
TILELINK: GENERATING EFFICIENT COMPUTE-COMMUNICATION OVERLAPPING KERNELS USING TILE-CENTRIC PRIMITIVES

Size Zheng^{1*} Jin Fang^{1*} Xuegui Zheng¹ Qi Hou¹ Wenlei Bao¹ Ningxin Zheng¹ Ziheng Jiang¹
Dongyang Wang¹ Jianxi Ye¹ Haibin Lin¹ Li-Wen Chang¹ Xin Liu¹

ABSTRACT

Large deep learning models have achieved state-of-the-art performance in a wide range of tasks. These models often necessitate distributed systems for efficient training and inference. The fundamental building blocks for distributed model execution are intra-layer parallel operators. The most effective approach to enhancing the performance of intra-layer parallel operators involves overlapping computation with communication. The overlapping can be achieved through either operator decomposition or kernel fusion. While decomposing operators is straightforward to implement, it often results in suboptimal performance. On the other hand, fusing communication kernels with compute kernels demands significant expertise and is error-prone.

In this paper, we propose TILELINK to enable efficient compilation and generation of overlapped compute-communication kernels. TILELINK is composed of frontend and backend. In the frontend, TILELINK decouples the design space of communication and computation, linking these two parts via tile-centric primitives. In the backend, TILELINK translates these primitives into low-level communication instructions, integrating the communication and computation components to achieve overlapped execution. In experiments, TILELINK achieves from $1.17\times$ to $20.76\times$ speedup to non-overlapping baseline and achieves performance comparable to state-of-the-art overlapping libraries on GPUs.

1 INTRODUCTION

Large deep learning models keep growing in both model size and performance. These models have achieved state-of-the-art results in a wide range of domains including natural language processing (OpenAI, 2023; Rivière et al., 2024; Dubey et al., 2024; DeepSeek-AI et al., 2024), vision processing (Radford et al., 2021; Ataallah et al., 2024; Bai et al., 2023; Lu et al., 2024), and reasoning (OpenAI, 2024; Shao et al., 2024). The substantial sizes of these models, coupled with their immense computational demands, necessitate parallel execution across distributed systems. Various parallel methods have been proposed to accelerate distributed processing by exploiting both intra-layer and inter-layer parallelism (Rajbhandari et al., 2020; Narayanan et al., 2021; Huang et al., 2019; Rasley et al., 2020).

Since intra-layer parallelism forms the foundation of parallel computing, a significant body of work has focused on exploring it (Narayanan et al., 2021; NVIDIA, 2022b; Liu et al., 2023). While parallel execution enhances overall performance, communication between devices still incurs

significant overhead, limiting further improvements in computational efficiency (Chen et al., 2024; Chang et al., 2024). Previous work (Chang et al., 2024) indicates that communication overhead constitutes approximately 10% to 50% of the total execution overhead even in machines equipped with high-speed inter-device links.

Overlapping communication with computation is an effective strategy for enhancing computational efficiency. The core idea is to map communication and computation to distinct hardware units, allowing them to operate concurrently. To handle data dependency between communication and computation operators, synchronization or barriers are inserted into the loop of data transfer and computation. Previous work on overlapping communication with computation mainly uses two techniques: operator decomposition and kernel fusion.

Operator decomposition (NVIDIA, 2022b; Wang et al., 2023; Chen et al., 2024) involves breaking down both communication and computation kernels into smaller, homogeneous kernels. The data dependencies are then distributed across multiple communication-computation kernel pairs. The smaller kernels, once split, can be dispatched to different streams, allowing communication and computation kernels to operate on separate data shards simultaneously. Operator decomposition can be easily imple-

*Equal contribution ¹ByteDance Seed. Correspondence to: Xin Liu <liuxin.ai@bytedance.com>.

mented on modern deep learning frameworks such as PyTorch (Ansel et al., 2024) or TensorFlow (Abadi et al., 2016), enabling systematic exploration of the entire design space for communication-computation overlap, including model-level, layer-level, and operation-level as pointed out in previous work (Chen et al., 2024). However, synchronization between these decomposed kernels necessitates host intervention, introducing non-negligible overhead at runtime. Furthermore, the performance of decomposed kernels may be degraded due to low cache utilization and resource quantization inefficiency.

On the other hand, the kernel fusion method (Jangda et al., 2022; Chang et al., 2024) combines communication and computation kernels into one fused kernel to overcome the above disadvantages. Within fused kernels, communication is mapped to either DMA (Direct Memory Access) engines or processing cores (e.g., streaming multiprocessors on a GPU), while computation is executed simultaneously on other processing cores. Data dependencies are managed using on-device barriers, and processing cores responsible for data transfer communicate with computation cores through atomic or communication instructions. This method is efficient in terms of performance but often requires high-level hardware expertise to implement efficient kernels, and it struggles to keep pace with rapid algorithm development.

To address the challenges inherent in existing approaches, we propose TILELINK, a framework designed to enhance the development efficiency of overlapping kernels through compilation. TILELINK consists of two main components: a frontend and a backend. In the frontend, TILELINK decouples the design space of communication and computation kernels, enabling each to utilize distinct optimization strategies and tiling methods. To allow the communication and computation kernels to operate with different tile sizes, it relies on tailored barrier controls to maintain producer-consumer dependencies, ensuring correct and efficient execution. Typically, this fusion is achieved by directly programming in assembly. To automate the fusion of communication and computation kernels without requiring low-level assembly code, TILELINK offers a set of tile-centric primitives. These primitives provide abstract semantics for signaling and data communication between devices, while concealing low-level details such as pointer management and barrier control.

In the backend, TILELINK compiles tile-centric primitives into low-level hardware instructions, integrating them with the communication and computation kernels. To ensure correct data exchange and barrier manipulation operations, TILELINK employs a tile-centric mapping strategy, which includes shape mapping, rank mapping, and channel mapping. This tile-centric mapping can be either static or dynamic. Static mapping uses affine transformations at com-

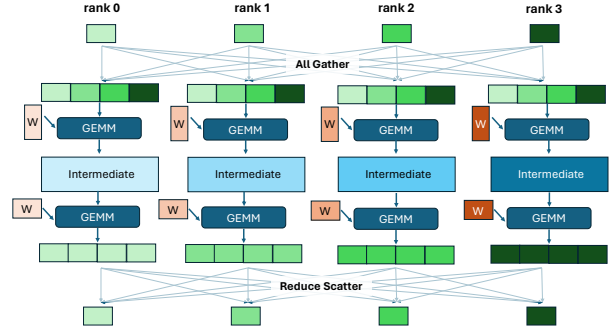


Figure 1. Intra-layer parallel FFN example.

pile time to map tile IDs to shape ranges, rank IDs, and communication channels. In contrast, dynamic mapping computes these mappings on-the-fly at runtime, allowing greater flexibility. To show the flexibility and generality of TILELINK, we implement a broad range of overlapped workloads using TILELINK, including self-attention, MLP (multilayer perceptron), and MoE (mixture of experts). In addition to programming efficiency, TILELINK also achieves high performance on GPUs. Evaluation on $8 \times \text{H800}$ GPUs shows that TILELINK can achieve from $1.17 \times$ to $20.76 \times$ speedups over non-overlapping baselines, achieving comparable or better performance to overlapping libraries, such as FLUX (Chang et al., 2024) and RingAttention (Liu et al., 2023). For end-to-end evaluation, we test eight different language models on $8 \times \text{H800}$ GPUs and the average speedup of TILELINK is $1.32 \times$ compared to PyTorch. We also benchmark TILELINK on two nodes of $8 \times \text{H800}$ (totally 16 GPUs) and the average speedup to PyTorch is $1.29 \times$.

2 BACKGROUND

2.1 Communication and Intra-Layer Parallelism

Operator-Centric Communication Primitives: Collective communications are frequently used in parallel execution of large models. Existing libraries (NVIDIA, 2024) and frameworks (Ansel et al., 2024; Abadi et al., 2016) employ operator-level primitives for common communication patterns, such as AllReduce, ReduceScatter, AllGather, and All2All. These primitives need to perform system synchronization before and after data transfer to follow the SPMD (Single Program, Multiple Data) programming model and integrate seamlessly with other operators. However, coarse-grained synchronization can cause computational units to be idle during communication, thus reducing computational efficiency. We call these communication primitives *operator-centric primitives*.

