OptimusNIC: Offloading Optimizer State to SmartNICs for Efficient Large-Scale AI Training

Achref Rebai
KAUST

Marco Canini
KAUST

Abstract

LLM training is a demanding workload that requires careful coordination among hardware components, ensuring high GPU utilization, rapid data transfer, and minimal memory overhead. This synergy leads to an efficient AI training system. Scaling LLMs encounters memory constraint challenges, particularly the optimizer state, whose size in bytes scales with a factor of 12× the number of model parameters. In this paper, we explore the impact of offloading the optimizer state and parameter update operation to a SmartNIC. OptimusNIC reduces communication overhead between GPUs, minimizes GPU computation (by handling the optimizer step externally), and significantly decreases memory requirements. These improvements allow GPUs to accommodate additional model layers and efficiently train and fine-tune models with minimal resources. In addition to examining the impact of OptimusNIC, we evaluate DeepSpeed's ZeRO-Infinity framework, identifying key limitations associated with using the CPU as the offloading target and we demonstrate why OptimusNIC offers a more efficient alternative. Furthermore, we analyze the usability of one of the target platforms for OptimusNIC: the NVIDIA BlueField-2 DPU.

CCS Concepts

• **Networks → In-network processing**.

Keywords

In-Network Computing, Training, Offloading, Memory management

ACM Reference Format:
Achref Rebai and Marco Canini. 2025. OptimusNIC: Offloading Optimizer State to SmartNICs for Efficient Large-Scale AI Training. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3721146.3721960

1 Introduction

Nowadays, distributed AI training systems try to balance between computation, communication, and memory usage to deliver the best performance with respect to some metrics. A distributed training workload comprises four principal steps, forward step, backward step, gradient aggregation, and parameter update. The four operations require storing multiple states: **model parameters**, **gradients**, and **optimizer state**.

Training large-scale AI models, particularly Large Language Models (LLMs), requires vast computational resources and efficient parallelism strategies. Traditional approaches such as Data Parallelism (DP) and Model Parallelism (MP) struggle to keep pace with the ever-growing model sizes, even with the latest GPU hardware. To address these limitations, state-of-the-art systems like FSDP [26], MegatronLM [22], and DeepSpeed [17] have combined advanced parallelism techniques (e.g., Pipeline Parallelism [11] and Tensor Parallelism [14]) and introduced model partitioning strategies. These innovations help scale training to larger models but come with significant memory and computation overheads.

A notable optimization introduced by frameworks like DeepSpeed ZeRO++ [24] is communication reduction by maintaining a secondary copy of the model and using parameter and gradient compression. This allows the system to reduce communication time. However, this comes at the cost of increased GPU workloads: GPUs must now execute additional compression kernels, leading to resource contention and sub-optimal performance making the framework struggle to scale efficiently [2, 25].

The core challenge is that GPUs are increasingly becoming overburdened with both core compute tasks (forward, backward passes, and optimizer state updates) and auxiliary compute tasks (communication, and compression). This resource contention not only limits the ability to train larger models efficiently but also increases training time and costs. Furthermore, the large memory requirements of these models force the use of more GPUs, driving up infrastructure costs and making large-scale training even more expensive.

Recent work has shown that offloading GPU operations to SmartNICs (network interface cards with compute capabilities) offer a promising solution by alleviating GPU workloads and optimizing resource utilization. FPGA-based SmartNICs were studied and used to offload compression

