USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Syncopate: Efficient Multi-GPU AI Kernels via Automatic Chunk-Centric Compute-Communication Overlap

Xinwei Qiang, Yue Guan, and Zhengding Hu, *University of California, San Diego;* Keren Zhou, *George Mason University and OpenAI;* Yufei Ding, *University of California, San Diego, and Meta;* Adnan Aziz, *Meta*

https://www.usenix.org/conference/osdi26/presentation/qiang

This paper is included in the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation.

July 13–15, 2026 • Seattle, WA, USA

ISBN 978-1-939133-55-7

Open access to the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Syncopate: Efficient Multi-GPU AI Kernels via Automatic Chunk-Centric Compute-Communication Overlap

Xinwei Qiang[1,*], Yue Guan[1,*], Zhengding Hu[1], Keren Zhou[3,4], Yufei Ding[1,2], Adnan Aziz[2]
[1]*University of California, San Diego,* [2]*Meta,* [3]*George Mason University,* [4]*OpenAI*
[1]*{x1qiang, yueguan, zhh068, yufeiding}@ucsd.edu*
[2]*{adnanaziz}@meta.com*
[3]*{kzhou6}@gmu.edu*

Abstract

Communication has become a first-order bottleneck in large-scale GPU workloads, and existing distributed compilers address it mainly by overlapping whole compute and communication kernels at the stream level. This coarse granularity incurs extra kernel launches, forces device-wide synchronizations at kernel boundaries, and leaves substantial slack when the slowest tile or kernel stretches the communication tail. We present Syncopate, a compiler and runtime that enable automatic fine-grained overlap around a single fused compute kernel. Syncopate introduces a communication chunk abstraction that decouples communication granularity from kernel structure and backend mechanisms, allowing chunk-level plans to be ported from existing distributed compilers, written directly by users, or instantiated from reusable templates. Given a local Triton kernel and a chunk schedule, Syncopate performs transformations to align computation with chunk availability. Implemented as a source-to-source compiler on Triton, Syncopate delivers an average end-to-end speedup of 1.3× and up to 4.7× on multi-GPU workloads. Our code is open-sourced at https://github.com/tie-pilot-qxw/syncopate.

1 Introduction

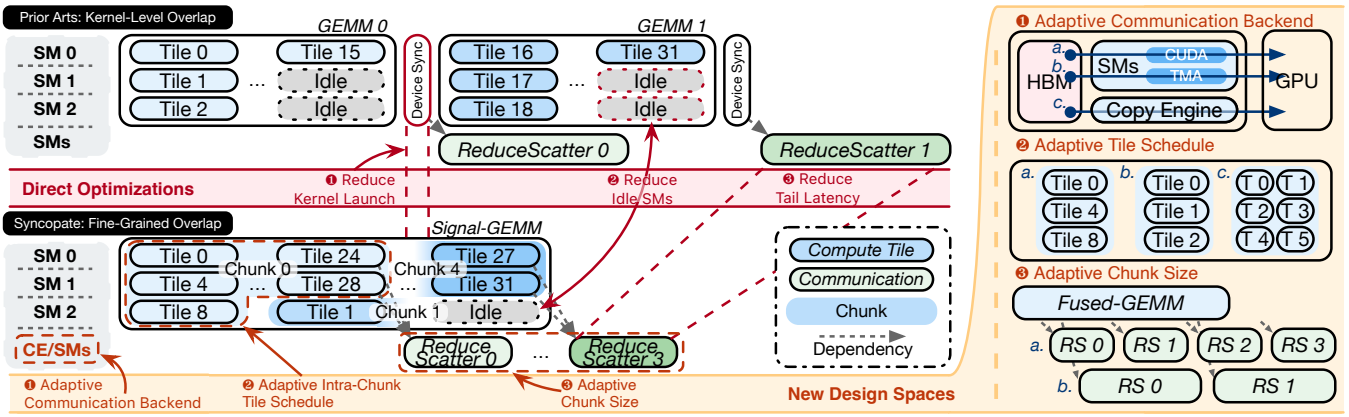
Communication has become a first-order bottleneck for training and serving large neural networks on multi-GPU systems. Even with high-bandwidth interconnects such as NVLink [23] and NVSwitch, collective operations like AllGather, ReduceScatter, and All-to-All frequently dominate end-to-end latency for tensor-parallel feed-forward layers and attention layers. To hide this cost, recent systems and distributed compilers aggressively search for schedules that overlap computation and communication at the kernel level. Given a computation graph and a device mesh, these compilers select parallelization strategies, insert communication kernels, and assign compute and communication kernels to streams so that multiple kernels can run concurrently. This kernel-level overlap [4, 10, 31, 36, 38, 47] has become the default mechanism for improving utilization in distributed settings.

However, kernel-level overlap is fundamentally insufficient for fully hiding communication latency. As illustrated in Fig. 1, this kernel-level scheduling forces a device-wide synchronization at every kernel boundary and incurs extra launch and sync overhead for each communication phase (❶). Moreover, by splitting computation into multiple shorter kernels, the work within each launch is further partitioned into waves of compute tiles on each SM, increasing the fraction of time SMs sit idle; even when part of the data needed by later kernels is already available, tiles in the current wave must wait for the slowest tile to finish (❷). Finally, the coarse granularity of kernel-level overlap leaves a long segment of communication at the end of the timeline that receives little or no overlap (❸).

This motivates us to overlap computation and communication at a finer intra-kernel granularity. Such fine-grained overlap opens up new design space for improving end-to-end efficiency. As shown by the yellow region in Fig. 1, Syncopate launches communication directly from within the fused kernel, rather than delegating to external communication libraries, giving the compiler explicit control over which hardware backend to use for each transfer (copy engine, tensor memory accelerator, or load/store on CUDA cores) as shown with ❶. This also allows us to explore much smaller communication granularities without incurring additional kernel-launch overhead, and to tune the chunk size that best balances link throughput against synchronization cost (❷). Finally, because tiles and communication are orchestrated inside a single kernel, we can reshape the intra-kernel tile schedule to track communication progress while still preserving locality in the register, shared-memory, and cache hierarchy (❸).

We present Syncopate, a compiler that turns the vision of fine-grained overlap for distributed kernel generation into a practical and general system. At the heart of Syncopate is the notion of a communication chunk, which represents a logical block of data associated with a particular communication operation and the tiles that produce or consume it. This abstrac-

*Xinwei and Yue contributed equally.



(a) Comparison of compute–communication overlap at different levels

(b) New design space for fine–grained overlap

Figure 1: Motivating example of Syncopate. Red numbers show the direct improvements gained by fine-grained overlap over kernel-level overlap, while orange numbers show the additional improvements from the new design space enabled by Syncopate.

tion is motivated by a key observation: fine-grained overlap requires a communication granularity that flexibly matches how tiles generate data and how communication backends consume it, rather than assuming that tile-level or full-kernel granularity is always appropriate. By making chunks explicit, Syncopate can represent a wide range of overlap patterns and enable the compiler to reason about when each chunk becomes available, which tiles produce or use it, and how these dependencies interact with fused multi-stage tensor programs and potentially irregular collectives. This abstraction exposes a small set of principled knobs, including chunk size, backend choice, and tile order, that define the design space later explored by our autotuner.

Building on this abstraction, Syncopate implements a source-to-source compiler and runtime that transform standard Triton [34] kernels and a high-level chunk-based communication plan into fused distributed kernels capable of fine-grained compute–communication overlap. The compiler restructures the kernel's tile execution to follow communication progress and selects appropriate backends to realize each chunk transfer, while a lightweight runtime executes these transfers and integrates seamlessly with PyTorch distributed [2] so that Syncopate kernels can replace standard operators with only minimal changes to user code. Syncopate further performs inter- and intra-chunk autotuning, adjusting chunk sizes, backend choices, SM allocations, and tile schedules, to consistently achieve high performance across diverse operators. This implementation strategy makes the abstract chunk-based model concrete, enabling rapid prototyping of new overlap policies while remaining compatible with real distributed workloads and production communication stacks.

In summary, this work makes the following contributions:

- We introduce a chunk-based abstraction that decouples high-level overlap intent from low-level implementation, enabling fine-grained compute-communication overlap inside distributed kernels.
- We design and implement a compiler pipeline and runtime that takes annotated local kernels and chunk-level communication plans, then generates efficient fused distributed kernels with inter- and intra-chunk autotuning.
- We evaluate Syncopate on a diverse set of distributed workloads and observe an average speedup of 1.3× on common operators, with improvements reaching up to 4.7× in the best cases.