Intra-Layer Parallelism with Operator-Centric Primitives: For large models (*i.e.*, Transformer-based models),

intra-layer parallelism is primarily applied to two components: the attention part and the FFN (feed-forward network) part, which is composed of MLP (multilayer perceptron) layers or MoE (mixture of experts) layers. For the attention part, the context (key and value) is sharded across devices. Before computation, these context shards are gathered to form a complete context for self-attention. This parallel algorithm is referred to as sequence-parallel (Narayanan et al., 2021; Liu et al., 2023).

For the FFN part, the weights of the two layers in the MLP or MoE are sharded across devices. First, input data is gathered from different ranks, followed by local computation using the corresponding weight shards. Finally, the partial results are reduced and scattered to the appropriate ranks. This algorithm, commonly used in previous work (Narayanan et al., 2021; Jangda et al., 2022; Wang et al., 2023), is depicted in Figure 1 and is referred to as tensor-parallel FFN. Using existing communication libraries and frameworks, tensor-parallel FFN is expressed as AllGather + GEMM (or GroupGEMM) followed by GEMM (or GroupGEMM) + ReduceScatter.

2.2 Communication and Computation Overlapping

Overlapping communication and computation has been extensively explored in prior studies (Chen et al., 2024; Jangda et al., 2022; Wang et al., 2023; Chang et al., 2024). Centauri (Chen et al., 2024) introduces a comprehensive three-level design space encompassing model-level, layer-level, and operation-level overlapping. Intra-layer parallelism serves as the foundational element across these levels of overlapping. TILELINK focuses on intra-layer overlapping; generalizing TILELINK’s techniques to inter-layer or model-level overlapping is feasible, but beyond the scope of this paper. For intra-layer parallelism, there are two main ways to achieve overlapping: operator decomposition and kernel fusion. As shown in Table 1, we summarize the features of representative studies and TILELINK.

Operator Decomposition: This approach splits the original communication and computation operators into smaller, fine-grained units. These smaller operators enable more precise synchronization control, allowing communication operators to execute in parallel with computation operators that do not have data dependencies. Operator decomposition is advantageous due to its straightforward implementation and compatibility with existing libraries and frameworks. However, using smaller operators can lead to inefficiencies, including low L2 cache utilization (Tillet et al., 2019a) and resource quantization inefficiency (Osama et al., 2023). Additionally, synchronization between kernels requires host intervention, introducing non-negligible overhead. Representative works employing operator decomposition include Dist-Einsum (Wang et al., 2023), Asynchronous Tensor Par-

Table 1. Feature comparison of TILELINK and previous work.

Name	Compile	Method	Primitive
CoCoNet	Yes	Fusion	No
Dist-Einsum	Yes	Decompose	operator-centric
Centauri	No	Decompose	operator-centric
FLUX	No	Fusion	No
Async-Torch	No	Decompose	operator-centric
TILELINK	Yes	Fusion	tile-centric

Table 2. Motivational example.

Configurations of TP MLP		
batch × sequence length	hidden dim	intermediate size
8192	4096	11008
Performance of Different Overlapping Techniques		
Method	Performance	
	AG+GEMM	GEMM+RS
Non-Overlap	0.676 ms	0.541 ms
Decomposition	1.301 ms	1.443 ms
Fusion (FLUX)	0.504 ms	0.610 ms
TILELINK (ours)	0.505 ms	0.504 ms
Lines of Code	FLUX ≈ 2,000 .cu	TILELINK (ours) ≈ 200 .py

allel PyTorch (Ansel et al., 2024), and Centauri (Chen et al., 2024).

Kernel Fusion: This approach fuses communication and computation kernels. Typically, the fused kernel allocates part of the processing cores to communication tasks and the remaining cores to computation tasks. Cores assigned to different tasks use on-device barriers to communicate execution states. The fused kernel eliminates the need for host intervention during synchronization, improves cache utilization, and mitigates resource quantization inefficiency, potentially achieving better performance than the operator decomposition method. However, developing fused kernels on modern accelerators, such as GPUs, presents challenges. On one hand, low-level control over barriers and hardware-related communication instructions demands a high level of expertise. On the other hand, improper fusion design may lead to performance degradation due to resource conflicts between communication and computation cores. Consequently, only a few highly optimized libraries (Punniyamurthy et al., 2023; Chang et al., 2024) or domain-specific compilers (Jangda et al., 2022) support the kernel fusion method.

2.3 Code Generation Compilers

With the rapid advancement of code generation compilers (Ragan-Kelley et al., 2013; Chen et al., 2018; Tillet et al., 2019b), generating high-performance code for attention or FFN has become practical. Although previous overlapping compilers such as CoCoNet (Jangda et al., 2022) and Dist-Einsum (Wang et al., 2023) can generate overlapped kernels,

they are restricted to fixed overlapping patterns without programming flexibility at the operator level. In contrast, TILELINK uses tile-centric primitives and enables efficient compilation for a variety of workloads.

2.4 Motivational Example

To illustrate the benefits of TILELINK, we use a tensor-parallel MLP layer as a motivational example. The input shape of the MLP layer, detailed in Table 2, corresponds to the configuration used in the LLaMA-7B model. This MLP layer is implemented as AllGather + GEMM (AG + GEMM) followed by GEMM + ReduceScatter (GEMM + RS), as depicted in Figure 1. We compare the performance of different techniques for these two parts in Table 2. *Non-Overlap* is to use cuBLAS (NVIDIA, 2022a) and NCCL (NVIDIA, 2024) with no overlapping. *Decomposition* uses the operator decomposition technique, with performance results taken from Async-TP PyTorch (Liang et al., 2024). *Fusion* refers to the kernel fusion technique, measured using the open-source library FLUX (Chang et al., 2024).

On one hand, we compare the performance achieved by different techniques. As shown in the Table 2, the decomposition technique delivers the lowest performance, while the fusion technique achieves the best results for AG + GEMM. TILELINK achieves the best performance for GEMM + RS and comes very close to FLUX for AG + GEMM (about 99%). These findings demonstrate that TILELINK is capable of delivering performance that is comparable to or better than previous approaches. On the other hand, we compare the lines of code required by FLUX and TILELINK. FLUX involves approximately 2,000 lines of CUDA code, whereas TILELINK achieves similar performance with only around 200 lines of Python code, resulting in a roughly 10 \times improvement in programming efficiency. This motivational example highlights the significant advantages of TILELINK.

3 FRONTEND PRIMITIVES

In this Section, we explain the frontend of TILELINK. We first explain the decoupled design space. Then, we present TILELINK’s tile-centric primitives.

3.1 Decoupled Design Space

There are two ways to design compute-communication fusion kernels. One is to tightly couple the optimization choices of the two parts, including tile size, tile order, and resource mapping, while the other is to decouple computation and communication kernel design. TILELINK adopts the latter one because the decoupled design space enables more flexibility in kernel design and could result in better performance.

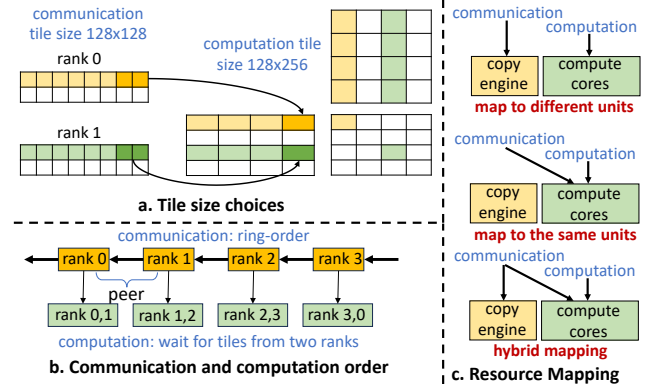


Figure 2. Examples of the three design sub-spaces of communication and computation.

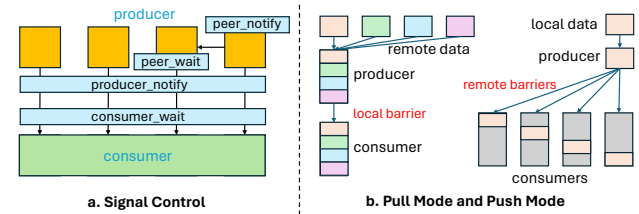


Figure 3. Tile-centric primitives support different signal control and data transfer directions.

We divide the decoupled design space into three subspaces: tile size, tile order, and resource mapping. For each of these subspaces, the communication and computation components can make independent choices to optimize their performance. In the tile size subspace, the communication and computation components can choose different tile sizes. For example, as illustrated in Figure 2a, the communication part transfers a tile of 128×128 at a time, while the computation part consumes a tile of size 128×256 at a time. This differentiation in tile size helps each component achieve optimal performance by aligning with the number of processing cores it uses. For instance, given an AllGather + GEMM problem with the tensor size of $M \times N \times K$, where the AllGather part binds dimension M , K to processing cores, and the GEMM part binds dimension M , N . If the communication component uses more cores, a smaller tile size will be beneficial, because all core resources can be fully utilized; conversely, if it uses fewer cores, a larger tile size will be more effective.

