PARALLELKITTENS: SYSTEMATIC AND PRACTICAL SIMPLIFICATION OF MULTI-GPU AI KERNELS

Stuart H. Sul [1] **Simran Arora** [1] **Benjamin F. Spector** [1] **Christopher Ré** [1]

ABSTRACT

Inter-GPU communication has become a major bottleneck for modern AI workloads as models scale and improvements in hardware compute throughput outpace improvements in interconnect bandwidth. Existing systems mitigate this through compute-communication overlap but often fail to meet theoretical peak performance across heterogeneous workloads and new accelerators. Instead of operator-specific techniques, we ask whether a small set of simple, reusable principles can systematically guide the design of optimal multi-GPU kernels. We present PARALLELKITTENS (PK), a minimal CUDA framework that drastically simplifies the development of overlapped multi-GPU kernels. PK extends the THUNDERKITTENS framework and embodies the principles of multi-GPU kernel design through eight core primitives and a unified programming template, derived from a comprehensive analysis of the factors that govern multi-GPU performance—data-transfer mechanisms, resource scheduling, and design overheads. We validate PK on both Hopper and Blackwell architectures. With fewer than 50 lines of device code, PK achieves up to $2.33\times$ speedup for data- and tensor-parallel workloads, $4.08\times$ for sequence-parallel workloads, and $1.22\times$ for expert-parallel workloads.

1 INTRODUCTION

A few years ago, GPU compute utilization was often limited by *intra-GPU* memory access. However, IO-aware algorithms like FlashAttention (Dao et al., 2022), domain-specific languages (DSLs) that support efficient mapping of operators to hardware (Tillet et al., 2019; Nvidia, 2024; Spector et al., 2025), and the continued scaling of AI models have left *inter-GPU* communication as the primary remaining bottleneck. Even with high-speed interconnects like NVLink (Nvidia) and compute-friendly phases like prefill, communication can occupy over 50% of execution time in large language model (LLM) workloads, leaving GPU compute idle (Chang et al., 2024). The problem is compounded by the relatively slow improvements in communication hardware: from the Nvidia A100 (Nvidia, 2020) to the B200 (Nvidia, 2025), BF16 tensor core performance improved by $7.2\times$ and High Bandwidth Memory (HBM) bandwidth by $5.1\times$, while intra-node inter-GPU communication (NVLink) improved by only $3\times$ and inter-node communication (PCIe/InfiniBand) by just $2\times$.

To mitigate communication overhead, prior methods *overlap* inter-GPU communication with intra-GPU computation for common operators like General Matrix Multiplication (GEMM), attention, and Mixture-of-Experts (MoE) layers (Liu et al., 2024; Chang et al., 2024; Zhao et al., 2025; Zheng et al., 2025a; Zhang et al., 2025; Aimuyo et al., 2025). These approaches reduce non-overlapped communication time in data, tensor, sequence, and expert parallelism (Shoeybi et al., 2019; Liang et al., 2025), which are common strategies for distributing industry-scale training and inference across many GPUs. However, prior works either (i) rely on bespoke kernels for specific AI operators and depend on complex low-level primitives (e.g., CUTLASS, NVSHMEM, Linux IPC), (ii) employ compiler-based approaches that fail to adapt to new accelerators—occasionally generating kernels slower than non-overlapped baselines—or (iii) utilize off-the-shelf libraries, resulting in up to $4.08\times$ slower performance than hand-tuned implementations.

As hardware shifts toward unified multi-GPU systems—illustrated by Nvidia's roadmap from NVL72 to NVL144 (2026) and NVL576 (2027) (Nvidia, 2025)—we would need simple, general principles and programming primitives that enable peak-performance multi-GPU operations. In this work, we identify three key principles for designing efficient multi-GPU kernels and analyze each in detail (Sec. 3.1).

1. **Transfer mechanism.** Inter-GPU networking relies on three mechanisms—copy engines, tensor memory accelerators (TMA), and register-level instructions—that differ in maximum bandwidth, effective message granularity, supported functionality, and compute occupancy. Understanding these trade-offs and choosing the right

[1]Department of Computer Science, Stanford University. Correspondence to: Stuart H. Sul <ssul@cs.stanford.edu>.

Proceedings of the 9^{th} MLSys Conference, Bellevue, WA, USA, 2026.

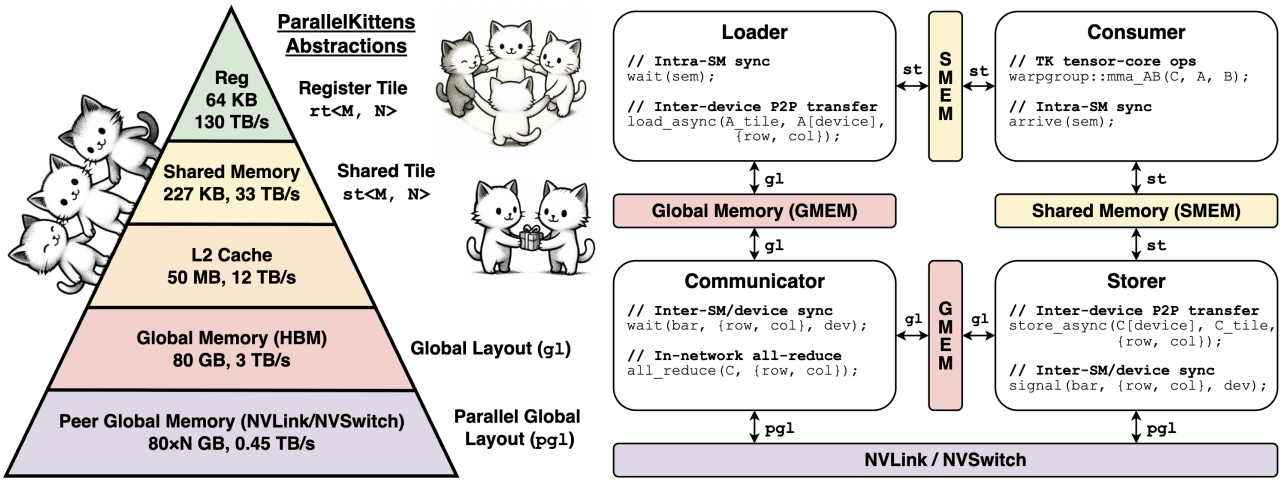

Figure 1. We study the principles for high performance multi-GPU kernels and introduce PARALLELKITTENS (PK), an opinionated collection of programming primitives to encapsulate these principles. The GPU memory hierarchy and corresponding PK abstractions are shown on the left (Sec. 3.2.1), and the PK program template with its key multi-GPU kernel components is shown on the right (Sec. 3.2.3).

mechanism is crucial for peak performance. For instance, copy engines achieve the highest efficiency (81% of theoretical maximum) but require large messages ($\geq$ 256 MB) for saturation. TMA attains near-peak throughput (74%) with only 2 KB messages (Figure 2). Register-level instructions operate efficiently at a 128 B granularity but need about 76 streaming multiprocessors (SMs) to saturate bandwidth (70%), whereas TMA needs 15 (Figure 3). However, only register-level instructions support in-network reduction. Existing systems do not capture these trade-offs. For instance, Triton Distributed, Flux, and CUTLASS rely on the copy engine for intra-node all-gather GEMM, and this design choice leads to worse performance than the non-overlapped baseline on smaller matrix sizes (Figure 7).

2. **Scheduling.** The distribution of compute and communication work across SMs must be chosen based on workload characteristics. We identify *inter-SM* and *intra-SM* overlapping as the two primary scheduling strategies, trading off compute utilization and communication versatility. Intra-SM overlapping is preferred when computation and communication granularities align; for example, in GEMM reduce-scatter, intra-SM overlapping outperforms inter-SM by $1.2\times$. In contrast, inter-SM overlapping enables communication patterns that can significantly reduce transfer size. For instance, leveraging in-network reduction through inter-SM overlapping achieves a $3.62\times$ performance improvement for GEMM all-reduce (Figure 5) and $1.57\times$ for all-gather GEMM. No prior work explores both scheduling strategies; existing methods either rely on a single type or omit device-side overlapping altogether, thereby failing to generalize (e.g., applying the Flux intra-SM overlapping design to GEMM all-reduce would lead to the slowdown above).

