DITRON: A FLEXIBLE AND VERSATILE DISTRIBUTED TENSOR COMPILER FOR LLM

Anonymous authors
Paper under double-blind review

ABSTRACT

As the performance scaling of individual devices slows down, distributed system acceleration has emerged as the mainstream. With the increasing maturity of parallel optimizations (e.g., tensor parallelism, sequence parallelism, expert parallelism) in recent years, researchers have found that the key to scalability lies in optimizing the overlap between computation and communication. Developing sophisticated overlapping kernels is challenging and exceeds the capabilities of most researchers, hindering the development of model architectures and parallel strategies. To address this issue, we present DITRON, a flexible and versatile distributed compiler that offers high-level programming interfaces for overlapping kernels. DITRON provides programming abstractions at three levels: (1) fine-grained tile-level programming within the scale-up domain; (2) chunk-level data transfer for the scale-out domain; (3) task-level distributed MegaKernel generation for entire LLM. DITRON inherits Triton's programming model, enabling the transformation of original Triton kernels into parallel kernels with minimal modifications to the source code. Evaluation results show that overlapping kernels developed with DITRON are $1.27\times$ –$19.18\times$ faster than non-overlapping versions, and even outperform expert-tuned CUDA libraries by 6%–30%. End-to-end inference yields a 5%–30% speedup over vLLM. Moreover, DITRON has been validated for training tasks on over 10,000 GPUs, proving robust capabilities for large-scale industrial deployment and saving millions of GPU hours monthly.

1 INTRODUCTION

The speed of large language model (LLM) training depends on the running speed of distributed programs; only with the support of high-speed training can models with diverse architectures be put into practical application (Dubey et al., 2024; Comanici et al., 2025; Rivière et al., 2024; DeepSeek-AI et al., 2024b; Qwen-Team, 2024; OpenAI, 2023; Anthropic, 2024). In the past, compiler work (such as Triton (Tillet et al., 2019) and TileLang (TileLang-Team, 2025)) and libraries (such as CUTLASS/CuTeDSL (Nvidia, 2022) and ThunkderKitten (Spector et al., 2024)) focus on finding ways to improve kernel performance on a single device, while communication in distributed clusters relied on separate libraries (NVIDIA, 2024) or compilers (Shah et al., 2025). As the scale of the cluster expands, communication overhead gradually surpasses computation overhead (according to Chang et al. (2024), communication can take $20\% - 80\%$ runtime in training and inference), diluting the benefits gained from improved computation kernel performance. As a result, it is necessary to jointly optimize computation and communication and overlap their latency with each other.

Although previous work (Jangda et al., 2022; Google, 2025; Wang et al., 2023; Chang et al., 2024; Zheng et al., 2025; Zhang et al., 2025; Zhao et al., 2025; Gond et al., 2025) provides domain-specific languages (DSL) or hand-tuned kernels for overlapping, users still struggle to develop new kernels based on these tools. This difficulty arises from rigid DSL constraints and the high level of expertise required for low-level kernel programming. The root cause behind this challenge is that these frameworks lack suitable programming abstractions that align well with both the algorithmic characteristics of LLMs and the hardware attributes of distributed clusters.

The major difference between distributed programming and single-device programming lies in the existence of non-uniform memory access (NUMA), which can be viewed as two additional levels of

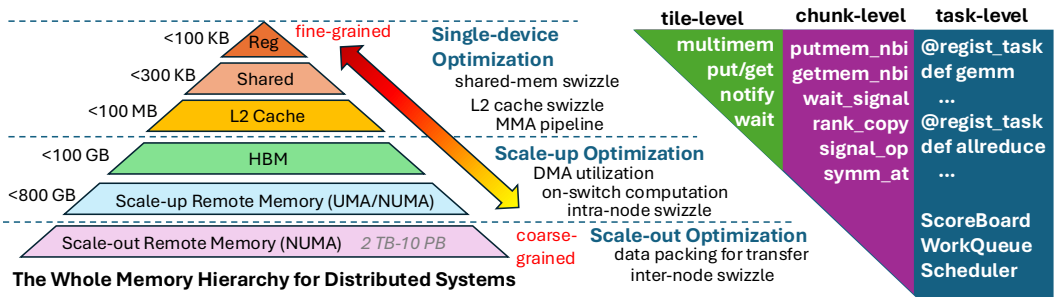


Figure 1: DITRON provides three levels of programming abstractions, which corresponds to the hardware hierarchy of distributed systems.

the memory hierarchy: scale-up remote memory and scale-out remote memory (shown in Figure 1). We summarize distributed NUMA programming model as the following:

1. Memory consistency by signals: Global synchronization in distributed GPU cluster is expensive in terms of latency. As a result, in-HBM signals are used for peer-to-peer synchronization. Signals are constructed and accessible by all the GPUs.

2. Non-blocking P2P copy and blocking multi-memory operations: On one hand, due to the inconsistency and long latency of remote memory access, peer-to-peer (P2P) data transfer between GPUs are asynchronous, making it possible to overlap with other computation and communication operations. On the other hand, modern GPUs equipped with NVLink Sharp support on-switch reduce computation and broadcast for acceleration, which is called multi-memory operations.

3. Resource partition for task pipeline: Communication and computation compete with each other for hardware resources, including streaming multiprocessors (SM), L2 cache, and HBM bandwidth. Efficient programming requires separate dedicated SMs for communication. The GPU is logically partitioned into multiple parts; each part is responsible for one task (computation or communication), forming a producer-consumer pipeline.

We aim to provide a new programming interface for distributed programming. The principle of our design is to offer minimal programming abstractions while maintaining consistency with existing compilers (Tillet et al., 2019) for computation when falling back to single-device programming, ensuring that the programming complexity is manageable for most researchers. Through extensive practice and refinement, we have developed the DITRON compiler, with its key abstractions shown in Figure 1:

1. Tile-level: For computation, DITRON inherits all the tile-level programming primitives from existing compilers (Tillet et al., 2019), thus retaining standard optimizations for single devices. For communication, we first introduce multi-memory primitives to leverage on-switch communication acceleration; second, we add tile-level remote memory access primitives; third, we incorporate wait/notify signal operations for intra-node synchronization.

2. Chunk-level: For data transfer, DITRON supports non-blocking put memory (to remote) and get memory (from remote) operations as well as chunk-level signal operations for inter-node synchronization. We also introduce new optimizations including low-latency protocol and intra/inter-node swizzling.

3. Task-level: Whether for computation or communication, distinct tasks ought to be mapped to separate hardware resources and pipelined through sophisticated scheduling. DITRON offers a task-level interface that supports task registration, dependency control, and task scheduling (all with minimal manual intervention) while generating efficient MegaKernels in real time.

The contributions of DITRON are twofold. First, we provide a simple yet efficient programming interface via compiler techniques, enabling researchers to add flexible parallel acceleration to their single-device code with minimal manual effort. This capability can potentially encourage more exploration of model architectures and the design of new parallel strategies. Second, through extensive validation and evaluation, DITRON achieves a $1.27\times - 19.18\times$ average speedup over non-overlapped

kernels across a wide range of workloads, and even outperforms expert-tuned overlapping libraries implemented in CUDA by $6\% - 30\%$. It also delivers a $5\% - 30\%$ end-to-end performance improvement for vLLM (Kwon et al., 2023) under large batch sizes. Validation across more than $10,000$ GPUs demonstrates DITRON's scalability and robustness for practical industrial deployment, saving millions of GPU hours per month. Additionally, DITRON supports AMD GPUs and PCIe-connected GPUs: on AMD GPUs, it achieves a $2\% - 38\%$ speedup over RocmBLAS+RCCL; on L20 GPUs, it delivers an average speedup of $8.33\times$ over CuBLAS+NCCL. DITRON is publicly available at `https://github.com/[blinded-for-review]`[1].

2 HARDWARE AND SOFTWARE IN DISTRIBUTED SYSTEM

A distributed system consists of multiple levels of hardware, the basic building block of which include GPUs, host CPUs, and switches. The overview of a GPU cluster is shown in Figure 2.

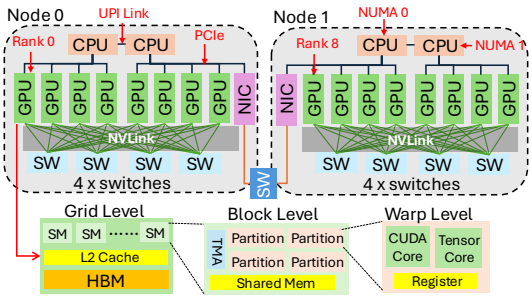


Figure 2: GPU Cluster Overview.

Computation and Communication Hardware: In distributed systems, the primary computing hardware is the GPU, which can be divided into three levels: warp, block, and grid. The warp level controls computing units such as CUDA Cores (for scalar and vector calculations) and Tensor Cores (for matrix multiplication). Block-level is responsible for warp partition and shared memory control. The grid level manages workload mapping across streaming multiprocessors (SMs) via scheduling and controls L2 cache access order through swizzling. The exploitation of single-GPU hardware resources to achieve high performance has been well studied in prior works (Nvidia, 2022; Dao et al., 2022; DeepSeek-AI, 2025; Spector et al., 2024), and we recommend readers refer to these for further details.

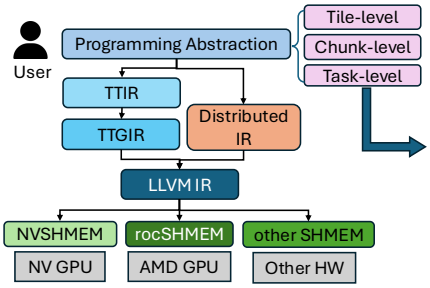
The communication aspect can be categorized into two domains: the intra-node scale-up domain and the inter-node scale-out domain. Within the scale-up domain, there are two memory access modes: UMA and NUMA. In UMA, the address space of each GPU is shared among all GPUs, enabling data copying between any two GPUs via standard load/store instructions or DMA engines (dedicated hardware units for direct memory access in GPUs). In NUMA, however, each GPU's memory space is symmetric. To access data from another GPU, local data pointers in HBM must first be mapped to the symmetric address space and then translated to the remote GPU's address.

UMA access via DMA engines is triggered by host CPUs rather than GPU SMs, thereby freeing up more resources for computation. In contrast, UMA access through load/store instructions (multi-memory access) occupies SMs, with reduction and broadcast operations achievable via dedicated hardware known as NVLink Sharp (NVIDIA, 2018). By contrast, NUMA access is always initiated by GPU-side SMs, leaving fewer SMs available for computation.

For inter-node domain, only NUMA mode is supported. GPU's memory space is mapped to symmetric address space and data copy is done by network interface cards (NICs) and network switches, allowing asynchronous (non-blocking, or *nbi* for shot) data copy from local GPU to remote GPU (putmem) or from remote to local (getmem).

Overlapping Library and Compiler: DITRON draws inspiration from three lines of research: compute-communication overlapping libraries (Chang et al., 2024; Zhang et al., 2025; Gond et al., 2025; Hong et al., 2025), single-device compilers (Tillet et al., 2019), and compute-communication overlapping compilers (Wang et al., 2023; Jangda et al., 2022; Google, 2025; Zheng et al., 2025). FLUX (Chang et al., 2024), COMET (Zhang et al., 2025) are libraries developed using CUTLASS (Nvidia, 2022), which enable overlapping by inlining wait/notify PTX assembly into CUTLASS matrix multiplication code and hand-crafting communication kernels using NVSHMEM (NVIDIA, 2025). The lack of programming abstraction makes it hard to develop new kernels

[1]The code is open-sourced, and the URL is anonymized to comply with double-blind review requirements.



Abstraction	Functionality	Examples
Tile-level	Tile Signal Control	wait(signal, value), notify(signal, value)
	NVLink Sharp	multimem_ld_reduce(tile), multimem_st(tile, data)
	Tile Swizzle	pid_m, pid_n = swizzle(pid_m, pid_n)
Chunk-level	DMA Engine	rank_copy(dst, src, data_size)
	Memory Mapping	symm_ptr = symm_at(ptr, rank)
	Chunk Signal Control	wait_signal(signal, rank, value)
	NIC Data Transfer	putmem_nbi(dst, src, data_size, rank)
Task-level	Task Management	register_task(func, config)
	Scoreboard	wait_deps(task), release_tile(task)
	Schedule	fetch_task(work_queue)

Figure 3: DITRON compiler stack includes (1) programming abstraction; (2) distributed IR; (3) LLVM IR and hardware-specific OpenSHMEM implementation. The right side shows examples for three levels of programming abstractions.

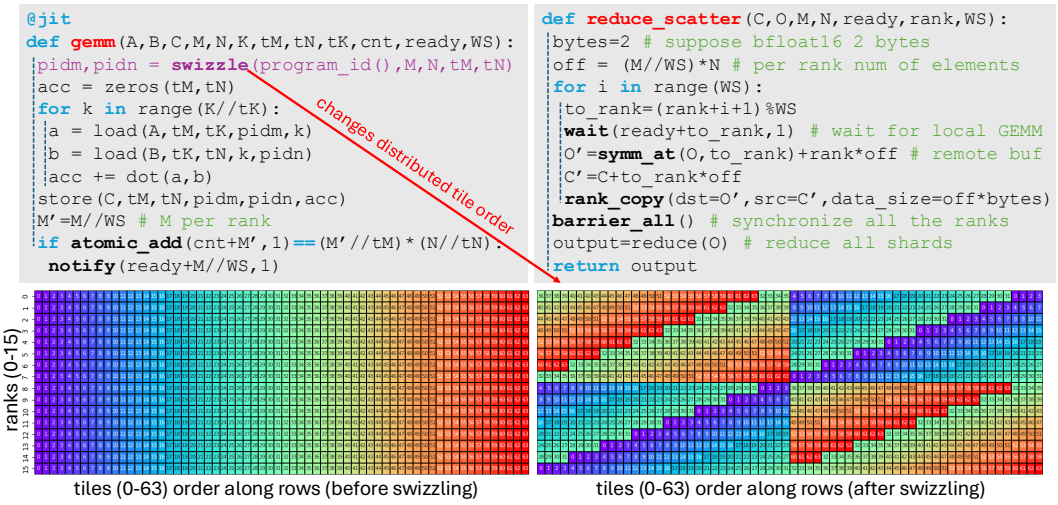


Figure 4: Simplified GEMM+ReduceScatter code example and distributed swizzling. The ReduceScatter requires that each GPU rank i takes 4 data chunks ($[4i, 4i+4)$) eventually. Tiles are processed from left to right. Without swizzling, all ranks start from data tiles for rank 0 and the other ranks are blocked, while with swizzling, all the ranks can start without blocking. A clearer enlarged swizzle view is presented in the Appendix B and the full code example is in Appendix E.2.

using these libraries. Triton (Tillet et al., 2019) compiler provides a set of NumPy-like primitives to reduce programming complexity. TileLink (Zheng et al., 2025) extends Triton's primitives to intra-node communication and enables overlapping kernels such as AllGather-GEMM and GEMM-ReduceScatter. The programming abstraction of TileLink only supports the *tile-level* in Figure 1. Similarly, other compilers such as CoCoNet (Jangda et al., 2022), Dist-Einsum (Wang et al., 2023), Pallas (Google, 2025), and Mirage (Wu et al., 2025) design new DSLs that only cover part of the abstractions in Figure 1. But in this work, we aim to support all the three levels of abstractions.