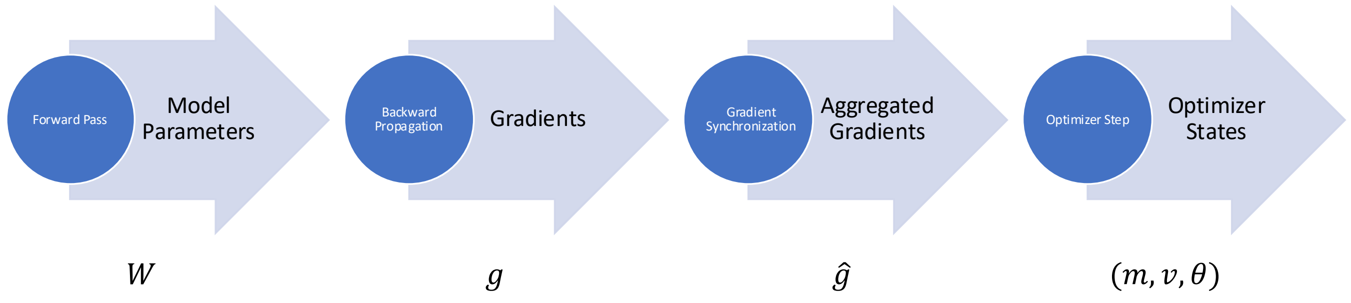

Figure 1: Distributed Training Iteration Steps

operations [16, 19, 20]. In-network aggregation and collective communication offloading [9, 13, 15, 21] has also been studied as an approach to reduce communication overhead in distributed training systems. By performing operations such as gradient aggregation directly within network devices like SmartNICs or programmable switches, these methods can optimize data transfers between GPUs.

Optimizer state offloading was first studied in ZeRO Infinity [18], which introduced the idea of offloading optimizer state, gradients, and parameters to CPU or NVMe storage, significantly reducing the memory footprint per GPU, and enabling the training of larger models. However, this approach introduces I/O operations and faces increased communication overhead between the GPU and host memory. Our measurements (§ 5) show that non-overlapped communication in ZeRO-Infinity accounts for up to 27% of the optimizer step's time.

Building upon the promising idea of optimizer offload, we introduce OptimusNIC, which offloads critical operations from GPUs to SmartNICs. Unlike existing solutions, OptimusNIC stores optimizer state across SmartNICs while also offloading parameter updates and gradient communication, reducing GPU resource contention and improving training efficiency. SmartNICs are particularly well-suited for this role because they are closer to the network than CPUs, reducing the overhead associated with transferring optimizer state to general-purpose CPUs. Since gradient reduction happens directly on the SmartNIC, optimizer state updates can be performed locally, avoiding unnecessary data movement between GPUs and CPUs. By shifting these operations away, OptimusNIC allows GPUs to focus exclusively on forward and backward computations, improving overall efficiency. Additionally, memory requirements per GPU are reduced by 4×, as the optimizer state is stored on SmartNICs, enabling the system to host larger models or train/finetune models using fewer resources.

In this work-in-progress paper, we primarily explore the impact of OptimusNIC on GPU memory requirements (§ 3.1), computation (§ 3.2), and communication time (§ 3.3). We highlight the various challenges (§ 4) associated with these aspects, along with testing and profiling ZeRO-Infinity (§ 5) to assess its efficiency and overheads. These insights pave the way for more effective strategies in optimizing large-scale model training.

2 Background

2.1 Distributed Training

Distributed training enables scaling machine learning models across multiple devices by dividing computation and memory workloads. The process consists of four key steps (Fig. 1):
Forward Pass: Computes predictions using stored model parameters (W), which remain unchanged during this phase.
Backward Propagation: Computes gradients (g) via backpropagation for updating model parameters.
Gradient Aggregation: Synchronizes gradients ($\hat{g}$) across GPUs to maintain consistency, introducing communication overhead.
Optimizer Step: Updates model parameters using the optimizer state (m, v, θ), which require significant memory storage.

Efficiently managing these operations is crucial to reducing memory and communication overheads in large-scale training systems. To scale training further, modern distributed training frameworks employ a combination of parallelism strategies:
Data Parallelism (DP) replicates the model across multiple GPUs, where each GPU processes different mini-batches of data and synchronizes gradients after backpropagation.
Tensor Parallelism (TP) splits individual layers across multiple GPUs, allowing them to collaboratively compute matrix multiplications for large model parameters.

Pipeline Parallelism (PP) partitions the model across multiple GPUs, with different GPUs processing different layers sequentially in a pipelined manner to optimize resource utilization and minimize memory overhead.
3D Parallelism combines DP, TP, and PP to maximize efficiency when training large-scale models.

2.2 Memory Requirements

Memory requirements for mixed-precision training (the defacto standard for LLM pre-training [7, 8]) primarily consist of **model parameters** (W), **gradients** (g), and **optimizer state** (m, v, θ). Assuming as per best practices **half-precision** (16-bit) for parameters and gradients and **full precision** (32-bit) for optimizer state, the total bytes required per GPU for a model with D parameters is:

$$M = 2D + 2D + 12D = 16D.$$

ZeRO mitigates this by partitioning memory across N_p devices, Specifically, ZeRO-Stage 3 partitions all model states: optimizer state, gradients, and parameters across data-parallel processes, reducing the per-device requirement to $\frac{M}{N_p}$ bytes.