3. **Design overheads.** Widely used communication libraries (e.g., NCCL, NVSHMEM) encapsulate design choices—specifically in synchronization and buffering—that aim for simplicity but degrade performance. We show that the choices in prior libraries can cause over $1.7\times$ performance loss in pure communication kernels (e.g., all-reduce) and up to $4.5\times$ higher communication latency. By adopting a design that enables explicit user control over memory allocation and synchronization, these overheads can be substantially reduced.

Building on these insights, we introduce **PARALLELKITTENS** (PK), an opinionated collection of C++ embedded programming primitives that extends the THUNDERKITTENS (TK) framework (Spector et al., 2025) (Sec. 3.2). PK exposes only the most efficient transfer mechanisms for each functionality (e.g., TMA for point-wise communication, register operations for in-network acceleration), provides minimal synchronization primitives and a general program template that simplifies achieving both inter- and intra-SM overlapping scheduling, and offers full control over performance-critical components (e.g., NVLink transfers) while abstracting away non-essential multi-GPU complexities (e.g., inter-process communication and virtual memory exchange).

We validate PK across diverse parallel AI workloads on both Hopper and Blackwell architectures, including data, tensor, sequence, and expert parallelism (i.e., fused parallel GEMMs, distributed attention variants, and MoEs). Compared with the strongest baselines, PK achieves up to $2.33\times$ higher compute throughput (FLOP/s) for data- and tensor-parallel workloads, $4.08\times$ for sequence-parallel workloads, and $1.22\times$ for expert-parallel workloads, effectively reducing non-overlapped communication time down to 1%, 9%, and 15%, respectively. PK matches the performance of

the strongest hand-optimized kernels (Flux, Comet, CUTLASS), outperforms compiler-based approaches (Triton Distributed) by 1.07–5.63×, and surpasses communication library-based approaches (xDiT, YunChang) by 1.01–4.08× across varying problem sizes.

Each PK kernel required fewer than 50 lines of additional device code beyond the original single-GPU GEMM or attention kernels. The complete implementation of PK, including its kernels, is fully open-sourced[1] and has been adopted at Cursor for large-scale in-house training of Composer 2 (Chan et al., 2026).

To summarize, our contributions are:

- A detailed analysis of multi-GPU programming that decomposes performance into interpretable factors (transfer mechanisms, scheduling strategies, and design overheads) and validates each through microbenchmarks.
- PARALLELKITTENS, a minimal collection of multi-GPU primitives and a unified programming template that extends the familiar THUNDERKITTENS framework.
- Kernels built with PARALLELKITTENS that match or surpass hand-optimized kernel performance while substantially reducing code complexity.

2 BACKGROUND

We provide background on modern datacenter GPUs and review prior efforts on optimizing multi-GPU AI kernels.[2]

2.1 GPU Architecture

A GPU kernel loads data from HBM, performs computation, and writes the results back to HBM. Multi-GPU kernels distribute the workload across multiple GPUs and access the HBMs of all devices.

GPU hierarchy. GPU kernels execute tens of thousands of hardware *threads* in parallel across over a hundred *streaming multiprocessors* (SMs). Memory farther from the SM provides greater capacity at higher latency. Each SM contains 64 KB of registers private to individual threads and accessible every clock cycle. Threads are organized into *thread blocks*, each executing on a single assigned SM. Threads in a thread block communicate via 227 KB of shared memory (SMEM), a per-SM on-chip SRAM offering up to 33 TB/s of bandwidth. All threads share a 50 MB L2 cache (≈12 TB/s) connected to 80 GB HBM (3 TB/s). Threads can also access *peer GPU HBM* over NVLink (450 GB/s unidirectional), enabling the development of multi-GPU kernels.

GPU networking. Multi-GPU systems rely on a hierarchy of interconnects. *PCIe* (64 GB/s) is the channel for CPU-to-GPU (e.g., kernel launches, host-initiated transfers) and multinode communication over InfiniBand/TCP. *NVLink* (450 GB/s) provides point-to-point connections between GPUs and the NVSwitch; *NVSwitch* interconnects all NVLink endpoints into a non-blocking fabric for full GPU-to-GPU communication. NVSwitch also supports in-network, off-device acceleration for multicast and reduction. Unless otherwise noted, all inter-GPU communication in this paper occurs via NVLink/NVSwitch.

Execution overlap. GPUs contain various execution units specialized for different compute, memory, and communication operations. For compute, *Tensor Cores* perform tiled matrix multiplications, while *CUDA Cores* handle element-wise arithmetic. For memory, the *Tensor Memory Accelerator* (TMA) performs bulk data transfers between SMEM and HBM and can be invoked asynchronously by a single thread. Alternatively, a per-GPU *copy engine* (dedicated DMA unit) moves large contiguous regions of device memory independently of the SMs and is invoked from the host.

Within an SM, threads can concurrently issue instructions to different execution units. Achieving optimal performance therefore depends on effectively overlapping their use to hide non-critical operations and maximize the throughput of critical ones. We distinguish *inter-SM overlapping*, where entire SMs are dedicated almost exclusively to compute, memory, or communication tasks, from *intra-SM overlapping*, where different threads within the same SM concurrently drive compute, memory, or inter-GPU traffic (Bauer et al., 2011). These resources saturate at different rates, creating opportunities for various overlapping strategies.

2.2 Related Works

We are inspired by the extensive amount of work that accelerates multi-GPU AI workloads.

Operator-specific kernels. Many prior works hand-tune particular AI operators by overlapping computation and communication, e.g., TP-Async (He et al., 2024), Flux (Chang et al., 2024), Ring Attention (Liu et al., 2024), DeepEP (DeepSeek-AI et al., 2024), Comet (Zhang et al., 2025), FlashDMoE (Aimuyo et al., 2025), and several distributed GEMM kernels from CUTLASS (Thakkar et al.). These approaches employ techniques ranging from overlapping host-triggered copies with device kernels, to highly optimized on-device schedulers and device-initiated com-

[1] https://github.com/HazyResearch/ThunderKittens

[2] Unless otherwise specified, we use the Nvidia HGX H100 (Tsu, 2022) platform with 8×H100 80GB SXM GPUs, 4th generation NVLink/NVSwitch, and 5th generation PCIe as our running example; however, in general, the principles extend to other modern platforms (e.g., Blackwell architecture) and hardware vendors (e.g., AMD GPUs).

Table 1. The observed NVLink bandwidth utilization (GB/s) when using all SMs to transfer 1GB of data, and its ratio to the theoretical maximum (450 GB/s for H100s, 900 GB/s for B200s).

METHOD	H100 BW (RATIO)	B200 BW (RATIO)
COPY ENGINE	368.82 (82%)	726.13 (81%)
TMA OP	350.01 (78%)	669.12 (74%)
REGISTER OP	342.68 (76%)	628.35 (70%)

munication (Potluri et al., 2017). While these systems deliver strong performance for specific targets, they demand complex implementations and offer limited reusable abstractions. For instance, FlashDMoE is optimized only for TF32 precision, with BF16/FP16 support still under development five months after its release. In contrast, PK distills general principles applicable across diverse workloads, achieving speedups comparable to hand-optimized kernels while simplifying implementation.

Scheduling frameworks. Frameworks such as Megatron-LM (Shoeybi et al., 2019), FlexFlow (Jia et al., 2019), and NanoFlow (Zhu et al., 2025) automate parallelization and scheduling, and are complementary to PK. These systems primarily orchestrate bulk collective operations (e.g., NCCL), which require synchronization before and after data transfers, and employ stream-level overlap. NanoFlow offers finer-grained scheduling by partitioning SMs among compute, memory, and network operations to saturate available bandwidth without full occupancy (i.e., inter-SM overlapping). However, achieving peak kernel performance also requires intra-SM warp specialization with device-initiated, tile-level transfers; PK provides that layer.