3 DITRON

We present DITRON, a flexible and versatile compiler designed for compute-communication overlapping kernel programming. An overview of DITRON is provided in Figure 3, which also includes primitive examples for the three levels of programming abstractions. In what follows, we first elaborate on the programming abstractions. Figure 4 presents a simplified code example for GEMM+ReduceScatter, upon which we then explain the swizzle optimizations. Finally, we detail the implementation specifics of DITRON.

3.1 THREE LEVELS OF PROGRAMMING ABSTRACTIONS

Tile-level Abstraction: In addition to the tile-level computation abstraction inherited from Triton (Tillet et al., 2019), DITRON further provides abstraction for tile-level signal control and distributed data transfer:

(1) Scale-up domain signals are viewed as tensors located in HBM. The signals can be used for spin-locks (through *wait* and *notify* primitive) to control producer-consumer execution. One signal can be shared by a tile of data, indicating that the whole tile of data is controlled by this signal.

(2) Distributed data is treated as tiles that reside in the HBM of remote GPUs. Although the data is physically located in remote HBM, local kernels can still hold a view (or pointers) of remote tiles. DITRON provides primitives to fetch the remote tiles and perform reduction/broadcast via NVLink Sharp (through *multimem_ld_reduce* and *multimem_st* primitives).

(3) Distributed data tiles can be swizzled like regular tiles to enhance performance. For distributed communication, local HBM is treated as a cache for remote memory, and data from remote memory arrives in local HBM at distinct steps. Consequently, to leverage temporal locality, tiles corresponding to earlier steps should be scheduled before other tiles, which is achieved via the *swizzle* primitive.

Chunk-level Abstraction: To leverage the hardware features of DMA engines and NICs, DITRON further provides chunk-level abstraction. Tile-level data transfer in scale-out domain is inefficient because the low-level hardware is designed for 1-D contiguous data transfer instead of N-D tile transfer. N-D tile transfer should be translated to multiple small 1-D data transfer at a low level, hampering the achieved bandwidth. Our chunk-level abstraction design includes:

(1) Data chunk is composed of multiple data tiles. The data tiles are packed together to form a larger data chunk before communication. For example, in our GEMM+AllToAll kernel (designed for Ulysses sequence parallel (Jacobs et al., 2023)), data should be packed along head dimension before AllToAll communication and concatenated along sequence dimension after communication.

(2) Distributed data chunks are symmetric, the address of which can be mapped to remote GPU's address space. GPUs across nodes can also share the symmetric view of data chunks (through *symm_at* primitive).

(3) Distributed data chunk transfer is asynchronous (through *rank_copy* and *put/getmem_nbi*) and can be synchronized by dedicated signal operations (through *wait_signal*). As a result, the data chunks can also be swizzled to exploit temporal locality, where scale-up remote memory becomes cache for scale-out remote memory.

Task-level Abstraction: Computation and communication kernels require dedicated hardware resources to achieve best performance. Manually scheduling the resource partition and execution order is error-prone and inefficient. DITRON provides lightweight task-level abstraction to help combine the computation and communication into one optimized MegaKernel:

(1) Both computation kernel and communication kernel can be converted to tasks. DITRON uses a unified interface to wrap kernels as tasks. The interface encapsulates (through *register_task*) the kernel's input/output tensors, the number of tiles within the kernel, and the tile dependency to its producer kernels.

(2) Tasks should be executed in producer-consumer order, which is controlled by a software scoreboard. DITRON provides scoreboard implementation and corresponding *wait_deps* and *release_tile* primitives. The dependency of tasks is captured during task registration.

(3) The mapping from task to physical SM can be scheduled and optimized through a scheduler. DITRON provides a scheduler interface that supports compile-time round-robin and zig-zag scheduling. Customized scheduling can be also added by users. Besides, DITRON also supports runtime scheduling (via *fetch_task* primitive) that chooses tasks for each SM dynamically.

3.2 SWIZZLING OPTIMIZATION

As noted in previous sections, the tile-level and chunk-level abstraction designs incorporate distributed tile/chunk swizzle optimizations for distributed scenarios to enhance locality in remote memory. DITRON supports two swizzle modes: gather mode and scatter mode. The key insight

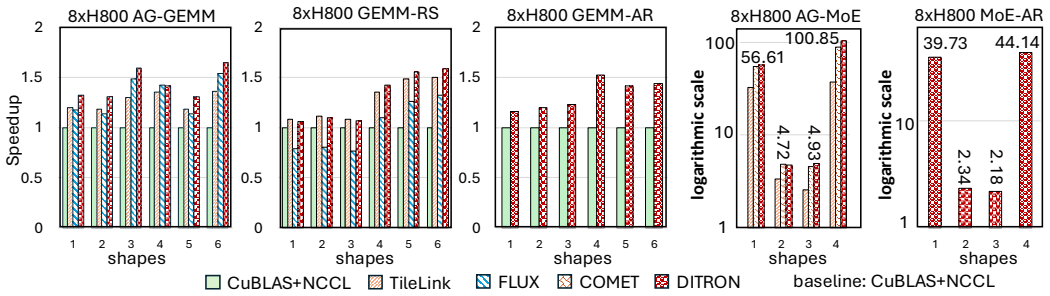


Figure 5: Evaluation of single workload on $8\times$ H800 GPUs. Workloads are AllGather+GEMM (AG-GEMM), GEMM+ReduceScatter (GEMM-RS), GEMM+AllReduce (GEMM-AR), AllGather+MoE (AG-MoE), MoE+AllReduce (MoE-AR). A logarithmic scale is used for the speedup of MoE due to its wide span, and the specific values are listed in the Appendix D.

is that gather mode aims to delay long-distance data transfers since computation depends on communication, the latter must be initiated as early as possible. In contrast, scatter mode prioritizes long-distance data transfers because communication can overlap with subsequent computation if started early, it is given precedence.

For the simplified GEMM+ReduceScatter example in Figure 4, we adopt a global perspective of all data tiles across all parallel devices. The heatmaps at the bottom illustrate 16 GPUs, each tasked with computing 64 result tiles. Each rank processes tiles from left to right. Without swizzling, only the data tiles required by rank 0 are processed initially, while other tiles remain unready, which blocks the execution of ranks dependent on these tiles. With swizzling, the order is rescheduled such that each rank has at least one data tile computed initially.

Non-perfect Shape Problem: For real-world workloads, input shapes may not be perfectly divisible by tile sizes, and cross-rank tiles/chunks exist. Swizzling for such workloads differs slightly. In gather mode, cross-rank tiles/chunks are postponed because they depend on multiple communication steps. In scatter mode, by contrast, they are prioritized: since multiple ranks rely on these tiles/chunks, processing them earlier allows subsequent computations to start sooner. We put the swizzling views for non-perfect tiling in Appendix B.

3.3 Implementation Details of Ditron

Ditron is implemented with around 59,000 lines of Python code and 7,000 lines of C++ code, consisting primarily of a compiler stack and an optimized kernel library.

Compiler Stack: The compiler stack of Ditron is illustrated in Figure 3. Users write NumPy-like programs using our three levels of abstractions, which cover both computation and communication. We have implemented a distributed IR for compilation: single-device semantics (e.g., *dot*, *load/store*) in the programs are lowered to standard Triton IR (TTIR) and Triton GPU IR (TTGIR), while distributed semantics are lowered to our distributed IR. This distributed IR adheres to the OpenSHMEM standard, which is a communication standard designed for distributed systems. OpenSHMEM has been implemented in efficient low-level communication libraries by NVIDIA (NVSHMEM) and AMD (rocSHMEM), with support for other hardware achievable via custom SHMEM libraries. The distributed IR is then lowered to LLVM IR, where we leverage LLVM's *CallExtern* capability to invoke external symbols from the SHMEM library.

Other Optimizations for Kernels:

(1) Low-latency (LL) Protocol: State-of-the-art communication libraries NCCL (NVIDIA, 2024) supports the LL and LL128 protocols, which are synchronization-free but incur approximately 50% bandwidth waste. Ditron also supports the LL protocol for latency-oriented kernels (e.g., batch size 1).

(2) Device-to-device (D2D) copy elimination: We observe that data movement via default driver APIs (cudaMemCpy) or PyTorch APIs (copy) can lead to stragglers and unexpected SM utilization,

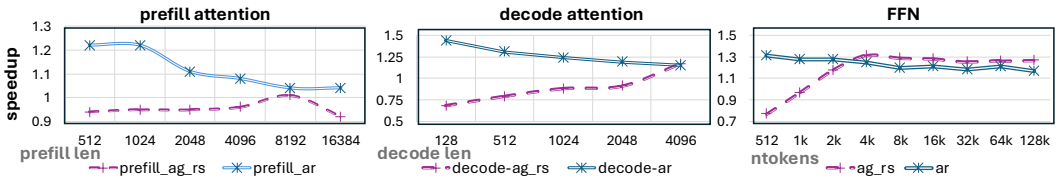


Figure 6: Module-level TP evaluation for Qwen3-32B: Results for prefill and decode attention show that DITRON with AllReduce achieves better speedups over CuBLAS+NCCL compared to using AllGather and ReduceScatter. For MLP, results indicate that the combination of AllGather and ReduceScatter performs better for large shapes, while AllReduce is superior for small shapes.

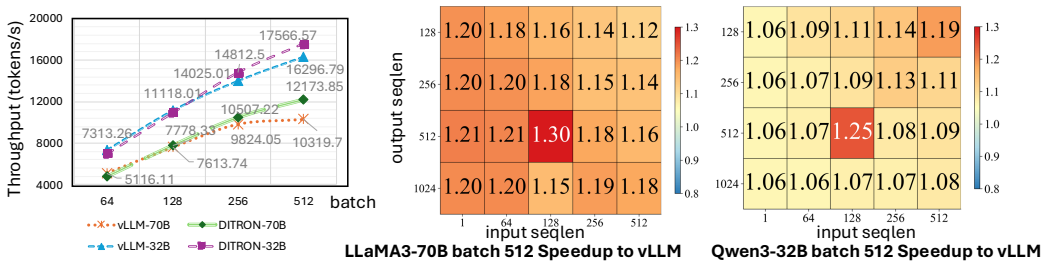


Figure 7: End-to-end TP evaluation results for Qwen3-32B and LLaMA3-70B on 8 H800s: DITRON demonstrates advantages over vLLM for large batch sizes and large models.

rendering overlap ineffective. Thus, we integrate these copies into our communication or computation kernels to prevent slowdowns.

(3) PCIe non-atomic synchronization and ring communication: The PCIe link provides no hardware guarantee for data transfer order, thus not supporting atomic operations. To deploy DITRON on PCIe-based GPUs such as the RTX 4090 and L20, we implement barriers using volatile load/store operations, eliminating the need for atomic operations. Additionally, PCIe-based communication requires a ring topology, and cross-UPI communication should be offloaded to the NIC. DITRON supports ring-based communication for AllGather and ReduceScatter, and implements cross-UPI optimizations to achieve high performance.

4 EXPERIMENTS

DITRON supports kernel implementation and optimization for various parallel strategies (TP, SP, EP). In our experiments, we evaluate DITRON under different parallel configurations. For inference, where the primary parallel strategy is TP within a single node, we first test 5 distinct workloads for GEMM and MoE. Second, we integrate these into attention modules and FFN modules to assess module-level performance. Third, we evaluate end-to-end inference using Qwen3-32B and LLaMA3-70B. For training, we test TP, SP, and EP kernels implemented by DITRON across 8 to 128 GPUs, examining both weak and strong scaling. Finally, we use DITRON's kernels to scale the training of a Qwen3-like 640B MoE model to 11,520 GPUs. DITRON also supports AMD GPUs and PCIe GPUs, which is discussed at the end of the evaluation section.

4.1 EVALUATION FOR INFERENCE

Single Workload Evaluation: We evaluated 5 distinct workloads on 8 H800 GPUs across 26 configurations, with each workload featuring multiple shape configurations derived from real-world models such as LLaMA (Dubey et al., 2024), Mixtral (Jiang et al., 2024), GPT (OpenAI, 2023), Qwen (Yang et al., 2025), and DeepSeek (DeepSeek-AI et al., 2024a). Detailed shape configurations are provided in Appendix C.2. Our baselines include CuBLAS (NVIDIA,

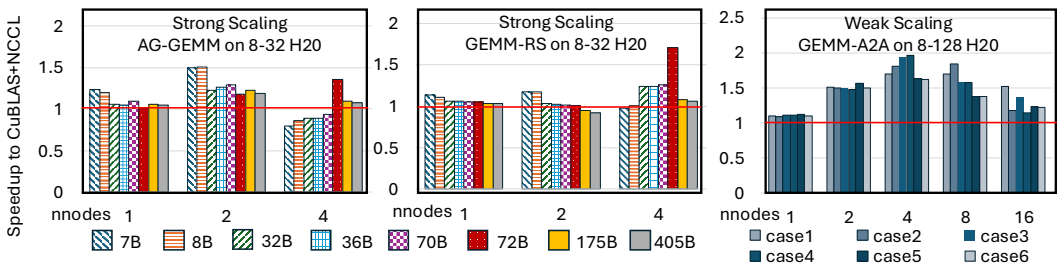


Figure 8: Scaling TP and SP workloads on H20 clusters: Results are presented as speedups over CuBLAS+NCCL, with shapes derived from various real-world LLMs (details in Appendix C.2). DITRON maintains speedups for AG-GEMM across up to 16 GPUs. For 32 GPUs, speedups are achievable only with shapes from large LLMs. The speedup of GEMM-RS remains consistent across 8–32 GPUs. GEMM-A2A exhibits the best weak scaling due to its scalable AllToAll.

2022)+NCCL (NVIDIA, 2024) (non-overlapping), TileLink (Zheng et al., 2025), FLUX (Chang et al., 2024), and COMET (Zhang et al., 2025), with results presented in Figure 5.

Overall, for AG-GEMM, the geometric speedup of DITRON is $1.43\times$ to CuBLAS+NCCL, $1.13\times$ to TileLink, and $1.09\times$ to FLUX. For GEMM-RS, the geometric speedup of DITRON is $1.27\times$ to CuBLAS+NCCL, $1.02\times$ to TileLink, and $1.30\times$ to FLUX. TileLink and FLUX have no support for AllReduce. For GEMM-AR, the geometric speedup to CuBLAS+NCCL is $1.32\times$. The speedup of AG-GEMM and GEMM-RS mainly comes from the overlapping of communication instead of faster GEMM (actually, we use Triton's GEMM and the GEMM is slightly slower than that of CuBLAS and FLUX), nearly 87.5% communication latency is hidden by computation for large input shapes. The speedup of GEMM-AR comes from faster AllReduce implemented using DITRON, which supports both one-shot algorithm and two-shot algorithm via multi-memory reduction/broadcast provided by NVLink Sharp. For AG-MoE, the geometric speedup of DITRON is $19.18\times$ to CuBLAS+NCCL, $1.89\times$ to TileLink, and $1.06\times$ to COMET. For MoE-AR, the average speedup to CuBLAS+NCCL is $13.89\times$

Module-level Evaluation: We incorporate the aforementioned workloads into attention modules and FFN modules. The QKV projection and output projection in attention modules are replaced by DITRON kernels. Prefill and decode performance is presented in Figure 6. Prefill input lengths range from 512 to 16k, and decode output lengths range from 128 to 4k. The two lines represent the difference in speedups over CuBLAS+NCCL between using AllGather+ReduceScatter and using AllReduce. The results indicate that for attention modules, both prefill and decode prefer AllReduce communication because the reduction dimension for GEMM (head dimension size in attention) is small (128), making GEMM latency not enough to hide communication latency of AllGather or ReduceScatter. The geometric mean speedup of DITRON's attention module (using AllReduce) to CuBLAS+NCCL is $1.12\times$ for prefill and $1.26\times$ for decode. On the other hand, GEMMs in FFN modules use large intermediate size and the latency of GEMMs can hide communication latency when given enough input tokens (large batch or sequence length), so overlapping AllGather and ReduceScatter is better than using AllReduce for sequence length larger than 2k. The speedup for 128k tokens of DITRON is $1.17\times$ using AllReduce and $1.27\times$ using AllGather+ReduceScatter.

