DynaFlow: Transparent and Flexible Intra-Device Parallelism via Programmable Operator Scheduling

Yi Pan [1 2] **Yile Gu** [1] **Jinbin Luo** [2] **Yibo Wu** [1] **Ziren Wang** [1] **Hongtao Zhang** [1] **Ziyi Xu** [2] **Shengkai Lin** [1 2]
Baris Kasikci [1] **Stephanie Wang** [1]

Abstract

Intra-device parallelism addresses resource under-utilization in ML inference and training by overlapping the execution of operators with different resource usage. However, its wide adoption is hindered by a fundamental conflict with the static, sequential programming model of existing frameworks. Integrating these strategies requires invasive, model-specific code overhauls, representing an intractable engineering cost. This is further amplified by the high sensitivity of strategies to execution contexts (e.g., workload, model architecture, hardware), forcing developers to implement and maintain multiple specialized solutions. To address this, we propose DynaFlow, a framework that enables the transparent and flexible integration of intra-device parallelism by decoupling the logical model definition from the physical execution schedule. DynaFlow introduces a flexible frontend with annotations for graph partitioning and a programmable interface for defining custom intra-device parallelism strategies. Its efficient backend manages complex control/data-flow asynchronously, uses custom memory management to eliminate copy overheads, and preserves compatibility with optimizations like CUDA Graphs and TorchInductor. We demonstrate that DynaFlow can integrate representative parallelism strategies into 6 state-of-the-art ML systems with minimal code changes, achieving up to a 1.29x throughput improvement. DynaFlow is publicly available at https://github.com/uw-syfi/DynaFlow.

1 Introduction

As modern machine learning (ML) models such as Large Language Models (LLMs) and diffusion models have grown exponentially in scale, they must rely increasingly on techniques such as distributed execution (Abadi et al., 2016; Dean et al., 2012; Li et al., 2014; Huang et al., 2019; Narayanan et al., 2021), sparsity (Fedus et al., 2022; Lepikhin et al., 2020; Zaheer et al., 2020; Zhang et al., 2023; Ge et al., 2023), and long context generation (Beltagy et al., 2020; Peng et al., 2023; Xiao et al., 2023; Tang et al., 2024). Thus, models are shifting from purely compute-bound to a sequence of operators with highly diverse resource requirements; compute-bound operators like matrix multiplications are often interleaved with memory-bound operators like decode attention, or delayed by network-bound communication. This heterogeneity means that at any given moment, some resources in a single device (e.g., compute units, memory, or network bandwidth) remain idle, leading to significant inefficiencies and degrading model inference and training throughput.

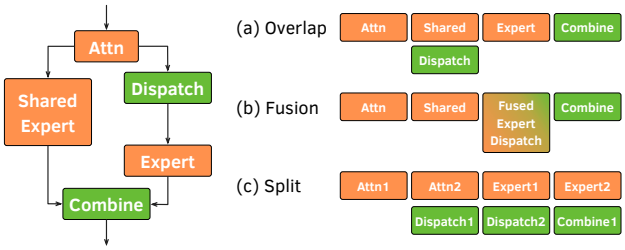


Figure 1. Representative intra-device parallelism strategies: (a) Overlapping computation and communication on different streams; (b) Fine-grained kernel fusion; (c) Splitting the input batch for concurrent execution.

To address this, recent research has explored **intra-device parallelism**, a class of strategies that aims to maximize resource utilization within a single device. Techniques such as *overlapping* computation with communication (Chen et al., 2024; DeepSeek-AI, 2025; Team et al., 2025), fine-grained kernel *fusion* (Gond et al., 2025; Chang et al., 2024; Zhang et al., 2025), or further *splitting* the input batch for concurrent execution (Zhu et al., 2025; Gond et al., 2025; DeepSeek-AI, 2025) have shown significant throughput improvement. By breaking the sequential execution order, these approaches overlap the execution of heterogeneous operations and achieve higher hardware utilization.