2 Background and Related Works

2.1 Distributed Compilers

As model sizes grew, tensor compilers [5, 6, 9, 29, 40, 43] incorporated basic multi-GPU support [1, 16, 27, 30, 37, 41], typically by composing single-GPU kernels with predefined collective primitives. Systems like Alpa [44], Mercury [13], and other recent distributed compilers extend this idea by searching over communication patterns and schedules at the level of whole kernels: given a parallelization strategy, they first construct a communication plan containing AllGather, ReduceScatter, or All-to-All operators, and then explore different mappings of compute and communication kernels to streams in order to maximize kernel-level overlap (Table 1). This design has made distributed training far more accessible, but it also bakes in a rigid abstraction boundary: communication is planned as a sequence of full-kernel collectives whose launch and completion times are the basic units of scheduling. As a result, all these distributed compilers focus their search on the communication plan at the kernel level and are fundamentally blind to finer-grained opportunities inside kernels, such as overlapping per-shard or per-tile communication with computation, reusing remote data across tiles, or exploiting topology-aware pipelining within a fused kernel. Once a kernel is chosen and its associated collective is placed, the compiler can only treat it as an atomic black box, leaving

Table 1: Comparison of projects on distributed operations.

Projects	Granularity	Compute	Communication	Schedule	Performance
Automatic Approaches					
Alpa [44]	Kernel	Auto	Auto	Template	✔
Mercury [13]	Kernel	Auto	Auto	Auto	✔✔
Manual Implementations					
Flux [3]	Tile	Manual	Manual	Manual	✔✔
AsyncTP [38]	Tile	Manual	Manual	Manual	✔✔
FlashOverlap [14]	Chunk	Manual	Manual	Manual	✔✔✔
Domain Specific Languages					
ThunderKittens [32,33]	Tile	Manual	Manual	Manual	✔✔✔
TritonDistributed [45,46]	Chunk	Manual	Manual	Manual	✔✔✔
Syncopate	Chunk	Auto	Auto	Template	✔✔✔

Table 2: Comparison of various GPU communication mechanisms.

	Hardware	Programming	Collective	Bandwidth
Copy Engine	Copy Engine	Host Launch	✗	✔✔✔
TMA	SM	Async. Instruction	✗	✔✔
Load/Store	SM	Sync. Instruction	✔	✔

significant intra-kernel overlap potential untapped even under an "optimal" kernel-level schedule.

2.2 Manual Kernel Designs

A complementary line of work pushes beyond kernel-level overlap and instead hand-crafts fine-grained pipelines that interleave communication and computation at the level of tiles, tokens, or shards. Flux [3] fuses GEMM with collectives at tile granularity and over-decomposes work to maximize overlap for both training and inference, while Comet [42] targets MoE with a shared-tensor abstraction and NVSHMEM-backed buffers to overlap token-wise communication with tile-wise compute at production scale. FlashOverlap [14] uses lightweight readiness signaling and layout reordering to trigger overlap with standard NCCL collectives without modifying existing compute kernels, and systems like TritonDistributed [45,46] introduce tile-centric or OpenSHMEM-style primitives to Triton [34] that make it easier to author overlapped kernels (e.g., fused AllGather/GEMM or GEMM/ReduceScatter) in a domain-specific language.

Despite these advances, all of these systems fundamentally rely on manual, operator-specific engineering: experts must design fused kernels, choose tiling and buffering schemes, and reason about synchronization and communication protocols for each new model architecture or hardware platform. The resulting implementations are highly optimized but difficult to generalize or retarget, and they do not provide a general compiler abstraction for expressing and reusing fine-grained overlap patterns across operators. Emerging fine-grained DSLs and primitives greatly lower the barrier to writing overlapped kernels, but they still place the burden of discovering effective overlap strategies, encoding dependency structure, and validating correctness squarely on the programmer. In practice, users still need to write or modify DSL-level communication and computation code, explicitly manage signal/wait, buffer ownership, and memory ordering, and revalidate these low-level details for each operator and communication pattern.

2.3 Communication Backends

Modern GPU systems [19] expose several mechanisms for moving tensors across devices, each with distinct performance and programmability trade-offs (Tbl. 2). Copy Engines saturate NVLink [23] at 400 GB/s per direction on H100 and run independently of the SMs. Therefore, they do not consume compute resources and are ideal when communication can be decoupled from computation. However, they are typically driven by host APIs and can only transfer contiguous data, so high-dimensional strided tensors must be decomposed into many smaller transfers, each requiring a separate launch costing around 2-3μs, which can significantly reduce the effective bandwidth as each transfer time is also very short.

Tensor Memory Accelerator (TMA) [19] paths achieve high bandwidth using dedicated asynchronous tensor-copy hardware, and can achieve a throughput of 300+ GB/s with only about 16 SMs issuing TMA instructions. This makes TMA attractive for overlapping structured tensor movement with computation within a node. At the same time, TMA must be launched by SM threads and does not currently support inter-node communication or in-network (switch-based) collective reduction, which limits its applicability to intra-node, point-to-point patterns.

Finally, plain load/store-based communication, often combined with registers and shared memory, attains slightly lower peak bandwidth than copy engines or TMA [33] but is significantly more flexible. These mechanisms integrate naturally with switch-based collective reduction (NVSHARP [20]), enabling fine-grained per-shard communication and in-network reductions. The downside is that they consume SM resources and are synchronous from the issuing warp's perspective, so they are harder to pipeline and require careful scheduling to hide communication latency.

3 Motivation

To fully exploit modern GPUs, distributed training systems must overlap computation and communication. Prior work mainly relies on coarse, kernel-level partitioning of computation kernels, which leaves substantial performance on the table. In this section, we revisit the motivating example in Figure 1 using detailed microbenchmarks (Figure 2), and show three key insights:

> ★***Insight 1: Limitations of Kernel-Level Overlap.*** *Kernel-level overlap is fundamentally limited by SM under-utilization and kernel-launch overheads.*

Figure 2(a) reports SM utilization for different GEMM

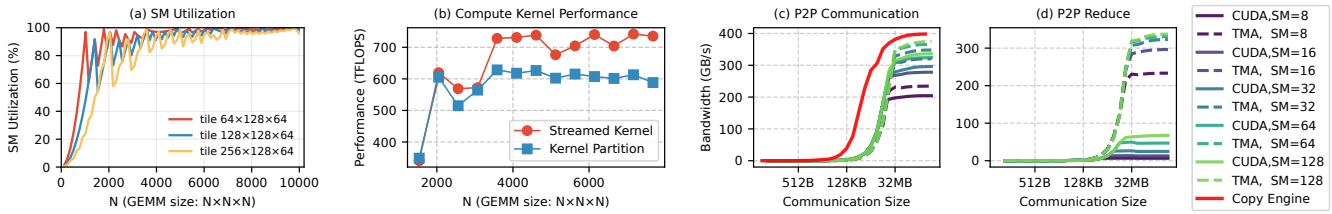


Figure 2: Motivation experiment results. (a) SM utilization under different GEMM sizes and tile sizes. (b) Performance comparison between a streamed GEMM kernel and a kernel-partitioned baseline. (c,d) Bandwidth of different communication backends under varying message sizes.

sizes under several commonly used tile-size configurations. Large GEMMs provide enough tile waves to saturate the SMs across all configurations. As the GEMM size decreases, fewer tile waves are generated, and the partially filled last wave dominates a larger portion of execution, causing SM utilization to drop due to wave quantization. Partitioning a GEMM into many sub-kernels forces each launch to operate on a smaller shape, pushing execution into exactly this low-utilization regime. This effect directly limits the benefit of kernel-level partition-based overlap (corresponding to ❷ in Figure 1), since overlapping many small kernels simply wastes SM capacity.

Figure 2(b) compares the end-to-end performance of (i) a baseline that partitions GEMM into multiple small kernels for overlap and (ii) a streamed GEMM kernel that internally pipelines tiles. Although both variants execute the same arithmetic operations, the kernel-partitioned baseline incurs substantial performance loss due to extra kernel launches (❶) and the SM under-utilization observed in Figure 2(a). In contrast, the streamed kernel maintains high utilization by exposing fine-grained compute tiles within a single launch, enabling overlap without fragmenting the workload. These results show that simply launching more kernels is not an effective path toward overlap; we need mechanisms that expose intra-kernel concurrency while preserving efficient GPU execution.

> ★***Insight 2: Granularity and Backend Effects.*** *Communication efficiency varies sharply with transfer granularity and backend selection.*

Communication efficiency varies sharply with transfer granularity and backend behavior. Figures 2(c) and 2(d) show the achieved bandwidth of different communication backends as we vary the transfer size and the number of SMs. Each backend shows distinct scaling behavior: some reach peak bandwidth at moderate transfer sizes, while others require larger transfers or more SMs to reach their full potential. Moreover, different backends support different communication patterns, such as point-to-point transfers versus reductions.

Taken together, these granularity and backend effects imply that the optimal configuration depends jointly on (i) the compute tile size, which determines the cadence at which results become available, (ii) the communication transfer size, which trades off latency and bandwidth, and (iii) the choice of communication backend. Coarse-grained kernel-level overlap cannot flexibly coordinate these parameters, since compute and communication are implemented as separate kernels with rigid interfaces. Intra-kernel overlap, by coordinating computation and communication at the same time, can align tile production with backend-specific sweet spots and select transfer granularities that sustain high utilization across SMs and copy engines.

