CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Improving Computation–Communication Overlap in Multi-GPU Machine Learning: A Resource-Centric Approach

Using Green Contexts and Tensor Memory Accelerator for Resource-Aware Multi-GPU Communication

Master's thesis in Computer science and engineering

Keyvan Dadashzadeh

Yuehong Zhou

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Master's thesis 2026

Improving Computation–Communication Overlap in Multi-GPU Machine Learning: A Resource-Centric Approach

Using Green Contexts and Tensor Memory Accelerator for Resource-Aware Multi-GPU Communication

Keyvan Dadashzadeh

Yuehong Zhou

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Improving Computation–Communication Overlap in Multi-GPU Machine Learning: A Resource-Centric Approach
Using Green Contexts and Tensor Memory Accelerator for Resource-Aware Multi-GPU Communication
Keyvan Dadashzadeh, Yuehong Zhou

Supervisor: Minyu Cui, Computer and Network Systems
Examiner: Miquel Pericas, Computer and Network Systems

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2026

Improving Computation–Communication Overlap in Multi-GPU Machine Learning: A Resource-Centric Approach
Using Green Contexts and Tensor Memory Accelerator for Resource-Aware Multi-GPU Communication
Keyvan Dadashzadeh
Yuehong Zhou
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The rapid growth of deep learning (DL) model size and parameter count has made single-GPU training increasingly impractical due to limited memory and computational capacity. Consequently, large-scale DL models are typically across multiple GPUs. However, multi-GPU execution introduces additional inter-GPU communication overhead, which can become a major performance bottleneck. One common approach to mitigate this overhead is to overlap communication with computation. In such approaches, partial results produced during computation are communicated immediately, while the remaining computation continues in parallel. Since both computation and communication consume GPU resources, efficient overlap requires careful resource allocation. This raises an important question: how should GPU resources be allocated between the computation and communication paths so that overlap can be performed efficiently?

This thesis addresses this question by investigating GPU resource utilization during computation-communication overlap, with a particular focus on the GPU's main execution resources, Streaming Multiprocessors (SMs). Building on prior work, FlashOverlap, the thesis proposes two methods, GCOverlap and Oh Overlap. GCOverlap uses CUDA Green Contexts to partition SMs between computation and communication, reducing interference between the two execution paths. Oh Overlap centers on the communication side by using the Tensor Memory Accelerator (TMA) to reduce the SM resources needed for collective communication.

The evaluation shows that GCOverlap can improve operator-level performance for several GEMM+AllReduce and GEMM+ReduceScatter workloads by isolating computation and communication on separate SM partitions. The results also show that Oh Overlap achieves higher bandwidth than its NCCL counterpart for large messages while using fewer SM resources. In overlapping execution, Oh Overlap achieves speedups between 1.1× and 1.4× over the non-overlap baseline, outperforming both the baseline and the original FlashOverlap design in the tested workloads.

Overall, this thesis demonstrates that computation–communication overlap is not only a scheduling problem, but also a GPU resource-management problem. Reducing contention and lowering the resource cost of communication can improve overlap efficiency and provide a useful direction for future multi-GPU training systems.

Keywords: GPU, distributed training, compute-communication overlap, collective communication, TMA, resource partition.

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Minyu, for her guidance, feedback, and support throughout this thesis project. We would also like to thank our examiner, Miquel, for the time and effort spent evaluating this work.

On a personal note, I would like to express my deepest gratitude to my mother for her unconditional love, patience, and support. Her encouragement has been an important source of strength throughout my studies and during the completion of this thesis.

Yuehong Zhou

I am deeply grateful to my father and mother for their constant support and encouragement during the past six months, despite the difficult circumstances they faced in my home country. Without their support, this thesis would not have been possible.

Keyvan Dadashzadeh

Gothenburg, June 2026

Declaration of AI-assisted Tools

AI-assisted writing and visualization tools, such as ChatGPT and Gemini, were used during the preparation of this thesis to support grammar checking, language editing, clarity improvements, and the generation or refinement of some illustrative figures. These tools were used only to improve wording, sentence structure, readability, and the clarity of visual explanations.

Platforms Used for Experimentation

All experiments presented in this thesis were carried out on two computing platforms: Alvis and Vera. The use of Alvis was supported by project allocation `NAISS 2026/4-335` and the use of Vera was supported by project allocation `C3SE 2026/1-12`. Both allocations were granted by the National Academic Infrastructure for Supercomputing in Sweden (NAISS).

Contents

List of Figures

List of Tables

1

Introduction

Efficient distributed GPU execution is a central challenge in modern deep learning training. As models and workloads grow, performance depends on not only raw computational throughput, but also on how effectively computation and communication are coordinated across GPUs.

This thesis studies computation-communication overlap as a way to mitigate multi-GPU communication overhead. It focuses on how GPU execution resources are shared during overlap and investigates two approaches for improving overlap efficiency.

1.1 Problem Statement

Modern deep learning workloads are trained at increasingly large scales, with state-of-the-art (SoA) models now reaching the trillion-parameter regime [1], [2]. Under this trend, training such models on a single GPU is no longer practical in terms of both memory capacity and execution time. First, GPU memory is limited and often cannot store large model parameters, intermediate activations, and gradient information at the same time, especially during backpropagation. Second, even when memory is sufficient, the throughput of a single GPU is usually not enough to finish training within a reasonable time. As a result, single-GPU training is no longer adequate for modern large-scale models. Distributed training across multiple GPUs has become the standard practice [3], [4].

However, multi-GPU training introduces communication overhead between devices. Collective communication operations, such as all-reduce, all-gather, and all-to-all, are required to synchronize gradients, exchange activations, or redistribute data across devices [4] As model and batch sizes grow, the amount of data that must be communicated per iteration also increases. Consequently, the time spent in these collective operations can become comparable to, or even larger than, the local compute time. When this happens, communication turns into a key scalability bottleneck.

To reduce this overhead, modern systems often try to overlap communication with computation so that communication latency can be hidden under useful computation. Techniques such as multi-stream execution and gradient bucketing are widely used for this purpose [3], [5]. However, the achievable overlap is often limited in prac-

tice. One reason is that computation and communication tasks compete for shared GPU resources, including Streaming Multiprocessors (SMs), memory bandwidth, and execution pipelines. Another reason is that many existing training pipelines still use relatively coarse-grained scheduling, conservative dependency control, and large communication buckets. As a result, collective operations are often launched only after sizable compute kernels complete, which reduces the actual overlap that can be achieved in practice.

1.2 Motivation

Computation-communication overlap is widely adopted to reduce communication overhead in distributed training, but overlap does not make communication free. On modern GPUs, collective operations often require GPU execution resources to transfer data between devices. As a result, communication kernels may interfere with computation operations when both execute at the same time. If this interference is too much, the benefit of overlap can be limited, even when computation and communication are launched concurrently.

This motivates a resource-aware view of computation-communication overlap. Rather than treating overlap only as a scheduling problem, this thesis further studies how GPU resources are shared between the computation path and the communication path during concurrent execution. This thesis focuses on SM resources, since SMs are the GPU's primary execution unit and are used by both computation and collective communication kernels. The thesis studies the computation-communication overlap at the GEMM-plus-collective operator level. In transformer-based large language models (LLMs), the dominant computations are spent in large matrix multiplications, which consist of floating-point additions and multiplications [4]. To complete such heavy workloads within a reasonable amount of time, training is often parallelized across multiple GPUs, where different parts of the computation are distributed to different devices and executed in parallel [3]. Collective communication operations are essential for exchanging data across GPUs. This operator-level setting makes it possible to directly analyze the interaction between computation and communication without the additional complexity of a full training framework.

1.3 Purpose and Contributions

This project aims to investigate how computation-communication overlap in distributed machine learning workloads can be improved in multi-GPU systems at the *operator level*. Building on FlashOverlap [6] as the baseline overlap mechanism, the project focuses on improving GPU resource utilization during overlap rather than introducing a completely new overlap strategy. All evaluations are performed at the GEMM-plus-collective operator level.

The thesis explores two directions. The first is GCOverlap, which uses CUDA Green Contexts to partition SM resources into dedicated compute and communication subsets. The goal is to study whether explicit SM isolation can reduce resource con-

tention between GEMM kernels and collective communication kernels, and thereby improve overlap efficiency.

The second is Oh Overlap, which builds a collective communication library on top of newer GPU hardware features in Hopper architecture, in particular the Tensor Memory Accelerator (TMA). The goal is to investigate whether communication can be implemented with lower SM-side resource usage, so that more GPU execution resources remain available for computation during overlap.

In particular, the main objectives and contributions are as follows:

- Study whether SM-level resource partitioning via CUDA Green Contexts can improve compute-communication overlap efficiency;
- Design and evaluate a TMA-based collective communication backend that reduces the SM-side cost of intra-node communication.

1.4 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 presents the technical background needed to understand the thesis, including GPU architecture, the basics of matrix multiplication, and different systems and mechanisms for GPU communication. Chapter 3 discusses different categories of prior work that aim to improve GPU communication efficiency or propose new techniques for computation–communication overlap.

Chapter 4 describes the design and implementation of the two methods proposed in this thesis. Chapter 5 presents the experimental setup and evaluates the proposed methods. Finally, Chapter 6 concludes the thesis by discussing the main findings, limitations, and directions for future work.

2

Technical Background

This chapter introduces the background needed for the rest of the thesis. We first introduce the GPU architecture and explain how GPU programs are executed and what resources they use. Then, we discuss General Matrix Multiplication (GEMM) and the libraries commonly used to implement it efficiently on GPUs. After that, we describe communication between multiple GPUs within a single machine. Finally, we discuss GPU resource partitioning features that are relevant to this work.

2.1 GPU Architecture and Execution Resources

2.1.1 Streaming Multiprocessors

The fundamental execution unit of an NVIDIA GPU is the *Streaming Multiprocessor* (SM). A GPU contains multiple SMs, which are organized into larger hardware groups called *Graphics Processing Clusters* (GPCs). Each SM can execute many warps and thread blocks concurrently. Internally, an SM provides execution units, registers, shared memory, and scheduling logic for parallel kernel execution [7].

After a CUDA kernel is launched, its thread blocks are scheduled onto available SMs. A thread block executes on one SM, while different blocks from the same grid can run on different SMs in parallel.

SMs represent the main execution resources shared by GPU kernels. Computation kernels use SMs to perform arithmetic operations, such as the operations used in matrix multiplication. Communication-related kernels may also use SMs to move data, copy buffers, or prepare data for transfer to another GPU. Therefore, when multiple kernels execute at the same time, they may compete for SM resources.

2.1.2 GPU Memory Hierarchy

GPU kernels access data through a hierarchy of memory spaces with different latency, capacity, and visibility. In machine learning workloads, input data such as weights, activations, and intermediate results are usually stored in device *global memory* (GMEM). This memory is accessible to all SMs and threads and persists beyond a single kernel launch. Therefore, inputs and final outputs are commonly stored in GMEM so that they can be used by later kernels. Inside the GPU, data moves between device memory, the shared L2 cache, the per-SM L1/shared-memory

region, and the register files. This GPU-side memory organization is illustrated in Figure 2.1.

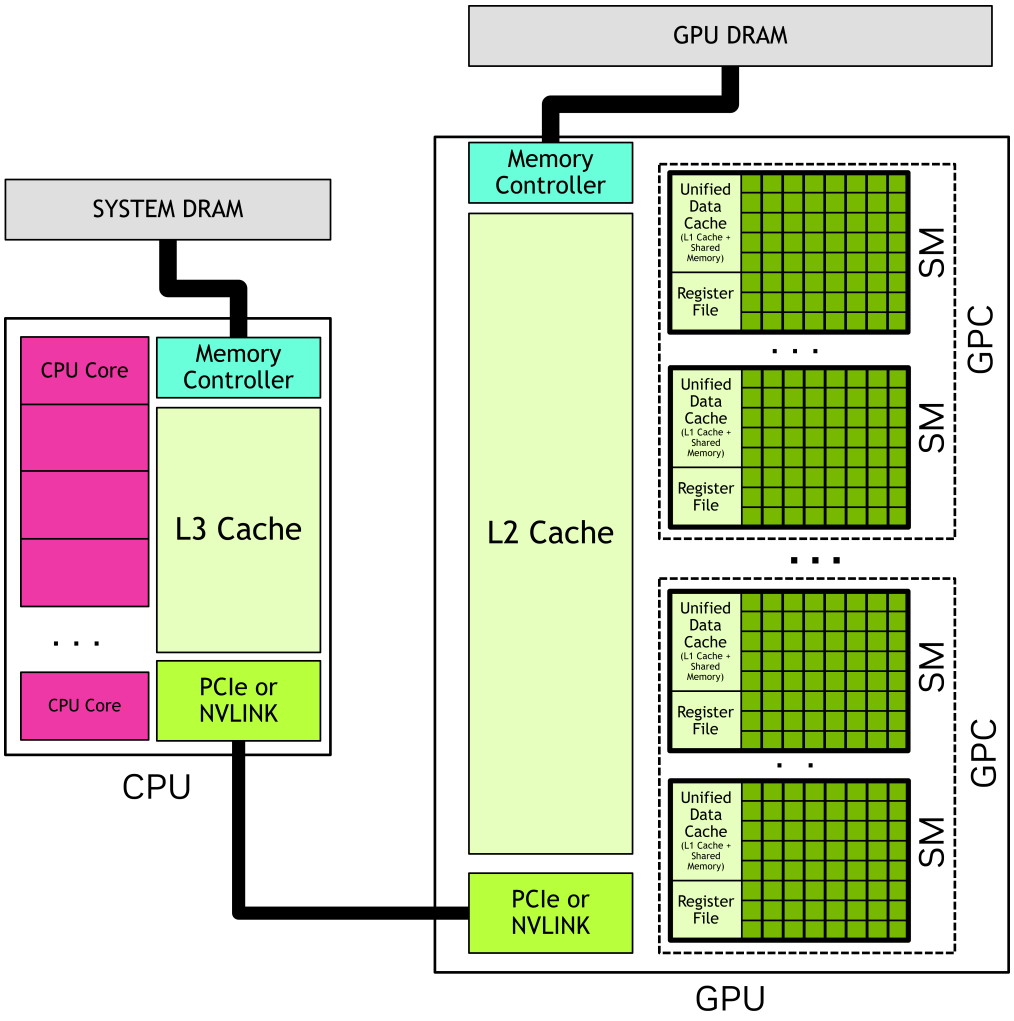

Figure 2.1: High-level GPU memory-system view, including device memory, L2 cache, unified L1/shared memory, and register files [7].

The GPU memory hierarchy follows a common trade-off: larger memory spaces are usually farther from computation and have higher latency, while smaller memory spaces are closer to computation and provide higher bandwidth. Figure 2.2 summarizes this trade-off for an H100 GPU, showing representative capacity, bandwidth, and access-latency differences across the hierarchy [8].

At the bottom of the hierarchy, GMEM provides the largest capacity and is used for large tensors, intermediate results, and final outputs. However, accessing GMEM is expensive compared to on-chip storage. The L2 cache is shared across the GPU and helps reduce repeated accesses to device memory. Moving closer to the SMs, NVIDIA GPUs provide an L1 cache and *shared memory* (SMEM). On recent architectures, SMEM is partitioned from a unified data cache, and the remaining portion serves as L1 cache [7].

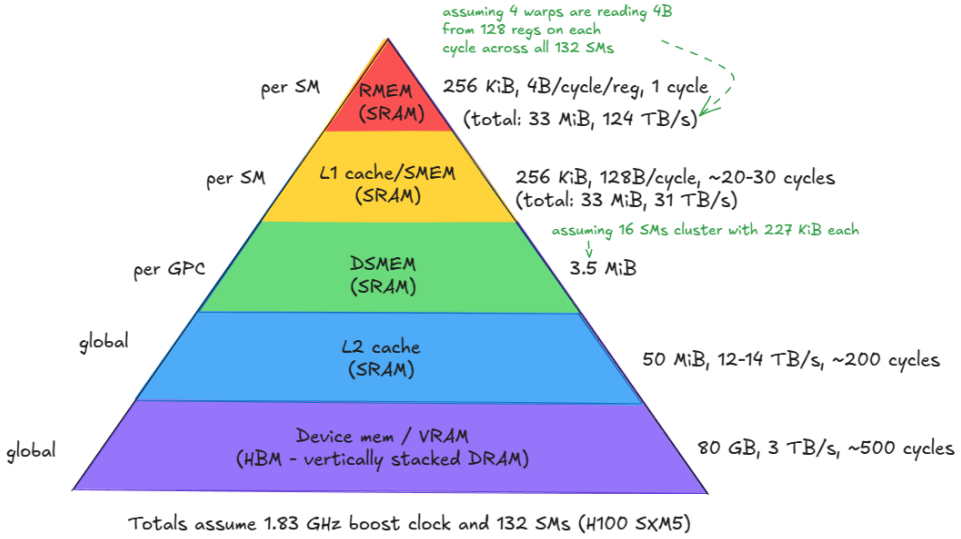

Figure 2.2: Simplified H100 GPU memory hierarchy from device memory to registers, including approximate capacity, bandwidth, and access latency [8].

Registers are the fastest storage and are private to each thread. They are typically used for temporary values, loop variables, and accumulator fragments. Shared memory, in contrast, is visible to all threads within the same thread block and has the same lifetime as the block. It is commonly used as a programmer-managed cache: a block loads a small region of data from global memory into shared memory, reuses it for computation, and then replaces it with the next region.

Starting with Hopper-generation GPUs, NVIDIA introduced distributed shared memory (DSM). In this model, thread blocks that belong to the same cluster are able to access each other's shared memory through mechanisms discussed later in this chapter. DSM should not be understood as a completely separate physical memory level like global memory or L2 cache. Rather, it extends how shared memory can be accessed among cooperating thread blocks on Hopper-generation GPUs [9].

2.1.3 Data Movement Mechanisms

Data movement is a central part of GPU performance. In many GPU kernels, arithmetic operations are only one part of the total execution time; data must also be moved between global memory, shared memory, registers, and, in multi-GPU systems, between different GPUs. Modern NVIDIA GPUs therefore provide several mechanisms for moving data, ranging from normal thread-level load/store instructions to specialized asynchronous copy mechanisms such as the Tensor Memory Accelerator (TMA) [7], [9].

2.1.3.1 Thread-Level Load and Store Instructions

The most basic data movement mechanism is performed through normal load and store instructions. In this case, individual GPU threads issue memory instructions to move data between memory spaces. For example, a thread can load data from global

memory into registers, store register values into shared memory, or write final results back to global memory. This form of movement is flexible because each thread can compute its own address and access arbitrary memory locations. However, it also consumes instruction issue bandwidth and execution resources from the SM [7].

In computation kernels, thread-level data movement is commonly used to load input values, move temporary values through registers, and store output values after computation. Registers are especially important because they hold temporary values and partial results close to the executing thread. However, moving data through registers means that the movement is directly tied to GPU instruction execution. Therefore, this mechanism competes with arithmetic instructions for scheduling and register resources.

2.1.3.2 Copy Engines

Another form of data movement is performed by *copy engines*. Copy engines are hardware units used for large memory transfers, such as host-to-device, device-to-host, or device-to-device copies. These transfers are usually launched through CUDA memory-copy operations rather than by individual threads inside a kernel. Therefore, copy engines are useful for coarse-grained data movement, where a large contiguous buffer needs to be transferred between memory regions [7].

Copy engines are different from in-kernel data movement mechanisms. They are suitable for moving large buffers outside the main body of a kernel, but they are not the mechanism used by a thread block to move intermediate data during kernel execution. Inside a kernel, data movement is usually performed either by thread-level load and store instructions or, on newer NVIDIA GPUs, by specialized mechanisms such as the Tensor Memory Accelerator (TMA), which is discussed later in this section.

2.1.3.3 Asynchronous Copy and Tensor Memory Accelerator

NVIDIA GPUs support asynchronous data movement inside kernels. Unlike normal load and store instructions, an asynchronous copy allows a thread to start a memory movement operation and continue with other work before the copy has completed. The copied data cannot be safely consumed until the corresponding completion mechanism has finished. This makes asynchronous copies useful for building in-kernel data movement pipelines, where data for a later step can be requested before it is needed [7], [10].

At the lower software level, these mechanisms are described in Parallel Thread Execution (PTX), NVIDIA's low-level virtual instruction set. PTX exposes asynchronous data movement in several forms. Earlier forms support asynchronous movement between global memory and shared memory, while newer bulk and tensor forms support larger and more structured transfers. The *Tensor Memory Accelerator* (TMA), introduced with Hopper-generation GPUs, is closely related to these newer bulk and tensor forms because it provides hardware support for moving tensor-shaped data efficiently [9], [10].

TMA is designed to support efficient movement of structured multidimensional data. Instead of having many threads manually calculate addresses and issue individual memory operations, a small number of threads can initiate a larger transfer described by a tensor map. The tensor map describes the layout of the data, including dimensions, strides, and other layout information. Once the operation is issued, address generation and data movement are handled by the hardware. This high-level difference between thread-based memory movement on A100 and TMA-based memory movement on H100 is illustrated in Figure 2.3 [7], [9].

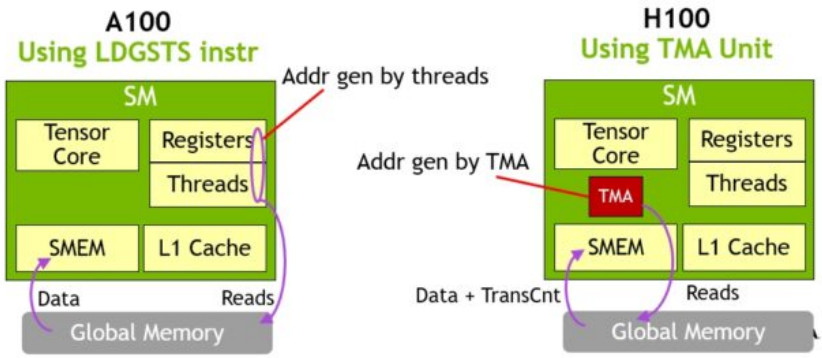

Figure 2.3: High-level comparison between thread-based data movement on A100 and TMA-based data movement on H100 [9].

TMA does not support arbitrary movement between all memory spaces. Instead, it supports specific source and destination combinations. Table 2.1 gives a compact view of the bulk TMA copy directions described in the CUDA documentation.

Table 2.1: Bulk TMA copy directions between memory spaces.

Source	**Destination**
Global memory	Thread-block shared memory
Global memory	Cluster shared memory
Thread-block shared memory	Global memory
Thread-block shared memory	Cluster shared memory

TMA supports both one-dimensional bulk transfers and multidimensional tensor transfers. One-dimensional bulk transfers operate on contiguous byte ranges and have alignment requirements. For example, the CUDA documentation specifies that global-memory addresses, shared-memory addresses, and the transfer size must satisfy 16-byte alignment requirements for one-dimensional bulk operations. Multidimensional tensor transfers extend this idea to structured tensor data. They support tensor ranks from 1D to 5D and use a tensor map to describe the multidimensional layout [7], [10].

Because TMA operations are asynchronous, correctness depends on explicit completion handling. A kernel can issue one or more TMA operations and continue executing independent work, but it must wait before consuming the destination data or reusing memory that may still be involved in an outstanding operation.

The completion mechanism depends on the direction of the transfer. For example, transfers from global memory into shared memory are commonly completed through a shared-memory barrier, while transfers from shared memory back to global memory are completed through a bulk async-group mechanism. Table 2.2 summarizes these completion mechanisms at a high level [7], [10].

Table 2.2: Completion mechanisms for common bulk TMA copy directions.

Source	Destination	Completion mechanism
Global memory	Thread-block shared memory	Shared-memory barrier
Global memory	Cluster shared memory	Shared-memory barrier
Thread-block shared memory	Global memory	Bulk async-group
Thread-block shared memory	Cluster shared memory	Shared-memory barrier

TMA is especially useful for computation kernels where the input data has a regular multidimensional layout. A small number of threads can issue the data movement operation, while other threads continue with independent work or wait until the data becomes available. This supports producer-consumer style kernels, where some threads are responsible for data movement and others are responsible for computation [7], [9].

2.1.3.4 Cluster-Level TMA and Distributed Shared Memory

TMA also supports data movement at the thread-block cluster level. In Hopper GPUs, thread blocks can be organized into clusters, and blocks in the same cluster can access *distributed shared memory* (DSM). This allows data to be moved into a cluster-level shared-memory space and then used by cooperating thread blocks. Conceptually, this avoids some cases where data would otherwise need to be exchanged through global memory. However, using DSM requires explicit synchronization so that one thread block does not read data before the producing block or data movement operation has completed. Figure 2.4 illustrates the difference between traditional global-memory-based exchange and Hopper cluster-level exchange through DSM [9], [10].

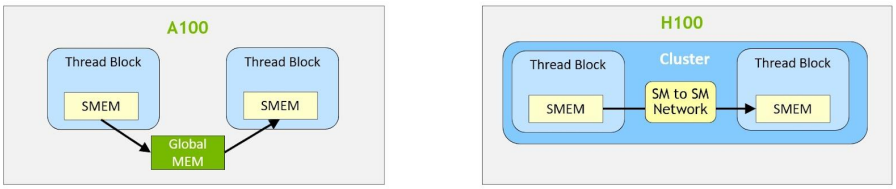

Figure 2.4: Comparison between traditional thread-block data exchange through global memory on A100 and cluster-level data exchange through distributed shared memory (DSM) on H100 [9].

2.1.3.5 TMA Reduction Operations

TMA can also combine data movement with reduction-style operations. In this mode, data is read from shared memory and used to update a destination memory

region through an elementwise reduction. Table 2.3 summarizes the main TMA reduction directions.

Table 2.3: Main TMA reduction directions and completion mechanisms.

Source	**Destination**	**Completion mechanism**
Thread-block shared memory	Global memory	Bulk async-group
Thread-block shared memory	Cluster shared memory	Memory-barrier based completion

Supported reduction operations include addition, minimum, maximum, increment, decrement, and bitwise operations, depending on the data type and the exact operation form. PTX specifies that tensor reduction operations are non-blocking and that each elementwise reduction has relaxed memory-ordering semantics. Therefore, reduction operations also require explicit completion handling before the program can safely depend on their results [10].

