*photonics*

MDPI

*Article*

# Mercury: Accelerating 3D Parallel Training with an AWGR-WSS-Based All-Optical Reconfigurable Network

**Shi Feng, Jiawei Zhang *, Huitao Zhou, Xingde Li and Yuefeng Ji**

State Key Lab of Information Photonics and Optical Communications, Beijing University of Posts and Telecommunications (BUPT), Beijing 100876, China; fengshi2016@bupt.edu.cn (S.F.); zhouhuitao@bupt.edu.cn (H.Z.); 2025010093@bupt.cn (X.L.); jyf@bupt.edu.cn (Y.J.)
* Correspondence: zjw@bupt.edu.cn

**Abstract**

The network traffic of 3D parallel training in large-scale deep learning, featuring burstiness, hot-spots, and periodic large-bandwidth patterns, severely challenges network efficiency, necessitating a high-performance and flexible optical network solution. To address this, this paper proposes Mercury, a hybrid optical network based on physical optical components: its optical timeslot switching (OTS) subnet uses an arrayed waveguide grating router (AWGR) and tunable lasers for dynamic traffic, while the optical circuit switching (OCS) subnet relies on wavelength selective switches (WSSs) for low-latency high-bandwidth transmission, which is coordinated by selective valiant load balancing (S-VLB) and most efficient path configuration (MEPC) mechanisms. Validated via simulations and FPGA-based testbed experiments, Mercury outperforms the Sirius network by reducing epoch training time (e.g., 179s with five jobs) and relieving OTS congestion through offloading large flows to OCS. This work demonstrates that Mercury provides a flexible, high-performance physical optical solution for 3D parallel training of large-scale deep learning models.

**Keywords:** distributed deep learning; optical data center; hybrid optical switching; training acceleration; load balancing

## 1. Introduction

The exponential growth of deep neural networks (DNNs), spearheaded by models like GPT-4 and BERT, has catapulted artificial intelligence into the era of ultra-large-scale architecture. GPT-4, for instance, boasts an astonishing 1.8 trillion parameters [1], pushing the boundaries of training capability. As the parameter size, number of layers, and training dataset scale of large models continue to grow, both distributed deep learning (DDL) traffic and training resource demands have increased significantly, posing greater challenges to artificial intelligence data center networks (AI-DCNs). To support the training of such large-scale models, multiple distributed training strategies have been proposed. Data parallelism (DP) [2] addresses single-device memory limitations by replicating model parameters across multiple devices, enabling the parallel processing of data batches and gradient synchronization. Tensor parallelism (TP) [3] partitions model parameters and computations, such as matrix dimensions in transformers, to balance computational loads for high-density operations. Pipeline parallelism (PP) [4], to ease the computation burden of numerous neural layers, leverages micro-batch scheduling to execute forward and backward propagations in a sequential, layered manner, effectively overlapping computation and communication by dividing layers across devices. These strategies converge within

Check for updates

Received: 3 February 2026
Revised: 10 March 2026
Accepted: 10 March 2026
Published: 16 March 2026

**Copyright:** © 2026 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.

advanced hybrid frameworks, such as DeepSpeed, where data, parameters, and network layers are systematically distributed, which is known as 3D parallelism [5]. This integrated approach has become essential for training neural networks that would otherwise be computationally intractable.

However, 3D parallelism introduces distinct communication challenges that put a severe strain on AI data center networks in multiple dimensions. DP and TP rely on symmetric all-reduce operations (e.g., ring all-reduce) for gradient synchronization and tensor activation exchanges, respectively. This establishes fixed communication dependencies between specific GPU pairs at each pipeline stage, forming "hotspot links" that are prone to congestion. The communication flow introduces sequential dependencies: during forward propagation, pipeline stages cannot pass activation values to the next stage until TP communication is complete, causing latency accumulation. During back propagation, DP requires each layer's TP to synchronize gradients before proceeding, and layers can only propagate errors to the next stage after TP synchronization has occurred, where even minor delays cascade through the pipeline and cause cumulative delays. For PP traffic, asymmetric patterns (e.g., unidirectional activation that transfers between consecutive layers on critical GPU pairs) lead to inefficient utilization of bidirectional network links and can potentially trigger head-of-line blocking in asymmetric paths. Additionally, PP exacerbates communication burstiness due to the dynamic overlap of micro-batch forward/backward propagations across devices. Specifically, simultaneous activation transfers between the pipeline stages and the abrupt release of delayed micro-batch backlogs, both of which are inherent in PP's layered execution, generate unpredictable traffic surges. Collectively, these factors degrade the network resource scheduling efficiency, impair latency performance, reduce adaptability, amplify scheduling overhead, and necessitate the near-real-time reconfiguration capabilities between GPUs. Though communication matrices can be precomputed via GPU placement and neural network structure [6], dynamic computation–communication interdependencies (with causal timing links across iterations) require a rapid network reconfiguration. For instance, ring all-reduce in data parallelism generates heavy GPU-pair traffic, and customized paths cut latency by avoiding congestion stalls that are in training. Reconfigurable data center networks (RDCNs) are based on optical switching and thus emerge to address such traffic-driven dynamic needs. RDCNs schemes are divided into two types: traffic-unaware static schemes, which rely on predefined rules/offline models without using real-time adaptation, and traffic-aware dynamic schemes, which adjust via real-time/predicted traffic (facing control overhead and scalability challenges). Traffic-unaware static schemes include RotorNet [7], Shoal [8], Microsoft's Sirius [9], Cornell's Shale [10], REACToR [11], and Ben Yoo's H-LION [12], all of which adopt offline routing tables without real-time traffic awareness, whose the key limitation of mismatched fixed topologies and hotspots leading to link overload, resource waste, and failure in DDL with periodic traffic, while traffic-aware dynamic schemes that optimize DCN performance include c-Through [13], Helios [14], TopoOpt [15], Lightwave Fabrics [16], Google's Jupiter [17], TPUv4 [18], PULSE [19], HPN [20], NegotiaToR [21], and other works [22–24], though they face fundamental limitations in modern AI workloads such as 3D parallel training: state-of-the-art Jupiter architectures and Google's TPUv4 clusters still depend heavily on high-speed electrical switching fabrics (TPUv4 additionally using a fixed cube-based electro-optical topology with static inter-superPod optical links) and only improve throughput by scaling the electrical port bandwidth rather than by architectural innovation, alongside common issues including centralized control bottlenecks, inadequate traffic estimation, and hardware–software mismatches; thus a hybrid architecture is essential, as it combines periodic updates for structured patterns and lightweight

real-time corrections, leveraging 3D training periodicity to preconfigure optical paths, balancing proactive allocation efficiency and real-time agility for large-scale AI training.

However, DNN training with 3D parallelism imposes stricter temporal dependencies and dynamic communication patterns that exceed existing RDCN capabilities. It introduces two key challenges: deterministic yet adaptive communication sequences (balancing periodicity and real-time burst reconfiguration) and port resource contention that is worsened by static allocations. The current traffic-aware RDCNs struggle due to centralized control bottlenecks, insufficient micro-batch scheduling agility, and limited scalability validation. This work bridges this gap with a scalable, low-latency solution (validated via hardware and large-scale simulations), extending our research prior to earlier work [25] (Optical Fiber Conference 2025). Its main new contributions are summarized below:

- This work conducts a detailed analysis of DDL training traffic models at the micro-batch level under 3D parallel strategies. Based on these models, we introduce a hybrid network architecture integrating an OCS subnetwork and an OTS subnetwork. This optical network is further augmented with a two-tier scheduling algorithm, which couples periodic topology optimization for predictable 3D traffic patterns with distributed real-time adjustments.
- We develop a high-fidelity OMNeT++ based simulation framework to model 3D parallel training across 64 GPUs under Mercury, enabling a large-scale validation of the hybrid architecture's scalability and demonstrating a remarkable speedup effect compared to the representative RDCN work. The proposed scheme is also validated using a three-node FPGA-based hardware testbed. Experimental results also demonstrate a significant speedup in DDL job training compared to the state-of-the-art solutions.

