SmartNIC-GPU-CPU Heterogeneous System for Large Machine Learning Model with Software-Hardware Codesign

Anqi Guo
Department of Electrical & Computer Engineering
Boston University
Boston, Massachusetts, USA
anqiguo@bu.edu

Yuchen Hao
Meta Platforms
Menlo Park, California, USA
haoyc@meta.com

Xiteng Yao
Department of Electrical & Computer Engineering
Boston University
Boston, Massachusetts, USA
xtyao@bu.edu

Shining Yang
Boston University
Boston, Massachusetts, USA
shiningy@bu.edu

Jianyu Huang
Meta Platforms
Menlo Park, California, USA
jianyuhuang@meta.com

Tony (Tong) Geng
Department of Electrical and Computer Engineering
University of Rochester
Rochester, New York, USA
tong.geng@rochester.edu

Martin Herbordt
Department of Electrical & Computer Engineering
Boston University
Boston, Massachusetts, USA
herbordt@bu.edu

Abstract

The rapid growth of large machine learning models, from billions to trillions of parameters, has led to powerful AI capabilities that increasingly impact everyday life. However, this expansion in model size has surpassed the capacity of GPU memory. As a result, GPU clusters—built by aggregating multiple GPUs—have scaled up significantly to accommodate these models. To address this scalability challenge, and to make large-model training more widely accessible, researchers have proposed heterogeneous systems. These systems leverage CPUs and secondary memory to offload storage and computation onto these devices, thereby reducing the total number of GPUs required for training. Despite their promise, such heterogeneous systems have so far faced challenges in achieving high efficiency and performance.

In response, we address this problem with SmartNIC-GPU-CPU (SGC), a heterogeneous system, enhanced with SmartNICs, for training large machine learning models with software-hardware codesign. SGC increases system performance and efficiency while simultaneously reducing power consumption and overall costs.

In SGC, SmartNICs serve as an intermediate layer that seamlessly connects the heterogeneous components. By implementing optimization techniques such as prefetching, buffering, and dynamic scheduling and control, SmartNICs streamline the data pipeline, minimizing idle times and overlapping communication with computation. In addition, system configuration software optimizes the system and model settings for maximal efficiency, given different system specifications. Experiments demonstrate that the SGC system achieves an improvement of over 1.6× in training throughput over the baseline for a 100B parameter model.

ICS '25, Salt Lake City, UT, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3729514

CCS Concepts

• **Hardware → Networking hardware**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies → Machine learning**.

Keywords
Heterogeneous System, SmartNIC, Machine learning

ACM Reference Format:
Anqi Guo, Yuchen Hao, Xiteng Yao, Shining Yang, Jianyu Huang, Tony (Tong) Geng, and Martin Herbordt. 2025. SmartNIC-GPU-CPU Heterogeneous System for Large Machine Learning Model with Software-Hardware Codesign. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3729514

1 Introduction

In recent years, large-scale deep learning (DL) has advanced rapidly and come to dominate the AI application landscape. Large models provide significantly better sample efficiency and performance [24]. With the rise of attention-based deep learning models, model size has grown exponentially. For instance, large language models like Meta's LlaMA [51, 52], OpenAI's ChatGPT [35], and Google's LaMDA [50] have enabled impressive AI capabilities that are transforming our daily lives. The functionality of these powerful models is largely attributable to their immense size: 70 billion parameters for LlaMA 2, 175 billion for GPT-3, 540 billion for PaLM, and 1.8 trillion for GPT-4. This trend of ever-increasing model size (see Figure 1) is expected to continue for some time [24].

These large deep-learning models far exceed the memory capacity of a single GPU. Even with GPU clusters, training systems still face fundamental limitations in fitting models with billions to trillions of parameters into the GPUs' aggregate memory. To address this constraint, memory-efficient optimizations such as the Zero Redundancy Optimizer (ZeRO) [39] and Pytorch Fully Sharded Data Parallel (FSDP) [61] have been introduced, trading extra communication for memory utilization. These approaches allow model size to scale proportionally with the number of devices, making it feasible to train large models on a cluster of GPUs.

Although ZeRO enables the training of large-scale models, the memory wall still poses a significant challenge. Today's models demand hundreds of gigabytes to multiple terabytes of GPU memory just to hold their parameters. Yet, GPU memory capacity is not keeping pace with the exponential growth in model size (as shown in Figure 1). One way to handle these massive models is to keep scaling up the number of GPUs; however, this approach quickly becomes impractical. First, performance does not scale linearly with the number of GPUs due to diminishing returns and communication overhead. Second, as cluster size grows, system overhead and communication demands become major bottlenecks. Third, expanding hardware capacity proportionally increases power consumption, which is already considerable. Lastly, although technologies like NVLink can improve the

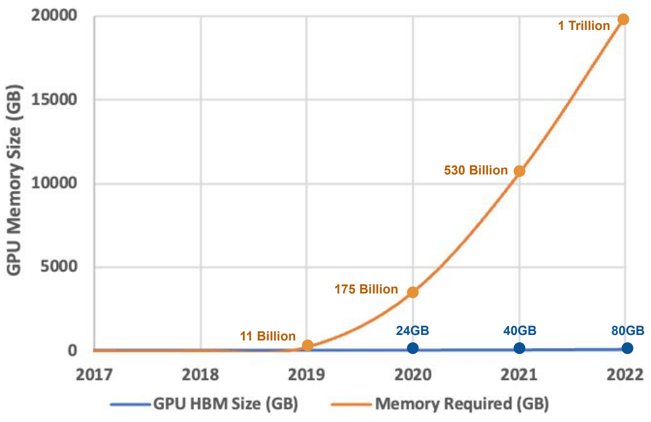


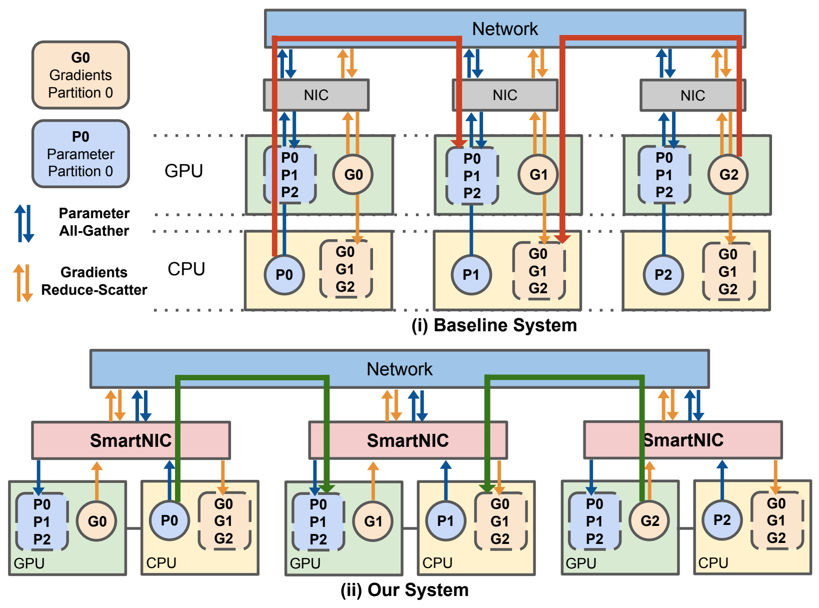
Figure 1: GPU memory wall. The memory requirements of ML models have increased exponentially, significantly outpacing the growth of GPU memories.

throughput of GPU clusters [26], the cost of the technology is high. All these reasons combined make such GPU clusters impractical and, for individuals and small companies, potentially out of reach.

