2021

True shared memory architecture for next-generation multi-GPU systems

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

TRUE SHARED MEMORY ARCHITECTURE FOR NEXT-GENERATION MULTI-GPU SYSTEMS

by

MD SAIFUL AREFIN MOJUMDER

B.Sc., Bangladesh University of Engineering & Technology, 2013
M.Sc., Bangladesh University of Engineering & Technology, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2021

Approved by

First Reader

Ajay Joshi, Ph.D.
Associate Professor of Electrical and Computer Engineering

Second Reader

Martin Herbordt, Ph.D.
Professor of Electrical and Computer Engineering

Third Reader

Tali Moreshet, Ph.D.
Senior Lecturer and Research Assistant Professor of Electrical and Computer Engineering

Fourth Reader

David R. Kaeli, Ph.D.
Distinguished Professor of Electrical and Computer Engineering
Northeastern University

Acknowledgments

First, I would like to express my sincere gratitude to my PhD adviser, Prof. Ajay Joshi, for his tireless support and encouragement throughout the course of the PhD program. During the ups and downs of the PhD life, Prof. Joshi was a constant source of guidance and assistance in enhancing my research endeavours and skills. Over the years, I have enjoyed working with and learning from him. Undoubtedly, this thesis would not have been complete without him.

I would like to thank my collaborators, Prof. David Kaeli, Prof. John Kim, Prof. José L. Abellán, Amir Kavyan Ziabari, Yifan Sun, and Trinayan Baruah for their sincere support, assistance and guidance throughout my PhD work. I learnt how to think about big picture and focus on solving problems from Prof. Kaeli. Prof. Kim provided the access to the multi-GPU system that we extensively used throughout this work. José's persistent scrutiny of my work always kept me on the right track and I learned the fact that "no detail is too small in research" from him. Amir helped me ramp up in research during the early stage of my PhD. I particularly enjoyed and learnt a lot working with Yifan Sun who was always there to help.

I want to thank the rest of my thesis committee memebers, Prof. Martin Herbordt and Prof. Tali Moreshet, for their precious time and insightful feedback. Additionally, I learnt the basics of computer architecture from the course taught by Prof. Moreshet, while the valuable lesson I learnt from the 'High Performance Programming with Multicore and GPUs' taught by Prof. Herbordt helped me design experiments for my research.

Many thanks to my labmates, Yenai Ma, Leila Delshadtehrani, Marcia Sahaya Luis and Furkan Eris at Boston University. I would also like to thank all the members of the ICSG research lab, the PEAC Lab research group, and the CAAD research group.

Finally, I want to express my gratitude to my parents for their endless love, unconditional support and incessant encouragement throughout my life. In particular, my mother

has been the constant source of inspiration for all the achievements in my life. I want to specially thank my wife, Tasmiah Nuzhath, who stood by me during this uncertain journey and was a persistent source of support and encouragement.

TRUE SHARED MEMORY ARCHITECTURE FOR NEXT-GENERATION MULTI-GPU SYSTEMS

MD SAIFUL AREFIN MOJUMDER

Boston University, College of Engineering, 2021

Major Professor: Ajay Joshi, PhD
Associate Professor of Electrical and Computer Engineering

ABSTRACT

Machine learning (ML) is now omnipresent in all spheres of life. The use of deep neural networks (DNNs) for ML has gained popularity over the past few years. This is because DNNs are capable of efficiently solving complex problems such as image processing, object detection, language processing, etc. To train these DNN workloads, graphics processing units (GPUs) have become the most widely used platform. A GPU can support a large number of parallel threads that execute simultaneously to achieve a very high throughput. However, as the sizes of the DNN workloads grow, a single GPU is no longer adequate to provide fast training, and developers resort to using multi-GPU (MGPU) systems that can reduce the training time significantly. Consequently, to keep pace with the growth of DNN applications, GPU vendors are actively developing novel and efficient MGPU systems.

To better understand the challenges associated with designing MGPU systems for DNN workloads, in this thesis, we first present our efforts to understand the behavior of the DNN workloads, in particular, the training of DNN workloads on MGPU systems. Using the DNN workloads as benchmarks, we observe the evolution of MGPU system architecture. Based on our profiling and characterization of DNN workloads on exist-

ing high-performance MGPU systems, we identify the computation- and communication-intensiveness of the DNN workloads and the hardware- and software-level inefficiencies present in the existing MGPU systems. We find that the data movement across multiple GPUs and high remote data access cost leading to NUMA effects, data duplication, and inefficient use of GPU memory leading to memory capacity issues, and the complexity in programming MGPUs pose serious limitations in the execution of ever-scaling DNN workloads on MGPU systems.

To overcome the limitations of existing MGPU systems, we propose to unify the main memory of GPUs to design an MGPU system with true shared memory (MGPU-TSM). Our proposed MGPU-TSM system demonstrates a significant performance boost (3.8× for a 4 GPU system) over the best-performing existing MGPU system. This is because MGPU-TSM system eliminates the NUMA effects and the necessity for data duplication. To provide seamless data sharing across multiple GPUs and ease programming of MGPU-TSM, we propose a light-weight coherence protocol called MGCC. MGCC is a timestamp-based protocol that provides both intra- and inter-GPU coherence. We implement a number of hardware features including unified memory controller, request tracker and timestamp storage unit to support MGCC. Using both standard and synthetic stress benchmarks, we evaluate the MGPU-TSM system with MGCC leveraging sequential as well as relaxed consistency. Our evaluation of a 4-GPU system using MGPUSim simulator suggests that our proposed coherent MGPU system achieves up to 3.8× improved performance than current best-performing MGPU system while the stress tests performed using synthetic benchmarks suggests that MGCC leads to up to 46.1% performance overhead.

Contents

List of Tables

List of Figures

List of Abbreviations

AWS	.....	Amazon Web Services
BP	.....	Backward Propagation
CPU	.....	Central Processing Unit
CU	.....	Compute Unit
DL	.....	Deep Learning
DNN	.....	Deep Neural Network
DtoH	.....	Device-to-Host
FP	.....	Forward Propagation
GPU	.....	Graphics Processing Unit
HtoD	.....	Host-to-Device
MC	.....	Memory Controller
MCM	.....	Multi-Chip Module
ME	.....	Metric Evaluation
Memcpy	.....	Memory copy
MGCC	.....	Multi-GPU Cache Coherence
MGPU	.....	Multiple Graphics Processing Units
ML	.....	Machine Learning
P2P	.....	Peer-to-Peer
SGD	.....	Stochastic Gradient Descent
SM	.....	Streaming Multiprocessor
SWMR	.....	Single-Writer-Multiple-Reader
TSM	.....	True Shared Memory
TSU	.....	Timestamp Storage Unit
UM	.....	Unified Memory
WU	.....	Weight Update

Chapter 1

Introduction

1.1 A Brief History of GPUs

Graphics Processing Units (GPUs) have been used in computing systems for more than four decades. GPUs were first used in 1978 (Booth et al., 1985) to accelerate video processing. At the beginning, GPUs (although not officially called GPU) targeted the gaming applications. Among the early versions of GPU, the IBM Professional Graphics Controller (PGA) was launched in the market in 1984 and it took over the video processing tasks from the central processing unit (CPU) (James, 1987), (McClanahan, 2010). With the introduction of OpenGL in 1989, Silicon Graphics Inc. (SGI) pioneered the development of the graphics pipeline (Crow, 2004). The graphics cards released by SGI were mostly limited to the workstations, and graphics hardware vendors such as 3DFX, NVIDIA, ATI, and Matrox took the opportunity to market the consumer 3D graphics accelerators in the 1990s (McClanahan, 2010). The adoption of the GPU hardware started growing with the release of games such as Quake and Doom that relied on these graphics boards (McClanahan, 2010). The first true GPU was released by NVIDIA in 1999 (GeForce256) (Dietrich, 1999) and ATI released its first true GPU in 2000 (Radeon 7500) (Crayton et al., 2004). With the introduction of these two GPUs, the gaming industry started to surge. NVIDIA took the big step of making GPU pipeline programmable in 2001 (Crow, 2004). With the introduction of CUDA (Kirk et al., 2007) and OPENCL (Munshi, 2009), programming GPUs became easier.

Although initially GPUs were developed to accelerate 3D image processing for gaming

applications (Blythe, 2008), over time both academia and industry started using GPUs for general purpose computing (Booth et al., 1985), (Mišić et al., 2012). GPUs started to benefit a wide variety of applications, including image processing (Viola et al., 2003), (Strzodka et al., 2003), signal processing (Manocha, 2003), (Govindaraju et al., 2006), computer vision (Fung et al., 2002), (Woetzel and Koch, 2004), oil and gas exploration (Lin and Hall, 2007), (Deschizeaux and Blanc, 2007), linear algebra (Hoff III et al., 2001), (Bolz et al., 2003) physics (Zeller, 2005), (Hagen et al., 2006), chemistry (Harris et al., 2002), (Kim and Lin, 2003), databases and data mining (Govindaraju et al., 2005), (Govindaraju et al., 2005), biomedical applications (Tran et al., 2004), (Cates et al., 2004), and life sciences (Mosegaard and Sorensen, 2005), (Li et al., 2007). Nowadays, GPUs are ubiquitous and are extensively used in almost every computation domain, especially after the widespread adoption of machine learning (ML). Although ML applications leveraged GPUs in the late 1990s (Zheng and Pekhimenko, 1997), the introduction of high-level frameworks such as Caffe (Jia et al., 2014) and MXNet (Chen et al., 2015a) has enabled users from different application domains to use GPUs for training computation-intensive ML applications. To support the growth of ML applications, GPU vendors have introduced several innovations in the GPU architecture, designed powerful GPU systems such as NVIDIA DGX-1 (NVIDIA, 2016), NVIDIA DGX-2 (Choquette and Gandhi, 2020), AMD MI100 (AMD, 2020), etc. as well as developed highly optimized software libraries (NVIDIA, 2018; NVIDIA, 2008) and runtime support to ease programming the GPUs.

Both industry and academia commonly used single-GPU systems up until 2004. A typical single GPU has thousands of parallel threads that run concurrently to achieve very high throughput. However, die size restricts the number of parallel computing cores known as compute units (CUs)[1] or streaming multiprocessors (SMs) in a single GPU (Arunkumar et al., 2017). With the continued growth of applications that demand extremely high com-

[1]Throughout this manuscript, we use CU to refer to compute cores

puting resources, the compute resources of a single GPU were no longer adequate. This led to the design and adoption of Multiple Graphics Processing Unit (MGPU) systems. Today, MGPU systems have become the platform of choice for accelerating a variety of applications, including ML (Chen et al., 2015b), (Al-Rfou et al., 2016), (Abadi et al., 2016), (Paszke et al., 2019), (Jia et al., 2014), (Tokui et al., 2015), graph applications (Che et al., 2013), (Xu et al., 2014), (Shi et al., 2019), (Wang et al., 2019), medical applications (Valero-Lara, 2014), (Saikia and Kanhirodan, 2014), (Zhu et al., 2019) and large-scale simulations (Zhu et al., 2018), (Yamazaki et al., 2014), (Zhang et al., 2018). Hence, MGPU systems have become an integral part of online services such as Amazon Web Services (AWS) (Amazon, 2015), Microsoft Azure (Copeland et al., 2015), and Google Cloud (Krishnan and Gonzalez, 2015).

In fact, MGPU systems have enabled researchers from different domains to solve complex, large and time-consuming problems that were previously impossible due to lack of computing resources. For instance, Goyal et al. (Goyal et al., 2017) trained ResNet-50 in 1 hour using 256 GPUs, which would have otherwise taken more than a week using a single GPU and months using high-end CPUs. As a result, there have been tremendous efforts from both industry and academia to improve the MGPU system design so that MGPU systems can continue to help solve the continually growing complex applications.

1.2 Background

In this thesis, we present our work on the profiling of ML applications on MGPU systems, hardware and software design choices that limit the performance of MGPU systems while running the ML applications, and our novel memory system and associated coherence policies that improve the MGPU system design. To better understand our novel contributions to the field of MGPU systems, here we provide a background on running deep learning (DL) workloads on MGPU systems and MGPU system architecture.

1.2.1 Deep Learning using MGPU Systems

DL is a branch of ML that has become increasingly popular for solving complex data-intensive problems. Deep Neural Network (DNN) model, a type of DL model, is commonly used today in a number of application domains. To maximize the potential of the DNN models, we need to train them carefully. Both single-GPU and MGPU systems are used for training DNNs (Jia et al., 2014). In this section, we provide an overview of DNNs, the process of training DNNs using MGPU systems and the associated challenges.

DNN and its Training Process

A DNN has multiple layers of neurons. Neurons in a layer are connected to the neighboring layers by weighted edges. Each layer applies a set of mathematical operations, such as dot-product, convolution, max-pooling or sigmoid to the layer's inputs. A DNN can be trained to classify input data samples with high accuracy. Training a DNN is an iterative process of updating the parameters (weights) of each layer. The iterative process consists of the following stages in each iteration:

1. Forward Propagation (FP)
2. Backward Propagation (BP)
3. Weight Update (WU)
4. Metric Evaluation (ME)

In FP, each layer performs a set of linear and non-linear operations using a set of weight parameters (randomly initialized in the beginning, but updated during each BP step in the training process) on the input data. The common type of layers in a model include: the convolution layers, the fully connected layers, and the activation layers. The observed output is then compared with the expected output. The difference between the two is then

fed back into the network, from the last layer back to the first layer. This is the BP stage. The outputs of the BP stage are the local gradients of the network parameters, suggesting how each parameter should change its value to reduce the difference between observed and expected output, i.e., improve neural network classification accuracy. After a complete backward pass, the gradients are used to update the weights during the WU stage. This process of updating weights using gradients is based on the Stochastic Gradient Descent (SGD) algorithm. During the ME stage, performance metrics such as training accuracy are calculated. This is performed for each batch of data. Since our evaluation only focuses on performance rather than algorithm efficiency and the ME stage only adds a fixed amount of execution time, we do not include the ME stage in our study.

The FP, BP, and WU stages are repeated multiple times until the output error rate is less than a desired value. For DNNs with large training datasets it is expensive to update the weights after performing FP and BP for each input-output pair of the whole training set. Hence, training data is randomly sampled into mini-batches. All inputs within a mini-batch go through the FP and BP stages. The gradients are accumulated for all input-output pairs within a mini-batch and WU is performed only once.

MGPU DNN Training

MGPU systems enable faster DNN training compared to single-GPU systems because the training is distributed and parallelized across multiple GPUs. In this work, we do not dive deep into how the DNN training algorithm works. We focus on how the training data is managed and moved in a typical DNN training process. Although the exact stages of the training process differ from framework to framework, the overall approach is the same. The timeline for training DNNs using the synchronous SGD algorithm with four GPUs is shown in Figure 1·1. When the algorithm starts, the CPU randomly generates the internal parameters of the network model (not shown in Figure 1·1, as this is a one time process). The network model is broadcasted to all the GPUs. The CPU also loads k mini-batches

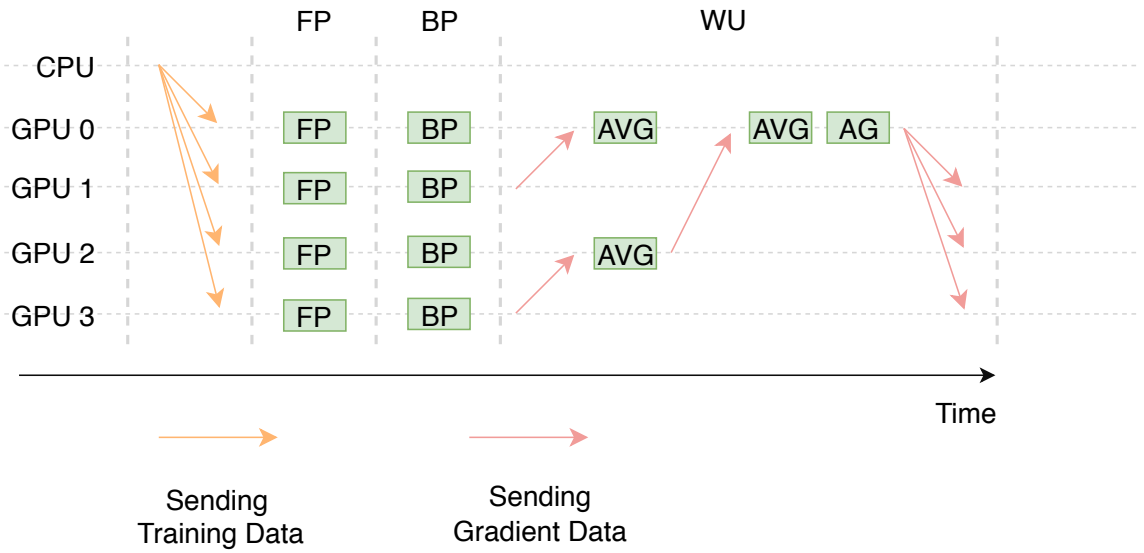


Figure 1·1: The timeline of an epoch during MGPU DNN training using the data-parallelism approach with synchronous SGD. FP, BP, AVG, and AG represent forward propagation, backward propagation, averaging, and add gradients, respectively. (This figure is not drawn to scale.)

of the training data, where k equals to the number of GPUs in the system, and sends one mini-batch to each GPU (see the left-most arrows in the figure). All the GPUs perform FP and BP to calculate the gradients. The size of the gradient data should be approximately equal to the size of data in the neural network model (Glorot and Bengio, 2010).

The gradients calculated by each GPU is not the same and we need to calculate the average gradient. The average is calculated with a *reduction* approach. For example, if four GPUs are used, the gradients calculated by GPU1 will be moved to GPU0 and GPU0 takes the average of the gradients from GPU0 and GPU1. Simultaneously, GPU2 collects the gradients from GPU3 and calculates the average. Finally, GPU0 collects the averaged result from GPU2 and then calculates the average. GPU0 updates the neural network data with averaged gradients and then, it broadcasts the updated network model to the remaining three GPUs. Once all the GPUs have the next mini-batch of the training set sent from the CPU, the next iteration will start. The process is repeated for a specific number of

epochs.[2] Here, the number of epochs depends on the desired accuracy of training and convergence of the training algorithm. Today, a large number of deep learning frameworks, including Caffe (Jia et al., 2014), CNTK (Yu et al., 2014), TensorFlow (Abadi et al., 2016), Torch (Collobert et al., 2011), MXNet (Chen et al., 2015a), Pytorch (Paszke et al., 2019), and Theano (Al-Rfou et al., 2016), use GPUs to reduce the training time of DNNs.

1.2.2 MGPU System Architecture

Over the past few years, NVIDIA and AMD, the two major GPU vendors, have introduced a number of high performance MGPU systems such as NVIDIA Pascal-based DGX-1 (Foley and Danskin, 2017), NVIDIA Volta-based DGX-1 (NVIDIA, 2017), NVIDIA Volta-based DGX-2 (NVIDIA, 2018a), NVIDIA Ampere-based DGX-2 (Choquette and Gandhi, 2020), and AMD MI100 (AMD, 2020). These MGPU systems primarily target the ML domain. To support the ML workloads, GPU vendors have integrated a number novel hardware and software features in these MGPU systems. The noteworthy hardware features include NVLink interconnect (Foley and Danskin, 2017) and AMD Infinity Fabric Link (AMD, 2020) to accelerate data transfers across GPUs, and dedicated tensor cores (Markidis et al., 2018) and matrix cores (AMD, 2020) for fast and efficient matrix-related computations. On the software side, GPU vendors have introduced variety of software libraries including NCCL (NCCL, 2018), RCCL (AMD, 2021) cuDNN (NVIDIA, 2018), and cuBLAS (NVIDIA, 2008).

Figure 1·2 is representative of existing MGPU systems (Young et al., 2018; Milic et al., 2017). In this example MGPU system, there are 4 GPUs, where each GPU has 64 CUs in this example. Each CU has its own private L1$. L2$ is distributed and has multiple (8 in this example) banks. It is shared across all the CUs. The L1$s and the L2$ banks are connected via a network (a Xbar in this example). There is a memory controller connected

[2] An epoch is a complete pass through all the data in the training dataset. Each epoch involves processing of multiple mini-batches of data.

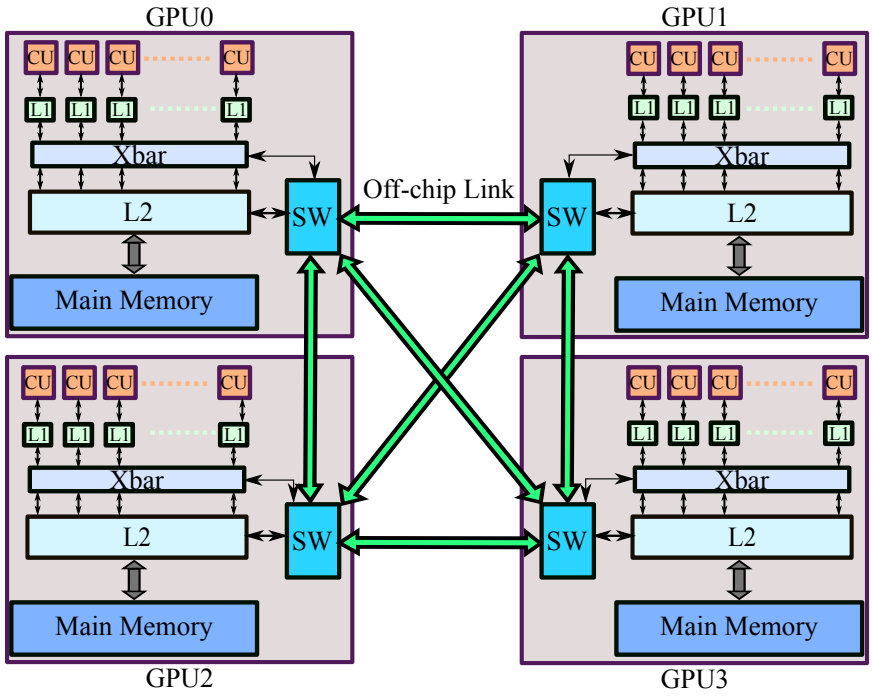


Figure 1·2: Conventional MGPU system. Switch (SW) handles the remote access requests from one GPU to another GPU. PCIe or NVLink is used as the off-chip link.

with each L2$ bank and the memory controller serves the requests it receives from L2$ to read or write the data from main memory.

Thus, we have a total of 8 memory controllers in this example. Each GPU has its own native memory (8 GB for each GPU in this example). Hence, each memory controller controls 1 GB of main memory. The L2$ banks and the main memory of a GPU are connected using high bandwidth and low-latency connections. Each GPU in the MGPU system can access the other GPUs' memory through low-bandwidth high-latency links (PCIe or NVLink) by means of Remote Direct Memory Access (RDMA). These off-chip links have 5× to 20× lower bandwidth (for transferring data between GPUs, and between CPU and GPU) than the bandwidth of the links used for accessing local main memory of a GPU (Young et al., 2018), (Buono et al., 2017). When the CU of a GPU requests data that resides on a remote GPU, the request goes to its remote direct memory access (RDMA)

engine via the L1$ of the CU. This RDMA engine communicates with the RDMA engine residing on the remote GPU. The remote RDMA engine routes the request to the L2$ and then to main memory (if data is not already in the L2$) of the remote GPU. The response from the main memory of the remote GPU is cached in the L2$ of the remote GPU, and then routed to the L1$ of the local GPU via the two RDMA engines (remote and then local). Note that the CPU memory in a typical MGPU RDMA-based system is placed on the CPU side. Hence, off-chip links are used to transfer data between CPU and the GPUs.

1.2.3 Remote Memory Access Mechanisms in an MGPU System

As mentioned before, GPUs in an MGPU system (as described in Section 1.2.2) need to communicate with each other due to the data sharing requirements of the applications. Hence, GPU vendors have introduced a number of mechanisms, summarized in Table 1.1, to enable data sharing across multiple GPUs. This section briefly describes different inter-GPU data transfer/access mechanisms and their pitfalls.

P2P Memcpy

Using the Peer-to-Peer (P2P) memcpy method, one GPU can directly copy data to/from another GPU's memory from/to its own memory. Hence, multiple copies of data exist in different GPUs' memory. The data is transferred using the RDMA engine using the off-chip links. In case of any modification of the copied data, the programmer has to manually update the data in all locations by copying back and forth to maintain coherence. Thus, this method leads to programming complexity.

P2P Direct Access

Using the P2P direct access method, one GPU can directly access the data that resides on remote GPU's memory without copying the data in its own memory. The details about P2P direct access using RDMA has been described in Section 1.2.2. As the same data may

Table 1.1: Comparison of different communication mechanisms in existing MGPU systems. We compare the main memory usage and programmability of each mechanism w.r.t. P2P memcpy (baseline for comparison represented by '–'), and remote memory (RM) access latency and bandwidth w.r.t. local main memory access latency and bandwidth. '✗', '✓', and '✓✓' indicate 'no', 'fair' , and 'good', respectively.

Method	RM Access Latency	RM Access Bandwidth	Data Duplication	Improves Programmability	Improves GPU Mem. Usage
P2P Memcpy	High	Low	Yes	–	–
P2P Direct	High	Low	Partial	✓	✓
Zerocopy	Extremely high	Low	No	✓✓	✗
Unified Memory	Extremely High	Low	No	✓✓	✓

reside in local GPU's cache as well as remote GPU's cache and main memory, handling data races among multiple GPUs is programmer's responsibility as existing GPUs lack coherence support at hardware level. However, this method allows programmer to avoid multiple copies of data in GPUs' memory leading to relatively easier programming.

Zerocopy

Zerocopy method does not use GPU memory. Rather, all the GPUs in the system use the Host or CPU memory. This method addresses the memory capacity issue of GPUs. The memory is pinned to a particular GPU and that GPU can directly access the data using PCIe links. Caches are bypassed for this method. As a result, this method results in extremely slow data access and may introduce serialization delays if multiple GPUs need to access same address space.

Unified Memory

To ease the programming of MGPU systems, NVIDIA introduced unified memory which is a software abstraction of shared main memory. With unified memory, all the physically separate main memory of CPU and GPUs are presented as a single memory space to the programmer. Hence, the programmer can use a single pointer from any of the devices

(CPU or GPU) to access a desired memory location irrespective of its physical location. This method is facilitated by the recently introduced user transparent page fault support in GPUs (Chien et al., 2019). If a GPU wants to access a memory location that is outside its local memory region, a page fault occurs. The GPU runtime along with the CPU handles the page fault and provides the required page to the GPU. However, the unified memory is known to be inefficient in terms of performance even though it eases programmability (Baruah et al., 2020).

1.2.4 Coherence and Consistency in GPU and MGPU Systems

Coherence is required to ensure that all the processors in a system see the same view of memory. In case of shared data, cache coherence ensures that all the components have the same view of shared data. On the other hand, memory consistency is a contract between the hardware and the software. A consistency model ensures that a hardware keeps its promises to maintain certain memory orderings that the software knows. As the CUs of a GPU share the L2 cache and copy data to their private L1$, we need a coherence protocol to make sure all the CUs see the same view of the shared data. The GPU and MGPU consistency model dictates what a programmer should expect from the GPU or MGPU in terms of ordering of memory operations when writing programs. The programmer need to provide appropriate barriers/synchronizations accordingly to obtain the correct result from the hardware.

This section provides the overview on memory coherence and consistency, and discusses support for coherence and consistency on existing GPUs.

Coherence

Consider a system having multiple cores and the system is executing a multi-threaded application. In this multi-threaded application execution environment, coherence mechanism decides how updates to a single memory location are propagated. We use the single-writer–multiple-reader (SWMR) invariant to define coherence (Nagarajan et al., 2020).

SMWR invariant is defined as: at a given logical time, only one core can write to a given location and other cores can only read the same location (Nagarajan et al., 2020). The typical responsibilities (Nagarajan et al., 2020) of a cache coherence protocol involve:

1. When a memory location is updated, the new value must be propagated to all-sharers. This can be done in either of the two ways:
   (a) Update the private copies (known as write-update)
   (b) Invalidate the private copies (known as write-invalidate)
2. When a write operation is performed, the updated value must be globally visible i.e. visible to all threads and processors
3. A write operation must be logically seen by all thread at once (known as write-atomicity (Adve and Gharachorloo, 1996)). This can be achieved by invalidating or updating all private copies of data before completing a write operation.

Consistency

While memory coherence dictates how updates to a single memory location are propagated, the memory consistency model defines a set of allowed behaviors of multi-threaded programs to be correctly executed in a shared memory system (Nagarajan et al., 2020). A memory consistency model tells the programmer what to expect from a system with shared memory. In particular, a memory consistency model specifies the valid ordering of memory operations (i.e. read and write) to different memory locations. There are different memory consistency models such as sequential consistency (SC), relaxed consistency (RC), total store order (TSO) consistency, etc. According to Lamport (Lamport, 1979), a multiprocessor is sequentially consistent if "the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program."

Sequential consistency represents the most strict form of memory consistency. Relaxed consistency models permit a multiprocessor to violate the program order of memory operations. The TSO consistency model is a form of relaxed consistency model that maintains the ordering of store operations but supports reordering of load operations. This model is a compromise between strict sequential consistency and an extremely relaxed consistency model that allows all types of memory reordering. In our work, we support sequential consistency model as well as the relaxed memory consistency model that allows re-ordering of memory operations (see Section 4.3)

Coherence and Consistency in GPUs and MGPUs

Single GPU: Existing GPUs support data-race-free (DRF) weak consistency (a form of relaxed consistency) without coherence (Alsop et al., 2016), (Singh et al., 2015). The programming model assumes that there is no inter-thread communication during kernel execution. Under the DRF consistency model in existing GPUs, a sequentially consistent execution is guaranteed only if program is data-race free. Hence, to make the program data race free the programmer must annotate all data races as synchronization operations by using fences or atomic operations (Singh et al., 2015). The programming model also completely relies on the programmer to take precautionary steps to manually support coherence if needed. There are several ways a programmer can achieve coherence manually. First, the programmer can disable private caching of data in the L1$ and perform the coherence action in shared L2$. Note that modern GPUs support some accesses as coherent accesses at the shared L2$ (Tabbakh, 2018). Bypassing L1$ and executing memory operation at shared L2$ lead to additional traffic and performance degradation. Second, the programmer can perform kernel-stopping coarse-grained synchronization to maintain coherence. This results in poor performance as multiple kernels need to be launched. Third, the programmer can use atomic operation, but it does not support complex instructions.

However, all of the methods are inefficient and create additional burden on programmer

when programming a single GPU.

MGPU: The DRF consistency model along with lack of coherence support exacerbates the performance issues and programming complexities in an MGPU system. The programmer must manually address any consistency issues in a program. To maintain coherence across multiple GPUs, the programmer can again bypass the cache when accessing the remote data and directly access the data from a remote GPU's main memory. However, it leads to significant latency in memory access because of the involvement of high-latency low-bandwidth off-chip links. Another frequently used method is to manually copy data back and forth across multiple GPUs and use kernel-stopping synchronization to maintain coherence. This results in inefficient use of GPU memory because of the existence of multiple copies of the same data in different GPUs' memory.

1.3 Challenges in Existing MGPU Systems

Data sharing across multiple GPUs during kernel execution leads to programming challenges as the programmer must choose between programmability and performance. In this section, we examine the challenges in existing MGPU systems using the DNN training process as an example. We highlight how different communication mechanisms trade-off programmability for performance. Finally, we show how remote memory accesses in an MGPU system is extremely expensive.

1.3.1 DNN Training on MGPU Systems

Training a DNN on a MGPU system introduces various challenges:

- The programmer has to explicitly distribute the data (input data and network model data) among multiple GPUs. The programmer can either distribute the input data onto multiple GPUs while replicating the network model in each of the GPUs (Yadan

et al., 2013) (referred to as data parallelism), or assign different parts of the neural network model to distinct GPUs (referred to as model parallelism) (Yadan et al., 2013).

- Both approaches require data to be transferred and synchronized across GPUs. The programmer needs to carefully handle data transfer and synchronization to ensure correctness of computations.