Despite their substantial efficiency gains, the wide adop-

[1]University of Washington [2]Shanghai Jiao Tong University. Correspondence to: Yi Pan <conlesspan@sjtu.edu.cn>.

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

tion of these intra-device parallelism strategies has been hindered by a fundamental programming model mismatch. State-of-the-art ML systems like vLLM (Kwon et al., 2023) or SGLang (Zheng et al., 2024) are built upon a sequential programming model with its implied sequential execution order, which conflicts with the non-sequential nature of intra-device parallelism. Consequently, the integration requires invasive code overhauls. For example, implementing dual-batch overlap (DeepSeek-AI, 2025) in SGLang (Team, 2025b) took more than two months and 1.3K lines of specialized code for one model. Replicating this effort across models and strategies imposes prohibitive engineering costs.

Worse yet, this prohibitive engineering effort is further amplified because no single intra-device parallelism strategy is universally optimal. The effectiveness of a strategy varies with the execution context: the model architecture, workload, and hardware (Figure 2). For example, the high-level execution schedule is workload-sensitive: splitting the input batch for overlapping benefits large batch sizes (e.g., LLM inference prefill) but degrades small-batch performance due to higher memory I/O. Hardware can also dictate the optimal choice: on A100 GPUs, partial overlap (all-reduce and RMSNorm) outperforms full overlap (which also overlaps GEMM) due to SM resource contention. Conversely, on H100 GPUs, full overlap yields higher throughput by leveraging `multimem` instructions to offload reduction and free SMs (Ishii & Wells, 2022). This sensitivity forces developers to maintain multiple and more specialized solutions, compounding the engineering burden.

These challenges call for a solution that is both transparent and flexible—one that allows service providers and ML engineers to (1) integrate advanced intra-device parallelism into their existing stack with minimal code changes, and (2) express and adjust this diverse set of strategies to fit specific contexts through a unified abstraction.

Our key idea for solving these challenges is to **decouple operator execution from the model implementation**. Framework developers can retain their existing sequential and imperative programming models. However, rather than adhere to the static, sequential operator order, we introduce a dynamic, programmable execution substrate between the logical model and its physical realization. This substrate enables users to flexibly orchestrate operator execution and define custom, non-sequential execution plans — without altering the model's definition itself.

Achieving this goal presents a core design challenge requiring careful co-design. The flexibility granted by (a) an expressive frontend for defining custom scheduling granularities and execution orders must be reconciled with (b) the need for an efficient backend that can transparently manage the resulting control- and data-flow complexity with negligible overhead, all while remaining fully compatible with other optimizations like CUDA Graphs.

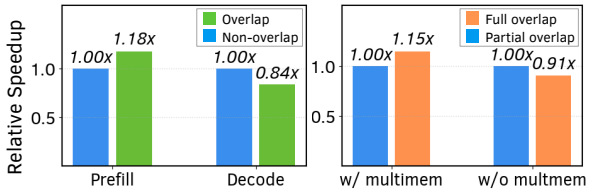


Figure 2. Performance of different intra-device parallelism strategies under different execution contexts on serving Llama-3-70B with 4 GPUs and tensor parallelism.

To address the challenge, we introduce **DynaFlow**, a transparent, flexible, and efficient framework for integrating intra-device parallelism into existing ML systems. DynaFlow enables programmers to easily implement and deploy complex scheduling strategies without modifying core model logic through several key designs as follows: (1) a flexible scheduling frontend that partitions the execution graph into schedulable subgraphs using simple user annotations, providing a unified Python-native interface to dynamically define execution orders; and (2) an efficient execution backend that asynchronously manages complex control- and data-flow dependencies. It incorporates system-level memory management to preallocate intermediate tensors to avoid unnecessary data copies and preserve compatibility with static optimizations like CUDA graphs and TorchInductor by applying them at the subgraph level.