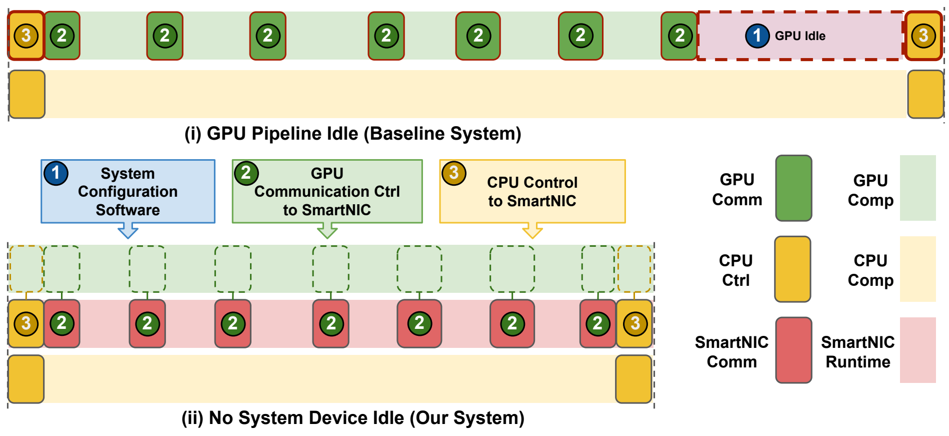
To reduce the required scale of GPU clusters and mitigate the GPU memory wall, heterogeneous training of deep learning has emerged. These systems leverage CPUs and NVMe memory for offloading model states and parameters. Building on the foundation of Zero Redundancy Optimizer (ZeRO) [39], ZeRO-Offload [42] and ZeRO-Infinity [40] facilitate large model training on small-scale GPU clusters by offloading data and computations to CPUs using memory optimization techniques. These heterogeneous systems, however, suffer from low efficiency and modest performance. This is due mainly to two limitations:

(1) Limited System Data Exchange Efficiency (Figure 2a): The architecture of a heterogeneous GPU system, with the CPU serving as an offload engine, is illustrated in Figure 2a(i). The CPU stores model parameters, gradients, and optimizer states. The GPU serves as the primary computation device for forward and backward propagation. Because ZeRO partitions the model across devices, parameters residing on the CPU frequently need to be exchanged among multiple GPUs, creating potential bottlenecks. Each partition stage triggers additional data exchanges between heterogeneous system components, leading to communication barriers and reduced data exchange efficiency.

(2) Limited System Node Computation and Control Efficiency (Figure 2b): In heterogeneous GPU systems with offload engines, the *delayed parameter update* (DPU) technique [42] allows the CPU's data path for parameter updates to be decoupled from the GPUs' forward/backward propagation computation (Figure 4). However, because the CPU and GPU have different computational power and resource



(a) System Data Exchange Efficiency


3
2
1 GPU Idle
(i) GPU Pipeline Idle (Baseline System)
1 System Configuration Software
2 GPU Communication Ctrl to SmartNIC
3 CPU Control to SmartNIC
GPU Comm
GPU Comp
CPU Ctrl
CPU Comp
SmartNIC Comm
SmartNIC Runtime
(ii) No System Device Idle (Our System)

(b) System Node Computation and Control Efficiency

Figure 2: System Comparison. (a) Addition of SmartNICs as an intermediate layer. (b) SmartNICs provide dynamic scheduling and control.

needs, their execution latencies often mismatch, causing one to be idle while waiting for the other. Moreover, the GPU's communication operator requires significant memory, computation resources, and control overhead. Collectively, these factors contribute to the diminishing of the system node's computation and control efficiency.

Advanced network interface cards, known as SmartNICs, combine communication, control, and computation and are useful in numerous domains [3, 17, 22, 44, 58]. Given their capabilities and their network-facing placement within the node, SmartNICs are well-suited to address the system efficiency challenges just described. SmartNICs enhance standard NICs, which typically lack application-level control beyond basic networking functions, leaving system control to the CPU. This reliance increases overhead and inefficiencies in coordinating the CPU offload engine, GPU, and network. Simply adding SmartNICS, however, addresses only point-to-point communication latency. Currently, no distributed system design leverages SmartNICs to overcome the broader limitations in heterogeneous GPU systems.

We propose a SmartNIC-GPU-CPU (SGC) system for large machine learning models that, together with system configuration software, breaks the boundaries between the various

(i) Same GPU number, Same Model Size
→ **Higher Performance and Efficiency**

	GPU	GPU-CPUoffload	**SGC System**
Throughput	High	Low	**High**
Latency	Low	High	**Low**
Power Consumption	High	Medium	**Low**

(ii) Same Model Size
→ **Fewer GPU needed, High Performance and Efficiency**

	GPU	GPU-CPUoffload	**SGC System**
System Scale Required	Large	Small	**Small**
Performance	High	Low	**High**
Power Consumption	High	Medium	**Low**

(iii) Same GPU Number
→ **Larger Model Supported, With Comparable Performance**

	GPU	GPU-CPUoffload	**SGC System**
Model Size	Small	Large	**Large**
Performance	High	Low	**Comparable Good**

Table 1: System Goals

device types and aims to provide: (i) Higher system performance and efficiency with lower system cost and power consumption (Table 1i); (ii) High performance with fewer of GPUs (Table 1ii); and (iii) Support of larger models while retaining good performance. (Table 1iii).

Several design techniques and optimizations are proposed. **To improve system data exchange efficiency (Figure 2a(ii)):** (1) SmartNICs act as an intermediate layer that breaks the boundary between distributed heterogeneous components in the system and facilitates seamless connectivity between GPUs and CPU offload engines; (2) A SmartNIC prefetch mechanism that proactively initiates communication before parameters are needed by the GPU or CPU, compacting the pipeline with less data waiting and largely mitigating communication bottlenecks via overlap of computation and communication; and (3) A SmartNIC buffering technique that stores duplicated communication data, reducing overall system communication workload. **To improve system device computation and control efficiency (Figure 2b(ii)):** (1) SmartNICs provide dynamic scheduling and control, allowing both GPUs and CPUs to concentrate on computation with reduced interruptions and overhead; and (2) System configuration software that optimizes system and model settings for maximal efficiency (minimizing device idle time) given different system specifications.

To summarize, the key contributions of this work include:

- A heterogeneous system where the SmartNICs facilitate seamless integration of CPU, GPU, and communication; specifically, in training large machine learning models including LLMs and DLRMs;
- A SmartNIC design that supports dynamic workload and communication scheduling and control of the compute components;
- System configuration software that optimizes system and model setup based on resources and hardware specifications.

Results show that the SGC system benefits training throughput and scalability with an improvement of over 1.6× in throughput over the baseline (without SmartNIC) for a 100B parameters model.

2 Background and Motivation

2.1 Zero Redundancy Optimizer (ZeRO)

Zero Redundancy Optimizer (ZeRO) is a memory optimization strategy that eliminates memory redundancies across data parallel processes by partitioning model states. By introducing reasonable additional communications, these strategies can efficiently scale the model size in proportion to the number of devices. ZeRO distributes the training batch across multiple GPUs in a way similar to data parallel training. Instead of duplicating models, however, ZeRO partitions model states across all GPUs and uses communication collectives to gather parameters as needed during the various training phases. It offers a more general solution that does not require users to modify the model extensively, thus providing improved compute efficiency and scalability.

ZeRO operates in three stages corresponding to three model states. Figure 3 shows an example of the workflow with 3 nodes. In this example, the model is partitioned into three parts and distributed among three data parallel processes. Zero has 3 stages that represent how the model is partitioned among nodes. In ZeRO-1, the optimizer states are partitioned on top of data parallelism, with each process owning a partition of the entire optimizer. Consequently, each optimizer partition updates only the corresponding parameter partition, following an All-Gather operation to update all processes' parameters. ZeRO-2, the second stage, partitions both optimizer states and gradients. Each process owns a partition of the gradients, requiring a Gradients Gather operation to collect all computed gradients across processes. After gathering, the corresponding optimizer works on the parameter update for its partition. In the final stage, ZeRO-3, layer parameters are partitioned and owned by data parallel processes. Broadcast communication collectives are initiated by the parameter partition owner prior to each forward and backward pass to share parameters with other data parallel processes. This process repeats until the completion of all

