DITRONGEN: COMPILER–LLM COORDINATION FOR ADAPTIVE KERNEL GENERATION AND COMPUTATION–COMMUNICATION OVERLAP

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimizing training and inference of large artificial intelligence models on distributed platforms is challenging because inter-device communication can limit efficiency of computation. High-level frameworks typically optimize computation and communication separately, while manually implementing overlapped kernels demands substantial expertise. We present DitronGen, a framework that coordinates a compiler and an LLM agent for adaptive kernel generation and computation–communication overlap. Starting from single-device programs with user-specified tensor placements, the compiler infers communication requirements and adapts computation and communication implementations to hardware architecture, interconnect topology, and communication dimensions. The LLM agent generates synchronization code using Triton-distributed primitives, adapting synchronization operations and their placement to the data dependencies and control flow of the generated kernels to enable tile-level overlap across different communication scenarios. This division of responsibilities coordinates compiler-based generation of computation and communication kernels with LLM-based generation confined to synchronization. DitronGen can be used as a custom backend for `torch.compile`. On NVIDIA Hopper GPUs, it achieves speedups of 1.13×–3.70×, 4.07×–15.3×, and 2.55×–5.59× over Torch Inductor for GEMM, Group GEMM, and Attention, respectively, and up to 1.14× and 5.22× for dense MLP and MoE layers.

1 INTRODUCTION

Distributed execution provides high computation and memory capacity needed by large AI models. Existing high-level programming frameworks for AI applications often separately program computation and communication, optimizing and generating code for them independently. However, dependencies between faster intra-device computation and slower inter-device communication can leave computing units waiting for data, reducing resource utilization and limiting overall performance for training and inference. Computation–communication overlap mitigates this problem by allowing certain computation and communication tasks to proceed concurrently. In particular, tile-level overlap allows computation to consume available tiles of data before an entire communication operation of all tiles completes, or communication to transfer completed tiles of results while computation continues on other tiles.

Implementing such overlap remains challenging. Previous works (Chang et al., 2024; Zhang et al., 2025) provide specialized overlapping kernels for particular operators and communication patterns, while developing these kernels needs expertise and is time-consuming. Triton-distributed (Zheng et al., 2025) provides distributed data access and synchronization primitives that simplify writing these kernels. But using these primitives still requires developers to construct suitable communication implementations and determine how to synchronize them with computation.

We present DitronGen, a framework that coordinates a compiler and an LLM agent for adaptive kernel generation and computation–communication overlap. It can be used as a custom backend for `torch.compile` (Ansel et al., 2024), allowing developers to retain a high-level computation interface while delegating communication inference and low-level kernel overlap to the framework.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Adaptive generation in DitronGen addresses variation in both kernel implementations and synchronization requirements. Starting from single-device programs with user-specified tensor placements, the framework infers communication requirements, generates communication implementations from predefined pattern descriptions, and adapts computation kernel templates to distributed execution. Hardware architecture, interconnect topology, and data layout guide these implementation choices, including communication along arbitrary tensor dimensions. The resulting kernels can have different producer–consumer dependencies and control flows. These require the framework to adaptively generate synchronization code for overlap.

Compiler–LLM coordination provides a division of responsibilities for this adaptive generation. Communication inference, communication code generation, and computation template adaptation are handled by the compiler using explicit rules and reusable implementations. Synchronization insertion must account for dependencies expressed through the generated kernels. Implementing these variations entirely through compiler transformations would require dedicated analysis and rewriting logic. DitronGen assigns this task to an LLM agent guided by expert-authored synchronization rules. The agent generates synchronization code using Triton-distributed primitives and adapts their insertion points to the compiler-generated kernels.

We make the following contributions in this paper:

- A framework that coordinates compiler-based kernel generation with LLM-assisted synchronization generation, connecting high-level PyTorch programs to distributed execution with tile-level overlap.
- An adaptive kernel generation method that infers communication requirements and generates computation and communication kernels tailored to hardware architecture, interconnect topology, and communication dimensions.
- An automated synchronization generation method that uses Triton-distributed primitives to enable computation–communication overlap across different communication patterns and tensor dimensions.

On NVIDIA Hopper GPUs, DitronGen achieves speedups of 1.13×–3.70×, 4.07×–15.3×, and 2.55×–5.59× over Torch Inductor for GEMM, Group GEMM, and Attention, respectively. It also achieves up to 1.14× and 5.22× speedups for dense MLP and MoE layers.