Additionally, ZeRO-Infinity extends these capabilities by enabling the offloading of model states to CPU or NVMe memory, further alleviating GPU memory constraints.

2.3 SmartNICs

SmartNICs are specialized network adapters that integrate compute capabilities, allowing for offloading tasks traditionally handled by CPUs or GPUs. These devices enhance system efficiency by reducing data transfer overhead and enabling in-network computation.

Several types of SmartNICs exist [12], each catering to different computational needs. We consider two primary targets:
NVIDIA BlueField (BF) SmartNICs: BlueField SmartNICs combine a System-on-Chip (SoC) ARM-based compute cores with high-speed networking capabilities, enabling offloading of network processing tasks and hardware accelerators for specific security (e.g., encryption) and storage (e.g., compression) operations.
FPGA-based SmartNICs: These SmartNICs leverage Field Programmable Gate Arrays (FPGAs) for high customizability and efficiency in specific workloads. Examples include the Xilinx Alveo™ series, which provides flexible hardware acceleration for AI and networking tasks. While more power-efficient than general-purpose processors, FPGA-based SmartNICs require extensive engineering effort for adoption.

3 Opportunities

Our work explores the impact of offloading the optimizer state in three key areas:
Memory Efficiency: By reducing GPU memory usage, OptimusNIC enables either larger models to fit within the same hardware or efficient fine-tuning with fewer GPUs, making it particularly beneficial for workloads where parameter-efficient fine-tuning (PEFT) methods are not assumed.
Compute Offloading: Computing updates to optimizer state within SmartNICs frees up cycles on GPUs for forward and backward passes, improving resource utilization.
Communication Overhead: Offloading optimizer state frees GPU memory, allowing more layers to be hosted per GPU. This reduces the need for intermediate activation transfers between GPUs, minimizing per-layer communication overhead and improving training efficiency.

3.1 Lower Memory Per GPU

The memory usage can be significantly reduced by leveraging OptimusNIC to offload the optimizer state. In a model-partitioned setting, such as ZeRO, memory requirements for the optimizer state, gradients, and model parameters are distributed across N_P parallel workers. This partitioning results in a per-GPU memory footprint corresponding to $\frac{16D}{N_P}$ bytes.

With OptimusNIC offloading the optimizer state, the memory requirement for the optimizer state ($\frac{12 \cdot D}{N_P}$) is entirely removed from the GPU, leaving only the gradients and model parameters. Thus the per-GPU memory requirement becomes $\bar{M} = \frac{4 \cdot D}{N_P}$ bytes.

This optimization achieves a 4× reduction in per-GPU memory requirement, freeing up resources to host larger models and reducing the need for additional hardware. By combining this approach with ZeRO's partitioning strategies, OptimusNIC enhances both memory efficiency and scalability in distributed training.

3.2 Sparing GPU Cycles

To estimate the compute time savings from offloading computation, we experiment with a model composed entirely of TransformerDecoderLayer, which serve as the fundamental building blocks of large language models (LLMs). Each decoder layer contains 151M parameters, and we use a total of 4 decoder layers. The experiments run on NVIDIA A100 GPUs while we vary sequence length in $[64, 128, 256, 512]$.

For each configuration, we measure the ratio of time consumed by the optimizer step relative to the total iteration time. Table 1 demonstrates that this ratio decreases as sequence length increases. This trend is attributed to the growing computational cost of forward and backward passes, which dominate the overall iteration time for longer sequences. We conclude that the optimizer step remains a computational bottleneck but accounts for a smaller fraction of the overall training time. This suggests that offloading the optimizer step could provide performance gains.

Table 1: Ratio of optimizer step time to total iteration time.

Sequence Length	64	128	256	512
Optimizer Step Ratio (%)	39.75	26.55	16.08	9.08

3.3 Less Communication Volume

Using the same model as in the previous experiment, we compared the training performance on a single GPU versus a 4-GPU setup using model parallelism. We use a factor of 4 to represent the scaling benefit due to lower memory requirements, which implies that less communication would occur. The model is partitioned across 4 NVIDIA A100 GPUs, with each GPU hosting one partition. We evaluate the impact of model partitioning on forward and backward passes while we vary sequence length in $[64, 128, 256, 512]$. We measure the time required for both the forward and backward passes, calculating the median over 50 trials for each configuration.