This capability is important because it allows part of a reduction to be expressed as a data movement operation rather than as a separate sequence of normal thread-level arithmetic instructions followed by stores. For communication-heavy workloads, this is attractive because reductions are common in collective operations. However, the mechanism is hardware-specific and must obey the supported data types, memory directions, tensor-map format, alignment requirements, and synchronization rules [7], [10].

2.2 CUDA Execution Model

Compute Unified Device Architecture (CUDA) is NVIDIA's programming model for writing GPU-accelerated applications. In CUDA, the CPU is referred to as the *host*, while the GPU is referred to as the *device*. The host launches functions called *kernels*, which execute on the GPU. CUDA exposes a hierarchy of parallel execution units, including grids, thread blocks, threads, warps, and, on newer GPUs, thread block clusters [7].

2.2.1 Kernels, Grids, Thread Blocks, and Warps

A CUDA *kernel* is executed by many GPU *threads*. When launching a kernel, the programmer specifies an execution configuration that defines the number of *thread blocks* and the number of threads per block. The full set of thread blocks launched by one kernel forms a *grid*, as shown in Figure 2.5. Each thread block is assigned to a Streaming Multiprocessor (SM) for execution, and different blocks from the same grid can run on different SMs in parallel [7].

Threads inside a block are further grouped into warps. A *warp* is the basic scheduling unit used by the SM. In NVIDIA GPUs, a warp contains 32 threads. The SM schedules warps and issues their instructions to the available execution units. This means that although CUDA exposes individual threads to the programmer, many low-level execution and performance properties are determined at warp granularity.

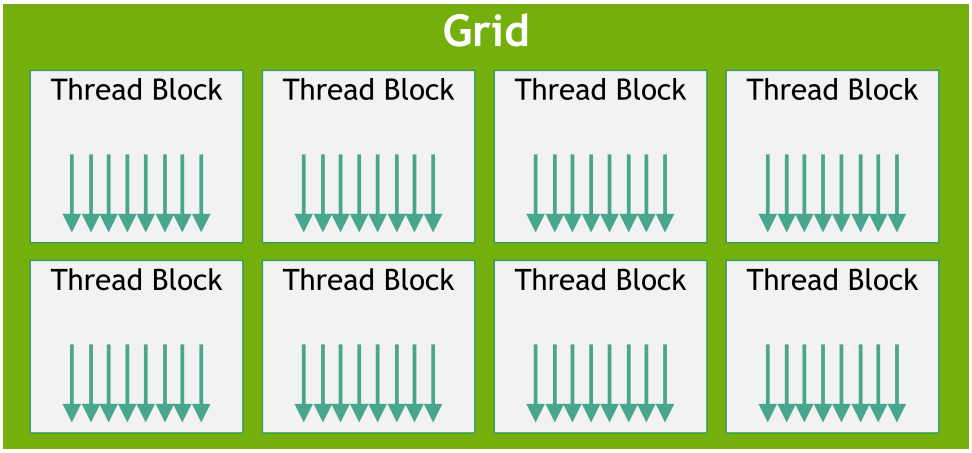

Figure 2.5: A CUDA grid composed of multiple thread blocks [7].

Thread blocks are important because they define the scope of cooperation between threads. Threads in the same block can communicate through shared memory and synchronize with block-level barriers. This block-level execution model is used by many GPU algorithms to break a large computation into smaller pieces that can run independently on different SMs.

2.2.2 Thread Block Clusters

Hopper-generation GPUs introduce *thread block clusters*. A cluster is a group of thread blocks that are scheduled together and can cooperate more closely than independent thread blocks. This extends the CUDA execution hierarchy from grid, block, and thread to grid, cluster, block, and thread, as shown in Figure 2.6 [7].

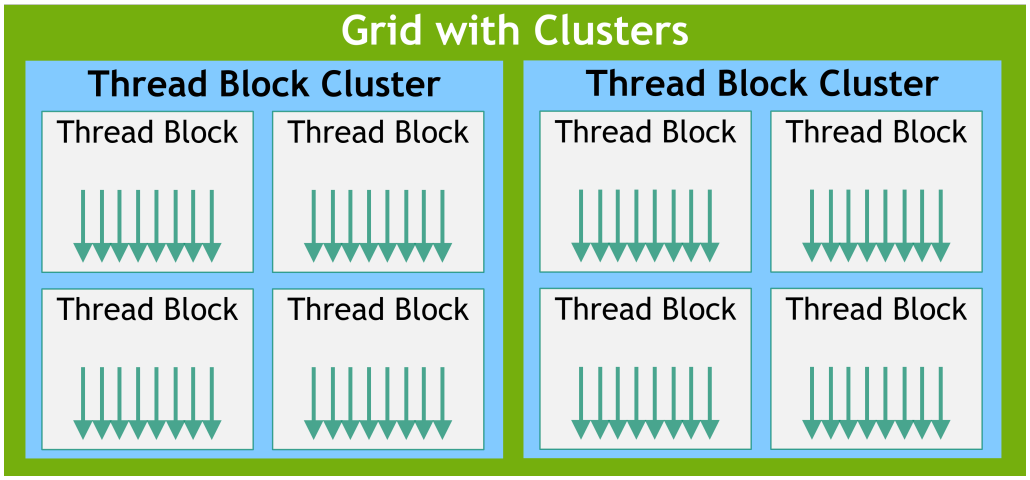

Figure 2.6: A CUDA grid organized into thread block clusters [7].

Thread block clusters are important because they allow blocks in the same cluster to access distributed shared memory (DSM). In the traditional CUDA model, shared memory is visible only to threads inside the same thread block. With DSM, blocks

in the same cluster can access each other's shared memory. This enables more coordination between blocks without always going through global memory [7].

2.2.3 CUDA Streams and Asynchronous Execution

CUDA operations are submitted to *streams*. A stream is an ordered sequence of operations. Operations submitted to the same stream execute in issue order, while operations submitted to different streams may execute concurrently if there are no dependencies and if enough hardware resources are available [7].

Streams allow a program to organize independent GPU work without forcing all operations to execute sequentially. For example, a program may launch one kernel in one stream and another kernel, memory copy, or communication-related operation in another stream. If these operations do not depend on each other, the GPU may be able to make progress on them at the same time.

However, using multiple streams does not automatically guarantee concurrency. The operations still depend on the available hardware resources and on the dependencies expressed by the program. When one operation needs data produced by another operation, the programmer must use synchronization mechanisms such as CUDA events or stream ordering to ensure that the consumer starts only after the producer has finished.

2.3 General Matrix Multiplication and Execution Libraries

General Matrix Multiplication (GEMM) is one of the most fundamental computational operations in high-performance computing and deep learning [11], [12], [13]. Its basic form is:

$$C = A \times B$$

where $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$, and $C \in \mathbb{R}^{M \times N}$. The parameters M, N, and K describe the problem size of the matrix multiplication.

In deep learning models, especially Transformer architectures, GEMM is used in many key components, including the linear layers in MLP blocks, the Query/Key/Value projections in the attention mechanism, the attention output projection, and embedding or feature transformation layers. Since these components are executed repeatedly during both training and inference, GEMM accounts for a large portion of the total floating-point operations (FLOPs) of the model [6].

On NVIDIA GPUs, GEMM execution is accelerated by specialized hardware units such as Tensor Cores. These units are designed to speed up matrix multiply-accumulate operations, which are central to GEMM execution.

2.3.1 Tile-Based GEMM Execution

In machine learning workloads, matrices are often too large to be processed efficiently as a single task. Therefore, modern GPUs partition GEMM into many smaller computational units called *tiles*. A tile is a sub-block of the output matrix C. Instead of computing the entire output matrix at once, the GPU computes the matrix multiplication tile by tile, decomposing a large workload into many smaller tasks [13], [14].

This tiled execution model is important because different output tiles can be assigned to different SMs for parallel execution. Each tile requires a corresponding region of matrix A and matrix B, and the computation proceeds by accumulating partial products across the K dimension. This structure allows the GPU to expose parallelism across output tiles while also reusing data through registers and shared memory inside each tile computation [13], [14].

2.3.2 GEMM Libraries and CUTLASS

Developers usually do not implement every GEMM kernel from scratch. Instead, they often rely on optimized GPU linear-algebra libraries. cuBLAS provides high-performance GEMM routines through a library interface, where the user calls a predefined routine and the library selects an optimized implementation internally [11]. This is useful for applications that need fast GEMM execution without directly constructing a custom GPU kernel.

CUTLASS follows a different approach. It is a CUDA C++ template library for building high-performance matrix multiplication and related kernels from reusable components [12], [13]. CUTLASS is useful when the programmer needs more control over the GEMM structure, such as the tile shape, data movement strategy, epilogue behavior, or kernel schedule. In CUTLASS 3.x, this composability is also supported by CuTe, which provides lower-level layout and tensor abstractions used to describe tiled computation and data movement [15], [16].

In CUTLASS terminology, a CUDA thread block is often referred to as a *Cooperative Thread Array* (CTA). A CTA-level GEMM tile is the portion of the output matrix assigned to one CTA or, in newer kernels, to a related scheduling unit such as a thread-block cluster. The work inside this tile is then divided across warps, warp groups, and individual instructions depending on the kernel design.

2.3.2.1 Mainloop, Epilogue, and Tile Completion

High-performance CUTLASS GEMM kernels can be understood as having two major phases: the mainloop and the epilogue [13], [15]. The core matrix multiplication happens in the *mainloop*. In this phase, the kernel iterates over the K dimension. During each iteration, it stages the required portions of matrices A and B from global memory into faster on-chip storage, such as shared memory and registers. The staged fragments are then consumed by matrix multiply-accumulate (MMA) instructions. In Tensor Core-based GEMM kernels, these MMA instructions are issued cooperatively by a warp or warp group, depending on the GPU architecture

and kernel schedule, while the arithmetic itself is executed by Tensor Cores. For example, Ampere-generation kernels commonly use warp-level MMA instructions, while Hopper-generation CUTLASS 3.x kernels may use warp-group-level MMA instructions in warp-specialized schedules. The partial results are accumulated in registers during the mainloop [10], [13], [15].

After the mainloop finishes accumulating the output values, the *epilogue* is executed. The epilogue is responsible for producing the final output values and writing them back to global memory. Depending on the GEMM implementation, the epilogue may also apply additional operations such as scaling, bias addition, activation functions, type conversion, or other elementwise transformations [13], [15].

Figure 2.7 shows a simplified view of the GEMM hierarchy, including the mainloop and epilogue. The important point is that a GEMM output is not produced as one single monolithic result. Instead, the output matrix is divided into tiles, and each tile may go through several internal steps before it is finally written to global memory. This means that tile completion is tied to the epilogue path, not only to the end of the matrix-multiplication mainloop [13].

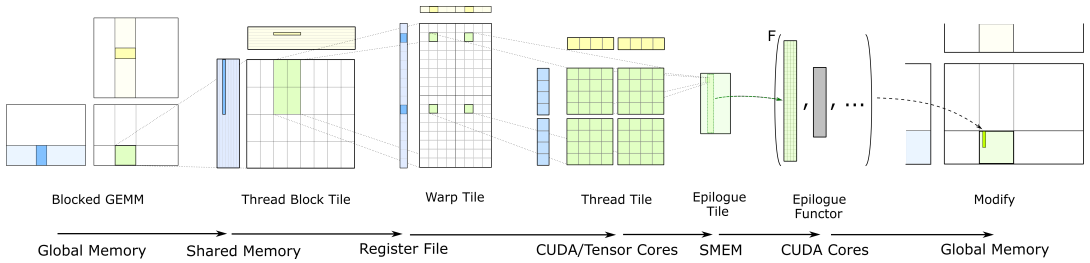

Figure 2.7: Simplified GEMM hierarchy, including the mainloop and epilogue stages [13].

2.3.2.2 CUTLASS 3.x GEMM Hierarchy

CUTLASS 3.x introduces a composable GEMM hierarchy designed to support newer GPU features such as TMA, warp-group matrix multiply instructions, and thread block clusters. Instead of describing a GEMM kernel only through the hardware hierarchy of CTAs, warps, and instructions, CUTLASS 3.x organizes GEMM kernels through device, kernel, collective, tiled MMA/copy, and atom-level abstractions. Figure 2.8 shows this layered organization [15], [16].

At the outer levels, the device and kernel abstractions describe how the GEMM operation is launched and scheduled on the GPU. The collective level describes the main computation and data movement pattern used by the kernel. Below that, tiled MMA and tiled copy abstractions describe how matrix multiplication and memory movement are performed over structured regions of data. Finally, the atom level represents the lowest-level building blocks, such as individual copy operations or matrix multiply operations. This hierarchy makes CUTLASS 3.x flexible for composing GEMM kernels with architecture-specific features while still reusing common building blocks [15], [16].

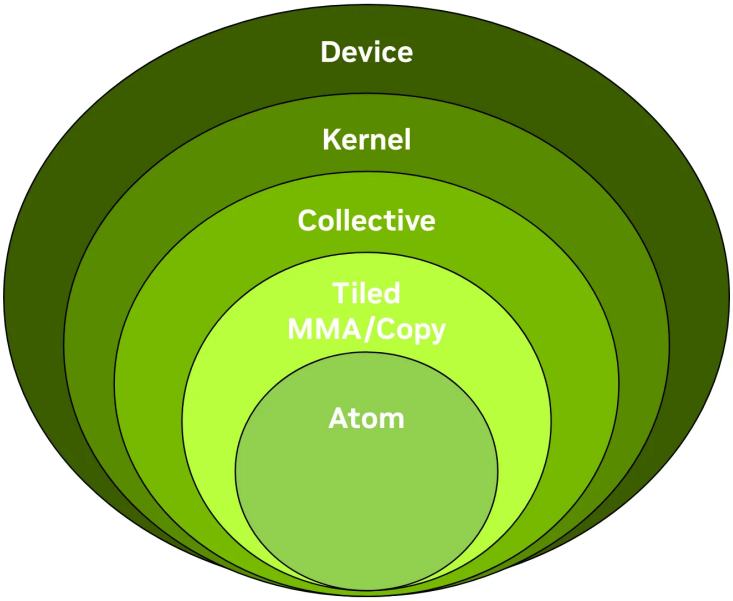

Figure 2.8: CUTLASS 3.x GEMM abstraction hierarchy [16].

2.3.2.3 Kernel Schedules and Tile Schedulers

CUTLASS GEMM kernels can be understood from two scheduling perspectives. The first is the *tile scheduler*, which decides which output tile is assigned to which CTA or thread-block cluster. The second is the *kernel schedule*, which describes how the work inside the kernel is organized, including data movement, matrix multiplication, synchronization, and the epilogue. This distinction is important because the tile scheduler affects the order in which output tiles are produced, while the kernel schedule affects how each tile is computed internally [15], [16].

In a normal tiled GEMM schedule, the output matrix is divided into independent tiles, and each CTA is assigned one tile to compute. The CTA loads the required portions of matrices A and B, iterates over the K dimension, accumulates the partial result, executes the epilogue, writes the output tile to global memory, and then terminates. This approach is simple and efficient when the number of output tiles is large enough to keep all SMs busy. However, if the number of tiles does not divide evenly across the available SMs, the final wave of work may underutilize the GPU. This effect is commonly referred to as *wave quantization*. For example, if a GPU has 100 SMs and the GEMM contains 2048 output tiles, then the first 20 waves can use all 100 SMs, but the final wave contains only 48 tiles. As a result, only 48 SMs are active during that final wave while the remaining SMs are idle [13], [16].

The *Stream-K* tile scheduler addresses this uneven distribution of work. Instead of being purely tile-centric, Stream-K is more work-centric because it can further partition work along the K dimension. This means that multiple CTAs may contribute to the computation of the same output tile. This can improve SM utilization for some GEMM shapes, but it also introduces additional complexity because partial results from multiple CTAs may need to be combined correctly [16].

On Hopper-generation GPUs, CUTLASS also supports warp-specialized kernel schedules. In these kernels, different warps or warp groups inside the same CTA or cluster are assigned different roles. Some warps act as *producer* warps, which are responsible for moving data from global memory into shared memory, often using TMA. Other warps act as *consumer* warps, which perform the matrix multiply-accumulate operations using the data staged in shared memory. This separation creates an internal producer-consumer pipeline inside the GEMM kernel. The pipeline improves performance by allowing data movement and computation to make progress at the same time, but it also makes the kernel structure more complex than designs where most warps perform similar work [9], [13], [15].

This producer-consumer organization can also improve resource allocation. Producer warps mainly issue data movement operations and may require fewer registers than consumer warps. This can leave more register capacity available for the consumer side, where the matrix multiply-accumulate operations and accumulator fragments require significant register storage [13], [15].

CUTLASS provides different variants of warp-specialized schedules, including *ping-pong* and *cooperative* schedules. In a *ping-pong* schedule, producer and consumer stages alternate across pipeline buffers. While one buffer is being used for computation, another buffer can be filled with data for a later stage. In a *cooperative* schedule, the producer-consumer organization is still present, but multiple warp groups cooperate more directly in the computation pipeline. This can improve performance for some GEMM shapes, but it also requires more careful coordination between data movement, computation, and synchronization [13], [15], [16].

2.4 Multi-GPU Interconnects and Memory Access

In multi-GPU systems, GPUs often need to exchange data during training and inference. The performance of this communication depends heavily on the underlying communication medium, because the available bandwidth, latency, and topology can directly affect the total execution time. Broadly, GPU communication can be categorized into *inter-node* and *intra-node* communication. Inter-node communication happens between GPUs located in different machines and is usually carried over networking technologies such as InfiniBand, Ethernet, Remote Direct Memory Access (RDMA)-capable networks, or, in simpler cases, the TCP/IP stack. Intra-node communication happens between GPUs inside the same machine and typically uses PCIe, NVLink, or NVSwitch. This thesis focuses on intra-node GPU communication and the hardware mechanisms that support it [5], [17].

2.4.1 PCIe

PCI Express, or PCIe, is the standard system interconnect used to connect GPUs to the CPU and the rest of the host system. In many GPU servers, PCIe provides the basic path between the CPU and each GPU. It can also be used for GPU-to-

GPU communication when the system topology supports it. For example, a PCIe Gen5 x16 connection provides about 64 GB/s bandwidth in each direction, or about 128 GB/s total bidirectional bandwidth [18].

Compared with GPU-specific interconnects, PCIe is more general-purpose and widely supported, but it usually provides lower bandwidth and higher latency for GPU-to-GPU communication. Therefore, when communication-heavy workloads run on systems where GPUs communicate mainly through PCIe, the communication cost can become a major bottleneck. In distributed deep learning, this affects collective operations such as AllReduce, ReduceScatter, AllGather, and All-to-All.

2.4.2 NVLink

NVLink is NVIDIA's high-bandwidth interconnect for GPU-to-GPU communication. Unlike PCIe, which is a general-purpose system interconnect, NVLink is designed to provide a faster communication path between accelerators. In multi-GPU systems, NVLink allows GPUs to exchange data with higher bandwidth and lower latency than PCIe in many cases [19]. For example, NVIDIA reports that fourth-generation NVLink on H100 provides up to 900 GB/s of GPU-to-GPU bidirectional bandwidth [20].

NVLink is especially important in multi-GPU workloads because GPUs frequently exchange intermediate tensors, partial results, or synchronization data during execution. These transfers are sensitive to inter-GPU bandwidth and latency, so the availability of NVLink can significantly affect communication performance. Figure 2.9 illustrates an example system where GPUs are connected to CPUs through PCIe, while GPUs also communicate with each other through NVLink.

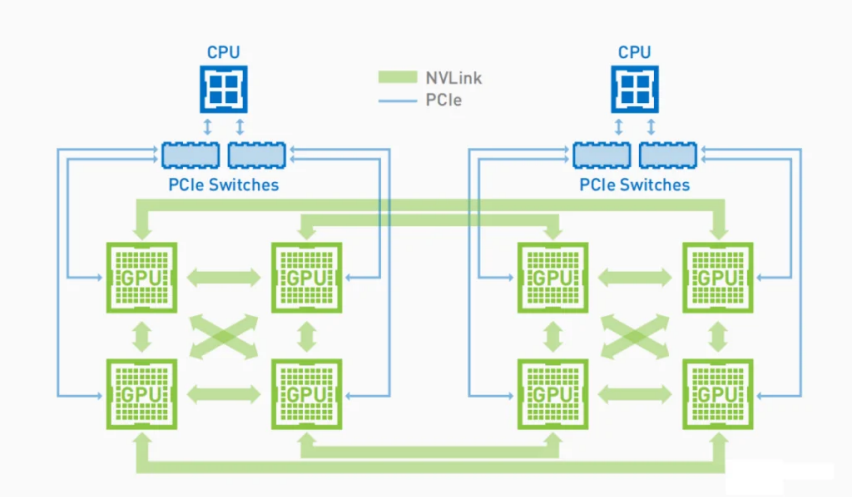

Figure 2.9: Example intra-node GPU system where PCIe is used for CPU–GPU communication and can also provide a GPU–GPU communication path, while NVLink provides a direct high-bandwidth GPU–GPU communication path [21].

However, NVLink by itself is still a point-to-point communication medium. It provides high-bandwidth links between devices, but the communication operation still

needs to be implemented by software or lower-level GPU communication mechanisms. One common software layer is NVIDIA Collective Communication Library (NCCL), which provides optimized collective communication operations for NVIDIA GPUs [17]. Therefore, the same collective operation may behave differently depending on whether the GPUs are connected through PCIe, direct NVLink links, or a switched NVLink fabric.

2.4.3 NVSwitch

NVSwitch extends NVLink from direct point-to-point connections into a switched fabric. In systems with many GPUs, directly connecting every GPU pair with dedicated NVLink links is difficult. NVSwitch solves this problem by acting as a high-bandwidth switch between GPUs. Instead of relying only on direct GPU-to-GPU links, GPUs communicate through the NVSwitch fabric. NVIDIA describes NVSwitch as an NVLink switch with 18 NVLink ports, where each port provides 50 GB/s of bidirectional bandwidth, giving 900 GB/s of aggregate switch bandwidth [22]. Figure 2.10 shows the difference between direct GPU-to-GPU connectivity and an NVSwitch-based topology [19].

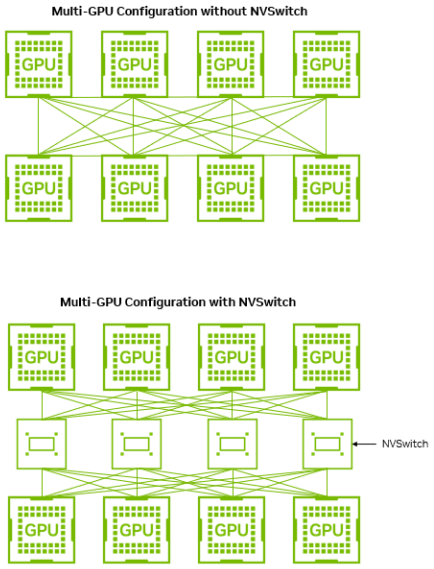

Figure 2.10: Comparison between a topology with direct NVLink connections between GPUs and a topology using NVSwitch [19].

The main advantage of NVSwitch is that it provides more scalable and more uniform intra-node connectivity. This is important for collective communication, where multiple GPUs communicate at the same time. Operations such as AllReduce and ReduceScatter are not only limited by the bandwidth of one link, but also by how efficiently traffic can be routed among all participating GPUs. NVSwitch helps by providing a high-bandwidth fabric that communication libraries can use to move data between GPUs.

Newer NVSwitch systems also support collective acceleration features. One important example is switch-assisted reduction, where part of a reduction operation can

be performed inside the NVSwitch fabric instead of being performed entirely by GPU SMs. This is useful for collectives such as AllReduce or ReduceScatter because reduction normally requires data from multiple GPUs to be combined. If the interconnect can perform part of this reduction, then the GPUs may spend fewer SM resources on communication-side reduction work [7].

NVSwitch also supports multicast-style communication in supported systems. Multicast means that one data movement operation can deliver the same data to multiple destination GPUs. This is useful when the same value or tensor region must be sent to several GPUs. Instead of sending the same data repeatedly through separate point-to-point transfers, multicast can reduce redundant traffic by using the switch fabric more efficiently [7].

At the lower software level, these features can be exposed through memory-semantic communication mechanisms. For example, NVLink SHARP and related multi-memory addressing mechanisms allow some communication operations to be expressed as memory operations rather than only as traditional host-launched collectives. In this model, a GPU operation can target a special multi-memory address, and the interconnect can perform operations such as multicast or reduction across multiple GPUs. Conceptually, this makes some collective behavior look closer to a memory operation, while the NVSwitch fabric performs part of the communication or reduction work [7], [10].

However, multicast memory addresses have a limitation. A memory address that is defined as a multicast memory region cannot be accessed through the same TMA-based data movement path used by GEMM kernels. Instead, multicast access relies on a different set of PTX instructions from the TMA instructions. The fundamental problem with this approach is that GEMM libraries, such as CUTLASS, have specialized kernels designed for execution on a single GPU using hardware resources such as TMA. Marking a memory region as multicast and requiring a different loading or reduction mechanism would require changes to these libraries so that they can adapt to the new memory-access path, which is cumbersome and unrealistic [10].

One important aspect to consider is that marking a memory region as a multicast memory region does not mean that part of the memory is isolated. Instead, the marking creates an alias for that memory region. This means that another alias can be introduced for the same memory region. This is possible using the Virtual Memory Management (VMM) APIs of CUDA [7].

2.4.4 CUDA Virtual Memory Management for Multi-GPU Access

CUDA Virtual Memory Management (VMM) provides lower-level control over how GPU memory is allocated, mapped, and accessed. Instead of only allocating memory through a single high-level allocation call, VMM separates the virtual address range from the physical memory allocation. This allows a program to reserve a virtual address range, create a physical memory allocation, map that allocation into the virtual address range, and define which GPUs are allowed to access it. Figure 2.11

shows this idea [7].