We implement DynaFlow as a `torch.compile` backend, enabling its transparent integration into any PyTorch-based system. To demonstrate its efficiency and flexibility, we use DynaFlow to implement four representative intra-device parallelism strategies, covering the categories in Figure 1, into 6 state-of-the-art ML systems. With only minimal code changes, our approach improves end-to-end throughput by up to 1.29x compared to the original systems. Moreover, DynaFlow matches and even outperforms existing, highly specialized implementations by up to 1.1x. In summary, this work makes the following key contributions:

- We identify a fundamental conflict between the static, sequential programming model of ML frameworks and the needs of intra-device parallelism, proposing to resolve it by decoupling the execution schedule from the model implementation.
- We design and implement DynaFlow, a framework featuring a programmable frontend for defining custom parallelism and an efficient backend that preserves compatibility with low-level optimizations.
- We demonstrate DynaFlow's effectiveness by integrating representative intra-device parallelism strategies into 6 major ML systems, achieving up to a 1.29x throughput improvement with minimal engineering effort.

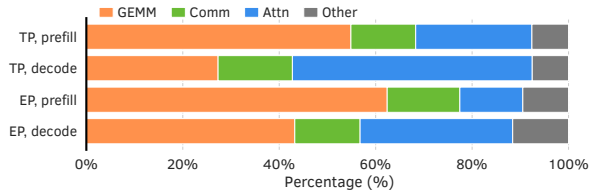


Figure 3. Execution time breakdown of serving a Llama-3-8B model on 2 GPUs with tensor parallelism (TP); (b) a DeepSeek-V2-Lite model on 2 GPUs with expert parallelism (EP). The batch size is 512 and the sequence length is 1024.

2 Background and Motivation

Modern large-scale ML models are composed of a sequence of operators with highly diverse resource requirements. These are broadly categorized by their resource bottlenecks (Figure 3): (1) compute-bound (e.g., general matrix multiplication), (2) memory-bound (e.g., decode-attention and normalization), and (3) network-bound (e.g., all-reduce in tensor parallelism and all-to-all in expert parallelism). A primary inefficiency in ML serving and training stems from the *sequential execution* of these heterogeneous operators, creating stalls on one resource (e.g., compute units) while another (e.g., the network fabric) is active. Figure 3 shows that such stalls on network communication could leave up to 15% performance on the table.

2.1 Intra-Device Parallelism Strategies

To address these inefficiencies, researchers have developed a diverse set of intra-device parallelism strategies. The goal is to maximize the overall resource utilization by overlapping the execution of heterogeneous operations and thereby improving the end-to-end performance.

These strategies take a variety of forms: (1) overlapping heterogeneous operations such as compute-bound GEMMs with network-bound all-to-all (Figure 1a), (2) fusing multiple kernels with different resource usage into a single one (Figure 1b), or (3) splitting an input batch into smaller micro-batches to further break dependencies and create more overlapping opportunities (Figure 1c). By enabling the parallel execution of compute, memory, and network operations, these strategies can, in principle, eliminate most resource stalls and deliver substantial throughput improvements. However, each strategy has its own scheduling and implementation challenges, making the choice of when to apply them nontrivial. We outline these next.

Overlapping. Overlapping can be successfully applied when operators do not depend on each other and require different resources, usually compute vs. communication. For example, in MoE models with shared experts, the shared expert (compute) can be launched in parallel with the dispatch to the top-k experts (communication) (Figure 1a). This pattern is also common in data-parallel training, to launch gradient reduction for later layers in parallel with the backward pass for earlier layers (Abadi et al., 2016; Sergeev & Del Balso, 2018; Li et al., 2020), or to prefetch the next layer's weight shards in parallel with computation (Rajbhandari et al., 2020; Zhao et al., 2023).

There are two challenges in applying overlapping. First is the decision of the overlap mechanism. For example, on NVIDIA GPUs, using CUDA streams is simple but can result in lower end-to-end performance due to resource contention. Meanwhile, CUDA green contexts ensure resource isolation between concurrent kernels but requires the user to specify the number of SMs per context, which can be challenging to tune. Overlapping also increases peak memory usage compared to a sequential execution, as an operator's results must be buffered until the downstream operator's resource is available again.

