aerospace

MDPI

Article

GPU-Accelerated Eclipse-Aware Routing for SpaceWire-Based OBC in Low-Earth-Orbit Satellite Networks

Hyeonwoo Kim [1], Heoncheol Lee [1,2,*] and Myonghun Han [3]

[1] School of Electronic Engineering, Kumoh National Institute of Technology, Gumi 39177, Gyeongbuk, Republic of Korea
[2] Department of IT Convergence Engineering, Kumoh National Institute of Technology, Gumi 39177, Gyeongbuk, Republic of Korea
[3] Agency for Defense Development, Daejeon 34186, Republic of Korea
* Correspondence: hclee@kumoh.ac.kr; Tel.: +82-54-478-7476

Abstract: Low-Earth-Orbit (LEO) satellite networks offer a promising avenue for achieving global connectivity, despite certain technical and economic challenges such as high implementation costs and the complexity of network management. Nonetheless, real-time routing remains challenging because of rapid topology changes and strict energy constraints. This paper proposes a GPU-accelerated Eclipse-Aware Routing (EAR) method that simultaneously minimizes hop count and balances energy consumption for real-time routing on an onboard computer (OBC). The approach first employs a Breadth-First Search (BFS)–based K-Shortest Paths (KSP) algorithm to generate candidate routes and then evaluates battery usage to select the most efficient path. In large-scale networks, the computational load of the KSP search increases substantially. Therefore, CUDA-based parallel processing was integrated to enhance performance, resulting in a speedup of approximately 3.081 times over the conventional CPU-based method. The practical applicability of the proposed method is further validated by successfully updating routing tables in a SpaceWire network.

Keywords: LEO-SN; routing; K-Shortest Path; energy; GPU; OnBoard computer; SpaceWire

check for updates

Academic Editor: M. Reza Emami

Received: 5 March 2025
Revised: 7 May 2025
Accepted: 8 May 2025
Published: 9 May 2025

Citation: 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. https://doi.org/10.3390/aerospace12050422

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

As the global demand for fast and stable communication networks continues to rise, satellite-based networks are gaining attention as a complementary solution to traditional communication infrastructure. In particular, LEO satellite networks, which can effectively cover the entire globe while providing low latency, are emerging as a key technology for next-generation satellite communications [1,2]. LEO satellite networks enable high-speed internet services even in regions with underdeveloped terrestrial infrastructure. Additionally, large-scale satellite deployments, such as mega-constellations, can connect the world into a single network, while advancements in laser communication technology facilitate high-speed data transmission. Due to these advantages, LEO satellites are being utilized across various fields, including disaster relief, logistics tracking, and maritime and aviation communications [3].

Routing algorithms are as essential in LEO satellite networks as they are in terrestrial networks. However, the unique characteristics of LEO satellites make it challenging to directly apply conventional routing algorithms. Compared to Medium-Earth-Orbit (MEO) and Geostationary-Earth-Orbit (GEO) satellites, LEO satellites are positioned closer to the Earth's surface, resulting in reduced latency and lower path loss [4,5]. While these attributes offer advantages, they also introduce significant challenges. To achieve global

coverage, LEO satellite networks require the deployment of hundreds or even thousands of satellites. Furthermore, the high mobility of LEO satellites leads to a highly dynamic network topology, posing a critical challenge in satellite routing research. Due to their continuous movement relative to the Earth, inter-satellite links frequently change, creating a dynamic topology that must be efficiently managed [6–8]. As a result, extensive research is being conducted to develop routing strategies that account for the specific characteristics of LEO satellite networks.

Energy constraints are also a critical factor in LEO satellite networks. When a satellite is exposed to sunlight, it generates energy through solar panels, which is used for data processing and communication while simultaneously charging its onboard battery. However, when the satellite enters the Earth's shadow region, known as an eclipse, it can no longer receive solar energy and must rely solely on stored battery power to maintain its operations. During this period, the number of charge and discharge cycles of the battery is limited, a phenomenon referred to as the Depth of Discharge (DoD) cycle. As DoD increases, the maximum number of charge-discharge cycles decreases, leading to a reduction in the battery's overall lifespan. Therefore, to maintain a low DoD level and prolong battery life, excessive energy consumption must be avoided during eclipse periods [9–11].

This paper proposes an Eclipse-Aware Routing (EAR) approach based on the K-Shortest Path (KSP) algorithm, which incorporates satellite battery status and eclipse conditions into the routing decision process. Unlike conventional shortest path algorithms that determine only a single optimal route, the KSP algorithm identifies a set of K shortest path candidates. This feature is particularly beneficial in LEO satellite networks, where frequent disruptions in inter-satellite links are likely due to high mobility.

The primary goal of this work is to enhance the robustness and energy efficiency of routing under realistic LEO conditions, where dynamic topology changes and energy constraints coexist. The KSP algorithm was selected to support this goal by enabling proactive route planning with multiple alternatives, allowing the system to adapt quickly to link failures and reduce the need for repeated full-route recalculations.

By preemptively securing multiple alternative routes, network disconnections can be effectively mitigated [12]. Subsequently, the routing algorithm selects the optimal path by considering the battery levels of the satellites and their eclipse states. The proposed routing algorithm ensures both accuracy and efficiency while balancing energy consumption across the network. By accounting for eclipse conditions, it helps distribute energy usage more evenly, thereby extending the network's operational lifespan and improving the overall efficiency of satellite operations.

To simulate the operation in an actual satellite environment, the routing algorithm was implemented on an OnBoard Computer (OBC). While it efficiently determined routing paths at relatively small node scales, real-time applications became challenging for large-scale nodes due to computationally intensive operations. To address this, parallel processing was performed using a GPU to enable real-time routing. Rather than redesigning the routing algorithm for reduced complexity, a parallel acceleration strategy was adopted due to the high computational cost of path search. GPU-based execution allows massive parallelism with predictable memory access and minimal synchronization, making it suitable for onboard environments with strict timing requirements and limited CPU resources. In satellite environments, real-time data processing is crucial, and relying solely on traditional CPU-based computation may not ensure high performance [13,14]. This paper enhances real-time routing performance by leveraging GPU-based parallel computation to mitigate bottlenecks in the algorithm, thereby improving the overall performance of the satellite network.

Finally, an experiment was conducted by integrating the routing algorithm with the SpaceWire Brick Mk4, a SpaceWire [15] device for spacecraft communication networks. The experiment involved modifying the routing tables of remote devices according to the paths derived from the algorithm's results [16]. This demonstrated the feasibility of networking in a more realistic satellite environment. The main contributions of this work are as follows:

- Proposing an Eclipse-Aware Routing (EAR) algorithm that considers both satellite battery levels and eclipse conditions for energy-efficient path selection in LEO satellite networks.
- Implementing a BFS-based K-Shortest Path (KSP) algorithm optimized for GPU-based parallel processing to enable real-time routing.
- Demonstrating the feasibility of the proposed method through integration with a SpaceWire-based onboard system and dynamic routing table updates.

2. Problem Description

2.1. Routing Problem in the LEO Network

As mentioned earlier, LEO satellite networks operate in a dynamic topology environment, where inter-satellite connections continuously change over time. Specifically, previously established links may be disconnected, while new links are formed as time progresses. Due to this characteristic, conventional routing techniques used in static networks cannot be directly applied. Instead, a routing approach that can dynamically reflect the real-time connectivity status between satellites is required.