> ★***Insight 3: A Unified Unit for Intra-Kernel Overlap.*** *Effective intra-kernel overlap therefore requires a communication unit with tunable granularity and a stable interface across communication backends.*

The combined granularity and backend effects indicate that effective overlap requires a communication unit whose size can match both the rate at which tiles produce data and the efficiency points of different communication backends. At the same time, this unit must provide a stable boundary between the high-level communication schedule and its backend-specific realization, so that schedules need not be rewritten for each backend. We therefore introduce a chunk abstraction that offers both tunable intra-kernel granularity and a unified interface for mapping communication onto diverse backends.

4 Overview

Syncopate is a compiler and runtime framework that turns locally written Triton kernels into distributed, fine-grained overlapped kernels. Rather than asking programmers to manually fuse communication and computation, Syncopate takes an existing local kernel and a high-level distribution specification as input, and automatically synthesizes an intra-kernel schedule that interleaves tile-wise communication with computation according to the available communication backends.

Input and User Interface. On the compute side, Syncopate consumes unmodified local Triton kernels annotated with

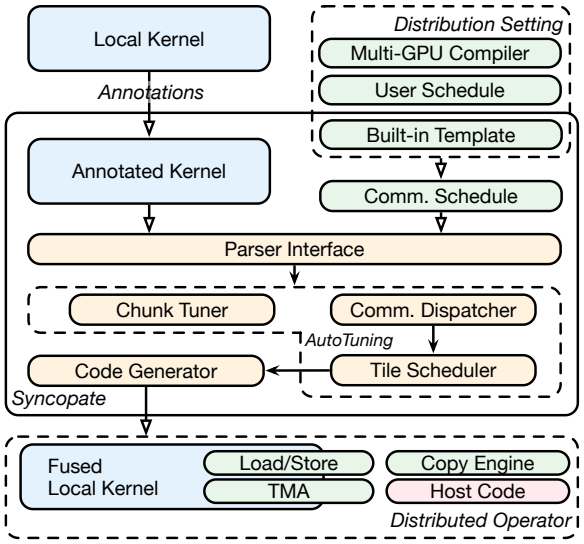


Figure 3: System overview of Syncopate.

Listing 1: Annotated Local Triton Kernel API.

```
@triton.jit
def kernel_gemm(a_ptr, b_ptr, ...):
    start_pid = tl.program_id(axis=0)
    # @sy.axis_count M block=BLOCK_SIZE_M
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    # @sy.tile_id persistent
    tile_id = start_pid - NUM_SMS
    ...
    a_desc = tl.make_tensor_descriptor(a_ptr, ...)
    ...
    for _ in range(0, k_tiles * tiles_per_SM):
        tile_id += NUM_SMS
        # @sy.dispatch begin
        # @sy.pid_map M=pid_m N=pid_n
        pid_m, pid_n = get_pid_mn(tile_id,
                                  num_pid_m, ...)
        # @sy.dispatch end
        offs_am = pid_m * BLOCK_SIZE_M
        offs_bn = pid_n * BLOCK_SIZE_N
        offs_k = ki * BLOCK_SIZE_K
        a = a_desc.load([offs_am, offs_k])
        b = b_desc.load([offs_bn, offs_k])
        accumulator = tl.dot(a, b.T, accumulator)
```

Listing 2: Communication Schedule Example.

```
def all_gather_1d_swizzle(shape,dtype,axis,rank,...):
    plan = DevicePlan(dev=rank)
    plan.tensors_involved[buf] = (torch.Size(shape))
    local = shard(rank)
    plan.local_regions.setdefault(buf,[]).append(local)
    for i in range(mesh):
        peer = (i + rank) % mesh  # 1D swizzle
        if peer == rank: continue
        r = shard(peer)
        plan.add_op(Transfer(
                        op=TransferOp.PULL,
                        dst_buf=buf, dst_region=r,
                        src_buf=buf, src_region=r,
                        peer=peer, shard_idx=peer,
                    ...))
    return plan
```

lightweight scheduling metadata (Listing 1). Programmers write kernels as if they were running on a single device, using standard Triton primitives for indexing, tiling, and tensor descriptors. Lightweight Syncopate annotations (e.g., axis counts, tile identifiers, and dispatch regions) are required to identify the logical tiles and iteration structure but do not change the kernel's semantics. On the distribution side, Syncopate uses a communication plan that encodes the desired global data movement pattern and device topology (Listing 2). This plan is expressed using a small API defined by the users or imported directly from higher-level compilers searching parallel schedule.

To end users, Syncopate exposes a small set of APIs for (1) registering local kernels with their annotations, (2) constructing or selecting predefined communication plans (such as 1D/2D AllGather or ReduceScatter swizzles), and (3) compiling these into executable distributed kernels. High-level frameworks can wrap these APIs so that model authors only specify tensor partitioning and desired collectives, while Syncopate automatically generates the corresponding overlapped kernels. This separation of concerns allows experts to encode distribution and communication strategies once, while ordinary users invoke the resulting distributed operators through familiar, library-style calls.

Syncopate Architecture and Output. Given a local kernel and its communication plan, Syncopate lowers them into a unified dependence representation over tiles, shards, and communication operations. The compiler then searches for a fine-grained schedule that maps tiles to devices, assigns communication operations to appropriate backends (e.g., copy engine, TMA, or load/store), and inserts the necessary synchronization to respect both compute and communication dependencies. The output preserves the original numerical semantics while tightly pipelining asynchronous communication and computation around a single fused compute kernel. From the user's perspective, the generated operator can be invoked with the same signature as the original local kernel, plus standard distributed-runtime arguments (e.g., rank, world size, mesh), and integrated into existing training or inference code without further changes.

5 Syncopate: Fine-Grained Overlap Compiler

Syncopate automatically transforms locally written kernels into fine-grained overlapped distributed kernels by aligning the execution of local computation tiles with a global communication schedule. At a high level, we introduce a *chunk-centric* compilation pipeline that treats communication and computation symmetrically: communication is described as transfers of logical chunks, while computation is expressed as tiles that consume and produce these chunks. The compiler then derives dependencies between chunks and tiles,

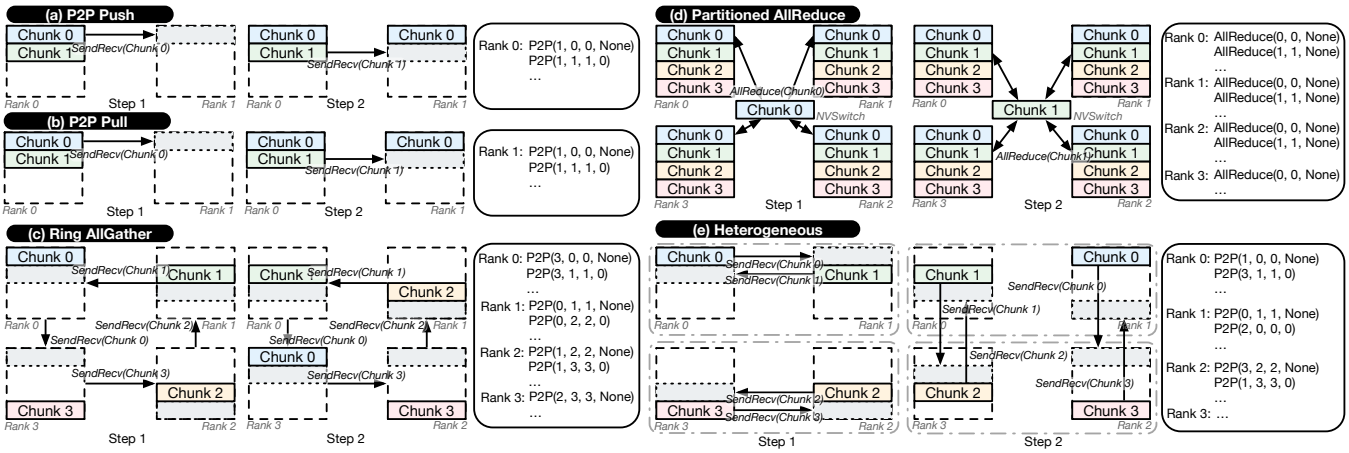


Figure 4: Communication schedule abstraction. (a) and (b) illustrate the same point-to-point exchange expressed as push and pull variants, respectively. (c) shows a ring-based AllGather pattern. (d) represents a partition-based AllReduce schedule. (e) depicts a heterogeneous swizzled AllGather pattern that pipelines communication across multiple hierarchy levels.

rewrites the tile scheduler to follow the communication order, inserts the necessary synchronization, and finally explores several implementation choices to generate high-performance overlapped code.