Fusion. Fusion overlaps heterogeneous operators that have a sequential dependency at fine granularity, within a single custom kernel definition. This is commonly applied to cases where computation is followed by a collective communication, such as a GEMM followed by an all-reduce or reduce-scatter (Figure 1b). Such cases are common in both model-parallel training and inference (Gond et al., 2025; Chang et al., 2024; Zhang et al., 2025; Jangda et al., 2022; Wu et al., 2025; Spector et al., 2025).

Fusion requires the substitution of multiple high-level operators with an expert-designed custom kernel. This can be challenging for framework developers to maintain, as there is a combinatorial number of possible operator subsequences that can be substituted, and many possible implementations. For example, vLLM attempts to mitigate this issue for MoE kernels by exposing an internal pluggable abstraction called FuseMoEModularKernel (vLLM, 2025). However, this is a one-off choice for MoE kernels that does not generalize to vLLM's other operators, and such internal abstractions require careful design to remain compatible with other optimizations such as splitting.

Splitting. An alternative way to overlap sequential heterogeneous operators is to split execution along the batch dimension, so that the resulting *micro-batches* do not depend on each other and can then be overlapped. This strategy is again common in both model-parallel training and inference (Liu et al., 2024; Zhu et al., 2025; DeepSeek-AI, 2025; Wang et al., 2022; Liang et al., 2024). For example, Nanoflow (Zhu et al., 2025) overlaps compute-, memory-, and communication-bound operators for LLM inference, while DualPipe (Liu et al., 2024) combines expert and pipeline parallelism to overlap compute- and communication-bound operators between forwards and

backwards passes.

Splitting introduces the same challenges as overlapping, as well as an additional choice of when and at what granularity to apply batch splitting. Batch splitting requires an additional read of the model weights per additional micro-batch, so naive application can worsen performance (Figure 2a). Also, asymmetric micro-batches can improve performance under heterogeneous operator durations, so control over the micro-batch sizes is critical. Finally, splitting the input batch may be selectively applied to operators; this introduces a challenge of eliminating memory copies from batch splitting and merging.

2.2 Our Approach and Challenges

Despite the clear benefits of intra-device parallelism, these strategies are not widely adopted in popular, high-performance ML systems like vLLM (Kwon et al., 2023) or SGLang (Zheng et al., 2024). Adoption is so far limited to targeted optimizations for specific models or model parallelism strategies. This adoption barrier exists because intra-device parallelism is fundamentally more challenging to integrate than existing parallelization techniques or optimizations. Established parallelisms, for example, typically involve *modular changes* to local operators, such as replacing linear layers for tensor parallelism (TP), and rely on *static configurations*, like a fixed TP degree. Intra-device parallelism strategies, in contrast, require invasive modifications to the model's global execution order. Furthermore, their optimal usage must be decided dynamically at runtime to adapt to the characteristics of each incoming batch. This creates a **dynamicity gap** that static compiler-based approaches, such as XLA, struggle to handle, as they assume a more stable execution graph. This combination of architectural invasiveness and the need for runtime dynamicity makes manual integration intractable.

DynaFlow's goal is to enable efficient, simple, and dynamic choice of intra-device parallelism strategy. We propose to solve this by decoupling operator execution from the model implementation. This approach introduces a new abstraction layer that breaks the coupling between the static, logical definition of a model and its dynamic, physical execution. DynaFlow provides a **transparent** interface for applying intra-device parallelism, while simultaneously offering the **flexibility** required to express a diverse and growing set of dynamic, context-aware schedules.

Although this decoupled approach is powerful, its practical, high-performance realization presents several design challenges to address.

First, **defining the scheduling granularity** is a critical challenge. An overly *fine-grained* granularity, such as individual operators (e.g., `torch.add`, `torch.arange`), creates an unmanageably complex scheduling task for the user, risks high dispatch and scheduling overhead, and can obstruct optimizations like kernel fusion. Conversely, an overly *coarse-grained* granularity, such as entire model layers, is simpler but fails to expose the very intra-layer overlapping opportunities (e.g., between a computation and its following communication) that the system is intended to exploit.