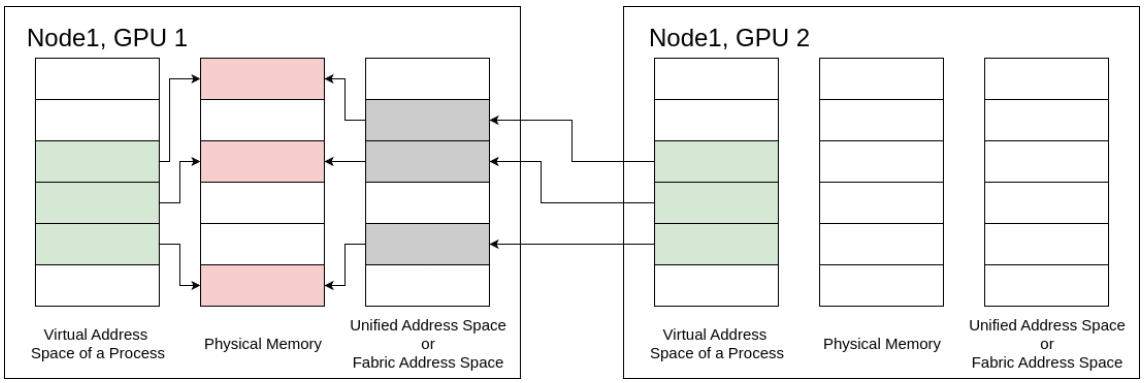

Figure 2.11: Example of CUDA virtual memory mapping for sharing memory across different address ranges [7].

This mechanism is useful in multi-GPU systems because the same physical memory allocation can be exposed through different virtual address mappings. For example, memory physically allocated on one GPU can be mapped into an address range that is accessible from another GPU, if the hardware and access permissions support it. This makes it possible for kernels or communication mechanisms on one GPU to access memory that belongs to another GPU.

In the context of this thesis, VMM is important because it provides a way to build memory regions that are visible across GPUs. This is different from ordinary local GPU memory, where a pointer usually refers to memory owned by one device. With VMM, the programmer can explicitly control memory mappings and access permissions, which is useful when implementing fine-grained communication or when creating aliases for the same underlying memory region.

VMM is also useful when working with special memory mappings such as multicast or communication-oriented aliases. Marking a memory region for multicast-style access does not necessarily create a separate physical allocation. Instead, it can create another virtual mapping for the same underlying memory. Therefore, one virtual address can be used for normal local access, while another virtual address can be used for communication-specific operations.

However, VMM should not be understood as a replacement for the communication hardware itself. Even if memory from another GPU is mapped into the address space, accesses to that memory still travel through the underlying interconnect, such as PCIe, NVLink, or NVSwitch. Therefore, VMM provides address mapping and access control, while the actual performance still depends on the physical communication path.

2.5 Collective Communication Primitives

To run a model on multiple GPUs, training and inference systems use parallelism strategies that distribute computation, data, or model components across devices.

This distribution can reduce per-GPU memory pressure, increase throughput, and make larger models practical. However, distributing the computation does not make each GPU fully independent. In most parallel execution strategies, each GPU only holds part of the data, parameters, or intermediate results needed for the complete computation. Therefore, GPUs must communicate with each other to exchange partial results, synchronize state, or route data to the devices where it is needed.

Different parallelism strategies create different communication needs. In *data parallelism*, the model is replicated across GPUs, and each GPU processes a different portion of the input batch. This improves throughput because more samples can be processed at the same time, but the model replicas must remain consistent across GPUs. In *tensor parallelism* (TP), tensors or operations inside a layer are partitioned across GPUs. This reduces the amount of computation and memory required per GPU for a layer, but intermediate results may be split across devices and must be exchanged during the layer execution. In *pipeline parallelism*, different layers or groups of layers are assigned to different GPUs, and micro-batches flow through the pipeline stages. This improves memory scalability for deep models, but it introduces communication between neighboring stages and may suffer from pipeline bubbles. In *expert parallelism*, different experts in a Mixture-of-Experts model are placed on different GPUs. This increases model capacity without activating every parameter for every token, but it requires tokens and intermediate results to be routed between GPUs [23].

These strategies are not mutually exclusive. Large-scale training and inference systems often combine data, tensor, pipeline, and expert parallelism to balance memory usage, computation, and communication. The main focus of this thesis is tensor parallelism (TP), because in this setting the output of a computation on one GPU often depends on partial tensor data produced by other GPUs. This creates fine-grained communication needs close to the computation itself.

The communication patterns introduced by these parallelism strategies are usually expressed through collective communication primitives. A collective operation defines how data is exchanged among a group of GPUs. Common primitives include `AllReduce`, `AllGather`, `ReduceScatter`, and `All-to-All`. NVIDIA Collective Communication Library (NCCL) provides optimized implementations of these collectives for NVIDIA GPU systems [24].

2.5.1 AllReduce

AllReduce is an operation where each GPU contributes an input tensor, and all tensors are combined using a reduction operation such as summation, as shown in Figure 2.12. After the reduction, every GPU receives the same complete reduced result [24].

AllReduce is commonly used for gradient synchronization in distributed deep learning. In data-parallel training, each GPU computes gradients using a different mini-batch. These gradients must be summed or averaged across GPUs before updating the model parameters. PyTorch DistributedDataParallel, for example, uses AllRe-

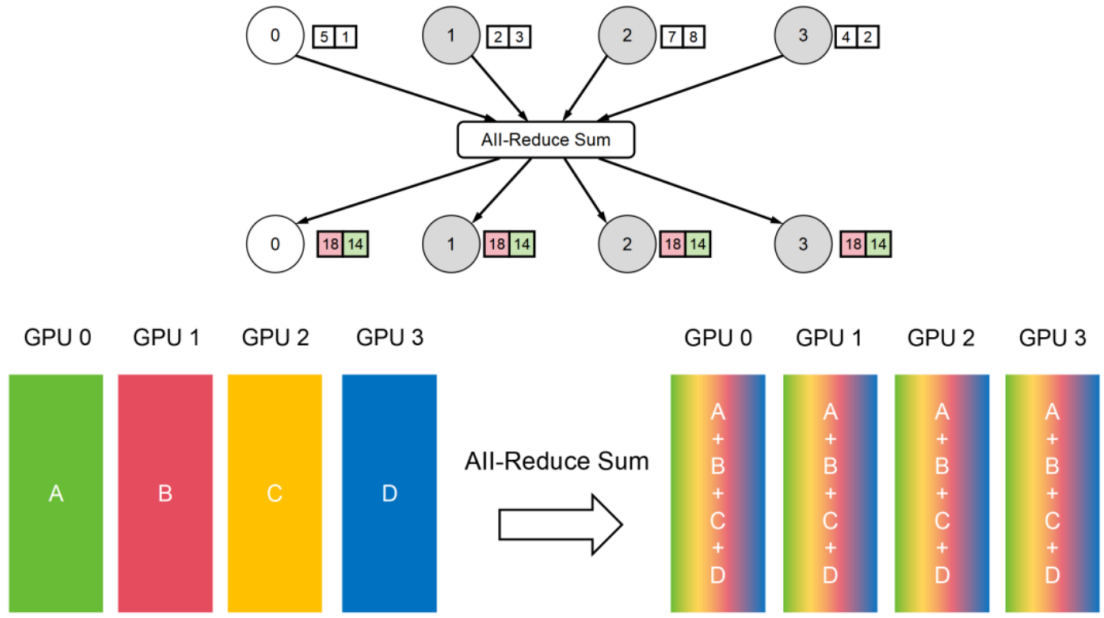

Figure 2.12: AllReduce across four GPUs. The input tensors are reduced element-wise, and the complete reduced result is available on every GPU [25].

duce to compute gradient summation across processes [26]. In tensor parallelism, different GPUs may compute partial results of a linear layer or attention projection, and AllReduce can be used to combine these partial results so that each GPU obtains the complete output.

2.5.2 ReduceScatter

With ReduceScatter, each GPU first contributes an input tensor, and the tensors are reduced element-wise across GPUs. The final reduced tensor is then split into equal-sized chunks, and each GPU receives one chunk of the reduced result [24].

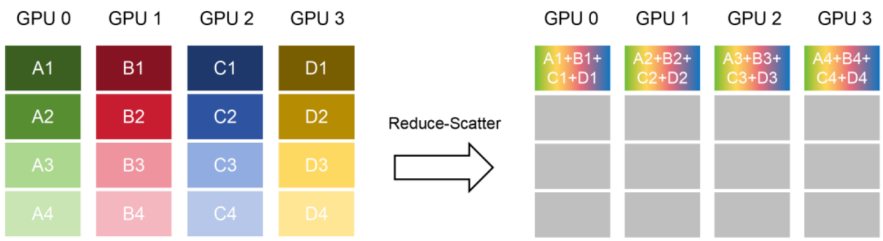

Figure 2.13: ReduceScatter across four GPUs. The input tensors are reduced element-wise, and the final reduced tensor is scattered across GPUs [25].

ReduceScatter is useful when each GPU only needs a shard of the reduced result. For example, distributed optimizers can use ReduceScatter to reduce parameter gradients while keeping only the gradient shard needed by each GPU, followed by AllGather to reconstruct updated parameters when needed [23].

2.5.3 AllGather

With AllGather, each GPU contributes one chunk of data, and all chunks are collected from all GPUs. After the operation, every GPU receives the complete tensor formed by concatenating the chunks from all GPUs, as shown in Figure **??**. In NCCL, the gathered output is ordered by rank index [24].

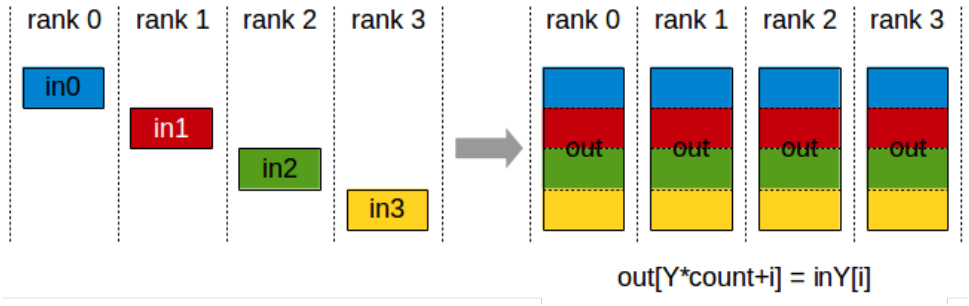

Figure 2.14: AllGather across four GPUs. Each GPU contributes one chunk, and every GPU receives the concatenation of all chunks.[24]

AllGather is commonly used when data has been distributed across multiple GPUs and each GPU later needs the complete tensor. For example, in tensor parallelism or distributed optimizer settings, each GPU may hold only part of an activation, gradient, or parameter tensor. AllGather collects these parts and makes the full tensor available on every GPU. It is also often used together with ReduceScatter: ReduceScatter reduces the data and distributes different chunks to different GPUs, while AllGather collects the chunks back when the complete result is needed [23], [24].

2.5.4 All-to-All

All-to-All (Figure 2.15) is used when data needs to be redistributed across GPUs. Each GPU sends different parts of its input tensor to different destination GPUs, and at the same time receives data from all other GPUs. After the operation, every GPU obtains a new tensor composed of the corresponding chunks from all GPUs [24].

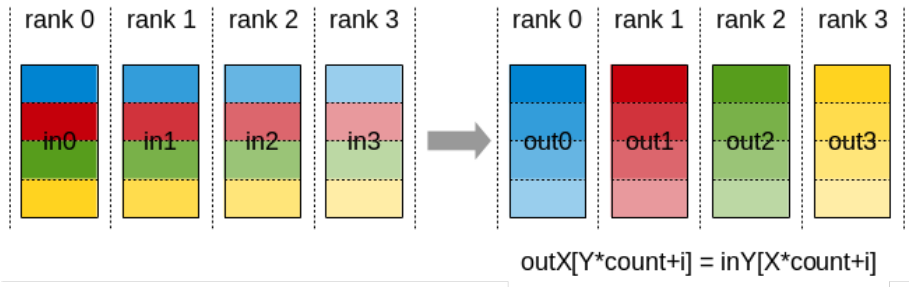

Figure 2.15: All-to-All across four GPUs.[24]

In expert parallelism for Mixture-of-Experts models, tokens may be routed to different experts located on different GPUs. After local computation, All-to-All can be

used to exchange token data so that each GPU receives the data required for the next computation stage.

2.6 GPU Resource Partitioning and Control

Modern GPUs execute many kernels on a shared pool of hardware resources. When multiple kernels run concurrently, they may compete for resources such as SMs, cache capacity, memory bandwidth, and scheduling opportunities. This contention can make kernel execution less predictable and may reduce the benefit of concurrency.

One way to reduce such interference is to partition or control GPU resources. Instead of allowing all workloads to use the GPU in the same unrestricted way, different mechanisms provide different levels of resource control. Some mechanisms partition the GPU at the hardware level, while others focus on process-level sharing or finer-grained control of SM resources within an application. This section discusses three NVIDIA mechanisms related to GPU resource partitioning and control: Multi-Instance GPU (MIG), Multi-Process Service (MPS), and Green Context (GC).

2.6.1 Multi-Instance GPU

Multi-Instance GPU (MIG) provides hardware-level partitioning. It partitions supported GPUs into multiple GPU instances, each with dedicated compute and memory resources. This provides a strong isolation boundary because several resources, including memory-related resources, are divided between the created instances [27].

MIG is mainly useful when the goal is to isolate different workloads or users on the same physical GPU. However, this strong isolation also makes MIG less flexible for fine-grained coordination inside a single distributed workload. In particular, MIG does not support P2P communication between instances on different GPUs, and NCCL is not fully supported with MIG [27]. Therefore, MIG is not the main resource-control mechanism considered in this thesis.

2.6.2 Multi-Process Service

Multi-Process Service (MPS) is mainly used for sharing a GPU among multiple CUDA processes. It reduces context-switch overhead and can improve the performance of multiple small workloads running at the same time [28].

MPS targets process-level sharing rather than explicit partitioning of resources inside a single application. It is useful when several CUDA processes need to use the same GPU more efficiently. However, MPS is not designed to give fine-grained control over how different kernels inside one application use the GPU resources. Therefore, although MPS provides a form of GPU resource control, it is less suitable for studying low-level resource partitioning between computation and communication kernels within the same workload.

2.6.3 Green Context

Green Context (GC) provides a lighter-weight way to partition GPU SM resources into separate logical parts [7]. Unlike MIG, GC does not require hardware-level GPU instance creation. Unlike MPS, GC can be used to control resource usage within an application rather than only between different CUDA processes. Therefore, GC provides a more explicit and controllable form of SM resource management for kernels inside one workload.

In this thesis, Green Context is the main resource-partitioning mechanism considered. GC can divide the available SMs into logical partitions, as shown in Figure 2.16. These partitions can then be assigned to different execution paths or kernel groups.

Figure 2.16: Green Context static resource partitioning [7].

For example, an application may use one partition for compute-intensive kernels such as GEMM kernels and another partition for communication-related kernels such as NCCL kernels. In this case, the goal is to reduce contention between the compute path and the communication path by preventing both types of kernels from freely competing for the same SMs.

In this work, we use two partitions:

- **Compute Context / Compute Partition**: runs compute-intensive kernels such as GEMM.
- **Communication Context / Communication Partition**: runs communication-related kernels such as NCCL kernels.

CUDA streams can be used to run kernels concurrently. For example, a GEMM kernel can run in one stream while a communication kernel runs in another stream. However, using different streams does not by itself prevent these kernels from competing for the same SMs, cache, and memory bandwidth [5], [6]. With GC, the kernels can still overlap in time, but they can be assigned to different SM subsets. This makes the resource usage more controlled and reduces SM-side contention between GEMM and communication kernels [6], [7].

2.7 Summary

In this chapter, we discussed the main technical background needed for the rest of the thesis. We first described how CUDA programs execute on NVIDIA GPUs, including kernels, thread blocks, warps, streams, and thread block clusters. We also discussed the main GPU resources used by these programs, such as SMs, registers, shared memory, global memory, and the memory hierarchy. From this discussion, we observed that GPU computation is not only limited by arithmetic throughput, but also by how data is moved and synchronized across different memory spaces and execution units.

We then focused on GEMM, which is one of the dominant computation patterns in deep learning workloads. We discussed how GEMM is executed using tiled computation and how optimized libraries such as CUTLASS structure GEMM kernels through the mainloop, epilogue, tile scheduling, and warp-specialized execution. These details are important because GEMM kernels are already highly optimized and use GPU resources carefully. Therefore, overlapping communication with GEMM requires understanding how computation, data movement, and resource usage interact inside the GPU.

Next, we discussed intra-node multi-GPU communication. We reviewed PCIe, NVLink, NVSwitch, and CUDA VMM as mechanisms that affect how GPUs exchange data and access memory across devices. We also described collective communication primitives such as AllReduce, ReduceScatter, AllGather, and All-to-All. These primitives express the communication patterns required by distributed machine learning parallelism strategies. In particular, we focused on tensor parallelism (TP), where GPUs often need to exchange partial tensor results during layer execution.

Finally, we discussed GPU resource partitioning and control mechanisms, including MIG, MPS, and Green Context. We observed that these mechanisms operate at different levels: MIG provides hardware-level partitioning, MPS supports process-level sharing, and Green Context provides finer-grained control over SM resources within an application. Since this thesis studies the interaction between compute kernels such as GEMM and communication kernels such as NCCL kernels, Green Context is the main resource-control mechanism considered in the rest of the work.

Overall, the concepts discussed in this chapter provide the foundation for understanding the computation, communication, and resource-management challenges studied in the following chapters.

3

Related Works

In multi-GPU model training, computation and communication are tightly coupled. Large models are distributed across multiple GPUs, which requires communication between GPUs to share intermediate computation results. Therefore, training speed depends not only on how quickly computation takes place, but also on how effectively communication between GPUs is performed.

Prior work has addressed the efficiency of training in multi-GPU environments from different perspectives. Based on these perspectives, we categorize the existing work into four major categories. First, *software-based compute and communication overlap* techniques attempt to hide communication latency by overlapping communication with computation kernels. While software-based overlap can provide considerable speedup, it often requires substantial hand-written kernel implementation. Therefore, another line of work introduces *compiler and programming abstractions* to reduce engineering effort. Then, *hardware-assisted overlap and collective operation* either exploits existing hardware features or proposes new hardware mechanisms to facilitate communication without significantly affecting computation kernels. Finally, *communication libraries and backends* try to optimize the communication that happens between GPUs, mostly independently of computation.

3.1 Software-based Compute and Communication Overlap

Communication libraries improve communication speed and utilize bandwidth efficiently, but they do not remove the dependency between computation and communication. Traditionally, distributed deep learning on multiple GPUs is executed sequentially: after a computation kernel finishes execution, communication takes place to produce the final desired output. We refer to this execution model as the **baseline**.

This has motivated a line of work that attempts to overlap communication with computation. More specifically, instead of executing communication only after computation finishes, communication is performed alongside computation. Communicating partial results while the remaining computation continues can hide communication latency, thereby achieving speedup compared to the baseline. However, coordination or synchronization is needed between computation and communication kernels

so that communication libraries perform collective operations at the correct time and on valid intermediate results, ensuring data correctness. Moreover, CCLs also require GPU resources, such as SMs, to perform communication, which affects how computation kernels allocate resources [29], [30], [31]. Therefore, existing works can be categorized around three important questions. First, at which point during computation should communication start? Second, should communication happen inside the computation kernel or separately? Third, how much implementation effort is needed to perform the overlap? Based on these questions, prior work can be grouped into operator decomposition, kernel fusion, and signaling-based overlap.

3.1.1 Operator Decomposition

One of the fundamental overlap techniques is operator decomposition, in which the computation is separated into multiple chunks. In this approach, communication for previous chunks is performed while computation for the current chunks is still executing. This approach is attractive because it can use existing high-performance tools for each task. For example, cuBLAS can be used for computation, while various CCLs can be used for communication with tuned settings based on the workload. Prior work commonly uses this approach as a straightforward way to overlap computation and communication by decomposing operators and scheduling them on different CUDA streams [6], [32], [33].

However, one of the basic issues with this approach is that many synchronization operations and kernel launches are needed to perform the overlap. While it can provide speedup compared to the baseline in many cases, it cannot always utilize the available resources efficiently. As a result, although decomposition provides overlap with less implementation effort, it may not fully utilize GPU resources compared to more fine-grained approaches such as kernel fusion [29], [32], [33].

3.1.2 Kernel Fusion

Kernel fusion places communication inside the computation kernel. In this approach, the computation kernel starts communicating partial results depending on which work unit has already completed, such as a tile, a group of tiles, or a workgroup. Kernel fusion reduces the amount of synchronization needed between the computation and communication stages, but at the cost of forming a larger kernel and requiring substantial low-level implementation effort. Moreover, kernel fusion typically injects communication logic into the epilogue or other parts of the computation kernel, which can interfere with optimizations already applied to computation kernels in frameworks such as cuBLAS and CUTLASS [4], [6], [29].

FLUX follows this direction by introducing the reduction operation at the end of the computation kernel. This requires peer-to-peer access among the participating GPUs. The computation unit on which FLUX performs communication is tile-level, meaning that it provides more fine-grained overlap [29]. Although FLUX provides fine-grained overlap by operating at tile level, it has two main limitations. First, it requires low-level manual implementation to perform communication at the end of

the GEMM kernel. Second, transferring data at tile granularity may not fully utilize the bandwidth of the shared communication medium between GPUs.

While FLUX addresses general GEMM cases, Comet, which builds on ideas from FLUX, addresses overlap for Mixture-of-Experts (MoE) models. MoE workloads are more irregular because token routing results in more dynamic patterns of communication and computation. Comet addresses this by analyzing data dependencies and allocating different CTAs for communication and computation, in contrast to FLUX, where communication is performed as part of the same computation CTA [34].

TokenWeave points out another limitation of fused-kernel approaches, as demonstrated in FLUX. It argues that fused kernels are capable of hiding communication latency when there is a considerable amount of computation. This means that kernel fusion may not be an appropriate approach for fully utilizing GPU resources in distributed inference workloads such as LLM inference. Instead of relying solely on kernel fusion, TokenWeave implements a token-splitting technique, where the computation of one subset of tokens is performed while the communication of another computed subset is transferred. Moreover, it fuses the AllReduce collective with RMSNorm to form a larger kernel and provide more overlap opportunities [31].

3.1.3 Signaling-based Overlap

Although kernel fusion enables fine-grained compute-communication overlap, it introduces two main challenges. First, because communication is integrated into the computation kernel, implementing fused kernels requires substantial low-level engineering effort. In addition, such implementations may be difficult to maintain as optimized computation libraries such as CUTLASS evolve. Second, computation kernels in frameworks such as CUTLASS and cuBLAS are highly tuned for specific problem sizes and hardware configurations. Adding communication logic to these kernels may interfere with existing optimizations and reduce computation efficiency [6].

FlashOverlap addresses some of these limitations using a lightweight signaling mechanism. Instead of performing communication directly inside the computation kernel, FlashOverlap signals when tiles of the GEMM output become ready. The communication path can then transfer groups of ready tiles while the remaining computation continues. Since tiles may complete in an order different from the memory layout expected by communication libraries, FlashOverlap reorders completed tiles into a contiguous buffer before invoking communication. This allows the system to preserve much of the original computation kernel performance while remaining compatible with existing communication libraries [6].

However, signaling-based overlap still requires modifications to the computation path. The computation kernel must expose tile-completion information, and the system must coordinate signaling, reordering, and communication to ensure correctness.

3.2 Compiler and Programming Abstractions

The software-based overlap approaches discussed in the previous section can improve performance and provide opportunities for overlap. However, software-based approaches suffer from one particular limitation: manual implementation effort is often required to make overlap possible. To shed more light on this issue, both kernel fusion and signaling mechanisms require a way to notify the communication logic when it should start transferring data. This means that some degree of modification to the computation kernel is usually needed. This becomes cumbersome to maintain because the manual implementation may need to change across different operators, GPU architectures, and communication patterns. For this reason, another line of work addresses this issue either by designing compilers that support automatic kernel generation or by providing programming abstractions and tools that make it easier to implement overlap techniques [32], [33], [35], [36].

To start with, TileLink provides a set of tile-centric primitives for describing the interaction between computation and communication. Instead of forcing developers to manually write fused kernels, TileLink separates the computation and communication design spaces and connects them through a series of tile-centric primitives. Its frontend allows developers to describe this interaction using high-level primitives, and its backend lowers these primitives to actual low-level communication primitives. As a result, TileLink generates fused kernels without requiring developers to manually implement them [32].

AutoOverlap also aims to reduce the engineering effort needed to support fine-grained overlap. AutoOverlap introduces a chunk-based abstraction in which the transfer granularity is at the chunk level. It uses compiler and runtime transformations to align computation with communication availability and support this chunk-based abstraction. The attractive point of AutoOverlap is that it allows local kernels, such as Triton kernels, to be transformed into distributed kernels that support fine-grained overlap [33].

Another set of works focuses on providing easier programming abstractions for implementing overlap. For example, ThunderKittens provides tile-based abstractions that hide some of the complexity of CUDA programming and allow easier kernel implementation [35]. ParallelKittens extends this work to support multiple GPUs. ParallelKittens provides more reusable components and inter-GPU communication primitives. The goal of these systems is not to target specific operators, but to introduce reusable building blocks for designing efficient kernels with overlap capabilities across different workloads [36].

3.3 Hardware-Assisted Overlap and Collective Operation

All of the methods described so far use GPU resources to perform overlap. One reason for this is that current CCLs need GPU resources, particularly SMs, to perform

data transfers. On the other hand, overlap techniques, whether software-based or compiler-supported, need synchronization mechanisms to notify the communication logic. As a result, computation kernels receive fewer resources and may also be required to perform synchronization [3], [30], [31].

Hardware-assisted approaches attempt to reduce these overheads by exploiting available hardware or proposing new hardware mechanisms for moving part of the tracking, triggering, data movement, or collective operations into hardware. Instead of relying on software-based approaches to determine when intermediate results are ready, hardware mechanisms can monitor and track progress and start communication. Moreover, collective operations such as ReduceScatter can be performed by specialized hardware to reduce collective-operation overhead [3], [37], [38].

T3 is a representative example of this direction. It proposes a hardware-software co-design for transparent tracking and triggering of fine-grained compute-communication overlap. In T3, hardware tracks the progress of the producer computation and triggers communication once the corresponding output data is ready. The communication is then performed through a DMA-based mechanism, reducing the need to use SM resources for communication coordination [3].

Furthermore, ParallelKittens showcases an example implementation of an AllReduce operation with the help of NVSwitch hardware. In this implementation, the reduction operation is performed through NVSwitch rather than consuming SM resources [36].