2 Related Work

Computation–communication overlap. Prior work explores overlap at different levels. Application-level approaches such as DistriFusion (Li et al., 2024) and PipeFusion (Fang et al., 2025) exploit model-specific execution structures to overlap computation and communication. Graph-level approaches, including Centauri (Chen et al., 2024), asynchronous tensor parallelism in PyTorch (PyTorch, 2026), and others (Jiang et al., 2024; Wang et al., 2022), expose overlap through operation decomposition and scheduling, at the cost of additional kernel launches and synchronization. Kernel-level approaches such as FLUX (Chang et al., 2024) and Comet (Zhang et al., 2025) overlap computation and communication at tile granularity through specialized implementations. DitronGen also targets kernel-level overlap, but automatically generates computation, communication, and synchronization code for different distributed execution scenarios.

ML compilers and distributed programming. Prior works (Chen et al., 2018; Denniston et al., 2016; Tillet et al., 2019) provide compilation techniques for generating efficient computation code. DISTAL (Yadav et al., 2022) supports distributed code generation and graph-level computation–communication overlap through a dedicated programming and scheduling interface. Triton-distributed (Zheng et al., 2025) extends tile-level programming with distributed data access and synchronization primitives. DitronGen uses its primitives in an automated generation workflow.

LLMs for kernel generation and optimization. KernelBench (Ouyang et al., 2025) and TritonBench (Li et al., 2025) provide benchmarks for evaluating LLM-based GPU kernel generation. K-search (Cao et al., 2026) and KernelBand (Ran et al., 2026) explore search-based generation and optimization, while QiMeng-GEMM (Zhou et al., 2025) and Geak (Wang et al., 2025) use agent-based

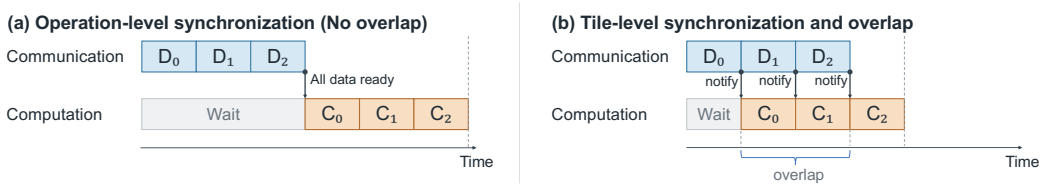

Figure 1: Tile-level computation–communication overlap. (a) Computation waits for the entire communication operation. (b) Tile-level synchronization allows computation to consume ready data while subsequent communication proceeds.

Table 1: Triton-distributed primitives used in this paper.

Primitive	**Explanation**
`rank`	Return the rank of the current device.
`num_ranks`	Return the total number of devices.
`symm_at`	Map a local data pointer to the corresponding pointer on a remote device.
`notify`	Set a signal to notify a consumer.
`wait`	Wait until a signal equals the specified value.
`consume_token`	Use the token returned by `wait` to establish a data dependency.

pipelines. These approaches do not target distributed scenarios. DitronGen explores compiler–LLM coordination for distributed kernel generation.

3 PRELIMINARIES

PyTorch compilation and distributed tensors. The `torch.compile` interface compiles PyTorch functions for optimized execution (Ansel et al., 2024). Torch Dynamo captures computation as FX graphs, which a compilation backend lowers to executable code. Torch Inductor is the default backend. PyTorch distributed tensors associate logical tensors with placements across devices while retaining a single-device-style interface. DitronGen uses the captured graphs and tensor placement metadata as inputs to its custom compilation backend.

Tile-level computation–communication overlap. Figure 1 illustrates overlap through tile-level synchronization. With operation-level synchronization, computation waits for the entire communication operation to complete. With tile-level synchronization, computation can consume ready data tiles while other tiles are transferred. Readiness signals and waits enforce producer–consumer dependencies at tile granularity. The same principle applies when computation produces data for communication.

Triton and Triton-distributed. Triton (Tillet et al., 2019) provides a tile-level programming abstraction and compiler for writing efficient computation kernels. Triton-distributed (Zheng et al., 2025) extends this model with facilities for distributed execution, including rank information, remote data access, and synchronization. These facilities allow developers to express tile-level synchronization needed for computation–communication overlap. Table 1 summarizes the primitives used in this paper.