Second, we must design a **general and unified abstraction** for defining the schedule. This interface must be expressive enough to capture the diverse and growing set of intra-device parallelism strategies. Crucially, it must also support *dynamism*, allowing the execution schedule to be adapted at runtime based on the specific context (e.g., workload, hardware, or model architecture).

Finally, this flexibility **must not compromise performance**. Managing the complex *control- and data-flow dependencies* introduced by intra-device parallelism strategies can add high runtime overhead with naive scheduling. Also, *data-flow overheads*, such as the memory copies for splitting and merging different micro-batches across operators, can easily negate any gains from parallelism. Moreover, a non-sequential and dynamic schedule can conflict with essential operator graph optimizations that assume a static graph, such as CUDA Graphs and TorchInductor. Addressing these challenges is the core of our system design, which we present in the following section.

3 DESIGN

3.1 Overview

The overview and the workflow of DynaFlow are depicted in Figure 4. It addresses the challenges outlined in §2.2 through a decoupled architecture. The frontend (§3.2) augments vanilla Python code with a unified, dynamic programming abstraction for expressing diverse parallelism strategies and scheduling granularities, thus addressing the problems of engineering cost and inflexibility. The backend (§3.3) focuses on efficient execution, managing the complex control- and data-flow dependencies introduced by dynamic scheduling, minimizing data-flow overheads like memory copies, and ensuring compatibility with low-level optimizations such as CUDA graphs and TorchInductor.

Integrating DynaFlow into an existing ML system involves three phases. First, during model initialization, a developer uses the frontend's annotation APIs to partition the model's computational graph into schedulable subgraphs. Second, the developer implements a custom scheduling policy within a Python-native scheduler function, using the frontend's high-level APIs to define the desired execution order and overlapping patterns. Third, at runtime, DynaFlow intercepts the model's forward call, invoking this user-defined scheduler. The user scheduler dynamically builds an exe-

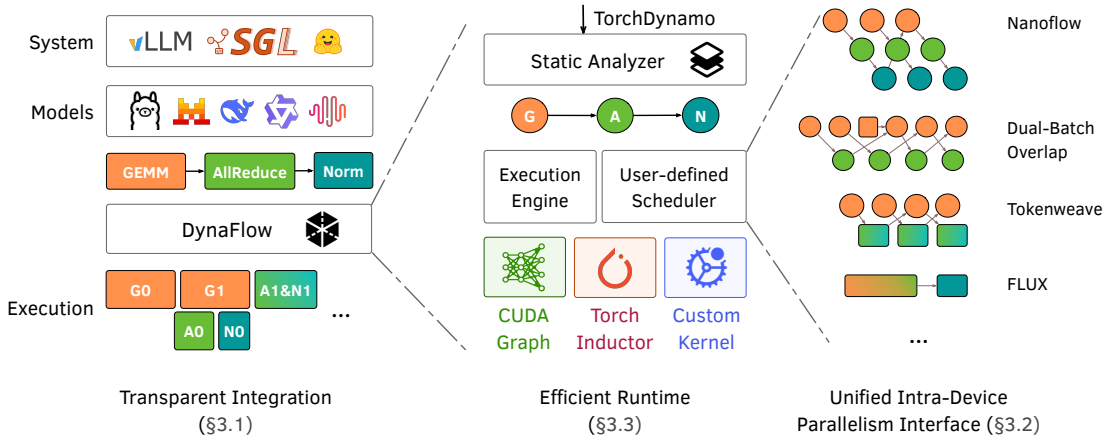


Figure 4. Overview of the DynaFlow system design and workflow.

cution plan and dispatches subgraphs to the asynchronous backend, which manages all low-level execution details.