In the tile order subspace, the communication component may utilize a different tile order compared to the computation component. For instance, communication can adopt various data transfer orders, such as ring order, full-mesh all-to-all order, or other patterns, while the computation

Table 3. Tile-centric primitives in TILELINK

Name	Usage	Explanation
<i>producer_tile_notify</i>	<i>producer_tile_notify(tile_id, mode)</i>	Mark producer tile done and notify its consumer tile, consumer tile is marked ready when all the producer tiles it depends on are done
<i>consumer_tile_wait</i>	<i>consumer_tile_wait(tile_id)</i>	Consumer tile is blocked until all its dependent producer tiles done
<i>peer_tile_notify</i>	<i>peer_tile_notify(tile_id, rank)</i>	Mark current tile done and notify its peer tiles in another rank
<i>peer_tile_wait</i>	<i>peer_tile_wait(tile_id, rank)</i>	Block current tile until its peer tile in another rank is done
<i>rank_notify</i>	<i>rank_notify(tile_id, rank)</i>	Tell another rank that data at tile_id is ready
<i>rank_wait</i>	<i>rank_wait(rank)</i>	Block current rank until another rank is done
<i>tile_push_data</i>	<i>tile_push_data(tensors, tile_id, data)</i>	Send a tile of data to one (p2p) or all the (broadcast) remote tensors
<i>tile_pull_data</i>	<i>data = tile_pull_data(tensors, tile_id)</i>	Load one (p2p) or all the (broadcast) tiles of data from remote tensors
<i>rank_copy_data</i>	<i>rank_copy_data(src, dst)</i>	Copy data from src rank to dst rank

component can begin processing data tiles from any rank. There is a trade-off associated with the choice of tile order. If the computation component waits for data tiles from multiple ranks, it may achieve better cache efficiency when operating on larger chunks of data; however, this approach may result in longer wait time. Conversely, if the computation component only waits for data tiles from a single rank, it can start computations earlier, but this may lead to lower overall computation efficiency. Figure 2b shows an example, where communication uses ring order and computation waits for data from two ranks at each iteration of computation.

In the resource binding subspace, the communication and computation components can be mapped to either different units or the same unit, as illustrated in Figure 2c. If the communication component utilizes the copy engine (DMA), it avoids resource conflicts with the computation component. However, this approach involves host interference, which introduces additional overhead. On the other hand, if the communication component employs compute cores for data copying, it may lead to resource conflicts with the computation component but eliminates host overhead. This strategy is particularly suitable in scenarios where the computation component cannot fully utilize all available processing cores.

3.2 Tile-Centric Primitives

Decoupling the design space of communication and computation introduces synchronization challenges. Since the two components utilize different tile sizes, tile orders, and resource mappings, synchronizing them necessitates complex low-level programming with communication instructions. For example, on GPUs, instructions such as *ld.global.acquire* and *red.release* are required. However, the programming model for these instructions does not align with that of code generation compilers (Chen et al., 2018; Tillet et al., 2019b), as existing compilers lack support for a memory consistency model.

To address this issue, TILELINK offers a set of tile-centric

primitives. These primitives introduce memory consistency semantics and adhere to the tile-level abstraction utilized in the compiler, distinguishing them from the operator-centric primitives provided by previous frameworks (Ansel et al., 2024; Abadi et al., 2016) and libraries (NVIDIA, 2024). The primitives of TILELINK are summarized in Table 3. They are categorized into two groups: signal primitives and data primitives. Each group contains both device-side primitives and host-side primitives.

3.2.1 Signal Primitives

Signal primitives are designed to manage barriers between communication and computation. They include *producer(peer)_tile_notify*, *consumer(peer)_tile_wait*, and *rank_notify(wait)*. For device-side primitives, the *producer_tile_notify* and *consumer_tile_wait* primitives are applied to producer-consumer relationships, such as those between the tiles of AllGather and GEMM. The *peer_tile_notify* and *peer_tile_wait* primitives are primarily used for tiles of the same operator across different ranks, enabling users to construct various tile orders. For host-side primitives, the *rank_notify(wait)* primitive is used to manage barriers between the copy engine and compute cores. When communication is mapped to the copy engine, these primitives facilitate the control of tile orders between communication and computation. Figure 3a shows the signal control between communication and computation parts.

Notify primitives require either *mode* argument or *rank* argument to clarify which remote ranks to notify. TILELINK provides two choices for *mode* argument: *p2p* and *broadcast*. *p2p* means that only one target rank will be notified, which is calculated by the offset of the given *tile_id* in the global tensor view; *broadcast* means that all the ranks will be notified.

Memory Consistency: In parallel executions, memory operations performed by different processes/threads may become visible to others in a non-uniform order. The memory consistency model specifies constraints to prevent contradictions in the observed order of operations across processes/threads.

The signal primitives provide strict memory consistency semantics. The notify primitives carry release semantics, ensuring that any memory access occurring before *producer(peer)_tile_notify* and *rank_notify* cannot be executed after these notify primitives. Conversely, the wait primitives carry acquire semantics, ensuring that any memory access following *consumer(peer)_tile_wait* and *rank_wait* cannot be executed before these wait primitives. This strict memory consistency must also be taken into account during backend compilation, which will be discussed later.

3.2.2 Data Primitives

Data primitives facilitate data transfer and include *tile_push(pull)_data* and *rank_copy_data* primitives. These primitives control the resource mapping and tile sizes of the transferred data. The device-side *tile_push(pull)_data* primitive maps communication to processing cores, while the host-side *rank_copy_data* primitive maps communication to the copy engine. There are two modes for data transfer—pull and push—each suited for different synchronization methods. In the pull mode, the producer reads data from all other ranks and notifies its consumer using local barriers. In contrast, the push mode allows the producer to write local data to all other ranks while notifying its remote consumers of the data’s arrival. Figure 3b illustrates the differences between the two modes. The choice between pull and push modes may impact performance (as pointed out in FLUX (Chang et al., 2024)), depending on factors such as data shapes, tiling strategies, and available hardware resources. Notably, the *rank_copy_data* primitive supports both modes through peer-to-peer copying, with the data transfer direction indicated by the order of the source and destination pointers.

4 BACKEND MAPPING

The backend of TILELINK handles the compilation of both communication and computation components into low-level device codes. To enable code generation for distributed systems, TILELINK employs a tile-centric mapping technique that links parts of the communication and computation. In this section, we first explain the tile-centric mapping approach and the compilation process used by TILELINK. Next, we describe how TILELINK ensures memory consistency. Finally, we briefly summarize additional compilation techniques applied for single device.

4.1 Tile-Centric Mapping

TILELINK uses a tile-centric mapping approach to compile frontend primitives into low-level code. Tile-centric mapping consists of three components: shape mapping, rank mapping, and channel mapping. Shape mapping associates each *tile_id* with a specific tensor shape slice. Rank mapping

links each *tile_id* to a device rank. Channel mapping assigns each *tile_id* to a communication barrier. We use f_S, f_R, f_C to represent these three mappings, respectively. Depending on the workload type, different mapping functions should be used. We classify the different mappings into two groups: static mapping and dynamic mapping.

Static mapping refers to mappings that can be decided at compile time. Static mapping is commonly used when data sharding strategy is fixed such as tensor-parallel MLP and sequence-parallel self-attention. We use affine operations to handle static mapping (f_S, f_R, f_C are affine). For example, for AllGather (pull mode) + GEMM (problem size $M \times N \times K$) on R ranks with C channels per rank (each rank corresponds to C barriers), the producer AllGather uses tile size $Tm_p \times Tn_p$, and the input tensor is sharded along M dimension. Given producer tile *tile_id_p*, the shape range, source rank, and channel can be computed as follows:

$$\begin{aligned} M_{per_rank} &= \lceil \frac{M}{R} \rceil, & M_{per_channel} &= \lceil \frac{M}{R * C} \rceil, \\ range_M &= [tile_id_p * Tm_p, tile_id_p * Tm_p + Tm_p), \\ src_rank &= \lfloor \frac{tile_id_p}{\lfloor \frac{M_{per_rank}}{Tm_p} \rfloor} \rfloor, & channel &= \lfloor \frac{tile_id_p}{\lfloor \frac{M_{per_channel}}{Tm_p} \rfloor} \rfloor. \end{aligned}$$

Similarly, we can compute the mapping from consumer *tile_id_c* to shape range, rank, and channel.