4 OVERVIEW

Figure 2 presents the DitronGen workflow. It serves as a custom backend for `torch.compile`, taking as input an FX graph captured by Torch Dynamo together with metadata of tensor placement and target hardware configuration. It uses predefined communication pattern descriptions, computation templates, and synchronization skills. At JIT compilation time, the compiler first infers communication requirements from operator semantics and tensor placements, inserting the corresponding communication nodes into the graph. It then generates communication implemen-

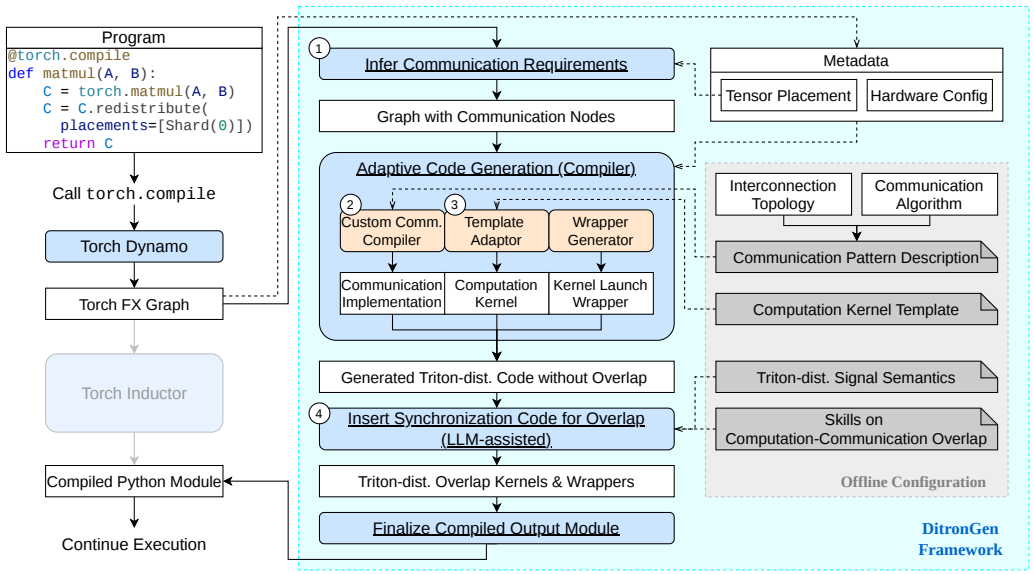

Figure 2: Overview of DitronGen framework.

tations from predefined pattern descriptions and computation kernels from single-device templates, adapting them to the communication requirements, data layout, and target hardware. The wrapper generator also supplies the execution wrappers and buffers needed for synchronization. The compiler-generated code without synchronization is passed to the LLM agent. Guided by expert-authored skills, the agent generates synchronization operations using Triton-distributed primitives and adapts their insertion points to the data dependencies and control flow expressed in the code, enabling tile-level overlap. The resulting kernels and wrappers form the compiled implementation returned to `torch.compile` for subsequent execution.

5 Technique Details

5.1 Inferring Communication Requirements

DitronGen first infers the communication required to execute a computation under the specified tensor placements. For each computation node in the FX graph, the compiler reads the placement metadata of its input and output tensors. When a subsequent `redistribute` node specifies the desired output placement, that target placement is used for communication inference. The compiler then matches the operator semantics and placements against predefined rules to determine the communication operation, the tensor and dimension involved, and whether communication precedes or follows computation. The resulting communication nodes are inserted into the graph for subsequent code generation.

For a tensor with global shape $[M, N]$, $[M_S, N]$ denotes sharding along the first dimension, and $[M, N_S]$ denotes sharding along the second. We write All-Gather(T, d) and Reduce-Scatter(T, d) for communication on tensor T along dimension d, using zero-based dimension indices. Table 2

Table 2: Placement-based communication inference for GEMM.

Case	A	B	C	Communication
1	$[M_S, K]$	$[K, N_S]$	$[M, N_S]$	All-Gather (A, 0)
2	$[M, K_S]$	$[K_S, N]$	$[M_S, N]$	Reduce-Scatter (C, 0)
3	$[M, K_S]$	$[K, N_S]$	$[M, N_S]$	All-Gather (A, 1)
4	$[M_S, K]$	$[K_S, N]$	$[M_S, N]$	All-Gather (B, 0)
5	$[M_S, K]$	$[K, N_S]$	$[M_S, N]$	All-Gather (B, 1)

lists representative inference rules for GEMM ($C = AB$). In Case 4, each device holds a subset of the rows of A spanning the full reduction dimension K, but only a shard of B along dimension K. The compiler therefore inserts All-Gather(B, 0) along the reduction dimension K before GEMM. Gathering B makes the full reduction dimension available to each device, allowing it to compute its assigned output rows and produce the requested placement $[M_S, N]$.