In this paper, routing is performed based on the KSP algorithm. Unlike traditional shortest path algorithms that identify only a single shortest route, the KSP algorithm generates a set of K shortest path candidates and selects the optimal route from them. This approach is particularly advantageous in LEO satellite networks, where inter-satellite connections are highly dynamic and may suddenly become unavailable. By precomputing multiple alternative paths, network stability can be ensured [17,18]. There are various methods for implementing the K-Shortest Path (KSP) algorithm, with Yen's algorithm and Eppstein's algorithm being among the most widely used [19,20]. Beyond these classical approaches, advanced variations have been proposed to improve computational efficiency or enrich path diversity. For example, Katoh et al. introduced a technique that accelerates path enumeration by leveraging advanced data structures, significantly reducing the overhead in listing top-k paths [21]. In a different direction, Liu et al. proposed a generalized framework called K-Shortest Paths with Diversity (KSPD), which incorporates similarity constraints between paths to generate more diverse and meaningful alternatives [22]. These approaches highlight the ongoing efforts to optimize KSP computation for different practical needs.

A BFS-based KSP algorithm was chosen in this study, primarily for its compatibility with GPU-based parallel processing. While algorithms like Yen's and Eppstein's offer theoretical efficiency, they often involve recursive or heap-intensive structures, which are less amenable to parallel execution on architectures such as CUDA. In contrast, the BFS-based approach allows for flat memory access, minimal branching, and straightforward path expansion, making it particularly suitable for implementation on resource-constrained onboard systems.

Future work will explore comparative implementation and benchmarking of these alternative KSP algorithms, including diversity-aware strategies, to better understand the trade-offs between computational complexity, parallelizability, and routing quality in LEO satellite environments.

The BFS-based KSP algorithm implemented in this paper operates by first searching for K shortest paths to the target node and then selecting the optimal path based on specific

criteria. In this paper, the initial path selection is based on the number of hops, and an additional path selection strategy is applied to consider energy consumption.

Additionally, modern onboard networks consist of various elements, each performing different functions while being interconnected through the onboard network communication infrastructure. SpaceWire is one of the widely used technologies in this infrastructure due to its compact structure, efficiency, and ease of understanding and implementation [15].

In particular, SpaceWire-based onboard networks must dynamically reflect changes in network topology in real time through dynamic routing. In this study, a BFS-based KSP algorithm was applied to precompute multiple alternative paths and select the most suitable route for the SpaceWire routing table. By aligning with SpaceWire's routing mechanism, this approach enables the construction of optimal data transmission paths, enhancing both network stability and real-time data transfer performance.

Furthermore, while small-scale SpaceWire networks are relatively simple to design and operate, as the network size increases, traffic management and routing optimization become critical challenges. Therefore, performing thorough simulations before deploying the actual network is essential for verifying network performance and implementing effective routing strategies compatible with the SpaceWire protocol.

2.2. Energy Efficiency Problem in the LEO Network

In LEO satellite networks, satellites recharge their batteries using solar energy. However, when entering an **eclipse** region, they can no longer receive solar energy, leading to a gradual depletion of battery reserves. These energy constraints significantly impact network stability and operational lifespan. If a specific satellite excessively consumes its battery power, it can degrade the overall network performance.

Conventional routing algorithms primarily consider the shortest path or minimum latency; however, they do not account for the energy consumption of individual satellites. As a result, certain satellites may be repeatedly included in routing paths, leading to an imbalance in battery depletion. This imbalance can cause some satellites to drain their batteries prematurely, and in the worst-case scenario, it may result in network disconnection [23].

To address this issue, this paper proposes the EAR approach, which distributes energy consumption more evenly across satellites. This method determines the optimal data transmission path by considering each satellite's eclipse duration, expected transmission load, and energy consumption rate.

Algorithm 1 illustrates the computational flow of the EAR approach. This algorithm selects a routing path that ensures stable data transmission while minimizing energy consumption in LEO satellite networks. First, the algorithm initializes the energy consumption values for transmission, reception, and baseline operations. Then, it searches for K shortest paths available at the given time and calculates the energy consumption for satellites included in each path. To account for eclipse conditions, the algorithm determines whether each satellite is in eclipse and, if so, computes the eclipse duration. It then estimates the expected number of data transmissions during this period. Based on this estimation, the algorithm calculates the total energy consumption by summing the baseline operation energy and transmission/reception energy. Although the Depth of Discharge (DoD) is not explicitly computed, it is indirectly reflected in this energy model—satellites remaining in eclipse for extended periods with high communication load are considered at higher risk of battery depletion. For each path, the satellite with the highest estimated energy usage is identified, and the maximum value among all paths is used as a routing metric. The path with the lowest such value is selected to prevent overuse of energy-critical satellites, thus maintaining balanced energy consumption and avoiding early battery degradation. This approach is similar to previous studies that indirectly estimate DoD based solely on eclipse

duration or reduce DoD risk by excluding or penalizing energy-critical satellites without direct DoD computation [9,10].

Algorithm 1. Eclipse Aware Routing—Pseudo-code representing the Eclipse Aware Routing method.

```
                eclipseMatrix[time][nodes]
                KShortestPaths[]
Input           sampleTime
                NoE, TxE, RxE
                nTimeStep
Output          minPath
1.              Initialize Constants: NoE, TxE, RxE
2.              Initialize Data Structures: expectedConsumedEnergy[][] <= 0;
3.
4.              For pathIndex = 0 ~ topPath
5.                testingPath <= shortestPaths[pathIndex];
6.                For satIndex = 0 ~ num_sats_in_path
7.                  testingSat <= testingPath[satIndex];
8.
9.                  If eclipseMatrix[timeIndex, testing Sat] == 1
10.                   Determine eclipseStartIndex and eclipseEndIndex;
11.                   eclipseTime = (eclipseEndIndex − eclipseStartIndex) * sampleTime;
12.                   expectedTxNum = round(eclipseTime * randi(10, 1, 1) * 0.01);
13.
14.                   Compute Energy Consumption:
15.                     NominalE = eclipseTime * NoE;
16.                     CommunicationE = (expectedNum + 1) * (TxE + RxE)
17.                     ExpectedConsumedEnergy[pathIndex, satIndex] = NominalE +
18.             CommunicationE;
19.               End if
20.             End for
21.             End for
22.
23.             Compute Maximum Energy Consumtion per Path:
24.             For pathIndex = 0 ~ topPath
25.               maxEnergy[pathIndex] = max(ExpectedConsumedEnergy[pathInex]);
26.             End for
27.
28.             Select Optimal Path:
29.             minPath <= pathIndex with min(maxEnergy);
30.
31.             Return minPath;
```

2.3. Overall Structure of the Algorithm and the Need for Parallelization

In this paper, a BFS-based KSP algorithm is combined with EAR to implement an efficient routing method for LEO satellite networks. The overall flow of the implemented algorithm is illustrated in Figure 1 and consists of three main stages: graph initialization, KSP search, and energy consumption calculation.

First, to perform routing, the algorithm initializes a graph that represents the connectivity of the satellite network. The input data is stored in a text file (.txt) format and

includes the total number of satellites, the source node, the destination node, the number of paths to search (K), and inter-satellite connection information. Based on this data, the algorithm updates the node and edge information to construct the graph, preparing it for the subsequent KSP search.

