2024

Software and hardware codesign of SmartNIC-based heterogeneous HPC clusters with machine learning case studies

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

SOFTWARE AND HARDWARE CODESIGN OF SMARTNIC-BASED HETEROGENEOUS HPC CLUSTERS WITH MACHINE LEARNING CASE STUDIES

by

ANQI GUO

B.S., Lanzhou University, 2018
M.S., Boston University, 2020

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2024

Approved by

First Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Second Reader

Roscoe Giles, PhD
Professor of Electrical and Computer Engineering

Third Reader

Tali Moreshet, PhD
Senior Lecturer and Research Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Tong Geng, PhD
Assistant Professor of Electrical and Computer Engineering and Computer Science
University of Rochester

When pride comes, then comes disgrace, but with the humble is wisdom.

Proverbs 11:12

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Martin Herbordt. Throughout my PhD journey, his boundless help and guidance have been invaluable. His inspiration and detailed understanding of my project kept me motivated, and his consideration of my personal life deeply touched me. I am honored and proud to have had him as my advisor.

I would also like to extend my thanks to my other committee members, Professor Roscoe Giles, Professor Tali Moreshet, and Professor Tong Geng, for their invaluable feedback, generous support, and precious time. I am immensely grateful to all my collaborators and co-authors, especially Prof. Tong Geng, Ang Li, Yuchen Hao, Jianyu Huang, Cheng Tan, Cheng Yang, Qingqing Xiong, Pouya Haghi, and Chunshu Wu. Their contributions, expertise, and insights have significantly enriched the quality of my research.

Additionally, I would like to express my appreciation to all my friends and members of the CAAD lab for their support and encouragement. Special thanks to my friends Pouya, Chunshu, Sahan, Reza, Hafsah, and Zihao for the wonderful years shared during my PhD life.

Lastly, I want to thank my family for their unconditional and limitless support. I am grateful to my dad and mom for their love and unwavering support of my decisions. I could not be who I am without my dearest family.

SOFTWARE AND HARDWARE CODESIGN OF SMARTNIC-BASED HETEROGENEOUS HPC CLUSTERS WITH MACHINE LEARNING CASE STUDIES

ANQI GUO

Boston University, College of Engineering, 2024

Major Professor: Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

ABSTRACT

Machine learning has evolved significantly recently and has penetrated every aspect of science, technology, and daily life. As application prediction demands higher accuracy and more complex tasks, larger models are proposed to meet these requirements. Deep learning applications like recommendation models and large language models have evolved with trillions of parameters and consume up to terabytes of memory. These models have outpaced the growth of GPU memories: GPU clusters, which aggregate GPU memory, have therefore grown exponentially to accommodate these large models. The Memory wall refers to the point at which the demand for memory exceeds the available capacity, creating a bottleneck for training ever-larger deep learning models. Heterogeneous deep learning training has become a key approach to addressing the limitations of GPU clusters, especially as models grow in size and complexity. By combining the strengths of CPUs, GPUs, and NVMe memory, heterogeneous systems aim to overcome the required scale of GPU clusters and mitigate the memory wall limitation by offloading model states and parameters and making it

possible to train ever-growing large-size models on limited resources. However, such heterogeneous system performance is limited by the data exchange, computation, and control efficiency.

Advanced network interface cards, known as SmartNICs, have emerged to mitigate network challenges in scale-out data centers. The placement of SmartNICs as a network-facing computational component within a node allows them to efficiently manage communication between different parts of the distributed system, offloading tasks from the central processors and reducing network traffic bottlenecks. As SmartNICs continue to evolve, they are expected to play a crucial role in enabling more scalable and efficient operations in large-scale data centers, addressing the growing demands of modern applications like machine learning and big data analytics.

In this thesis, we propose heterogeneous smartNIC-based systems for coupling software and hardware for machine learning applications. We explore the heterogeneous system design space in four steps: examining the practical capabilities of emerging smartNIC, integrating host-detached smartNICs into CPU-centric systems, facilitating SmartNICs in GPU-centric systems, and exploring SmartNICs beyond computation offload with heterogeneous global control and disaggregated memory systems. Our proposal involves software-hardware codesign of SmartNIC-based systems, enhancing system performance through dynamic scheduling and control, enabling both GPU and CPU to focus on computation with reduced interruptions. The smartNIC serve as an intermediary layer, breaking barriers between heterogeneous system components and facilitating seamless connectivity between GPUs and CPU offload engines. Additionally, the introduction of a caching system reduces communication workload and memory bandwidth pressure. Furthermore, SmartNICs are attached to the switch level with disaggregated memory, forming a heterogeneous global control system. This system aims to minimize system barrier and synchro-

nization overhead while maximizing communication-computation overlap and model FLOPs utilization for higher system performance.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuits
BSP	Bulk Synchronous Parallel
CGRA	Coarse Grained Reconfigurable Array
DIMM	Dual Inline Memory Module
DL	Deep Learning
DLRM	Deep Learning Recommendation Model
DM	Disaggregated Memory
DMA	Direct Memory Access
DNN	Deep Neural Network
DPU	Delayed Parameter Update
DSP	Digital Signal Processor
FDM	Fetch Disaggregated Memory
FPGA	Field Programmable Gate Array
GCN	Graph Convolutional Network
GCU	Global Control Unit
GNN	Graph Neural Network
GPU	Graphics Processing Unit
HLS	High Level Synthesis
HNIC	Headless Network Interface Card
HPC	High Performance Computing
IP	Internet Protocol
LLM	Large Language Model
MD	Molecular Dynamics
MGT	Multi Gigabit Transceiver
ML	Machine Learning
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communication Library
NIC	Network Interface Card
OS	Operating System
PE	Processing Element
RDMA	Remote Direct Memory Access
RTL	Register Transfer Level
SDN	Software-Defined Networking

SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessors
SmartNIC	Smart Network Interface Card
SoC	System-on-a-Chip
SpMM	Sparse-dense Matrix Multiplication
SpMV	Sparse Matrix Vector multiplication
UDP	Unreliable Datagram Protocol
ZeRO	Zero Redundancy Optimizer

Chapter 1

Introduction

1.1 Characteristics of Deep Learning Computations

In the past few years, the field of large-scale deep learning (DL) has undergone a dramatic transformation, emerging as a dominant force in the contemporary AI landscape. These large models offer remarkable improvements in sample efficiency, enabling them to achieve superior performance with less training data (Kaplan et al., 2020). A key driver of this transformation has been the exponential growth in the size of attention-based deep learning models. As a result, state-of-the-art AI models like Meta's LlaMA (Touvron et al., 2023a; Touvron et al., 2023b), OpenAI's ChatGPT (OpenAI, 2024), and Google's LaMDA (Thoppilan et al., 2022) have delivered astonishing capabilities that are reshaping our daily experiences.

The remarkable performance of these models is in large part due to their sheer size. For example, LlaMA 2, with its 70 billion parameters, and GPT-3, with 175 billion parameters, exemplify the scale at which modern deep learning operates. Google's PaLM, with 540 billion parameters, and OpenAI's latest offering, GPT-4, with a staggering 1.8 trillion parameters, are pushing the boundaries even further. These models represent a trend toward ever-larger AI architectures. *But with them comes not only advanced functionality, but also significant challenges in terms of computational resources and scalability.*

As AI continues to permeate diverse aspects of our lives—from language processing and recommendation systems to healthcare and robotics—the demand for

increasingly large models shows no signs of abating. This upward trajectory in model size reflects both the technological advancements in hardware and software, as well as the growing appetite for more complex and capable AI systems. The next generation of AI applications is expected to continue driving this trend, with researchers and engineers striving to optimize both performance and efficiency at unprecedented scales.

1.2 Challenges of Deep Learning Computations

Machine learning applications create significant obstacles for large-scale distributed systems in three areas: communication, memory bandwidth, and computation efficiency, which we now introduce.

- **Communication:** Communication-intensive operations on such huge data sets — operations such as All-to-All, Broadcast, and All-Gather — require massive data exchanges, leading to communication bottlenecks.
- **Memory:** Some models contain a trillion parameters, consuming terabytes of memory, which puts considerable pressure on memory bandwidth due to frequent memory accesses and buffer usage.
- **Computation Efficiency:** These models involve various computational tasks —ranging from regular parallel computations, like dense matrix multiplication, to more irregular, low-arithmetic-intensity tasks like sparse matrix multiplication, data reshaping, flattening, and transposing. All of these pose distinct efficiency challenges.

While communication, memory systems, and computational efficiency all pose significant challenges when creating efficient and performant systems for large-scale machine learning applications, limitations of current memory systems, in particular,

the *memory wall*, are exacerbating the already critical problems in the other two areas. Historically, the memory wall referred to as the so-called von Neumann bottleneck is a metaphor for the limited bandwidth between compute units and the data upon which they operated (Mudigere et al., 2022). Lately, however, the term memory wall also refers to the point at which the demand for memory exceeds the available *capacity*, creating a bottleneck for training ever-larger deep learning models.

While advanced parallel strategies like 3D-parallel training (Song et al., 2023; Narayanan et al., 2021; Smith et al., 2022) and the Zero Redundancy Optimizer (ZeRO) (Rajbhandari et al., 2020) have facilitated the training of large-scale models, the problem of the memory wall remains a significant obstacle. Despite progress in parallelization techniques, current hardware limitations are not evolving quickly enough to meet the burgeoning demands of cutting-edge models. Modern deep learning models require immense memory resources. To give a sense of the scale needed, fitting a single model can require hundreds of gigabytes to tens of terabytes of GPU memory, depending on the model's size and complexity. GPU memory capacity, however, is not expanding as fast as these models. As a result, maintaining pace with this model growth would require an exponential increase in the size of GPU clusters. Consider a one trillion parameter model: to accommodate a computation with this storage requirement, more than 300 high-end GPUs with 80GB of memory each would be needed. For emerging models of even larger scale, e.g., a model with a hundred trillion parameters, thousands of such GPUs would be required. Note that the problem we describe here is that the requirement for thousands of GPUs is to handle storage alone; for computational purposes, many fewer might be satisfactory.

Beyond this problem of memory/compute resource imbalance, however, this exponentially increasing demand for compute nodes leads to a multitude of issues for the entire system design.

- First, adding more GPUs does not necessarily result in linear performance gains due to the overhead of coordinating between them. Synchronization and communication across a larger cluster of GPUs can slow down overall performance.

- Second, system overhead and communication bottlenecks emerge as critical issues. In large GPU clusters, data must be transferred and synchronized among different processes, resulting in significant communication that can slow down computation.

- Third, larger-scale systems require substantial power, raising concerns about energy efficiency and sustainability. The power consumption of these clusters can become prohibitively high, contributing to higher operational costs and environmental impact. Moreover, scaling up indefinitely means that cooling and infrastructure requirements also increase, presenting additional technical and logistical challenges.

- Finally, the memory wall creates barriers for researchers and companies looking to build and access large models. This is especially true for smaller institutions or startups with limited resources. The high cost of hardware, combined with the increasing complexity of system design and management, can hinder innovation and restrict access to advanced AI technologies.

1.3 Heterogeneous Architectures for Deep Learning

Heterogeneous deep learning training has become a key approach to addressing the limitations of GPU clusters, especially as models grow in size and complexity. By combining the strengths of CPUs, GPUs, and NVMe memory, heterogeneous systems (an example shown in Figure 1·1) aim to overcome the various problems just described that restrict the amount of data that can be processed within a single training step,

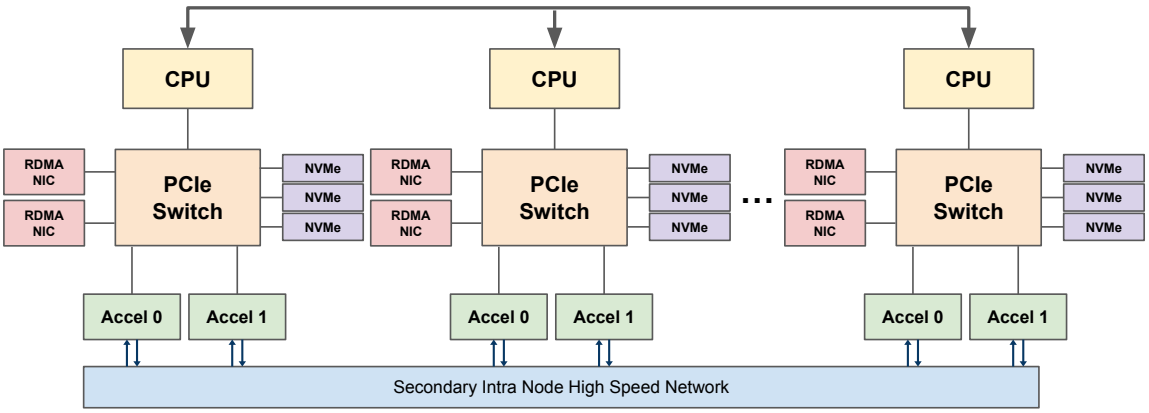


Figure 1·1: An example of heterogeneous systems.

most notably as exacerbated by the memory wall. ZeRO (Rajbhandari et al., 2020) is one such approach. In ZeRO large models are segmented across different GPUs to minimize memory redundancy. Subsequent enhancements like ZeRO-Offload (Ren et al., 2021b) and ZeRO-Infinity (Rajbhandari et al., 2021) have refined this approach by allowing offloading of model states and parameters to CPUs and NVMe storage. This heterogeneous approach, however, brings its own set of challenges, leading to bottlenecks affecting efficiency and performance. Two of these are now described.

One major limitation is the system's **data exchange efficiency**. Since the GPU is primarily responsible for computations, it also has the task of coordinating data transfers among different components. In ZeRO-Offload, for example, while CPUs store the model parameters, gradients, and optimizer states, it is still the GPU's responsibility to initiate, fetch, and communicate data to and from these CPUs. This creates a bottleneck, as all data must go through the GPU in order to facilitate the high-speed network like NvLink, resulting in delays and inefficiencies in the data exchange process. This bottleneck becomes more pronounced when model training is partitioned, necessitating frequent exchanges of data between CPUs and GPUs. Because the GPU acts as the communication hub, it must manage all data movement, leading to a slowdown of the data transfer operations. The process of sending and

receiving data via the GPU can consume significant resources, further straining the system's communication efficiency.

Another major limitation involves the system's **computation and control efficiency**. While offloading techniques, like ZeRO-Offload, employ delayed parameter updates to decouple CPU parameter updates from GPU computation, the GPUs still play a central role in controlling and scheduling CPU tasks. This creates an imbalance where the CPUs and GPUs might have varying workloads, resulting in idle times. Additionally, the GPUs' communication operations require substantial memory and computational resources, adding overhead to the system's operation. The varying computational power and complexity between the CPU and GPU paths can lead to different execution latencies, disrupting the training workflow's synchronization. This can cause one part of the system to wait for the other, diminishing the overall efficiency. The complex communication and control structure in heterogeneous systems, along with the significant overhead required by GPU-based communication operators, collectively reduce the efficiency and performance of these advanced deep learning training systems.

1.4 Addressing Computational Challenges in Deep Learning

We now summarize our discussion so far. For efficient processing of large-scale machine learning applications, GPU-based heterogeneous systems are being used. But while in production use, they still have a number of limitations. A central tenet of this dissertation is that *the limitations can be addressed, in part, by augmenting these heterogeneous systems with Smart Network Interface Cards (SmartNICs).*

SmartNICs represent a significant advance in data center networking. SmartNICs are designed to address the growing challenges of communication in large-scale, distributed environments. Unlike traditional NICs, which primarily handle basic net-

work data transfers, SmartNICs offer additional computational capabilities and can offload certain tasks from the CPU, thereby reducing network congestion and improving overall system efficiency (Broadcom, 2019; NVIDIA, 2020; Marvell, 2020; Mellanox, 2020; Netronome, 2020). Advanced SmartNICs are particularly valuable for computationally intensive applications like machine learning and streaming data analytics. By allowing the programmer to incorporate domain-specific computing capabilities, SmartNICs can handle a range of tasks that traditionally would have required additional processing power from the CPUs or other system components. These include, e.g., operations related to data preprocessing, encryption, compression, and other support for communication protocols such as the Message Passing Interface (MPI) (Caulfield et al., 2016; Jaganathan et al., 2003; Xiong et al., 2019; Xiong et al., 2018b; Xiong et al., 2018a; Schonbein et al., 2019; Wei et al., 2023).

Much of the utility of SmartNICs is due to their most obvious characteristic: their position and role within the node. The placement of SmartNICs as a network-facing computational component node allows them to bypass innumerable hardware and software layers with the potential of drastically reducing the latency of communication-oriented operations. This enables SmartNICs, e.g., to efficiently manage communication between different parts of a distributed system, offloading tasks from the central processors and reducing network traffic bottlenecks. (Nvidia, 2021; Intel, 2022; Xilinx, 2022b) As SmartNICs continue to evolve, they are expected to play a crucial role in enabling more scalable and efficient operations in large-scale data centers, addressing the growing demands of modern applications like machine learning and big data analytics. (Caulfield et al., 2016; Krishnan et al., 2020; Wei et al., 2023)

Specifically as related to the applications addressed in this dissertation: as machine learning models and data analysis tasks grow in complexity and scale, Smart-

NICs could serve as an efficient solution to manage the increasing demands on network and system resources. Furthermore, SmartNICs are well-positioned to address scalability challenges in training and inference like deep learning recommendation models (DLRMs). These models often require high-speed data transfers and low-latency communication across distributed systems, which can stress traditional network infrastructures. With their enhanced communication support and built-in compute capabilities, SmartNICs can help optimize data flows, reduce latency, and improve the overall scalability of ML workloads.

1.5 Problems of Using SmartNICs in Large-Scale Systems

In the last section, we stated that the function of NICs in distributed systems is to address point-to-point communication latency and to offload network functions from the CPU. In this they are successful. *SmartNICs* are currently being used to add to this capability the offload of certain *application-level* network functions, e.g., collectives such as scatter, gather, broadcast, and reduce (Nvidia, 2024a). However, these functions alone do not solve the most critical challenges in large-scale heterogeneous systems: communication bottlenecks, memory bandwidth pressure, and improving compute efficiency. In fact, *the use of SmartNICs is so much in its infancy that there remain a number of basic questions that need to be answered.*

- What are the capabilities of the components within current production SmartNICs?
- Can SmartNICs be decoupled from the rest of the node to reduce the overhead caused by intranode and internode communication?
- Given SmartNICs' unique position within the node, can this be exploited to make the entire **node** within GPU-CPU-Memory-SmartNIC systems more effi-

cient?

- Similarly, given SmartNICs' unique position with the system (network-facing), can this be exploited to make the entire GPU-centric heterogeneous **system** more efficient?

- Using SmartNICs to address the aforementioned problems is likely to require complex designs. What should the design process be? Is it possible to create a unified framework for creating such designs?

1.6 Thesis

So far we have determined that processing large-scale machine-learning applications has particular challenges that appear to be best addressed through the use of GPU-centric heterogeneous systems. These heterogeneous systems, however, have numerous inefficiencies that are being exacerbated by the current memory wall. We propose to address these inefficiencies, in part, by using SmartNICs. The use of SmartNICs for application processing, however, is in its infancy with comparatively little understanding of even their basic capabilities.

In this dissertation, we propose to address the challenges and begin to answer these questions through software-hardware codesign of various SmartNIC-based heterogeneous high-performance computing (HPC) systems with machine learning applications as case studies.

Our hypothesis is that software-hardware codesign enhances system efficiency, resulting in improved performance. Given that applications exhibit diverse workflows, computational characteristics, communication patterns, and performance bottlenecks, a generalized approach alone may not fully exploit the system's capabilities or maximize performance efficiency. Therefore, our approach begins by diving into the application, and conducting performance profiling to identify performance bottlenecks

and algorithm characteristics. On the software side, we focus on optimizing algorithms for improved computational efficiency, adopting hardware-friendly strategies where applicable. Simultaneously, from a hardware and system perspective, we design hardware and system configurations that align with software optimizations. By integrating these two aspects, we propose a software-hardware codesign methodology aimed at fully optimizing system efficiency and achieving higher performance. This approach proposes an integral optimization process that addresses both software and hardware aspects, leading to enhanced system performance and efficiency.

The heterogeneous systems that are the target of this dissertation have component switches, disaggregated memories, and heterogeneous nodes. The latter consists of GPUs, CPUs, and SmartNICs. These systems are built with the goal of improving performance and efficiency, lowering power consumption, mitigating data exchange overheads, and overlapping computation and communication. We explore the system design space in four steps with each step advancing the performance and capabilities of the system and each showcasing broadly a hardware-software codesign approach. The case studies chosen to demonstrate this codesign approach encompass a range of advanced machine learning applications: graph neural networks (GNN), deep learning recommendation models, large language models (LLM), and further generalized large machine learning models.

Our thesis is that high-performance and high-efficiency inference and training of large machine learning models can be achieved by software-hardware codesign of SmartNIC-based heterogeneous high-performance computing systems.

We now introduce four aspects of applying SmartNIC-based heterogeneous high-performance computing systems to improve their performance and efficiency in pro-

cessing large-scale machine learning applications.

Exploration 1: What are the practical capabilities of emerging SmartNICs? SmartNICs have evolved into components that are powerful, but also complex and heterogeneous. SmartNICs are designed to process a wide range of applications. They thus incorporate networking support to address basic NIC functionality, but also CPUs, programmable logic, vector processing, HBM, and DDR memory. Moreover, the components themselves are heterogeneous. For example, the Xilinx Versal ACAP platform has three processors: a traditional FPGA, a cluster of CPUs, and a Coarse Grained Reconfigurable Array (CGRA) of hardware blocks useful in Machine Learning Computations. The problem addressed in this thrust is these SmartNIC architectures are still rarely used to their fullest capability, i.e., with applications that exercise all of these diverse components, even within the devices. Specifically, we study the mapping of graph neural networks to the Xilinx Versal ACAP platform.

Exploration 2: How can SmartNICs be integrated efficiently into the CPU-centric nodes? Just as SmartNICs and their components have become ever more powerful and complex, the same is true of the nodes themselves (Park et al., 2023). This means that there is potentially much overhead in the interaction among these components. Of special interest to our study of SmartNICs is the interaction between the SmartNIC and the host CPU whose workload is being offloaded to the SmartNIC. The second thrust is thus the investigation into the use of distributed SmartNICs as a system *detached from the host*, demonstrating the capabilities of offloading application control and coupling computation and computation. Graph neural networks and deep neural networks (DNNs) are used as applications.

The proposed system is a user-friendly framework for neural network inference on FPGA-Centric SmartNIC (FCsN) that can perform computation, communication, and control altogether at the same time, allowing flexible and fine-grained task cre-

ation, distribution, and execution across multiple SmartNIC devices. By avoiding CPU intervention, the result is maximally hiding the computation latency with network communication for streaming applications at line-rate and achieving high FPGA utilization and high performance at the system level. On the software side, FCsN uses a data-centric programming model and is equipped with Python-based programming APIs. On the hardware side, FCsN is equipped with a hardware-based SmartNIC runtime to achieve CPU-detached scheduling and supports high-performance execution of NN kernels at line-rate. The current FCsN framework focuses on Neural Network applications but has the potential of extending to a general framework as many scientific applications share similar basic kernel functions as NN applications.

Exploration 3: Can SmartNICs facilitate connectivity in GPU-centric nodes? The use of SmartNICs to offload computations is well-established. In this exploration we investigate a potentially higher value role: facilitating cooperation among CPUs and GPUs within a node. In this node design, a software/hardware co-design system leverages SmartNIC capabilities for coupling computation and communication, GPUs accelerators, and CPUs and NVMe as the offload engines. Deep learning based recommendation models and large language models are used as applications.

We introduce a software-hardware co-design of a heterogeneous SmartNIC system for scalable machine learning inference and training that improves the system performance and efficiency, lowers the power consumption and budget, and mitigates the data exchange overhead. The SmartNIC acts 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. A set of SmartNIC optimization techniques of prefetching, caching, and SmartNIC computation kernels exploits locality to reduce data movement, enhance computation and communication

overlap, relieve memory access intensity, and improve GPUs' computation efficiency.

Exploration 4: SmartNICs beyond computation offload – What are the potential benefits of global control and disaggregated memory? A problem in large systems is that as long as overall control remains within the nodes it is doomed to have substantial overhead in making decisions. This is the result of the many layers of hardware and software that must be traversed in order to, first, collect information needed to make decisions, and, second, to transmit those decisions to the rest of the nodes. Moving control into the SmartNICs cuts through many of those overhead layers.

The fourth exploration again involves heterogeneous SmartNIC-based systems but with global control and disaggregated memory. This setup utilizes a *headless* SmartNIC attached directly to a network switch, which allows for centralized system and application control: an improvement over previous designs that lacked such centralized global coordination.

This exploration includes a second aspect. By incorporating disaggregated memory that is also linked to this global control, the system gains several advantages. It facilitates efficient application management and data distribution, reducing the need for extensive synchronization and lowering barrier-related overhead. This design also decreases communication workload, promoting greater overlap between computation and communication tasks.

In summary, we explore how to leverage SmartNICs in large-scale ML systems, demonstrating their critical role in enhancing system performance and efficiency. Our four-step exploration not only augments existing systems but also addresses their limitations. Rather than simply adding SmartNICs without fully utilizing their capabilities, we employ a software-hardware codesign approach. This methodology begins by analyzing applications to identify bottlenecks and inefficiencies. By integrating soft-

ware and hardware, we propose SmartNIC designs that optimize system efficiency and improve performance through greater computation and communication overlap, reduced communication workload, and enhanced application control efficiency.

1.7 Contributions and Outline

The overall contributions of this dissertation are as follows:

- We characterize and employ heterogeneous configurable devices to explore the practical capabilities of emerging SmartNICs.
- We explore and design distributed SmartNICs as a system detached from their hosts, demonstrating the capabilities of offloading application control and coupling computation and computation.
- We propose the software-hardware codesign heterogeneous system with GPU and CPU offload engine that leverages SmartNICs as an intermediate layer that breaks the boundary between distributed heterogeneous components and facilitates seamless connectivity between GPUs and CPU. We introduce optimization techniques facilitated by SmartNICs, including caching systems, prefetching, and computation kernels.
- We propose a heterogeneous SmartNIC-based system with system-level control and disaggregated memory. By attaching a headless SmartNIC to the switch, we establish a centralized system for application management with efficient application handling and data distribution.
- Despite the challenges associated with utilizing SmartNICs in various capabilities, our common methodology enables SmartNIC-based systems to achieve high system efficiency through software-hardware codesign. We demonstrate

the critical role of SmartNICs in large-scale ML systems through higher model training performance and scalability.

The rest of the dissertation is organized as follows. We start with background and related work in Chapter 2. The next 5 chapters follow the four system design space exploration steps. Chapter 3 introduces the exploration SmartNIC capabilities as a device and using the Xilinx Versal ACAP platform showcases the heterogeneous component with the ability to handle various applications. Chapter 4 explores the distributed host-detached SmartNIC systems that the SmartNIC detached from the host, demonstrating the capabilities of offloading control and computation on SmartNICs. The next two chapters explore heterogeneous SmartNIC systems cooperating with GPU with software-hardware codesign, first for DLRMs (Chapter 5) and then for large machine learning applications (Chapter 6). Chapter 7 explores heterogeneous SmartNIC-based systems with global control and disaggregated memory. With the headless SmartNIC attached to the switch, the system is augmented with system global control and centralized disaggregated memory that the previous system was not able to achieve. Finally, Chapter 8 summarizes the entire dissertation and discusses future work.

Chapter 2

Background and Related Work

In this chapter, we provide the background of common components of this dissertation. These include SmartNICs, FPGAs and their use in HPC and SmartNICs, machine learning applications, machine learning parallelization strategies, and machine learning training systems. Background and previous work are described further in each of the next chapters.

We first introduce the different categories of SmartNICs. And then parallel training strategies including data parallel, model parallel (tensor parallel), pipeline parallel, and 3D parallel with their communication patterns. We also introduce the ML applications that we targeted as software-hardware codesign and lastly, we discuss the machine learning training systems that are used.

2.1 Smart Network Interface Cards (SmartNICs)

NICs are fundamental hardware components that enable computers and other devices to connect to the network. NICs serve as the bridge between a device and the network, translating data from the devices' internal architecture into a format suitable for transmission over the network and vice versa. They play a crucial role in facilitating communication and data exchange. Traditionally, NICs have been simple devices focused on basic network communication tasks such as sending and receiving data packets and handling basic network protocols like Ethernet and Wi-Fi. These cards are typically installed in a computer's expansion slot for integration into the

motherboard.

With the increasing demands of modern computing, including high-speed data transfers, virtualization, cloud computing, high performance computing, and distributed applications, NICs have evolved to support more complex tasks. Advanced NICs, known as SmartNICs, offer additional computation capabilities, allowing them to offload specific tasks from the main CPU. This advancement has enabled NICs to play a critical role in improving system efficiency and performance, particularly in HPC, data centers, and cloud environments.

SmartNICs are capable of performing functions such as encryption, compression, network traffic analysis, and even machine learning, reducing the load on the system's primary processors. This evolution reflects the changing landscape of networks, where NICs are no longer just simple network interfaces but are becoming integrated components in system architecture, designed to optimize both communication and computation.

There are dozens of commercial FPGA-based SmartNICs: Xilinx has introduced the Alveo U25 (Xilinx, 2020) and SN1000(Xilinx, 2022b) SmartNICs with FPGA programmable logic and ARM core. Intel released the FPGA-based Intel Infrastructure Processing Units (Intel IPUs) (Intel, 2021) and Intel FPGA SmartNIC (Intel, 2022). Broadcom provides the Stingray SmartNIC (Broadcom, 2019) with an 8-core ARM CPU and P4 packet processing engine. Other commercial SmartNICs have also been developed with the aim of near-network processing (NVIDIA, 2020; Marvell, 2020; Mellanox, 2020; Netronome, 2020).

Researchers have also proposed using SmartNICs as computation resources to offload networking functions and applications. Catapult (Caulfield et al., 2016) uses FPGA-based network solution to offload network applications. Work by (Jaganathan et al., 2003) proposed a configurable network protocol on intelligent NICs. COPA

(Krishnan et al., 2020) provides a software/hardware framework that makes the underlying FPGA hardware (SmartNIC device) agnostic to middleware. FCsN (Guo et al., 2022b; Guo et al., 2022a) proposed a high-performance FPGA centric SmartNIC framework, which supports domain-specific computation, low-latency communication, and host-detached scheduling. INCA (Schonbein et al., 2019) provides general-purpose compute capabilities for SmartNICs that can be utilized when the network is inactive. sPIN (Hoefler et al., 2017) provides a portable programming model to offload simple packet processing. Work in (Wei et al., 2023) presents the first holistic study of a representative off-path SmartNIC, specifically the Bluefield-2, from a communication-path perspective.

SmartNICs have been built with various hardware architectures including SmartNICs with application-specific integrated circuits (ASICs), FPGAs, Graphics Processing Units (GPUs), and Systems-on-Chip (SoC) that combine one or more CPUs with the standard NIC functions. These are now briefly described.

2.1.1 ASIC-based SmartNICs

ASICs are known for delivering high efficiency and performance, primarily due to their specialized nature, which comes at the cost of flexibility. In contrast to FPGAs, ASICs are custom-designed for specific tasks with fixed logic, meaning they are optimized for particular functions but lack the adaptability of an FPGA. Traditionally, developing ASICs involves a lengthy and rigid process, where the entire design, including specifications, test methodologies, and operational scope, is determined at the outset. This contrasts with the agile development approach possible with FPGAs. The ASIC-based SmartNICs are tailored to meet specific networking functions, making them ideal for specific mature applications in HPC, cloud data centers, and advanced networking environments.

ASIC-based SmartNICs operate by handling a range of networking tasks, includ-

ing packet processing, encryption, compression, and load balancing, with specialized hardware logic. This specificity allows for lower latency and faster data transfer rates, as the ASIC is designed to execute these tasks with minimal overhead. Consequently, ASIC-based SmartNICs are a popular choice for applications requiring high throughput and low-latency communication. However, ASICs also face certain limitations due to their specialized design as they lack the flexibility to adapt to new protocols or changing network requirements without redesigning the chip. In summary, while ASIC-based NICs offer high performance and cost efficiency, their development cycles can be inflexible and they are generally limited to data plane operations.

2.1.2 FPGA-based SmartNICs

FPGA-based SmartNICs offer a unique blend of flexibility and programmability. Unlike traditional NICs and ASIC-based SmartNICs, FPGA-based SmartNICs provide the ability to reconfigure their hardware logic, allowing them to adapt to changing network requirements and evolving workloads.

FPGA-based SmartNICs are designed to meet the needs of modern data centers and HPC by enabling customized network functions and offloading various tasks including network protocols, security requirements, and data processing tasks that are continually evolving. By leveraging the programmable nature of FPGAs, these SmartNICs can be updated and reconfigured with new functionalities without hardware changes. A key advantage of FPGA SmartNICs is their ability to perform complex network processing tasks at high speed because of the inherent parallelism and customizable logic of FPGAs. This makes them suitable for packet processing, encryption/decryption, compression network monitoring, and so on. Additionally, they can be programmed to support emerging network technologies and custom protocols, offering a level of flexibility not achievable with traditional ASIC-based NICs. The reconfigurable nature of FPGA-based SmartNIC also provides a future-proofing

advantage, allowing them to keep pace with new developments in network technology. As data centers and cloud platforms become more complex, FGPA-based SmartNICs can adapt to changing demands, providing a scalable and flexible solution for networking challenges.

2.1.3 SoC-based SmartNICs (DPU)

SoC-based SmartNICs, sometimes also known as Data Processing Units (DPUs) (Nvidia, 2021), represent a significant evolution in network technology, integrating advanced processing capabilities directly onto network hardware. Unlike traditional network interface cards which primarily focus on network connectivity, SoC-based SmartNIC combines networking with additional resources, providing a versatile and powerful solution for modern data centers, high-performance computing, and cloud environments. Bluefield SmartNICs leverage ARM cores and NVIDIA's DPU offers enhanced performance and security (Forbes, 2020).

At the cores of SoC-based SmartNICs is the concept of a System-on-Chip, where multiple components — such as CPUs, memory, network interfaces, and other specialized accelerators — are integrated into a single chip. This design allows SoC-based SmartNICs to offload a wide range of tasks from the main server CPU, significantly improving overall system performance and efficiency. The key advantage of SoC-based SmartNIC is its flexibility and versatility. They can handle various network-related tasks, such as packet processing, encryption, load balancing, and virtualized network functions, with the added benefit of programmability and scalability. This flexibility is crucial in modern data centers, where network workloads are becoming increasingly complex and dynamic, requiring adaptable hardware solutions. SoC-based SmartNICs or DPUs are often equipped with dedicated CPUs and memory, allowing them to run software and execute complex tasks without replying to the host server's resources. This capability enables offloading of resource-intensive processes like virtual

machine management, storage processing, and security functions, reducing the burden on the main CPU and freeing up resources for other applications. The result is a more efficient system with reduced latency and increased throughput. Additionally, SoC-based SmartNICs are designed to support modern data center architectures, such as software-defined networking (SDN) and network function virtualization. By enabling programmability and customization, there SmartNIC allows data center operators to adapt to evolving network requirements without requiring significant hardware changes. This adaptability is crucial in environments where scalability and rapid response to new workloads are essential.

2.2 FPGAs and their Utility in Communication

FPGAs for several decades have been critical components in computer and networked communication. This is due to the fact that their key attributes are also crucial in communication. Configurable hardware enables rapid update, e.g., to support new protocols and applications. The massive off-chip communication capability enables high-bandwidth and low-latency data transfers. And the tight coupling of computation and communication delivers unmatched application-application performance. Only a few cycles are needed to get from application logic to the network interface.

2.2.1 Xilinx Versal ACAP Platform

As the SmartNICs are evolving with more powerful and heterogeneous devices including ARM cores, programmable logic, and vector processing, we use Xilinx Versal ACAP heterogeneous SoC platform to explore the potential of SmartNIC as a device. The Versal ACAP (Gaide et al., 2019; Xilinx, 2024b) is a fully software-programmable, heterogeneous compute platform that combines (1) the processor system (PS) - scalar engines that include the ARM processor for general-purpose processing, (2) programmable logic (PL) - adaptable engines that include the pro-

grammable logic blocks providing bit-level flexibility and memory, and (3) Artificial Intelligence Engines (AIEs) optimized for computation-intensive processing with leading-edge memory and interfacing technologies and GTY transceivers for communication.

The PL kernels can be C/C++ kernels or RTL kernels. Its programming model is the same as traditional FPGA. Xilinx AiEs are an array of VLIW professors with SIMD vector units., which are highly optimized for compute-intensive applications. The AIE array provides three levels of parallelism: (1) SIMD - vector registers that allow multiple elements to be computed in parallel, (2) instruction level - VLIW architecture that allows multiple instructions to be executed in a single clock cycle, and (3) multi-core - AIE array where up to 400 AIEs can execute in parallel. The AIE kernels are C/C++ programs written using specialized intrinsic calls (Xilinx, 2024c) or AIE APIs (Xilinx, 2024a) for the VLIW processor. So the PS plays the role of CPU, the PL implements all the FPGA functions, and the AIEs are responsible for the computational acceleration like a GPU.

2.2.2 FPGAs in HPC and Communication

Field-Programmable Gate Arrays (FPGAs) have long been studied for HPC acceleration due to their many advantages (VanCourt et al., 2004; VanCourt and Herbordt, 2004; VanCourt et al., 2005). Although GPUs currently dominate the HPC landscape, FPGAs have shown significant potential to become key components of next-generation HPC systems, especially with advances in programmability (Yang et al., 2017; Sanaullah and Herbordt, 2018b; Sanaullah and Herbordt, 2018c; Sanaullah and Herbordt, 2018a; Sanaullah et al., 2018a; Herbordt, 2019), improved design tools (VanCourt and Herbordt, 2005; VanCourt and Herbordt, 2006), middleware solutions (Haghi et al., 2020b; Haghi et al., 2020a), more general optimization methods (Shahzad et al., 2022; Munafo et al., 2023; Shahzad et al., 2024), and use in the data

center (Shahzad et al., 2021).

Researchers have demonstrated the efficiency and benefits of FPGAs in various scientific computing applications, such as Adaptive Mesh Refinement (Wang et al., 2019b; Wang et al., 2019c), Algebraic Multigrid (Haghi et al., 2020a), Bioinformatics (Herbordt et al., 2006; Mahram and Herbordt, 2012), and security-related tasks (Wolfe et al., 2020; Patel et al., 2022b; Patel et al., 2022a).

Of special note are applications that both require high performance and are communication constrained, i.e., applications for which strong scaling is a particular challenge. The FPGA capabilities of combining accelerator-level compute capability — with low-latency communication between application and physical layers — are a flexible and powerful way to support strong scaling. Deployments include large FPGA-augmented clouds and clusters with either direct FPGA-FPGA interconnects (George et al., 2016; Sheng et al., 2017b; Sheng et al., 2017a; Sheng et al., 2018; Meyer et al., 2023; Kikuchi et al., 2023) or network-facing FPGAs (e.g. on SmartNICs) (Putnam, 2014; Caulfield et al., 2016).

Modeling molecules over long timescales is a widely studied application of this kind, having been addressed specifically with dedicated ASIC-based solutions for classical Molecular Dynamics (MD) (Dror et al., 2010; Shaw et al., 2014; Ohmura et al., 2014; Shaw et al., 2021). Much work has also been done in demonstrating FPGAs as a cost-effective alternative in Molecular Docking (VanCourt and Herbordt, 2006b; Sukhwani and Herbordt, 2008; Sukhwani and Herbordt, 2010), long-range force calculations (VanCourt and Herbordt, 2006a; Gu and Herbordt, 2007; Humphries et al., 2014; Sheng et al., 2014; Sanaullah et al., 2016), range-limited force calculations (Chiu and Herbordt, 2009; Chiu and Herbordt, 2010; Wu et al., 2021b; Wu et al., 2022; Wu et al., 2024b), and integrated MD systems (Yang et al., 2019; Pascoe et al., 2020; Wu et al., 2020; Wu et al., 2021a; Wu et al., 2023).

2.2.3 SmartSwitches: A Smart Communication Alternative

A growing trend in HPC is the importance of the network at the switch level in application support. There has been work that offloads collective processing into switches (Graham et al., 2016; De Sensi et al., 2021). Current in-switch processing, however, is limited in a number of ways. There have been commercial implementations that support collectives, and this support covers a set of scalar and fixed function operations (Faraj et al., 2009; Graham et al., 2016). Academic work (Stern et al., 2017; Haghi et al., 2022) demonstrates support for user-defined extensions, but the resulting switches are confined to inline (aka streaming (Krishnan et al., 2020)) processing.

The IBM BlueGene systems (Almási et al., 2005) offload collectives into the fixed function network router. Recent work by Mellanox (Graham et al., 2016) offloads MPI collectives to fixed logic switches using reduction trees for short message sizes. Increasing switch flexibility has long been a goal of the networking community, culminating, in part, in programmability using P4 (Bosshart et al., 2014).

In-switch computing is becoming an active area of research. The work in (Li et al., 2019) provides an in-switch computing paradigm, implemented on NetFPGA, to accelerate the aggregation of gradients used in the training phase of reinforcement learning. The work in (De Sensi et al., 2021) designs a flexible programmable switch on top of PsPIN (Di Girolamo et al., 2021) building blocks to accelerate Allreduce with custom operators and data types. The work (Liu et al., 2020) presents a Remote Direct Memory Access (RDMA) compatible in-network reduction architecture to accelerate distributed DNN training, in which the FPGAs are connected to the switch and the switch is configured to route the packets that need to be aggregated to the FPGA.

FPGAs have also emerged as promising substrates for coordinating communication and processing offloaded data in Smart Switches (Haghi et al., 2020b; Haghi

et al., 2020a; Haghi et al., 2021; Haghi et al., 2022; Haghi et al., 2023; Haghi et al., 2024; Guo et al., 2022a; Guo et al., 2022b; Guo et al., 2023).

2.3 Machine Learning Parallel Strategies

Machine learning training involves processing large volumes of data to train complex models that require significant computational resources. To scale these computations and reduce training time, parallel strategies are employed. These strategies leverage multiple processing units to perform training tasks concurrently, leading to more efficient use of resources and faster results. Various parallel strategies have emerged including data parallel, model parallel (tensor parallel), and pipeline parallel, and so on. This section introduces the machine learning training parallel strategies and the communication patterns.

2.3.1 Data Parallelism

Data parallelism (Valiant, 1990) is a popular technique used to speed up training on large mini-batches when each mini-batch is too large to fit on a GPU. Under data parallelism, a mini-batch is split up into smaller-sized batches that are small enough to fit on the memory available on different GPUs on the network. Each GPu holds an identical copy of the network parameters and runs the forward and backward pass. At the end of the backward pass, each GPU sends the computed gradients to the main node or a parameter server. The parameter server aggregates the gradients and computes the updates to the network parameters using some variant of stochastic gradient descent. The updated parameters are then sent to each GPU and the process is repeated for a fresh mini-batch.

Increasing the mini-batch size in proportion to the number of available workers often results in a nearly linear increase in training data throughput. But training with larger batches introduces challenges to the optimization process that lead to decreased

accuracy or longer convergence times which downgrade the benefit of higher throughput (Keskar et al., 2017). To address this, various techniques aimed at maintaining model accuracy with the efficiency of large-batch training are introduced (Goyal et al., 2018; You et al., 2017; You et al., 2020). For large scalability, data parallelism with activation checkpointing, where activations are recomputed during the backward pass instead of being stored during the forward pass to rescue memory consumption (Chen et al., 2016).

2.3.2 Tensor Parallelism

Model parallelism is a technique used to distribute neural network computations across multiple devices to handle large-scale models that can't fit entirely on a single GPU. There are two primary categories within model parallelism: layer-wise pipeline parallelism and distributed tensor computation, also known as tensor parallelism. In tensor parallelism, specific parts of a neural network, like model weights, gradients, and optimizer states, are divided among multiple devices. This approach is particularly useful when a single parameter, such as a large embedding table or a dense softmax layer with a high number of classes, consumes a significant portion of GPU memory. Tensor parallelism mitigates these bottlenecks by distributing these memory-heavy components across different GPUs, thus balancing the memory load and improving efficiency.

This strategy is also critical for extremely large models, where pipeline parallelism alone is insufficient. In such cases, splitting the model into smaller layers with pipeline parallelism might not effectively distribute the workload, leading to deep pipelines with high overhead and reduced throughput. For instance, with models like GPT-3 that require partitioning across dozens of instances, relying solely on pipeline parallelism and micro-batching can become impractical, as the resulting pipeline depth and associated overhead make the training process cumbersome and inefficient.

Tensor parallelism provides a more effective solution in these scenarios by distributing large tensors or operations across multiple devices, allowing for scalable model training without overloading any single GPU's memory capacity. This approach enables the efficient training of complex models, achieving a better balance in memory usage and reducing the overhead associated with deep pipeline parallelism. As neural networks continue to grow in size and complexity, tensor parallelism plays an increasingly important role in managing the demands of modern large-scale machine learning training. Mesh-TensorFlow (Shazeer et al., 2018) introduces a language for specifying a general class of distributed tensor computation in TensorFlow (Abadi et al., 2016). FlexFlow (Jia et al., 2019) is a deep learning framework using tensor parallelism that provides a method to pick the best between tensor parallel and pipeline parallel. Megatron-LM (Shoeybi et al., 2020) uses tensor model parallelism that is orthogonal and complimentary to pipeline model parallelism for language models.

2.3.3 Pipeline Parallelism

In contrast to tensor parallelism which splits individual weights, pipeline parallelism keeps individual weights intact but partitions the set of weights. In pipeline model parallelism, groups of operations are performed on one device before the outputs are passed to the next device in the pipeline where a different group of operations are performed. Each batch of training data is divided into micro-batches that can be processed in parallel by the pipeline stages. Once a stage completes the forward pass for a micro-batch, the activation memory is communicated to the next stage in the pipeline. Similarly, as the next stage completes its backward pass on a micro-batch, the gradient with respect to the activation is communicated backward through the pipeline. Each backward pass accumulates gradients locally. Next, all data parallel groups perform reductions of the gradients in parallel. Lastly, the optimizer updates the model weights.

Several methods (Harlap et al., 2018; Chen et al., 2019) incorporate a parameter server (Li et al., 2014) to support pipeline parallelism in distributed machine learning. For instance, the GPipe framework (Huang et al., 2019) for TensorFlow addresses inconsistencies by using synchronous gradient descent. This synchronization ensures that gradients are updated uniformly across all pipeline stages. However, adopting this approach necessitates extra logic to manage the complex interactions between communication and computation within the pipeline. It involves careful coordination to ensure that data is transmitted and processed in the correct order, optimizing the pipeline's efficiency and maintaining consistent model training.

2.3.4 3D Parallelism

3D parallelism training integrates three distinct types of parallelism—data parallelism, model parallelism (also known as tensor parallelism), and pipeline parallelism—to expedite and enhance the efficiency of training large deep learning models. By capitalizing on multiple devices, this technique enables simultaneous processing of different facets of the model, leading to significantly reduced training times and the ability to tackle large-scale models that would be impractical or infeasible with single-device training. Moreover, 3D parallelism optimizes hardware utilization by distributing the workload across available GPUs.

However, implementing 3D parallelism presents challenges due to its complexity, necessitating meticulous planning and optimization efforts. Communication overhead is a particular concern, as coordinating and transmitting data between devices can introduce additional latency, potentially offsetting some of the performance gains.

Notable works employing 3D parallelism include training large language models such as Megatron-Turing NLG (Smith et al., 2022) and GPT-3 (Brown et al., 2020), which demand extensive computational resources. Similarly, in image recognition and computer vision tasks, the speedup provided by 3D parallelism significantly benefits

the training of complex convolutional neural networks (CNNs) (Oyama et al., 2020). In natural language processing (NLP), tasks like machine translation and text summarization, characterized by large datasets and intricate models, also stand to gain from the advantages offered by 3D parallelism (Song et al., 2023; Narayanan et al., 2021).

In conclusion, 3D parallelism represents a potent technique for accelerating deep learning training, empowering researchers and practitioners to train increasingly sophisticated and robust models.

2.3.5 Zero Redundancy Optimizer (ZeRO)

Zero Redundancy Optimizer (Rajbhandari et al., 2020) is a memory optimization parallel 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 proportionately to the number of devices. ZeRO distributes the training batch across multiple GPUs, similar to data parallel training. However, instead of duplicating models, the ZeRO partitions model states across all GPUs and utilizes communication collectives to gather parameters when needed during various phases of the training process. It offers a more generic solution that does not require users to modify the model extensively for implementation, providing improved compute efficiency and scalability.

With data parallel, each process possesses duplicated models. After backward propagation, an All-Reduce operation is employed to calculate gradients used in the optimizer, updating parameters in each process. ZeRO operates in three stages corresponding to three model states, where the model is partitioned into three parts and distributed among three data parallel processes.

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 before each forward and backward pass to share parameters with other data-parallel processes. This process repeats until the completion of all forward pass operations. After each process completes loss computation, a parameter Broadcast is issued before each backward pass partition.

Work has been done to propose a heterogeneous system leveraging CPU or NVMe memory has been explored to augment the system's memory capacity. Such a system enables the training of significantly large models on a limited number of GPUs.

ZeRO-Offload (Ren et al., 2021b) presents a heterogeneous training approach with the CPU as an offload engine, based on the ZeRO-2 foundation to offload optimizer states, gradients, and the computation of parameter updates on the CPU. Figure 6·5 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.

2.3.6 Communication Patterns in Machine Learning Applications

Parallel machine learning training relies heavily on efficient communication between devices to synchronize computation or distribute data across distributed computing nodes. Among the key communication patterns used in parallel training are All-Reduce, Reduce Scatter, All-Gather, and Broadcast. These communication patterns

play a vital role in coordinating data exchange and synchronization among distributed nodes, enabling efficient parallelization of machine learning algorithms.

All-Reduce is a fundamental communication pattern frequently employed in distributed optimization algorithms like stochastic gradient descent (SGD). It involves aggregating data from multiple nodes through a reduction operation, such as summation or averaging, performed across all nodes. The aggregated result is then distributed back to each node, ensuring synchronization of gradients and facilitating collaborative model updates. All-Reduce plays a crucial role in achieving convergence and improving the efficiency of distributed training.

Broadcast is used where data is transmitted from a single source node to all other nodes in the network. This pattern is often employed for distributing global model updates or initializations to all nodes before training begins. By broadcasting data to all nodes, Broadcast ensures consistent starting conditions across the distributed environment, facilitating synchronized training across multiple computing nodes.

All-Gather is a communication pattern that involves collecting data from all nodes and combining it into a single global dataset. This pattern is commonly used for aggregating local computations or gathering statistics from distributed nodes. By collecting data from all nodes, All-Gather enables collective decision-making or parameter updates, contributing to the overall convergence and performance of distributed training algorithms.