Table 2 indicates that while model parallelism enables training larger models that exceed the memory capacity of a single GPU, it also introduces significant communication overhead that negatively impacts training efficiency. Specifically, our findings show that the forward pass experiences a slowdown of up to 7.8%, while the backward pass sees up to 5% degradation. The slowdown is particularly pronounced at smaller sequence lengths, suggesting that the cost of inter-GPU communication outweighs the computational benefits of distributing the model. Moreover, as sequence length increases, the performance gap persists, demonstrating that communication remains a limiting factor even for larger input sizes. These results emphasize that, for models that fit within a single GPU's memory, avoiding model parallelism can lead to better computational efficiency.

4 Challenges

Using SmartNICs to offload the optimizer state and update step presents certain challenges. Compared to GPUs and CPUs, current SmartNICs have limited memory and compute power. We are curious to explore to what extent SmartNICs may represent a promising target for offloading, and if they fall short, what margin of improvements are necessary to flip the calculus.

4.1 Do SmartNICs Have Enough Memory?

As discussed in the previous section, offloading the optimizer state means that a SmartNIC would store 3× more state than a GPU (i.e., $12D$ vs. $4D$ bytes). This may be reasonable if we assume a well balanced server design where each GPU is paired to an accompanying SmartNIC, which appears to agree with current trends at industry [5, 6].

Beyond a relative perspective, we must be mindful of absolute memory requirements. Currently, high-end GPUs like A100 and H100 have up to 80 and 120 GB of memory, respectively. As an example, consider the LLaMA 70B model (1120 GB of model states). Let's reason how this model may fit in two NVIDIA DGX A100 nodes—each equipped with 8 GPUs, totaling GPU 1280 GB of memory. By employing a 3D parallelization strategy with ($DP = 2$, $TP = 2$, $PP = 4$), the model is distributed across 8 GPUs. This setup assigns approximately 8.75 B parameters per GPU (140 GB).

ZeRO (Stage 3) reduces the memory footprint per GPU, yielding 70 GB per GPU. While this requirement is lower than an A100's 80 GB memory, this allocation is too tight, as additional memory is needed for activations and other memory overheads. This is where OptimusNIC becomes advantageous, as it offloads the optimizer state accounting for 35 GB per GPU—thereby enabling the model to be run on two DGX A100 nodes without scaling out three nodes.

To achieve this, a SmartNIC needs several GBs of memory. Current commodity SmartNICs like NVIDIA BlueField-2 and -3 have 16 and 32 GBs of onboard RAM, respectively. As this capacity is significant yet insufficient, this indicates that it may be necessary to leverage host memory for additional storage. This challenge can be mitigated through an efficient prefetching strategy to overlap CPU–SmartNIC communication and minimize latency. An plausible realization could consider FPGA-based SmartNICs, known for their high programmability and flexibility. While most high-end FPGAs currently provide 16 GB to 32 GB of onboard memory, they enable custom optimizations that can help manage memory constraints effectively. A commercially available candidate for this workload is AMD's Xilinx Alveo™ V80 [1] Compute Accelerator Card, which combines 32 GB of HBM with 32 GB of DDR4 expansion memory, providing ample bandwidth and storage.

4.2 Can A BlueField Run Optimizer Steps?

Aside from memory capacity, offloading optimizer update steps require compute capability. While we expect a SmartNIC to be far less capable than a GPU or CPU (particularly for SoC-based SmartNICs like the BlueField design), we wish to quantify its performance and analyze the underlying technological trends.

We experiment by executing the optimizer step on a BlueField-2 (BF-2) SmartNIC with 8 ARMv8 A72 cores at 2.75 GHz. We note that the optimizer step is an element-wise operation over the model parameters and thus it can be easily parallelized across multiple cores. Specifically, we measure, while varying the thread count, the time required for an optimizer step in a simple transformer model, which consists of one TransformerDecoderLayer (4096 embedding size, 302M parameters). The optimizer step time is measured over 25

Table 2: Breakdown and relative slowdown of forward and backward pass in single- vs. 4-GPU model parallelism.