- The input data is fed to the GPUs as mini-batches (also called batches). Each mini-batch consists of a certain number of unique inputs chosen by the programmer from the dataset. The choice of mini-batch size has implications on training time, GPU memory usage and training accuracy. Recent works (Goyal et al., 2017), (You et al., 2017), (Smith et al., 2017) have shown that batch size can be increased without losing accuracy.

- Although we can parallelize the computation required for training DNNs, the GPUs still need to communicate with each other during the different phases of training. High-end MGPU systems support different methods and libraries for communication. Depending on the size of neural networks, communication can pose significant bottlenecks. To minimize the communication time, both hardware-level (i.e. NVLinks) and software-level (i.e. NCCL library) solutions have been introduced. Nonetheless, the training algorithm requires more hardware-specific optimizations to achieve the peak performance from an MGPU system.

To understand the programming complexities and inefficiencies in an MGPU system, we use Algorithms 1, 2 and 3, where we consider three different ways a programmer can perform the WU stage.

Algorithm 1: Using Memcpy*

Initialization: `weights` in CPU ;
Copy `weights` from CPU to GPU0 → `wGPU0` ;
Copy `weights` from CPU to GPU1 → `wGPU1` ;
FP+BP on GPU0 using `wGPU0` → `gGPU0` ;
FP+BP on GPU1 using `wGPU1` → `gGPU1` ;
Copy `gGPU1` from GPU1 to GPU0 → `gGPU0Copy` ;
WU on GPU0 using `(gGPU0,gGPU0Copy)` → `wGPU0` ;
Copy `wGPU0` from GPU0 to GPU1 → `wGPU1` ;

Algorithm 2: Using P2P direct access*

Initialization: `weights` in CPU ;
Copy `weights` from CPU to GPU0 → `wGPU` ;
FP+BP on GPU0 using `wGPU` → `gGPU0` ;
FP+BP on GPU1 using `wGPU` → `gGPU1` ;
WU on GPU0 using `(gGPU0,gGPU1)` → `wGPU` ;

Algorithm 3: Using shared main memory*

Initialization: `weights` in CPU ;
FP+BP on GPU0 using `weights` → `g0` ;
FP+BP on GPU1 using `weights` → `g1` ;
WU on GPU0 using `(g0, g1)` → `weights` ;

*In the pseudocode, the right arrows point to destination variables of an operation.

We will assume a 2-GPU MGPU system here. Algorithm 1 shows that when using memcpy, the programmer must maintain coherence explicitly by periodically copying data to GPU1's memory. Thus, there is an additional copy of data i.e. SGD (`gGPU1`) in GPU0's memory, leading to additional memory usage. Nonetheless, this mechanism can be efficient in terms of kernel runtime because P2P memcpy can run asynchronously. Algorithm 2 shows how P2P direct access with RDMA can eliminate the data copy step, but at the expense of accessing data using off-chip links. Still, the programmer must transfer the data from the CPU to the GPUs. Algorithm 3 illustrates that a shared main memory could ease programmability and eliminate explicit GPU-to-GPU or CPU-to-GPU data transfers. Note that UM and Zerocopy solutions use Algorithm 3. UM, as proposed by NVIDIA,

eases programming with a software abstraction, but suffers from performance degradation due to inefficient page-fault support and expensive remote accesses (Baruah et al., 2020). A Zerocopy solution does not use GPU memory at all. The GPUs access *pinned* CPU memory using the off-chip (PCIe) links (Negrut et al., 2014). We argue that we need a solution which would not trade-off programmability to gain performance.

1.3.2 RDMA Access Cost

In this section, using the data access latency metric, we present the performance impact of remote data access in an MGPU system as the previous section demonstrates the necessity of communication across multiple GPUs in the exiting MGPU systems. Here, we run the commonly-used matrix multiplication kernel `SGEMM` (used in training DNNs), from NVIDIA's cuBLAS library (NVIDIA, 2008), on an MGPU system with V100 GPUs (compute capability of 7.0). We use two GPUs connected through NVLink 2.0 (total of 50 GB/s unidirectional bandwidth). The conclusions of our analysis are broadly applicable to systems with more than 2 GPUs that use GPU-GPU RDMA.

The computations in the `SGEMM` kernel consist of three matrices A, B, and C. In our experiment, we distribute the matrices in the memory of two GPUs (GPU0 and GPU1) and examine the performance degradation caused by different degrees of remote access (using P2P direct access as an example) when the `SGEMM` is executed on GPU0. We use the `aL-bR` format to represent `a`% local access and `b`% remote access for GPU0, where `a` and `b` are integers. We evaluate the following four matrix distributions across memory:

1. Matrices A, B and C are in GPU0's memory. This leads to 100% local access for GPU0 (`100L-0R`).

2. Matrices A and B are in GPU0's memory, and C is in GPU1's memory (`67L-33R`).

3. Matrix A is in GPU0's memory, and matrices B and C are in GPU1's memory (`33L-67R`).

4. Matrices A, B and C are in GPU1's memory. This leads to 100% remote access for GPU0 (`0L-100R`).

Figure 1·3 shows the runtime for the `SGEMM` kernel execution with different matrix sizes for the above four matrix distributions. For smaller matrix sizes, accessing remote memory is very expensive because of the fixed remote access overhead. The runtime of `SGEMM` for the `0L-100R` distribution for a 4k×4k matrix is 27× longer than that of the `100L-0R` distribution. On the other hand, the runtime of `SGEMM` for the `0L-100R` distribution for the 32k×32k matrix is 12.2× longer than that of the `100L-0R` distribution. Here, the fixed remote access overhead gets amortized. From these experiments, we can see the significant impact of remote accesses on performance, and in turn, argue that to improve the performance of applications, we need to avoid remote accesses as much as possible.

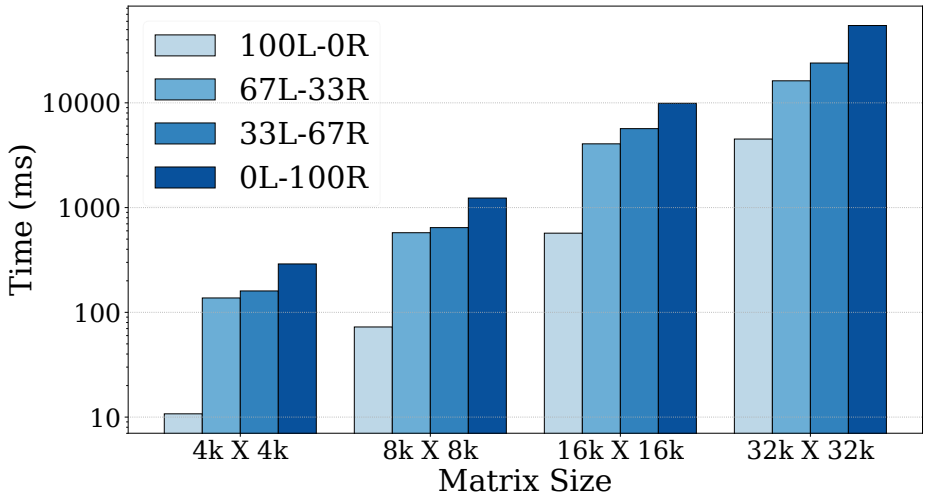


Figure 1·3: *Runtime of `SGEMM` kernel from cuBLAS library for different matrix sizes. Each bar corresponds to a different distribution of local and remote memory accesses.*

1.4 Thesis Contributions

The first part of this thesis focuses on understanding the behaviour of DNN applications, the performance limiting factors for training DNN workloads in MGPU systems and proto-

typing a potential solution to eradicate the performance bottlenecks. Based on the insights gained from DNN workload characterization, we propose and evaluate a novel and efficient MGPU system with True Shared Memory (MGPU-TSM) in the second part of the thesis. Finally, the third part of the thesis concentrates on solving one of the major challenges, i.e. coherence, in the proposed MGPU system. The key contributions of the thesis are provided in the sections below.

1.4.1 DNN Workload Characterization on MGPU Systems

To understand the characteristics of emerging workloads and the performance limitations in existing MGPU systems, we perform workload characterization on different generations of MGPU system. We breakdown the characterization in two parts– the first part presents workload characterization on MGPU systems prior to the launch of NVIDIA Volta-based MGPU systems, and the second part of the workload characterization is performed on the NVIDIA Volta-based MGPU systems.

Workload Characterization on Pre-Volta MGPU Systems

The key contributions of our workload characterization on the Pre-Volta system are as follows:

- We evaluate three different MGPU systems (NVIDIA Kepler, Titan Z and DGX-1) using different DNN workloads, and observe that communication among GPUs, as well as between the CPU and GPUs, can consume up to 40.6% of the total execution time.
- We develop and use synthetic workloads to evaluate data transfer mechanisms between two NVIDIA GPUs, and between a CPU and a GPU. This evaluation is useful for DNN algorithm developers to find the best communication mechanism for training their algorithms in existing MGPU systems.

- By forcing the GPUs in the MGPU system to use data from a shared main memory space, we demonstrate that a prototype (real) MGPU system with shared main memory improves performance for synthetic MNIST, Cifar10 and Imagenet datasets on a MLP network by 3.7×, 3.2× and 3.5×, on average, (across all three MGPU systems when using 2 GPUs) as compared to their respective baseline that uses P2P memcopy for gradient synchronization.

Workload Characterization on DGX-1 Volta MGPU Systems

For this characterization, we profile the three stages – Forward Propagation (FP), Backward Propagation (BP), and Weight Update (WU) of the DNN training process on a Volta-based NVIDIA DGX-1 system with 8 GPUs. The contributions of this work include:

- We compare the impact of P2P and NCCL based communication methods on the training time of DNN workloads (LeNet, AlexNet, GoogLeNet, ResNet and Inception-v3) on NVIDIA's Volta-based DGX-1 system. We profile these workloads to isolate and quantify the computation-intensive and the communication-intensive portion of the training process to identify the software and hardware-level bottlenecks.

- Our evaluation shows that MGPU communication latency cannot be hidden by simply increasing the computation-intensiveness of the workloads or compute capability of the GPUs. We also show that only increasing the bandwidth (BW) of the interconnect network in the MGPU system cannot completely eliminate the communication bottleneck. We also need an efficient implementation by the developers/programmers of DNN workloads to take advantage of the high BW interconnect.

- We quantify the impact of growing network size and increasing batch size on memory usage, and identify memory capacity to be a key limiting factor that hinders the speedup of the training of DNNs on MGPU systems.

1.4.2 MGPU Systems with True Shared Memory

In this section, we highlight our proposal to eliminate the major performance bottlenecks that we observe from our DNN workload characterization efforts. To simplify programming, eliminate the costly remote data movement latency, increase the memory utilization efficiency, and avoid redundant data copies, our solution is to physically unify the main memory of the GPUs. We refer to this system as a *MGPU system with true shared memory (MGPU-TSM)*. Unlike existing MGPU systems that rely on RDMA to access non-local GPU memory, an MGPU-TSM system allows all GPUs to directly access the entire physical main memory of the system, thus eliminating non-uniform memory access (NUMA) effects observed in traditional MGPU systems. The major contributions of this work are as follows:

- We are the first to demonstrate the performance benefits of MGPU-TSM. Our MGPU-TSM system eliminates NUMA accesses to main memory as well as the need to transfer data back and forth between the CPU's or GPUs' main memory.
- We implement the MGPU-TSM system architecture with 4 GPUs in MGPUSim simulator (Sun et al., 2019) and compare the performance of MGPU-TSM with existing MGPU systems using standard benchmarks. We demonstrate that MGPU-TSM achieves $3.81\times$ better performance on average versus an existing MGPU system that uses RDMA direct access.

1.4.3 Coherence in MGPU-TSM

To provide efficient support for both intra-GPU and inter-GPU coherence, in this thesis, we propose MGCC- a new timestamp-based hardware-level cache coherence scheme for MGPU-TSM systems. MGCC uses hardware-level coherence support for inter-GPU and intra-GPU data sharing using a single-writer-multiple-reader (SWMR) invariant. The contributions of the work are as follows:

- We propose a novel timestamp-based hardware-level inter-GPU and intra-GPU coherence protocol named MGCC, a scheme that ensures seamless data sharing in an MGPU-TSM system. MGCC leverages the concept of a *logical timestamp* (Plakal et al., 1998)

- MGCC introduces a new cache-level logical time counter to reduce traffic in the memory hierarchy. MGCC can reduce request traffic by up to 41.7% and response traffic by up to 3.1%. MGCC includes a novel timestamp storage unit (TSU) to keep track of cache block timestamps. We strategically place the TSU outside the critical path of memory requests. The TSU is accessed in parallel with the MM, thereby avoiding any performance overhead.

- MGCC also provides support for tracking memory requests to avoid potential load-after-store and store-after-load issues. Our implementation supports sequential and existing relaxed memory consistency models, but MGCC can also work as a building block for other memory consistency models, such as release consistency and total store ordering (TSO) consistency.

- We evaluate MGCC using both standard and synthetic benchmark suites with the MGPUSim simulator (Sun et al., 2019). Compared to an existing MGPU system supporting RDMA, for standard benchmarks, our MGPU-TSM system with MGCC delivers 3.7× and 3.0× better performance, on average, using RC and SC, respectively. Our MGCC protocol only introduces a 2% and 21% performance overhead, on average, for supporting RC and SC, respectively, when running standard benchmarks. Compared to MGPU systems utilizing the HMG coherence protocol (Ren et al., 2020), an MGPU-TSM system utilizing MGCC performs, on average, 2.4× better, when running standard benchmarks. Stress tests performed using our synthetic benchmark suite show that, compared to a non-coherent MGPU-TSM, a coher-

ent MGPU-TSM system with MGCC has 26.6% and 46.1% performance overhead for supporting RC and SC, respectively. However, this is a reasonable cost to pay for significantly lowering programmer burden.

1.5 Related Work

1.5.1 DNN Workload Characterization on MGPU Systems

As mentioned before, DNN has emerged as the most ubiquitous workload in the modern computing system. So, there are ongoing efforts to understand the behavior of DNN workloads on single GPU and MGPU systems, and to develop new software and hardware to achieve better performance.

Several prior works (Shi et al., 2016), (Bahrampour et al., 2015), (Kim et al., 2017), (Sze et al., 2017), (Li et al., 2016) have studied GPU-accelerated ML frameworks. Shi et. al. (Shi et al., 2016) benchmarked a selection of these frameworks including Caffe, CNTK, MXNet, TensorFlow, and Torch, using three popular types of deep neural networks (FCNs, CNNs, and RNNs) on two CPU platforms and three GPU platforms. The goal was to provide a guide for selecting appropriate combinations of the hardware platforms and software tools. Bahrampour et. al. (Bahrampour et al., 2015) present a comparative study of Caffe, Neon, TensorFlow, Theano, and Torch, from three different aspects, namely extensibility, hardware utilization, and performance. Kim et. al. (Kim et al., 2017) analyze the GPU performance characteristics of Caffe, CNTK, TensorFlow, Theano, and Torch from the perspective of a representative CNN model, focusing on the performance characteristics of these five frameworks on a single GPU system and a MGPU system. Unlike previous approaches, they measure layer-wise execution times, as well as the processing time, of an input batch. While these studies provide users some insight into the strengths and limitations of these deep learning frameworks, they do not introduce hardware or software modifications to improve the performance of these frameworks.

In comparison to the prior works, our DNN characterization work focuses on the both previous and current generations of MGPU systems to provide insight into the gradual evolution of MGPU architectures to support the emerging DNN workloads. Apart from that, unlike previous works, our characterization and profiling provide insight into communication– and computation–intensiveness of the DNN workloads.

1.5.2 NUMA Effects in MGPU System

MGPU system architecture is evolving over time along with the architecture improvement of individual GPUs. However, the works related to MGPU system architecture mostly focus on reducing the NUMA effects in existing MGPU systems. Hence, a number of recent works from both academia and industry have attempted to address the NUMA effects in existing MGPU systems, but they did not succeed in eliminating the NUMA effects completely. This is because eliminating NUMA effect requires drastic changes in the memory architecture and interconnect network architecture, and the previously proposed works mostly focus on achieving incremental improvements.

Milic et al. propose a NUMA-aware multi-socket GPU solution to resolve performance bottlenecks related to NUMA memory placement in multi-socket GPUs (Milic et al., 2017). The proposed system dynamically adapts inter-socket link bandwidth and caching policies to avoid NUMA effects. The proposed system exploits the changing application phase behavior in terms of inter-socket bandwidth demand and data locality in L1 and L2 caches by dynamically adapting the inter-socket link bandwidth and L1 and L2 caching policies, respectively. Our proposed MGPU-TSM system with MGCC completely eliminates the impact of NUMA on performance. Arunkumar et al. (Arunkumar et al., 2017) and Ren et al. (Ren et al., 2020) propose an MCM-GPU, where multiple GPU modules are integrated in a package to improve energy efficiency. As in MCM-GPU, our proposed MGPU-TSM can take advantage of novel integration technologies to improve energy efficiency and performance.

The recently introduced NVIDIA DGX-2 (NVIDIA, 2018a), (Choquette and Gandhi, 2020) system still suffers from NUMA effects as each GPU has its own local main memory and multiple GPUs still need to communicate via NVLinks. Although there has been improvement in the bandwidth of NVLink and the interconnect topology for connecting multiple GPUs to each other, the remote data access is still expensive as demonstrated in Section 1.3.2. Similarly, AMD introduced AMD Infinity Fabric Link (AMD, 2020) to improve the communication performance across MGPUs as well as between CPU and GPU. Kim et.al. (Kim et al., 2014) propose an MGPU system with unified memory network (UMN) that connects GPUs' main memory modules (knows as Hybrid Memory Cube or HMC (Pawlowski, 2011)) to provide faster connectivity across different GPUs and CPU. In particular, the authors propose a sliced flattened butterfly topology for the UMN by removing local HMC channels to ensure improved scalability. However, unlike our proposed MGPU-TSM system, the proposed UMN cannot provide GPUs with uniform memory access (UMA) to all the main memory modules. Arunkumar et al. (Arunkumar et al., 2019) also argue the need to improve inter-GPU communication. Our evaluation of MGPU-TSM also agrees with the argument.

In summary, our work on MGPU-TSM system is unique in the sense that no prior work comprehensively explored the impact of uniform memory access (UMA) to all the CPU and GPUs in the system.

1.5.3 MGPU Memory System and Coherence

In recent years, coherence in MGPU systems have gained significant attention. To support coherence in MGPU systems, researchers have proposed architectural changes from both software– and hardware–perspective in the memory design. In this section, we first discuss prior works related to CPU and GPU coherence, and then we discuss the prior works related to coherence in the MGPU domain.

CPU and GPU Coherence

A number of works proposed directory-based cache coherence for APUs (Alsop et al., 2018), (Power et al., 2013), (Hechtman et al., 2014)[3]. In contrast, we are the first to propose an efficient timestamp-based cache coherence protocol for an MGPU. Kumar et al. (Kumar et al., 2015) and Boroumand et al. (Boroumand et al., 2019) propose protocols for coherence between a CPU and GPU. These works are complementary to our own and can be leveraged to address CPU-GPU coherence issues. Qian et al. (Qian et al., 2010) proposes ScalableBulk, a directory-based cache coherence protocol for multicore CPUs. Since GPUs have heavier traffic patterns than CPUs, invoking different directories in a GPU environment to maintain coherence would lead to severe performance degradation. Thus, we believe ScalableBulk is not suitable for GPUs.

MGPU Coherence

NUMA-Aware multi-socket GPU (Milic et al., 2017) maintains inter-GPU coherence by extending SW-based coherence for L1$s to the L2$s. The resulting coherence traffic lowers application performance. Similarly, MCM-GPU (Arunkumar et al., 2017) leverages the software-based L1$ coherence protocol for its L1.5$. The flushing of the caches and coherence traffic hurt system scalability. Young et al. (Young et al., 2018) propose CARVE, a method where part of a GPU's memory is used as a cache for shared remote data and the GPU-VI protocol is used for coherence. This protocol does not scale well with an increase in the amount of read-write transactions and false sharing. Also, the CARVE method can cause performance degradation for workloads with large memory footprint as it reduces effective GPU memory space. HMG (Ren et al., 2020) is a recent hardware-managed cache coherence protocol for distributed L2$s in GPUs using a scoped memory model consistency. HMG proposes to extend a simple VI-like protocol to track sharers in a hierarchical

[3] An APU contains a CPU and a GPU in the same die

way, and achieves a cost-effective solution in terms of on-chip area overhead, inter-GPU coherence traffic reduction and high performance. This protocol, however, relies on error-prone scoped memory consistency model that increases programming complexity. Sinclair et al. (Sinclair et al., 2015) propose a memory consistency model for GPUs without using user-managed scopes of a variable in the memory hierarchy, while Alsop et al. (Kumar et al., 2015) propose a lazy release consistency model for GPUs. Both of these studies only consider single CPU and single GPU systems. Exploring efficient MGPU memory consistency models is an open challenge and results from these studies can be leveraged to address that challenge. We demonstrate the impact of RC and SC consistency in this work and our flexible MGCC protocol can be used as a building block for exploring other consistency models. To reduce coherence traffic, Singh et al. propose timestamp-based coherence (TC) protocol for intra-GPU coherence (Singh et al., 2013). As this protocol relies on a globally synchronized clock across all CUs, maintaining clock synchronization is a challenging task for large MGPU systems. To address this, Tabakh et al. (Tabbakh et al., 2018) propose a logical timestamp based coherence protocol (G-TSC). However, the G-TSC protocol is designed for single GPU systems and does not scale well for MGPU systems unlike our MGCC protocol.

All the prior MGPU coherence-related work focused on imposing a light-weight coherence protocol to improve NUMA effects. However, MGPU-TSM with MGCC provides the means to completely eradicate NUMA effects and impose a very light-weight low-overhead coherence protocol.

1.6 Organization

The rest of the thesis is organized as follows:

- Chapter 2 discusses the workload characterization efforts on different generations of MGPU systems to quantitatively and qualitatively understand the inefficiencies in

existing MGPU systems.

- Chapter 3 describes our proposed solution MGPU-TSM to overcome the inefficiencies in existing MGPU systems.
- Chapter 4 presents theory and evaluation of our proposed MGCC coherence protocol for MGPU-TSM.
- Chapter 5 discusses the future reserach direction and concludes the thesis.

Chapter 2

DNN Workload Characterization on Existing MGPU Systems

In this chapter, we present our DNN workload characterization methodology and profiling results when running DNN workloads on different generations of NVIDIA MGPU systems. The work in this chapter is motivated by the fact that DNN workloads have become ubiquitous and MGPU system architectures are evolving to support the growth of DNN workloads. Our objective is to understand the DNN workloads, the impact and limitations of MGPU hardware to train DNN workloads, and provide insights into the improvements necessary to develop next-generation MGPU systems. In Section 2.1, we discuss our characterization and evaluation of the training of DNNs on three NVIDIA pre-Volta MGPU systems– Kepler, Titan Z and DGX-1. Section 2.2 presents the in-depth profiling of DNN workloads on NVIDIA DGX-1 MGPU system with 8 V100 GPUs. Section 2.3 presents our prototype solution, true shared memory (TSM) system, to lower the communication overhead.

2.1 Workload Characterization on Pre-Volta MGPU Systems

DNN workloads (LeCun et al., 2015), (Jordan and Mitchell, 2015), (Libbrecht and Noble, 2015), (Kourou et al., 2015), (Alsheikh et al., 2014) are presently being used in a number of application domains. To maximize the potential of DNNs, we need to carefully perform training. However, training DNN algorithms is very challenging. Apart from the fact that the training involves tuning several parameters that require experience and intuition,

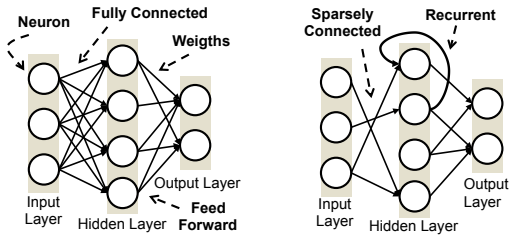
it involves performing thousands of matrix operations, which can be very time consuming. Given the inherent parallelism available on today's GPUs, we can reduce the cost of these matrix operations, and in turn, accelerate the DNN training process (Chen and Lin, 2014), (Schmidhuber, 2015), (Zhang et al., 2015). However, as the size of DNN workloads scales up, single GPUs are not enough, and we need to consider moving to MGPU systems for training (Coates et al., 2013). Nonetheless, performance of DNN workloads does not scale with GPU count, as MGPU system performance is limited by its memory system, number of compute units and communication among different GPUs. Our work focuses on identifying the bottlenecks in MGPU systems when running DNN workloads and developing solutions to remove these bottlenecks.

2.1.1 DNNs

In this section, we provide an overview of three different artificial neural network (ANN) based DNN algorithms used for evaluating MGPU systems. In particular, we focus on the Multi-Layer Perceptron (MLP) Networks, which are good at handwritten character recognition (Singh and Sachan, 2014); Convolutional Neural Networks (CNNs), which excel in accurate image understanding (Krizhevsky et al., 2012) and speech recognition (Sainath et al., 2013); and Recurrent Neural Networks (RNNs) (Hochreiter and Schmidhuber, 1997), which are typically employed for efficient natural language processing (Graves et al., 2013).

Multi-Layer Perceptron (MLP)

A typical MLP network is shown in Figure 2·1a. It consists of an input layer of artificial neurons (ANs), one or more hidden layers of ANs (example shows one hidden layer) and an output layer of ANs. Each neuron in the hidden layer(s) and output layer computes a weighted sum over values arriving on its input edges, applies an activation function to compute an output, and propagates this output to neurons in the next layer. An MLP has two modes of operation: *inference or feedforward computation* and *training*.



(a) (b)

Figure 2·1: Artificial Neural Networks commonly used in Deep Learning. (a) Multi Layer Perceptron (MLP). (b) Convolutional Neural Network (CNN): AlexNet (Krizhevsky et al., 2012). CONV = Convolutional layer, MP = Max-Pooling layer, and FC = Fully-Connected Layer. (c) Recurrent Neural Network (RNN) with sparsely-connected neurons.

Inference/Feedforward Computation: To begin processing, an input vector is sent to the first hidden layer (the input layer in an NN is pass-through). Each AN computes its output, z, by applying an *activation function* (e.g., a sigmoid), σ, to an input–weight inner product, y:

$$y = \sum_{\forall \text{ weights}} \text{weight} \times \text{input} \tag{2.1}$$

$$z = \sigma(y) \tag{2.2}$$

Training using Error Back Propagation and Weight Updates: Learning, or training, involves modifying the weights of inter-neuron connections in an attempt to minimize an output cost function (e.g., the mean squared error (MSE)). The challenge of learning is then rapidly determining the relative contribution of each weighted connection towards this cost function, i.e., the partial derivative of the cost function with respect to each weight. By deliberately choosing differentiable activation functions, output derivatives (or errors) *backpropagate* through the network in a single backward sweep, much faster than via the chain rule. In the update step, weights are moved against their individually computed derivatives to decrease the cost function.

During training, each output neuron uses its expected output value, $E[z_{out}]$, to compute MSE cost function derivative, δ. This derivative requires the activation function derivative, σ', which is defined in terms of known parameters for a sigmoid (Equation 2.3). The output error, $E[z_{\text{out}}] - z_{\text{out}}$, is amplified with an inverse hyperbolic tangent function that penalizes large errors (Equation 2.4):

$$\sigma'_{\text{sigmoid}}(y) = \sigma_{\text{sigmoid}}(y) \times (1 - \sigma_{\text{sigmoid}}(y)) \tag{2.3}$$

$$\delta_{\text{out}} = \sigma'(y_{\text{out}})(E[z_{\text{out}}] - z_{\text{out}}) \tag{2.4}$$

Output derivatives are broadcast *backwards* along the input connections of an output neuron, multiplied by their connection weights, and accumulated at the previous layer nodes (see Equation 2.5). Scaled by the activation function derivative (σ') this product forms the cost function derivative for each hidden node:

$$\delta_{\text{hidden}_{i-1}} = \sigma'\left(y_{\text{hidden}_{i-1}}\right) \sum_{\forall \text{weights}} \delta_{\text{out}_i} \times \text{weight}_i \tag{2.5}$$

Weights can be updated after every input–output pair—as in stochastic gradient descent—or the accumulated weight updates of all input–output pairs can be batched and applied at once—as in standard gradient descent. Partial weight updates, Δw, for each input–output training pair are determined by multiplying the neuron-specific cost function derivative, δ, by the current *input* seen along that connection:

$$\Delta w = \delta \times \text{input} \tag{2.6}$$

All accumulated partial weight updates are finally scaled by a user-specified learning rate (divided by the number of training pairs) and added to the old weight value:

$$\text{weight} += \frac{\text{learning rate}}{\text{\# training pairs}} \times \sum_{\forall \text{ training pairs}} \Delta w \tag{2.7}$$

This process of error back propagation and weight updates is repeated until the cost func-

tion is minimized.

Convolutional Neural Network (CNN)

CNN is a feed-forward sparsely-connected ANN which is composed of multiple types of layers. Figure 2·1b shows a popular CNN called AlexNet (Krizhevsky et al., 2012), which is made up of 5 convolutional layers (CONV) which are used to create a higher-level abstraction of the input data, called a *feature map* or *fmaps* (features in Figure 2·1b); 3 Max-Pooling (MP) layers, used for ignoring small shifts and distortions (noise) in the input data; and 3 fully-connected (FC) layers, which are used for classification.

Inference/Feedforward Computation: As described in (Sze et al., 2017), each CONV layer involves computation of high-dimensional convolutions, where a set of 2D input feature maps (*ifmaps*), called a channel, is convoluted with a distinct 2D *filter*, and the results of the convolution at each point are summed across all the channels. The result of this computation is one channel of the output feature map (*ofmap*). Finally, multiple stacks of ifmaps may be processed together as a batch to improve reuse of the filter weights. The computation is described in Equation 2.8, assuming the shape parameters in Table 2.1.

Table 2.1: Shape parameters of a CONV/FC layer

Shape Parameter	Description
N	Batch size of 3D fmaps
M	# of 3D filters / # of ofmap channels
C	# of ifmap/filter channels
P	ifmap plane width/height
R	filter plane width/height (=P in FC)
E	ofmap plane width/height (=1 in FC)

Here, **O**, **I** and **W** are the matrices for the ofmaps, ifmaps and filters, respectively.

$$\mathbf{O}[z][u][x][y] = f(\sum_{k=0}^{C-1}\sum_{i=0}^{R-1}\sum_{j=0}^{R-1}\mathbf{I}[z][k][Ux+i][Uy+j] \times \mathbf{W}[u][k][i][j])$$
$$0 \leq z < N; 0 \leq u < M; 0 \leq x, y < E; E = (P-R+U)/U \quad (2.8)$$

The MP computation involves estimating the maximum value in a given set of elements of a vector, matrix or cube, so that the output is a smaller representation of the quadrant (just a single value). This is a filter applied after the CONV layers to avoid noise and extract the most significant information for the next layer (CONV or FC, as it is shown in Figure 2·1b). Finally, a small number of FC layers are used for classification purposes, where Equation 2.8 still holds for the computation with $P = R$, $E = 1$ and $U = 1$.

Training using Error Back Propagation and Weight Updates: The backward propagation of a CNN involves estimating the gradients for the last layer of the CNN, and propagating the gradients back to the first layer. The principles of this computation are based on Equation 2.7 and depend on the type of layer in the CNN (CONV, MP and FC). For the sake of brevity, we do not include the formulation of each case. For further details, we refer the reader to (LeCun et al., 1998).

Recurrent Neural Network (RNN)

An RNN is categorized as a feed-back or recurrent ANN. As shown in Figure 2·1c, an RNN contains cyclic connections that pass the previous outputs of a hidden layer back to the current input of the hidden layer. These recurrent connections can take the time dimension into account to predict sequential data, which is very useful for speech recognition and machine translations tasks.