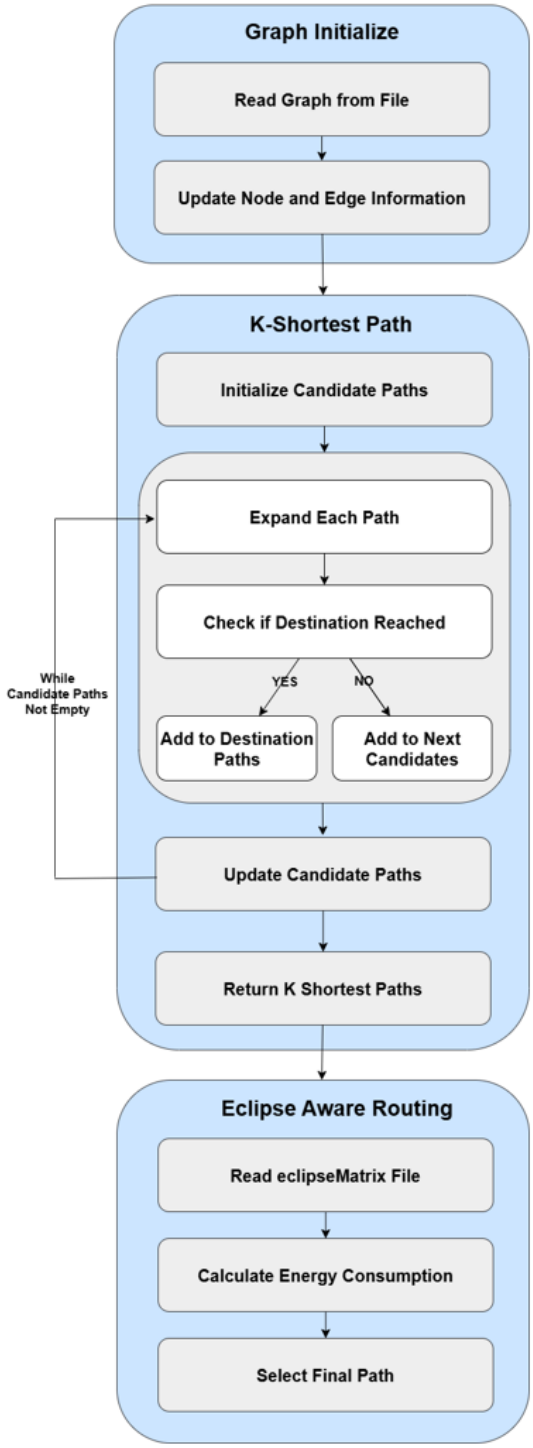

Figure 1. Block diagram of a conventional Eclipse-Aware Routing algorithm.

In the next step, the algorithm performs the KSP search based on BFS to find K shortest paths. Starting from the source node, the BFS search expands along the paths, and each candidate path is managed as a list. When a path reaches the destination node, it is sorted

based on the path cost, and the top K paths are selected. Finally, the K candidate paths are returned, and the optimal path is determined through energy consumption analysis.

Finally, to select the most energy-efficient path among the K shortest paths, the algorithm analyzes the energy consumption of each satellite, as previously described. First, the eclipseMatrix file is loaded to determine whether each satellite is in an eclipse state at a given time. Then, the expected energy consumption of each satellite is calculated based on the energy required for transmission, reception, and baseline operations. In particular, the battery depletion is predicted by considering the duration of the eclipse. The algorithm evaluates the maximum energy consumption for each path and ultimately selects the path that minimizes energy usage.

The performance evaluation results of applying this algorithm to an OBC simulating a LEO satellite network environment are presented in Figure 2. To analyze the execution time of the code, the number of satellites was set to 150, and the number of shortest paths to search (K) was set to 15 for profiling. The analysis revealed that the KSP search process accounted for 92.53% of the total execution time. In contrast, the graph initialization and energy consumption calculation processes incurred relatively lower computational costs.

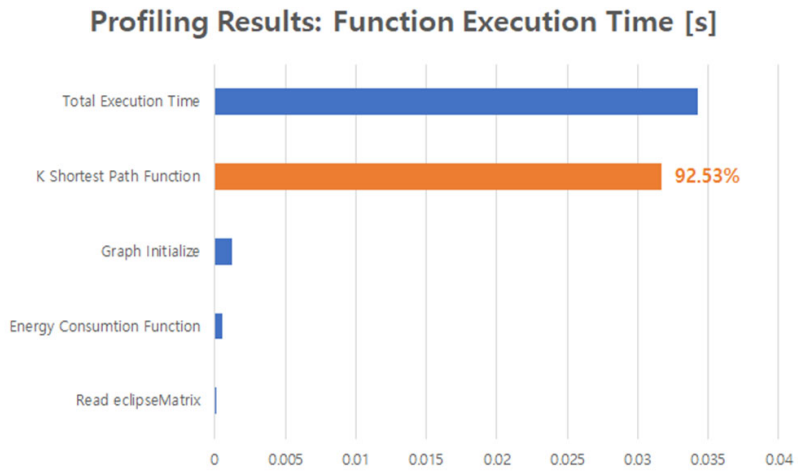

Figure 2. Profiling results of the conventional Eclipse-Aware Routing algorithm.

Therefore, this paper identifies the KSP search process as the primary bottleneck determining the overall algorithm's performance. To optimize this process, a GPU-based parallelization approach is proposed. By leveraging the parallel computing capabilities of the GPU, tasks such as path expansion and candidate path management can be executed more efficiently. This optimization enhances the path search speed compared to conventional computation methods.

3. Proposed Method

3.1. BFS-Based K-Shortest Path Algorithm

To perform efficient path search in a LEO satellite network environment, a BFS-based KSP algorithm was applied. BFS operates by expanding adjacent nodes step by step from the source node, making it suitable for finding K shortest paths. The algorithm implemented in this paper searches for K shortest paths from the source node to the destination node, and Algorithm 2 illustrates the overall computational flow of this process.

First, an initial candidate path list is created, including only the source node, while the list of discovered paths reaching the destination is initialized as empty. Then, as long as the candidate path list is not empty and the number of discovered paths reaching the destination remains below K, the algorithm performs BFS-based exploration. During the search, the algorithm expands paths based on the last node of the currently explored

path, identifying adjacent nodes. To prevent cycles, nodes already included in the path are excluded. If an expanded path reaches the destination, it is added to the discovered path list; otherwise, it is stored in the next candidate path list for further expansion in the subsequent iteration. Once the search concludes, as shown in line 22, the candidate path list is updated with the next candidate paths, and this process repeats until the optimal paths are found. Finally, the paths that reach the destination are stored in the discovered path list. After the search is complete, the paths are sorted based on their costs, and the K paths with the lowest costs are selected and returned.

To provide a clearer explanation of the BFS-based KSP search process, Figure 3 is used to analyze the path exploration procedure, representing a portion of the actual graph used in this study's experiments. The gray node represents the starting node where the search begins, assigned as level 0, while the blue nodes indicate adjacent nodes expanded in the first step, corresponding to level 1. The yellow nodes represent nodes further expanded in the second step, forming candidate paths to the destination, with the numbers in the upper right corner of each node denoting the order in which they were explored. In BFS-based search, the exploration expands level by level from the starting node, forming paths while preventing cycles to ensure optimal route selection. Due to the nature of BFS, all possible paths are explored while expansion occurs based on the shortest distance, making it an efficient method for finding the K shortest paths.

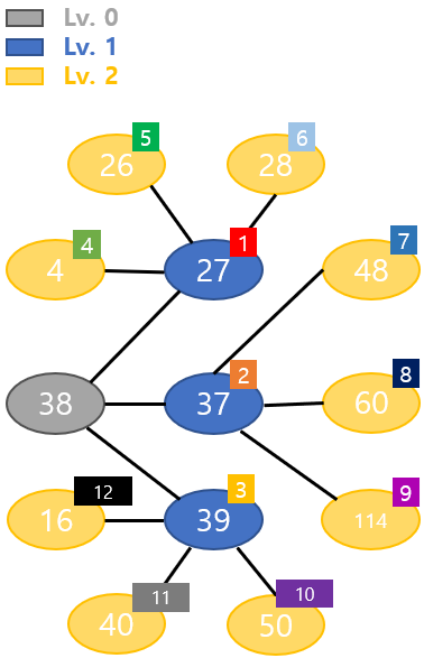

Figure 3. Example graph representing sequential KSP search.