Multi-GPU programming primitives. DSLs and libraries have been proposed to simplify multi-GPU kernel development (Nvidia, 2025a; Nvidia; Zheng et al., 2025a; AMD, 2025; Google, 2025). Triton Distributed (Zheng et al., 2025a) and TileLink (Zheng et al., 2025b) extend Triton (Tillet et al., 2019) with OpenSHMEM-style one-sided operations, enabling compiler-based generation of multi-GPU kernels. However, these approaches lack explicit workload distribution control (e.g., warp or SM specialization) needed for optimal overlap. Also, our benchmarks show that Triton Distributed, originally tuned for H800 GPUs, fails to adapt efficiently to other architectures such as H100s (Sec. 4). In contrast, PK provides a lightweight C++ layer that enables direct control over communication workload distribution, enabling arbitrary scheduling and optimization across Hopper and Blackwell GPUs. NCCLX (Si et al., 2025) complements PK by accelerating *inter-node* collectives for large clusters ($\geq$ 100k GPUs), but does not exploit device-initiated asynchronous overlapping (via TMA) or in-network acceleration, both critical for fine-grained overlap with peak bandwidth utilization.

3 PARALLELKITTENS

We present our analysis of the design tradeoffs of multi-GPU kernels and present PARALLELKITTENS.

3.1 Analysis

We start with a general, high-level cost model that provides a roadmap for the analysis.

3.1.1 Cost Model

The objective of designing a multi-GPU kernel is to minimize its total wall-clock time T_{kernel}, which reflects the combined cost of compute, memory, and communication operations. The key contributors include:

$$T_{\text{kernel}} = T_{\text{launch}} + \max(T_{\text{comp}}, T_{\text{mem}}, T_{\text{comm}}) + T_{\text{non-overlap}} + T_{\text{sync}}$$

In this simple model, T_{launch} denotes the per-kernel launch cost, including host-side latency and per-thread block setup and teardown (e.g., tensor memory allocation and pipeline fill/drain phases). T_{comp}, T_{mem}, and T_{comm} represent the full-pipeline time spent on computation, memory access, and communication, respectively. Ideally, these components overlap so that the total time equals the maximum of the three, but $T_{\text{non-overlap}}$ accounts for operations that cannot be overlapped. The cost of each component (e.g., T_{comm}) depends on the work size (S_{comm}) and achievable bandwidth (B_{comm}), i.e., $T_{\text{comm}} = S_{\text{comm}}/B_{\text{comm}}$. Finally, T_{sync} captures the synchronization overhead across SMs or devices.

These costs are controlled by three design decisions: first, the specific **transfer mechanism** that we select to move data between GPUs (Sec. 3.1.2); second, the kernel **scheduling strategy** for overlapping computation and communication (Sec. 3.1.3); and third, the communication abstraction's **design choices**, including peer-memory allocation, management, and access (Sec. 3.1.4).

3.1.2 Transfer Mechanism

We now discuss the choice of communication mechanism.

Host versus device-initiated communication. The per-GPU copy engine is host-initiated and supports only contiguous memory transfers. As shown in Table 1, it achieves the highest throughput for large, all-at-once data movements. However, when fine-grained communication is required (e.g., all-to-all communication in MoEs), performance degrades significantly because additional overhead is incurred for data rearrangement or repeated transfer invocations. Figure 2 illustrates this behavior. To sustain over 80% bandwidth utilization, the transfer granularity must be at least 256 MB when using the copy engine, whereas device-side methods achieve comparable utilization with only 2 KB.

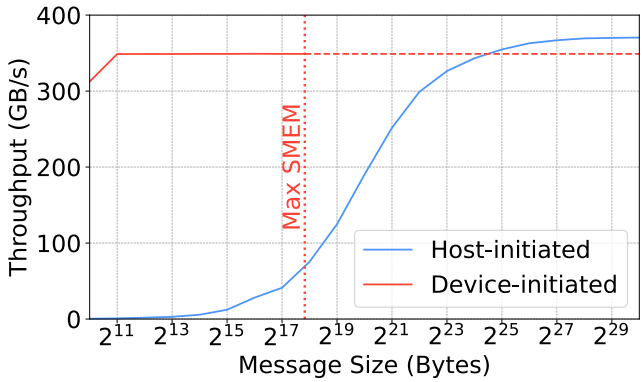

Figure 2. Observed memory bandwidth utilization for a 1 GB peer-to-peer transfer over NVLink. For device-initiated (TMA) transfers, the maximum supported message size is 227 KB; throughput values beyond this limit are held constant for visual comparison.

Consequently, PK relies exclusively on device-side communication for the following reasons. First, host-initiated transfers are suitable primarily for large contiguous data blocks (e.g., weight movements in fully sharded data parallelism (Zhao et al., 2023)). In such cases, overlapping computation and communication is often trivial: the host transfer and device kernel can be launched on separate streams without kernel modifications. Second, although the copy engine has the advantage of not occupying SM resources, only a small number of SMs are needed to saturate the interconnect bandwidth using device-initiated communication, as shown in Figure 3. Moreover, intra-SM overlapping enables computation to proceed concurrently with that communication.

Device-initiated communication mechanisms. There are two main mechanisms for device-initiated communication:

1. The first is via the Tensor Memory Accelerator (TMA), which supports NVLink transfers and NVSwitch-accelerated broadcasts. A key advantage of TMA is that it can be launched *asynchronously* by a *single thread* without increasing register pressure, allowing other threads in the same SM to overlap the execution of compute or memory work (intra-SM overlap).
2. The second is via plain register-level instructions (e.g., `ld`, `st`). As shown in Table 1, they are relatively inefficient, achieving about 70% of the peak bandwidth on B200 GPUs. Because these instructions are synchronous and operate at the register level, saturating NVLink bandwidth requires full SM occupancy—thousands of threads issuing instructions concurrently—as well as higher register pressure and manual memory coalescing.

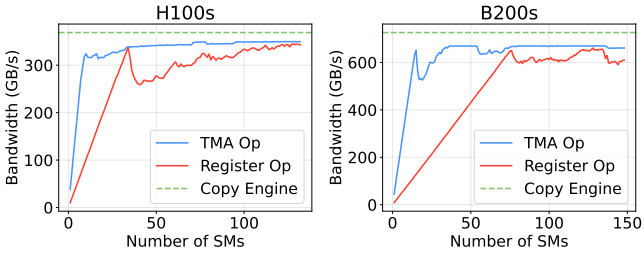

Figure 3. The number of SMs it takes to saturate NVLink Bandwidth, using different communication mechanisms.

Table 2. Comparison of multi-GPU transfer mechanisms (copy engine (CE), TMA, and register-based operations) and their supported functionalities. Here, P2P transfer denotes bulk data movement between two GPUs; in-fabric broadcast refers to multicast from a single source GPU to all GPUs; P2P reduction denotes pointwise reduction to a single destination GPU; in-fabric reduction refers to off-device aggregation of data across all participating GPUs; elementwise transfer denotes byte-level communication between two GPUs.

FUNCTIONALITY	CE	TMA	REG
P2P TRANSFER	✓	✓	✓
IN-FABRIC BROADCAST	✓	✓	✓
P2P REDUCTION	×	✓	✓
IN-FABRIC REDUCTION	×	×	✓
ELEMENTWISE TRANSFER	×	×	✓

We find that these mechanisms excel in different scenarios. As illustrated in Figure 3, register-level operations require 3.2–5.1× more SMs than TMA to saturate NVLink bandwidth, leaving little opportunity for intra-SM overlap. Register instructions are therefore useful when neither the copy engine nor TMA provides the required functionality. A representative case is NVSwitch in-network reduction (e.g., `multimem.ld_reduce` and `multimem.red`), which can substantially speed up workloads like all-reduce. Existing communication libraries do not exploit this design space; for instance, NVSHMEM relies exclusively on register-level operations for intra-node data transfers. Table 2 summarizes the functionalities supported by each mechanism.