Reduce-Scatter is another important communication pattern used for distributing global data, such as gradients gathering and redistribution. Reduce Scatter operates by dividing the data into smaller subsets and distributing these subsets to different nodes in the network. Each destination node receives a portion of the original data, often corresponding to a subset of the overall dataset or a specific segment of the model parameters. Reduce-Scatter is particularly useful in scenarios where global

data needs to be distributed across multiple nodes for parallel processing.

2.4 Machine Learning Applications

2.4.1 Graph Convolutional Networks (GCNs)

Graph Convolution Networks have emerged as a powerful tool in the realm of deep learning, particularly for tasks involving graph-structured data. With the rise of complex data structures such as social networks, biological networks, and citation networks, traditional deep learning architectures faced challenges in effectively capturing the inherent relationships and dependencies within these graphs. GCNs address this issue by introducing graph convolutional operations, enabling the integration of graph structure into the learning process. At its core, a GCN operates on a graph, which consists of nodes representing entities and edges representing connections or relationships between these entities. The key idea behind GCNs is to learn node representations by aggregating information from neighboring nodes, leveraging the graph's topology. This aggregation process is achieved through graph convolutional layers, which adapt the convolution operation from regular grids (as in image data) to irregular graph structures.

GCNs are composed of stacked graph convolutional layers. Each GCN layer follows the Aggregation and Combination paradigm. Particularly, the widely used 2-layer GCN model is

$$X^{(2)} = \sigma(\tilde{A} \cdot \sigma(\tilde{A} \cdot X^{(0)} \cdot W^{(1)}) \cdot W^{(2)}),$$

where $W^{(l)} \in \mathbb{R}^{h_{l-1} \times h_l}$ is the weight matrix of the l-th layer and $X^{(l)}$ is the feature vector of the l-th layer. $\tilde{A} = D^{-\frac{1}{2}} \cdot A \cdot D^{-\frac{1}{2}}$. Here $A = A + I$ is the self-loop adjacency matrix; D is the Laplacian matrix with $D_{ii} = \sum_j A_{ij}$; and σ denotes non-linear activation functions.

Multiplying (A $\times$ X)W first results in a sparse-sparse matrix multiplication that

produces a large dense matrix. Previous work (Geng et al., 2020a; Geng et al., 2021b) found that the order of computation, A $\times$ (X $\times$ W), greatly reduces the scale of computation since both are sparse-dense matrix multiplications (SpMM). We therefore first compute the product of feature and weight matrices, which is the called combination phase. Subsequently, matrix A is multiplied with the result of the combination phase (updated feature matrix); this is called aggregation. Similar to the prior art, we follow a 2-layer vanilla GCN model (Geng et al., 2020a). From now on, for a SpMM computation, we refer to the first (sparse) matrix as LHM (left-hand matrix) with the size m $\times$ k and the second (dense) matrix as RHM (right-hand matrix) with size k $\times$ n.

GCNs typically operate on large, irregular input graphs with relatively small models, i.e., few layers. This contrasts with DNNs, where both the model and the collection of input samples are large, but each sample (e.g., an image) is small. Sometimes, these graphs are so large they cannot be stored in the memory of a single node (Jia et al., 2020). Consequently, training often involves sampling techniques to fit data into a single device's memory, though this reduces accuracy (Jia et al., 2020). Conversely, inference generally processes the entire graph in one batch. More than 90% of infrastructure costs on AWS are attributed to inference, with less than 10% due to training (AWS, 2019; Gasteiger et al., 2022). Moreover, inference is necessary during training. Other approaches are proposed for graph based application (Wu et al., 2024a; Song et al., 2024)

2.4.2 Deep Learning Recommendation Models

Recommendation systems have become an essential component of many online platforms, from e-commerce websites to streaming services and social media networks. The primary goal of these systems is to predict user preferences and provide personalized recommendations, thereby enhancing user experience and engagement. Tra-

ditional recommendation methods, such as collaborative filtering and content-based filtering, have laid the groundwork for this field. However, these approaches often struggle with the challenges posed by large-scale and sparse data, as well as the need to capture complex, non-linear user-item interactions.

The advent of deep learning has revolutionized many areas of artificial intelligence, and recommendation systems are no exception. Deep learning-based recommendation models (Naumov et al., 2019) use the power of deep neural networks to address the limitations of traditional methods and offer significant improvements in recommendation accuracy and scalability. By leveraging vast amounts of data and sophisticated neural network architecture, DLRMs can learn intricate patterns and relationships within the data, enabling them to make more accurate and relevant recommendations.

DLRMs utilize a variety of components to achieve their performance. Embedding layers transform high-dimensional sparse categorical features (such as user IDs and item IDs) into dense, low-dimensional vectors. These embeddings capture semantic similarities and relationships between features, facilitating more effective interaction modeling. Interaction layers, often involving deep neural networks, process these embeddings to capture higher-order feature interactions that are critical for accurate recommendations. The architecture of DLRMs typically includes multiple hidden layers that learn representations at different levels of abstraction. These layers can incorporate various neural network types, including fully connected layers, convolutional layers, and recurrent layers, depending on the specific requirements of the recommendation task.

DLRMs require large memory bandwidth and capacity (Yang et al., 2020a; Zhao et al., 2020; Yin et al., 2021; Lan et al., 2020; Weinberger et al., 2009). Hashing functions are optimized by work like (Weinberger et al., 2009) Works like (Sethi et al., 2022) optimize embedding partitioning and placement techniques. Works like

(Lim et al., 2019; Lin et al., 2020) using various quantization schemes to reduce communication volume. These works address embedding operators using software and algorithm solutions that do not fundamentally solve the DLRM bottleneck in a hardware aspect. Much attention is given to using GPUs as computation accelerators. Work of (Mudigere et al., 2022) introduced a software-hardware co-design system using GPU for distributed training. Work (Kwon and Rhu, 2022) proposed a software runtime system that manages GPU DRM as a fast *scratchpad*. There are works that explore using storage technology to enhance the performance embedding operator of DLRM. Work (Eisenman et al., 2019) presents a storage system that reduces the DRAM footprint using Non-volatile Memory. Work (Wilkening et al., 2021) proposed a near-data processing solution that improves the performance of underlying SSD storage for embedding table operators. However, these works are not focused on the communication bottleneck as the DLRM scales up. Work (Zhu et al., 2021) presents an FPGA cluster for recommendation inference for embedding lookups and computation. Work (Jiang et al., 2021) proposed a recommendation inference engine using FPGA's high bandwidth memory and pipelined dataflow. These works are not targeting scalability as the recommendation model grows even more significant.

2.4.3 Large Language Models (LLMs)

Large language models have emerged as powerful tools in natural language processing, revolutionizing various tasks such as text generation, translation, summarization, and question-answering. These models, typically based on deep learning architectures, exhibit remarkable capabilities in understanding and generating human-like text. The development of LLMs has been primarily driven by advancements in deep learning algorithms, the availability of massive datasets, and increased computational resources.

The foundation of modern LLMs can be traced back to the advent of recurrent neural networks (RNNs) and their ability to effectively model sequential data. How-

ever, it was the introduction of transformer architectures that marked a significant breakthrough in the field of NLP. The transformer model, proposed by Vaswani et al. (Vaswani et al., 2017), relies on self-attention mechanisms to capture long-range dependencies within input sequences, enabling parallelization and more efficient training on large-scale datasets.

One of the key features of LLMs is their ability to learn complex patterns and relationships from vast amounts of text data through unsupervised pretraining followed by fine-tuning task-specific data. Pretraining involves training the model on a diverse corpus of text, such as books, articles, and websites, to learn general language representations. This pre-trained model is then fine-tuned on downstream tasks by providing task-specific supervision, allowing it to adapt its learned representations to the specific task requirements (Devlin et al., 2019).

During inference, LLMs utilize learned representations to generate coherent and contextually relevant text. This process involves feeding input tokens into the model, which then employs its learned parameters to predict the most likely continuation of the sequence based on the provided context. Beam search and sampling techniques are commonly used to generate diverse and fluent outputs, with techniques such as temperature scaling used to control the level of randomness in the generated text (Holtzman et al., 2020).

2.4.4 Machine Learning Acceleration

FPGAs find extensive use in neural network acceleration, e.g., with optimized perceptrons (Sanaullah et al., 2018c), for medical diagnosis (Sanaullah et al., 2018b), for in-brain neural spike sorting (Liu et al., 2016), for training (Geng et al., 2018a; Geng et al., 2018b; Wang et al., 2020b), using CGRAs for QNNs (Geng et al., 2020b), GNNs (Geng et al., 2020a; Geng et al., 2021b), and binarized NNs (Geng et al., 2019b; Geng et al., 2019a; Geng et al., 2021b). Researchers have proposed various

model regularization approaches for CNNs and RNNs, achieving efficient acceleration of regularized models with FPGAs (Shi et al., 2020b). See also (Geng et al., 2021a) for a survey.

2.5 Machine Learning Training Systems

Large-scale machine learning training systems have emerged as essential infrastructure for tackling the challenges posed by massive datasets and complex models in various domains. These systems are instrumental in training models with vast amounts of data, enabling advancements in fields such as image recognition, natural language processing, recommendation systems, and autonomous vehicles. Central to the efficiency and scalability of large-scale ML training systems is the utilization of advanced hardware and communication protocols. In recent years, technologies such as remote direct memory access have gained prominence for their ability to facilitate high-throughput, low-latency communication between computing nodes in distributed systems. RDMA enables direct memory access between nodes without involving the CPU, reducing communication overhead and enhancing overall system performance. Furthermore, the advent of graphics processing units and their associated programming framework, CUDA (Compute Unified Device Architecture), has revolutionized the landscape of ML training. GPUs offer massively parallel processing capabilities, allowing for accelerated computations of complex neural network architectures. CUDA provides a platform for developers to harness the computational power of GPUs, enabling efficient implementation of parallel algorithms for training deep learning models.

In distributed GPU environments, efficient communication among devices is important for achieving optimal performance. The NVIDIA Collective Communication Library (NCCL) is a communication library designed for multi-GPU systems, offering optimized algorithms for collective operations such as all-reduce, broadcast, reduce-

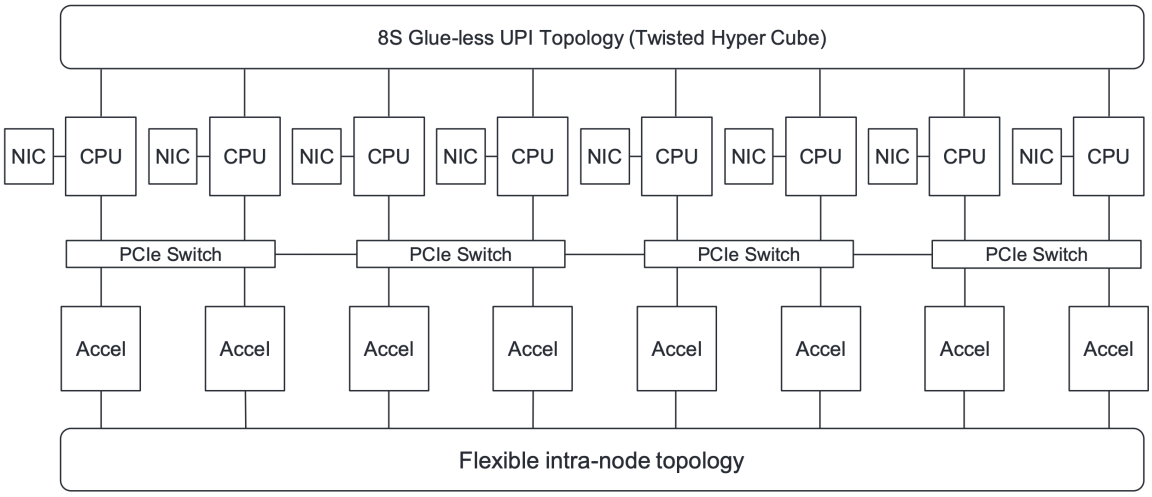


Figure 2·1: The system architecture of Zion which is a high-performance hardware platform for DLRM training. (Mudigere and Zhao, 2019)

scatter, and all-gather. NCCL facilitates efficient data exchange and synchronization among GPUs, with low communication overhead and high distributed training throughput. Large-scale ML training systems leverage these hardware advancements and communication protocols to train complex models, such as large language models on massive datasets. Distributed training frameworks, such as TensorFLow and PyTorch, provide the infrastructure for distributed training of large machine learning models. These frameworks implement parallelization strategies such as data parallelism and model parallelism, distributing computation and memory across multiple GPUs and nodes. RDMA-enabled networking solutions, coupled with efficient communication libraries like NCCL, facilitate high-bandwidth, low-latency communication among devices, enabling seamless coordination and synchronization during training.

Zion, introduced as a high-performance hardware platform for Deep Learning Recommendation Models (Mudigere and Zhao, 2019), offers a robust architecture designed to meet the demands of training complex models. Illustrated in Figure 2·1,

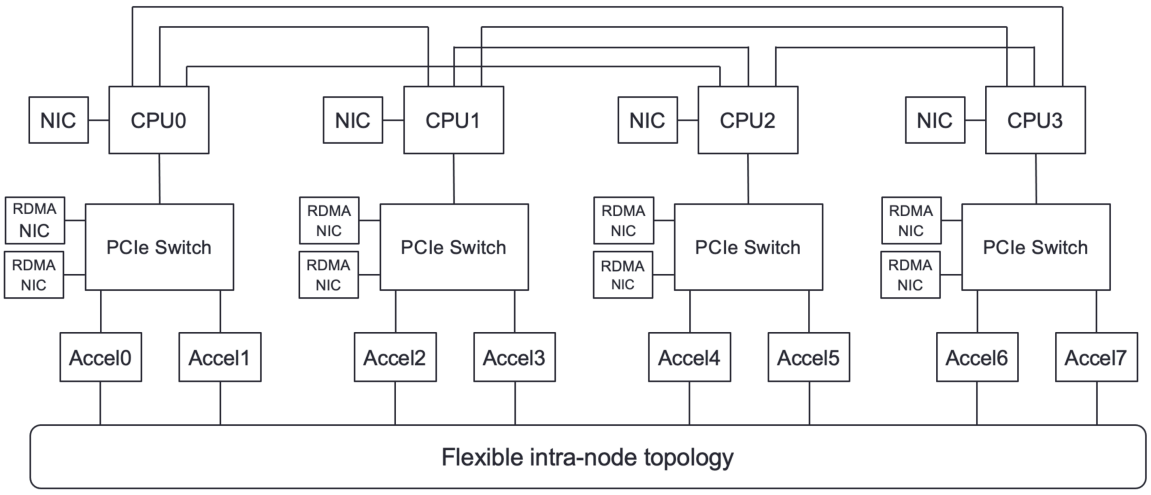


Figure 2·2: The system architecture of ZionEX with improved network capabilities. (Mudigere and Zhao, 2019; Mudigere et al., 2022)

a Zion node boasts 8 CPU sockets accompanied by 1.5 TB of memory, 8 GPUs, and 8 network interface cards (NICs). This design facilitates efficient training of DLRMs by strategically offloading compute-heavy layers onto GPUs while leveraging CPUs for handling large embedding operators in DRAM. This approach enables Zion to accommodate TB-scale DLRMs on a single node. However, while formidable at the individual node level, Zion faces limitations when it comes to distributed training, hindering its scalability to meet the escalating demands of ML training.

To address the scalability challenges, ZionEX (Mudigere et al., 2022) emerges as an evolutionary step forward, as depicted in Figure 2·2. Unlike its predecessor, ZionEX is engineered with scalability in mind, offering both scale-up and scale-out network capabilities. Notably, ZionEX employs dedicated Remote Direct Memory Access over Converged Ethernet (RDMA over Converged Ethernet, or RoCE) NICs for each GPU, connected via PCIe switches. This configuration enables isolated inter-node connectivity, separate from the common data-center network, and supports more efficient RDMA/GPUDirect communication protocols (Naumov et al., 2020). Furthermore, ZionEX nodes can be interconnected through a dedicated back-

end network, forming a cluster tailored for distributed and scalable training scenarios. This architectural enhancement not only streamlines inter-node communication but also mitigates the bottlenecks associated with networking constraints and data-center infrastructure limitations, paving the way for more seamless and efficient distributed ML training.

ZeRO-Infinity (Rajbhandari et al., 2021) introduces a parallel deep learning training system designed to overcome the GPU memory limitation by leveraging heterogeneous memory systems, including GPU memory, CPU memory, and extensive NVMe storage within model GPU clusters. With ZeRO-Infinity, each NVIDIA V100 DGX-2 node can accommodate up to one trillion parameters, representing a staggering 50-fold increase over 3D parallelism. A key feature of ZeRO-Infinity is its utilization of the powerful offload mechanism known as the infinity offload engine. This engine intelligently partitions model states, enabling them to be offloaded to CPU or NVMe memory as needed, or retained within GPU memory based on memory requirements.

MegaScale (Jiang et al., 2024) presents the design, implementation, and operational insights garnered from the development and deployment of MegaScale, a production-grade system tailored for training large language models at an unprecedented scale exceeding 10,000 GPUs. Adopting a comprehensive full-stack approach, the paper co-designs algorithmic and system components spanning model blocks, optimizer design, computation and communication overlap, operator optimization, data pipelines, and network performance tuning. Notably, MegaScale addresses numerous stability challenges that manifest uniquely at a large scale, emphasizing the critical role of in-depth observability in resolving such issues. Demonstrating its prowess, MegaScale is capable of training a massive 175 billion parameter LLM model utilizing 12,288 GPUs.

Chapter 3

SmartNIC Capabilities as Devices

SmartNICs are becoming increasingly powerful and diverse in architecture. In addition to traditional networking functions, SmartNICs are designed to offload various network and compute tasks from the main CPU, enhancing overall system performance and efficiency. Key components include ARM cores, which run various network functions, control plane tasks, and user-defined applications; Network Processing Units (NPUs), specialized processors designed to handle network-related tasks such as packet processing, traffic management, and security functions; and programmable logic, such as FPGAs or other programmable hardware components, which can be customized to implement specific network functions, data processing tasks, and accelerate specific workloads.

Applications like Graph Convolution Neural Networks have different phases and parts, each with its own optimal hardware. In this exploration, we investigate the practical capabilities of emerging SmartNICs and focus on mapping applications to their architecture. We utilize Xilinx Versal Adaptive Compute Acceleration Platforms (ACAPs) to explore SmartNIC capabilities as devices.

3.1 Motivation

In the past few years, GNNs have achieved great success in many applications such as node classification, link prediction, graph classification, and clustering. Among various kinds of GNNs, graph convolutional network, is one category of models that

re-define the notion of convolution for graph data and has attracted substantial efforts from both the industrial and academic communities due to their unique ability to extract latent information from graph data. GCNs have various applications, including citation networks, social network analysis, chemistry, computer vision, and natural language processing.

Despite the popularity of GCNs, accelerating GCN inference is still challenging: GCNs inherit the irregular computational pattern and processing dataflow of graph analytics, resulting in inefficiency on CPUs and GPUs. This is due especially to three factors: (1) irregular data access patterns due to executing on non-Euclidean data, (2) workload imbalance due to skewed distribution of graph degrees, and (3) hybrid computation patterns due to diverse features of different GCN phases. In particular, the Aggregation (or message passing) phase performs vector additions where vectors are fetched with irregular strides, while the Combination (or node embedding) phase can be either dense or sparse-dense matrix multiplication.

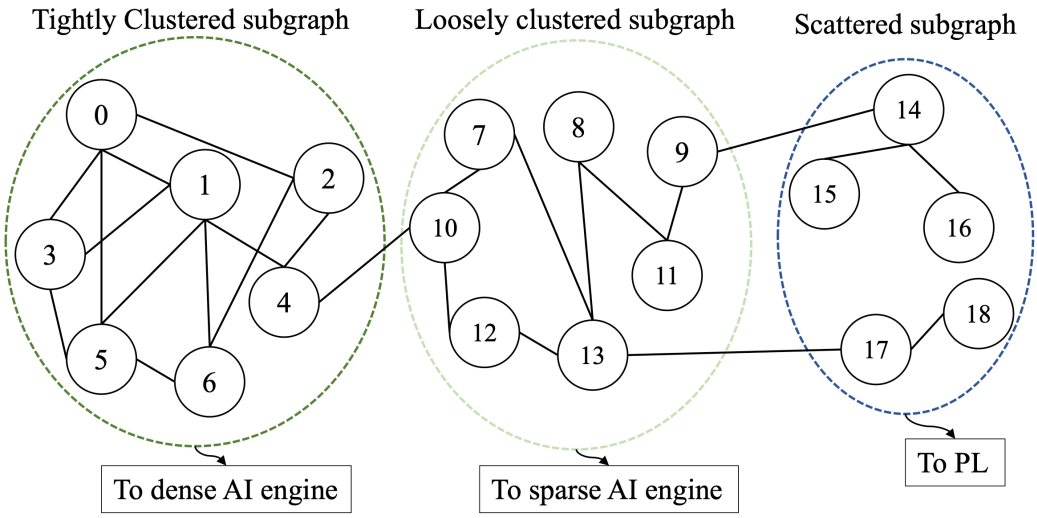


Figure 3·1: Overview of three types of subgraph.

There have been many efforts on GCN acceleration using both GPUs and FPGAs. Researchers have pointed out that the irregularity from graph topology, the resulting

poor data locality, and the serious workload imbalance are the problems. By leveraging FPGA hardware flexibility, existing work has well addressed these problems. However, we observe that besides the irregularity, the heterogeneity of graph structure is also a significant performance limiter. As shown in Figure 3·1, a graph can have tightly clustered components, loosely clustered components, and scattered nodes: it is therefore challenging to use a unified hardware architecture/device to accelerate all parts of the graph computation.

A few works have implemented GCN accelerator on FPGAs (Zhang et al., 2021; Zeng and Prasanna, 2020). However, the overall performance is significantly bounded due to the low frequency of FPGAs compared to CPUs and GPUs. Also, single-instruction multiple-data (SIMD) processing in CPUs can provide high frequency and computation power. Its utility, however, is reduced as the target computation strays from dense, regular operations. This is also the case to some extent in the analogous modes in GPUs and FPGAs. Overall, the heterogeneity of GCN implies that emerging heterogeneous hardware such as Xilinx ACAP may provide an opportunity for further acceleration.

To this end, we propose H-GCN, an accelerator designed to mirror the heterogeneous computing paradigm of GCNs. In particular, H-GCN leverages the heterogeneity of the Versal ACAP to efficiently process different types of subgraphs. The computation of tightly clustered components is mapped onto dense AIEs to fully utilize their high frequency and parallelism from SIMD and very-long instruction word (VLIW) processors. The computation of loosely clustered components is executed on sparse AIEs to reduce computation latency. The computation of scattered nodes is finished on programmable logic (PL) to utilize its programming flexibility. Its performance is not be bounded by the low frequency since the proportion of scattered nodes is relatively small.

H-GCN is an ultra-efficient, systolic tensor-based hardware accelerator that incorporates the features of the PL and AIE for fully utilizing the ACAP's heterogeneous compute capability in GCN computation.

- We study the heterogeneity of graphs and heterogeneity-aware GNN acceleration.
- We are the first to study the use of the AIE compiler in graph processing and sparse matrix processing.
- We design a lightweight grouping strategy to enable sparse tensor computation on the Versal AIEs.
- We develop an efficient method to process tiles of a sparse matrix to enable an automatic mapping of SpMM onto the systolic tensor array.
- Experimental results show that compared with CPU and GPU solutions (i.e., PyG-CPU, PyG-GPU, DGL-CPU, and DGL-GPU), H-GCN achieves up to 3376.3× and 128.7× speedups, respectively. Compared with a state-of-the-art FPGA accelerator, H-GCN achieves 1.4-12.7× speedup on the tested graph datasets.

3.2 Background and Related Work

3.2.1 Xilinx Versal ACAP

Figure 3·2 shows the Xilinx Versal ACAP architecture. ACAP (Gaide et al., 2019; Xilinx, 2024b) is a fully software-programmable, heterogeneous compute platform that combines three components: (1) the Processor System (PS)—Scalar Engines that include the ARM processors, (2) Programmable Logic (PL)—Adaptable Engines that include the programmable logic blocks and memory, and (3) Artificial Intelligence Engines (AIEs) with leading-edge memory and interfacing technologies.

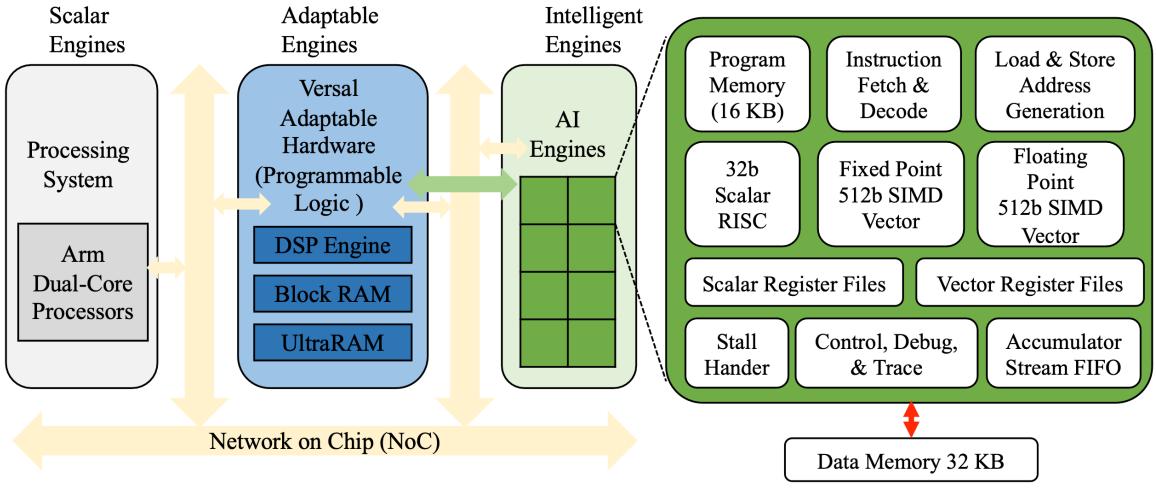


Figure 3·2: Xilinx Versal Adaptive Compute Acceleration Platforms (ACAPs).

The PL kernels can be C/C++ kernels or RTL kernels. Its programming model is the same as traditional FPGA. Xilinx AIEs are an array of VLIW processors with SIMD vector units, which are highly optimized for compute-intensive applications. The AIE array provides three levels of parallelism: (1) SIMD - vector registers that allow multiple elements to be computed in parallel, (2) instruction level - VLIW architecture that allows multiple instructions to be executed in a single clock cycle, and (3) multi-core - AIE array where up to 400 AIEs can execute in parallel. The AIE kernels are C/C++ programs written using specialized intrinsic calls (Xilinx, 2024c) or AIE APIs (Xilinx, 2024a) for the VLIW processor. In this chapter, we mainly use intrinsic calls to implement our AIE kernels and use the Vitis AI compiler "AIE" to compile these codes.

In general, if we compare ACAP to a conventional computing system, the PS plays the role of CPU, the PL implements all the FPGA functions, and the AIEs are responsible for the computational acceleration like GPU. Thus, ACAP illustrates a strong heterogeneity. However, there is no work that takes advantage of such strong heterogeneity in GCN acceleration. In addition, intrinsic calls or APIs are designed

and optimized for dense computation, so there is no prior work that optimizes sparse computation on the AIEs.

3.2.2 Related Work

There has been ongoing research on designing dedicated hardware architecture to accelerate GCNs. For example, HyGCN (Yan et al., 2020) designs hybrid architecture with individual modules for Aggregation and Combination, respectively, to tackle the hybrid computing pattern of Graph Neural Networks. AWB-GCN (Geng et al., 2020a) proposes an autotuning strategy to solve the workload imbalance in GCN acceleration. BoostGCN (Zhang et al., 2021) uses hardware-aware partition centric feature aggregation scheme to increase on-chip data reuse. I-GCN (Geng et al., 2021b) reorders graphs using islandization to improve the data locality so as to achieve better on-chip data reuse and less off-chip memory access. Islandization targets low-frequency, fine-grained, highly flexible PL devices and requires fine-grained hardware architecture, which is not suitable for 2D-mesh AIEs. In the evaluation, we will compare our work with HyGCN, AWB-GCN, BoostGCN, and I-GCN.

Different from all prior work, the proposed H-GCN can fully enable the computational power of the emerging heterogeneous compute platform—Xilinx Versal ACAP—for GCN acceleration by leveraging its strong heterogeneity (e.g., ARM processor, FPGA, and SIMD vector units). To fully explore the capability of ACAP, we propose to mix sparse/dense systolic tensor arrays to accelerate the hybrid computing pattern of GCNs.

In addition, there are a few applications that already leveraged Versal ACAPs. For example, Corradi and Jensen (Corradi and Jensen, 2020) implemented real-time synthetic aperture and plane wave ultrasound imaging on the AIEs. However, there has been no work that explores the way to implement and optimize sparse computation on AIEs.

3.3 System Architecture

3.3.1 Overview of Our Proposed Architecture

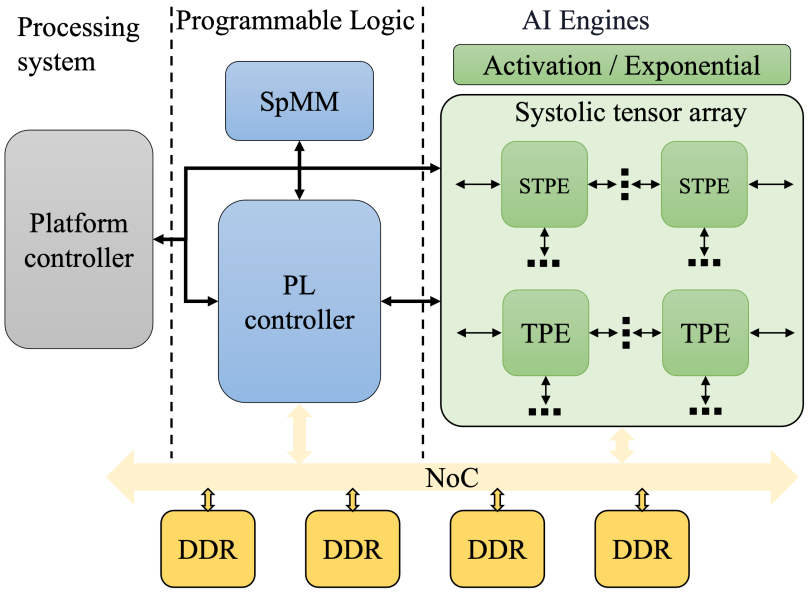


Figure 3·3: Overview of the hardware system design

Figure 3·3 shows the overview architecture of our proposed H- GCN. It consists of a platform controller in the processing system, a sparse-dense matrix-matrix multiplications unit and a PL controller in programmable logic, a sparse/dense systolic tensor array and activation/exponential unit implemented in the AIEs, a network on chip (NoC), four DDR4 SDRAM. The platform controller is used to control the whole system and send instructions to the SpMM unit, PL controller, and sparse/dense systolic tensor array to control their executions and collect their statuses. Specifically, the PL controller controls SpMM unit to cooperate with the sparse/dense systolic tensor arrays to perform all GCN computations. It starts the SpMM unit when it detects that the sparse or dense systolic tensor array has generated enough data. We were inspired by MatRaptor to design our SpMM unit, which adopts row-wise product approach. The PL controller also includes a DDR controller to work with the

NoC to perform data reading and writing. Moreover, the sparse/dense systolic tensor array, which is interconnected side-by-side in a chain/ring fashion, targets the acceleration of both dense and sparse matrix addition and multiplication. It includes both sparse systolic tensor array and dense systolic tensor array; the sparse systolic tensor array is designed for sparse-dense matrix-matrix multiplications in GCNs, while the dense systolic tensor array is mainly for dense-dense matrix-matrix multiplications in GCNs. In addition, our system first performs graph reordering to improve the data locality/reuse and then maps different computations, i.e., dense matrix-matrix multiplication and our optimized SpMM onto different computation engines, i.e., AIEs and PL, based on the matrix density.

3.3.2 Input Graph Reordering

Graph reordering is to optimize both the computation order and the data layouts (e.g., graph-level data locality (Arai et al., 2016)) by modifying the order of vertices. Our goal of reordering is to group the vertices with more shared neighbors together to improve the data reuse when conducting aggregation reductions. The intrinsic reason that the reordering method can provide better temporal reuse is based on the fact that real-world graphs exhibit a "community" structure (Girvan and Newman, 2002), which means some vertices may share neighbors or have a closer relationship to each other; thus, by grouping them together, the data locality during execution will be significantly improved. Note that graph reordering does not change the graph structure but only affects the execution order in the graph.

We perform the graph reordering and sort vertices into a community based on their degrees (Chiang et al., 2019) at the training stage only once using mt-metis (Lasalle and Karypis, 2013). mt-metis is the latest release of an OpenMP version of Metis partitioning and ordering routines.

Figure 3·4 shows the effect of reordering on the Cora and Pubmed dataset (Bo-

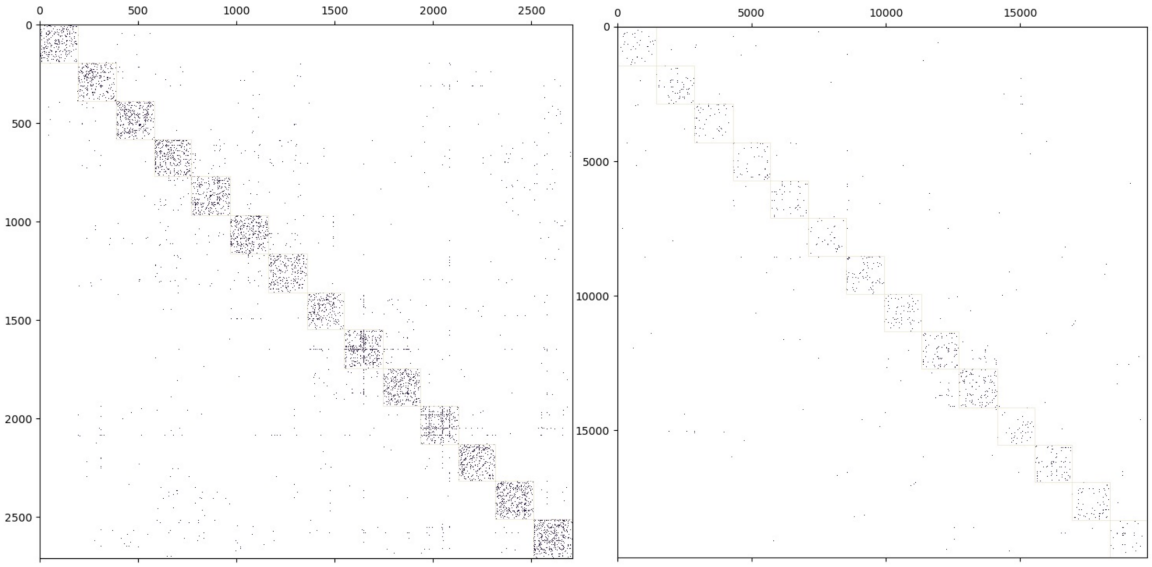


Figure 3·4: The effect of reordering on Cora (left) and Pubmed (right)

jchevski and Günnemann, 2018). It illustrates that most of the vertices are concentrated in the diagonal area forming relatively dense rectangular areas (each dense area is marked with an auxiliary line in the figure). The effect of concentrating vertices in rectangular areas has three advantages: (1) The potential of data reuse is increased. (2) The denser the data distribution, the higher the computational efficiency of the AIEs. (3) The numbers of vertices in different rectangular areas are relatively similar, which can effectively avoid the workload-imbalance issue. After the reordering, to fully utilize the resources of PL and AIEs, we will map the feature aggregation of the vertices in the dense rectangular areas and in the remaining areas onto the AIEs and the PL, respectively. Note that both computations can be performed completely in parallel.

3.3.3 AIE-based Sparse Tensor Engine

The computation mode of GCNs is two-phase matrix multiplication. The essence of matrix multiplication is multiply-accumulate (MAC) operations. Matrix multiplication can be further decomposed into vector operations. An AIE provides a

floating-point 512-bit SIMD vector unit, particularly two intrinsic calls, FPMAC and FPMUL, for vector multiplication and accumulation operations on the vector unit. FPMAC performs multiplication and accumulation for single-precision real number real-time floating-point vectors. FPMUL does multiplication for single precision real times real floating-point vectors. Those intrinsic calls are designed and optimized for dense matrix multiplication.

After the graph reordering, the density of rectangular areas is still lower than 10% based on our extensive profiling results. Thus, we propose a lightweight strategy that enables efficient SpMM on AIEs, which improves the computation efficiency by avoiding zeros be involved in the computation and fully utilizes the high-frequency, single-instruction-multiple-data AIEs. It is worth noting that, without our work, SpMM on AIEs is much slower than running the corresponding dense GEMM directly. Besides, we also use the row-wise SpMM and the traditional sparse storage format CSR to increase the generality of our sparse tensor engine.

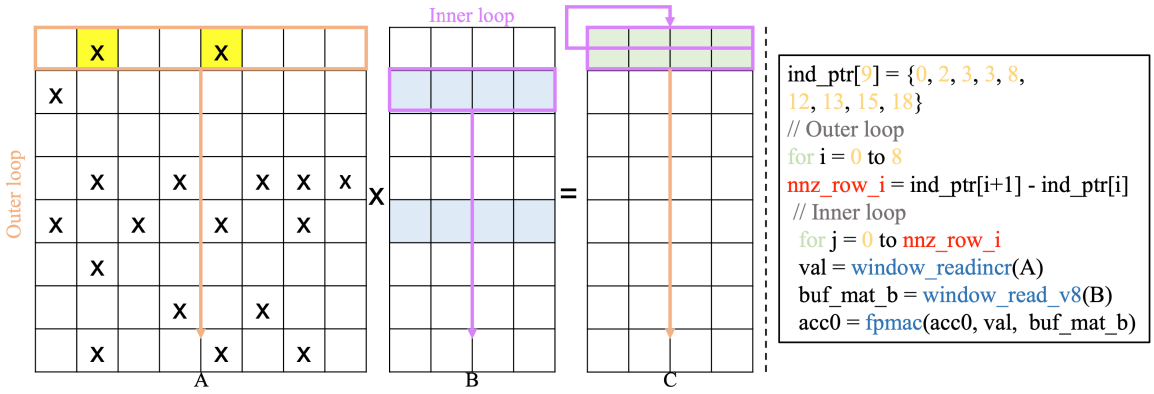


Figure 3·5: Row-wise sparse-dense matrix multiplication

In the Sparse row-wise product approach, all the non-zero elements from a single row of matrix A are multiplied with corresponding rows of matrix B, where the row index of matrix B is determined by the column index of the non-zero value from matrix A. The results are accumulated in the corresponding row of the output matrix

(i.e., C[i;:] = PNk=0 A[i;k]·B[k;:]). Note that multiple rows can be computed in parallel. Figure 3·5 illustrates an example of row-wise SpMM. The challenges of implementing row-wise SpMM include: (1) The number of the innermost loops is not fixed because the number of non-zeros in each row of matrix A is not fixed. The compiler cannot use pipeline or loop flattening to optimize such loops with a variable number of loops, resulting in the final performance being worse than the dense matrix multiplication with the same size, even though we have theoretically reduced the number of calculations. (2) CSR format leads to random row data accesses, which causes low memory bandwidth utilization.

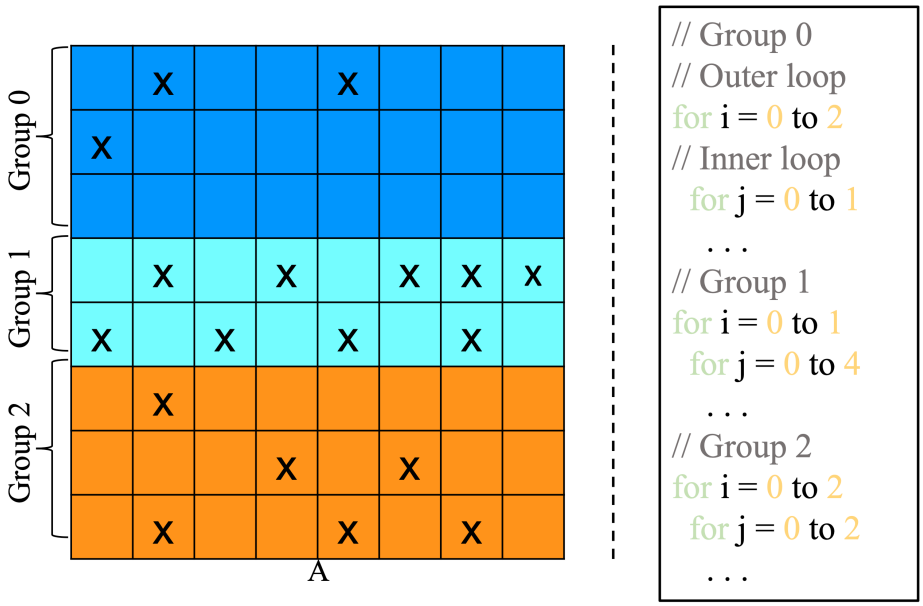


Figure 3·6: Grouped sparse-dense matrix and corresponding program

We note that although directly flattening the outermost loop (each row of A corresponding to a loop) can make the innermost loop fixed, each AIE has limited programming space, and direct expansion will cause compilation failures due to insufficient programming space. To solve this issue, we design a lightweight strategy (shown in Figure 3·6) that divides the outermost loop into multiple loops with fixed number of innermost loops. This allows the compiler to fully optimize both loops.

We propose to use "moving average" to divide the rows of matrix A into multiple groups. Our goals include (1) each group contains as many rows as possible to save programming space, and (2) each group has as little padding as possible to reduce unnecessary calculations on zeros.

Algorithm 1: Proposed grouping algorithm.

Inputs : A: input array; $nnzs\_rows$: non-zeros of each row; $rows$: the number of rows of A; τ: threshold of changing group

Outputs: $group\_dic$: dictionary of group information; $density$: density after padding

```
1  moving_ave ← MovingAverage(); group_dic ← dict();
     idx_g ← 0
2  for i ← 0 to rows − 1 do
3  |   if not exist(nnzs_rows) then
4  |   |   nnz_row_i ← count_nonzero(A[i, :])
5  |   else
6  |   |   nnz_row_i ← nnzs_rows[i]
7  |   end
8  |   pre_ave ← cur_ave
9  |   cur_ave ← moving_ave.update(nnz_row_i)
10 |   if pre_ave == 0 then
11 |   |   pre_ave ← cur_ave   # Prevent division by 0.
12 |   end
13 |   if abs(cur_ave − pre_ave)/pre_ave ≥ τ then
14 |   |   group_dic[idx_g] ← g; g ← [ ]     # update group.
15 |   |   moving_ave.reset(); moving_ave.update(nnz_row_i)
16 |   else
17 |   |   g.append(i)
18 |   end
19 end
20 widensity ← calc_density(group_dic)
```

Figure 3·7: Algorithm 1: Proposed grouping algorithm.

We describe the proposed grouping algorithm in Algorithm 3·7 in detail. We do not need to calculate the non-zeros of each line if *nnzs_rows* already exist (lines 3-7).

We use *pre_ave* to record the previous moving average, and cur ave saves the current moving average (lines 8-9). Moreover, we also need to prevent dividing by zero since *RESET()* function will set *cur_ave* to zero (line 15).

If the change of the moving average exceeds threshold T, we put the data from row j to row i-1 into a group, and we will pad each row in this group to ensure the same number of non-zero elements in each row, where j is the first row of this group (lines 13-18).

3.3.4 Sparse Systolic Tensor Array on AIEs

Two-dimensional (2D) systolic array is a pipelined 2D array of processing elements (PEs). The classical systolic array is generalized into a family of systolic tensor arrays by replacing the traditional scalar PEs with tensor PEs (TPEs). Each TPE is responsible for processing one tile of tensor or matrix. When using the systolic tensor array to perform matrix operations, TPEs in the same row are required to perform exactly the same calculation mode (e.g. MAC) because one tile of data will flow to each TPE in the same row in turn. It is very easy to satisfy such requirements when performing dense matrix multiplication because each TPE only needs to perform vector-based MAC operations. However, it is difficult to meet such requirements when performing SpMM using systolic tensor array, since each tile has a completely different number of non-zero elements and computational models.

To solve this issue, we propose an efficient method to process tiles of a sparse matrix to enable mapping SpMM onto the systolic tensor array automatically. Our idea is to pad the tiles in the same row as little as possible to make them have the same calculation pattern. Algorithm 3·8 describes the simplified workflow of automatic pre-processing of tiles and corresponding tensor PEs generation. We generate different sparse or dense codes for the systolic tensor PEs in the same row as the distributions of non-zeros in different tiles are different. Specifically, (1) we count the non-zeros of

Algorithm 2: Proposed automatic tensor PEs generation algorithm.

Inputs : A: input sparse matrix; $rows$: the number of rows of A; $cols$: the number of columns of A; $tile\_size$: tile size; δ: ratio by which the number of non-zeros changes. p: coverage percentage; d: density threshold of generating dense tensor PE.

Outputs: Sparse or dense code for systolic tensor PEs in the same row.

```
1  tiles_row = rows/tile_size; tiles_col = cols/tile_size
2  for i ← 0 to tiles_row do
3      nnzs_rows ← [0] × tile_size
4      for j ← 0 to tile_size do
5          nnzs_row_j ← [0] × tiles_col
6          for k ← 0 to tiles_col do
7              nnzs_row_j[k] ←
                 count_nonzero(A[i × tile_size + j, k × tile_size :
                 (k + 1) × tile_size])
8          end
9          ave_nnz ← sum(nnzs_row_j)/len(nnzs_row_j)
10         max_nnz ← max(nnzs_row_j)
11         if max_nnz/ave_nnz ≥ δ then
12             nnzs_rows[j] ← find_nnz(nnzs_row, p)
13         else
14             nnzs_rows[j] ← max_nnz
15         end
16     end
17     group_dic, density ← grouping(nnzs_rows)
18     if density ≥ d then
19         gen_dense_tensor_PE(i)
20     else
21         gen_sparse_tensor_PE(i, group_dic)
22     end
23 end
```

Figure 3·8: Algorithm 2: Proposed automatic tensor PEs generation algorithm

tiles in the same row (lines 6-8). (2) We calculate the average non-zeros (ave_nnz) and maximum non-zeros (max_nnz) of all tiles in the same row (lines 9-10). (3) We attempt to find a suitable number of non-zeros (line 12) for all tiles in the same row if the difference between ave_nnz and max_nnz is larger than the pre-defined ratio δ; if we cannot find a suitable number, we will select max_nnz as ideal non-zeros for all

tiles in the same row (line 14). The purpose of this step is to reduce padding as much as possible. The function *FIND_NNZ* is to find the number of non-zeros which covers p percentage of all tiles in the same row. The remaining non-zeros are calculated by SpMM in PL. (4) We use the grouping algorithm described in Algorithm 1 to group the rows (enable efficient SpMM on each AIE) after generating the number of non-zeros in each row (line 17), and obtain the final density after padding. (5) We directly use dense tensor PE for those tiles if their final density is larger than d; otherwise, we use sparse tensor PE to process those tiles (lines 18-22). Based on our profiling experiments, there is no speedup of using spare tensor PE when density higher than 50% (shown in Figure 3·10).

3.3.5 Pipelining SpMM Chains

As described in the previous section and Equation 1, SpMM chains $A \cdot (X \cdot W)$ are executed on three different hardware, i.e., dense systolic tensor array, sparse systolic tensor array, and PL for SpMM. Figure 3·9 illustrates how to map such computation pattern onto the AISEs. Note that that ":" means all indices along this axis. For instance, B[:, 0:32] means a slice from B containing 32 columns across all the rows. Note that there are 400 AIEs distributed in 8 rows and 50 columns. The upper 4 lines of the AIEs are used to implement the mixed sparse or dense systolic tensor PEs (STPEs/TPEs) to perform the computation of $A \cdot B$. We use Algorithm 3·8 to automatically generate corresponding STPEs/TPEs based on the sparsity. According to our experiment, over 90% of the generated systolic tensor PEs are sparse. The remaining 4 lines of the AIEs are used to implement the dense systolic tensor PEs to perform the computation of $X \cdot W$, where B is the intermediate variable generated by $X \cdot W$.

The tile size (i.e., the size of a tile in the blocked matrix-matrix multiplication) of $A \cdot B$ (SpMM) is 64×64. The reasons for choosing this tile size are: (1) A is represented

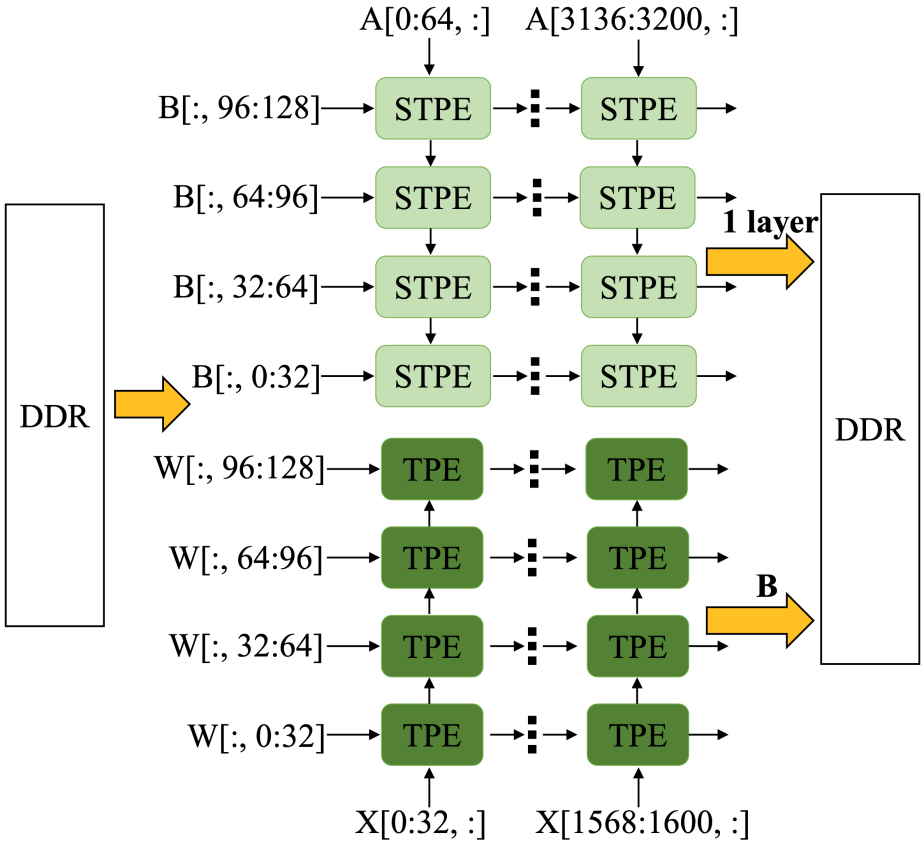


Figure 3·9: Proposed computation mapping strategy and pipelining

in a CSR format, so such a tile of A can be completely stored in the on-chip memory of an AIE. (2) Feeding a large amount of data can ensure the computation efficiency of the AIE. The tile size of $X \cdot W$ (dense matrix-matrix multiplication) is 32×32, which is the maximum size that an AIE can hold after forming systolic tensor array. The remaining 4 lines of the AIEs will be reconfigured to STPEs/TPEs after finishing the entire $X \cdot W$, maximizing the use of all AIE resources. Note the matrix size equals the tile size multiplied by the number of tensor PEs.

