Intellectual Property
ISL	Inter-Satellite Link
KSP	K-Shortest Path
LEO	Low Earth Orbit
LUT	Look-Up Table
LUTRAM	Look-Up Table Random Access Memory
MAC	Multiply–Accumulate
MLP	Multilayer Perceptron
MPNN	Message-Passing Neural Network
MPSoC	Multiprocessor System-on-Chip
NSFNET	National Science Foundation Network
OBC	Onboard Computer
PC	Personal Computer
PL	Programmable Logic
PS	Processing System
PS–PL	Processing System–Programmable Logic
PTQ	Post-Training Quantization
PYNQ	Python Productivity for Zynq
Q-value	Action-value
SELU	Scaled Exponential Linear Unit
SoC	System-on-Chip

References

1. Wang, Q.; Li, W.; Yu, Z.; Abbasi, Q.; Imran, M.; Ansari, S.; Sambo, Y.; Wu, L.; Li, Q.; Zhu, T. An Overview of Emergency Communication Networks. *Remote Sens.* **2023**, *15*, 1595. [CrossRef]
2. Kodheli, O.; Lagunas, E.; Maturo, N.; Sharma, S.K.; Shankar, B.; Mendoza Montoya, J.F.; Merlano Duncan, J.C.; Spano, D.; Chatzinotas, S.; Kisseleff, S.; et al. Satellite Communications in the New Space Era: A Survey and Future Challenges. *IEEE Commun. Surv. Tutor.* **2021**, *23*, 70–109. [CrossRef]
3. Pritchard-Kelly, R.; Costa, J. Low Earth Orbit Satellite Systems: Comparisons with Geostationary and Other Satellite Systems, and Their Significant Advantages. *J. Telecommun. Digit. Econ.* **2022**, *10*, 1–22. [CrossRef]
4. Shayea, I.; El-Saleh, A.A.; Ergen, M.; Saoud, B.; Hartani, R.; Turan, D.; Kabbani, A. Integration of 5G, 6G and IoT with Low Earth Orbit (LEO) Networks: Opportunity, Challenges and Future Trends. *Results Eng.* **2024**, *23*, 102409. [CrossRef]
5. Cao, X.; Li, Y.; Xiong, X.; Wang, J. Dynamic Routings in Satellite Networks: An Overview. *Sensors* **2022**, *22*, 4552. [CrossRef] [PubMed]
6. Qi, X.; Ma, J.; Wu, D.; Liu, L.; Hu, S. A Survey of Routing Techniques for Satellite Networks. *J. Commun. Inf. Netw.* **2016**, *1*, 66–85. [CrossRef]
7. Westphal, C.; Han, L.; Li, R. LEO Satellite Networking Relaunched: Survey and Current Research Challenges. *ITU J. Future Evol. Technol.* **2023**, *4*, 711–744. [CrossRef]
8. Zhu, Z.; Rao, Z.; Xiao, S.; Yao, Y.; Xu, Y.; Meng, W. Intelligent Routing Methods for Low-Earth Orbit Satellite Networks Based on Machine Learning: A Comprehensive Survey. *Ad Hoc Netw.* **2025**, *178*, 103995. [CrossRef]
9. Xu, P.; Feng, M.; Zhou, J.; Xiao, L.; Xiao, P.; Jiang, T. Inter-Satellite Routing for LEO Satellite Networks: A GNN and DRL Integrated Approach. In Proceedings of the 2024 IEEE/CIC International Conference on Communications in China (ICCC), Hangzhou, China, 7–9 August 2024; pp. 1346–1351. [CrossRef]
10. Ding, Z.; Liu, H.; Tian, F.; Yang, Z.; Wang, N. Fast-Convergence Reinforcement Learning for Routing in LEO Satellite Networks. *Sensors* **2023**, *23*, 5180. [CrossRef] [PubMed]
11. Ding, Z.L.; Liu, H.J.; Tian, F.; Yang, Z.J. SDDRL-SR: A High-Reliability Satellite Routing Algorithm Based on Deep Reinforcement Learning. In Proceedings of the 2024 IEEE 99th Vehicular Technology Conference (VTC2024-Spring), Singapore, 24–27 June 2024; pp. 1–6. [CrossRef]
12. Chen, Y.; Cao, H.; Wang, L.; Chen, D.; Liu, Z.; Zhou, Y.; Shi, J. Deep Reinforcement Learning-Based Routing Method for Low Earth Orbit Mega-Constellation Satellite Networks with Service Function Constraints. *Sensors* **2025**, *25*, 1232. [CrossRef] [PubMed]