3.1.3 *Scheduling*

We now examine workload scheduling strategies for multi-GPU kernels. There are two main ways to overlap compute and communication within a kernel:

1. *Intra-SM* overlapping partitions the threads within an SM into two pools: one issuing compute/memory instructions and the other issuing communication instructions.
2. *Inter-SM* overlapping partitions the SMs into two pools: one for computation and the other for communication.

While prior work primarily uses inter-SM overlapping (Zhu et al., 2025; Zhang et al., 2025), we find that each provides different compute-communication trade-offs depending on workload characteristics.[3]

[3]The focus on inter-SM overlap in prior work is largely due to limitations in pre-Hopper architectures, which lacked single-thread bulk asynchronous transfers and therefore required entire warps or thread blocks to participate, increasing register pressure.

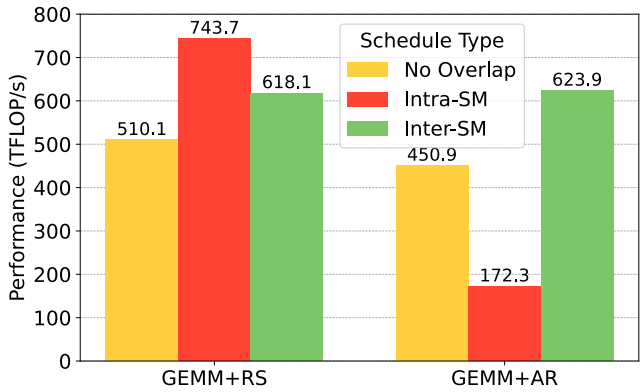

Figure 4. GEMM reduce-scatter (RS) and all-reduce (AR) performance across overlapping schedules. Measured on 8×H100 GPUs with local GEMM shape $N \times N \times N/8$ ($N = 32768$) and element type BF16.

Intra-SM overlapping. Intra-SM overlapping is effective when the ideal communication pattern aligns with that of computation, allowing communication to be naturally embedded within the computation pipeline. In such cases, it is superior to inter-SM overlapping for two main reasons:

1. Unlike in inter-SM overlapping, all compute units (i.e., tensor cores) across all SMs are busy in an intra-SM overlapping scheme. This is crucial because, unlike communication bandwidth, compute throughput scales linearly with the number of SMs that perform computation.

2. Inter-SM communication incurs additional synchronization overhead T_{sync}, as it must traverse the HBM. Our microbenchmarks show that a single intra-SM synchronization using `mbarrier` objects incurs approximately 64 ns of latency, whereas inter-SM synchronization through the HBM takes about 832 ns.

We illustrate these effects using a kernel that fuses a GEMM with a reduce-scatter (RS). Figure 4 (left) shows that the GEMM+RS kernel achieves higher compute throughput under an intra-SM overlapping schedule, due to higher compute utilization and lower synchronization overhead.

We further show that intra-SM overlapping can almost completely hide communication overhead in certain regimes. Consider an $M \times N \times K$ GEMM+RS fused kernel with per-iteration tiles of size $m \times n \times k$. In a typical GEMM kernel, an output tile region is selected, and the $m \times n \times k$ sub-GEMM is executed K/k times before the result is stored.

Given the per-element size s, sustained tensor core throughput R (in FLOP/s), and per-GPU NVLink bandwidth B (in bytes/s), the compute and communication times for producing a single output tile of size $m \times n$ are given by:

$$T_{\text{comp\_tile}} = \frac{2mnk}{R} \times \frac{K}{k} = \frac{2mnK}{R}$$

Table 3. Measured BF16 GEMM and GEMM+RS performance (ms).

$M\&N$	K	GEMM	GEMM+RS	COMM RATIO
32768	512	2.071	6.483	**68%**
32768	1024	2.918	6.613	**56%**
32768	2048	5.567	7.531	**26%**
32768	4096	11.78	11.828	**<1%**
32768	8192	23.285	25.325	**8%**

$$T_{\text{comm\_tile}} = \frac{smn}{B}$$

From this, communication can be completely hidden by computation when $T_{\text{comp\_tile}} \geq T_{\text{comm\_tile}}$, i.e.,

$$K \geq \frac{sR}{2B}$$

For BF16 GEMM on H100 GPUs, $s = 2$, $R = 989 \times 10^{12}$, and $B = 450 \times 10^9$, implying that communication is hidden when $K \gtrsim 2197$. We verify this empirically in Table 3, where we ablate our fused GEMM+RS kernel against a standalone GEMM kernel. The results show that at $K = 2048$, the non-overlapped communication ratio drops by roughly half, and beyond that, communication becomes nearly fully hidden. The residual communication time near $K = 2048$ arises from atomic additions required for output tile accumulation, which prevent complete overlap.

Inter-SM overlapping. While intra-SM overlapping fully utilizes GPU compute, it constrains communication to follow the computation pattern. This leads to two potential drawbacks: the inability to exploit in-network acceleration and suboptimal L2 caching behavior. Inter-SM overlapping mitigates these issues but introduces a partitioning tradeoff: deciding how many SMs to allocate to communication versus computation.

In-network acceleration. Recent networking hardware integrates compute directly into the interconnect fabric, enabling in-network reductions and collective offload within switches and link controllers (Nvidia, 2025b; 2024). This transforms the interconnect from a passive data mover into an active participant in collectives. For communication-heavy kernels such as fused GEMM all-reduce (AR), in-network reduction can significantly reduce bandwidth usage. However, performing it within the same SM is impractical due to register pressure, limited occupancy, and inter-GPU synchronization costs. A more effective approach is to accumulate partial results in HBM, signal completion after each local write, and delegate a few specialized SMs to execute a single in-network all-reduce once all devices finish.

This tradeoff is shown in Figure 4 (right). Intra-SM overlapping issues N atomic writes to N destinations for each output tile, where N is the number of GPUs. Even with a

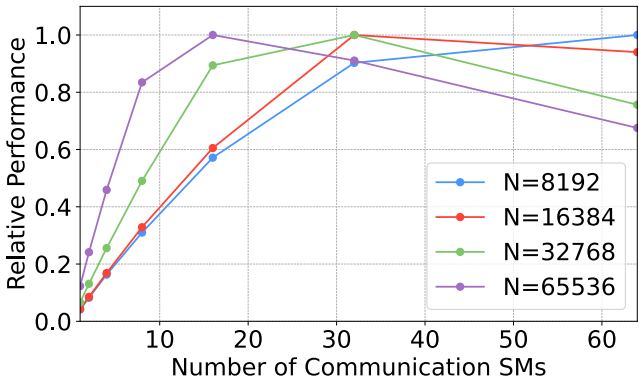

Figure 5. Performance comparison of different inter-SM schedules on all-gather (AG) GEMM ($N \times N/8 \times N$).

fully interconnected NVSwitch fabric, each GPU is limited by its 450 GB/s per-port bandwidth, causing concurrent peer writes to serialize at the destination. Inter-SM overlapping reduces T_{comm} by roughly a factor of N, typically outweighing the cost of dedicating a few SMs to communication.

Remote cache reuse. Another limitation of intra-SM overlapping arises from the *far-sided* nature of L2 caching for peer HBM accesses. Data fetched from a peer GPU is cached only on the source device, not on the requester. Consequently, every remote access is bottlenecked by NVLink bandwidth. A representative case appears in Ring Attention (Liu et al., 2024), where key and value (KV) tensors are reused across multiple attention blocks. Letting each thread block independently load them from remote GPUs leads to redundant transfers and rapid interconnect saturation. Instead, performing bulk transfers of the next block's K and V tensors to local HBM using communication-dedicated SMs, while the remaining SMs compute, substantially reduces T_{comm} and improves L2 reuse, as shown in Sec. 4.2.

SM partitioning. Inter-SM overlapping requires balancing SMs between communication and computation. As shown in Figure 5, the optimal split depends on input size: larger workloads favor more compute SMs, while smaller ones need proportionally more SMs for communication. PK allows users to automatically search for the optimal SM allocation at runtime through a unified program template.