Note that A is constant during the inference of a certain graph, once a partial result pB of B is calculated, we can start the multiplication of pB with A on STPEs/TPEs and PL for SpMM immediately without waiting for the entire $X \cdot W$ to finish. Therefore, we can exploit the parallelism between consecutive SpMMs–X

Dataset	# Vertices	A's Density	# Features
Cora	2,708	0.14%	1,433
Flickr	89,250	0.011%	500
Citeseer	3,327	0.08%	3,703
Reddit	232,965	0.04%	602
Pubmed	19,717	0.023%	500
Yelp	716,847	0.0027%	300
Amazon	1,569,960	0.011%	200

Table 3.1: Test Graph Datasets

$\cdot W$ and $A \cdot (X \cdot W)$—in a layer through fine-grained pipelining, as shown in Figure 7. When generating a tile (i.e., 32×32) of intermediate data B, we perform $A \cdot B$ immediately. This pipelining design has two major benefits: (1) It gains extra parallelism and reduces the overall latency. (2) It avoids a part of hardware stalls.

3.4 Experimental Evaluation

In this section, we first introduce the experimental setup and analyze the performance impact of graph reordering and mapping methodologies. Then, we compare the performance of H-GCN with the state-of-the-art GCN accelerators.

3.4.1 Experimental Setup

The Graph accelerator evaluation covers a widely used spectrum of mainstream graph datasets (Geng et al., 2021b; Zhang et al., 2020) including Cora, Citeseer, Pubmed (Yang et al., 2016), Flickr, Reddit, Yelp, and AmazonProducts (Amazon) (Zeng et al., 2020). Details of these datasets are listed in Table 3.1.

Similar to the previous works (Geng et al., 2020a; Yan et al., 2020), the solution is evaluated solution on a two-layer Vanilla-GCN model (Kipf and Welling, 2017) with the hidden dimension of 128.

We use the Xilinx Versal VCK5000 (data center development card) (AMD, 2022)

and its development kit for implementation. VCK5000 features the Xilinx Versal ACAP XCVC1902 device. XCVC1902 device contains 400 AIEs distributed in 8 rows and 50 columns. For PL resources, XCVC1902 device includes 1,968 DSP engines, 1,799,680 CLB Flip-Flops (FFs), 899,840 LUTs, and 34 MB Block RAM. VCK5000 board is equipped with four discrete DDR4 with 72-bit memory interface. The external memory has 100 GB/s peak memory bandwidth with four memory channels. Each channel can provide 25 GB/s peak memory bandwidth. We compile our design using Vitis unified software platform 2020.2.

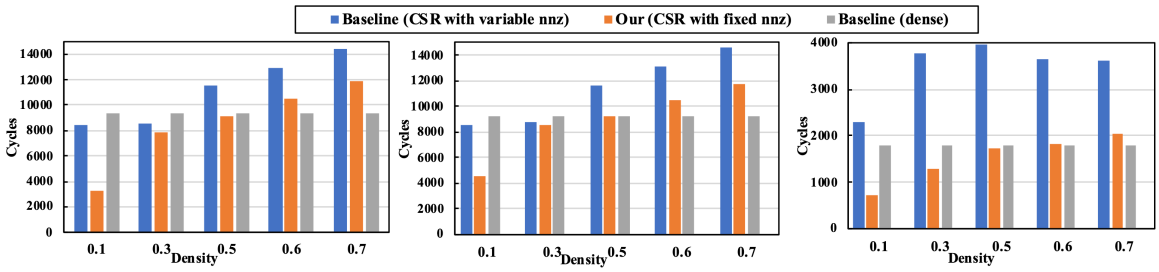
We compare our H-GCN with two advanced, well-optimized geometric deep learning frameworks, i.e., PyG (Fey and Lenssen, 2019) and DGL (Wang et al., 2020a), on general-purpose processors (i.e., CPU and GPU) and the state-of-the-art GCN accelerators, i.e., HyGCN (Yan et al., 2020), AWB-GCN (Geng et al., 2020a), I-GCN (Geng et al., 2021b), and BoostGCN (Zhang et al., 2021). The CPU platform is equipped with two 28-core Intel Xeon Gold 6238R @2.2GHz processors with 384 GB DRAM. The GPU platform is equipped with an NVIDIA RTX 2060 SUPER with 8 GB memory. We denote PyG and DGL running on CPU and GPU platforms as PyG-CPU, DGL-CPU, PyG-GPU, and DGL-GPU, respectively. PyTorch version and CUDA version are 1.11.0 and 11.3, respectively.

3.4.2 Implementation Details

First, we map different partitioned computations to different engines as follows: (1) when the density is higher than 50%, we map the computation of tightly clustered subgraphs onto dense AIEs; when the density is lower than 50% but higher than 1.0%, we map the computation of loosely clustered subgraphs onto sparse AIEs; and when the density is lower than 1.0%, we map the computation of scattered nodes onto PL. Second, we follow three steps to conduct this allocation: (1) we compile the code of AIEs for the computation of clustered or loosely clustered nodes (after reordering)

using the Vitis AI compiler; (2) we compile the HLS kernels of PL for the computation of scattered nodes using the v++ command; and (3) we use the v++ command to link the compiled objects with the target platform (i.e., VCK5000). Third, the frequency of NoC, PL, and AIEs is 800 MHz, 273 MHz, and 1GHz, respectively. The hardware resource utilization and frequency are obtained from the generated report by place and route. Note that the frequencies of PL and NoC are defined by our design choice, while AIEs - an array of VLIW processors with SIMD vector units - have a fixed frequency of 1 GHz. Fourth, the SpMM module only accounts for 15.3%, 84.6%, 14.7%, and 26.6% of BRAM, DSP, FFs, and LUTs, respectively. Last, the evaluation results shown in the following discussion are based on simulations. Xilinx provides a profiling tool called "Vitis Analyzer" (Xilinx, 2022d), which can accurately model the execution time of AIEs.

3.4.3 Speedup of Sparse Tensor Engine



(a) Matrix size: 64 x 64 64 x 8 (b) Matrix size: 32 x 32 32 x 32 (c) Matrix size: 16 x 16 16 x 16

Figure 3·10: Speedups of sparse tensor engine with different grouping strategies under different matrix sizes

First, we evaluate the impact of the grouping algorithm on the overall speedup. We perform the experiments on different matrix sizes and densities as illustrated in Figure 3·10. Since an AIE can only hold up to $64 \times 64 + 64 \times 8$ floating-point numbers, we test matrix sizes up to 64. Compared to the original dense algorithm, our grouping algorithm (i.e., CSR-fixed-nnz) provides 2.9×, 2.1×, and 2.5× speedup

COMPARISON OF INFERENCE TIMES (T) IN μS AND ENERGY EFFICIENCY (E) IN GRAPHS/KJ. OOM IS SHORT FOR "OUT OF MEMORY".

Dataset	PyG-CPU		DGL-CPU		PyG-GPU		DGL-GPU		HyGCN		AWB-GCN		I-GCN		BoostGCN		H-GCN (our work)	
	T	E	T	E	T	E	T	E	T	E	T	E	T	E	T	E	T	E
Flickr	3.5E5	17.37	2.4E5	25.43	1.6E4	3.51E2	1.1E4	5.1E2	N/A	N/A	N/A	N/A	N/A	N/A	2.01E4	N/A	1.02E4	1.0E3
Reddit	6.5E6	0.83	5.4E5	11.26	OoM	N/A	6.6E4	87.07	2.89E5	5.17E2	5.0E4	1.5E2	4.6E4	2.2E2	9.81E4	N/A	4.18E4	2.46E2
Yelp	5.9E6	1.03	8.6E5	7.09	OoM	N/A	2.5E5	23.12	N/A	N/A	N/A	N/A	N/A	N/A	1.93E5	N/A	1.2E5	85.85
Amazon	OoM	N/A	2.9E6	2.1	OoM	N/A	OoM	N/A	N/A	N/A	N/A	N/A	N/A	N/A	7.94E5	N/A	5.15E5	19.93E

Table 3.2: Comparison of inference times and energy efficiency

over the original dense method on matrices of size 64, 32, and 16, respectively, when density is 0.1.

The row-wise SpMM with variable loops (i.e., CSR-variable-nnz), however, is much slower than the dense method even though we theoretically avoid computation on zeros. This is because the Vitis AIE compiler cannot use pipelining or loop flattening to optimize those variable loops.

The speedup gradually decreases to 1 as the density increases, and the speedup disappears when the density is higher than 50%. The reasons are the increase in non-zero elements leads to increases in both the overhead of random access data and the computational delay. Thus, we switch to dense matrix-matrix multiplication when the density is higher than 50%.

We also evaluate the impact of sparsity on the effective FLOPS of an AIE. The effective FLOPS is 7.1 GFLOPS per AIE for dense matrix multiplication. We calculate the effective FLOPS based on nonzeros. FLOPS will increase as the density increases. This is because SpMM needs to convert to dense vector operations for executing on AIEs. For example, the effective FLOPS per AIE for SpMM of 32×32 by 32×32 is 1.6 GFLOPS, 2.5 GFLOPS, 3.1 GFLOPS, 3.4 GFLOPS, 3.5 GFLOPS, and 3.7 GFLOPS, when the density is 10%, 20%, 30%, 40%, 50%, and 60%, respectively.

3.4.4 Comparison with State of The Art

We evaluate the inference latency, and energy efficiency of H-GCN and compare it with other approaches (including software and accelerator solutions).

COMPARISON OF INFERENCE TIMES (T) IN μS AND ENERGY EFFICIENCY (E) IN GRAPHS/KJ.

Method	Cora		Citeseer		Pubmed	
	T	E	T	E	T	E
PyG-CPU	1.1E4	5.36E2	1.7E4	3.65E2	5.7E4	1.07E2
DGL-CPU	7.5E3	8.08E2	2.4E4	2.50E2	2.9E4	2.07E2
PyG-GPU	2.2E3	2.55E3	2.7E3	2.16E3	3.7E3	1.53E3
DGL-GPU	4.1E3	1.39E3	4.6E3	1.23E3	4.96E3	1.15E3
H-GCN	1.1E2	9.18E4	2.9E2	3.56E4	1.03E3	9.93E3

Table 3.3: Comparison of inference times and energy efficiency

First, the "T" columns in Table 3.2 show that H-GCN outperforms the best accelerator I-GCN by 1.1× in terms of inference latency. Moreover, compared with other prior accelerators, H-GCN provides speedups of 1.5× - 2.3× (1.9× on average) over BoostGCN, 1.2× over AWB-GCN, and 6.9× over HyGCN. In addition, H-GCN significantly outperforms PyG and DGL on both CPU and GPU: it achieves average speedups of 79.5× over PyG-CPU, 12.2× over DGL-CPU, 1.59× over PyG-GPU, and 1.58× over DGL-GPU.

The performance improvement is because of (1) the better data locality and hence higher data reuse after the graph reordering, (2) the full use of AIEs via efficient sparse systolic tensor computation, and (3) our proposed scheduling approach for reducing the number of stalls in the overall pipeline.

The "E" columns in Table 3.2 show that H-GCN is 1.12× and 1.64× more energy-efficient than I-GCN and AWB-GCN, respectively, which were previously the most energy-efficient solutions. This is due to the ACAP's more efficient dynamic power management (Xilinx, 2022c). Note that we measure the energy efficiency of H-GCN by using Xilinx Power Estimator (Xilinx, 2022c).

For relatively small graphs, dataflow accelerators such as I-GCN normally preload the graph data into their on-chip buffer and thereby avoid off-chip data access achieving lower inference latency. Therefore, we compare H-GCN with CPU and GPU

platforms for Cora, Citeseer, and Pubmed. Table 3.3 compares inference latency and energy efficiency of relatively small graphs in CPU and GPU platforms. It achieves average speedups of 71.1× over PyG-CPU, 59.8× over DGL-CPU, 10.9× over PyG-GPU, and 19.2× over DGL-GPU.

3.4.5 Performance Breakdown

To demonstrate that the performance improvement is due to the proposed method rather than the graph reordering, we map the computation of dense rectangular areas into AIEs without the approach (using dense systolic tensor array). The inference time of Cora, Citeseer, Pubmed, Flickr, Reddit, Yelp, and Amazon increases by 2.0×, 2.9×, 4.3×, 5.9×, 1.9×, 4.3×, and 3.9×, respectively.

We compare the performance of SpMM (i.e., 64×64 by 64×32) on PL and AIEs with different sparsities. Specifically, when the densities are 0.1%, 0.5%, 1.0%, 5.0%, and 10.0%, the run times of PL are 0.18μs, 0.88μs, 1.75μs, 8.41μs, and 16.82μs, respectively. The run times of AIE are 1.1μs, 2.07μs, 3.84μs, 7.97μs, and 10.44μs, respectively. This illustrates that SpMM on PL is faster than on AIE when the density is less than 1.0%. Thus, we propose to use "density" as our criterion to determine whether to map SpMM onto PL or AIE.

In addition, we propose to prefetch and cache data through the PL controller because the theoretical PL-AIE bandwidth can reach 1.3 TB/s, whereas AIE-NoC bandwidth is only around 12 GB/s. Our evaluation shows that PL-DDR bandwidths of Cora, Citeseer, Pubmed, Flickr, Reddit, Yelp, and Amazon are 72.6 GB/s, 71.9 GB/s, 69.3 GB/s, 81.7 GB/s, 79.0 GB/s, 74.5 GB/s, and 75.7 GB/s, respectively. Note that since Xilinx provides DDR controller IP, we implement our own DDR controller on PL. To calculate the throughput we use RTL simulations to measure the total clock cycles for transferring the graph data.

GRAPH REORDERING TIME (mS).

Cora	Citeseer	Pubmed	Flickr	Reddit	Yelp	Amazon
11.5	11.2	33.6	193	648	1650	7310

Table 3.4: Graph reordering time

3.4.6 Overhead of Graph Reordering

Finally, we evaluate the time overhead of the graph re-ordering, as shown in Table 3.4. Note that as aforementioned, the graph reordering can be integrated into the training process (Chiang et al., 2019), so we take this overhead as the offline overhead. The OpenMP version of Metis takes advantage of multiple cores/threads in the CPU to reorder large graphs in parallel. For the Amazon dataset with 1,569,960 vertices, the graph reordering on 56 CPU cores only takes 7.31 seconds.

Since graphs can evolve dynamically, especially for inductive GNNs, we will support this online graph reordering in our future work. Specifically, we plan to use the host's CPU to reorder the initial graph offline (only once) and the ACAP's ARM CPU to fine-tune the order online (multiple times) as the graph evolves. This will help eliminate the communication cost of transferring node indices between the host and ACAP.

3.5 Summary with Discussion of HW/SW Codesign

The diverse structure of graphs significantly limits the performance of GCN inference. Typical graphs contain tightly clustered subgraphs, loosely clustered subgraphs, and scattered nodes, making it impractical to use a single hardware architecture or device to accelerate all parts of GCN computation. To address these challenges, we introduce H-GCN, an ultra-efficient, systolic tensor-based hardware accelerator with a heterogeneous computation paradigm tailored to GCNs. By leveraging the heterogeneity of

the Xilinx Versal ACAP, we can efficiently process these three types of subgraphs.

Software: On the software side, we identify the bottlenecks in GCN inference and analyze data density through graph reordering preprocessing.

Hardware: On the hardware side, we provide tailored hardware designs to meet different requirements and run matrix multiplication kernels with varying data densities.

Codesign: This software-hardware co-design approach achieves high performance and efficiency for executing heterogeneous applications. Our broad experiments have demonstrated that, compared with a state-of-the-art FPGA accelerator, H-GCN achieves speedups of 1.1-2.3×.

Chapter 4

Distributed Host-Detached SmartNIC Systems

Many distributed system applications suffer from communication bottlenecks. While some simple compute kernels have been successfully offloaded, these approaches often leave the majority of control, scheduling, and management tasks to the host CPUs. This not only places an additional burden on the host CPUs but also results in suboptimal utilization of the SmartNICs. In our second exploration step, we are investigating the efficient integration of SmartNICs into CPU-centric systems. We explore using distributed SmartNICs as independent systems, demonstrating their capabilities in offloading application control and integrating computation with communication.

Previous works have utilized SmartNICs but often faced issues with decoupling from the host. For instance, Intel's COPA (Krishnan et al., 2020) provides a framework for kernel acceleration offloading on SmartNICs, and Portals (Barrett et al., 2012) is a low-level network API for high-performance networking that offloads fundamental operations to support MPI, focusing solely on communication operations. However, these solutions are not capable of handling application control.

In this exploration, we introduce a framework for neural network inference on FPGA-centric SmartNICs. Our framework, FCsN, is a high-performance system that supports application computation, low-latency communication, and host-detached scheduling.

4.1 Motivation

Network communication is increasingly becoming the performance bottleneck for scaling out HPC and warehouse applications, as enormous amounts of CPU cycles are devoted to processing network packets and contributing to long per-packet latency (Figure 4·1(a)). To reduce such latency, advanced network interface cards known as SmartNICs have been introduced for handling networking functions such as TCP Segmentation Offload (Sidler et al., 2015) and Generic Receive Offload (Haghi et al., 2020b). In particular, in cloud computing, SmartNICs can support SR-IOV that forwards packets directly to Virtual-Machines bypassing the hypervisors (Figure 4·1(b)).

Lately, it has been found that if an FPGA can be integrated into the NIC, not only more complex network protocols but also some data-intensive computation, such as reduction and scaling, can be efficiently realized when processing the network packets, often at line-rate and without introducing significant overhead (Figure 4·1(c)). With high-bandwidth and low-latency access to network data through Multi-Gigabit Transceivers (MGTs), and programming logic with embedded hard cores, FPGA-based SmartNICs can be viewed as network-focused streaming-processing accelerators, in addition to network support devices. This is particularly useful for domain-specific computations, such as in machine learning and streaming data analytics, as the FPGAs can be reconfigured as customized accelerators.

Nevertheless, existing FPGA-based SmartNICs are constrained by three limitations. (i) *Host-control*: Although the offloading of some simple compute kernels has been demonstrated, this work generally assumes a host-device programming model, leaving the majority of control, scheduling, and management tasks to the host CPUs. This not only incurs an extra burden on the host CPUs but also leads to poor utilization of the SmartNICs for handling the control dependencies with the host through PCIe and software stacks. (ii) *Limited scalability*. Existing SmartNIC applications

rarely involve the offload of non-local tasks, missing opportunities for system-level designs that can span a distributed cluster, eliminate unnecessary data movement, and support more efficient scheduling and workload balance. (iii) *Programmability.* As the control is performed by the host, most existing SmartNICs only handle relatively simple kernels. Little support is offered to the system and software users for designing flexible domain-specific acceleration solutions.

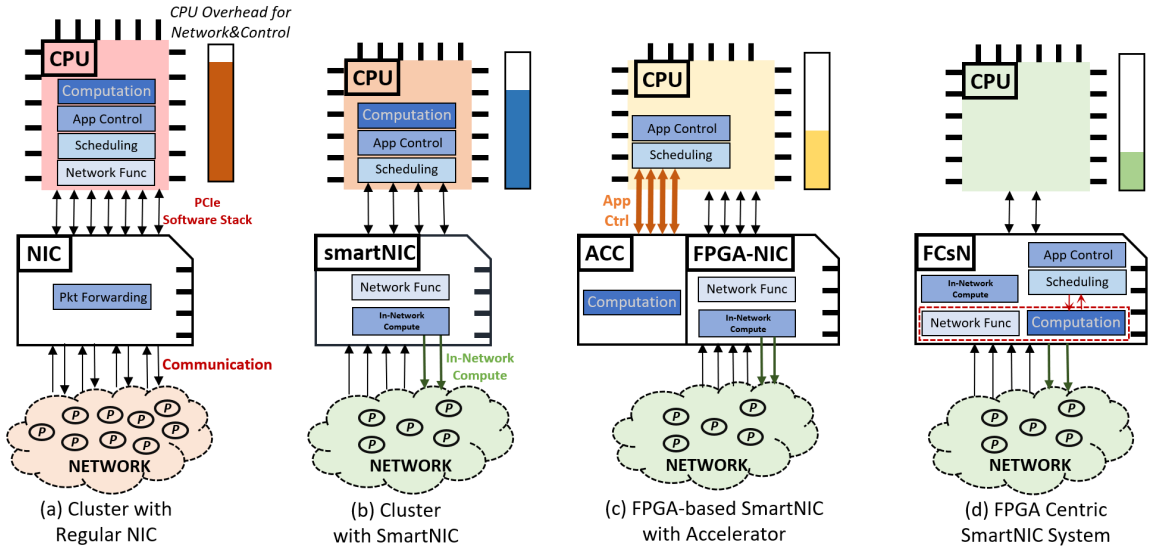


Figure 4·1: **(a)** CPU handles computation and network functions; the NIC is under-utilized and the network traffic is heavy resulting in communication bottlenecks. **(b)** SmartNIC handles network functions and performs simple in-network computing. The CPU is in charge of kernel execution. **(c)** FPGA-based SmartNIC acts as a SmartNIC and accelerator with partially offloaded CPU computations. However, extra overheads between the CPU and FPGA-NIC are introduced by the CPU's intervention of application control logic and scheduling. **(d)** FCsN SmartNIC not only handles network functions, but also offloads application computations, application control, kernel scheduling, and task initiation. CPU cycles are saved, the overhead between the NIC and the CPU is diminished, and FPGA-NIC resources are fully utilized.

In this chapter, we address these problems by presenting a user-friendly **F**PGA-**C**entric **s**mart**N**IC framework (**FCsN**) that can perform computation, communication, and control altogether at the same time, allowing flexible and fine-grained task

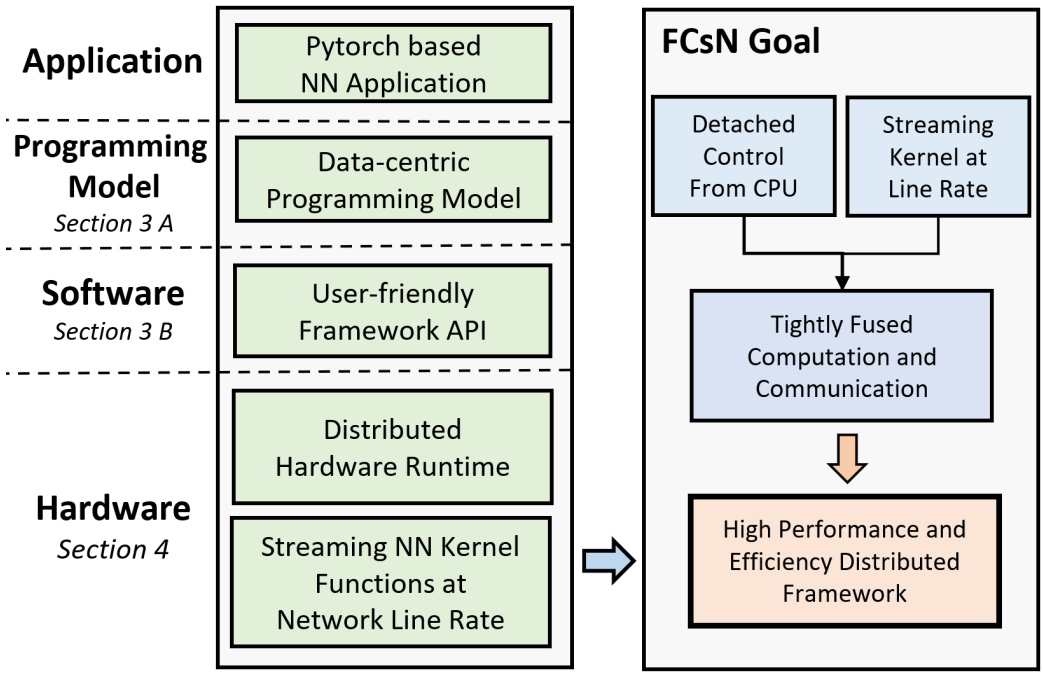


Figure 4·2: Overview of FPGA-Centric SmartNIC Design

creation, distribution, execution, and finalization across multiple SmartNIC devices. This results in maximally hiding the computation latency with network communication for streaming applications at line-rate, and achieving high FPGA utilization and high performance at the system level by avoiding CPU intervention (Figure 4·1(d)). Figure 4·2 illustrates the design stack of FCsN. On the software side, FCsN uses a data-centric programming model (Section 3 A) and is equipped with Python-based programming APIs (Section 3 B); on the hardware side, FCsN is equipped with a hardware-based SmartNIC runtime (Section 4 C) to achieve CPU-detached scheduling and support high-performance execution of NN kernels at line-rate (Section 4 D). The current FCsN framework focuses on Neural Network applications, but it has the potential of extending to a general framework as most scientific applications share similar basic kernel functions as NN applications. Contributions to this work are as follows:

- We propose, FCsN, a user-friendly and high-performance FPGA-centric SmartNIC framework that supports domain-specific computation, low-latency communication, and host-detached scheduling;
- We propose a hardware-based FPGA-centric SmartNIC runtime that enables asynchronous and fine-grained task scheduling and so avoids the control dependency with CPUs;
- We propose a series of streaming NN kernels that provide acceleration at line rate and maximally overlap computation latency with network communication for NN applications;
- Evaluations using neural network applications including general DNNs and GNNs with commonly-used models and datasets on systems with FCsN support and realized on Alveo U280 FPGAs. These show that FCsN can achieve 10× speedups over the standard MPI-based system baseline.

4.2 Background and Related Work

Besides the compute-centric approach, another type of work targets a data-centric approach allowing different tasks to work on the same set of data. The authors in (Tan et al., 2021) propose an asynchronous reconfigurable accelerator based on a ring topology for data-driven high-performance computing. However, this work does not provide network functions and it is basically a method of bringing computation to data. Our work here advances upon (Tan et al., 2021) by providing network functions, keeping up with line rate, and providing a full example of GNN implementation on an FPGA-based SmartNIC system.

4.3 Programming Model & Software Support

In this section, we discuss possible programming models and introduce FCsN's Python-based programming interface.

4.3.1 Programming Models

We investigate two programming models: compute-centric and data-centric. In the compute-centric programming model, a process is assigned to a processor or an entire node and communication happens through message passing. Computations follow a series of steps: parallel computation (each node participates in a portion of assigned tasks) and barrier synchronization to align the execution of nodes. This model works well for BSP applications with easily partitionable data and regular computation. However, many workloads have irregular behavior with skewed data distribution, high synchronization intensity, and irregular communication patterns (Haghi et al., 2021). The data-centric model (Tan et al., 2021) is an alternative. It brings computation to the data rather than the reverse and so minimizes data movement and eliminates unnecessary communication. The application is partitioned into tasks circulating among the system and finishes when all tasks have been executed. The data-centric model suits applications with less structured data and irregular behaviors (e.g., sparse matrices) that introduce unpredictable data access. For example, modern NN applications, which are ever more optimized to reduce computation, are becoming correspondingly more irregular and communication-bound, especially in large-scale processing (Geng et al., 2021a). In these are our initial application targets for FCsN, we consider using the data-centric model.

With the data-centric programming model, applications can be split into two parts, tasks and data, and partitioned into fine-grained tasks executing in a lightweight kernel; these can easily be optimized to achieve network line rate. With the fusion of

compute-kernel execution and communication, the distributed application executes in a streaming manner. During initialization, the tasks can be configured, as needed, into several micro-services, e.g., convolution and aggregation engines. The information for the task is initiated on each node's NIC and the runtime is dynamically generated.

Data is distributed on each node in the application and system initialization phase. When the task demands remote data, the RDMA handler initiates a data request to the remote node. The corresponding task will be temporarily held waiting until the demanded data is fully gathered from the system. The task packets contain the task execution parameters and demanded data information. When the task packet arrives with its target data, the node consumes the packet and injects the task into the compute pipeline. In this way, the data-centric approach brings the compute tasks to each node asynchronously instead of sending large chunks of data into the network.

4.3.2 FPGA programming interface

To support a user-friendly interface, a middle layer API coordinates activity between the host CPU and underlying hardware, including system and kernel initialization, hardware status checking, data syncing, and hardware control. A list of API functions is shown in Table 4.1.

FCsN supports most of the major kernels used in Neural Network processing, including 2D convolution, dense and sparse matrix multiplication, graph aggregation, norm, and non-linear element-wise activation functions. The APIs can be easily and seamlessly integrated into PyTorch-based NN applications. Based on the NN application's need, corresponding kernel function tasks can be configured. Before a kernel's execution, data that needs to be carried by tasks are synced to on-chip *'Spawn_MEM'*. *'Kernel_start'* starts the task spawner module based on task data from *'Spawn_MEM'* and dynamically generates kernel task runtimes based on *'Task_id'*.

SOFTWARE PROGRAMMING API AND NEURAL NETWORK KERNELS

Function	Description
Software Programming API	
Overlay_handler = **System_init** (*Node_num*)	FCsN multi-node environment initialization setup
Board_init(*Overlay_handler*, *Bit_file*)	Configure network function, check board status and program the board with binary file
Check_Kernel()	Check kernel status
Data_handler = **Host_data_preprocess** (*data*)	Host preprocess data
Mem_handler = **Board_MEM_init** (*Overlay_handler*, *Data_handler*)	Allocate on-chip Memory, sync and distribute data from host to chip.
Spawn_MEM = **Board_MEM_init** (*Overlay_handler*, *Task_carrying_data*)	Allocate Task Spawner Memory with task required data.
Kernel_Start (*Overlay_handler*, *Task_id*, *Argv*)	Start the kernel with Task_id and control argument
Sync_to_Host(*Overlay_handler*, *Data_handler*)	Sync on-chip Memory data back to host Memory
System_Finalize(*Overlay_handler*)	Free overlay
Neural Network Kernels	
2D_convolution	Task_id = 0
Dense_Dense_Matrix_Multiply	Task_id = 1
Sparse_Dense_Matrix_Multiply	Task_id = 2
Sparse_Sparse_Matrix_Multiply	Task_id = 3
Function_Norm	Task_id = 4
Non_Linear_Activation	Task_id = 5
Aggregation_function	Task_id = 6

Table 4.1: Software API and neural network kernels

'argv' indicates the task spawner control arguments such as destination or data range.

To associate middle layer API with hardware, we use Xilinx Pynq (Xilinx, 2022a)

as API to program and interact with Xilinx XRT (Xilinx, 2024d) and Vitis platform FPGA. Pynq is an open-source Python-based library for programming both the embedded processors and the overlays. We use Vitis HLS to implement basic hardware NN kernel functions. The Vitis compiler compiles the HLS kernel into an xo file, creating an overlay by linking with other kernels, e.g. network function, DMA engine, and so on.

4.4 FPGA-based SmartNIC Architecture

In this section, we describe the hardware architecture and supported basic NN kernel functions with streaming execution capability.

4.4.1 Architecture Overview

The FCsN architecture consists of a static shell that provides the basic infrastructure and dynamic user logic (PR-Region) and is shown in Figure 4·3. Within the dynamic user logic, the overlay is split into network function, hardware runtime, and NN kernel compute engines. Neural network kernel functions are configured according to the application's requirement during the initialization phase.

4.4.2 Network Functions

Also implemented and tested is light-weight, low-latency transceiver support using Xilinx Aurora IP (Xilinx, 2024e). The network module uses UDP as the layer 4 transport protocol.

4.4.3 Hardware Runtime Support

The hardware runtime is split into two stages, runtime initialization and runtime execution. Runtime initialization enables the host to program the dynamic application chip area with the user's application binary and sync data from host memory

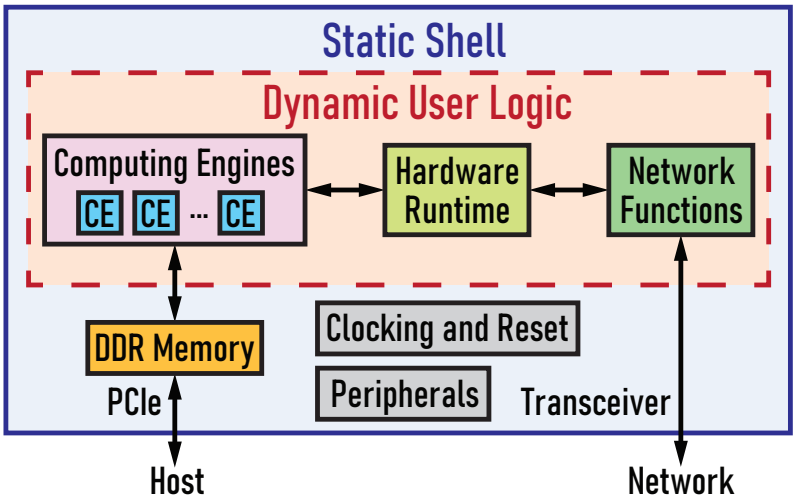


Figure 4·3: FPGA-based SmartNIC Overlay

to FPGA's memory through DMA support with the static shell. Runtime execution handles application task spawning, scheduling, and application control logic, and manages RDMA data.

Runtime initialization loads the binary, allocates resources, and prepares for runtime execution. After the user provides the on-chip design (and compiles the kernel into a binary), the kernel is inserted into the dynamic user logic region and connected with network functions through an AXI stream. Distributed application kernel tasks are assigned to each node and partitioned data in each node is preloaded onto the chip. Runtime execution invokes the corresponding task spawner and dynamically generates tasks with preloaded information (Figure 4·4).

As described in the previous section, the kernel should be able to keep up with the network line rate. To achieve this, data is preloaded into the BRAMs and partitioned in order to be able to fetch the demanded data in one cycle. BRAM optimization avoids data conflict within each packet's tasks.

After the runtime initialization, task-specific information is distributed among each node based on the user's specification. The runtime task spawner dynamically

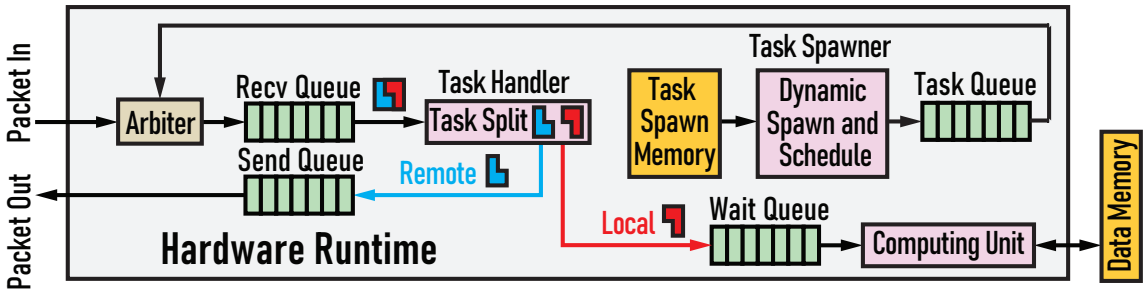


Figure 4·4: Data-Centric Hardware Runtime Overview

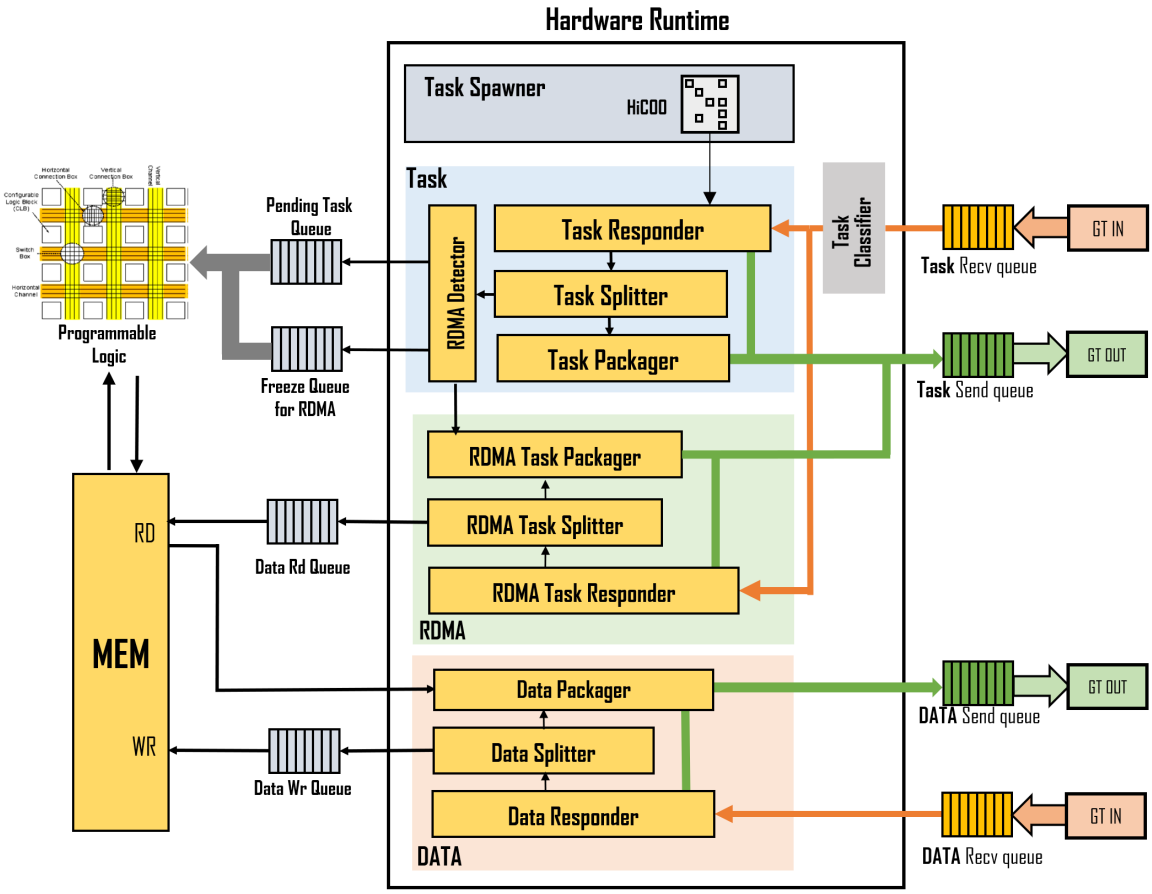


Figure 4·5: Hardware Runtime Architecture

generates tasks based on the on-chip memory's task info. In our design, we wrap each task as a 512-bit packet with a 32-bit application header and 480-bit payload. The header contains information on task type, task ID, task destination, and needed data range. The payload is formatted by different kernels and the corresponding kernel will

parse the payload. The packets are classified into three types (Task packet, RDMA packet, and Data packet). Task packets execute the application kernel task when they arrive at the destination node or find the corresponding data range. RDMA packets perform the RDMA data fetching request from other nodes. Data packets send the requested data back to the requesting source. The application data is stored in FPGA on-chip memory, so requesting data does not need to interrupt the host. The packets describe the destination node either with the destination node id or with a data range that the task is working with. A remote data header field is also preserved if the task needs to request remote data from other nodes. When there are multiple kernels supported or needed by the application, the task ID identifies which operation needs to be executed. The hardware runtime monitors the network and retrieves packets when there is a valid task packet. After the hardware runtime has parsed the packet, the task handler consumes the task if the current node contains the demanding data (Figure 4·4).

As we have three types of task packets (task packet, RDMA packet, and data packet), each packet has its corresponding runtime handler. Figure 4·5 shows the architecture of the hardware runtime.

Task Handler: When there is a new valid packet received from the network, it is first appended to the receive queue. The Task Classifier pops the first packet and appends it to the corresponding handler. Within each handler, the Task Responder checks the integrity of the packet and whether the packet can be fully or partially executed in the current node (Figure 4·4). (1) Partially executed means the current node does not have all the data it needs. In this case, the Task Splitter splits the packet, consumes the local task, and generates a new packet with the rest of the unavailable data on the current node. The Task Packager wraps the task with the requested information, pushes it into the send queue, and sends it into the network.

(2) The consuming task is sent to an RDMA Detector to check if the task needs a remote node's data. (i) If the task does not need remote data, it is appended to the pending task queue and waits to be assigned to the desired compute unit. (ii) If the task requests remote data, the task is temporarily pushed to a freeze queue and waits for the requested data to be available on the current node. The RDMA Handler generates a corresponding RDMA task.

RDMA Handler: An RDMA task requests remote data. It is analyzed in the RDMA Task Responder to determine whether the requesting data is available on the local node. After consuming the RDMA task, the task is pushed into a data read queue and waits for the memory controller to issue its requested data. When the data has been retrieved, it is wrapped by the Data Packager and sent to the requesting source node.

Data Handler: Data Handler handles data packets when the requesting data from the current node's RDMA task has been successfully retrieved. Meanwhile, the RDMA Detector is informed that new data has arrived and checks whether there are tasks ready to be computed.

4.4.4 Neural Network Kernel Support

In FCsN, streaming kernel execution at the network line rate leverages the SmartNIC capabilities. We provide optimized kernel functions for Neural Network applications including 2D Convolution, Matrix Multiplication, Function Norms, Non-linear element-wise Activation, and Aggregation functions.

With the aim of achieving stream execution at line rate, the compute pipeline is required to have the capability of consuming one task packet each cycle. Within the pipeline, sub-tasks in each packet should not have memory conflicts or other hazards that stall the pipeline and cause packets to drop. Also, the latency of each task needs to be fixed with predetermined pipeline stages. Details of supported NN function

kernels are as follows:

2D Convolution: In 2D convolution, a small matrix of kernel weights "slides" over the 2D input data, performing an element-wise multiplication with the part of the input it is currently over, and then summing up the results into a single output pixel. The weight kernel and 2D input data are preloaded into BRAM. 2D input data is fed into the kernel pipeline in a streaming manner. In FCsN with its distributed approach, halo exchange happens between the edges of partitioned 2D input data. 2D convolution halo tasks are generated to carry halo exchange data to the neighbor nodes.

Dense and Normal Sparse Matrix Multiplication: With the distributed matrix multiplied between matrix A and B in FCsN data-centric programming model, matrix B can be treated as 'data' distributed in the system, and matrix A can be broken down into tasks. The task contains matrix A's value and the corresponding demand portion of matrix B. In a Neural Network, two types of matrices are involved, dense and sparse. We recommend generating tasks with the sparse matrix if there is one that introduces fewer tasks and higher efficiency. Task tokens carries the non-zero element in the sparse matrix or a slot of dense matrix data. When the current node contains the task demanding data range, it enters the matrix multiply pipeline and finishes the operation.

Function Norms: Norms, such as layernorm and batchnorm, provide the neural networks with the capability of adjusting the statistical distribution of the input/output data for better performance (Rannen-Triki et al., 2019). They are realized through vector dot product (to calculate statistics such as sum and mean) and element-wise multiplication and addition (to scale and add biases). Their implementation can be modified from 2D Convolution and Matrix Multiplication kernels to realize the line rate.

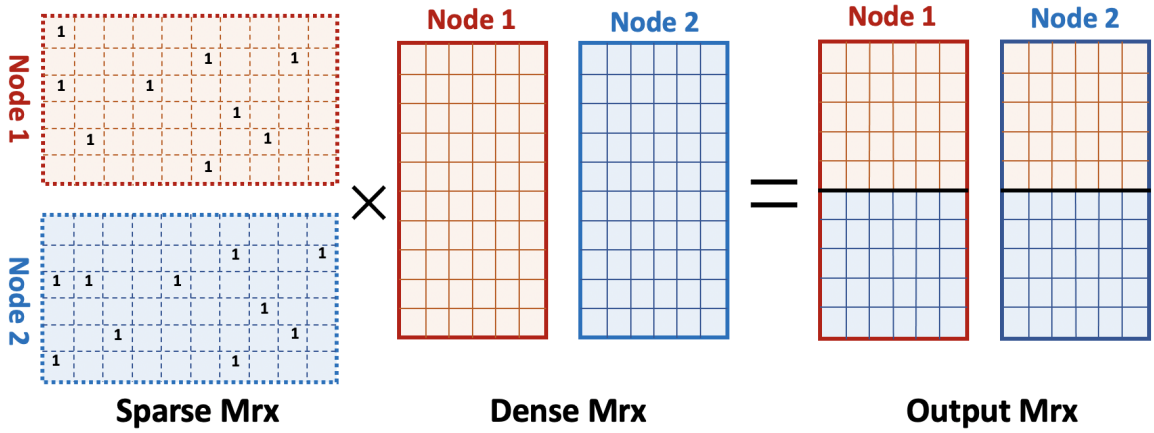


Figure 4·6: Aggregation Function Overview

Non-linear element-wise activation functions: Activation functions are mathematical equations that add non-linearity to the neural network models for improved prediction accuracy (Nwankpa et al., 2018). In the case of ReLU (Agarap, 2019), the activation function determines whether the output neurons should be activated or not, based on their signs. The activation function can be inserted in the computation pipeline acting on the output of the neuron.

Some operations combine the function of nonlinear element-wise activation and function norms, such as softmax (Liu et al., 2017). It can reuse and combine the previously implemented kernel with line rate capability.

4.4.5 Aggregation Functions

Within GNN workloads, aggregation is an extreme case of Sparse Matrix Multiply with irregular data access which is hard to achieve streaming execution (Geng et al., 2020a; Geng et al., 2021b). Details of achieving streaming kernel execution at line rate with aggregation function are described in this section.

Aggregation Function Operation: Aggregation is a matrix multiplication between an extremely sparse adjacency matrix (>= 99%) and dense weight matrix

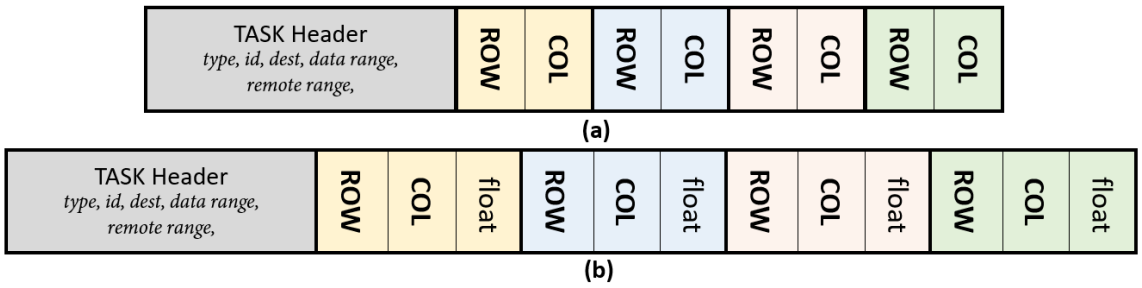


Figure 4·7: Aggregation Task Packet Format

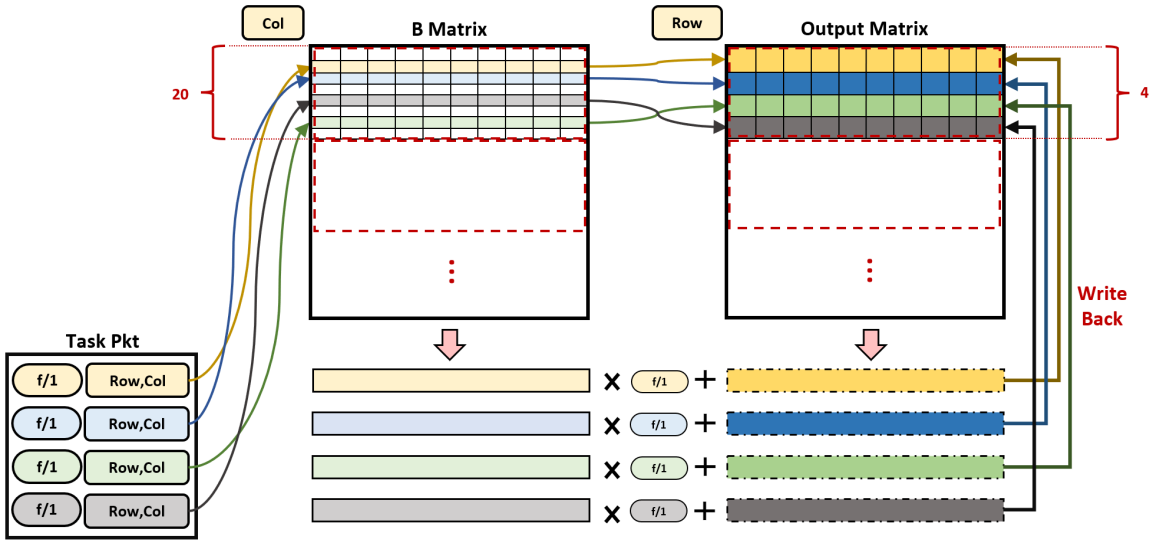


Figure 4·8: Aggregation Function Operation

(shown in Figure 4·6). In FCsN we distribute the aggregation operations by partitioning the sparse matrix with each node containing a portion of it. The size of the weight matrix is reasonable to be duplicated and stored with on-chip memory.

To facilitate streaming compute kernels with the data-centric model, the sparse adjacency matrix is decomposed into tasks traversing in the network acting on distributed weight matrix in the system. Task packet headers include the aggregation operations' task id and the data range that the task works with. The payload of the task packet consists of non-zero elements of the adjacency matrix. As Boolean sparse matrix (values are 0 or 1), the payload contains the row id and the column

id of the non-zero element (As Figure 4·7(a)). For a non-Boolean sparse matrix, the payload consists of the coordinates of the non-zero elements and their values (As Figure 4·7(b)).

Figure 4·8 illustrates the operation of aggregation tasks working with a local distributed weight matrix. Within each non-zero element in the task packet, one row of dense matrix B and one row of the output matrix is fetched. The column id of the non-zero element indicates which row of matrix B is fetched. The Boolean SPGEMM does a vector add on these two vectors and writes to the output matrix updating the value of the same fetched row. The non-Boolean SPGEMM does a scalar vector multiply and a vector add with output matrix row. The result will be written back to the output matrix based on the row id.

Non-conflict Streaming Execution: Our aim of achieving streaming execution of kernel function at line rate requires parallel execution of non-zero elements in each task with non-conflict, non-stall pipeline. However, several issues need to be solved in the aggregation function to achieve streaming pipeline. (1) Reading matrix B or output matrix may conflict between non-zero elements in each packet task. Streaming pipeline at line rate requires non-zero elements of task packets fetching their demanding data in one cycle to avoid pipeline stall. However, there is no latency guarantee of memory access for multiple elements. (2) Writing back to the output matrix may conflict between non-zero elements if more than one element in the task is updating the same row. There is no guarantee of the latency of updating the same row multiple times. A reduction can be inserted before updating the output matrix, but we still can't determine how many levels of reduction is needed. (3) Read after write (RAW) hazard may happen between reading and writing operations on the output matrix among different task packets in the pipeline.