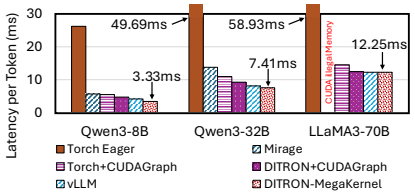


Figure 9: Batch 1 inference latency for Qwen3-8B, Qwen3-32B, LLaMA-70B using DITRON's distributed Megakernel on $8\times$ H800.

End-to-end Evaluation: We evaluate DITRON using end-to-end models LLaMA3-70B (Dubey et al., 2024) and Qwen3-32B (Yang et al., 2025). The results are shown in Figure 7. Our baseline is vLLM (Kwon et al., 2023), which is a strong baseline for TP inference as vLLM employs efficient AllReduce kernels designed by experts. For batch size less than 128, vLLM is slightly better than DITRON in throughput. But for batch sizes larger than 128, DITRON achieves $5\% - 30\%$ speedup

to vLLM. Specially, for batch size 512, we show the detailed speedup under different input lengths and output lengths for LLaMA3-70B and Qwen3-32B in Figure 7. The speedup translates to 12k tokens/s throughput for LLaMA3-70B and 17k tokens/s throughput for Qwen3-32B.

Distributed MegaKernel Evaluation: For single batch inference, the hardware resources are usually underutilized, task-level scheduling that produces MegaKernel can eliminate kernel launch overhead, increase SM activity, and improve end-to-end performance. We compare DITRON's distributed MegaKernel with PyTorch (Eager mode and CUDAGraph mode), DITRON+CUDAGraph, and Mirage (Wu et al., 2025) as shown in Figure 9. The geometric speedup is $6.28\times$ to Torch Eager, $1.73\times$ to Mirage, $1.33\times$ to Torch+CUDAGraph, $1.11\times$ to DITRON+CUDAGraph, and $1.10\times$ to vLLM. We also add MegaKernel code examples in Appendix E.4.

4.2 EVALUATION FOR TRAINING

Scale Single Workload: For training, we first evaluate both strong scaling and weak scaling performance for TP (AllGather+GEMM, GEMM+ReduceScatter), SP (GEMM+AllToAll), and EP (MoE Dispatch and MoE Combine) using DITRON. The results for TP and SP are shown in Figure 8 and the results for EP are shown in Figure 10. The detailed shape configurations are put in Appendix C.2. More results (on both H800 and H20) are put in Appendix D.

TP workloads renders heavy communication among nodes and the low bandwidth of network communication makes scaling non-beneficial. As a result, we only observe speedups of AllGather+GEMM and GEMM+ReduceScatter for 8-32 GPUs (strong scaling, number of total tokens is 32768), which ranges from $0.80\times$ to $1.71\times$ to CuBLAS+NCCL. For cases where the speedup is lower than 1, the main cause is GEMM becomes too small for each GPU after sharding, which cannot hide communication latency. As a comparison, GEMM+AllToAll (GEMM-A2A) gives consistent speedups for weak scaling (sequence length per rank remains unchanged) from 8 GPUs to 128 GPUs, where the shape of GEMM keeps constant. As for EP (we set 8192 tokens per rank, topk 8, hidden size 7168), dispatch and combine performance remains similar for strong scaling (globally 512 experts in total) and weak scaling (8 experts per rank). The speedup to PyTorch+NCCL implementations ranges from $1.04\times$ to $4.70\times$.

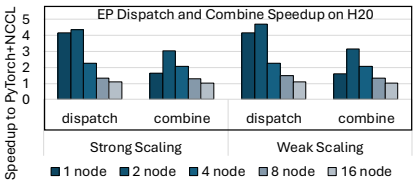


Figure 10: Scaling EP dispatch and combine using DITRON.

End-to-end Training of MoE Moodels: DITRON has been validated in industrial training settings on $11,520$ H20 GPUs using a training task that scales a Qwen3-like [2] MoE model to around 640B (33B activated). The achieved MFU is around 0.29. Compared to native PyTorch implementations, DITRON helps saving millions of GPU hours per month.

4.3 SUPPORT FOR OTHER PLATFORMS

DITRON can be transferred to more hardware platforms due to the flexibility of our distributed IR and the compatibility of dependent Triton compiler. For now, we manage to support AMD GPUs and PCIe GPUs (L20). We put our preliminary results in Appendix D. Overall, on AMD GPUs the speedup to RocmBLAS+RCCL ranges from $2\% - 38\%$; on L20 GPUs the average speedup to CuBLAS+NCCL is $8.33\times$.

5 CONCLUSION

Distributed inference and training becomes necessary and requires more researchers to be able to program distributed kernels. This work presents DITRON, a flexible and versatile distributed compiler with tile-level, chunk-level, and task-level interfaces for overlapping kernels. Researchers can use DITRON to program efficient distributed kernels for tensor parallel, sequence parallel, and expert parallel with performance comparable to or better than expert-tuned kernels.

[2] Details omitted to protect commercial secrets

Ethics Statement

This research focuses on compiler optimization techniques for distributed computing systems, aiming to enhance the computational efficiency of training and inference for LLMs. Our work does not directly involve human subjects, personally identifiable information, or sensitive datasets. We believe that DITRON has positive ethical implications. By significantly improving the efficiency of computation-communication overlap, our tool can effectively reduce the total compute time and energy consumption required for large-scale models, which helps mitigate the environmental impact of AI development.

Reproducibility Statement

We are committed to ensuring the reproducibility of our research. To this end, we have made the following efforts:

1. The complete source code for the DITRON compiler has been made publicly available upon submission. For the consideration of anonymity, we choose not to include any effective link or code package with this submission. However, we are willing to supplement the code package at any time, provided that the paper is accepted for publication or with the permission of the reviewers and the organizing committee during review process.

2. The appendix of the paper provides several simplified code examples, including All-Gather GEMM, GEMM Reduce-Scatter, GEMM All-Reduce, and MegaKernel, which clearly demonstrate the practical usage of DITRON's programming abstractions.

References

AMD. URL https://github.com/ROCm/rocSHMEM.

Anthropic. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet, 2024.

Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. FLUX: fast software-based communication overlap on gpus through kernel fusion. CoRR, abs/2406.06858, 2024. doi: 10.48550/ARXIV.2406.06858. URL https://doi.org/10.48550/arXiv.2406.06858.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. arXiv preprint arXiv:2507.06261, 2025.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. CoRR, abs/2205.14135, 2022. doi: 10.48550/arXiv.2205.14135. URL https://doi.org/10.48550/arXiv.2205.14135.

DeepSeek-AI. Deepseek deepgemm. https://github.com/deepseek-ai/DeepGEMM, 2025.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, Hao Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, Tao Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L.

Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, and Xiaowen Sun. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. CoRR, abs/2405.04434, 2024a. doi: 10.48550/ARXIV.2405.04434. URL https://doi.org/10.48550/arXiv.2405.04434.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. Deepseek-v3 technical report. CoRR, abs/2412.19437, 2024b. doi: 10.48550/ARXIV.2412.19437. URL https://doi.org/10.48550/arXiv.2412.19437.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. The llama 3 herd of models. CoRR, abs/2407.21783, 2024. doi: 10.48550/ARXIV.2407.21783. URL https://doi.org/10.48550/arXiv.2407.21783.

Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. Tokenweave: Efficient compute-communication overlap for distributed llm inference, 2025. URL https://arxiv.org/abs/2505.11329.

Google. Pallas, 2025. URL https://docs.jax.dev/en/latest/pallas/index.html.

Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, Guohao Dai, and Yu Wang. Flashoverlap: A lightweight design for efficiently overlapping communication and computation. CoRR, abs/2504.19519, 2025. doi: 10.48550/ARXIV.2504.19519. URL https://doi.org/10.48550/arXiv.2504.19519.

Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. CoRR, abs/2309.14509, 2023. doi: 10.48550/ARXIV.2309.14509. URL https://doi.org/10.48550/arXiv.2309.14509.

Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (eds.), ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022, pp. 402–416. ACM, 2022. doi: 10.1145/3503222.3507778. URL https://doi.org/10.1145/3503222.3507778.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts. CoRR, abs/2401.04088, 2024. doi: 10.48550/ARXIV.2401.04088. URL https://doi.org/10.48550/arXiv.2401.04088.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

NVIDIA. Nvidia nvswitch: Technical overview. Technical report, NVIDIA, 2018. URL https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

NVIDIA. cuBLAS, 2022. URL https://developer.nvidia.com/cublas.

Nvidia. Cutlass, 2022. URL https://github.com/NVIDIA/cutlass.

NVIDIA. Nvidia collective communications library. https://developer.nvidia.com/nccl, 2024.

NVIDIA. NVSHMEM, 2025. URL https://docs.nvidia.com/nvshmem/api/using.html.

OpenAI. GPT-4 technical report. CoRR, abs/2303.08774, 2023. doi: 10.48550/ARXIV.2303.08774. URL https://doi.org/10.48550/arXiv.2303.08774.

Qwen-Team. Qwen2.5 technical report. arXiv preprint arXiv:2412.15115, 2024.

Morgane Rivière, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozinska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucinska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju-yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjösund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, and Lilly McNealus. Gemma 2: Improving open language models at a practical size. CoRR, abs/2408.00118, 2024. doi: 10.48550/ARXIV.2408.00118. URL https://doi.org/10.48550/arXiv.2408.00118.

Aashaka Shah, Abhinav Jangda, Binyang Li, Caio Rocha, Changho Hwang, Jithin Jose, Madan Musuvathi, Olli Saarikivi, Peng Cheng, Qinghua Zhou, Roshan Dathathri, Saeed Maleki, and Ziyue Yang. Msccl++: Rethinking gpu communication abstractions for cutting-edge ai applications, 2025. URL https://arxiv.org/abs/2504.09014.

Benjamin Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable AI kernels. CoRR, abs/2410.20399, 2024. doi: 10.48550/ARXIV.2410.20399. URL https://doi.org/10.48550/arXiv.2410.20399.

TileLang-Team. Tilelang, 2025. URL https://github.com/tile-ai/tilelang.

Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. Triton: an intermediate language and compiler for tiled neural network computations. In Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (eds.), Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019, pp. 10–19. ACM, 2019. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (eds.), Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023, pp. 93–106. ACM, 2023. doi: 10.1145/3567955.3567959. URL https://doi.org/10.1145/3567955.3567959.

Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A multi-level superoptimizer for tensor programs. In 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25), Boston, MA, July 2025. USENIX Association.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jian Yang, Jiaxi Yang, Jingren Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. CoRR, abs/2505.09388, 2025. doi: 10.48550/ARXIV.2505.09388. URL https://doi.org/10.48550/arXiv.2505.09388.

Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. CoRR, abs/2502.19811, 2025. doi: 10.48550/ARXIV.2502.19811. URL https://doi.org/10.48550/arXiv.2502.19811.

Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, and Xin Liu. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives, 2025. URL https://arxiv.org/abs/2503.20313.

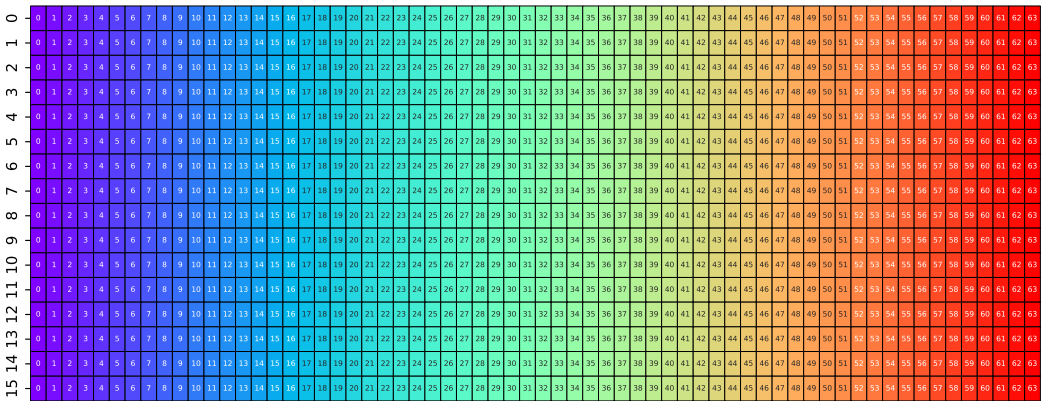


Figure 11: Global view of ranks and data tiles without swizzling. Corresponding to Figure 4. Each row corresponds to one rank; each column corresponds to one tile. Each rank computes GEMM from the left tile to the right tile. This example is for GEMM+ReduceScatter.

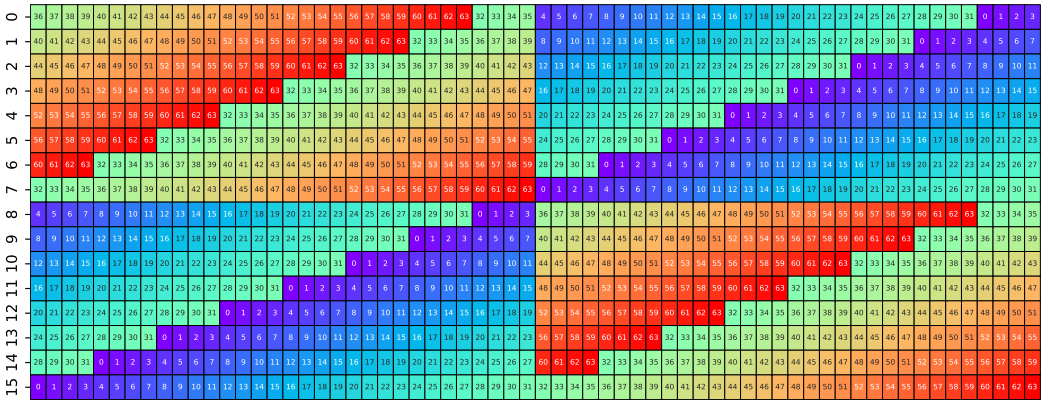


Figure 12: Global view of ranks and data tiles with swizzling. Corresponding to Figure 4. This example is for GEMM+ReduceScatter.

A Use of Large Language Models

In preparing this manuscript, LLM were used as a general-purpose writing assistance tool, primarily to improve the clarity and phrasing of certain sentences. All authors have reviewed and edited the final manuscript and take full responsibility for the entire content of the paper.