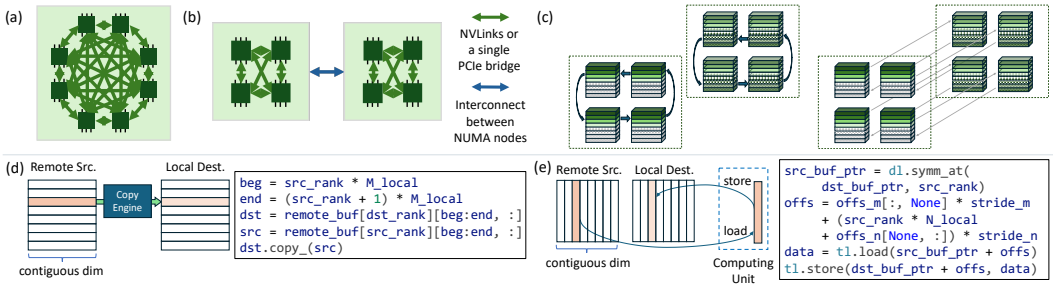

Figure 3: (a) Full-mesh topology. (b) A NUMA topology with two internally fully connected groups. (c) All-Gather on the NUMA topology. (d) Transfer of contiguous dimension uses the copy engine. (e) Transfer of non-contiguous dimension uses a generated Triton kernel.

Algorithm 1 Adaptive Communication Code Generation

Part I: Predefine communication pattern descriptions
Input: Set of supported communication patterns $\mathcal{U}$, set of supported topologies $\mathcal{T}$
Output: Library of communication pattern descriptions $\mathcal{D}$
1: **for all** $(u, t) \in \mathcal{U} \times \mathcal{T}$ **do**
2: Define description $\mathcal{D}[u, t]$ using constructs such as `Pull` and `Push`

Part II: Generate Implementation at JIT compilation
Input: Communication operation o, tensor metadata m, target hardware configuration h
Output: Communication code C
3: $P \leftarrow$ SELECTDESCRIPTION$(\mathcal{D}, o.\text{type}, h)$; $C \leftarrow \emptyset$
4: **for all** constructs s in P in description order **do**
5: **if** s is a `Pull` or `Push` transfer **then**
6: $Dim \leftarrow$ RESOLVETRANSFERDIMENSION(s, m)
7: **if** ISCONTIGUOUS$(Dim, m.\text{strides})$ **then** Emit a DMA transfer using `copy_` into C
8: **else** Emit a Triton kernel using `symm_at` and indexed loads/stores into C
9: **else** Lower s into C

5.2 Implementing Communication Operations

Algorithm 1 summarizes the process of implementing communication operations. We predefine communication pattern descriptions for different operations and interconnect topologies. At JIT compilation time, we select descriptions and generate code according to target hardware configuration and tensor metadata. This allows the compiler to adapt to both topology and tensor layout.

Communication pattern descriptions use constructs including `Pull` and `Push` to specify transfers. Different descriptions encode different algorithms for the same collective according to interconnect topology. For All-Gather on the full-mesh topology in Figure 3(a), a description can specify direct transfers between arbitrary devices. For the NUMA topology in Figure 3(b), which comprises two internally fully connected groups, the description instead specifies the two-stage procedure in Figure 3(c). Devices first gather data within each group through ring transfers. Each device then exchanges the gathered data with a peer in the other group, reducing contention on inter-group link.

Collective communication libraries such as NCCL (Nvidia, 2026) typically only support operations along the first tensor dimension. Applying these operations along other dimensions requires additional splitting and reshaping. DitronGen supports arbitrary communication dimensions, either contiguous or non-contiguous. At JIT compilation time, the compiler resolves each transfer's tensor dimension, and determines whether its data region is contiguous. For contiguous transfers,

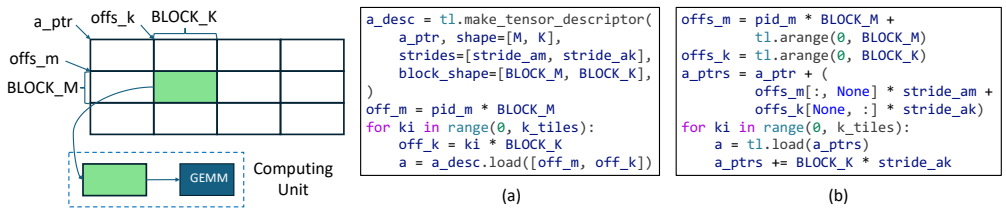

Figure 4: Hardware-adaptive data loading for GEMM. (a) Descriptor-based loading using TMA on Hopper GPUs. (b) Pointer-based loading on Ada Lovelace GPUs.

it emits DMA transfers through the PyTorch `copy_` method, using the GPU copy engine (Figure 3(d)). For non-contiguous transfers, it generates a Triton kernel that uses GPU computing units (Figure 3(e)). The kernel accesses remote memory through Triton-distributed's `symm_at` primitive and uses stride-based indexed loads and stores to transfer the required elements.