3.2 Frontend Design

The DynaFlow frontend provides the abstraction for defining and controlling intra-device parallelism. During model initialization, it takes a TorchDynamo-traced computational graph as input. TorchDynamo extracts PyTorch graphs from Python bytecode and is supported by many state-of-the-art ML systems like vLLM and SGLang; this design thus ensures broad compatibility. The frontend then enables developers to implement custom parallelism strategies through the mechanisms detailed in the following subsections.

3.2.1 Graph Partition

The fine-grained TorchDynamo-traced computational graph presents a design challenge of determining the correct scheduling granularity, as discussed in §2.2. Fortunately, we observe that the targets of intra-device parallelism strategies are at the level of logical, coarse-grained operators (e.g., an RMSNorm or an Attention), instead of their constituent tensor arithmetic operations in the fine-grained graph (e.g., sqrt, div, dot). This is because overlapping at too fine-grained granularity with tiny operators provides more overhead than benefit. As a result, these desired logical blocks usually correspond to one of two common patterns: (1) a `nn.Module`, like `torch.nn.RMSNorm`, or (2) a PyTorch API call, usually to invoke a custom kernel implementation, such as `F.scaled_dot_product_attention`.

Therefore, our API is designed to provide annotations that directly target these patterns, allowing developers to reuse their logical code structure as the basis for scheduling. As detailed in Figure 5, DynaFlow provides `SplitModule` to partition the graph at module boundaries and `SplitFunc` to partition around specific function calls. For all other cases where partition boundaries do not align with these two common patterns, `dynaflow.mark` is provided as a Python context manager to wrap any code sections.

```python
# Split on a function call with specific pattern
class SplitFunc:
    pattern: str
# Split on a module of the specific type
class SplitModule:
    target_cls: type[torch.nn.Module]
# Split on a code block
@contextmanager
def mark(tag: str) -> Generator:
    pass
```

Figure 5. APIs for graph partition.

```python
# Initialize parallel execution for N micro-batches
def split(batch_sizes: list[int]):
    pass
# Get operators ready to execute for a micro-batch
def get_ready_ops(ubatch_idx: int) -> list[op]:
    pass
# Dispatch one or more ready operators to execute
def execute(operators: tuple[op], stream=None,
            replace_func=None):
    pass
```

Figure 6. APIs for programmable operator scheduling.

3.2.2 Programmable Scheduling

The DynaFlow frontend provides a unified and dynamic abstraction for scheduling the partitioned subgraphs. Designing this abstraction requires a balance between flexibility and complexity. One design alternative is to expose all subgraph executables to users directly. This would provide maximum control but force the user to manually manage all complex data-flow (e.g., storing and merging intermediate tensors) and control-flow dependencies, which is ineffective to reduce the engineering effort. Meanwhile, a declarative approach such as graph rewriting rules is not flexible enough because it is difficult to adapt to different runtime contexts.

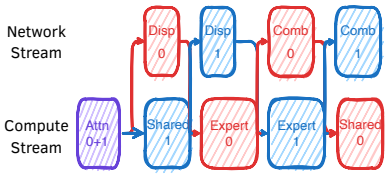
DynaFlow's design strikes a balance: it provides a set of high-level APIs to abstract this complexity, but embeds them

```python
def get_split_rules(self) -> list[MatchingRule]:
    return [
        MatchingRule(
            condition=SplitFunc(r".*_dispatch"),
            tag="disp"
        ),
        MatchingRule(
            condition=SplitFunc(r".*_expert"),
            tag="expert"
        ),
        MatchingRule(
            condition=SplitFunc(
                r".*_combine(?:_with_shared)?"),
            tag="combine"
        ),
        # "shared", "attn" are omitted
    ]
```

(a) DBO Graph Partition

```python
async def schedule(self, batch_size: int):
    if batch_size <= THRESHOLD:
        # Fallback to sequential execution
        pass
    split([bs_1, bs_2])
    while not finished:
        op0, op1 = get_ready_ops(…)
        match (op0.tag, op1.tag):
            case ("attn", "attn"):
                execute((op0, op1))
            case ("disp", "shared"):
                op_c, op_n = op1, op0
            case ("comb", "expert"):
                op_c, op_n = op1, op0
            # Some similar cases omitted
        execute((op_c,), compute_stream)
        execute((op_n,), network_stream)
```