3.4 Communication Libraries and Backends

Collective Communication Libraries (CCLs) provide the collective communication primitives needed for sharing intermediate results during the training of large models on multi-GPU systems. Typical operations supported by CCLs include AllReduce, AllGather, ReduceScatter, and All-to-All [24]. We categorize existing work into three major groups based on how communication is performed: general-purpose CCLs, programmable CCLs, and resource-efficient or specialized CCLs that often propose in research systems.

To start with, the NVIDIA Collective Communication Library (NCCL) is one of the most well-known CCLs. NCCL provides high-performance collective communication primitives that are optimized for NVIDIA GPU hardware. One of the most important features of NCCL is that it supports different transport mechanisms, topologies, and protocols, which are selected automatically [17], [39]. On the other hand, AMD developed a CCL that is highly optimized for AMD GPUs, called RCCL [40]. Both NCCL and RCCL are developed by GPU manufacturing and design companies, meaning they are constantly updated to utilize the latest hardware and GPU features. Because of these features and their ease of use, these general-purpose CCLs widely used as default communication backends for distributed deep learning training.

While general-purpose CCLs are widely used, they provide limited control over how

communication is performed because low-level APIs and implementation details, such as scheduling and protocol decisions, are not exposed to developers. To address this limitation, the Microsoft Collective Communication Library (MSCCL) and its extension, MSCCL++, provide more programmable communication abstractions for defining custom communication algorithms [41], [42]. Moreover, MSCCL++ separates low-level communication primitives from high-level abstractions, allowing users to describe their communication patterns and algorithms while balancing performance, portability, and programmability.

Although both general-purpose and programmable CCLs are useful for ease of use and flexibility in introducing communication primitives for distributed training, they have the limitation of not exposing a global uniform memory view across GPUs. Due to this limitation, collective operations typically need to be orchestrated from the host side, and it is not directly possible to execute communication operations within GPU kernels. NVSHMEM addresses this limitation by providing a global address space across GPUs and enabling them to perform memory operations, such as get, put, and atomic operations, from within GPU kernels [5], [43]. This approach enables more fine-grained and asynchronous communication, as well as more generalized algorithms that operate on memory operations rather than depending only on network-level primitives. However, this approach places the burden of synchronization and correct data movement on the programmer.

Recent works focus not only on the communication algorithm itself, but also on how the algorithm executes communication. ResCCL argues that most traditional CCLs allocate a static number of CTAs based on data size and use inefficient internal scheduling mechanisms [30]. ResCCL addresses this problem by scheduling the execution of communication primitives while considering data dependencies between internal communication channels. Furthermore, it provides flexible CTA allocation and generates lightweight communication kernels with the goal of reducing pipeline bubbles among communication channels, improving bandwidth utilization, and reducing SM usage.

Overall, existing CCLs address communication between multiple GPUs and aim to improve and scale communication performance independently of computation kernels. This motivates another line of work, discussed in the next section, that attempts to overlap communication with computation.

3.5 Positioning of this thesis

From the works discussed in this chapter, we observe that different approaches provide different ways of improving or enabling overlap between computation and communication. Communication libraries mainly optimize the communication backend itself, software-based approaches try to expose overlap opportunities at different granularities, compiler and programming abstractions reduce the implementation effort, and hardware-assisted approaches move part of the communication or synchronization work closer to the hardware. However, one aspect that is less commonly addressed directly is the resource usage and contention between computation and

communication. Ideally, the communication path should use as few GPU execution resources as possible without sacrificing communication efficiency, so that more resources can remain available for the computation path.

This observation naturally leads to hardware-assisted solutions, which can be considered close to optimal in terms of reducing software overhead and SM usage. However, hardware solutions also have drawbacks, especially because they require new hardware support or changes to the GPU system design. Instead, this thesis stays mostly on the software side and explores how existing software features and newer GPU hardware features can be used to improve resource utilization in overlap scenarios.

The important distinction between this thesis and existing work is that we do **not** introduce yet another overlap technique or compiler-based approach. Instead, we introduce two ideas that aim to improve resource utilization in environments where computation and communication overlap. More specifically, we focus on controlling contention and reducing the usage of SMs, which are a fundamental execution resource of the GPU, between computation and communication kernels. For this purpose, we propose two methods: *GCOverlap* and *Oh Overlap*. The first method uses Green Context to reduce competition for SMs between computation and communication kernels, thereby improving execution time. The second method introduces a new CCL backend that is built on top of newer GPU hardware features, such as TMA, with the goal of reducing the SM resources needed by communication.

As stated above, this thesis is not intended to introduce a new overlap mechanism by itself. However, for testing and implementation, this thesis relies on FlashOverlap[6]. This means that our methods either build on top of FlashOverlap or are evaluated in a FlashOverlap-style environment. We choose FlashOverlap for two main reasons. First, due to the complexity of GEMM kernel-builder libraries such as CUTLASS, the testing platform should be easy to integrate with these libraries and should be as non-invasive as possible. When computation and communication logic are tightly entangled, it becomes difficult and time-consuming to incorporate the overlap logic into GEMM kernel-builder libraries, especially because these libraries are complex and evolve quickly. For this reason, FlashOverlap provides an attractive signaling-based approach that requires less invasive modification to the GEMM computation path.

Second, our methods are designed around the separation between computation and communication. Therefore, the testing platform should also be designed with this separation in mind. This is different from kernel-fusion approaches, where communication logic is directly inserted into the computation kernel, and also different from some compiler-based approaches that generate fused kernels. In contrast, FlashOverlap already separates the computation path from the communication path by using tile-completion signals. This makes it an appealing choice for implementing and evaluating the resource-efficient methods proposed in this thesis.

Finally, although the two proposed methods are developed in different environments, they address the same broader goal: improving GPU resource utilization when computation and communication are executed together. GCOverlap approaches this goal by introducing an isolation mechanism through Green Context, which assigns

computation and communication kernels to different SM subsets and reduces direct competition for SM resources. Oh Overlap approaches the same goal from the communication-backend side by reducing the SM usage of communication kernels through a new CCL backend. Unlike a method that is tied only to one specific overlap technique, Oh Overlap is designed as a more general-purpose CCL backend and can potentially be used in other communication scenarios as well. These two methods are also complementary: GCOverlap controls how GPU resources are divided between computation and communication, while Oh Overlap reduces the amount of GPU resources needed by the communication path itself. Therefore, they can potentially be combined to further improve resource utilization in overlapping scenarios.

4

Methods

This chapter presents the two systems developed in this thesis to address GPU resource utilization in compute-communication overlap on modern GPU hardware. Both systems are studied in the context of FlashOverlap [6], which provides the baseline overlap mechanism and the tile-level signaling model used for testing and integration.

First, the GCOverlap, a system for overlapping GEMM computation with collective communication in distributed training. The work is inspired by and builds upon FlashOverlap [6], which provides the baseline overlap mechanism and core ideas. We extend it by introducing Green Context SM partitioning to eliminate resource contention between compute and communication, an offline configuration search to find the optimal tile segmentation, and a runtime execution model that drives the overlap.

Second, Oh Overlap addresses the communication side of the problem from a different direction. It implements a collective communication layer that uses TMA, a hardware feature introduced with NVIDIA Hopper GPUs, to perform data movement and reduction with fewer SM-side resources. The goal of Oh Overlap is to reduce the amount of GPU execution resources required by the communication path, so that more resources can remain available for computation when communication and computation are executed together.

Since FlashOverlap is used as the main testing and integration platform in this thesis, its compatibility with the required GPU features also matters. The original FlashOverlap implementation was developed for an earlier GPU environment, while Oh Overlap requires TMA support, which is available from Hopper-generation GPUs onward. Therefore, the implementation of the two methods follows the GPU architecture required by each method. GCOverlap uses the original FlashOverlap setup, while Oh Overlap requires changes to make the FlashOverlap signaling mechanism work on Hopper GPUs. Part of this chapter therefore describes the changes needed to support FlashOverlap-style GEMM-side signaling on Hopper. Although the two systems are implemented differently, they share the same high-level goal of improving GPU resource utilization during compute-communication overlap.

The rest of this chapter is organized as follows. We first describe FlashOverlap, which provides the baseline signaling-based overlap mechanism used in this work. We then present the design of GCOverlap, including Green Context setup, tile seg-

mentation, configuration search, and runtime execution. After that, we describe the changes needed to support FlashOverlap-style signaling on Hopper GPUs. Finally, we present Oh Overlap, including its memory setup, task abstraction, TMA pipeline, collective planning, communication kernel variants, and runtime tuning process.

4.1 FlashOverlap

FlashOverlap [6] serves as the baseline overlap mechanism in this work. It enables fine-grained overlap between a single GEMM and and dependent communication primitives such as AllReduce or ReduceScatter by tracking the completion of GEMM output tiles. Instead of waiting for the entire GEMM output to finish, FlashOverlap triggers communication when a tunable group of completed output tiles becomes ready.

A key observation in FlashOverlap is that GEMM tiles complete in waves. A wave is a set of tiles that execute concurrently and finish within a short time window. Triggering communication for every individual tile would maximize overlap opportunity, but it would also fragment communication and reduce bandwidth utilization. Therefore, FlashOverlap groups one or more waves into a wave group and starts communication only after all tiles in the group have completed. These groups are determined according to the GEMM shape, tiling configuration, and tile execution order.

FlashOverlap uses a signaling mechanism to track when each wave group becomes ready. During GEMM execution, tiles update a counting table according to their group assignment. When the counter of a group reaches the number of tiles in that group, the computation kernel sends a signal that allows the communication path to launch the corresponding collective operation. This mechanism preserves the original GEMM mainloop and enables communication to overlap with the remaining GEMM computation.

FlashOverlap also requires reordering. Because optimized GEMM kernels may use block swizzling, the tile completion order may not match the logical memory order of the GEMM output. Moreover, NCCL collectives operate on contiguous communication buffers. FlashOverlap therefore reorders completed tiles before communication so that the data belonging to a group is placed contiguously, and then restores or adjusts the data order after communication. Figure 4.1 illustrates this reordering mechanism.

For ReduceScatter, this reordering is more involved because rows must remain complete on the destination GPU; FlashOverlap handles this by using subtiles and correcting the row order when needed.

4.2 System Overview of GC overlap

To eliminate SM contention in the FlashOverlap, we introduce SM partitioning using CUDA Green Context.

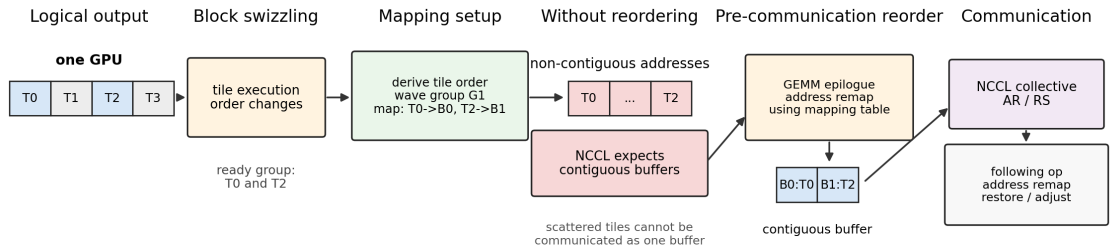

Figure 4.1: Simplified reordering in FlashOverlap. Block swizzling can make a ready wave group non-contiguous in memory, so FlashOverlap uses a mapping table to pack completed tiles into a contiguous NCCL buffer before communication and restores or adjusts the output order afterward.

Given a GPU with N_{total} SMs and a user-specified compute fraction $\alpha \in (0, 1)$, we allocate:

$$N_{\text{comm}} = 2\,\lfloor N_{\text{total}} \cdot (1-\alpha)/2 \rfloor, \tag{4.1}$$

$$N_{\text{compute}} = N_{\text{total}} - N_{\text{comm}}, \tag{4.2}$$

where the rounding ensures N_{comm} is even (required by GC).

The runtime first retains the primary CUDA context and obtains the full SM resource using the CUDA driver API. It then calls `cuDevSmResourceSplitByCount` to split out the communication SM resource, while the remaining SM resource is used for computation. From these two resource descriptors, the runtime creates two Green Context handles, `gc_compute` and `gc_comm`, and derives one `CUcontext` from each handle.

Two non-blocking streams are then created with cuGreenCtxStreamCreate: the `compute_stream` inside `gc_compute`, and `comm_stream` inside `gc_comm`. This means the SM isolation is established by the CUDA context and stream resources before any GEMM or communication kernel is launched; the GEMM kernel itself is not manually restricted to a subset of SMs.

The communication library is initialized while the communication Green Context is active. This ensures that the library resources used by the collective kernels are associated with the communication partition. As a result, collective operations submitted to `comm_stream` run on the communication SM subset rather than on the full primary context.

4.2.1 Tile Segmentation and Communication Triggering

CUTLASS partitions the output matrix $\mathbf{C} \in \mathbb{R}^{M \times N}$ into $T = \lceil M/B_M \rceil \times \lceil N/B_N \rceil$ tiles of size $B_M \times B_N$, each computed independently by one SM. As described before, FlashOverlap observes that tiles do not complete randomly: across runs, the N_{comp} compute SMs tend to finish tiles in consistent batches, which they call *waves*.

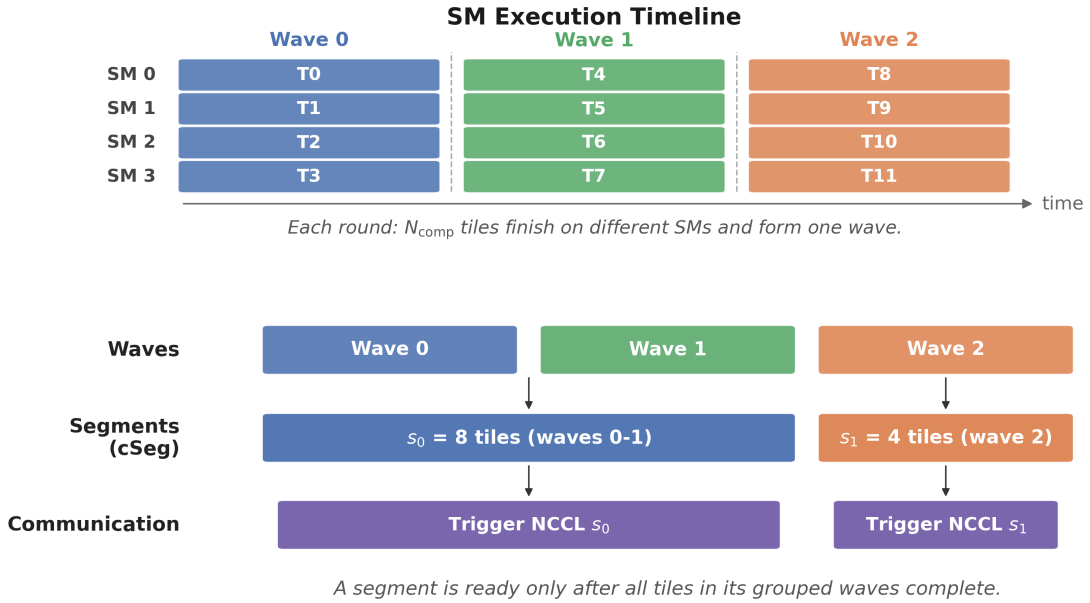

Figure 4.2: Tile, Wave, Segment hierarchy. Tiles completed in the same execution round form a wave, and completed wave groups are used as communication segments for triggering NCCL.

Exploiting this regularity, they partition these T tiles into k consecutive segments:

$$\text{cSeg} = [s_0,\ s_1,\ \ldots,\ s_{k-1}], \qquad \sum_{i=0}^{k-1} s_i = T.$$

And as soon as segment s_i finishes, the $s_i \cdot B_M B_N$ output elements are ready in memory and a partial collective is immediately launched for those elements (Figure 4.2).

Each wave consists of N_{comp} tiles executing in parallel (one per SM). Consecutive waves are grouped into segments; upon completion of each segment, the corresponding NCCL collective is immediately dispatched.

4.2.2 Overlap Configuration Search

Given a GEMM shape (M, N, K), and those tiling parameters (B_M, B_N), and SM allocation N_{compute}, we seek the segmentation that minimizes latency.

4.2.2.1 Tile Order Profiling

Before searching for the communication segmentation, the runtime needs to estimate which output tiles are likely to be completed in each GEMM wave. This information is required because the physical tile completion order may differ from the logical tile index order due to tiling, swizzling, and SM scheduling effects.

This information was obtained in offline probing mode. Each tile records a global completion order when it finishes. This global order can then be converted into a wave index according to the SMs that are assigned to the computation. The probing

process is repeated 10 times to check whether each tile is assigned to the same wave across all runs. These stable tiles are saved as a list:

$$h = [h_0, h_1, \ldots, h_{T'-1}],$$

where $T' \leq T$ denotes the number of stable tiles included in the hint, which describes the predictable tile completion order and is used to build reorder arrays/segments.

4.2.2.2 Communication Configuration

After obtaining a stable tile completion order, the system further searches for a suitable communication segmentation, the `cSeg` configuration. Different `cSeg` configurations change the number of tiles contained in each segment, thereby affecting the communication trigger timing, communication granularity, and the final overlap efficiency.

The optimal `cSeg` configuration depends on:

- the GEMM shape and tiling configuration $(M, N, K, B_M, B_N, \text{Algo})$;
- the number of compute SMs N_{compute} assigned by Green Context;
- the NCCL bandwidth curve, since different message sizes achieve different throughput;
- the communication primitive, such as AllReduce or ReduceScatter, and the world size.

One approach from the FlashOverlap is to enumerate all possible `cSeg` configurations and measure each candidate directly, selecting the one with the lowest latency. This method can provide reliable results, but they also point out that when the number of tiles is large or the GEMM shape is large, the candidate space grows rapidly, leading to excessive tuning.

Therefore, for large search spaces, FlashOverlap and this work adopt a fast search strategy. Specifically, the system first uses a lightweight analytical model to score candidate communication segmentations and estimate their overlapped execution latency. Then, only a small number of promising candidates are retained for the subsequent measurement stage. In this way, the tuning overhead can be reduced while still preserving candidates that are close to the optimal `cSeg` configuration.

4.2.2.3 Tuning and Predictive Search for Configurations

The predictive search consists of the following steps: coarse enumeration, diverse top-$k$selection, neighborhood refinement, last real GPU measurement, and the pseudocode in the Appendix 1.

4.2.2.3.1 Coarse Enumeration. The coarse enumeration makes the search tractable, similarly to FlashOverlap, we introduce a group of waves by:

$$g = \max\left(1, \min\left(\left\lceil \frac{W}{B_{\text{bins}}} \right\rceil, W - 1\right)\right), \qquad \widetilde{W} = \left\lceil \frac{W}{g} \right\rceil.$$

This reduces the search from all partitions of W waves to bounded ordered compositions of $\widetilde{W}$ groups. Then we enumerates the ordered partitions of $\widetilde{W}$, that is:

$$\mathbf{p} = [p_1, \ldots, p_{K_s}]$$

Where each p_i represents the number of coarse waves contained in the ith segment. That converts into tile level:

$$s_i = p_i \cdot g \cdot N_{\text{compute}}, \quad i < K_s,$$

and the last segment takes the remaining tiles:

$$s_{K_s} = T - \sum_{i<K_s} s_i.$$

Also, the candidates with only one segment size are discarded, because in such a case, the communication triggers after all the computation is finished, which means the communication and computation happen in serial. We also discard front-heavy candidates in which the first segment contains more than half of the total number of tiles. Such a large leading segment delays the first communication launch, shortens the available overlap window, and therefore reduces pipeline efficiency.

4.2.2.3.2 Diverse Top-k Selection. After coarse enumeration, each valid candidate is scored by a latency estimator. The estimator first predicts the overlap latency using the NCCL bandwidth curve and a pipeline latency model. For inefficient segmentation schemes, such as those with too much or overly large trailing segmentation, we add an estimated latency cost.

After scoring all candidate partitions, the system retains the top-K diverse candidates for refinement. Diversity is considered in terms of segment count and partition structure so that the refinement stage does not rely on a single potentially inaccurate analytical prediction.

4.2.2.3.3 Neighborhood Refinement. The neighborhood refinement further explores the local neighborhood of each retained candidate. Two types of perturbations are applied: a *shift*, which transfers one coarse-wave unit between two adjacent segments; and a *split*, which bisects a segment into two roughly equal parts. Each perturbed composition is scored by the same latency estimator and used to update the candidate pool. This process is repeated for several rounds, with the diverse top-K candidates from each round seeding the next.

Finally, each selected candidate is benchmarked on GPU and the one with the lowest measured latency is saved as the final `cSeg` configuration.

4.2.3 Overlap Execution

Before runtime execution, the configuration (B_M, B_N, Algo, h, `cSeg`) produced by the offline search (§4.2.2) is loaded from disk and used to initialize the tile reorder array

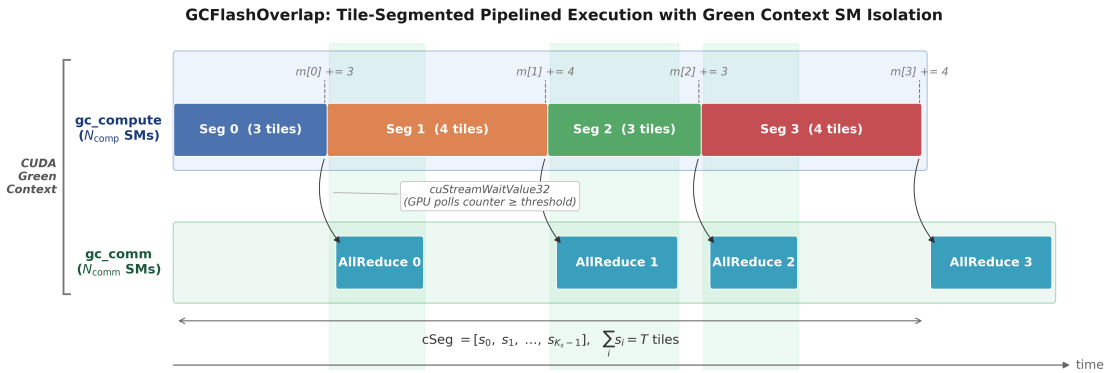

Figure 4.3: GCFlashOverlap execution timeline

and segment tensors on each GPU. Figure 4.3 illustrates the resulting execution timeline.

In the runtime implementation, the hint h is materialized as a device reorder array `RA`, which maps each logical output tile to its packed output position. The segment-size vector `cSeg` is stored in two copies: a host-side copy, used by the runtime to compute communication offsets and message sizes, and a device-side copy, used by the GEMM epilogue to identify the segment of each completed tile. A device array `MM` stores the segment completion counters.

At runtime, the overlap is driven by two independent CUDA streams: `compute_stream`, bound to `gc_compute`, and `comm_stream`, bound to `gc_comm`. Because each stream operates exclusively within its own SM partition, the two can execute concurrently without any SM-level contention.

4.2.3.1 Compute Path

A modified CUTLASS kernel is launched on `compute_stream`. The reorder array derived from the hint h is not used to force CUTLASS to execute tiles in a new order. Instead, the kernel keeps the selected CUTLASS schedule and changes the output mapping in the epilogue. For each logical output tile, the epilogue visitor computes the logical tile id, looks up its packed tile id in the reorder array, and stores the tile to the corresponding packed position in the output tensor. As a result, tiles predicted to become ready in the same communication segment occupy a contiguous memory range, which can be passed directly to NCCL.

After the packed tile store is completed, one representative thread determines which segment contains the packed tile and atomically increments the corresponding per-segment counter m_i in GPU memory. Thus m_i counts completed tiles in segment i, and the communication side can wait for m_i to reach the segment size s_i before launching the collective for that segment.

4.2.3.2 Communication Path

Before execution begins, the CPU pre-enqueues all K_s wait–launch pairs onto `comm_stream`: for each segment i, a `cuStreamWaitValue32` call that hardware-polls the counter m_i until it reaches s_i, followed immediately by the corresponding NCCL collective (AllReduce or ReduceScatter) over the $s_i \cdot B_M B_N$ output elements of that segment. Once all pairs are enqueued the CPU returns; the GPU executes them in order as each counter condition is satisfied, without further CPU involvement.

4.2.3.3 Synchronization and Completion

After all K_s collectives have been issued on `comm_stream`, a CUDA event is recorded on `comm_stream` and `compute_stream` waits on it. This ensures that the fully reduced output is visible to any downstream kernel before execution proceeds, without requiring a host-side barrier.

4.3 Porting FlashOverlap to CUTLASS 3.x for H100 GPUs

The previous sections described FlashOverlap and its signaling-based overlap model. In this section, we focus on the changes needed to make this design work with CUTLASS 3.x GEMM kernels on H100 GPUs. The original FlashOverlap implementation was mainly built around CUTLASS 2.x kernels on A100 GPUs. However, CUTLASS 3.x changes several parts of the GEMM implementation, including the kernel hierarchy, scheduling model, and epilogue structure. In addition, some of the APIs and assumptions used by the original implementation are deprecated or no longer directly compatible with CUTLASS 3.x.

Because of these changes, the original FlashOverlap implementation could not be used by simply recompiling it for H100. The main challenge was the transition from the CUTLASS 2.x GEMM structure to the CUTLASS 3.x GEMM structure used on H100. In particular, the CUTLASS 3.x kernels used in this work rely on newer mechanisms such as TMA-based memory movement, warp-specialized schedules, new tile schedulers, and a different epilogue path. These differences affect how tile completion can be detected and how the output reorder logic can be inserted.

Therefore, we ported FlashOverlap at two levels. First, we modified the GEMM-side implementation so that tile completion can be detected correctly in CUTLASS 3.x kernels on H100. Second, we modified the tool flow used to profile kernels, observe tile completion order, and generate the reorder array used by the overlap runtime.

4.3.1 CUTLASS 3.x GEMM Integration

FlashOverlap depends on knowing when regions of the GEMM output become ready. As discussed in the technical background, GEMM results become visible through the epilogue, where the accumulated values are finalized and written to global memory.

Therefore, in our CUTLASS 3.x port, we keep the same general idea as the original FlashOverlap design: the GEMM mainloop is left unchanged, and the overlap-related logic is added around the epilogue.

The main challenge is deciding when an output tile can safely be marked as ready for communication. The original FlashOverlap implementation assumes that one logical output tile becomes ready after one tile-level completion event. This assumption is not sufficient for the CUTLASS 3.x kernels used in this work. In these kernels, the epilogue work for one logical output tile may be split across multiple epilogue warp groups. Therefore, the same logical tile may be written in multiple parts.