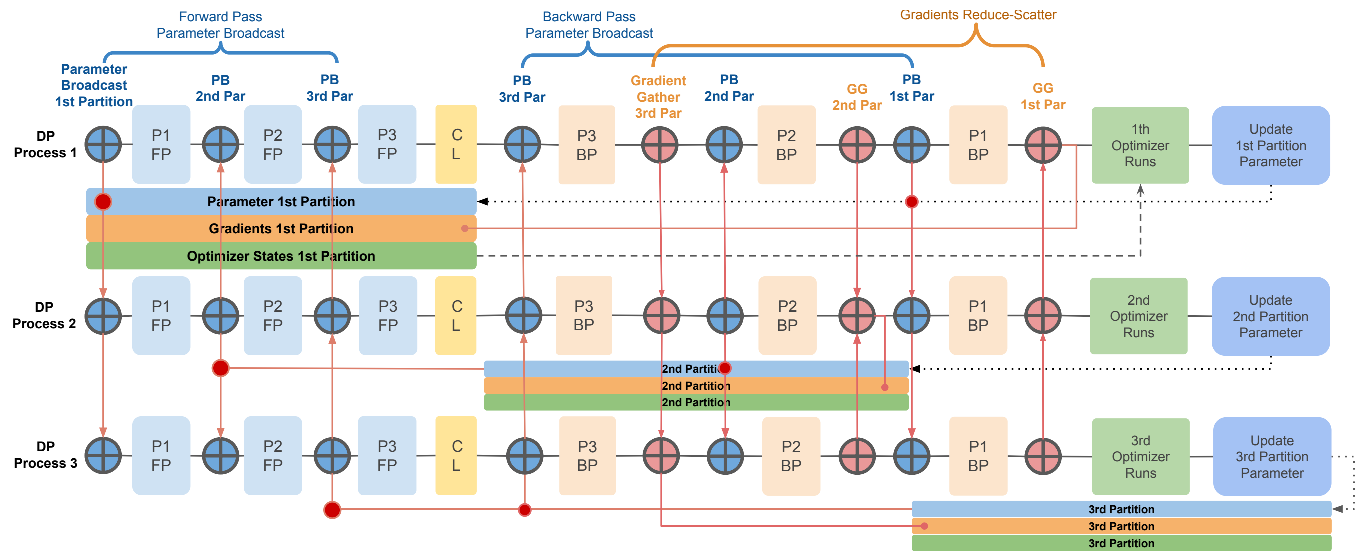


Figure 3: ZeRO Overview with No Bandwidth Aggregation for a three node system. P1FP: Forward Propagation of Partition 1. CL: Compute Loss. P3BP: Backward Propagation of Partition 3

forward pass operations. After each process completes its loss computation, a parameter Broadcast is issued before each backward pass partition.

Further work has been done with heterogeneous systems that leverage CPU or NVMe memory to augment the system's memory capacity. Such systems enable the training of significantly larger models on a limited number of GPUs. ZeRO-Offload [42] presents a heterogeneous training approach with the CPU as an offload engine; it is based on a ZeRO-2 foundation to offload optimizer states, gradients, and the computation of parameter updates on the CPU. Figure 5 illustrates the ZeRO-Offload workflow. This method addresses the constraints of limited GPU memory, alleviating the challenge of requiring a large number of GPUs to store optimizer states and gradients.

In ZeRO-Infinity [40], parameters are partitioned across all data parallel processes, using All-Gather instead of Broadcast to gather parameters for each process. The aggregated PCIe bandwidth across all nodes transfers its partition of parameters from the CPU to the GPU in parallel for the All-Gather operation. This aggregated bandwidth effectively mitigates the PCIe bottleneck, addressing the challenges associated with having a single PCIe link.

2.2 Delayed Parameter Update

The computation on the offload engine for parameter updates follows the forward and backward propagation on the GPU. However, this offload engine computation can potentially become a bottleneck during training. To address this issue and to hide the offload engine computation, ZeRO-Offload

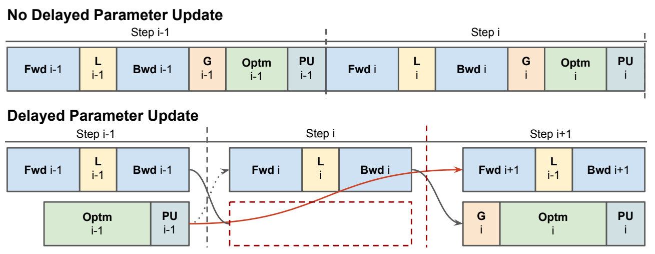


Figure 4: Delayed Parameter Update (DPU). (PU: Parameter Update, G: Gradients, L: Compute Loss)

employs a one-step delayed parameter update (DPU), as shown in Figure 4.

With no DPU, after the backward pass, gradients are generated, and the optimizer updates the parameters for the next iteration. The computation of the optimizer states depends on the output from the backward pass. With the introduction of DPU, the pipeline is divided into two independent data paths; optimizer state computation and parameter updates no longer depend on forward and backward propagation. At step $i + 1$, parameter updates use gradients from step i, while the forward and backward passes use parameters updated at step $i - 1$. From this step onward, the model at the $(i + 1)$th step is trained using parameters updated at the $(i - 1)$th step. The two data paths run in parallel with no dependency.

2.3 Limitations of the Existing Systems

Although existing heterogeneous systems that use CPU offload engines can enable large-scale model training, they have certain performance and efficiency constraints.

Limited system data exchange efficiency. During forward and backward pass computations, parameter data exchanges are required at each node to collect partitioned parameters from all the other CPUs. The parameter data exchanges involve the transfer from the CPU offload engine of each node to the GPU of every other node, with the GPU managing the control of its CPU offload engine, the data copy, and the initiation of communication.

After each partial backward propagation, gradients are produced in partitions for each process. A Reduce-Scatter distributes the gradients to their destination data parallel processes from each GPU to every other node's CPU offload engine. The data movement occurs from each node's GPU to every other node's CPU, with the GPU handling the Reduce-Scatter operator and data copy to the CPU offload engine. Even though the GPU does not participate in the parameter update computation using generated gradients, it still needs to initiate communication. Also, data flows through the GPU, which forwards it to the offload engine.

As seen in Figure 2a(i), such barriers between the system's heterogeneous components limit the system data exchange efficiency in two cases: (1) Data movement of parameter partitions from the offload engine to the GPU for forward and backward propagation computation. The movement starts from each node's CPU offload engine, passes through each node's GPU, and ends at every other node's GPU. (2) Data movement of generated gradients from GPU to CPU offload engine for parameter updates computation. The movement starts from each node's GPU, passes through every other node's GPU, and ends at every other node's CPU.

Limited system device computation and control efficiency. Delayed parameter update is used to maintain computational efficiency. This approach allows the CPU offload engine's parameter update computation data path to be detached from the GPU's data path, as illustrated in the upper part of Figure 6. Although these two data paths are independent computation paths, the CPU data path is still under the GPU's control. This control dependency reduces computational efficiency and increases overhead on both sides.

Also, variations in computational power and complexity between the GPU and CPU computation paths, along with different model configurations, batch sizes, and hardware specifications, can result in divergent execution latencies and lengthy idle times. In certain system and model configurations, the GPU datapath may finish faster than the CPU, leading to the training performance being consistently bottlenecked by the CPU, regardless of the GPU's power and the utilization of optimization techniques. This idle device time becomes a bottleneck, limiting system performance. Additionally, the GPUs initiate communication operators

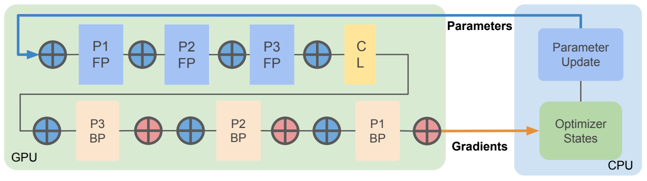


Figure 5: ZeRO Offload with CPU as Offload Engine.

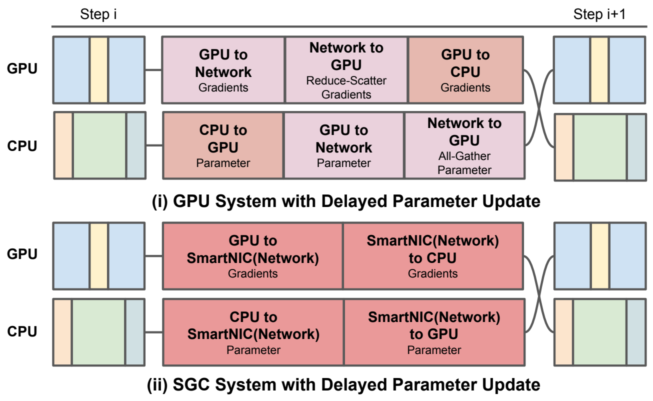

Step i
Step i+1
GPU
CPU
GPU to Network Gradients
Network to GPU Reduce-Scatter Gradients
GPU to CPU Gradients
CPU to GPU Parameter
GPU to Network Parameter
Network to GPU All-Gather Parameter
(i) GPU System with Delayed Parameter Update
GPU to SmartNIC(Network) Gradients
SmartNIC(Network) to CPU Gradients
CPU to SmartNIC(Network) Parameter
SmartNIC(Network) to GPU Parameter
(ii) SGC System with Delayed Parameter Update

Figure 6: System Comparison with DPU

for data exchange, necessitating memory allocation for exchanged data and computation resources for data management. Together, these factors diminish system computation and control efficiency.

3 SmartNIC-GPU-CPU System

This section gives an overview of the SmartNIC-GPU-CPU (SGC) heterogeneous system.