However, the BFS-based KSP algorithm has inherent limitations due to its sequential processing nature. Since it expands paths step by step, the search time can become significantly longer when executed sequentially. This issue becomes more pronounced in large-scale satellite networks with a high number of nodes and an increased number of paths to explore, leading to a rapid increase in computational time. As the BFS search process requires managing and expanding all candidate paths, the computational complexity grows exponentially, potentially affecting system performance. To address this bottleneck, this paper proposes a GPU-based parallel computing approach to accelerate the BFS-based KSP search. The details of this approach are discussed in the following section.

Algorithm 2. BFS-based K-Shortest Path Algorithm in CPU—Pseudo-code representing the BFS-based K-Shortest Path algorithm in CPU.

```
            graph_adj[]
Input       src, dst, k
            num_nodes
Output      k_paths[]
1.          Initialize candidate_paths with {src}
2.          Initialize destination paths as empty list
3.
4.          While candidate_paths is not empty AND size(destination_paths) < k
5.            Initialize next_candidate_paths as empty list
6.
7.            For each path in candidate_paths
8.              last_node = last node in path
9.              For each neighbor in graph_adj[last_node]
10.               If neighbor is NOT in path (prevent cycles)
11.                 new_path = path + neighbor
12.                 new_path_cost = path.cost + edge_cost(last_node, neighbor)
13.
14.                 If neighbor == dst
15.                   Add new_path to destination_paths
16.                 Else
17.                   Add new_path to next_candidate_paths
18.                 End
19.               End
20.             End
21.
22.           candidate_paths = next_candidate_paths
23.         End
24.
25.         Sort destination_paths by cost
26.         Return K paths from destination_paths
```

3.2. Graph Representation

To enable efficient graph search in a GPU environment, this study applies the Compressed Sparse Row (CSR) transformation technique. While the traditional adjacency list approach used in CPU-based implementations is intuitive and memory-efficient, it can lead to performance degradation in GPU-based parallel searches due to non-contiguous memory access. In contrast, the CSR method converts graph connectivity information into a continuous array format, making it an optimized graph representation for efficient execution on GPUs [24].

In CPU implementations, graphs are typically stored using a 2D vector structure, where each node maintains a variable-sized vector for its adjacency list. However, this approach can lead to inefficient memory access patterns when performing searches on a GPU. Specifically, since each node's adjacency list varies in size, memory fragmentation occurs, and the parallel search process suffers from random memory access, resulting in inevitable performance degradation [25]. To address this issue, the CSR (Compressed Sparse Row) technique is applied, converting the graph into three contiguous arrays, optimizing it for efficient memory access in a GPU environment.

In the CSR format, a graph is stored using three arrays, as illustrated in Figure 4. The h_offsets array stores the starting position of each node's adjacency list, allowing quick determination of where a specific node's adjacency list begins in the h_edges array. The h_edges array contains all adjacency lists in a contiguous format, storing the destination nodes of all edges, so scanning from h_offsets[i] to h_offsets[i + 1] enables efficient retrieval of all adjacent nodes for a given node. The h_weights array, maintaining the same order as h_edges, stores the edge weights, ensuring quick access to the weight information corresponding to each adjacent node. By applying CSR transformation, the memory inefficiencies and challenges of parallel search encountered in the traditional adjacency list method are resolved, enabling fast and optimized path searching in a GPU environment.

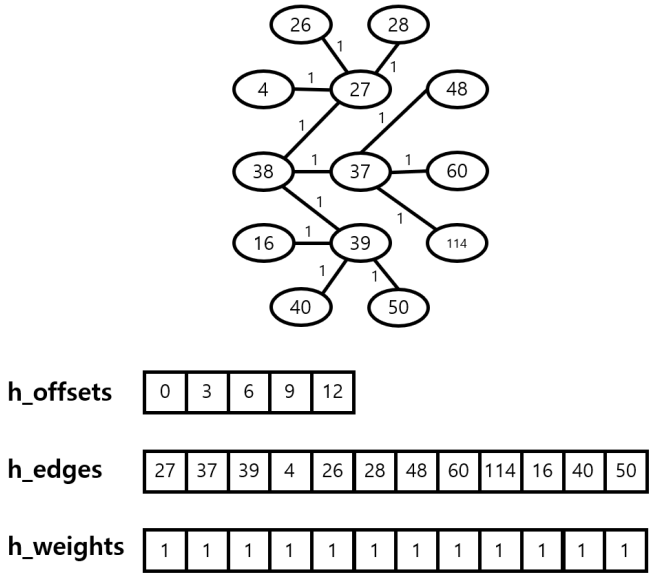

Figure 4. CSR format conversion example.

3.3. Configuration of the CUDA Kernel

To parallelize KSP search, this study proposes a CUDA-based BFS search method. In conventional CPU-based BFS search, candidate paths must be expanded sequentially, leading to increased search time as the number of nodes or K grows. In contrast, by leveraging CUDA, multiple candidate paths can be expanded in parallel, significantly improving search speed.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and API developed by NVIDIA, designed to efficiently handle large-scale computations by utilizing GPUs. Unlike traditional CPU computations, which execute tasks sequentially across a limited number of cores, GPUs leverage thousands of cores to perform parallel processing, making them highly effective for graph-based computations such as BFS search [26]. In this study, CUDA is employed to parallelize KSP search, enabling a more efficient exploration process compared to conventional CPU-based methods.

The overall workflow of the proposed method can be explained using the block diagram presented in Figure 5. Similar to the CPU-based approach, the process consists of three main stages: graph initialization, KSP search, and energy consumption calculation. First, the CPU reads the graph data, converts it into the CSR format as previously described, and transfers it to the GPU. In the KSP search stage, instead of performing sequential expansion, a CUDA kernel is executed, enabling parallel expansion of candidate paths. Each thread is responsible for handling a single candidate path, checking its adjacent nodes, and generating new candidate paths. Finally, the eclipse state and energy consumption of

the satellites are calculated in a similar manner to the CPU-based approach, and the most optimal path is selected.

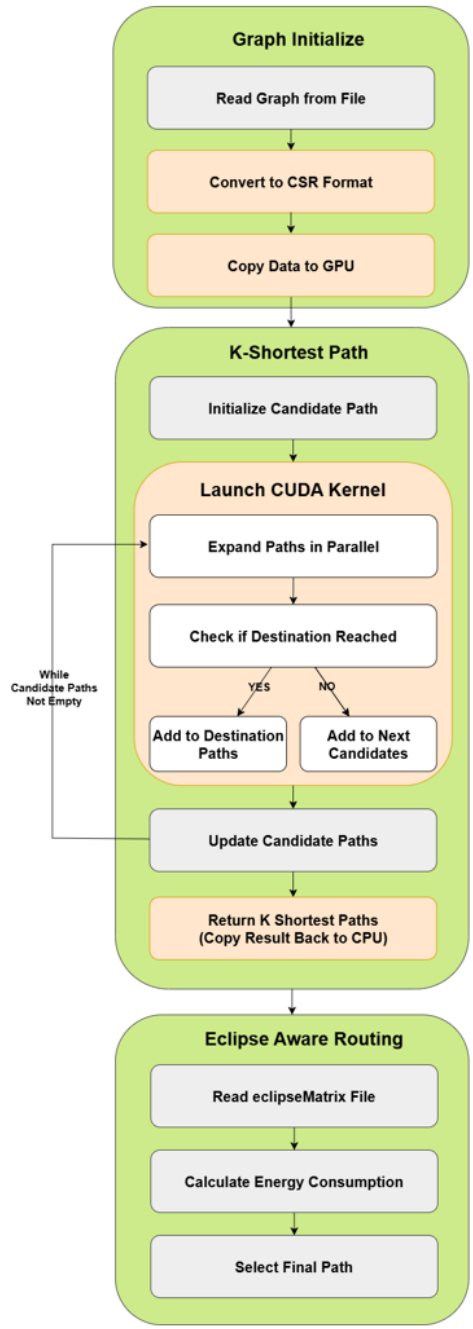

Figure 5. Block diagram of GPU-accelerated Eclipse-Aware Routing algorithm.