To overcome these issues, task spawner in hardware runtime and BRAM optimiza-

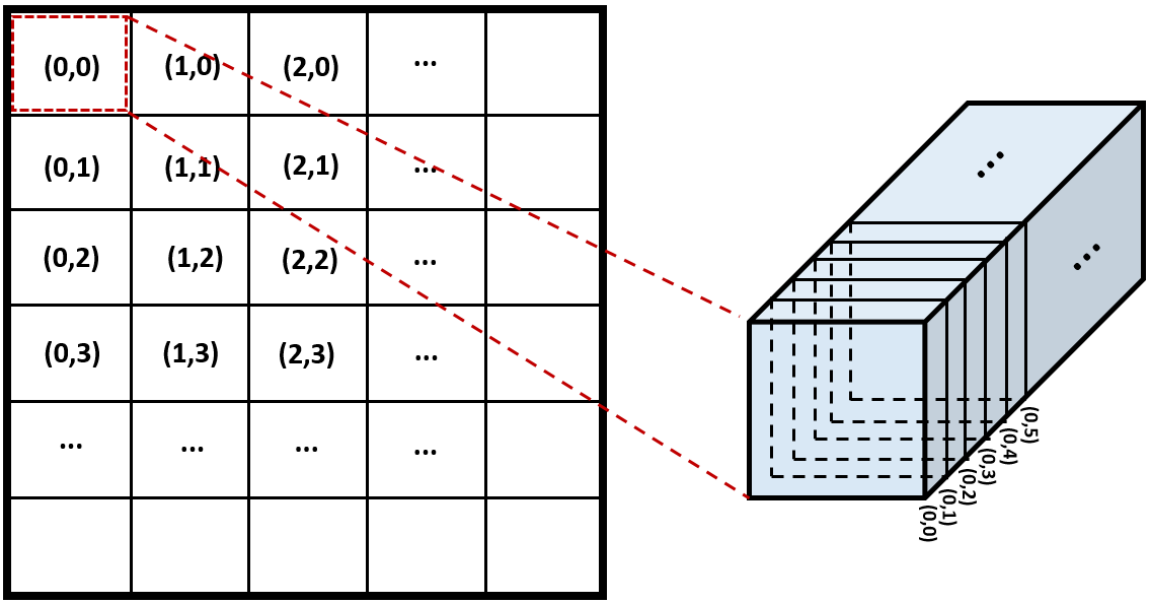


Figure 4·9: Block tensor format

tion in computing units cooperate with each other to achieve non-conflict streaming execution of aggregation function. On-chip DDR access is slow and unpredictable, so we load the dense matrix data to Block RAM to achieve fetching data in one cycle. However, each packet contains several non-zero elements, simply loading B and the output matrix to BRAM can still not fetch several rows of data in one cycle. So we partition BRAM into blocks in an interleave manner and the task spawner selects interleave non-zero elements to avoid memory conflict.

Before generating tasks, the adjacency matrix is tiled into blocks as the storage format for sparse tensors. As Figure 4·9, within each block of tensors, the data locality is improved, and the performance of compute units is enhanced as well. We are tiling the sparse matrix column-wise and with a block size of 20. Tiling the matrix column-wise and with a block size of 20 helps the task spawner generate tasks with simple logic and BRAM access with non-conflict. (Figure 4·10)

Accessing B Matrix: The block size of each tile adjacency matrix is 20 column-wise. With a block size of 20x20 (Red block in Figure 4·10), the non-zero element in

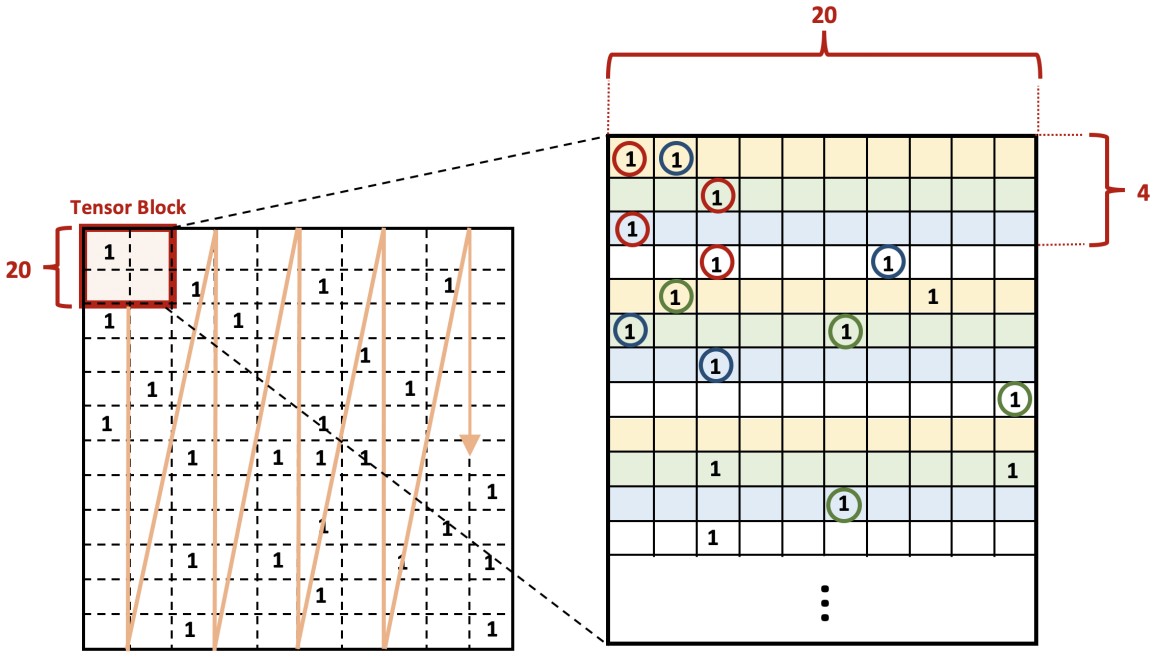


Figure 4·10: Runtime Task Spawner

each column block will only access the first 20 rows of the matrix B. In Vitis HLS, we optimize the BRAM with pragma to partition the BRAM cyclic so that every 20 rows of matrix B can be fetched in parallel. The task spawner will generate tasks with non-zero elements within each block tile column-wise as the orange arrow in Figure 4·10.

Accessing and Updating Output Matrix: After we define the column-wise Block tiling, the non-zero selection needs to ensure non-conflicting output matrix access in each block. The row id of the non-zero element determines which row of the output matrix will be read out and written to. Each packet is confined with a maximum of 4 non-zero elements. (Use 4 elements as an example) So we also partition the output matrix cyclic factor as 4 and within every cyclic four rows of the output matrix can be independently read and written. Task spawner selects non-zero elements based on the remainder of element row id divided by cyclic factor 4 to confirm interleave data access on the output matrix BRAM. (Same color of circle in Figure 4·10 indicates

non-zero elements in the same packet) In this way, the task spawner makes sure each element in the same task does not access the same partitioned region of the output matrix BRAM.

4.4.6 Aggregation Kernel Architecture

When there are valid tasks in the wait queue, the compute unit fetches the task packets from the wait queue, parses the task, and consumes it. Reasoning that the task spawner issues non-zero elements with different partitioned BRAM regions, the compute unit is able to fetch data from BRAM and write back in one cycle with no conflict between 4 non-zero elements in each task. Four independent data paths compute each element of task packets in parallel. To avoid RAW hazards, a RAW detector temporarily holds the last cycle's row id and vector value. If the non-zero element demands the row that is written back, it uses the stored vector in the RAW detector instead of fetching it from the output matrix BRAM.

4.5 Evaluation

4.5.1 Experiment setup

We evaluated the FCsN framework with NN kernel functions and NN applications using an Alveo U280 cluster with 2-4 nodes; systems with 8 and 16 nodes are evaluated with our cycle-accurate simulator with verified results collected in real systems. The FCsN hardware runtime and NN kernels are implemented in Vitis HLS. The baseline CPU results are evaluated with a 16-core 32-thread Intel® Xeon® Gold 6226R CPU. Current state-of-the-art distributed Pytorch-based NN applications use MPI as the backend. To eliminate the communication overhead and variance of different NIC or SmartNIC configurations, we implemented hand-tuned MPI code mapped to the cores of the same CPU. We also added a multi-node GPU with MPI using Nvidia Tesla

Table 4.2: Resource Utilization, Frequency: 294MHz

	BRAM_18K	DSP	FF	LUT
NN Function Kernels	2,754 (66%)	2,724 (30%)	1,182,054 (42%)	834,876 (60%)
Hardware Runtime	717 (15%)	0 (0%)	87,813 (3%)	64,089 (3%)

V100s. The evaluations are with respect to four models: compute-centric (C-C) with MPI on CPU, GPU with MPI, computer-centric (C-C) with FCsN, and data-centric (D-C) with FCsN.

4.5.2 Performance and Resource Utilization

The baseline CPU MPI results evaluate the distributed Pytroch-based NN approach with the compute-centric model. We simplified and optimized the MPI code with the same computation and communication pattern as Pytorch. The multi-GPU result is achieved by using MPI as the message-passing interface. Compute-centric FCsN model follows the same computation and communication methods as the CPU baseline. However, the computation, control logic, and communication are offloaded onto the SmartNIC, with no CPU control involved. This result shows the improvement of the FCsN framework with detached host control. Lastly, FCsN with data-centric model indicates the performance of the data-centric model of streaming kernel execution with tightly fused computation and network pipeline detached from host control.

Figure 4·11 shows the normalized speedups for these evaluation models with NN kernels and NN applications. CPU with MPI and GPU with MPI have lower performance and scalability because of the communication and software overhead. The data-centric model gains more speedup than the compute-centric model as the system size scales up since data movement and communication overhead increase. The FCsN data-centric model shows better speedups and scalability due to the streaming execution of task tokens that minimize data movement, and the avoidance of PCIe, software stack, host control, and synchronization.

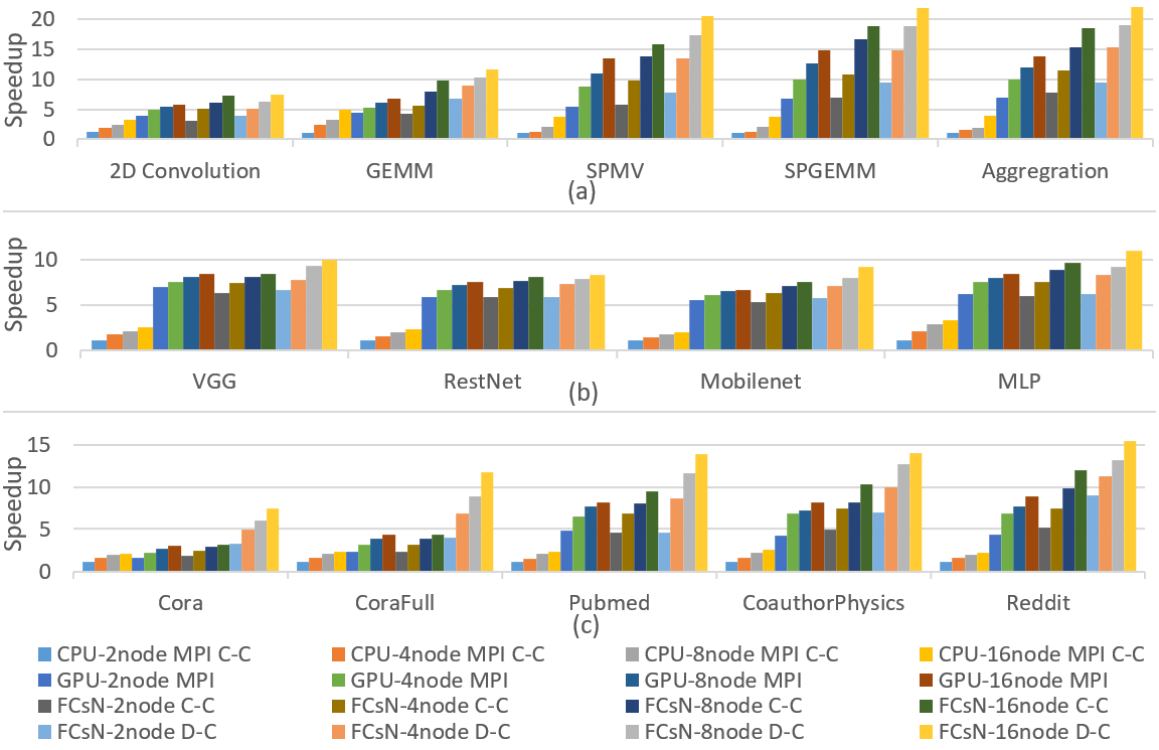


Figure 4·11: NN Kernel and Application Model Evaluation Speedup

Kernel Performance: Figure 4·11(a) shows the speedups of NN kernel functions with matrix size of 2048x2048. Since 2D convolution has less communication in 2D convolution and GEMM has regular data access and computation, FCsN with the data-centric model has limited speedup. However, in irregular kernels like SPMV, SPGEMM, and aggregation, FCsN gains more speedup due to the offloaded control, asynchronous tasks, and streaming computation. In the 2D convolution function kernel, the speedup with FCsN in data-centric model is 2.3× compared with the CPU version, 1.27× compared with GPU and 1.1× compared with FCsN in compute-centric mode.

The distributed GEMM kernel requires data movement between nodes. FCsN has less overhead handling communication than the CPU and GPU approaches. Compute-centric FCsN has a speedup of 2× compared to the CPU baseline. Data-

centric FCsN provides higher performance than CPU, GPU, and FCsN in compute-centric with streaming computation with a speedup of 2.3× over the CPU baseline and 1.7× over the GPU. Sparse matrix-vector multiplication (SPMV), sparse matrix multiplication (SPGEMM) and aggregation have similar data distribution and execution. Computations are decomposed into tasks and vectors or dense matrices are distributed as data in data-centric model. These kernels follow similar speedup trends over the baseline. The average speedup over CPU is 3.8× for compute-centric FCsN and 6.7× for data-centric FCsN.

Application Performance: Figure 4·11(b) shows the performance of neural network applications (VGG, RestNet, Mobilenet, MLP (Simonyan and Zisserman, 2015; He et al., 2016; Howard et al., 2017; Popescu et al., 2009)) using FCsN provided function kernels. The speedups of VGG (2D-convolution), RestNet (2D-convolution), MobileNet (GEMM) and MLP (GEMM) are 3.36×, 3.55×, 4.3×, and 2.9×, respectively, for FCsN compute-centric over the CPU baseline. The speedup of FCsN data-centric over CPU baseline is 4×, 3.6×, 4.6× and 3.3×. We evaluated GNN models using the aggregation kernels with five datasets in Figure 4·11(c), Cora, CoraFull, Pubmed, CoautherPhysics, and Reddit (Sen et al., 2008; Dwivedi et al., 2020). The size of datasets increases in order. The speedups of FCsN compute-centric over CPU are 1.52×, 1.86×, 4.82×, 5.08× and 8.5×. The speedup of FCsN data-centric over baseline are 5.38×, 5.08×, 6.04×, 6.64× and 10.13×.

Resource Utilization: Table 4.2 shows resource utilization of NN function kernels and the FCsN hardware runtime. The BRAMs and LUT resources are used largely to avoid memory access conflicts.

4.6 Summary with Discussion of HW/SW Codesign

In this chapter, we introduce a user-friendly FPGA-Centric SmartNIC framework (FCsN) for Neural Networks, featuring a lightweight distributed hardware runtime and a data-centric programming model that operates independently of CPUs. Utilizing a task circulation execution model, this framework tightly integrates communication and computation, distributing kernel execution in a streaming manner at network line rate. The hardware-based FPGA-centric SmartNIC runtime enables asynchronous and fine-grained task scheduling which allows FCsN to be detached from the host.

Software: On the software side, we identify bottlenecks in GCN inference and generate execution task tokens based on the hardware configuration.

Hardware: On the hardware side, we design the architecture with interleaved memory data storage and non-conflicting streaming kernels.

Codesign: This software-hardware co-design approach ensures conflict-free kernel execution, seamlessly integrated with the network pipeline. FCsN leverages these characteristics to achieve high performance and efficiency for irregular and data-intensive neural network applications.

Chapter 5

Heterogeneous GPU-SmartNIC System: DLRMs

As previously mentioned, large-scale machine learning systems suffer from communication bottlenecks. In the second step of our exploration, we found that while SmartNICs are capable of handling application control and communication, they have limited computational power. FPGA-based SmartNICs alone do not provide sufficient processing power, and GPUs are not well-suited for control tasks, including general communication. In the third exploration step, we explore SmartNICs facilitating connectivity in GPU-centric systems.

Previous work, such as the Zion GPU Cluster by Meta (Mudigere et al., 2022), provides a high-performance hardware platform for training deep learning recommendation models. Zion supports RDMA SmartNICs that improve communication latency, but the system does not fully utilize the capabilities of SmartNICs. Similarly, Microsoft's ZeRO-Offload (Ren et al., 2021b) and ZeRO-Infinity (Rajbhandari et al., 2021) are heterogeneous GPU systems that leverage CPUs and NVMe memory to scale large models on limited resources, but they suffer from low efficiency and performance.

In this chapter, we address these challenges by proposing a software and hardware co-design system that fully leverages SmartNIC capabilities. This system integrates computation and communication, using GPUs as accelerators to maximize efficiency and performance. In this exploration, we are targeting to overcome the three critical

challenges of GPU-based clusters with deep learning recommendation model as an application.

5.1 Motivation

Personalized Recommendation systems (Resys) are one of the critical applications deployed for vital services in technology platforms. Applications and online services like product recommendations, video and music recommendations, news feeds, and search services are supported by recommendation systems. (Gupta et al., 2020; Hazelwood et al., 2018; Naumov et al., 2020; Park et al., 2018; Covington et al., 2016; Gomez-Uribe and Hunt, 2016; Smith and Linden, 2017) As recommendation prediction requirements and datasets grow, deep learning recommendation models (Naumov et al., 2019) have shown substantial advantages and provide ranking and click-through rate (CTR) prediction used by major companies that provide online services.

DLRM's model size is significantly larger than traditional deep neural networks due to its data-intensive embedding operators that take hundreds of Gigabytes or even Terabytes of capacity. The model size surpasses the on-chip memory of accelerators and the growth of the accelerator's HBM cannot keep up with the ever-growing DLRM size, shown in Figure 5·4 (Sethi et al., 2022; Mudigere et al., 2022). Therefore, distributed DLRM inference and training require large-scale multi-node systems. As a result, the scalability issue caused by the communication bottleneck mainly hinders the development of DLRM.

DLRMs bring three challenges to large-scale distributed systems: (1) *Communication bottleneck.* Parallel strategy like data parallelism is unfeasible for DLRM because the replications of the model are too large to fit into the accelerator's on-chip memory. The combination of model and pipeline parallel is used to satisfy the data-dependent

behavior of the embedding operator (Figure 5·1). The embedding operators are partitioned and distributed to each node and use all-to-all communication exchanging each share of embedding tables (EMT). The all-to-all communication incurs a massive amount of data exchange resulting in communication as the bottleneck of the entire application. With the development of the DLRM model, the model size is growing even larger to improve the prediction quality with the system adding more nodes. The exponential growth of communication volume by all-to-all communication makes this critical path become even worse. (2) *Memory bandwidth challenge.* DLRMs contain up to trillions of parameters and consume up to terabytes of memory capacities. The embedding operators require high memory bandwidth and frequent access to embedding tables. Meanwhile, the embedding operators also lead to significant runtime overhead from launching thousands of CUDA kernels. (3) *Computation efficiency challenge.* Compared with other machine learning models, DLRM exhibits lower arithmetic intensity with irregular computations like data reshaping, flattening, and transposing. These irregular operations are primarily memory copies and make GPU's computation less efficient with large hardware resource consumption.

Advanced network interface cards known as SmartNICs have emerged to mitigate network communication in scaled-out data centers. Moreover, SmartNIC with computation capabilities is particularly useful for domain-specific computation such as machine learning and streaming data analytics (Xilinx, 2020; Xilinx, 2022b; Intel, 2021; Intel, 2022; Broadcom, 2019; Nvidia, 2021). As SmartNICs continue to advance in power, they offer a significant opportunity to act as heterogeneous communication and computation coupled devices in distributed systems that overcome the communication bottleneck of DLRM's scalability.

Even though SmartNICs present significant potential for improving DLRM's communication performance, simply adding SmartNIC to the distributed system only mit-

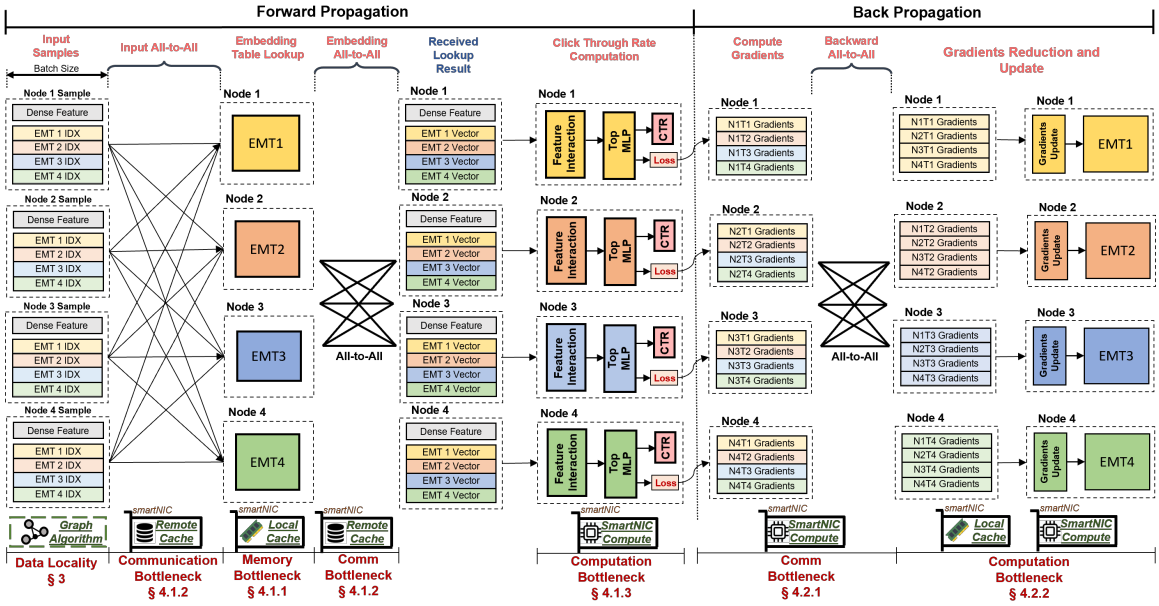


Figure 5·1: Deep Learning Recommendation Model Workflow Overview (EMT: Embedding Table, MLP: Multilayer Perceptron, CTR: Click Through Rate(Prediction), N1T1 Gradients: Computed loss gradients of Embedding Table 1 from Node 1) The heterogeneous SmartNIC system targets the memory bottleneck, communication bottleneck, computation bottleneck of forward propagation and backward propagation (Section 4). A graph algorithm improves the data locality of batches (Section 3).

igates the point-to-point communication latency. Currently, there isn't a distributed system design that fully leverages the abundant SmartNIC resources to overcome DLRM's communication bottleneck, mitigating memory bandwidth pressure and improving computation efficiency.

Other existing work (Yang et al., 2020b; Shi et al., 2020a; Sethi et al., 2022) uses software solutions targeting the communication bottleneck by reducing the embedding table size or the communication volume of all-to-all and all-reduce collection. However, these approaches have limited performance improvement, and the software solution can not fundamentally resolve the performance bottleneck. There are also works (Eisenman et al., 2019; Wilkening et al., 2021) that introduce storage technologies to optimize the embedding operator's performance. However the memory

DLRM Challenge	Cache System		SmartNIC Computation	Graph Algorithm
	Remote Cache	Local Cache		
Communication	**+++**		**++**	+
Memory	++	++		+
Computation			+	+

Figure 5·2: SmartNIC Design and DLRM Challenges

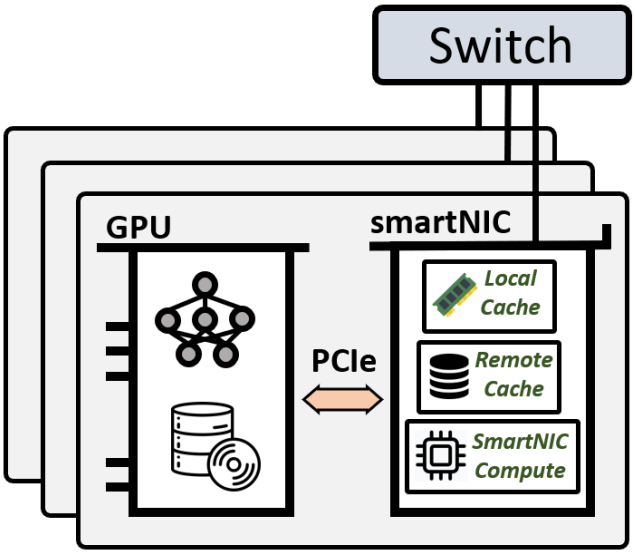


Figure 5·3: Heterogeneous SmartNIC System Overview

bandwidth and latency can't catch up with GPU's HBM, and as the model size grows, memory bandwidth could become another critical bottleneck. Besides, the current GPU cluster (Sergeev and Balso, 2018; Abadi et al., 2016; Mudigere et al., 2022) for DLRM suffers from large volumes and frequent communication bottlenecks.

In this chapter, we introduce a software-hardware co-design of heterogeneous SmartNIC system for scalable DLRM inference and training that overcomes communication bottlenecks, mitigates memory bandwidth pressure, and improves computation efficiency. A set of SmartNIC designs of cache systems (including local

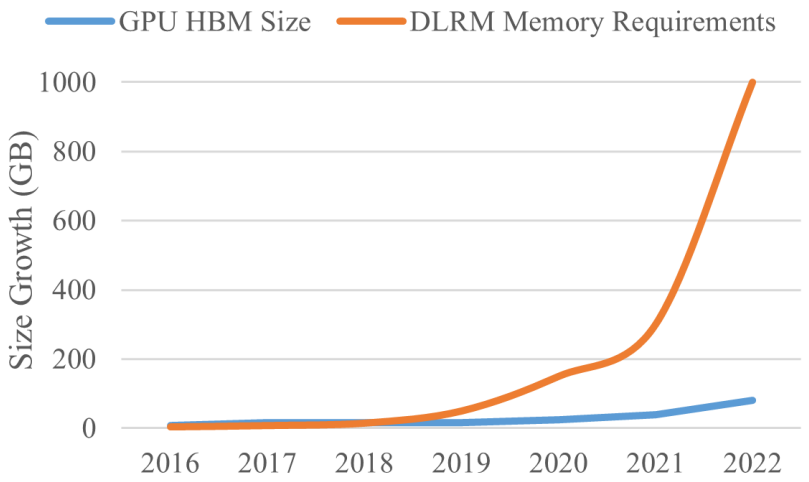


Figure 5·4: DLRM memory capacity requirements and GPU HBM growth

cache and remote cache) and SmartNIC computation kernels exploits the locality of DLRM to reduce data movement, relieve memory access intensity, and improve GPU's computation efficiency. Figure 5·2 indicates the techniques used.

(1) Remote Cache. The large volume and intense all-to-all communication primarily contribute to the communication bottleneck of distributed DLRM systems. The remote cache on SmartNIC buffers frequently used remote embedding lookup results, reducing communication workloads and alleviating both networking and memory bandwidth pressure.

(2) Local Cache. The local cache on SmartNIC caches the popular local embedding tables allowing direct retrieval of embedding lookup results from the SmartNIC, instead of interrupting the GPU. This vastly reduces the memory bandwidth burden on the GPU's HBM, improving overall node memory bandwidth.

(3) SmartNIC computation. The SmartNIC's irregular computation kernels complement the system node's computation resources, improving GPU efficiency by offloading irregular computations and minimizing GPU's kernel calling overhead and hardware usage. Additionally, the computation kernels reduce gradient loss updates

in backward propagation and decrease the workload of backward all-to-all communication. On top, we also introduce a **graph algorithm** that enhances the data locality of DLRM batches by clustering similar samples. More data reuse reduces embedding lookup requests and communication volume, increases cache hit rate, and eases system memory bandwidth pressure. High data locality batches also benefit GPU computation efficiency.

This set of techniques can work together to produce a synergistic effect, resulting in an outcome greater than the sum of their individual contributions.

To summarize, our contributions to this exploration are as follows:

- We propose a scalable software-hardware co-design heterogeneous SmartNIC system for both forward propagation and backward propagation of DLRMs.
- Our system introduces a set of techniques for SmartNIC design that overcomes the communication bottleneck of distributed DLRMs mitigates the memory bandwidth pressure and improves computation efficiency. A graph algorithm improves the data locality of batches and optimizes the overall system performance with high data reuse.
- The evaluation results show that our heterogeneous SmartNIC system can achieve 2.1 × latency speedup for inference and 1.6 × throughput speedup for training.

5.2 Background

5.2.1 Deep Learning Recommendation Model

Figure 5·5 shows the overview of the deep learning recommendation system (DLRM). DLRM consists of two types of inputs, dense feature, and sparse feature, and predicts the probability that a user would interact with an item which is referred to as the

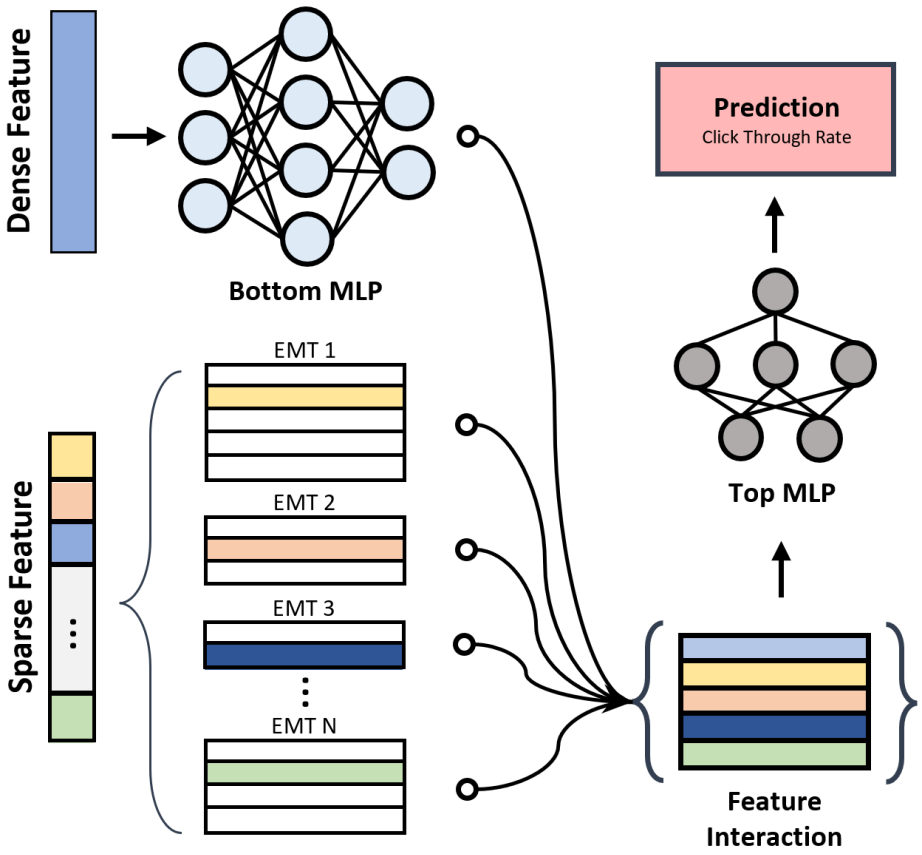


Figure 5·5: Deep learning recommendation models overview

Click Through Rate (CTR). The dense feature contains continuous data like the user's age or the current time. Sparse features are categorical features such as posts, pages, or an item. The categorical features contain up to thousands of categorical features that contribute to an embedding operator (or an embedding table). The categorical features are mapped to an embedding vector from its corresponding embedding table (EMT). Sparse features use a group of embedding table indexes to fetch the sparse embedding vectors. Afterward, feature interaction combines the output of the bottom MLP and feeds it into the top MLP for CTR prediction.

5.2.2 Distributed DLRM System Challenges

The embedding operators are partitioned and distributed to the system for its large size. Samples as a batch unit are required to access each part of different embedding parameters causing data-dependent behavior of the embedding operators. This results in a combination of model and pipeline parallelism for the distributed DLRMs shown in Figure 5·1. In forward propagation, input all-to-all, and embedding all-to-all exchanges each portion of embedding parameters with samples within each node. Click-through rate computation waits until all embedding operators are gathered for each sample. These all-to-alls contribute to the major bottleneck of DLRM's inference. In backward propagation, trained embeddings are required to synchronize before the next batch iteration starts. Another backward all-to-all is used to update the trained embeddings for each embedding table. These total three all-to-alls are the throughput performance killer of DLRM's training. As the prediction demand rises, embedding operators, grow even bigger, and more nodes are added to meet the increasing query requests. Scalability is an urgent issue that hinders the development of DLRMs.

SmartNICs have emerged in distributed data centers recently with abundant communication and computation capability. The SmartNICs have developed with ARM core, FPGA, domain-specific accelerators, and on-chip memories (Xilinx, 2020; Xilinx, 2022b; Intel, 2021; Intel, 2022; Broadcom, 2019; Nvidia, 2021). Besides accelerating packet processing, SmartNIC creates an opportunity of offloading application workloads. However, adding SmartNICs to the system only improves limited point-to-point communication performance. It is challenging to fully leverage the SmartNIC communication and computation coupled capability for applications in the system.

The left side of Figure 5·1 shows the workflow of forward propagation. The embedding operator of DLRM is too large to fit into the accelerator's memory. It is

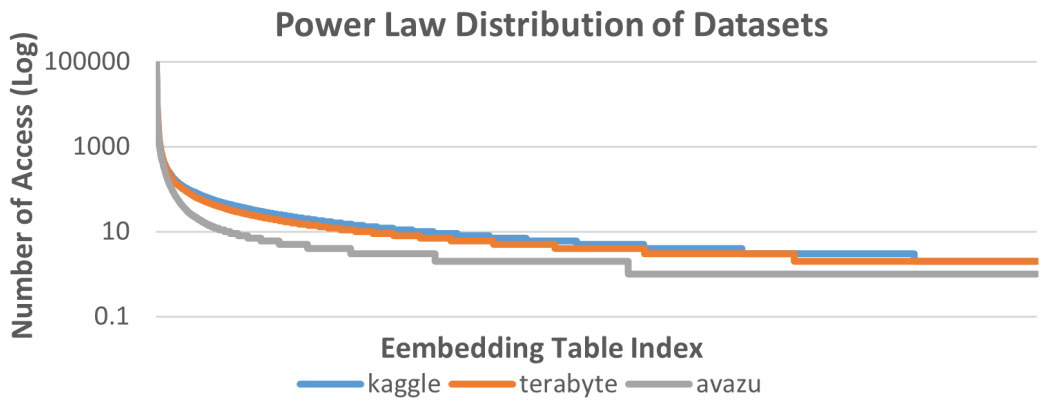


Figure 5·6: Power Law Distribution of Datasets.

unrealistic to duplicate the entire model on each node. The embedding layer is partitioned and distributed to each node to exploit model parallelism. The MLP layer is duplicated in each node for data parallelism. All-to-all communication is required to connect the output of the embedding layer and the input of the MLP layer. The right side of Figure 5·1 shows the workflow of backward propagation. After forward propagation, losses of the embedding vector from each sample are sent to the embedding table destination through backward all-to-all communication. Each embedding table gathers updated embedding gradients from the system for the final embedding layer output of the current training batch. DLRM training is throughput-limited, and the goal is to train as many samples as possible.

5.3 Characteristics of DLRM Data Power Law Distribution

The frequency distribution of a categorical feature's categorical embedding follows a power law distribution shown in Figure 5·6. A small fraction of embeddings takes over most of the embedding access. It is typically the case that a large portion of the users are interested in a small portion of popular items. (Pages, Movies, and best-sell products) This characteristic leads to data reuse opportunities for popular embedding vectors and architecture to exploit the locality to overcome DLRM's communication

bottleneck.

5.4 Graph Algorithm

This section discusses our graph algorithm that enhances the data locality within batches of queries.

5.4.1 Graph Mini Batch

DLRM uses batch as a processing unit to use the parallelism of hardware resources and enable high throughput. Although the sample's sparse features follow power law distribution, batches of forward and backward propagation are formed by fetching samples in a sequential manner with no relations between each other. Space for improvement of enhancing the data locality within the batch to improve the overall system performance further. Batches with better data locality can enhance every phase of DLRM with less embedding lookup, memory bandwidth pressure, communication workloads, and higher computation efficiency.

Instead of fetching samples sequentially from the dataset, we inserted the graph algorithm to generate a mini-batch of samples with high data locality. A reasonably larger size of samples is pre-loaded for the graph batch to select a group of closely related samples for mini-batch. As Figure 5·7, the blue table indicates a batch of samples that the graph algorithm applies to. The table can be viewed as a lookup incidence matrix with rows of samples and columns of the lookup index.

Based on the lookup incidence matrix, a hypergraph is formed where samples are nodes, and embedding table lookup indexes are edges shown as the middle graph in Figure 5·7. Sample nodes are connected to index edges corresponding to the lookup table. The hyperedges with higher degrees are formed because DLRM sparse features follow power law distribution with a higher chance of multiple samples lookup the same popular indexes. As the lookup table indicates, embedding table 0 with index

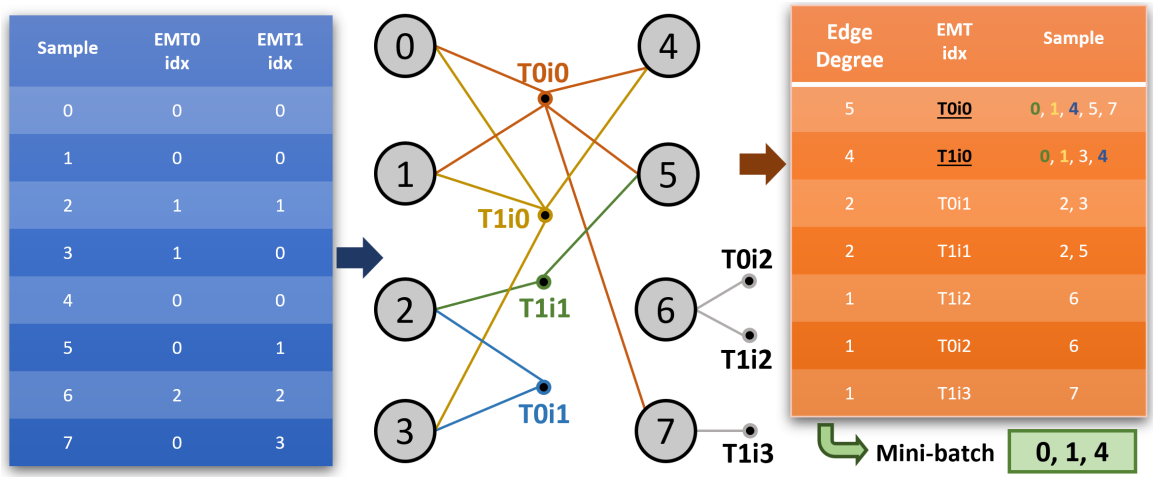


Figure 5·7: Graph Algorithm (T0i0: Embedding Table (EMT) 0, index 0) Left blue table indicates a sample batch that can be viewed as an incidence matrix. The orange table indicates the scoreboard ranking popularity of edges in the hypergraph. A mini-batch of samples with high similarity is generated as input of forward propagation.

0 (edge T0i0) is requested by sample nodes 0, 1, 4, 5, 7, and table 1 index 0 (edge T1i0) is requested by sample nodes 0, 1, 3, 4. The hypergraph is generated as each of the embedding tables' indexes are iterated. Within the graph, the edge with high degrees means more data reuse as a single embedding lookup can serve more sample nodes. Sample nodes that share more overlapped edges have more similarities with better data locality. Based on these two characteristics, the graph algorithm filters samples with similarity and more overlapped popular embedding lookup index. Graph algorithm saves lookup requests, and mitigates all-to-all communication workloads and memory bandwidth pressure.

Algorithm 1 indicates the simplified graph algorithm workflow. The embedding lookup incidence matrix is generated along with pre-loading the batch of samples from the dataset with a counter attached to each embedding index to register the degree of index edges. After loading the batch samples, we do hyperedge degree sorting using the edge degree counter. The table on the right indicates the popularity of the EMT

Algorithm 1 Graph algorithm

```
Mini_Batch[mini_batch_size]
Incidence_Matrix[batch_size][EMT_idx]
Hyperedge_Cntr[batch_size]
Hyperedge_Threshold = n

# Feed samples from the dataset to the incidence matrix
while len(Incidence_Matrix)<batch_size do
    Incidence_Matrix ← sample
    Update Hyperedge_Cntr[EMT_idx]
end while

# Get similar samples
Sample_List[batch_size] # Initiated as a priority queue
for hyper_edge in
sorted(Hyperedge_Cntr[0:Hyperedge_Threshold]) do
    for sample_id in Hyperedge_Sample[hyper_edge] do
        Sample_List ← sample_id
    end for
end for
Mini_Batch ← Sample_List[0:mini_batch_size]

# Delete mini_batch samples
for Sample_id in Mini_Batch do
    Update Hyperedge_Cntr, Incidence_Matrix
end for

return Mini_Batch
```

index from top to bottom. Embedding table 0 index 0 (T0i0) is the most popular index with sample nodes 0, 1, 4, 5, and 7 connected with high similarity between each other. However, in the real-world case, if the mini-batch only picks samples based on the top edges' degrees, samples in the mini-batch could not be closely related as each sample has more sparse features. So we introduce the hyperedge threshold that picks the top n popular embedding table index and do a node sorting that picks sample nodes that appear most often in these indexes. Sample list a priority queue to keep updated as the number of appeared samples in descending order. As traversing the edges, the priority queue stores sample nodes from the most similar to the least. Mini batch fetches the first mini-batch size of samples to feed into the forward propagation pipeline. As Figure 5·7 indicates, edge T0i0 and edge T1i0 are selected as popular indexes. Within these two indexes, sample nodes 0, 1, and 4 appeared most with a high similarity between each other. The mini-batch fetches these three nodes with a mini-batch size of 3. The mini-batch samples will be deleted from the lookup incidence matrix, hyperedge counter will be updated for the next batch. New samples will be fetched from the dataset and forwarded to the next iteration.

5.4.2 Refresh Batch

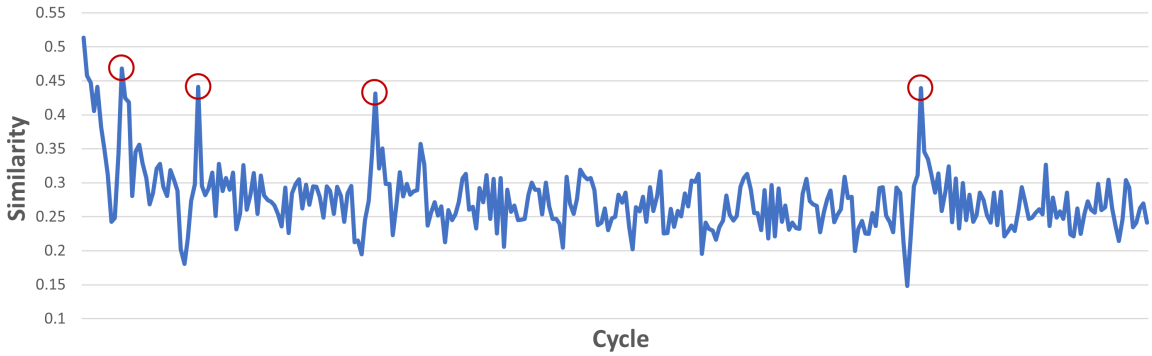


Figure 5·8: Similarity of Samples within Mini Batches. The red circle indicates a refreshing batch with new samples.

As in the previous section, the graph algorithm will always find the most similar samples within batches. So as iterations of the graph algorithm go, samples that have less common sparse features will be always left in the batch. These samples will not be picked by the graph algorithm with fewer data reuse chances. Besides these, the inference of DLRM also has latency requirements. Samples that are left too long in the batch will not be served causing users to never hear back from requests.

To address these issues, we introduce the batch refreshing mechanism. The graph algorithm will select samples within the batch one by one until it is empty and refile the batch from the dataset when one of the two criteria has been reached. Firstly, we use a downgrading factor to trace the similarity of the mini-batch selected by the graph algorithm. As the downgrade factor reaches a threshold that means the similarity of the mini-batch is too low to form a high data reuse opportunity. The refresh batch will be triggered to avoid the continuation of worse data reuse within the mini-batch. Secondly, a timing counter is attached to each sample. When any of the samples has waited too long to serve, refresh batch will be triggered to consume the batch and refill new samples. As Figure 5·8, batch similarities are dropping as the timeline goes. The red circle means the refresh batch mechanism is triggered to fresh the samples within the batch and lift the performance of the graph algorithm by finding similar mini-batches.

5.5 System Hardware Architecture

The scalability of DLRM is the central issue that hinders the recommendation system's development as the model size grows even more significant to provide more accurate predictions. The main bottlenecks that impact the scalability of DLRM are memory, communication, and computation. In this section, we introduce three aspects of design on SmartNIC for both forward and backward propagation: local

cache, remote cache, and SmartNIC computation kernel, which address those three bottlenecks.

5.5.1 Forward Propagation

Local Cache: DLRM has a high demand on memory bandwidth because the embedding operators request frequent and high-volume embedding table lookups. The embedding tables can also consume up to terabytes of memory. If the system uses the GPU as an accelerator, the memory bandwidth and capacity requirement exceed GPU's on-chip memory capability. In order to handle the embedding layer, multiple GPUs are aggregated to meet the memory requirement introduced by the large-size embedding tables. Embedding tables are distributed to each GPU's HBM memory. As in the previous section, each node receives embedding lookup from every other node. As the scale of the system grows, and the size of the DLRM embedding layer grows, embedding lookup requests increase which lays more burden on the memory. The memory could be the bottleneck of the system's scalability. Large amounts of embedding lookup can also introduce significant memory access overheads. For large-scale GPU-based clusters, nodes are connected through network interface cards. Embedding lookup requests are sent through the network, received by NICs, and sent to GPU through PCIe. As more frequent requests, the overhead of PCIe and GPU memory access could also aggravate the embedding operator's performance.

We introduce local cache on SmartNICs to address the memory bottleneck of DLRM. The local cache can relieve the memory pressure of the GPU, and reduce the frequent lookup overhead of PCIe and GPU memory access. Embedding operator requests are first received by the SmartNIC and sent to GPU for lookup. Local cache on SmartNIC buffers the lookup result from GPU. So before generating the lookup index list, the lookup index checks if it is hit on the local cache on SmartNIC. If yes, there is no need to send it to GPU, embedding vectors are ready on SmartNIC. After

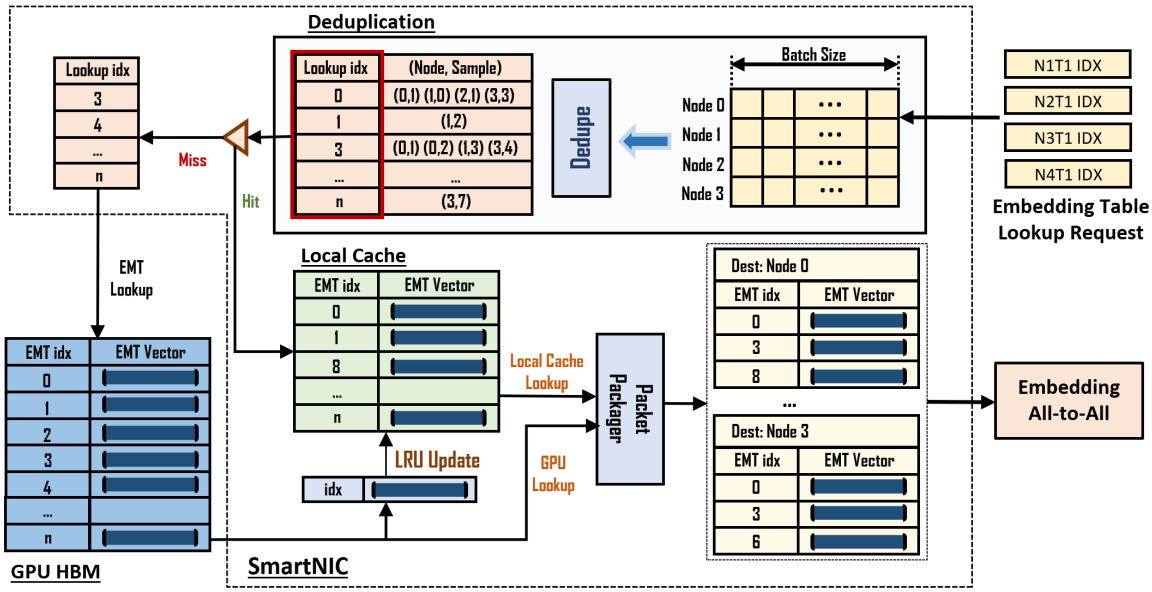


Figure 5·9: Local cache on SmartNIC buffers popular embeddings. Hitting on local cache saved lookup requests to GPU's HBM. Dedupe kernel eliminates redundant duplicated index lookup.

the GPU's memory lookup, the local cache updates its buffer result using the least recently used (LRU) policy. It is very likely that lookup indexes are hit on the local cache because it buffers the most popular index of the embedding table. With the introduction of local cache, less GPU interruption and less GPU memory pressure mitigate the memory bottleneck of the embedding operator.

Before the input all-to-all phase, each node in the system sends the corresponding embedding lookup based on the distribution of the embedding table. After all-to-all, each node receives the lookup requests from other nodes with batches of lookup indexes. As the power law distribution nature, a large amount of duplicated lookups between nodes and samples take up memory bandwidth. Dedupe kernel on SmartNIC uses a deduplication table to keep the record of the lookup index with its corresponding source node and sample ids. The list of lookup indexes is checked if it is hit on the local cache.

Afterward, a list of lookup indexes is summarized from the table and will be used

to issue a lookup request to the GPU's memory. If hit, the lookup result will be sent to the packet packager for preparing the lookup result packets and updating the local cache based on LRU. If missed, indexes are temporarily saved to the GPU lookup request table. As deduplication finishes, lists of lookups will be sent to GPU. When GPU sends the lookup result back to SmartNIC, the lookup vector will be inserted into the local cache on SmartNIC updating cache, and the result will be formatted by SmartNIC using the deduplication table to check the belonging source node and sample index. As every lookup index from the dedupe table finishes, the embedding all-to-all phase exchanges the lookup result between each node.

Remote Cache: The size of DLRM is significantly large because of the embedding operator which can take up to terabytes of memory capacity. The size is growing even more significantly with more embeddings providing more accurate predictions. It is impossible to do data parallelism that replicates the entire model on a single device's memory. Model and pipeline parallel are introduced using large-scale system and distributing the embedding tables to every node's memory. This large system forms a large pool of memory capacity and shares the memory capacity and lookup bandwidth. As Figure 5·1, all-to-all communication is needed for each node's queries gathering all distributed embedding tables' lookup results. All-to-all communication pattern introduces enormous workload pressure on the system's network and will grow exponentially as each node's batch grows. With the development of the DLRM model growing even more extensive, data exchange communication would become the performance killer of the overall system.