3.1 Breaking System Boundaries

During the training of large machine learning models using the ZeRO-Offload strategy, there is a substantial and frequent exchange of data among system components. In a GPU system without SmartNICs, all data exchanges must be initiated and facilitated by GPUs as illustrated in the upper part of Figure 6, when the current CPU sends offloaded parameters to remote GPUs (for the next iteration's forward and backward propagations), the offloaded parameters need to be retrieved by the current GPU that initiates the communication process to deliver the data to the other GPUs. A similar scenario occurs when gradients are generated by the current GPU and transmitted to other CPUs for parameter updates. The gradients are sent by the current GPU through the network, received by the destination GPU, and passed to the CPU. Such cross-device routing, combined with the limited interface between heterogeneous components, significantly reduces data exchange efficiency.

By contrast, SGC introduces SmartNICs as an intermediate layer to break the boundaries between GPUs and CPUs (Figure 2aii). Our model is very similar to ZeRO-offload. However, our communication from the CPU is no longer dependent on the GPU and can be efficiently handled by the SmartNIC. This decoupling allows the communication sender and receiver to be different types of devices, with the SmartNIC managing the communication process. Moreover, with the implementation of delayed parameter update (shown in Figure 4), the GPU and CPU data paths no longer depend on each other. Offloading control tasks from the GPU to the SmartNIC not only boost the computational efficiency of both devices but also reduces overhead and hardware utilization. As illustrated in the bottom part of Figure 6, GPUs and CPUs can initiate data transfers through the SmartNIC, further compacting the overall system pipeline by reducing stages and overhead.

3.1.1 Data exchange between system components. As illustrated in Figure 7, parameter gathering communication is required before each partition of forward propagation. The GPU fetches data from the CPU offload engine (FC) and initiates the communication operator. Once the communication is complete, the computation stream is ready for the first partition's forward propagation. In the SGC system, the GPU retrieves data from the SmartNIC instead of fetching it from the CPU. Upon receiving the fetch signal from the GPU, the SmartNIC initiates the corresponding communication operator, fetching data from the CPU and initiating the parameter gather communication. Once the destination node's SmartNIC receives the parameters, the data is sent directly to the GPU. Once all parameters are received, the Fetch SmartNIC stage on the GPU is complete and ready for the first partition's forward propagation.

Another parameter-gathering communication is required for each partition's backward propagation, following the same data exchange operations for backward propagation. After the backward propagation generates the gradients, the GPU forwards the gradients to the SmartNIC. After the Reduce-Scatter operator, the destination node's SmartNIC receives the results and sends them to the current CPU offload engine for the next iteration's parameter update.

3.1.2 Offload parameters for free. In a GPU system with a CPU offload engine, parameter offloading introduces overhead due to the additional PCIe data transactions. During the parameter gather communication operator, parameters are stored in CPU memory, and the GPU fetches the data through PCIe. Another PCIe transaction is then required to transfer the data from the GPU to the NIC, creating a dependency between the two PCIe data transfers. However, with the introduction of a SmartNIC, these two PCIe data exchanges can overlap. The SmartNIC fetches parameters from CPU memory simultaneously with the GPU fetch from the CPU. Even if the offload engine is not used, the parameter-gathering communication operator still involves a PCIe data exchange between the GPU and the NIC. With the support of a SmartNIC, the PCIe data exchange path becomes equivalent to a GPU system without an offload engine. This means that, with the assistance of a SmartNIC, we can achieve efficient parameter offloading essentially for free.

3.2 Collective Communication Support

Several collective communication operations are used during training, including parameter Broadcast, parameter All-Gather, and gradient Reduce-Scatter. In the system without SmartNIC, these operations are handled by the GPU, with data gathering and communication initiation. In the SGC architecture, we offload the collective communication operators to the SmartNIC.

Parameter broadcast and parameter All-Gather. We introduce the *Fetch Parameter* operator for GPU. When the GPU requires gathering parameters from the system, the GPU sends a *Fetch Parameter* signal to the SmartNIC with metadata indicating the current GPU stage and requesting data information. The SmartNIC fetches parameter data from the CPU offload engine and manages the parameter gather communication operator, distributing partitioned parameters to other system nodes. Once the SmartNIC receives all the gathered parameters, the result is forwarded to the GPU.

Gradient reduce-scatter. After backward propagation, gradients are generated and require a Reduce-Scatter operator to collect the full gradients for each partition. As the GPU generates gradients, they are sent to the SmartNIC to handle the reduce-scatter operator. The SmartNICs send and receive gradients with a gradient reduction computation. The gradient reduction computation kernel computes the final gradients, forwarding them to the local CPU in the *Write CPU* (WC) stage. Once the CPU receives all the gradients, it possesses all the necessary data for the subsequent iteration's parameter update.

4 SGC Optimizations

In this section, we present optimization techniques that increase the SGC system efficiency.

4.1 SmartNIC Prefetch

In the previous section, we discussed how the GPU fetches from the SmartNIC instead of the CPU offload engine during parameter gathering. Each node's SmartNIC fetches partitioned parameters from the CPU and initiates the parameter gather communication operator. Consequently, the SmartNIC possesses awareness of both the application and system status. Operating as a network-facing device, the SmartNIC knows about the completion status of the ongoing communication operator. *SmartNIC prefetch* enables the SmartNIC

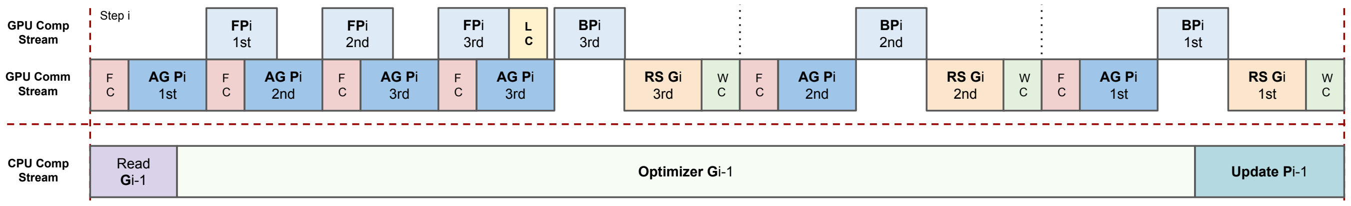


Figure 7: GPU with CPU offload engine system pipeline with three partitions. (FC: Fetch from CPU. WC: Write to CPU. AG: All-Gather. RS: Reduce scatter. FP: Forward propagation. LC: Computing loss. BP: Backpropagation.)

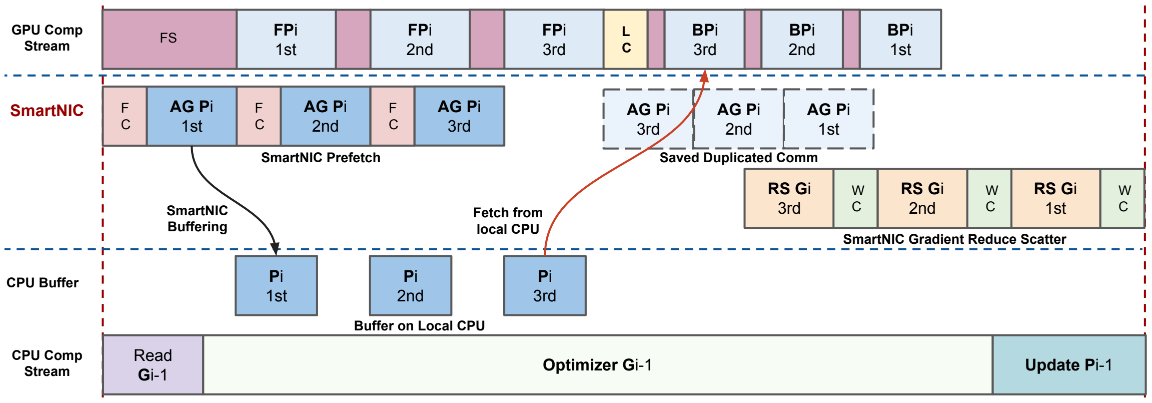


Figure 8: SGC Heterogeneous System Pipeline. (FS: Fetch from SmartNIC. FC: Fetch from CPU. WC: Write to CPU. AG: All-Gather. RS: Reduce scatter. FP: Forward propagation. LC: Computing loss. BP: Backpropagation.)

to pre-initiate the parameter gather communication immediately after the completion of the preceding communication.

The SmartNIC control pipeline aligns with the GPU's datapath, improving communication and computation overlap. As shown in Figure 8, the GPU pipeline initiates during forward propagation with the *Fetch SmartNIC* (FS) signal, signifying the initiation of a new GPU stage. This triggers the SmartNIC to execute the *Fetch CPU* (FC) operation, fetching the local CPU's offloaded parameter partition. With the acquired data, the *All-Gather* communication operator is activated to gather parameters across system nodes. Note that in this example, we employ bandwidth aggregation, utilizing All-Gather instead of Broadcast to gather the parameters.