5.3 Generating Computation Kernels

For each computation node, DitronGen generates code from a predefined single-device kernel template. These templates support the input shapes required by each device, while a template adapter selects the data-loading implementation according to the GPU architecture. Figure 4 illustrates this hardware adaptation for GEMM. On NVIDIA Hopper GPUs, the adapter selects descriptor-based loading through Triton's `make_tensor_descriptor` API to use the Tensor Memory Accelerator (TMA) for transferring data tiles from global to on-chip memory. On NVIDIA Ada Lovelace GPUs, it selects pointer-based loading using explicit address calculations and `tl.load`. This selection adapts memory access to the target hardware while retaining the computation expressed by the template. DitronGen also provides templates for Group GEMM and self-attention.

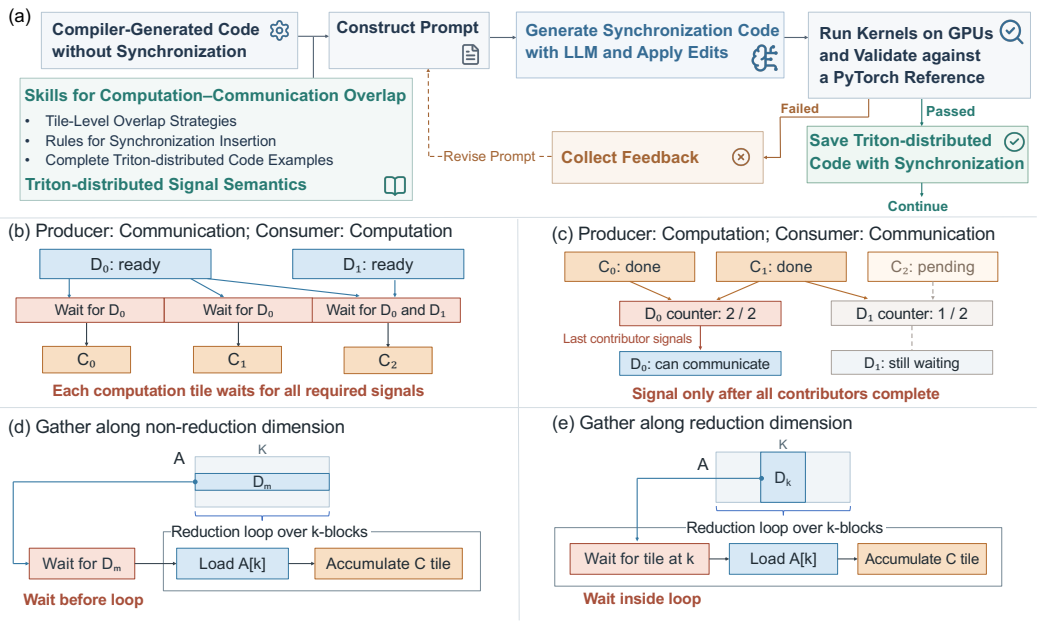

Figure 5: (a) Workflow of our custom agent for generating and inserting synchronization for overlap. (b-c) Different producer–consumer tile dependencies. (d-e) Different wait placements of All-Gather along non-reduction and reduction dimensions.

5.4 Inserting Synchronization for Computation–Communication Overlap

DitronGen inserts synchronization for overlap with an LLM agent. The key challenge is to adapt synchronization to the data dependencies and control flow expressed in the generated code, ensuring that consumers wait for all necessary dependencies. As shown in Figure 5(a), guided by expert-authored skills, the agent constructs a prompt from compiler-generated code. It then generates and inserts synchronization operations, runs the modified kernels on GPUs, and validates their outputs against a PyTorch reference. Successful implementations are saved; failures provide feedback for prompt revision and another generation attempt. Among rules for synchronization insertion in skills, the following two considerations are central.

Producer–consumer tile dependencies. Communication uses large tiles for inter-device bandwidth, while computation uses small tiles that fit into on-chip memory, resulting in different tile partitions. When communication produces data for computation (Figure 5(b)), each computation tile waits for every communication tile supplying its inputs. When computation produces data for communication (Figure 5(c)), each communication tile maintains a completion counter. Completing a computation tile increments the counters of all communication tiles that depend on it, and the last contributor signals readiness. The agent applies these rules to coordinate producers and consumers under the generated tile partitions.