Sequence	Forward Pass			Backward Pass		
Length	Single GPU [ms]	4 GPUs [ms]	Slowdown [%]	Single GPU [ms]	4 GPUs [ms]	Slowdown [%]
512	1084.21	1141.36	5.27	1906.43	1956.76	2.64
256	543.35	556.59	2.44	957.61	970.48	1.34
128	274.22	281.03	2.48	483.22	489.62	1.32
64	143.75	154.96	7.80	245.84	258.28	5.06

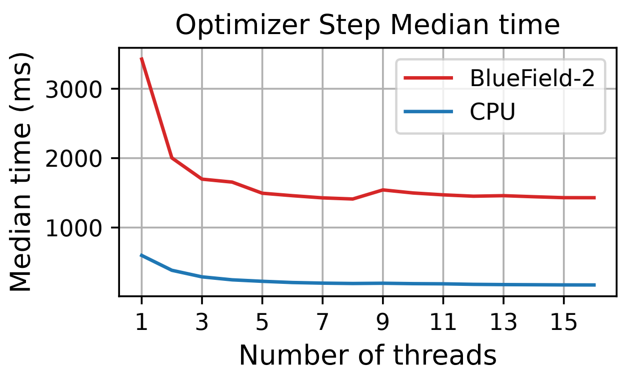

Figure 2: Median optimizer step time (BF-2 vs. CPU).

trials. As a baseline, we compare against an AMD EPYC 7763 64-core CPU.

Figure 2 shows that as the threads increase, the optimizer step time decreases in both cases, but the BF-2 remains slower than the CPU by a factor of approximately 5–8×. This indicates that for offloading on BF-2 to be viable, the optimizer step workload needs to be spread over several SmartNICs to recover baseline performance.

Meanwhile, SmartNIC technology is improving. BF-3 [3] features 16 Arm Cortex-A78 cores, doubling the core count of BF-2 [4], which is equipped with 8 Arm Cortex-A72 cores. In terms of CPU frequency, BF-2 cores can reach up to 2.75 GHz, while BF-3 cores operate at frequencies up to 2.0 GHz in the E-Series and up to 2.133 GHz in the P-Series. Additionally, BF-3 transitions from DDR4 to DDR5 memory, significantly enhancing memory bandwidth. These improvements make BF-3 a viable candidate for offloading optimizer computations and bridging the performance gap between SmartNICs and high-end CPUs. However, we are currently unable to experiment with BF-3 due to hardware unavailability.

The CPU achieves a peak performance of 166 ms using 33 cores, while the best performance on BF-2 reaches 1429 ms. High performance AI nodes typically feature multiple GPUs, with each GPU linked to its own SmartNIC. Distributing the optimizer step across all SmartNICs helps close this performance gap, enabling a more scalable and efficient offloading strategy. This is made possible by in-network gradient aggregation [21], which allows the resulting gradients to be efficiently distributed across the node's SmartNICs using a reduce-scatter collective operation. Combined with the hardware improvements in BF-3, these SmartNICs present a strong candidate for OptimusNIC, enhancing both performance and scalability.

5 ZeRO-Infinity Analysis

In the previous section, we analyzed the optimizer step performance on BF-2 and argued that a set of BF-3 SmartNICs can collectively achieve performance comparable to high-end data center CPUs. In this section, we assess the impact of CPU-based offloading, highlight its limitations, and demonstrate that OptimusNIC presents a more efficient option.

To explore the impact of offloading the optimizer state on training performance, we use ZeRO-Stage 3 as our baseline since ZeRO-Infinity is built on top of it and relies on its mechanisms for offloading the optimizer state. We compare two configurations: ZeRO-Stage 3 alone and ZeRO-Stage 3 with optimizer state offloading to CPU memory. In this experiment, we measure the end-to-end iteration time of two models: one with a single(Model 1) and another(Model 2) with two TransformerDecoderLayer layers. The experiments run on two nodes, each equipped with two NVIDIA A100 GPUs and an AMD EPYC 7763 64-core processor.

Table 3 shows that while ZeRO-Infinity reduces the memory requirement per GPU by offloading the optimizer state to the CPU, it also introduces significant computational overhead, leading to a notable increase in iteration time. The optimizer step, in particular, experiences a substantial slowdown due to communication delays and the lack of overlap between computation and data transfers. As a result, the total iteration time increases by 59.2% for the first model and 51.7% for the second model when offloading is enabled. This performance degradation is primarily driven by the optimizer step, which slows down by more than 12× compared to executing on the GPU. These findings highlight the limitations of CPU-based offloading and underscore the need for a more efficient alternative, such as OptimusNIC, which can mitigate these overheads.