The SmartNIC pipeline keeps track of the current stage, ensuring seamless progression. On completion of the All-Gather, the SmartNIC initiates the gather for the second partition's parameters. This SmartNIC capability of parameter prefetching eliminates the need to wait for the GPU to complete the forward propagation of the first partition before initiating the All-Gather for the second partition.

Once the prefetched parameters have been gathered, they are temporarily buffered on the SmartNIC, waiting for the GPU to initiate the FS stage signal for the second partition. Subsequently, the data is forwarded to the GPU for the next stage of the forward pass. As the data is transmitted to the GPU, the SmartNIC can execute the next pipeline stage, prefetching the parameters for the next partition. This scenario assumes that the forward propagation stage is longer than the communication stage, allowing the SmartNIC to ensure that the GPU has initiated the next stage and received all data before proceeding to the subsequent SmartNIC pipeline stage. However, there may also be cases where the communication stage takes longer than the GPU computation stage. In such instances, the GPU is ready for the next partition's forward propagation computation, but the gathered data is not yet available. Consequently, the GPU will remain in the FS stage until the SmartNIC has all the necessary data prepared for forwarding.

During backward propagation, a duplicated parameter gathering operation is necessary before initiating backward propagation (Figure 3). This is because, to conserve GPU memory, the gathered parameters are deleted after the completion of forward propagation. Consequently, an additional duplicated parameter gather operation is essential for each backward propagation step. Following the SmartNIC pipeline, the parameter gathering for backward propagation is started once the SmartNIC concludes the parameter gathering communication for the last partition. The SmartNIC will wait until the GPU fetches the parameters.

4.2 SmartNIC Buffering Technique

As shown in Figure 3, forward and backward propagation involve two duplicate parameter-gathering processes. Due to GPU memory management considerations, the gathered parameters are deleted after each partition's forward propagation computation. Consequently, another duplicate parameter gathered is essential for conducting backward propagation. With increased model size and system scale, communication constitutes a significant portion of the latency. This duplication in communication may impede the system's FLOPs throughput, with GPUs waiting for data to arrive. Even when using the SmartNIC design, where parameter All-Gather and gradients Reduce-Scatter overlap, parameter gathering still consumes a substantial share of the network bandwidth, potentially compromising the performance of both communication operators. To address this redundancy, we introduce the SmartNIC buffering technique.

As illustrated in Figure 8, in the forward propagation phase, the All-Gather parameter operator is completed by the SmartNIC, which retains the first partition of the gathered parameters. The SmartNIC then concurrently forwards these parameters to the GPU for the first partition's forward propagation and to the CPU's allocated memory to temporarily store the first partition parameters. This process is repeated until all partition forward propagations are completed. When the GPU starts backward propagation, the FS stage signals the SmartNIC to request the required data for backward pass computation. As the parameters are already buffered locally on the CPU, the SmartNIC signals the CPU to transfer the third partition's parameter data from CPU memory to the corresponding GPU memory. This process continues until the backward propagations are all complete. Upon the CPU sending the parameters to the GPU, the parameter copy is deleted for reuse in the next iteration. This approach avoids initiating a new round of communication during backward propagation, allowing the GPU to fetch parameter data from the local CPU directly.

5 System Configuration Software

5.1 System Metrics

The large-scale heterogeneous system consists of hardware components with varying capabilities and power; the connecting technologies between these components also differ. Numerous system configurations may present diverse bottlenecks that restrict overall performance based on different machine-learning model setups.

To address this, we propose a system configuration software. This software, tailored to different system configurations, identifies potential bottlenecks in system components or connections. It then suggests system and model configuration adjustments to minimize bottlenecks and enhance overall system efficiency. The software utilizes several metrics to evaluate system performance, including:

(i) Model Size, (ii) Batch Size, (iii) Data Parallel Process Number (Model Partition Number), (iv) GPU Computation Power, (v) CPU Computation Power, (vi) Heterogeneous Components Connection Configuration.

5.2 Configuration Metrics

Computation Datapath. There is a computational power and complexity disparity between the GPU and CPU data paths in heterogeneous training systems. This misaligned factor can lead to idle periods on the GPU or CPU, wasting computational resources and reducing efficiency. The GPU computation complexity is $O(B\Psi)$, where B is batch size and Ψ is model parameters. The CPU complexity is $O(\Psi)/P$, where P is the number of model partitions. The GPU workload increases as the batch size grows, but CPU computation decreases proportionally with more partitions. In a configuration with a small batch size and few partitions, the CPU path can have much higher latency, causing the GPU to idle as the next step cannot start until both paths finish. This leads to a wastage of valuable GPU computational power. Configuration software can identify such bottlenecks, offering recommended configuration thresholds based on various metrics to eliminate GPU idle time.

Data exchange bandwidth trade-off. Our current system configuration assumes a PCIe bandwidth larger than the network bandwidth. However, design choices and trade-offs arise when employing advanced network configurations or different system device connections. In the previous section, we introduced the SmartNIC buffer technique to buffer parameters that require duplicated parameter gather communication. These buffered parameters reside on the local CPU, and when the GPU requires them, they are transferred via PCIe. This approach proves advantageous when PCIe bandwidth surpasses the network bandwidth.

In scenarios where the system utilizes advanced network connections, such as NVLink, offering a greater bandwidth than the PCIe setup, the SmartNIC buffer technique may impact system performance and become a bottleneck. Fetching the same amount of data from the local CPU could incur higher latency than initiating the network communication parameter All-Gather. Therefore, the configuration software can adjust the pipeline to swap the SmartNIC buffer with the parameter gather communication operation.

Advancements in device connection technologies introduce high-bandwidth connections like NVLink [33] for data exchange between heterogeneous components such as GPU and CPU. This benefits the SmartNIC buffer technique, ensuring that data exchange between GPU and CPU does not compromise overall system performance. The SmartNIC buffer also saves network bandwidth during backward propagation when the parameter All-Gather and gradients Reduce-Scatter occur simultaneously. This technique removes parameter gathering communication by utilizing local GPU-CPU data transactions, thereby preserving more network bandwidth for gradients Reduce-Scatter.

System scale and offload engine. The partition number could largely affect the system performance, depending on the given training model size. With a small data parallel process number, the size of offloaded partitioned model states increases, stored on the offload engine. This results in a larger volume of PCIe data exchange between the offload engine and the communication device during the parameter gathering phase, involving fetching partitioned parameters from the CPU. The potential bottleneck in the communication phase arises from the latency in PCIe data exchange. However, as mentioned earlier, advanced device connection techniques between the GPU and CPU can enhance the data exchange efficiency between the GPU and offload engine. Therefore, a small system scale with more parameters offloaded onto the offload engine can still provide comparable performance.

6 Evaluation

This section evaluates the SGC system and demonstrates that it achieves excellent training efficiency and scalability for large-scale machine learning model training.

6.1 Experimental Methodology and Setup

The baseline GPU system is built on PyTorch FSDP with CPU offload engines (ZeRO-Offload), utilizing ZeRO as the data parallel strategy [61]. The large transformer-based model follows Meta LLaMA [51]. The recommendation model is based on the Deep Learning Recommendation Model (DLRM) [31].

A standard problem in computer architecture research, especially with respect to large-scale systems, is the limited access to the target systems under study, in this case, heterogeneous CPU-GPU-SmartNIC systems with at least hundreds of nodes. We ensure evaluation accuracy by integrating real-world performance benchmarked parameters obtained from commercial CPUs, GPUs, and FPGAs- the latter for prototyping SmartNIC devices- into a cycle-accurate simulator.

Our experimental methodology consists of the following steps. First, we profile the ML modeling workflow with the workload of interest. We measure the timing for each model phase using a GPU cluster running PyTorch FSDP with open-source models. The large transformer-based model is based on Meta LLaMA [29]; the recommendation model follows DLRM [6]. Next, we decompose the heterogeneous SGC system into its homogeneous components—-CPU, GPU, SmartNIC, and network—-analyzing their respective tasks and workloads. We benchmark each component using real-world hardware under various workload sizes to derive, for each type of device, its roofline model and latency distribution. For CPUs we use the Intel(R) Xeon(R) Gold 6226R CPU; for GPUs the NVIDIA Tesla V100 Tensor Core GPU. For SmartNIC prototyping, we use the AMD Alveo U280 FPGA, configured via High-Level Synthesis (Vitis HLS). The FPGA is configured as a SmartNIC utilizing the XUP Vitis Network Example (VNx) as the network kernel[57]. To benchmark network performance, we utilize the Open Cloud Testbed (OCT) [34], which consists of 24 AMD Alveo U280 FPGAs connected via 100 GbE data center switches using QSFP28 passive DAC cables. We execute collective communication benchmarks across various data sizes and communication patterns. The SmartNIC design is implemented using Vitis HLS, leveraging the user logic integrated within XUP.