Inference/Feedforward Computation: An RNN computes the hidden layer vector sequence $H = (h_1, ..., h_T)$ that maps an input sequence $X = (x_1, ..., x_T)$ to an output sequence $Y = (y_1, ..., y_T)$, by iterating the following equations from $t = 1$ to T:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1}) \tag{2.9}$$

$$y_t = W_{hy}h_t \tag{2.10}$$

Where W_{xh} is the weight matrix between the input layer and the hidden layer, and W_{hy} corresponds to the weight matrix between the hidden layer and the output layer. Moreover, W_{hh} represents the weight matrix of recurrent connections between two consecutive time steps of the hidden layer states. Finally, σ is the hidden layer activation function ($f(.)$), an element-wise $sigmoid()$ function.

In this work, we will focus on RNNs with Long Short Term Memories (LSTMs) (Hochreiter and Schmidhuber, 1997) due to their popularity to obtain high prediction accuracy (Graves et al., 2013). LSTM can maintain long-term memory by relying on memory cells in order to decide what to remember, what to forget and what to output. For that, a standard LSTM cell implements input, forget and output gates. We omit a discussion of these gates for the sake of brevity. For further information, we refer the reader to (Hochreiter and Schmidhuber, 1997). An LSTM extends Equation 2.9 and Equation 2.10 as follow:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1}) \tag{2.11}$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1}) \tag{2.12}$$

$$\vec{c_t} = tanh(W_{xc}x_t + W_{hc}h_{t-1}) \tag{2.13}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \vec{c_t} \tag{2.14}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1}) \tag{2.15}$$

$$h_t = o_t \odot tanh(c_t) \tag{2.16}$$

where σ is the $sigmoid()$ function, and f, i, $\vec{c}$, o are the output vectors of forget, input, memory cell and ouput gates, respectively. W_{ab} represents a weigh matrix (e.g., if $a = x, b = i$, W is the weight matrix between input gate i and input vector x).

Training using Error Back propagation and Weight Updates: The backward pass of an LSTM-RNN computes the loss function of Equation 2.7 that depends on the output of the hidden layer at both current time and next time steps. For further information, we refer the reader to (Hochreiter and Schmidhuber, 1997).

2.1.2 Evaluation Methodology

MGPU System Description

In our evaluation, we use a Kepler system (one NVIDIA K40c and one NVIDIA K20c GPU), a Titan system (2 NVIDIA Titan Z GPUs) and a NVIDIA DGX-1 system (8 NVIDIA P100 GPUs). Table 2.2 shows the specifications of the Kepler, Titan and DGX-1 MGPU systems. The Kepler system uses a PCIe generation 2 (PCIe 2.0) interconnect, while the Titan system uses a PCIe generation 3 (PCIe 3.0) interconnect. Both systems consist of GPUs of same NVIDIA Kepler architecture with similar compute capability. We use compute capability to refer to the number of cores and clock rate of a GPU. Given that the bandwidth of PCIe 3.0 interconnect is twice the bandwidth of of PCIe 2.0, the comparison between the Titan and Kepler systems provide some insight into the potential impact of network bandwidth when running DNN workloads in a MGPU system. The DGX-1 system has higher compute capability and interconnect bandwidth. In terms of compute capability, each P100 GPU in the DGX-1 system is 2.73$\times$ and 2.36$\times$ faster than K20c and Titan Z GPUs, respectively. Each unidirectional link in the NVLink is 1.25$\times$ and 2.5$\times$ faster than each of the unidirectional PCIe 3.0 and PCIe 2.0 links, respectively. Comparing performance across the DGX-1, Titan and Kepler systems help understand the importance of both increased compute capability and bandwidth. More importantly, this comparison

Table 2.2: MGPU Systems evaluated in this work.

System	#GPUs	#Cores per GPU	Clock Speed (MHz)	Memory (GB)	Memory BW (GB/s)	Inter-connect	Archi-tecture
Kepler	1 (K40c) 1 (K20c)	2880 2496	745 706	12 5	288 208	PCIe 2	Kepler
Titan	2 (Titan Z)	2880	705	6	363	PCIe 3	Kepler
DGX-1	8 (P100)	3584	1328	16	732	NVLink 1	Pascal

provides valuable insights for future research directions for both GPU hardware (architecture for MGPU systems) and software (management of computation and communication resources in MGPU systems).

Workloads

In this work, we use the network model for MLP, LeNet, ResNet, AlexNet, GoogLeNet and Inception-v3 available with the MXNet framework. We use both real-world datasets (MNIST and CIfar10) and the synthetic ImageNet dataset for training the networks to evaluate the performance of different MGPU systems. We use MXNet release version 0.11.0, which requires cuDNN 5 for CUDA 8.0. The overall computation time and communication time for the training of the workloads depends on the batch size, the number of weights in the network, the input dataset and the number of epochs. Since one epoch is representative of the entire training process using the entire dataset, we train the networks for only one epoch. We use a different combination of datasets and networks. Hence, the total number of weights in a network will vary based on the input dataset. Table 2.3 shows the number specification of the workloads used in our work. We show the approximate number of weights for the CNN networks. The 3-layer-MLP network has one hidden layer which has 128 neurons. The LSTM network has 2 stacked RNN layers with 200 neurons in the hidden layer. We use MNIST, Cifar10, synthetic ImageNet and PennTreeBank dataset in

our evaluation.

Table 2.3: Specifications of the CNN Networks used in this Work. More details about the CNN workloads can be found in Section 2.2.1.

Network	Conv. Layers	FC Layers	Weights
LeNet	2	2	∼60k
ResNet-110	109	1	∼1.7M
GoogLeNet	21	1	∼7M
AlexNet	5	3	∼61M

Profiling Tools

We use *nvprof* and NVIDIA's Visual Profiler to profile the workloads. We calculate the total computation time and communication time (i.e. host-to-device data copy time + device-to-host data copy time + device-to-device data copy time) using the profilers. We use the internal timer of MXNet to obtain the run time of a DNN workload for one epoch. Profiling applications with *nvprof* requires a large amount of GPU memory (as opposed to the applications running standalone). As a result, it is not possible to profile larger workloads using *nvprof* or the Visual Profiler. Nonetheless, we gain insights about the performance of MGPU systems by profiling smaller workloads, and based on those insights, we analyze the performance of larger workloads run on a MGPU system.

2.1.3 Evaluation Results

In this section, we first present our evaluation of the Kepler, Titan and DGX-1 MGPU systems using DNN workloads. We explore the impact of changing the batch size on communication and computation, while training a DNN workload. We present the results obtained from both *nvprof* and the MXNet timer for different DNN workloads, as run with 2 GPUs on the Kepler, Titan and DGX-1 systems. We show how the workloads scale on

the DGX-1 system for 2, 4 and 8 GPUs; then, we analyze and identify the factors affecting scaling.

Effect of Changing Batch Size

Here, we show the effect of batch size for training a neural network to gain insights into the behavior of larger workloads that we will evaluate in the later sections. First, we evaluate the performance of a 3-layer MLP and LeNet (CNN) network with MNIST dataset to understand the effect of batch size on the degree of computation and communication. We use 2 GPUs from each of the three MGPU systems for this evaluation.

Figure 2·2a shows the computation time for both MLP and CNN workloads for the three MGPU systems. For smaller batch sizes, the number of kernel launches is large, although the size of the kernels is very small in terms of computation. Hence, most of the GPU computing resources remain under-utilized for small batch sizes. The computation time saturates after the batch size reaches 128 for both workloads. There are two factors that affect the computation time. First, as the batch size increases, each GPU needs to process fewer batches, which helps to reduce computation time. Second, each GPU has to perform more computations per batch. This increases the computation time. After the batch size of 128, these two factors cancel each other out. But at smaller batch sizes kernel launching overheads and under-utilization of GPU's compute resources lead to performance degradation.

The workload with LeNet network is more computationally-intensive than the MLP workload. Given that the Kepler system performance is limited by the weaker K20c GPU with 2496 cores, the Titan system with 2880 CUDA cores performs 1.4× faster than the Kepler MGPU system for the LeNet network with a batch size of 64. As workload with MLP network requires lesser amount of computation than the LeNet network, the Titan system achieves only a 1.06× speedup over the Kepler system for a batch size of 64. Each P100 GPU in the DGX-1 system has a 1.9× higher clock rate as compared to the Titan Z or

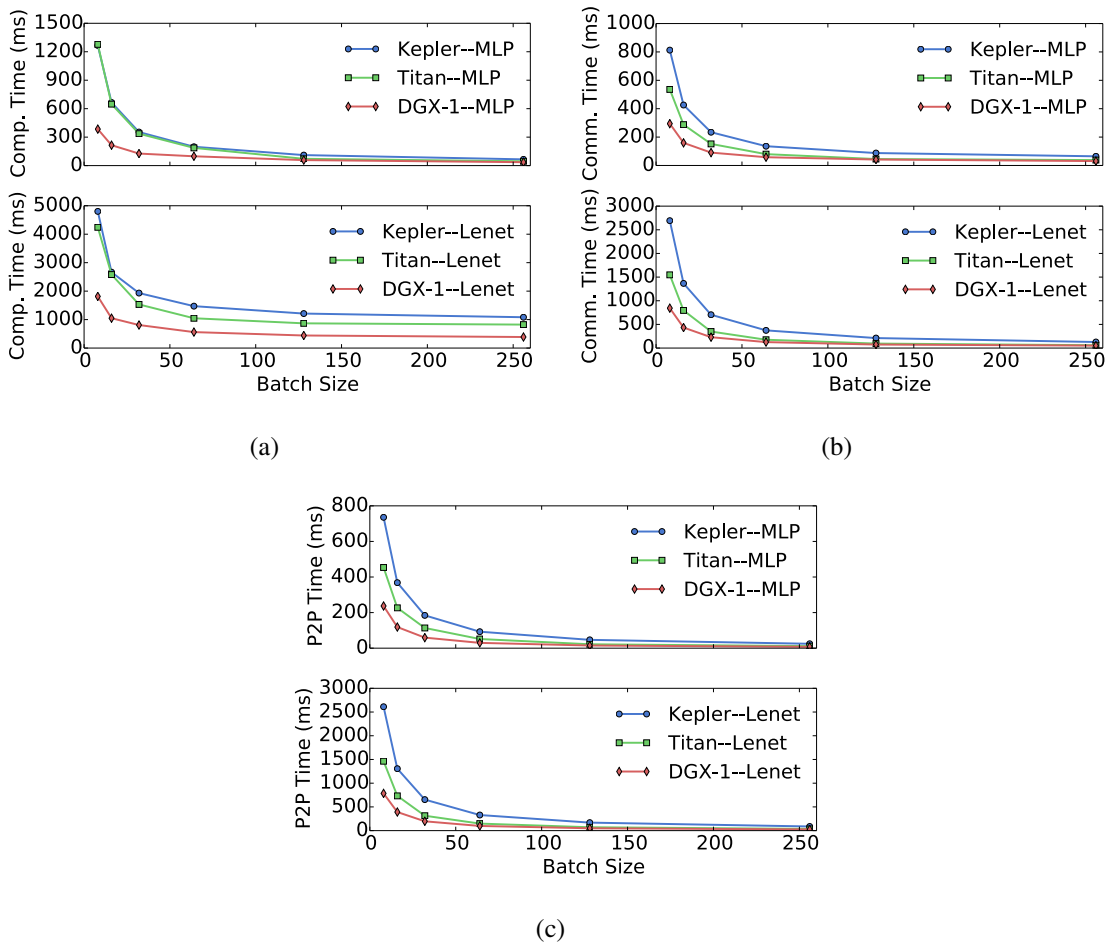


Figure 2·2: Effect of changing batch size for MNIST dataset for training in a 3-layer-MLP and LeNet network using 2 GPUs of the Kepler, Titan and DGX-1 system.

the K20c GPUs. Furthermore, P100 GPU has 1.44× and 1.24× more CUDA cores than the K20c and Titan Z, respectively. Hence, in terms of clock rate and compute resources, each P100 GPU in the DGX-1 system is expected to obtain 2.73× and 2.36× faster computation as compared to the K20c and the Titan Z GPUs, respectively. The DGX-1 system performs 2.02× and 2.63× faster than Kepler system for the MLP and CNN workload, respectively, with a batch size of 64. For the same batch size, the DGX-1 system computes 1.9× and 1.87× faster than the Titan system for the MLP and CNN workload, respectively.

Figure 2·2b and Figure 2·2c show the communication time and P2P memcpy time, re-

spectively, for the three MGPU systems for both workloads. At smaller batch sizes the overall communication time is high becasue of a large number of P2P memcpy for gradient synchronization. As the batch size increases, the number of P2P memcpy decreases. After the batch size of 128, the total P2P memcpy time is small relative to the overall communication time, as communication is dominated by the CPU-to-GPU memcpy time. The P2P memcpy time also allows us to compare the performance of PCIe 2.0, PCIe 3.0 and NVLink 1.0. For the LeNet network workload, using a batch size of 64, Titan and DGX-1 systems require 2.22× and 3.34× less P2P memcpy time, respectively, as compared to the Kepler system.

Scaling of Workloads to 2 GPUs

Table 2.4 shows a breakdown of the runtime in terms of communication time and computation time obtained using *nvprof* for different workloads employing 1 and 2 GPUs on Kepler, Titan and DGX-1 MGPU systems. We also report P2P memcpy time for 2 GPU cases. Note that the overhead for the CUDA API calls is different for different MGPU systems under consideration because of the CPU configuration and network used in each MGPU systems of the three MGPU systems. Hence, comparing the total run time may not be fair. We use runtime to calculate scaling factor for different number of GPUs within the same MGPU system.

For both the MNIST and Cifar10 datasets with 3-layer MLP networks, none of the three MGPU systems achieve good scaling. This happens because our datasets are too small in terms of the input image size, and the network is also very small in terms of number of layers and neurons per layer. Consequently, the GPUs launch a number of very lightweight kernels to perform forward and backward propagation to train the workload. If we compare the P2P memcpy time for 2 GPUs across the three MGPU systems, we see that DGX-1 system achieves 3.2× and 1.94× shorter communication times than Kepler and

Table 2.4: Profiling results for different workloads on 1 and 2 GPUs in Kepler, Titan and DGX-1 MGPU systems

Dataset	Network	Batch Size	GPU	Run Time (ms)	Comm. Time (ms)	Comp. Time (ms)	P2P Memcpy Time (ms)	Scaling Factor
MNIST	MLP	64	Kepler K20c	845	43	370	–	1
			Kepler 2 GPU	750	135.7	198.4	92.5	1.13
			Titan Z 1 GPU	639	26.5	331.5	–	1
			Titan Z 2 GPU	609	87.7	191.3	56.8	1.05
			P100 1 GPU	1090	24.8	192	–	1
			P100 2 GPU	880	55	98	29.3	1.2
MNIST	LeNet (CNN)	64	Kepler K20c	2795	44.1	2411	–	1
			Kepler 2 GPU	2006	374.8	1468	329.3	1.4
			Titan Z 1 GPU	2259	24.4	1874	–	1
			Titan Z 2 GPU	1401	165.6	976.4	139.4	1.6
			P100 1 GPU	2336	26	1017	–	1
			P100 2 GPU	1261	125.8	567	98.4	1.85
Cifar10	MLP	64	Kepler K20c	2263	102.1	420	–	1
			Kepler 2 GPU	1951	305.2	195	200.7	1.16
			Titan Z 1 GPU	2256	51.4	416.9	–	1
			Titan Z 2 GPU	2080	179.3	302.1	100.6	1.08
			P100 1 GPU	2219	53.6	225.6	–	1
			P100 2 GPU	2000	127.4	110.8	58.8	1.1
Cifar10	LeNet (CNN)	64	Kepler K20c	3769	102.7	2582.9	–	1
			Kepler 2 GPU	3251	383.2	1263.7	279.8	1.16
			Titan Z 1 GPU	3307	52.61	1813	–	1
			Titan Z 2 GPU	2596	217.4	1185	149.1	1.27
			P100 1 GPU	2747	57	915.6	–	1
			P100 2 GPU	2111	145.7	512.2	83	1.3
Pen-Tree-Bank	LSTM (RNN)	128	Kepler K20c	23800	17.5	23127	–	1
			Kepler 2 GPU	14700	1208.8	13605	1163	1.6
			Titan Z 1 GPU	20981	13.7	19449	–	1
			Titan Z 2 GPU	11596	501.5	10387	466.0	1.81
			P100 1 GPU	14800	17.0	10059.2	–	1
			P100 2 GPU	8900	373.6	5040.8	332.6	1.7

Titan systems, respectively. This is because of the higher bandwidth provided by NVLink for P2P memcpy.

To show how workloads with heavier kernels (kernels that perform more computations) can amortize the kernel launch overhead and scale better in a MGPU system, we kept the dataset the same, but use a larger LeNet network that increases computation. For example, the GPUs in the DGX-1 system perform 5.8× more computation, with an increase in communication time of only 2.3× for the workload with LeNet network as compared to the

Table 2.5: Evaluation results for different CNN workloads using 2 GPUs of the MGPU systems

Dataset	Network	Batch Size	GPU	Run Time (s)	Scaling Factor
ImageNet	LeNet	16	Kepler K20c	20.4	1
			Kepler 2 GPU	17.2	1.18
			Titan Z 1 GPU	14.4	1
			Titan Z 2 GPU	10.4	1.38
			P100 1 GPU	10.6	1
			P100 2 GPU	9.6	1.1
ImageNet	AlexNet	16	Kepler K20c	14.4	1
			Kepler 2 GPU	10.4	1.38
			Titan Z 1 GPU	10.1	1
			Titan Z 2 GPU	7.4	1.36
			P100 1 GPU	5.8	1
			P100 2 GPU	6.0	0.97
ImageNet	GoogLeNet	16	Kepler K20c	25.9	1
			Kepler 2 GPU	13.2	1.96
			Titan Z 1 GPU	16.8	1
			Titan Z 2 GPU	9.2	1.83
			P100 1 GPU	7.4	1
			P100 2 GPU	4.4	1.68
ImageNet	ResNet	16	Kepler K20c	75	1
			Kepler 2 GPU	38.3	1.95
			Titan Z 1 GPU	55.6	1
			Titan Z 2 GPU	28.6	1.94
			P100 1 GPU	18.4	1
			P100 2 GPU	10.1	1.82
Cifar10	ResNet	16	Kepler K20c	263.8	1
			Kepler 2 GPU	147.8	1.5
			Titan Z 1 GPU	173.2	1
			Titan Z 2 GPU	103.5	1.67
			P100 1 GPU	147	1
			P100 2 GPU	90.4	1.63

workload with the MLP network.

As mentioned in Section 2.1.2, larger workloads cannot be profiled using *nvprof* because of insufficient GPU memory. Table 2.5 shows the results for the 2 GPU configurations on the Kepler, Titan, and DGX-1 systems processing the ImageNet dataset, while considering different networks, and Cifar10 dataset with ResNet network. The batch size cannot be increased beyond 16 per GPU on the Kepler system because of the 5GB memory on the K20c GPU. The scaling factor for the 2-GPU Kepler system, as compared to the single K20c GPU, is close to 2 when training the GoogLeNet and ResNet networks using ImageNet datasets. The AlexNet network does not scale when using a batch size of

16 because this network has only 5 convolutional layers, but approximately 61M weights. Hence, the P2P memcpy for the gradient synchronization requires a large amount of time. In case of DGX-1 system, a batch size of 16 per GPU cannot utilize the compute resources fully for any of the workloads.

To evaluate the effect of input data size on scaling, we change the dataset to Cifar10, which is a smaller input dataset. As the input data size become smaller, the amount of computation is reduced. Hence, this workload does not scale beyond 1.67 in any of the three MGPU systems.

Scaling in DGX-1 MGPU System

In this section, we present results for workload scalability beyond 2 GPUs in DGX-1 system. We use weak scaling for this study.[1] We consider the ImageNet dataset to train AlexNet, GoogLeNet, and ResNet networks. To investigate the performance bottlenecks in the DGX-1 system, we vary the batch size per GPU to train different CNN networks on 1, 2, 4 and 8 GPUs, separately.

In previous sections, we have shown that the smaller workloads such as Cifar10 with LeNet and the 3-layer-MLP network, ImageNet dataset with AlexNet and the GoogLeNet network do not scale well for 2 GPUs in the DGX-1 MGPU system for a batch size of 16. Here, we increase the batch size of the ImageNet dataset.

Table 2.6 shows the results of scaling on 2, 4 and 8 GPUs for different workloads. Beyond 2 GPUs, the workload with the AlexNet network does not scale well as the batch size is increased from 32 to 64. This because the network is very small and cannot utilize the P100 GPUs' computing resources.

The GoogLeNet and ResNet workloads tend to show improved scaling as the batch size

[1]Strong scaling of the training of DNNs is the speedup in training time as we increase the GPU count while keeping the size of the input dataset fixed. Weak scaling is the speedup in training as we increase the GPU count and the size of the input dataset by a factor equal to the GPU count.

[2]We do not report results for smaller MLP and RNN networks that do not scale well for 2 GPUs. We only report the results for the workloads that have potential to scale beyond 2 GPUs.

Table 2.6: Scaling in DGX-1 MGPU system for different CNN workloads[2].

Dataset	Network	Batch Size	# of GPUs	Run Time (s)	Scaling Factor
ImageNet	AlexNet	32	1	21	1
			2	14.0	1.5
			4	10.1	2.1
			8	13.7	1.53
ImageNet	AlexNet	64	1	34	1
			2	20.3	1.67
			4	15.6	2.18
			8	22.9	1.48
ImageNet	GoogLeNet	32	1	46.3	1
			2	24.8	1.87
			4	13.5	3.43
			8	10.0	4.63
ImageNet	GoogLeNet	64	1	85.2	1
			2	45.9	1.86
			4	23.6	3.61
			8	14.6	5.84
ImageNet	ResNet	32	1	130.1	1
			2	65.8	1.98
			4	34.0	3.83
			8	20.9	6.22
ImageNet	ResNet	64	1	246.9	1
			2	127.1	1.94
			4	64.5	3.83
			8	34.8	7.1

increases. The GoogLeNet achieves a scaling of 3.61 when the batch size is increased to 64 from 32 with 4 GPUs, but it does not scale similarly for 8 GPUs. Our largest network, which in terms of the number of convolutional networks is ResNet, achieves the highest scaling of 7.1 for 8 GPUs. Hence, our evaluation of DGX-1 system in this section shows that workloads with larger number of weights and small number of convolutional layers struggle to scale in a MGPU systems. This is because of the communication bottlenecks that is present in the existing MGPU systems.

2.1.4 Summary

With the widespread adoption of DNNs in applications, it has become imperative to provide fast and efficient training of the DNNs. Because of the inherently parallel nature of DNNs, MGPU systems can train DNNs much faster than high-end CPUs. However, even

on MGPU systems, the training of DNNs consumes a large amount of time as it requires numerous data transfers between CPU and GPUs, and among multiple GPUs. To further accelerate the training of DNNs, we need to improve throughput and reduce the communication time in MGPU systems.

In this work, we evaluate three different high performance MGPU systems– an NVIDIA Titan with 2 GPUs, a Kepler-based cluster with 2 GPUs, and a DGX-1 with 8 GPUs, using a variety of DNN workloads, with the goal of identifying the performance bottlenecks of these systems. We profile MLP–, CNN– and RNN–based workloads using these MGPU systems. We show that for DNN workloads, communication among the GPUs and between the CPU and GPUs can consume up to 40.6% of total execution time. Based on this observation, we evaluate different types of communication mechanisms (zerocopy, memcpy and unified memory) using our synthetic workloads that mimic the communication pattern of these DNN workloads.

2.2 Workload Characterization on DGX-1 Volta MGPU Systems

In this section,we focus on DNN workload characterization on NVIDIA's Volta-based DGX-1 system. In particular, we present the limits and opportunities for training the following five DNN workloads: GoogLeNet, AlexNet, Inception-v3, ResNet and LeNet on the DGX-1 system. These workloads represent a wide mixture of computation and communication behavior. We present a thorough analysis of how well the training of various DNNs scales with GPU count on the DGX-1 multi-GPU system. We identify hardware-level (i.e. computation, communication and memory capacity) and software-level (i.e. NCCL library, the programming framework for DNNs, etc.) bottlenecks present in the training of the various DNNs using multi-GPU systems. The detailed descriptions about the background of this work have been presented in Section 1.2.1 ans Section 2.1.1. Hence, we present our evaluation methodology and results in the following sections.

2.2.1 Evaluation Methodology

To gain insights into the factors affecting the speedup of DNN training in a multi-GPU environment, we compare two most popular inter-GPU communication methods , i.e., P2P memcpy and NCCL. With the help of `nvprof` profiler, we isolate the computation-intensive and the communication-intensive portion of the workloads to identify the computation and communication bottlenecks in the underlying hardware.

Evaluation Platform

Our evaluations are performed on NVIDIA's Volta-based DGX-1 system (NVIDIA, 2016). The DGX-1 system has two 20-Core Intel Xeon E5-2698 v4 CPUs and eight Tesla V100 GPUs. Each Tesla V100 GPU incorporates 80 streaming multiprocessors (SMs) and delivers 17.7 TFLOPs single precision computing capability. The Tesla V100 GPU also features eight tensor cores that serve as dedicated hardware that can accelerate matrix operations. With the tensor cores, a Tesla V100 GPU can achieve a throughput of 125 TFLOPs, which is 7$\times$ faster than only using the traditional single precision computing devices. Since we focus on the DNN workloads where matrix multiplication is the most common operation, the tensor cores are utilized to accelerate the training of the DNNs.

Figure 2·3 shows a high-level view of the network topology of the Volta-based DGX-1 system. As depicted in Figure 2·3, the CPUs in the DGX-1 system use the PCIe bus to communicate with the GPUs, while the GPUs are connected with high-bandwidth peer-to-peer NVLink connections. Each NVLink connection delivers 25 GB/s data transfers in each direction. For GPU pairs that have more than one connection, NVLink can aggregate the connections and provide a 50 GB/s virtual connection. Each CPU has direct access to only four GPUs. To access the other four GPUs it needs the help of the other CPU. Some GPUs have two direct connections between them (e.g. between GPU 0 and GPU 2), while some GPUs have only one direct connection between them (e.g. between GPU 2

and GPU 3). Moreover, some GPUs may not have a direct connection between them (e.g. between GPU 3 and GPU 4), and they require the help of another GPU or two CPUs for communication. A maximum of one intermediate node (two hops) is required to connect any pair of GPUs.

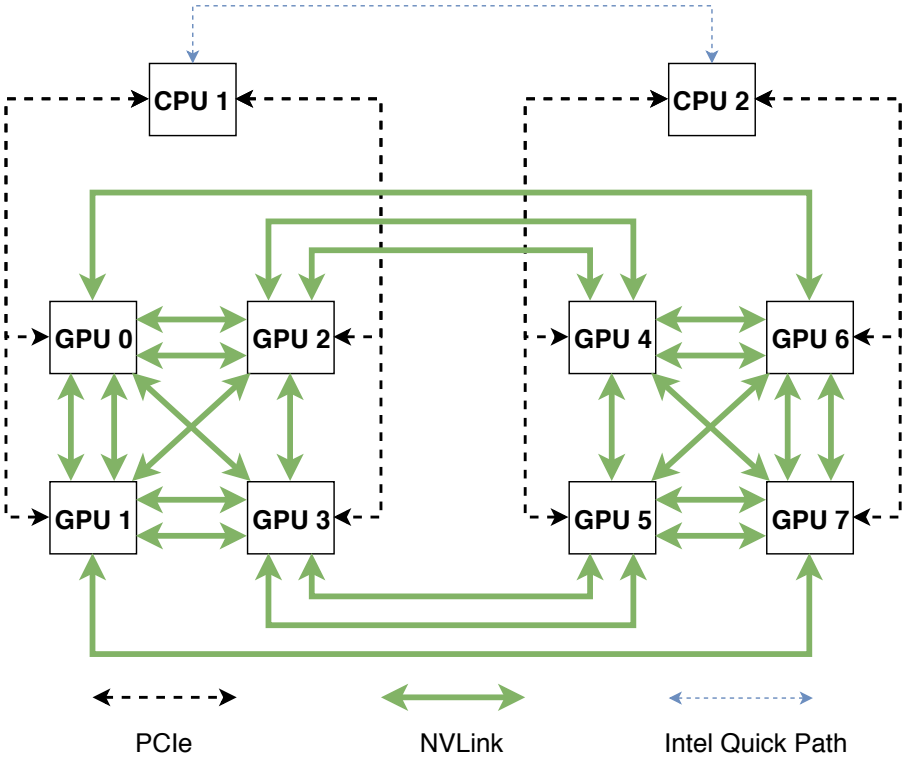


Figure 2·3: Network Topology in a DGX-1 System.

Framework and Tools

For our evaluation, we use the NVIDIA container image of MXNet, release 18.04 and CUDA 9.0.176. The DNN frameworks run on the cuDNN 7.1.1 (NVIDIA, 2018) and cuBLAS 9.0.333 (NVIDIA, 2008). The MXNet framework uses `Broadcast` and `AllReduce` communication collectives from NCCL 2.1.15. We collect the profiling data using the `nvprof` (NVIDIA, 2018b) profiler. We use the NVIDIA System Management Interface `nvidia-smi` to monitor memory usage of GPUs.

Table 2.7: Description of the networks. (Conv = Convolution, Incep = Inception, and FC = Fully Connected)

Network	Layers	Conv Layers	Incep Layers	FC Layers	Weights
LeNet	5	2	0	2	60K
AlexNet	8	5	0	3	60M
GoogLeNet	22	3	9	1	4M
Inception-v3	48	7	11	1	24M
ResNet	110	107[1]	0	1	55M

[1] Conv layers with residual input from previous layers

As discussed in Section 1.2.1, we only execute and profile the training of DNNs for FP, BP, and WU stages. As FP, BP, and WU stages are repeatedly executed, the accuracy of the network will improve. However, the time spent during each of the three stages within an epoch will remain the same.

Workloads and Datasets

For our evaluation, we use five popular neural networks used for image classification. Table 2.7 specifies the number of layers and parameters in each neural network. Layers in LeNet and AlexNet are connected serially and consist of 3×3 to 11×11 kernels in the Convolution layers for feature extraction. LeNet and AlexNet have a higher number of parameters because of their relatively larger number of fully connected layers compared to other neural networks in our evaluation.

GoogLeNet and Inception-v3 networks have both Convolution layers and Inception layers for feature extraction. Typical Inception layers consist of small parallel convolution kernels (1×1 to 5×5) followed by concatenation layer to concatenate features extracted from the parallel convolution kernels. Inception layers allow the network to use both local features (small convolution kernels) as well as highly abstracted features (larger convolution kernels). GoogLeNet and Inception-v3 require a smaller number of parameters compared to AlexNet because of the inception layers.

ResNets are very deep neural networks with residual blocks. A residual block is created

by combining the output of the current layer and the output(s) from one or more previous layers using a shortcut connection (i.e. a direct connection skipping any layers between the current layer and the previous layer(s)). Residual block allows training very deep networks without degrading the initially extracted features. It also requires a smaller number of parameters compared to other neural networks.

We consider both strong scaling and weak scaling in our evaluation. We use 256K images from Imagenet dataset to train our networks for evaluating the strong scaling. The input image dimension is $299 \times 299 \times 3$ for Inception-v3 and ResNet, and $224 \times 224 \times 3$ for the other networks. For evaluating weak scaling, we use 256K, 512K, 1024K and 2048K images for 1, 2, 4 and 8 GPUs, respectively.

2.2.2 Evaluation Results

In this section, we present our evaluation and analysis of the training of 5 DNN workloads using NVIDIA's Volta-based DGX-1 multi-GPU system. Here, we discuss the comparison of the P2P and the NCCL communication methods, quantify the NCCL overhead, show the breakdown of training time into FP+BP and WU stages, determine memory usage, and provide analysis on weak scaling for training the DNNs. We also provide in-depth insights and guidance for future endeavors for accelerating the training of DNN workloads using multi-GPU systems.

Comparison of P2P and NCCL

In this subsection, we compare the training time for the DNN workloads with the batch sizes of 16, 32 and 64 using the P2P and the NCCL communication methods and identify the factors that affect the training of the DNNs on the DGX-1 multi-GPU system.