## 2. Network Architecture and Traffic Modeling

### *2.1. Network Architecture*

This paper first details our proposed Mercury network architecture (Figure 1), which extends from our previous work [25]. This architecture is a novel integration of OTS networks and OCS networks, tailored to the unique requirements of distributed machine learning workloads. Hosts house multiple GPUs as computational engines, with integrated NICs handling the communication, data processing, and buffering. Each NIC serves as the gateway for all network-bound traffic. Intra-host communication (GPU-to-GPU and GPU-to-NIC) relies on the high-speed PCIe bus protocol [26] for rapid data transfer. The network-wide time synchronization aligns GPU-hosts, dividing time into equal-duration timeslots as the minimum scheduling unit, converting dynamic traffic management into a periodic, slot-based framework with standardized temporal windows for data transmission and resource allocation; all incoming frames are split into fixed-size cells that are assigned to corresponding timeslots, ensuring deterministic scheduling and low overhead. The OTS network employs an AWGR as the high-speed interconnection fabric, leveraging its wavelength-routing property for passive optical transmission, which reduces energy consumption and simplifies scheduling (fixed wavelength-to-port mapping eliminates complex real-time routing). Fast tunable lasers, composed of SOA arrays and lasers, enable nanosecond-scale wavelength switching by controlling SOA on/off states (pre-tuning to the target wavelength, then deactivating the original SOA and activating the target [9]). The OTS subnetwork also supports 400 Gb/s/800 Gb/s high-speed NICs through advanced modulation formats, WDM, and optical frequency combs, overcoming the channel spacing limitation of AWGRs. Hosts are grouped into pods, each with AWGRs equal to the total number of pods; each AWGR connects to m hosts from one pod and links them to local $m$ hosts in its own pod, with each host having $n$ ports (equal to total pods) for direct communication with all hosts in the target pods, delivering high parallelism, scalability

and fault tolerance. Complementing the OTS network, the OCS network (implemented with WSSs) provides an alternative connectivity path for NICs via additional ports. WSSs outperform MEMS-based switches [27] by supporting dual-dimensional (wavelength and spatial) connectivity, enabling efficient wavelength-division multiplexing to maximize fiber utilization and adopting a Spanke structure [28] for spatial non-blocking operation; while WSS introduces inherent insertion loss, the Mercury architecture is hardware-agnostic and fully compatible with low-insertion-loss OCS alternatives such as MEMS, thus ensuring its feasibility and scalability in large-scale intelligent computing and DDL scenarios. Mercury achieves excellent scalability: its hybrid control architecture avoids centralized bottlenecks with the central module only handling low-frequency OCS reconfiguration, and the distributed protocol features low complexity with manageable per-node loads. The flat network design enables an incremental expansion via AWGR deployment without hierarchical forwarding bottlenecks and adapts well to large-scale deployments with tens of thousands of GPUs.

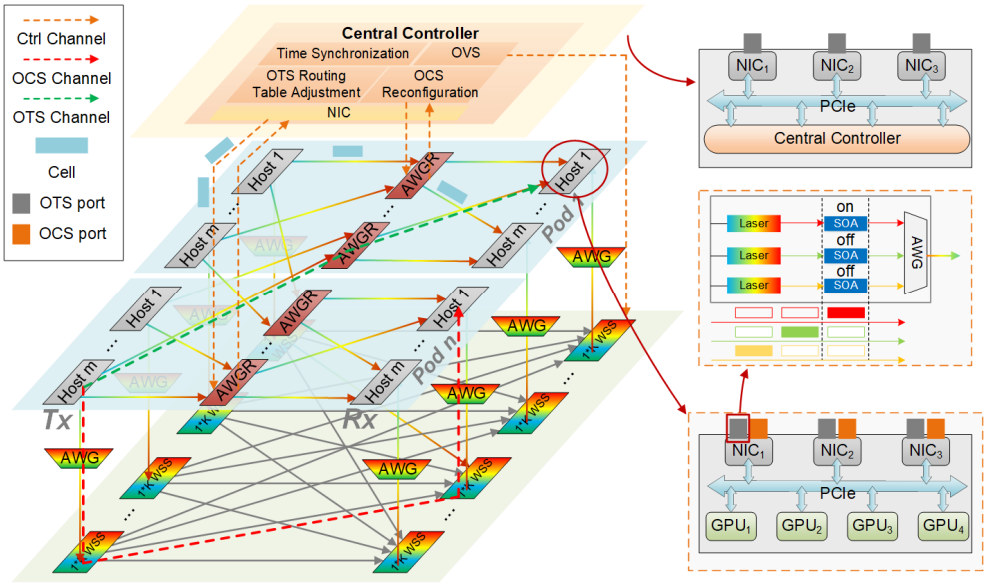



**Figure 1.** Mercury: hybrid OTS/OCS-based reconfigurable datacenter networks.

### *2.2. Traffic Modeling*

DDL traffic has two key characteristics: periodicity in communication matrix topology (e.g., fixed inter-GPU connections) and predictability in traffic volume (e.g., consistent gradient sizes), determined by model architecture, parallelization strategies, and hardware mapping (GPU allocation policies). Analyzing these factors underpins the dynamic optical network reconfiguration [3,29]. This paper assumes a model architecture based on transformer blocks [29] (core of modern LLMs like GPT/BERT) and adopts 3D parallelism (data/tensor/pipeline). To analyze inter-GPU communication patterns, we assume $d$ as the data parallelism dimension, $t$ as the tensor parallelism dimension, and $p$ as the pipeline parallelism dimension. As detailed in refs. [3,29], we decompose traffic into micro-batch-level flows during iterative training. For a text dataset with batch size $b$, sequence length $s$, and hidden dimension $h$, each micro-batch first generates a $(b, s, h)$ tensor via the embedding layer and positional encoding. The tensor enters the $L$ transformer blocks, which are partitioned into $p$ pipeline stages (GPUs), with each stage processing $L/p$ blocks. Since each GPU contains multiple transformer blocks, each transformer block completes all internal computations before initiating synchronization and communication. This approach ensures that data dependencies within a block are fully resolved before exchanging intermediate results across GPUs, aligning with the pipeline parallelism schedule where each

stage processes the $L/p$ blocks sequentially. We define $m$ as the number of micro-batches per iteration, where one iteration completes both forward and backward propagations for all micro-batches. We illustrate it with pipeline parallelism dimension $p = 3$, tensor parallelism dimension $t = 2$, and data parallelism dimension $d = 2$.

In forward propagation (Figure 2a), GPUs in pipeline stage 1 (1, 2, 7, and 8) first generate embedding vectors via look-up tables (no computation), followed by positional encoding, with the element-wise addition of embeddings and positional encodings referenced. Based on input $X$, the multi-head attention layer of the first transformer block for the current micro-batch performs activation computation and computes projection matrices $Q/K/V$, and the attention scores $S_i$ are calculated from $Q_i/K_i/V_i$. Matrix $Z_i$ is derived by weighting $A_i$ and $V_i$, then all-reduce merges $Z_i$ from tensor parallel members, and the output projection is done via the $W_O$ matrix. Tensor parallel members execute all-gather operations (each of the $L/p$ transformer blocks on a single device run independently, with the data volume per operation $(t-1)/t \cdot b \cdot s \cdot h \cdot S_e$ where $S_e$ is bytes per element, e.g., eight for FP64). The attention layer output enters the MLP layer. After a GPU completes MLP and the attention layer computations for all transformer blocks (step ① in Figure 2a), pipeline stage 1 synchronizes activation tensors via ring all-reduce (two communication rounds: scatter-reduce/all-gather among members like GPU pairs 1–2 or 7–8, total steps $2(t-1)$). The synchronized tensors are sent to the next pipeline stage (Step ②) for repeated computation/communication, and forward propagation finishes after the final synchronization (Step ⑤), followed by immediate backward propagation; for parallel speed optimization, each subsequent micro-batch starts forward propagation once the previous one completes computation in the current pipeline stage to overlap computation and communication across stages.

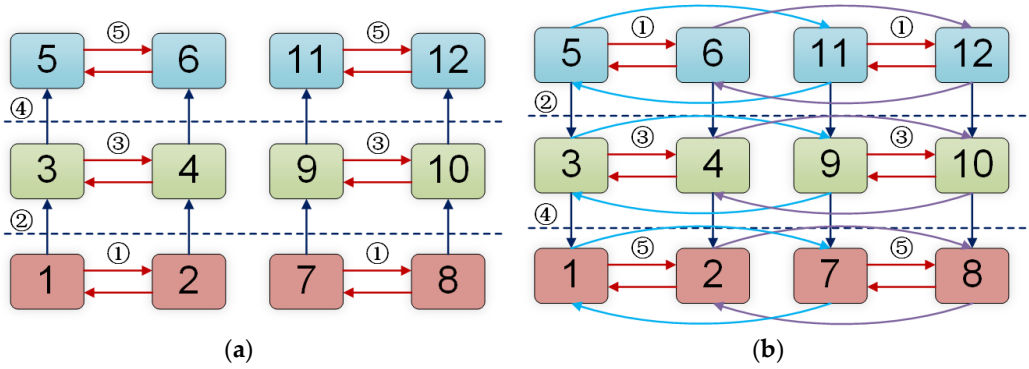



**Figure 2.** Parallelism mapping of GPUs with $(p, t, d) = (3, 2, 2)$: (**a**) forward propagation and (**b**) backward propagation.

In backward propagation (Figure 2b), for each micro-batch at the last pipeline stage, loss $L$ is first computed to derive $\frac{\partial L}{\partial O}$. At each transformer layer, MLP layer parameter derivatives are calculated for parameter updates, then all-reduce aggregates MLP output gradients among tensor parallel members, and the attention layer uses MLP-derived gradients to compute $Q/K/V$ matrix gradients. After the last stage of micro-batch computation, synchronous synchronization involves all-reduce operations, with $2 \cdot (d-1)$ communication steps and $S_e \cdot N/p \cdot t$ bytes per step ($N$ = total model parameters). Once a pipeline stage finishes a gradient computation of a micro-batch, it passes gradients to the preceding stage ②. When the first pipeline stage completes all gradient synchronization, embedding table synchronization between each pipeline stage is required (marking an end to the training iteration). Thus, training traffic shows iteration-level periodicity and intra-iteration temporal correlation (interleaved computation/communication), posing network design challenges to handle periodic synchronization bursts and time-dependent traffic (minimiz-

ing overhead). Table 1 summarizes the communication traffic volume (frequency (times) and per-communication data volume (bytes)) of the three parallelization methods per iteration per device, with DP assuming gradient accumulation (local micro-batch gradient computation before global ring all-reduce synchronization). For the detailed numerical values presented in this table, please refer to Appendix A.