3.1.4 *Minimizing design overheads*

Ideally, abstractions should preserve the developer's ability to achieve peak hardware performance. In practice, however, certain design choices in widely used communication libraries like NCCL and NVSHMEM—particularly in synchronization and buffering—constrain this ability.

Two-way synchronization and intermediate buffering. Many multi-GPU communication libraries impose synchronization and buffering constraints. For example, NCCL enforces two-way synchronization for every operation: both

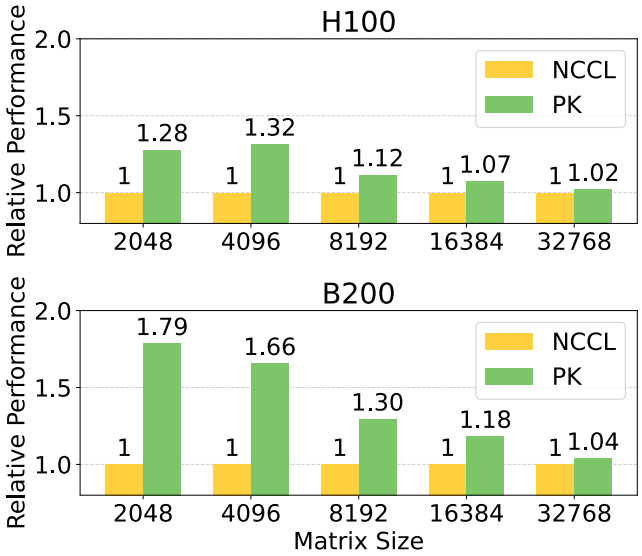

Figure 6. All-reduce sum kernel comparison (BF16).

sender and receiver must be ready and acknowledge each other before data transfer begins, even for point-to-point communication. In addition, to reduce peer memory exchange overhead, NCCL employs small pre-allocated intermediate buffers (communication channels), introducing extra data movement. While such overheads are masked for large inputs, they become significant in fine-grained communication. PK avoids these issues by using pre-allocated destination buffers, enabling direct, one-way transfers without intermediate staging. As shown in Figure 6, this design improves the performance of pure communication kernels such as all-reduce by up to $1.79\times$.

Peer-memory access and synchronization. NVSHMEM, the de facto low-level standard for multi-GPU communication, also introduces additional overhead in its public API functions. Each remote peer access performs a global memory load (`__ldg`) to retrieve the peer address and enforces a group synchronization (e.g., `__syncthreads`). By keeping peer addresses in registers and removing unnecessary synchronizations, PK eliminates these costs, achieving up to $4.5\times$ lower element-wise NVLink access latency and about 20 GB/s higher bandwidth utilization.

3.2 Abstractions

We introduce PARALLELKITTENS (PK), a collection of abstractions that generalizes the tile-based programming principles proposed in THUNDERKITTENS—and successor systems such as CuTe DSL and TileLang—to the multi-GPU setting. PK provides a minimal and complementary set of primitives for efficient multi-GPU communication. These abstractions expose high-performance communication mechanisms, enable the workload scheduling patterns described earlier, and minimize performance overheads by design. PK hides low-level complexity that does not impact

performance, while preserving full user control through its CUDA/C++ embedded design.

3.2.1 Data Structure

PK defines a data structure for each level of the GPU memory hierarchy, as illustrated in Figure 1 (left). At the *register* level, the minimum unit of execution is a 16×16 tile, consistent with the original TK design. At the *shared memory* level, users operate on shared tiles that enable asynchronous, tile-granularity loads from and stores to peer HBM by a single thread. Store operations optionally support atomic reductions on peer memory and multicast to multiple devices via in-network broadcast. These operations preserve tensor-core–friendly layouts to remain efficient within local compute pipelines. At the *HBM* level, PARALLELKITTENS introduces the **Parallel Global Layout (PGL)**, which represents identically shaped and sized memory regions allocated across all devices. PGL serves as the central data structure enabling asynchronous P2P transfers, broadcasts, and synchronous in-fabric multicasts and reductions over tile-indexed regions. All data abstractions enforce essential principles such as coalesced interconnect access, swizzling to minimize bank conflicts and match tensor-core layouts, and fully device-initiated communication.

3.2.2 Multi-GPU Operations

We introduce eight new primitives that are sufficient to implement all kernels demonstrated in Sec. 4. We also note that the original THUNDERKITTENS operators are extended to remain fully compatible with the aforementioned data structures.

P2P communication primitives

- `store_async(dst, src, coord)`
- `store_add_async(dst, src, coord)`

Network-accelerated communication primitives

- `reduce(dst, dst_coord, src, src_coord)`
- `all_reduce(dst_and_src, coord)`

Inter-device and inter-SM synchronization primitives

- `signal(bar, coord, dev_idx, val)`
- `signal_all(bar, coord, val)`
- `wait(bar, coord, dev_idx, expected)`
- `barrier(bar, coord, dev_idx)`

Because all PK data structures are tile-based, the new primitives also operate at tile granularity, ranging from 16×16 (the minimum tile) up to the shared-memory limit (about 256×256). All operations are device-initiated and use coordinates (`coord`) represented as `int4` values specifying tile indices in local or remote HBM. P2P primitives are asynchronous and single-threaded, enabling fusion with other operations (e.g., tensor-core compute), whereas network-accelerated primitives require at least warp-level participation for optimal throughput. Synchronization primitives provide simple signaling and waiting mechanisms, enabling users to design arbitrary workload scheduling schemes. A complete API description is provided in Appendix C.

3.2.3 Program Template

We provide a unified program template for implementing a wide range of multi-GPU kernels. As shown in Figure 1 (right), the template defines four worker components—*loader*, *storer*, *consumer*, and *communicator*—each encapsulating a common warp/SM specialization. The *loader* performs local or peer HBM reads, while the *storer* handles local or peer HBM writes. When either component accesses peer HBM, intra-SM overlapping is employed. The *communicator* occupies one or more SMs exclusively to perform dedicated communication, enabling inter-SM overlapping. Finally, the *consumer* issues tensor- or CUDA-core–based local compute. Beyond providing a structural pattern, the template automates common low-level tasks, including kernel configuration, shared memory and TMA setup, barrier and synchronization management, and tuning for optimal SM/warp partitioning. This allows users to focus solely on the per-tile compute and communication logic. A detailed description of the template and an example kernel are provided in Appendix D.

3.2.4 Utilities

We provide inter-process communication (IPC) and PyTorch utilities for seamless integration with multi-process execution (e.g., via `torchrun`). These utilities manage low-level OS driver interactions and support pre-allocation of multi-GPU memory, enabling direct P2P communication without intermediate staging overheads. Appendices E and F provide further implementation details.

4 EXPERIMENTS

We demonstrate that PK generalizes across a diverse range of multi-GPU AI workloads by implementing representative kernels with its abstractions and comparing them against existing frameworks and hand-optimized baselines.

All experiments were conducted using $8\times$Nvidia H100 80GB SXM GPUs, interconnected via 4th-generation NVLink and NVSwitch, using CUDA 12.8 and PyTorch 2.8.0. All matrix multiplications use BF16 as the element type and FP32 as the tensor core accumulator type. For brevity, we denote the GEMM shape as $M \times N \times K$, where the first operand has dimensions $M \times K$ and the

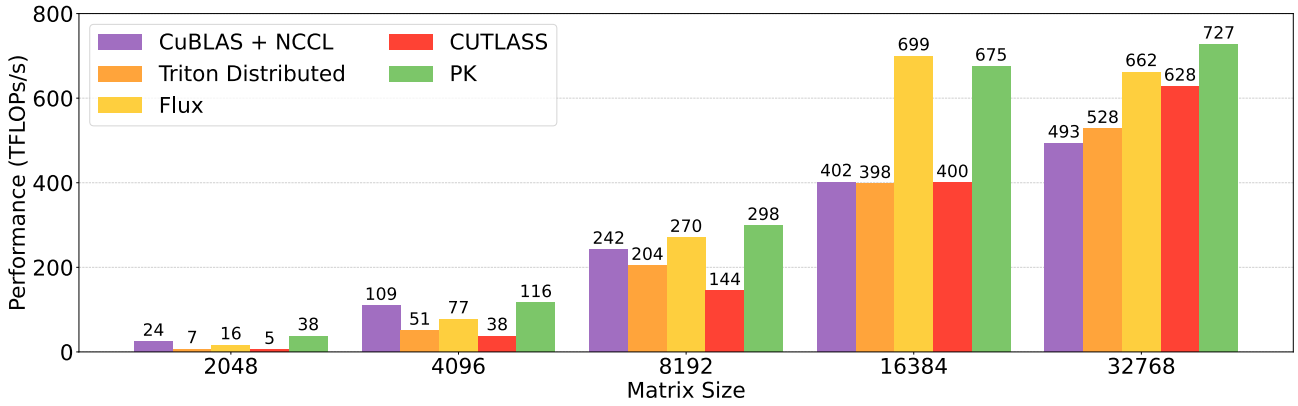

Figure 7. AG + GEMM performance. Local GEMM size is $N \times N/8 \times N$, with N given on the X-axis.