In forward propagation, there are two all-to-all communications. The input all-to-all requests the embedding table lookup for each node's sample queries' sparse features. The embedding all-to-all returns embedding lookup results to each of the sample's original source nodes. The computation kernels wait until the all-to-all phase

is finished to move forward and compute the prediction results.

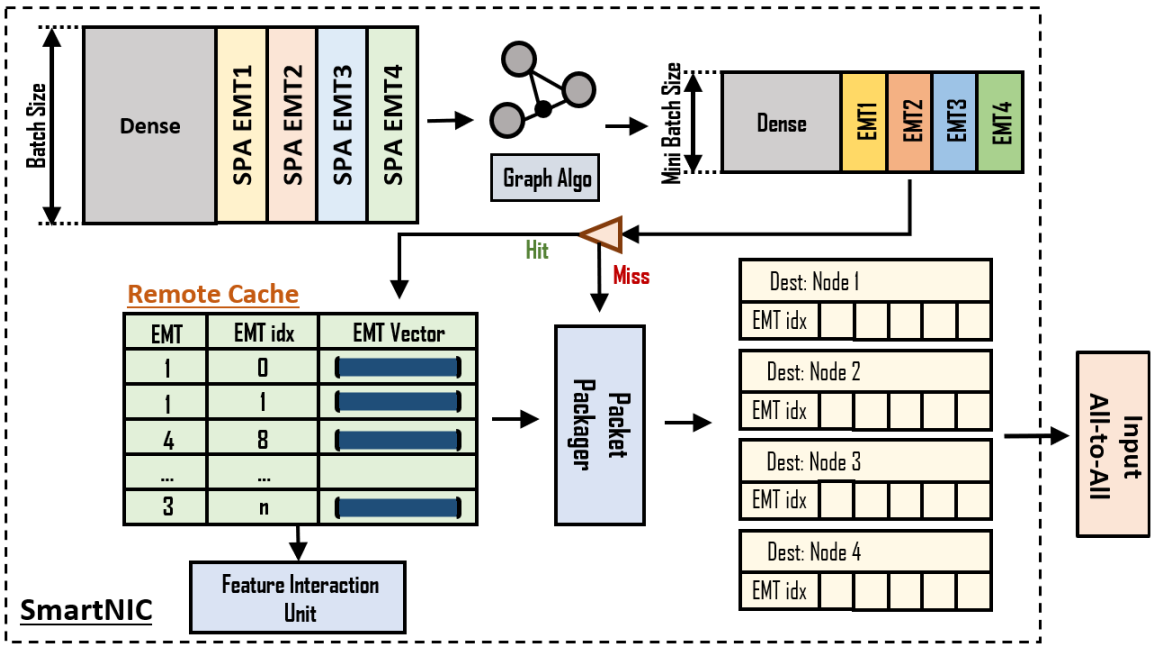


Figure 5·10: Remote cache on SmartNIC buffers remote embedding tables. Queries check if remote embedding is hit on the remote cache before issuing the remote lookup request.

We introduce the remote cache on SmartNIC to address the communication bottleneck by reducing the all-to-all communication workloads. After embedding all-to-all, remote embedding table lookup results are sent back to the original source node. The results are buffered by the remote cache on SmartNIC. The remote cache uses the least recently used (LRU) policy to update the storing vectors as iteration goes on. As the system warms up, the remote cache on SmartNIC stores the most popular remote embedding table's vectors. Figure 5·10 indicates the workflow of how sample queries make use of the remote cache. After the graph algorithm generates a mini-batch of query samples, samples first check if the lookup is hit on the remote cache. If it hits, this lookup request is not needed and can directly get the remote embedding results from the remote cache. The result will be held until embedding all-to-all finishes. Since DLRM samples follow power law distribution, popular remote lookup has a

higher chance will be requested in the future. The remote cache that buffers the popular embeddings reduces the unnecessary duplicated remote embedding lookup request. If the lookup is missed on the remote cache, it will be forwarded to the packet packager for remote lookup.

SmartNIC Computation: DLRM generally exhibits lower arithmetic intensity compared to the traditional DNN model. DLRM consists of various computations including matrix multiplication, no-linear activation, and irregular computation like data reshape matrix flattening, and matrix transposing. These irregular computations can not use GPU's hardware resources efficiently as GPUs are good at massive parallel scalar computations. The irregular compute introduces frequent memory copy that takes enormous amounts of hardware resources and on-chip memory and can not use the GEMM operator on the GPU. Kernel calling overheads are another factor that largely hinders the computation efficiency of accelerators.

We introduce computation kernel and data management kernel (shown in Figure 5·11) on SmartNIC to minimize the overhead of GPU and improve the GPU's computation efficiency. After phase embedding all-to-all, each node receives the embedding lookup result from the remote node and is ready to send the result to GPU for prediction calculation. Instead of dumping the unmanaged raw data to GPU, feature interaction which includes irregular computation operations are offloaded to SmartNIC with dedicated hardware for higher computation efficiency. The raw data are reorganized by the kernel on SmartNIC before forwarding to GPU. The computation kernel combines the dense feature and sparse features as a matrix. The matrix is transposed as the input to feature interaction using matrix multiplication kernel. After feature interaction, the result is flattened to convert the 2D matrix into a 1D vector which is the input to the top MLP. These dedicated irregular kernels improve the computation efficiency and save hardware resources for GPU.

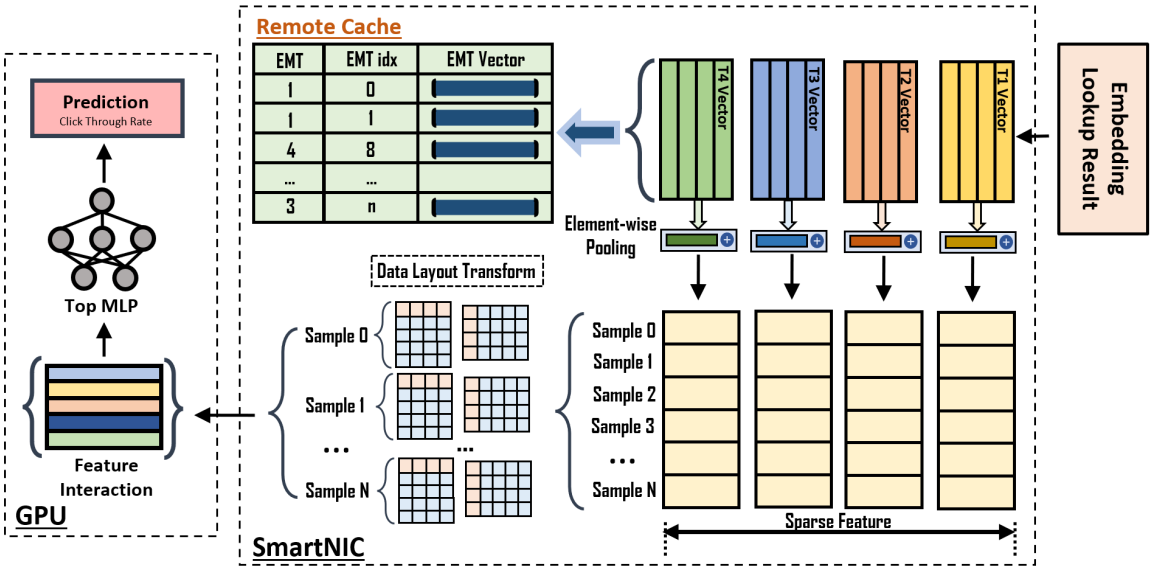


Figure 5·11: SmartNIC computation with irregular computation Kernels including data layout transform, matrix transposing, matrix flattening, element-wise pooling. Remote Cache is updated with received remote embeddings after embedding all-to-all.

5.5.2 Backward Propagation

DLRM training workflow differs from the inference that synchronization is needed between batches of training samples. In the backward propagation, kernels for training are supported in the training core of SmartNIC. Embedding losses are calculated after forward propagation and used to update the embedding table's value. Backward all-to-all gathers every node's partial embedding losses and collectively updates the distributed embedding value. The next batch of training samples will be issued when all embedding tables have been updated. The remote cache would not be effective in training because every node's value is partially updated embedding value. The updated embeddings are located in the embedding table's original node.

In the forward propagation of the training process, the local cache on SmartNIC reduces the system memory pressure. In backward propagation, the SmartNIC computation kernel improves throughput with two levels of reduction which are local

loss reduction and global loss reduction using the reduction computation kernel on SmartNIC.

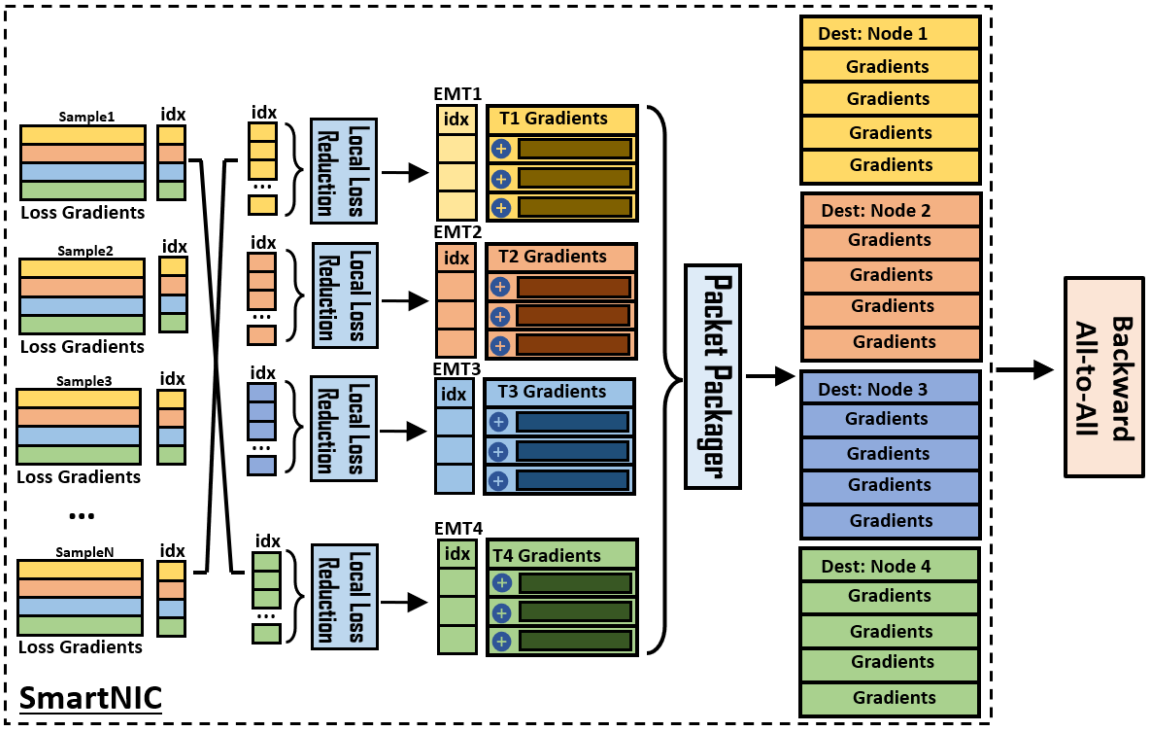


Figure 5·12: Backward propagation local gradient loss reduction using SmartNIC computation

Local Loss Reduction: The embedding losses are generated after forward propagation. After the GPU's loss calculation, losses are sent to SmartNIC for backward all-to-all communication updating the corresponding embedding table. DLRM sample follows power law distribution, it is highly possible that samples within batches on each node update the embedding value. Local loss reduction kernel combines the losses of the same embedding vector and calculates the gradient losses of locally global losses. The packet generator packs updating losses for each embedding table after all samples have been handled and sent to the destination. Local loss reduction kernel reduces the workload of backward all-to-all and saves GPU overhead and resources.

Global Loss Reduction: Each node in the system receives embedding losses

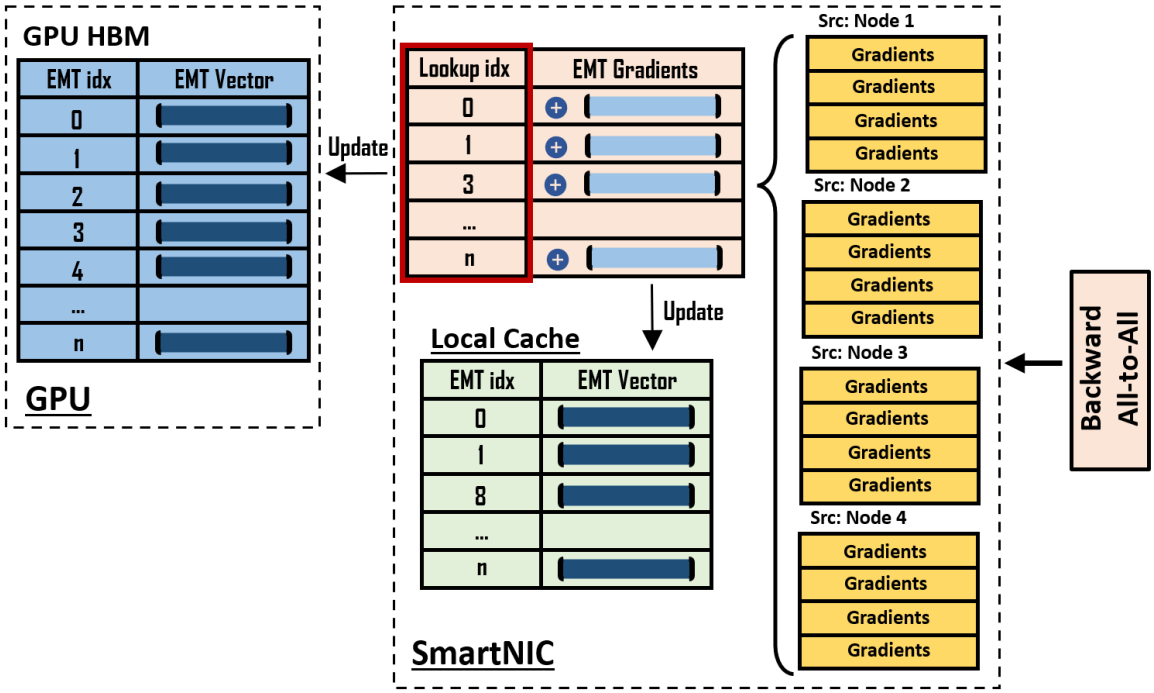


Figure 5·13: Backward propagation global gradient loss reduction using SmartNIC computation. The gradient loss is updated both in the local cache on SmartNIC and GPU's HBM.

from every other node after backward all-to-all. A table keeps updating each embedding vector's losses as losses are received from the network. As every loss has been received, the final loss gradients update the embedding table's value. The local cache is used in the forward propagation process and stores the embedding vector value from the GPU's HBM using the LRU policy. So the local cache guaranteed stores the embedding vector that needs to be updated by the loss gradients. The final loss will be updated on both the local cache on SmartNIC and GPU's HBM. The updated local cache buffers the updated embedding for future batches. With the local cache and reduction kernel on SmartNIC, memory bandwidth pressure is largely saved, and minimize the overhead of PCIe and GPU's kernel interruption.

5.5.3 Design Space Exploration: SmartNIC-Centric Framework

We also did a design space exploration by offloading the whole application onto the SmartNIC. Even though large amounts of PCIe, kernel calls, and software stack overheads are saved, the performance can not catch up with heterogeneous architecture with GPU because of the limitation of hardware resources and computation latency of SmartNICs compared to GPU as an accelerator.

5.6 Evaluation

5.6.1 Experimental Setup

The DLRM model evaluation is based on Meta Research open source code (Naumov et al., 2019). The CPU baseline uses 16-core 32-thread Intel® Xeon® Gold 6226R, and the GPU baseline uses Nvidia RTX8000. To evaluate SmartNIC hardware, we use Xilinx Alveo U280 FPGA with hardware implementation of runtime, cache system, and computation kernels using High Level Synthesis (HLS). For scalability, we used MPI as the backend for CPU and GPU evaluation. Multiple Alveo U280 FPGAs are connected through transceivers as a cluster to evaluate the multi-SmartNIC system.

Since there is no exact heterogeneous SmartNIC-GPU system, we evaluated the real SmartNIC parameters on the muti-node Alveo U280 cluster and plugged it into a simulator based on DLRM open-source code (Naumov et al., 2019) with GPU as an accelerator.

We evaluated our system with three well-established recommendation model datasets: Criteo Kaggle, Criteo Terabyte and Avazu with both inference and training. Table 5.1 shows the parameters of these datasets.

The evaluation figures use the following abbreviations: BL=baseline, GA=graph algorithm, LC=local cache on SmartNIC, RC=remote cache on SmartNIC, SC=SmartNIC computation, FS=fully SmartNIC system.

Table 5.1: DLRM Datasets Parameters

Dataset	Kaggle	Terabyte	Avazu
Dense Feature	13	13	1
Sparse Feature	26	26	21
EMT Rows	33.8M	226M	9.3M
Row Dim	16	64	16

5.6.2 Performance Evaluation

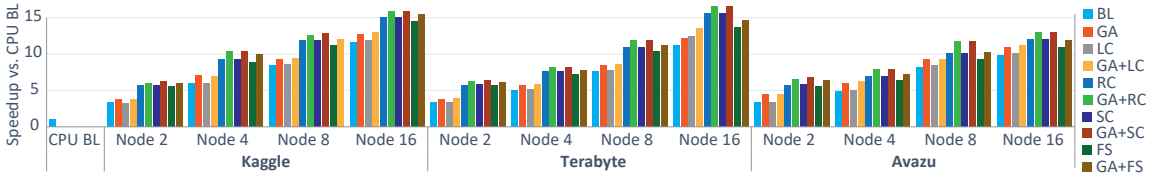


Figure 5·14: Inference Scalability.

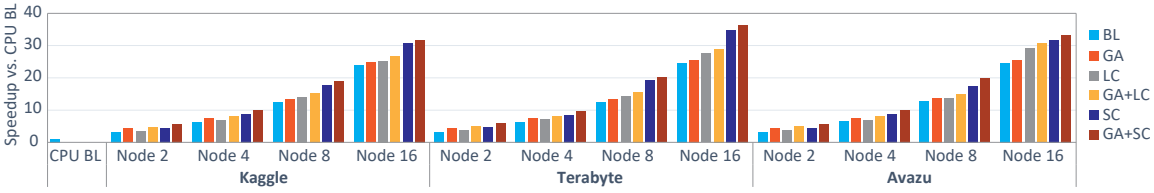


Figure 5·15: Training Scalability.

Graph Algorithm: The graph algorithm improves system performance in inference by 1.2 × on average, with less impact in training. It's more practical and effective in inference due to high and random query requests in real-time, as the algorithm rearranges batch clusters of similar queries, reducing system communication and memory pressure. In training, the dataset samples are given with more straightforward data pre-processing schemes achieving comparable data locality. The graph algorithm demonstrates better performance in inference in our system, making it a practical way to boost inference performance.

Forward Propagation: This section evaluates the system performance with forward propagation using graph algorithm, local cache, remote cache, SmartNIC computation, and fully SmartNIC centric design.

(1) Local Cache: Figure 5·16 illustrates the latency speedup of using local cache on SmartNIC and graph algorithm with respect to three datasets. The forward propagation latency performance is significantly impacted by the local cache on SmartNIC. The speedup remains steady as the size increases. This is mainly because batches of the forward propagation are pipelined. The latency of batches is hidden if the lookup request misses the local cache on SmartNIC and fetches the lookup result from GPU. Using graph algorithm improves the latency by speedup of average 1.2 × of three datasets as it enhances the data locality within batches, reducing remote communication and memory access, thus improving overall latency. Figure 5·18 indicates the Hit/Miss rate of local cache on SmartNIC and GPU's memory. As the size of the local cache increases, fewer memory accesses are needed from the GPU's memory. The local cache relieves the memory bandwidth pressure of the GPU.

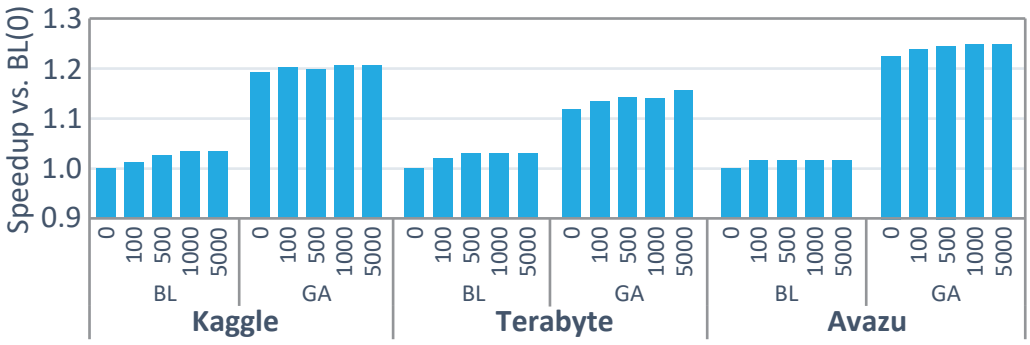


Figure 5·16: Latency Speedup of Forward Propagation Using Local Cache on SmartNIC

(2) Remote Cache: Figure 5·19 shows the latency speedup of forward propagation with Remote Cache on SmartNIC and graph algorithm. Remote cache largely caches the remote embedding lookup because of the power law distribution characteristic. When the query sample hits the remote cache, the remote cache lookup communication request is saved, reducing both queries' latency and the system's communication workload, and improving communication efficiency with less communication conges-

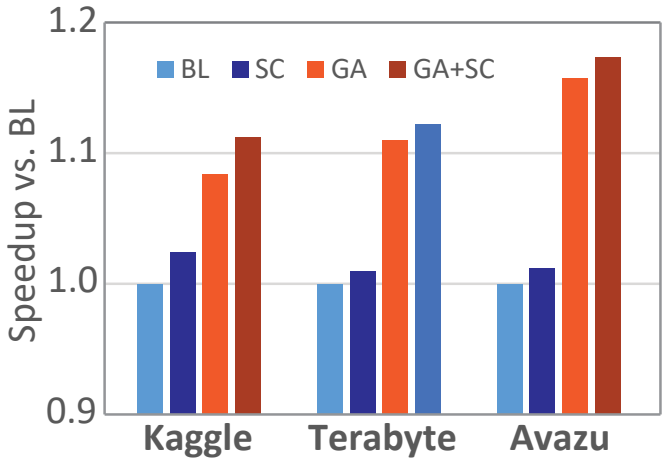


Figure 5·17: Latency Speedup of Forward Propagation using SmartNIC Computation

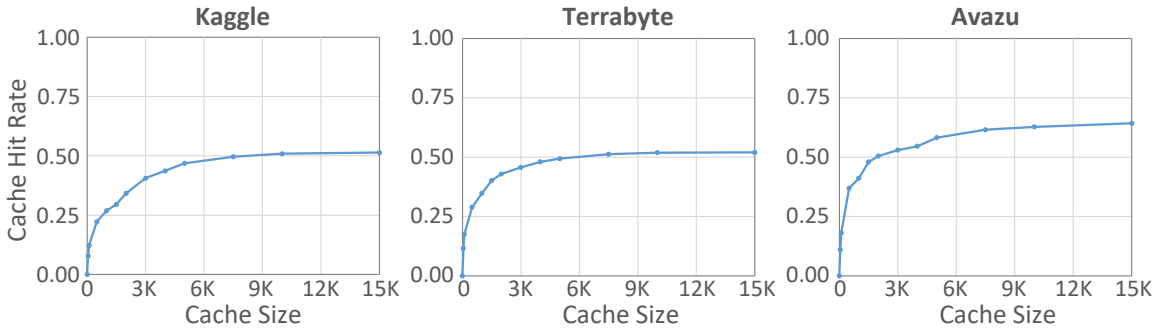


Figure 5·18: Forward Propagation Hit Rate of Local Cache on SmartNIC. Missing on local cache indicates issuing embedding lookup on GPU's HBM.

tion. Figure 5·20 shows saved remote lookup requests using remote cache and graph algorithm. As the remote cache size increases, more lookups are saved. Graph algorithm clusters similar samples to reuse lookups more effectively. Figure 5·21 shows the decline of the local cache hit rate as remote cache size increases. This is because embedding table accesses follow power law distribution. A larger remote cache caches more popular embedding indices, causing less popular embeddings to remain in GPU's HBM. Queries for remote embeddings are more likely to request these unpopular embeddings.

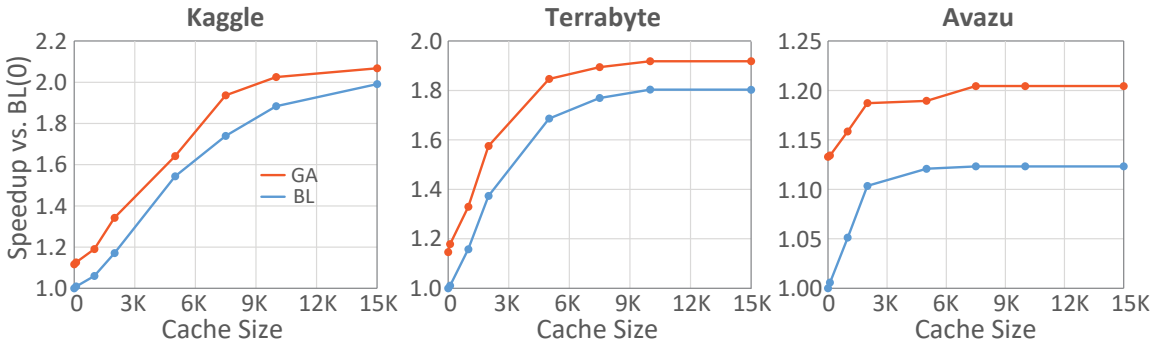


Figure 5·19: Latency Speedup of Forward Propagation Using Remote Cache on SmartNIC

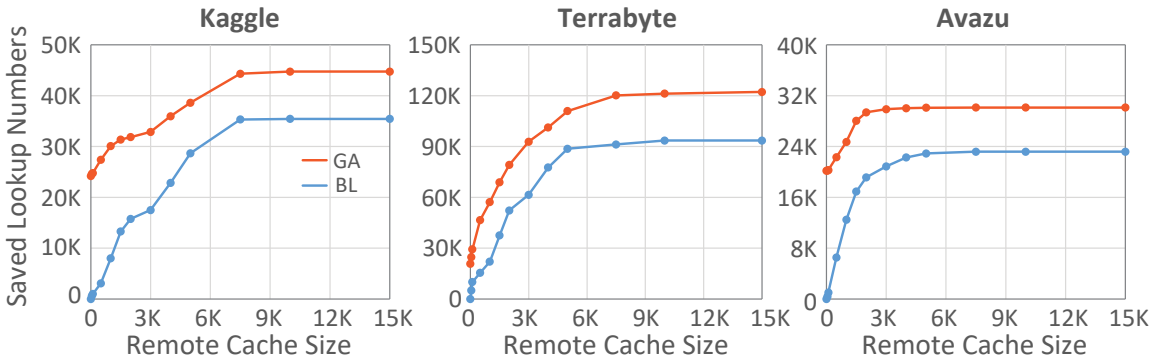


Figure 5·20: Saved Lookup Numbers (Remote Embedding Lookup Requests) of Forward Propagation Using Remote Cache

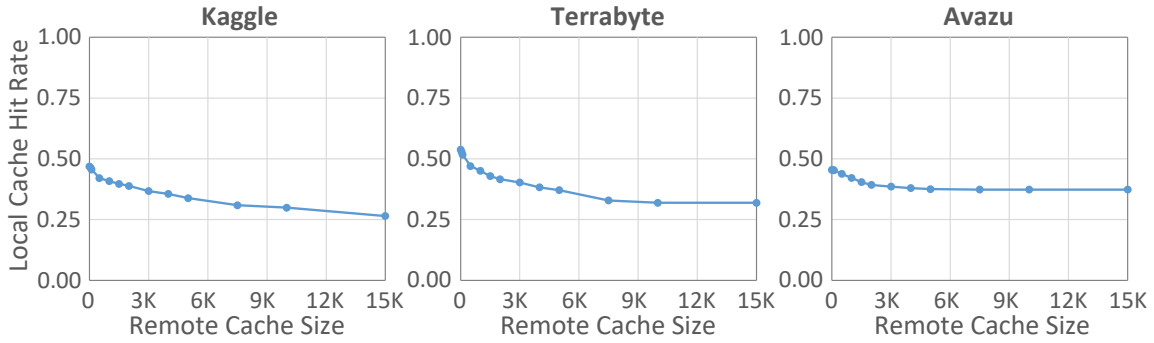


Figure 5·21: Hit Rate of Local Cache After Using Remote Cache of Forward Propagation

(3) SmartNIC Computation: Figure 5·17 shows the speedup of using SmartNIC computation in forward propagation. The figure indicates that there is not a noticeable effect on latency performance compared to training. This is because inference

batches are processed in a pipeline, and the main bottlenecks are embedding lookup and communication. The latency reduction from the SmartNIC irregular computation kernel only accounts for a small portion of the entire process. However, during training, SmartNIC computation is crucial for reducing gradients, which will be covered later.

(4) Forward Propagation Overall Speedup: Figure 5·22 indicates the overall latency speedup of the techniques across three datasets. The speedup is higher for Kaggle and Terabyte compared to Avazu. Graph algorithm improves the data locality of samples within batches and enhances the overall system latency speedup. The figure highlights that remote cache and graph algorithm significantly enhance forward propagation by reducing communication workloads and increasing data reuse. Fewer communication requests significantly reduce the all-to-all bottleneck. Meanwhile, local cache and SmartNIC computation show limited impact on forward propagation as inference batches are processed in a pipeline, obscuring the latency between them. These two techniques mainly alleviate GPU memory bandwidth pressure, kernel overhead, and hardware resource utilization.

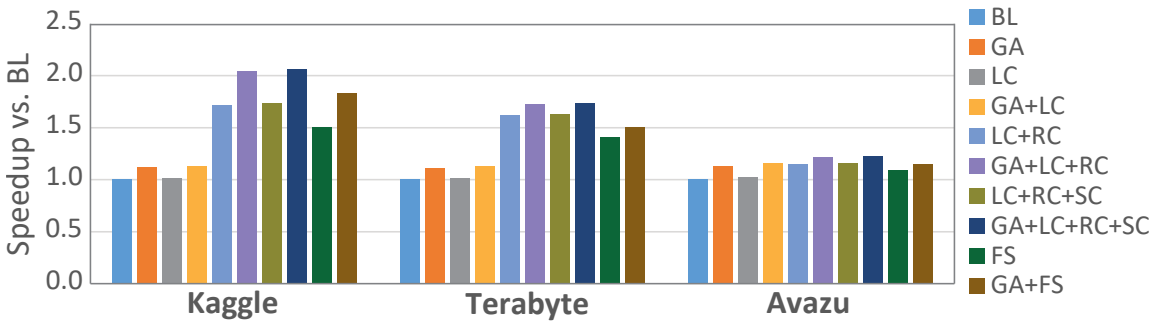


Figure 5·22: Latency Speedup of Forward Propagation Comparison Between Using Graph Algorithm, Local Cache, Remote Cache, SmartNIC Computation and Fully SmartNIC System

Backward Propagation: In backward propagation, samples are trained in sequential batches, with each batch starting only after the previous one is completed.

During this process, the embedding table's gradient losses are updated. The remote cache is not beneficial in this stage as the cached embedding vectors become outdated after backward propagation.

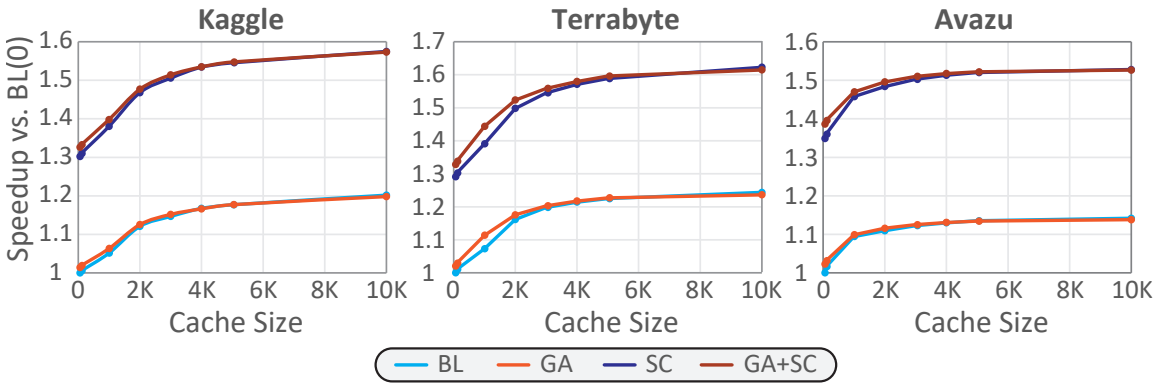


Figure 5·23: Throughput Speedup of Backward Propagation Using Local Cache on SmartNIC

Figure 5·23 shows the training throughput speedup of graph algorithm, local cache, and using SmartNIC compute with three datasets. As the figure indicates, local cache size increases, the throughput speedup increases accordingly. This results from more embedding table requests being serviced by the local cache of the SmartNIC, reducing the pressure on GPU memory bandwidth and embedding lookup overhead. Figure 5·24 indicates the hit/miss rate of local cache on SmartNIC.

SmartNIC computation is crucial in improving training throughput by handling irregular computation and reducing gradient loss updates. Two levels of reduction, including gradient reduction on the local node and global gradient reduction of all other nodes, are performed on the SmartNIC.

Figure 5·25 indicates an overall throughput speedup. Graph algorithm improves the overall throughput speedup by an average of 1.1 ×. Local cache improves the throughput speedup by 1.3 ×. SmartNIC computation improves the throughput by 1.4 ×. Overall the throughput speedup can reach 1.5 ×.

We also evaluated the effect of batch size on the throughput speedup with Smart-

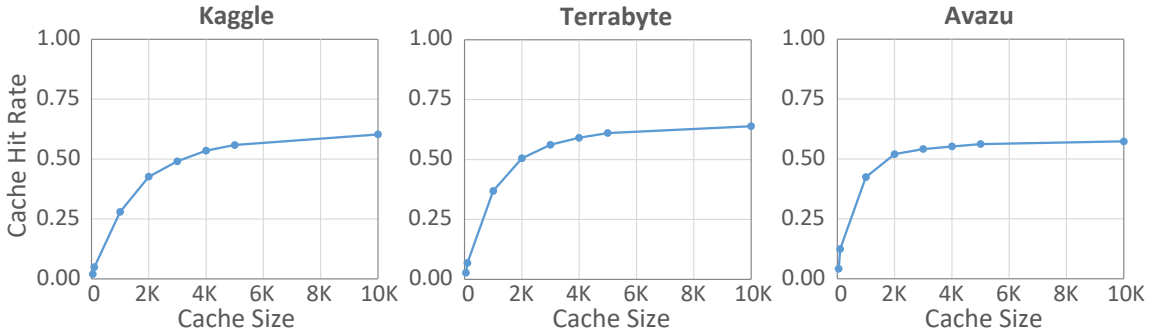


Figure 5·24: Backward Propagation Hit Rate of Local Cache on SmartNIC. Missing local cache indicates missing embedding lookup on GPU's HBM.

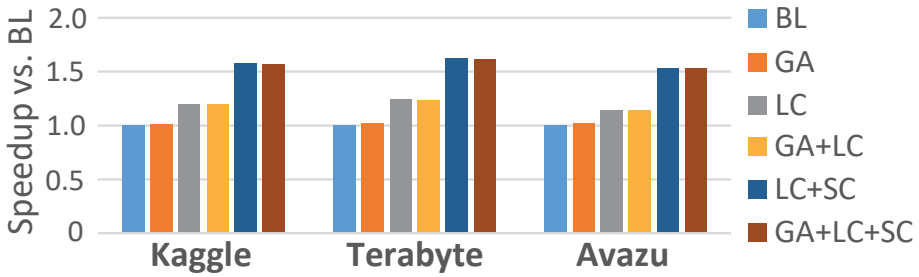


Figure 5·25: Throughput Speedup of Backward Propagation Using Graph Algorithm, Local Cache and SmartNIC Computation Kernels

NIC computation. The results show that as batch size increases, the speedup becomes bounded by the computation bottleneck.

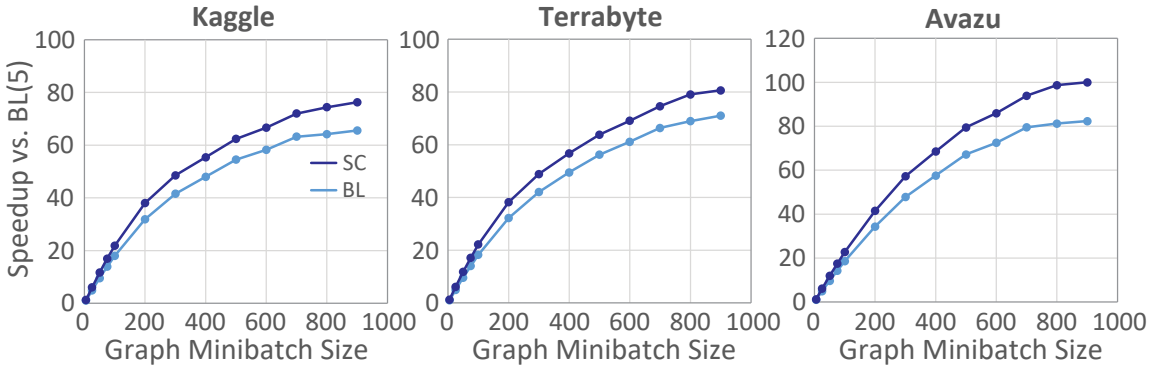


Figure 5·26: Throughput Speedup with Training Batch Sizes

5.6.3 System Scalability

We evaluated the system's scalability of inference and training using 2, 4, 8, and 16 nodes. The embedding tables are evenly distributed among each node. As a reference, we use 2-node CPU MPI as an overall baseline.

Figure 5·14 shows the inference scalability. We tested the same workloads with the number of nodes increasing in the system. As the number of nodes increases, the per-node embedding table size decreases with less memory bandwidth pressure. However, communication workloads increase as the node number scales up. The inference latency speedup shows better scalability using techniques on SmartNICs.

Figure 5·15 shows the scalability of training throughput speedup. As the system scales out, all-to-all communication and backward propagation are mainly the bottleneck that limits the scalability of training throughput. SmartNIC computation plays an important role in keeping the scalability of the system's throughput speedup. The system demonstrates better scalability with the use of SmartNIC techniques.

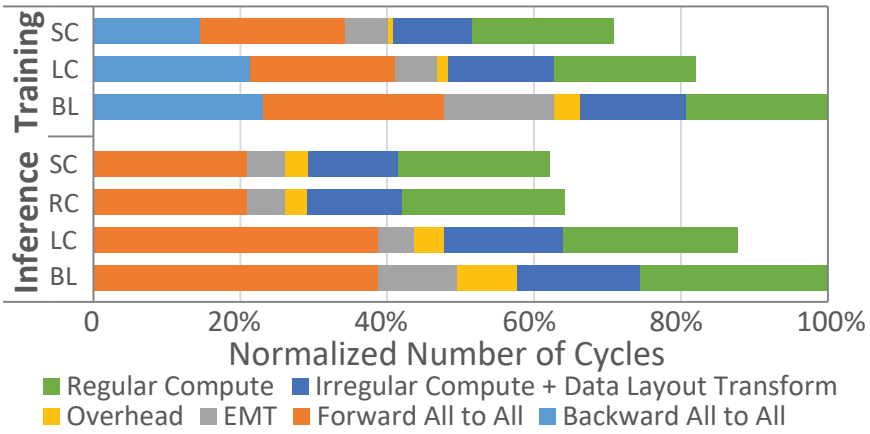


Figure 5·27: Time breakdown of DLRM inference and training. Overhead includes NIC to GPU PCIe latency, kernel calling overheads, host to device, etc. Backward all-to-all refers to gradient loss updating in backward propagation.

Figure 5·27 shows a time breakdown analysis for these techniques for inference and

training. In inference, all-to-all communication takes nearly 40 % of the total time. The local cache on SmartNIC reduces the embedding lookup time. The remote cache saves a significant amount of all-to-all communication time, as popular embeddings are stored locally on SmartNIC and do not require communication requests. Both the local cache and remote cache also eliminate overhead latency, such as the host-to-NIC latency via the PCIe bus. As training, SmartNIC computation reduces both backward all-to-all for gradients update and irregular computation latency.

5.7 Related Work

Much attention is given to using GPUs as computation accelerators. Work of (Mudigere et al., 2022) introduced a software-hardware co-design system using GPU for distributed training. Work (Kwon and Rhu, 2022) proposed a software runtime system that manages GPU DRM as a fast *scratchpad*. There are works that explore using storage technology to enhance the performance embedding operator of DLRM. Work (Eisenman et al., 2019) presents a storage system that reduces the DRAM footprint using Non-volatile Memory. Work (Wilkening et al., 2021) proposed a near-data processing solution that improves the performance of underlying SSD storage for embedding table operators. However, these works are not focused on the communication bottleneck as the DLRM scales up. Work (Zhu et al., 2021) presents an FPGA cluster for recommendation inference for embedding lookups and computation. Work (Jiang et al., 2021) proposed a recommendation inference engine using FPGA's high bandwidth memory and pipelined dataflow. These works are not targeting scalability as the recommendation model grows even more significant.

5.8 Summary with Discussion of HW/SW Codesign

DLRMs are one of the critical applications in large-scale online services and have evolved as the single largest machine learning application. In this chapter, we proposed software-hardware co-design heterogeneous SmartNIC system that targets the scalability challenges of DLRM, including communication, memory, and computation. The graph algorithm clusters similar queries in batches with higher system efficiency and performance.

Software: On the software side, we identify bottlenecks in DLRM and recognize that data follows a power-law distribution. The graph algorithm is utilized to improve data locality.

Hardware: On the hardware side, we provide a SmartNIC cache system including local cache and remote cache for embedding buffering. The computation kernels execute irregular computation and data reformatting for GPU's computation.

Codesign: By combining these, we design the SmartNIC cache system based on the nature of data distribution and further enhance efficiency using the graph algorithm. Our system extends the performance boundaries of current software and hardware platforms with less communication workload and memory bandwidth pressure.

Chapter 6

Heterogeneous GPU-CPU-SmartNIC Systems: LLMs

In the third exploration step, which we began discussing in the previous chapter, we aim to address the two key limitations of heterogeneous GPU systems extended with offload engines for large memory capacity, using large machine learning models like language models as applications. Language models have significantly larger model sizes compared to deep learning recommendation models. We explored a heterogeneous SmartNIC system integrating GPU and CPU cooperation. The two limitations we seek to overcome in these systems are limited data exchange efficiency and limited computation and control efficiency. We propose a software and hardware co-design system that leverages SmartNIC capabilities, utilizing GPUs as accelerators and CPUs as offload engines.

6.1 Motivation

In recent years, large-scale deep learning has advanced rapidly and come to dominate the current AI application landscape. Large models provide significantly better sample efficiency and better performance (Kaplan et al., 2020). With the rise of attention-based deep learning models, model size has grown exponentially. For instance, large language models like Meta's LlaMA (Touvron et al., 2023a; Touvron et al., 2023b), OpenAI's ChatGPT (OpenAI, 2024), and Google's LaMDA (Thoppilan et al., 2022) have enabled impressive AI capabilities that are transforming daily

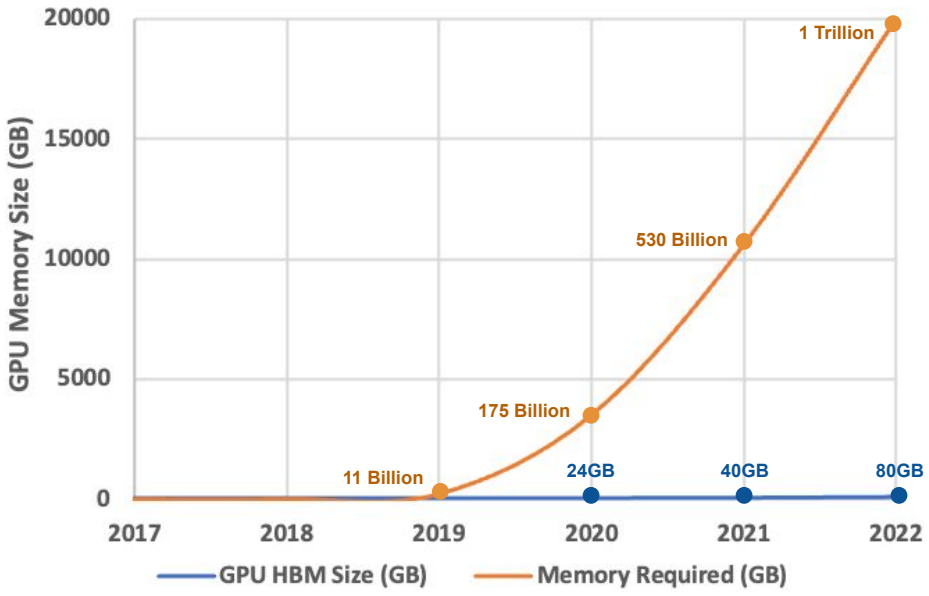


Figure 6·1: GPU memory wall

life. The functionality of these powerful models is largely attributable to their immense size - 70 billion parameters for LlaMA 2, 175 billion parameters for GPT-3, 540 billion parameters PaLM, and 1.8 trillion parameters for GPT-4. This trend of ever-increasing model size is expected to continue extensively in (at least) the near future.

These large deep learning models far exceed the memory capacity of a single GPU. Although training parallelism strategies are used, the system still faces fundamental limitations in fitting models with billions to trillions of parameters into the confined cluster of aggregated GPUs' memory. To address this, memory-efficient optimizations such as the Zero Redundancy Optimizer (Rajbhandari et al., 2020) and Pytorch Fully Sharded Data Parallel (FSDP) (Zhao et al., 2023) have been introduced, trading extra communication for efficient 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, we are hitting the

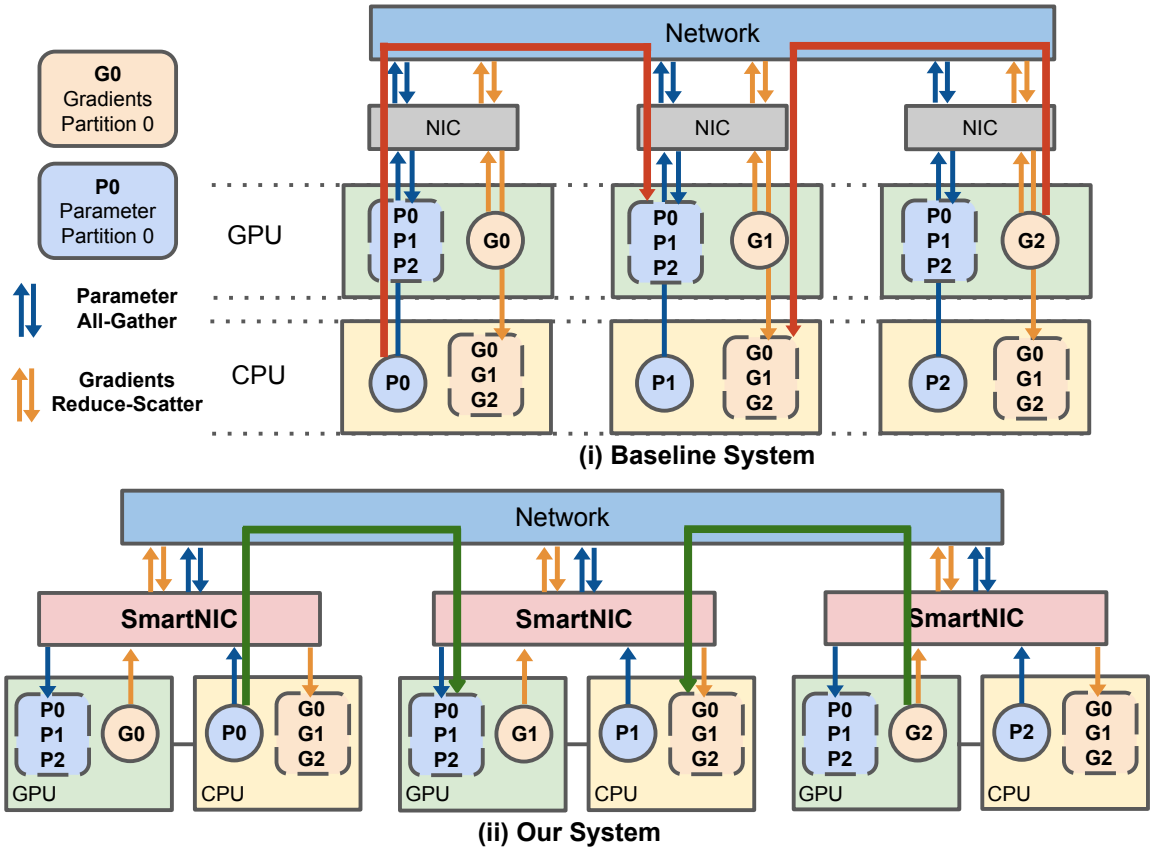


Figure 6·2: System Data Exchange Efficiency

memory wall. The current models require hundreds of GBs to tens of TBs of GPU memory just to fit the parameters, but GPU memory capacity is not increasing nearly fast enough to keep pace with the exponential growth in model size shown in Figure 6·1. To accommodate such massive models, the scale of GPU clusters increases exponentially as well. However scaling up GPU clusters indefinitely is not feasible. The performance of the GPU cluster does not increase proportionally with the number of GPUs in the system. System overhead and communication emerge as critical bottlenecks affecting performance. Simultaneously, larger-scale systems introduce substantial power consumption with limited power usage efficiency. The GPU memory wall further hinders researchers and most companies from building and accessing large models.