B Swizzle

Swizzling optimization has been mentioned in previous work (Jangda et al., 2022; Chang et al., 2024; Zheng et al., 2025). But their swizzling methods are limited to single node and there lack a clear explanation of the swizzling for non-perfect tiling scenarios (which is the most common for LLM training and inference).

Figure 11 and Figure 12 show an enlarged view of Figure 4, which correspond to swizzle optimization for 16 GPUs GEMM+ReduceScatter with perfect shapes (no cross-rank tiles). For this case, we use $M\_per\_rank = 1024, N\_per\_rank = 256, tile\_M = tile\_N = 256$.

For non-perfect shapes, some tiles are needed by multiple ranks for subsequent ReduceScatter. These tiles should be permuted to the left so that they can be computed earlier and transferred

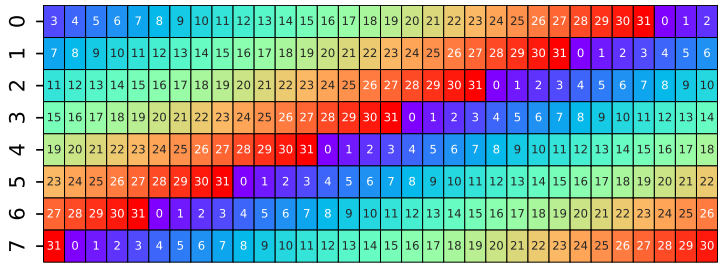


Figure 13: Single node global view of swizzling for non-perfect tiling. The tile orders are different from that of perfect tiling. Cross-rank tiles are permuted to left. This example is for GEMM+ReduceScatter.

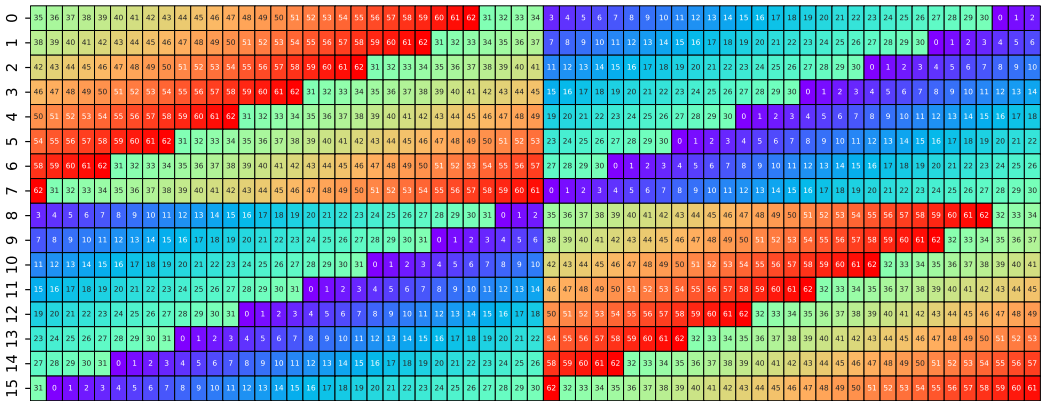


Figure 14: Two nodes global view of swizzling for non-perfect tiling. This example is for GEMM+ReduceScatter.

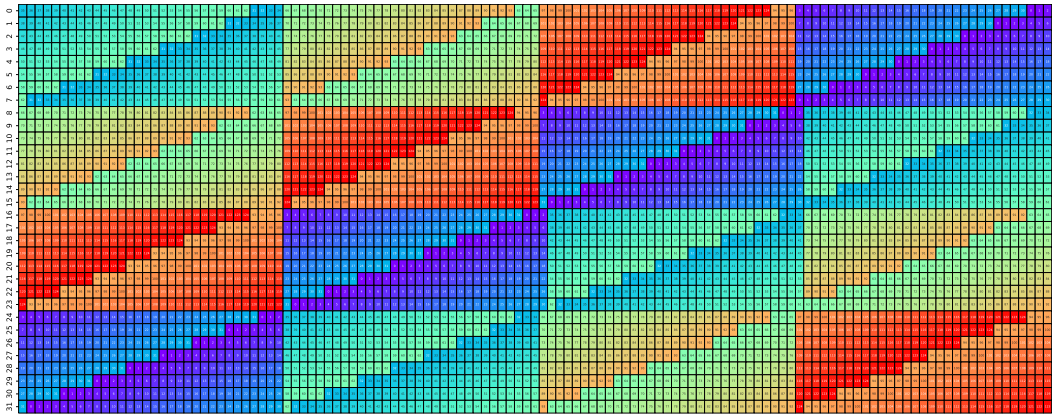


Figure 15: Four nodes global view of swizzling for non-perfect tiling. This example is for GEMM+ReduceScatter.

to the ranks that need them. Figure 13 shows swizzle results for 8 GPUs; Figure 14 shows swizzling for 16 GPUs; and Figure 15 shows swizzling for 32 GPUs. For this case, we use $M\_per\_rank = 997, N\_per\_rank = 256, tile\_M = tile\_N = 256$.

Besides GEMM+ReduceScatter, we also implement similar swizzling for AllGather+GEMM, and AllGather+MoE. The swizzle logic is implemented in DITRON through efficient kernels. We show the core code for swizzle as follows.

First, swizzle code for AllGather+GEMM. DITRON is implemented based on Triton, so we still use *triton.jit* as compile entry. We extend programming interface with multiple primitives for tile-level or even thread-level control, such as *laneid, warp_prefix_sum_kernel, __ballot_sync, __shfl_sync_i32*.

```python
@triton.jit(do_not_specialize=["rank"])
def threadblock_swizzle_allgather_gemm_kernel(
    tiled_m,
    M,
    rank,
    WORLD_SIZE: tl.constexpr,
    NNODES: tl.constexpr,
    BLOCK_SIZE_M: tl.constexpr,
    DEBUG: tl.constexpr = False,
):
    LOCAL_WORLD_SIZE = WORLD_SIZE // NNODES
    node_id = rank // LOCAL_WORLD_SIZE
    M_per_rank = M // WORLD_SIZE
    M_per_node = M // NNODES
    node_start = node_id

    lane_id = laneid()

    if lane_id < NNODES:
        n = (lane_id + node_start) % NNODES
        M_node_start = M_per_node * n
        M_node_end = M_per_node * (n + 1)
        tiled_m_node_start = M_node_start // BLOCK_SIZE_M
        prev_tiled_m_node_end = (M_node_start - 1) // BLOCK_SIZE_M
        tiled_m_node_end = (M_node_end - 1) // BLOCK_SIZE_M
        next_tiled_m_node_start = M_node_end // BLOCK_SIZE_M

        if lane_id == 0 and M_node_start != 0:
            if prev_tiled_m_node_end == tiled_m_node_start:
                tiled_m_node_start += 1

        if lane_id == 0 and M_node_end != M:
            if next_tiled_m_node_start == tiled_m_node_end:
                tiled_m_node_end -= 1

        if lane_id != NNODES - 1 and M_node_end != M:
            if next_tiled_m_node_start == tiled_m_node_end:
                tiled_m_node_end -= 1
        if DEBUG and lane_id == NNODES - 1:
            print("tiled_m_node_end", tiled_m_node_end)

        swizzled_tiled_m_size = tiled_m_node_end - tiled_m_node_start + 1
    else:
        swizzled_tiled_m_size = 0

    if DEBUG and lane_id < NNODES:
        print("swizzled_tiled_m_size", swizzled_tiled_m_size, lane_id)
    swizzled_tiled_m_size_accum = (warp_prefix_sum_kernel(swizzled_tiled_m_size, lane_id, NNODES) -
                                   swizzled_tiled_m_size)
    if DEBUG and lane_id < NNODES:
        print("swizzled_tiled_m_size_accum", swizzled_tiled_m_size_accum)

    tiled_m_size_l = __shfl_down_sync_i32(0xFFFFFFFF, swizzled_tiled_m_size, NNODES - node_start)
    tiled_m_size_r = __shfl_up_sync_i32(0xFFFFFFFF, swizzled_tiled_m_size, node_start)
    tiled_m_size = 0
    if lane_id < node_start:
        tiled_m_size = tiled_m_size_l
    elif lane_id < NNODES:
        tiled_m_size = tiled_m_size_r

    if DEBUG and lane_id < NNODES:
        print("tiled_m_size", tiled_m_size)

    tiled_m_size_accum = warp_prefix_sum_kernel(tiled_m_size, lane_id, NNODES) - tiled_m_size
    mask = __ballot_sync(0xFFFFFFFF, tiled_m < swizzled_tiled_m_size_accum)
    n = ffs(mask) - 1 - 1
    if DEBUG and lane_id < NNODES:
        print("tiled_m_size_accum", tiled_m_size_accum)
        print("n", n, tiled_m, swizzled_tiled_m_size_accum, mask)

    # map node
    nid = (n + node_start) % NNODES
    node_offset = __shfl_sync_i32(0xFFFFFFFF, swizzled_tiled_m_size_accum, n)

    tile_size = __shfl_sync_i32(0xFFFFFFFF, swizzled_tiled_m_size, n)

    tiled_m_intra_node = tiled_m - node_offset
    local_rank = rank % LOCAL_WORLD_SIZE
    m_start = M_per_node * nid + M_per_rank * local_rank
    tiled_m_start = tl.cdiv(m_start, BLOCK_SIZE_M)
    swizzled_node_offset = __shfl_sync_i32(0xFFFFFFFF, tiled_m_size_accum, nid)
```

```
    rank_offset = max(0, tiled_m_start - swizzled_node_offset)  # this may < 0, bad

    # map rank
    tiled_m_intra_node_new = (tiled_m_intra_node + rank_offset) % tile_size
    return swizzled_node_offset + tiled_m_intra_node_new
```

Then, swizzle code for GEMM+ReduceScatter.

```
@triton.jit(do_not_specialize=["rank"])
def threadblock_swizzle_gemm_reduce_scatter_kernel(
    tiled_m,
    M,
    rank,
    WORLD_SIZE: tl.constexpr,
    NNODES: tl.constexpr,
    BLOCK_SIZE_M: tl.constexpr,
    DEBUG: tl.constexpr = False,
):
    LOCAL_WORLD_SIZE = WORLD_SIZE // NNODES
    node_id = rank // LOCAL_WORLD_SIZE
    M_per_rank = M // WORLD_SIZE
    M_per_node = M // NNODES
    node_start = node_id + 1

    lane_id = laneid()

    if lane_id < NNODES:
        n = (lane_id + node_start) % NNODES
        M_node_start = M_per_node * n
        M_node_end = M_per_node * (n + 1)
        tiled_m_node_start = M_node_start // BLOCK_SIZE_M
        # if tiled_m_start_node overlaps with previous node, then we need to add 1 to
        ↪  tiled_m_start_node
        prev_tiled_m_node_end = (M_node_start - 1) // BLOCK_SIZE_M
        if lane_id != 0 and M_node_start != 0:
            if prev_tiled_m_node_end == tiled_m_node_start:
                tiled_m_node_start += 1

        tiled_m_node_end = (M_node_end - 1) // BLOCK_SIZE_M
        next_tiled_m_node_start = M_node_end // BLOCK_SIZE_M
        if lane_id == NNODES - 1 and M_node_end != M:
            if next_tiled_m_node_start == tiled_m_node_end:
                tiled_m_node_end -= 1

        swizzled_tiled_m_size = tiled_m_node_end - tiled_m_node_start + 1
    else:
        swizzled_tiled_m_size = 0

    if DEBUG and lane_id < NNODES:
        print("swizzled_tiled_m_size", swizzled_tiled_m_size, lane_id)
    swizzled_tiled_m_size_accum = (warp_prefix_sum_kernel(swizzled_tiled_m_size, lane_id, NNODES) -
                                   swizzled_tiled_m_size)
    if DEBUG and lane_id < NNODES:
        print("swizzled_tiled_m_size_accum", swizzled_tiled_m_size_accum)
    # thread 0 hold node `node_start` size
    # thread 1 hold node `node_start + 1` size
    # ...
    # thread NNODES - node_start hold node `0` size
    # thread NNODES - 1 hold NNODES `node_start - 1` size

    #  => thread 0 want to hold node 0 size

    tiled_m_size_l = __shfl_down_sync_i32(0xFFFFFFFF, swizzled_tiled_m_size, NNODES - node_start)
    tiled_m_size_r = __shfl_up_sync_i32(0xFFFFFFFF, swizzled_tiled_m_size, node_start)
    tiled_m_size = 0
    if lane_id < node_start:
        tiled_m_size = tiled_m_size_l
    elif lane_id < NNODES:
        tiled_m_size = tiled_m_size_r

    if DEBUG and lane_id < NNODES:
        print("tiled_m_size", tiled_m_size)

    tiled_m_size_accum = warp_prefix_sum_kernel(tiled_m_size, lane_id, NNODES) - tiled_m_size
    mask = __ballot_sync(0xFFFFFFFF, tiled_m < swizzled_tiled_m_size_accum)
    n = ffs(mask) - 1 - 1
    if DEBUG and lane_id < NNODES + 1:
        print("tiled_m_size_accum", tiled_m_size_accum)
        print("n", n, tiled_m, swizzled_tiled_m_size_accum, mask)

    # map node
    nid = (n + node_start) % NNODES
    node_offset = __shfl_sync_i32(0xFFFFFFFF, swizzled_tiled_m_size_accum, n)

    tile_size = __shfl_sync_i32(0xFFFFFFFF, swizzled_tiled_m_size, n)

    tiled_m_intra_node = tiled_m - node_offset
    local_rank = rank % LOCAL_WORLD_SIZE
    m_start = M_per_node * nid + M_per_rank * (local_rank + 1)
    tiled_m_start = m_start // BLOCK_SIZE_M
```

```
    swizzled_node_offset = __shfl_sync_i32(0xFFFFFFFF, tiled_m_size_accum, nid)
    rank_offset = max(0, tiled_m_start - swizzled_node_offset)  # this may < 0, bad

    # map rank
    tiled_m_intra_node_new = (tiled_m_intra_node + rank_offset) % tile_size
    return swizzled_node_offset + tiled_m_intra_node_new
```