Figure 2·4 shows the total training time for 5 different workloads using 1, 2, 4 and 8 GPUs. To analyze the results, first we start with the smallest network (LeNet) and discuss the impact of increasing the number of GPUs for a given batch size on LeNet using the P2P

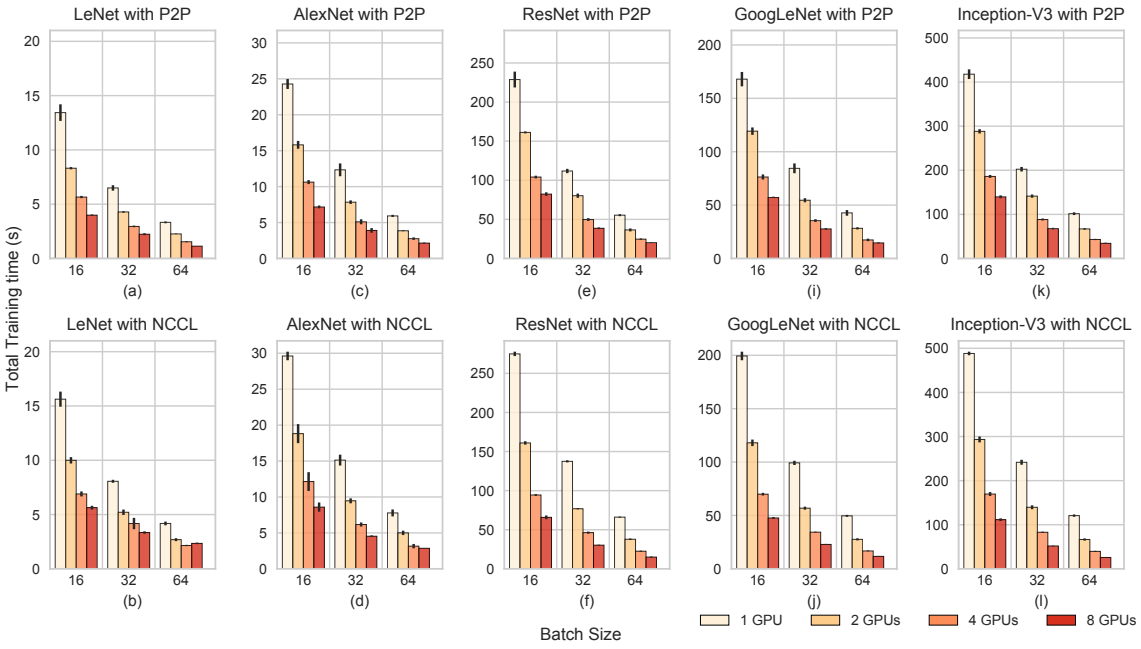


Figure 2·4: Training time per epoch for 5 different workloads on the Volta-based DGX-1 system using the P2P and the NCCL-based communication. Each bar represents the mean training time of 5 repetitions. The standard deviation is shown by the black line on top of each bar.

communication method. Then we discuss the impact of increasing batch size for training LeNet with P2P for a given GPU count. Afterwards, we discuss the effect of increasing both the batch size and the number of GPUs on the 5 workloads. Then, we compare the impact of the two communication methods for all the workloads, different batch sizes and different number of GPUs. Finally, we provide insights obtained from our analysis.

For training LeNet with a batch size of 16, as we increase the number of GPUs, the overall training time decreases for both P2P and NCCL. With P2P we can speed up the training time by factors of 1.62×, 2.37× and 3.36× for 2, 4 and 8 GPUs, respectively. On the other hand, with NCCL we achieve speedup factors of 1.56×, 2.27× and 2.77× for 2, 4 and 8 GPUs, respectively. This is understandable because LeNet is a very small network with only 2 convolution layers. GPUs do not have sufficient computation for this workload to hide the latency of communication. As a result, communication time dominates the

training time. Hence, the training time does not decrease linearly for this workload as we increase the number of GPUs. P2P outperforms NCCL for this workload due to the overhead associated with incorporating NCCL into MXNet. In Section 2.2.3, we quantify this overhead.

For all GPU counts, we observe that the time spent in training LeNet decreases almost linearly as we increase the batch size from 16 to 32 to 64. For instance, for training LeNet using 4 GPUs with P2P, as we increase the batch size from 16 to first 32 and then 64, the training time decreases by a factor of 1.92× and 3.67×, respectively. With the increased batch size, two factors affect computation. The first factor is the total number of batches that the GPU needs to process decreases for a fixed dataset. The second factor is that the number of images each GPU needs to process increases. While a decrease in the number of batches helps reduce computation time, an increase in the number of images per batch leads to increased utilization of GPU compute cores, provided that the cores are not already saturated. If the compute cores become saturated, increasing the batch size may lead to a computation bottleneck. Moreover, as the number of batches decreases with a larger batch size, the frequency of inter-GPU communication decreases. However, the amount of data that needs to be transferred for each WU from one GPU to another remains constant. This is because the number of gradients and weights is independent of the batch size, and depends solely on the DNN. Hence, increasing the batch size helps decrease the communication time.

Using the P2P communication method, the training time for all workloads under study decreases as the batch size increases. As we increase the number of GPUs from 1 to 2, for all the workloads, we observe up to a 1.8× speedup in the training time. However, we do not get the same rate of speedup when using 4 and 8 GPUs. This is because P2P memcopy becomes increasingly communication heavy as we increase the number of GPUs. For instance, consider the 4 GPU case. In the MXNet implementation using P2P, the gradients are

aggregated on GPU0. Hence, the other 3 GPUs transfer their locally computed gradients to GPU0 using a reduction operation. Then, GPU0 updates the weights using the gradients and transfers the updated weights to all the GPUs. The transfer of the gradients and the weights is parallelized using an asynchronous data transfer between GPUs. Note that the DGX-1 system provides asymmetric interconnects between different pairs of GPUs. This can cause some of the GPUs to become idle during DNN training. The BW for communication between GPU0 and GPU1, and GPU0 and GPU2, is twice the BW rate between GPU0 and GPU3 (see Figure 2·3). As a result, after updating weights, GPU3 has to wait longer than GPU1 and GPU2 to receive the updated weights. This causes GPU1 and GPU2 to remain idle until GPU3 receives the updated weights.[3]

For the training with 8 GPUs, the situation is even worse because of the lack of direct connectivity using NVLink between all the GPUs. For instance, GPU0 has direct NVLink connections with GPU1, GPU2, GPU3, and GPU6. When weights are transferred from GPU0 to GPU4, GPU5, and GPU7, direct P2P memory transfers cannot be used. Instead, weights are transferred using a device-to-host (DtoH) memory copy followed by a host-to-device (HtoD) memory copy over the slow PCIe interconnect.[4] Hence, the communication time can be significantly longer for training a DNN using 8 GPUs if the number of weights is large. MXNet tries to overcome this issue by performing multi-stage transfers through NVLink. More precisely, since GPU1 has no direct NVLink connection with GPU7, GPU0 first transfers the weights to GPU1 and then GPU1 transfers the weights to GPU7. This multi-stage transfer requires some additional time, which results in a non-linear speedup as we increase the GPU count from 2 to 4 to 8.

For training LeNet with a batch size of 16, P2P achieves a better speedup of training

[3]It is possible that GPU1 and GPU2 execute FP and BP immediately after receiving the updated weights. In that case, before synchronizing the gradients, GPU1 and GPU2 have to remain idle after executing FP and BP, until GPU3 finishes executing FP and BP.

[4]This is a limitation of the design of DGX-1. The “routers" within each GPU do not have the ability to route a packet to another node, and thus, the communication for all non-1-hop packets go through the CPU. This not a fundamental limitation, but instead, a design decision.

time than NCCL as we increase the GPU count. P2P achieves better training time than NCCL even if we increase the batch size for training LeNet. This applies for training AlexNet as well, because AlexNet has only 5 convolution layers and a large number of weights ($\sim$60M). This implies that if the number of computation-intensive layers is small, the overhead associated with incorporating the NCCL library cannot be amortized and P2P will outperform NCCL. Note that DNNs with a small number of computationally-intensive layers (i.e., Lenet, AlexNet) achieve non-linear speedup as we increase the GPU count to 4 and 8. This is because the amount of computation is not sufficient to hide the latency of communication, as well as synchronization, among the GPUs. However, for workloads with a larger number of computation-intensive layers, NCCL continuously outperforms P2P, as we increase the batch size for 4 and 8 GPUs. For a batch size of 16, training of GoogLeNet is 1.1$\times$ and 1.2$\times$ faster when using NCCL as compared to P2P for 4 and 8 GPUs, respectively. For both ResNet and Inception-v3, the training with a batch size of 16 is 1.1$\times$ and 1.25$\times$ faster using NCCL than using P2P with 4 and 8 GPUs, respectively. This is because NCCL pipelines the data transfer for updating and transferring the weights using `AllReduce` and `Broadcast` operations, respectively. This implies that the process of pipelining data transfers can amortize the NCCL overhead if there are sufficient number of data transfers [5].

Our evaluation based on training time provides the following insights:

- Increasing batch size reduces training time for an epoch linearly for all the workloads we evaluated in this work. Hence, to accelerate DNN training, hardware support is needed to facilitate training with larger batch sizes.

- Whether increasing the number of GPUs to train a particular network will lead to faster training depends on the computation-intensity of the workload and the communication method.

[5] More layers with weights mean more number of data transfers.

- With the increase of computation-intensive layers in DNN workloads, although the overall training time increases, we can reduce training time by increasing the number of GPUs.
- With NCCL, training time decreases significantly for 4 and 8 GPUs if the DNN workload has a sufficiently large number of computation-intensive layers.
- NCCL implementation has additional overhead compared to P2P implementation. NCCL should be used for training with more than 4 GPUs if the DNN network model is large, otherwise, P2P is sufficient.

2.2.3 NCCL Overhead

In the previous section, we explained that using NCCL for communication comes with additional overhead. When NCCL is used as the communication method, MXNet uses different source code (i.e. variables, functions, and kernels) compared to P2P even if only one GPU is used. MXNet uses `AllReduce` and `Broadcast` collectives available in NCCL for aggregating gradients and transferring updated weights to the GPUs, respectively. In particular, two kernels (`ReduceKernel` and `BroadcastKernel`) are used when NCCL is used as the communication method. These kernels leverage the P2P direct memory access where one GPU can directly use the data on another GPUs memory without transferring the data to its own memory. Note that the P2P communication method that we compare with NCCL uses P2P direct data transfers which is different from P2P direct memory access. Since the kernels executed by the GPUs are different for NCCL and P2P method, the CUDA runtime API overhead varies as data is accessed using different methods. For DNN training, the overhead for NCCL is larger than the overhead for P2P. However, NCCL overcomes this overhead by pipelining the data transfer as we increase the GPU count. In this section, we measure that additional NCCL overhead by comparing the training times on a single GPU for P2P and NCCL. This result explains why the use of NCCL cannot help reduce the

training time for all types of workloads.

Table 2.8 shows the NCCL overhead over P2P for training the 5 DNNs with different batch sizes on a single GPU. The percent overhead varies by a value of less than 3.6 for large networks (i.e. GoogLeNet, Inception-v3 and ResNet) with the increase of batch size while the percentage of overhead increases with batch size for smaller networks (LeNet and AlexNet). This is because as the batch size increases, the overall computation time reduces significantly for these smaller workloads and the NCCL overhead becomes more significant. This additional overhead is also present in the multi-GPU training.

Table 2.8: NCCL overhead compared to P2P for the workloads executed on a single GPU.

Network	Batch Size	NCCL Overhead (%)
LeNet	16	16.4
LeNet	32	24
LeNet	64	26.7
AlexNet	16	21.8
AlexNet	32	21.8
AlexNet	64	31.8
ResNet	16	20.1
ResNet	32	22.9
ResNet	64	19.3
GoogLeNet	16	18.7
GoogLeNet	32	17.5
GoogLeNet	64	16.2
Inception-v3	16	16.9
Inception-v3	32	19.4
Inception-v3	64	18.9

In the MXNet implementation, the P2P memcopy method requires hundreds of GBs of data copy per epoch from 3 GPUs (for training with 4 GPUs) or 7 GPUs (for training with 8 GPUs) to one of the GPUs' (GPU0) memory, whereas using NCCL, one GPU simply reads another GPU's memory and directly uses the data for computation. As NCCL reduces the communication time by leveraging the pipelining in data transfer, training a network using 2 GPUs cannot benefit much from the pipelining, rather it suffers from additional NCCL overhead. However, if a network is large, NCCL benefits significantly from the pipelining

and overcomes the NCCL overhead when training the network with 4 and 8 GPUs. Hence, with 4 and 8 GPUs, we observe a better speedup in the training time of ResNet, GoogLeNet and, Inception-v3 using NCCL compared to training time using P2P.

2.2.4 Training Time Breakdown

In this section, we show the breakdown of the total training time into computation (FP+BP) and communication (WU) time. The dataset contains a fixed set of 256K images from the Imagenet dataset for this experiment. Figure 2·5 shows the breakdown of the total training time for the 5 workloads for different batch sizes and GPU counts when using NCCL-based communication. Since our comparison between P2P and NCCL shows that NCCL has the potential to significantly decrease the training time and achieve larger speedup compared to P2P as we increase the network size and GPU count, in this subsection, we only consider NCCL-based communication. We use the `nvprof` profiler for this analysis.

During the FP stage of DNN training, the outputs (i.e. feature maps) of different layers are generated for an input batch of data (i.e. images). During the BP stage, the error at the final output layer is calculated and back-propagated to compute the gradients. Hence, FP and BP are the compute-intensive portions of DNN training. During the WU stage the gradients are aggregated and synchronized using `AllReduce` operations from the NCCL

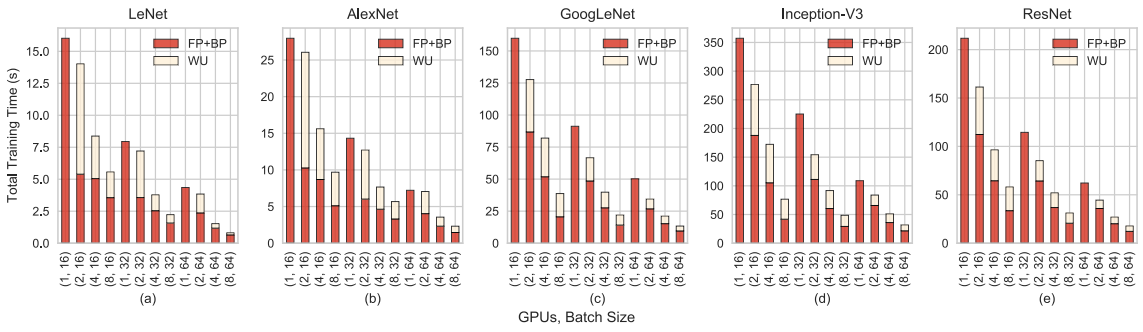


Figure 2·5: Breakdown of training time into computation (FP stage and BP stage) time and communication (WU stage) time. The X-axis represents (GPU count, Batch Size).

library and the updated weights using the aggregate gradients are broadcasted to all GPUs using `Broadcast` operations from the NCCL library. Hence, the amount of computation is negligible in the WU stage. So, we make the assumption that the time spent in the WU stage is primarily for communication.[6] Note that for the single GPU case the WU is nearly two orders of magnitude lower than the FP and BP stage (Shi and Chu, 2017) because updating weights is simple matrix addition operation (i.e. $Y = aX + B$, where a is scalar and Y and B are vectors) and does not involve any inter-GPU communication. Hence, in our evaluation, we do not report the time spent in the WU stage for single GPU training. To analyze the results, first, we discuss the impact of increasing the number of GPUs on the FP+BP and WU stages for training LeNet with a given batch size. Then, we discuss the effect of increasing the batch size on the FP+BP and WU stages for training LeNet for a given GPU count. We discuss the effect of both batch size and GPU count across all the workloads. Finally, we present the insights obtained from the breakdown of training time into FP+BP and WU stages,

For the training of LeNet with a batch size of 16, we observe more than two-fold improvement for FP+BP time as the number of GPUs increases from 1 to 2. This is because the training with 1 GPU suffers from 21.8% additional NCCL overhead that we have shown in Section 2.2.3. However, as the number of GPUs further increases, the time required for FP+BP decreases non-linearly. Our profiling results show that the `cudaStreamSynchronize` API consumes most of the time among all APIs.[7] Table 2.9 shows the percent overhead of `cudaStreamSynchronize` for training LeNet with a batch

[6] As MXNet allows overlap of BP and WU, some of the communication latency can be hidden. The WU stage takes into account the hidden latency. Hence, the actual communication time is larger than the time required for the WU stage.

[7] While training DNNs using GPUs, the training process is conducted with the help of multiple CUDA streams. Each stream is responsible for a unique set of tasks (i.e., one CUDA stream performs FP, while another performs BP). All the tasks assigned to a particular stream execute sequentially, but the different streams can be executed in parallel. The `cudaStreamSynchronize` API is used to maintain synchronization of the streams with the CPU or host thread. It holds off execution in the CPU or host thread until all the CUDA tasks assigned to the stream referenced by `cudaStreamSynchronize` finish execution. This overhead can be amortized by assigning more tasks to the stream before synchronization.

size of 16 using 1, 2, 4 and GPUs. LeNet with a compute utilization of only 18.3% fails to amortize this CUDA API overhead, and so we observe a non-linear scaling of time spent in the FP+BP stages. The time spent in the WU stage decreases almost linearly as we increase the number of GPUs from 2 to 4 to 8, for a batch size of 16.

Table 2.9: `cudaStreamSynchronize` API overhead for training LeNet with a batch size of 16, 32 and 64 using 1, 2, 4 and 8 GPUs.

Batch Size	GPU Count	Time (%)
16	1	89.2
	2	94.1
	4	86.7
	8	76.4
32	1	86.7
	2	91.9
	4	78.6
	8	68.8
64	1	81.6
	2	86.1
	4	69.8
	8	54.4

As we increase the batch size for training LeNet, both the time for FP+BP stage and time for WU decreases linearly. This is expected because doubling the batch size halves not only the number of batches each GPU processes, but also the number of times each GPU needs to communicate with other GPUs for a fixed dataset. Since LeNet is neither a computation-intensive (only 2 convolution layers) nor a communication-intensive (only $\sim$60k parameters) workload, batch sizes of 32 and 64 cannot saturate the compute cores or the NVLink BW. Nonetheless, Table 2.8 shows that as we increase the batch size, percentage of time spent for `cudaStreamSynchronize` decreases. This is because with the increased batch size, each CUDA stream performs more computations or tasks while the number of times synchronization of streams is required reduces.

As the number of computation-intensive layers in the workload increases, we observe that time spent in FP+BP stage reduces almost linearly for all GPU count (for Inception-v3, we achieve near ideal linear scaling for the batch sizes of 16 and 32). However,

the time spent in the WU stage achieves ideal linear scaling only for AlexNet which has ∼60M weights and only 8 layers. As we increase the GPU count from 2 to 4 to 8, although for other workloads we do not obtain linear scaling, we observe that from ResNet→GoogLeNet→Inception-v3, the WU stage achieves better speedup. From our observation, it is evident that workloads with more weights per layer (for layers that contain weights) show better speedup in the WU stage. This is because transferring a small amount of data is a waste of NVLink BW if the layers have a small number of weights.

Based on our evaluation using `nvprof`, in this section we provide the following insights:

- Computation time for FP+BP dominates the training time as we increase the number of GPUs for the workloads under study.
- In order to achieve close to ideal linear scaling, FP+BP stages need to utilize GPU compute cores efficiently (i.e. we can increase GPU compute utilization by increasing the amount of work in each GPU by increasing the batch size and correspondingly, reduce the number of data transfers).

2.2.5 Memory Usage Analysis

During our evaluation, we observe that the memory capacity of GPUs limits the maximum batch size that can be used to train a DNN. Hence, in this section, we perform an in-depth evaluation of memory usage by GPUs prior to the start of the training (pre-training) and during training. During the pre-training stage, the network model is transferred to the GPUs from the CPU and during the training stage additional GPU memory space is required to house the feature maps and temporary results. We varied the batch size to see the impact of batch size on both the pre-training and the training stage memory usage for 5 DNN workloads. During training, since one of the GPUs (typically GPU0) is responsible for coordinating the other GPUs, it requires additional memory.

Table 2.10 shows the memory usage for 4 GPUs during the pre-training and training

phase. Note that all the GPUs require the same amount of memory during the pre-training phase. Additionally, there is a less than 5% difference in the memory usage of P2P mem-copy and NCCL based communication methods. Hence we are only reporting memory usage for NCCL-based communication method. We observe that during the training of DNNs, all the GPUs except GPU0 consume the same memory irrespective of the number of GPUs used for training. Furthermore, for training using 2, 4 or 8 GPUs, GPU0 consumes almost the same amount of memory. Hence, the 4 GPU memory results are representative of the memory usage for training with 2, 4 and 8 GPUs.

Table 2.10: Memory usage when using the NCCL-based communication method during the pre-training stage and the training stage of DNNs when using 4 GPUs. The memory usage of all GPUs is the same for the pre-training stage. GPUz refers to the memory usage of a GPU during the pre-training, where z can take any value from 0 to 3. GPU0 refers to the memory usage of the GPU0 during training while GPUx refers to memory usage of the remaining GPUs, where x can take any value from 1 to 3.

Network	Batch Size	Pre-training GPUz (GB)	Training GPU0 (GB)	Training GPUx (GB)	Additional Mem. Usage in GPU0 w.r.t. GPUx (%)	Increase in Mem. Usage w.r.t. the Batch Size of 16 (%)
LeNet	16	1.37	2.76	1.96	41.1	–
LeNet	32	1.38	2.84	2.04	39.4	3.0
LeNet	64	1.40	2.89	2.36	22.7	4.8
AlexNet	16	1.24	2.15	1.55	39.2	–
AlexNet	32	1.25	2.36	1.76	34.5	9.9
AlexNet	64	1.27	2.97	2.37	25.6	38.2
ResNet	16	1.08	3.62	3.29	10.1	–
ResNet	32	5.98	5.66	5.63	6.2	56.1
ResNet	64	11.06	9.48	9.15	3.5	161.5
GoogLeNet	16	0.92	2.35	2.24	4.7	–
GoogLeNet	32	0.94	3.64	3.55	2.5	55.2
GoogLeNet	64	0.97	6.17	6.07	1.6	162.8
Inception-v3	16	1.04	3.89	3.60	7.9	–
Inception-v3	32	1.06	6.70	6.06	10.5	72.3
Inception-v3	64	1.09	11.01	10.78	2.4	183.3

As we increase the batch size, the memory usage increases for all workloads. While the increase in the pre-training memory usage is insignificant, the memory usage increases significantly during training. For instance, increasing the batch size from 16 to 64 increases the GPU memory consumption by a factor of 1.83$\times$ for Inception-v3. This is because with an

increased batch size, GPUs produce more feature maps or intermediate results. For all the workloads, GPU0 uses more memory than the other GPUs used in training. This is because MXNet uses GPU0's memory for gradient aggregation and weight update. As we increase batch size, the increase in the memory required for feature maps is significantly large. But the additional memory required for gradients does not increase proportionately. Hence, the percentage of additional memory usage by GPU0 decreases with increased batch size.

As the number of feature maps increases with the increase in the number of layers, the memory usage increases significantly. Note that the increase in feature maps do not necessarily depend on the number of layers, rather it depends on total nodes (neurons) in different layers. For a batch size of 64, GPU0 requires a memory usage of 2.37GB to train AlexNet while GPU0 requires 11GB of memory to train Inception-v3. Memory required for feature maps can only be reduced by making algorithm-level changes.

During our evaluation, we also tried to evaluate batch sizes larger than 64 per GPU for all the workloads. However, we could not train Inception-v3 and ResNet with a batch size larger than 64 per GPU, and we could not train GoogLeNet with a batch size larger than 128 per GPU, due to GPU memory limitations. Hence, future research should focus on both increasing memory capacity while preserving the memory BW from the hardware-level, as well as more efficient memory mapping from the software-level.

Our evaluation in this subsection provides the following insights:

- While increasing the batch size reduces the training time of DNNs for each epoch, the amount of GPU memory limits the maximum batch size that can be used for training DNN workloads.

- For larger workloads (i.e. ResNet, GoogLeNet, and Inception-v3), the memory required for intermediate outputs at different layers far exceeds the memory required for the network model.

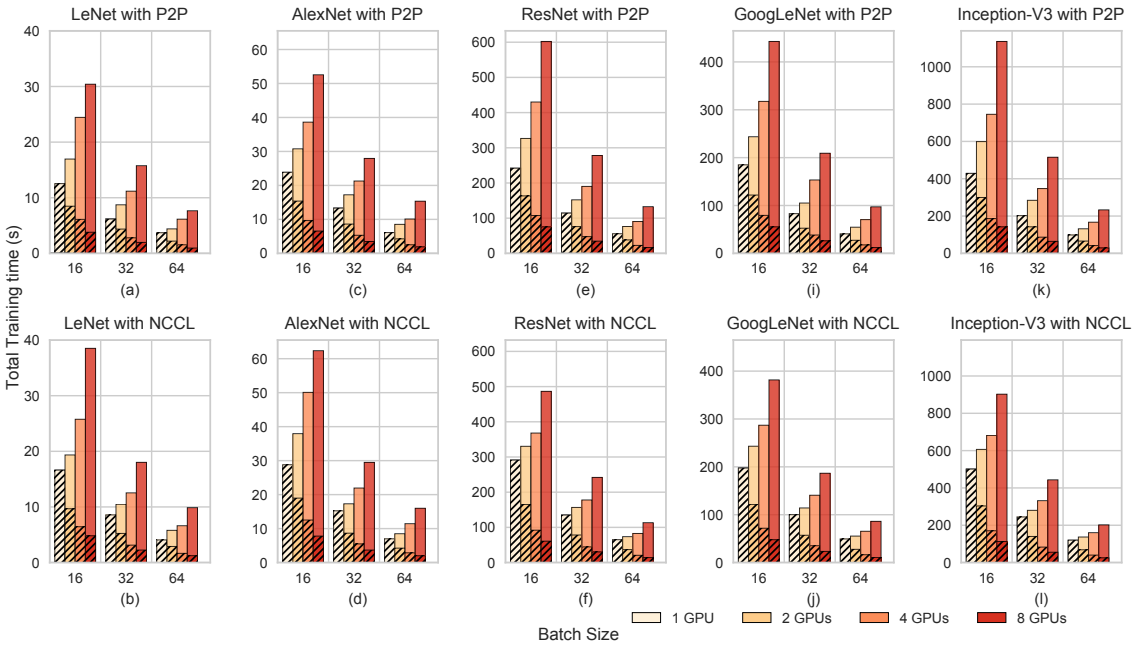


Figure 2·6: Weak scaling evaluation for the 5 workloads. The height of the 'entire bar' represents the total time per epoch for training with 256k, 512k, 1024k and, 2048k images using 1, 2, 4 and, 8 GPUs, respectively. The height of the 'hatched bar' represents the average time to train with 256k images. This facilitates the comparison between the training time for weak scaling with that for strong scaling.

2.2.6 Weak Scaling

We evaluated the weak scaling trends of the training time of the 5 DNN workloads by increasing the number of images in the dataset as we increase the number of GPUs. We use 256k, 512k, 1024k, and 2048k images for 1, 2, 4, and 8 GPUs, respectively. Figure 2·6 shows the average time for training with 256K images for both P2P and NCCL using 1, 2, 4 and 8 GPUs, as well as the total training time for evaluating weak scaling. Based on our evaluation of weak scaling we provide the following insights:

- The speedup for training LeNet with weak scaling is larger than that with strong scaling for all the batch sizes with both P2P and NCCL. As discussed in Section 2.2.4, the overhead associated with CUDA APIs affects training time of LeNet. As the

dataset size is increased for weak scaling, the overhead associated with creating and synchronizing CUDA streams gets amortized, which leads to a slightly improved training time over strong scaling training time for LeNet.

- When using weak scaling, AlexNet shows better training time for the batch sizes of 32 and 64 compared to strong scaling. With a small number of computation-intensive layers, AlexNet suffers from the overhead of CUDA APIs for creating and synchronizing streams. As we increase the number of batches, it amortizes some of the overheads. Since AlexNet has a large number of weights per layer, it utilizes the high BW of NVLink more efficiently than LeNet.

- In case of the relatively more computation-intensive workloads i.e. ResNet, GoogLeNet, and Inception-v3, when using weak scaling we achieve speedups that are less than 17% higher as compared to speedups with strong scaling using NCCL for all the batch sizes.

2.2.7 Accelerating Training of DNNs

Based on our evaluation, we make the following suggestions to accelerate DNN training:

- In Section 2.2.2 and Section 2.2.3, we have shown that NCCL does not always perform better than P2P because of additional overhead associated with NCCL. This overhead needs to be reduced. Apart from that, frameworks such as MXNet should be improved to leverage the best available communication method automatically for a given DNN workload.

- Increasing the GPU count does not improve the training time for smaller workloads as shown in Section 2.2.2. Hence, the size of the workload (i.e. number of compute-intensive layers, number of weights, etc.) should be taken into account to choose the proper GPU count.

- Our evaluation in Section 2.2.4 shows that for FP+BP stages do not achieve ideal linear speedup as we increase the batch size for a number of computation-intensive workloads. GPUs with more tensor cores and compute cores can help accelerate the FP+BP stage.

- Our memory analysis in Section 2.2.5 showed that GPU memory capacity can be a severe bottleneck for training DNNs as larger networks need to be trained with larger batch sizes to reduce training time. The maximum batch size that can be used to train a network is limited by GPU memory capacity for data parallel implementation of training. Hence, significant improvement in the memory technology is required to increase GPU memory capacity.[8]

- Our evaluations show that inefficiency in the implementation of high-level frameworks such as MXNet, may lead to under-utilization of GPU resources. For instance, additional memory consumption of GPU0 compared to other GPUs causes under-utilization of available GPU memory. This can be solved by a more efficient distribution of data.

- CUDA API overheads for maintaining synchronization consume a significant amount of training time. Faster synchronization mechanism needs to be developed to utilize the GPU resources more efficiently.

2.2.8 Summary

High performance MGPU systems are widely used to accelerate training of DNNs by exploiting the inherently massive parallel nature of the training process. Typically, the training of DNNs in MGPU systems leverages a data-parallel model in which a DNN is replicated on every GPU, and each GPU performs Forward Propagation (FP), Backward

[8]The memory required for training a DNN using GPUs can depend on how a framework is implemented. Hence, different frameworks may need a different amount of memory for training the same DNN. But the memory required for the output at each layer must be the same for all frameworks, for a given DNN.

Propagation (BP) and, Weight Update (WU). We analyze the WU stage that is composed of collective communication (e.g., allReduce, broadcast), which demands very efficient communication among the GPUs to avoid diminishing returns when scaling the number of GPUs in the system. To overcome this issue, different data transfer mechanisms and libraries have been introduced by NVIDIA, and adopted by high-level frameworks to train DNNs. In this work, we evaluate and compare the performance of peer-to-peer (P2P) data transfer method and NCCL library-based communication method for training DNNs on a DGX-1 system consisting of 8 NVIDIA Volta-based GPUs. We profile and analyze the training of five popular DNNs (GoogLeNet, AlexNet, Inception-v3, ResNet and LeNet) using 1, 2, 4 and 8 GPUs. We show the breakdown of the training time across the FP+BP stage and the WU stage to provide insights about the limiting factors of the training algorithm as well as to identify the bottlenecks in the multi-GPU system architecture. Our detailed profiling and analysis can help programmers and DNN model designers accelerate the training process in DNNs.

2.3 Evaluation of MGPU Systems Using Synthetic Workloads

Our evaluation in previous sections shows that communication overhead has a huge impact on the scalability of MGPU workloads. Hence, it is imperative to explore solutions that address communication bottlenecks in MGPU systems. In this section, we describe our synthetic workloads which we use to evaluate MGPU system performance and address communication bottleneck in the MGPU systems. We consider different data transfer mechanisms to develop our synthetic workloads. We describe our TSM solution and present an evaluation of the true shared memory (TSM) solution using three MGPU systems under study.