Wait placement for different communication dimensions. Consider All-Gather followed by GEMM, whose computation kernel loops over the reduction dimension. When All-Gather operates along a non-reduction dimension (Figure 5(d)), the required communication tile spans the reduction dimension, so its wait is placed before the loop. When All-Gather operates along the reduction dimension (Figure 5(e)), different iterations of the loop consume different communication tiles; waits therefore occur inside the loop, before the corresponding loads. This placement allows computation to proceed as its inputs become ready, preserving tile-level overlap without waiting for the entire collective.

6 Evaluation

6.1 Experimental Setup

We evaluate DitronGen on kernel-level benchmarks covering GEMM, Group GEMM, and sequence-parallel self-attention, and on layer-level benchmarks covering dense MLP and MoE layers. Experiments primarily use NVIDIA Hopper GPUs with a full-mesh interconnect topology. Selected GEMM cases are also evaluated on NVIDIA Ada Lovelace GPUs with a NUMA topology to assess adaptation across GPU architectures and interconnect topologies. Each platform consists of a single machine with eight GPUs. We use Torch Inductor as the main baseline, as it compiles the same PyTorch programs with distributed tensors. The agent for inserting synchronization uses DeepSeek-V4, with a maximum of five generation attempts. All benchmark timings measure execution only, excluding compilation and agent-side code generation.

Synchronization generation. Across the evaluated cases, the agent produced synchronization code that passed validation within one or two LLM calls, without reaching the retry limit. Agent-side generation took a few seconds to approximately two minutes, depending on the complexity of the input code.

6.2 Kernel-Level Benchmarks

GEMM on Hopper GPUs: We evaluate the GEMM cases in Table 2. For each case, we use shapes derived from open-source LLM workloads, listed in Table 3 (left). For All-Gather and Reduce-Scatter workloads, (M, N, K) in Table 2 is (S, I, H) and (S, H, I), respectively. Figure 6 (left) shows average speedups over Torch Inductor of 3.70× for All-Gather(A, 0) + GEMM, 1.18× for GEMM + Reduce-Scatter, and 1.13×–1.65× for the other three patterns.

We also evaluate each case on 50 randomly selected shapes, with results shown in Figure 6 (right). The red dashed lines show the average speedups. All-Gather(A, 0) + GEMM and GEMM + Reduce-

Table 3: GEMM and Group GEMM Test Shapes. S is sequence length. H is hidden size. I is intermediate size. E is number of experts.

GEMM				**Group GEMM**						
Model	S	H	I	Case #	Model	S	H	I	E	topk
LLaMA-7B	8192	4096	11008	1	–	8192	8192	8192	32	4
LLaMA3.1-8B	8192	4096	14336	2	Qwen1.5-MoE-A2.7B	8192	2048	1408	60	4
LLaMA2-13B	8192	5120	13824	3	Mixtral-8x7B	8192	14336	4096	8	2
LLaMA3.1-70B	8192	8192	28672	4	Qwen3-30B-A3B	8192	2048	768	128	8
Qwen2-72B	8192	8192	29568							