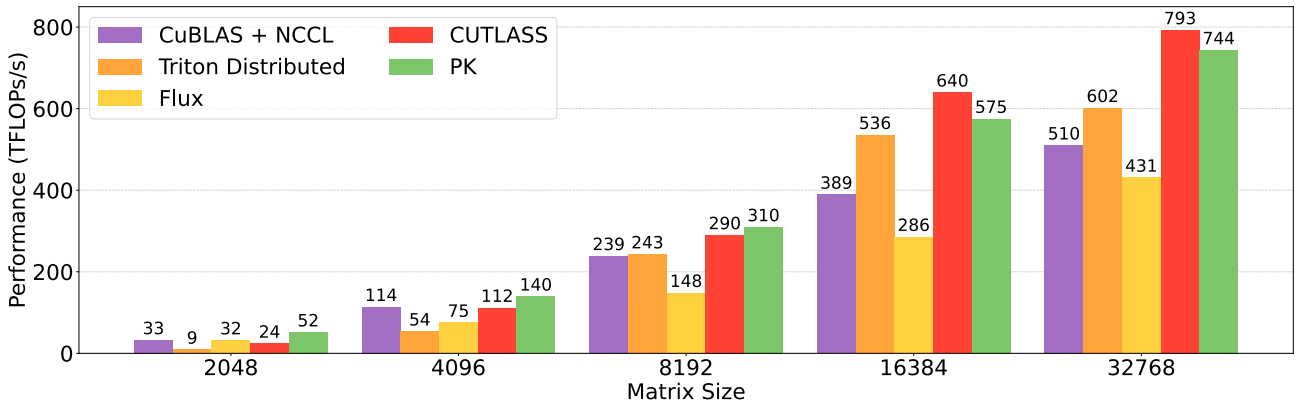

Figure 8. GEMM + RS performance. Local GEMM size is $N \times N \times N/8$, with N given on the X-axis.

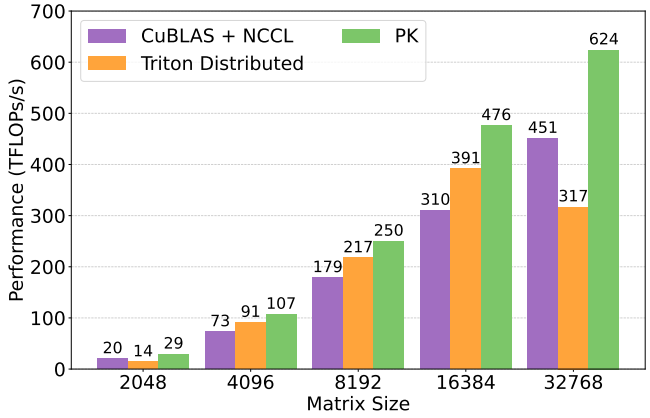

Figure 9. GEMM + AR performance. Local GEMM size is $N \times N \times N/8$, with N given on the X-axis.

second has dimensions $K \times N$. We report the observed average compute throughput.

Although the experiments in this section use H100 GPUs, PK is fully compatible with Blackwell GPUs and exhibits similar performance characteristics. We present performance results for various multi-GPU AI kernels on B200 GPUs in Appendices A and B.

4.1 Data and Tensor Parallelism

To efficiently scale large models, weights are often sharded across multiple devices using *tensor parallelism* (Shoeybi et al., 2019; Zheng et al., 2022), which partitions weight matrices along the row or column dimension. A common strategy combines this with *data parallelism* (Narayanan et al., 2021): inputs sharded by rows are first all-gathered (AG), followed by a GEMM with column-sharded weights, a nonlinear activation, and a second GEMM with row-sharded weights, after which a reduce-scatter (RS) or all-reduce (AR) is applied. Communication and computation are overlapped by pairing AG with the first GEMM (AG+GEMM) and RS or AR with the second (GEMM+RS, GEMM+AR).

For these workloads, we compare against the cuBLAS GEMM combined with NCCL as the non-overlapped baseline, compiler-based approaches (Triton Distributed), and hand-optimized kernels (Flux and CUTLASS). Flux and CUTLASS do not provide GEMM–AR kernels and are therefore omitted in those cases. Figures 7, 8, and 9 show the results. Overall, PK achieves a 1.06–1.68× speedup over the non-overlapped baseline and outperforms compiler-based approaches by 1.07–5.63×. Compared to hand-optimized kernels, PK matches or surpasses their perfor-

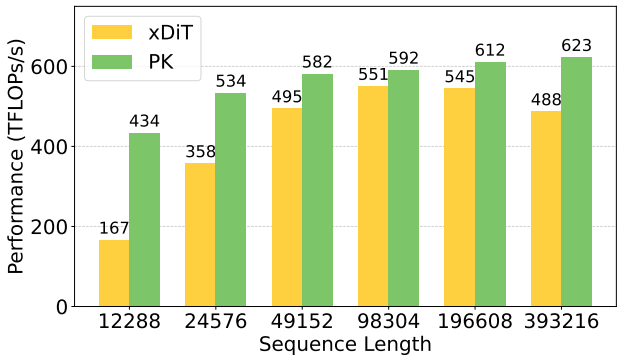

Figure 10. Ring Attention performance across sequence lengths ($B = 16$, $H = 16$, $D = 128$).

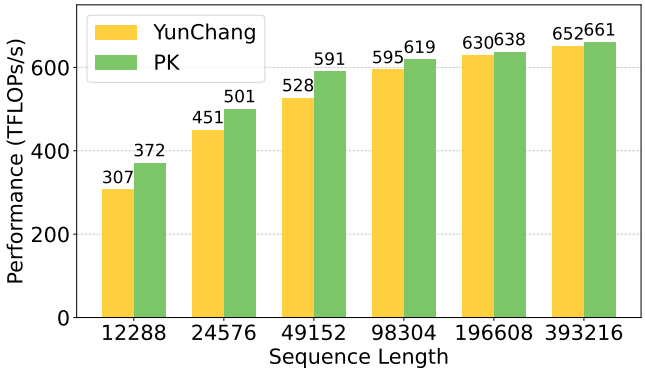

Figure 11. DeepSpeed-Ulysses attention layer performance across sequence lengths ($B = 16$, $H = 128$, $D = 128$).

mance, achieving 0.97–2.33× speedup over Flux and 0.90–7.39× over CUTLASS. We also note that AG+GEMM and GEMM+RS are often used back-to-back in practice, and no single baseline outperforms PK when both are combined.

We further observe that compiler-based approaches can exhibit inconsistent performance across diverse hardware platforms. For instance, Triton Distributed, originally developed for H800 GPUs, sometimes performs below the non-overlapped baseline on H100s. Hand-tuned kernels also show reduced efficiency on certain problem shapes.