2.3.1 Synthetic Workloads

We develop synthetic workloads that precisely mimic the communication pattern of MNIST, Cifar10 and ImageNet datasets with the MLP network, based on the study of the real-world workloads. We use these synthetic workloads as our baseline (using P2P memcpy for gradient synchronization) to evaluate the communication mechanisms (zero-copy and unified memory) supported by CUDA. Then we propose our TSM solution and evaluate it using synthetic workloads.

We do not try to mimic the exact computations performed by the GPUs, but we are able to reproduce the same volume of data transfers present in the DNN workloads in our synthetic workloads. We use the Cifar10 dataset and the 3-layer MLP network as an example to show how we calculate the weights in our synthetic workload. The size of each image in the Cifar10 dataset is 32×32, which can be represented by a matrix of size 1024×1. The first fully-connected layer in the MLP network has 128 neurons; the output of the first layer can be represented by a matrix of size 1×128. The weights for the first layer can be represented by a matrix of size 1024×128. Similarly, for fully-connected layers 2 and 3, with 64 and 1000 neurons, the size of weight matrices are 128×64 and 64×1000, respectively. The size of the weights is 794 KB. Similarly, the size of the weights for MNIST and ImageNet dataset with the MLP network are 482 KB and 11245.6 KB, respectively. We use the cuBLAS library to perform matrix multiplications when calculating the output of each layer and aggregating the gradients.

We design our synthetic workloads, running them on 2 GPUs. Figure 2·7 shows how we can distribute memory space for gradients and input data. Figure 2·7a shows the baseline P2P memcpy mechanism used by MXNet to update the gradients. A unique set of input data is transferred from the CPU to each GPU. The weight matrix is initialized to random values in the CPU, and the same weights are transferred from the CPU to the both GPUs. Each GPU performs matrix multiplications to produce the output in each layer; this mimics the

forward propagation phase for training of the DNN workloads. We transfer the gradients (represented by the weight matrices for our synthetic workload) from GPU1 to GPU0 to update weights in GPU0. We use P2P memcpy to transfer the gradients. Once the weights are updated, we transfer the updated weights from GPU0 to GPU1 using P2P memcpy.

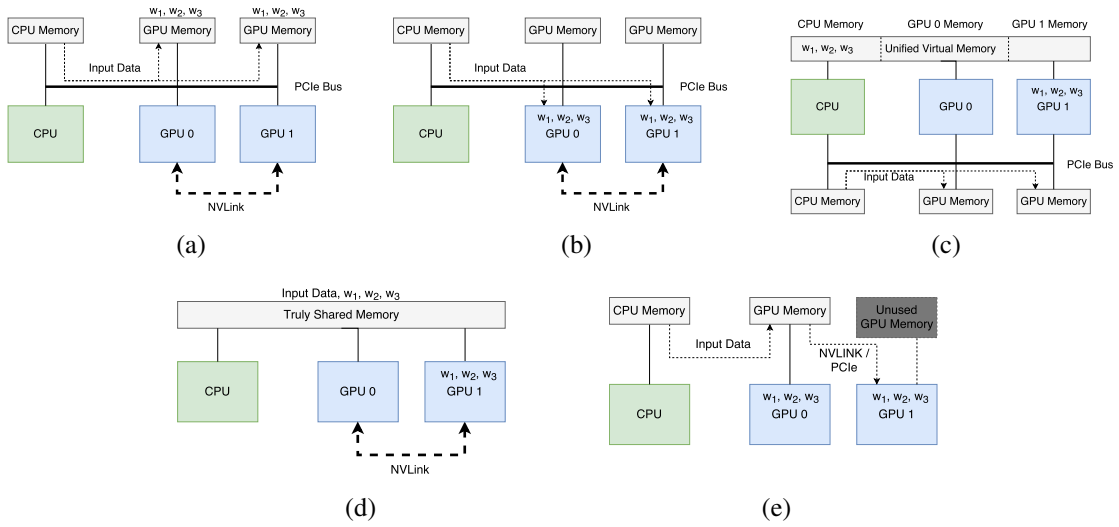


Figure 2·7: Distribution of input data and weights for synthetic workload representing (a) baseline P2P memcpy, (b) zerocopy, (c) unified virtual memory for gradient synchronization, (d) our proposed true shared memory model and, (e) implementation of our true shared memory model to improve performance of DL workloads

In Figure 2·7b, we use the zerocopy mechanism, supported in NVIDIA CUDA, to reduce the number of P2P memcpy. Zerocopy is a data access mechanism where a GPU can access the data from CPU's memory without copying to its own memory. To ensure data coherence, the data is pinned in GPU memory. This means if a GPU requires the data, it has to wait until another GPU that is using the data finishes using it. If the data is too large, our evaluation finds that serialization latency due to pinned memory can lead to severe performance degradation. In our synthetic workload using zerocopy, we use zerocopy for the weights since they have to be same for both GPUs. We can also avoid memcpy for gradient synchronization. The input data is transferred to the GPUs using memcpy. Hence, in

this workload, the weights are always in CPU memory. Although serialization latency can degrade performance, this method can be used for training large batch sizes. We need to fit both input data and network model data in GPU memory. However, we are restricted by the size of GPU memory, and may not be able to train the network with large batch sizes. But the zerocopy method provides a way to use CPU memory to hold the network model data and allows training with larger batches.

We use unified memory (UM) supported by CUDA 6.0 or higher in our synthetic workload, as shown in Figure 2·7c. Unified memory provides a virtualization based mechanism to support a single, shared, memory space that can be accessed using a single pointer. The user does not need the exact physical location of the data since the CUDA runtime will figure out how to transfer the data to the device. This transfer is transparent to the user. As a result, this mechanism makes the programming easier, but does not necessarily reduce the volume of data transferred across multiple devices. In our synthetic workloads, we keep the weight matrices in the unified memory. The input data is transferred to the GPUs using memcpy.

Finally, Figure 2·7d shows our proposed solution– true shared memory (TSM)– to efficiently use the memory space and improve performance. We propose using a true shared memory space for both CPU and GPUs to avoid the large number of data copies back and forth between the computing devices. To create a unified memory space synthetically, we first copy the input datasets and weights in one of the GPU's (GPU0) memories, as shown in Figure 2·7e. We enable peer-to-peer direct memory access so that GPU1 can directly use the memory of GPU0. Then each GPU works on a different set of data, but use the same weights in the GPU0's memory. They save the gradients in GPU0's memory. In this way, we have both the gradients and the weights in GPU0's memory and we can update the weights using the gradients without performing any additional data copies, as in the case of P2P memcpy.

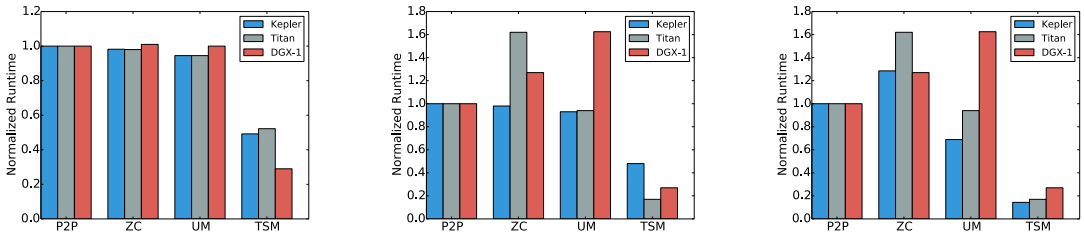


Figure 2·8: Performance comparison of different data transfer mechanisms among the three MGPU systems (2 GPUs of each system) using synthetic workloads that mimic (a) MNIST dataset (b) Cifar10 dataset (c) ImageNet dataset with MLP network.

2.3.2 Evaluation Results Using Synthetic Workloads

Figure 2·8 shows the normalized results for Kepler, Titan and DGX-1 systems. We mimic the communication patterns for training a batch size of 32 per GPU for all the synthetic workloads. The runtime is normalized with respect to the runtime of the corresponding baseline synthetic P2P workload in each of the three systems. We run our workloads on three different servers, where each server has its own customized network for CPUs and GPUs. Based on the complexity of the network for the CPUs and GPUs, the overhead for different CUDA API calls can vary, and so comparing absolute runtime values can be misleading.

Figure 2·8a and 2·8b show that zerocopy (ZC) and unified memory (UM) do not improve performance in any of the three systems studied, as compared to the baseline P2P memcpy method, when evaluting synthetic workloads. As discussed earlier, ZC causes serialization when executing the kernels, given that there is a race between two GPUs that try to get control of the data (weights). UM performs slightly worse than the baseline P2P memcpy method because UM does not decrease the number of data copies as compared to P2P memcpy. It also has additional CUDA runtime overhead (Landaverde et al., 2014).

Finally, our prototype solution, TSM, performs 2.07×, 1.95× and 3.33× better in terms

of runtimes compared to the respective baselines on the Kepler, Titan and DGX-1 system respectively, for workloads mimicking MNIST dataset. The Kepler system shows slightly better speedup versus the Titan system, because for the P2P memcpy method, the Kepler system uses PCIe 2.0 bandwidth as opposed to PCIe 3.0 bandwidth (2× bandwidth over PCIe 2.0 in the Titan system). Since we avoid a large number of P2P memcpy in TSM, the normalized runtime improves more in Kepler system compared to Titan system.

Based on our results for workloads mimicking the ImageNet dataset run with a MLP network, as shown in Figure 2·8c, we can make the following observations. First, ZC is a huge bottleneck on the GPUs, as a large number of weights for the ImageNet dataset on the MLP network is pinned to one GPU (e.g., GPU0) and the other GPU (GPU1) cannot access it for the entire time that GPU0 is using the data. Second, UM also performs worse than the baseline because of the overhead associated with large amounts of user-transparent data migration. In a DGX-1 system, UM performance is even worse compared to the Kepler and Titan systems. This is because of the GPU page fault support on the Pascal GPUs. Previously, data had to be transferred to the devices before executing a kernel. Equipped with the new page fault support, if the data is not present in the GPU memory, the GPU page fault occurs. Hence, there is the overhead for serving GPU page faults and transferring the data to the GPUs. Third, the Kepler system achieves better performance for the TSM solution, as compared to the baseline. This is not surprising because for the baseline P2P method, the Kepler system uses the lowest bandwidth interconnect among the three systems. Hence, the Kepler system requires the most time for the P2P method. But in the case of the DGX-1 system, its baseline P2P method benefits from the higher NVLink bandwidth. Nonetheless, for the ImageNet dataset our TSM solution achieves 3.7× better performance than our baseline. Speedups of the 3 datasets, as run on our TSM solution, achieves a 3.7×, 3.2×, 3.5× speedup versus the corresponding baseline P2P method in Kepler, Titan and DGX MGPU systems, respectively.

We consider a MLP network in our synthetic workload. However, we should mention that the data transfer mechanism is similar in CNN and RNN networks in MXNet and other frameworks. Based on the amount of computation and communication, the size of the network and the data set, and the number of weights in different layers of network, different DL workloads may scale differently. For computationally-intensive workloads such as CNN, the computation time increases quickly, based on the amount of communication. Hence, CNN workloads show better scaling in MGPU systems. Nonetheless, our TSM method will still improve overall performance.

Although we have used 2 GPUs in our evaluation, our proposed TSM solution will improve the performance for MGPU systems with more than 2 GPUs. As the number of GPUs is increased, the number data transfers increase. For the P2P memcpy method using 4 GPUs, for example, the CPU has to transfer the input data to 4 GPUs. After the calculation of gradients by all 4 GPUs, to upgrade the weights, the gradients have to be transferred to one of the GPUs and the upgraded weights need to be transferred to other GPUs. These transfers involve a number of CPU-to-GPU and P2P memcpy. As our TSM solution improves performance by reducing the number of data transfers, MGPU systems with more than 2 GPUs will benefit from TSM significantly.

The significance of our proposed solution is two-fold. First, it encourages computer architects to develop new systems with true shared memory for both CPU and GPUs. When the network model does not fit in the GPU memory, it degrades the GPU performance for training (Dean et al., 2012). Our TSM solution introduces a unified memory space for both CPU and GPUs, and removes the limitations that are imposed by the size of the GPU's native memory. Second, the developers of the DL frameworks or workloads can gain insight into the communication mechanisms best suited for training DL workloads. Users can choose the type of data transfer that is most suitable for their workload, when using the existing MGPU systems.

Chapter 3

True Shared Memory for MGPU System

In this chapter, we present the overall design of our proposed MGPU-TSM system architecture to overcome the limitations of the MGPU systems that we have listed in Chapter 2. While the concept of shared main memory has been used in APU designs (Hechtman and Sorin, 2013), where a CPU and an integrated GPU share the same main memory, we extend this shared memory to CPUs with multiple discrete GPUs. Unified memory (NVIDIA, 2018) provides an abstraction that the main memory is unified, but the underlying main memory is still partitioned and each partition is only accessible by the CPU or GPU. In comparison, MGPU-TSM provides an architecture where the CPU and GPUs *truly* share the same physical main memory. We present our evaluation of MGPU-TSM with respect to the existing MGPU system design in this chapter. We also perform a preliminary evaluation of the thermal feasibility of an MGPU-TSM system.

3.1 MGPU-TSM Architecture

To create a low-latency UMA for multiple GPUs, we re-architect both the GPU memory hierarchy organization and the CPU memory. To explain our envisioned MGPU-TSM architecture, we consider an MGPU-TSM system consisting of 4 GPUs, 1 CPU and a total of 32GB of main memory[1], shared by the CPU and the GPUs. Figure 3·1b shows a logical implementation of the idea with respect to the existing MGPU systems in Figure 3·1a. In

[1] We are using a 32GB main memory just to explain the MGPU-TSM architecture and make a fair comparison with the MGPU system with RDMA, where the GPUs have the same compute and memory resources. Our MGPU-TSM system also works with a much larger main memory.

the existing MGPU systems, each GPU has its own local main memory and a GPU can access a remote GPU's main memory using a PCIe or NVLink connected with a switch. However, MGPU-TSM provides all the CPU and GPUs in the system access to the entire main memory of the system by reorganizing the memory modules and connectivity to the memory modules.

Figure 3·2 shows the overall layout of our MGPU-TSM architecture. We provided a dedicated write-through L1$ for each CU. All the L1$s are connected to the L2$s via a crossbar network. For our proposed MGPU-TSM system, we make changes to the memory hierarchy, starting from L2$, and including the main memory.

As mentioned earlier, GPUs typically have distributed L2$ banks, where each L2$ bank is connected to one memory controller. Each memory controller controls a region of main memory. However, this memory controller configuration is not designed appropriately for an MGPU-TSM system. This is because all the GPUs in the MGPU-TSM system have direct access to the entire main memory, so 4 memory controllers (one from each one of the 4 GPUs) can simultaneously send requests to the same memory location, resulting in load-after-store and store-after-load issues. What is lacking is the necessary control logic to correctly order the memory requests coming from the 4 discrete memory controllers, enforcing an ordering before sending the requests to the DRAM. To prevent this from happening, we need to re-architect the memory controllers.

We move the memory controllers from the GPU side to the memory side and group

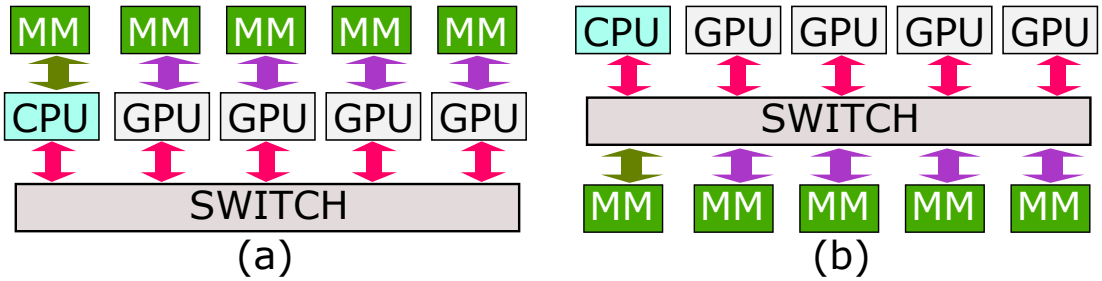


Figure 3·1: (a) Conventional MGPU system vs. (b) MGPU-TSM.

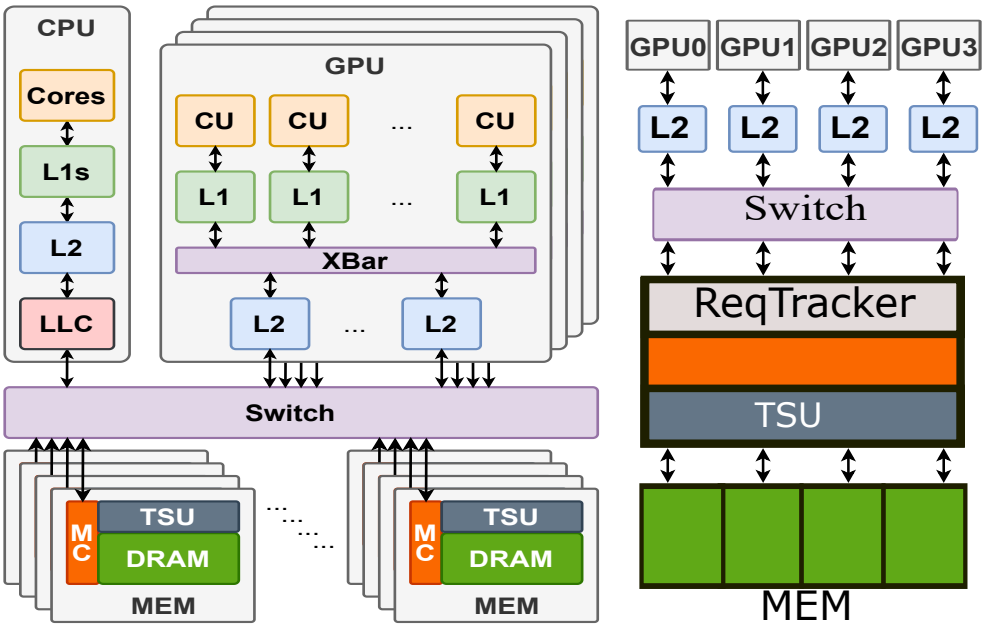


Figure 3·2: A high-level representation of our MGPU-TSM architecture (left). The description of the TSU is provided in Section 4.2.6.

MCs from different GPUs that target the same main memory region into a unified MC. Thus, we end up with 8 unified memory controllers from 32 normal memory controllers for the entire system, where each MC manages 1/8th of main memory and connects to the corresponding L2$ bank on the GPU side.[2] In each unified memory controller, we implement request tracker logic (see ReqTracker in Figure 3·2) that keeps track of in-flight memory transactions. We implement MSHR-style logic in the ReqTrack design to track the requests being serviced. This tracking ensures that load-after-store and store-after-load issues across multiple GPUs are properly addressed for a given cache line. At any given time, a memory controller can serve four different requests, which is the case if we have 4 separate memory controllers. The ReqTracker keeps track of the in-flight transactions using a hash table. If a new request arrives for a cache line that is in-flight for another request, the ReqTracker does not allow it to be scheduled and thus avoids potential load-after-store

[2]An MGPU-TSM system architecture with unified memory controllers can be implemented using 2.5D integration or PCB-like integration, where the unified memory controllers are on a separate chiplet or chip, respectively.

or store-after-load problems. The exact implementation of ReqTracker varies based on the underlying memory consistency model which we discuss later in Section 4.2.5.

Each L2$ bank is connected to a switch (SW), with a dedicated bidirectional link. Each memory controller is also connected to the switch by a bidirectional link. Thus, each memory access requires a two-hop communication in each direction, from the L2$ to the switch and from the switch to memory controller for memory requests, and from the memory controller to the switch and then from switch to L2$ for memory responses. The timestamp storage unit (TSU) is also part of the memory controller. The TSU is an essential component to maintain coherence in the MGPU system. The TSU's operation will be described in detail in Section 4.2.6.

The key advantage of our TSM lies in physically-unified main memory, which provides uniform memory access (UMA) across the system. This physically-unified design completely eliminates remote accesses via RDMA, as well as CPU-to-GPU data transfers.

3.2 Evaluation Methodology

In this section, we describe the RDMA-based (`MGPU-RDMA`) and TSM-based (`MGPU-TSM`) MGPU system configurations, the simulator used, and application benchmarks selected to compare the two MGPU configurations.

3.2.1 MGPU System Configurations

Table 3.1 shows the architecture of each GPU in both `MGPU-RDMA` and `MGPU-TSM` configurations. Both configurations leverage a relaxed memory consistency model, which allows re-ordering of memory requests as long as there are no read-write dependencies between the requests. There is no support for coherence at the hardware level and the programmer needs to be aware of potential coherence issues and write programs accordingly. Both configurations use a write-back cache replacement policy for L2$. To maintain a fair com-

Table 3.1: GPU Architecture.

Component per GPU	Configuration	Count	Component per GPU	Configuration	Count
CU	1.0 GHz	64	L1 Vector $	16KB 4-way	64
L1 Scalar $	16KB 4-way	8	L1I$	32KB 4-way	8
L2$	256KB 16-way	8	DRAM	512MB HBM	8
L1 TLB	1 set, 32-way	48	L2 TLB	32 sets, 16-way	1

parison, we limit the total L2-to-main memory bidirectional bandwidth for `MGPU-TSM` to 32 GB/sec, as the RDMA transactions are limited by the 32GB/sec BW of PCIe in `MGPU-RDMA`. Hence, in our MGPU-TSM evaluation, the total L2-to-main memory bandwidth for 4 GPUs is limited to 128 GB/sec. By doing so, we highlight the true benefit of `MGPU-TSM` versus a `MGPU-RDMA` implementation. We later evaluate the impact of the availability of higher bandwidth interconnects, in terms of performance, for our `MGPU-TSM` system in Section 4.4.1.

3.2.2 Simulation Platform

We used the MGPUSim (Sun et al., 2019) simulator for our evaluation. MGPUSim has been validated against real AMD MGPU systems (Macri, 2015; Sun et al., 2019). MGPUSim supports the `MGPU-RDMA` configuration. The MGPUSim simulator faithfully models the PCIe 4.0 interconnects used for the `MGPU-RDMA` configuration. We extended the simulator and its memory hierarchy to support `MGPU-TSM`.

3.2.3 Standard Application Benchmarks

In terms of workloads, we use a mix of memory-bound and compute-bound benchmarks. Our suite includes 11 workloads in total (see Table 3.2), selected from the Hetero-Mark (Sun et al., 2016), PolyBench (Pouchet, 2012), SHOC (Danalis et al., 2010), and DNNMark (Dong and Kaeli, 2017) benchmark suites. We use this suite to compare the performance of `MGPU-RDMA` and `MGPU-TSM`. These 9 benchmarks have a memory footprint of 64 MB or higher, which is more than 8$\times$ the capacity of all the L2$'s of the 4 GPUs combined (8MB). In addition, these benchmarks present a diverse range of sharing pat-

Table 3.2: Application benchmark suite used for evaluation. Memory represents the footprint of the GPU memory required by a benchmark.

Benchmark (abbr.)	Suite	Type	Memory
Advanced Encryption Standard (**aes**)	Hetero-Mark	Compute	71 MB
Matrix Transpose and Vector Multiplication (**atax**)	PolyBench	Compute	64 MB
Breadth First Search (**bfs**)	SHOC	Memory	574 MB
BiCGStab Linear Solver (**bicg**)	PolyBench	Compute	64 MB
Finite Impulse Response (**fir**)	Hetero-Mark	Memory	67 MB
Matrix Multiplication (**mm**)	AMDAPPSDK	Memory	192 MB
Matrix Transpose (**mt**)	AMDAPPSDK	Memory	128 MB
Rectified Linear Unit (**relu**)	DNNMark	Memory	67 MB
Simple Convolution (**conv**)	AMDAPPSDK	Memory	145 MB

terns across the GPUs. This helps stress the memory hierarchy and thoroughly compare `MGPU-RDMA` and `MGPU-TSM`.

3.3 Evaluation Results

In this section, we compare the performance of the `MGPU-RDMA` and `MGPU-TSM` system. As we can see from Figure 3·3, `MGPU-TSM` achieves 3.81× better performance on average versus `MGPU-RDMA`. The `MGPU-TSM` configuration benefits from two factors:

1. The physically unified shared memory of `MGPU-TSM` eliminates the need for the CPU-to-GPU and GPU-to-CPU data transfers.

2. TSM is organized such that it eliminates remote data accesses, so avoiding costly RDMA transfers.

To better characterize the performance differences between `MGPU-TSM` and `MGPU-RDMA` when running individual benchmarks, for each workload we determine the number of RDMA transactions that access remote L2$, and the number of DRAM transactions that access the local DRAM. Note that some of the RDMA transactions cause the remote L2$ to send the request to the remote GPU's main memory, if there are L2$ misses. Table 3.3

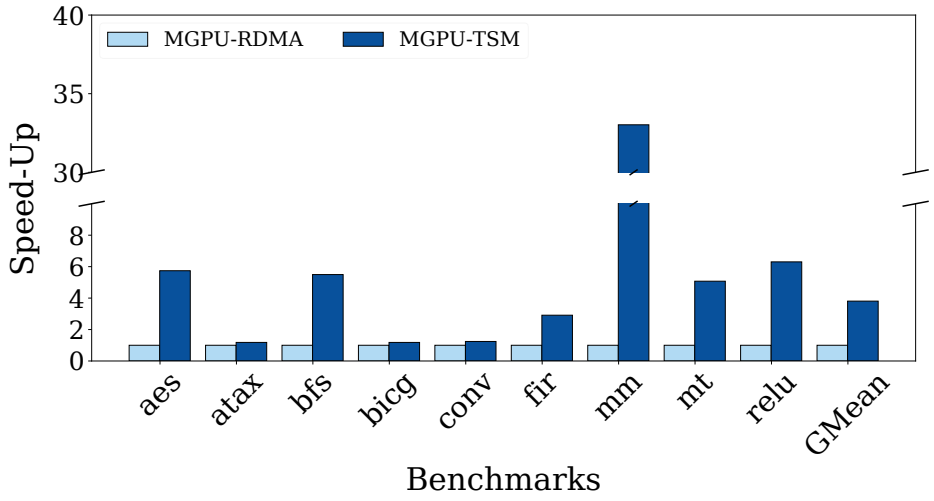


Figure 3·3: Speedup of our MGPU-TSM system w.r.t. a MGPU-RDMA system.

shows the number of RDMA and DRAM transactions per 100M instructions for each workload. First, every benchmark requires that the GPUs use remote accesses to collaboratively execute the applications. Second, the memory-bound benchmarks, which are **mm**, **bfs**, **relu**, **mt**, and **fir**, clearly benefit the most from MGPU-TSM. Among these memory-bound benchmarks, **mm** achieves the most performance benefit with MGPU-TSM because **mm** has larger number of remote accesses than the number of main memory accesses with MGPU-RDMA. Third, the compute-bound benchmarks, such as **atax** and **bicg**, benefit only moderately from MGPU-TSM. These benchmarks have sufficient computations to hide most the memory access latency. Nonetheless, all the benchmarks under study see some benefits from MGPU-TSM, as compared to MGPU-RDMA. Finally, the DRAM transaction count to the RDMA transaction count (D/R ratio) indicates the relative number of DRAM accesses for each RDMA access for the benchmarks. The **mm** benchmark generates 1.5× more RDMA accesses than DRAM accesses, which implies that the **mm** benchmark suffers when using RDMA, experiencing a high direct access penalty, even though a high percentage of RDMA accesses result in cache hits in a remote L2$. Hence, we have fewer DRAM transactions than RDMA transactions for **mm**.

Table 3.3: RDMA and DRAM transaction counts for `MGPU-RDMA`, per 100M instructions.

Benchmark	#RDMA	#DRAM	D/R Ratio
aes	494,16	1,903,438	3.85
atax	15,142,282	18,938,385	1.25
bfs	38,416	9,468,865	246.5
bicg	15,142,276	18,939,477	1.25
conv	3,652,412	17,915,660	4.91
fir	7,402,806	10,169,010	1.37
mm	4,961,351	3,301,306	0.67
mt	1,573,680	4,194,339	2.66
relu	1,652.258	4,405,017	2.67

Table 3.4: MGPU-TSM components obtained from publicly available product specifications.

Component	**Name**	**Tech. Node** (nm)	**Area** (mm^2)	**Power** (W)
GPU	RX 5700	7	151	180
CPU	Ryzen 9 3950X	7	144*	105
Memory	HBM 2.0	14	92	21.4*

* Determined through technology scaling rules.

3.4 Thermal Feasibility of MGPU-TSM

In this section, we provide an initial evaluation of thermal behavior of an MGPU-TSM system to understand its thermal feasibility.

For this preliminary evaluation, we use an MGPU-TSM system consisting of 4 GPUs, 1 CPU and 4 HBM stacks, for a total of 32 GB memory. The specifications of the GPU, CPU and HBM stacks are provided in Table 3.4. We used components that were recently released on the market. We can leverage 2.5D integration technology to build this MGPU-TSM system. The maximum manufacturable interposer size is 50mm $\times$ 50mm (Cochet et al., 2014). As technology scales, we expect to see lower area and power for the future CPU, GPU and memory with similar performance; hence, an MGPU-TSM system will be easier to build.

To understand if our proposed MGPU-TSM system is viable, we evaluated the physi-

cal design and thermal behavior of the system. The detailed thermal-aware placement and routing (Coskun et al., 2018) of different components are beyond the scope of this thesis. However, we intuitively placed the components on the silicon interposer to evaluate the thermal behavior at the maximum power scenario using Hotspot (Huang et al., 2006). Figure 3·4 shows the thermal map resulting from our intuitive placement. The center of the interposer is the hottest part of the system. We, therefore, place the CPU between the GPUs and spread out the HBMs in the remaining open space. We assumed a square-shaped chip for the CPU and GPU.

Figure 3·4 shows that at the maximum power consumption, our intuitive placement results in a maximum temperature of 88.6°C. Typical systems can tolerate temperature up to 100°C. Hence, even our intuitive placement results in a viable system. Note that in our analysis we do not consider any sophisticated cooling method such as liquid cooling which is becoming common for high power density systems (Fan et al., 2018), (Gullbrand et al., 2019), (Khalaj and Halgamuge, 2017). With the application of efficient liquid cooling mechanism, we expect the temperature to be lowered. Hence, we can pack more GPUs or HBMs onto the interposer and increase the throughput density (Flop/mm^2) of the system.

3.5 Summary

The sizes of GPU applications are rapidly growing. They are exhausting the compute and memory resources of a single GPU, and are demanding the move to multiple GPUs. However, the performance of these DNN workloads scales sub-linearly with GPU count because of the overhead of data movement across multiple GPUs. Moreover, a lack of hardware support for coherence exacerbates the problem because a programmer must either replicate the data across GPUs or fetch the remote data using high-overhead off-chip links. To address these problems, we propose an MGPU system with true shared memory (MGPU-TSM), where the main memory is physically shared across all the GPUs. We

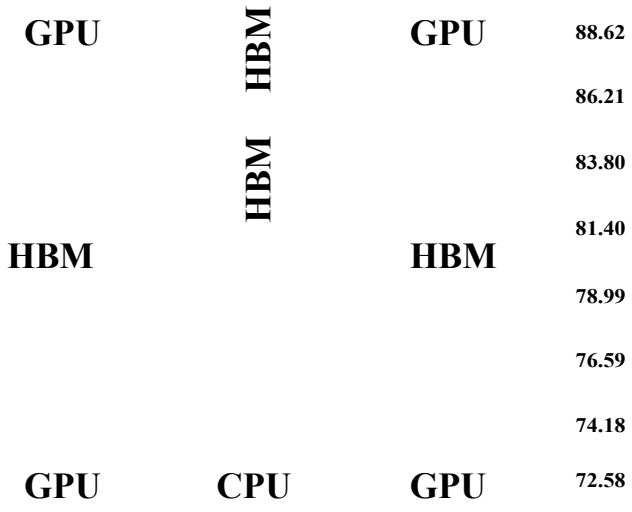


Figure 3·4: Thermal map for an MGPU-TSM system with 4GPUs, 1 CPU and 4 HBM stacks on an interposer (50mm $\times$ 50mm) using 2.5D integration technology.

eliminate remote accesses and avoid data replication using an MGPU-TSM system, which simplifies the memory hierarchy. Our analysis shows that MGPU-TSM with 4 GPUs performs, on average, 3.8$\times$ better than the current best performing MGPU configuration for standard application benchmarks.