If the communication side is notified after only the first part of the tile is written, it may start communicating data before the full tile is complete. This would break correctness because the communication operation could read incomplete output data. For this reason, the port must track all epilogue completions that belong to the same logical tile and only mark the tile as ready after all expected parts have finished.

To handle this, we replace the original single tile-level signal with an arrival-counting mechanism. Each logical output tile has a counter. Whenever an epilogue warp group finishes its part of that tile, it increments the counter. The tile is considered complete only when the counter reaches the expected number of arrivals. In our implementation, this expected number is obtained from the CUTLASS epilogue metadata, specifically `NumEpilogueWarpGroups`. This ensures that the communication side sees a tile as ready only after the full logical tile has been written.

The implementation is done by wrapping the CUTLASS collective epilogue. The wrapper keeps the original epilogue behavior as much as possible, but adds two pieces of logic. First, in the `store` path, it computes the logical tile identifier and applies the reorder mapping. Instead of writing each tile to its original output position, the wrapper redirects the tile to the packed output position expected by the FlashOverlap runtime. This packed layout makes tiles from the same communication segment contiguous in memory.

Second, in the `store_tail` path, the wrapper updates the completion information. In simple terms, `store` is where the output store is prepared and issued, while `store_tail` is reached when that epilogue store work for one epilogue warp group is being completed. Therefore, `store_tail` is a suitable place to record that one part of the logical tile has finished. At this point, one representative thread increments the arrival counter for the corresponding reordered tile. If the counter reaches `NumEpilogueWarpGroups`, the wrapper marks the logical tile as complete and updates the segment-level counter used by the communication side.

This design preserves the main structure of the original GEMM kernel. The GEMM mainloop and most of the CUTLASS epilogue behavior are left unchanged. The added logic only redirects output tiles into the packed order and tracks when each logical tile is fully complete. The key change is that tile readiness is no longer defined as "the epilogue reached this tile once." Instead, a tile is ready only after all expected epilogue warp-group arrivals for that tile have completed.

4.3.2 Kernel Scheduler Support and Limitations

CUTLASS 3.x provides several scheduling choices for GEMM kernels. Two kernels may have the same high-level GEMM shape but still behave differently depending on their tile shape, cluster shape, number of stages, mainloop schedule, epilogue schedule, and tile scheduler. Therefore, the port also needs to understand and select the correct kernel variant, not only the matrix dimensions.

Our implementation supports the non-cooperative CUTLASS 3.x schedules used in this work. These schedules provide a usable epilogue completion point through the `store` and `store_tail` path, which makes it possible to track tile completion without rewriting the whole GEMM kernel.

We currently do not support cooperative CUTLASS 3.x schedules. The reason is that cooperative schedules do not provide the same clean per-tile completion point through the epilogue path used by our implementation. Since our progress tracking depends on `store_tail`, cooperative schedules would require deeper modification inside the GEMM kernel itself, rather than only modifying the epilogue store path. We therefore leave support for cooperative schedules as future work.

4.3.3 ReduceScatter Limitation

The original FlashOverlap design includes support for ReduceScatter-style overlap. However, in our CUTLASS 3.x port, we do not fully support the original fine-grained ReduceScatter reordering scheme. The main difficulty is that ReduceScatter requires the GEMM output to be arranged according to the final scattered layout expected by the communication operation. This may require row-level or sub-row-level reordering, depending on how the output matrix is partitioned across GPUs.

This becomes difficult in the CUTLASS 3.x kernels used in this work because the epilogue work for one logical output tile may be split across multiple epilogue warp groups. Supporting the original ReduceScatter layout would require adding more detailed row-level remapping inside the epilogue, so that different parts of the tile could be placed according to the final scattered layout. This would make the epilogue significantly more invasive and would also increase the amount of bookkeeping needed during output stores.

For this reason, the current port does not implement full ReduceScatter-specific reordering. Instead, we treat ReduceScatter with the same tile-level reordering mechanism used by the other overlap path. This means that the port can still monitor tile completion and reorder tiles at tile granularity, but it does not provide the complete row-level ReduceScatter layout optimization. We leave a full CUTLASS 3.x ReduceScatter implementation as future work.

4.3.4 Porting the FlashOverlap Offline Tool Flow

Porting the GEMM-side signaling logic is not enough by itself. FlashOverlap also depends on an offline tool flow that selects a suitable GEMM kernel, observes the order in which output tiles complete, and generates the reorder array used by the

runtime. This tool flow is tied to the GEMM implementation and therefore also had to be modified for the CUTLASS 3.x kernels used on H100.

For each GEMM shape, we first use the CUTLASS profiler to find suitable candidate kernels. Unlike the CUTLASS 2.x case, a CUTLASS 3.x kernel is not described only by the CTA tile shape and warp shape. The selected kernel may also depend on the cluster shape, stage count, mainloop schedule, epilogue schedule, and tile scheduler. Therefore, we added a kernel matching step that parses the profiler result and maps it to one of the CUTLASS 3.x kernel variants supported by our implementation.

After selecting a candidate kernel, the offline tool runs the GEMM multiple times and records the order in which output tiles finish. This completion order is important because the runtime uses a packed output layout for communication. More specifically, the reorder array is a mapping from each logical output tile to the position where that tile should be stored in the packed layout. By using this mapping, tiles that are expected to finish close to each other can be placed next to each other in memory, so that each communication segment becomes a contiguous memory region. The reorder array is generated offline for each GEMM shape and problem configuration, because it depends on the tile completion order of that specific configuration.

The tool also uses a parameter called the minimum group size. This parameter defines the minimum number of waves that should be grouped into a communication segment during the search, except possibly for the final remaining segment. Since one wave corresponds to approximately one tile per compute SM, if the minimum group size is G and the number of compute SMs is S, then the nominal monitoring window contains about $G \times S$ tiles. These windows are used to observe which tiles consistently finish within the same region of the execution order.

For the CUTLASS 3.x kernels used in this work, the tile completion order is less stable than in the original A100 implementation. Because these kernels use more complex scheduling mechanisms, some tiles near the boundary of two windows may move between neighboring windows across different runs. Therefore, using the full nominal window directly can produce a reorder array that is too optimistic, because it may assume that boundary tiles are stable even when they are not.

To handle this, the newly developed search tool computes an effective group size from the stable tiles inside the nominal windows. A tile is considered stable for a window only if it appears in that same window in every monitored run. If confirmation is enabled, the tile must also remain stable across the confirmation round. For each window, the tool counts the number of stable tiles and converts this into a number of stable waves by dividing by the number of compute SMs. The final effective group size is the minimum stable wave count across the full windows.

For example, suppose the nominal minimum group size is 10. The tool first monitors windows corresponding to 10 waves of tiles. If only 9 waves are consistently stable inside those windows, then the effective group size becomes 9. The predictive search then uses this effective group size instead of the original nominal group size. In this way, the algorithm keeps the stable part of each nominal window and avoids relying on unstable boundary tiles.

The reorder array is then built from the stable tile order. The stable tiles collected from all windows form the prefix of the reordered layout. Tiles that are not stable in any window are not forced into early communication groups. Instead, they are appended to the end of the reorder array. As a result, unstable tiles are handled in the tail of the execution, where their exact completion order is less important.

This modified offline tool flow makes the port more robust to run-to-run variation in tile completion order. The output of the tool includes the full tile order, the reorder map, the selected effective group size, and the communication segment sizes. These values are then used by the overlap runtime to pack completed tiles into the expected order and decide when communication segments can be launched.

4.4 Collective Communications Using TMA

In this section, we describe the collective communication layer that we developed on top of Hopper's Tensor Memory Accelerator (TMA). The goal of this layer is to support intra-node collective operations, including `AllReduce`, `AllGather`, and `ReduceScatter`, while using fewer SM-side resources than traditional communication kernels.

Existing collective communication libraries usually perform communication by launching GPU kernels that use many CTAs and threads to move data, reduce values, and synchronize progress. This approach is effective, but it also means that communication competes with computation for SMs and scheduling resources. Hopper introduces TMA as a hardware-supported mechanism for asynchronous data movement and reduction. This makes it possible to build collective operations where most of the data movement is issued by a small number of threads and then carried out by the hardware. We call our TMA-based collective communication layer *Oh Overlap* (`ooverlap`).

4.4.1 Group, Node, Buffer, and Peer Visibility

The communication layer separates the system into three main concepts: *groups*, *nodes*, and *buffers*. A *group* represents the set of GPUs that participate in a collective operation. It stores the participating devices, the group size, and the information needed to synchronize the participants. A *node* represents one participant in that group. Each node is associated with one GPU rank and is responsible for launching the local part of the collective operation. A *buffer* represents a memory region that participates in the collective. The buffer may be allocated by `ooverlap`, or it may wrap an existing allocation provided by the user.

A key requirement for our design is that each participant must be able to access the memory regions of the other participants. This is necessary because TMA can only operate on addresses that are visible from the GPU that issues the TMA operation. Therefore, before a collective operation starts, each GPU must have a valid address for both its own local buffer and the buffers owned by the other GPUs.

To support this, we use a small set of memory-management helper functions around

CUDA Virtual Memory Management (VMM). These helpers handle the low-level steps needed to create peer-visible memory regions, including reserving virtual address ranges, creating physical memory allocations, mapping those allocations into the virtual address space, and setting access permissions for the participating GPUs. These operations are hidden behind the buffer abstraction so that the rest of the communication layer does not need to directly manage VMM calls.

For same-process multi-GPU execution, we use VMM to create peer-visible allocations. In this mode, the memory is physically owned by one GPU, but the virtual address mapping and access permissions are set so that the other GPUs in the group can also access the same allocation. This allows the collective layer to treat local and remote buffers in a uniform way: each buffer is simply an addressable memory region, even if the physical memory belongs to another GPU.

For multi-process execution, the implementation also supports sharing memory information between processes. A buffer owned by one process can be exported and then imported by another process. The imported mapping gives the receiving process a valid GPU-accessible pointer to the remote memory. For this part, we reused and adapted helper components from ThunderKittens [35], including the multi-process broker used to share buffer information between processes and the VMM helper functions used for exporting, importing, and mapping memory. These helpers simplify the setup phase and allow different processes to establish a consistent view of the peer buffers.

In both the same-process and multi-process cases, the purpose of the setup phase is to give each participant the addresses it needs before launching the collective kernel: the address of its own local buffer and the mapped addresses of the peer buffers. After this setup, the collective execution logic does not need to know whether a pointer comes from a local allocation, a VMM allocation, or an imported interprocess communication (IPC) mapping. It only sees a set of addressable buffers and can issue TMA operations to local or peer memory through the same interface.

4.4.2 Chunks, Windows, and Tasks

The collective execution is organized around three concepts: *chunks*, *windows*, and *tasks*. These concepts are used to keep the GPU kernels simple. The main design idea is that the CPU-side planning code decides what work should be done, while the GPU kernel only executes the tasks it receives. In other words, the GPU kernel behaves like a worker: it does not decide the collective algorithm, the shard ownership, or the communication pattern. It only receives task descriptions and executes them.

A *chunk* is the smallest unit of data movement or reduction in the communication pipeline. When TMA moves or reduces data, it operates on one chunk at a time. The chunk size is a tuning parameter. A smaller chunk can expose finer-grained progress, while a larger chunk can reduce overhead and improve transfer efficiency.

A *window* is a group of consecutive chunks. Windows are used as a progress unit above individual chunks. Instead of signaling after every chunk, the system can

signal after a full window has completed. This reduces synchronization overhead while still allowing later work to know when a region of data is ready.

A *task* describes a unit of work across nodes. While chunks and windows describe how data is divided inside one operation, a task describes what operation should be performed between buffers. For example, a task may copy a window range from one buffer to another, or reduce a window range from a peer buffer into a local buffer. Therefore, the task acts as the interface between the high-level collective operation and the lower-level TMA execution path, which is described later.

This gives the implementation a simple hierarchy. The high-level collective operation is translated into tasks. Each task covers a range of windows, and each window contains multiple chunks. The GPU kernel receives these tasks and executes the corresponding chunk-level TMA operations. Figure 4.4 illustrates this hierarchy for a two-GPU AllReduce example.

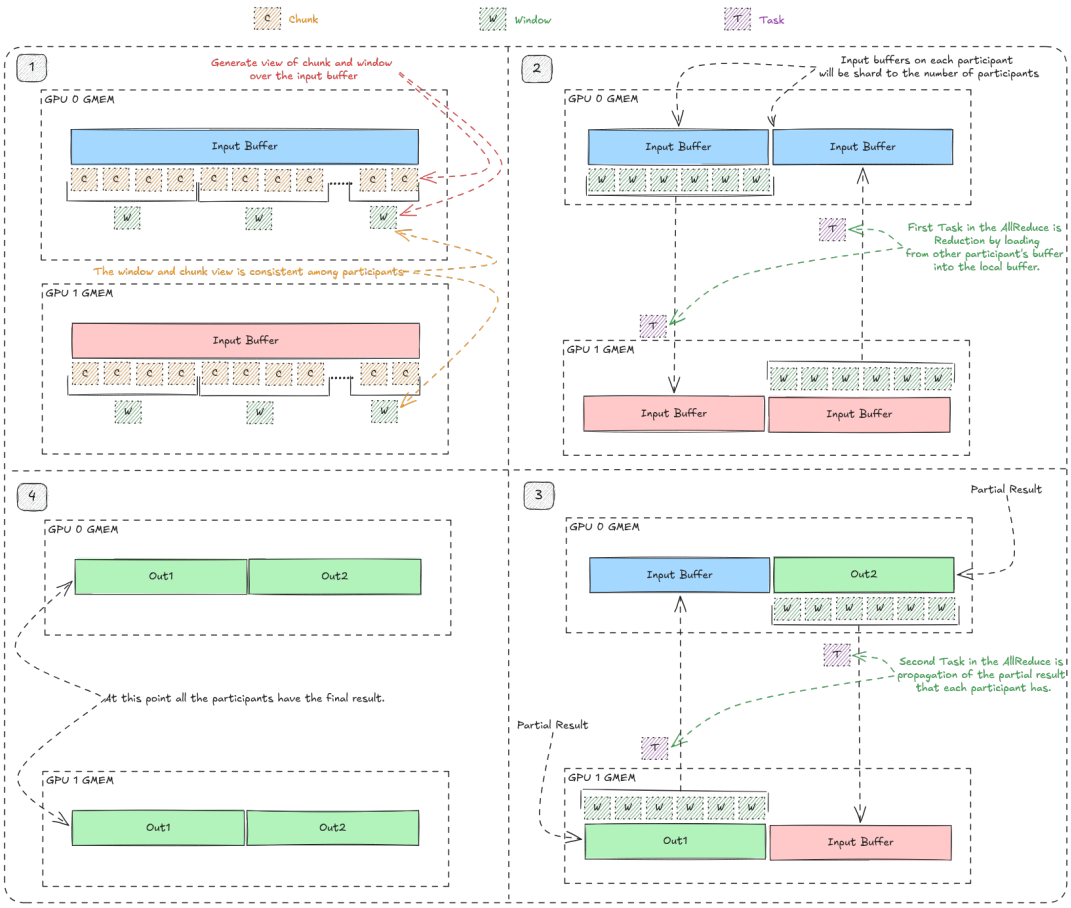

Figure 4.4: Chunk, window, and task hierarchy in a two-participant AllReduce example. Chunks are the smallest data movement units, windows group consecutive chunks, and tasks describe the work performed between participants.

4.4.3 Pipelined TMA Execution

A direct implementation of TMA communication would issue a transfer and then wait for it to finish before issuing the next one. This would waste time because every TMA operation has a completion delay. To reduce this waiting time, our implementation uses a staged pipeline. The low-level wrapper around the TMA PTX instructions[10] is adapted from ThunderKittens[35], while the pipeline organization is built around the needs of our collective kernels.

In this pipeline, a TMA load moves data from global memory(GMEM) into shared memory(SMEM). The global-memory source can be either the local buffer or a peer buffer, depending on the task's details. After the data is available in shared memory, the next step is either a TMA store or a TMA reduction from shared memory back to global memory. The destination can also be either local memory or peer memory.

The pipeline allocates a shared-memory staging buffer with size $D \times C$, where D is the stage depth and C is the chunk size. Each stage stores one chunk in shared memory. Therefore, the product of the stage depth and the chunk size must fit within the shared memory available to each CTA. This constraint is important because increasing the stage depth or chunk size can improve overlap, but it also increases shared-memory usage.

The pipeline uses half of the stages for future loads and the other half for stores or reductions that are currently being completed. In practice, this means the prefetch distance is $D/2$. This choice comes from the completion mechanism of the TMA store path. TMA store completion is ordered like a FIFO: when multiple store or reduction operations are in flight, the kernel can wait for a certain number of the oldest operations to complete while allowing newer operations to remain in flight. For example, if D store or reduction operations are in flight, waiting for $D/2$ operations means that the oldest $D/2$ operations are guaranteed to have completed, while the remaining operations may still be progressing.

Because each shared-memory stage is reused across different chunks, the kernel must ensure that a stage is no longer being used by an outstanding store or reduction before issuing a new TMA load into that stage. Otherwise, the new load could overwrite shared-memory data that is still needed by a previous store or reduction operation. This is why the prefetch distance is limited to $D/2$ without adding extra waiting: half of the stages can be used for future loads, while the other half may still be involved in store or reduction operations.

The execution proceeds as follows. The kernel first issues loads for future chunks up to the prefetch distance. Later, when the current chunk reaches the execution point, its load has already been issued earlier. The kernel waits for the current load to complete, then issues the corresponding store or reduction, and then moves to the next chunk. At the same time, it continues issuing loads for future chunks whenever the corresponding shared-memory stage is safe to reuse.

This pipelined structure hides part of the waiting time required by TMA operations. Loads for future chunks are issued several iterations before their data is needed, and stores or reductions are also allowed to remain in flight. The kernel only waits when

it needs to use loaded data or when it needs to reuse a shared-memory stage for another chunk.

Figure 4.5 shows an example of this pipeline with four stages and ten chunks. In this example, the stage depth is $D = 4$, so the prefetch distance is $D/2 = 2$. The figure shows that loads for later chunks are issued before they are needed, while stores or reductions from earlier chunks are still in flight.

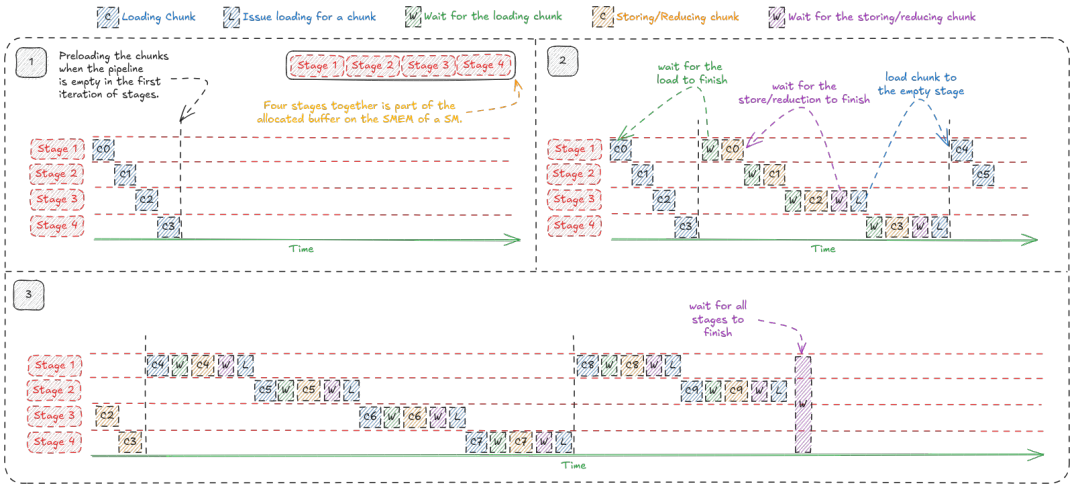

Figure 4.5: Example of the staged TMA pipeline with four stages and ten chunks. The pipeline issues future loads while previous chunks are being stored or reduced, and it waits only when loaded data is needed or when a shared-memory stage must be reused.

4.4.4 Shard-Based Collective Planning

As shown in Figure 4.4, the collective operation is planned over shards of the input buffers. The main planning strategy is based on sharding. For a collective with N participants, the logical tensor is divided into N shards. Each participant becomes responsible for one shard. The participant that owns a shard controls the lifecycle of that shard: it gathers the required data, performs the reduction or copy operations for that shard, and then distributes the result if the collective requires it.

The implementation supports both *in-place* and *out-of-place* collectives. In the in-place case, the input and output use the same buffer. This is attractive because it avoids allocating and moving data through an additional output buffer. However, it also makes synchronization more important, because the same memory region may be read by some participants and written by others during the collective. In the out-of-place case, the input and output buffers are separate, which makes the data dependency easier to reason about, but it introduces additional memory traffic and generally provides lower performance than the in-place path.

The shard-based design reduces the amount of synchronization needed between GPUs. The participants do not need to coordinate after every chunk or every task. Instead, synchronization is mainly needed at the boundaries of the collective operation. Before the collective starts, all participants must make sure that the input

buffers and peer mappings are ready. This is handled through ready signals and a collective epoch. After the collective is launched, each participant must also make sure that all participants have finished the collective before the output is consumed or the input buffer is reused. This final synchronization is especially important for in-place collectives, where early reuse of the buffer could break correctness.

The *plan* is the component that turns a user-level collective operation into a sequence of executable tasks. The user requests an operation such as `AllReduce`, `AllGather`, or `ReduceScatter`. For reduction-based collectives, the user also specifies the reduction operation, such as sum, minimum, or maximum, depending on the supported data type. The plan then uses this information, together with the group size, the identity of the local node, the tensor size, and the selected launch configuration, to decide which tasks must be executed by the local GPU.

The *launch configuration* describes how the communication kernel should run, including the number of CTAs, the number of threads per CTA, the chunk size, the number of chunks per window, the TMA stage depth, and the selected execution path. The execution path may use TMA-based copy, TMA-based reduction, or a thread-based fast copy path, which we describe it later.

The plan itself does not move data. It only describes the work. For example, it decides that a certain window range should be reduced from a peer buffer into the local shard, or that a reduced local shard should be copied back to the peer buffers. The GPU kernel then receives this task list and executes it without needing to understand the full collective algorithm. This separation is useful because more advanced scheduling strategies can be implemented in the planner without changing the GPU kernel. This is important because GPU kernels are harder to modify and maintain, especially when the scheduling policy becomes more complicated.

4.4.5 Direction of TMA Transfers

TMA-based communication can be implemented in two directions. The first direction is to load data from peer memory and store or reduce it into local memory. The second direction is to load data from local memory and store or reduce it into peer memory. Both directions are useful, but they have different synchronization and performance behavior.

In our design, reduction phases usually use the peer-to-local direction. The local GPU loads the corresponding shard from each peer and reduces it into its own local destination. This direction is attractive because the destination is local, the owner of the shard controls the accumulation, and the synchronization is easier to reason about.

For broadcast-like phases, such as the distribution step of `AllReduce` or the main data movement in `AllGather`, the local-to-peer direction is also useful. In this case, the rank that owns a reduced shard writes the result to the corresponding region of the peer buffers. This avoids forcing every peer to separately pull the same shard. The performance trade-off between these directions is evaluated later in the results chapter.

4.4.6 Communication Kernel Variants

The communication layer contains several execution paths because different message sizes and resource budgets favor different strategies.

The main path is the *TMA copy and reduction path*. In this path, a small number of threads issue TMA operations for chunk-sized regions. For copy operations, the kernel loads a chunk from global memory into shared memory and then stores it to the destination global-memory address. For reduction operations, the kernel loads a chunk into shared memory and then uses TMA-supported reduction to update the destination. This path is the main design used to reduce SM-side communication work.

The second path is the *sequential fast global-memory copy path*. This path does not use TMA. Instead, it uses normal GPU threads to perform optimized global-memory copies. This path is useful because, when more CTAs are allowed for communication, a carefully optimized thread-based copy can sometimes provide higher bandwidth than the TMA path. The implementation uses vectorized global-memory loads and stores, loop unrolling, and cache-bypass-style memory accesses to avoid unnecessary L1 allocation. This makes it a useful alternative for pure copy phases such as the distribution part of `AllReduce` or `AllGather`.

The third path is the *fast global-memory add path*. This path performs reduction using thread-level vectorized arithmetic instead of TMA reduction. The implementation includes this path as a building block, but it is not fully integrated into the main collective plans used in this thesis. We therefore treat it as future work for extending the thread-based reduction path.

The implementation also includes an overlapped all-reduce variant. In the basic all-reduce plan, the reduction phase and copy-back phase are executed sequentially. In the overlapped version, part of the CTAs perform the reduction while another part performs the copy-back. These two groups communicate through window-level signals. When the reduction side finishes a window, it publishes a signal. The copy-back side waits for the corresponding window signal and then copies that completed window to the peer buffers. This allows the distribution phase to begin before the full reduction phase has completed.

This window-level signaling is different from global synchronization. The signal is local to a window range, so the copy-back CTAs do not need to wait for the entire shard to finish. This design is useful for overlapping the two phases of `AllReduce` inside the communication kernel itself.

4.4.7 Tuning and Runtime Configuration

The performance of the collective layer depends on both the launch configuration and the selected execution path. These choices affect not only communication latency, but also the amount of GPU resources consumed by the communication kernel. For example, using more CTAs may improve communication bandwidth, but it also consumes more SM resources, which can reduce the resources available for

computation kernels.

To select these parameters, we use an offline tuning process. For each collective operation and data size, the tuner evaluates different launch configurations and records their measured latency. Each configuration specifies how the communication kernel should execute the generated tasks, including its resource usage, chunk and window sizes, and execution path. The best configurations are saved and later used by the runtime.

At runtime, the communication layer retrieves the saved tuning results and selects a configuration based on the requested collective operation and tensor size. The runtime uses the closest tuned data size to the requested size. If no usable tuned entry is found, the implementation falls back to a default configuration. The selection can also respect resource constraints. For example, the user can restrict the maximum number of CTAs or threads that the communication kernel is allowed to use. The runtime filters the saved configurations using these constraints before selecting the final configuration.

The tuning policy supports two selection modes. In the best-performance mode, the runtime chooses the configuration with the lowest measured latency among the valid candidates. In the efficiency mode, the runtime first finds the best latency and then keeps configurations whose latency is within a small tolerance of that best value, such as five percent. Among those near-best configurations, the runtime chooses the one that uses fewer CTAs and fewer threads. This allows the system to trade a small amount of communication performance for lower SM resource usage, which is important when communication is overlapped with GEMM computation.