(b) DBO Scheduler



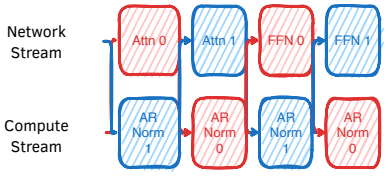
(c) DBO Execution

```python
def get_split_rules(self) -> list[MatchingRule]:
    return [
        # The final one does not output residual
        MatchingRule(
            condition=(
                SplitFunc(r"all_reduce"),
                SplitMod(RMSNorm)
            ),
            hook=_is_final_ar_norm,
            tag="ar_add_rms_norm_final"
        ),
        MatchingRule(
            condition=(
                SplitFunc(r"all_reduce"),
                SplitMod(RMSNorm)
            ),
            tag="ar_add_rms_norm"
        ),
    ]
```

(d) Tokenweave Graph Partition

```python
async def schedule(self, batch_size: int):
    if batch_size <= THRESHOLD:
        # Fallback to sequential execution
        pass
    split([chunk_size_1, chunk_size_2])
    while not finished:
        op0, op1 = get_ready_ops(…)
        if op0.tag.startswith("ar_…"):
            op_c, op_n = op1, op0
        else:
            op_c, op_n = op0, op1
        is_final = op_n.tag.endswith("final")
        execute((op_c,), compute_stream)
        execute(
            (op_n,)),
            network_stream,
            fused_ar_norm if is_final else …
        )
```

(e) Tokenweave Scheduler



(f) Tokenweave Execution

Figure 7. Examples of using DynaFlow's API to define DBO (up) and Tokenweave (down), with the desired execution order.

within a fully Python-native frontend to preserve flexibility. To implement a custom strategy, a developer inherits from a base class, `OpSchedulerBase`, and overrides its `schedule` method. Inside this method, the developer interacts with the DynaFlow backend using a set of high-level APIs (detailed in Figure 6).

The core primitives we provide includes: (1) `split([`$bs_1, bs_2, \cdots, bs_n$`])`, which initializes n logical micro-batches, (2) `get_ready_ops(i)`, which queries the backend for a list of subgraphs whose control-flow dependencies have been met for the `i`-th micro-batch, and (3) `execute(operators, replace_func=...)`, which dispatches one or more ready operators to the backend. This API enables flexible execution patterns. Denote op_x^y as the x-th operator of the y-th batch. `execute(`op_i^0`);` `execute(`op_i^1`)` represents a micro-batched execution of op_i while `execute((`op_i^0`,` op_i^1`))` merges the two micro-batches into a single batch. Similarly, `execute((`op_i^0`,` op_j^1`), kernel)` supports fusion by replacing the execution of the i-th and j-th operators with a custom callable `kernel`. If `execute` is called on different operators without a custom kernel, the system falls back to a sequential execution. As later described in §3.3, the backend remains responsible for managing all data-flow dependencies.

This API provides both expressiveness and dynamism. To demonstrate how this unified interface can capture diverse and complex strategies, we present two representative examples in Figure 7.

Example 1: Splitting and Overlapping with DBO. Our first example, dual-batch overlap (DBO) (Liu et al., 2024), primarily uses splitting and overlapping. The goal is to execute the compute-intensive attention layer as a single large batch while splitting the MoE layer into two micro-batches to overlap its communication and computation. Figure 7(a,b,c) shows how this complex logic can be implemented in DynaFlow with just $\tilde{2}0$ lines of Python. The user first defines the split rules to partition the graph into attention and different operators in MoE. Then, during runtime, the scheduler checks the `batch_size` to dynamically decide whether to apply the optimization. If so, it uses `split()` to create two micro-batches and enters a loop, repeatedly using `get_ready_ops()` and `execute()` to dispatch compute and network operators to different streams according to the DBO pattern. This concise implementation stands in sharp contrast to the invasive rewrites required in prior systems.