**Table 1.** Communication Demands in an Iteration.

| Type | Times | Volume/Bytes |
|---|---|---|
| TP | $L/p \times m \times 7 \times (t-1)$ | $(t-1)/t \times b \times s \times h \times S_e$ |
| DP | 1 | $2 \times (p-1)/p \times N/t \times S_e$ |
| PP | $2 \times m \times (p-1)$ | $b \times s \times h \times S_e$ |

## 3. Control Mechanism

The Mercury network adopts a collaborative, centrally distributed control mechanism to achieve an efficient operation. Notably, this mechanism is characterized by two key design features: a hybrid of traffic-aware and on-demand control (centralized) and a traffic-unaware execution (distributed), coupled with synergy between coarse-grained resource configuration and fine-grained data transmission. Its core lies in coordinated path configuration and traffic allocation strategies through static scheduling and dynamic adjustment, where OCS reconfiguration is used for phase switching, and task on–off matches the low-dynamic workload changes in fixed-parallelism AI training, while nanosecond-scale OTS slot scheduling adapts to fine-grained real-time traffic dynamics—achieving hierarchical matching between reconfiguration time constants and workload dynamics, thus ensuring efficient network resource utilization and a low-latency transmission. This section elaborates on the core aspects of this mechanism, including the basic scheduling framework, the distributed selective-VLB mechanism, and the centralized OCS path configuration algorithm.

### *3.1. Mercury Scheduling Framework*

The network operation architecture is illustrated in Figure 3. Its data plane comprises OTS and OCS subnetworks, while the control plane relies on a centralized arbiter operating over the OTS network. As shown in Figure 3a, the arbiter (master) achieves time synchronization with each NIC (slave) via sequential timestamp exchanges; once global synchronization is established, time is partitioned into equal discrete slots, enabling timeslot-based transmission. Prior to network scheduling, job training tasks were allocated by mapping the 3D parallel groups of each job to GPUs across different hosts. To minimize cross-host communication overhead in 3D parallel training, we propose a strict priority job deployment strategy: TP groups are deployed first, followed by PP groups, and finally DP groups. This sequential approach ensures that higher priority components (with more frequent communication) are prioritized for intra-host placement, with cross-host deployment allowed only when resource constraints make intra-host placement impossible. Intra-node communication between GPUs relies on dedicated high-speed internal interconnects such as NVLink and PCIe, rather than the Mercury optical network. Upon receiving packets from external nodes, the NIC forwards traffic that is destined for local GPUs directly through the on-board high-speed bus with negligible latency.

Within the OTS network, all host NICs and the arbiter follow a static round-robin routing table (Figure 3b) to tune wavelengths, enabling conflict-free port forwarding via the AWGR [9]. This ensures that all of the hosts and the arbiter communicate with each other at least once per cycle. To address the non-uniform traffic matrix that is introduced by 3D parallel training, load balancing is implemented using selective valiant load balancing (S-VLB), which is a modified version of the standard valiant load balancing in [9], with its logic adapted based on

OCS path configurations derived from the MEPC algorithm. The OTS network is designated to handle two types of traffic: bursty small flows generated after GPU computation (e.g., cross-rack TP/PP traffic) and traffic waiting for OCS configurations to complete.

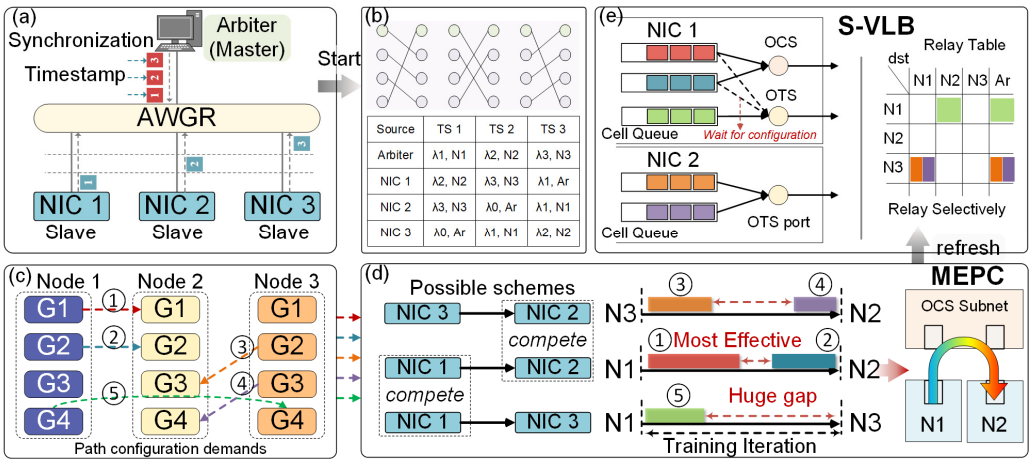



**Figure 3.** Mercury scheduling framework: (**a**) time synchronization; (**b**) forwarding table of OTS; (**c**) OCS configuration demands; (**d**) MEPC process; and (**e**) S-VLB.

By virtue of the static wavelength routing table, the arbiter receives one cell from each NIC per cycle. NICs encapsulate OCS link requests and S-VLB relay information in cell headers; even during arbiter-bound slots with no data to transmit, an empty cell carrying control messages in its header is still generated to ensure a timely information exchange. Upon receiving job arrival notifications and associated path requests, the arbiter executes the MEPC algorithm to determine OCS reconfiguration schemes. This algorithm not only maximizes OCS bandwidth utilization but also accelerates transmission and alleviates the OTS network load. The reconfiguration results are distributed to all NICs via the round-robin table within one cycle, prompting each NIC to update its local S-VLB logic. During OCS reconfiguration, the OTS network temporarily carries traffic that cannot be routed through OCS. For instance, in the scenario depicted in Figure 3c, transmission demands ①/② (node 1 to node 2), ③/④ (node 3 to node 2), and ⑤ (node 1 to node 3) are encapsulated into specific fields of cell headers before being sent to the arbiter. The arbiter then determines OCS link configurations using the MEPC algorithm and distributes the results to all NICs. During the OCS reconfiguration period, the OTS subnetwork absorbs the reconfiguration latency by temporarily carrying the traffic that will be served by the upcoming OCS links; the NICs subsequently adjust their local load balancing strategies via S-VLB in a distributed manner.

### *3.2. Most Efficient Path Configuration (MEPC) Algorithm*

During the network operation, each host NIC dynamically monitors the volume of cells that are destined for specific targets in its local DRAM buffer. When the number of cells accumulated for a particular destination reaches the predefined threshold $Q$, the NIC triggers a path configuration request encapsulating details such as the source–destination pair and traffic volume into the header of a cell. This request is transmitted to the arbiter during the specific timeslots for arbiter communication in the round-robin schedule. If no data cells are available for transmission to the arbiter during these slots, the NIC generates a "fake cell" (an empty data cell with a header carrying request information, including service type) to ensure the timely delivery of configuration requests and enable an accurate traffic demand prediction; this mechanism guarantees path demands are communicated to the arbiter independent of actual data traffic. Specifically, the service type carried in the cell header is inherently tied to a fixed OCS link demand: since each service originates from

a designated GPU, the corresponding inter-GPU connection requirement is determined immediately upon service initiation, establishing a one-to-one mapping between the service type (along with the source–destination pair) and the required OCS link configuration. This deterministic mapping enables the MEPC algorithm to precompute and configure OCS links for DP bursty traffic in advance, maintaining control continuity. For fixed 3D parallel DDL tasks, OCS-traffic mismatches are extremely rare due to highly predictable traffic demands; even if such mismatches arise, the OTS subnetwork can automatically absorb the unprovisioned traffic via fine-grained scheduling, with only slight performance impacts that do not disrupt the system's overall stability.

The selection of threshold $Q$ is tightly linked to the characteristics of the traffic. Most of the traffic destined for specific targets corresponds to DP flows, known for their large, bursty volumes and periodic communication patterns. $Q$ is set to match the volume of a single DP communication burst. This calibration ensures that $Q$ effectively filters transient, small-scale traffic (which can be efficiently handled by the OTS network via S-VLB). Triggering OCS path requests only for substantial DP flows would benefit from the high bandwidth and low latency of direct optical connections. By aligning $Q$ with the intrinsic volume of DP communication, the mechanism avoids unnecessary OCS reconfigurations (reducing control overhead) and ensures that OCS resources are reserved for the traffic that truly demands them. Upon receiving these path requests (whether via data cells or fake cells), the arbiter consolidates all pending demands and initiates the MEPC algorithm to determine the optimal OCS configurations. The algorithm first enumerates potential path schemes, considering the resource constraints (each source/destination port can only support one path at a time) and the specific demands that are triggered by the $Q$ threshold. Since the requested paths are primarily for DP flows with known volume characteristics, MEPC prioritizes schemes that minimize idle gaps for these high-volume flows. This ensures that the selected OCS configurations not only maximize bandwidth utilization (by reducing idle slots) but also directly address the most resource-intensive traffic, thereby alleviating the burden on the OTS network. The final configuration results are then distributed to all NICs via the round-robin schedule, as described earlier, to update their local path states.