Chapter 4

Coherence in MGPU-TSM

In the previous chapter, we demonstrated that our proposed MGPU-TSM system can achieve 3.81× average speed-up as compared to a contemporary MGPU system. To ease the programmability and data sharing within and across multiple GPUs, we add efficient and scalable intra-GPU and inter-GPU hardware coherence support for MGPU-TSM by means of our new MGCC protocol. To present how it works, we assume an example MGPU-TSM system with 4 GPUs. Our MGCC protocol is based on the G-TSC protocol (Tabbakh et al., 2018). To better understand MGCC, we first briefly explain the operation of G-TSC protocol (Tabbakh et al., 2018), which has been proposed to maintain coherence in a single GPU system. Afterwards, we explain our MGCC protocol in detail and then, we present a thorough evaluation of an MGPU-TSM system with MGCC protocol.

4.1 Timestamp-Based Coherence in a Single GPU System

G-TSC protocol assigns a logical timestamp (`warpts`) to each CU in the GPU. Table 4.1 provides definitions for the terminology used to describe the G-TSC and also our proposed MGCC protocol.

Read Operation: A read request from a CU contains the `warpts` and the address. Each block in the L1\$[1] has a read timestamp (`rts`) and write timestamp (`wts`). If the block is

[1]Throughout this paper, we use L1\$ to refer to L1\$ vector cache unless specified otherwise.

Table 4.1: Terminologies and definitions

Term	Definition
physical time	The wall clock time of an operation.
logical time	The logical counter maintained by a component (e.g., CU and cache).
warpts	The current logical time of a CU. In the G-TSC protocol, the memory operations are ordered based on the warpts.
cts	The current logical time of a cache. Each cache has a cts that is updated based on the last memory operation.
block	An entry containing address, data, and associated timestamps in the caches.
wts	The write timestamp of a cache block. It represents the logical time when a write operation is visible to the processors.
rts	The read timestamp of a cache block. It represents the logical time until which reading the cache block is valid.
lease	The difference between rts and wts. The data in the cache block is valid only if cts or warpts is within the lease.
RdLease	Lease assigned to a block after a read operation is executed.
WrLease	Lease assigned to a block after a write operation is executed.
memts	The memory time stamp that represents the logical read timestamp that the memory assigns to a cache block.

present in the L1$ and the warpts falls within the range between wts and rts (i.e. the lease), the read request is considered a cache hit. Otherwise, the read request is treated as an L1$ miss and is forwarded to the L2$. This read request to L2$ contains the address, the wts of the block and the warpts. If the wts value for the request is set to 0, it means a compulsory miss occurred at L1$, and L2$ must respond with the data and the timestamps. A non-zero value for wts implies the block exists in L1$, but the timestamp expired.

Upon receiving a read request, the L2$ checks if it has the requested cache block. If the block does not exist in the L2$, then the L2$ sends a read request to the MM. If the warpts of the request from L1$ is within the lease for that block in the L2$, the L2$ also compares the wts from the L1$ request and the wts of the block in the L2$. If both wts values are the same, it means that the data was not modified by another CU and simply that the lease expired in the L1$. In that case, the L2$ extends the lease by increasing the rts and sends the new rts and wts values to the L1$. If the wts values do not match, it implies that the data was modified by a different CU. Hence, L2$ sends both data and new rts and

`wts` to the L1$.

Write Operation: Write requests are handled using a write-though policy from L1$ to L2$, and L1$ adopts a no-write-allocate policy. To have a write hit in a cache, the `warpts` of the cache block needs to be within the `lease` of the requested cache block. Otherwise, it is considered a write miss. When there is a write hit in the L1$, the data is written in the L1$, but the access to the data is locked until L2$ is updated and L2$ sends the updated timestamps to the L1$ for the data. It is necessary to lock access to the block to ensure that the `warpts` is updated correctly using the `wts` value that the L1$ receives from the L2$. Any discrepancy in updating `warpts` may result in an incorrect ordering of memory access requests. If there is an L1$ write miss, the data is directly sent to the L2$ to complete the write operation. For an L2$ write miss, the L2$ sends a write request to the MM. If we get a write hit at L2$, the L2$ writes the data to the block in the L2$ and updates the timestamps for that block. L2$ then sends the updated `rts` and `wts` values to the L1$.

For both read and write operations, the responses from L1$ to CU contain the `wts` value from the most recent memory operation. Based on this `wts` value, CU updates its `warpts`.

4.1.1 Applicability of G-TSC Protocol in MGPU System

This G-TSC protocol (Tabbakh et al., 2018), which was designed for intra-GPU coherence, cannot be readily applied to MGPU systems. Maintaining coherence across multiple GPUs is more challenging as the L1$s of a GPU can only interact with their own L2$. We need to maintain coherence across multiple L2$s in different GPUs as well as in the shared MM. A straightforward extension of G-TSC would be to add timestamps to each block of data in the MM, but that would lead to significant area overhead as we would need space to store the timestamps of each block of data in the MM. G-TSC also needs to maintain a logical time counter at the compute unit which needs to send timestamps i.e. `warpts` back and forth between CUs and L1$s leading to additional traffic overhead.

4.2 MGCC Protocol for Coherence in MGPU-TSM

We define the MGCC protocol using a single-writer-multiple-reader (SWMR) invariant. The terms used to explain the MGCC protocol are defined in Table 4.1. Unlike the G-TSC protocol, we do not have a `warpts` but assign a timestamp `cts` to each of the L1$s and L2$s. Each CU has a private L1$, hence the `cts` for an L1$ is equivalent to the `warpts` in the G-TSC protocol. Managing timestamps at the caches allows us to reduce timestamp traffic between the L1$ and CU as well as between the L1$ and L2$ by eliminating the need for sending `cts` with requests and responses to maintain coherence as compared to G-TSC protocol which sends `warpts` with every request. We adopt write-through (WT) cache policy for L2$.[2] The memory operations are ordered based on the logical time, in particular, `cts`. If two requests have the same `cts` value, the cache uses physical time to order them. The key idea is that the block is only valid in the cache if the `cts` is within the valid `lease` period. Additionally, while G-TSC simply provides the same lease for both reads and writes, we provide different lease values for reads and writes. By doing so, we can exploit the temporal locality of data. To elaborate, each write operation moves the logical time counter ahead by the write lease value. If we use the same lease for read as for write, each write operation will lead to self-invalidation of the previously read block. Figure 4·1 shows the transactions between CUs, L1$s, L2$s, and MM for read and write operations. We explain these transactions with the help of Algorithms 1–5. We assume sequential consistency for this illustration.

4.2.1 Read Operations

L1$: Figure 4·1a shows the transactions between a CU and the L1$ for read operations. As shown in Algorithm 1, a cache hit at L1$ occurs only when there is an address (tag)

[2]We could have used write-back (WB) cache policy as well. However, it would require additional complexity to handle the L2$ evictions and dirty data in the L2$ leading to additional traffic or stalling. We leave that for future work.

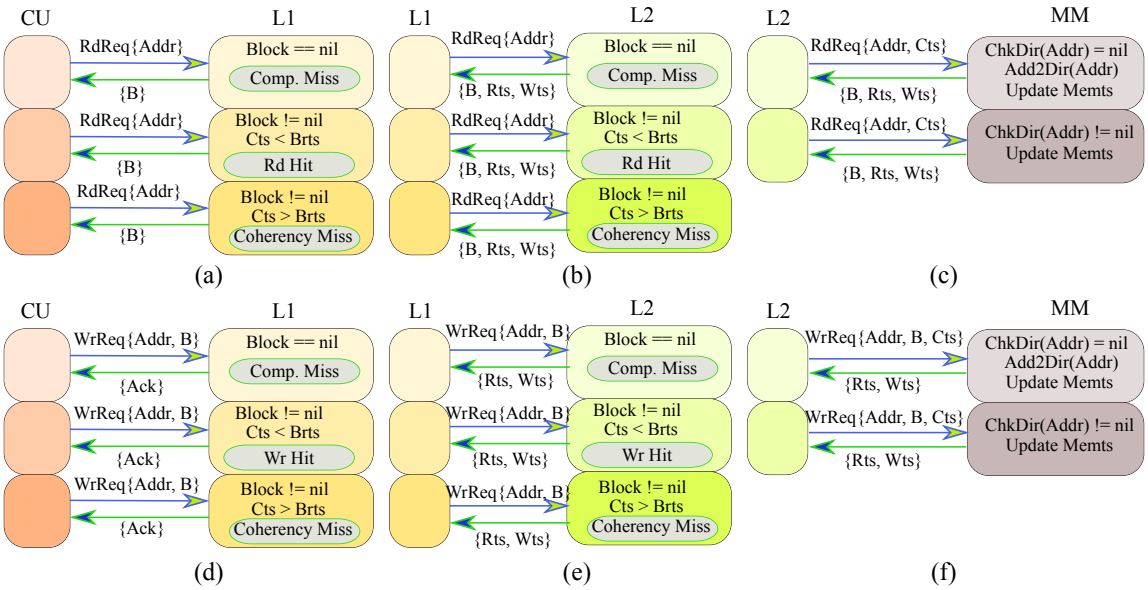


Figure 4·1: Transactions between (a) a CU and an L1$ for read operations, (b) an L1$ and an L2$ for read operations, (c) an L2$ and the MM for read operations, (d) a CU and an L1$ for write operations, (e) an L1$ and an L2$ for write operations, and (f) an L2$ and the MM for write operations.

match and the current timestamp, `cts`, is within the lease period of the cache block. If there is a tag hit, but the `cts` is not within the lease period, we fetch the cache block from L2$ with new `rts` and `wts` values. For an L1$ miss, we fetch the cache block with its `rts` and `wts` values from L2$.

L2$: Algorithm 2 shows how read requests are handled by the L2$. The L2$ hit or miss is similar to that of L1$. Figure 4·1b shows the transactions between the L1$ and the L2$ for read requests. If there is a cache hit and the lease is valid, the L2$ sends the cache block, `rts`, and `wts` to the L1$. If there is a cache miss in the L2$, then the L2$ sends a request to the MM. After fetching the cache block from MM, the L2$ responds to the L1$ with the cache block, `rts`, and `wts`. If there is a tag match, but `cts` is not within the lease period in L2$, we re-fetch the data with new timestamps from the MM. This re-fetching of data ensures coherence in case another GPU modified the data in the MM. Note that G-TSC protocol only fetches renewed timestamps from L2$ if data has not been modified

Algorithm 1: Read Request to L1

```
Initialization: cts = 0;
Fetch RdReqFromCU{Addr};
if Block(Addr) == nil or  cts > rts(Block) then
    Send RdReqToL2{Addr};
    Fetch RspFromL2{Data, rts, wts};
    Bwts = max[cts, wts];
    Brts = max[wts + 1, rts];
    Send RspToCU{Block{Data}};
else if cts <= rts(Block) then
    Send RspToCU{Block{Data}};
```

Algorithm 2: Read Request to L2

```
Initialization: cts = 0;
Fetch RdReqFromL1{Addr};
if Block(Addr) == nil or  cts > rts(Block) then
    Send RdReqToMM{Addr};
    Fetch RspFromMM{Data, Mrts, Mwts};
    Bwts = max[cts, Mwts];
    Brts = max[wts + 1, Mrts];
    Send RspToL1{Block{Data, Brts, Bwts}};
else if cts <= rts(Block) then
    Send RspToL1{Block{Data, Brts, Bwts}};
```

Algorithm 3: Read or WriteRequest to MM

```
Initialization: Memts = 0;
Fetch RdReqFromL1{Addr};
if TSU(Addr) == nil then
    AddEntryToTSUBlockAddr;
if Req==ReadReq then
    MemtsEntry = memts + RdLease;
    Mrts = MemtsEntry; Mwts = Mrts - RdLease;
else if Req==WriteReq then
    MemtsEntry = memts + WrLease;
    Mrts = MemTsEntry; Mwts = Mrts - WrLease;
Send RspToL2{Block{Data, rts, wts}};
```

Algorithm 4: Write Request to L1

```
Initialization: cts = 0;
Fetch WrReqFromCU{Addr};
if cts <= rts(Block) then
    WriteToBlock;
    LockAccesstoBlock;
    Send WrReqToL2{Addr};
    Fetch RspFromL2{rts, wts};
    Bwts = max[cts, wts];
    Brts = max[wts+1, rts];
    cts = max[cts, Bwts];
    UnlockAccessToBlock;
    Send RspToCU{Block{Data}};
else
    Send WrReqToL2{Addr};
    Fetch RspFromL2{Block, rts, wts};
    WriteBlockToCache;
    Bwts = max[cts, wts]
    Brts = max[wts+1, rts];
    cts = max[cts, Bwts];
    Send RspToCU{Block{Data}};
```

Algorithm 5: Write Request to L2

```
Initialization: cts = 0;
Fetch WrReqFromL1{Addr};
if cts <= rts(Block) then
    WriteToBlock;
    LockAccesstoBlock;
    Send WrReqToMM{Addr};
    Fetch RspFromMM{Mrts, Mwts};
    Bwts = max[cts, Mwts];
    Brts = max[wts+1, Mrts];
    cts = max[cts, Bwts];
    UnlockAccessToBlock;
    Send RspToL1{Block{Data, Brts, Bwts}};
else
    Send WrReqToMM{Addr};
    Fetch RspFromMM{Block, rts, wts};
    WriteBlockToCache;
    Bwts = max[cts, wts]
    Brts = max[wts+1, rts];
    cts = max[cts, Bwts];
    Send RspToL1{Block{Data, Brts, Bwts}};
```

by another CU. However, such re-fetching requires CUs or L1$s to send the `warpts` with each request (which we eliminated to reduce traffic) and adds more complexity when we deal with a deeper memory hierarchy.

MM: Figure 4·1c shows the transactions to and from the MM for read requests from the L2$. Algorithm 3 explains how a read request from the L2$ is handled by the MC. The MM tracks the timestamp of each block accessed by the L2$s of all the GPUs using the TSU. The TSU stores the read address and the timestamp (`memts`) of the block, but not data itself. `memts` is used to keep track of the lease of a block sent to the L2$s. If there is no entry for the requested address in the TSU (i.e., the block has never been requested by the L2$s), it adds the address and then updates the `memts` of the block using the `Mrts`[3] allocated for the read operation. If there is already an entry in the TSU for the requested address, the TSU extends the `memts` of the entry using the `Mrts` for the read operation.

4.2.2 Write Operations

L1$: Figure 4·1d shows the transactions that take place for write requests to the L1$. We adopt a write-through (WT) cache policy for both the L1$s and L2$s. Algorithm 4

[3]Read timestamp, `Mrts` and write timestamp, `Mwts` are design parameters for the MGCC protocol; depending on the implementation, these values can be statically or dynamically assigned.

illustrates how write requests to L1$ are handled. Due to the WT policy, a write request to L1$ triggers a write request from L1$ to L2$, irrespective of a cache hit or miss. If the `cts` is within the `lease`, it is a write hit. In case of a write hit, the data is written immediately to the cache block in the L1$ and a write request is sent to the L2$. Access to the block is locked until the L1$ receives a write response, along with the new timestamps, from the L2$. The access is locked by adding an entry to the miss-status-holding-register (MSHR). In the case of a write miss in the L1$, the L1$ sends the request to the L2$. Once the L2$ returns both the block and its timestamps to the L1$, the L1$ writes data to the appropriate location and updates its `cts`.

L2$: Figure 4·1e shows the transactions that take place for write requests to the L2$s. Algorithm 5 demonstrates how a write request to the L2$ is serviced. As we are using a WT policy for the L2$, a write request to the L2$ triggers a write request from the L2$ to the MM, irrespective of whether the access is a cache hit or miss. Again, the cache hit and miss conditions are the same as in the case of the L1$. If the access is a cache hit, the data is written to the block in the L2$ and a write request is sent to the MM. The L2$ updates the timestamp of the cache block using the response that it receives from the MM. The access to the block is locked until the write response and the timestamps are received from the MM. If the L2$ access results in a cache miss, L2$ sends a write request to the MM. The write request includes the data and address. The MM sends a response with the block and updated timestamps. Next, the L2$ issues a write to the block and updates its timestamps using the response from the MM.

MM: Figure 4·1f shows the transactions to and from the MM for a write request from the L2$. Algorithm 3 explains how a write request from the L2$ is serviced by the MC. If there is no matching entry for the requested address in the TSU, then the TSU adds the address and updates the timestamp of the block using the `lease` for the write operation. If

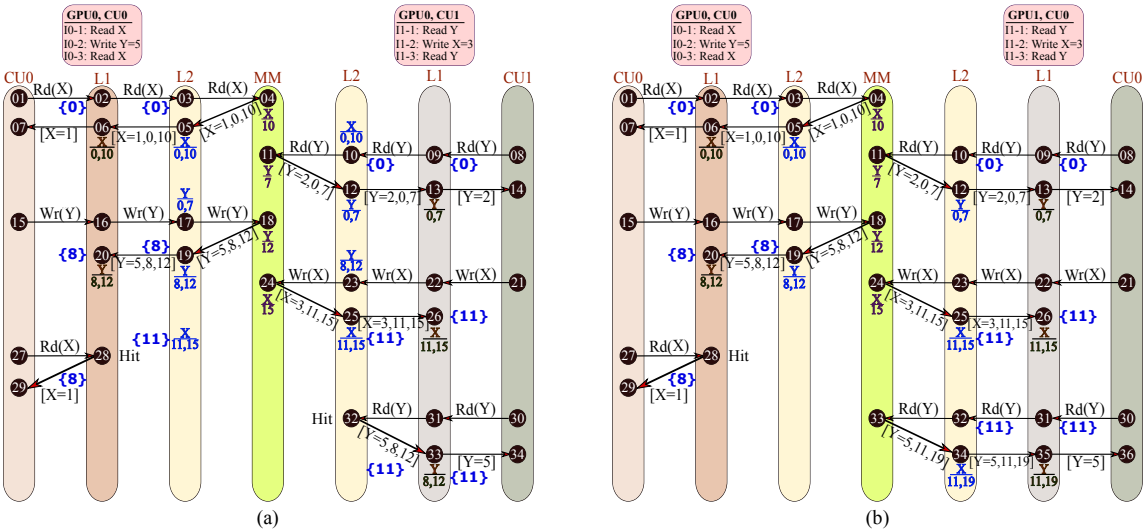


Figure 4·2: The timeline for (a) the intra- and (b) inter-GPU coherence. [] represents response traffic in [Data, `wts`, `rts`] or [Data] format, {} represents the updated `cts` of a cache. In (a), the two L2$ instances refer to the same physical L2$.

there is an entry present in the TSU for the requested address, the MM increases the `memts` of the entry using the `lease` for a write operation.

4.2.3 Intra-GPU Coherence

We use instruction sequences identical to those described by Tabbakh et al. (Tabbakh et al., 2018) to explain both intra-GPU and inter-GPU coherence using MGCC. Here, we first present how intra-GPU coherence is maintained using our MGCC protocol. Figure 4·2(a) shows the instructions and the sequence of steps for maintaining intra-GPU coherence. In this example, we have two compute units, CU0 and CU1. Both CU0 and CU1 belong to GPU0. Each CU has a private L1$, but the L2$ is shared between the two CUs. In Figure 4·2(a), we show two L2$s for the sake of explanation, but both L2$s are the same L2$. CU0 executes 3 instructions, I0-1, I0-2, and I0-3, which read location [X], write to location [Y] and read location [X], respectively. Similarly, CU1 executes 3 instructions, I1-1, I1-2, and I1-3, which are: read location [Y], write location [X], and read location

[Y], respectively. Both L1$ and L2$ have initial cts values of 0. ❶ to ㉞ correspond to different memory events that occur during the execution of the three instructions. At ❶, CU0 issues a read to location [X]. This request misses in the L1$. So at ❷, the L1$ sends a read request to L2$. As the request misses in L2$ as well, the L2$ sends a read request to the MM at ❸. At ❹, the MM sends the response to the L2$ with rts and a wts values of 10 and 0, respectively (we choose these values for the timestamps just as an example. Our protocol works correctly for any values of rts and wts). Based on the cache block and the timestamps received from MM, at ❺ L2$ updates its cts, the block's rts and wts, and responds to L1$ with the updated rts and wts values, along with the cache block. Similarly, the L1$ updates its cts, and rts and wts values for the cache block at ❻. The CU finally receives the data from L1$ at ❼. Instruction I1-1 from CU1 issues a read from location [Y] and follows the same steps as I0-1. The CU1 receives the data through steps ❽ to ⓮. We assume a different timestamp values (wts= 0, rts= 7) for location [Y] in this example.

CU0 requests to write to location [Y] at ⓯. The write request from a CU is served by the MM, regardless of whether it is a cache hit at L1$ or L2$ as the MM is responsible for updating the lease for a cache block to maintain global visibility of updates. At ⓰, the L1$ of CU0 sends a write request to L2$. This results in a cache hit at L2$ as the location [Y] was previously read by CU1 and cts $\leq$ rts. At ⓱, the L2$ sends a write request to the MM for location [Y]. The MM updates the value and timestamps for location [Y]. We assume a lease of 5 for write operations in this example, but the protocol can work with any value of the lease. At ⓲, the MM sends the response with rts= 12 and wts= 8 for the block containing [Y] to the L2$. Then the L2$ updates the timestamps for [Y] and sets cts= 8 at ⓳ and sends the updated timestamps to the L1$ of CU0. At ⓴, the L1$ updates the timestamps for the block containing [Y] and the associated cts= 8. Note that we do not show the actions to lock and unlock a block in the diagram for clarity. Every

write request to a block in the cache must lock access to the block in both L1$ and L2$ until receiving a response from the MM. At step ㉑, there is a write request (I1-2) from the CU1 at location [X]. This follows the same steps followed by I0-2. The response to the write request is executed in steps ㉒ to ㉖. Now, both L1$ and L2$ of CU1 have a `cts` value of 11 after completing the write request to location [X]. At ㉗, there is a read request for location [X] from CU0. At ㉘, the `cts` value is 8 and the block for location [X] has a `rts` value of 10. Hence, it is a cache hit in L1$. Note that the advantage of using a logical timestamp is that it allows the scheduling of a memory operation in the future by assigning a larger `wts` value. Hence, the previous write on [X] by CU1 will be visible later to L1$ of CU0 as it has a `cts` value lower than the assigned `wts` value to the block for the write request by CU1 at ㉔. Since the `cts` of the L1$ of CU0 is smaller than the `cts` value of the L1$ of CU1 at this point, the read by CU0 of the L1$ happens before the write by the L1$ of CU1. The data is sent to CU0 by L1$ at ㉙. At ㉚, CU1 sends a request to read location [Y]. This request creates a coherence miss in L1$. This is because the `cts` is 11, but the block for location [Y] has a `rts` of 7. At ㉛, L1$ sends a read request to L2$. This request results in a cache hit at L2$, since L2$ has a `cts` value of 11 and the block for [Y] has `rts`= 12 and `wts`= 8. The execution order of the instructions in this example is I0-1 $\rightarrow$ I1-1 $\rightarrow$ I0-2 $\rightarrow$ I0-3 $\rightarrow$ I1-2 $\rightarrow$ I1-3.

4.2.4 Inter-GPU Coherence

In this example, we use the same instructions as in the previous example for intra-GPU coherence. CU0 of GPU0 executes instructions I0-1, I0-2, and I0-3. However, instructions I1-1, I1-2, and I1-3 are executed by the CU0 of GPU1 in this example. Thus, we have two different L2$s, one connected to GPU0 and one connected to GPU1. Figure 4·2b shows the instructions and the sequence of execution for explaining inter-GPU coherence. The read request from CU0 of GPU0 to read location [X] and read request from CU0 of GPU1 to read location [Y] follow the exact same steps (steps ⑪ - ⑭) as in the case of intra-GPU

coherence. The write request from CU0 of GPU0 at ⓯ and the write request from CU0 of GPU1 at ㉑ are also handled in the same manner as in the case of intra-GPU coherence. The only difference is that the data for the write of [X] and for the write of [Y] reside in different L2$s. The read request (I0-3) by CU0 of GPU0 at ㉗ still produces a cache hit in L1$. The read request issued by CU0 of GPU1 (I1-3) results in a different set of execution steps. This is because at ㉜, there is no longer an L2$ hit, as the lease (`rts`= 7, `wts`= 0) expired for a `cts`= 11. Hence the data for [Y] has to be fetched from the MM. The MM has the updated value written by CU0 of GPU0. This value is received by CU0 of GPU1, and thus it becomes coherent with CU0 of GPU0. The execution order for both the instructions in this example is again I0-1 $\rightarrow$ I1-1 $\rightarrow$ I0-2 $\rightarrow$ I0-3 $\rightarrow$ I1-2 $\rightarrow$ I1-3.

4.2.5 Request Tracker Operation

The request tracker (ReqTrack) logic (illustrated in Figure 3·2) is physically implemented in hardware. It works as the first stage of the MC that receives requests from 4 different L2$ banks. Upon receiving multiple requests it compares the timestamp of the received requests and place the requests in ascending order of timestamp values in a buffer to be served by the MC. Thus, it helps maintain sequential consistency based on both physical and logical time order as it considers the physical time of the arrival of request and then logical time for responding to the requests. For relaxed consistency model, it allows reordering cache blocks with same timestamps or an independent block with smaller `wts` for read operations as read operation on independent cache block is permissible under RC.

4.2.6 TSU Implementation

The TSU is physically placed between the ReqTrack and MM in the same chiplet or chip that houses the memory controllers. We could have chosen to place the TSU in the DRAM layers, but this would increase memory access latency. We designed the TSU as an 8-way set associative cache. The TSU needs to store the `memts` for all of the blocks in all the L2$s

in the MGPU system. We use 16 bits for each `memts`. Since we have 8 distributed L2$ modules in each GPU, each way of the TSU keeps track of the timestamps of the cache blocks in one of the L2$ modules. For example, for an MGPU with a 2MB L2$ per GPU, we need 64KB of space for the timestamps in the TSU for each GPU. As TSU logic only searches for the presence of the timestamp of a block and generates or updates timestamps, the latency for accessing the TSU is identical to a L3$ hit time of 40 cycles (Levinthal, 2009). We conservatively assume a 50 cycle access latency for TSU.

Figure 4·3 shows the operation of the TSU. A request from the memory controller is sent to the TSU and the DRAM layer in parallel. The TSU responds with the timestamp for the cache block, and in parallel with the DRAM layer, responds with the cache block. Thus, the TSU never impacts the critical path of the DRAM access, and so does not add any performance overhead. The eviction of TSU entries is related to the eviction of L2$ entries. When there is an eviction from the L2$ of a GPU, the TSU also evicts the timestamp for that cache block if it is not shared with other GPUs. The TSU logic determines the block sharers using the `memts` value (if the value of `memts` is greater than one lease period, it is assumed to be shared). In case the TSU is full, the TSU evicts the cache block with lowest `memts` value.

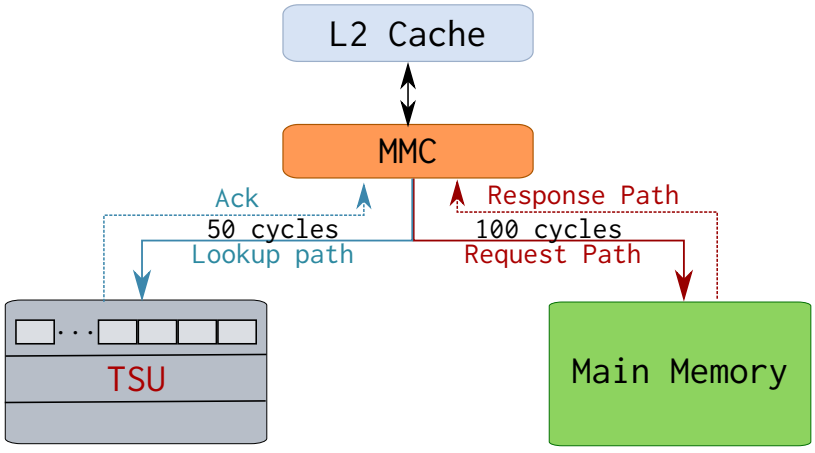


Figure 4·3: Time Stamp Unit (TSU). The TSU operates independently and in parallel with the memory access.

4.2.7 Timestamp Design

We use 16-bit fields for each one of the timestamps, `rts` and `wts`. We need 1KB of storage for each L1$ of size 16KB and 8KB of storage per L2$ of size 256KB for holding the read and write timestamps. For each cache timestamp (`cts`), we use 64 bits. For an example GPU with 64 CUs, the GPU requires a total of 72 `cts` entries (64 for the 64 private L1$s belonging to each CU and 8 for the L2$). Hence, we need a total of 576B (512B for all L1$ and 64B for 8 L2$) to represent all the `cts` values for the entire GPU.

Assuming 64B cache block size, 4B for ACK, 4B for metadata and 8B address, from L2$ to L1$ or from MM to L2$, MGCC increases the network traffic by 5% and 5.26% for each read response and each write response, respectively. For each read and each write request from L2$ to MM, MGCC introduces 16.7% and 2.63% additional traffic, respectively. If the timestamp value overflows, instead of flushing the cache, we simply re-initialize the timestamps to 0. This re-initialization results in a cache miss for one of the cache blocks. However, given we are using a write-through policy for writes in both L1$ and L2$, there is no chance of losing data belonging to the cache block experiencing the overflow. We just need to do an extra MM access.

In summary, MGCC has the following novelties:

- MGCC replaces CU-level timestamps with cache-level timestamps to reduce CU-to-L1 traffic.
- MGCC introduces a novel request tracker in hardware to address load-after-store and store-after-load orderings.
- MGCC implements a novel TSU to keep track of the timestamp values of the cache block and the TSU is efficiently accessed such that it does not add any performance overhead.

- Unlike G-TSC, MGCC eliminates the logic to extend the lease to simplify the coherence protocol and help reduce request traffic.

4.3 Evaluation Methodology

In this section, we describe the evaluation methodology for the proposed MGCC protocol that provides intra- and inter-GPU hardware coherence support for MGPU-TSM. We use the MGPUSim simulator and standard benchmarks as described in Section 3.2. We describe the MGPU-TSM system configurations[4], simulation platform and synthetic benchmarks to stress test the MGCC protocol in this section. For individual GPUs, the specification remains same as shown in Table 3.1.

4.3.1 MGPU System Configurations

We evaluate the following different MGPU configurations to evaluate the MGCC protocol:

1. MGPU-TSM system with WB L2$, no coherence and relaxed consistency (`TSM-WB-NC-RC`).
2. MGPU-TSM system with WT L2$, no coherence and relaxed consistency (`TSM-WT-NC-RC`).
3. MGPU-TSM system with WT L2$, MGCC-based coherence and relaxed consistency (`TSM-WT-C-RC`).
4. MGPU-TSM system with WT L2$, MGCC based coherence and sequential consistency (`TSM-WT-C-SC`).
5. MGPU system with RDMA, HMG (Ren et al., 2020) coherence and scope-based consistency (`HMG`) [5].

[4]To name the MGPU systems we use the following notation: `C` = cache-coherence support, `NC` = No coherence support, `WT` = L2$ with write-through policy, and `WB` = L2$ with write-back policy.

[5]HMG is the most recently proposed solution for efficient HW cache-coherent support in MGPU systems.

The TSM-WB-NC-RC and TSM-WT-NC-RC configurations are used to compare L2$ write-back (WB) policy with L2$ write-through (WT) policy in a MGPU-TSM system. This comparison is necessary to understand the impact of changing L2 cache write policy from write-back to write-through, which is a necessary condition for our proposed MGCC protocol. We compare the TSM-WT-NC-RC and TSM-WT-C-RC configurations to determine the overhead of MGCC coherence protocol. The TSM-WT-C-SC highlights the impact of strict sequential consistency protocol as compared to the relaxed consistency protocol. The MGPU system with HMG represents the most recently proposed solution that uses a VI-based coherence protocol to improve performance of RDMA-based MGPU system. In the case of HMG, to maintain coherence, each GPU's L2$ is connected to the RDMA engine to send a remote request as the HMG protocol uses RDMA via L2$. For RDMA connections between L2$ and MM, we use PCIe 4.0 links. To maintain fairness of comparison, we provide all the variable system-level scope for the HMG as all the MGPU-TSM configurations provide coherence support across the entire memory hierarchy.