At last, the swizzle code for AllGather+MoE.

```
@triton.jit(do_not_specialize=["rank"])
def threadblock_swizzle_ag_moe_kernel(
    # input
    ntokens_by_rank_by_expert_ptr,
    # output
    expert_id_ptr,
    tiled_m_ptr,
    segment_start_ptr,
    segment_end_ptr,
    ntiles_ptr,
    # workspace buffer
    ntokens_by_expert_by_rank_acc_ptr,
    ntiles_by_expert_acc_ptr,
    ntiles_by_expert_by_stage_ptr,
    ntiles_by_expert_by_stage_acc_ptr,
    rank,
    N_EXPERTS: tl.constexpr,
    TP_SIZE: tl.constexpr,
    LOCAL_TP_SIZE: tl.constexpr,
    NTILES_NEXT_POW_OF_2: tl.constexpr,
    BLOCK_SIZE_M: tl.constexpr,
    DEBUG: tl.constexpr = False,
):
    """
    tile_index = g(expert_id, stage, index): if tile_index is grouped by expert then stage

    but how to map (stage, index) => tile_index_in_expert?

    first map tile_index_in_expert => (stage, off_in_expert_by_stage, index_in_expert_in_stage)

    tile_index => (expert_id, tile_index_in_expert)                : ntokens grouped by expert_id by rank
                => (expert_id, stage)                              : get_stage_id
                => (expert_id, off_in_expert_by_stage)             : cumsum by stage
                => (expert_id, off_in_expert_by_stage, index_in_expert_in_stage)  : atomic_add for index_in_expert_in_stage
                => tile_index_new : by function g

    tile_index -> (expert_id, rank_offset) grouped by (expert_id, rank)
    rank_offset -> segment_start, segment_end, stage

    so we can remap tile_index as:
    tile_index_new = g(f(tile_index))
    """
    thread_idx = tid(0)
    N_EXPERTS_NEXT_POW_OF_2: tl.constexpr = next_power_of_2(N_EXPERTS)
    TP_SIZE_NEXT_POW_OF_2: tl.constexpr = next_power_of_2(TP_SIZE)
    offs_by_expert = tl.arange(0, N_EXPERTS_NEXT_POW_OF_2)
    mask_by_expert = offs_by_expert < N_EXPERTS
    offs_by_rank = tl.arange(0, TP_SIZE_NEXT_POW_OF_2)
    mask_by_rank = (offs_by_rank < TP_SIZE)
    offs_by_expert_by_rank = offs_by_expert[:, None] * TP_SIZE + offs_by_rank[None, :]
    mask_by_expert_by_rank = mask_by_expert[:, None] & mask_by_rank[None, :]
    offs_by_rank_by_expert = offs_by_rank[:, None] * N_EXPERTS + offs_by_expert[None, :]
    mask_by_rank_by_expert = mask_by_rank[:, None] & mask_by_expert[None, :]
    ntokens_by_rank_by_expert = tl.load(ntokens_by_rank_by_expert_ptr + offs_by_rank_by_expert,
                                        mask=mask_by_rank_by_expert)

    ntokens_by_expert_by_rank = ntokens_by_rank_by_expert.T
    ntokens_by_expert_by_rank_acc = tl.cumsum(ntokens_by_expert_by_rank, axis=1)
    ntokens_by_expert = tl.sum(ntokens_by_rank_by_expert, axis=0)
    ntiles_by_expert = tl.cdiv(ntokens_by_expert, BLOCK_SIZE_M)
    ntiles_by_expert_acc = tl.cumsum(ntiles_by_expert, axis=0)
    ntiles = tl.sum(ntiles_by_expert)

    tl.store(ntokens_by_expert_by_rank_acc_ptr + offs_by_expert_by_rank,
             ntokens_by_expert_by_rank_acc,
             mask=mask_by_expert_by_rank)
    tl.store(ntiles_by_expert_acc_ptr + offs_by_expert, ntiles_by_expert_acc)

    # # for each tiled_m in expert eid => stage id / segment_start / segmeng_end / tiled_m
    tile_index = tl.arange(0, NTILES_NEXT_POW_OF_2)
    mask_tile_idx = tile_index < ntiles

    __syncthreads()
    # tile_index -> (expert_id, offset_by_expert, tile_index_in_expert) -> stage, segment_start, segment_end
    expert_id, off_by_expert, off_in_expert =
    bisect_right_with_offset_kernel(ntiles_by_expert_acc_ptr, tile_index,
                                                                    N_EXPERTS)

```

```
    stage, segment_start, segment_end = get_tile_stage(
        off_in_expert,
        rank,
        TP_SIZE,
        LOCAL_TP_SIZE,
        BLOCK_SIZE_M,
        ntokens_by_expert_by_rank_acc_ptr + expert_id * TP_SIZE,
    )

    # histogram by expert by stage
    tl.store(ntiles_by_expert_by_stage_ptr + offs_by_expert_by_rank, 0, mask=mask_by_expert_by_rank)
    __syncthreads()
    off_in_expert_in_stage = tl.atomic_add(ntiles_by_expert_by_stage_ptr + expert_id * TP_SIZE +
    ↪  stage, 1,
                                           sem="relaxed", scope="gpu", mask=mask_tile_idx)
    __syncthreads()

    # do some cumsum
    ntiles_by_expert_by_stage = tl.load(ntiles_by_expert_by_stage_ptr + offs_by_expert_by_rank,
                                        mask=mask_by_expert_by_rank, other=0)
    ntiles_by_expert_by_stage_acc = tl.cumsum(ntiles_by_expert_by_stage, axis=1)
    __syncthreads()
    tl.store(
        ntiles_by_expert_by_stage_acc_ptr + offs_by_expert_by_rank,
        ntiles_by_expert_by_stage_acc,
        mask=mask_by_expert_by_rank,
    )
    __syncthreads()

    off_in_expert_by_stage = tl.where(
        stage == 0,
        0,
        tl.load(ntiles_by_expert_by_stage_acc_ptr + expert_id * TP_SIZE + stage - 1,
        ↪  mask=mask_tile_idx, other=0),
    )

    tile_index_by_expert_by_stage = off_by_expert + off_in_expert_by_stage + off_in_expert_in_stage
    __syncthreads()

    tl.store(expert_id_ptr + tile_index_by_expert_by_stage, expert_id, mask=mask_tile_idx)
    tl.store(tiled_m_ptr + tile_index_by_expert_by_stage, tile_index, mask=mask_tile_idx)
    tl.store(segment_start_ptr + tile_index_by_expert_by_stage, segment_start, mask=mask_tile_idx)
    tl.store(segment_end_ptr + tile_index_by_expert_by_stage, segment_end, mask=mask_tile_idx)
    thread_idx = tid(0)
    if thread_idx == 0:
        tl.store(ntiles_ptr, ntiles)
    if DEBUG and thread_idx < ntiles:
        print("expert_id", expert_id)
```

C Evaluation Setup and Workload Configurations

C.1 Evaluation Setup

The H800 clusters used in our evaluation are equipped with NVLink of uni-directional bandwidth of 200 GB/s. There are 8 NICs in one node and each GPU can communicate with another GPU in other nodes at a uni-directional bandwidth of 50 GB/s. The H20 clusters used in our evaluation are equipped with NVLink of uni-directional bandwidth of 450 GB/s. There are 4 NICs in one node and each GPU can communicate with another GPU in other nodes at a uni-directional bandwidth of 25 GB/s.

Table 1: Shapes for AllGather+GEMM on H800.

No.	From Model	M	N	K
1	LLaMA3-7B	8192	11008	4096
2	LLaMA3.1-8B	8192	14336	4096
3	LLaMA3.1-70B	8192	28672	8192
4	LLaMA3.1-405B	8192	53248	16384
5	Mistral-7B	8192	14336	4096
6	Qwen2-72B	8192	29568	8192

C.2 Workload Configuration

The input shapes for AllGather+GEMM used in Figure 5 are listed in Table 1. The input shapes for GEMM+ReduceScatter and GEMM+AllReduce used in Figure 5 are listed in Table 2. The input shapes for AllGather+MoE and MoE+AllReduce used in Figure 5 are listed in Table 3.

Table 2: Shapes for GEMM+ReduceScatter and GEMM+AllReduce on H800.

No.	From Model	M	N	K
1	LLaMA3-7B	8192	4096	11008
2	LLaMA3.1-8B	8192	4096	14336
3	LLaMA3.1-70B	8192	8192	28672
4	LLaMA3.1-405B	8192	16384	53248
5	Mistral-7B	8192	4096	14336
6	Qwen2-72B	8192	8192	29568

Table 3: Shapes for AllGather+MoE and MoE+AllReduce on H800.

No.	From Model	num_tokens	hidden_dim	intermediate_size	num_experts	topk
1	Qwen1.5-MoE-A2.7B	8192	2048	1408	60	4
2	Mixtral-8x7B	8192	14336	4096	8	2
3	Mixtral-8x22B	8192	16384	6144	8	2
4	DeepSeek-MoE	8192	1408	2048	64	6

Table 4: Shapes for strong scaling AllGather+GEMM.

Label	From Model	M	N	K
7B	LLaMA3-7B	32768	11008	4096
8B	LLaMA3.1-8B	32768	14336	4096
32B	Qwen3-32B	32768	25600	5120
36B	Seed-OSS-36B	32768	27648	5120
70B	LLaMA3.1-70B	32768	28672	8192
72B	Qwen2-72B	32768	29568	8192
175B	GPT-3-175B	32768	49152	12288
405B	LLaMA3.1-405B	32768	53248	16384

Table 5: Shapes for strong scaling GEMM+ReduceScatter.

1	From Model	M	N	K
7B	LLaMA3-7B	32768	4096	11008
8B	LLaMA3.1-8B	32768	4096	14336
32B	Qwen3-32B	32768	5120	25600
36B	Seed-OSS-36B	32768	5120	27648
70B	LLaMA3.1-70B	32768	8192	28672
72B	Qwen2-72B	32768	8192	29568
175B	GPT-3-175B	32768	12288	49152
405B	LLaMA3.1-405B	32768	16384	53248

Table 6: Shapes for strong scaling GEMM+AllToAll.

Label	seqlen	heads	head_size
case 1	262144	64	128
case 2	491520	64	128
case 3	262144	32	128
case 4	491520	32	128
case 5	262144	16	128
case 6	491520	16	128

For scaling experiments, the shapes for strong scaling AllGather+GEMM are shown in Table 4. The shapes remain the same from 8 GPUs to 32 GPUs. The shapes for weak scaling are the same as those in strong scaling except that the value of M varies with the number of GPUs. We use $M = 8192$ for 8 GPUs, $M = 16384$ for 16 GPUs, and $M = 32768$ for 32 GPUs. The shapes for strong scaling GEMM+ReduceScatter are shown in Table 5. The shapes remain the same for 8 GPUs and 32 GPUs. For weak scaling, the configurations for M are the same as weak scaling AllGather+GEMM.

The shapes for strong scaling GEMM+AllToAll are shown in Table 6. The shapes remain the same from 8 GPUs to 128 GPUs. The shapes for weak scaling are the same as those in strong scaling except that the seqlen varies with the number of GPUs. For 8 GPUs, seqlens are 65536 and 122880; for 16 GPUs, they are 131072 and 245760; for 32 GPUs, they are 262144 and 491520; for 64 GPUs, they are 524288 and 983040; for 128 GPUs, they are 1048576 and 1966080.

D Detailed Experiment Resutls

D.1 Detailed Evaluation Results for MoE

The detailed results for AllGather+MoE in Figure 5 are shown in Table 7. The detailed results for MoE+AllReduce in Figure 5 are shown in Table 8. The shapes for each case are shown in Table 3.

Table 7: Latency of AllGather+MoE (ms).

	CuBLAS+NCCL	TileLink	COMET	Ditron
1	18.12	0.56	0.33	0.32
2	7.12	2.14	1.48	1.51
3	8.49	3.30	1.85	1.72
4	24.27	0.65	0.28	0.24

Table 8: Latency of MoE+AllReduce (ms).

	CuBLAS+NCCL	TileLink	Ditron
1	22.04	1.44	0.445
2	7.99	3.70	2.603
3	8.97	4.32	3.289
4	28.64	0.84	0.441

Table 9: Strong scaling results (speedup to CuBLAS+NCCL) for AllGather+GEMM on H800.

From Model	1 Node	2 Node	4 Node
LLaMA3-7B	1.31	1.11	0.63
LLaMA3.1-8B	1.35	1.12	0.61
Qwen3-32B	1.59	1.32	0.65
Seed-OSS-36B	1.64	1.32	0.64
LLaMA3.1-70B	1.67	1.36	0.77
Qwen2-72B	1.67	1.39	1.03
GPT-3-175B	1.46	1.68	1.00
LLaMA3.1-405B	1.44	1.76	1.02

D.2 Detailed Evaluation Results for Scaling

In Figure 8 we shave shown strong scaling results for AllGather+GEMM on H20. Here we show strong scaling results on H800 and weak scaling results on both H20 and H800 in Table 9, Table 10, and Table 11.

Table 10: Weak scaling results (speedup to CuBLAS+NCCL) for AllGather+GEMM on H20

From Model	1 Node	2 Node	4 Node
LLaMA3-7B	1.23	1.50	0.80
LLaMA3.1-8B	1.19	1.50	0.86
Qwen3-32B	1.06	1.22	0.89
Seed-OSS-36B	1.05	1.26	0.89
LLaMA3.1-70B	1.10	1.29	0.93
Qwen2-72B	1.01	1.18	1.36
GPT-3-175B	1.05	1.23	1.09
LLaMA3.1-405B	1.05	1.19	1.08

Table 11: Weak scaling results (speedup to CuBLAS+NCCL) for AllGather+GEMM on H800

From Model	1 Node	2 Node	4 Node
LLaMA3-7B	1.30	0.98	0.63
LLaMA3.1-8B	1.30	1.02	0.61
Qwen3-32B	1.53	1.14	0.65
Seed-OSS-36B	1.57	1.15	0.64
LLaMA3.1-70B	1.61	1.25	0.77
Qwen2-72B	1.66	1.29	1.03
GPT-3-175B	1.43	1.58	1.00
LLaMA3.1-405B	1.34	1.63	1.02

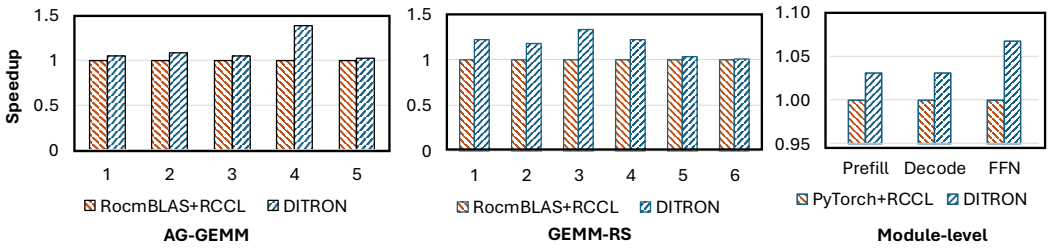


Figure 16: Evaluation results on 8×AMD MI308X GPUs. AllGather+GEMM and GEMM+ReduceScatter shapes are in Table 1 and Table 2. Module-level evaluation uses configurations from Qwen3-32B.

D.3 Evaluation Results on AMD GPU

DITRON can be deployed to AMD GPUs through RocSHMEM (AMD) and Triton for AMD. For evaluation, we use shapes from Table 1 and Table 2. Our baseline is RocmBLAS+RCCL. Overall, the speedup ranges from $1.02\times$ to $1.38\times$. The geometric speedup to baseline is $1.11\times$ for AllGather+GEMM and $1.16\times$ to GEMM+ReduceScatter. For module-level evaluation, we use Qwen3-32B. The speedup for prefill attention is $1.03\times$, the speedup for decode attention is $1.03\times$, and the speedup for FFN is $1.07\times$.

D.4 Evaluation Results on L20 GPU

We evaluate 6 different workloads on L20, including GEMM+ReduceScatter (Table 12), AllToAll (Table 13), AllGather+MoE (Table 14), MoE+ReduceScatter (Table 15), MoE+AllReduce (Table 16), and AllToAll+GEMM (Table 17). The geometric mean speedup of all the workloads is $8.33\times$.