For the example shown in Figure 3c, each NIC dynamically detects local path configuration requirements by threshold $Q$ and encapsulates these requirements into cell headers to report to the arbiter. The requirements are as follows: ①/② from node 1 to node 2, ③/④ from node 3 to node 2, and ⑤ from node 1 to node 3. These requirements are identified by the arbiter through specific fields in the cell headers. The core logic of the MEPC algorithm is executed in Figure 3d: requirements ③/④ can share the path from node 3 to node 2, and requirements ①/② can share the path from node 1 to node 2 by interleaving their computation and communication phases. Since each source or destination port can only be configured with one path at a time, there is resource competition among these schemes. The MEPC algorithm evaluates the proportion of idle time slots (idle gap) of each scheme within each scheduling cycle and prioritizes the configuration scheme with the smallest idle gap to maximize the utilization of path resources. Finally, since the OCS reconfiguration delay is relatively constant, we encapsulate the effective timestamp of the OCS links into control frames together with the configuration results output by the MEPC algorithm and deliver them to the corresponding node NICs to guide the switching of their port states.

### *3.3. Selective Valient Load Balancing (S-VLB)*

S-VLB is designed to address bursty, hotspot-prone traffic in 3D parallel training that is marked by frequent TP/PP communication with temporal fluctuation. It combines slot-based transmission with dynamic path selection, enabling efficient traffic distribution across the OTS–OCS integrated network. The OTS subnetwork, built on an AWGR-based

optical timeslot architecture with round-robin wavelength tuning, ensures conflict-free communication but requires targeted load balancing to handle non-uniform 3D parallel traffic patterns. Traditional VLB [9,30] disperses traffic by randomly redirecting packets through intermediate nodes, but this introduces unnecessary overhead in networks with predictable flows (e.g., periodic large-bandwidth traffic in training iterations). Unlike the traffic-aware relay within AWGRs in work [21], the high-bandwidth traffic in Mercury is groomed by introducing an OCS subnetwork. S-VLB enhances VLB by limiting the load balancing to traffic that cannot use established OCS direct connections, thus reducing redundancy. S-VLB also ensures the continuity of the traffic that waits for the OCS configuration.

In S-VLB, each NIC inspects locally generated traffic (stored in DRAM) and encapsulates forwarding requests into cell headers, sent during its allocated round-robin timeslots to avoid collision. Unlike traditional VLB's random intermediate node selection, S-VLB prioritizes OCS-direct destinations: if a direct OCS path exists, traffic is sent directly without relaying. The arbiter, functioning as both a control plane node and a specialized relay, is included in the round-robin schedule to handle globally coordinated traffic. Upon receiving requests, nodes (including the arbiter) grant forwarding permission only if the sum of the queued packets for the target destination and issued grants is below threshold $Q$, preventing queue overflow. Granted cells are moved from DRAM to dedicated FIFO queues for the intermediate node (or directly to OCS-connected peers) and transmitted in corresponding timeslots, with OCS-direct traffic prioritized to bypass intermediate hops.

A key strength of S-VLB is its tight integration with the OCS subnetwork via the MEPC algorithm. It dynamically adapts to OCS topology changes: traffic destined to the nodes with established OCS links skips load balancing, reducing latency and conserving OTS bandwidth; during OCS reconfiguration, S-VLB temporarily handles all traffic to avoid disruptions, and the arbiter, alongside its control role, maintains relays for globally optimized paths. S-VLB uses a hierarchical queue structure (Figure 4): incoming cells are stored in DRAM that is partitioned by final destination for efficient look-up, with dedicated FIFO queues per destination. OCS-direct cells route directly to output FIFOs, while relayed cells use intermediate node FIFOs. High priority is given to OCS-connected traffic, with temporary redirection through S-VLB during reconfiguration to maintain connectivity. This design has multiple advantages: selective load balancing reduces unnecessary hops for OCS-served flows; OTS bandwidth is conserved for bursty traffic; direct OCS links minimize latency for high bandwidth traffic; and the distributed congestion control mechanism, paired with centralized OCS configuration, scales efficiently to large systems. S-VLB thus balances flexibility for bursty traffic and efficiency for periodic flows, aligning with 3D parallel training demands.

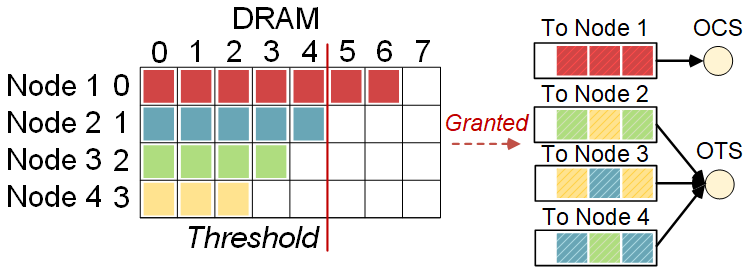



**Figure 4.** The implementation of S-VLB.

## 4. Performance Evaluation

In this section, we present a comprehensive evaluation of the Mercury network, encompassing both simulation analyses based on OMNet++ and hardware-validated results from an FPGA-based testbed. We adopt the representative Sirius network [9] as

the primary benchmark for its well-founded representativeness and strong comparability: as a state-of-the-art fully optical reconfigurable data center network, it employs pure distributed scheduling based on VLB with traffic-unaware uniform forwarding, which makes it a typical baseline for conventional optical load-balancing networks. Given that Mercury's core scheduling innovation lies in S-VLB and a hybrid centralized-distributed scheduling mechanism, we systematically compare the performance of Mercury against Sirius in terms of latency characteristics under varying traffic loads to directly and fairly evaluate the performance gains brought by our key design optimizations. The evaluation methodology, experimental configurations, and detailed results are elaborated as follows.

### *4.1. Simulation Setup*

Figure 5a illustrates an 8-node simulated network constructed on the OMNet++ platform, where the 8 nodes are partitioned into 2 pods, with each pod comprising 4 hosts and 2 AWGRs, each AWGR equipped with 4 input ports and 4 output ports, respectively. Each host accommodates 8 GPUs, resulting in a total of 64 GPUs across the entire network. And a network-wide arbiter is deployed, which connects to the AWGR (the one that links all GPUs within the respective pod) in each distinct pod. The physical distance between each host and each AWGR within the same pod is set to 5 m, while the distance between each host and the AWGRs belonging to different pods is configured as 25 m, with the port bandwidth for both hosts and the arbiter set to 10 Gbps. The traffic input is configured as transformer-based jobs, which employ 3D parallelism; each GPU is assumed to serve a single job at its maximum computational load, and simulation experiments are conducted with 1, 2, 3, 4, and 5 jobs respectively, with the parallel dimension configurations (p, t, and d) for each job under different job counts designed to fully utilize all 64 GPUs as detailed in Table 2. We selected the classic distributed control-based network Sirius with the same scale as the comparison object, which adopts the traditional VLB mechanism for traffic forwarding.

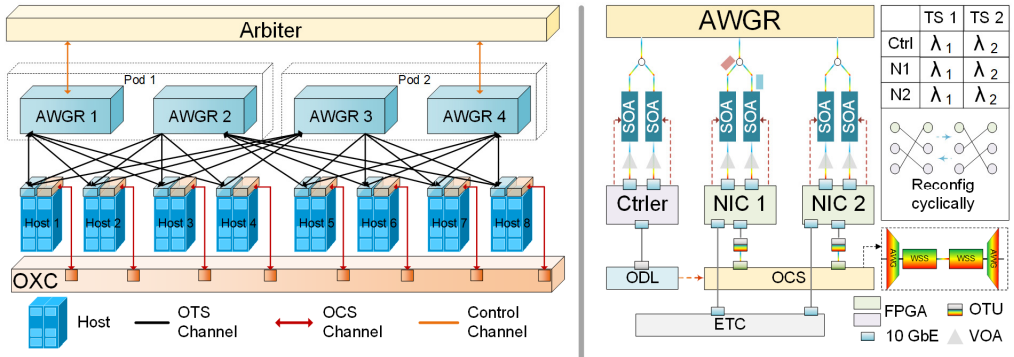



**Figure 5.** Evaluation scenarios: (**a**) simulation topology and (**b**) testbed topology.

**Table 2.** GPUs Setup under Varied Job Counts.

| Job Num | (p, t, d) Per Job | GPUs Per Job |
|---|---|---|
| 1 | (4, 4, 4) | 64 |
| 2 | (4, 4, 2) | 32 |
| 3 | (2, 5, 2) | 20 |
| 4 | (2, 4, 2) | 16 |
| 5 | (2, 3, 2) | 12 |