Dynamic mapping refers to mappings computed at runtime, which are essential for workloads with dynamic data sharding requirements. For example, in the MoE data sharding strategy, dynamic routing determines the data distribution, and each tile may require tokens from any other rank. It is impossible to determine from which ranks to gather data or at which channel to wait for a barrier at compile time. Consequently, the mapping must be computed at runtime. To support dynamic mapping, TILELINK transforms these mappings into lookup tables, whose values can be filled at runtime, while the access operations to these lookup tables are determined at compile time. Formally, dynamic mapping is

$$\begin{aligned} range &= [f_S_low[tile_id], f_S_high[tile_id]), \\ rank &= f_R[tile_id], & channel &= f_C[tile_id], \end{aligned}$$

where f_S_low, f_S_high, f_R and f_C are lookup tables, the values of them will be filled at runtime by other dynamic logics (e.g., dynamic routing).

4.2 Compilation for Memory Consistency

In backend compilation, the frontend primitives with memory consistency semantics are compiled to corresponding device instructions such as *ld.global.acquire* and *red.release*. However, directly translating these primitives is not enough to guarantee memory consistency. For most computation

```

GEMM Kernel:
Input: local tokens[M,K], weights[K,N]
Output: remote gemm_out[M,N]
Reduce Kernel:
Input: local buffer[rank][M,N]
Output: local out[M_per_rank,N]
1 if block_id < SM_id - 20 # GEMM kernel:
2   tid_m, tid_n = calc_tid(M, N, BLOCK_M, BLOCK_N)
3   acc = zeros(BLOCK_M, BLOCK_N)
4   for k in range(K / BLOCK_K):
5     data = load(tokens, [tid_m, k])
6     weight = load(weights, [k, tid_n])
7     acc += dot(data, weight)
8   store(gemm_out, [tid_m, tid_n], acc)
9   producer_tile_notify([tid_m, tid_n], "p2p")
10 else: # ring reduce kernel
11   to_rank = (rank - 1 + WORLD_SIZE) % WORLD_SIZE
12   tid_m, tid_n = calc_tid(M_per_rank, N, BLOCK_M', BLOCK_N')
13   for stage in range(WORLD_SIZE):
14     seg = (rank + stage + 1) % WORLD_SIZE
15     tid_m_global = tid_m + seg * (M_per_rank / BLOCK_M')
16     consumer_tile_wait(tid_m_global, tid_n)
17     data = load(gemm_out, [tid_m_global, tid_n])
18     if stage != 0:
19       peer_tile_wait([tid_m_global, tid_n], rank)
20     data += load(buffers[rank], [tid_m_global, tid_n])
21     if stage == WORLD_SIZE - 1:
22       store(out, data, [tid_m, tid_n])
23     else:
24       tile_push_data(
25         buffers[to_rank], [tid_m_global, tid_n], data)
26       peer_tile_notify([tid_m_global, tid_n], to_rank)

```

Figure 4. GEMM+RS overlapping kernel using TILELINK.

kernels, multi-stage pipeline is applied to enhance load-compute balance and improve overall performance. Compiling original programs into multi-stage version requires operator reordering, during which some memory access operations may be reordered before or after TILELINK primitives unexpectedly. To address this issue, TILELINK enforces strict data dependencies between its primitives and their following load/store operations so that its primitives can be correctly reordered and unrolled by pipelining passes.

4.3 Other Compilation Optimizations

Apart from the aforementioned techniques, TILELINK also leverages strategies for single-device optimization to achieve high performance, which has been well-studied in previous work (Chen et al., 2018; Tillet et al., 2019b). The optimizations primarily include two aspects: memory optimization and pipeline optimization. Memory optimization involves the automatic allocation of on-chip register buffers and shared memory buffers for computation. Data access to global buffers is coalesced, and the access pattern to shared memory is transformed to avoid bank conflicts. Pipeline optimization involves rearranging data load/store operations and computations to form a multi-stage pipeline. Local data copies are mapped to dedicated asynchronous engines, such as the Tensor Memory Accelerator (TMA) of GPUs. Computation is mapped to high-performance units, such as the Tensor Core units of GPUs.

5 KERNEL DESIGN WITH TILELINK

To demonstrate the flexibility and generality of TILELINK, we present how to design overlapped kernels for GEMM + ring ReduceScatter, AllGather + MoE, and AllGather KV + self-attention. These three examples are representative because they utilize different tile orders (ring and all-to-all), different mappings (static and dynamic), and different hardware resources (device and host).

Figure 4 shows the pseudo code for the GEMM + ring ReduceScatter kernel. Both computation and communication use SMs, we use 20 SMs for communication in this example (see line 1). The producer GEMM stores partial outputs in the local tensor and notifies its consumer using *producer_tile_notify* (line 9). The consumer ReduceScatter waits for its producer at line 16. Once the data from the producer is ready, the consumer kernel performs a local reduction (line 20) and passes the partial results to its previous rank (line 24). Signal control between peer ranks is managed using the primitives *peer_tile_wait* and *peer_tile_notify* at lines 19 and 26, respectively. This example uses static mapping and demonstrates how to program communications in two directions: producer-consumer and peer-to-peer.

Figure 5 shows the pseudo code for AllGather + MoE. Again, both computation and communication use SMs, and we use 20 SMs for communication (see line 1). Note that MoE requires dynamic routing (*topk_ids* in inputs) to select experts for each token, necessitating dynamic mapping. We use *table* to denote the lookup tables for shape mapping, rank mapping, and channel mapping. All the primitives involved should take *table* as arguments so that TILELINK can generate correct code using the dynamic mappings. Additionally, the *table* is required by the *load* primitive because it uses the shape mapping in *table* to gather the correct tokens (line 11) and the correct *top_ids* (line 12) for the corresponding tokens, which are needed by the current tile.

Figure 6 shows the pseudo code for AllGather KV + self-attention (sequence parallel). In this example, communication use copy engine. we use host primitives to trigger copy engines. The communication and computation run on two different streams. Communication is done using *rank_copy_data* primitive, and the tile size for communication part is simply divide KVCache sequence length (*S*) by the total number of ranks (*WORLD_SIZE*). For computation part, the tile size is different. Tile-centric mapping is used to guarantee the correct barrier operations between communication and computation parts.

These examples show that TILELINK is flexible in overlapping kernel design and reduces programming effort, thanks to our tile-centric primitives and mappings.

```

AllGather Kernel:
Input: remote token_shards[WORLD_SIZE, M_per_rank, H]
       lookup tables: table = <f_S, f_R, f_C>
Output: local tokens[M, H]
MoE Kernel:
Input: local tokens[M, H], topk_ids[M, TOPK],
       local weights[E, H, D]
Output: local out[M*TOPK, H]
1 | if block_id < 20 # AllGather kernel :
2 |   tid_m, tid_n = calc_tid(M, H, BLOCK_M, BLOCK_N)
3 |   data = tile_pull_data(token_shards, [tid_m, tid_n], table)
4 |   store(tokens, data, [tid_m, tid_n])
5 |   producer_tile_notify([tid_m, tid_n], "p2p", table)
6 | else # MoE kernel
7 |   tid_m, tid_n = calc_tid(M*TOPK, H, BLOCK_M', BLOCK_N')
8 |   acc = zeros(BLOCK_M', BLOCK_N')
9 |   for k in range(K / BLOCK_K):
10 |     consumer_tile_wait([tid_m, tid_n], table)
11 |     a = load(tokens, [tid_m, tid_n], table)
12 |     expert_offs = load(topk_ids, [tid_m], table)
13 |     b = load(weights + expert_offs, [tid_z, tid_n])
14 |     acc += dot(a, b)
15 |   store(out, [tid_m, tid_n], acc)
    
```

Figure 5. AG + MoE overlapping kernel using TILELINK.

```

AllGather:
Input: remote K_shards, V_shards
       [WORLD_SIZE, B, H, S_per_rank, D]
Output: local K, V[B, H, S, D]
Self-attention Kernel:
Input: local Q_shard[B, H, S_per_rank, D], local K, V[B, H, S, D]
Output: local O[B, H, S_per_rank, D]

1 | def compute_func(): # Define Self-attention kernel
2 |   tid_z, tid_m_local = calc_tid(
3 |     B*H, S_per_rank, BLOCK_Z, BLOCK_M)
4 |   acc = zeros(BLOCK_Z, BLOCK_M, D)
5 |   q = load(Q_shard, [tid_z, tid_m_local])
6 |   for tid_n in range(S/BLOCK_N):
7 |     consumer_tile_wait([tid_z, tid_n])
8 |     k = load(K, [tid_z, tid_n])
9 |     v = load(V, [tid_z, tid_n])
10 |    acc = tile_flash_attn(q, k, v, acc) # use flash-attn
11 |    store(O, [tid_z, tid_m_local], acc)
12 |
13 | with comm_stream(): # AllGather Comm with host primitives:
14 |   for r in range(num_ranks):
15 |     for R, L in zip([K_shards, V_shards], [K, V]):
16 |       if r != rank_id:
17 |         rank_copy_data(
18 |           R[r, :, :, :],
19 |           L[:, :, r*S_per_rank:(r+1)*S_per_rank, :])
20 |         rank_notify([r], rank_id)
21 | with compute_stream():
22 |   compute_func() # Call Self-attention kernel
    
```

Figure 6. AG KV + self-attention overlapping Kernel.