Table 12: GEMM+RS performance on 8×L20 GPUs.

M	N	K	CuBLAS+NCCL	DITRON	Speedup
8192	8192	29568	18.63 ms	7.34 ms	2.54×

Table 13: All2All performance on 8×L20 GPU.

num_tokens	hidden_size	num_experts	topk	NCCL	DITRON	speedup
8	3584	128	8	0.42 ms	0.09 ms	4.51×

Table 14: AG+MoE performance on 8×L20 GPU (shape configurations see Table 3).

No.	num_tokens	CuBLAS+NCCL	DITRON	speedup
1	2048	21.10 ms	0.58 ms	36.26×
2	2048	43.91 ms	2.67 ms	16.47×
3	2048	51.11 ms	3.44 ms	14.86×
4	2048	21.43 ms	0.43 ms	49.84×

Table 15: MoE+RS performance on 8×L20 GPU.

num_tokens	hidden_size	intermediate_size	num_experts	topk	CuBLAS+NCCL	DITRON	speedup
8192	2048	1536	32	2	15.80 ms	1.71 ms	9.26×

Table 16: MoE+AR performance on 8×L20 GPU.

num_tokens	hidden_size	intermediate_size	num_experts	topk	CuBLAS+NCCL	DITRON	speedup
8192	2048	1536	32	2	18.19 ms	6.54 ms	2.78×

Table 17: A2A+GEMM performance on 8×L20 GPU.

DType	M	N	K	CuBLAS+NCCL	DITRON	speedup
Int8	7168	9216	3072	7.11 ms	2.15 ms	3.31×
FP8	7168	9216	3072	8.06 ms	2.17 ms	3.71×

E CODE EXAMPLES

In this section, we provide some simplified code examples, including All-Gather GEMM, GEMM Reduce-Scatter and GEMM All-Reduce. For each example, the code is generally divided into three main parts: the host-side launcher, the producer kernel, and the consumer kernel.

E.1 ALL-GATHER GEMM

Here we provide the code example for intra-node All-Gather GEMM. The following is the host-side Python function for launching the All-Gather GEMM operation. It orchestrates the producer (communication) and consumer (computation) kernels.

```python
def ag_gemm(a, b, ctx: AllGatherGEMMTensorParallelContext, use_cooperative=True):
    """allgather gemm
    Allgather global matrix A and do matmul with local matrix B, produces local matrix C

    Args:
        a (torch.Tensor<float>): local matmul A matrix. shape: [M_per_rank, K]
        b (torch.Tensor<float>): local matmul B matrix. shape: [N_per_rank, K]
        ctx: (AllGatherGEMMTensorParallelContext, Optional): if not provided, created immediately

    Returns:
        C (torch.Tensor<float>): local matmul C matrix. shape: [M, N_per_rank]
    """

    assert a.shape[1] == b.shape[
        1], f"tensor_B should has shape (col_major) [{b.shape[0]}, {a.shape[1]}], but get
        ↪  [{b.shape}]"
    assert a.dtype == b.dtype, f"Dtype of input and weight must be same: tensor_A dtype {a.dtype},
    ↪  tensor_B dtype {b.dtype}"

    M_per_rank, K = a.shape
    N_per_rank, _ = b.shape

    assert a.shape[0] * ctx.num_ranks <= ctx.max_M and a.shape[1] == ctx.K, f"Shape of tensor_A must
    ↪  not exceed the maxmize M of ctx: tensor_A shape [{a.shape}], ctx shape [{ctx.max_M},{ctx.K}]"
    assert b.shape[0] == ctx.N_per_rank, f"N_per_rank of tensor_B must match that of ctx: tensor_B
    ↪  shape [{b.shape[0]}], ctx shape [{ctx.N_per_rank}]"
    assert ctx.tensor_dtype == a.dtype, f"dtype of ctx must match that of ctx: tensor_A dtype
    ↪  {a.dtype}, ctx dtype {ctx.tensor_dtype}"

    C = torch.empty([ctx.num_ranks * M_per_rank, N_per_rank], dtype=a.dtype, device=a.device)

    local_copy_and_barrier_all(ctx.local_rank, ctx.rank, ctx.num_ranks, a, ctx.symm_workspace,
    ↪  ctx.symm_comm_buf, ctx.symm_barrier, M_per_rank, K, ctx.phase, is_internode=False,
    ↪  use_cooperative=use_cooperative)
    ctx.phase += 2

    current_stream = torch.cuda.current_stream()
    ctx.ag_intranode_stream.wait_stream(current_stream)

    cp_engine_producer_all_gather_full_mesh_pull(ctx.rank, ctx.num_ranks, a, ctx.symm_workspaces,
    ↪  ctx.symm_barriers, ctx.ag_intranode_stream, for_correctness=ctx.for_correctness,
    ↪  all_gather_method=ctx.all_gather_method)

    M_per_rank, K = a.shape
    M = M_per_rank * ctx.num_ranks
    grid = lambda META: (triton.cdiv(M, META["BLOCK_SIZE_M"]) * triton.cdiv(ctx.N_per_rank,
    ↪  META["BLOCK_SIZE_N"]), )
    kernel_consumer_gemm[grid](ctx.symm_workspace[:M], b, C, M, ctx.N_per_rank, ctx.K,
    ↪  ctx.symm_workspace.stride(0), ctx.symm_workspace.stride(1), b.stride(1), b.stride(0),
    ↪  c.stride(0), C.stride(1), ctx.rank, ctx.num_ranks, ctx.symm_barrier, ctx.BLOCK_M,
    ↪  ctx.BLOCK_N, ctx.BLOCK_K, ctx.GROUP_SIZE_M, num_stages=ctx.stages, num_warps=ctx.warps)
    current_stream.wait_stream(ctx.ag_intranode_stream)

    return C
```

The producer kernel below is responsible for the All-Gather communication. It copies the local data tensor to the symmetric memory (`remote_tensor_buffers`) of other GPUs and then sets a signal (`_set_signal_cuda`) to notify the consumer kernels that the data is ready.

```python
def cp_engine_producer_all_gather_full_mesh_pull(
    rank,
    num_ranks,
    local_tensor: torch.Tensor,
    remote_tensor_buffers: List[torch.Tensor],
    barrier_buffers: List[torch.Tensor],
    stream: torch.cuda.Stream,
    for_correctness=False,
):
    M_per_rank, N = local_tensor.shape
```

```

    rank_orders = [(rank + i) % num_ranks for i in range(num_ranks)]

    with torch.cuda.stream(stream):
        if for_correctness:
            # fake a slow communication case
            # test if the computation is waiting for the correct communication
            _add_noise_workload_debug()
        for src_rank in rank_orders:
            if src_rank == rank:
                continue
            dst = remote_tensor_buffers[rank][src_rank * M_per_rank:(src_rank + 1) * M_per_rank, :]
            src = remote_tensor_buffers[src_rank][src_rank * M_per_rank:(src_rank + 1) * M_per_rank,
            ↪  :]
            dst.copy_(src)
            _set_signal_cuda(barrier_buffers[rank][src_rank], 1, stream)
```

The consumer kernel first waits on a signal (`dl.wait`) indicating that the required data are available. Once the data is ready, it proceeds with the GEMM computation on the data.

```
@triton.jit(do_not_specialize=["rank"])
def kernel_consumer_gemm(
        a_ptr, b_ptr, c_ptr,
        M, N, K,
        stride_am, stride_ak,
        stride_bk, stride_bn,
        stride_cm, stride_cn, rank, WORLD_SIZE: tl.constexpr, barrier_ptr,
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,  #
        GROUP_SIZE_M: tl.constexpr,  #
):
    a_dtype = a_ptr.dtype.element_ty
    b_dtype = b_ptr.dtype.element_ty
    c_dtype = c_ptr.dtype.element_ty
    tl.static_assert(a_dtype == b_dtype, "A and B must have the same dtype")

    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m

    m_per_rank = M // WORLD_SIZE
    m_offset = m_per_rank * rank
    pid_m_offset = tl.cdiv(m_offset, BLOCK_SIZE_M)
    pid_m = (pid_m + pid_m_offset) % num_pid_m

    offs_am = pid_m * BLOCK_SIZE_M
    rank_beg = offs_am // m_per_rank
    rank_end = (min(offs_am + BLOCK_SIZE_M, M) - 1) // m_per_rank
    token = dl.wait(barrier_ptr + rank_beg, rank_end - rank_beg + 1, "gpu", "acquire", waitValue=1)

    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)

    a_ptrs = dl.consume_token(a_ptrs, token)

    if a_dtype == tl.int8:
        accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.int32)
    else:
        accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator += tl.dot(a, b)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk

    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)

    tl.store(c_ptrs, accumulator.to(c_dtype), mask=c_mask)
```

E.2 GEMM Reduce-Scatter

Here we provide the code example for GEMM Reduce-Scatter. The following is the host-side Python function for launching the GEMM Reduce-Scatter operation.

```python
def gemm_rs_op(input, weight, ctx: GEMMReduceScatterTensorParallelContext, persistent: bool = True,
               fuse_scatter: bool = False):
    world_size = ctx.rs_ctx.world_size
    local_world_size = ctx.rs_ctx.local_world_size
    rs_stream = ctx.rs_stream
    output_dtype = ctx.output_dtype
    num_gemm_sms = ctx.num_gemm_sms

    M, local_K = input.shape
    N = weight.shape[0]
    assert N == ctx.rs_ctx.N

    assert M % world_size == 0
    assert weight.shape[1] == local_K
    M_per_rank = M // world_size
    current_stream = torch.cuda.current_stream()
    rs_stream.wait_stream(current_stream)

    output = torch.empty((M_per_rank, N), dtype=output_dtype, device=input.device)
    workspace = torch.zeros((world_size, ), dtype=torch.int32, device=input.device)
    gemm_out = ctx.get_gemm_out_buf(input)
    scatter_signal = ctx.rs_ctx.scatter_signal_buf

    triton_config = triton.Config(
        {
            "BLOCK_SIZE_M": ctx.BLOCK_M, "BLOCK_SIZE_N": ctx.BLOCK_N, "BLOCK_SIZE_K": ctx.BLOCK_K,
            "GROUP_SIZE_M": ctx.GROUP_M
        }, num_stages=ctx.stages, num_warps=8)
    triton_config = update_triton_config(M, N, local_K, input.dtype, world_size, local_world_size,
    ↪  triton_config)
    gemm_rs_producer(input, weight, gemm_out, scatter_signal, workspace, world_size,
    ↪  local_world_size, fuse_scatter, triton_config)

    if not fuse_scatter:
        with torch.cuda.stream(rs_stream):
            reduce_scatter_2d_op(gemm_out, ctx.rs_ctx, output)
        current_stream.wait_stream(rs_stream)
    else:
        nvshmem_barrier_all_on_stream(current_stream)
        ring_reduce(gemm_out, output, ctx.rs_ctx.local_rank, local_world_size)
        nvshmem_barrier_all_on_stream(current_stream)
    return output
```

The producer kernel below is responsible for the GEMM computation. When `FUSE_SCATTER` is enabled, it directly writes the output to the symmetric memory of the destination rank.

```python
@triton.jit(launch_metadata=_matmul_launch_metadata, repr=_gemm_rs_non_persistent_repr)
def kernel_gemm_rs_producer(
    # Pointers to matrices
    a_ptr,  # [M, K]_Ti
    b_ptr,  # [K, N]_Ti
    c_ptr,  # [M, N]_To
    # Matrix dimensions
    M,
    N,
    K,
    # The stride variables represent how much to increase the ptr by when moving by 1
    # element in a particular dimension. E.g. `stride_am` is how much to increase `a_ptr`
    # by to get the element one row down (A has M rows).
    stride_am,
    stride_ak,  #
    stride_bk,
    stride_bn,  #
    stride_cm,
    stride_cn,
    barrier_ptr,
    counter_ptr,
    FUSE_SCATTER: tl.constexpr,
    LOCAL_WORLD_SIZE: tl.constexpr,
    WORLD_SIZE: tl.constexpr,
    # Meta-parameters
    BLOCK_SIZE_M: tl.constexpr,
    BLOCK_SIZE_N: tl.constexpr,
    BLOCK_SIZE_K: tl.constexpr,  #
    GROUP_SIZE_M: tl.constexpr,
):
    """Kernel for computing the matmul C = A x B.
    A has shape (M, K), B has shape (K, N) and C has shape (M, N)
    """
    tl.static_assert(a_ptr.dtype.is_ptr(), "A should be a pointer")
    tl.static_assert(b_ptr.dtype.is_ptr(), "B should be a pointer")
    tl.static_assert(c_ptr.dtype.is_ptr(), "C should be a pointer")
    a_dtype = a_ptr.dtype.element_ty
    b_dtype = b_ptr.dtype.element_ty
    tl.static_assert(a_dtype == b_dtype, "A and B should have the same dtype")

    rank = dl.rank()
    NNODES = WORLD_SIZE // LOCAL_WORLD_SIZE

    pid = tl.program_id(axis=0)
```

```
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)

    M_per_rank = M // WORLD_SIZE
    pid_m, pid_n = swizzle_2d(pid, num_pid_m, num_pid_n, GROUP_SIZE_M)

    if NNODES != 1:  # with complex threadblock swizzle logic
        pid_m = threadblock_swizzle_gemm_reduce_scatter_kernel(pid_m, M, rank, WORLD_SIZE, NNODES,
        ↪  BLOCK_SIZE_M)
    else:
        pid_m_offset = (rank + 1) * M_per_rank // BLOCK_SIZE_M
        pid_m = (pid_m + pid_m_offset) % num_pid_m

    # Create pointers for the first blocks of A and B.
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)

    # Iterate to compute a block of the C matrix.
    if a_ptr.dtype.element_ty == tl.int8:
        accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.int32)
    else:
        accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)

    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        # Load the next block of A and B, generate a mask by checking the K dimension.
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        # We accumulate along the K dimension.
        accumulator += tl.dot(a, b)
        # Advance the ptrs to the next K block.
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk

    # Write back the block of the output matrix C with masks.
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    out_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)

    if not FUSE_SCATTER:
        tl.store(c_ptrs, accumulator, mask=out_mask)

        # inc barrier
        segment_start = pid_m * BLOCK_SIZE_M // M_per_rank
        segment_end = (min((pid_m + 1) * BLOCK_SIZE_M, M) - 1) // M_per_rank
        __syncthreads()
        segment = segment_start + tid(axis=0)
        if segment <= segment_end:
            m_start = M_per_rank * segment
            m_end = M_per_rank * (segment + 1) - 1
            tiled_m_start = m_start // BLOCK_SIZE_M
            tiled_m_end = m_end // BLOCK_SIZE_M
            tiled_m_size = tiled_m_end - tiled_m_start + 1
            val = atomic_add(counter_ptr + segment, 1, semantic="release", scope="gpu")
            if val == num_pid_n * tiled_m_size - 1:
                # or use other signal op semantic
                st(barrier_ptr + segment, 1, scope="gpu", semantic="release")
    else:
        rank_start = pid_m * BLOCK_SIZE_M // M_per_rank
        rank_end = (min((pid_m + 1) * BLOCK_SIZE_M, M) - 1) // M_per_rank
        for cur_rank in range(rank_start, rank_end + 1):
            m_start = max(M_per_rank * cur_rank, pid_m * BLOCK_SIZE_M)
            m_end = min(M_per_rank * (cur_rank + 1) - 1, (min((pid_m + 1) * BLOCK_SIZE_M, M) - 1))
            remote_c_ptr = dl.symm_at(c_ptr, cur_rank)
            mask_offset = m_start - pid_m * BLOCK_SIZE_M
            remote_offs_cm = m_start % M_per_rank + rank * M_per_rank + tl.arange(0, BLOCK_SIZE_M) -
            ↪  mask_offset
            remote_c_ptrs = remote_c_ptr + stride_cm * remote_offs_cm[:, None] + stride_cn *
            ↪  offs_cn[None, :]
            remote_mask = (offs_cm[:, None] <= m_end) & (offs_cm[:, None] >= m_start) &
            ↪  (offs_cn[None, :] < N)
            tl.store(remote_c_ptrs, accumulator, mask=remote_mask)
```