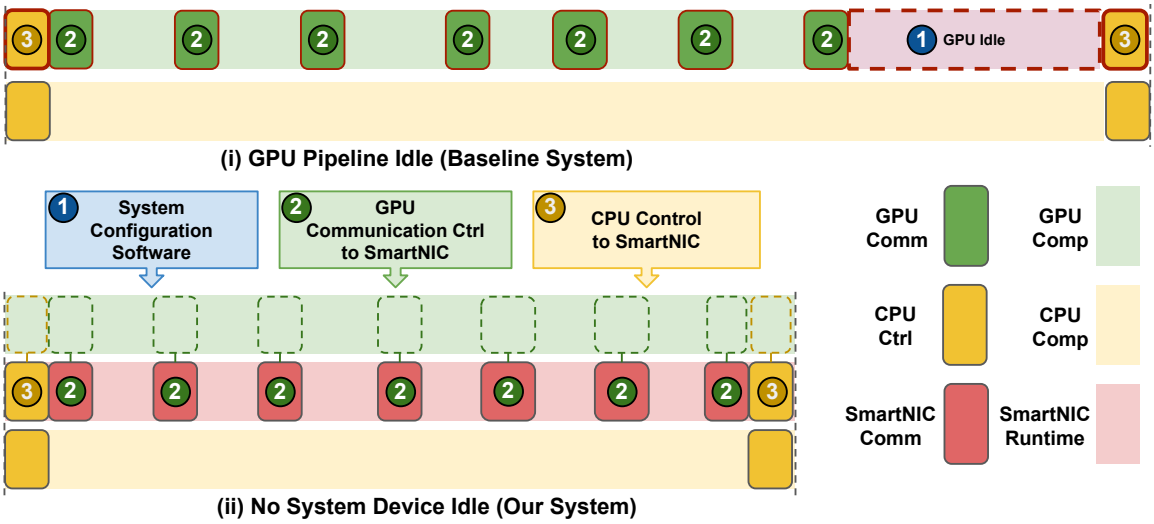


Figure 6·3: System Device Computation and Control Efficiency

To reduce the required scale of GPU clusters and mitigate the GPU memory wall limitation, heterogeneous deep learning training has emerged to leverage offload engines like CPUs and NVMe memory by offloading model states and parameters. Building on the foundation of Zero Redundancy Optimizer (Rajbhandari et al., 2020), ZeRO-Offload (Ren et al., 2021b) and ZeRO-Infinity (Rajbhandari et al., 2021) facilitates large model training on small-scale GPUs by offloading data and computations to the CPU with the memory optimization technique. These techniques make it possible to train ever-growing large-size models on limited resources. However, such a heterogeneous system suffers from low efficiency and limited performance, which is mainly due to two critical limitations:

(1) Limited System Data Exchange Efficiency (Figure 6·2) The architecture of a heterogeneous GPU system, with the CPU serving as an offload engine, is illustrated in Figure 6·2i. The CPU stores model parameters, gradients, and optimizer states. The GPU serves as the primary computation device for forward and backward propagation and acts as a communication device fetching offloaded data from the CPU. When employing ZeRO as the model training strategy, the model is

partitioned, requiring data exchanges between different heterogeneous system components at each partition stage. These exchanges move data from the CPUs to other GPUs and vice versa. However, even when not directly involved, the GPU must initiate, fetch, and communicate all data transfers, creating a bottleneck where data must pass through the GPU for both sending and receiving. This leads to inefficient system data exchange.

(2) Limited System Device Computation and Control Efficiency (Figure 6·3) In heterogeneous GPU systems with offload engines, to maintain system efficiency, the delayed parameter update (DPU) technique (Ren et al., 2021b) allows the CPU's data path for parameter updates to be detached from dependency and parallelized from the GPU's forward/backward propagation (Figure 6·6). However, despite this separation, the GPU still controls and schedules the CPU's computation, leading to reduced efficiency in computation and control on both sides. Additionally, the two data paths' varying computational power and complexity result in differing execution latencies, potentially causing idle periods for the GPU or CPU between training steps. Furthermore, the GPU's communication operator requires significant memory, computation resources, and controlling overhead. Collectively, these factors contribute to an overall diminishing of system device computation and control efficiency.

Advanced network interface cards known as SmartNIC have emerged to combine communication, control, and computation, which is useful for domain-specific computation. Such capabilities and placement in the node (network-facing) point to their use in overcoming the system efficiency challenge in training and serving large-scale heterogeneous GPU systems. However, adding SmartNICs to a distributed system only addressed point-to-point communication latency. Currently, no distributed system design leverages SmartNIC resources to overcome the heterogeneous GPU system

with an offload engine.

In this chapter, we introduce a SmartNIC-GPU-CPU heterogeneous system (SGC system) for large machine learning models with software-hardware co-design that improves the system performance and efficiency, lowers the power consumption and budget, and mitigates the data exchange overhead. Our system breaks the boundary of heterogeneous devices in the system and aims to provide the following:

1. Higher system performance and efficiency with less budget and power consumption. (Table 6.1i)

2. High performance with with less number of GPUs (Table 6.1ii)

3. Support larger models with good performance. (Table 6.1iii)

Several designs and optimization techniques are proposed to target the two performance and efficiency limitations of existing GPU heterogeneous systems with offload engines. **To improve system data exchange efficiency (Figure 6·2ii):** (1) SmartNIC acts 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) SmartNIC prefetch techniques initiate communication proactively before parameters are needed by the GPU or CPU, compacting the pipeline without data waiting and largely mitigating communication bottlenecks via overlap of computation and communication. (3) SmartNIC buffering technique that stores the duplicated communication data transfer, reducing overall system communication workload. **To improve system device computation and control efficiency (Figure 6·3ii):** (1) SmartNIC provides dynamic scheduling and control, allowing both GPU and CPU to concentrate on computation with reduced interruptions and overhead. (2) System configuration software optimizes system and model settings for maximal efficiency given different system specifications, minimizing device idle time.

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

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

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

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

Table 6.1: Our System Goal

To summarize, the contributions of this Chapter include:

- A highly efficient and high-performance SmartNIC-GPU-CPU heterogeneous system that trains large-scale machine learning models with a limited number of GPUs, low power consumption, and cost-effective budget allocation.

- SmartNIC design incorporates advanced techniques such as dynamic scheduling and control of heterogeneous system device components, a SmartNIC prefetch system that optimizes the system pipeline by eliminating data waiting, and SmartNIC buffering techniques that effectively alleviate communication bottle-

necks.

- System configuration software that optimizes system and model setup based on hardware specifications and resources to maximize efficiency.

6.2 Background

6.2.1 Parallelization Strategy

There are various approaches to parallel machine learning models to tackle the large-size model and improve the training efficiency in the scale-out system. Data Parallelism works for a model that fits into the device memory for training. It shards the data across all the processing nodes with the duplicated model. The processing is done in parallel and all setups are synchronized at the end of each training step. Model Parallelism is suitable when a model cannot fit into a single device's memory, involving the vertical splitting of the model among processes. Pipeline Parallelism divides training examples into small batches and pipelines the execution of each set across multiple machines. Tensor Parallelism partitions tensors into chunks, distributing them to system devices, where each shard resides on a designated GPU, processed in parallel, and synced at the end of the step. 3D Parallelism combines various parallelism strategies, allowing scalability to trillions of parameters with efficiency. However, it requires significant code modification and substantial effort to balance pipeline stages.

Zero Redundancy Optimizer (ZeRO): Zero Redundancy Optimizer is a memory optimization parallel 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 proportionately to the number of devices. ZeRO distributes the training batch across multiple GPUs, similar to data parallel training. However, instead of duplicating models, ZeRO par-

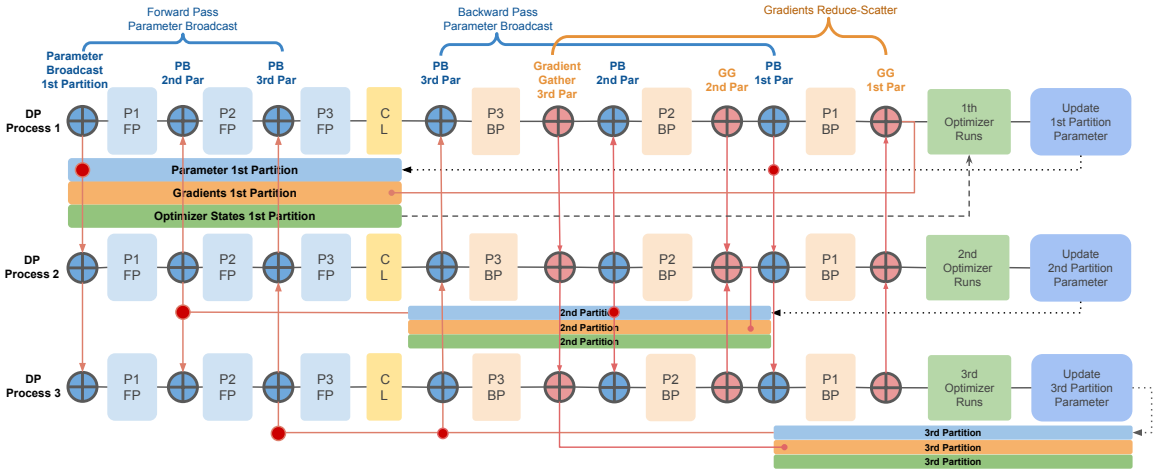


Figure 6·4: ZeRO Overview with No Bandwidth Aggregation. (P1FP: Forward Propagation of Partition 1. CL: Compute Loss. P3BP: Backward Propagation of Partition 3)

titions model states across all GPUs and utilizes communication collectives to gather parameters when needed during various phases of the training process. It offers a more generic solution that does not require users to modify the model extensively for implementation, providing improved compute efficiency and scalability.

ZeRO operates in three stages corresponding to three model states and Figure 6·4 indicates an overview of the workflow, where the model is partitioned into three parts and distributed among three data parallel processes.

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 process 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 before each forward and backward pass to share parameters with other data-parallel processes. This process repeats until the completion of all forward pass operations. After each process completes loss computation, a parameter Broadcast is issued before each backward pass partition.

Gradients are generated on each process after the backward pass, and a Gradients Gather operation gathers them to the corresponding process. The partitioned optimizer then works on the parameter updates allocated to its partition. The complete ZeRO workflow is illustrated in Figure 6·4, where the model is partitioned into three parts and distributed among three data parallel processes.

Heterogeneous System Training: Work has been done to propose a heterogeneous system leveraging CPU or NVMe memory has been explored to augment the system's memory capacity. Such a system enables the training of significantly large models on a limited number of GPUs.

ZeRO-Offload (Ren et al., 2021b) presents a heterogeneous training approach with the CPU as an offload engine, based on the ZeRO-2 foundation to offload optimizer states, gradients, and the computation of parameter updates on the CPU.

Figure 6·5 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.

While the ZeRO parallel strategy allows for large model sizes proportional to the number of devices, it necessitates aggregating GPU memory at a scale that can accommodate both the model and residual states for training. However, while using ZeRO as a parallel strategy, the expansion of model size surpasses the growth rate of GPU High Bandwidth Memory (HBM) size, resulting in an exponentially increasing size of the GPU system required to train ever-growing deep learning models. Efforts

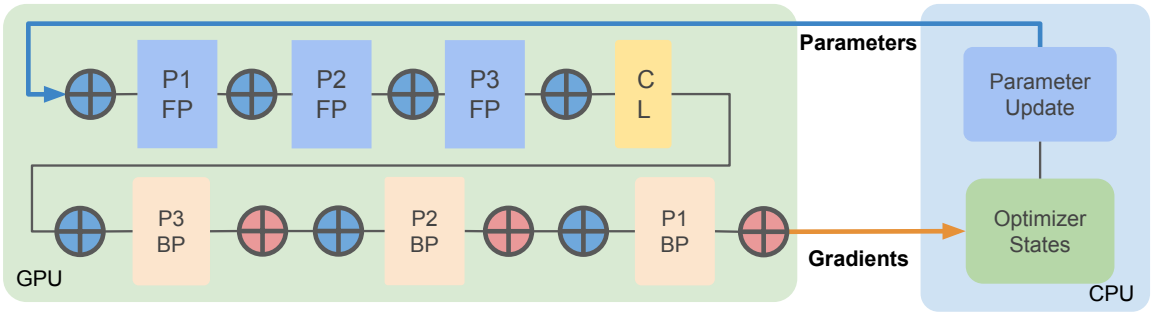


Figure 6·5: ZeRO Offload with CPU as Offload Engine

have been made to explore heterogeneous system training, leveraging CPU or NVMe memory to extend the system's capacity and enable the training of significantly larger models on a limited number of GPUs.

Several crucial considerations must be considered during offloading: (1) The CPU computation throughput is orders of magnitude slower than the GPU. (2) Minimizing the communication volume between CPU and GPU memory is essential.

The computational complexity of deep learning training per iteration typically follows O(MB), where M represents the model size, and B is the batch size. In contrast, the complexity for optimizer computation and parameter updates is O(M). Leveraging mixed precision, the forward and backward passes use fp16 as the parameter type, and the optimizer maintains a copy of fp32 as the parameter type for updating. This configuration is optimized for offloading the optimizer state and parameter updates onto the CPU, thereby minimizing CPU computation complexity and communication volume between the CPU and GPU.

Bandwidth Aggregation: ZeRO-Infinity takes a step further in addressing several limitations of ZeRO-Offload (Ren et al., 2021b). Firstly, ZeRO-Offload still necessitates GPU memory to store parameters, limiting the model size by the number of parameters that GPU memory can accommodate. Secondly, ZeRO-Offload faces constraints introduced by PCIe bandwidth limitations. ZeRO-Infinity overcomes these

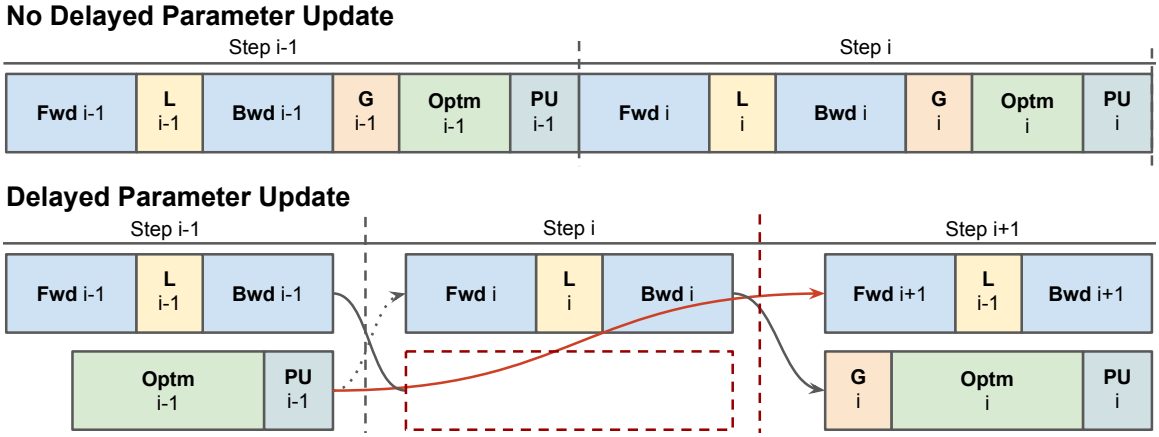


Figure 6·6: Delayed Parameter Update. (PU: Parameter Update, G: Gradients, L: Compute Loss)

limitations by introducing bandwidth aggregation. In ZeRO-Infinity (Rajbhandari et al., 2021), parameters are partitioned across all data-parallel processes, utilizing All-Gather instead of Broadcast to gather parameters for each process. The aggregated PCIe bandwidth across all nodes in the system 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 a single PCIe link.

6.2.2 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 hide the offload engine computation, ZeRO-Offload employs a one-step delayed parameter update, as depicted in Figure 6·6.

With no DPU, after the backward pass, gradients are generated, and the optimizer updates the parameters for the next iteration. The optimizer states' computation depends on the output from the backward pass. With the introduction of a DPU,

the pipeline is divided into two independent data paths, where the optimizer states computation and parameter updates no longer depend on forward and backward propagation. At step $i+1$, parameter updates utilize gradients from step i, while the forward and backward pass uses parameters updated at step $i-1$. From this step afterward, the model at the $(i+1)$ th step is trained using parameters updated at the $(i-1)$ th step, and the two data paths run in parallel with no dependency.

6.2.3 Limitations of Existing Systems

Heterogeneous systems with nodes comprising a GPU with an offload engine facilitate the training of large-scale machine learning models, but their performance and efficiency are constrained.

Limited System Data Exchange Efficiency: During the forward and backward pass computation, parameter data exchanges are required at each node to collect partitioned parameters from all other CPUs. The parameter data exchange involves 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, data copy, and initiation of communication.

After each partial backward propagation, gradients are produced in corresponding partitions for each process. A Reduce-Scatter is required to distribute the gradients to their destination data-parallel process 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, and data flows through the GPU, forwarding it to the offload engine.

As Figure 6·2i, such barriers between the system's heterogeneous components limit the system data exchange efficiency in two cases:

(1) The 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) The data movement of gradients generated from GPU to parameter updates computation on the CPU offload engine. 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: The delayed parameter update is employed in a heterogeneous GPU system with an offload engine to maintain computational efficiency. This approach allows the CPU offload engine's parameter update computation data path to be detached and parallelized from the GPU's data path, as illustrated in the upper Figure 6·7. Although these two data paths are independent computation paths, the CPU data path is still under GPU's control and scheduling. This control dependency reduces computational efficiency and increases overhead on both sides.

Furthermore, 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 diverse execution latencies and potential idle device time for the GPU or CPU between each training step. 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 GPU initiates communication operators for data exchange, necessitating memory allocation for exchanged data and computation resources for data management. This utilization of GPU hardware resources introduces overhead,

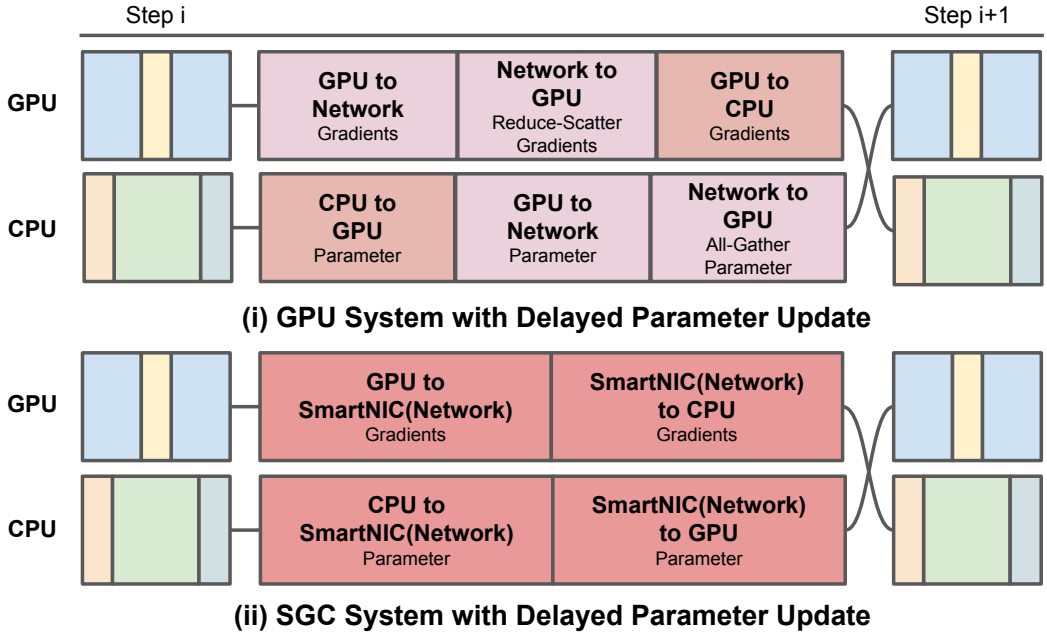


Figure 6·7: System Comparison with DPU

thereby constraining the GPU's efficiency. Collectively, these factors contribute to the diminished device computation and control efficiency of the system.

6.3 Memory Requirements Analysis

Current large-scale models encompass model sizes ranging from millions to trillions of parameters. Training these extensive models necessitates a substantial number of GPUs, and a GPU memory pool size from Gigabytes to Terabytes. This section delves into understanding the significant memory consumption factors. The memory consumption during training can be categorized as due to model states and residual states. The analysis is based on the utilization of mixed precision and the Adam Optimizer.

6.3.1 Model States

The model states take most of the training model memory consumption and it is consist of (1) Optimizer States, (2) Gradients and (3) Parameters. Assuming that we have a model that contains ψ parameters.

When using mixed precision training, both parameters and gradients are represented in FP16. Following backward propagation, the optimizer states incorporate the generated gradients to update the current iteration's parameters. These optimizer states store a copy of momentum, variance, parameters, and gradients with FP32 precision. To accommodate the FP16 copies of parameters and gradients, a memory space of 4ψ bytes is necessary. For the optimizer states, a total space of 16ψ bytes is required to store all relevant information. Consequently, the overall model storage requirement amounts to 20ψ bytes. Fitting a 100 billion-parameter transformer-based model necessitates 64 NVIDIA V100 GPUs, while storing a model with 1 trillion parameters requires 512 GPUs.

6.3.2 Residual States

In order to make the training process run, extra residual memory is required. There are mainly three categories, including (1) Activations, (2) Temporary Buffer, and (3) Memroy Fragmentation. Activations are the primary source of memory consumption during the training process, with memory requirements highly contingent on factors such as model architecture, batch size, and sequence length (Rajbhandari et al., 2021). While activation memory can be minimized using techniques like activation checkpointing (Chen et al., 2016), a GPT-like model with 100 billion parameters necessitates around 60GB of memory for a batch size of 32 (Rajbhandari et al., 2020).

Temporary buffer space is essential for holding computation parameters before

applying operators in the model. Adequate memory is needed for parameters and gradients to conduct both forward and backward propagation. Furthermore, for communication operators like gradients Redcue-Scatter and parameter All-Gather, memory must be allocated for the execution of these operations. Memory fragmentation occurs when memory is divided into chunks, and there isn't sufficient contiguous memory to execute a given operator. This can lead to running out of memory, even when memory is technically available.

6.4 SmartNIC-GPU-CPU Heterogeneous System

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

6.4.1 Breaking the System Boundary

During the training of large machine learning models using the ZeRO strategy, there is a substantial and frequent exchange of data among system components. In a GPU system with no SmartNIC, all data exchanges must be initiated and facilitated by GPUs. As illustrated in the upper Figure 6·7, when the current CPU sends offloaded parameters to other remote GPUs for the next iteration's forward and backward propagation, 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 happens when gradients are generated by the current GPU and must be 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. The boundary of heterogeneous device components limits the data exchange efficiency.

The SGC system, with SmartNICs, breaks such boundaries by acting as an intermediate layer between GPUs and CPUs, seamlessly connecting system heterogeneous

components (Figure 6·2ii). The 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 (depicted in Figure 6·6), the GPU and CPU data paths no longer depend on each other. Offloading the CPU control from GPU to SmartNIC alleviates burdens on both GPU and CPU. Controlling applications on the SmartNIC, not only does it enhance the computational efficiency of the system's GPU and CPU but also reduces hardware utilization and control overheads. As illustrated in the bottom of Figure 6·7, GPU and CPU can initiate data transfers through the SmartNIC, compacting the overall system pipeline with fewer stages and reduced overhead.

The SGC system design simplifies the datapath pipelines for both GPU and CPU, allowing each to concentrate on computation rather than control and communication. Integrating SmartNIC to coordinate heterogeneous components and handle control unifies system and application control with communications, enhancing overall system efficiency. Additionally, this approach optimizes GPU hardware utilization, enabling more efficient handling of computation workloads.

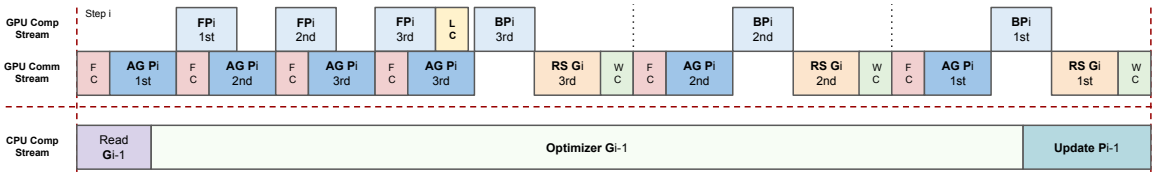


Figure 6·8: GPU with CPU offload engine system pipeline with three partitions. (FC: Fetch from CPU. WC: Write to CPU.)

Data Exchange Between System Components: As illustrated in Figure 6·8, 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 SmartNIC-GPU-CPU (SGC) system, the GPU retrieves data from the SmartNIC instead of fetching 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 directly sent to the GPU. Once all parameters are received, the Fetch SmartNIC stage on 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.

Parameter Offload: 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 fetching 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.

6.4.2 Collective Communication Support

In the training process, several collective communication operations are involved, 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 SmartNIC-GPU-CPU (SGC) architecture, we empower the SmartNIC with support for collective communication operators, offloading these operators onto 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 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 gradients 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.

6.5 Efficiency Optimizations

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

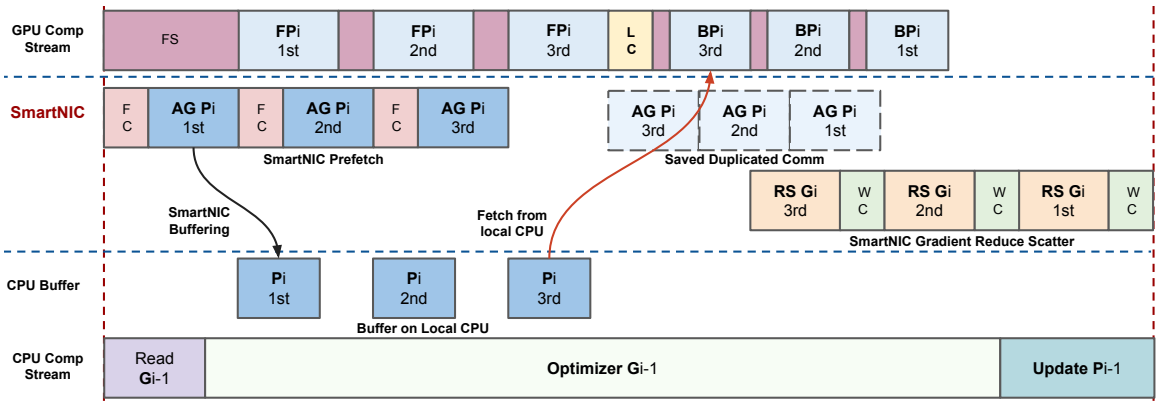


Figure 6·9: SGC Heterogeneous System Pipeline. (FS: Fetch from SmartNIC. FC: Fetch from CPU. WC: Write to CPU.)

6.5.1 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 is knowledgeable about the completion status of the ongoing communication operator. Therefore, we introduce SmartNIC prefetch, empowering the SmartNIC to pre-initiate the parameter gather communication immediately after the completion of the preceding communication.

We enabled a SmartNIC control pipeline that aligns with the GPU's datapath, improving communication and computation overlap. As depicted in Figure 6·9, 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. (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. Upon the completion of the All-Gather operation, the SmartNIC proceeds to initiate the gathering operation 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, awaiting the GPU to initiate the *Fetch SmartNIC* 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 *Fetch SmartNIC* stage until the SmartNIC has all the necessary data prepared for forwarding.

During backward propagation, a duplicated parameter gathering operation is necessary before initiating the backward propagation (Figure 6·4). 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. SmartNIC will remain waiting until the GPU fetches the parameters.

6.5.2 SmartNIC Buffering Technique

As depicted in Figure 6·4, forward and backward propagation involves 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 gathering 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 our SmartNIC design, where parameter All-Gather and gradients Reduce-Scatter overlap, the parameter gathering still consumes a substantial share of the network bandwidth, potentially compromising the performance of both communication operators. To address this redundancy and enhance system performance, we introduce the SmartNIC buffering technique.

As illustrated in Figure 6·9, 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 *Fetch SmartNIC* 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 all backward propagations are 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.

6.6 System Configuration Software

6.6.1 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:

- Model Size
- Batch Size
- Data Parallel Process Number (Model Partition Number)
- GPU Computation Power

- CPU Computation Power
- Network Configuration
- Heterogeneous Components Connection Configuration

6.6.2 Configuration Metrics Impact and Trade Off

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.

Advances in device connection technologies include high-bandwidth connections like NVLink (NVIDIA, 2023) for data exchange between heterogeneous components such as GPU and CPU. This benefits the SmartNIC buffering technique, ensuring that data exchanged between GPU and CPU does not compromise overall system performance. The SmartNIC buffer also saves network bandwidth during backward propagation when parameter All-Gather and gradient 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.

6.7 Evaluation

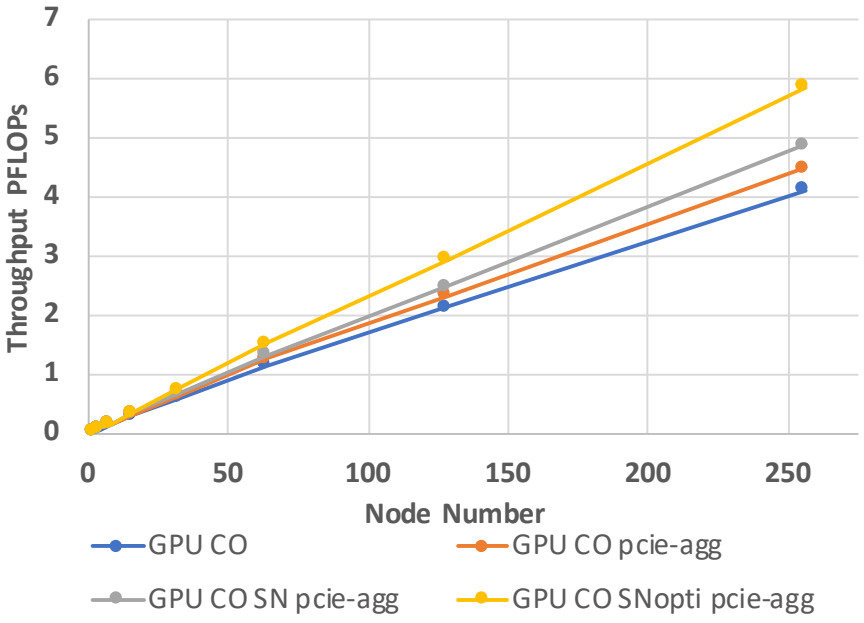


Figure 6·10: 10B Model Throughput (PFLOPs)

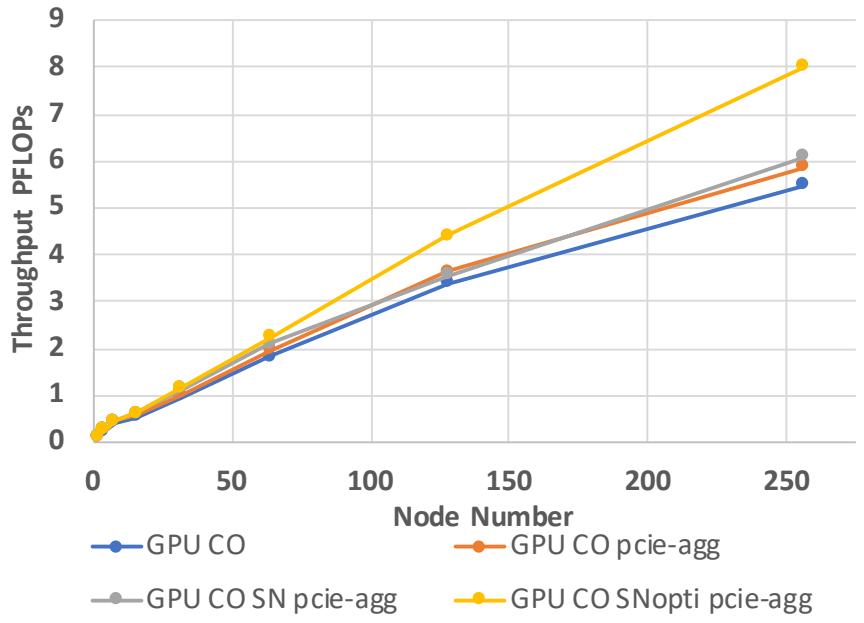


Figure 6·11: 100B Model Throughput (PFLOPs)

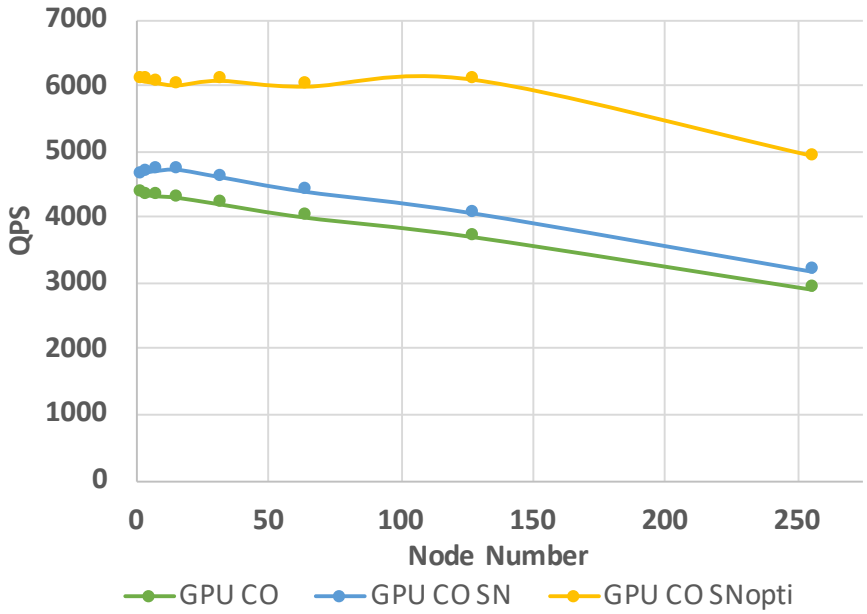


Figure 6·12: DLRM Kaggle Sample/GPU/Second (QPS)

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

6.7.1 Experimental Setup

The baseline for the GPU system is built on PyTorch FSDP with CPU as an offload engine, employing ZeRO as the data-parallel strategy (Zhao et al., 2023). The large transformer-based model is based on Meta LLaMA (Touvron et al., 2023a), and the recommendation model is based on Deep Learning Based Recommendation Model (DLRM) (Naumov et al., 2019). To evaluate the SmartNIC hardware, we utilized the Xilinx Alveo U280 FPGA, incorporating a configured hardware implementation of runtime, control logic, and collective communication kernel through High-Level Synthesis (Vitis HLS). Since there is no exact heterogeneous SmartNIC-GPU-CPU system, we conducted evaluations using real-world tested parameters on each component device and integrated them into a cycle-accurate simulator aligned with the baseline system. The system details are Intel(R) Xeon(R) Gold 6226R CPU, NVIDIA

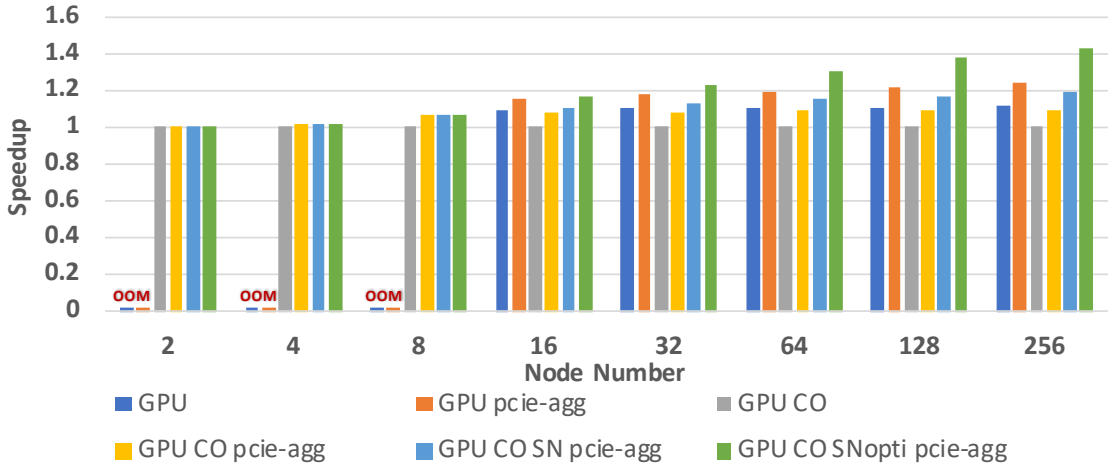


Figure 6·13: 10B Model Training Iteration Latency Speedup

Tesla V100 Tensor Core GPU, Bidirectional 32 GBps PCIe and 200GbE Network.

Hardware	**Specification**
GPU	NVIDIA Tesla V100 Tensor Core GPU
GPU Memory	32 GB 900 GB/sec HBM2
CPU	Intel(R) Xeon(R) Gold 6226R CPU
CPU Memory	2933 MHz DDR4
PCIe	Bidirectional 32 GBps PCIe
Network	200GbE

Table 6.2: System Hardware Details

The evaluation Figure uses the following abbreviations: GPU = GPU only system with no offload engine, GPU pcie-agg = GPU only system with PCIe bandwidth aggregation, GPU CO = GPU system with CPU offload (Baseline), GPU CO pcie-agg = GPU system with CPU offload and PCIe bandwidth aggregation, GPU CO SN pcie-agg = SmartNIC-GPU-CPU system, GPU CO SNopti pcie-agg = SmartNIC-GPU-CPU system with optimization techniques.

6.7.2 Performance

The system performance evaluation is based on the three system goals as Table 6.1 and the system scalability.

Higher Performance With Same Scale: Figure 6·13 illustrates the training latency speedup for a 10-billion-parameter transformer-based model. When the system scale is less than 16 nodes, the model is too large to fit into the GPU-only system, as indicated by the *OOM* label in the Figure. At this point, the performance across systems is nearly identical because the CPU computation data path becomes the bottleneck. With a smaller system scale, 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. The parameter All-Gather communication operator used by pcie-agg requires less network bandwidth than the parameter Broadcast.

The SGC system, incorporating SmartNIC optimization techniques, outperforms 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 enhancement is primarily attributed 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 6·14 presents the training latency speedup for a model with 100 billion parameters. With a system scale of fewer than 16 nodes, the CPU computation data path constrains the system performance. However, as the system scales from 16 to 256 nodes, the SGC system achieves a notable speedup over the baseline, reaching up to 1.6 ×. With larger model sizes, the communication and PCIe latency becomes

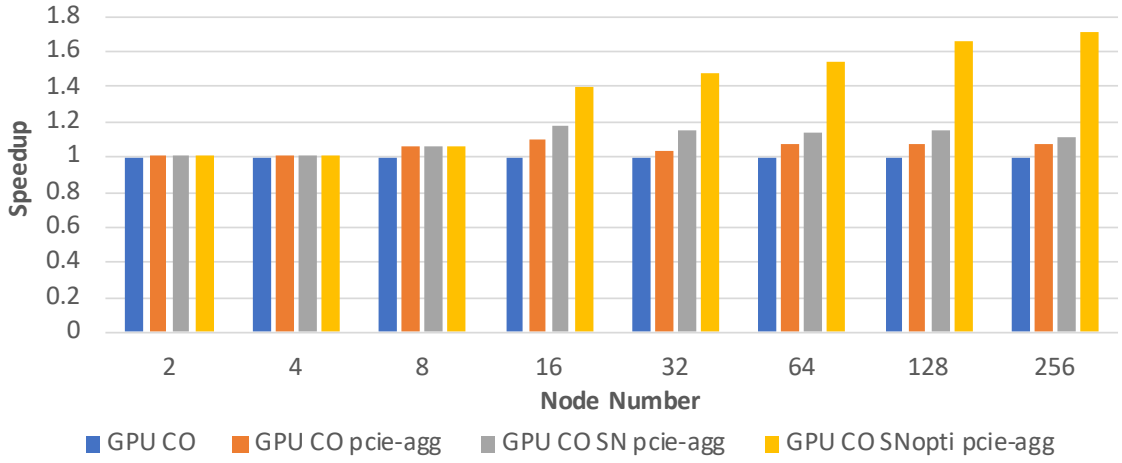


Figure 6·14: 100B Model Training Iteration Latency Speedup

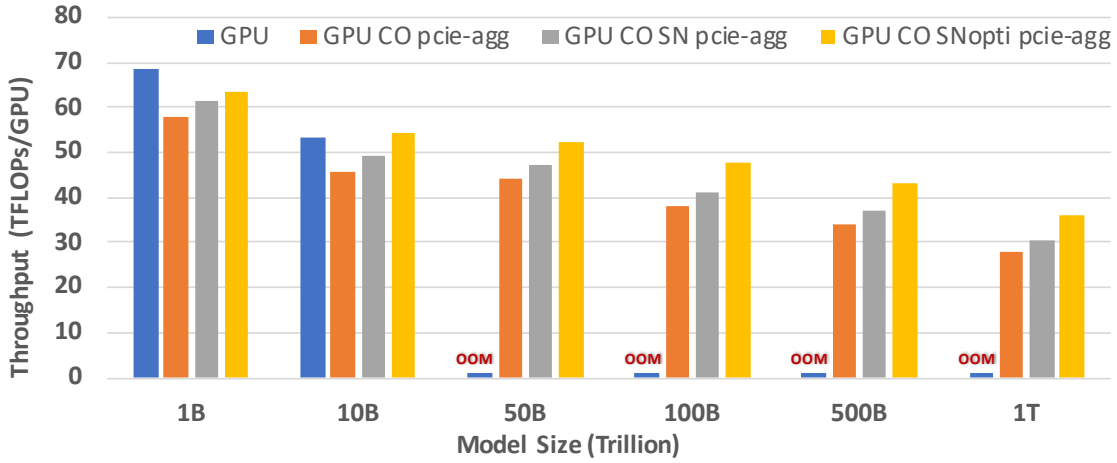


Figure 6·15: 16 Node System Training Throughput

a significant factor in the overall training latency. The SGC system improves the overlapping of communication and computation stages, contributing to its enhanced performance.

High Performance with Less Number of GPUs Required: Figure 6·15 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 scale. Achieving the same

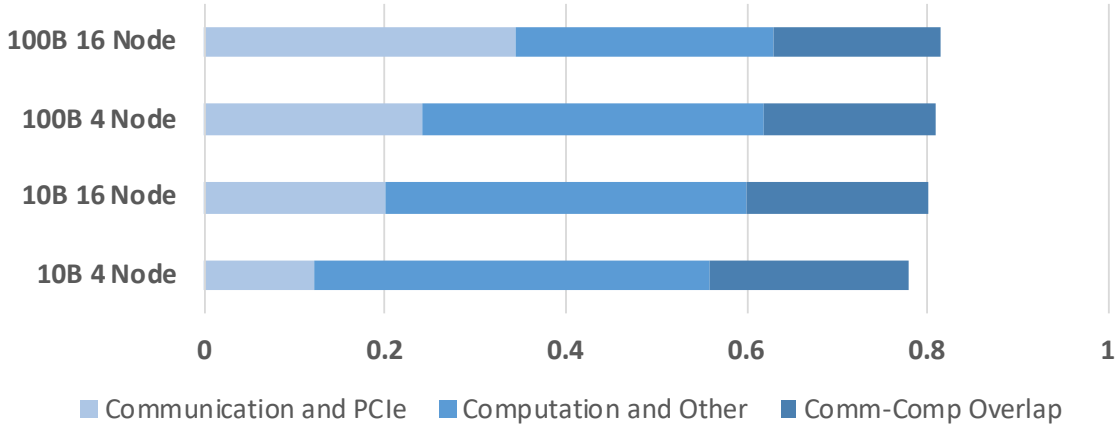


Figure 6·16: GPU with Offload Engine Latency Breakdown

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

Support Larger Model Size and Achieving Comparable Performance: Figure 6·15 indicates that SGC system with 16 nodes could support larger model sizes up to trillions of parameters. As the model size increases, the SGC system achieves higher training throughput over the baseline. Meanwhile, the SGC system can train large models with almost comparable throughput to the GPU-only system. The 16-node GPU-only system could train up to 10 billion size model, 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.

System Scalability: Figure 6·16 shows the training latency breakdown. 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 portion in 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.

Figures 6·10, 6·11, and 6·12 present 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 minimizing system overhead.

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

6.7.3 System Software Configuration

As in the previous Section, there are differences between GPU and CPU computation power and complexity. When the CPU computation path constrains the system's performance, the benefits of the SGC system with optimization techniques may be limited, as the GPU data path can not start the next training iteration unless the CPU is completed. The upper part of Figure 6·17 illustrates the normalized GPU data path latency over CPU data path latency. A value less than 1 indicates that CPU data path dominates the system latency. Both system scale and batch size influence this ratio. As the system scales up, the CPU data path experiences 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

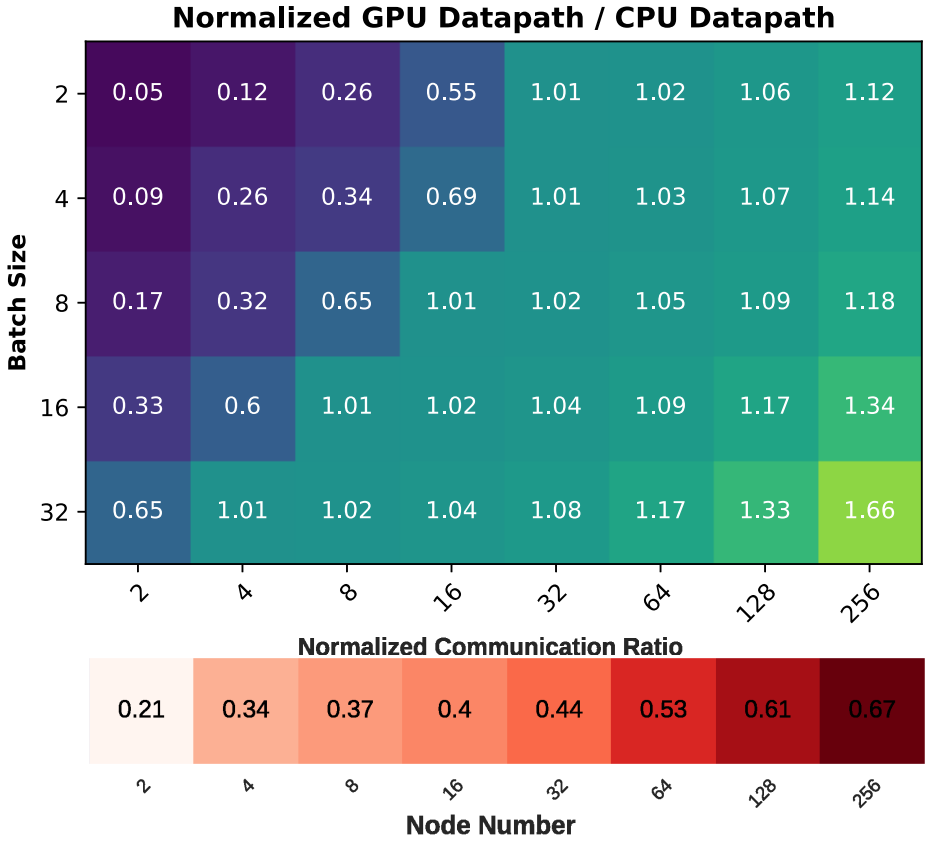


Figure 6·17: System Configuration Software with 10B Model

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 6·17 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, aiming to achieve higher efficiency and improved system performance.

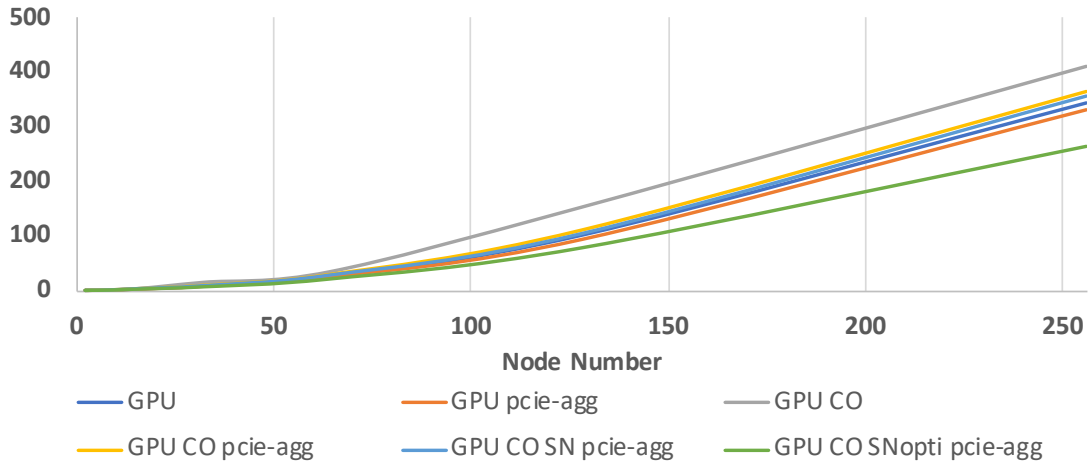


Figure 6·18: Normalized Power Consumption with 100B Model

6.7.4 Power Consumption

Figure 6·18 indicates the normalized power consumption comparison across various systems. As the system scales, the SGC system exhibits lower power consumption than others. The SmartNIC optimization in SGC contributes to power savings, particularly due to the SmartNIC buffer technique, which replaces network communication with local PCIe data fetching, thereby reducing network communication workloads. This advantage becomes more pronounced as the system scales up, as larger scales introduce increased energy consumption for network data exchange. Additionally, the SmartNIC prefetch helps compact the system pipeline by reducing device data waiting time and system bubble, further contributing to power efficiency.

6.8 Related Work

Parallel strategies are used to train large models at scale (Shoeybi et al., 2020; Shazeer et al., 2018; Wang et al., 2019a; Huang et al., 2019; Harlap et al., 2018; Lai et al., 2023; Song et al., 2023). To scale up the model training, work (Chen et al., 2016) saves memory form activation by recomputing from the saved checkpoints. Mixed precision

(Micikevicius et al., 2018) is also proposed to compress the model for less memory requirement. There have been works on CPU memory-based training approaches (Hildebrand et al., 2020; Huang et al., 2020; Jin et al., 2018; Peng et al., 2020; Ren et al., 2021a; Rhu et al., 2016; Wang et al., 2018; Fang et al., 2023), L2L (Pudipeddi et al., 2020) enable multi-billion parameter training by managing memory usage in GPU layer by layer. ZeRO-Offload (Ren et al., 2021b) and ZeRO-Infinity (Rajbhandari et al., 2021) is the state-of-the-art for large model training based on ZeRO (Rajbhandari et al., 2020) parallel training strategy. PyTorch FSDP (Zhao et al., 2023) advanced the DDP (Li et al., 2020) model wrapper enables training of large model side using PyTorch (Paszke et al., 2019)

6.9 Summary with Discussion of HW/SW Codesign

In this chapter, we propose a SmartNIC-GPU-CPU heterogeneous system for training large machine learning models using a software-hardware co-design approach. The SmartNIC serves as an intermediary layer, seamlessly connecting the GPU and CPU. A set of optimization techniques streamlines the system pipeline, reducing data wait times and maximizing the overlap between communication and computation. Additionally, system configuration software optimizes both system and model settings to achieve maximal efficiency.

Software: On the software side, we identify bottlenecks in ZeRO parallel training and implement control flow mechanisms that break system boundaries between heterogeneous components, minimizing device idle time.

Hardware: On the hardware side, we provide a SmartNIC dynamic application control runtime with hardware design optimizations such as parameter prefetching, buffering, and collective communication kernels.

Codesign: By integrating these elements, we implement system hardware based

on ZeRO and optimize control flow, software, and model configurations to match the system hardware. This combination pushes the boundaries of current software and hardware platforms for efficient machine learning model training.

Chapter 7

Heterogeneous Global Control and Disaggregated Memory System

In the previous Chapters, we propose SmartNIC-GPU-CPU (SGC) systems to address the limitations of heterogeneous GPU systems with CPU offload, such as with data exchange and computation control. Recent systems have, in part, addressed these issues. For example, PyTorch FSDP (Zhao et al., 2023), shown in Figure 7·1, optimizes large model training with support for CPU offload by overlapping communication and computation. Despite these advances, however, existing systems still face challenges in achieving high training efficiency. Among these challenges are the following.