5

Results

This chapter evaluates the two overlap mechanisms developed in this thesis. We evaluate GCOverlap and compare it with a sequential baseline and FlashOverlap to quantify when CUDA Green Context based SM isolation improves fine-grained GEMM-communication overlap. Then evaluate Oh Overlap on H100, including the CUTLASS 3.x FlashOverlap port, the overhead introduced by reorder and signaling logic, and the performance of the TMA-based communication path compared with NCCL.

5.1 Evaluation of GCOverlap

This section evaluates GCOverlap on A100 GPUs and compares it against a sequential baseline and FlashOverlap across AllReduce and ReduceScatter workloads at two tensor-parallel scales.

5.1.1 Setup

We evaluate GCOverlap on a single multi-GPU node equipped with NVIDIA A100 SXM 80GB GPUs. The experiments are conducted on the Alvis high-performance computing platforms under the NAISS project allocation `NAISS2026-4-335`. Each A100 GPU contains 108 SMs. Unless otherwise stated, all experiments are conducted using GPUs within the same node, connected through NVLink.

The software stack includes CUDA 12.9.1, NCCL 2.27.7. The implementation is built as a CUDA/C++ extension and uses CUTLASS-based GEMM kernels together with NCCL collectives.

We compare three execution schemes: Baseline, FlashOverlap, and GCOverlap.

Table 5.1: Systems under evaluation

System	Description
Baseline	Sequential GEMM then NCCL collective; no overlap
FlashOverlap	Overlap with tile-level pipelining; GEMM and communication share the same SM pool
GCOverlap	FlashOverlap combined with CUDA Green Context SM isolation for GEMM and communication

The baseline follows a sequential execution model. The GEMM kernel is executed first, and the collective communication is launched only after the entire GEMM output has been produced. There is no overlap between computation and communication. FlashOverlap follows the wave-level signaling mechanism of FlashOverlap as we presented in previous chapters. This scheme overlaps computation and communication in time, but both paths still share the same pool of SMs. GCOverlap uses the same fine-grained overlap mechanism as FlashOverlap but places the GEMM path and the communication path into separate GC. A fixed subset of SMs is assigned to GEMM, while the remaining SMs are reserved for signaling and collective communication.

5.1.2 Workload and metric

The evaluated workload consists of a GEMM followed by a collective communication operation. We focus on two representative communication primitives, AllReduce and ReduceScatter, and evaluate them with GCOverlap. To analyze how GEMM shape affects overlap performance, we vary the GEMM dimensions across multiple sweeps. All experiments use one tensor-parallel rank per GPU; therefore, TP=2 and TP=4 correspond to two-GPU and four-GPU settings, respectively.

The primary metric is end-to-end latency in milliseconds, including GEMM, signaling, synchronization, and collective communication. Speedup is reported relative to the sequential baseline. Thus, a speedup greater than 1.0 means that overlap improves performance, while a speedup below 1.0 means that the overlap mechanism is slower than the sequential baseline. Offline profiling and search are not included in the runtime latency because they are performed before deployment and the selected configuration is loaded at runtime. The cSeg configuration selected by the search is a practical heuristic: the search uses a lightweight analytical model to prune the candidate space and then measures a small number of finalists, so the selected configuration may not be globally optimal. The reported speedups therefore reflect this heuristic-chosen configuration rather than an oracle-optimal one.

5.1.3 Overall Performance

The tested GEMM shapes are summarized in Table 5.2. In total, the operator-level evaluation covers around 40 unique GEMM shapes across different tensor-parallel scales and collective primitives. The full shape-level results are listed in Appendix A.2.

Table 5.2: GEMM shape regions in the operator evaluation.

Primitive	**AllReduce**		**ReduceScatter**	
	TP=2	**TP=4**	**TP=2**	**TP=4**
M ($\times$1024)	1–64	1–64	8–64	4–96
N ($\times$1024)	1–48	1–48	4–48	4–32
K ($\times$1024)	1–32	1–32	1–32	1–8

Figure 5.1 reports a few GEMM+AllReduce workloads under TP=2 and TP=4.

These workloads are selected to cover different GEMM shape regimes and tensor-parallel scales. Each GCOverlap bar uses the best measured SM partition for that GEMM shape.

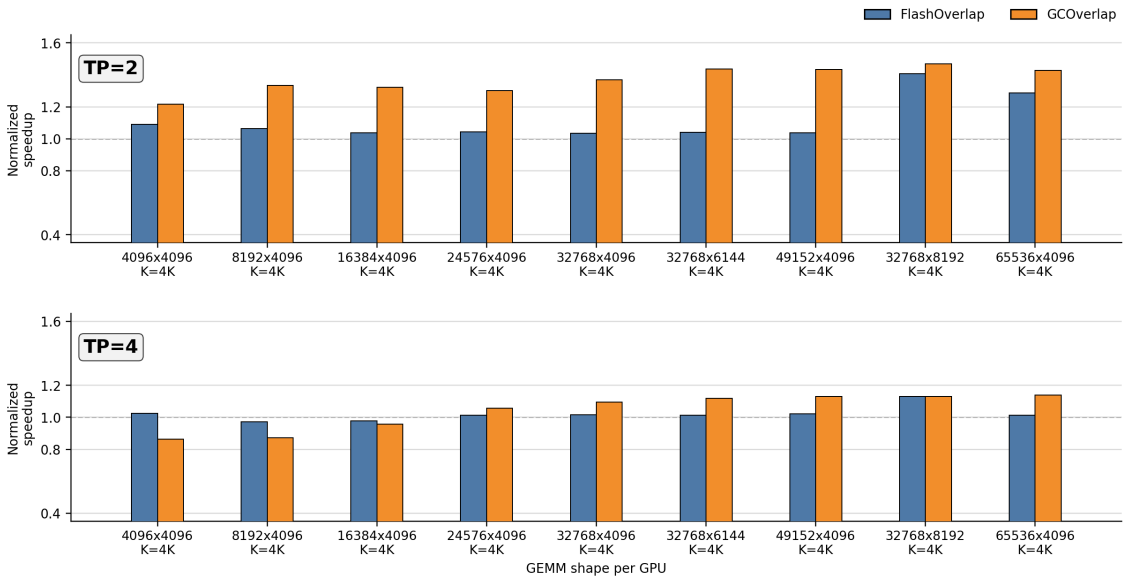

Figure 5.1: Operator-level speedup comparison for GEMM+AllReduce under TP=2 and TP=4.

Under TP=2, GCOverlap improves over FlashOverlap in all selected cases. FlashOverlap achieves 1.03–1.41× speedup over the sequential baseline, while GCOverlap reaches 1.22–1.47×. The additional gain of GCOverlap over FlashOverlap ranges from 6.1 to 39.5 percentage points. Under TP=4, the benefit is more workload-dependent: GCOverlap improves 6 out of the 9 selected cases, and the difference relative to FlashOverlap ranges from -16.2 to +12.8 percentage points. This indicates that, when more GPUs participate, the overlap opportunity becomes more sensitive to the communication path and the selected SM partition.

We also evaluate dimension-wise sweeps over K, M, and N. Figure 5.2 shows the corresponding trends for GEMM+AllReduce under TP=2 and TP=4. The K sweep fixes $M = N = 8192$, the M sweep fixes $N = K = 4096$, and the N sweep fixes $M = 8192$ and $K = 4096$.

Under TP=2, GCOverlap shows the clearest benefit when M or N increases. As M or N increases, the GEMM output size and the amount of communication increase, making communication progress more important. In these cases, isolating communication-side SMs helps reduce contention with the GEMM kernel and improves overlap efficiency.

The benefit is smaller in the K sweep, where increasing K mainly increases GEMM compute work without increasing the output size. The workload becomes more compute-dominated and the relative opportunity for communication overlap decreases.

Under TP=4, the curves are less favorable. GCOverlap improves some large-M

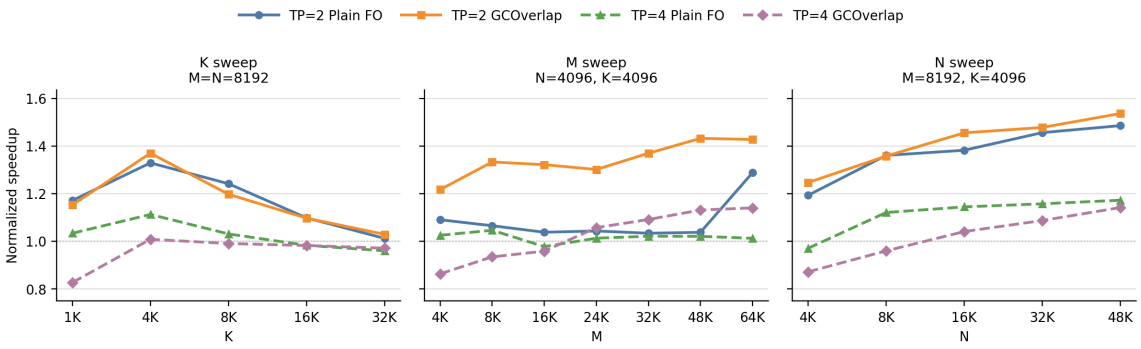

Figure 5.2: Dimension-wise trends for GEMM+AllReduce workloads under TP=2 and TP=4.

cases but often provides smaller or negative gains in the K and N sweeps. This indicates that, at larger tensor-parallel scale, performance is increasingly affected by communication-path latency, NCCL scheduling, and inter-rank synchronization rather than only SM-side contention.

Figure 5.3 shows that GCOverlap improves ReduceScatter overlap on several representative shapes. For TP=2, FlashOverlap achieves a 1.12-1.34× speedup, while GCOverlap reaches 1.13–1.41×, corresponding to an additional gain of 1.2 to 18.9 percent compared with the FlashOverlap. The benefit is less pronounced under TP=4.

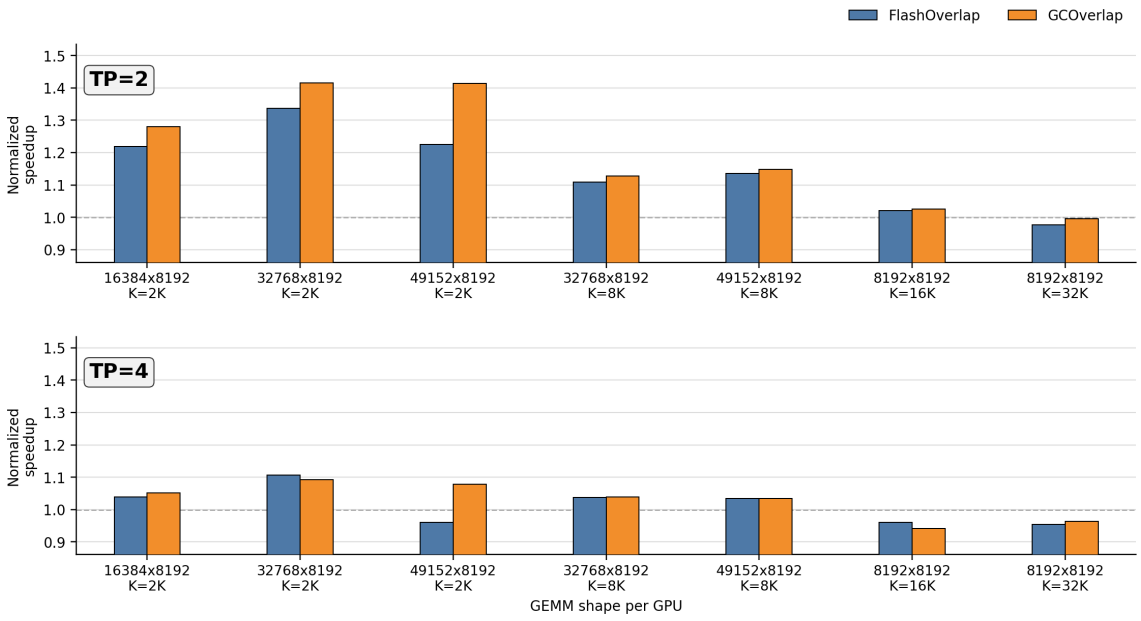

Figure 5.3: Selected operator-level speedups for GEMM+ReduceScatter workloads.

In communication-heavy AllReduce scenarios, reserving a small number of SMs for the communication path can help reduce delays caused by contention with the GEMM kernel. This makes the progress of communication kernels more predictable and can improve the overall overlap efficiency.

For ReduceScatter, the trend is less uniform. Compared with AllReduce, ReduceScatter usually involves a smaller communication volume, so FlashOverlap can already hide a large portion of the communication overhead in many cases. In particular, in the TP=4 results, GCOverlap provides smaller gains and can even be slower than FlashOverlap in some cases. The weaker TP=4 ReduceScatter behavior is discussed separately as a boundary case in Section 5.1.5.

5.1.4 Sensitivity to SM partition

GCOverlap introduces a trade-off that does not exist in FlashOverlap. If too many SMs are reserved for the communication path, the compute partition becomes smaller and the GEMM kernel itself slows down. In contrast, if too few SMs are reserved for communication, NCCL communication kernels may not make progress in time, causing stalls in the overlap pipeline. Therefore, the performance of GCOverlap depends on the choice of the SM partition ratios.

Figure 5.4 illustrates why GCOverlap requires workload-specific SM partitioning. For each workload, we identified the number of communication SMs that yielded the best GCOverlap performance and then counted how many times these optimal communication SMs appeared.

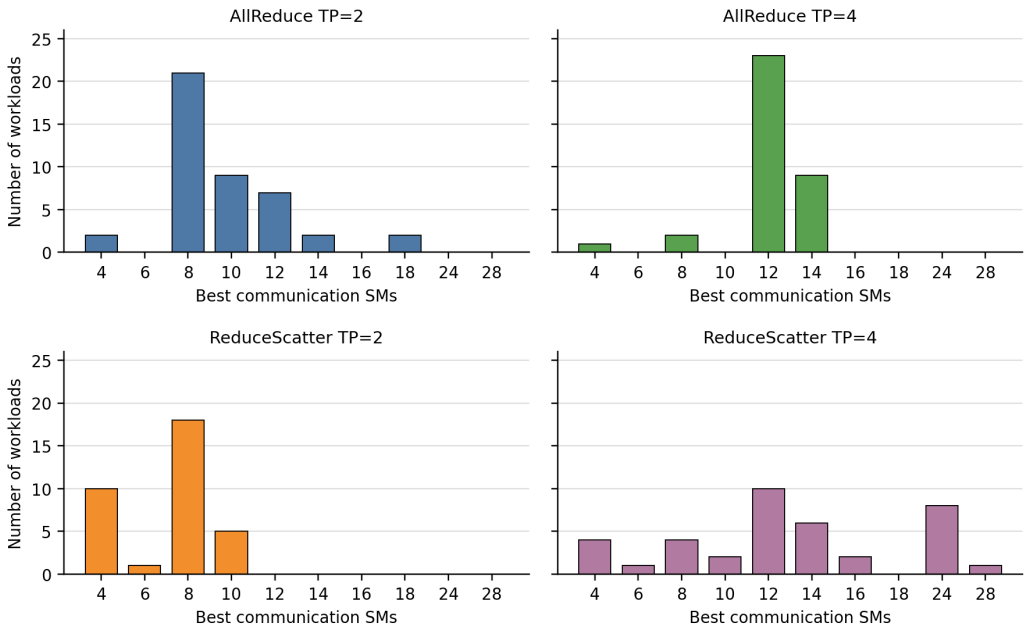

Figure 5.4: Distribution of the best communication-SM allocation across collected workloads.The distribution shows that the best SM partition varies across TP settings and collective primitives.

For AllReduce TP=2, the most common optimal partition is 8 communication SMs, with only a few workloads requiring more SMs. For AllReduce TP=4, the optimal partition tends to be 12 or 14 communication SMs. AllReduce generally needs more communication-side resources when more GPUs participate.

ReduceScatter shows a different pattern. Under TP=2, the selected partition is concentrated around smaller communication partitions: 8 or 4 communication SMs are selected in most cases. Under TP=4, however, the selected partitions become much more dispersed. The best communication-SM count ranges from 4 to 28 SMs. This suggests that ReduceScatter under TP=4 is more shape-sensitive: some workloads can use a small communication partition, while others require a much larger communication-side allocation to make progress.

This variation indicates that the optimal partition depends on both the collective primitive and the tensor-parallel scale. Therefore, GCOverlap uses profiling-based partition selection instead of a single static SM split.

5.1.5 Boundary Cases and Sensitivity Analysis

We also observe several boundary cases where GCOverlap is competitive with, but does not clearly outperform, FlashOverlap. These cases are most common under TP=4, especially for ReduceScatter. With four GPUs, the collective operation involves more and stronger synchronization requirements, so the dominant bottleneck can shift from SM contention to communication-path latency, NCCL scheduling, or cross-GPU synchronization. In this regime, reserving SMs for the communication path does not always improve progress enough to compensate for the smaller GEMM compute partition.

Competitive or non-improving cases also appear in compute-dominated shapes, such as those with large K, as well as in small shapes. When K is large, GEMM accounts for most of the total latency, and the relative contribution of communication becomes smaller. Therefore, even if FlashOverlap or GCOverlap can hide part of the communication, the overall speedup is limited. In this case, further reducing the compute-side SM allocation may slow down GEMM and offset the benefit of communication isolation. For small shapes, both GEMM and communication have short durations, so the fixed overheads introduced by SM isolation, kernel scheduling, and synchronization may not be sufficiently amortized. Similarly, ReduceScatter usually involves less communication volume than AllReduce, leaving less room for GCOverlap to further improve.

5.2 Evaluation of Oh Overlap and CUTLASS 3.x FlashOverlap Port

This section evaluates the two main components developed for the H100 platform. First, we evaluate the CUTLASS 3.x port of FlashOverlap and quantify the overhead introduced by the reorder and signaling logic. Second, we evaluate Oh Overlap, our TMA-based collective communication layer, and compare it against NCCL under different communication and resource settings. Finally, we evaluate the end-to-end operator-level overlap where the ported FlashOverlap GEMM is combined with either NCCL or Oh Overlap.

5.2.1 Experimental Setup

All experiments in this section were performed on the Vera system at C3SE [44] using two NVIDIA H100 GPUs on the same node[1]. The selected GPU pair is connected through NVLink, with the system topology reporting a bonded set of 12 NVLink connections between the two GPUs. Unless otherwise stated, all measurements in this section use this two-GPU intra-node setting.

Table 5.3: Experimental environment for the Oh Overlap and CUTLASS 3.x FlashOverlap evaluation.

Component	**Configuration**
System	Vera, C3SE
GPU type	NVIDIA H100
Number of GPUs	2
GPU interconnect	NVLink, bonded 12-link connection between the selected GPUs
CUDA	12.4
NCCL	2.31

Although the Vera H100 nodes contain eight H100 GPUs, this evaluation is restricted to two GPUs. At the time of writing, Vera did not allow us to allocate more than four GPUs on this system. In addition, using three or four GPUs often resulted in asymmetric topology, where some GPU pairs were connected through NVLink while others communicated through PCIe paths. Our current implementation assumes a symmetric topology, so asymmetric multi-GPU configurations would require a more advanced planning and scheduling policy. We therefore leave larger-scale and asymmetric-topology evaluation for future work.

5.2.2 GEMM Overhead of Reordering and Signaling

Before evaluating end-to-end overlap, we first measure the standalone GEMM performance of the CUTLASS 3.x port. This is important because the overlap mechanism is only useful if the GEMM kernel itself remains competitive. In particular, the FlashOverlap-style port adds output reordering and tile-completion signaling to the GEMM epilogue, so we need to quantify the overhead introduced by this additional logic.

We compare three cases:

- **cuBLAS baseline**: a standard cuBLAS GEMM, used as a highly optimized vendor-library reference.
- **Plain CUTLASS 3.x GEMM**: our CUTLASS 3.x-based GEMM implementation, without output reordering or tile-completion signaling.
- **Reorder+signal GEMM**: the same CUTLASS 3.x-based GEMM implementation, but with FlashOverlap-style output reordering and tile-completion signaling enabled.

[1]The experiments were conducted under the C3SE project allocation `C3SE 2026/1-12`.

Table 5.4 shows representative results from the GEMM sweep. The full shape-level table is provided in Appendix A.3. For the plain and reorder+signal GEMM cases, we exclude cooperative CUTLASS 3.x schedules, as discussed in the methods chapter. The ratio "Plain/cuBLAS" shows how close the selected non-cooperative CUTLASS 3.x GEMM is to cuBLAS. The ratio "Reorder/Plain" shows the overhead introduced by reordering and signaling. Ratios above 1.0 indicate faster execution for the numerator.

Table 5.4: Representative standalone GEMM performance for cuBLAS, plain CUTLASS 3.x GEMM, and reorder+signal GEMM.

Shape (M, N, K)	**cuBLAS ms**	**Plain ms**	**Reorder ms**	**Plain/cuBLAS**	**Reorder/Plain**
4096, 2048, 1024	0.0364	0.0396	0.0595	0.919	0.666
8192, 2048, 1024	0.0726	0.0803	0.1045	0.904	0.768
8192, 2048, 4096	0.2968	0.2811	0.3012	1.056	0.933
8192, 2048, 8192	0.5982	0.5472	0.5820	1.093	0.940
8192, 4096, 1024	0.1430	0.1530	0.1942	0.935	0.788
8192, 8192, 1024	0.2796	0.3155	0.4212	0.886	0.749
16384, 2048, 4096	0.5494	0.5570	0.5993	0.986	0.929
16384, 4096, 2048	0.5389	0.5714	0.6203	0.943	0.921
16384, 8192, 2048	1.1402	1.3467	1.4230	0.847	0.946
32768, 4096, 8192	4.6245	5.5003	5.2783	0.841	1.042
32768, 8192, 4096	4.7162	6.3615	6.2208	0.741	1.023
49152, 8192, 8192	14.9764	17.9477	18.3411	0.834	0.979

The results show two main trends. First, the plain non-cooperative CUTLASS 3.x kernels are usually close to cuBLAS, but they do not always match it. This is expected because cuBLAS is free to select highly tuned internal kernels, while our implementation is restricted to the CUTLASS 3.x schedules supported by the completion-tracking mechanism. Second, the reorder+signal path introduces additional overhead. This overhead comes from output reordering, atomic progress counters, and the synchronization needed to expose tile completion to the communication path. The overhead is more visible for smaller or less compute-dominated shapes, while it becomes less significant when the GEMM work per tile increases.

5.2.3 TMA Copy and Reduction Microbenchmarks

We next evaluate the low-level communication building blocks used by Oh Overlap. These experiments measure the bandwidth of transferring a buffer between two GPUs under CTA limits. For copy, we compare TMA-based copy against a vectorized thread-based copy and NCCL send/receive-style communication. For reduction, we compare TMA-based reduction against a vectorized FP16 reduction kernel. The goal is to understand how much bandwidth each approach can achieve under limited CTA resources and which transfer direction is more suitable for the collective implementation.

Figure 5.5 shows two-GPU copy bandwidth under CTA limits. The experiment evaluates two directions: loading from local memory and storing to remote memory, and loading from remote memory and storing to local memory.

Under an 8-CTA limit, TMA copy is able to utilize the available bandwidth when loading from remote memory and storing into local memory. In this direction, TMA

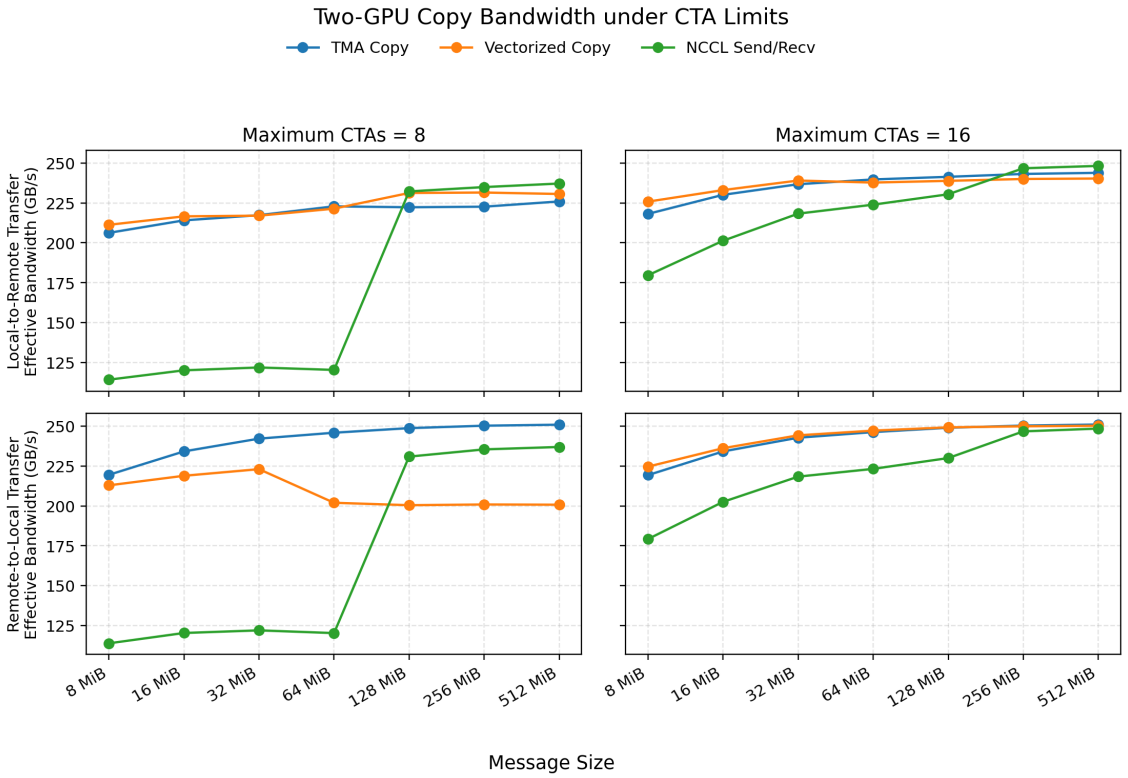

Figure 5.5: Two-GPU copy bandwidth under CTA limits. The comparison includes TMA copy, vectorized copy, and NCCL send/receive-style communication.

is competitive with, and in several cases faster than, the vectorized copy and NCCL paths. In the reverse direction, where the GPU loads local data and stores to remote memory, the three approaches eventually converge to a similar peak bandwidth. However, for smaller buffer sizes in both directions, TMA copy and vectorized copy achieve higher bandwidth than NCCL. Under the 8-CTA limit, NCCL catches up after around 64 MiB, which suggests that it switches to a more bandwidth-efficient internal transfer strategy for larger messages.