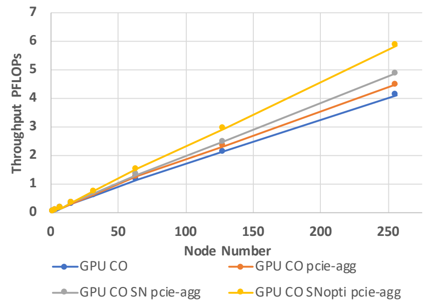
Finally, after obtaining real-world performance parameters, we integrate these hardware roofline model benchmarks (GPU, CPU, smartNIC, network) into our cycle-accurate simulator. This simulator models large-scale systems by incorporating ML model phases, workload data sizes, and computation dependency flows. Each system component is connected via FIFOs, simulating bandwidth, latency, and contention of hardware connections. These include bidirectional 32 GBps PCIe links, GPU kernel launch overhead, software runtime overhead, microarchitecture-level cycle latency on smartNICs, contention for hardware resources and the network contention, collective communication packet chunking, and bandwidth and latency for collectives.

To validate the simulator, we conducted experiments with real heterogeneous (CPU-GPU-FPGA) clusters, ranging in size from one to eight nodes. Workloads were validated at multiple levels from single kernel to complete application. This ensures an accurate correlation between the simulator and real-world cluster performance.

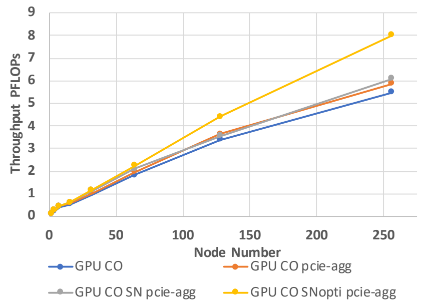
The GPU system with CPU offload (ordinary NICs) serves as the baseline. The evaluation figures use the following abbreviations:
pcie-agg = PCIe bandwidth aggregation enabled
CO = CPU offloading enabled
SN = SmartNIC enabled
SNopti = SmartNIC enabled with optimization techniques.

6.2 Performance

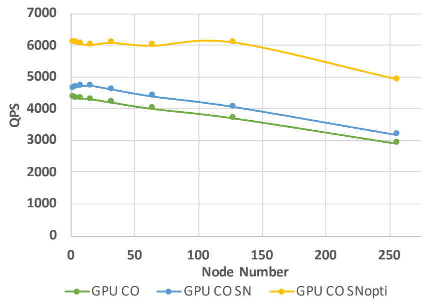
6.2.1 Performance with same system size. Figure 10 illustrates the speedup in training latency for a 10 billion parameter transformer-based model. When the system size is less than 16 nodes, the model is too large to fit into the



(a) 10B Model Throughput (PFLOPs)



(b) 100B Model Throughput (PFLOPs)



(c) DLRM Kaggle Sample/GPU/Second (QPS)

Figure 9: Training Throughput

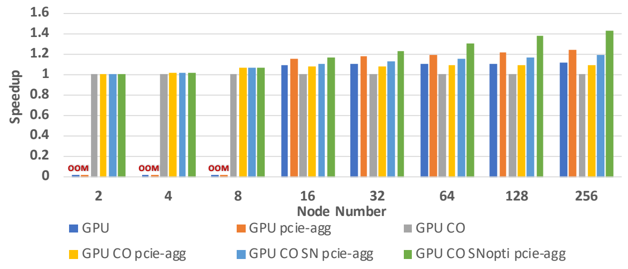


Figure 10: 10B Model Training Latency Speedup

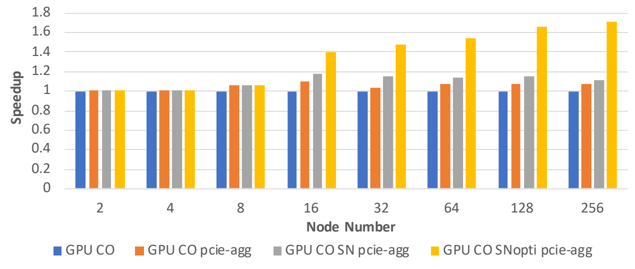


Figure 11: 100B Model Training Latency Speedup

GPU-only system, as indicated by the *OOM* label. At this point, the performance across systems is nearly identical because the CPU computation data path becomes the bottleneck. With a smaller system size, each CPU data path handles a heavier workload. As the system scales beyond 16 nodes, the GPU-only system shows improved training latency compared to the GPU system with an offload engine. The latter experiences overhead from PCIe data exchange and inefficiencies due to the boundary between GPUs and offload engines. By leveraging PCIe bandwidth aggregation, the speedup increases as the system utilizes every node's PCIe bandwidth instead of just one. Also, the parameter All-Gather communication operator used by pcie-agg requires less network bandwidth than the parameter Broadcast.

The SGC system, incorporating SmartNIC optimization, outperforms both the baseline and the GPU-based system. This improvement becomes more pronounced as the system scales up to 256 nodes, achieving a 1.4 × speedup over the baseline. This improvement is primarily due to higher SGC system efficiency. The SmartNIC prefetch optimizes system pipelines by enhancing computation and communication overlap. Additionally, the SmartNIC buffer reduces duplicated communication through local PCIe data exchange, thereby minimizing GPU data waiting time.

Figure 11 presents the training latency speedup for a model with 100 billion parameters. With a system size of fewer than 16 nodes, the CPU computation data path constrains the system's performance. However, as the system scales from 16 to 256 nodes, the SGC system achieves a notable speedup over the baseline, reaching a 1.6 × speedup. With larger model sizes, communication and PCIe latency become significant factors in overall training latency. The SGC system improves the overlapping of communication and computation stages, contributing to its enhanced performance.

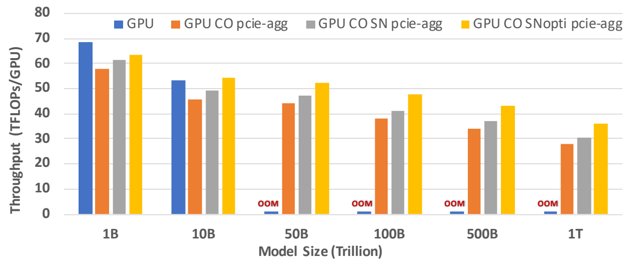


Figure 12: 16 Node System Training Throughput

6.2.2 *High Performance with Fewer GPUs.*

Figure 12 illustrates the system's support for various model sizes with 16 nodes. The GPU-based system is constrained to fitting the model sizes of less than 10 billion parameters. In contrast, the SGC system demonstrates the capability to train models with up to 1 trillion parameters using the same 16-node system. Achieving the same capability with a GPU-only system

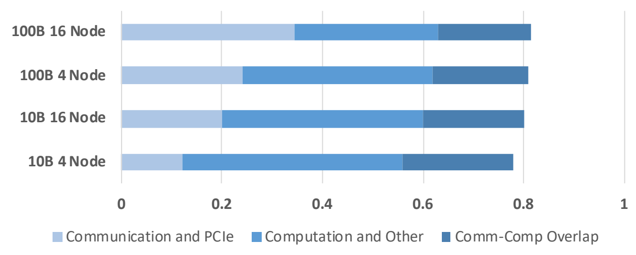


Figure 13: GPU with Offload Engine Latency Breakdown

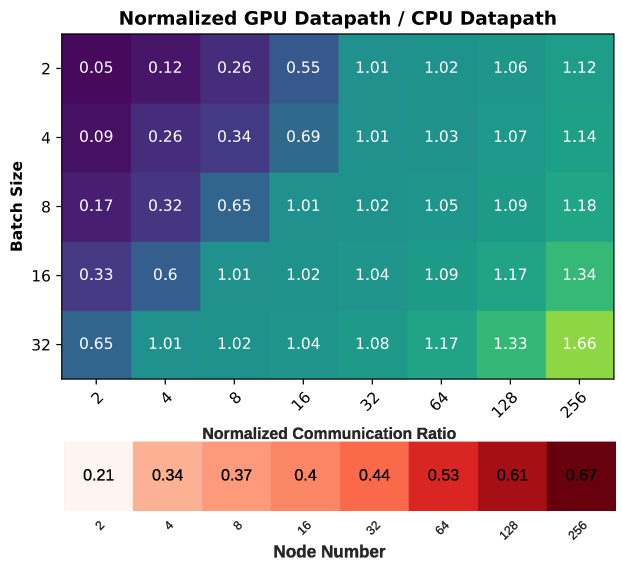
would necessitate 320 GPUs just to fit the model for training. With increasing model sizes, the SGC system achieves higher training throughput over the baseline, and the GPU-only system requires more GPUs to fit the large models.