4.3.2 Simulation Platform

We extended the MGPUSim simulator and the memory hierarchy of GPUs to support MGCC. We verified our implementation of MGCC using unit, integration, and acceptance tests provided with the simulator. We also extended the simulator to support the HMG protocol by implementing a hash function that assigns a home node for a given address, directory support for tracking sharers and invalidation support for sending messages to the sharers as needed.

4.3.3 Synthetic Benchmarks

We use standard application benchmarks (see Section 3.2.3) as well as synthetic benchmarks to evaluate our MGCC protocol. The publicly available benchmark suites mentioned in Section 3.2.3 have been developed considering the lack of hardware-level coherence sup-

port in GPUs. Hence, these benchmarks cannot necessarily harness the potential benefit of the hardware-level coherence protocol like MGCC. To stress test our MGCC protocol, we develop a synthetic benchmark suite called **Xtreme**. There are three benchmarks in the Xtreme suite[6]. All the benchmarks in the Xtreme suite perform a basic vector operation: $C = A + B$, where A, B and C are floating point vectors. We describe the basic operation of the **Xtreme** benchmarks with a simple example. For each example, we assume the following:

1. There are two GPUs: GPU_X and GPU_Y.
2. Both GPU_X and GPU_Y are equipped with two CUs each: CU_{X0}, CU_{X1}, and CU_{Y0} and CU_{Y1}, respectively.
3. There are three vectors A, B and C that are used to compute $C = A + B$ using both GPU_X and GPU_Y.
4. Each of the three vectors, A, B and C, are split into 4 slices: A_0, A_1, A_2 and A_3; B_0, B_1, B_2 and B_3; and C_0, C_1, C_2, and C_3.
5. At the beginning of the program, CU_{X0} reads A_0, B_0, and C_0; CU_{X1} reads A_1, B_1, and C_1; CU_{Y0} reads A_2, B_2, and C_2; CU_{Y1} reads A_3, B_3, and C_3.

The three **Xtreme** benchmarks work as follows:

Xtreme1

① CU_{X0} performs $C_0 = A_0 + B_0$; Similarly, CU_{X1} operates on A_1, B_1 and C_1; CU_{Y0} operates on A_2, B_2 and C_2; and CU_{Y1} operates on A_3, B_3 and C_3.
② Repeat step ① 10 times.

[6]All the benchmarks in the **Xtreme** suite perform repeated writes to and reads from the same location. This extreme behavior is typically uncommon in regular GPU benchmarks, and so the name Xtreme.

③ CU_{X0} performs $A_0 = C_0 + B_0$; Similarly, CU_{X1} operates on A_1, B_1 and C_1; CU_{Y0} operates on A_2, B_2 and C_2; and CU_{Y1} operates on A_3, B_3 and C_3.

④ Repeat step ③ 10 times.

With **Xtreme1**, we evaluate the impact of consecutive writes to the same location by a CU. There is no data sharing between the CUs or the GPUs. When there is a write to any location, the corresponding, current logical time (`cts`) of the L1$ and L2$ step ahead and generate read misses. Steps ② and ④ force coherence misses in the caches.

Xtreme2

① CU_{X0} performs $C_0 = A_0 + B_0$; Similarly, CU_{X1} operates on A_1, B_1 and C_1; CU_{Y0} operates on A_2, B_2 and C_2; and CU_{Y1} operates on A_3, B_3 and C_3.

② CU_{X0} performs $A_1 = C_1 + B_1$;

③ Repeat step ② 10 times.

④ Repeat step ①

With **Xtreme2**, we stress test MGCC for intra-GPU coherence. There is a SWMR invariant dependency between CU_{X0} and CU_{X1} at ②, CU_{X0} writes to a location that was previously read by CU_{X1}. Step ③ forces coherence misses.

Xtreme3

① CU_{X0} performs $C_0 = A_0 + B_0$; Similarly, CU_{X1} operates on A_1, B_1 and C_1; CU_{Y0} operates on A_2, B_2 and C_2; and CU_{Y1} operates on A_3, B_3 and C_3.

② CU_{X0} performs $A_3 = C_3 + B_3$;

③ Repeat step ② 10 times.

④ Repeat step ①

With **Xtreme3**, we stress test MGCC for inter-GPU coherence. The difference between **Xtreme2** and **Xtreme3** is that at ② CU_{X0} writes to a location that was previously read by CU_{X1} and CU_{Y1}, respectively.

In our evaluation, we vary vector sizes from 384KB to 24MB for A, B and C so that we can examine the impact of capacity misses at different levels of the memory hierarchy.

4.4 Evaluation

In this section, using standard benchmarks and synthetic benchmarks, we discuss the impact of the MGCC protocol on the MGPU-TSM .

4.4.1 Standard Application Benchmarks

In this section we evaluate the impact of the MGCC protocol using the standard application benchmarks. Then, we present a more in depth evaluation of MGCC using `Xtreme` benchmark suite which contains the applications that require coherence.

Figure 4·4a shows the speedup for different MGPU configurations, as compared to `TSM-WB-NC-RC`. We compare 4 different MGPU-TSM configurations: `TSM-WB-NC-RC`, `TSM-WT-NC-RC`, `TSM-WT-C-RC` and `TSM-WT-C-SC` assuming a 4-GPU system.[7] We use a `WrLease` of 5 and a `RdLease` of 10 for this evaluation.[8] On an average, the `TSM-WT-NC-RC`, `TSM-WT-C-RC`, and `TSM-WT-C-SC` configurations achieve a 0.98×, 0.97×, and 0.79× the `TSM-WB-NC-RC` performance, respectively.

WB vs. WT

As mentioned before, our MGCC protocol requires L2$ to be write-through to ensure a simpler timestamp-based coherence protocol. Hence, we compare the `TSM-WB-NC-RC` and `TSM-WT-NC-RC` configurations and observe that `TSM-WT-NC-RC` achieves 98× the performance, on average, as that of the `TSM-WB-NC-RC`. From Figure 4·4b and Figure 4·4c, we see that for these two configurations, we have similar L1 and L2 cache miss rates. However, Figure 4·4d shows that the average cache access latency (ACAL) for L1$ increases

[7] An evaluation of an MGPU-TSM system with 8 GPUs and 16 GPUs is presented later in this section.

[8] Please refer to Section 4.4.2 for details on why we choose these lease values.

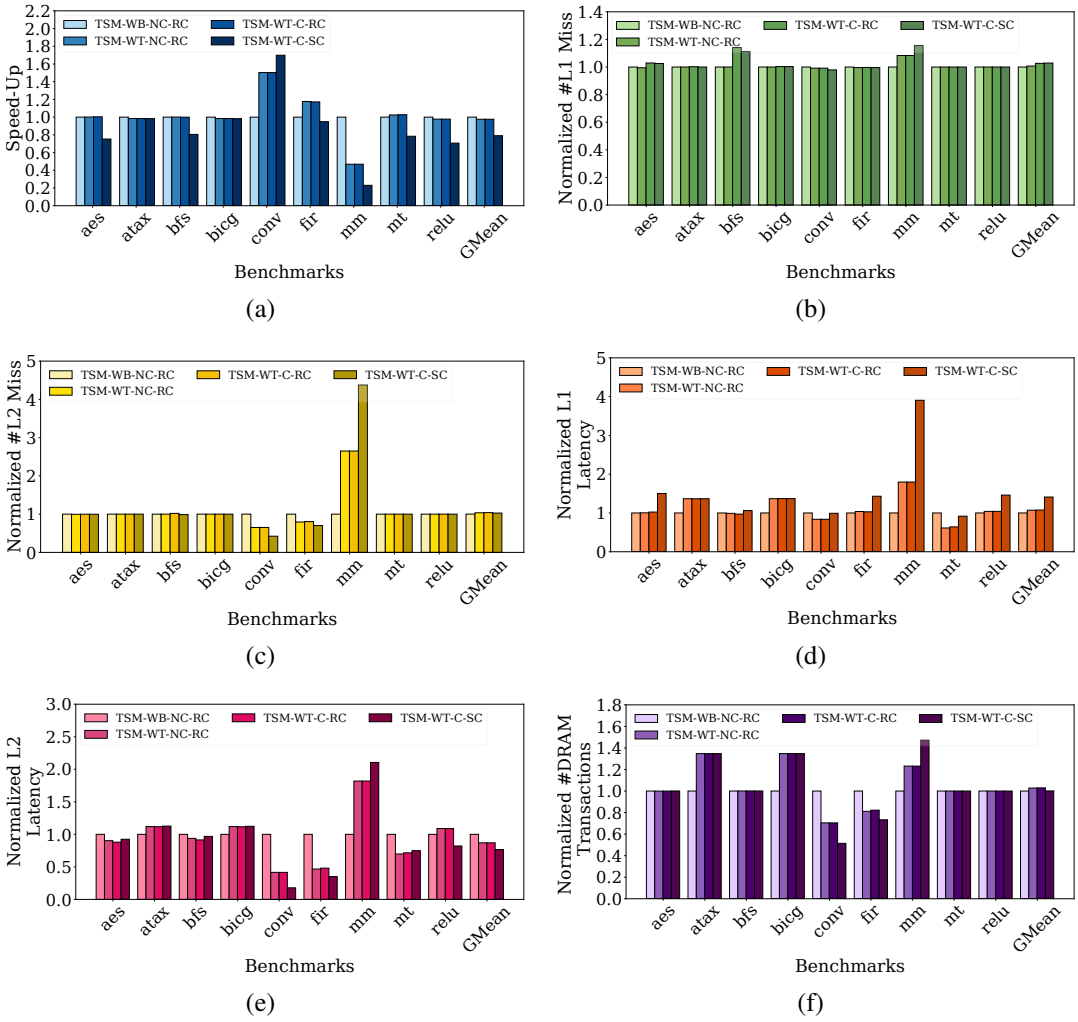


Figure 4·4: Comparison across different MGPU-TSM configurations. The results are normalized w.r.t. the `TSM-WB-NC-RC`

for `TSM-WT-NC-RC` compared to `TSM-WB-NC-RC`. This happens because of the write operations. With WB policy, L2$ sends the write acknowledgement signal to the L1$. However, with the WT policy, the MM has to send the write acknowledgement to the L2$ and then L2$ sends the write acknowledgement to the L1$. Thus, the L1$ ACAL increases. The same phenomenon is responsible for increased L2 cache access latency for benchmarks including **aes, atax, bfs, mm,** and **mt** as observed from Figure 4·4e. Figure 4·4e also shows some benchmarks such as **conv, fir** and **relu** have lower L2$ ACAL for WT as compared

to WB. For a read or write miss in the L2$ with a WB policy, first, the L2$ performs a write to MM to generate a cache eviction if there is either a conflict or capacity miss. Only then the L2$ can service the pending read or write transactions. The L2$ generating the WB becomes a bottleneck when there are frequent cache evictions. This is why **conv, fir** and **relu** have lower L2$ ACAL for WT. Consequently, L1$ with WT, on average, suffers from 7.1% increased ACAL for WT, whereas L2$ achieves 13.0% lower ACAL for WT compared to WB.

Coherence Overhead

Now, we analyze the `TSM-WT-C-RC` configurations versus the `TSM-WT-NC-RC` configurations to understand the impact of introducing the MGCC protocol for relaxed consistency model. From Figure 4·4(a), we observe that there is only 1% performance gap between the `TSM-WT-NC-RC` and `TSM-WT-C-RC` configurations. Two factors are primarily responsible for this low overhead:

1. Under the existing relaxed consistency model, the read and write requests for different cache lines can be reordered unless the programmer puts explicit barrier. As these existing workloads already have the required barriers for both `TSM-WT-NC-RC` and `TSM-WT-C-RC`, we do not see much performance degradation due to MGCC.

2. Most of the benchmarks are streaming in nature and we chose data sizes significantly larger than the total L2$ capacity of all GPUs combined. Hence, there are significantly more frequent capacity misses in the L1 and L2 caches compared to the coherence misses.

From Figure 4·4a, we observe that `TSM-WT-C-SC` has, on average, 19.0% lower performance than the `TSM-WT-C-RC`. As sequential consistency prohibits reordering of memory operations, we see the additional performance overhead. For both `TSM-WT-C-RC` and `TSM-WT-C-SC`, we observe similar increase (average increase of 1.92% L1$ and 0.4% L2$

misses for TSM-WT-C-RC, and 2.15% L1$ and 0.1% L2$ misses for TSM-WT-C-SC) compared to TSM-WT-NC-RC in cache misses because of the MGCC protocol as shown in Figure 4·4b and 4·4c. However, Figure 4·4d and 4·4e show that the L1$ ACAL, on average, increases by 31.2%, whereas L2$ ACAL decreases by 11.8% for TSM-WT-C-SC compared to TSM-WT-C-RC. Note that the relation between the speed-up is not necessarily linearly dependent on ACAL. This is because ACAL measures average cache access latency and with the introduction of MGCC the number of cache accesses also increase, which can result in lower ACAL. For TSM-WT-C-SC, in Figure 4·4f, we observe slightly lower (on average, 2.7%) DRAM transaction count as compared to TSM-WT-C-RC. This is because the MC for TSM-WT-C-SC is capable serving multiple requests with a single access to DRAM. Since multiple requests can stay pending for the same cache block, any pending read-after-write request can be served along with the previous write request and we do not take those transactions into account for DRAM transaction count.

To sum up, despite the additional overhead introduced by the MGCC protocol, the TSM-WT-C-RC and TSM-WT-C-SC, on average, achieve a speed-up of 3.7× and 3×, respectively in comparison to existing non-coherent MGPU system with RDMA (evaluated in Section 3.3).

TSM-WT-C-RC vs. HMG

We compare the performance achieved from our proposed coherent MGPU-TSM system to the performance achieved by the most recently proposed MGPU system HMG coherence protocol. To make this comparison fair, we use relaxed consistency for MGPU-TSM with MGCC. Figure 4·5 shows that HMG achieves, on average, only 41.2% performance in comparison to TSM-WT-C-RC. **atax** and **bicg** benchmarks achieve 1.5× and 1.85× speedup, respectively for HMG. HMG caches data fetched using RDMA direct access from remote GPU's L2$ to its own L2$ unlike the RDMA which caches the data into L1$. These two benchmarks have data sharing across multiple cores which benefit from caching the data into the shared

L2$ instead of the private L1$. Note that HMG performs, on average, 1.53× better than RDMA. Overall, TSM-WT-C-RC has not only 2.41× average performance improvement over HMG but also additional benefit of easing programmability.

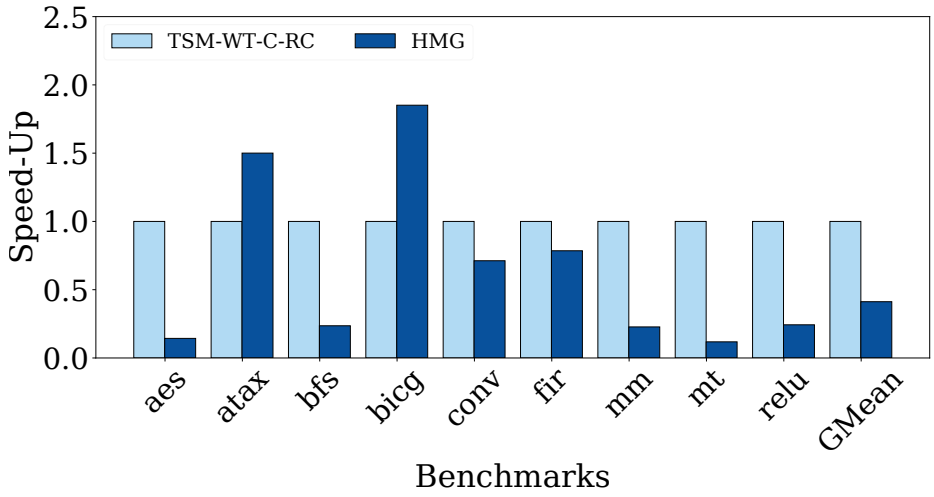


Figure 4·5: Comparison of MGPU-TSM with MGCC to the state-of-the-art MGPU with HMG. Figure shows the speedup of HMG w.r.t. TSM-WT-C-RC

Bandwidth Sensitivity

As mentioned before, we evaluate the MGPU-TSM system with the same bandwidth and latency of the MGPU system with RDMA to maintain fairness of comparison. Practically,

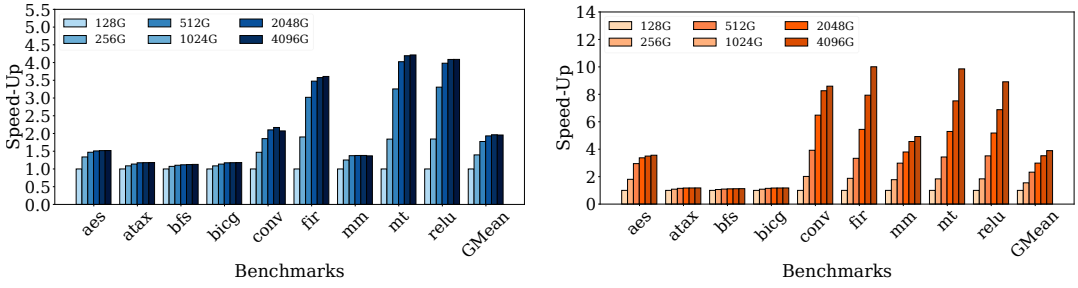


Figure 4·6: Study of bandwidth sensitivity of MGPU-TSM with 4 GPUs for different system bandwidth ranging from 128GB/s to 4096GB/s in the case of RC (a) and SC (b)

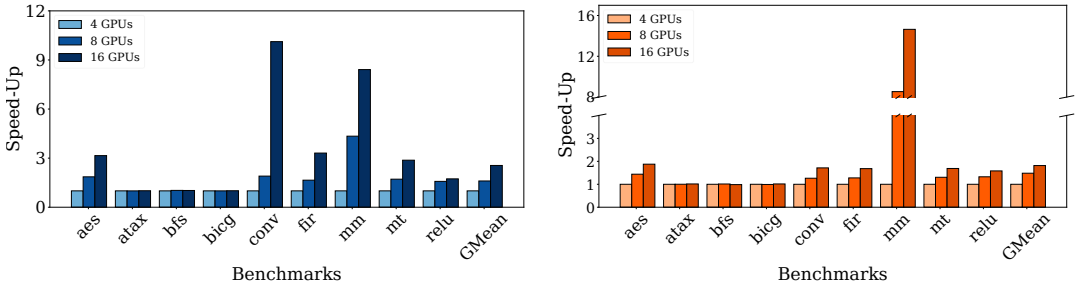


Figure 4·7: Scalability of coherent MGPU-TSM configurations. The results are normalized w.r.t. coherent MGPU-TSM with 4 GPUs.

the system-wide BW for MGPU-TSM is expected to be significantly more[9]. Hence, we perform a BW sensitivity study for MGPU-TSM by increasing the L2-to-MM BW while keeping the latency constant. Figure 4·6 shows the speed-up of MGPU-TSM with 4 GPUs using RC and SC for 256GB/s, 512GB/s, 1TB/s, 2TB/s and 4TB/s versus 128GB/s. With RC, the MGPU-TSM achieves 1.40×, 1.77×, 1.93×, 1.96× and 1.96× speed-up while with SC, the MGPU-TSM achieves 1.55×, 2.33×, 2.99×, 3.52× and 3.90× speed-up for 256GB/s, 512GB/s, 1TB/s, 2TB/s and 4TB/s, respectively. Lack of reordering support mandated by the SC makes the BW more influential on performance for SC as compared to RC. Some compute-intensive benchmarks such as **atax**, **bfs** and **bicg** do not require the larger BW and hence do not achieve significant performance boost for large BW. On the other hand, the relatively more memory-intensive workloads achieve up to 10× speed-up for the BW of 4TB/s versus 128GB/s. Thus, we conclude larger system BW can significantly improve the performance of MGPU-TSM which even with low bandwidth constraints, already performs noticeably better than existing RDMA based MGPU system.

[9] Arunkumar et al. (Arunkumar et al., 2017) describes that 1.5TB/s is practically achievable in a package level integration of an MGPU system. However, we expect larger BW will be possible in future with the progress of silicon-photonic link technology.

Scalability

We use strong scaling to explore the scalability of the MGPU-TSM system with MGCC protocol by varying the GPU count while keeping the size of the workloads constant. Figure 4·7 shows the scalabilty of MGCC protocol for RC and SC for the standard benchmarks. For 8 and 16 GPUs, `TSM-WT-C-RC` achieves, on average, 1.6× and 2.6× speed-up while `TSM-WT-C-SC` achieve 1.5× and 1.8× speed-up compared to the respective 4 GPU configuration. However, three benchmarks, **aes, atax** and **bicg** do no show noticeable scalability. This happens because we are using strong scaling and the data size itself for these benchmarks are not scalable to larger configurations[10]. We verified this by running the non-coherent configurations. On the other hand, MM achieves more than ideal speed-up using 8 and 16 GPUs for both consistency models. This happens because as we distribute fixed amount of work groups across multiple GPUs, each GPU processes less number of workgroups and require less frequent cache evictions because of capacity misses. Overall, our evaluation suggests that the MGCC protocol is scalable assuming that we increase the interconnect and memory bandwidth as well as capacity accordingly.

4.4.2 Xtreme Benchmarks

As discussed before, the standard MGPU benchmarks have been developed with the assumption that there is no hardware support for coherence and a weak-consistency programming model is employed. Hence, we use our synthetic benchmark suite, **Xtreme**, to evaluate the impact of our proposed MGCC protocol for some of the extreme cases of applications, where we need coherence to ensure the correctness of the computation. We use MGPU-TSM with 4 GPUs for this evaluation. Figure 4·8, 4·9 and 4·10 show the comparison of speed-up for `TSM-WT-NC-RC`, `TSM-WT-C-RC` and `TSM-WT-C-SC` for all three **Xtreme**

[10]Increasing the data size could be an option to evaluate the scalability more thoroughly, but we are fundamentally limited by the amount of DRAM memory in the server as increasing the GPU count as well as data size leads to exponential increase in the memory usage by the simulator.

benchmarks. With Xtreme benchmarks, we evaluate three different scenarios:

1. The data size (192KB, 384KB, 768KB) is small, so there are neither L1$ nor L2$ capacity or conflict misses.

2. The data size (1,536KB) is large enough to cause L1$ capacity and conflict misses, but not large enough to cause L2$ capacity or conflict misses.

3. The data size (24,576KB) is large enough to cause both L1$ and L2$ capacity or conflict misses.

Xtreme1

The repeated writes to the same cache location in **Xtreme1** cause the `cts` of both the L1$s and L2$s to step ahead in logical time, leading to coherence misses for the data that was read before. For the **Xtreme1** (see Figure 4·8a) the `TSM-WT-C-RC` and `TSM-WT-C-SC` have up to 21.6% and 23.8% higher runtime than `TSM-WT-NC-RC`, respectively for the vector sizes of 384KB and 768KB, respectively. These two vector sizes are within the capacity of L1$ and demonstrate the impact of L1 cache misses due to coherence. For 1,536KB vector size, the runtime increases by 8.2% and 19.8% for `TSM-WT-C-RC` and `TSM-WT-C-SC`, respectively compared to `TSM-WT-NC-RC`.

Figures 4·8b-f further corroborate this behavior as there are significant increases in the L1$ and L2$ cache misses as well as latencies for these vector sizes. As the L1$ suffers from capacity misses for 1,536KB vector size, the impact of L1$ misses due to coherence is reduced by capacity misses which is present in both coherent and non-coherent cases. For larger vector size of 24,576KB, the `TSM-WT-C-RC` has 7.84% lesser runtime than `TSM-WT-NC-RC`. This is because the timestamp based ordering leads to more cache hits by avoiding eviction of cache blocks with valid timestamp as can be seen from Figure 4·8b and 4·8c. On the other hand, the `TSM-WT-C-SC` has up to 46.1% increased runtime due to sequential accesses. The number of DRAM transactions do not increase much as long

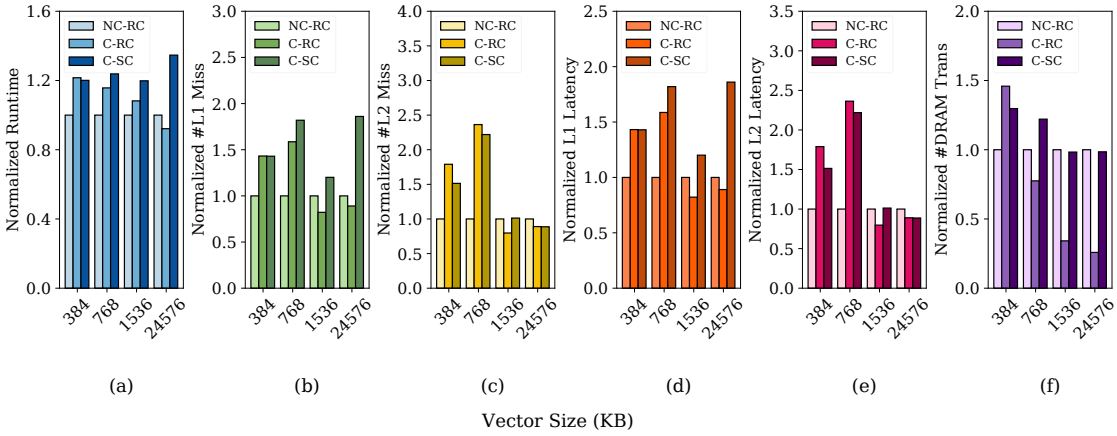


Figure 4·8: Evaluation of MGPU-TSM configurations using **Xtreme1** benchmarks for different vector sizes per GPU.

as the vector sizes are within L2$ capacity. As we increase the vector size significantly above the L2$, the capacity and conflict misses dominate over coherence misses and the performance overhead due to MGCC protocol reduces.

Xtreme2

The **Xtreme2** benchmark evaluates the performance impact of intra-GPU coherence. Figure 4·9a shows that the intra-GPU coherence can lead up to 25.1% and 22.1% coherence overheads for `TSM-WT-C-RC` and `TSM-WT-C-SC`, respectively for the vector size of 768KB per GPU among the vector sizes below L1$ capacity. Figures 4·9b-f shows increased cache miss rates, latency and DRAM transactions. We see a trend similar to the **Xtreme1** benchmark for the vector sizes of 1,536KB and 24,576KB in this intra-GPU coherence case.

Xtreme3

The **Xtreme3** benchmark evaluates the performance impact of inter-GPU coherence. From Figure4·10a we observe up to 26.6% and 22.1% coherence overheads for `TSM-WT-C-RC` and `TSM-WT-C-SC`, respectively for the vector size of 368KB and 768KB per GPU, respec-

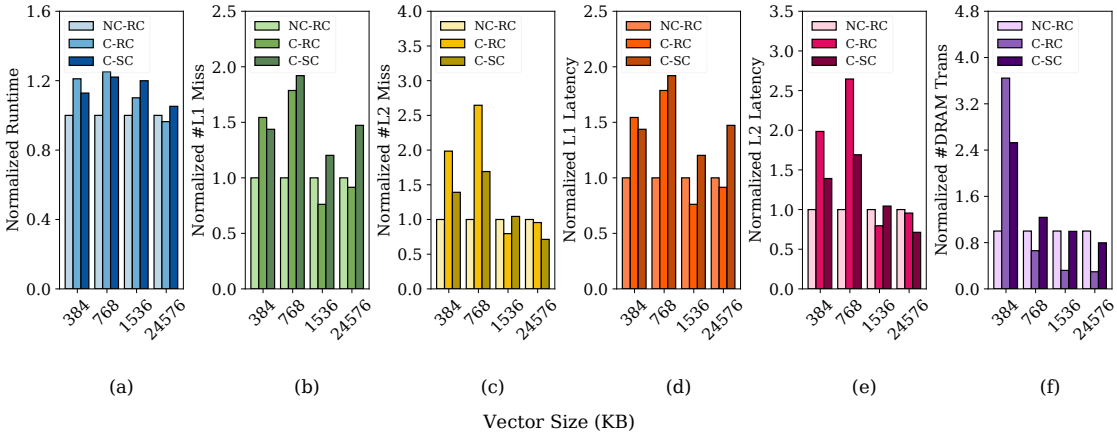


Figure 4·9: Evaluation of MGPU-TSM configurations using **Xtreme2** benchmarks for different vector sizes per GPU.

tively. Increased cache miss rates, latency and DRAM transactions are responsible for this overhead as seen from Figures 4·10(b)-(f). For vector sizes of 1,536KB and 24,576KB, we see a similar trend as in the case of **Xtreme1** and **Xtreme2**.

For both **Xtreme2** and **Xtreme3**, we observe that `TSM-WT-C-SC` performs better than `TSM-WT-C-RC` for vector sizes of 384KB and 768KB because of the additional synchronization overhead associated with `TSM-WT-C-RC` to maintain correctness of operations. To sum up, even with the worst case overhead, the MGPU-TSM performance significantly better than the RDMA based systems.

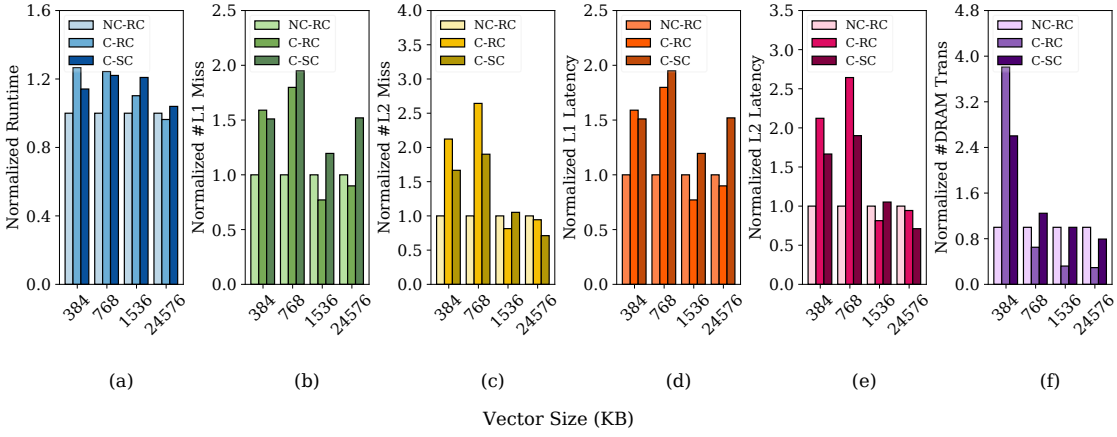


Figure 4·10: Evaluation of MGPU-TSM configurations using **Xtreme3** benchmarks for different vector sizes per GPU.

Sensitivity to Timestamps

We used {RdLease, WrLease} = {5, 10} for our evaluations. We examined the impact of using different (RdLease, WrLease) values using the coherence-aware **Xtreme** benchmarks for different vector sizes. Figure 4·11 shows a representative example of timestamp sensitivity for a vector size of 384KB as this vector size demonstrated significant variation of runtime for all the **Xtreme** benchmarks and across all the MGPU-TSM configurations. Our evaluation suggests that we need to maintain a smaller difference between RdLease and WrLease. In terms of absolute values of RdLease and WrLease, a large value of the RdLease can help an application that performs significantly smaller number of writes than number of reads accessing the same cache block. On the other hand, a small value of the RdLease results in more coherence misses. We choose a WrLease value that is smaller than RdLease value based on the fact that if a CU or a GPU writes to a cache block, it may write to the same cache block in the future. This choice of WrLease, in turn, prevents making cts too large, potentially causing many coherence misses. For all of the configurations {WrLease, RdLease} value of {5,10} demonstrated the least runtime for majority of the cases and consequently, we use these timestamp values for our evaluation.