### *4.2. Experimental Setup*

This experimental testbed, as presented in our previous work [25] and illustrated in Figure 6, is used to construct the Mercury network. The testbed comprises 2 NICs, 1 AWGR, 1 arbiter, and 1 OXC (incorporating 2 WSS). Each NIC is implemented using a VCU118 FPGA board with two ethernet interfaces, each of which can be split into four independent 10 Gbps ethernet interfaces; we configure each NIC with two 10 GbE ports connected to an ethernet test center (ETC), where each 10 GbE port receives traffic from a GPU emulated by the ETC, while another 10 GbE port of the NIC is connected to the OXC. SOA-based fast tunable lasers mounted on an expander board are linked to the NICs via FMC ports, enabling the NICs to switch cells from the ingress to the egress of the AWGR through these tunable lasers. Additionally, the arbiter communicates with an OpenDayLight (ODL) SDN controller via a UART interface for optical path configuration, and OCS management is further realized through an OpenFlow-extended protocol together with the flat network architecture, which reduces hardware complexity and ensures engineering feasibility. The AWGR/SOA-based OTS architecture follows the well-validated mature scheme in ref. [9], with its operating principle verified in prior optical slot-switching testbeds; the SOA module is integrated with a custom closed-loop temperature controller that stabilizes the chip temperature at 25 $\pm$ 0.1 °C, effectively suppressing thermal gain drift caused by temperature variations. Specifically, the ETC emulates 4 GPUs per host (i.e., 4 traffic flows) running two jobs with a DP + MP configuration (2, 2). In this setup, a fast tunable optical switch, implemented by combining the SOA with multi-wavelength optical modules at FPGA ports, is connected to the AWGR to form an optical time-slot subnet, while a WSS interconnected with the AWG forms an OCS subnet, with the two host nodes and the arbiter executing the wavelength routing table as shown in the subgraph of Figure 5b.

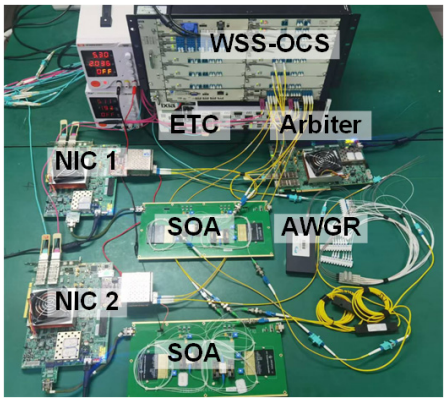



**Figure 6.** Testbed of FPGA-based Mercury Networks.

For the parameter assumptions of model training tasks, the number of transformer layers, denoted as $L$, is set to 12 [31], with the hidden dimension $h$ specified as 256/512/1024/2048, which is designed to represent varied model scales. The sequence length $s$ is set to 512 tokens, covering the input length requirements of mainstream NLP tasks such as text classification and question answering. The storage size of a single parameter $S_e$ is 2 bytes, adopting FP16 precision, which is a commonly used format in deep learning training to accelerate computation and reduce memory footprint, aligning with the hardware characteristics of GPUs. The total number of model parameters is assumed to be $1.1 \times 10^8$, which occupies approximately 27.5 MB under FP16 precision, a value far below the actual memory capacity of GPUs, ensuring that memory does not become a bottleneck. The hardware is assumed to be NVIDIA V100 GPUs, with a computational power of 125 TFLOPs for FP16 (half-precision) operations. Additionally, the micro-batch size $b$ is set to 8. The above parameters are summarized in Table 3.

**Table 3.** Traffic Parameters.

| Parameters | Values |
|---|---|
| Number of transformer layers, $L$ | 12 |
| Hidden layer dimension, $h$ | 256/512/1024/2048 |
| Number of multi-heads, $a$ | 12 |
| Sequence length, $s$ | 512 tokens |
| Size of a micro-batch, $b$ | 8 |
| Storage size of a parameter, $S_e$ | 2 bytes |
| Number of model parameters, $N$ | $1.1 \times 10^8$ (27.8 MB) |
| Computation power of a GPU | 125 TFLOPs |

### *4.3. Results and Analysis*

This section presents simulation results across diverse settings, analyzing how various factors influence model training time. As shown in Figure 7, we compare the DP processing time of our proposed network mechanism with Sirius under different job counts, using a hidden layer size of 256. Since DP traffic has the highest bandwidth demand among 3D parallel flows, and the time to complete DP directly affects the iteration end time, Mercury's ability to handle DP traffic largely determines the overall training time. The simulation results in Figure 7 demonstrate that for all job counts, Mercury significantly outperforms Sirius in DP processing time, completing DP tasks in nearly half the time of Sirius. This advantage stems from the fundamental difference in traffic handling: while Sirius's VLB strategy scatters DP traffic to other nodes for relaying, Mercury can directly offload DP flows through OCS, drastically reducing propagation latency.

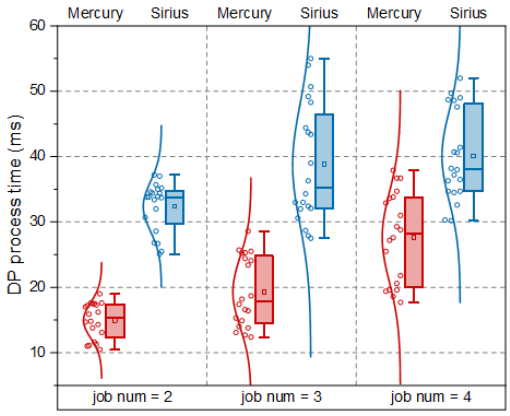



**Figure 7.** DP processing time versus job count.

Figure 8 illustrates the epoch training time versus the number of DDL jobs under different model scales, where model size is adjusted by varying the number of hidden h layers. The simulations are conducted on an 8-node network with 8 GPUs per host. An epoch is defined as a cycle of 1000 iterations traversing the entire training dataset. The results showcase how epoch training time evolves with increasing DDL jobs across diverse model scales. Figure 8a compares epoch training times under different job counts with a hidden layer dimension of 256. It can be observed that the Mercury network achieves lower latency with 1 job, where Mercury fully establishes connections for cross-host demands, resulting in a significant latency reduction of 47.5%. As the number of jobs increases, although the percentage of latency reduction gradually decreases, the absolute reduction value grows. When there are 5 jobs, OCS path competition emerges, but link multiplexing still enables a latency reduction of up to 179 s. Figure 8b,c present results with hidden layer dimensions that are expanded to 512 and 1024, respectively. The results demonstrate that Mercury consistently delivers lower latency regardless of model scale,

which aligns with the trend of increasing model parameters. As the number of jobs increases, network communication patterns grow increasingly complex. Mercury alleviates the burden on the OTS network by offloading large-bandwidth traffic through the OCS subnetwork. This allows the OTS network, with its flexibility in timeslot scheduling, to focus on handling the remaining diverse small flows, such as efficiently balancing loads via the S-VLB mechanism. This synergy leverages the strengths of both networks: OCS carries stable high-bandwidth traffic, while OTS flexibly adapts to dynamically changing flow patterns. As a result, Mercury maintains overall network efficiency even when faced with complex communication matrices caused by growing job counts.

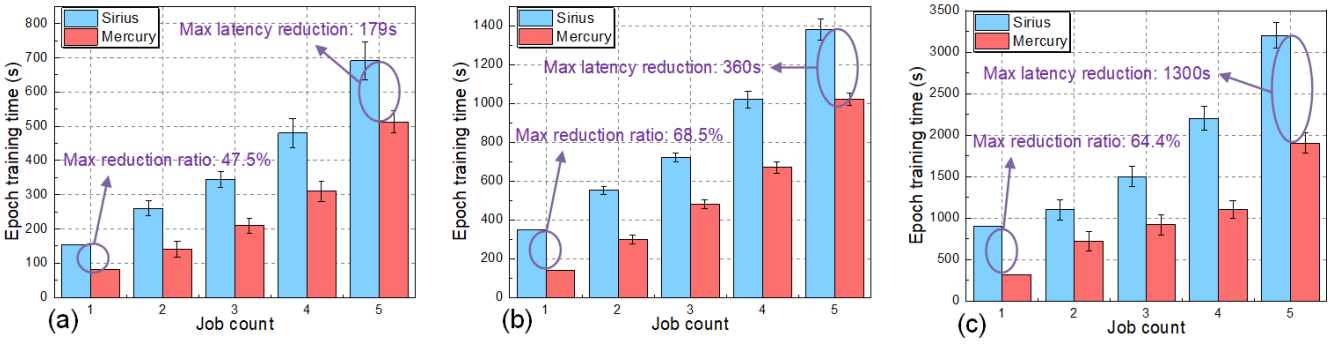



**Figure 8.** Epoch training time versus DDL job counts under different model scales: (**a**) h = 256; (**b**) h = 512; and (**c**) h = 1024.

To demonstrate the compatibility of the proposed architecture with varying model scales, we compare the epoch training time performance of Mercury and Sirius across different model sizes. The corresponding results are presented in Figure 9. Figure 9a shows the trend of epoch training time as the number of hidden layers increases, with two jobs running. The simulation results indicate that our proposed scheme achieves a maximum latency reduction of 779 s, while also revealing that Mercury's performance improves as the model scale grows. Similar trends are observed in Figure 9b,c for three and four jobs, respectively. This is because larger model parameters correspond to larger volumes of gradient exchange traffic. Traditional load balancing in Sirius struggles to handle such loads, while Mercury effectively alleviates pressure and reduces latency by establishing OCS paths to offload hot-spot traffic. These results confirm that Mercury exhibits strong generalization capability, effectively adapting to the ever-increasing model scales.