First, current systems operate on a both a large scale and with global synchronization barriers and control. However, any particular compute unit only has access to local information, necessitating extensive communication to perform global operations. As systems scale significantly, they are limited by their Model FLOPs Utilization (MFU) (Jiang et al., 2024). MFU is the ratio of the observed throughput to the theoretical maximum throughput of 100% of peak FLOPs.

Second, All-gather and Reduce-Scatter are commonly used global communication patterns for parameter gathering and gradient reduction. As the system scales up, the communication workload increases significantly, consuming a large portion of the execution latency even with prefetch enabled. This larger scale introduces more collective workload among more processes, leading to longer communication time.

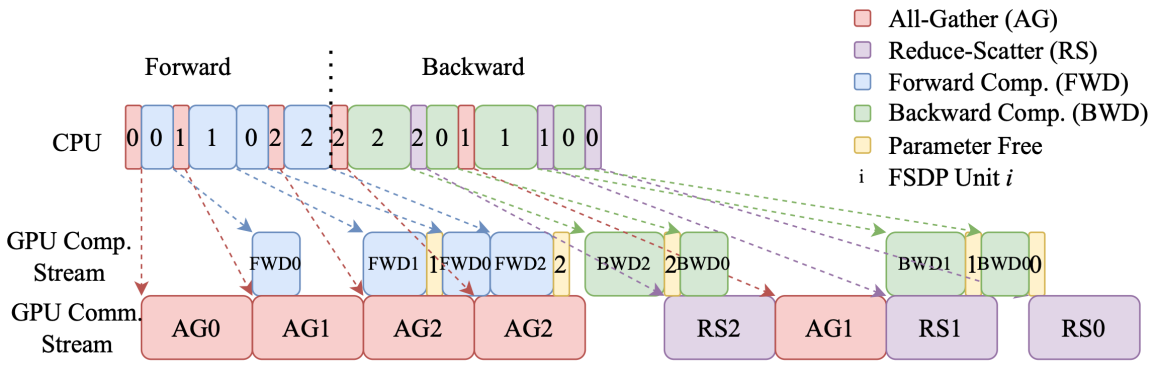


Figure 7·1: PyTorch FSDP Communication and Computation Overlap Pipeline (Zhao et al., 2023)

Third, local control, scheduling, and communication operators consume considerable hardware resources and on-chip memory. The larger the system, the greater the duplication.

Fourth, in-network computing opportunities are wasted as computing is relegated to nodes that are necessarily at the edge of the cluster. Since these in-switch, in-network opportunities are described elsewhere (Haghi et al., 2023; Haghi et al., 2024), we concentrate here on the other three challenges.

We propose a heterogeneous global control and disaggregated memory system featuring global control, scheduling, and parameter storage at the switch level. This system offers fine-grained software-hardware co-design, providing the following benefits: (1) centralized global application control, (2) centralized synchronization, including barriers, and (3) improved communication-computation overlap. All this should reduce overhead, especially communication volume, and open up the possibilities for better use of run-time performance measures, which, in turn, may lead to new load balancing strategies.

7.1 Overview of design techniques and optimizations

Several system design techniques and optimization are proposed.

1. Global control and memory system for parameter storage and update

Application control and scheduling are managed at the global level, with parameters stored in disaggregated memory at the switch level. Instead of collecting parameters from scattered local nodes and redistributing data for each leaf node's computation, we use a global parameter storage system that distributes data directly to the leaf nodes. Parameter All-Gather is thus replaced with Broadcast while gradient Reduce-Scatter is replaced with Reduce. As described elsewhere (Haghi et al., 2022) this significantly reduces the communication volume.

2. Global one-sided synchronization

(2a) Asynchronous global communication.

Unlike All-Gather and Reduce-Scatter, which require synchronization across all processes, the proposed system operates asynchronously. GPUs request data directly from, and send gradients to, disaggregated memory without the need for global synchronization. A global control unit within the switch manages parameter requests and gradient reduction asynchronously from the leaf nodes. When a leaf node requests parameters, the disaggregated memory sends them, potentially using buffering in the switch if multiple nodes request the same parameter. Similarly, when a leaf node generates gradients, they are sent to the global control unit for reduction. The switch asynchronously receives gradients from the leaf nodes and performs the reduction. Once all gradients are received, they are sent to the optimizer for parameter updates.

The proposed system leverages already deployed parameter updates by using the previous iteration's parameters for the current update. Disaggregated memory stores all parameters needed for the current iteration, making them readily accessible to leaf nodes. The global control unit manages asynchronous data requests from each leaf node, eliminating the need for synchronization during gradient reduction.

Here is an operation from a leaf node's perspective. It first retrieves parameters

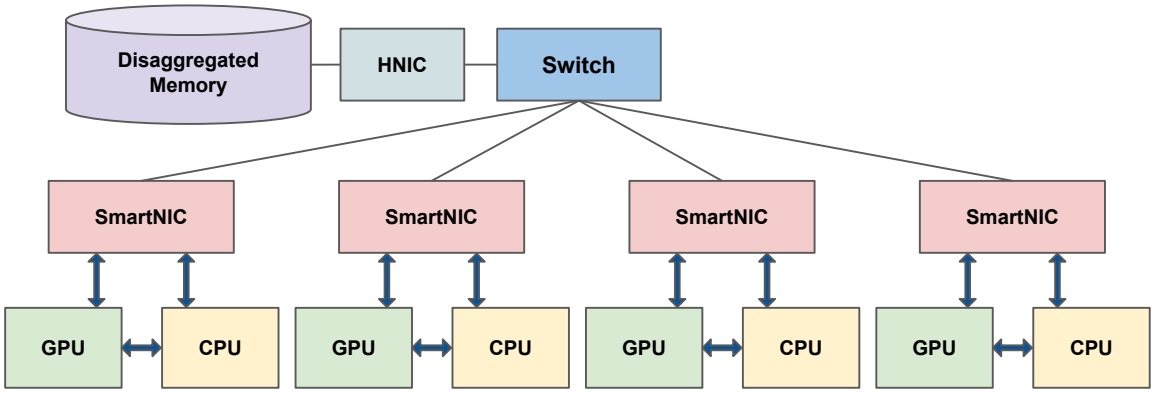


Figure 7·2: Architecture of Heterogeneous Global Control and Disaggregated Memory System

from the global control (GCU) and disaggregated memory (DM) as needed. It then sends generated gradients back to the GCU and DM for reduction and subsequent parameter updates. This process is entirely asynchronous, so leaf nodes do not wait for each other, significantly reducing training overhead. Once the request and send operations are complete, the leaf node can proceed to the next phase of the application.

(2b) Global barrier at the switch.

Traditional training methods often use synchronization points, or barriers, during initialization and between iterations, setting up communication groups among leaf nodes with a communication library. Our system, featuring global control attached to the switch, eliminates the need for such global synchronization. Instead, a global barrier table is maintained. When a leaf node sends a barrier signal, it can proceed to the next application phase without waiting for other nodes. The global control unit ensures that as long as the longest asynchronous phase of each leaf node falls within a single iteration, the nodes can execute their application phases asynchronously.

3. Data prefetching and buffering

Expanding on our previous work with SmartNICs, the global control unit (GCU) collaborates with leaf nodes' SmartNICs through prefetching and buffering, effectively

overlapping communication and computation to maximize model FLOPs utilization.

The GCU oversees parameter requests from each leaf node, promptly sending the requested parameters. Simultaneously, it tracks the application phase of each leaf node, enabling it to proactively initiate the transfer of parameters for the next phase immediately after fulfilling a current request. These parameters are buffered in the local node's SmartNIC, readily accessible when needed by the leaf node. Subsequently, after fetching the buffered parameters, the SmartNIC can proactively request prefetching of parameters for the subsequent phase from the GCU. This strategy substantially overlaps communication and computation, leading to enhanced operator utilization on the leaf nodes.

In summary, the GCU with DM system aims to allow leaf nodes to focus on computation, with parameter requests and gradient send-outs handled by the GCU. This system's global control and memory at the system level provide higher efficiency, lower overhead (synchronization, barriers), and reduced communication workload. Additionally, it facilitates easier checkpointing and fault tolerance through the combined information of global control and SmartNICs.

7.2 Background and Related Work

In general, disaggregated systems refer to any computer system where there is non-uniformity in resource deployment. As such, disaggregated systems originated almost simultaneously with the first clusters with self-contained nodes. In these early clusters, rather than each node having identical resources, e.g., CPU, memory, and I/O, some resources, e.g., the secondary storage, would be shared among all the nodes. In other words, the secondary storage would be disaggregated.

Staying with our disaggregated storage example, let's look at the advantages and disadvantages. The first advantage is that we can optimize nodes for the common

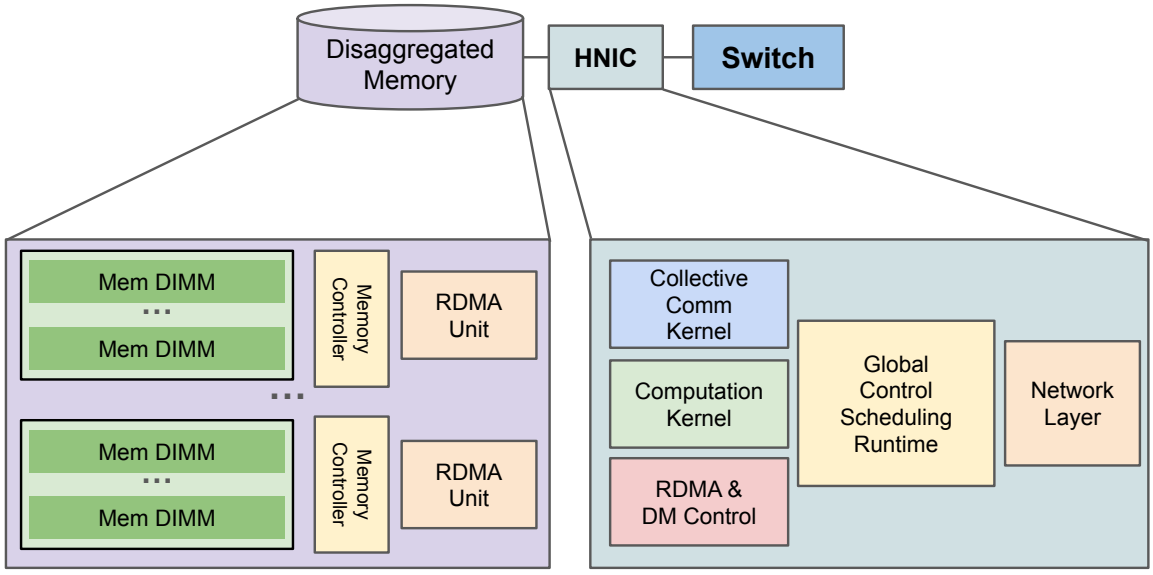


Figure 7·3: Global Control (GCU) and Disaggregated Memory (DM) Architecture

case. If nodes rarely need high-performance secondary storage, then it can be eliminated. The second advantage is that the disaggregated components themselves can be optimized. Building on economies of scale, large secondary storage can be organized in various ways (e.g., RAID types). Products can be built around, say, Network Attached Storage (NAS). A third advantage is that queueing delay is reduced, as is true for any system with multiple servers. A disadvantage is that peak performance decreases as there is added latency as accesses are through the network (remote versus local).

A short step away from separation of components by type is having heterogeneous servers. Let's assume that a compute cluster needs to support various types of application loads, each of which is optimally served by a different set of capabilities. These capabilities might be having many GPUs per node, or having large memories per node, or having a particular type of interconnect. One option is to have multiple different servers each of which handles its particular workload type. Another option is to be able to "mix-and-match" nodes of different types.

Disaggregated systems are currently undergoing much study (Guo et al., 2022c; Ke et al., 2022; Ewais and Chow, 2023; Shen et al., 2023; Gao et al., 2016). A survey (Blagodurov et al., 2021) describes disaggregated memory in data centers: "Compute units are decoupled from the memory hierarchy, with all components connected by the datacenter fabric." It gives examples especially of HPE's "The Machine" and the emerging CXL standard. A recent workshop had a number of overviews, concentrating on system-wide connectivity with embedded photonics (Bergman, 2023), composable disaggregated infrastructure though the OpenFabrics Management Framework (Aguilera et al., 2023), and commercial hardware switching support for CXL (Jiang, 2023).

7.2.1 Disaggregated Memory

Disaggregated memory architecture represents a new form of approach in computing, with the aim of optimizing memory utilization and performance. Traditional memory architectures often encounter challenges in scalability and efficiency, particularly as the demand for larger memory capacities and higher performance increases. Disaggregated memory addresses these challenges by decoupling memory resources from individual compute nodes, allowing for more flexible and efficient memory management.

The system can be categorized into two components: compute nodes (CNs) and memory nodes (MNs) (Kwon and Rhu, 2018; Kwon and Rhu, 2019). Compute nodes provide high-performance processors but have limited memory capacity, while memory nodes supply numerous high-capacity DRAM devices. Compute nodes in the distributed system access memory resources via a high-speed network, using protocols such as RDMA (Remote Direct Memory Access) or specialized interconnects like InfiniBand or Ethernet with RDMA support (RoCE).

The key components of disaggregated memory include the RDMA unit, memory

controller, and multiple memory Dual Inline Memory Modules (DIMMs) for parallel memory accesses. RDMA enables direct memory access, bypassing the CPU and offloading data transfer tasks to the NIC, thereby significantly reducing overhead and latency. The memory controller is responsible for coordinating and managing memory resources, ensuring efficiency and parallel access to multiple memory DIMMs. This parallelization capability is crucial for maximizing memory bandwidth and enhancing overall system performance.

7.2.2 Ring-Based collective communication

The Nvidia Collective Communication Library (Nvidia, 2024b; Nvidia, 2024a) facilitates efficient collective communication among clusters of GPUs. One of its implementations utilizes a ring-based logical topology to execute collective communication tasks such as Broadcast, All-Reduce, All-Gather, and Reduce-Scatter. In this setup, each GPU is connected to its adjacent GPUs in a logical ring, forming a circular pathway for data transfer. Within the ring, data flows sequentially from one GPU to the next. This ring-based approach minimizes communication overhead by reducing the number of direct connections required among GPUs. It ensures that each GPU communicates solely with its neighboring GPUs, streamlining data transfer complexity and enabling scalable communications with an increasing number of GPUs.

Figure 7·4 demonstrates the workflow of Reduce-Scatter using a ring-based logical topology with a cluster of four GPUs forming a circular communication pathway. Initially, each GPU begins with its own data chunk, with the same buffer size allocated for the Reduce-Scatter operator. The first step involves passing data around the ring, with each GPU sending its data to the next GPU in the sequence. For example, GPU0 sends its data to GPU1, GPU1 to GPU2, and so forth. Upon receiving data, each GPU applies reduction operations such as summation, maximum, or minimum, to combine the incoming data with its current data. For instance, when GPU0 receives

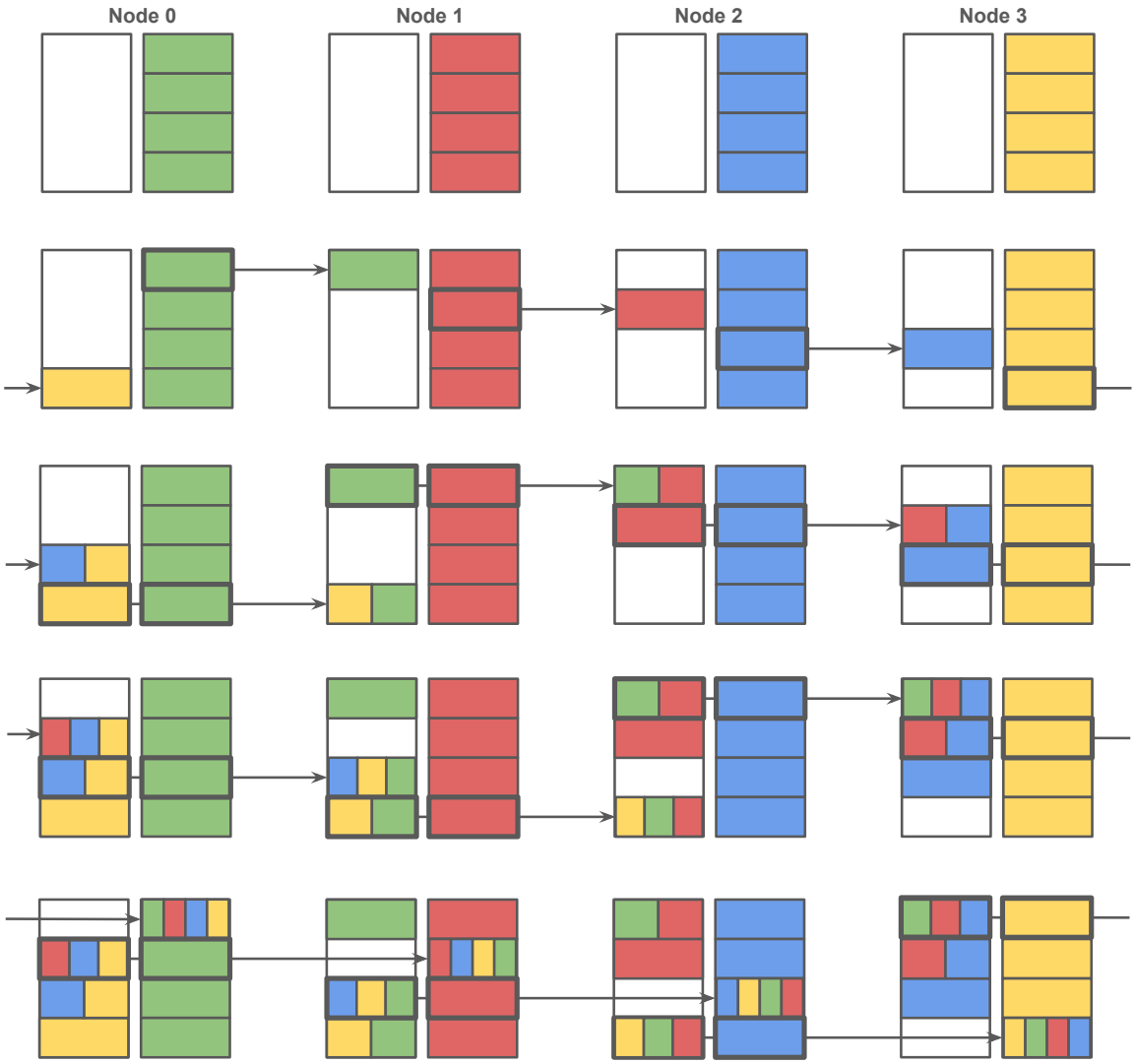


Figure 7·4: Ring-Based NCCL Reduce-Scatter Workflow.

data from GPU3, it performs the reduction operation on the received data along with its own data, and then forwards the result to GPU1. This process is replicated across all nodes in the system. Subsequent steps continue circulating data around the ring, with each GPU applying the reduction operation upon receiving new data. After completing a full loop around the ring, each GPU contains a portion of the overall reduced data. This setup enables further parallel processing, as each GPU independently operates on its designated portion.

Figure 7·5: Ring-Based NCCL All-Gather Workflow.

Figure 7·5 illustrates the workflow of All-Gather using a ring-based logical topology. Initially, each GPU begins with its own data segment. In the first step, each GPU sends its data segment to the next GPU in the ring while simultaneously receiving data from its preceding neighbor in the ring. For example, GPU0 sends its segment to GPU1, GPU1 sends its data to GPU2, and so forth. This process is replicated across all nodes in the system. As data segments are received, each GPU accumulates them in a buffer, eventually containing the complete collection of data from all GPUs. Subsequent steps involve transferring data around the ring, with each

GPU passing the previously received data to the next node. This circulation continues until completing a full loop, ensuring that every GPU has received all data segments from all other GPUs. As a result, each GPU possesses a concatenated collection of data segments from every GPU in the ring.

All-Reduce achieves the desired outcome where each node ends up with the same complete dataset derived from the combination and reduction of all nodes' contributions. All-Reduce collective communication involves combining Reduce-Scatter followed by All-Gather. Combining Figure 7·4 and Figure 7·5 illustrates the full steps of performing All-Reduce, with each node's assigned data chunk circling around the system nodes twice.

7.3 Heterogeneous Global Control and Disaggregated Memory System

7.3.1 Disaggregated Memory with Switch

In the baseline system, such as a GPU cluster or when utilizing a CPU as an offload engine, parameters are divided and distributed among the leaf nodes within the system. To perform each step of forward and backward propagation, global communication methods like All-Gather and Reduce-Scatter are employed to collect parameters in each leaf node or gather corresponding partition gradients. With our proposal of global control, we bring data closer to the control unit, enhancing system and application scheduling efficiency through data dispatch.

As depicted in Figure 7·2, a centralized disaggregated memory is connected to the headless NIC (HNIC) within the switch. The controlling kernel in the HNIC retrieves the corresponding parameters from the disaggregated memory using the memory controller on the HNIC. Instead of each leaf node issuing an All-Gather communication operator to gather data, a parameter request signal from the leaf node

GPU is sent to the global control HNIC for data retrieval. As the HNIC handles the data request, parameters fetched from the disaggregated memory are broadcast to each node with reduced communication workload compared to All-Gather. The same approach is applied to gradients with a Reduce-Scatter. As each leaf node generates its corresponding portion of the gradients, these gradients are sent to the HNIC for reduction. Subsequently, the compute unit on the HNIC computes the optimizer, calculating the updated parameters for the next iteration. These updated parameters are written to the disaggregated memory using the HNIC's memory controller for the next iteration's data request.

Figure 7·3 depicts the architecture of the HNIC connected to the switch and the disaggregated memory unit linked to the HNIC. The RDMA unit manages the RDMA data fetch requests from the leaf nodes. With the memory controller attached to the RDMA unit, multiple banks of memory DIMMs are organized to facilitate parallel data fetching with high bandwidth. By attaching multiple disaggregated memory units to the HNIC, the system can scale its parameter storage size according to the system scale and model requirements. We provide an example of a potential disaggregated memory configuration. The Intel Server Board S2600WF supports up to 24 DDR DIMM slots, with DDR5 DIMMs capable of reaching 64 GB/s and capacities ranging from 64 GBs to 512 GBs per DDRM memory. The total memory capacity can thus reach up to 12 TBs per node.

To evaluate the performance of the disaggregated memory, we can attach multiple instances of a DM to the HNIC. Combined with ZeRO, the number of data parallel processes equals the number of instances, with each instance storing a partition of the overall parameters. This aligns with GPU-based ZeRO, where each GPU stores a portion of the overall parameters. The RDMA engine and memory controller fetch each partition in parallel from each instance of a DM.

7.3.2 Global Control with Switch

With global control at the switch level, system control and application scheduling can be managed by the centralized switch. However, currently, there isn't a programmable switch that can provide such capabilities. We therefore propose attaching an HNIC to a dedicated port of the switch. As packets flow through the switch, the leaf node's request signals or data will be routed to the HNIC for processing and scheduling. The control kernel on the HNIC keeps track of the data and signals flowing through the entire network from each leaf node. Utilizing this information, the global control on the HNIC understands the status of each node and its corresponding application phase.

With this global control, the system no longer requires global synchronized communication patterns like all-gather and reduce-scatter, which necessitate all leaf nodes waiting for each other to start and end at a global barrier. As the attached disaggregated memory stores model parameters, the global synchronized communication can be swapped to asynchronous broadcast and reduce. Previously, all-gather gathered the distributed partitioned parameters from each leaf node and synchronized between each other before forwarding them to the next propagation computation. With GCU and DM, this is swapped for synchronized broadcast. The parameters are stored in disaggregated memory attached to global control. When a GPU is ready for the next propagation computation, it sends a signal to the global control requesting the corresponding data partition. The HNIC fetches the data partition and forwards it to the leaf node. With the GCU keeping track of each node's status and application phase, this data forwarding is asynchronous, with each GPU potentially fetching data at different times. This avoids leaf nodes waiting for each other with a global barrier. To achieve this, a global barrier table is maintained in the HNIC to ensure that the asynchronous data fetching from leaf nodes occurs within the same iteration.

As each leaf node generates gradients after backward propagation, reduce-scatter is required to reduce each partition's gradients. With GCU and DM, reduce-scatter is replaced with reduction. As mentioned earlier, leaf nodes execute the application pipeline asynchronously with different timestamps. Correspondingly, the gradients are generated asynchronously as well. The generated gradients are sent to the global HNIC right after each backward propagation stage. The reduction in the HNIC is executed as each leaf node's gradients arrive at the HNIC. To achieve this, a vector of partition gradients is maintained in the HNIC, with each element representing a partition of the parameter's gradients. Since leaf nodes might be in different execution pipeline stages with different parameter partition gradients, the corresponding partition will update the corresponding index of the vector of partition gradients. Once all partition gradients are received from each leaf node and reduced, the final gradients are forwarded to the parameter update computation kernel for parameter update computation. After the parameter update, the updated partition parameter is forwarded to the disaggregated memory by the memory controller in HNIC for the next iteration.

The HNIC GCU keeps track of each leaf node's status, including its parameter data request and gradient generation. It maintains a global barrier table and global gradient vector, ensuring that the maximum timestamp difference between leaf nodes is less than one iteration, thereby ensuring error-free execution of the application. This approach allows leaf nodes to proceed without waiting for each other for synchronized global communication or global barrier, thereby minimizing both computation and communication overlap to maximize efficiency.

Figure 7·3 depicts the architecture of the HNIC attached to the switch. The HNIC includes a network layer responsible for handling the transport layer protocols between switches. The global control and scheduling unit on the HNIC track requests

signals from leaf nodes, RDMA data fetching from leaf nodes, and the ongoing packet flow through the switch. As this information is collected at runtime, the global control at the switch level captures the current phase of the application and the status of each leaf node. The runtime manages asynchronous timestamp requests from every node and monitors the global status of each leaf node, ensuring that the asynchronous difference between each node is less than one iteration. Through delayed parameter updates, the updated parameter is only available for one iteration ahead. If the difference exceeds one iteration, the optimizer's updated parameter will not be able to provide parameters beyond one iteration. Additionally, in systems utilizing multi-level switch configurations, there may be sub-switches responsible for handling sub-global regions of the system, as shown in Figure 7·9. The global control kernel can be reconfigured for such sub-region switches. For example, by combining Data Parallel, Model (Tensor) Parallel, and Pipeline Parallel, model (tensor) parallel and pipeline parallel processes can be mapped to the sub-region, while the global switch connects different data parallel processes. The sub-region switch manages control and scheduling for processes in the model parallel, along with data exchange between MP.

In addition to the runtime, the selective communication kernel manages gradient reduction (Gradient Reduce-Scatter in the baseline). A global reduction table is maintained in the kernel to track the reduction of different partition gradients. The gradients are received asynchronously by the HNIC with different timestamps. Once all gradients for the corresponding partition are received, the reduced gradients are sent to the computation kernel for optimization with parameter updates. This computation kernel can also be relocated to an attached GPU or CPU to leverage computation resources. After parameter update, the parameters for the next iteration are ready to be used and forwarded to the disaggregated memory for RDMA data fetch. The RDMA and DM control handle the leaf node's RDMA data fetching

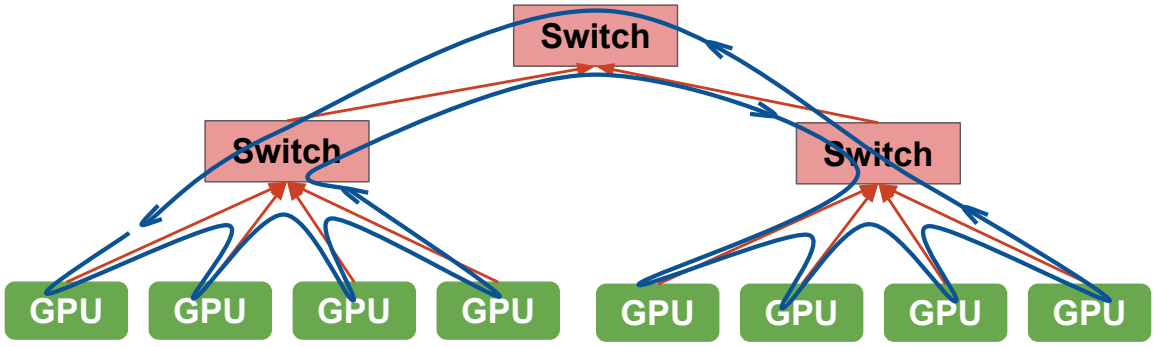


Figure 7·6: Ring-Based Collectives Topology in NCCL

request and coordinate with the RDMA unit in the disaggregated memory.

7.3.3 Global Control and Disaggregated Memory Collective Communication

Using ZeRO as the model training parallel strategy involves partitioning and distributing parameters across the system, with each leaf node containing a partition of the parameters. Consequently, an All-Gather operation is necessary to gather the corresponding partition's parameters for both forward and backward propagation. After each portion of the backward propagation, the partition's gradients are generated by each leaf node through its data parallel process. Reduce-Scatter is then employed to gather the corresponding portion of gradients, which are used to update the matching parameter portion. These global collective communications necessitate global communication patterns. Ring-based or tree-based collective patterns are commonly used to gather or reduce the data distributed among the leaf nodes in the system.

Figure 7·6 illustrates an example of the Nvidia Collective Communication Library (NCCL) ring-based collective with a two-level switch GPU cluster. In this example, the distributed GPUs are interconnected through switches in a ring fashion. For All-Gather operations, each data chunk unit traverses every node once in the ring, ensuring that all data chunks are gathered to each node once the process is complete.

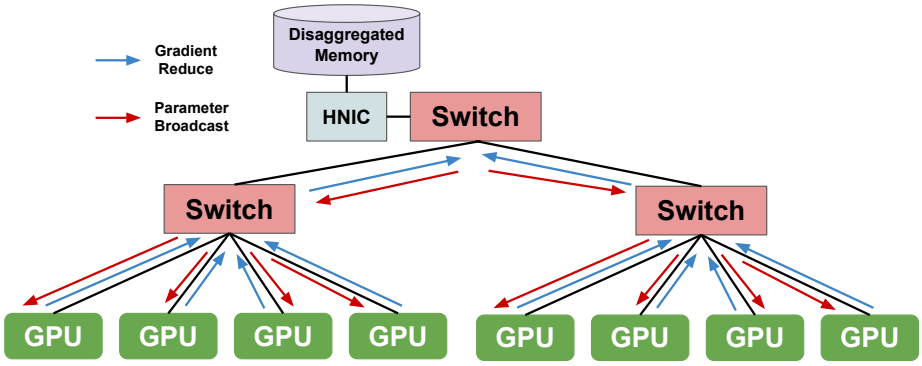


Figure 7·7: Global Control with Disaggregated Memory System Collective Communication Topology

Reduce-Scatter follows a similar pattern, where each data partition traverses each node, reducing the data until it reaches the destination node.

The heterogeneous GCU with DM system enhances the switch with headless functionality and migrates the leaf node's local control to the global switch. The DM serves as model parameter storage near the GCU, leading to higher system control efficiency and simplified data dispatch to the leaf nodes for computation. To distribute parameters to each leaf node for forward and backward propagation, a parameter broadcast is employed to dispatch parameters to the system. Compared to the baseline system using a ring topology with the All-Gather communication pattern, parameter broadcast not only reduces communication volume but also minimizes the number of hops data traverses to reach every system node in a ring topology. Similarly, after backward propagation, the baseline system uses the Reduce-Scatter communication pattern to reduce the corresponding partition of gradients for each leaf node. In the GCU-DM system, a global gradient reduction is conducted to reduce the scattered gradients in the system, as shown in Figure 7·7. Following global reduction, the HNIC's computation kernel (or computation device attached to the HNIC) computes the updated parameter for the next iteration using an optimizer. The updated parameter is then stored in the disaggregated memory. The global

gradient reduction results in less communication workload and fewer communication hops compared to Reduce-Scatter in a ring topology.

7.3.4 System Control, Scheduling and Optimization Techniques

The previous section provided an overview of the heterogeneous global control and disaggregated memory system. This section will delve into the detailed workflow and pipeline stages of the system components, including GPU, CPU, SmartNIC, HNIC GCU, and DM. It is important to note that the goal of the system is not to propose an entirely new system to replace GPU-based clusters. Instead, we are enhancing the existing system by introducing new application-level pipeline stages in GPU, CPU, and NICs to support heterogeneous device cooperation with global control and disaggregated memory support. Therefore, there will not be significant modifications to the existing system. We will add new pipeline stages at the software level and implement corresponding hardware logic in SmartNICs and HNICs to support global offload and data fetching from global disaggregated memory. For GPU, the computation stages will remain the same, but the communication operators will be swapped with new stages that issue data requesting signals to the global control and disaggregated memory. The SmartNICs and HNICs attached to the switch will handle communication and data fetching. As data arrives at the GPU, it will be ready to proceed with computation. The local SmartNIC collaborates with the HNIC to enable data prefetching and buffering, facilitating higher computation and communication overlap and reducing pipeline bubbles. In the following subsections, we will divide the application into phases and explain the workflow and pipeline stages of the heterogeneous devices.

1. Forward propagation and backward propagation

As in the previous section, ZeRO partitions the model parameters into N partitions, where N represents the number of data-parallel processes. Similarly, the dis-

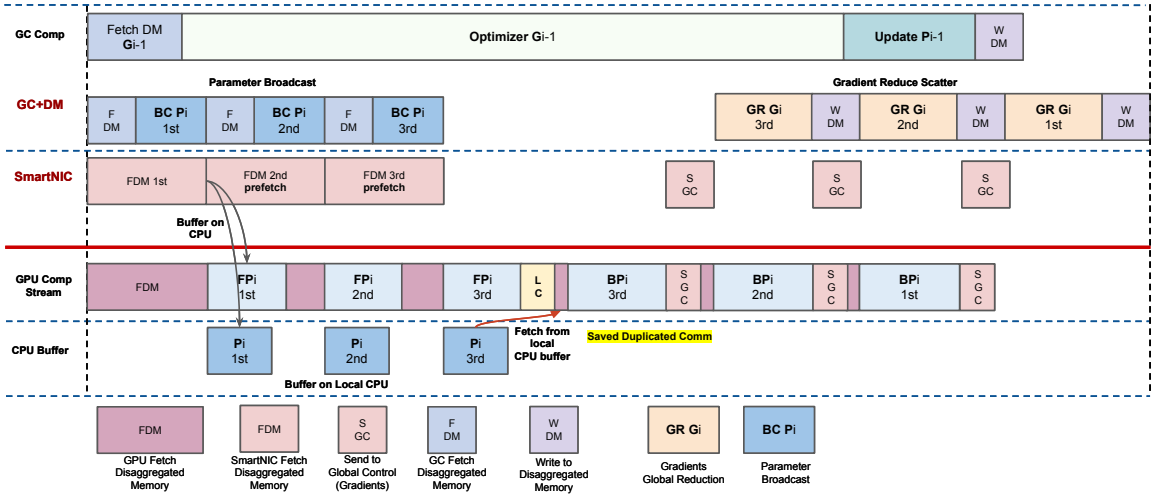


Figure 7·8: Heterogeneous Global Control and Disaggregated Memory System Pipeline. (GCU: global control; DM: disaggregated memory attached to global control)

aggregated memory attached to the HNIC is instantiated into N instances, aligned with the number of data parallel processes. Due to constraints on the GPU HBM memory, both the forward and backward propagation processes are partitioned into N parts, with the corresponding portion of parameter data fetching required before forward and backward propagation computation begins. This partitioning ensures efficient utilization of resources and enables parallel processing across the system.

Figure 7·8 illustrates the GPU compute stream, outlining the GPU pipeline. In this figure, three data parallel processes are used as an example, so the forward propagation and backward propagation are divided into three parts accordingly. In the baseline GPU system, the All-Gather communication operator is used to gather parameters from every other GPU to assemble the parameters required for the first forward propagation. In the GCU-DM system, we introduce the *GPU Fetch Disaggregated Memory* (FDM) pipeline stage. Here, the GPU emits a data fetch signal to the system to request data. This signal can be considered as an RDMA read operation from the global DM. The signal is initially received by the local Smart-

NIC, which handles the parameter fetching from the system's DM. Communication is managed by the local SmartNIC instead of the GPU. Once the data arrives, the parameter is written directly to the GPU by the SmartNIC along with a notification. This approach allows the GPU to conserve resources that would otherwise be used to handle the communication operator. The GPU initiates the data requesting signal and allocates sufficient buffer size to receive parameters written by the SmartNIC. This setup optimizes the data fetching process and enhances overall system efficiency.

When the local SmartNIC receives the RDMA read request signal from the local GPU, it collaborates with the global control and the disaggregated memory to complete the data fetching process. The communication operator is initiated on the SmartNIC, triggering the RDMA read request (FDM stage) to the GCU. Upon receiving the signal, the FDM stage is activated, initiating the RDMA kernel with a read request to the DM. The memory controller processes the data fetching from the memory DIMM and returns it to the RDMA unit. Since each GPU is requesting the same chunk of parameters for the first portion of the forward propagation, the parameter broadcast communication from the GCU distributes the parameter portion to each local SmartNIC. Subsequently, the local SmartNIC manages the data writing to the corresponding allocated buffer of the local GPU.

It is important to note that in the baseline system, the parameter All-Gather is replaced by parameter broadcast, resulting in reduced communication volume in the network and fewer data hops. Additionally, the GCU may receive RDMA read requests asynchronously, as there is no requirement for global barrier synchronization among leaf nodes with different timestamps. The GCU handles such asynchronous requests. Furthermore, the HNIC buffers the requested data to prevent multiple RDMA data fetches from the disaggregated memory, thus avoiding performance degradation. This optimized process ensures efficient data fetching and enhances system perfor-

mance.

After the SmartNIC receives the requested parameters from the GCU, the data are written directly to the GPU's buffer, accompanied by a notification to proceed with the forward propagation computation. This streamlined process ensures that the GPU can seamlessly access the required parameters without additional latency or overhead. A similar workflow is repeated for the backward propagation stage, where the relevant parameter portions are gathered to facilitate the backward computation. This ensures that the necessary parameter data is available to the GPU for the completion of the backward propagation process. Overall, this approach enhances the efficiency and effectiveness of both forward and backward propagation stages in the system.

2. Parameter Prefetch

After each forward and backward propagation, the GPU initiates the next portion of parameter fetching request. However, during the computation phases, the network remains idle, presenting an opportunity for optimization. To address this, we introduce parameter prefetching techniques. In this approach, the local SmartNIC and global control collaborate to prefetch data ahead of the GPU's request. After the SmartNIC completes writing the requested parameters to the GPU's buffer, it initiates the prefetching process for the next portion of parameters. This prefetching is triggered autonomously by the SmartNIC itself once the current data fetching is completed. Upon receiving the parameter prefetching request from the local SmartNIC, the global control ensures that the data is buffered on the SmartNIC, awaiting the GPU's request. When the GPU finishes its propagation computation, it sends the next portion of the *FDM* signal to the local SmartNIC. Since the data is already available on the local SmartNIC due to prefetching, it can be immediately written to the GPU's buffer without delay. This prefetching technique effectively overlaps

computation and communication, minimizing the number of pipeline bubbles and reducing the GPU's data waiting time.

3. Parameter Buffering

Before each partition of the forward propagation, communication is necessary to gather the corresponding parameters required for that specific portion of the propagation. Similarly, for backward propagation, another round of communication is needed to gather the partition parameters, causing a duplication of communication efforts that can stall the backward propagation pipeline. To address this issue, we propose parameter buffering on the CPU as an offload engine during forward propagation, with the aim of reusing the buffered parameters for the backward pass.

As shown in Figure 7·8, after the SmartNIC completes the first FDM stage, it forwards the parameters for the first partition to the GPU for the initial forward pass. Simultaneously, the SmartNIC also forwards the same data to the CPU for buffering. The decision to use the CPU as an offload engine is due to its capability of handling larger memory capacities, ensuring that all portions of the parameters can be buffered efficiently. After all forward propagation steps are completed, just before initiating the backward pass, the GPU sends the FDM signal to the SmartNIC. Since the SmartNIC has knowledge of the buffered parameters on the CPU, it instructs the CPU to migrate the corresponding portion of the parameters to the GPU's allocated buffer. This buffering technique effectively eliminates duplicated communication volume and reduces data transfer energy consumption.

4. Gradient Reduction

In ZeRO, gradients are generated by each data-parallel process, with each process owning a different partition of the parameters. Consequently, the gradients corresponding to these parameters need to be gathered using a Reduce-Scatter. However, in the proposed system, the parameters are stored in a DM attached to the top-level

switch, rather than being distributed among each leaf node. As a result, a *Global Reduction* operation is required to gather the gradients generated across the distributed nodes and reduce them for updating the parameters in the next iteration.

In the pipeline illustrated in Figure 7·8, once the GPU completes backward propagation, the gradients are generated and then sent to the local SmartNIC through a *Send to Global Control* (SGC) stage. The GPU writes this data to the allocated buffer on the SmartNIC. Once the data transfer is completed, the gradients are forwarded to the GCU for global reduction. Within the global control, a table is maintained to record the received gradients from the leaf nodes. Since each node might send its gradients asynchronously, the global control ensures that gradients belonging to the same portion of the parameter are reduced to the same buffer. Given that leaf nodes operate asynchronously and are not in the same phase, gradients belonging to different portions of the parameter are managed using multiple buffers, updated by leaf nodes at different pipeline stages. When the buffer table indicates that gradients from all leaf nodes have been received and reduced, the resulting gradient is written to the disaggregated memory for further parameter update computation, and the buffer is released.

5. Parameter Update

In ZeRO, delayed parameter updates are implemented to allow parallelization of forward and backward passes with parameter updates. This approach eliminates dependencies between these processes, resulting in higher efficiency. We enable global control with a computation option, which is focused on parameter update computation. This can be achieved by attaching another computation device, such as a GPU, to the HNIC, or by enabling a computation kernel directly on the HNIC. Attaching a GPU to the HNIC provides higher computational power, thus preventing parameter updates from becoming a bottleneck to the entire system. The computation device

fetches the gradients generated from the last iteration from the disaggregated memory. After computing the updated parameter using an optimizer, the parameters are written back to the disaggregated memory for the next iteration.

7.4 Heterogeneous GCU and DM System

In today's dynamic landscape of machine learning, tackling the challenges of faster training times and handling larger datasets has become increasingly crucial. Traditional single-threaded or single-machine training methods often struggle to keep up with the scale and complexity of modern machine learning tasks. To overcome these hurdles, researchers and engineers are turning towards parallelization techniques, with 3D parallel training emerging as a particularly promising approach for maximizing efficiency and scalability. 3D parallel training integrates data parallelism, model parallelism (tensor parallelism), and pipeline parallelism. By leveraging these three parallelization techniques, systems can effectively train large machine learning models with enhanced efficiency.

Data parallelism involves splitting datasets into smaller chunks and distributing them among data parallel processes. Each process possesses a full copy of the model and executes identical operations on its assigned data portion. Subsequently, the gradients generated by each process are aggregated and utilized to update the model parameters. These updated parameters are then redistributed to the processes for the subsequent iteration.

Model parallelism, or tensor parallelism, comes into play when the model size surpasses the memory capacity of individual devices. In such cases, the model is partitioned across multiple devices, enabling the training of larger models.

Pipeline parallelism divides the training process into stages or partitions the model into layers, assigning each segment to a distinct device or processor. Data batches

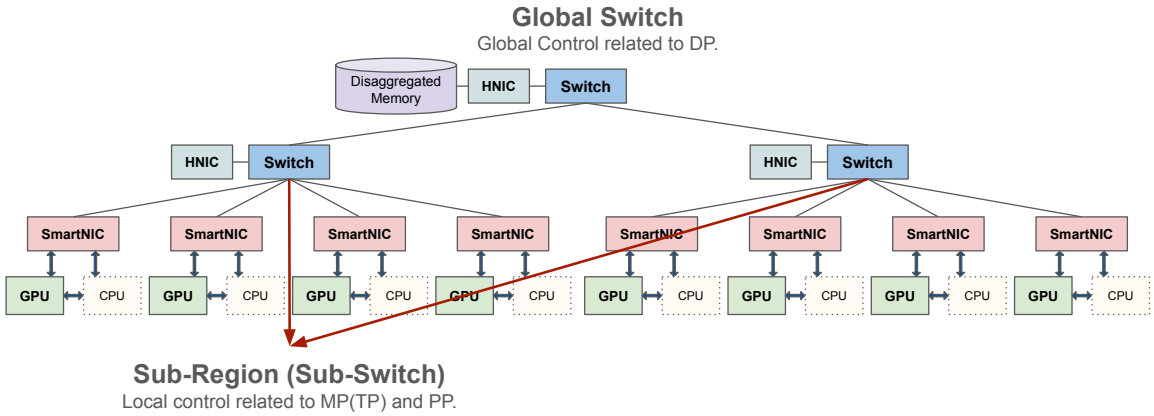


Figure 7·9: GC-DM System with Multiple Regions. The global switch, which controls the entire system and application, and the sub-region switches manage specific sub-regions of the local nodes.

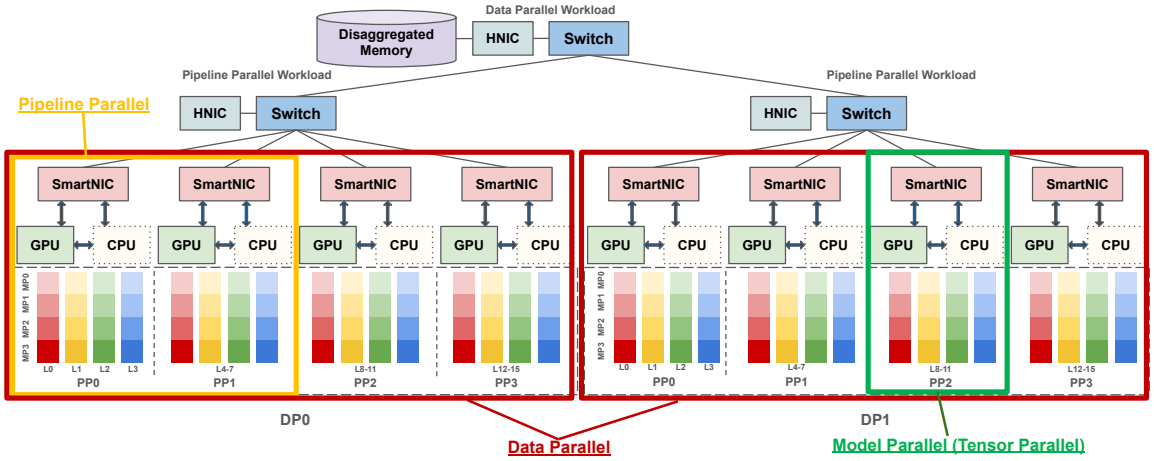


Figure 7·10: The GCU-DM System with 3D Parallel Training as an Example. The global switch handles data parallel processes, and the sub-region switch manages pipeline parallel processes.

are subdivided into micro-batches and fed through the pipeline in a sequential manner. This setup creates a pipeline where different stages are processed concurrently, minimizing idle periods and increasing overall throughput.

As depicted in Figures 7·9 and 7·10, our system architecture divides the system switches into two main components: the global switch and the sub-region switches. The global switch assumes control over the entire system and application, while the

sub-region switches manage specific sub-regions of the local nodes. In the context of 3D parallel training, the global switch oversees the data parallel processes, treating each sub-region as a distinct data parallel process. Meanwhile, the sub-region switches are responsible for managing the pipeline parallel processes within their respective regions. The tensor or model parallel process is situated within each leaf node. This arrangement is preferred for tensor parallelism due to its requirement for more frequent communication, which is efficiently managed at the local node level.

The global switch assumes responsibility for application control and parameter fetching from the attached disaggregated memory. As data are fetched, the sub-switch dispatches the parameters to the leaf nodes based on their pipeline stage. By delegating communication tasks between pipeline stages to the sub-switches, the system optimizes communication flow. Data movement is confined to within specific regions, minimizing unnecessary hops and enhancing efficiency. This topology-aware approach ensures that communication within the system is streamlined and optimized.

7.5 Evaluation

In this section, we provide an analysis comparing NCCL ring-based collectives with our heterogeneous global control and disaggregated memory system. We examine three collective operations: All-Reduce, parameter gathering collectives, and gradient reduction collectives. Our system demonstrates a significant reduction in communication workload, i.e., approximately an order of magnitude less than that of a ring-based cluster.

7.5.1 All-Reduce

In the ring-based collective communication model, each node operates under the assumption of a direct connection within a ring topology. Instead of employing All-Reduce operations, our system uses a global Reduce with Broadcast approach to

achieve comparable results. This section provides a comparison between the ring-based collective communication and our system.

In ring-based All-Reduce, N hosts are arranged into a logical ring, and each host sends to its neighbor $(N-1)$ messages, each of size Z/P where Z is the number of elements to be reduced and P is the data chunk size that each packet carries. As the message traverses the whole system twice, so the total amount of data sent by each host is $Z/P * 2 * (N-1)$. The total amount of the packet volume in the system is $Z/P * 2 * N(N-1)$ which is approximately $Z/P * 2 * N^2$.

Using the system shown in Figure 7·6 as an example two level of switches are used to connect a cluster of devices. The total number of the hops that packets need to traverse is $2 * H$ between nodes with extra $2 * S * H$ between switches where H represents the unit of each hop and S represents the number of the second level switch connecting to the first level switch. So the total number of the communication volume with hops is $Z/P * 2 * (N^2 * 2 * H + 2 * S * H)$ which is $Z/P * 4(N^2H + SH)$.

In our system, the collective communication is switch-centric as seen in Figure 7·7. The sub-region switch collects data from leaf nodes and the global switch reduces data from sub-region switches. The All-Reduce consists of Reduction with a Broadcast to leaf nodes. N hosts send the Z/P messages to the sub-region switch where the sub-region switch's HNIC handles data reduction. The sub-reduced data are forwarded from sub-regions to the global region switch for final data reduction. In total, there are $Z/P * (N+S)$ messages for reduce. As the global switch reduces data, Broadcast communication is used to distribute the final results back to the leaf node with the same amount of as Reduce. The total number of messages is $Z/P * 2 * (N+S)$. The total communication volume is $Z/P * 2(N+S) * H$ where H represents the hop that connects each device or switch.

7.5.2 Parameter gathering and gradients reduction collectives

The ring-based All-Reduce consists of a Reduce-Scatter followed by an All-Gather. So each of the two collective communications can be viewed as half of the communication volume of the All-Reduce which is $Z/P * 2(N^2H + SH)$. In our system, the All-Gather is replaced with a Broadcast, and Reduce-Scatter is swapped by a global Reduce with each taking $Z/P * (N + S) * H$ of the total communication volume.

7.6 Summary with Discussion of HW/SW Codesign

In this chapter, we introduce a global control system with disaggregated memory, integrating a SmartNIC with the switch to empower it with global control and computation capabilities. This system supports fine-grained software-hardware co-design at the system level, enhancing global application control. It reduces system barriers and synchronization overhead while increasing the overlap between communication and computation, leading to higher model FLOPs utilization and reduced communication workload.