Table 3: Performance Breakdown of ZeRO-Infinity: Impact of Offloading on Training Iteration Time

Model	Forward (ms)	Backward (ms)	Optimizer Step (ms)	Total (ms)
Model 1 (No Offloading)	147.283	255.772	18.593	411.589
Model 1 (With Offloading)	152.419	243.095	255.822	655.335
Model 2 (No Offloading)	200.493	550.389	27.788	777.762
Model 2 (With Offloading)	205.796	554.082	406.518	1180.145

Table 4: Optimizer Step Profiling Results

Operation	Self CPU %	Self CPU Time (ms)
`CPUAdam.step`	26.02	59.344
`aten::copy_`	19.60	44.699
`aten::isinf`	9.94	22.662
`cudaMemcpyAsync`	7.75	17.676
`aten::any`	7.20	16.432
`aten::ne`	6.92	15.791
`aten::mul_`	6.75	15.403
`aten::abs`	6.74	15.382
`aten::eq`	6.71	15.296

In ZeRO-Infinity the optimizer step is performed after performing the backward step across all the layers. During the optimizer step, gradients computed on the GPU must be transferred to the CPU for processing. This transfer is followed by parameter updates, which are then sent back to the GPU. The absence of overlap between these operations results in GPU idleness while waiting for the CPU to complete the optimization process. Consequently, the GPU, designed for high-throughput computations, remains underutilized, leading to increased iteration time and reduced training efficiency.

To better understand the overhead caused by the optimizer step during training, we profiled the operations performed as part of the optimizer step in a ZeRO Infinity training run using Model 1. Table 4 shows that the majority of execution time is spent on parameter update operations including `CPUAdam.step` and others. Besides, a significant portion of time (27.35%) is spent on tensor copies, with `cudaMemcpyAsync` and `aten:: copy_` contributing 7.75% and 19.60%, respectively. These operations introduce substantial communication overhead due to frequent data transfers stalling the GPU while waiting for the updated parameters to start the next training iteration. These overheads grow almost linearly with the number of TransformerDecoderLayer layers making it difficult to scale training for LLMs.

In-network aggregation enables gradients to reside in SmartNIC memory, reducing the need for frequent data transfers. Since gradients are aggregated layer by layer meaning each model layer's gradients are sent for aggregation once its backward pass is complete, the optimizer step can be executed on OptimusNIC immediately after aggregation. This approach allows overlapping not only gradient communication and computation but also the optimizer step itself. Results from Table 3 show that the optimizer step time is always shorter than the combined duration of the forward and backward passes. This indicates that overlapping these operations is feasible, as the peak performance of multiple SmartNICs and CPU-based optimizer steps are comparable.

6 Related Work

LuWu [23] addresses these inefficiencies by offloading the optimizer state and parameters to an in-network optimizer node, utilizing SmartNIC-SmartSwitch co-optimization. This design enables the in-network optimizer to manage parameter storage and updates, subsequently broadcasting the updated parameters to GPU workers. While this approach alleviates some of the CPU-related bottlenecks, it introduces potential communication constraints due to the centralized nature of the in-network optimizer node, akin to bandwidth limitations observed in parameter-server architectures.
Cerebras Systems Weight Streaming [10] disaggregates parameter storage from compute using their MemoryX service, enabling the training of larger models without requiring parameters to reside in compute memory. While weight streaming minimizes the need for parameter replication and reduces communication bottlenecks, it relies heavily on Cerebras-specific hardware like the Wafer-Scale Engine (WSE-2) and SwarmX interconnect fabric to achieve these gains. These solutions, while addressing specific issues in distributed training, underline the necessity for more efficient and generalized approaches like OptimusNIC, which optimally balances communication, computation, and memory requirements across standard hardware setups.

7 Conclusion

While frameworks like ZeRO-Infinity attempt to alleviate memory constraint issues by offloading optimizer state and parameters to CPU or NVMe memory, such solutions often introduce significant bottlenecks, including heavy I/O operations and limited overlap between computation and communication. These inefficiencies emphasize the need for more advanced, hardware-optimized approaches. OptimusNIC presents a promising solution by offloading optimizer state management and parameter updates to SmartNICs. This approach reduces communication delays, minimizes GPU resource contention, and enhances memory efficiency, enabling GPUs to focus on compute-heavy tasks. As a future direction, the design and implementation of OptimusNIC should prioritize optimizing the partitioning of the optimizer state across multiple SmartNICs and host memory while ensuring efficient execution of optimizers on SmartNICs. These improvements will enable the training of more complex models by minimizing bottlenecks and maximizing hardware utilization, ultimately enhancing scalability and performance.