5.1 Communication Schedule Abstraction

We first define a communication-side abstraction that captures how data moves across devices independently of any particular local kernel implementation. This abstraction is built around the notion of a *chunk*, an intermediate layout between the global logical tensor and the local computation tiles. By operating at this intermediate granularity, the abstraction is expressive enough to describe a wide range of distributed schedules while remaining compatible with both partition-based and loop-based IRs in existing distributed compilers.

Definition. A *chunk* is a logical block of data that is communicated as a unit. Each chunk contains one or more tiles, where a tile is the basic unit of computation in the local kernel. Importantly, the chunk size in the communication schedule specifies *logical* transfers; the same logical chunk may later be implemented using different physical communication patterns or backends during lowering.

Conceptually, a chunk can be represented as: `chunk = Chunk(sizes=[...] , layout=..., tensor=...)`. Based on this abstraction, we define communication operators over chunks. We consider two primary classes of operators: point-to-point (P2P) transfers and collective communications.

- **P2P transfer** is represented as `P2P(src_rank, dst_rank, src_chunk, dst_chunk, dependency)`. This operator moves a chunk from a source rank to a destination rank, optionally guarded by a dependency on other chunks or operations. Note that for a pair of P2P operations on the source and destination ranks, we only include the operation on one side. If the P2P operation is defined on the source side, it represents a push operation; otherwise, it represents a pull operation. This will lead to different implementation choices during lowering.
- **Collective communication** is represented as `Collective(collective_type, src_chunk, dst_chunk, ranks, dependency)`. This operator applies a collective operation (e.g., AllGather, ReduceScatter) over a set of ranks on a given chunk with explicit dependency control. When explicitly defined as collective operations, the compiler can leverage the optimized collective implementations provided by the communication backends.

For both operator types, the `dependency` field encodes any ordering constraints that must be respected between communication operations. Specifically, it is represented as a `(rank, index)` tuple, indicating that the current operation cannot start until the specified operation on the given rank has completed. This allows us to express complex communication patterns, such as ring exchanges or multi-stage collectives, by chaining dependencies between chunks.

Upon this abstraction, a *communication schedule* is defined as a sequence of chunk-level communication operations with their associated dependencies on each rank as `schedule := [rank:Int, operations:List[CommOp]]:List`. Since there is no restriction on the operation for each rank, the communication schedule can express heterogeneous communication patterns where different ranks perform different operations on different chunks at different times.

Expressiveness. Despite its simplicity, the chunk abstraction is sufficiently expressive to capture a wide range of communication schedules covering all the overlap patterns used in practice (Fig. 4). To elaborate, (a) and (b) show the same P2P communication between two ranks expressed as push and pull operations, respectively, demonstrating the flexibility of

pull/push semantics. (c) illustrates a ring-based AllGather pattern, which is a common pattern for asynchronous distributed operators [18], where each rank sends and receives chunks in a pipelined manner, with dependencies ensuring the correct order of operations. (d) depicts a partition-based collective AllReduce pattern, where each rank contributes a chunk to the collective operation and performs the accumulation over the communication fabric. This pattern is often used in partition-based distributed compilers for kernel-level overlap. Finally, (e) shows a complex heterogeneous swizzled AllGather pattern advancing (c). By utilizing the port abstraction, each rank processes communication at different mesh hierarchy levels, enabling fine-grained pipelining and overlap across multiple dimensions. With this abstraction, different collective patterns, pipelined P2P exchanges, and hybrid schemes that combine intra-node and inter-node communication can all be written as sequences of chunk-level P2P and collective operators with explicit dependencies. Because chunks are defined in terms of logical tensor regions rather than concrete buffers, the same schedule can be reused across different kernels and tensor shapes, and later specialized by the compiler.

Lowering from Higher-Level Compiler IRs. Communication schedules in this abstraction can either be defined manually or automatically derived from existing distributed compiler IRs. In the manual case, users construct chunk objects and communication operators directly using our API as shown in Listing 2. In most cases, manual schedules are parameterized communication algorithms, not fully unrolled per-world-size plans: ring-based algorithms, for example, can be expressed with loops over ranks, chunks, or mesh levels, and Syncopate specializes them to the concrete world size, topology, and chunk configuration. Syncopate also provides predefined templates such as 1D/2D AllGather or ReduceScatter swizzles for the common communication patterns. The user can instantiate these templates with different chunk sizes, mesh topologies, communication axes, and pipeline stages to generate reusable communication schedules.

When integrating with a higher-level distributed compiler, Syncopate provides frontends for both partition-based and loop-based IRs. For partition-based IRs, we analyze the global data partitioning and the implied communication pattern between partitions to infer chunk sizes, participating ranks, and the corresponding P2P or collective operators. For loop-based IRs, we traverse loop nests, identify communication points, and group the communicated regions into chunks according to the chosen granularity. In both cases, the result is a uniform chunk-level schedule that decouples the high-level communication intent from any particular implementation (Listing 3). Specifically, the collective operators can be directly inserted ("direct") into our communication plan, or they can be further lowered to P2P communication operators using our templates ("template") or using some collective synthesis algorithms ("synth") such as TACOS [39].

Listing 3: Lowering from Higher-Level IR.

```
def emit_steps(steps, mesh, path="template"):
    comm = CommPlan()
    for step in steps:
        if step.is_p2p(): # push/pull/local_copy
            for rank in mesh:
                emit_p2p(comm.plans[rank], step, rank)
        else:  # collective
            for rank in mesh:
            # Path in ["direct", "synth", "template"]
                emit_collective(path, comm.plans[rank],
      step, rank)
    return comm

def lower_partition_ir(part_ir, axis_info, mesh, path="
     template"):
    steps = []
    for tensor in part_ir.tensors:
        layout = part_ir.placement[tensor]
        meta = axis_info[tensor]
        steps.extend(parse_partition_to_steps(tensor,
     layout, meta))
    return emit_steps(steps, mesh, path)

def lower_loop_ir(loop_ir, mesh, path="template"):
    steps = []
    for node in walk(loop_ir):
        steps.extend(parse_comm_intents(node))
    return emit_steps(steps, mesh, path)
```

5.2 Chunk-based Code Generation

Given a chunk-level communication schedule, the next step is to reorganize local computation so that tiles are executed in an order that aligns with the arrival and consumption of chunks. Intuitively, we want the kernel to compute exactly those tiles whose data has already been communicated, and to defer tiles whose input chunks are still in flight. This *compute chunk scheduling* bridges the gap between the communication abstraction and the tile-level structure of the original local kernel.

Compute Kernel Annotations. To make tile-level scheduling explicit, users provide lightweight annotations on the local computation kernel using Syncopate's API. These annotations do not change the numerical semantics of the kernel; instead, they expose its tiling structure and iteration order to the compiler. Although expressed as Python comments, they follow a structured directive format analogous to OpenMP [25] pragmas, allowing the compiler to reliably parse and verify them. Concretely, we require three pieces of information:

- **Tile size:** the logical shape of each tile along the relevant dimensions (e.g., GEMM blocks), which allows us to map tiles to chunks.
- **Tile index identifier:** a program variable (or tuple of variables) that uniquely identifies the tile being processed in a given iteration.
- **Tile scheduler:** the loop or control structure that advances the tile index and determines the order in which tiles are visited.

These annotations can often be derived from existing index-

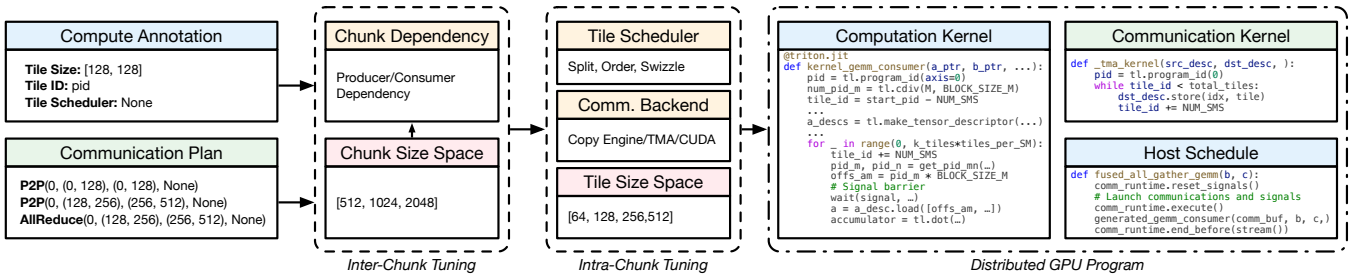


Figure 5: Compilation pipeline. In this example, we show communication using specialized SM as an independent kernel synchronized with signals. It can also be a fused kernel depending on the communication backend.

ing expressions and loop bounds with minor code changes, as illustrated in the overview section. These annotations are compiler metadata. They only expose the tile space, the tile identifier, and the mapping from a tile id to logical tensor-axis coordinates. Syncopate uses this information to relate the local Triton tile scheduler to the chunk-level schedule.