6 IMPLEMENTATION

TILELINK is implemented in Python on top of Triton (Tillet et al., 2019b). We extend Triton’s language features by implementing tile-centric primitives at the Python level, while the tile-centric mapping mechanism is realized through Python Abstract Syntax Tree (AST) transformations. The current implementation can be readily adapted to other compiler frameworks such as TVM (Chen et al., 2018) and MLIR (Lattner et al., 2020).

As shown in Figure 7, the compiler takes as input a pure Python program combining TILELINK’s primitives with Triton’s native primitives. A special parameter *BlockChannel* is provided to serve as the tile-centric mapping context for

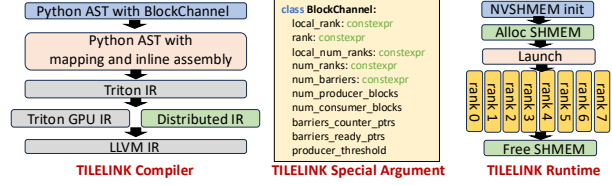


Figure 7. Compilation and Runtime of TILELINK.

computation and communication. The *BlockChannel* parameter encapsulates distributed mapping metadata including current process rank, total world size, synchronization barrier configurations, and producer/consumer block relationships. The Python program is parsed into an AST and translated into Triton IR. During translation, the *BlockChannel* parameter is decomposed to construct the tile-centric mapping using embedded metadata. TILELINK’s primitives are converted into Triton’s *ElementwiseInlineAsmOp*. The Triton IR is then lowered to both Triton GPU IR and a new Distributed IR introduced by TILELINK. This Distributed IR is used to translate the special instructions expressed via *ElementwiseInlineAsmOp* into LLVM IR, which is further compiled into PTX for NVIDIA GPUs. Support for additional backend architectures can be achieved by translating the LLVM IR into target-specific low-level assembly. At runtime, NVSHMEM (NVIDIA, 2025) is used to initialize the distributed execution environment and allocate shared memory. The generated code is launched across all processes to perform concurrent computation and communication, followed by proper shared memory deallocation after completion.

7 EVALUATION

7.1 Experiment Setup

In the evaluation, we use three benchmarks: MLP layer, MoE layer, and self-attention. The input shapes for these layers are listed in Table 4. We use input configurations derived from real workloads such as LLaMA (Dubey et al., 2024), Gemma (Rivière et al., 2024), and Qwen (Yang et al., 2024). We use Async-TP PyTorch as the baseline for the decomposition method (Centauri (Chen et al., 2024) and Dist-Einsum (Wang et al., 2023) are not publicly available), FLUX (Chang et al., 2024) as the baseline for the fusion technique (CoCoNet (Jangda et al., 2022) is available but its source code has been deprecated), and cuBLAS+NCCL as the baseline for non-overlap. We use consistent parallel configurations for all the baselines.

7.2 Single Layer Performance

MLP Layer: the MLP layer is composed of two parts, the first part is mainly composed of AllGather + GEMM, the second part is mainly composed of GEMM + ReduceScatter, there is one activation layer (e.g., SiLUMul or GeLUMul)

Table 4. Benchmark Shapes. S is sequence length, H is hidden dimension length, I is intermediate size, E is number of experts.

Configurations of MLP					
Name	S	H	I	Source Model	
MLP-1	8192	4096	11008	LLaMA-7B	
MLP-2	8192	4096	14336	LLaMA-3.1-8B	
MLP-3	8192	3584	14336	Gemma-2-9B	
MLP-4	8192	4608	36864	Gemma-2-27B	
MLP-5	8192	8192	28672	LLaMA-3.1-70B	
MLP-6	8192	8192	29568	Qwen-2-72B	
Configuration of MoE					
Name	S	H	I	E	topk
MoE-1	8192	2048	1536	8	2
MoE-2	8192	2048	1536	32	2
MoE-3	8192	2048	1536	32	5
MoE-4	8192	4096	2048	8	2
MoE-5	8192	4096	2048	32	2
MoE-6	8192	4096	2048	32	5
Configuration of self-attention					
Name	heads	head dim	sequence length choices		
Attn-1	32	128	16k, 32k, 64k, 128k		
Attn-2	64	128	16k, 32k, 64k, 128k		

between these two parts. We evaluate the two parts separately and also evaluate the full performance of the MLP layer. The results on $8 \times H800$ cluster are shown in Figure 8. For AG + GEMM, Async-TP PyTorch cannot produce a speedup because the decomposed GEMMs are too small to fully utilize the device. Also, according to our tracing results, Async-TP PyTorch uses too many host-driven synchronizations and thus incurs non-negligible overhead to the overlapped kernel. FLUX achieves the highest speedup ($1.34 \times$ over cuBLAS+NCCL) due to its highly optimized implementation. TILELINK also achieves a speedup over cuBLAS+NCCL ($1.27 \times$), reaching 94.5% of FLUX’s performance. Note that TILELINK only requires hundreds of lines of Python code, while FLUX requires thousands of lines of CUDA code. The overlapped kernel generated by TILELINK maps AllGather to the DMA engine.

As for GEMM + ReduceScatter, TILELINK gives the best performance: $1.25 \times$ over cuBLAS+NCCL, $2.22 \times$ over Async-TP PyTorch, and $1.28 \times$ over FLUX. TILELINK decouples the design space of GEMM and ReduceScatter, enabling each part to find their best optimizations, while FLUX uses a tightly coupled fusion kernel for this case, which performs sub-optimally in evaluation. The overlapped kernel generated by TILELINK maps the ReduceScatter to both DMA engine and SMs (streaming multiprocessors), which is a hybrid resource mapping: scatter is done using DMA, and reduction is done on SMs. Combining both parts with intermediate activation, TILELINK achieves performance comparable to FLUX (101.4%) and a $1.24 \times$ speedup over cuBLAS+NCCL. These results show that TILELINK can achieve performance comparable to state-of-the-art fusion libraries with significantly less code (as pointed out in the

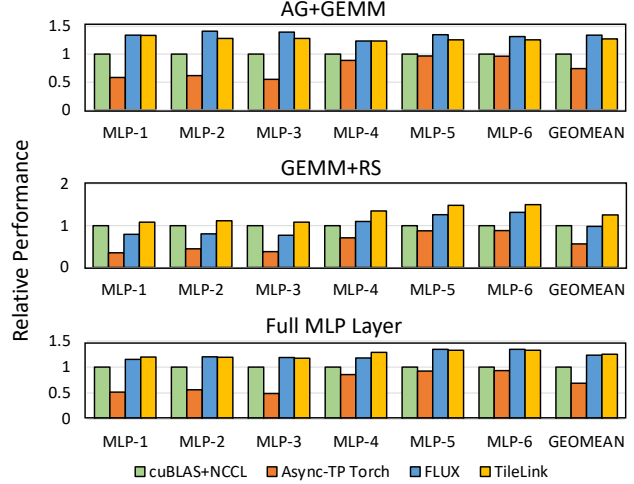


Figure 8. Performance Results of MLP layers (AG+GEMM and GEMM+RS) on $8 \times H800$.