13. Han, C.; Xiong, W.; Yu, R. Deep Reinforcement Learning-Based Multipath Routing for LEO Megaconstellation Networks. *Electronics* **2024**, *13*, 3054. [CrossRef]
14. Yan, M.; Deng, L.; Hu, X.; Liang, L.; Feng, Y.; Ye, X.; Zhang, Z.; Fan, D.; Xie, Y. HyGCN: A GCN Accelerator with Hybrid Architecture. In Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 22–26 February 2020; pp. 15–29. [CrossRef]
15. Zhang, B.; Kannan, R.; Prasanna, V.K. BoostGCN: A Framework for Optimizing GCN Inference on FPGA. In Proceedings of the 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Orlando, FL, USA, 9–12 May 2021; pp. 29–39. [CrossRef]
16. Tian, T.; Zhao, L.; Wang, X.; Wu, Q.; Yuan, W.; Jin, X. FP-GNN: Adaptive FPGA Accelerator for Graph Neural Networks. *Future Gener. Comput. Syst.* **2022**, *136*, 294–310. [CrossRef]
17. Procaccini, M.; Sahebi, A.; Giorgi, R. A Survey of Graph Convolutional Networks (GCNs) in FPGA-Based Accelerators. *J. Big Data* **2024**, *11*, 163. [CrossRef]
18. Veyette, M.J., Jr.; Aylor, K.; Stafford, D.; Herrera, M.; Jumani, S.; Lineberry, C.; Macklen, C.; Maxwell, E.; Stiles, R.; Jenkins, M. AI/ML for Mission Processing Onboard Satellites. In Proceedings of the AIAA SCITECH 2022 Forum, San Diego, CA, USA and Virtual, 3–7 January 2022. [CrossRef]
19. Ahmad, S.; Boppana, V.; Ganusov, I.; Kathail, V.; Rajagopalan, V.; Wittig, R. A 16-nm Multiprocessing System-on-Chip Field-Programmable Gate Array Platform. *IEEE Micro* **2016**, *36*, 48–62. [CrossRef]
20. Kim, H.; Park, J.; Lee, H.; Won, D.; Han, M. An FPGA-Accelerated CNN with Parallelized Sum Pooling for Onboard Realtime Routing in Dynamic Low-Orbit Satellite Networks. *Electronics* **2024**, *13*, 2280. [CrossRef]
21. Jo, B.; Lee, H.; Roh, B.; Han, M. Efficient Inference of Neural Networks with Cooperative Integer-Only Arithmetic on a SoC FPGA for Onboard LEO Satellite Network Routing. *Aerospace* **2026**, *13*, 277. [CrossRef]
22. Park, J.H.; Lee, H.; Han, M. Parallel Eclipse-Aware Routing on FPGA for SpaceWire-Based OBC in LEO Satellite Networks. *J. Sens. Actuator Netw.* **2025**, *14*, 73. [CrossRef]
23. Kim, H.; Lee, H.; Han, M. GPU-Accelerated Eclipse-Aware Routing for SpaceWire-Based OBC in Low-Earth-Orbit Satellite Networks. *Aerospace* **2025**, *12*, 422. [CrossRef]
24. Gilmer, J.; Schoenholz, S.S.; Riley, P.F.; Vinyals, O.; Dahl, G.E. Neural Message Passing for Quantum Chemistry. In Proceedings of the 34th International Conference on Machine Learning (ICML), Sydney, NSW, Australia, 6–11 August 2017; pp. 1263–1272.

25. Almasan, P.; Suárez-Varela, J.; Rusek, K.; Barlet-Ros, P.; Cabellos-Aparicio, A. Deep Reinforcement Learning Meets Graph Neural Networks: Exploring a Routing Optimization Use Case. *Comput. Commun.* **2022**, *196*, 184–194. [CrossRef]
26. Xing, R.; Xu, M.; Zhou, A.; Li, Q.; Zhang, Y.; Qian, F.; Wang, S. Deciphering the Enigma of Satellite Computing with COTS Devices: Measurement and Analysis. In Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom), Washington, DC, USA, 30 September–4 October 2024; pp. 420–435. [CrossRef]
27. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018; pp. 2704–2713. [CrossRef]
28. Wu, H.; Judd, P.; Zhang, X.; Isaev, M.; Micikevicius, P. Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. *arXiv* **2020**, arXiv:2004.09602. [CrossRef]