6.2.3 Support larger model Size with comparable performance. Figure 12 indicates that an SGC system with 16 nodes could support model sizes up to trillions of parameters. As the model size increases, the SGC system achieves higher training throughput over the baseline. Also, the SGC system can train large models with throughput almost equivalent to that of the GPU-only system. The 16-node GPU-only system could train up to 10 10-billion size models, but the 16-node SGC system achieves comparable throughput with a training model size of 100 billion. This improvement is largely due to the more efficient pipeline stage overlap and reduced communication latency.

6.2.4 System Scalability. Figure 13 shows the breakdown in training latency for the GPU with a CPU offload engine system. The PyTorch FSDP supports GPU data prefetching, indicated by *comm-comp overlap*. For both 10 billion and 100 billion models, as the system scales from 4 nodes to 16 nodes, there is an increased latency fraction due to communication and PCIe data exchange. With larger model sizes, data exchange constitutes a larger portion of the overall execution time, making it challenging to achieve efficient overlap between computation and communication.

Figure 9 presents the evaluation of system scalability. The SGC system exhibits better scalability for both 10-billion and 100-billion model sizes. As the model size increases, SGC maintains nearly linear training throughput even as the system scales up. Notably, for the 100 billion model, the speedup is even more pronounced, given that communication latency constitutes a substantial portion of the overall training latency. This aligns with the latency breakdown, highlighting that SGC achieves better overlapping of communication with computation and so minimizes system overhead.

For DLRM, the all-to-all communication is used by the embedding layer, which is the system performance bottleneck. The SGC system maintains the training throughput as the system scales up and displays a higher throughput than the baseline system.



Batch Size \ Node Number	2	4	8	16	32	64	128	256
2	0.05	0.12	0.26	0.55	1.01	1.02	1.06	1.12
4	0.09	0.26	0.34	0.69	1.01	1.03	1.07	1.14
8	0.17	0.32	0.65	1.01	1.02	1.05	1.09	1.18
16	0.33	0.6	1.01	1.02	1.04	1.09	1.17	1.34
32	0.65	1.01	1.02	1.04	1.08	1.17	1.33	1.66

Node Number	2	4	8	16	32	64	128	256
Normalized Communication Ratio	0.21	0.34	0.37	0.4	0.44	0.53	0.61	0.67

Figure 14: System Configuration Software with 10B Model

6.3 System Software Configuration

As shown in the previous sections, When the CPU computation path constrains the system's performance, the benefits of the SGC system may be limited, as the GPU data path cannot start the next training iteration until the CPU is completed. The upper part of Figure 14 illustrates the normalized GPU data path latency over CPU data path latency. A value less than 1 indicates that the CPU data path dominates. Both system and batch size influence this ratio. As the system scales up, the CPU data path has a reduced workload. Increasing the batch size results in a higher computation workload for the GPU data path. To avoid a CPU computation bottleneck, we select a system computation configuration with a ratio greater than 1, allowing the SGC system to provide more benefits for training performance.

Despite the system's performance being less constrained by the CPU data path, as we configure the system on a larger scale, communication latency increases as additional nodes are incorporated into the system. The bottom part of Figure 14 illustrates the ratio of communication latency (including the overlapped communication latency) to overall latency. By combining these two figures, the configuration software can recommend configurations based on system and model metrics to achieve higher efficiency and improved performance.

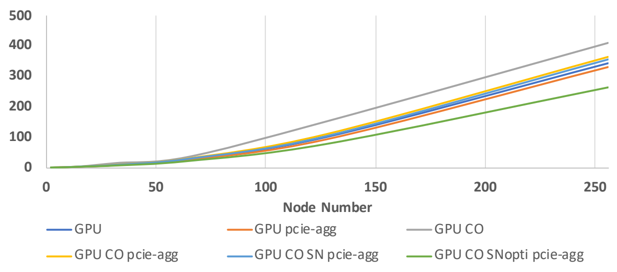


Figure 15: Normalized Power Consumption

6.4 Power Consumption

Figure 15 compares the normalized power consumption across various systems. As the system scales, the SGC system exhibits lower power consumption. The SmartNIC optimization in SGC contributes to power savings, particularly due to the SmartNIC buffering technique, which replaces network communication with local PCIe data fetch, thereby reducing network communication workloads. This advantage becomes more pronounced as the system scales up, as larger sizes increase energy consumption for network data exchange. Also, the SmartNIC prefetch helps compact the system pipeline by reducing device data waiting time and system bubbles, further contributing to power efficiency.

The SGC system introduces higher processing power onto the smartNIC, potentially increasing its power consumption and cost. However, as the SGC system improves system efficiency and reduces overall ML workload latency, this design choice could in fact lower total power consumption. Also, the current focus for the SGC system is on ML training and inference without sharing with other tasks. With more flexibility come interesting issues with network resource contention.

7 Related Work

Parallel strategies are used to train large models at scale [15, 19, 25, 46–48, 54]. To scale up model training, work in [4] saves memory from activation by recomputing from saved checkpoints. Mixed precision [30] has been proposed to compress models to reduce memory requirements. CPU memory-based training approaches [7, 16, 18, 23, 37, 41, 43, 53], such as L2L [38], enable multi-billion parameter training by managing memory usage in the GPU layer by layer. ZeRO-Offload [42] and ZeRO-Infinity [40] are state-of-the-art for large model training based on the ZeRO [39] parallel training strategy. The PyTorch FSDP [61] advanced DDP [28] model wrapper enables training of large models using PyTorch [36]

SmartNICs have evolved with various powerful computing resources. Nvidia has introduced DPUs [32] for data center AI and networking workloads. Broadcom provides the Stingray SmartNIC [2] with an 8-core ARM CPU and P4 packet processing engine. Many commercial FPGA-based SmartNICs have been released [1, 45]. For example, Xilinx has introduced the Alveo U25 [55] SN1000 [56] SmartNICs with FPGA programmable logic and ARM core. Intel released FPGA-based Intel Infrastructure Processing Unit (Intel IPU) [20] and Intel FPGA SmartNIC [21]. Commercial and research systems have deployed SmartNICs as computation resources to offload applications [3, 9–11, 17, 22, 44, 59] and system functions [8, 27, 60]. Related approaches to computing in the network augment the switch [5, 12–14, 49].

8 Conclusion

We propose a SmartNIC-GPU-CPU heterogeneous system for large machine learning models training with software-hardware codesign. The SmartNICs act as an intermediate layer that seamlessly connects the GPU and CPU. A set of optimization techniques compacts the system pipeline with less data waiting and largely overlapping communication and computation. Additionally, system configuration software optimizes the system and model settings with maximal system efficiency. Evaluation results show that the SGC system achieves better training throughput and scalability over the baseline system.

While our proposed system demonstrates considerable gains in training large models using a modest GPU cluster, it is primarily tailored to settings where GPU resources are limited (e.g., 8–128 nodes). In these scenarios, the SmartNIC-based design effectively addresses the memory bottleneck and communication overhead that arise in smaller-scale deployments. However, this approach is not intended to compete with large-scale GPU clusters connected by high-speed interconnects such as NVLink. Consequently, the primary contribution of our work is to enable data scientists and organizations with limited GPU resources to train models that exceed the capacity of GPU high-bandwidth memory, rather than to provide a one-size-fits-all solution for ultra-large-scale high-performance clusters.

Acknowledgments

We are most grateful to the anonymous reviewers for their help in guiding the revision of the manuscript. This work was supported, in part, by the National Science Foundation through awards CCF-1919130, CCF-2151021, and CCF-2326494; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, in support of the MEERCAT Microelectronics Science Research Center, under Contract DE-AC05-76RL01830; and by AMD and Intel both through donated FPGAs, tools, and IP.

References

[1] C. Bobda, J. Mandebi, P. Chow, M. Ewais, N. Tarafdar, J.C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser, M.C. Herbordt, H. Shahzad, P. Hofstee, B. Ringlein, J. Szefer, A. Sanaullah, and R. Tessier. 2022. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM*

Transactions on Reconfigurable Technology and Systems 15, 3 (2022), 1–42. doi:10.1145/3506713

[2] Broadcom. 2019. Stingray PS250 2x50-Gb High-Performance Data Center SmartNIC. https://docs.broadcom.com/doc/PS250-PB.