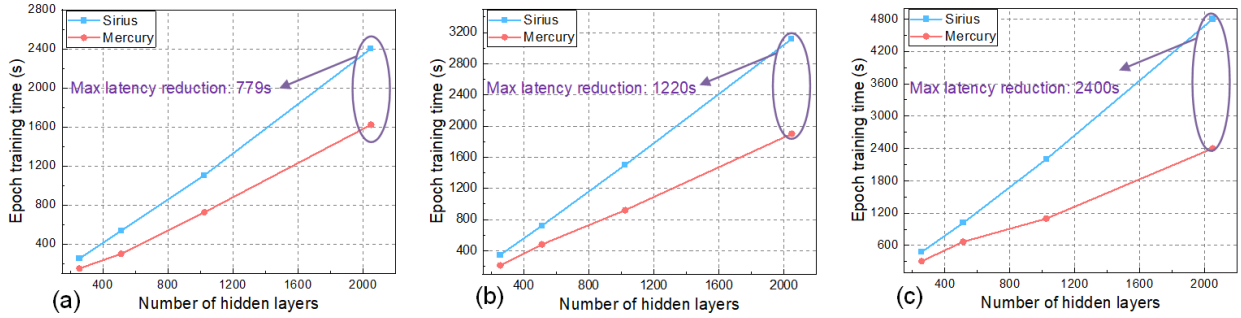



**Figure 9.** Epoch training time versus model scales under different DDL job counts: (**a**) job num. = 2; (**b**) job num. = 3; and (**c**) job num. = 4.

We present the experimental results in the FPGA-based testbed under different settings and discuss the impact on epoch training time. We tested the proposed scheme on a three-node experimental platform (including an arbiter) equipped with an FPGA, WSS, AWGR, and SOA-based fast switching optical switches. The experimental results are presented

in Figure 10. Figure 10a shows the SOA on/off control signals observed in the FPGA's ILA. These signals dynamically switch the SOA through bit transitions, achieving dynamic wavelength switching and laying the foundation for the OTS network. To address the CDR (clock and data recovery) challenge and resolve the latency mismatch between nanosecond-scale SOA switching and the second-scale link setup of commercial optical transceivers, we adopt a customized CDR scheme based on mature mechanisms in [9] and use a separate FPGA board for centralized frequency distribution to all nodes; combined with phase caching and pre-compensation and timestamp synchronization via initialization packets, this design ensures data consistency between the transceiving ends of optical timeslots and eliminates the second-scale synchronization overhead of commercial optical modules. The effectiveness of this implementation can be verified through the oscilloscope in Figure 10a: the yellow and cyan traces represent the control levels of different SOAs, and their high-low level transitions indicate that SOA on/off switching has enabled optical time-slot transmission. Due to the small scale of the experimental platform, we simulated distributed training services at this scale. Using ETC, we emulated transmission between four GPUs across two nodes, achieving conflict-free transmission of cells carrying control information to the arbiter and cells destined for other nodes. Figure 10b shows the distribution of packet latency, which appears stable and extremely low. We gradually increased the flow bandwidth to simulate traffic generated by model training with different hidden layer dimensions and collected the number of packets transmitted per training iteration. The results show that Mercury outperforms Sirius by a large margin in reducing latency. Figure 11a shows the CDF of a single training iteration completion time for Mercury and Sirius under a hidden layer size of 256, where Mercury achieves a nearly 1.5× speedup in iteration time. Figure 11b illustrates the stability of the system's packet loss rate: it maintains zero packet loss until approaching the performance limit, and packet loss occurs when the line rate nears the port capacity due to the increasing proportion of protocol and control overhead, which prevents the system from reaching the theoretical maximum line rate.

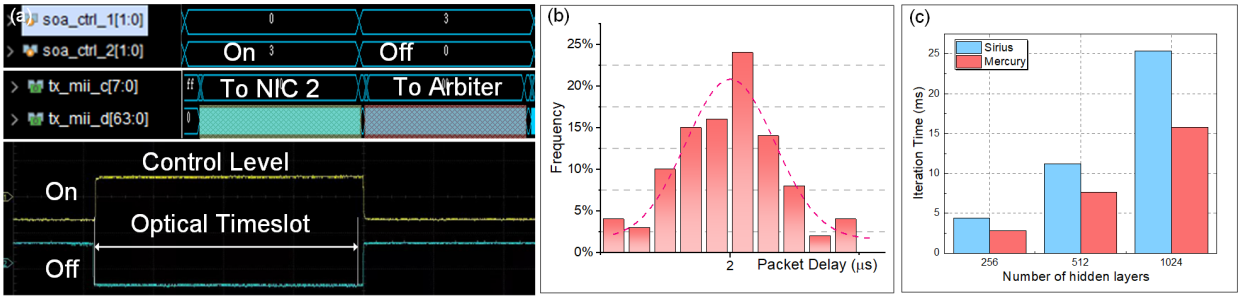



**Figure 10.** Experimental results: (**a**) Signal trace captured in ILA and oscillator; (**b**) packet delay distribution; and (**c**) iteration time under different numbers of hidden layers.

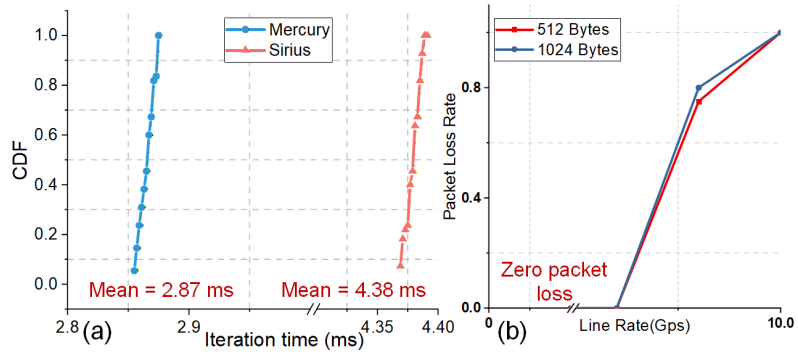



**Figure 11.** Experimental results: (**a**) CDF of iteration time and (**b**) packet loss rate with different cell sizes.

## 5. Conclusions

This paper presents Mercury, a hybrid optical network architecture designed to address the traffic challenges in 3D parallel training, including burstiness, hotspots, and mixed bandwidth demands. By integrating an OCS subnetwork for high-bandwidth, periodic flows and an OTS subnetwork for dynamic, bursty traffic, Mercury achieves efficient resource utilization while minimizing latency. The core innovations of Mercury lie in two key mechanisms: (1) S-VLB, which selectively applies load balancing to traffic that cannot utilize OCS direct links, prioritizing OCS-connected flows to reduce overhead and leverage low-latency optical paths, and (2) the MEPC algorithm, which dynamically optimizes OCS path allocations based on traffic demands, maximizing resource utilization and adapting to changes in network topology. Both simulation results and FPGA-based testbed experiments validate the effectiveness of Mercury. Under varying model scales and job counts, Mercury consistently outperforms the state-of-the-art Sirius network in epoch training time, DP processing latency, and link utilization efficiency. Specifically, it reduces epoch training time by up to 47.5% for small job counts and maintains significant latency advantages (e.g., 179 s reduction for five jobs) even as traffic complexity grows. The hybrid design also alleviates the burden on the OTS network, allowing its flexibility to be fully utilized for handling complex communication matrices induced by increasing job counts.

In summary, Mercury's synergy between OCS and OTS subnets provides a high-performance, scalable network solution for large-scale distributed training. Notably, Mercury is not tailored to a specific 3D parallelism strategy but dynamically reconfigures based on traffic patterns generated by the selected parallelization approach; its hybrid OCS/OTS granularity enables adaptation to diverse training phases and dynamic parallelism scenarios, while its advantages are most pronounced for structured traffic patterns similar to 3D parallelism, with diminished benefits for highly unstructured traffic. Future work will focus on scaling the architecture to larger node counts and integrating advanced machine learning-driven traffic prediction to further optimize OCS reconfiguration timing.

**Author Contributions:** Conceptualization, S.F.; methodology, S.F.; validation, S.F., H.Z. and X.L.; writing—original draft preparation, S.F.; writing—review and editing, S.F., J.Z. and Y.J.; supervision, J.Z. and Y.J. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by "the National Natural Science Foundation of China, grant number 62271078" and "State Key Laboratory of IPOC, grant number IPOC2025ZZ03, IPOC2025ZJ07".

**Data Availability Statement:** The data underlying the results presented in this paper are not publicly available at this time but may be obtained from the authors upon reasonable request.

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

## Appendix A

This appendix elaborates on the detailed derivation processes, calculation formulas and implementation steps for the communication traffic volume (in bytes) and occurrence frequency (in times) generated during the 3D parallel communication process described in Section 2.2 of the main text. All specific numerical results of traffic volume and frequency presented in Table 1 of the main text are derived from the calculation methods and parameter settings detailed in this appendix. The calculations are strictly based on the intrinsic data interaction rules, gradient synchronization mechanisms and parameter update logic of the three parallelization strategies (data parallelism, model parallelism and pipeline parallelism) in 3D parallel training and are combined with the standard configuration of training hardware and the setting of micro-batch size in actual AI model training to ensure the rationality and practicality of the calculated traffic metrics [29–31].