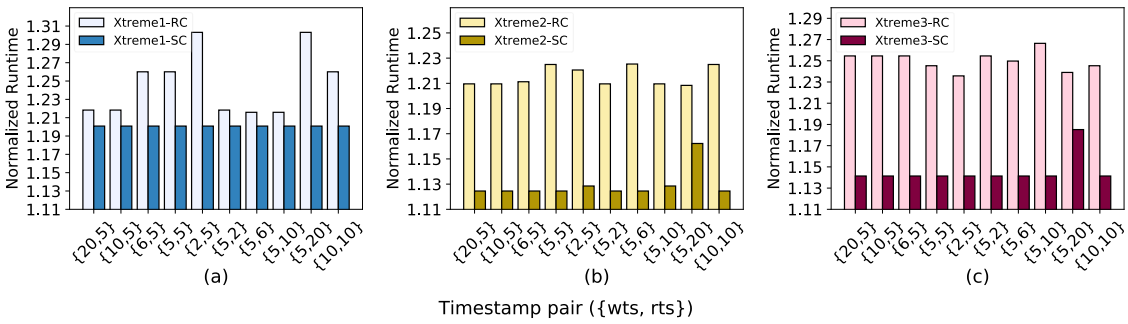


Figure 4·11: Timestamp sensitivity for a vector size of 384KB. The runtime is normalized w.r.t. the TSM-WT-NC-C. {wts,rts} refers to {WrLease,RdLease}

4.5 Summary

To enable seamless sharing of data in an MGPU system with TSM (MGPU-TSM system), we introduce a novel lightweight timestamp-based coherence protocol called MGCC. Our MGCC protocol replaces the compute unit (CU) level logical time counters with cache-level logical time counters to reduce coherence traffic. Furthermore, MGCC uses a novel timestamp storage unit (TSU) in the main memory to perform coherence transactions, with no additional performance overhead. We evaluate a MGPU-TSM system equipped with MGCC using MGPUSim simulator. For a set of standard MGPU benchmarks, a 4-GPU MGPU system with TSM and MGCC, utilizing a relaxed consistency model, achieves a 3.8× and 2.4× speedup versus a 4-GPU MGPU system with existing RDMA, and with the recently proposed HMG coherence protocol, respectively. We also evaluated the scalability of MGCC using different GPU counts (8 and 16) using standard MGPU benchmarks. Broadly speaking, MGCC scales well and achieves, on average, up to a 1.6× and 2.6× speedup for 8 and 16 GPUs, respectively compared to a 4-GPU configuration with MGCC. Furthermore, we stress tested our MGCC protocol using a custom synthetic benchmark suite to evaluate its impact on the overall performance.

Chapter 5

Summary and Future Work

5.1 Summary of the thesis

The advancement of data-intensive applications such as machine learning has forced the researchers to develop non-traditional software and hardware solutions to support these applications. At a high level, this thesis has two aims – first to understand and explain the behaviour of the training and inference operations in these machine learning applications, and second to use this understanding to propose novel memory systems and coherence protocols for next-generation MGPU systems.

In the first part of the thesis, we perform workload characterization on different generations of MGPU systems to understand the application behavior and the evolution of GPU architectures. We started with the evaluation of the performance bottlenecks in the Kepler, Titan and DGX-1 MGPU systems for training DNN workloads. We used MNIST, Cifar10, Imagenet and PennTreeBank datasets, and MLP, Lenet, Alexnet, Googlenet, Resnet and LSTM networks implemented using MXNet frameworks in our evaluation. We found that a number of workloads do scale linearly as the number of GPU is increased because of communication bottlenecks. We investigated various data transfer mechanisms (P2P memcopy, zerocopy and unified memory) using synthetic workloads, which mimic the communication pattern of DNN workloads, to understand the communication bottlenecks. Finally, we demonstrate a scaled down MGPU system with true shared memory (TSM) reduces the run time by more than 3×, on average, for all the three NVIDIA MGPU systems– Pascal, Titan Z and DGX-1. This result motivated the unification the physical memory space for

the heterogeneous systems consisting of CPUs and GPUs. We continued our effort to perform more in-depth performance analysis of MGPU systems for training DNNs using the contemporary NVIDIA DGX-1 system consisting of 8 NVIDIA V100 GPUs. In this work, we performed a comprehensive analysis to understand the computation and communication pattern of training DNN workloads on the Volta-based DGX-1 MGPU system and characterized data movement and memory usage of five DNN workloads (GoogLeNet, AlexNet, Inception-v3, ResNet and LeNet). We used the MXNet framework for training DNN workloads using the data parallelism approach. We compared NCCL library based communication with P2P memcopy based communication among the GPUs. Based on our evaluation, we found that the MGPU scalability heavily depends on the neural network architecture, batch size, and the GPU-to-GPU communication method. We concluded that workloads scale better with NCCL-based communication than P2P for 4 and 8 GPUs. NCCL introduced significant overhead for DNN training, especially when using 1 or 2 GPUs for training. We observed that GPU memory capacity puts significant limitation on scaling the size of DNN workloads. Furthermore, the lack of hardware-level coherence support on GPUs exacerbated the memory capacity issues and programming complexity. We also showed that remote data access degrades MGPU system performance significantly.

In the second part of this thesis, we proposed a novel MGPU system, where the CPU and GPUs physically share the main memory to overcome the NUMA effects introduced by remote memory accesses by GPUs. We refer to such a system as a multi-GPU system with true shared memory (MGPU-TSM). This MGPU-TSM system eliminated both the programmer's burden of unnecessary data replication and the expensive remote memory accesses. We showed that our proposed MGPU-TSM system with 4 GPUs is capable achieving, on average, 3.8$\times$ better performance than existing MGPU system with 4 GPUs for standard benchmarks.

To ensure seamless sharing of data across and within multiple GPUs, we proposed

MGCC, a novel timestamp-based hardware coherence protocol in the final part of this thesis. To support the MGCC protocol on MGPU-TSM system, we designed and implemented a number of novel hardware features including unified memory controllers and a request tracker logic to avoid coherence issues, and timestamp storage units (TSUs) to store and access timestamps without adding any performance overhead. For standard benchmarks, an MGPU-TSM system (that has 4 GPUs and uses MGCC) performed on average, 3.8× better with relaxed consistency and 3.0× better with sequential consistency, than the non-coherent conventional MGPU system with same number of GPUs. In addition, compared to a coherent MGPU system using the state-of-the-art HMG coherence protocol, an MGPU system that uses MGCC had 2.4× higher performance. Our scalability study showed that our MGCC coherence protocol scales well in terms of GPU count. We developed synthetic benchmarks that leverage data sharing to examine the impact of our MGCC protocol on performance. For the worst case scenario in our synthetic benchmarks, the proposed MGPU-TSM with MGCC suffered up to a 46.1% performance overhead.

5.2 Future Directions

5.2.1 Workload Characterization and Benchmarking

GPU and MGPU systems are increasingly being used for accelerating applications from both current and emerging domains. In the emerging domains, graph neural network-based applications (Liu et al., 2020) and Homomorphic Encryption-based applications (Al Badawi et al., 2020) have recently been leveraging GPU and MGPU acceleration. An understanding of the behavior of the applications in these emerging domains is necessary to gain insight into the architectural features that needs to be integrated into the next-generation MGPU systems as well as software features necessary to provide optimized performance. In our work, we have created stress-test benchmarks that can be used to characterize the behavior of these emerging applications.

To gain in-depth insights into the performance issues of MGPU systems, we need benchmarks that can capture a wide range of application behavior. In our work, we continually faced challenges in finding proper benchmarks to understand MGPU system performance. Hence, we created synthetic benchmarks. An interesting research direction is to create an extensive MGPU benchmark suite that stress tests different architectural features in an MGPU system. Recently, few attempts such as Tartan (Li et al., 2018), (Li et al., 2019), MLPerf (Mattson et al., 2020), etc. have been made to standardize the GPU and MGPU benchmarking and performance analysis. We need more efforts in this area as MGPU systems and its applications are continuously evolving.

5.2.2 MGPU System Design

To build efficient MGPU systems with TSM, a number of design considerations need to be taken into account and future work should focus on solving the design challenges reported in the subsequent sections.

Thermal Considerations

High-end MGPU systems consume a high amount of power and produce a significant amount of heat. Hence, these systems can easily lead to the dark silicon problem (Henkel et al., 2015). We, therefore, need to perform in-depth thermal analysis of these systems to understand the thermal feasibility of the proposed MGPU-TSM system. Our preliminary evaluation suggests that 2.5D chip integration technology can be leverage to develop thermally feasible MGPU systems (Eris et al., 2018), (Coskun et al., 2018), (Ma et al., 2021).

Future research should look into emerging technologies such as 2.5D chip integration technology with both passive and active interposers to build large MGPU systems, more efficient routing and placement of components on the interposer, in-depth thermal analysis of the MGPU systems and efficient means of cooling for such systems.

Interconnection Network

In our work, we considered a simple switch for connectivity between the L2 and main memory. This switch provides point-to-point connectivity between the GPUs and the main memory. Finding the optimal network topology for the MGPU-TSM system is an open research problem. Future work should focus on exploring a variety of symmetric network topologies including mesh, crossbar, clos, ring, butterfly, etc. as well as asymmetric topologies (Ziabari et al., 2015a). These network topologies will influence the available bandwidth for the main memory, which in turn will impact overall system performance. As GPU count increases, we will need to use multiple interposers. Consequently, this would require the design of intra-interposer and inter-interposer communication methods. A promising technology for inter-interposer communication is the photonic link technology that provides high-bandwidth low-latency communications (Chittamuru et al., 2017), (Abellán et al., 2016), (Zhang et al., 2016), (Ziabari et al., 2015b), (Koka et al., 2010), (Joshi et al., 2009).

CPU-GPU Interaction and Consistency

In this work, we did not take into account the CPU and GPU interactions. Today's heterogeneous system architectures already support a CPU-GPU collaborative computing environment (Mittal and Vetter, 2015). As part of future work we need to develop hardware and software features that provide support for CPU-MGPU-TSM collaborative computing environment. Our MGCC policy only supports coherence between GPUs. We will need to develop coherence policies that are suitable for the CPU-MGPU system. Similarly, our MGPU-TSM system with MGCC protocol supports sequential consistency and relaxed consistency. One possible future research direction is not only to explore efficient memory consistency models, but also to build tools to easily verify MGPU consistency models.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Abellán, J. L., Coskun, A. K., Gu, A., Jin, W., Joshi, A., Kahng, A. B., Klamkin, J., Morales, C., Recchio, J., Srinivas, V., et al. (2016). Adaptive tuning of photonic devices in a photonic noc through dynamic workload allocation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(5):801–814.

Adve, S. V. and Gharachorloo, K. (1996). Shared memory consistency models: A tutorial. *computer*, 29(12):66–76.

Al Badawi, A., Veeravalli, B., Lin, J., Xiao, N., Kazuaki, M., and Mi, A. K. M. (2020). Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):379–391.

Al-Rfou, R., Alain, G., Almahairi, A., Angermüller, C., Bahdanau, D., and et al., N. B. (2016). Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688.

Alsheikh, M. A., Lin, S., Niyato, D., and Tan, H.-P. (2014). Machine learning in wireless sensor networks: Algorithms, strategies, and applications. *IEEE Communications Surveys & Tutorials*, 16(4):1996–2018.

Alsop, J., Orr, M. S., Beckmann, B. M., and Wood, D. A. (2016). Lazy release consistency for gpus. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE.

Alsop, J., Sinclair, M., and Adve, S. (2018). Spandex: a flexible interface for efficient heterogeneous coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 261–274. IEEE.

Amazon, E. (2015). Amazon web services. *Available in: http://aws. amazon. com/es/ec2/(November 2012)*.

AMD (2020). Amd instinct™ mi100 accelerator.

AMD (Accessed: January 4, 2021). Rocm communication collectives library. *URL: https://github.com/ROCmSoftwarePlatform/rccl*.

Arunkumar, A., Bolotin, E., Cho, B., Milic, U., Ebrahimi, E., Villa, O., Jaleel, A., Wu, C.-J., and Nellans, D. (2017). Mcm-gpu: Multi-chip-module gpus for continued performance scalability. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 320–332. ACM.

Arunkumar, A., Bolotin, E., Nellans, D., and Wu, C.-J. (2019). Understanding the future of energy efficiency in multi-module gpus. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 519–532. IEEE.

Bahrampour, S., Ramakrishnan, N., Schott, L., and Shah, M. (2015). Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*.

Baruah, T., Sun, Y., Dinçer, A. T., Mojumder, S. A., Abellán, J. L., Ukidave, Y., Joshi, A., Rubin, N., Kim, J., and Kaeli, D. (2020). Griffin: Hardware-software support for efficient page migration in multi-gpu systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 596–609. IEEE.

Blythe, D. (2008). Rise of the graphics processor. *Proceedings of the IEEE*, 96(5):761–778.

Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. (2003). Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM transactions on graphics (TOG)*, 22(3):917–924.

Booth, K. S., Cowan, W. B., and Forsey, D. R. (1985). Multitasking support in a graphics workstation. In *First IEEE International Conference on Computer Workstations*, pages 82–89.

Boroumand, A., Ghose, S., Patel, M., Hassan, H., Lucia, B., Ausavarungnirun, R., Hsieh, K., Hajinazar, N., Malladi, K. T., and Zheng, H. (2019). Conda: efficient cache coherence support for near-data accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 629–642.

Buono, D., Artico, F., Checconi, F., Choi, J. W., Que, X., and Schneidenbach, L. (2017). Data analytics with nvlink: An spmv case study. In *Proceedings of the Computing Frontiers Conference*, pages 89–96.

Cates, J. E., Lefohn, A. E., and Whitaker, R. T. (2004). Gist: an interactive, gpu-based level set segmentation tool for 3d medical images. *Medical image analysis*, 8(3):217–231.

Che, S., Beckmann, B. M., Reinhardt, S. K., and Skadron, K. (2013). Pannotia: Understanding irregular gpgpu graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195. IEEE.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015a). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015b). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chen, X.-W. and Lin, X. (2014). Big data deep learning: challenges and perspectives. *IEEE access*, 2:514–525.

Chien, S., Peng, I., and Markidis, S. (2019). Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57. IEEE.

Chittamuru, S. V. R., Desai, S., and Pasricha, S. (2017). Swiftnoc: a reconfigurable silicon-photonic network with multicast-enabled channel sharing for multicore architectures. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(4):1–27.

Choquette, J. and Gandhi, W. (2020). Nvidia a100 gpu: Performance & innovation for gpu computing. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–43. IEEE Computer Society.

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. (2013). Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345.

Cochet, K. R. P., McCleary, R., Rogoff, R., and Roy, R. (2014). Lithography challenges for 2.5d interposer manufacturing. In *2014 IEEE 64th Electronic Components and Technology Conference (ECTC)*, pages 523–527.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376.

Copeland, M., Soh, J., Puca, A., Manning, M., and Gollob, D. (2015). *Microsoft Azure*. Springer.

Coskun, A., Eris, F., Joshi, A., Kahng, A. B., Ma, Y., and Srinivas, V. (2018). A cross-layer methodology for design and optimization of networks in 2.5 d systems. In *Proceedings of the International Conference on Computer-Aided Design*, page 101. ACM.

Crayton, M., Foss, M., Lindquist, D., and Tostado, K. (2004). Graphical processing units: An examination of use and function. *Proceedings of CA 2004 the Fall 2004 ENGR 3410 Computer Architecture Class*, page 17.

Crow, T. S. (2004). Evolution of the graphical processing unit. *A professional paper submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science, University of Nevada, Reno. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.368rep=rep1type=pdf*.

Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. (2010). The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231.

Deschizeaux, B. and Blanc, J.-Y. (2007). Imaging earth's subsurface using cuda. *GPU Gems*, 3:831–850.

Dietrich, S. (1999). Vertex blending under directx 7 for the geforce 256. *Technical Presentations*, 99.

Dong, S. and Kaeli, D. (2017). Dnnmark: A deep neural network benchmark suite for gpus. In *Proceedings of the General Purpose GPUs*, pages 63–72. ACM.

Eris, F., Joshi, A., Kahng, A. B., Ma, Y., Mojumder, S., and Zhang, T. (2018). Leveraging thermally-aware chiplet organization in 2.5 d systems to reclaim dark silicon. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1441–1446. IEEE.

Fan, Y., Winkel, C., Kulkarni, D., and Tian, W. (2018). Analytical design methodology for liquid based cooling solution for high tdp cpus. In *2018 17th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, pages 582–586. IEEE.

Foley, D. and Danskin, J. (2017). Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17.

Fung, J., Tang, F., and Mann, S. (2002). Mediated reality using computer graphics hardware for computer vision. In *Proceedings. Sixth International Symposium on Wearable Computers,*, pages 83–89. IEEE.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Govindaraju, N. K., Larsen, S., Gray, J., and Manocha, D. (2006). A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 89–es.

Govindaraju, N. K., Lloyd, B., Wang, W., Lin, M., and Manocha, D. (2005). Fast computation of database operations using graphics processors. In *ACM SIGGRAPH 2005 Courses*, pages 206–es.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, speech and signal processing (ICASSP)*, pages 6645–6649. IEEE.

Gullbrand, J., Luckeroth, M. J., Sprenger, M. E., and Winkel, C. (2019). Liquid cooling of compute system. *Journal of Electronic Packaging*, 141(1):010802.

Hagen, T. R., Lie, K.-A., and Natvig, J. R. (2006). Solving the euler equations on graphics processing units. In *International Conference on Computational Science*, pages 220–227. Springer.

Harris, M. J., Coombe, G., Scheuermann, T., and Lastra, A. (2002). Physically-based visual simulation on graphics hardware. In *Graphics Hardware*, volume 2002, pages 1–10.

Hechtman, B. A., Che, S., Hower, D. R., Tian, Y., Beckmann, B. M., Hill, M. D., Reinhardt, S. K., and Wood, D. A. (2014). Quickrelease: A throughput-oriented approach to release consistency on gpus. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 189–200. IEEE.

Hechtman, B. A. and Sorin, D. J. (2013). Evaluating cache coherent shared virtual memory for heterogeneous multicore chips. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 118–119. IEEE.

Henkel, J., Khdr, H., Pagani, S., and Shafique, M. (2015). New trends in dark silicon. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (Dac)*, pages 1–6. IEEE.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hoff III, K. E., Zaferakis, A., Lin, M., and Manocha, D. (2001). Fast and simple 2d geometric proximity queries using graphics hardware. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 145–148.

Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., and Stan, M. R. (2006). Hotspot: A compact thermal modeling methodology for early-stage vlsi design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513.

James, J. M. (1987). The third generation of pc graphics controllers. *IEEE computer graphics and applications*, 7(10):24–27.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA. ACM.

Jordan, M. I. and Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260.

Joshi, A., Batten, C., Kwon, Y.-J., Beamer, S., Shamim, I., Asanovic, K., and Stojanovic, V. (2009). Silicon-photonic clos networks for global on-chip communication. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 124–133. IEEE.

Khalaj, A. H. and Halgamuge, S. K. (2017). A review on efficient thermal management of air-and liquid-cooled data centers: From chip to the cooling system. *Applied energy*, 205:1165–1188.

Kim, G., Lee, M., Jeong, J., and Kim, J. (2014). Multi-gpu system design with memory networks. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 484–495. IEEE.

Kim, H., Nam, H., Jung, W., and Lee, J. (2017). Performance analysis of cnn frameworks for gpus. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 55–64. IEEE.

Kim, T. and Lin, M. C. (2003). Visual simulation of ice crystal growth. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 86–97.

Kirk, D. et al. (2007). Nvidia cuda software and gpu parallel computing architecture. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, volume 7, pages 103–104.

Koka, P., McCracken, M. O., Schwetman, H., Zheng, X., Ho, R., and Krishnamoorthy, A. V. (2010). Silicon-photonic network architectures for scalable, power-efficient multi-chip systems. *ACM SIGARCH Computer Architecture News*, 38(3):117–128.

Kourou, K., Exarchos, T. P., Exarchos, K. P., Karamouzis, M. V., and Fotiadis, D. I. (2015). Machine learning applications in cancer prognosis and prediction. *Computational and structural biotechnology journal*, 13:8–17.

Krishnan, S. and Gonzalez, J. L. U. (2015). *Building your next big thing with google cloud platform: A guide for developers and enterprise architects*. Springer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Kumar, S., Shriraman, A., and Vedula, N. (2015). Fusion: design tradeoffs in coherent cache hierarchies for accelerators. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 733–745.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9):690–691.

Landaverde, R., Zhang, T., Coskun, A. K., and Herbordt, M. (2014). An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Levinthal, D. (2009). Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18.

Li, A., Song, S. L., Chen, J., Li, J., Liu, X., Tallent, N. R., and Barker, K. J. (2019). Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110.

Li, A., Song, S. L., Chen, J., Liu, X., Tallent, N., and Barker, K. (2018). Tartan: evaluating modern gpu interconnect via a multi-gpu benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–202. IEEE.

Li, J.-M., Wang, X.-J., He, R.-S., and Chi, Z.-X. (2007). An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, pages 855–862. IEEE.

Li, X., Zhang, G., Huang, H. H., Wang, Z., and Zheng, W. (2016). Performance analysis of gpu-based convolutional neural networks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 67–76. IEEE.

Libbrecht, M. W. and Noble, W. S. (2015). Machine learning in genetics and genomics. *Nature Reviews. Genetics*, 16(6):321.

Lin, J. C.-R. and Hall, C. (2007). Multiple oil and gas volumetric data visualization with gpu programming. In *Visualization and Data Analysis 2007*, volume 6495, page 64950U. International Society for Optics and Photonics.

Liu, H., Lu, S., Chen, X., and He, B. (2020). G3: when graph neural networks meet parallel graph processing systems on gpus. *Proceedings of the VLDB Endowment*, 13(12):2813–2816.

Ma, Y., Delshadtehrani, L., Demirkiran, C., Abellán, J. L., and Joshi, A. (2021). Tap-2.5 d: A thermally-aware chiplet placement methodology for 2.5 d systems. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). (To appear)*.

Macri, J. (2015). Amd's next generation gpu and high bandwidth memory architecture: Fury. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–26. IEEE.

Manocha, D. (2003). Interactive geometric and scientific computations using graphics hardware. *SIGGRAPH Course Notes*, 11.

Markidis, S., Der Chien, S. W., Laure, E., Peng, I. B., and Vetter, J. S. (2018). Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE.

Mattson, P., Reddi, V. J., Cheng, C., Coleman, C., Diamos, G., Kanter, D., Micikevicius, P., Patterson, D., Schmuelling, G., Tang, H., et al. (2020). Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16.

McClanahan, C. (2010). History and evolution of gpu architecture. *A Paper Survey*, 9. *http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf*.

Milic, U., Villa, O., Bolotin, E., Arunkumar, A., Ebrahimi, E., Jaleel, A., Ramirez, A., and Nellans, D. (2017). Beyond the socket: Numa-aware gpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–135. ACM.

Mišić, M. J., Đurđević, Đ. M., and Tomašević, M. V. (2012). Evolution and trends in gpu computing. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 289–294. IEEE.

Mittal, S. and Vetter, J. S. (2015). A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35.

Mosegaard, J. and Sorensen, T. S. (2005). Gpu accelerated surgical simulators for complex morphology. In *IEEE Proceedings. VR 2005. Virtual Reality, 2005.*, pages 147–153. IEEE.

Munshi, A. (2009). The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE.

Nagarajan, V., Sorin, D. J., Hill, M. D., and Wood, D. A. (2020). A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 15(1):1–294.

NCCL (2018). Nvidia collective communications library (nccl). *https://developer.NVIDIA.com/nccl*.

Negrut, D., Serban, R., Li, A., and Seidl, A. (2014). Unified memory in cuda 6.0. a brief overview of related data access and transfer issues. *SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09. https://sbel.wisc.edu/wp-content/uploads/sites/56-9/2018/05/TR-2014-09.pdf*.

NVIDIA (2016). Nvidia dgx-1 with tesla v100 system architecture. *https://pdfs.semanticscholar.org/2e4e/fbcd8f52446b276129e1272512b916ddf093.pdf*.

NVIDIA (2017). Nvidia tesla v100 gpu architecture. *https://computing.llnl.gov/tutorials/sierra/volta-architecture-whitepaper.pdf*.

NVIDIA (2018). Nvidia cudnn. *https://developer.NVIDIA.com/cudnn*.

NVIDIA (2018). NVIDIA Unified Memory. *http://on-demand.gputechconf.com/gtc/-2018/presentation/s8430-everything-you- -need-to-know-about-unified-memory.pdf* .

NVIDIA (2018a). Ontap ai–nvidia dgx-2 pod with netapp aff a800. *https://resources.NVIDIA.com/en-us-netapp/ontap-ai-dgx-2-pod-with-netapp-whitepaper*.

NVIDIA (2018b). Profiler user's guide. *https://docs.NVIDIA.com/cuda/profiler-users-guide/index.html*.

NVIDIA, C. (2008). Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31. *https://developer.NVIDIA.com/cublas*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037.

Pawlowski, J. T. (2011). Hybrid memory cube (hmc). In *2011 IEEE Hot chips 23 symposium (HCS)*, pages 1–24. IEEE.

Plakal, M., Sorin, D. J., Condon, A. E., and Hill, M. D. (1998). Lamport clocks: verifying a directory cache-coherence protocol. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 67–76.

Pouchet, L.-N. (2012). Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*.

Power, J., Basu, A., Gu, J., Puthoor, S., Beckmann, B. M., Hill, M. D., Reinhardt, S. K., and Wood, D. A. (2013). Heterogeneous system coherence for integrated cpu-gpu systems. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 457–467. IEEE.

Qian, X., Ahn, W., and Torrellas, J. (2010). Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 447–458. IEEE.

Ren, X., Lustig, D., Bolotin, E., Jaleel, A., Villa, O., and Nellans, D. (2020). Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems. In *2020 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE.

Saikia, M. J. and Kanhirodan, R. (2014). High performance single and multi-gpu acceleration for diffuse optical tomography. In *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, pages 1320–1323. IEEE.

Sainath, T. N., r. Mohamed, A., Kingsbury, B., and Ramabhadran, B. (2013). Deep convolutional neural networks for lvcsr. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8614–8618.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

Shi, J., Yang, R., Jin, T., Xiao, X., and Yang, Y. (2019). Realtime top-k personalized pagerank over large graphs on gpus. *Proceedings of the VLDB Endowment*, 13(1):15–28.

Shi, S. and Chu, X. (2017). Performance modeling and evaluation of distributed deep learning frameworks on gpus. *arXiv preprint arXiv:1711.05979*.

Shi, S., Wang, Q., Xu, P., and Chu, X. (2016). Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104.

Sinclair, M. D., Alsop, J., and Adve, S. V. (2015). Efficient gpu synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 647–659.

Singh, A., Aga, S., and Narayanasamy, S. (2015). Efficiently enforcing strong memory ordering in gpus. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 699–712.

Singh, G. and Sachan, M. (2014). Multi-layer perceptron (mlp) neural network technique for offline handwritten gurmukhi character recognition. In *2014 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–5.

Singh, I., Shriraman, A., Fung, W. W., O'Connor, M., and Aamodt, T. M. (2013). Cache coherence for gpu architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590. IEEE.

Smith, S. L., Kindermans, P.-J., and Le, Q. V. (2017). Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*.

Strzodka, R., Droske, M., and Rumpf, M. (2003). Fast image registration in directx9 graphics hardware. *Journal of Medical Informatics & Technologies*, 6.

Sun, Y., Baruah, T., Mojumder, S. A., Dong, S., Gong, X., Treadway, S., Bao, Y., Hance, S., McCardwell, C., Zhao, V., Barclay, H., Ziabari, A. K., Chen, Z., Ubal, R., Abellán, J. L., Kim, J., Joshi, A., and Kaeli, D. (2019). Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 197–209, New York, NY, USA. ACM.

Sun, Y., Gong, X., Ziabari, A. K., Yu, L., Li, X., Mukherjee, S., McCardwell, C., Villegas, A., and Kaeli, D. (2016). Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. (2017). Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint arXiv:1703.09039*.

Tabbakh, A. (2018). *Efficient Memory Coherence and Consistency Support for Enabling Data Sharing in GPUs*. PhD thesis, University of Southern California.

Tabbakh, A., Qian, X., and Annavaram, M. (2018). G-tsc: Timestamp based coherence for gpus. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 403–415. IEEE.

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5.

Tran, J., Jordan, D., and Luebke, D. (2004). New challenges for cellular automata simulation on the gpu. *SIGGRAPH, Los Angeles. ACM. Poster. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.9597rep=rep1type=pdf*.

Valero-Lara, P. (2014). Multi-gpu acceleration of dartel (early detection of alzheimer). In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 346–354. IEEE.

Viola, I., Kanitsar, A., and Groller, M. E. (2003). *Hardware-based nonlinear filtering and segmentation using high-level shading languages*. IEEE.

Wang, P., Zhang, L., Li, C., and Guo, M. (2019). Excavating the potential of gpu for accelerating graph traversal. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 221–230. IEEE.

Woetzel, J. and Koch, R. (2004). Multi-camera real-time depth estimation with discontinuity handling on pc graphics hardware. In *2004 International Conference on Pattern Recognition (ICPR) (1)*, pages 741–744.

Xu, Q., Jeon, H., and Annavaram, M. (2014). Graph processing on gpus: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 140–149. IEEE.

Yadan, O., Adams, K., Taigman, Y., and Ranzato, M. (2013). Multi-gpu training of convnets. *arXiv preprint arXiv:1312.5853*.

Yamazaki, I., Dong, T., Solcà, R., Tomov, S., Dongarra, J., and Schulthess, T. (2014). Tridiagonalization of a dense symmetric matrix on multiple gpus and its application to symmetric eigenvalue problems. *Concurrency and computation: Practice and Experience*, 26(16):2652–2666.

You, Y., Zhang, Z., Hsieh, C.-J., Demmel, J., and Keutzer, K. (2017). 100-epoch imagenet training with alexnet in 24 minutes. *ArXiv e-prints*.

Young, V., Jaleel, A., Bolotin, E., Ebrahimi, E., Nellans, D., and Villa, O. (2018). Combining hw/sw mechanisms to improve numa performance of multi-gpu systems. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 339–351. IEEE.

Yu, D., Eversole, A., Seltzer, M., Yao, K., Huang, Z., Guenter, B., Kuchaiev, O., Zhang, Y., Seide, F., Wang, H., et al. (2014). An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*.

Zeller, C. (2005). Cloth simulation on the gpu. In *ACM SIGGRAPH 2005 Sketches*, pages 39–es.

Zhang, C., Zhang, S., Peters, J. D., and Bowers, J. E. (2016). $8\times$ $8\times$ 40 gbps fully integrated silicon photonic network on chip. *Optica*, 3(7):785–786.

Zhang, C.-L., Xu, Y.-P., Xu, Z.-J., He, J., Wang, J., and Adu, J.-H. (2018). A fuzzy neural network based dynamic data allocation model on heterogeneous multi-gpus for large-scale computations. *International Journal of Automation and Computing*, 15(2):181–193.

Zhang, H., Hu, Z., Wei, J., Xie, P., Kim, G., Ho, Q., and Xing, E. (2015). Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216*.

Zheng, B. and Pekhimenko, G. (1997). Ecornn: Efficient computing of lstm rnn on gpus. *Memory*, 9:1735–1780.

Zhu, Y., Wang, Q., Li, M., Jiang, M., and Zhang, P. (2019). Image reconstruction by mumford–shah regularization for low-dose ct with multi-gpu acceleration. *Physics in Medicine & Biology*, 64(15):155017.

Zhu, Y.-L., Pan, D., Li, Z.-W., Liu, H., Qian, H.-J., Zhao, Y., Lu, Z.-Y., and Sun, Z.-Y. (2018). Employing multi-gpu power for molecular dynamics simulation: an extension of galamost. *Molecular Physics*, 116(7-8):1065–1077.

Ziabari, A. K., Abellán, J. L., Ma, Y., Joshi, A., and Kaeli, D. (2015a). Asymmetric noc architectures for gpu systems. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, pages 1–8.

Ziabari, A. K. K., Abellán, J. L., Ubal, R., Chen, C., Joshi, A., and Kaeli, D. (2015b). Leveraging silicon-photonic noc for designing scalable gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 273–282.

CURRICULUM VITAE