A more detailed execution process follows the pseudo code in Algorithm 3. The CUDA-based BFS search operates similarly to the CPU-based BFS, but the key difference is that candidate path expansion is performed in parallel. In CPU-based BFS, as shown in Figure 3, paths are expanded one at a time, causing the computation time to increase exponentially as the number of candidate paths grows. In contrast, during CUDA kernel execution, each candidate path is expanded in parallel across multiple threads. Each thread locates the adjacent nodes of the last node in its assigned path using the CSR format, then generates new candidate paths accordingly. The CSR format optimizes memory access and prevents cycle formation by ensuring that only nodes not already included in the path are added. Since multiple threads modify the candidate path list simultaneously, atomic

operations are applied to maintain data consistency. This prevents synchronization issues when adding new paths, ensuring safe and efficient search execution. Once the search is complete, the destination list is copied back to the CPU, where the shortest paths are sorted, and the top K paths are selected and returned.

Algorithm 3. BFS-based K-Shortest Path Algorithm in GPU—Pseudo-code representing the BFS-based K-Shortest Path algorithm in GPU.

```
          graph_offsets[], graph_edges[]
Input     src, dst, k
          num_nodes
Output    k_paths[]
1.        Copy graph data to GPU memory
2.        Initialize device candidate_paths with {src}
3.        Initialize device destination_paths as empty list
4.        Initialize device new_candidates as empty list
5.
6.        While candidate_paths is not empty AND size(destination_paths) < k
7.          Set new_candidate_count = 0
8.
9.          Launch CUDA kernel:
10.           For each path in candidate_paths (Parallelized by threadIdx.x)
11.             last_node = last node in path
12.             start_idx = graph_offsets[last_node]
13.             end_idx = graph_offsets[last_node + 1]
14.
15.             For i = start_idx to end_idx
16.               neighbor = graph_edges[i]
17.
18.               If neighbor is NOT in path (prevent cycles)
19.                 new_path = path + neighbor
20.                 new_path_cost = path.cost + edge_cost(last_node, neighbor)
21.
22.                 If neighbor == dst
23.                   AtomicAdd(destination_paths_count)
24.                   Store new_path in destination_paths
25.                 Else
26.                   AtomicAdd(new_candidate_count)
27.                   Store new_path in new_candidates
28.                 End
29.               End
30.             End
31.           End
32.         End
33.
34.         Swap candidate_paths with new_candidates
35.       End
36.
37.       Copy destination_paths to CPU memory
38.       Sort destination_paths by cost
39.       Return K paths from destination_paths
```

Figure 6 clearly illustrates the difference between CPU-based and CUDA-based search methods. As shown in Figure 3, the conventional CPU-based BFS search expands nodes sequentially, processing each node one at a time. Specifically, in level 0 (starting node), nodes in level 1 (e.g., 27, 37, 39) are explored one by one, followed by the expansion of level 2 nodes in the same manner. In contrast, the proposed CUDA-based BFS search expands all nodes in level 1 simultaneously, followed by a parallel expansion of all level 2 nodes. Since each level's exploration is performed in parallel, the search time is significantly reduced, allowing the K shortest paths to be found much faster. This confirms that CUDA-based KSP search operates more efficiently, particularly in large-scale graph search problems.

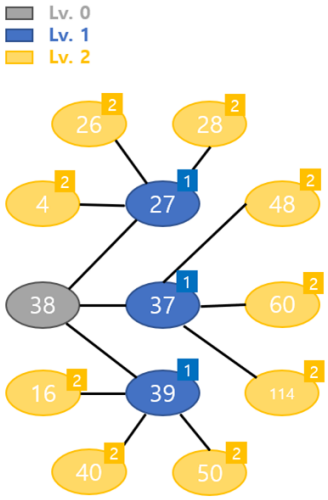

Figure 6. Example graph for proposed parallelized KSP search.

4. Experimental Results

To ground the simulations in realistic orbital dynamics, real TLE (Two-Line Element) data were used to model a small-scale LEO network, capturing genuine orbital motion and link availability patterns. For larger-scale evaluations, where comprehensive public TLE data are not available, synthetic constellations were generated. In these synthetic scenarios, each satellite was assigned 3–5 plausible inter-satellite links based on typical orbital spacing and connectivity patterns, preserving the essential dynamics of real LEO deployments. This hybrid approach balances empirical realism with scalability for assessing routing performance across network sizes.

4.1. Experimental Setups

In this paper, experiments on satellite network routing were conducted using CUDA-based KSP search and the EAR algorithm. The overall structure of the experimental setup is illustrated in Figure 7, while Figure 8 shows the actual equipment used in the experiment. The experimental environment consisted of an OBC device with both CPU and GPU, two SpaceWire Brick Mk4 devices, and a PC utilizing only a CPU.

The OBC with CPU and GPU was responsible for executing the routing algorithm and measuring performance. Additionally, it was used to modify the routing tables of remote devices via the SpaceWire network. Detailed experimental results related to SpaceWire are provided in Section 4.3. The experiment was conducted by first generating candidate paths through a BFS-based KSP search and then selecting the optimal route using the EAR algorithm.

The OBC with CPU and GPU employed in this experiment was the Jetson Xavier NX, which was used to simulate a satellite OBC with limited computational resources. As it is equipped with an NVIDIA GPU, it enabled parallel processing using CUDA. The PC

with only a CPU functioned as the receiving device, supplying power to the remote router and verifying the correctness of routing table modifications through the SpaceWire GUI. Additionally, the PC was used to validate the accuracy of the EAR algorithm by comparing its results with those obtained in the OBC environment. Since the accuracy validation involved ensuring that both environments produced the same results, the performance of the PC was not a critical factor in this experiment, and its specifications are therefore omitted.

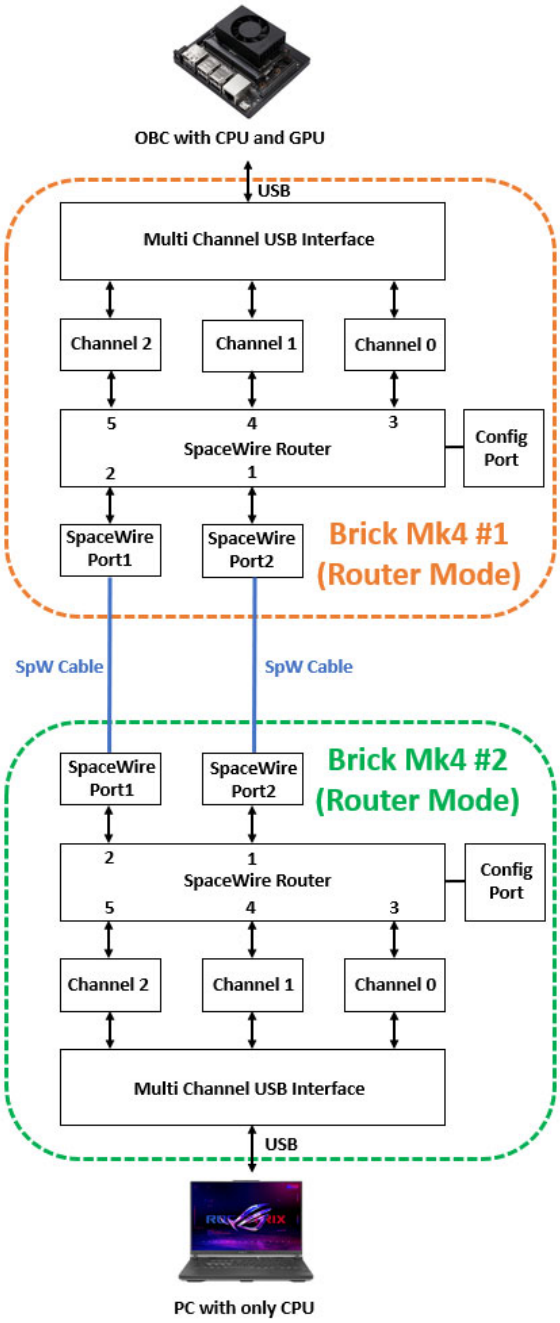

Figure 7. Structure for experimental environments.