Under sufficiently large reduction axes, the non-overlapped portion of communication time in PK falls below 1%. The communication component of our kernels (excluding GEMM) is implemented in fewer than 50 lines of device code, using the primitives introduced in Sec. 3.2. PK employs intra-SM overlap for GEMM+RS, and inter-SM overlap for AG+GEMM and GEMM+AR.

4.2 Sequence Parallelism

Modern AI workloads increasingly involve inputs with long sequence lengths, requiring a single sequence to be distributed across multiple devices. While sharding along the sequence dimension has minimal impact on MLP or MoE layers, attention layers require each token to attend to all others within the same sequence. This necessitates sequence-parallel approaches such as Ring Attention (Liu et al., 2024) and DeepSpeed-Ulysses (Jacobs et al., 2024). In our evaluation, we compare against the state-of-the-art implementations: xDiT (Fang et al., 2024) for Ring Attention and YunChang (Fang & Zhao, 2024) for DeepSpeed-Ulysses.

Ring Attention. In Ring Attention, key–value (KV) tensors are partitioned across devices, with each GPU computing blockwise attention on its local shard while concurrently transmitting it to a peer. The baseline xDiT implementation overlaps computation and KV exchange coarsely by launching NCCL P2P sends and FlashAttention-3 kernels on separate CUDA streams. In contrast, PK can fuse these into a single kernel with explicit inter-SM overlap, precisely allocating SMs between computation and communication, deciding how they synchronize, and auto-tuning this partitioning for optimal performance. As shown in Figure 10, this yields a 1.07×–4.08× speedup over the baseline—evaluated at total sequence lengths (shown on the X-axis)[4] evenly partitioned across 8 devices—and reduces the non-overlapped communication fraction down to 9%.

DeepSpeed-Ulysses. In DeepSpeed-Ulysses, an all-to-all exchange occurs before and after self-attention. Everything except self-attention is sequence-sharded, while self-attention remains head-sharded. The main bottleneck is the fine-grained all-to-all; as NCCL does not natively support this along the inner dimension, the baseline relies on tensor reshaping before and after communication. Using PK, we implement a fine-grained all-to-all kernel that removes this overhead. As shown in Figure 11, this yields a 1.01×–1.39× speedup, evaluated at total sequence lengths (shown on the X-axis) evenly split across 8 devices. The complete kernel remains under 50 lines of device code.

4.3 Expert Parallelism

To scale architectures with MoE layers (Shazeer et al., 2017), multiple experts are distributed evenly across devices, a strategy known as expert parallelism. However, this approach requires costly scattering and gathering of tokens before and after the expert MLP layers. Several approaches mitigate this by overlapping token communication with GEMM computation (Zhao et al., 2025; Zhang et al., 2025; Aimuyo et al., 2025). We compare against COMET (Zhang et al., 2025), the state-of-the-art fine-grained overlapping strategy for expert parallelism. For demonstration, we evaluate the first half of the MoE layer: overlapping token dispatch with the first expert MLP. PK employs an inter-SM scheme to enable this. As shown in Figure 12, where the total set of input tokens (shown on the X-axis) is initially partitioned

[4] Sequence lengths are intentionally set as multiples of 768 because this is required by the original TK attention forward kernel.

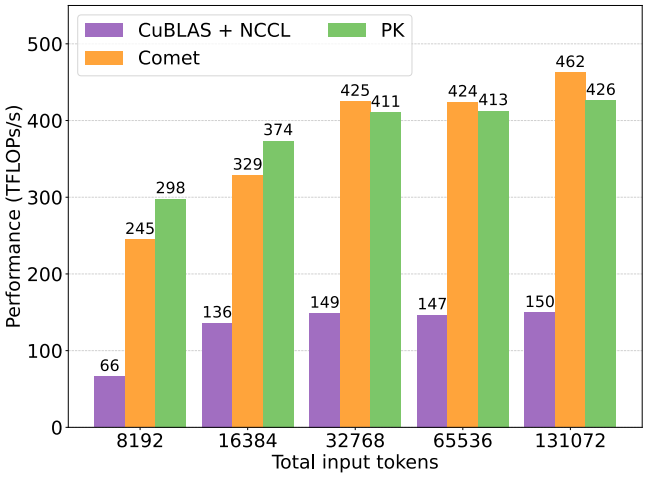

Figure 12. Expert-parallel token dispatch + GEMM performance (TopK $= 8$, $N_{\text{experts}} = 256$, $H = 7168$, $H_{\text{expert}} = 2048$).

evenly across devices, PK matches or surpasses the hand-tuned baseline, achieving 0.92–$1.22\times$ the performance of Comet, with fewer than 40 lines of device code added to a grouped GEMM kernel.

5 CONCLUSION

This work presents PARALLELKITTENS, a minimal and systematic framework for building high-performance multi-GPU kernels. By formalizing the design space through three key principles—transfer mechanisms, scheduling strategies, and design overheads—we demonstrate that a small set of primitives can match or surpass the performance of hand-optimized kernels while greatly simplifying implementation. As this work focuses on intra-node execution, extending these abstractions to inter-node communication remains an important direction for future work. At the same time, intra-node systems are rapidly scaling, as shown by Nvidia's NVL72 and upcoming NVL144, NVL576 architectures, which makes the study of efficient intra-node kernel design increasingly critical for distributed AI workloads.

Our framework and kernels are open sourced at: https://github.com/HazyResearch/ThunderKittens.

ACKNOWLEDGEMENTS

We are grateful to Cursor and Together AI for making this work possible. We thank Dylan Lim for his assistance with the initial implementation of PGL operations. We thank Yasa Baig, Kelly Buchanan, Francois Chaubard, Mayee Chen, Catherine Deng, Andy Dimnaku, Owen Dugan, Daniel Y. Fu, Roberto Garcia, Ronny Junkins, Ishane Khare, Hermann Kumbong, Jerry Liu, Avanika Narayan, Jon Saad-Falcon, and Alex Waitz for helpful feedback and discussions during this work. We gratefully acknowledge the support of NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF2247015 (Hardware-Aware), CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); US DEVCOM ARL under Nos. W911NF-23-2-0184 (Long-context) and W911NF-21-2-0251 (Interactive Human-AI Teaming); ONR under Nos. N000142312633 (Deep Signal Processing); Stanford HAI under No. 247183; NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, Google Cloud, Salesforce, Total, the HAI-GCP Cloud Credits for Research program, the Stanford Data Science Initiative (SDSI), and members of the Stanford DAWN project: Meta, Google, and VMware. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of NIH, ONR, or the U.S. Government.

REFERENCES

Aimuyo, O. J., Oh, B., and Singh, R. FlashDMoE: Fast Distributed MoE in a Single Kernel. *arXiv preprint arXiv:2506.04667*, June 2025.

AMD. Iris: First-class multi-gpu programming experience in triton. https://github.com/ROCm/iris, 2025.

Bauer, M., Cook, H., and Khailany, B. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2011. doi: 10.1145/2063384.2063400.

Chan, A., Shalaby, A., Wettig, A., Sanger, A., Zhai, A., Ajay, A., Nair, A., Snell, C., Lu, C., Shen, C., Jia, E., Cassano, F., Liu, H., Chen, H., Wildermuth, H., Jackson, J., Li, J., Katz, J., Yao, J., Hejna, J., Warner, J., Vering, J., Frans, K., Danilek, L., Wright, L., Cen, L., Melas-Kyriazi, L., Truell, M., de Jong, M., Jain, N., Schmidt, N., Wang, N., Muennighoff, N., Rybkin, O., Loh, P., Kravtsov, P., Yadav, R., Shah, S., Kottler, S., Rush, A. M., Zhang, S., Jain, S., Sankar, S., Heule, S., Sul, S. H., Asif, S., Rong, V., Zhu, W., Lin, W., Wu, Y., Volkov, Y., Zemlyanskiy, Y., Holbrook, Z., and Zhang, Z. Composer 2 Technical Report. *arXiv preprint arXiv:2603.24477*, March 2026.