In the forward propagation process, GPUs in pipeline stage 1 first determine embedding vectors via look-up tables, a computation-free operation, followed by positional encoding. The computational complexity, measured in FLOPs (where one addition and one multiplication constitute one FLOP), involves an element-wise addition of embedding vectors and positional encodings, yielding the cost of $b \times s \times h$ FLOPs. Subsequently, with the input $X$, the activation computation for the multi-head attention layer in the first transformer block of the current micro-batch proceeds, involving the $Q/K/V$ projections for all $a$ heads. All heads are distributed due to the tensor that is parallel with column-wise partitioning, where each GPU is equipped with $a/t$ heads. Therefore, the size of projection weight matrices $W^i_{Q/K/V}$ on each device becomes $h \times (h \cdot a/t)$. $Q^i/K^i/V^i$ are obtained by the multiplication between $X$ and $W^i_{Q/K/V}$. The attention score $S^i$ is calculated based on $Q^i$ and $K^i$. Following the computation of attention scores, a SoftMax operation is applied to generate matrix $A^i$. Matrix $Z^i$ is then derived by performing a weighted computation between $A^i$ and matrix $V^i$. Subsequently, an all-reduce operation is executed to merge the $Z^i$ matrices from all tensor parallel members, followed by an output projection via the $W_O$ matrix. The FLOPs for matrix multiplication are calculated by considering that each element in the resulting $m \times p$ matrix from the multiplication of an $m \times n$ matrix and an $n \times p$ matrix involves $n$ multiplications and $n - 1$ additions, summing to approximately $2n$ floating-point operations per element, leading to a total of $2 \cdot m \cdot n \cdot p$ FLOPs. The mentioned matrices are listed in Table A1. Following the computation, an all-gather operation is required among the tensor parallelism members. Each of the $L/p$ transformer blocks on a single device must independently perform an all-gather operation, with the data volume transmitted per operation being $(t - 1)/t \cdot b \cdot s \cdot h \cdot S_e$ (where Se denotes the number of bytes per element, e.g., 8 for FP64).

**Table A1.** Matrices in transformer layers.

| Layer | Matrix | Size |
|---|---|---|
| Attention | $X$, input | $\mathcal{R}^{b\times s\times h}$ |
| | $W_Q/W_K/W_V$, projection matrices | $\mathcal{R}^{h\times h}$ |
| | $W^i_Q/W^i_K/W^i_V$, projection matrices on device $i$ | $\mathcal{R}^{h\times (h/t)}$ |
| | $Q^i/K^i/V^i$, projections on device $i$ | $\mathcal{R}^{b\times s\times (h/t)}$ |
| | $S^i$, attention scores on device $i$ | $\mathcal{R}^{b\times s\times s}$ |
| | $A^i$, softmax results on device $i$ | $\mathcal{R}^{b\times s\times s}$ |
| | $Z^i$, weighted results on device $i$ | $\mathcal{R}^{b\times s\times (h/t)}$ |
| | $Z$, weighted result | $\mathcal{R}^{b\times s\times h}$ |
| | $O$, output results of attention layer | $\mathcal{R}^{b\times s\times h}$ |
| | $W_O$, output matrix of attention layer | $\mathcal{R}^{h\times h}$ |
| MLP | $W^i_1$, hidden layer weight matrix | $\mathcal{R}^{h\times (4h/t)}$ |
| | $W^i_2$, output layer weight matrix | $\mathcal{R}^{(4h/t)\times h}$ |
| | $H^i$, hidden layer output | $\mathcal{R}^{b\times s\times (4h/t)}$ |
| | $G^i$ activation output | $\mathcal{R}^{b\times s\times (4h/t)}$ |
| | $O^i$, MLP output | $\mathcal{R}^{b\times s\times h}$ |

The output from the attention layer then reaches the MLP layer and starts the activation within the MLP layer. The $W_1$ matrix is typically designed to project the input feature dimension from $h$ (the hidden dimension of the model) to an expanded dimension (commonly $4h$), which is a standard practice to enhance the model's expressive capacity. When tensor parallelism is applied with $t$ GPUs (column-wise partitioning), the weight matrix is split along its column dimension (the expanded dimension 4h) to distribute computational load across GPUs. Specifically, each GPU in the tensor parallel group only retains a fraction of the columns. Thus, under such tensor parallelism, the size of the $W_1$ matrix on each GPU becomes $h \times 4h/t$. Following the computation of $W_1$, the tensor dimension

is compressed back to $h$ using the $W_2$ matrix. The matrix dimensions and corresponding FLOPs for both expansion and compression operations are detailed in Table A2. After completing the computation of MLP and attention layer activations for all transformer blocks within a GPU, the current pipeline stage must synchronize the activation tensors across the tensor parallel group via ring all-reduce, which involves two communication rounds: scatter-reduce and all-gather among tensor parallel members. The total number of communication steps is $2(t-1)$, with each step requiring a communication volume of $S_e \cdot b \cdot s \cdot h \cdot (t-1)/t$. Upon tensor synchronization, activated tensors are forwarded to the next pipeline stage for repeated computation and tensor parallel communication; once the final tensor synchronization concludes, the micro-batch's forward propagation finishes, and the backward propagation starts immediately. To optimize parallel speed, each subsequent micro-batch initiates its forward propagation as soon as the prior one completes computation in the current pipeline stage, enabling overlapping computation and communication across pipeline stages.

**Table A2.** FLOPs demands in forward propagation.

| Layer | Operation | FLOPs |
|---|---|---|
| Attention | $Q^i/K^i/V^i = X \cdot W^i_{Q/K/V}$ | $2 \cdot b \cdot s \cdot h \cdot (h/t)$ |
| | $Si = Qi \cdot (Ki)T/\sqrt{h/a}$ | $2 \cdot b \cdot s \cdot (h/t) \cdot s$ |
| | $Zi = Ai \cdot Vi$ | $2 \cdot b \cdot s \cdot (h/t) \cdot s$ |
| | $Concat\ Zi$ | / |
| | $O = Z \cdot W_O$ | $2 \cdot b \cdot s \cdot h^2$ |
| MLP | $H^i = X \cdot W^i_1$ | $8 \cdot b \cdot s \cdot h^2/t$ |
| | $G^i = GELU(H^i)$ | $24 \cdot b \cdot s \cdot (h/t)$ |
| | $O^i = G^i \cdot W^i_2$ | $8 \cdot b \cdot s \cdot h^2/t$ |

During the backward propagation, for each micro-batch at the last pipeline stage, the loss $L$ must first be computed to derive the derivative $\frac{\partial L}{\partial O}$. At each transformer layer, the derivatives of MLP layer parameters are calculated to facilitate parameter updates. An all-reduce operation is then executed among the tensor parallel group members to aggregate the MLP output gradients. Subsequently, the attention layer receives gradients from the MLP layer to compute the gradients of Q/K/V matrices. The corresponding FLOPs requirements are tabulated in Table A3. After the current micro-batch in the last pipeline stage completes computation, two strategies exist for gradient synchronization within the data parallel group: (1) asynchronous synchronization, where each device updates parameters immediately after its own micro-batch computation without waiting for others, enabling faster training at the cost of potential convergence instability and (2) synchronous synchronization, where all devices wait until gradients from all micro-batches in the current stage are computed and aggregated via all-reduce. In both strategies, each all-reduce entails $2 \cdot (d-1)$ communication steps and transfers $S_e \cdot N/(p \times t)$ bytes per step, where $N$ denotes the total model parameters. Whenever the current pipeline stage finishes gradient computation for a micro-batch, it passes gradients to the preceding stage, adhering to the pipeline parallelism's computation-communication overlap principle.

Once the first pipeline stage completes gradient synchronization for all micro-batches, an embedding table synchronization between the first and last stages of each pipeline is required, marking the completion of a training iteration. Consequently, the traffic pattern in the entire training process exhibits periodicity at the iteration level, while the traffic within each iteration shows temporal correlation with interleaving computation

and communication. This interplay poses higher challenges for the network design, as it necessitates an efficient handling of both periodic synchronization bursts and time-dependent traffic sequences to minimize training overhead.