Table 1 summarizes the key specifications of the Jetson Xavier NX. To maximize performance, the device was configured in 20 W 6-core mode. It is equipped with an NVIDIA Volta GPU and a Carmel ARM CPU, enabling parallel computation using CUDA. Additionally, it supports 16 GB of LPDDR4x memory, providing a suitable environment for large-scale graph search computations.

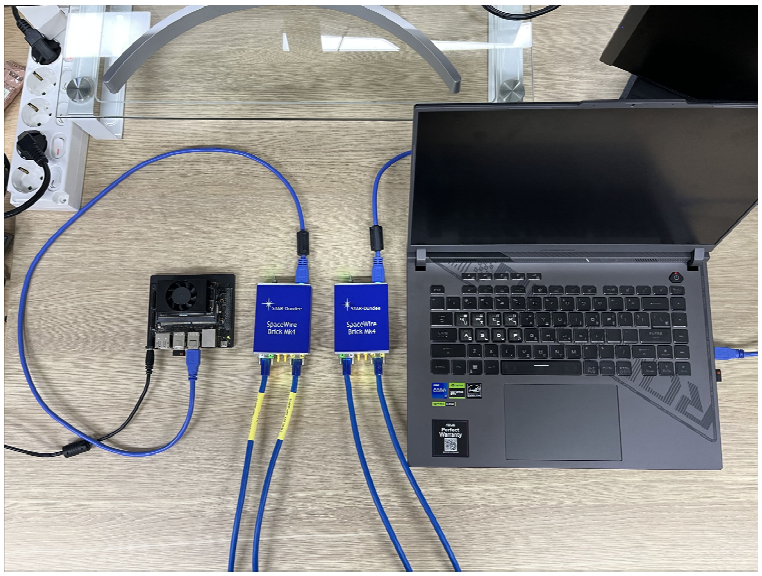

Figure 8. Configured real experimental environment.

Table 1. NVIDIA Jetson Xavier NX 16 GB board technical specification.

MODE	20 W 6CORE
CUDA Version	10.2v
AI Performance	21TOPS
GPU	384-core NVIDIA Volta GPU (48 Tensor Cores)
CPU	6CORE NVIDIA Carmel ARM v8.2 64-bit
Memory	16 GB 128-bit LPDDR4x (59.7 GB/s)
Power consumption	10 W–20 W

In this paper, the computational resources of the Jetson Xavier NX were utilized to evaluate the performance of CUDA-based KSP search and the EAR algorithm. The experimental results demonstrated that real-time routing is feasible even in constrained satellite environments.

4.2. Eclipse-Aware Routing Results

The proposed EAR algorithm was initially implemented in MATLAB 2024a and later ported to C/C++ for execution in an OBC environment. The implementation was further optimized using CUDA for GPU acceleration. This section presents a comparison between the execution results obtained in the PC using only a CPU environment and those from GPU-based execution on the OBC with both CPU and GPU. This comparison serves to validate the algorithm's correctness and analyze its execution time. For the experiments, the number of nodes was set to 150, and the number of candidate paths K was set to 15. Additionally, the weights between nodes were uniformly set to 1. The reason for configuring a relatively small-scale network was to facilitate a direct comparison between a PC with only a CPU and the OBC environment for verification.

Figure 9 presents the experimental results obtained in PC with only CPU and OBC with CPU and GPU environments. The table on the left shows the results from the PC with only CPU, while the table on the right displays the results from the OBC with CPU and GPU. The first path, which has a cost of 6, is identical in both environments. However, it can be observed that the ordering of paths with a cost of 8 differs between the two implementations. This discrepancy arises due to implementation-specific factors, but as

indicated by the arrows in the center of the figure, the candidate paths remain the same, ensuring correctness. To rigorously verify that both environments yield identical results, an exhaustive evaluation was conducted for all paths with a cost of 8 using the K-shortest path (KSP) algorithm. Similarly, for paths with a cost of 10, the ordering difference can result in a variation in the 15th path. The EAR algorithm is applied after extracting the top K shortest path candidates through KSP exploration.

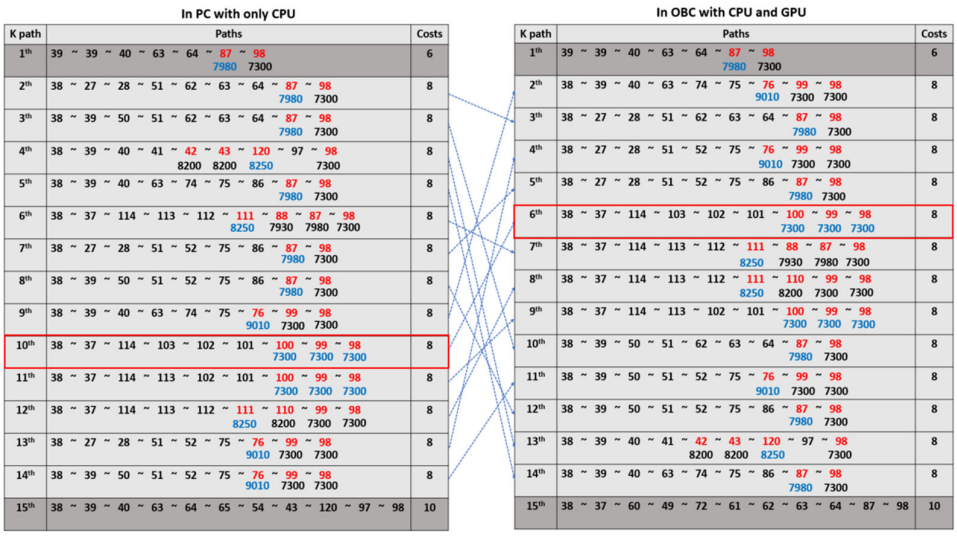

Figure 9. Validation by comparing execution results PC with only CPU and OBC with CPU and GPU.

In Figure 9, the nodes highlighted in red indicate eclipse nodes, identified using the eclipseMatrix. As described in Section 2.2, the EAR algorithm selects eclipse nodes from each path and estimates their expected power consumption. It then determines the satellites consuming the highest power along each path. The numbers below the red-marked nodes represent the estimated power consumption. The satellites with the highest power consumption in each path are highlighted in blue. Ultimately, the path with the lowest maximum energy consumption among these blue-highlighted satellites is selected as the final route. Consequently, the 10th path is selected in PC with only CPU, whereas the 6th path is chosen in OBC with CPU and GPU. However, as explained earlier, this is merely a difference in ordering, and both implementations ultimately select the same path. This approach is designed to reduce the burden on satellites experiencing severe battery depletion while optimizing the overall energy balance within the network.

Figure 10 compares the total execution time of the GPU-accelerated algorithm using CUDA and the traditional CPU-based algorithm. To ensure a fair comparison, the execution time was measured on the same OBC device, evaluating two cases: one using only the CPU and the other utilizing both the CPU and GPU for parallel execution. The gray bars represent the execution time when using only the CPU, while the green bars show the execution time when utilizing the GPU with CUDA. When evaluating networks with 1000 to 4000 nodes, the CUDA-based implementation is observed to be slower than the CPU-based implementation. This performance degradation is attributed to differences in the two implementations. In the CUDA implementation, additional steps such as CSR transformation and data copying introduce overhead. Specifically, CUDA requires data transfers between the host (CPU) and the device (GPU), which are handled using the cudaMemcpy function. Since the CPU and GPU operate in separate memory spaces, this data transfer is essential but incurs significant memory overhead. In smaller network environments, this overhead can have a substantial impact on performance. However, when the network size was increased to 5000 nodes, the CUDA-based algorithm achieved

a 3.081× speedup compared to the CPU-based algorithm. This result demonstrates that CUDA-based KSP exploration is highly effective in performing parallel computations in large-scale graph environments.