Chang, L., Bao, W., Hou, Q., Jiang, C., Zheng, N., Zhang, X., Song, Z., Jiang, Z., Lin, H., and Liu, X. FLUX: Fast Software-based Communication Overlap On GPUs Through Kernel Fusion. *arXiv preprint arXiv:2406.06858v1*, June 2024.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J., Liang, J., Guo, J., Ni, J., Li, J., Wang, J., Chen, J., Chen, J., Yuan, J., Qiu, J., Li, J., Song, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Wang, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Wang, Q., Zhu, Q., Chen, Q., Du, Q., Chen, R., Jin, R., Ge, R., Zhang, R., Pan, R., Wang, R., Xu, R., Zhang, R., Chen, R., Li, S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Pan, S., Wang, T., Yun, T., Pei, T., Sun, T., Xiao, W., Zeng, W., Zhao, W., An, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Li, X., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Zhang, X., Chen, X., Nie, X., Sun, X., Wang, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yu, X., Song, X., Shan, X., Zhou, X., Yang, X., Li, X., Su, X., Lin, X., Li, Y., Wang, Y., Wei, Y., Zhu, Y., Zhang, Y., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Yu, Y., Zheng, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Tang, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Wu, Y., Ou, Y., Zhu, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Zha, Y., Xiong, Y., Ma, Y., Yan, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Wu, Z., Ren, Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Huang, Z., Zhang, Z., Xie, Z., Zhang, Z., Hao, Z., Gou, Z., Ma, Z., Yan, Z., Shao, Z., Xu, Z., Wu, Z., Zhang, Z., Li, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Gao, Z., and Pan, Z. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437*, December 2024.

Fang, J. and Zhao, S. A unified sequence parallelism approach for long context generative ai. *arXiv preprint arXiv:2405.07719*, 2024.

Fang, J., Pan, J., Sun, X., Li, A., and Wang, J. xdit: an inference engine for diffusion transformers (dits) with massive parallelism. *arXiv preprint arXiv:2411.01738*, 2024.

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

He, H., Wright, L., Wehrstedt, L., Liu, T., and Liang, W. Introducing async tensor parallelism in pytorch. https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor-parallelism-in-pytorch/209487/1, September 2024.

Jacobs, S. A., Tanaka, M., Zhang, C., Zhang, M., Song, S. L., Rajbhandari, S., and He, Y. System optimizations for enabling training of extreme long sequence transformer models. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing (PODC '24)*, pp. 121–130, New York, NY, USA, 2024. Association for Computing Machinery. doi: 10.1145/3662158.3662806. URL https://doi.org/10.1145/3662158.3662806.

Jia, Z., Zaharia, M., and Aiken, A. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of the 2nd SysML Conference*, 2019.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., Purandare, S., Nadathur, G., and Idreos, S. Torchtitan: One-stop pytorch native solution for production ready LLM pretraining. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=SFN6Wm7YBI.

Liu, H., Zaharia, M., and Abbeel, P. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=WsRHpHH4s0.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

Nvidia. Nvidia NVLink and NVLink Switch. https://www.nvidia.com/en-us/data-center/nvlink/.

Nvidia. Nvshmem. https://developer.nvidia.com/nvshmem.

Nvidia. Nvidia ampere architecture in-depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/, May 2020.

Nvidia. Nvidia CuTe. https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/00_quickstart.md, 2024.

Nvidia. Advancing performance with nvidia sharp in-network computing. https://developer.nvidia.com/blog/advancing-performance-with-nvidia-sharp-in-network-computing/, 2024.

Nvidia. Nvidia blackwell architecture technical brief. https://resources.nvidia.com/en-us-blackwell-architecture, 2025.

Nvidia. Company Overview. https://s201.q4cdn.com/141608511/files/doc_presentations/2025/08/Q226-NVDA-Company-Overview-Final.pdf, August 2025.

Nvidia. Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2025a.

Nvidia. Nvidia nvlink and nvlink switch. https://www.nvidia.com/en-us/data-center/nvlink/, 2025b.

Potluri, S., Goswami, A., Rossetti, D., Newburn, C. J., Venkata, M. G., and Imam, N. Gpu-centric communication on nvidia gpu clusters with infiniband: A case study with openshmem. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pp. 253–262. IEEE, 2017. doi: 10.1109/HiPC.2017.00037.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017. doi: 1701.06538. URL https://arxiv.org/abs/1701.06538.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, September 2019.

Si, M., Balaji, P., Chen, Y., Chu, C.-H., Gangidi, A., Hasan, S., Iyengar, S., Johnson, D., Liu, B., Ren, R., Shetty, A. J., Steinbrecher, G., Wang, Y., Wu, B., Xie, X., Yang, J., Yang, M., Yu, K., Yu, M., Zhao, C., Bland, W., Boyda, D., Gumudavelli, S., Kannan, P., Lumezanu, C., Miao, R., Qu, Z., Ramesh, V., Samoylov, M., Seidel, J., Sundaresan, S., Tian, F., Tan, Q., Zhang, S., Zhao, Y., Zheng, S., Zhu, A., and Zeng, H. Collective communication for 100k+ gpus. *arXiv preprint arXiv:2510.20171*, October 2025.

Spector, B. F., Arora, S., Singhal, A., Parthasarathy, A., Fu, D. Y., and Ré, C. Thunderkittens: Simple, fast, and adorable kernels. In *The Thirteenth International Conference on Learning Representations*, April 2025. URL https://openreview.net/forum?id=0fJfVOSUra.

Thakkar, V., Ramani, P., Cecka, C., Shivam, A., Lu, H., Yan, E., Kosaian, J., Hoemmen, M., Wu, H., Kerr, A., Nicely, M., Merrill, D., Blasig, D., Qiao, F., Majcher, P., Springer, P., Hohnerbach, M., Wang, J., and Gupta, M. Cutlass: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass.

Tillet, P., Kung, H. T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

Tsu, W. Introducing Nvidia HGX H100: An Accelerated Server Platform for AI and High-Performance Computing. https://developer.nvidia.com/blog/introducing-nvidia-hgx-h100-an-accelerated-server-platform-for-ai-and-high-performance-computing/, April 2022.

Zhang, S., Zheng, N., Lin, H., Jiang, Z., Bao, W., Jiang, C., Hou, Q., Cui, W., Zheng, S., Chang, L.-W., Chen, Q., and Liu, X. Comet: Fine-grained Computation-communication Overlapping for Mixture-of-Experts. *Proceedings of the 8th MLSys Conference*, March 2025.

Zhao, C., Zhou, S., Zhang, L., Deng, C., Xu, Z., Liu, Y., Yu, K., Li, J., and Zhao, L. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.

Zheng, L., Li, Z., Zhang, H. Z., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y. X., Zhuo, D., and Xing, E. P. Alpa: Automating inter-and Intra-Operator parallelism for distributed deep learning. *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

Zheng, S., Bao, W., Hou, Q., Zheng, X., Fang, J., Huang, C., Li, T., Duanmu, H., Chen, R., Xu, R., Guo, Y., Zheng, N., Jiang, Z., Di, X., Wang, D., Ye, J., Lin, H., Chang, L.-W., Lu, L., Liang, Y., Zhai, J., and Liu, X. Triton-distributed: Programming Overlapping Kernels on Distributed AI Systems with the Triton Compiler. *arXiv preprint arXiv:2504.19442*, June 2025a.

Zheng, S., Fang, J., Zheng, X., Hou, Q., Bao, W., Zheng, N., Jiang, Z., Wang, D., Ye, J., Lin, H., Chang, L.-W., and Liu, X. TileLink: Generating Efficient Compute-Communication Overlapping Kernels using Tile-Centric Primitives. *arXiv preprint arXiv:2503.20313*, March 2025b.

Zhu, K., Gao, Y., Zhao, Y., Zhao, L., Zuo, G., Gu, Y., Xie, D., Tang, T., Xu, Q., Ye, Z., Kamahori, K., Lin, C.-Y., Wang, Z., Wang, S., Krishnamurthy, A., and Kasikci, B. NanoFlow: Towards Optimal Large Language Model

Serving Throughput. *arXiv preprint arXiv:2408.12757*, May 2025.