The consumer kernel performs the reduce-scatter communication operation. After the GEMM computation is complete, this kernel reduces the partial results across ranks and scatters them to produce the final output.

```
def reduce_scatter_multi_node(input: torch.Tensor, ctx: ReduceScatter2DContext, output:
↪  Optional[torch.Tensor] = None):
    """
    A hierarchical reduce-scatter implementation that overlaps the intra-node scatter
    with the local reduce and the inter-node p2p(after reduce). It also provides a rank-wise
    signal and supports overlap with gemm.
    """
    M, N = input.shape
    M_per_rank = M // ctx.world_size
```

```

    current_stream = torch.cuda.current_stream()
    ctx.reduction_stream.wait_stream(current_stream)

    # directly reduce_scatter to output if nnodes == 1
    out_each_node = output if ctx.nnodes == 1 else None
    if not has_fullmesh_nvlink():
        rs_result_per_node = reduce_scatter_for_each_node_ring(input, ctx, out_each_node)
    else:
        rs_result_per_node = reduce_scatter_for_each_node(input, ctx, out_each_node)

    if ctx.nnodes == 1:
        return rs_result_per_node

    nvshmem_barrier_all_on_stream(current_stream)
    if output is None:
        output = torch.empty((M_per_rank, N), dtype=input.dtype, device=input.device)
    ring_reduce(rs_result_per_node, output, ctx.node_id, ctx.nnodes)
    return output


def reduce_scatter_2d_op(input: torch.Tensor, ctx: ReduceScatter2DContext, output: Optional[torch.Tensor] = None):
    M, N = input.shape
    assert input.dtype == ctx.dtype
    assert ctx.max_M >= M and ctx.N == N
    assert M % ctx.world_size == 0

    nvshmem_barrier_all_on_stream(torch.cuda.current_stream())
    output = reduce_scatter_multi_node(input, ctx, output)
    ctx.reset_barriers()
    return output
```

E.3 GEMM ALL-REDUCE

Here we provide the example for GEMM All-Reduce. The following is the host-side Python function for launching the All-Reduce GEMM operation.

```
def gemm_allreduce_op(ctx: GemmARContext, a, b, gemm_config: triton.Config, copy_to_local=True, USE_MULTIMEM_ST=True):

    current_stream = torch.cuda.current_stream()
    ar_stream = ctx.ar_stream
    ar_stream.wait_stream(current_stream)

    M, N = a.shape[0], b.shape[0]
    # Check constraints.
    assert a.shape[1] == b.shape[1], "Incompatible dimensions"
    assert a.dtype == b.dtype, "Incompatible dtypes"
    symm_c = ctx.get_gemm_out_buf(a, b)
    symm_ar_out = ctx.symm_ar_out_buf
    gemm_barrier = ctx.gemm_barrier_buf
    multi_st_barrier = ctx.multi_st_barrier_buf
    NUM_COMM_SMS = ctx.NUM_COMM_SMS
    ar_out = torch.empty((M, N), dtype=a.dtype, device=a.device)
    BLOCK_SIZE_M = gemm_config.all_kwargs()["BLOCK_SIZE_M"]
    BLOCK_SIZE_N = gemm_config.all_kwargs()["BLOCK_SIZE_N"]
    # add mask in `consumer_all_reduce` can remove this constraint
    assert N % BLOCK_SIZE_N == 0
    persistent_gemm_notify(a, b, symm_c, gemm_barrier, gemm_config)
    with torch.cuda.stream(ar_stream):
        consumer_all_reduce(symm_c, symm_ar_out, ar_out, gemm_barrier, multi_st_barrier, BLOCK_SIZE_M=BLOCK_SIZE_M,
                            BLOCK_SIZE_N=BLOCK_SIZE_N, NUM_COMM_SMS=NUM_COMM_SMS, USE_MULTIMEM_ST=USE_MULTIMEM_ST)
    current_stream.wait_stream(ar_stream)
    if USE_MULTIMEM_ST and copy_to_local:
        ar_out.copy_(symm_ar_out.reshape(-1)[:M * N].reshape(M, N))
    nvshmem_barrier_all_on_stream(current_stream)
    if USE_MULTIMEM_ST and not copy_to_local:
        return symm_ar_out.reshape(-1)[:M * N].reshape(M, N)
    return ar_out
```

The producer kernel below is responsible for the GEMM computation. It performs computation and notifies the consumer kernels when tiles are ready for all-reduce.

```
@triton.jit(do_not_specialize=[])
def kernel_persistent_gemm_notify(a_ptr, b_ptr, c_ptr, gemm_barrier_ptr,  #
                                  M, N, K,  #
                                  stride_am, stride_ak,  #
                                  stride_bn, stride_bk,  #
                                  stride_cm, stride_cn,  #
                                  BLOCK_SIZE_M: tl.constexpr,  #
                                  BLOCK_SIZE_N: tl.constexpr,  #
```

```
                                BLOCK_SIZE_K: tl.constexpr,  #
                                GROUP_SIZE_M: tl.constexpr,  #
                                NUM_GEMM_SMS: tl.constexpr,  #
                                ):
    start_pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    k_tiles = tl.cdiv(K, BLOCK_SIZE_K)
    num_tiles = num_pid_m * num_pid_n

    tile_id_c = start_pid - NUM_GEMM_SMS

    offs_k_for_mask = tl.arange(0, BLOCK_SIZE_K)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n

    for tile_id in tl.range(start_pid, num_tiles, NUM_GEMM_SMS):
        pid_m, pid_n = _compute_pid(tile_id, num_pid_in_group, num_pid_m, GROUP_SIZE_M, NUM_GEMM_SMS)
        start_m = pid_m * BLOCK_SIZE_M
        start_n = pid_n * BLOCK_SIZE_N
        offs_am = start_m + tl.arange(0, BLOCK_SIZE_M)
        offs_bn = start_n + tl.arange(0, BLOCK_SIZE_N)
        offs_am = tl.where(offs_am < M, offs_am, 0)
        offs_bn = tl.where(offs_bn < N, offs_bn, 0)
        offs_am = tl.max_contiguous(tl.multiple_of(offs_am, BLOCK_SIZE_M), BLOCK_SIZE_M)
        offs_bn = tl.max_contiguous(tl.multiple_of(offs_bn, BLOCK_SIZE_N), BLOCK_SIZE_N)

        accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
        for ki in range(k_tiles):
            offs_k = ki * BLOCK_SIZE_K + tl.arange(0, BLOCK_SIZE_K)
            a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
            b_ptrs = b_ptr + (offs_bn[:, None] * stride_bn + offs_k[None, :] * stride_bk)

            a = tl.load(a_ptrs, mask=offs_k_for_mask[None, :] < K - ki * BLOCK_SIZE_K, other=0.0)
            b = tl.load(b_ptrs, mask=offs_k_for_mask[None, :] < K - ki * BLOCK_SIZE_K, other=0.0)
            accumulator = tl.dot(a, b.T, accumulator)

        tile_id_c += NUM_GEMM_SMS
        pid_m, pid_n = _compute_pid(tile_id_c, num_pid_in_group, num_pid_m, GROUP_SIZE_M,
        ↪  NUM_GEMM_SMS)
        offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
        offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
        c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
        c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
        c = accumulator.to(c_ptr.dtype.element_ty)
        tl.store(c_ptrs, c, mask=c_mask)
        __syncthreads()

        thread_idx = tid(0)
        gemm_barrier_idx = pid_m * num_pid_n + pid_n
        if thread_idx == 0:
            st(gemm_barrier_ptr + gemm_barrier_idx, 1, scope="gpu", semantic="release")
```

The consumer kernel performs the all-reduce operation. It waits for GEMM tiles to be ready, then performs reduction across all ranks.

```
@triton.jit(do_not_specialize=[])
def consumer_all_reduce_kernel(
    symm_input_ptr,
    symm_ar_out_ptr,
    ar_out_ptr,  #
    gemm_barrier_ptr,
    multi_st_barrier_ptr,  #
    M,
    N,  #
    BLOCK_SIZE_M: tl.constexpr,  #
    BLOCK_SIZE_N: tl.constexpr,  #
    NUM_COMM_SMS: tl.constexpr,
    USE_MULTIMEM_ST: tl.constexpr,
):
    rank = dl.rank()
    world_size = dl.num_ranks()
    pid = tl.program_id(0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_tiles = num_pid_m * num_pid_n
    thread_idx = tid(0)
    block_dim = num_warps() * 32
    VEC_SIZE: tl.constexpr = 128 // tl.constexpr(symm_input_ptr.dtype.element_ty.primitive_bitwidth)

    tl.static_assert(BLOCK_SIZE_N % VEC_SIZE == 0)
    VEC_PER_ROW = BLOCK_SIZE_N // VEC_SIZE
    src_data_mc_ptr = libshmem_device.remote_mc_ptr(libshmem_device.NVSHMEMX_TEAM_NODE,
    ↪  symm_input_ptr)
    if not USE_MULTIMEM_ST:
        for tile_id in range(pid, num_tiles, NUM_COMM_SMS):
            pid_m = tile_id // num_pid_n
            pid_n = tile_id % num_pid_n
            if thread_idx < world_size:
                peer_gemm_barrier_ptr = dl.symm_at(gemm_barrier_ptr, thread_idx)
                while ld(peer_gemm_barrier_ptr + tile_id, scope="sys", semantic="acquire") != 1:
```

```
                pass
            __syncthreads()
            tile_m = min(M - pid_m * BLOCK_SIZE_M, BLOCK_SIZE_M)
            cur_tile_nelem = tile_m * BLOCK_SIZE_N
            for idx in range(thread_idx, cur_tile_nelem // VEC_SIZE, block_dim):
                row_id = idx // VEC_PER_ROW
                col_id = idx % VEC_PER_ROW
                offset = (row_id + pid_m * BLOCK_SIZE_M) * N + col_id * VEC_SIZE + pid_n *
                ↪ BLOCK_SIZE_N
                val0, val1, val2, val3 = multimem_ld_reduce_v4(src_data_mc_ptr + offset)
                st_v4_b32(ar_out_ptr + offset, val0, val1, val2, val3)
    else:
        symm_out_mc_ptr = libshmem_device.remote_mc_ptr(libshmem_device.NVSHMEMX_TEAM_NODE,
        ↪ symm_ar_out_ptr)
        for tile_id in range(pid + rank * NUM_COMM_SMS, num_tiles, NUM_COMM_SMS * world_size):
            pid_m = tile_id // num_pid_n
            pid_n = tile_id % num_pid_n
            if thread_idx < world_size:
                peer_gemm_barrier_ptr = dl.symm_at(gemm_barrier_ptr, thread_idx)
                while ld(peer_gemm_barrier_ptr + tile_id, scope="sys", semantic="acquire") != 1:
                    pass
            __syncthreads()

            tile_m = min(M - pid_m * BLOCK_SIZE_M, BLOCK_SIZE_M)
            cur_tile_nelem = tile_m * BLOCK_SIZE_N
            for idx in range(thread_idx, cur_tile_nelem // VEC_SIZE, block_dim):
                row_id = idx // VEC_PER_ROW
                col_id = idx % VEC_PER_ROW
                offset = (row_id + pid_m * BLOCK_SIZE_M) * N + col_id * VEC_SIZE + pid_n *
                ↪ BLOCK_SIZE_N
                val0, val1, val2, val3 = multimem_ld_reduce_v4(src_data_mc_ptr + offset)
                multimem_st_v4(symm_out_mc_ptr + offset, val0, val1, val2, val3)
        __syncthreads()

        # barrier on all blocks with same pid
        # 0. set barrier to all blocks with same pid on all peer ranks
        if thread_idx < world_size:
            peer_ptr = dl.symm_at(multi_st_barrier_ptr, thread_idx)
            st(peer_ptr + rank * NUM_COMM_SMS + pid, 1, scope="sys", semantic="release")

        # 1. wait barrier
        if thread_idx < world_size:
            multi_st_barrier_idx = thread_idx * NUM_COMM_SMS + pid
            while ld(multi_st_barrier_ptr + multi_st_barrier_idx, scope="sys", semantic="acquire") !=
            ↪ 1:
                pass
            st(multi_st_barrier_ptr + multi_st_barrier_idx, 0)

    # Each block can safely reset the part of the barriers it is waiting for.
    # In low latency kernel, we ensure that the gemm barriers used for two consecutive iteration are
    ↪ different,
    # it can be reset without any sync.
    for tile_id in range(pid + rank * NUM_COMM_SMS, num_tiles, NUM_COMM_SMS * world_size):
        peer_gemm_barrier_ptr = dl.symm_at(gemm_barrier_ptr, thread_idx)
        if thread_idx < world_size:
            st(peer_gemm_barrier_ptr + tile_id, 0, scope="sys", semantic="relaxed")
```