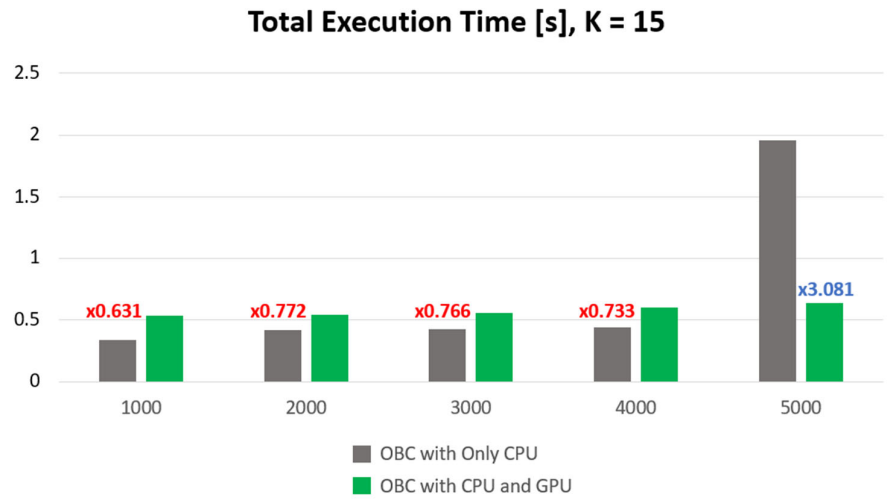

Figure 10. Compare total execution time of OBC with Only CPU vs. OBC with CPU and GPU.

4.3. SpaceWire-Based Experiment Results

In this paper, SpaceWire, a high-speed serial data communication protocol developed by the European Space Agency (ESA), is utilized for satellite and aerospace systems. SpaceWire is widely adopted due to its low latency, high reliability, and real-time data transmission capabilities, making it well-suited for inter-satellite communication. For this study, experiments were conducted using the SpaceWire Brick Mk4 device developed by STAR-Dundee. The SpaceWire Brick Mk4 is a USB-based SpaceWire interface device that facilitates data transmission between a PC and a SpaceWire network, making it a valuable tool for satellite system development and testing. Notably, it supports the Remote Memory Access Protocol (RMAP), enabling direct access to the memory of remote devices, thereby allowing dynamic modification of routing tables. Using this capability, the proposed algorithm was tested by selecting the optimal path and updating the routing tables of remote devices via the SpaceWire network.

The experimental scenario is described as follows. Figure 11 illustrates the process of the experiment. As shown in Figure 9, the EAR algorithm ultimately selects an optimal path. However, if only the KSP algorithm were applied without EAR, the shortest hop-count path would be chosen as the optimal route. In this case, assuming the receiving router is at node 38, the next node for data transmission would be node 39. However, when applying the EAR algorithm, which considers satellite battery status and eclipse conditions, a more energy-efficient route can be derived compared to the conventional shortest path approach. Consequently, instead of forwarding data to node 39, routing through node 37 becomes a more efficient choice. Based on this observation, the experiment was designed to assume node 38 as the receiving router and modify the routing table to direct data transmission to node 37.

Due to the limited availability of only two SpaceWire Brick Mk4 devices, replicating a fully operational satellite network was not feasible. Therefore, certain assumptions were made to conduct the experiment, and understanding these constraints is requested.

Figure 11 provides an overview of the internal structure of the SpaceWire Brick Mk4 device. The integrated SpaceWire router features a total of six ports, with port 0 designated as the configuration port and thus unavailable for use. This leaves five selectable ports for experimentation. In this study, port 4 was assumed to be connected to node 37, and the

routing table of the remote device was modified to direct traffic through port 4 according to the EAR algorithm's results.

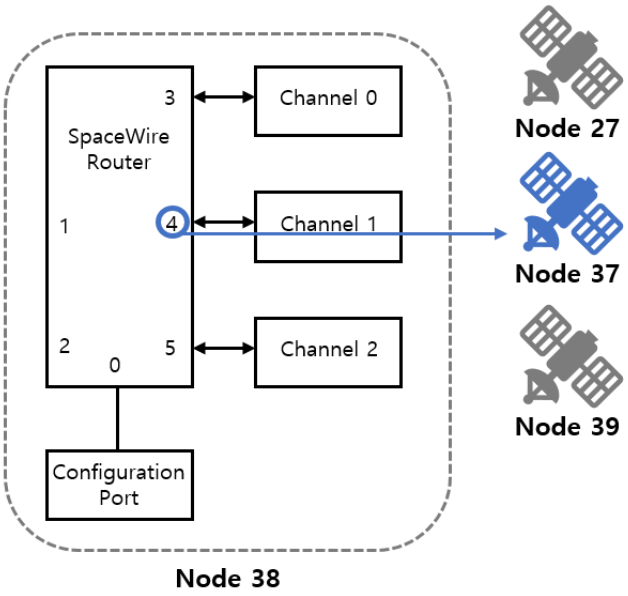

Figure 11. SpaceWire Brick Mk4 internal structure and experimental scenarios.

Figure 12 presents the SpaceWire GUI output, captured from a laptop connected to the receiving SpaceWire Brick Mk4. The experimental results confirmed that the routing table was successfully updated to direct data through port 4, as per the designed scenario, ensuring that data was correctly delivered to node 37. These findings demonstrate that the proposed accelerated EAR routing method is applicable within SpaceWire networks and is capable of dynamically adjusting remote device routing in real time.

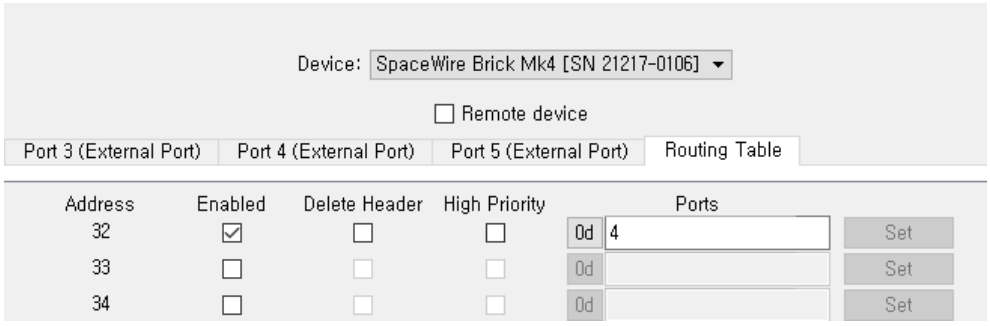

Figure 12. SpaceWire Device Configuration GUI screen viewed from the receiver's laptop.

5. Conclusions

This paper proposed a GPU-accelerated Eclipse-Aware Routing method that minimizes hop count while balancing energy consumption for real-time routing on an onboard computer to solve the problems of dynamic topology changes and energy constraints in LEO satellite networks. The proposed method finds the most efficient route based on not only hop counts but also satellite battery status and eclipse duration. Furthermore, to solve the real-time problem in the conventional method based on CPUs, the proposed method has been implemented on both a CPU and a GPU for parallel processing, which can reduce the computation time significantly. Experimental results showed that the GPU-based implementation operates up to 3.081 times faster than the CPU-based approach while maintaining routing accuracy, as confirmed by the consistency of results between the PC and OBC environments. Furthermore, an experiment using a SpaceWire-based communication

system demonstrated that the proposed algorithm can dynamically update routing tables in onboard networks, validating its feasibility in realistic satellite communication scenarios. In future works, research on extending the proposed method to include not only LEO satellites but also MEO and GEO satellites for more efficient network routing.