Dependency Parsing. With both the communication schedule and the annotated compute kernel in hand, Syncopate constructs a dependence graph over chunks and tiles according to the user-specified dependency intent in the plan, including how communication chunks depend on each other and how communication axes map to computation axes. For each chunk, we track its producer(s) and consumer(s), as well as any explicit ordering constraints encoded in the communication schedule (e.g., pipeline stages). For each tile, we determine which chunks it reads and writes based on its tile index, tensor layout, and the chunk-to-compute-axis mapping.

From this graph, the compiler identifies the minimal set of synchronization points needed to respect all data dependencies. Concretely, we insert wait operations in the kernel so that a tile that consumes a given chunk cannot start until the corresponding communication operator has completed. This synchronization can be implemented using different mechanisms depending on the chosen backend, but is always derived from the same chunk-level dependency structure.

Communication Code Generation. Once the communication schedule and tile scheduler have been aligned, Syncopate lowers the abstract chunk-level plan into concrete communication code. As illustrated in Fig. 7, the same logical schedule can be realized by several backends that differ in how they move chunks and how they allocate SM resources. Concretely, we support five realizations: (1) using the dedicated copy engine, (2) using TMA on a specialized SM, (3) using TMA on a co-located SM, (4) using operator-instruction load/store on a specialized SM, and (5) using operator-instruction load/store on a co-located SM. In all cases, tiles are produced on one side and consumed by operations on the other side, but the mechanism for signaling readiness and the division of compute versus communication work across SMs differ.

For each operator, Syncopate first builds a dependency graph over tiles and communication steps from the chunk schedule, then lowers this graph into backend-specific code that enforces all dependencies by construction. When targeting the copy engine or a specialized SM, the compiler emits global-memory signals and kernel launches so that communication progresses asynchronously relative to the main compute tiles. When targeting co-located SM backends, it instead generates shared-memory barriers and index bookkeeping to coordinate communication and computation within the same SM. Because all five realizations share the same logical schedule but expose different latency/bandwidth and resource tradeoffs, they form a search space for the autotuner: Syncopate automatically generates all valid implementations, measures their end-to-end performance, and selects the best-performing backend for each operator and hardware configuration.

Tile-Scheduler Swizzling. As visualized in Fig. 6, the communication plan and the original computation kernel typically induce different layouts over the global tensor: communication groups tiles into chunks based on where data needs to move, while the kernel groups tiles into waves based on its own traversal order. Prior work reconciles this mismatch by explicitly reordering data between communication and computation, paying extra global-memory traffic and synchronization. In contrast, Syncopate keeps the communicated chunks in-place and instead *swizzles* the tile scheduler at the intra-kernel level. We reorder the sequence of waves so that each chunk is consumed as soon as it arrives, and apply an intra-chunk swizzle that visits tiles in an order that preserves locality within the chunk. This chunk-based tile schedule aligns compute with communication progress without additional reordering kernels, enabling fine-grained overlap purely through scheduling. This transformation has two levels: at the chunk level, Syncopate reorders chunks to follow the communication schedule and consume chunks as they become ready; within each chunk, it applies a finer-grained tile scheduler, using the local kernel's original tile traversal by default while allowing alternative axis-based traversals when needed.

5.3 Communication-Centric Auto-Tuning

The chunk abstraction also provides a natural space for communication-centric auto-tuning. Because chunks sit exactly at the boundary between the global communication

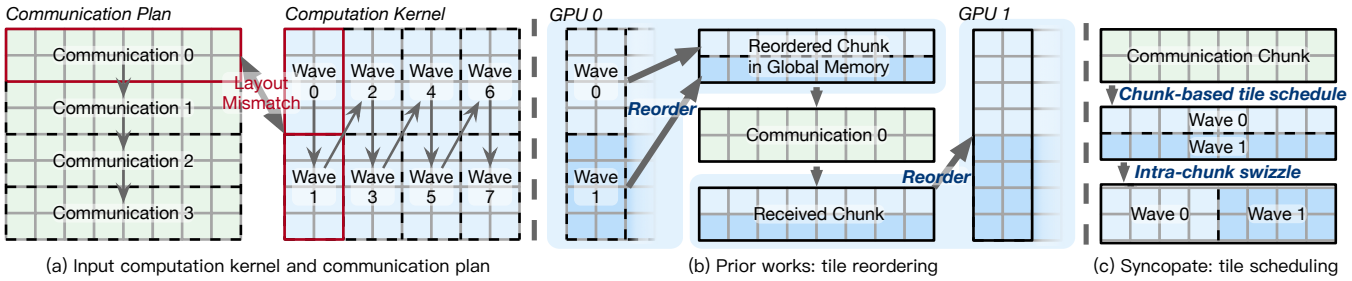


Figure 6: Tile scheduler transformation. (a) Computation and communication naturally follow different tile/chunk layouts, creating misalignment. (b) Prior approaches resolve this by inserting explicit data reordering between the two paths. (c) Syncopate instead rewrites the tile schedule to follow chunk order and applies intra-chunk swizzles for locality, aligning compute with communication progress without extra data movement.

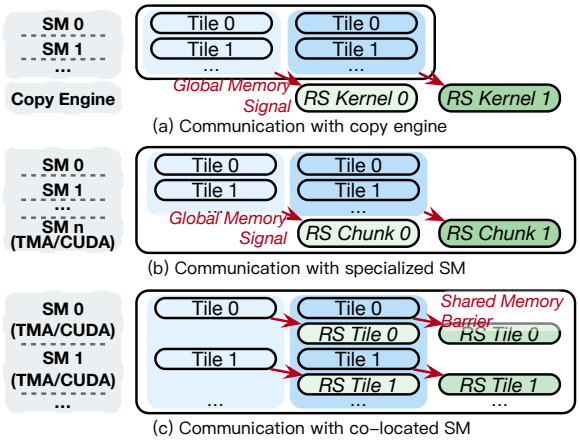


Figure 7: Communication backend selection. (a) Communication issued by the copy engine with global-memory signaling. (b) Dedicated SMs drive transfer. (c) Communication is co-located with compute on the same SM.

schedule and the local tile scheduler, changing chunk-level parameters simultaneously reshapes how data moves across ranks and how computation is ordered within each kernel. Rather than only tuning conventional kernel parameters (e.g., block sizes), Syncopate exposes a higher-level search space whose knobs directly control overlap and resource sharing. Given a local kernel and a chunk-level communication schedule, Syncopate uses an enumerate-and-measure tuning pass. The per-candidate compilation overhead is low because Syncopate performs lightweight source-to-source rewriting on Triton kernels and relies on the existing Triton JIT backend for code generation.

At the *inter-chunk* level, we tune the chunk size, shape, and split factor for each logical transfer. Larger chunks tend to achieve higher effective bandwidth on copy engines and TMA but reduce the granularity of overlap, while smaller chunks enable more fine-grained pipelining at the cost of higher per-chunk overhead and more synchronization. Different operators and model sizes favor different trade-off points: communication-heavy A2A-GEMM and GEMM-AR, for example, benefit from intermediate split factors that balance bandwidth and overlap, as confirmed by our sensitivity study in §6. The tuner searches this space under hardware-specific constraints (e.g., minimum efficient transfer size for copy engines and TMA alignment rules) and prunes configurations that would violate these hardware limits.

At the *intra-chunk* level, we tune both the computation tile configuration and how each chunk is realized by a communication backend. Given a fixed logical schedule, the compiler can instantiate each transfer using any of the backends in Fig. 7 (copy engine, intra- or inter-SM TMA, or CUDA load-/store on specialized or co-located SMs) and can vary the number of SMs assigned to communication when applicable. Some schedules benefit from using TMA for intra-node tensor movement and load/store-based communication for small, reduction-heavy shards, while others favor copy engines for large bulk transfers with minimal SM involvement. In parallel, the autotuner explores different tile sizes and intra-tile orders that better align compute waves with the chosen chunk layout, improving locality and avoiding long communication tails.

Crucially, all of these decisions operate on top of the same chunk-level dependence graph. Changing the backend, SM allocation, or tile order never requires re-deriving the global communication plan; instead, Syncopate reuses the existing schedule and regenerates backend-specific code that enforces the same dependencies. This separation of logical schedule from physical realization is what makes the search space both rich and manageable: as our ablation results show, reasonable but suboptimal settings can easily leave more than a factor of two in performance, while the tuned configuration found by our communication-centric autotuner consistently coincides with the most balanced point between computation, communication, and hardware utilization.