**Table A3.** FLOPs demands in backward propagation.

| Layer | Operation | FLOPs |
|---|---|---|
| MLP | $\frac{\partial L}{\partial W_2^i} = (\frac{\partial L}{\partial O})^T \cdot G^i$ | $2 \cdot b \cdot s \cdot h \cdot (4h/t)$ |
| | $\frac{\partial L}{\partial G^i} = \frac{\partial L}{\partial O} \cdot \left(W_2^i\right)^T$ | $2 \cdot b \cdot s \cdot h \cdot (4h/t)$ |
| | $GELU'(H^i)$ | $6 \cdot b \cdot s \cdot (4h/t)$ |
| | $\frac{\partial L}{\partial H^i} = \frac{\partial L}{\partial G^i} \odot GELU'(H^i)$ | $b \cdot s \cdot (4h/t)$ |
| | $\frac{\partial L}{\partial W_1^i} = X^T \cdot \frac{\partial L}{\partial H^i}$ | $2 \cdot b \cdot s \cdot h \cdot (4h/t)$ |
| | $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial H^i} \cdot \left(W_1^i\right)^T$ | $2 \cdot b \cdot s \cdot h \cdot (4h/t)$ |
| Attention | $\frac{\partial L}{\partial A} = \frac{\partial L}{\partial O^i} \cdot \left(V^i\right)^T$ | $2 \cdot b \cdot s^2 \cdot h/a$ |
| | $\frac{\partial L}{\partial Q^i} / \frac{\partial L}{\partial K^i} / \frac{\partial L}{\partial V^i}$ | $2 \cdot b \cdot s^2 \cdot h/a$ |
| | $\frac{\partial L}{\partial W_Q^i} / \frac{\partial L}{\partial W_K^i} / \frac{\partial L}{\partial W_V^i}$ | $2 \cdot b \cdot s^2 \cdot h/a$ |
| | $\frac{\partial L}{\partial X}$ | $6 \cdot b \cdot s^2 \cdot h/a$ |

Based on the above process descriptions, Table 1 in Section 2.2 summarizes the communication traffic generated by the three parallelization methods within one iteration at one device, including their communication frequencies and per-communication data sizes. For data parallelism (DP), we assume the adoption of gradient accumulation, where gradients from all micro-batches are computed locally before global synchronization via ring all-reduce.

## References


1. Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. Gpt-4 technical report. *arXiv* **2023**, arXiv:2303.08774. [CrossRef]
2. Rajbhandari, S.; Rasley, J.; Ruwase, O.; He, Y. ZeRO: Memory optimizations Toward Training Trillion Parameter Models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*; IEEE: New York, NY, USA, 2020.
3. Shoeybi, M.; Patwary, M.; Puri, R.; LeGresley, P.; Casper, J.; Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv* **2019**, arXiv:1909.08053.
4. Huang, Y.; Cheng, Y.; Bapna, A.; Firat, O.; Chen, M.X.; Chen, D.; Lee, H.; Ngiam, J.; Le, Q.V.; Wu, Y.; et al. Gpipe: Easy scaling with micro-batch pipeline parallelism. *arXiv* **2019**, arXiv:811.06965v5.
5. Smith, S.; Patwary, M.; Norick, B.; LeGresley, P.; Rajbhandari, S.; Casper, J.; Liu, Z.; Prabhumoye, S.; Zerveas, G.; Korthikanti, V.; et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv* **2022**, arXiv:2201.11990.
6. Li, W.; Liu, X.; Li, Y.; Jin, Y.; Tian, H.; Zhong, Z.; Liu, G.; Zhang, Y.; Chen, K. Understanding Communication Characteristics of Distributed Training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*; ACM: New York, NY, USA, 2024; pp. 1–8.
7. Mellette, W.M.; McGuinness, R.; Roy, A.; Forencich, A.; Papen, G.; Snoeren, A.C.; Porter, G. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*; ACM: New York, NY, USA, 2017; pp. 267–280.
8. Shrivastav, V.; Valadarsky, A.; Ballani, H.; Costa, P.; Lee, K.S.; Wang, H.; Agarwal, R.; Weatherspoon, H. Shoal: A network architecture for disaggregated racks. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*; ACM: New York, NY, USA, 2019; pp. 255–270.
9. Ballani, H.; Costa, P.; Behrendt, R.; Cletheroe, D.; Haller, I.; Jozwik, K.; Karinou, F.; Lange, S.; Shi, K.; Thomsen, B.; et al. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*; ACM: New York, NY, USA, 2020.
10. Amir, D.; Saran, N.; Wilson, T.; Kleinberg, R.; Shrivastav, V.; Weatherspoon, H. Shale: A practical, scalable oblivious reconfigurable network. In *Proceedings of the ACM SIGCOMM 2024 Conference*; ACM: New York, NY, USA, 2024; pp. 449–464.

11. Liu, H.; Lu, F.; Forencich, A.; Kapoor, R.; Tewari, M.; Voelker, G.M.; Papen, G.; Snoeren, A.C.; Porter, G. Circuit switching under the radar with {REACToR}. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*; ACM: New York, NY, USA, 2014; pp. 1–15.
12. Proietti, R.; Cao, Z.; Nitta, C.J.; Li, Y.; Ben Yoo, S.J. A scalable, low-latency, high-throughput, optical interconnect architecture based on arrayed waveguide grating routers. *J. Light. Technol.* **2015**, *33*, 911–920. [CrossRef]
13. Wang, G.; Andersen, D.G.; Kaminsky, M.; Papagiannaki, K.; Ng, T.S.E.; Kozuch, M.; Ryan, M. c-Through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*; ACM: New York, NY, USA, 2010; pp. 327–338.
14. Farrington, N.; Porter, G.; Radhakrishnan, S.; Bazzaz, H.H.; Subramanya, V.; Fainman, Y.; Papen, G.; Vahdat, A. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*; ACM: New York, NY, USA, 2010; pp. 339–350.
15. Wang, W.; Khazraee, M.; Zhong, Z.; Ghobadi, M.; Jia, Z.; Mudigere, D.; Zhang, Y.; Kewitsch, A. {TopoOpt}: Co-optimizing network topology and parallelization strategy for distributed training jobs. *arXiv* **2022**, arXiv:2202.00433.
16. Liu, H.; Urata, R.; Yasumura, K.; Zhou, X.; Bannon, R.; Berger, J.; Dashti, P.; Jouppi, N.; Lam, C.; Li, S.; et al. Lightwave fabrics: At-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*; ACM: New York, NY, USA, 2023; pp. 499–515.
17. Poutievski, L.; Mashayekhi, O.; Ong, J.; Singh, A.; Tariq, M.; Wang, R.; Zhang, J.; Beauregard, V.; Conner, P.; Gribble, S.; et al. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM 22)*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 66–85. [CrossRef]
18. Jouppi, N.; Kurian, G.; Li, S.; Ma, P.; Nagarajan, R.; Nai, L.; Patil, N.; Subramanian, S.; Swing, A.; Towles, B.; et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*; ACM: New York, NY, USA, 2023.
19. Benjamin, J.L.; Gerard, T.; Lavery, D.; Bayvel, P.; Zervas, G. PULSE: Optical circuit switched data center architecture operating at nanosecond timescales. *J. Light. Technol.* **2020**, *38*, 4906–4921. [CrossRef]
20. Qian, K.; Xi, Y.; Cao, J.; Gao, J.; Xu, Y.; Guan, Y.; Fu, B.; Shi, X.; Zhu, F.; Miao, R.; et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*; ACM: New York, NY, USA, 2024; pp. 691–706.
21. Liang, C.; Song, X.; Cheng, J.; Wang, M.; Liu, Y.; Liu, Z.; Zhao, S.; Cui, Y. NegotiaToR: Towards a simple yet effective on-demand reconfigurable datacenter network. In *Proceedings of the ACM SIGCOMM 2024 Conference*; ACM: New York, NY, USA, 2024; pp. 415–432.
22. Xue, X.; Calabretta, N. Nanosecond optical switching and control system for data center networks. *Nat. Commun.* **2022**, *13*, 2257. [CrossRef] [PubMed]
23. Porter, G.; Strong, R.; Farrington, N.; Forencich, A.; Chen-Sun, P.; Rosing, T.; Fainman, Y.; Papen, G.; Vahdat, A. Integrating microsecond circuit switching into the data center. *ACM SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 447–458. [CrossRef]
24. Chen, K.; Singla, A.; Singh, A.; Ramachandran, K.; Xu, L.; Zhang, Y.; Wen, X.; Chen, Y. OSA: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Trans. Netw.* **2013**, *22*, 498–511. [CrossRef]
25. Feng, S.; Zhang, J.; Zhou, H.; Li, X.; Ji, Y. Mercury: A Reconfigurable Datacenter Network with Collaborative Optical Timeslot Switching and Optical Circuit Switching. In *Proceedings of the Optical Fiber Communication Conference*; Optica Publishing Group: Washington, DC, USA, 2025.
26. Bhat, K.U.P.; Ravish, H.; Anirudha, V.; Prabhavathi, P. Design and Verification of Peripheral Component Interconnect Express (PCIe) 3.0. *Int. Res. J. Eng. Technol.* **2020**, *7*, 905–919.
27. Urata, R.; Liu, H.; Yasumura, K.; Mao, E.; Berger, J.; Zhou, X.; Lam, C.; Bannon, R.; Hutchinson, D.; Nelson, D.; et al. Mission Apollo: Landing Optical Circuit Switching at Datacenter Scale. *arXiv* **2022**, arXiv:2208.10041. [CrossRef]
28. Spanke, R. Architectures for large nonblocking optical space switches. *IEEE J. Quantum Electron.* **1986**, *22*, 964–967. [CrossRef]
29. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is all you need. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017)*; Advances in neural information processing systems; ACM: New York, NY, USA, 2017; p. 30.
30. Valiant, L.G.; Brebner, G.J. Universal schemes for parallel communication. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*; ACM: New York, NY, USA, 1981; pp. 263–277.
31. Devlin, J.; Chang, M.-W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2019**, arXiv:1810.04805. [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.