Author Contributions: Conceptualization, H.L. and M.H.; methodology, H.K. and H.L.; software, H.K.; validation, H.K. and H.L.; formal analysis, H.L.; investigation, M.H.; writing—original draft preparation, H.K.; writing—review and editing, H.L. 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) (UI237032TG).

Data Availability Statement: Data are contained within the article.

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

References

1. Hassan, N.U.L.; Huang, C.; Yuen, C.; Ahmad, A.; Zhang, Y. Dense Small Satellite Networks for Modern Terrestrial Communication Systems: Benefits, Infrastructure, and Technologies. *IEEE Wirel. Commun.* **2020**, *27*, 96–103. [CrossRef]
2. Cheng, H.; Xu, Z.; Guo, X.; Yang, J.; Xu, K.; Liu, S.; Jin, Z.; Jin, X. Research on Routing Equalization Algorithm of Inter-Satellite Partition for Low-Orbit Micro-Satellites. *Future Internet* **2022**, *14*, 207. [CrossRef]
3. Darwish, T.; Kurt, G.K.; Yanikomeroglu, H.; Bellemare, M.; Lamontagne, G. LEO Satellites in 5G and Beyond Networks: A Review from a Standardization Perspective. *IEEE Access* **2022**, *10*, 35040–35060. [CrossRef]
4. Xiao, Y.; Zhang, T.; Shi, D.; Liu, F. A LEO Satellite Network Capacity Model for Topology and Routing Algorithm Analysis. In Proceedings of the 14th International Wireless Communications & Mobile Computing Conference (IWCMC), Limassol, Cyprus, 25–29 June 2018; pp. 1431–1436. [CrossRef]
5. Ferreira, A.; Galtier, J.; Penna, P. Topological Design, Routing, and Handover in Satellite Networks. In *Handbook of Wireless Networks and Mobile Computing*; Wiley: Hoboken, NJ, USA, 2002; pp. 473–493. [CrossRef]
6. Gounder, V.V.; Prakash, R.; Abu-Amara, H. Routing in LEO-Based Satellite Networks. In Proceedings of the 1999 IEEE Emerging Technologies Symposium, Wireless Communications and Systems (IEEE Cat. No.99EX297), Richardson, TX, USA, 18–21 October 1999; pp. 22.1–22.6. [CrossRef]
7. Yang, Y.; Guo, X.; Xu, Z.; Zhu, Y.; Yi, X. Design and Implementation of LEO Micro-Satellite Network Routing Protocol Based on Virtual Topology. In Proceedings of the 2021 IEEE 5th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Xi'an, China, 15–17 October 2021; pp. 1490–1496. [CrossRef]
8. Tan, H.; Zhu, L. A Novel Routing Algorithm Based on Virtual Topology Snapshot in LEO Satellite Networks. In Proceedings of the 2014 IEEE 17th International Conference on Computational Science and Engineering, Chengdu, China, 19–21 December 2014; pp. 357–361. [CrossRef]
9. Mota Macambira, R.N.; Carvalho, C.B.; Rezende, J.F. Energy-Efficient Routing in LEO Satellite Networks for Extending Satellites Lifetime. *Comput. Commun.* **2022**, *195*, 463–475. [CrossRef]
10. Hussein, M.; Jakllari, G.; Paillassa, B. On Routing for Extending Satellite Service Life in LEO Satellite Networks. In Proceedings of the 2014 IEEE Global Communications Conference, Austin, TX, USA, 8–12 December 2014; pp. 2832–2837. [CrossRef]
11. Shergill, M.; Thompson, Z.; Song, G.; Zhu, T. Energy Efficient LoRaWAN in LEO Satellites. *arXiv* **2024**, arXiv:2412.20660. [CrossRef]
12. Huang, G.; Lu, W.; Jidong, X.; Zhang, G. Improved Route Selection Strategy Based on K Shortest Path. In Proceedings of the 2019 International Symposium on Networks, Computers and Communications (ISNCC), Istanbul, Turkey, 18–20 June 2019; pp. 1–4. [CrossRef]
13. Cao, J.; Zhang, S.; Chen, Q.; Wang, H.; Wang, M.; Liu, N. Computing-Aware Routing for LEO Satellite Networks: A Transmission and Computation Integration Approach. *IEEE Trans. Veh. Technol.* **2023**, *72*, 16607–16623. [CrossRef]
14. Yu, C.; Kim, D.; Lee, H.; Han, M. GPU-Accelerated CNN Inference for Onboard DQN-Based Routing in Dynamic LEO Satellite Networks. *Aerospace* **2024**, *11*, 1028. [CrossRef]
15. *ECSS-E-ST-50-12C*; SpaceWire Standard. ECSS-Space Engineering. SpaceWire-Links Nodes Routers and Networks. Publications Division ESTEC: Noordwijk, The Netherlands, 2008; pp. 11–12.
16. STAR-Dundee. SpaceWire Brick Mk4 Datasheet. Available online: https://www.star-dundee.com/wp-content/star_uploads/product_resources/datasheets/SpaceWire-Brick-Mk4.pdf (accessed on 7 May 2025).

17. Pizzi, S.; Rinaldi, F.; Iera, A.; Molinaro, A.; Araniti, G. Tackling Satellite Mobility in LEO-Based Non-Terrestrial Networks: Principles and Enhancements of Feeder Link Switch. *IEEE Veh. Technol. Mag.* **2024**, *19*, 83–91. [CrossRef]
18. Xiao, Z.; Yang, J.; Mao, T.; Xu, C.; Zhang, R.; Han, Z.; Xia, X.G. LEO Satellite Access Network (LEO-SAN) Toward 6G: Challenges and Approaches. *IEEE Wirel. Commun.* **2024**, *31*, 89–96. [CrossRef]
19. Yen, J.Y. Finding the k Shortest Loopless Paths in a Network. *Manag. Sci.* **1971**, *17*, 712–716. [CrossRef]
20. Eppstein, D. Finding the k Shortest Paths. *SIAM J. Comput.* **1998**, *28*, 652–673. [CrossRef]
21. Katoh, N.; Ibaraki, T.; Mine, H. An Efficient Algorithm for K Shortest Simple Paths. *ACM Trans. Algorithms* **2007**, *3*, 41. [CrossRef]
22. Liu, H.; Jin, C.; Yang, B.; Zhou, A. Finding Top-k Shortest Paths with Diversity. *IEEE Trans. Knowl. Data Eng.* **2017**, *30*, 488–502. [CrossRef]
23. Tsuchida, H.; Kawamoto, Y.; Kato, N.; Kaneko, K.; Tani, S.; Hangai, M.; Aruga, H. Improvement of Battery Lifetime Based on Communication Resource Control in Low-Earth-Orbit Satellite Constellations. *IEEE Trans. Emerg. Top. Comput.* **2022**, *10*, 1388–1398. [CrossRef]
24. Bell, N.; Garland, M. *Efficient Sparse Matrix-Vector Multiplication on CUDA*; Nvidia Technical Report NVR-2008-004; Nvidia Corporation: Santa Clara, CA, USA, 2008; Volume 2, p. 5.
25. Jang, B.; Schaa, D.; Mistry, P.; Kaeli, D. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Trans. Parallel Distrib. Syst.* **2011**, *22*, 105–118. [CrossRef]
26. Harish, P.; Narayanan, P.J. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *International Conference on High-Performance Computing*; Springer: Berlin/Heidelberg, Germany, 2007. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.