Software: On the software side, we implement an efficient global control ML training workflow based on the ZeRO parallel strategy, featuring centralized data dispatch.

Hardware: On the hardware side, we present an HNIC design with disaggregated memory and SmartNIC optimization techniques.

Codesign: By combining these elements, we implement hardware based on ZeRO and global control flow, optimizing software and model mapping according to the system hardware. This integrated approach maximizes efficiency and performance in machine learning model training.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The rapid expansion of AI applications, particularly in the realm of large-scale deep learning, has become a dominant force in contemporary AI. The emergence of large models is highlighted for their remarkable improvement in performance through giant model sizes. However, with the increasing size of these models, there arise significant challenges in terms of computational resources and scalability. The communication, memory bandwidth, and computation efficiency are challenging the computation cluster's performance. The memory wall has made this ever-growing computation cluster scaling issue even worse where current hardware limitations are struggling to keep pace with the exponential growth in model size.

In this dissertation, we proposed software-hardware codesign of SmartNIC-based heterogeneous high performance computing systems with machine learning applications as case studies. The system aims to improve performance and efficiency, lower power consumption and budget, and mitigate data exchange overheads and overlapping computation and communication. We explore the system design space in four steps with each step advancing the performance and the capabilities of the system. The case studies chosen to demonstrate the codesign approach encompass a range of advanced machine learning applications including graph neural networks, deep learning recommendation models, large language models and further generalized large machine learning models. Our thesis is that **high-performance and high-efficiency**

inference and training of large machine learning models can be achieved by software-hardware codesign of SmartNIC-based heterogeneous high-performance computing system.

The first exploration step explores the practical functionalities of emerging SmartNICs, which have evolved into powerful yet intricate and diverse components. These SmartNICs are engineered to handle a broad spectrum of applications, integrating networking support for fundamental NIC tasks alongside GPUs, programmable logic, vector processing, HBM, and DDR memory. Furthermore, the components themselves exhibit heterogeneity. To demonstrate the utility of these varied components within SmartNICs, we utilize the Xilinx Versal ACAP platform. This platform amalgamates a traditional FPGA with a cluster of CPUs and a Coarse Grained Reconfigurable Array of hardware blocks. We showcase the versatility of these heterogeneous components through the application of a Graph Convolution Neural Network. GCN combines both regular and irregular computation, encompassing tasks such as irregular data access patterns, workload balancing, and hybrid computation patterns. This makes GCN an ideal application for illustrating the utilization of heterogeneous hardware components. Through the mapping of subgraphs based on their density onto corresponding hardware elements of the Versal ACAP, we exemplify the efficient utilization of the diverse hardware components within heterogeneous devices.

The second exploration step explores how the SmartNICs integrated efficiently into the CPU-centric nodes. This involves examining the use of distributed SmartNICs as an independent system, showcasing their ability to offload application control and integrate computation and communication. We utilize graph neural networks and deep neural networks as our applications. Our proposed system is a user-friendly framework for neural network inference on FPGA-centric SmartNICs (FCsN) capable of performing computation, communication, and control simultaneously. This

approach allows for flexible and fine-grained task creation, distribution, and execution across multiple SmartNIC devices, bypassing CPU intervention. As a result, computation latency is minimized by overlapping it with network communication, enabling streaming applications to run at line-rate and achieving high FPGA utilization and system-level performance. On the software side, FCsN employs a data-centric programming model with Python-based programming APIs. On the hardware side, it features a hardware-based SmartNIC runtime for CPU-detached scheduling, supporting high-performance execution of neural network kernels at line-rate. While the current FCsN framework focuses on neural network applications, it has the potential to evolve into a general framework applicable to many scientific applications that share similar basic kernel functions with neural networks.

In the third phase of our exploration, we focus on the integration of SmartNICs into GPU-centric nodes to enhance connectivity and performance. Our node design employs a software/hardware co-design strategy, utilizing SmartNIC capabilities to synchronize computation and communication. This design integrates GPUs as accelerators, with CPUs and NVMe as offload engines, and applies deep learning-based recommendation models and large language models as primary examples.

We present a software-hardware co-design paradigm tailored for a heterogeneous SmartNIC system, specifically crafted for scalable machine learning inference and training. This paradigm aims to improve system performance and efficiency while reducing power consumption, costs, and data exchange overhead. Acting as a vital intermediary layer, the SmartNIC facilitates seamless connectivity between GPUs and CPU offload engines, bridging the gap between distributed heterogeneous components within the system.

To optimize performance, we implement a suite of SmartNIC optimization techniques, including prefetching, caching, and SmartNIC computation kernels. These

techniques capitalize on data locality, reducing data movement, enhancing the overlap of computation and communication, decreasing memory access intensity, and ultimately maximizing GPU computation efficiency.

In the fourth exploration, we investigated the previous system with SmartNICs beyond computation offload with heterogeneous global control and disaggregated memory system. This setup employs a headless SmartNIC attached directly to a network switch, enabling a centralized system and application control. This represents an improvement over previous designs that lacked centralized global coordination.

This exploration also incorporates disaggregated memory linked to global control, providing several benefits. It enhances efficient application management and data distribution, reducing the need for extensive synchronization and minimizing barrier-related overhead. Additionally, this design lowers the communication workload, promoting better overlap between computation and communication tasks.

8.2 Future Work

In this section, we briefly discuss future directions. In this dissertation, we explore the design space of software-hardware SmartNIC-based heterogeneous high-performance systems in four steps, each pushing the exploration boundary further. Our current work involves enabling switches with global control by attaching a SmartNIC to dedicated ports. This ongoing project needs to be tested and evaluated with machine learning models.

The goal of this system is not to replace existing GPU-based systems but to augment GPU clusters with global control by adding execution pipeline stages to both the GPU and the switch. This requires significant engineering effort, including exploring GPU CUDA kernels and developing corresponding SmartNIC designs to provide global control and synchronization for the GPU cluster. Our future work

aims to enable this system for large machine learning applications, such as large language models.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv, https://arxiv.org/abs/1603.04467.

Agarap, A. F. (2019). Deep learning using rectified linear units (relu). arXiv, https://arxiv.org/abs/1803.08375.

Aguilera, M. K., Amaro, E., Amit, N., Hunhoff, E., Yelam, A., and Zellweger, G. (2023). Memory disaggregation: why now and what are the challenges. *SIGOPS Operating Systems Review*, 57(1):38–46.

Almási, G., Heidelberger, P., Archer, C. J., Martorell, X., Erway, C. C., Moreira, J. E., Steinmacher-Burow, B., and Zheng, Y. (2005). Optimization of mpi collective communication on bluegene/l systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, page 253–262, New York, NY, USA. Association for Computing Machinery.

AMD (2022). VCK5000 Versal Development Card. https://www.xilinx.com/products/boards-and-kits/vck5000.html.

Arai, J., Shiokawa, H., Yamamuro, T., Onizuka, M., and Iwamura, S. (2016). Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *IEEE International Parallel and Distributed Processing Symposium*, pages 22–31.

AWS (2019). Deliver high performance ml inference with aws inferentia. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf.

Barrett, B., Brightwell, R., Underwood, K., and Hemmert, K. S. (2012). Poster: Portals 4 network programming interface. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1467–1467.

Bergman, K. (2023). Scalable and Adaptable Architectures for AI/HPC Advancement. In *3rd International Workshop on RESource DISaggregation in High Performance Computing (RESDIS)*.

Blagodurov, S., Ignatowski, M., and Salapura, V. (2021). The Time is Ripe for Disaggregated Systems. *Computer Architecture Today*. https://www.sigarch.org/the-time-is-ripe-for-disaggregated-systems/.

Bojchevski, A. and Günnemann, S. (2018). Deep Gaussian embedding of graphs: Unsupervised inductive learning via ranking. arXiv, https://arxiv.org/abs/1707.03815.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95.

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

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Caulfield, A. M., Chung, E. S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., and Burger, D. (2016). A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13.

Chen, C.-C., Yang, C.-L., and Cheng, H.-Y. (2019). Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform. arXiv, https://arxiv.org/abs/1809.02839.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training deep nets with sublinear memory cost. arXiv, https://arxiv.org/abs/1604.06174.

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C.-J. (2019). Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on*

Knowledge Discovery and Data Mining, KDD '19, page 257–266, New York, NY, USA. Association for Computing Machinery.

Chiu, M. and Herbordt, M. (2009). Efficient filtering for molecular dynamics simulations. In *2009 International Conference on Field Programmable Logic and Applications*. doi: 10.1109/FPL15426.2009.

Chiu, M. and Herbordt, M. (2010). Molecular dynamics simulations on high performance reconfigurable computing systems. *ACM Transactions on Reconfigurable Technologies and Systems*, 3(4):1–37. doi: 10.1145/1862648.1862653.

Corradi, G. and Jensen, J. A. (2020). Real Time Synthetic Aperture and Plane Wave Ultrasound Imaging with the Xilinx VERSAL™ SIMD-VLIW Architecture. In *2020 IEEE International Ultrasonics Symposium (IUS)*, pages 1–4.

Covington, P., Adams, J., and Sargin, E. (2016). Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA.

De Sensi, D., Di Girolamo, S., Ashkboos, S., Li, S., and Hoefler, T. (2021). Flare: flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21. ACM.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv, https://arxiv.org/abs/1810.04805.

Di Girolamo, S., Kurth, A., Calotoiu, A., Benz, T., Schneider, T., Beránek, J., Benini, L., and Hoefler, T. (2021). A RISC-V in-network accelerator for flexible high-performance low-power packet processing. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 958–971.

Dror, R., Grossman, J., Mackenzie, K., Towles, B., Chow, E., Salmon, J., Young, C., Bank, J., Batson, B., Deneroff, M., Kuskin, J., Larson, R., Moraes, M., and Shaw, D. (2010). Exploiting 162-Nanosecond End-to-End Communication Latency on Anton. In *SC '10: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*.

Dwivedi, V. P., Joshi, C. K., Laurent, T., Bengio, Y., and Bresson, X. (2020). Benchmarking graph neural networks. *CoRR*, abs/2003.00982.

Eisenman, A., Naumov, M., Gardner, D., Smelyanskiy, M., Pupyrev, S., Hazelwood, K., Cidon, A., and Katti, S. (2019). Bandana: Using non-volatile memory for storing deep learning models. In Talwalkar, A., Smith, V., and Zaharia, M., editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 40–52.

Ewais, M. and Chow, P. (2023). Disaggregated memory in the datacenter: A survey. *IEEE Access*, 11:20688–20712.

Fang, J., Zhu, Z., Li, S., Su, H., Yu, Y., Zhou, J., and You, Y. (2023). Parallel training of pre-trained models via chunk-based dynamic memory management. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):304–315.

Faraj, A., Kumar, S., Smith, B., Mamidala, A., Gunnels, J., and Heidelberger, P. (2009). MPI collective communications on the Blue Gene/P supercomputer: algorithms and optimizations. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 489–490, New York, NY, USA. Association for Computing Machinery.

Fey, M. and Lenssen, J. E. (2019). Fast graph representation learning with PyTorch geometric. arXiv, https://arxiv.org/abs/1903.02428.

Forbes (2020). Understanding the NVIDIA DPU. https://www.forbes.com/sites/forbestechcouncil/2020/12/10/understanding-the-nvidia-dpu/.

Gaide, B., Gaitonde, D., Ravishankar, C., and Bauer, T. (2019). Xilinx Adaptive Compute Acceleration Platform: Versal^{TM} Architecture. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 84–93.

Gao, P. X., Narayan, A., Karandikar, S., Carreira, J., Han, S., Agarwal, R., Ratnasamy, S., and Shenker, S. (2016). Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, Savannah, GA. USENIX Association.

Gasteiger, J., Qian, C., and Günnemann, S. (2022). Influence-based mini-batching for graph neural networks. In Rieck, B. and Pascanu, R., editors, *Proceedings of the First Learning on Graphs Conference*, volume 198 of *Proceedings of Machine Learning Research*, pages 9:1–9:19. PMLR.

Geng, T., Li, A., Shi, R., Wu, C., Wang, T., Li, Y., Haghi, P., Tumeo, A., Che, S., Reinhardt, S., and Herbordt, M. C. (2020a). AWB-GCN: a graph convolutional network accelerator with runtime workload rebalancing. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 922–936.

Geng, T., Wang, T., Sanaullah, A., Yang, C., Patel, R., and Herbordt, M. (2018a). A Framework for Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters with Work and Weight Load Balancing. In *28th International Conference on Field Programmable Logic and Applications*, pages 394–3944.

Geng, T., Wang, T., Sanaullah, A., Yang, C., Xu, R., Patel, R., and Herbordt, M. (2018b). FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters. In *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 81–84.

Geng, T., Wang, T., Wu, C., Yang, C., Song, S. L., Li, A., and Herbordt, M. (2019a). LP-BNN: Ultra-low-Latency BNN Inference with Layer Parallelism. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160-052X, pages 9–16.

Geng, T., Wang, T., Wu, C., Yang, C., Wu, W., Li, A., and Herbordt, M. C. (2019b). O3BNN: an out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 461–472, New York, NY, USA. Association for Computing Machinery.

Geng, T., Wu, C., Tan, C., Fang, B., Li, A., and Herbordt, M. (2020b). CQNN: a CGRA-based QNN Framework. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7.

Geng, T., Wu, C., Tan, C., Xie, C., Guo, A., Haghi, P., He, S. Y., Li, J., Herbordt, M., and Li, A. (2021a). A Survey: Handling Irregularities in Neural Network Acceleration with FPGAs. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8.

Geng, T., Wu, C., Zhang, Y., Tan, C., Xie, C., You, H., Herbordt, M., Lin, Y., and Li, A. (2021b). I-GCN: A Graph Convolutional Network Accelerator with Runtime Locality Enhancement through Islandization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1051–1063, New York, NY, USA. Association for Computing Machinery.

George, A., Herbordt, M., Lam, H., Lawande, A., Sheng, J., and Yang, C. (2016). Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects. In *2016 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA*, pages 1–7. doi: 10.1109/ HPEC.2016. 7761639.

Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826.

Gomez-Uribe, C. A. and Hunt, N. (2016). The Netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems*, 6(4).

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2018). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv, https://arxiv.org/abs/1706.02677.

Graham, R. L., Bureddy, D., Lui, P., Rosenstock, H., Shainer, G., Bloch, G., Goldenerg, D., Dubman, M., Kotchubievsky, S., Koushnir, V., Levi, L., Margolin, A., Ronen, T., Shpiner, A., Wertheim, O., and Zahavi, E. (2016). Scalable hierarchical aggregation protocol (SHArP): a hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10.

Gu, Y. and Herbordt, M. (2007). FPGA-based multigrid computations for molecular dynamics simulations. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 117–126. doi: 10.1109/ FCCM.2007.42.

Guo, A., Geng, T., Zhang, Y., Haghi, P., Wu, C., Tan, C., Lin, Y., Li, A., and Herbordt, M. (2022a). A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 01–08.

Guo, A., Geng, T., Zhang, Y., Haghi, P., Wu, C., Tan, C., Lin, Y., Li, A., and Herbordt, M. (2022b). FCsN: An FPGA-Centric SmartNIC Framework for Neural Networks. In *IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines*.

Guo, A., Hao, Y., Wu, C., Haghi, P., Pan, Z., Si, M., Tao, D., Li, A., Herbordt, M., and Geng, T. (2023). Software-hardware co-design of heterogeneous SmartNIC system for recommendation models inference and training. In *Proceedings of the 37th International Conference on Supercomputing*, page 336–347.

Guo, Z., Shan, Y., Luo, X., Huang, Y., and Zhang, Y. (2022c). Clio: a hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433, New York, NY, USA. Association for Computing Machinery.

Gupta, U., Wu, C.-J., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Hempstead, M., Jia, B., Lee, H.-H. S., Malevich, A., Mudigere, D., Smelyanskiy, M., Xiong, L., and Zhang, X. (2020). The architectural implications of Facebook's DNN-based personalized recommendation. In *IEEE International Symposium on High Performance Computer Architecture*, pages 488–501.

Haghi, P., Geng, T., Guo, A., Wang, T., and Herbordt, M. (2020a). FP-AMG: Fpga-based acceleration framework for algebraic multigrid solvers. In *IEEE 28th Annual*

International Symposium on Field-Programmable Custom Computing Machines, pages 148–156.

Haghi, P., Guo, A., Geng, T., Broaddus, J., Schafer, D., Skjellum, A., and Herbordt, M. (2020b). A Reconfigurable Compute-in-the-Network FPGA Assistant for High-Level Collective Support with Distributed Matrix Multiply Case Study. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 159–164.

Haghi, P., Guo, A., Geng, T., Skjellum, A., and Herbordt, M. C. (2021). Workload Imbalance in HPC Applications: Effect on Performance of In-Network Processing. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8.

Haghi, P., Guo, A., Xiong, Q., Yang, C., Geng, T., Broaddus, J. T., Marshall, R., Schafer, D., Skjellum, A., and Herbordt, M. C. (2022). Reconfigurable switches for high performance and flexible MPI collectives. *Concurrency and Computation: Practice and Experience*, 34(6):e6769.

Haghi, P., Krska, W., Tan, C., Geng, T., Chen, P. H., Greenwood, C., Guo, A., Hines, T., Wu, C., Li, A., Skjellum, A., and Herbordt, M. (2023). FLASH: fpga-accelerated smart switches with GCN case study. In *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, page 450–462, New York, NY, USA. Association for Computing Machinery.

Haghi, P., Tan, C., Guo, A., Wu, C., Liu, D., Li, A., Skjellum, A., Geng, T., and Herbordt, M. (2024). SmartFuse: Reconfigurable Smart Switches to Accelerate Fused Collectives in HPC Applications. In *38th ACM International Conference on Supercomputing (ICS)*. DOI: 10.1145/3650200.3656616.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. (2018). Pipedream: Fast and efficient pipeline parallel dnn training. arXiv, https://arxiv.org/abs/1806.03377.

Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., and Wang, X. (2018). Applied machine learning at Facebook: a datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture*, pages 620–629.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Herbordt, M. (2019). Advancing OpenCL for FPGAs: Boosting Performance with Intel FPGA SDK for OpenCL Technology. *The Parallel Universe Magazine*, January(35):17–32.

Herbordt, M. C., Model, J., Gu, Y., Sukhwani, B., and VanCourt, T. (2006). Single Pass, BLAST-Like, Approximate String Matching on FPGAs. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 217–226.

Hildebrand, M., Khan, J., Trika, S., Lowe-Power, J., and Akella, V. (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*, ASPLOS '20, page 875–890, New York, NY, USA. Association for Computing Machinery.

Hoefler, T., Di Girolamo, S., Taranov, K., Grant, R. E., and Brightwell, R. (2017). SPIN: High-Performance Streaming Processing In the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA. Association for Computing Machinery.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2020). The curious case of neural text degeneration. arXiv, https://arxiv.org/abs/1904.09751.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv, https://arxiv.org/abs/1704.04861.

Huang, C.-C., Jin, G., and Li, J. (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*, ASPLOS '20, page 1341–1355, New York, NY, USA. Association for Computing Machinery.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, z. (2019). Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Humphries, B., Zhang, H., Sheng, J., Landaverde, R., and Herbordt, M. (2014). 3D FFT on a Single FPGA. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. doi: 10.1109/ FCCM.2014.28.

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

Intel (2022). Intel® FPGA SmartNIC. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic.html.

Jaganathan, R. G., Underwood, K. D., and Sass, R. (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*, SC '03, page 22, New York, NY, USA. Association for Computing Machinery.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. (2020). Improving the accuracy, scalability, and performance of graph neural networks with Roc. In Dhillon, I., Papailiopoulos, D., and Sze, V., editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 187–198.

Jia, Z., Zaharia, M., and Aiken, A. (2019). Beyond data and model parallelism for deep neural networks. In Talwalkar, A., Smith, V., and Zaharia, M., editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13.

Jiang, J. (2023). CXL for Composable and Disaggregated AI Computing - XConn Technologies. In *3rd International Workshop on RESource DISaggregation in High Performance Computing (RESDIS)*.

Jiang, W., He, Z., Zhang, S., Preuß er, T. B., Zeng, K., Feng, L., Zhang, J., Liu, T., Li, Y., Zhou, J., Zhang, C., and Alonso, G. (2021). MicroRec: efficient recommendation inference by hardware and data structure solutions. In Smola, A., Dimakis, A., and Stoica, I., editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 845–859.

Jiang, Z., Lin, H., Zhong, Y., Huang, Q., Chen, Y., Zhang, Z., Peng, Y., Li, X., Xie, C., Nong, S., Jia, Y., He, S., Chen, H., Bai, Z., Hou, Q., Yan, S., Zhou, D., Sheng, Y., Jiang, Z., Xu, H., Wei, H., Zhang, Z., Nie, P., Zou, L., Zhao, S., Xiang, L., Liu, Z., Li, Z., Jia, X., Ye, J., Jin, X., and Liu, X. (2024). MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA. USENIX Association.

Jin, H., Liu, B., Jiang, W., Ma, Y., Shi, X., He, B., and Zhao, S. (2018). Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Transactions on Architecture and Code Optimization, 15(3): article 37*. https://doi.org/10.1145/3243904.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. arXiv, https://arxiv.org/abs/2001.08361.

Ke, L., Zhang, X., Lee, B., Suh, G. E., and Lee, H.-H. S. (2022). Disaggrec: Architecting disaggregated systems for large-scale personalized recommendation. arXiv, https://arxiv.org/abs/2212.00939.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. arXiv, https://arxiv.org/abs/1609.04836.

Kikuchi, K., Fujita, N., Kobayashi, R., and Boku, T. (2023). Implementation and performance evaluation of collective communications using CIRCUS on multiple FPGAs. In *Proceedings of the HPC Asia 2023 Workshops*.

Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks. arXiv, https://arxiv.org/abs/1609.02907.

Krishnan, V., Serres, O., and Blocksome, M. (2020). COnfigurable Network Protocol Accelerator (COPA): An Integrated Networking/Accelerator Hardware/Software Framework. In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 17–24.

Kwon, Y. and Rhu, M. (2018). Beyond the memory wall: a case for memory-centric hpc system for deep learning. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 148–161. IEEE Press.

Kwon, Y. and Rhu, M. (2019). A disaggregated memory system for deep learning. *IEEE Micro*, 39(5):82–90.

Kwon, Y. and Rhu, M. (2022). Training personalized recommendation systems from (GPU) scratch: Look forward not backwards. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, page 860–873, New York, NY, USA. Association for Computing Machinery.

Lai, Z., Li, S., Tang, X., Ge, K., Liu, W., Duan, Y., Qiao, L., and Li, D. (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):1466–1478.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2020). ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. arXiv, https://arxiv.org/abs/1909.11942.

Lasalle, D. and Karypis, G. (2013). Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 225–236.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, USA. USENIX Association.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. (2020). PyTorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018.

Li, Y., Liu, I.-J., Yuan, Y., Chen, D., Schwing, A., and Huang, J. (2019). Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 279–291, New York, NY, USA. Association for Computing Machinery.

Lim, H., Andersen, D. G., and Kaminsky, M. (2019). 3LC: lightweight and effective traffic compression for distributed machine learning. In Talwalkar, A., Smith, V., and Zaharia, M., editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 53–64.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. (2020). Deep gradient compression: Reducing the communication bandwidth for distributed training. arXiv, https://arxiv.org/abs/1712.01887.

Liu, S., Wang, Q., Zhang, J., Lin, Q., Liu, Y., Xu, M., Chueng, R. C. C., and He, J. (2020). NetReduce: RDMA-Compatible In-Network Reduction for Distributed DNN Training Acceleration. arXiv, https://arxiv.org/abs/2009.09736.

Liu, W., Wen, Y., Yu, Z., and Yang, M. (2017). Large-margin softmax loss for convolutional neural networks. arXiv, https://arxiv.org/abs/1612.02295.

Liu, Y., Sheng, J., and Herbordt, M. (2016). A Hardware Prototype for In-Brain Neural Spike-Sorting. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. doi: 10.1109/HPEC.2016.7761590.

Mahram, A. and Herbordt, M. (2012). FMSA: FPGA-Accelerated ClustalW-Based Multiple Sequence Alignment through Pipelined Prefiltering. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 177–183. doi: 10.1109/FCCM.2012.38.

Marvell (2020). Marvell® LiquidIO™ III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf.

Mellanox (2020). Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf.

Meyer, M., Kenter, T., and Plessl, C. (2023). Multi-FPGA designs and scaling of HPC challenge benchmarks via MPI and circuit-switched inter-FPGA networks. *ACM Transactions on Reconfigurable Technology and Systems*, 16(2):1–27.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. (2018). Mixed precision training. arXiv, https://arxiv.org/abs/1710.03740.

Mudigere, D., Hao, Y., Huang, J., Jia, Z., Tulloch, A., Sridharan, S., Liu, X., Ozdal, M., Nie, J., Park, J., Luo, L., Yang, J. A., Gao, L., Ivchenko, D., Basant, A., Hu, Y., Yang, J., Ardestani, E. K., Wang, X., Komuravelli, R., Chu, C.-H., Yilmaz, S., Li, H., Qian, J., Feng, Z., Ma, Y., Yang, J., Wen, E., Li, H., Yang, L., Sun, C., Zhao, W., Melts, D., Dhulipala, K., Kishore, K., Graf, T., Eisenman, A., Matam, K. K., Gangidi, A., Chen, G. J., Krishnan, M., Nayak, A., Nair, K., Muthiah, B., khorashadi, M., Bhattacharya, P., Lapukhov, P., Naumov, M., Mathews, A., Qiao, L., Smelyanskiy, M., Jia, B., and Rao, V. (2022). Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 993–1011, New York, NY, USA. Association for Computing Machinery.

Mudigere, D. and Zhao, W. (2019). HW/SW Co-design for future AI platforms - Large memory unified training platform (Zion). https://2019ocpregionalsummit.sched.com/event/Qyge.

Munafo, R., Shahzad, H., Sanaullah, A., Arora, S., Drepper, U., and Herbordt, M. (2023). Improved models for policy-agent learning of compiler directives in HLS. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. (2021). Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA. Association for Computing Machinery.

Naumov, M., Kim, J., Mudigere, D., Sridharan, S., Wang, X., Zhao, W., Yilmaz, S., Kim, C., Yuen, H., Ozdal, M., Nair, K., Gao, I., Su, B.-Y., Yang, J., and

Smelyanskiy, M. (2020). Deep learning training in Facebook data centers: Design of scale-up and scale-out systems. arXiv, https://arxiv.org/abs/2003.09518.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. (2019). Deep learning recommendation model for personalization and recommendation systems. arXiv, https://arxiv.org/abs/1906.00091.

Netronome (2020). Netronome® Agilio® SmartNICs. https://netronome.com/agilio-smartnics/.

NVIDIA (2020). NVIDIA Mellanox BlueField SmartNIC for Ethernet. https://network.nvidia.com/files/doc-2020/pb-bluefield-smart-nic.pdf.

Nvidia (2021). Nvidia BlueField-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

NVIDIA (2023). What is nvlink. https://blogs.nvidia.com/blog/2023/03/06/what-is-nvidia-nvlink/#:~:text=NVLink%20is%20a%20high%2Dspeed,and%20calculations%20to%20actionable%20results.&text=Accelerated%20computing%20%E2%80%94%20a%20capability%20once,research%20labs%20%E2%80%94%20has%20gone%20mainstream.

Nvidia (2024a). NCCL: Accelerated Multi-GPU Collective Communications. https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf.

Nvidia (2024b). NVIDIA Collective Communications Library (NCCL) . https://developer.nvidia.com/nccl.

Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. (2018). Activation Functions: Comparison of trends in Practice and Research for Deep Learning. arXiv, https://arxiv.org/abs/1811.03378.

Ohmura, I., Morimoto, G., Ohno, Y., Hasegawa, A., and Taiji, M. (2014). MDGRAPE-4: a special purpose computer system for molecular dynamics simulations. *Philosophical Transactions of the Royal Society A*, 372(20130387).

OpenAI (2024). GPT-4 Technical Report. arXiv, https://arxiv.org/abs/2303.08774.

Oyama, Y., Maruyama, N., Dryden, N., Mccarthy, E., Harrington, P., Balewski, J., Matsuoka, S., Nugent, P., and Van Essen, B. (2020). The case for strong scaling in deep learning: Training large 3D CNNs with hybrid parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 32(12).

Park, H., Cho, A., Jeon, H., Lee, H., Yang, Y., Lee, S., Lee, H., and Choo, J. (2023). HPC2 lusterscape: Increasing transparency and efficiency of shared high-performance computing clusters for large-scale ai models. In *2023 IEEE Visualization in Data Science (VDS)*, pages 21–29.

Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D., Law, J., Malani, P., Malevich, A., Nadathur, S., Pino, J., Schatz, M., Sidorov, A., Sivakumar, V., Tulloch, A., Wang, X., Wu, Y., Yuen, H., Diril, U., Dzhulgakov, D., Hazelwood, K., Jia, B., Jia, Y., Qiao, L., Rao, V., Rotem, N., Yoo, S., and Smelyanskiy, M. (2018). Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications. arXiv, https://arxiv.org/abs/1811.09886.

Pascoe, C., Stewart, L., Sherman, B., Sachdeva, V., and Herbordt, M. (2020). Execution of Complete Molecular Dynamics Simulations on Multiple FPGAs. In *IEEE High Performance Extreme Computing Conference*. https://par.nsf.gov/servlets/purl/10195301.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA. Curran Associates Inc.

Patel, R., Haghi, P., Jain, S., Kot, A., Krishnan, V., Varia, M., and Herbord, M. (2022a). Distributed hardware accelerated secure joint computation on the COPA framework. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7.

Patel, R., Haghi, P., Jain, S., Kot, A., Krishnan, V., Varia, M., and Herbordt, M. (2022b). COPA use case: Distributed secure joint computation. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–2.

Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., and Qian, X. (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*, page 891–905, New York, NY, USA. Association for Computing Machinery.

Popescu, M.-C., Balas, V. E., Perescu-Popescu, L., and Mastorakis, N. (2009). Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588.

Pudipeddi, B., Mesmakhosroshahi, M., Xi, J., and Bharadwaj, S. (2020). Training large neural networks with constant memory using a new execution algorithm. arXiv, https://arxiv.org/abs/2002.05645.

Putnam, A. (2014). A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 13–24. doi: 10.1109/ISCA.2014.6853195.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. (2020). ZeRO: memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. (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*, SC '21, New York, NY, USA. Association for Computing Machinery.

Rannen-Triki, A., Berman, M., Kolmogorov, V., and Blaschko, M. B. (2019). Function Norms for Neural Networks. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 748–752.

Ren, J., Luo, J., Wu, K., Zhang, M., Jeon, H., and Li, D. (2021a). Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 598–611.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. (2021b). ZeRO-Offload: Democratizing Billion-Scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564. USENIX Association.

Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. (2016). VDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture. (MICRO-49), article 18. doi: 10.1109/MICRO.2016.7783721.*

Sanaullah, A. and Herbordt, M. (2018a). An Empirically Guided Optimization Framework for FPGA OpenCL. In *2018 International Conference on Field Programmable Technology (FPT)*, pages 46–53. doi: 10.1109/FPT.2018.00018.

Sanaullah, A. and Herbordt, M. (2018b). FPGA HPC using OpenCL: Case Study in 3D FFT. In *9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, page 1–6. doi: 10.1145/3241793.3241800.

Sanaullah, A. and Herbordt, M. (2018c). Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. doi: 10.1109/HPEC.2018.8547646.

Sanaullah, A., Khoshparvar, A., and Herbordt, M. (2016). FPGA-Accelerated Particle-Grid Mapping. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 192–195. doi: 10.1109/ FCCM .2016.53.

Sanaullah, A., Sachdeva, V., and Herbordt, M. (2018a). SimBSP: Enabling RTL Simulation for Intel FPGA OpenCL Kernels. In *IEEE 4th Annual Heterogeneous High Performance Reconfigurable Computing*. doi: 10.1186/s12859-018-2505-7.

Sanaullah, A., Yang, C., Alexeev, Y., Yoshii, K., and Herbordt, M. (2018b). Real-Time Data Analysis for Medical Diagnosis using FPGA Accelerated Neural Networks. *BMC Bioinformatics*, 19 Supplement 18. doi: 10.1186/s12859-018-2505-7.

Sanaullah, A., Yang, C., Alexeev, Y., Yoshii, K., and Herbordt, M. C. (2018c). Application aware tuning of reconfigurable multi-layer perceptron architectures. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–9. doi: 10.1109/HPEC.2018.8547521.

Schonbein, W., Grant, R. E., Dosanjh, M. G. F., and Arnold, D. (2019). INCA: In-Network Compute Assistance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA. Association for Computing Machinery.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., and Eliassi-Rad, T. (2008). Collective classification in network data. *AI Magazine*, 29(3):93.

Sergeev, A. and Balso, M. D. (2018). Horovod: fast and easy distributed deep learning in TensorFlow. arXiv, https://arxiv.org/abs/1802.05799.

Sethi, G., Acun, B., Agarwal, N., Kozyrakis, C., Trippel, C., and Wu, C.-J. (2022). Recshard: Statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 344–358, New York, NY, USA. Association for Computing Machinery.

Shahzad, H., Sanaullah, A., Arora, S., Drepper, U., and Herbordt, M. (2024). AutoAnnotate: Reinforcement Learning based Code Annotation for High Level Synthesis. In *25th International Symposium on Quality Electronic Design*. DOI: 10.1109/ISQED60706.2024.10528738.

Shahzad, H., Sanaullah, A., Arora, S., Munafo, R., Yao, X., Drepper, U., and Herbordt, M. (2022). Reinforcement Learning Strategies for Compiler Optimization in High Level Synthesis. In *The Eighth Workshop on the LLVM Compiler Infrastructure in HPC*. DOI: 10.1109/LLVM-HPC56686.2022.00007.

Shahzad, H., Sanaullah, A., and Herbordt, M. (2021). Survey and Future Trends for FPGA Cloud Architectures. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. doi: 10.1109/HPEC49654.2021.9622807.

Shaw, D. E., Adams, P. J., Azaria, A., Bank, J. A., Batson, B., Bell, A., Bergdorf, M., Bhatt, J., Butts, J. A., Correia, T., Dirks, R. M., Dror, R. O., Eastwood, M. P., Edwards, B., Even, A., Feldmann, P., Fenn, M., Fenton, C. H., Forte, A., Gagliardo, J., Gill, G., Gorlatova, M., Greskamp, B., Grossman, J., Gullingsrud, J., Harper, A., Hasenplaugh, W., Heily, M., Heshmat, B. C., Hunt, J., Ierardi, D. J., Iserovich, L., Jackson, B. L., Johnson, N. P., Kirk, M. M., Klepeis, J. L., Kuskin, J. S., Mackenzie, K. M., Mader, R. J., McGowen, R., McLaughlin, A., Moraes, M. A., Nasr, M. H., Nociolo, L. J., O'Donnell, L., Parker, A., Peticolas, J. L., Pocina, G., Predescu, C., Quan, T., Salmon, J. K., Schwink, C., Shim, K. S., Siddique, N., Spengler, J., Szalay, T., Tabladillo, R., Tartler, R., Taube, A. G., Theobald, M., Towles, B., Vick, W., Wang, S. C., Wazlowski, M., Weingarten, M. J., Williams, J. M., and Yuh, K. A. (2021). Anton 3: Twenty Microseconds of Molecular Dynamics Simulation before Lunch. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/3458817.3487397.

Shaw, D. E., Grossman, J., Bank, J. A., Batson, B., Butts, J. A., Chao, J. C., Deneroff, M. M., Dror, R. O., Even, A., Fenton, C. H., Forte, A., Gagliardo, J., Gill, G., Greskamp, B., Ho, C. R., Ierardi, D. J., Iserovich, L., Kuskin, J. S., Larson, R. H., Layman, T., Lee, L.-S., Lerer, A. K., Li, C., Killebrew, D., Mackenzie, K. M., Mok, S. Y.-H., Moraes, M. A., Mueller, R., Nociolo, L. J., Peticolas, J. L., Quan, T., Ramot, D., Salmon, J. K., Scarpazza, D. P., Schafer, U. B., Siddique, N., Snyder, C. W., Spengler, J., Tang, P. T. P., Theobald, M., Toma, H., Towles, B., Vitale, B., Wang, S. C., and Young, C. (2014). Anton 2: Raising the bar for performance and programmability in a special-purpose Molecular Dynamics supercomputer. In *SC '14: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 41–53.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. (2018). Mesh-tensorflow: Deep learning for supercomputers. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Shen, J., Zuo, P., Luo, X., Su, Y., Gu, J., Feng, H., Zhou, Y., and Lyu, M. R.

(2023). Ditto: An elastic and adaptive memory-disaggregated caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 675–691, New York, NY, USA. Association for Computing Machinery.

Sheng, J., Humphries, B., Zhang, H., and Herbordt, M. (2014). Design of 3D FFTs with FPGA Clusters. In *IEEE High Performance Extreme Computing Conference*. doi: 10.1109/ HPEC.2014.7040997.

Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. (2017a). Collective Communication on FPGA Clusters with Static Scheduling. *ACM SIGARCH Computer Architecture News*, 44(4). doi: 10.1145/ 3039902.3039904.

Sheng, J., Yang, C., Caulfield, A., Papamichael, M., and Herbordt, M. (2017b). HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics. In *27th International Conference on Field Programmable Logic and Applications*. doi: 10.23919/ FPL.2017.8056853.

Sheng, J., Yang, C., and Herbordt, M. (2018). High Performance Dynamic Communication on Reconfigurable Clusters. In *28th International Conference on Field Programmable Logic and Applications*. doi: 10.1109/FPL.2018.00044.

Shi, H.-J. M., Mudigere, D., Naumov, M., and Yang, J. (2020a). Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '20, page 165–175, New York, NY, USA. Association for Computing Machinery.

Shi, R., Dong, P., Geng, T., Ding, Y., Ma, X., So, H. K.-H., Herbordt, M., Li, A., and Wang, Y. (2020b). CSB-RNN: a faster-than-realtime rnn acceleration framework with compressed structured blocks. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA. Association for Computing Machinery.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2020). Megatron-LM: Training multi-billion parameter language models using model parallelism. arXiv, https://arxiv.org/abs/1909.08053.

Sidler, D., Alonso, G., Blott, M., Karras, K., Vissers, K., and Carley, R. (2015). Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. arXiv, https://arxiv.org/abs/1409.1556.

Smith, B. and Linden, G. (2017). Two decades of recommender systems at amazon.com. *IEEE Internet Computing*, 21(3):12–18.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., Zhang, E., Child, R., Aminabadi, R. Y., Bernauer, J., Song, X., Shoeybi, M., He, Y., Houston, M., Tiwary, S., and Catanzaro, B. (2022). Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. arXiv, https://arxiv.org/abs/2201.11990.

Song, J., Yim, J., Jung, J., Jang, H., Kim, H.-J., Kim, Y., and Lee, J. (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*, ASPLOS 2023, page 560–573, New York, NY, USA. Association for Computing Machinery.

Song, R., Wu, C., Liu, C., Li, A., Huang, M., and Geng, T. (2024). DS-GL: advancing graph learning via harnessing nature's power within scalable dynamical systems. *the 51th IEEE/ACM International Symposium on Computer Architecture*.

Stern, J., Xiong, Q., Sheng, J., Skjellum, A., and Herbordt, M. (2017). Accelerating mpi reduce with fpgas in the network. In *Proc Workshop on Exascale MPI*.

Sukhwani, B. and Herbordt, M. (2008). Acceleration of a Production Rigid Molecule Docking Code. In *2008 International Conference on Field Programmable Logic and Applications*, pages 341–346.

Sukhwani, B. and Herbordt, M. (2010). FPGA Acceleration of Rigid Molecule Docking Codes. *IET Computers and Digital Techniques*, 4(3):184–195.

Tan, C., Xie, C., Geng, T., Marquez, A., Tumeo, A., Barker, K., and Li, A. (2021). ARENA: asynchronous reconfigurable accelerator ring to enable data-centric parallel computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(12):2880–2892.

Thoppilan, R., Freitas, D. D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., Li, Y., Lee, H., Zheng, H. S., Ghafouri, A., Menegali, M., Huang, Y., Krikun, M., Lepikhin, D., Qin, J., Chen, D., Xu, Y., Chen, Z., Roberts, A., Bosma, M., Zhao, V., Zhou, Y., Chang, C.-C., Krivokon, I., Rusch, W., Pickett, M., Srinivasan, P., Man, L., Meier-Hellstern, K., Morris, M. R., Doshi, T., Santos, R. D., Duke, T., Soraker, J., Zevenbergen, B., Prabhakaran, V., Diaz, M., Hutchinson, B., Olson, K., Molina, A., Hoffman-John, E., Lee, J., Aroyo, L., Rajakumar, R., Butryna, A., Lamm, M., Kuzmina, V., Fenton, J., Cohen, A., Bernstein, R., Kurzweil, R., Aguera-Arcas, B., Cui, C., Croak, M., Chi, E.,

and Le, Q. (2022). LaMDA: language models for dialog applications. arXiv, https://arxiv.org/abs/2201.08239.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. (2023a). LLaMA: open and efficient foundation language models. arXiv, https://arxiv.org/abs/2302.13971.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. (2023b). Llama 2: Open foundation and fine-tuned chat models. arXiv, https://arxiv.org/abs/2307.09288.

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

VanCourt, T., Gu, Y., and Herbordt, M. (2004). FPGA acceleration of rigid molecule interactions. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 300–301.

VanCourt, T., Gu, Y., and Herbordt, M. (2005). Three-dimensional template correlation: object recognition in 3D voxel data. In *Seventh International Workshop on Computer Architecture for Machine Perception (CAMP'05)*, pages 153–158.

VanCourt, T. and Herbordt, M. (2004). Families of FPGA-based algorithms for approximate string matching. In *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 354–364.

VanCourt, T. and Herbordt, M. (2005). LAMP: a tool suite for families of FPGA-based computational accelerators. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 612–617.

VanCourt, T. and Herbordt, M. (2006a). Application-dependent memory interleaving enables high performance in FPGA-based grid computations. In *IEEE Conference on Field Programmable Logic and Applications*, pages 395–401. doi: 10.1109/ FCCM.2006.25.

VanCourt, T. and Herbordt, M. (2006b). Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, v2006:1–10. doi: 10.1155/ ASP/2006/97950.

VanCourt, T. and Herbordt, M. (2006). Sizing of processing arrays for FPGA-based computation. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30:5998–6008.

Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. (2018). Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 41–53, New York, NY, USA. Association for Computing Machinery.

Wang, M., Huang, C.-c., and Li, J. (2019a). Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. ACM. article 26. https://doi.org/10.1145/3302424.3303953.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. (2020a). Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv, https://arxiv.org/abs/1909.01315.

Wang, T., Geng, T., Jin, X., and Herbordt, M. (2019b). Accelerating AP3M-Based Computational Astrophysics Simulations with Reconfigurable Clusters. In *IEEE 30th International Conference on Application-specific Systems, Architectures and Processors*, volume 2160-052X, pages 181–184.

Wang, T., Geng, T., Jin, X., and Herbordt, M. (2019c). FP-AMR: A Reconfigurable Fabric Framework for Adaptive Mesh Refinement Applications. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 245–253.

Wang, T., Geng, T., Li, A., Jin, X., and Herbordt, M. (2020b). FPDeep: Scalable Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters. *IEEE Transactions on Computers*, 69(8):1143–1158.

Wei, X., Cheng, R., Yang, Y., Chen, R., and Chen, H. (2023). Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, Boston, MA. USENIX Association.

Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. (2009). Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 1113–1120, New York, NY, USA. Association for Computing Machinery.

Wilkening, M., Gupta, U., Hsia, S., Trippel, C., Wu, C.-J., Brooks, D., and Wei, G.-Y. (2021). RecSSD: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 717–729, New York, NY, USA. Association for Computing Machinery.

Wolfe, P., Patel, R., Munafo, R., Varia, M., and Herbordt, M. (2020). Secret sharing mpc on fpgas in the datacenter. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 236–242, Los Alamitos, CA, USA. IEEE Computer Society.

Wu, C., Bandara, S., Geng, T., Guo, A., Haghi, P., Sherman, W., Sachdeva, V., and Herbordt, M. (2022). Optimized Mappings for Symmetric Range-Limited Molecular Force Calculations on FPGAs. In *International Conference on Field-Programmable Logic and Applications*. DOI: 10.1109/FPL57034.2022.00026.

Wu, C., Bandara, S., Geng, T., Sachdeva, V., Sherman, B., and Herbordt, M. (2021a). System-Level Modeling of GPU/FPGA Clusters for Molecular Dynamics Simulations. In *IEEE High Performance Extreme Computing Conference*. doi: 10.1109/HPEC49654.2021.9622838.

Wu, C., Geng, T., Bandara, S., Yang, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2021b). Upgrade of FPGA Range-Limited Molecular Dynamics to Handle Hundreds of Processors. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

Wu, C., Geng, T., Guo, A., Bandara, S., Haghi, P., Liu, C., Li, A., and Herbordt, M. (2023). FASDA: An FPGA-Aided, Scalable and Distributed Accelerator for Range-Limited Molecular Dynamics. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. DOI: 10.1145/3581784.3607100.

Wu, C., Geng, T., Sachdeva, V., Sherman, W., and Herbordt, M. (2020). A Communication-Efficient Multi-Chip Design for Range-Limited Molecular Dynamics. In *2020 IEEE High Performance extreme Computing Conference (HPEC)*.

Wu, C., Song, R., Liu, C., Yang, Y., Li, A., Huang, M., and Geng, T. (2024a). Extending power of nature from binary to real-valued graph learning in real world. In *The Twelfth International Conference on Learning Representations*.

Wu, C., Yang, C., Bandara, S., Geng, T., Haghi, P., Li, A., and Herbordt, M. (2024b). FPGA-Accelerated Range-Limited Molecular Dynamics. *IEEE Transactions on Computers*, 73(6):1544–1558. doi: 10.1109/TC.2024.3375613.

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

Xilinx (2022a). Pynq: Python productivity for adaptive computing platforms. https://pynq.readthedocs.io/en/latest/.

Xilinx (2022b). The Industry's First SmartNIC With Composable Hardware. https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html.

Xilinx (2022c). Xilinx Versal AI Engine Power. https://docs.xilinx.com/r/2020.2-English/ug1275-xilinx-power-estimator-versal/AI-Engine-Power.

Xilinx (2022d). Xilinx Vitis Analyzer. https://docs.xilinx.com/r/en-US/ug1076-ai-engine-environment/Compiling-and-Running-the-Graph-from-the-Command-Line.

Xilinx (2024a). AI Engine API Overview. https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding/AI-Engine-API-Overview.

Xilinx (2024b). AMD Versal™ Adaptive SoCs. https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal.html.

Xilinx (2024c). Versal ACAP AI Engine Intrinsics Documentation. https://docs.amd.com/r/2021.2-English/ug1079-ai-engine-kernel-coding/Intrinsics.

Xilinx (2024d). Xilinx runtime library (xrt). https://www.xilinx.com/products/design-tools/vitis/xrt.html.

Xilinx (2024e). Xilinx® logicore™ ip aurora 64b/66b core. https://www.xilinx.com/products/intellectual-property/aurora64b66b.html.

Xiong, Q., Bangalore, P., Skjellum, A., and Herbordt, M. (2018a). MPI Derived Datatypes: Performance and Portability Issues. In *25th European MPI Users' Group Meeting*. doi: 10.1145/ 3236367.3236378.

Xiong, Q., Skjellum, A., and Herbordt, M. (2018b). Accelerating MPI Message Matching Through FPGA Offload. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 191–1914. doi: 10.1109/FPL.2018.00039.

Xiong, Q., Yang, C., Patel, R., Geng, T., Skjellum, A., and Herbordt, M. (2019). GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 258–266. doi: 10.1109/FCCM.2019.00042.

Yan, M., Deng, L., Hu, X., Liang, L., Feng, Y., Ye, X., Zhang, Z., Fan, D., and Xie, Y. (2020). HyGCN: A GCN Accelerator with Hybrid Architecture. In *IEEE International Symposium on High Performance Computer Architecture*, pages 15–29.

Yang, C., Geng, T., Wang, T., Patel, R., Xiong, Q., Sanaullah, A., Lin, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2019). Fully Integrated FPGA Molecular Dynamics Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–31. doi: 10.1145/ 3295500.3356179.

Yang, C., Sheng, J., Patel, R., Sanaullah, A., Sachdeva, V., and Herbordt, M. (2017). OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. doi: 10.1109/ HPEC.2017. 8091078.

Yang, J. A., Huang, J., Park, J., Tang, P. T. P., and Tulloch, A. (2020a). Mixed-precision embedding using a cache. arXiv, https://arxiv.org/abs/2010.11305.

Yang, J. A., Park, J., Sridharan, S., and Tang, P. T. P. (2020b). Training deep learning recommendation model with quantized collective communications. In *Conference on Knowledge Discovery and Data Mining (KDD)*. https://dlp-kdd.github.io/assets/pdf/a11-yang.pdf.

Yang, Z., Cohen, W., and Salakhudinov, R. (2016). Revisiting semi-supervised learning with graph embeddings. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 40–48, New York, New York, USA. PMLR.

Yin, C., Acun, B., Wu, C.-J., and Liu, X. (2021). TT-Rec: tensor train compression for deep learning recommendation models. In Smola, A., Dimakis, A., and Stoica, I., editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 448–462.

You, Y., Gitman, I., and Ginsburg, B. (2017). Large batch training of convolutional networks. arXiv, https://arxiv.org/abs/1708.03888.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C.-J. (2020). Large batch optimization for deep learning: Training BERT in 76 minutes. arXiv, https://arxiv.org/abs/1904.00962.

Zeng, H. and Prasanna, V. (2020). GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 255–265*. ACM. https://doi.org/10.1145/3373087.3375312.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. (2020). Graph-SAINT: graph sampling based inductive learning method. arXiv, https://arxiv.org/abs/1907.04931.

Zhang, B., Kannan, R., and Prasanna, V. (2021). BoostGCN: A Framework for Optimizing GCN Inference on FPGA. In *IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 29–39.

Zhang, B., Zeng, H., and Prasanna, V. (2020). Hardware acceleration of large scale GCN inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 61–68.

Zhao, W., Xie, D., Jia, R., Qian, Y., Ding, R., Sun, M., and Li, P. (2020). Distributed hierarchical GPU parameter server for massive scale deep learning ads systems. In Dhillon, I., Papailiopoulos, D., and Sze, V., editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 412–428.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. (2023). PyTorch FSDP: experiences on scaling fully sharded data parallel. arXiv, https://arxiv.org/abs/2304.11277.

Zhu, Y., He, Z., Jiang, W., Zeng, K., Zhou, J., and Alonso, G. (2021). Distributed recommendation inference on FPGA clusters. In *31st International Conference on Field-Programmable Logic and Applications*, pages 279–285.

CURRICULUM VITAE