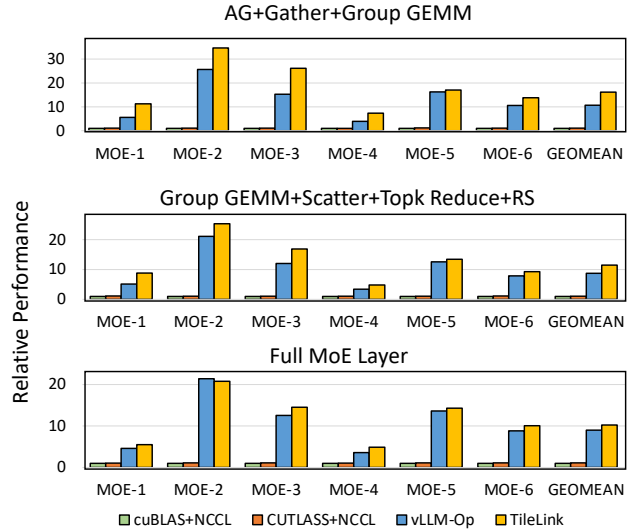


Figure 9. Performance Results of MoE layers (AG + Gather + GroupGEMM and GroupGEMM + Scatter + Reduce + RS) on $8 \times H800$.

motivational example of this paper).

MoE Layer: The MoE layer is much more complex than MLP layers and requires dynamic mapping during compilation. The MoE layer can also be divided into two parts: AG + Gather + Group GEMM and Group GEMM + Scatter + Topk Reduce + RS. There is a Gather operator in the first part and a Scatter + Topk Reduce operator in the second part because the dynamic routing shuffles tokens to

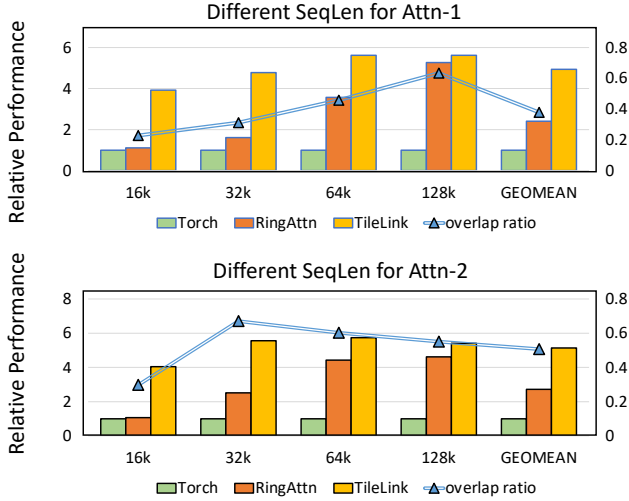


Figure 10. Performance Results and Overlap Ratio of Self-attention Layers on 8xH800.

different experts. These two operators can be fused into Group GEMM kernels. vLLM (Kwon et al., 2023) provides implementations for such fused Group GEMM operations.

For the first part, Figure 9 shows the evaluation results. The cuBLAS and CUTLASS baseline implementations do not fuse the gather and scatter operations into Group GEMM, resulting in a performance bottleneck. The results from vLLM show that such fusion can improve performance by 9.82x. TILELINK achieves even better performance than vLLM (an average of 1.51x improvement) because, in addition to the gather-scatter fusion, TILELINK also overlaps communication with computation. In the code generated by TILELINK, AllGather is mapped to the DMA engine.

For the second part, TILELINK achieves an average speedup of 1.31x over vLLM and 10.56x speedup over CUTLASS+NCCL. This part of MoE has two epilogues: Topk Reduce and RS. TILELINK overlaps three kernels using the tile-centric primitives, demonstrating that the primitives are versatile enough to create extended producer-consumer chains in practice. TILELINK maps Topk Reduce to SMs, and maps ReduceScatter to both the DMA engine and SMs. The full MoE performance is also shown in Figure 9, on average, TILELINK achieves a 1.14x speedup over vLLM. The maximal speedup over cuBLAS+NCCL is 20.76x. Note that existing libraries such as FLUX and Async-TP PyTorch do not support overlapping MoE layers. TILELINK supports MoE thanks to its flexible primitives and dynamic mappings.

Self-Attention Layer: Self-attention is composed of two batch GEMMs and one softmax, which are often fused

together using Flash-Attention techniques (Dao, 2024). Sequence-parallel self-attention consists of an AllGather component and a self-attention computation component. We first implement Flash-Attention in TILELINK on Hopper GPUs and then use TILELINK primitives to overlap AllGather and Flash-Attention. The performance results are shown in Figure 10. We test self-attention with different sequence lengths, from 16k to 128k, covering both short and long contexts. TILELINK shows consistent speedups over both the PyTorch non-overlap implementation (Torch) and RingAttention (Liu et al., 2023) (RingAttn) across all the sequence lengths. On average, TILELINK achieves a 5.04x speedup over Torch and a 1.97x speedup over RingAttn.

We also plot the overlap ratio for self-attention, where overlap ratio is defined as

$$ratio = \frac{comp_only_time + comm_only_time - overlap_time}{comm_only_time}$$

Overlap ratio can be used to measure how much communication overhead is hidden after overlapping. The results in Figure 10 shows that TILELINK can effectively overlap 43.9% communication overhead on average.

7.3 End-to-End Evaluation

We integrate TILELINK into PyTorch and evaluate end-to-end performance for 8 different LLMs on H800 clusters. We first evaluate the performance on a single node with 8xH800 GPUs. The results are shown in the left part of Figure 11. The first five LLMs are dense models, while the other three models are MoE models. Qwen1.5 uses shared experts in MoE, we combine MLP layer and MoE layer together to support shared experts. We use batch size 4 and sequence length 8192. The results show that on average, TILELINK achieves a 1.32x speedup over the PyTorch baselines. The average speedup of dense models is 1.20x, which aligns well with the speedup of single layer MLP. Although TILELINK achieves good speedups for self-attention, MLP layers dominate the performance of end-to-end evaluation (note that there are also large MLP layers before and after self-attention layer). The average speedup of MoE models is 1.54x, which is lower than the speedup of a single MoE layer. In MoE models, MLP layers and MoE layers each occupy about 50% of the total execution time, so the final speedup lies between the speedup of MLP and the MoE.

We also deploy TILELINK for multi-node evaluation. Tensor parallel is often used within one node due to the low inter-node bandwidth. So we use data parallel between two nodes and use tensor parallel in each node. The results on two nodes with 8xH800 GPUs show similar outcomes to those on a single node, as expected. We double the batch size for this evaluation. The overall speedup is 1.29x, which is slightly lower than a single node due to additional

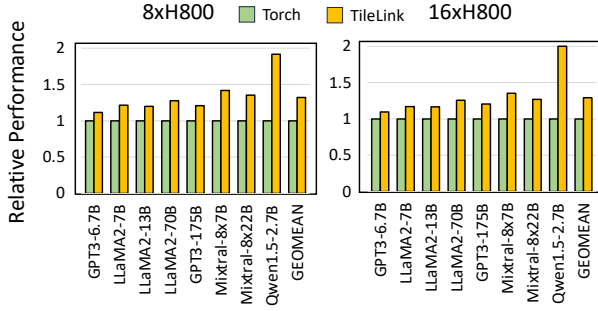


Figure 11. Performance Results of End-to-end Models on $8\times H800$ and $16\times H800$.

communication overhead between two nodes.

7.4 Discussion

Support for model-level communication: TILELINK can be extended to support model-level parallelism (e.g., pipeline parallelism). To achieve this, we can integrate NVSHMEM functionalities into TILELINK’s *tile_push_data* primitive and follow the same compilation techniques as TILELINK. We leave this for future work.

Support multiple backends: Currently, TILELINK targets only NVIDIA GPUs. To support more hardware, we can extend the low-level compilers (e.g., TVM, which supports more hardware than Triton), while keeping the primitives and compilation techniques of TILELINK unchanged.

8 RELATED WORK

Compute-communication overlapping has been studied for years. Early work focuses on CPU clusters with MPI programming model (Goumas et al., 2001; Lu et al., 2015; Marjanovic et al., 2010; Subramoni et al., 2017). With the fast advancement of LLMs, overlapping computation and communication on AI accelerators such as GPUs and NPUs has been proposed (Jangda et al., 2022; Wang et al., 2023; Chang et al., 2024; Punniyamurthy et al., 2023; Chen et al., 2024; Ansel et al., 2024).

Decomposition-based overlapping focuses on splitting operators into smaller ones and overlapping them by rearranging asynchronous pipelines. Dist-Einsum (Wang et al., 2023) implements overlapping kernels for MLP layers on Google TPUs; Async-TP PyTorch (Liang et al., 2024) provides implementations of overlapped AllGather GEMM and GEMM ReduceScatter; Centauri (Chen et al., 2024) systematically explore the three-level design space composed of model, layer, and operation overlapping. Decomposition-based method enables fast development and good compati-

bility with existing frameworks.