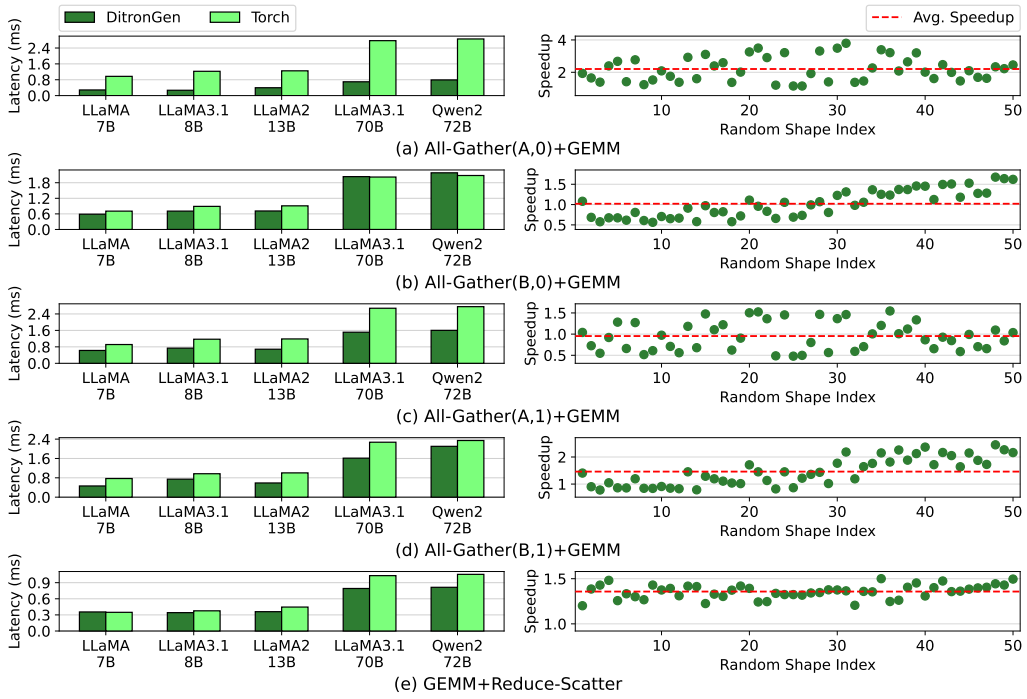

Figure 6: Results of GEMM on Hopper GPUs.

Scatter consistently outperform Torch Inductor, achieving average speedups of 2.20× and 1.36×, respectively. For the other three patterns, some shapes underperform the baseline because DitronGen uses fixed block sizes and pipeline stages without autotuning.

GEMM on Ada Lovelace GPUs: We evaluate three representative cases on Ada Lovelace GPUs: All-Gather(A, 0) + GEMM, All-Gather(A, 1) + GEMM, and GEMM + Reduce-Scatter. The two All-Gather cases use the copy engine and a generated Triton kernel for communication, respectively. As shown in Figure 7, these three cases achieve speedups over Torch Inductor of up to 10×, 6×, and 2.5×, respectively. Together with results on Hopper GPUs, these results demonstrate effective code generation across two platforms with different GPU architectures and interconnect topologies.

Group GEMM: We evaluate two MoE workloads: All-Gather + Gather + Group GEMM and Group GEMM + Scatter + TopK Reduce + Reduce-Scatter. Gather, Scatter, and TopK Reduce are local token-routing operations fused into our single-device Group GEMM templates. DitronGen generates code to overlap these fused computation kernels with All-Gather or Reduce-Scatter. Test shapes are shown in Table 3 (right).

Table 4 reports speedups of 4.07×–15.3× over Torch Inductor. These gains partly reflect the benefit of computation operator fusion: in the evaluated configuration, Torch Inductor uses cuBLAS and NCCL without fusing the local routing operations into Group GEMM, causing a performance bottleneck. In order to analyze the benefit of overlapping, we also compare DitronGen against

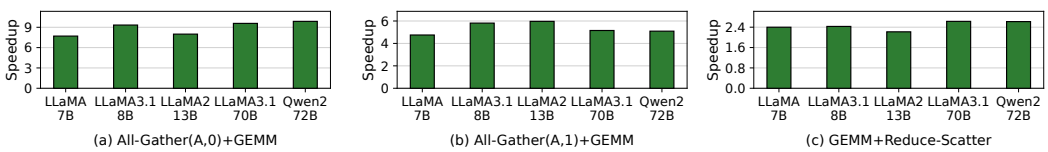

Figure 7: Results of GEMM on Ada Lovelace GPUs.

Table 4: Results of All-Gather + MoE Group GEMM and MoE Group GEMM + Reduce-Scatter.

	All-Gather + MoE Group GEMM Speedups over		**MoE Group GEMM + Reduce-Scatter** Speedups over	
Case #	Torch	Triton-Non-Overlap	Torch	Triton-Non-Overlap
1	4.23×	1.10×	6.34×	1.06×
2	6.07×	1.15×	7.43×	1.13×
3	9.01×	1.21×	4.07×	1.19×
4	9.35×	1.49×	15.3×	1.27×

Triton-Non-Overlap, which combines fused Triton computation kernels with NCCL communication without overlap. DitronGen achieves speedups of 1.06×–1.49× over this baseline.

Sequence-Parallel Self-Attention: We evaluate sequence-parallel self-attention with all-gather on the key and value tensors. We define a PyTorch custom operator to represent self-attention as a single FX node, enabling pattern-based kernel generation in DitronGen. Across the tested sequence lengths, with 32 attention heads and a head dimension of 128, as shown in Figure 8(a), DitronGen achieves an average speedup of 3.99× over Torch Inductor.

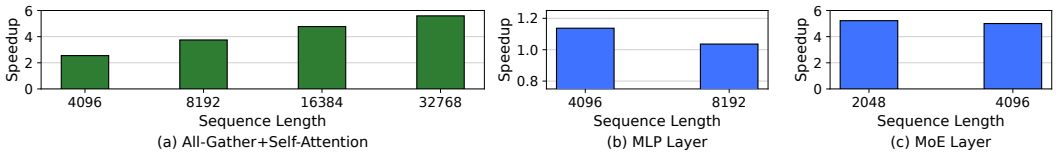

Figure 8: (a) Results of sequence parallel self-attention. (b) Results of MLP Layer. (c) Results of MoE Layer.