With a 16-CTA limit, the direction matters less for copy because all methods have enough CTA resources to approach the available bandwidth. The main difference remains in the small- and medium-size region: TMA and vectorized copy reach high bandwidth earlier, while NCCL needs larger buffers before it catches up.

Figure 5.6 shows the same experiment for FP16 reduction. In this experiment, reduction means element-wise adding the source buffer into the destination buffer. This case is especially important because reduction is the dominant communication component of `AllReduce` and `ReduceScatter`.

For reduction, the transfer direction has a stronger impact. With an 8-CTA limit, TMA reduction is able to utilize nearly the full available bandwidth when it loads from remote memory and reduces into local memory. In this direction, TMA clearly outperforms the vectorized FP16 reduction path. With a 16-CTA limit, both approaches reach similar bandwidth in the remote-to-local direction. In the reverse

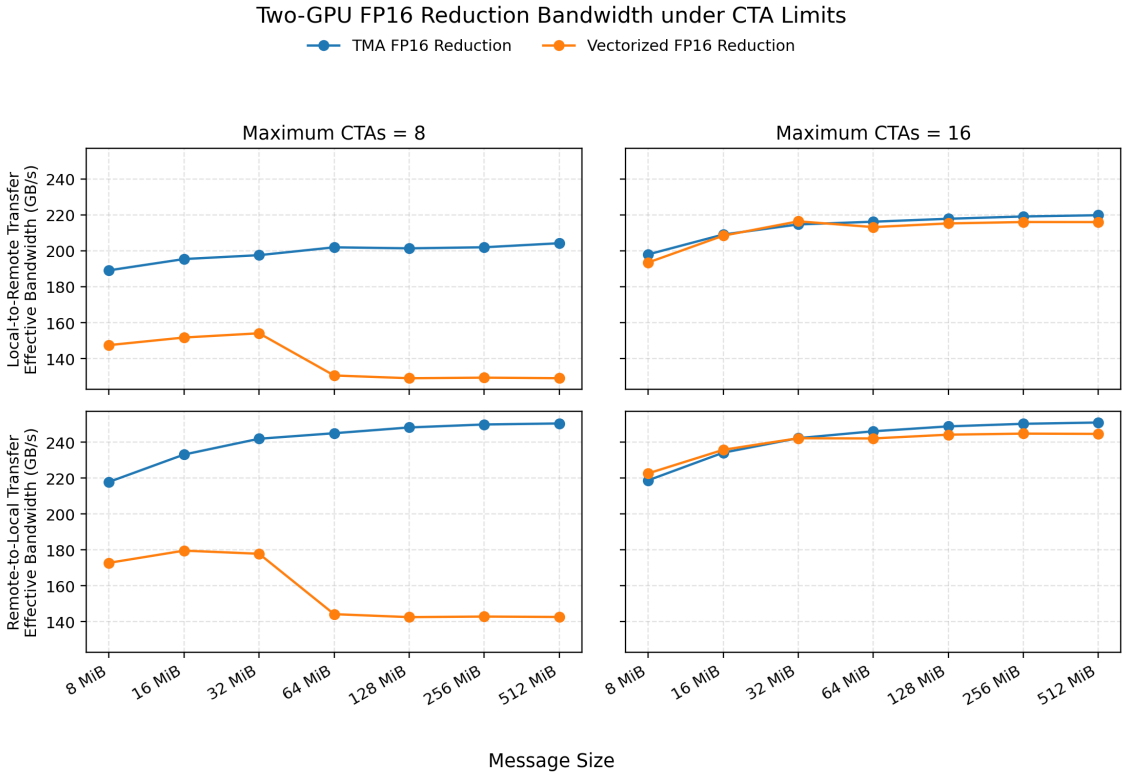

Figure 5.6: Two-GPU FP16 reduction bandwidth under CTA limits. The comparison includes TMA reduction and vectorized FP16 reduction.

direction, where the operation writes and reduces into remote memory, both approaches remain slower. We believe this is caused by the higher cost of remote write-and-reduce behavior compared with accumulating into local memory.

Based on these observations, Oh Overlap prioritizes the remote-to-local direction for reduction-heavy phases. In other words, the GPU that owns a shard loads peer data and reduces it into its local destination. For propagation phases, such as the copy-back phase of `AllReduce`, we use the opposite direction to reduce synchronization between participants. For `AllGather`, where there is no reduction, the implementation uses the direction that gives better copy bandwidth for the corresponding buffer size and CTA budget.

5.2.4 Collective Bandwidth for Large Messages

We evaluate Oh Overlap against NCCL for `AllReduce`, `ReduceScatter`, and `AllGather`. We separate the results into two regimes. For large messages, bandwidth is the main metric. For small messages, latency is the main metric.

Figure 5.7 compares effective per-rank bandwidth for large messages. The plot includes unrestricted execution as well as CTA-limited execution. For each CTA setting, Oh Overlap is compared against NCCL under the same restriction.

Oh Overlap consistently outperforms NCCL for large messages under the tested

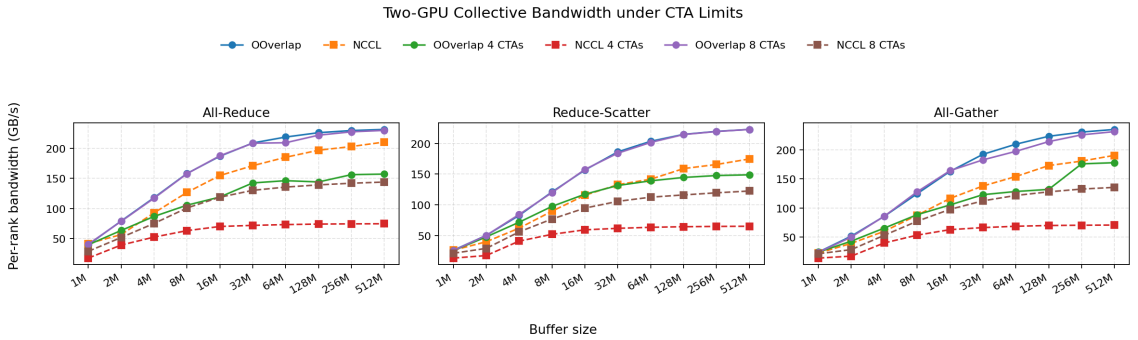

Figure 5.7: Large-message collective bandwidth for Oh Overlap and NCCL under unrestricted, 4-CTA, and 8-CTA settings.

CTA settings. The advantage is especially visible when the communication path is resource-constrained. This is expected because Oh Overlap performs most of its movement through TMA and requires fewer active threads than a conventional thread-driven communication kernel. As a result, it can maintain higher bandwidth when only a small number of CTAs are available.

Figure 5.8 reports the same result as speedup over NCCL for unrestricted execution. The speedup is above one for most medium and large message sizes across the three collective operations.

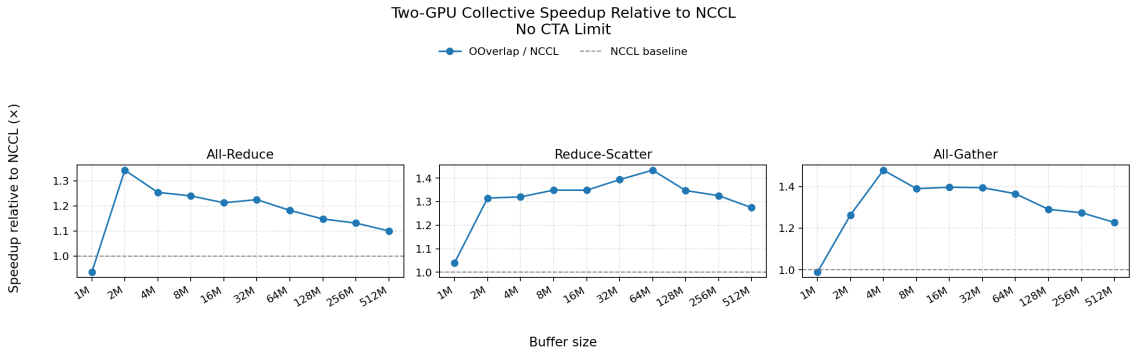

Figure 5.8: Speedup of Oh Overlap over NCCL for large-message collectives under unrestricted CTA usage.

Figures 5.9 and 5.10 show the CTA-limited cases. Under CTA limits, the advantage of Oh Overlap becomes larger because NCCL is more affected by the restricted number of communication CTAs. Across different message sizes and collectives, Oh Overlap provides gains ranging from modest improvements to large speedups in the mid-size and large-message regimes.

The main conclusion from these bandwidth experiments is that Oh Overlap is most effective when the message is large enough to amortize TMA setup costs and when the communication path is resource-constrained. This matches the goal of Oh Overlap: to achieve better bandwidth utilization with only a small number of SM resources, leaving more SMs available for GEMM during overlap.

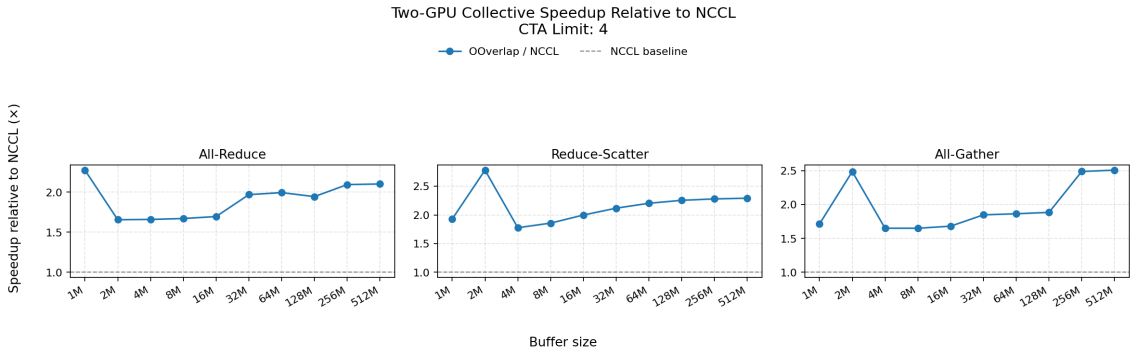

Figure 5.9: Speedup of Oh Overlap over NCCL for large-message collectives with a 4-CTA limit.

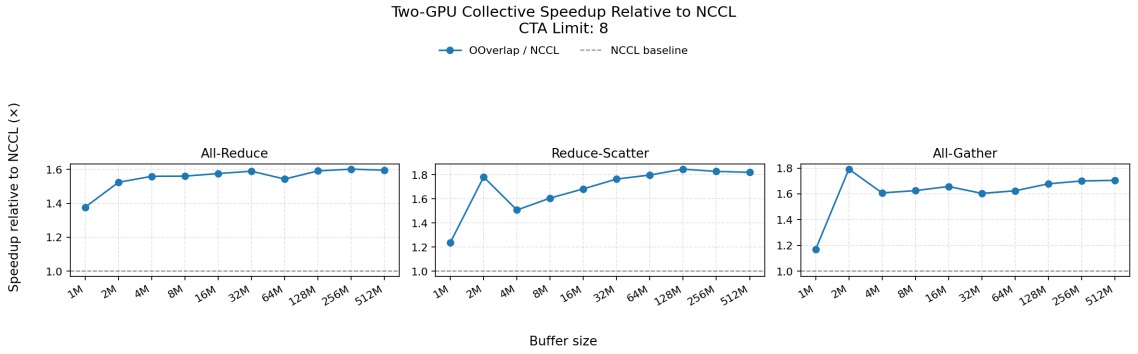

Figure 5.10: Speedup of Oh Overlap over NCCL for large-message collectives with an 8-CTA limit.

5.2.5 Collective Latency for Small Messages

For small messages, the behavior is different. Figure 5.11 reports the latency of Oh Overlap and NCCL for small buffer sizes.

Oh Overlap has higher latency than NCCL for small messages, typically by around 20–30% in the tested range. We believe this happens because TMA is primarily designed for structured bulk data movement. For very small messages, the fixed overhead of setting up and waiting on TMA operations is not sufficiently amortized. NCCL, in contrast, already has highly optimized paths for small collectives. Optimizing Oh Overlap for small messages is therefore left as future work.

5.2.6 Operator-Level Evaluation

We next evaluate operator-level overlap. The operator consists of a GEMM followed by a collective communication operation. The goal is to compare two overlap implementations: the CUTLASS 3.x FlashOverlap port using NCCL as the communication backend, and the same GEMM-side overlap mechanism using Oh Overlap.

The baseline in this experiment is our plain CUTLASS 3.x GEMM followed by NCCL communication after the GEMM finishes. We do not use cuBLAS followed by NCCL as the baseline, because cuBLAS may select cooperative kernels that our current

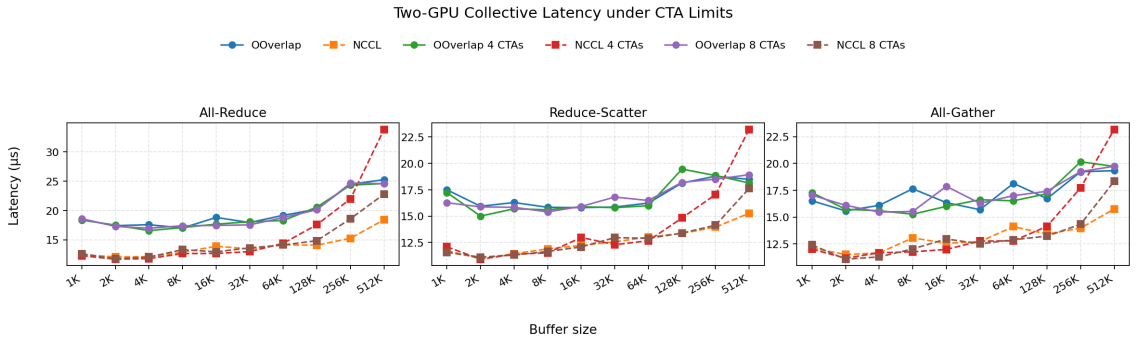

Figure 5.11: Small-message collective latency for Oh Overlap and NCCL under unrestricted, 4-CTA, and 8-CTA settings.

FlashOverlap port does not support. Using cuBLAS would therefore mix the effect of GEMM kernel selection with the effect of overlap. Instead, all systems in Table 5.5 use our CUTLASS 3.x GEMM implementation as the compute component. This makes the comparison focus on the additional cost of reordering and signaling, and on the difference between NCCL and Oh Overlap as the communication backend.

Table 5.5: Operator-level systems compared in the H100 evaluation.

System	**Description**
Non-overlap baseline	Plain CUTLASS 3.x GEMM followed by NCCL communication after GEMM completion
NCCL overlap	Reorder+signal GEMM with FlashOverlap-style overlap using NCCL
Oh Overlap	Reorder+signal GEMM with FlashOverlap-style overlap using Oh Overlap

Figure 5.12 reports the operator-level speedup normalized to the non-overlap baseline for the systems listed in Table 5.5.

The results show that the Oh Overlap backend usually performs better than the NCCL-overlap backend. Once the GEMM-side reordering and signaling are already enabled, replacing NCCL with Oh Overlap often reduces the remaining communication cost. In the best cases, Oh Overlap reaches around 10–40% speedup over the non-overlap baseline, with the maximum speedup being close to 1.4×. Some cases remain close to the baseline because the benefit depends strongly on the GEMM shape and on how much communication can actually be hidden.

This behavior is mainly affected by the balance between computation, communication, and reordering overhead. For smaller K, the GEMM performs less computation per output tile, so the cost of output reordering and signaling becomes more visible. In these cases, the overlap benefit can be limited because the additional bookkeeping is harder to hide. As K becomes larger, each output tile requires more computation, and the operator becomes more compute dominated. In that regime, communication becomes a smaller fraction of the total execution time, so there is less communication time left to hide. The strongest gains therefore appear in the

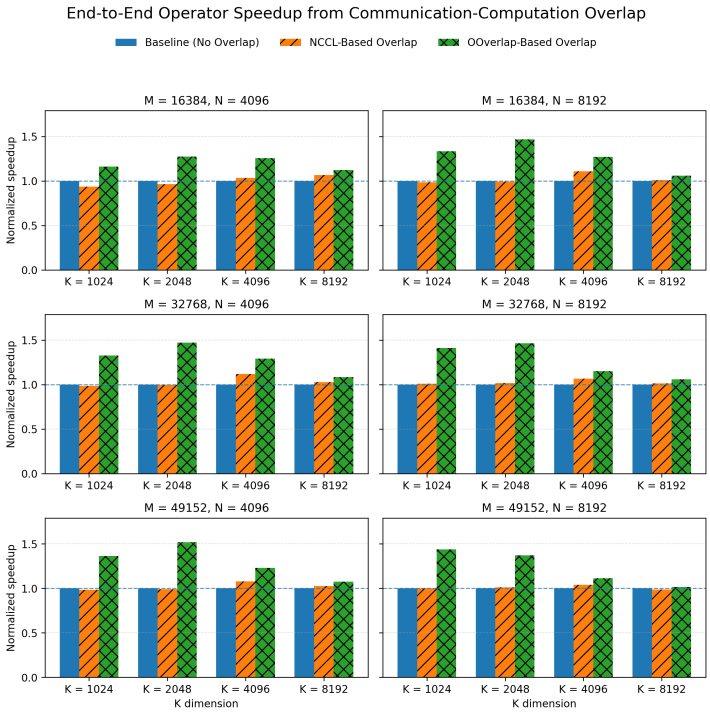

Figure 5.12: Operator-level speedup for GEMM+AllReduce using the CUTLASS 3.x FlashOverlap port with NCCL and Oh Overlap. The baseline is plain CUTLASS 3.x GEMM followed by NCCL AllReduce.

middle region, where the GEMM produces enough output tiles to expose overlap opportunities and the communication cost is still large enough to matter.

6

Conclusion

This thesis presented two techniques for improving GPU resource utilization in computation-communication overlap: GCOverlap and Oh Overlap. GCOverlap uses CUDA Green Context to partition SM resources between GEMM computation and collective communication. Oh Overlap provides a TMA-based collective communication layer that reduces the SM-side cost of intra-node communication.

The evaluation shows that GCOverlap can improve operator-level performance for several GEMM+AllReduce and GEMM+ReduceScatter workloads for TP=2 by reducing interference between compute and communication kernels. Oh Overlap, evaluated on two intra-node NVLink-connected H100 GPUs, shows that TMA-based collectives can provide higher bandwidth under small CTA budgets for large messages, while incurring higher latency for small messages. At the operator level, combining the CUTLASS 3.x FlashOverlap port with Oh Overlap achieves up to 1.4× speedup over the non-overlap baseline for the tested shapes.

6.1 Discussion

The results in this thesis show that computation-communication overlap is not only a scheduling problem, but also a resource-management problem. Starting communication earlier can hide part of the communication latency, but the communication path still consumes GPU execution resources. If communication kernels compete too aggressively with computation kernels, the benefit of overlap can be reduced.

This thesis explored this issue from two directions. GCOverlap reduces direct resource contention by assigning computation and communication to separate SM partitions. Oh Overlap reduces the amount of SM-side work required by the communication backend itself. These two approaches are therefore complementary: one controls how resources are divided, while the other reduces the resources needed by communication.

The results also show that the benefit of overlap depends strongly on the workload shape. When the GEMM is small or less compute dominated, the overhead of reordering, signaling, and synchronization becomes more visible. When the GEMM is very compute dominated, communication becomes a smaller fraction of the total operator time, leaving less communication latency to hide. The strongest gains appear between these two extremes, where the GEMM produces enough tiles to

expose overlap and the communication cost is still large enough to matter.

6.2 Limitations

The results in this thesis are operator-level measurements of isolated GEMM+collective latency. The thesis does not demonstrate end-to-end distributed training acceleration on a real model. The reported speedups therefore describe the latency reduction of isolated GEMM+collective operators, rather than direct model-level throughput improvement. This scope is a deliberate choice: the thesis prioritizes operator-level mechanism validation over full deployment evaluation, aiming to characterize when and why overlap helps before attempting to quantify the training-level impact.

This limitation is mainly due to the project timeline and the engineering effort required to implement and evaluate a complete end-to-end training setup. Although the available GPU systems could run model-level workloads, integrating the proposed methods into a realistic training loop would require additional implementation work beyond the operator-level framework used in this thesis. Therefore, the evaluation focuses on isolated GEMM+collective operators, where the behavior of the proposed mechanisms can be studied more directly.

The hardware setup also limits the generality of the results. Oh Overlap was evaluated only on two H100 GPUs, because the available H100 environment did not support experiments with more than two GPUs. As a result, the observed benefits may not directly transfer to larger intra-node GPU configurations, larger tensor-parallel settings, or H100 systems with more than two participating GPUs.

Another limitation is that the current implementations do not cover all possible kernel schedules and collective variants. For example, the CUTLASS 3.x port does not support cooperative schedules, and the ReduceScatter path does not implement the full fine-grained reordering scheme from the original FlashOverlap design. These limitations do not change the main conclusions, but they leave room for broader support in future work.

6.3 Future Work

There are several directions that follow naturally from the work in this thesis. The most direct extension is to integrate GCOverlap or Oh Overlap into an actual training loop and measure end-to-end step time. This would validate whether the operator-level gains translate into model-level throughput and help identify which shapes and collective patterns arise most frequently in practice.

Second, extending the evaluation to larger intra-node tensor-parallel configurations would clarify where each approach remains useful and where new challenges arise. In particular, evaluating Oh Overlap on more than two H100 GPUs would help show how the TMA-based collective layer behaves as the number of participating GPUs increases.

Another important direction is to extend Oh Overlap with support for `All-to-All` and evaluate it on Mixture-of-Experts (MoE) workloads. MoE models introduce more irregular and communication-intensive execution patterns. Testing Oh Overlap in this setting would help determine whether the TMA-based communication design is also beneficial for more dynamic communication patterns, not only for the collectives studied in this thesis.

Furthermore, future work should consider heterogeneous intra-node environments with different GPU hardware features and communication media. In such environments, collective communication needs more global-aware scheduling to optimize data movement according to the available topology, the hardware acceleration features on each GPU generation, and the communication capacity between different GPU pairs. Supporting this type of environment would require more topology-aware and hardware-aware collective planning.

GCOverlap currently uses a static SM partition that is fixed before execution. A dynamic partitioning scheme that adjusts the communication/compute SM split based on observed progress rates could improve robustness for workloads where the optimal partition varies across iterations.

Another direction is to broaden the CUTLASS kernel support in the FlashOverlap-style port. The current implementation supports only the CUTLASS 3.x schedules needed for the experiments in this thesis. Supporting more CUTLASS kernel variants, including cooperative schedules, would make the port more complete and would allow the overlap mechanism to be evaluated across a wider range of GEMM implementations.

Finally, another promising direction is to combine GCOverlap and Oh Overlap. A hybrid design could reserve dedicated SM resources for communication using Green Context, and run an Oh Overlap-style optimized collective inside the communication partition. This could potentially reduce both SM-side interference and communication overhead, leading to a more robust overlap mechanism across different shapes and tensor-parallel settings.

Bibliography

[1] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[2] A. @. M. Llama Team. "The llama 4 herd: The beginning of a new era of natively multimodal ai innovation." Accessed: 2025-12-12. [Online]. Available: https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

[3] S. Pati, S. Aga, M. Islam, N. Jayasena, and M. D. Sinclair, "T3: Transparent tracking & triggering for fine-grained overlap of compute & collectives," in *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024. DOI: 10.1145/3620665.3640410. [Online]. Available: https://arxiv.org/abs/2401.16677.

[4] K. Punniyamurthy, K. Hamidouche, and B. M. Beckmann, "Optimizing distributed ML communication with fused computation-collective operations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2024. [Online]. Available: https://arxiv.org/abs/2305.06942.

[5] N. Namashivayam, "GPU-centric communication schemes for HPC and ML applications," *arXiv preprint*, 2025. [Online]. Available: https://arxiv.org/abs/2503.24230.

[6] K. Hong et al., "Efficient and adaptable overlapping for computation and communication via signaling and reordering," in *Proceedings of the European Conference on Computer Systems*, ser. EuroSys '26, 2026. DOI: 10.1145/3767295.3769370. [Online]. Available: https://arxiv.org/abs/2504.19519.

[7] NVIDIA, *CUDA C++ Programming Guide*, https://docs.nvidia.com/cuda/cuda-programming-guide/, Accessed: 2026-05-31, 2026.

[8] A. Gordi, *Inside NVIDIA GPUs: Anatomy of high performance matmul kernels*, https://www.aleksagordic.com/blog/matmul, Accessed: 2026-05-28, 2024.

[9] M. Andersch et al., *NVIDIA hopper architecture in-depth*, https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, Accessed: 2026-05-28, 2022.

[10] NVIDIA, *Parallel Thread Execution ISA Documentation*, https://docs.nvidia.com/cuda/parallel-thread-execution/, Accessed: 2026-05-28, 2026.

[11] NVIDIA, *cuBLAS: Basic linear algebra on NVIDIA GPUs*, 2017. [Online]. Available: https://developer.nvidia.com/cublas.

[12] NVIDIA, *CUTLASS: CUDA templates for linear algebra subroutines*, 2017. [Online]. Available: https://github.com/NVIDIA/cutlass.

[13] NVIDIA, *Efficient GEMM in CUDA*, https://docs.nvidia.com/cutlass/latest/media/docs/cpp/efficient_gemm.html, Accessed: 2026-05-28, 2026.

[14] NVIDIA, *GPU performance background user's guide*, 2023. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background.

[15] NVIDIA, *CUTLASS 3.0 GEMM API*, https://docs.nvidia.com/cutlass/media/docs/cpp/gemm_api_3x.html, Accessed: 2026-05-28, 2026.

[16] C. Humpi et al., *CUTLASS 3.x: Orthogonal, Reusable, and Composable Abstractions for GEMM Kernel Design*, https://developer.nvidia.com/blog/cutlass-3-x-orthogonal-reusable-and-composable-abstractions-for-gemm-kernel-design/, Accessed: 2026-05-28, 2023.