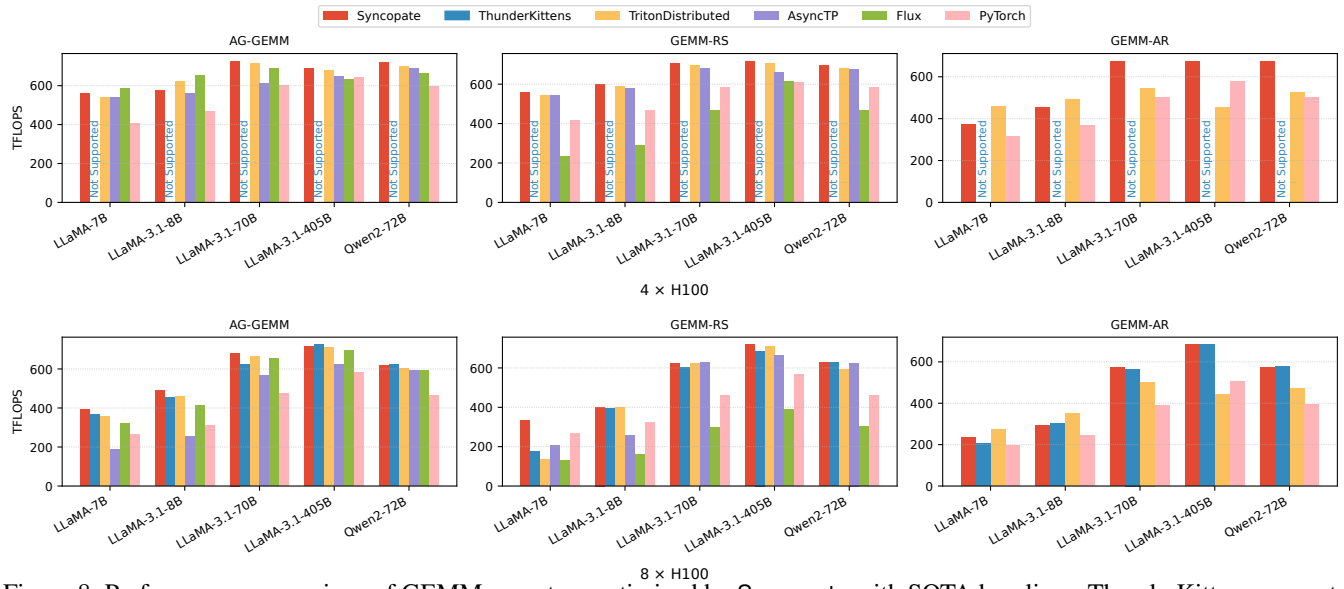


Figure 8: Performance comparison of GEMM operators optimized by Syncopate with SOTA baselines. ThunderKittens supports only 8 GPUs; 4-GPU is unsupported. When both settings are unsupported, the bar is omitted.

6 Evaluation

6.1 Experimental Setup

Testbed. We evaluate Syncopate on a server with 8 NVIDIA H100 GPUs connected via NVLink with an aggregate bandwidth of 900 GB/s, which is a representative single-node setting used in prior multi-GPU kernel studies [3, 13, 33, 46]. Unless otherwise stated, all measurements are taken on a single node using all 8 GPUs; in later experiments, we vary the number of active devices to study scalability and portability. Syncopate is implemented with CUDA v12.9, NVSHMEM v3.3.9, and PyTorch v2.7, and we run all baselines on the same software stack to ensure a fair comparison.

Workloads. We use Syncopate to optimize representative multi-GPU operators that dominate the cost of modern LLM workloads: general matrix multiplication (GEMM) and attention [7, 8, 28, 35]. For GEMM, we benchmark three distributed variants: AllGather–GEMM (AG-GEMM) and GEMM–ReduceScatter (GEMM-RS), and GEMM–AllReduce (GEMM-AR), which appear in tensor-parallel [31] or sequence parallel [17] feed-forward network (FFN) layers. For attention, we evaluate both head-parallel (HP) [15] and sequence-parallel (SP) schedules, including the overlapped RingAttention (Ring-Attn) [18] variant.

Operator shapes are derived from the FFN layers and attention layers of open-source Llama-3 [11] and Qwen [26] models, covering a range of hidden dimensions, head counts, and parallelism configurations that are typical of large-scale LLM deployments. For attention, we sweep over multiple sequence lengths to reflect common short- and long-context use cases under different distribution strategies. Overall, this workload suite exercises Syncopate across both regular GEMM-heavy and more irregular attention patterns.

Baselines. To assess the effectiveness of operators generated and tuned by Syncopate, we compare against both state-of-the-art manually engineered kernels and fully automatic compiler-based approaches. As manual baselines, we include built-in operators from fine-grained overlap DSLs such as ThunderKittens [32, 33] and TritonDistributed [45, 46], as well as highly optimized implementations including AsyncTP [38], Flux [3], and Triton kernels paired with NCCL [24] collectives.

To isolate the benefit of Syncopate's automatic fine-grained overlap, we further compare against automatic operators produced by existing distributed compiler frameworks, including Domino [36], Alpa [44], and Mercury [13]. For these comparisons, we transform the communication schedules found by each compiler into our chunk-level representation and reuse the same high-level plans in Syncopate, so that any performance difference reflects our intra-kernel overlap and backend-selection mechanisms rather than differences in global parallelization strategy.

Overlap Granularity. AsyncTP, Flux, TritonDistributed, and ThunderKittens perform fine-grained intra-kernel overlap by explicitly coordinating communication and computation inside specialized kernels or DSL programs. Domino, Mercury, and the PyTorch RingAttention baseline use inter-kernel overlap, while the remaining baselines execute communication through separate library collectives without overlap.

Base Triton Kernels. Our GEMM local kernels are off-the-shelf Triton GEMM kernels with only Syncopate annotations added. For attention, we start from standard Triton attention kernels and make minor modifications to support split-KV ex-

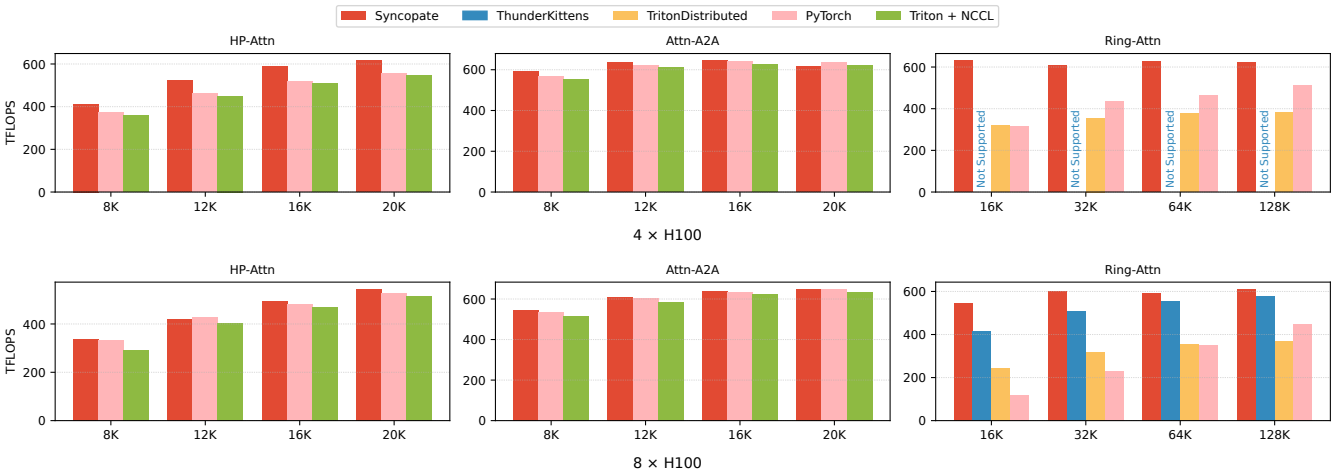


Figure 9: Performance comparison of operators optimized by Syncopate with SOTA baselines.

ecution, which is not exposed by the original kernels. In both cases, the kernels remain local compute kernels; Syncopate generates the communication logic, synchronization, backend selection, and tile-schedule transformations.

6.2 Performance Benchmark

Operator Results. Across all evaluated settings, Syncopate generates automatically optimized operators that are competitive with carefully hand-engineered baselines, while showing even larger gains over fully automatic distributed compilers. As summarized in Fig. 8 and Fig. 9, Syncopate sustains high TFLOPS on both GEMM and attention operators under multiple communication patterns (AG-GEMM, GEMM-RS, GEMM-AR, HP/SP attention, and Ring-Attn) and model configurations derived from Llama-3 and Qwen.