Acknowledgments

This publication is based upon work supported by the King Abdullah University of Science and Technology Research Funding (KRF) under Award No. ORA-CRG2022-5017. For computer time, this research used the resources of the Supercomputing Laboratory at KAUST.

References

[1] AMD. *AMD Alveo™ V80 Compute Accelerator Card.*
[2] Qiaoling Chen, Qinghao Hu, Guoteng Wang, Yingtong Xiong, Ting Huang, Xun Chen, Yang Gao, Hang Yan, Yonggang Wen, Tianwei Zhang, and Peng Sun. AMSP: Reducing Communication Overhead of ZeRO for Efficient LLM Training, 2024.
[3] NVIDIA Corporation. *NVIDIA BlueField-3 DPU Specifications.*
[4] NVIDIA Corporation. *NVIDIA BlueField-2 DPU Specifications.*
[5] NVIDIA Corporation. *NVIDIA DGX A100 User Guide*, 2023.
[6] NVIDIA Corporation. *NVIDIA DGX H100/H200 User Guide*, 2023.
[7] Abhimanyu Dubey et. al. The Llama 3 Herd of Models, 2024.
[8] Jinze Bai et al. Qwen Technical Report, 2023.
[9] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. Efficient Sparse Collective Communication and Its Application to Accelerate Distributed Deep Learning. In *SIGCOMM*, 2021.
[10] Stewart Hall, Rob Schreiber, and Sean Lie. Training Giant Neural Networks Using Weight Streaming on Cerebras Wafer-Scale Clusters, 2022. Cerebras Systems White Paper.
[11] Yanli Huang, Shuyong Zhu, Xuying Meng, and Yujun Zhang. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *NeurIPS*, 2019.
[12] Elie F. Kfoury, Samia Choueiri, Ali Mazloum, Ali AlSabeh, Jose Gomez, and Jorge Crichigno. A Comprehensive Survey on SmartNICs: Architectures, Development Models, Applications, and Research Directions. *IEEE Access*, 12:107297–107336, 2024.
[13] Mikhail Khalilov, Salvatore Di Girolamo, Marcin Chrapek, Rami Nudelman, Gil Bloch, and Torsten Hoefler. Network-Offloaded Bandwidth-Optimal Broadcast and Allgather for Distributed AI. In *SC*, 2024.
[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Commun. ACM*, 60(6), 2017.
[15] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-Network Aggregation for Multi-Tenant Learning. In *NSDI*, 2021.
[16] Rui Ma, Evangelos Georganas, Alexander Heinecke, Sergey Gribok, Andrew Boutros, and Eriko Nurvitadhi. FPGA-Based AI Smart NICs for Scalable Distributed AI Training Systems. *IEEE Computer Architecture Letters*, 21(2), 2022.
[17] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *SC*, 2020.
[18] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *SC*, 2021.
[19] Achref Rebai, Mubarak Adetunji Ojewale, Anees Ullah, Marco Canini, and Suhaib A. Fahmy. SqueezeNIC: Low-Latency In-NIC Compression for Distributed Deep Learning. In *Proceedings of the SIGCOMM Workshop on Networks for AI Computing*, 2024.
[20] Qingqing Ren, Shuyong Zhu, Xuying Meng, and Yujun Zhang. Gradient Compression for Distributed Training in Deep Learning. In *DCC*, 2022.
[21] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *NSDI*, 2021.
[22] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, 2020.
[23] Mo Sun, Zihan Yang, Changyue Liao, Yingtao Li, Fei Wu, and Zeke Wang. LuWu: An End-to-End In-Network Out-of-Core Optimizer for 100B-Scale Model-in-Network Data-Parallel Training on Distributed GPUs, 2024.
[24] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. ZeRO++: Extremely Efficient Collective Communication for Large Model Training. In *ICLR*, 2024.
[25] Dong Xu, Yuan Feng, Kwangsik Shin, Daewoo Kim, Hyeran Jeon, and Dong Li. Efficient Tensor Offloading for Large Deep-Learning Model Training based on Compute Express Link. In *SC*, 2024.
[26] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.*, 16(12), 2023.