[17] NVIDIA, *NVIDIA Collective Communication Library (NCCL) Documentation*, https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html, Accessed: 2026-05-28, 2026.

[18] Keysight Technologies, *How to Accelerate Your PCIe 5 Product Design and Testing*, https://www.keysight.com/blogs/en/tech/educ/pcie-5, Accessed: 2026-05-31, 2025.

[19] B. Slechta, N. Comly, A. Eassa, J. DeLaere, and S. Raj, *NVIDIA NVLink and NVIDIA NVSwitch Supercharge Large Language Model Inference*, https://developer.nvidia.com/blog/nvidia-nvlink-and-nvidia-nvswitch-supercharge-large-language-model-inference/, Accessed: 2026-05-28, 2024.

[20] NVIDIA, *Nvidia h100 pcie gpu product brief*, https://www.nvidia.com/content/dam/en-zz/Solutions/gtcs22/data-center/h100/PB-11133-001_v01.pdf, Accessed: 2026-05-31.

[21] FiberMall, *Understanding nvidia's nvlink and nvswitch evolution: Topology and rates*, https://www.fibermall.com/blog/nvidia-nvlink-and-nvswitch-evolution.htm, Accessed: 2026-05-28, 2025.

[22] NVIDIA, *Nvidia nvswitch: The world's highest-bandwidth on-node switch*, https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, Accessed: 2026-05-31.

[23] NVIDIA, *Megatron bridge parallelisms guide*, https://docs.nvidia.com/nemo/megatron-bridge/latest/parallelisms.html, 2026.

[24] NVIDIA, *Collective Operations – NCCL Documentation*, https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html, 2026.

[25] Xiaozao Jun, *What Are All-Reduce and All-to-All?* https://zhuanlan.zhihu.com/p/1976642212977713852, 2025.

[26] S. Li et al., "PyTorch Distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[27] NVIDIA, *Nvidia multi-instance gpu user guide: Deployment considerations*, https://docs.nvidia.com/datacenter/tesla/mig-user-guide/deployment-considerations.html, 2026.

[28] NVIDIA, *When to use mps: Static sm partitioning*, https://docs.nvidia.com/deploy/mps/when-to-use-mps.html, 2026.

[29] L.-W. Chang et al., "FLUX: Fast software-based communication overlap on gpus through kernel fusion," *arXiv preprint arXiv:2406.06858*, 2024. arXiv: 2406.06858 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2406.06858.

[30] T. Liu et al., "ResCCL: Resource-efficient scheduling for collective communication," in *Proceedings of the ACM SIGCOMM 2025 Conference*, ser. SIGCOMM '25, 2025. DOI: 10.1145/3718958.3750514. [Online]. Available: https://dl.acm.org/doi/10.1145/3718958.3750514.

[31] R. Gond, N. Kwatra, and R. Ramjee, "TokenWeave: Efficient compute-communication overlap for distributed llm inference," *arXiv preprint arXiv:2505.11329*, 2025. arXiv: 2505.11329 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2505.11329.

[32] S. Zheng et al., "TileLink: Generating efficient compute-communication overlapping kernels using tile-centric primitives," *arXiv preprint arXiv:2503.20313*, 2025. arXiv: 2503.20313 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2503.20313.

[33] X. Qiang, Y. Guan, Z. Hu, Y. Ding, and A. Aziz, "AutoOverlap: Enabling fine-grained overlap of computation and communication with chunk-based scheduling," *arXiv preprint arXiv:2601.20595*, 2026. arXiv: 2601.20595 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2601.20595.

[34] S. Zhang et al., "Comet: Fine-grained computation-communication overlapping for mixture-of-experts," *arXiv preprint arXiv:2502.19811*, 2025. arXiv: 2502.19811 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2502.19811.

[35] B. F. Spector, S. Arora, A. Singhal, D. Y. Fu, and C. Ré, "ThunderKittens: Simple, fast, and adorable ai kernels," *arXiv preprint arXiv:2410.20399*, 2024. arXiv: 2410.20399 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2410.20399.

[36] S. H. Sul, S. Arora, B. F. Spector, and C. Ré, "ParallelKittens: Systematic and practical simplification of multi-gpu ai kernels," *arXiv preprint arXiv:2511.13940*, 2025. arXiv: 2511.13940 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2511.13940.

[37] A. Agrawal, S. Aga, S. Pati, and M. Islam, "Conccl: Optimizing ml concurrent computation and communication with gpu dma engines," in *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2025, pp. 1–11. DOI: 10.1109/ISPASS64960.2025.00018.

[38] K. Hamidouche et al., "Gpu-initiated networking for NCCL," *arXiv preprint arXiv:2511.15076*, 2025. arXiv: 2511.15076 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2511.15076.

[39] Z. Hu et al., "Demystifying NCCL: An in-depth analysis of gpu communication protocols and algorithms," *arXiv preprint arXiv:2507.04786*, 2025. arXiv:

2507.04786 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2507.04786.

[40] AMD, *ROCm Communication Collectives Library (RCCL) Documentation*, https://rocm.docs.amd.com/projects/rccl/en/latest/, Accessed: 2026-05-28, 2026.

[41] M. Cowan, S. Maleki, M. Musuvathi, O. Saarikivi, and Y. Xiong, "MSCCL: Microsoft collective communication library," *arXiv preprint arXiv:2201.11840*, 2022. arXiv: 2201.11840 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2201.11840.

[42] C. Hwang et al., "MSCCL++: Rethinking gpu communication abstractions for ai inference," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '26, 2026. DOI: 10.1145/3779212.3790188. [Online]. Available: https://arxiv.org/abs/2504.09014.

[43] NVIDIA, *NVSHMEM Documentation*, https://docs.nvidia.com/nvshmem/, Accessed: 2026-05-28, 2026.

[44] Chalmers Centre for Computational Science and Engineering (C3SE), *Vera*, https://www.c3se.chalmers.se/about/Vera/, Accessed: 2026-05-31.

A

Appendix 1

A.1 Search Algorithm for GCOverlap

Algorithm 1 Predictive cSeg Search

Phase 1: Hint Probing

$\mathcal{H} \leftarrow \emptyset$
for each tiling candidate t **do**
  Run overlap kernel with uniform segments; record tile wave indices over 10 runs
  if tile order is consistent across all runs **then**
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{(t,\ h^{(t)})\}$
  end if
  if $|\mathcal{H}| = 3$ **then break**
  end if
end for
if $\mathcal{H} = \emptyset$ **then** $\mathcal{H} \leftarrow \{(0,\ \emptyset)\}$ ▷ fallback: empty hint
end if

Phase 2: Predictive Search (per hint candidate)

best $\leftarrow$ `null`
for each $(t, h) \in \mathcal{H}$ **do**
  Compute g, $\widetilde{W}$ ▷ grouping factor and normalized wave count
  $\mathcal{P} \leftarrow \emptyset$ ▷ candidate pool
  for each bounded composition $\mathbf{p}$ of $\widetilde{W}$ with $|\mathbf{p}| \le K_{\max}$ **do**
    if $\mathbf{p}$ is valid **then** ▷ no front-heavy or degenerate segments
      Expand $\mathbf{p}$ to tile-level `cSeg`
      $\hat{T} \leftarrow T_{\text{pred}}(\texttt{cSeg}) \cdot \lambda(\texttt{cSeg})$
      $\mathcal{P}$.`insert`(`cSeg`, $\hat{T}$) ▷ keep lower score on collision
    end if
  end for
  $\mathcal{S} \leftarrow$ DiverseTopK($\mathcal{P},\ K$) ▷ at least one per segment count
  for $r = 1$ **to** R **do** ▷ neighborhood refinement
    $\mathcal{N} \leftarrow \emptyset$
    for each `cSeg` $\in \mathcal{S}$ **do**
      Generate neighbors via *shift* and *split*; score each; update $\mathcal{P}$
      $\mathcal{N} \leftarrow \mathcal{N} \cup \{\text{neighbors}\}$
    end for
    $\mathcal{S} \leftarrow$ DiverseTopK($\mathcal{N},\ K$)
  end for
  $\mathcal{M} \leftarrow$ TailDiverseSelect($\mathcal{P},\ K_m$) ▷ cover all ρ buckets
  for each `cSeg` $\in \mathcal{M}$ **do** ▷ GPU measurement
    $d \leftarrow$ Measure($B_M^{(t)}, B_N^{(t)}, \text{Algo}^{(t)}, h,$ `cSeg`)
    if $d <$ *best.d* **then**
      best $\leftarrow (t, h,$ `cSeg`$, d)$
    end if
  end for
end for
return *best*

A.2 GCOverlap Shape-Level Results

This appendix lists the shape-level GCOverlap results used in the operator-level evaluation. For each GEMM shape, if multiple GCOverlap SM partitions were measured, the table reports the partition with the lowest GCOverlap latency. The speedups are normalized to the sequential baseline for the same shape.

A.2.1 AllReduce TP=2

Table A.1: AllReduce TP=2: selected PlainFlashOverlap and GCOverlap configuration per GEMM shape.

Shape (M, N, K)	**Base ms**	**Plain ms**	**Plain spd**	**GC ms**	**GC spd**	**Comm SMs**
(1024, 1024, 1024)	0.100	0.101	0.989	0.125	0.800	14
(1024, 2048, 4096)	0.222	0.246	0.901	0.274	0.810	8
(1024, 4096, 4096)	0.379	0.378	1.003	0.368	1.030	12
(2048, 1024, 1024)	0.146	0.149	0.985	0.173	0.847	14
(2048, 2048, 4096)	0.389	0.383	1.016	0.372	1.046	14
(2048, 4096, 4096)	0.695	0.582	1.194	0.588	1.182	12
(4096, 4096, 4096)	1.216	1.115	1.090	0.999	1.217	14
(4096, 8192, 8192)	3.158	2.574	1.227	2.786	1.133	8
(4096, 16384, 4096)	4.032	3.230	1.248	3.007	1.341	12
(4096, 16384, 8192)	5.853	4.808	1.217	5.022	1.166	8
(8192, 4096, 1024)	1.401	1.290	1.086	1.311	1.068	10
(8192, 4096, 4096)	2.247	2.184	1.029	1.685	1.333	12
(8192, 4096, 8192)	3.206	2.994	1.071	2.833	1.131	12
(8192, 4096, 16384)	5.193	4.653	1.116	4.942	1.051	8
(8192, 4096, 32768)	9.334	8.988	1.038	9.325	1.001	6
(8192, 8192, 1024)	2.680	2.337	1.147	2.326	1.153	10
(8192, 8192, 2048)	3.100	2.602	1.191	2.516	1.232	10
(8192, 8192, 4096)	4.080	3.060	1.333	2.978	1.370	8
(8192, 8192, 8192)	5.852	4.900	1.194	4.776	1.225	8
(8192, 8192, 16384)	9.937	9.049	1.098	9.058	1.097	4
(8192, 8192, 32768)	18.120	17.902	1.012	17.607	1.029	4
(8192, 11008, 4096)	5.354	3.989	1.342	3.748	1.428	12
(8192, 12288, 4096)	5.835	4.356	1.339	4.255	1.371	12
(8192, 16384, 4096)	7.764	5.616	1.382	5.334	1.456	10
(8192, 32768, 4096)	15.250	10.470	1.457	10.316	1.478	8
(8192, 49152, 4096)	22.769	15.323	1.486	14.817	1.537	8
(16384, 4096, 4096)	4.046	3.915	1.034	3.061	1.322	8
(16384, 4096, 8192)	5.964	4.915	1.213	4.978	1.198	10
(16384, 8192, 4096)	7.769	5.648	1.376	5.382	1.444	10
(24576, 4096, 4096)	5.924	5.678	1.043	4.551	1.302	12
(32768, 4096, 4096)	7.842	7.471	1.050	5.555	1.412	10
(32768, 6144, 4096)	11.505	11.051	1.041	8.012	1.436	8
(32768, 8192, 4096)	15.091	10.727	1.407	10.278	1.468	8
(32768, 12288, 4096)	22.785	15.789	1.443	15.803	1.442	8
(49152, 4096, 4096)	11.567	11.005	1.051	8.072	1.433	8
(49152, 8192, 4096)	22.600	17.581	1.285	15.242	1.483	8
(65536, 4096, 4096)	15.261	11.863	1.286	10.689	1.428	8
(65536, 8192, 4096)	30.183	23.480	1.285	20.353	1.483	8

A.2.2 AllReduce TP=4

Table A.2: AllReduce TP=4: selected PlainFlashOverlap and GCOverlap configuration per GEMM shape.

Shape (M, N, K)	**Base ms**	**Plain ms**	**Plain spd**	**GC ms**	**GC spd**	**Comm SMs**
(1024, 1024, 1024)	0.094	0.095	0.997	0.171	0.550	14
(1024, 1024, 4096)	0.142	0.145	0.977	0.238	0.598	14
(1024, 4096, 1024)	0.198	0.199	0.995	0.337	0.587	14

Shape (M, N, K)	**Base ms**	**Plain ms**	**Plain spd**	**GC ms**	**GC spd**	**Comm SMs**
(2048, 1024, 1024)	0.139	0.144	0.968	0.236	0.591	12
(2048, 4096, 4096)	0.607	0.562	1.080	0.705	0.860	18
(4096, 4096, 4096)	0.993	0.969	1.025	1.150	0.863	12
(8192, 4096, 4096)	1.819	1.759	1.034	1.905	0.955	12
(8192, 8192, 1024)	1.758	1.701	1.034	2.128	0.826	14
(8192, 8192, 4096)	3.196	2.873	1.113	3.170	1.008	14
(8192, 8192, 8192)	5.202	5.047	1.031	5.252	0.991	8
(8192, 8192, 16384)	9.284	9.445	0.983	9.452	0.982	8
(8192, 8192, 32768)	17.613	18.351	0.960	18.142	0.971	4
(8192, 16384, 4096)	6.129	5.356	1.144	5.889	1.041	12
(8192, 32768, 4096)	11.853	10.242	1.157	10.904	1.087	12
(8192, 49152, 4096)	17.802	15.183	1.172	15.592	1.142	12
(16384, 4096, 4096)	3.223	3.217	1.002	3.102	1.039	12
(16384, 8192, 2048)	4.136	3.353	1.234	4.022	1.028	12
(16384, 8192, 4096)	6.114	5.383	1.136	5.594	1.093	12
(24576, 4096, 4096)	4.659	4.599	1.013	4.406	1.057	12
(32768, 4096, 4096)	6.136	6.140	0.999	5.594	1.097	12
(32768, 6144, 4096)	8.887	8.862	1.003	7.936	1.120	12
(32768, 8192, 1024)	6.090	5.910	1.030	6.120	0.995	12
(32768, 8192, 2048)	7.893	6.305	1.252	6.900	1.144	12
(32768, 8192, 4096)	11.868	10.513	1.129	10.491	1.131	12
(32768, 12288, 4096)	17.767	15.470	1.148	15.388	1.155	12
(49152, 4096, 4096)	9.008	8.824	1.021	7.963	1.131	12
(49152, 8192, 2048)	11.617	9.172	1.267	10.193	1.140	12
(65536, 4096, 4096)	11.920	11.817	1.009	10.454	1.140	12

A.2.3 ReduceScatter TP=2

Table A.3: ReduceScatter TP=2: selected PlainFlashOverlap and GCOverlap configuration per GEMM shape.

Shape (M, N, K)	**Base ms**	**Plain ms**	**Plain spd**	**GC ms**	**GC spd**	**Comm SMs**
(4096, 16384, 4096)	3.276	2.633	1.244	2.881	1.137	8
(4096, 16384, 8192)	5.218	4.699	1.111	4.824	1.082	4
(8192, 4096, 1024)	1.012	0.896	1.129	0.869	1.164	10
(8192, 4096, 4096)	1.826	1.665	1.097	1.566	1.166	8
(8192, 4096, 8192)	2.839	2.768	1.026	2.738	1.037	4
(8192, 4096, 16384)	4.914	4.708	1.044	4.690	1.048	4
(8192, 4096, 32768)	9.133	8.985	1.016	8.876	1.029	4
(8192, 8192, 1024)	1.841	1.505	1.224	1.523	1.210	8
(8192, 8192, 4096)	3.237	2.692	1.202	2.668	1.214	10
(8192, 8192, 8192)	5.256	4.741	1.109	4.816	1.091	4
(8192, 8192, 16384)	9.224	9.037	1.021	8.997	1.025	4
(8192, 8192, 32768)	17.554	17.972	0.977	17.632	0.996	4
(8192, 11008, 4096)	4.273	3.407	1.254	3.611	1.183	10
(8192, 12288, 4096)	4.689	3.837	1.222	3.993	1.174	8
(8192, 16384, 4096)	6.192	5.018	1.234	5.151	1.202	10
(8192, 32768, 4096)	12.051	9.514	1.267	9.671	1.246	8
(8192, 49152, 4096)	18.122	14.191	1.277	14.348	1.263	8
(16384, 4096, 4096)	3.304	3.186	1.037	2.716	1.216	8
(16384, 4096, 8192)	5.242	4.687	1.118	4.726	1.109	4
(16384, 8192, 2048)	4.345	3.642	1.193	3.222	1.349	8
(16384, 8192, 4096)	6.144	5.290	1.161	4.965	1.237	10
(16384, 8192, 8192)	10.428	9.220	1.131	9.199	1.134	4
(32768, 4096, 4096)	6.284	5.310	1.183	5.176	1.214	6
(32768, 8192, 2048)	8.301	6.220	1.334	5.868	1.415	8
(32768, 8192, 4096)	12.071	10.017	1.205	9.635	1.253	8
(32768, 8192, 8192)	20.340	18.396	1.106	17.928	1.135	4
(49152, 4096, 4096)	9.190	8.923	1.030	7.435	1.236	4
(49152, 8192, 2048)	12.383	10.111	1.225	8.758	1.414	8
(49152, 8192, 4096)	18.229	15.336	1.189	14.486	1.258	8
(49152, 8192, 8192)	31.349	27.591	1.136	27.293	1.149	4
(65536, 4096, 4096)	12.174	10.332	1.178	9.793	1.243	8
(65536, 8192, 4096)	24.358	20.446	1.191	19.460	1.252	8

A.2.4 ReduceScatter TP=4

Table A.4: ReduceScatter TP=4: selected PlainFlashOverlap and GCOverlap configuration per GEMM shape.

Shape (M, N, K)	**Base ms**	**Plain ms**	**Plain spd**	**GC ms**	**GC spd**	**Comm SMs**
(4096, 8192, 1024)	0.725	0.655	1.108	0.710	1.022	24
(4096, 8192, 2048)	1.027	0.998	1.029	1.073	0.957	24
(4096, 8192, 8192)	2.627	2.637	0.996	2.856	0.920	12
(4096, 16384, 4096)	2.785	2.665	1.045	3.069	0.907	12
(4096, 16384, 8192)	4.804	4.865	0.987	5.184	0.927	10
(8192, 8192, 1024)	1.352	1.188	1.137	1.282	1.054	24
(8192, 8192, 2048)	1.825	1.635	1.116	1.829	0.998	12
(8192, 8192, 4096)	2.786	2.698	1.033	3.077	0.905	12
(8192, 8192, 8192)	4.775	4.829	0.989	5.158	0.926	14
(8192, 8192, 16384)	8.870	9.229	0.961	9.428	0.941	4
(8192, 8192, 32768)	17.156	17.983	0.954	17.817	0.963	4
(8192, 11008, 4096)	3.658	3.496	1.046	3.875	0.944	8
(8192, 12288, 4096)	4.025	3.896	1.033	4.200	0.958	12
(16384, 4096, 4096)	2.771	2.731	1.015	3.090	0.897	14
(16384, 4096, 8192)	4.815	4.723	1.020	5.130	0.939	8
(16384, 8192, 1024)	2.376	2.385	0.996	2.090	1.137	24
(16384, 8192, 2048)	3.328	3.205	1.038	3.164	1.052	14
(16384, 8192, 4096)	5.345	5.068	1.055	5.455	0.980	10
(16384, 8192, 8192)	9.661	9.319	1.037	9.629	1.003	6
(24576, 8192, 1024)	3.367	3.549	0.949	3.019	1.115	24
(24576, 8192, 2048)	4.802	4.279	1.122	4.581	1.048	14
(32768, 4096, 4096)	5.267	5.070	1.039	5.349	0.985	8
(32768, 8192, 1024)	4.295	4.446	0.966	3.709	1.158	28
(32768, 8192, 2048)	6.319	5.714	1.106	5.784	1.092	12
(32768, 8192, 4096)	10.362	9.845	1.052	10.217	1.014	12
(32768, 8192, 8192)	19.340	18.632	1.038	18.611	1.039	4
(40960, 8192, 1024)	5.387	5.285	1.019	4.449	1.211	24
(49152, 8192, 1024)	6.378	6.261	1.019	7.454	0.856	12
(49152, 8192, 2048)	9.359	9.748	0.960	8.680	1.078	16
(49152, 8192, 4096)	15.515	14.656	1.059	14.863	1.044	8
(49152, 8192, 8192)	28.897	27.923	1.035	27.947	1.034	4
(65536, 4096, 2048)	6.342	5.691	1.114	5.900	1.075	14
(65536, 8192, 1024)	8.413	8.306	1.013	8.279	1.016	16
(65536, 8192, 2048)	12.357	12.067	1.024	11.230	1.100	12
(98304, 8192, 2048)	18.511	19.285	0.960	17.327	1.068	12

A.3 Full GEMM Overhead Results

This appendix reports the full standalone GEMM sweep used in Section 5.2.2. The table compares cuBLAS, plain CUTLASS 3.x GEMM, and CUTLASS 3.x GEMM with FlashOverlap-style output reordering and tile-completion signaling. Ratios above 1.0 indicate faster execution for the numerator.

Table A.5: Full standalone GEMM performance for cuBLAS, plain CUTLASS 3.x GEMM, and reorder+signal GEMM.

Shape (M, N, K)	**cuBLAS ms**	**Plain ms**	**Reorder ms**	**Plain/cuBLAS**	**Reorder/Plain**
(4096, 2048, 1024)	0.0364	0.0396	0.0595	0.919	0.666
(8192, 2048, 1024)	0.0726	0.0803	0.1045	0.904	0.768
(8192, 2048, 2048)	0.1471	0.1471	0.1652	1.000	0.890
(8192, 2048, 4096)	0.2968	0.2811	0.3012	1.056	0.933
(8192, 2048, 8192)	0.5982	0.5472	0.5820	1.093	0.940
(8192, 4096, 1024)	0.1430	0.1530	0.1942	0.935	0.788
(8192, 4096, 2048)	0.2773	0.2846	0.3154	0.974	0.902
(8192, 4096, 4096)	0.5548	0.5806	0.6113	0.956	0.950
(8192, 4096, 8192)	1.1045	1.1867	1.2102	0.931	0.981
(8192, 8192, 1024)	0.2796	0.3155	0.4212	0.886	0.749
(8192, 8192, 2048)	0.5426	0.6555	0.7093	0.828	0.924
(8192, 8192, 4096)	1.1296	1.3081	1.3690	0.864	0.956

Shape (M, N, K)	**cuBLAS ms**	**Plain ms**	**Reorder ms**	**Plain/cuBLAS**	**Reorder/Plain**
(8192, 8192, 8192)	2.3092	2.6215	2.9384	0.881	0.892
(16384, 2048, 1024)	0.1445	0.1576	0.2015	0.917	0.782
(16384, 2048, 2048)	0.2770	0.2844	0.3124	0.974	0.910
(16384, 2048, 4096)	0.5494	0.5570	0.5993	0.986	0.929
(16384, 2048, 8192)	1.1115	1.1334	1.1642	0.981	0.974
(16384, 4096, 1024)	0.2829	0.3094	0.3947	0.914	0.784
(16384, 4096, 2048)	0.5389	0.5714	0.6203	0.943	0.921
(16384, 4096, 4096)	1.1130	1.2015	1.2458	0.926	0.964
(16384, 4096, 8192)	2.2507	2.3538	2.5323	0.956	0.930
(16384, 8192, 1024)	0.5638	0.6309	0.8554	0.894	0.738
(16384, 8192, 2048)	1.1402	1.3467	1.4230	0.847	0.946
(16384, 8192, 4096)	2.3115	2.6524	2.9898	0.872	0.887
(16384, 8192, 8192)	4.5786	5.2877	5.3414	0.866	0.990
(32768, 2048, 1024)	0.2895	0.3208	0.4215	0.902	0.761
(32768, 2048, 2048)	0.5605	0.5848	0.6435	0.958	0.909
(32768, 2048, 4096)	1.1138	1.1974	1.2316	0.930	0.972
(32768, 2048, 8192)	2.3043	2.3516	2.5224	0.980	0.932
(32768, 4096, 1024)	0.5734	0.6285	0.8321	0.912	0.755
(32768, 4096, 2048)	1.1450	1.2359	1.2653	0.927	0.977
(32768, 4096, 4096)	2.3286	2.4870	2.7660	0.936	0.899
(32768, 4096, 8192)	4.6245	5.5003	5.2783	0.841	1.042
(32768, 8192, 1024)	1.1805	1.3030	1.6649	0.906	0.783
(32768, 8192, 2048)	2.3602	2.8331	3.2922	0.833	0.861
(32768, 8192, 4096)	4.7162	6.3615	6.2208	0.741	1.023
(32768, 8192, 8192)	9.9277	11.9228	12.4415	0.833	0.958
(49152, 2048, 2048)	0.8255	0.9097	0.9712	0.907	0.937
(49152, 2048, 4096)	1.7070	1.8094	1.8894	0.943	0.958
(49152, 2048, 8192)	3.4554	3.6983	4.3077	0.934	0.859
(49152, 4096, 1024)	0.9080	0.9594	1.2387	0.946	0.775
(49152, 4096, 2048)	1.7318	1.8314	1.9586	0.946	0.935
(49152, 4096, 4096)	3.5522	4.0310	4.5258	0.881	0.891
(49152, 4096, 8192)	7.0908	8.2357	8.4533	0.861	0.974
(49152, 8192, 1024)	1.8355	2.0129	2.5340	0.912	0.794
(49152, 8192, 2048)	3.5655	4.4950	4.8972	0.793	0.918
(49152, 8192, 4096)	7.0225	9.1005	9.3774	0.772	0.971
(49152, 8192, 8192)	14.9764	17.9477	18.3411	0.834	0.979