Fusion-based overlapping uses kernel fusion techniques to fuse computation kernel with communication kernel. CoCoNet (Jangda et al., 2022) first proposes to fuse CUTLASS GEMM with NCCL kernels and produces state-of-the-art performance on V100 GPUs; FLUX (Chang et al., 2024) follows the idea of CoCoNet and implements high-performance overlapped kernels on A100 and H800 GPUs; another fusion library from AMD (Punniyamurthy et al., 2023) implements various overlapped kernels for DLRM and LLM on AMD GPUs. These studies require a long time to develop due to the lack of high-level programmable primitives. Compared to them, TILELINK provides flexible primitives and achieves comparable performance.

Overlapping compilers use compilation techniques to generate efficient overlapped kernels. CoCoNet compiles high-level operators and schedules into invocations of low-level CUTLASS GEMM and NCCL kernels. Dist-Einsum compiles DNN graphs to device code by decomposing original large operators into small operators and inserting synchronizations among them. These compilers provide little or no programming control for optimization choices such as tile sizes, tile orders, and resource bindings. On the other hand, code generation compilers (Chen et al., 2018; Tillet et al., 2019b; Lattner et al., 2020) and auto-tuners (Zheng et al., 2020b;a; 2022; Feng et al., 2023) provide mature code generation support for single device. Recent work from AMD (Punniyamurthy et al., 2023) also use Triton to generated overlapping kernels. Pallas (Google, 2025) is a distributed compiler that generates Triton code through compilation and supports computation-communication overlapping. However, the overlapping feature is currently only available on Google TPUs, not GPUs. TILELINK provides a set of tile-centric primitives and automatically compiles them into device code using tile-centric mappings, supporting a wide range of workloads.

9 CONCLUSION

To deploy large DNN models on distributed systems, overlapping communication and computation is of vital importance. Previous overlapping studies either bring sub-optimal performance or have difficulty in developing high-performance kernels. In this paper, we propose TILELINK to generate high-performance overlapped kernels. TILELINK uses a set of tile-centric primitives to enhance productivity and uses tile-centric mappings to generate low-level code. In experiments, TILELINK achieves from $1.17\times$ to $20.76\times$ speedups over non-overlapping baselines and comparable performance to state-of-the-art overlapping libraries.

REFERENCES

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In Keeton, K. and Roscoe, T. (eds.), *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 265–283. USENIX Association, 2016. URL <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>.
- Ansel, J., Yang, E. Z., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C. K., Maher, B., Pan, Y., Puhersch, C., Reso, M., Saroufim, M., Siraichi, M. Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., and Chintala, S. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In Gupta, R., Abu-Ghazaleh, N. B., Musuvathi, M., and Tsafirir, D. (eds.), *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pp. 929–947. ACM, 2024. doi: 10.1145/3620665.3640366. URL <https://doi.org/10.1145/3620665.3640366>.
- Ataallah, K., Shen, X., Abdelrahman, E., Sleiman, E., Zhu, D., Ding, J., and Elhoseiny, M. Minigpt4-video: Advancing multimodal llms for video understanding with interleaved visual-textual tokens. *CoRR*, abs/2404.03413, 2024. doi: 10.48550/ARXIV.2404.03413. URL <https://doi.org/10.48550/arXiv.2404.03413>.
- Bai, J., Bai, S., Yang, S., Wang, S., Tan, S., Wang, P., Lin, J., Zhou, C., and Zhou, J. Qwen-vl: A frontier large vision-language model with versatile abilities. *CoRR*, abs/2308.12966, 2023. doi: 10.48550/ARXIV.2308.12966. URL <https://doi.org/10.48550/arXiv.2308.12966>.
- Chang, L., Bao, W., Hou, Q., Jiang, C., Zheng, N., Zhong, Y., Zhang, X., Song, Z., Jiang, Z., Lin, H., Jin, X., and Liu, X. FLUX: fast software-based communication overlap on gpus through kernel fusion. *CoRR*, abs/2406.06858, 2024. doi: 10.48550/ARXIV.2406.06858. URL <https://doi.org/10.48550/arXiv.2406.06858>.
- Chen, C., Li, X., Zhu, Q., Duan, J., Sun, P., Zhang, X., and Yang, C. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In Gupta, R., Abu-Ghazaleh, N. B., Musuvathi, M., and Tsafirir, D. (eds.), *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pp. 178–191. ACM, 2024. doi: 10.1145/3620666.3651379. URL <https://doi.org/10.1145/3620666.3651379>.
- Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: an automated end-to-end optimizing compiler for deep learning. In Arpaci-Dusseau, A. C. and Voelker, G. (eds.), *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pp. 578–594. USENIX Association, 2018.
- Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL <https://openreview.net/forum?id=mZn2Xyh9Ec>.
- DeepSeek-AI, Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Deng, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Xu, H., Yang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Chen, J., Yuan, J., Qiu, J., Song, J., Dong, K., Gao, K., Guan, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Pan, R., Xu, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Zheng, S., Wang, T., Pei, T., Yuan, T., Sun, T., Xiao, W. L., Zeng, W., An, W., Liu, W., Liang, W., Gao, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Chen, X., Nie, X., and Sun, X. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *CoRR*, abs/2405.04434, 2024. doi: 10.48550/ARXIV.2405.04434. URL <https://doi.org/10.48550/arXiv.2405.04434>.
- Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A.,

- Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Rozière, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I. M., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., and et al. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024. doi: 10.48550/ARXIV.2407.21783. URL <https://doi.org/10.48550/arXiv.2407.21783>.
- Feng, S., Hou, B., Jin, H., Lin, W., Shao, J., Lai, R., Ye, Z., Zheng, L., Yu, C. H., Yu, Y., and Chen, T. Tensorir: An abstraction for automatic tensorized program optimization. In Aamodt, T. M., Jerger, N. D. E., and Swift, M. M. (eds.), *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pp. 804–817. ACM, 2023. doi: 10.1145/3575693.3576933. URL <https://doi.org/10.1145/3575693.3576933>.
- Google. Pallas, 2025. URL <https://docs.jax.dev/en/latest/pallas/index.html>.
- Goumas, G. I., Sotiropoulos, A., and Koziris, N. Minimizing completion time for loop tiling with computation and communication overlapping. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, USA, April 23-27, 2001*, pp. 39. IEEE Computer Society, 2001. doi: 10.1109/IPDPS.2001.924976. URL <https://doi.org/10.1109/IPDPS.2001.924976>.
- Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M. X., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d’Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 103–112, 2019. URL <https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html>.
- Jangda, A., Huang, J., Liu, G., Sabet, A. H. N., Maleki, S., Miao, Y., Musuvathi, M., Mytkowicz, T., and Saarikivi, O. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In Falsafi, B., Ferdman, M., Lu, S., and Wenisch, T. F. (eds.), *ASPLOS ’22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pp. 402–416. ACM, 2022. doi: 10.1145/3503222.3507778. URL <https://doi.org/10.1145/3503222.3507778>.
- Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In Flinn, J., Seltzer, M. I., Druschel, P., Kaufmann, A., and Mace, J. (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL <https://doi.org/10.1145/3600006.3613165>.
- Lattner, C., Pienaar, J. A., Amini, M., Bondhugula, U., Riddle, R., Cohen, A., Shpeisman, T., Davis, A., Vasilache, N., and Zinenko, O. MLIR: A compiler infrastructure for the end of moore’s law. *CoRR*, abs/2002.11054, 2020. URL <https://arxiv.org/abs/2002.11054>.
- 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 pre-training, 2024. URL <https://arxiv.org/abs/2410.06511>.
- Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context. *CoRR*, abs/2310.01889, 2023. doi: 10.48550/ARXIV.2310.01889. URL <https://doi.org/10.48550/arXiv.2310.01889>.
- Lu, H., Seo, S., and Balaji, P. MPI+ULT: overlapping communication and computation with user-level threads. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICSS 2015, New York, NY, USA, August 24-26, 2015*, pp. 444–454. IEEE, 2015. doi: 10.1109/