6.3 Layer-Level Benchmarks

We evaluate dense MLP and MoE layers in Qwen3 models by replacing their original operators with GEMM kernels for MLP layers and Group GEMM kernels for MoE layers. These kernels are generated offline by DitronGen. For each sequence length, we average speedups over Torch Inductor across models of the same type with different parameter counts. Average speedups range from 1.04× to 1.14× for dense MLP layers (Figure 8(b)) and from 5.00× to 5.22× for MoE layers (Figure 8(c)).

7 Conclusion

We presented DitronGen, a framework that coordinates a compiler and an LLM agent for adaptive kernel generation and computation–communication overlap. The compiler infers communication requirements and generates computation and communication implementations tailored to hardware and communication scenarios. The LLM agent generates synchronization code using Triton-distributed primitives, adapting synchronization operations and their placement to the generated code's dependencies and control flow. As a custom backend for `torch.compile`, DitronGen supports single-device programs with user-specified tensor placements. Future work may extend the framework to overlap multiple operators in the computation graph.

AI USE STATEMENT

In this work, an LLM agent is a component of the proposed DitronGen framework. The research method was developed fully by the authors, and generative AI tools were not used to implement the framework itself. During manuscript preparation, we used LLMs to polish writing, provide feedback on the presentation of our research methodology and experiments, and help interpret results. We did not use generative AI tools to help develop theoretical models or conceptual frameworks, propose or refine hypotheses, or assist with translation. Generating synthetic data sets, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, cleaning and reformatting dataset, and supporting qualitative and thematic data analysis are not applicable to this work. We have reviewed all AI-assisted work and verified that the text preserves our intended meaning and that the interpretations are supported by our experimental results. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

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, pp. 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model, 2026. URL https://arxiv.org/abs/2602.19128.

Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. Flux: Fast software-based communication overlap on gpus through kernel fusion, 2024. URL https://arxiv.org/abs/2406.06858.

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*, ASPLOS '24, pp. 178–191, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651379. URL https://doi.org/10.1145/3620666.3651379.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016. doi: 10.1145/2851141.2851157. URL https://doi.org/10.1145/2851141.2851157.

Jiarui Fang, Jinzhe Pan, Aoyu Li, Xibo Sun, and WANG Jiannan. Pipefusion: Patch-level pipeline parallelism for diffusion transformers inference. In *Advances in Neural Information Processing*

Systems, volume 38, Main Conference, pp. 98434–98455. Curran Associates, Inc., 2025. doi: 10.52202/085713-3289.

Chenyu Jiang, Ye Tian, Zhen Jia, Shuai Zheng, Chuan Wu, and Yida Wang. Lancet: Accelerating mixture-of-experts training via whole graph computation-communication overlapping. In *Proceedings of Machine Learning and Systems*, volume 6, pp. 74–86, 2024.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. TritonBench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1183. URL https://aclanthology.org/2025.findings-acl.1183/.

Muyang Li, Tianle Cai, Jiaxin Cao, Qinsheng Zhang, Han Cai, Junjie Bai, Yangqing Jia, Kai Li, and Song Han. Distrifusion: Distributed parallel inference for high-resolution diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7183–7193, June 2024.

Nvidia. Nvidia collective communications library, 2026. URL https://developer.nvidia.com/nccl.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: can llms write efficient gpu kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25, 2025.

PyTorch. Pytorch documentation, 2026. URL https://docs.pytorch.org/.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. Kernelband: Steering LLM-based kernel optimization via hardware-aware multi-armed bandits. In *Forty-third International Conference on Machine Learning*, ICML'26, 2026.

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, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel ai agent & evaluation benchmarks, 2025. URL https://arxiv.org/abs/2507.23194.

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, pp. 93–106, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450399159. doi: 10.1145/3567955.3567959. URL https://doi.org/10.1145/3567955.3567959.

Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Distal: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pp. 286–300, 2022. doi: 10.1145/3519939.3523437. URL https://doi.org/10.1145/3519939.3523437.

Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. In *Proceedings of Machine Learning and Systems*, volume 7. MLSys, 2025.

Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler, 2025. URL https://arxiv.org/abs/2504.19442.

Qirui Zhou, Yuanbo Wen, Ruizhi Chen, Ke Gao, Weiqiang Xiong, Ling Li, Qi Guo, Yanjun Wu, and Yunji Chen. Qimeng-gemm: Automatically generating high-performance matrix multiplication code by exploiting large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 22982–22990, 2025.