[3] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 7, 13 pages. doi:10.1109/MICRO.2016.7783710

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[5] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 1–16. doi:10.1145/3458817.3476178

[6] Facebook Research. 2019. DLRM: Deep Learning Recommendation Model. https://github.com/facebookresearch/dlrm

[7] Jiarui Fang, Zilin Zhu, Shenggui Li, Hui Su, Yang Yu, Jie Zhou, and Yang You. 2023. Parallel Training of Pre-Trained Models via Chunk-Based Dynamic Memory Management. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (Jan. 2023), 304–315. doi:10.1109/tpds.2022.3219819

[8] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[9] Anqi Guo, Tong Geng, Yongan Zhang, Pouya Haghi, Chunshu Wu, Cheng Tan, Yingyan Lin, Ang Li, and Martin Herbordt. 2022. FCsN: A FPGA-Centric SmartNIC Framework for Neural Networks. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Press, 1–2. doi:10.1109/FCCM53951.2022.9786193

[10] Anqi Guo, Tong Geng, Yongan Zhang, Pouya Haghi, Chunshu Wu, Cheng Tan, Yingyan Lin, Ang Li, and Martin Herbordt. 2022. A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 01–08. doi:10.1109/FPL57034.2022.00071

[11] Anqi Guo, Yuchen Hao, Chunshu Wu, Pouya Haghi, Zhenyu Pan, Min Si, Dingwen Tao, Ang Li, Martin Herbordt, and Tong Geng. 2023. Software-Hardware Co-design of Heterogeneous SmartNIC System for Recommendation Models Inference and Training. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 336–347. doi:10.1145/3577193.3593724

[12] Pouya Haghi, Anqi Guo, Qingqing Xiong, Chen Yang, Tong Geng, Justin T. Broaddus, Ryan Marshall, Derek Schafer, Anthony Skjellum, and Martin C. Herbordt. 2022. Reconfigurable switches for high performance and flexible MPI collectives. *Concurrency and Computation: Practice and Experience* 34, 6 (2022), e6769. doi:10.1002/cpe.6769 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6769

[13] Pouya Haghi, William Krska, Cheng Tan, Tong Geng, Po Hao Chen, Connor Greenwood, Anqi Guo, Thomas Hines, Chunshu Wu, Ang Li, Anthony Skjellum, and Martin Herbordt. 2023. FLASH: FPGA-Accelerated Smart Switches with GCN Case Study. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 450–462. doi:10.1145/3577193.3593739

[14] Pouya Haghi, Cheng Tan, Anqi Guo, Chunshu Wu, Dongfang Liu, Ang Li, Anthony Skjellum, Tong Geng, and Martin Herbordt. 2024. SmartFuse: Reconfigurable Smart Switches to Accelerate Fused Collectives in HPC Applications. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 413–425. doi:10.1145/3650200.3656616

[15] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *ArXiv* (2018). https://api.semanticscholar.org/CorpusID:47016772

[16] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 875–890. doi:10.1145/3373376.3378465

[17] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. 2017. sPIN: High-performance streaming Processing In the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 59, 16 pages. doi:10.1145/3126908.3126970

[18] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. doi:10.1145/3373376.3378530

[19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 10, 10 pages. doi:10.5555/3454287.3454297

[20] Intel. 2021. Intel® Infrastructure Processing Unit (Intel® IPU). https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[21] Intel. 2022. Intel FPGA SmartNIC. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic/n6000-pl-platform.html.

[22] Ranjesh G. Jaganathan, Keith D. Underwood, and Ron Sass. 2003. A Configurable Network Protocol for Cluster Based Communications using Modular Hardware Primitives on an Intelligent NIC. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (Phoenix, AZ, USA) *(SC '03)*. Association for Computing Machinery, New York, NY, USA, 22. doi:10.1145/1048935.1050173

[23] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. 2018. Layer-Centric Memory Reuse and Data Migration for Extreme-Scale Deep Learning on Many-Core Architectures. *ACM Trans. Archit. Code Optim.* 15, 3, Article 37 (Sept. 2018), 26 pages. doi:10.1145/3243904

[24] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeff Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. https://api.semanticscholar.org/CorpusID:210861095

[25] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. 2023. Merak: An Efficient Distributed DNN Training Framework With Automated 3D Parallelism for Giant Foundation Models. *IEEE Transactions on Parallel and Distributed Systems* 34, 5 (2023), 1466–1478. doi:10.1109/TPDS.2023.3247001

[26] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. doi:10.1109/TPDS.2019.2928289

[27] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/2934872.2934897

[28] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018. doi:10.14778/3415478.3415530

[29] Meta. 2025. LLaMA Github. https://github.com/meta-llama/llama

[30] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=r1gs9JgRZ

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091 [cs.IR] https://arxiv.org/abs/1906.00091

[32] Nvidia. 2021. NVIDIA BLUEFIELD-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[33] NVIDIA. 2023. What Is NVLink. https://blogs.nvidia.com/blog/2023/03/06/what-is-nvidia-nvlink/.

[34] OCT-FPGA. 2025. OCT Tutorials. https://github.com/OCT-FPGA/OCT-Tutorials

[35] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, and et al. Igor Babuschkin. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] https://arxiv.org/abs/1912.01703

[37] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. doi:10.1145/3373376.3378505

[38] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. 2020. Training Large Neural Networks with Constant Memory using a New Execution Algorithm. arXiv:2002.05645 [cs.LG] https://arxiv.org/abs/2002.05645

[39] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages. doi:10.5555/3433701.3433727

[40] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 59, 14 pages. doi:10.1145/3458817.3476205

[41] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 598–611. doi:10.1109/HPCA51647.2021.00057

[42] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. arXiv:2101.06840 [cs.DC] https://arxiv.org/abs/2101.06840

[43] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 18, 13 pages. doi:10.5555/3195638.3195660

[44] Whit Schonbein, Ryan E. Grant, Matthew G. F. Dosanjh, and Dorian C. Arnold. 2019. INCA: In-Network Compute Assistance. *SC19: International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), 1–13. https://api.semanticscholar.org/CorpusID:207940021

[45] Hafsah Shahzad, Ahmed Sanaullah, and Martin Herbordt. 2021. Survey and Future Trends for FPGA Cloud Architectures. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–10. doi:10.1109/HPEC49654.2021.9622807

[46] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/3a37abdeefe1dab1b30f7c5c7e581b93-Paper.pdf

[47] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *ArXiv* abs/1909.08053 (2019). https://api.semanticscholar.org/CorpusID:202660670

[48] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Kim, Youngsok Kim, and Jinho Lee. 2023. Optimus-CC: Efficient

Large NLP Model Training with 3D Parallelism Aware Communication Compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 560–573. doi:10.1145/3575693.3575712

[49] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: a data plane architecture for per-packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1099–1114. doi:10.1145/3503222.3507726

[50] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, and et al. Dehao Chen. 2022. LaMDA: Language Models for Dialog Applications. arXiv:2201.08239 [cs.CL] https://arxiv.org/abs/2201.08239

[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[52] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, and Isabel Kloumann et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[53] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: dynamic GPU memory management for training deep neural networks. *SIGPLAN Not.* 53, 1 (Feb. 2018), 41–53. doi:10.1145/3200691.3178491

[54] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 17 pages. doi:10.1145/3302424.3303953

[55] Xilinx. 2020. Alveo U25 SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u25.html.

[56] Xilinx. 2021. ALVEO™ SN1000 SmartNICs. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/xilinx-alveo-sn1000-product-brief.pdf.

[57] Xilinx. 2025. XUP Vitis Network Example. https://github.com/Xilinx/xup_vitis_network_example Accessed: 2025-03-16.

[58] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. 2023. Unleashing SmartNIC Packet Processing Performance in P4. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 1028–1042. doi:10.1145/3603269.3604882

[59] Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Anthony Skjellum, and Martin C. Herbordt. 2019. GhostSZ: A Transparent FPGA-Accelerated Lossy Compression Framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 258–266. doi:10.1109/FCCM.2019.00042

[60] Qingqing Xiong, Anthony Skjellum, and Martin C. Herbordt. 2018. Accelerating MPI Message Matching through FPGA Offload. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 191–1914. doi:10.1109/FPL.2018.00039

[61] 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. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. arXiv:2304.11277 [cs.DC] https://arxiv.org/abs/2304.11277