In Fig. 8, Syncopate is at or near the best-performing curve in almost every GEMM configuration. On common, heavily optimized AG-GEMM and GEMM-RS cases where manual kernels such as ThunderKittens, TritonDistributed, AsyncTP, and Flux already sit close to the hardware limits, Syncopate essentially matches their peak throughput on both 4- and 8-GPU settings, achieving on average 99.8% of the best baseline on 4 GPUs and 104% on 8 GPUs. For GEMM-AR, Syncopate is marginally below TritonDistributed on 7B/8B shapes, yet it scales more effectively and becomes the top-performing kernel on larger model configurations. These results indicate that our generic compilation pipeline can recover the same highly tuned overlap patterns that experts design by hand. The small advantage over the strongest baseline in some 8-GPU cases comes from the chunk-level abstraction. By grouping multiple tiles under a communication chunk, Syncopate reduces the number of fine-grained readiness checks and synchronization points while still preserving overlap.

In Fig. 9, we observe a similar trend for attention operators. Under standard HP attention with moderate sequence lengths, Syncopate closely tracks the best manual implementations, confirming that the chunk-based abstraction does not sacrifice performance even for workloads that already benefit from hand-optimized kernels. As the problem becomes harder, moving to Ring-Attn, longer sequences, and 8-GPU runs, the gap widens in favor of Syncopate. Our compiler maintains high TFLOPS while baseline kernels degrade more rapidly, since it can reshape chunks, rebalance compute and communication, and choose different backends as the sequence length and parallelism strategy change. Notably, on the most communication-intensive Ring-Attn settings, Syncopate delivers the best performance despite not being tailored specifically for this operator.

Integration Results. We further evaluate how Syncopate composes with existing automatic distributed compilers that search for the communication schedule among devices, using their communication plans as our inputs. As summarized in Fig. 10, for each of Domino, Alpa, and Mercury, we keep the original parallelization strategy and the searched communication schedule fixed, convert that schedule into our chunk-level representation, and let Syncopate generate the fine-grained overlapped kernels. Across both GEMM and attention workloads on 4- and 8-H100 configurations, integrating Syncopate consistently reduces end-to-end operator latency compared to the native implementations shipped with these systems, showing that chunk-based intra-kernel overlap exposes an additional optimization dimension on top of their global parallelization decisions.

This experiment also illustrates that Syncopate can be cleanly integrated with both partition-based and loop-based compiler stacks. Domino and Alpa operate on partitioned IRs and expose their communication plans as sequences of collectives between logical tensor partitions, while Mercury is built around a loop-centric IR for ring [18] and double-ring attention [12]. Because our interface only requires a well-defined, implementation-agnostic communication schedule and lightweight annotations on the local kernels, these systems can plug into Syncopate with minimal source modifications. This

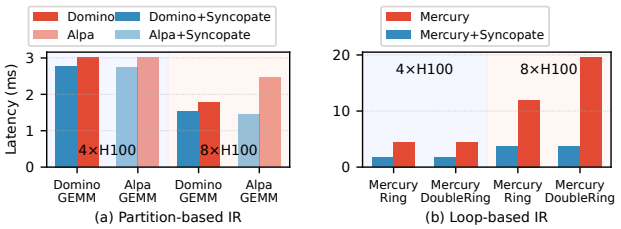


Figure 10: Evaluation of lowering partition-based and loop-based IRs generated by higher-level distributed compilers into Syncopate.

compatibility allows practitioners to reuse mature distributed compilers for global partitioning, while relying on Syncopate to automatically realize high-performance, fine-grained overlap within each generated operator.

6.3 Ablation and Sensitivity Studies

We next evaluate the effect of Syncopate's chunk-based code generation and communication-centric auto-tuning components through a series of ablation and sensitivity analyses in Fig. 11. Each subplot corresponds to one of the key design choices introduced in §5: communication backend selection and SM allocation, chunk size (split factor), and intra-tile scheduling.

Communication Backend and SM Tuning. Fig. 11(a) and (c) study how different realizations of the same logical communication schedule perform under the backends described in § 5.2, and how tuning the number of active SMs further refines this choice. In (a), for GEMM-RS and AG-GEMM, copy-engine and intra-SM TMA backends achieve the highest TFLOPS, while purely CUDA load/store realizations saturate at much lower throughput. The gap between the best and worst backend for the same logical schedule is comparable to the gap between our final implementation and several baselines in Fig. 8, indicating that backend selection alone can determine whether overlap is effective. This confirms that the ability to instantiate the same chunk schedule with different backends is crucial: no single mechanism dominates across operators, and picking a suboptimal backend can leave more than half of the available performance on the table. In (c), for a fixed backend, varying the number of SMs devoted to communication reveals a clear sweet spot where computation and communication are balanced. For both 405B and 70B GEMMs, allocating too few SMs underutilizes the link bandwidth, whereas allocating too many starves the main kernel. The optimal SM count also shifts with model size, which matches the design of our backend code generation: the autotuner treats SM allocation as a first-class knob and automatically selects a near-optimal point for each operator/hardware pair instead of relying on a single hard-coded ratio.

Chunk Size Tuning. Fig. 11(b) varies the number of chunks (split factor) used to realize the same high-level schedule for A2A-GEMM and GEMM-AR. Smaller split factors correspond to larger chunks with higher single-transfer efficiency but fewer overlap opportunities, while larger split factors increase overlap at the cost of per-chunk overhead. The curves exhibit a clear non-monotonic trend, with performance peaking at an intermediate split (e.g., 2–3 splits, about 128MB for GEMM-AR) and degrading when chunks become either too coarse or too fine. Notably, naive choices that are convenient to implement by hand (such as a single large chunk or splitting once per rank) sit far from the optimum. This behavior directly reflects the trade-off discussed in our chunk-based code generation: practitioners cannot reliably pick a single "good" chunk size by hand, whereas Syncopate searches this space automatically and selects the configuration that best matches the operator's compute/communication balance.

Intra-Tile Scheduling. Finally, Fig. 11(d) explores different intra-tile scheduling strategies for a representative GEMM configuration, varying the order in which tiles within each chunk are visited. Each point corresponds to a valid schedule with tile size on M, N, K dimensions and the pipeline stages that preserve program semantics but change locality and load balance. To represent different search candidates, we calculate the consumed shared memory size and plot the performance of these valid schedules. The wide spread in TFLOPS shows that tile order alone can introduce more than a 2 × performance difference, reinforcing the importance of the tile-scheduler transformation in § 5.2. High-performing schedules cluster around orders that align tile waves with the communication chunk order introduced in our chunk-based code generation, whereas poorly performing ones repeatedly revisit tiles in a way that destroys locality or creates long tails of unfinished work. By generating and evaluating these schedules automatically, Syncopate converges on tile orders that co-optimize cache reuse and SM utilization, without requiring users to reason manually about low-level tiling and swizzling policies.

Taken together, these studies demonstrate that Syncopate's chunk abstraction is not only expressive but also forms a practical search space: the same logical schedule can be realized via multiple backends, chunk sizes, SM allocations, and tile orders, and the differences between reasonable but non-optimal choices and the tuned configuration are often comparable to, or larger than, the gaps between systems in our main benchmarks. The auto-tuning framework in § 5.3 systematically explores this space using the code generation mechanisms of § 5.2, turning what would otherwise be a brittle, hand-tuned process into a robust, compiler-driven optimization.

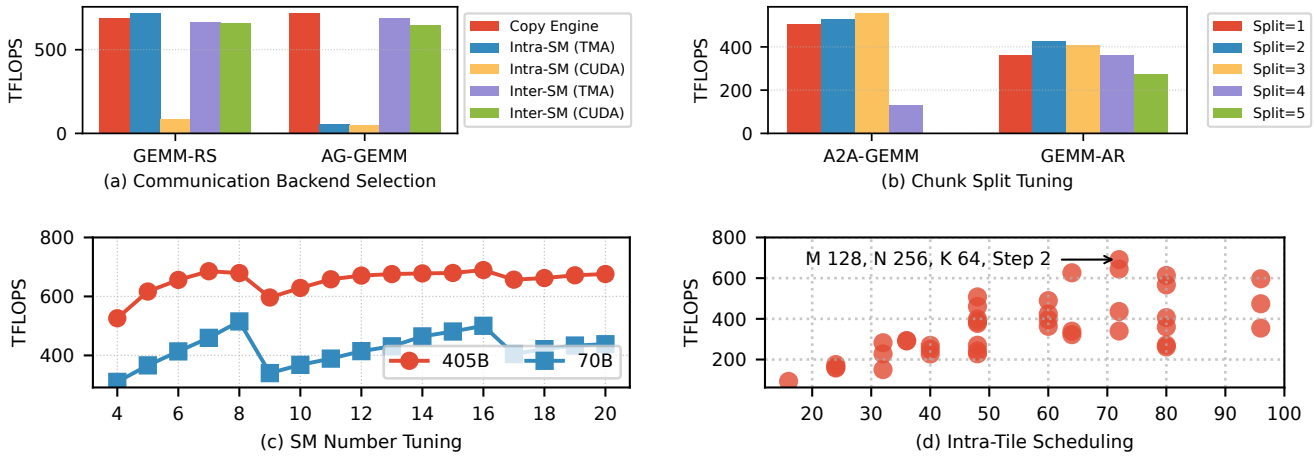


Figure 11: Ablation and sensitivity studies of Syncopate's auto-tuning design space.