- HPCC-CSS-ICISS.2015.82. URL <https://doi.org/10.1109/HPCC-CSS-ICISS.2015.82>.
- Lu, H., Liu, W., Zhang, B., Wang, B., Dong, K., Liu, B., Sun, J., Ren, T., Li, Z., Yang, H., Sun, Y., Deng, C., Xu, H., Xie, Z., and Ruan, C. Deepseek-vl: Towards real-world vision-language understanding. *CoRR*, abs/2403.05525, 2024. doi: 10.48550/ARXIV.2403.05525. URL <https://doi.org/10.48550/arXiv.2403.05525>.
- Marjanovic, V., Labarta, J., Ayguadé, E., and Valero, M. Overlapping communication and computation by using a hybrid mpi/smpss approach. In Boku, T., Nakashima, H., and Mendelson, A. (eds.), *Proceedings of the 24th International Conference on Supercomputing, 2010, Tsukuba, Ibaraki, Japan, June 2-4, 2010*, pp. 5–16. ACM, 2010. doi: 10.1145/1810085.1810091. URL <https://doi.org/10.1145/1810085.1810091>.
- 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. In de Supinski, B. R., Hall, M. W., and Gamblin, T. (eds.), *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, pp. 58. ACM, 2021. doi: 10.1145/3458817.3476209. URL <https://doi.org/10.1145/3458817.3476209>.
- NVIDIA. cuBLAS, 2022a. URL <https://developer.nvidia.com/cublas>.
- NVIDIA. Transformer Engine, 2022b. URL <https://github.com/NVIDIA/TransformerEngine>.
- NVIDIA. Nvidia collective communications library. <https://developer.nvidia.com/nccl>, 2024.
- NVIDIA. NVSHMEM, 2025. URL <https://docs.nvidia.com/nvshmem/api/using.html>.
- OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/ARXIV.2303.08774. URL <https://doi.org/10.48550/arXiv.2303.08774>.
- OpenAI. Openai o1. <https://openai.com/o1/>, 2024.
- Osama, M., Merrill, D., Cecka, C., Garland, M., and Owens, J. D. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the GPU. In Dehnavi, M. M., Kulkarni, M., and Krishnamoorthy, S. (eds.), *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPOPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, pp. 429–431. ACM, 2023. doi: 10.1145/3572848.3577479. URL <https://doi.org/10.1145/3572848.3577479>.
- Punniyamurthy, K., Hamidouche, K., and Beckmann, B. M. Optimizing distributed ml communication with fused computation-collective operations. *arXiv preprint arXiv:2305.06942*, 2023.
- Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. Learning transferable visual models from natural language supervision. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 8748–8763. PMLR, 2021. URL <http://proceedings.mlr.press/v139/radford21a.html>.
- Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. P. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pp. 519–530, 2013. doi: 10.1145/2491956.2462176. URL <https://doi.org/10.1145/2491956.2462176>.
- Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: memory optimizations toward training trillion parameter models. In Cuicchi, C., Qualters, I., and Kramer, W. T. (eds.), *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, pp. 20. IEEE/ACM, 2020. doi: 10.1109/SC41405.2020.00024. URL <https://doi.org/10.1109/SC41405.2020.00024>.
- Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deep-speed: System optimizations enable training deep learning models with over 100 billion parameters. In Gupta, R., Liu, Y., Tang, J., and Prakash, B. A. (eds.), *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pp. 3505–3506. ACM, 2020. doi: 10.1145/3394486.3406703. URL <https://doi.org/10.1145/3394486.3406703>.
- Rivière, M., Pathak, S., Sessa, P. G., Hardin, C., Bhupatiraju, S., Hussenot, L., Mesnard, T., Shahriari, B., Ramé, A., Ferret, J., Liu, P., Tafti, P., Friesen, A., Casbon, M., Ramos, S., Kumar, R., Lan, C. L., Jerome, S., Tsitsulin, A., Vieillard, N., Stanczyk, P., Girgin, S., Momchev, N., Hoffman, M., Thakoor, S., Grill, J., Neyshabur, B., Bachem, O., Walton, A., Severyn, A., Parrish, A., Ahmad,

- A., Hutchison, A., Abdagic, A., Carl, A., Shen, A., Brock, A., Coenen, A., Laforge, A., Paterson, A., Bastian, B., Piot, B., Wu, B., Royal, B., Chen, C., Kumar, C., Perry, C., Welty, C., Choquette-Choo, C. A., Sinopalnikov, D., Weinberger, D., Vijaykumar, D., Rogozinska, D., Herbison, D., Bandy, E., Wang, E., Noland, E., Moreira, E., Senter, E., Eltyshv, E., Visin, F., Rasskin, G., Wei, G., Cameron, G., Martins, G., Hashemi, H., Klimczak-Plucinska, H., Batra, H., Dhand, H., Nardini, I., Mein, J., Zhou, J., Svensson, J., Stanway, J., Chan, J., Zhou, J. P., Carrasqueira, J., Iljazi, J., Becker, J., Fernandez, J., van Amersfoort, J., Gordon, J., Lipschultz, J., Newlan, J., Ji, J., Mohamed, K., Badola, K., Black, K., Millican, K., McDonell, K., Nguyen, K., Sodhia, K., Greene, K., Sjöstrand, L. L., Usui, L., Sifre, L., Heuermann, L., Lago, L., and McNealus, L. Gemma 2: Improving open language models at a practical size. *CoRR*, abs/2408.00118, 2024. doi: 10.48550/ARXIV.2408.00118. URL <https://doi.org/10.48550/arXiv.2408.00118>.
- Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. doi: 10.48550/ARXIV.2402.03300. URL <https://doi.org/10.48550/arXiv.2402.03300>.
- Subramoni, H., Chakraborty, S., and Panda, D. K. Designing dynamic and adaptive MPI point-to-point communication protocols for efficient overlap of computation and communication. In Kunkel, J. M., Yokota, R., Balaji, P., and Keyes, D. E. (eds.), *High Performance Computing - 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18-22, 2017, Proceedings*, volume 10266 of *Lecture Notes in Computer Science*, pp. 334–354. Springer, 2017. doi: 10.1007/978-3-319-58667-0_18. URL https://doi.org/10.1007/978-3-319-58667-0_18.
- Tillet, P., Kung, H., and Cox, D. D. Triton: an intermediate language and compiler for tiled neural network computations. In Mattson, T., Muzahid, A., and Solar-Lezama, A. (eds.), *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pp. 10–19. ACM, 2019a. doi: 10.1145/3315508.3329973. URL <https://doi.org/10.1145/3315508.3329973>.
- Tillet, P., Kung, H., and Cox, D. D. Triton: an intermediate language and compiler for tiled neural network computations. In Mattson, T., Muzahid, A., and Solar-Lezama, A. (eds.), *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pp. 10–19. ACM, 2019b. doi: 10.1145/3315508.3329973. URL <https://doi.org/10.1145/3315508.3329973>.
- Wang, S., Wei, J., Sabne, A., Davis, A., Ilbeyi, B., Hechtman, B., Chen, D., Murthy, K. S., Maggioni, M., Zhang, Q., Kumar, S., Guo, T., Xu, Y., and Zhou, Z. Overlap communication with dependent computation via decomposition in large deep learning models. In Aamodt, T. M., Jerger, N. D. E., and Swift, M. M. (eds.), *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pp. 93–106. ACM, 2023. doi: 10.1145/3567955.3567959. URL <https://doi.org/10.1145/3567955.3567959>.
- Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., Dong, G., Wei, H., Lin, H., Tang, J., Wang, J., Yang, J., Tu, J., Zhang, J., Ma, J., Yang, J., Xu, J., Zhou, J., Bai, J., He, J., Lin, J., Dang, K., Lu, K., Chen, K., Yang, K., Li, M., Xue, M., Ni, N., Zhang, P., Wang, P., Peng, R., Men, R., Gao, R., Lin, R., Wang, S., Bai, S., Tan, S., Zhu, T., Li, T., Liu, T., Ge, W., Deng, X., Zhou, X., Ren, X., Zhang, X., Wei, X., Ren, X., Liu, X., Fan, Y., Yao, Y., Zhang, Y., Wan, Y., Chu, Y., Liu, Y., Cui, Z., Zhang, Z., Guo, Z., and Fan, Z. Qwen2 technical report. *CoRR*, abs/2407.10671, 2024. doi: 10.48550/ARXIV.2407.10671. URL <https://doi.org/10.48550/arXiv.2407.10671>.
- Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pp. 863–879. USENIX Association, 2020a. URL <https://www.usenix.org/conference/osdi20/presentation/zheng>.
- Zheng, S., Liang, Y., Wang, S., Chen, R., and Sheng, K. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In Larus, J. R., Ceze, L., and Strauss, K. (eds.), *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, pp. 859–873. ACM, 2020b. doi: 10.1145/3373376.3378508. URL <https://doi.org/10.1145/3373376.3378508>.
- Zheng, S., Chen, R., Wei, A., Jin, Y., Han, Q., Lu, L., Wu, B., Li, X., Yan, S., and Liang, Y. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In Salapura, V., Zahran, M., Chong, F., and Tang, L. (eds.),

ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022, pp. 874–887. ACM, 2022. doi: 10.1145/3470496.3527440. URL <https://doi.org/10.1145/3470496.3527440>.