E.4 REGISTER MEGAKERNEL TASK

In this section, we use linear (GEMM) as an example to show how to register kernels as tasks. The following code contains a tile-level GEMM kernel, a chunk-level (multiple tiles) GEMM kernel, and an interface kernel to keep unified function interface for all various tasks. These code are written by users and only minor modification is required to change users' original kernels.

```
@triton.jit
def tile_wise_matmul_compute(tile_id, a_ptr, b_ptr, c_ptr, M, N, K, BLOCK_SIZE_M, BLOCK_SIZE_N,
↪ BLOCK_SIZE_K,
                             NUM_STAGES):
    # linear: a (M, K) x b (N, K) -> c (M, N)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    k_tiles = tl.cdiv(K, BLOCK_SIZE_K)

    offs_k_for_mask = tl.arange(0, BLOCK_SIZE_K)

    pid_m = tile_id // num_pid_n
    pid_n = tile_id % num_pid_n
    start_m = pid_m * BLOCK_SIZE_M
    start_n = pid_n * BLOCK_SIZE_N
    offs_am = start_m + tl.arange(0, BLOCK_SIZE_M)
    offs_bn = start_n + tl.arange(0, BLOCK_SIZE_N)
    offs_am = tl.where(offs_am < M, offs_am, 0)
    offs_bn = tl.where(offs_bn < N, offs_bn, 0)
    offs_am = tl.max_contiguous(tl.multiple_of(offs_am, BLOCK_SIZE_M), BLOCK_SIZE_M)
    offs_bn = tl.max_contiguous(tl.multiple_of(offs_bn, BLOCK_SIZE_N), BLOCK_SIZE_N)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for ki in tl.range(0, k_tiles, num_stages=NUM_STAGES):
```

```
        offs_k = ki * BLOCK_SIZE_K + tl.arange(0, BLOCK_SIZE_K)
        a_ptrs = a_ptr + (offs_am[:, None] * K + offs_k[None, :])
        b_ptrs = b_ptr + (offs_bn[:, None] * K + offs_k[None, :])

        a = tl.load(a_ptrs, mask=offs_k_for_mask[None, :] < K - ki * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k_for_mask[None, :] < K - ki * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b.T, accumulator)

    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + N * offs_cm[:, None] + offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    c = accumulator.to(c_ptr.dtype.element_ty)
    tl.store(c_ptrs, c, mask=c_mask)


@triton.jit
def tile_range_matmul_compute_and_notify(tile_start, sb_base_ptr, a_ptr, b_ptr, c_ptr, M, N, K,
↪  BLOCK_SIZE_M,
                                         BLOCK_SIZE_N, BLOCK_SIZE_K, NUM_STAGES, TILE_READY_SIGNAL,
                                         ↪  NUM_SMS):
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_tiles = num_pid_m * num_pid_n
    for tile_id in tl.range(tile_start, num_tiles, NUM_SMS, flatten=True, warp_specialize=True):
        tile_wise_matmul_compute(tile_id, a_ptr, b_ptr, c_ptr, M, N, K, BLOCK_SIZE_M, BLOCK_SIZE_N,
        ↪  BLOCK_SIZE_K,
                                 NUM_STAGES)
        st(sb_base_ptr + tile_id, TILE_READY_SIGNAL, "gpu", "release")


@triton.jit
def linear_task_compute(task_base_info: TaskBaseInfo, scoreboard: Scoreboard, BLOCK_SIZE_M:
↪  tl.constexpr,
                        BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr, NUM_STAGES:
                        ↪  tl.constexpr,
                        ALIGNMENT_K: tl.constexpr):

    input: TensorDesc = task_base_info.get_tensor(0)
    weight: TensorDesc = task_base_info.get_tensor(1)
    output: TensorDesc = task_base_info.get_tensor(2)

    M = input.size(0)
    K = input.size(1, ALIGNMENT_K)
    N = weight.size(0)
    a_ptr = input.data_ptr(tl.bfloat16)
    b_ptr = weight.data_ptr(tl.bfloat16)
    c_ptr = output.data_ptr(tl.bfloat16)

    tile_id = task_base_info.tile_id_or_start
    tile_wise_matmul_compute(tile_id, a_ptr, b_ptr, c_ptr, M, N, K, BLOCK_SIZE_M, BLOCK_SIZE_N,
    ↪  BLOCK_SIZE_K,
                             NUM_STAGES)
    scoreboard.release_tile(task_base_info, tile_id)
```

Then, we can register the kernels as tasks using the following interface. Users are required to describe the problem size, data dependency, and the tiling configurations.

```
@registry.register_task(op_type="linear", task_cls=LinearTask, config_factory=linear_config_factory,
                        codegen_func=codegen_linear)
class LinearTaskBuilder(TaskBuilderBase):

    @classmethod
    def get_problem_size(cls, io_tensors: List[List['torch.Tensor']], extra_params: Dict[str, Any]):
        a, b = io_tensors[0]
        M, K = a.shape
        N, K = b.shape
        return (M, N, K)

    @classmethod
    def _build_tasks_impl(cls, device_prop, layer_id: int, dependency: TaskDependency, io_tensors,
    ↪  extra_params,
                          tile_wise=True, config_args={}) -> List[TaskBase]:
        assert tile_wise == True  # noqa: E712
        kernel_config = cls.create_config(**config_args)
        task_id = cls.get_task_id(layer_id)
        BLOCK_SIZE_M = kernel_config.BLOCK_SIZE_M
        BLOCK_SIZE_N = kernel_config.BLOCK_SIZE_N
        M, N, K = cls.get_problem_size(io_tensors, extra_params)
        num_tiles_m = cdiv(M, BLOCK_SIZE_M)
        num_tiles_n = cdiv(N, BLOCK_SIZE_N)
        num_tiles = num_tiles_m * num_tiles_n
        x, w = io_tensors[0]
        y = io_tensors[1][0]

        num_sm = device_prop.NUM_SMS
        tasks = []
        cls.log(
```

```
            f"Linear Task: M = {M}, N = {N}, K = {K}, num_tiles = {num_tiles}, num_sm = {num_sm},
            ↪  tile_wise = {tile_wise}, dependency = {dependency}, BLOCK_SIZE_M ={BLOCK_SIZE_M},
            ↪  BLOCK_SIZE_N = {BLOCK_SIZE_N}"
        )
        for tm in range(num_tiles_m):
            for tn in range(num_tiles_n):
                tile_id = tm * num_tiles_n + tn
                bm = min(BLOCK_SIZE_M, M - tm * BLOCK_SIZE_M)
                bn = min(BLOCK_SIZE_N, N - tn * BLOCK_SIZE_N)
                x_desc = InputDependencyDesc(x, require_full=False, start_indices=(tm * BLOCK_SIZE_M,
                ↪  0),
                                             data_sizes=(bm, K))
                w_desc = InputDependencyDesc(w, require_full=False, start_indices=(tn * BLOCK_SIZE_N,
                ↪  0),
                                             data_sizes=(bn, K))
                y_desc = OutputTilingDesc(tile_sizes=(BLOCK_SIZE_M, BLOCK_SIZE_N),
                                          start_indices=(tm * BLOCK_SIZE_M, tn * BLOCK_SIZE_N))
                inputs_dep = {x: x_desc, w: w_desc}
                outs_tile_mapping = {y: y_desc}
                tasks.append(
                    cls._create_task(layer_id, task_id, tile_id, num_tiles, kernel_config,
                    ↪  dependency, io_tensors,
                                     extra_params, inputs_dep, outs_tile_mapping))
        return tasks

    @classmethod
    def build_tasks(cls, device_prop: 'DeviceProp', layer_id: int, dependency: TaskDependency,
                    io_tensors: List[List['torch.Tensor']], extra_params: Dict[str, Any]) ->
                    ↪  List[TaskBase]:
        return cls._build_tasks_impl(device_prop, layer_id, dependency, io_tensors, extra_params)
```

Using the registration interface, multiple tasks can be added. And DITRON will **automatically** generate MegaKernel. Here we show the generated code. Note that the code is auto-generated, not written by users. The task-level primitives such as *fetch_task* and *scoreboard.wait_deps* are used here.

```
@triton.jit
def FETCH_TASK(work_queues, idx, INT_PER_TASK, NUM_SMS, MAX_NUM_TENSOR_DIMS,
↪  ENABLE_RUNTIME_SCHEDUER=False):
    sm_id = tl.program_id(axis=0)
    TASK_TYPE_OFFSET = 0
    LAYER_ID_OFFSET = 1
    TASK_ID_OFFSET = 2
    TILE_ID_OR_START_OFFSET = 3
    DEPEND_ENTRY_START_OFFSET = 4
    DEPEND_ENTRY_END_OFFSET = 5
    IO_TENSORS_OFFSET = 6

    if not ENABLE_RUNTIME_SCHEDUER:
        offset = INT_PER_TASK * NUM_SMS

        task_type = tl.load(work_queues + idx * offset + sm_id * INT_PER_TASK +
        ↪  TASK_TYPE_OFFSET).to(tl.int32)
        layer_id = tl.load(work_queues + idx * offset + sm_id * INT_PER_TASK +
        ↪  LAYER_ID_OFFSET).to(tl.int32)
        task_id = tl.load(work_queues + idx * offset + sm_id * INT_PER_TASK +
        ↪  TASK_ID_OFFSET).to(tl.int32)
        tile_id_or_start = tl.load(work_queues + idx * offset + sm_id * INT_PER_TASK +
        ↪  TILE_ID_OR_START_OFFSET).to(tl.int32)
        depend_entry_start = tl.load(work_queues + idx * offset + sm_id * INT_PER_TASK +
        ↪  DEPEND_ENTRY_START_OFFSET).to(tl.int32)
        depend_entry_end = tl.load(work_queues + idx * offset + sm_id * INT_PER_TASK +
        ↪  DEPEND_ENTRY_END_OFFSET).to(tl.int32)
        io_tensors_ptr = work_queues + idx * offset + sm_id * INT_PER_TASK + IO_TENSORS_OFFSET
    else:
        task_type = tl.load(work_queues + idx * INT_PER_TASK + TASK_TYPE_OFFSET).to(tl.int32)
        layer_id = tl.load(work_queues + idx * INT_PER_TASK + LAYER_ID_OFFSET).to(tl.int32)
        task_id = tl.load(work_queues + idx * INT_PER_TASK + TASK_ID_OFFSET).to(tl.int32)
        tile_id_or_start = tl.load(work_queues + idx * INT_PER_TASK +
        ↪  TILE_ID_OR_START_OFFSET).to(tl.int32)
        depend_entry_start = tl.load(work_queues + idx * INT_PER_TASK +
        ↪  DEPEND_ENTRY_START_OFFSET).to(tl.int32)
        depend_entry_end = tl.load(work_queues + idx * INT_PER_TASK +
        ↪  DEPEND_ENTRY_END_OFFSET).to(tl.int32)
        io_tensors_ptr = work_queues + idx * INT_PER_TASK + IO_TENSORS_OFFSET

    task_base_info = TaskBaseInfo(io_tensors_ptr, task_type, layer_id, task_id, tile_id_or_start,
    ↪  depend_entry_start, depend_entry_end, MAX_NUM_TENSOR_DIMS)
    return task_base_info


@triton.jit
def MEGA_TRITON_KERNEL(

    work_queue_start, # [1, ] int32, init with zero
    work_queues, # [MAX_INS, NUM_SMS, INS], int32
    num_tasks_per_wq, #[num_sms,]
    scoreboard_ptr,
```

```
    task_deps_ptr,  # [num_deps_entry_of_all_tasks, INT_PER_DEPS]

    INT_PER_DEPS: tl.constexpr,
    INT_PER_TASK: tl.constexpr,
    MAX_TASK_ID: tl.constexpr,
    MAX_NUM_TILES_PER_OP: tl.constexpr,
    MAX_NUM_TENSOR_DIMS: tl.constexpr,
    NUM_SMS: tl.constexpr,
    num_warps: tl.constexpr
):


    WARP_SIZE: tl.constexpr = 32
    NUM_THREADS: tl.constexpr = num_warps * WARP_SIZE
    scoreboard = Scoreboard(task_deps_ptr, INT_PER_DEPS, scoreboard_ptr, MAX_TASK_ID, MAX_NUM_TILES_PER_OP, tl.constexpr(1), NUM_THREADS)
    sm_id = tl.program_id(axis=0)


    num_tasks = tl.load(num_tasks_per_wq + sm_id)
    cur_task_idx = 0


    # early exit
    if cur_task_idx >= num_tasks:
        return

    cur_task_base_info = FETCH_TASK(work_queues, cur_task_idx, INT_PER_TASK, NUM_SMS, MAX_NUM_TENSOR_DIMS, ENABLE_RUNTIME_SCHEDUER=False)
    nxt_task_base_info = cur_task_base_info
    nxt_task_idx = cur_task_idx

    while cur_task_idx < num_tasks:



        task_type = cur_task_base_info.task_type

        # task kernel need to set signal for each tile

        scoreboard.wait_deps(cur_task_base_info)


        #### run task ####
        task_base_info = cur_task_base_info

        if task_type == 0: # RMSNormTask
            rmsnorm_task_compute(task_base_info, scoreboard, RMS_EPS=1e-06, BLOCK_SIZE_N = 2048)

        elif task_type == 6: # QKVProjTask
            fc1_task_compute(task_base_info, scoreboard, BLOCK_SIZE_M=16, BLOCK_SIZE_N=64,
                             BLOCK_SIZE_K=256, NUM_STAGES=5)

        elif task_type == 9: # QKNormRopeUpdateKVCacheTask
            rmsnorm_rope_update_kv_cache_task_compute(
                task_base_info, scoreboard, NUM_Q_HEADS=4, NUM_KV_HEADS=1, Q_HEAD_DIM=128,
                V_HEAD_DIM=128, PAGE_SIZE=1, MAX_NUM_BLOCKS_PER_SEQ=1024,
                Q_RMS_EPS=1e-06, K_RMS_EPS=1e-06
            )

        elif task_type == 2: # AttnSplitTask
            attn_gqa_fwd_batch_decode_split_kv_task_compute(
                task_base_info, scoreboard, SM_SCALE=0.08838834764831845, SOFT_CAP=0.0,
                NUM_Q_HEADS=4, NUM_KV_HEADS=1, Q_HEAD_DIM=128, V_HEAD_DIM=128, PAGE_SIZE=1,
                MAX_NUM_BLOCKS_PER_SEQ=1024, BLOCK_N=64, BLOCK_HEAD_DIM=128,
                BLOCK_DPE=0, BLOCK_DV=128, BLOCK_H=16, NUM_KV_SPLITS=32
            )

        elif task_type == 12: # AttnCombineTask
            attn_gqa_fwd_batch_decode_combine_task_compute(
                task_base_info, scoreboard, NUM_Q_HEADS=4, V_HEAD_DIM=128, BLOCK_DV=128,
                NUM_KV_SPLITS=32
            )

        elif task_type == 4: # OProjTask
            fc1_task_compute(task_base_info, scoreboard, BLOCK_SIZE_M=16, BLOCK_SIZE_N=128,
                             BLOCK_SIZE_K=64, NUM_STAGES=7)

        elif task_type == 11: # BarrierAllIntraNodeTask
            barrier_all_intra_node_task_compute(task_base_info, scoreboard)

        elif task_type == 7: # AllReduceTask
            allreduce_task_compute(task_base_info, scoreboard, BLOCK_SIZE=1024)

        elif task_type == 8: # AddTask
            add_task_compute(task_base_info, scoreboard, BLOCK_SIZE=512)

        elif task_type == 1: # MLPFC1Task
            fc1_task_compute(task_base_info, scoreboard, BLOCK_SIZE_M=16, BLOCK_SIZE_N=64,
                             BLOCK_SIZE_K=128, NUM_STAGES=6)

        elif task_type == 3: # SiLUMulUpTask
            silu_mul_up_task_compute(task_base_info, scoreboard, BLOCK_SIZE_M=8, BLOCK_SIZE_N=128)
```

```

        elif task_type == 5: # MLPFC2Task
            fc1_task_compute(task_base_info, scoreboard, BLOCK_SIZE_M=16, BLOCK_SIZE_N=64,
                             BLOCK_SIZE_K=256, NUM_STAGES=6)

        elif task_type == 10: # LinearTask
            linear_task_compute(task_base_info, scoreboard, BLOCK_SIZE_M=16, BLOCK_SIZE_N=128,
                             BLOCK_SIZE_K=128, NUM_STAGES=4, ALIGNMENT_K=16)


        # nxt task

        cur_task_idx = cur_task_idx + 1
        if cur_task_idx < num_tasks:
            cur_task_base_info = FETCH_TASK(work_queues, cur_task_idx, INT_PER_TASK, NUM_SMS,
            ↪  MAX_NUM_TENSOR_DIMS, ENABLE_RUNTIME_SCHEDUER=False)

```