7 Discussion and Future Work

Extending to other DSLs. Syncopate is implemented as a Triton source-to-source compiler, but the core abstraction only requires a tile-centric frontend that exposes tile shapes, tile identifiers, tile traversal order, and tile memory regions. Thus, DSLs such as CuTeDSL [21] or cuTile [22] could be supported with an additional frontend/lowering layer, as long as kernels are written in a tiled style. The chunk schedules, dependency construction, and backend-selection logic would remain reusable.

Scope of compute-memory overlap. Syncopate targets fine-grained overlap between tiled computation and explicit inter-GPU communication. Overlapping independent compute-bound and memory-bound kernels, as in NanoFlow-style optimizations [47], is largely orthogonal and can often be handled by stream-level scheduling. Such optimizations could be modeled by Syncopate only when the stages must be coordinated as chunked producer-consumer relationships with tile-level dependencies.

Dynamically shaped workloads. Syncopate currently specializes kernels to a fixed operator shape or a small set of common shape buckets. For variable sequence lengths, the chunk schedule can usually be re-instantiated with the new shape parameters; when chunk ranges are stored as device-side metadata, a lightweight update kernel can refresh these values before launching the generated kernel. A full dynamic-shape runtime that automatically manages shape buckets and metadata updates is left to future work.

Multi-node settings. Syncopate's chunk abstraction naturally extends to multi-node settings, since larger chunks are also a suitable unit for bandwidth-efficient NIC transfers. The main extension is not to generalize tiles or chunks, but to enrich the communication plan with a channel dimension that distinguishes intra-node and inter-node paths. Different channels can progress concurrently, allowing NVLink transfers and NIC transfers to be scheduled and overlapped independently. The compiler can then assign chunks to channels according to the topology and use different backend realizations for each channel. Supporting this setting requires additional backend and runtime support for inter-node communication, which we leave to future work.

Why a compiler abstraction? Syncopate is not meant to replace expert-written kernels for a fixed operator and hardware target. Its goal is to avoid re-encoding the same fine-grained overlap logic, signal/wait protocol, and backend-specific communication code for each schedule or parallel configuration. The chunk abstraction lets users or higher-level compilers specify the logical communication policy once, while Syncopate lowers it to concrete dependencies, synchronization, backend choices, and tile schedules.

8 Conclusion

Syncopate bridges the abstraction gap between communication planning and fine-grained overlapping by introducing a chunk-based interface that unifies communication plans with tiled GPU computation. By automatically aligning tile schedules with communication progress and selecting among heterogeneous backends, Syncopate turns intra-kernel overlap into a general compiler capability rather than a hand-engineered optimization. The framework integrates cleanly with existing distributed compilers, complementing their global parallelization strategies with backend-aware, fine-grained scheduling. This decoupled design provides a foundation for systems that co-schedule computation and communication and accommodate diverse communication backends.

References

[1] Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, et al. Partir: Composing spmd partitioning strategies for machine learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 794–810, 2025.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[3] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, et al. Flux: fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

[4] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 178–191, 2024.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.

[7] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[9] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[10] Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. Tokenweave: Efficient compute-communication overlap for distributed llm inference, 2025.

[11] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[12] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, et al. Loongtrain: Efficient training of long-sequence llms with head-context parallelism. *arXiv preprint arXiv:2406.18485*, 2024.

[13] Yue Guan, Xinwei Qiang, Zaifeng Pan, Daniels Johnson, Yuanwei Fang, Keren Zhou, Yuke Wang, Wanlu Li, Yufei Ding, and Adnan Aziz. Mercury: Unlocking multi-gpu operator optimization for llms via remote memory scheduling. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 1046–1061, New York, NY, USA, 2025. Association for Computing Machinery.

[14] Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, et al. Flashoverlap: A lightweight design for efficiently overlapping communication and computation. *arXiv preprint arXiv:2504.19519*, 2025.

[15] 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. *arXiv preprint arXiv:2309.14509*, 2023.

[16] 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 *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 402–416, 2022.

[17] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[18] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations*.

[19] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. Technical report, NVIDIA, mar 2022. White paper.

[20] NVIDIA. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) Rev 3.0.0, 2024. Accessed: 2025-02-10.

[21] NVIDIA. CuTe DSL. https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html, 2026. Accessed: 2026-04-18.

[22] NVIDIA. cuTile Python. https://docs.nvidia.com/cuda/cutile-python/, 2026. Accessed: 2026-04-18.

[23] NVIDIA Corporation. Nvidia nvlink high-speed interconnect: Application performance. Technical report, NVIDIA Corporation, 2015. Accessed: 2025-04-16.

[24] NVIDIA Corporation. *NVIDIA Collective Communications Library (NCCL)*, 2025. Version 2.26.2.

[25] OpenMP Architecture Review Board. Openmp application programming interface. https://www.openmp.org/specifications/, 2023.

[26] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[27] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Tim Harris, and Matei Zaharia. Distir: An intermediate representation and simulator for efficient neural network distribution. *arXiv preprint arXiv:2111.05426*, 2021.

[28] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[29] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[30] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers, 2018.

[31] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[32] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels, 2024.

[33] Stuart H. Sul, Simran Arora, Benjamin F. Spector, and Christopher Ré. Parallelkittens: Systematic and practical simplification of multi-gpu ai kernels, 2025.

[34] Philippe Tillet, H. T. Kung, and David Cox. 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*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[36] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. Domino: Eliminating communication in llm training via generic tensor slicing and overlapping, 2024.

[37] Haoran Wang, Lei Wang, Haobo Xu, Ying Wang, Yuming Li, and Yinhe Han. Primepar: Efficient spatial-temporal tensor partitioning for large transformer model training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 801–817, New York, NY, USA, 2024. Association for Computing Machinery.

[38] 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 *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[39] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. Tacos: Topology-aware collective algorithm synthesizer for distributed machine learning. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, MICRO '24, page 856–870. IEEE Press, 2024.

[40] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man 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)*, pages 21–38, 2025.

[41] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. Gspmd: General and scalable parallelization for ml computation graphs, 2021.

[42] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *arXiv preprint arXiv:2502.19811*, 2025.

[43] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[44] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[45] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, et al. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler. *arXiv preprint arXiv:2504.19442*, 2025.

[46] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025.

[47] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2025.

A Artifact Appendix

Abstract

The Syncopate artifact contains the source code for the chunk-centric compiler and runtime described in this paper, together with unit tests, end-to-end multi-GPU tests, experiment entry points, and a Dockerfile for the GPU software environment. A short CPU-only workflow validates the core communication-plan representation, signal planning, lowering logic, and compiler passes. On a multi-GPU NVIDIA Hopper-class system, the artifact also supports correctness and performance evaluation of generated GEMM and attention operators.

Scope

The artifact can be used to validate three central claims of the paper: (1) chunk-level communication plans can express reusable communication schedules and lower them to per-rank execution plans; (2) Syncopate can transform local tiled kernels into correct fine-grained overlapped multi-GPU kernels; and (3) the generated operators can be benchmarked across communication backends and scheduling choices. The CPU-only tests validate the compiler abstractions and passes, while the GPU tests validate generated-kernel correctness against unfused references and report execution time.

Reproducing all absolute performance numbers and comparisons in the paper requires a 4- or 8-GPU Hopper server with NVLink/NVSwitch connectivity and the external baseline implementations. Several external baselines are not vendored in the artifact; the repository documents this limitation and points to their upstream repositories.

Contents

The repository is organized as follows:

- `syncopate/communication/`: communication-plan descriptors, signal planning, code generation, and runtime support;
- `syncopate/computation/`: GEMM and attention kernel templates and the source-to-source transformation pass;
- `syncopate/interface/`: lowering from communication plans to per-rank schedules and tile schedules;
- `tests/`: CPU-only compiler tests and end-to-end multi-GPU correctness/performance tests;
- `experiments/`: figure-oriented experiment entry points for the operator, attention, compiler-integration, and ablation results; and
- `utils/`: the distributed launcher, communication microbenchmarks, and helper scripts.

Hosting

The artifact is publicly available under the MIT license at `https://github.com/tie-pilot-qxw/syncopate`. The version evaluated for OSDI '26 Artifact Evaluation is commit `ce2b13e496eb6629ab42aac2aec1d9e31e084ba6`. Evaluators and readers should check out this commit to obtain the fixed artifact version.

Requirements

The CPU-only validation requires Linux, Python 3.10 or newer, PyTorch, and `pytest`; it typically completes in under five minutes. Full generated-kernel and performance experiments require Linux, Docker with the NVIDIA Container Toolkit, and a single node with at least four NVLink/NVSwitch-connected NVIDIA Hopper-class GPUs; some experiments require eight GPUs.