Example 2: Splitting and Fusion with TokenWeave. While DBO showcases a common overlapping pattern, DynaFlow's unified API can also compose these techniques with operator fusion. We illustrate this using a

Algorithm 1 Data-Flow and Memory Management

```
function StaticAnalysis(G, M)
    for i in [0, #nano_batches] do
        for all tensor t in G do
            M[i][t].ref_count ← CalculateOutDegree(t, G)
            M[i][t].prealloc ← (t is input to merge point)
        end for
    end for
end function
function RuntimeExecute(op, idx, M)
    for all input M[idx][i] do
        M[idx][i].ref_count -= 1
        inputs[i] ← B[idx] if M[idx][i].prealloc
        // Garbage collection
    end for
    for all output M[idx][j] do
        if M[idx][j].prealloc then
            src ← GetDef(M[idx][j])
            B ← PreallocateBuffer(src)
            // Register runtime hook to replace the tensor
        end if
    end for
    // Execution
end function
```

TokenWeave-style (Gond et al., 2025) strategy, which fuses a communication-bound AllReduce with a memory-bound RMSNorm and overlaps this fused kernel with the next compute-bound operation. Implementing this requires two simple steps. First, the user defines the partitioning rule in the scheduler's configuration, as described in prose below, to group the target operators. Second, the scheduling logic uses the `replace_func` argument to dynamically substitute these operators with a custom fused kernel.

3.3 Backend Design

The DynaFlow backend is responsible for executing the dynamic schedules generated by the frontend, addressing the performance challenges identified in §2.2.

3.3.1 Data-Flow and Memory Management

Dynamic changes in the degree of intra-device parallelism between operators, enabled by the flexible frontend, introduce challenges in data-flow management. When the intra-device parallelism degree (i.e., number of micro-batches) differs between the producing and consuming operators, data must be re-sharded, typically via tensor splitting or concatenation. Naive implementations of these resharding operations incur substantial memory copy overheads (e.g., using `torch.cat`), which can negate performance gains from parallelism. Furthermore, accurately tracking the lifetime of intermediate tensors across these complex, dynamic execution paths is difficult, potentially leading to inefficient memory utilization and higher peak memory usage.

To address these challenges, the DynaFlow backend employs a coordinated data-flow and memory management system, detailed in Algorithm 1. This system operates in two phases. At initialization, the `StaticAnalysis` function traverses the computational graph to pre-compute metadata for each tensor, including its reference count for lifecycle management and a `prealloc` flag to identify tensors that are part of a future merge operation. At runtime, the `RuntimeExecute` function uses this metadata. It decrements reference counts for garbage collection and, for any output flagged with prealloc, it pre-allocates a contiguous buffer. A runtime hook then redirects the operator's output directly into the correct slice of this buffer, enabling zero-copy data resharding for subsequent consumers.

3.3.2 Compatibility with Low-Level Optimizations

Another challenge in dynamic scheduling is ensuring compatibility with other performance optimizations that intercept execution at the operator graph level, such as TorchInductor and CUDA Graphs. TorchInductor performs code generation for operator fusion while CUDA Graphs reduce kernel launch overhead. Both techniques require a static graph. This inherently conflicts with the dynamic, non-sequential schedules generated by DynaFlow.

DynaFlow addresses this conflict by applying static optimization techniques at the *subgraph* level. For TorchInductor, each subgraph is compiled once to obtain a callable that can be reused across different micro-batches. For CUDA graphs, to avoid data collisions during concurrent execution and maintain separate data-flow between micro-batches, we capture separate CUDA graphs for each micro-batch and subgraph. This can result in a large number of CUDA graphs and cause substantial memory usage. To address that, we reuse the CUDA graph pool (PyTorch, 2021) across all subgraphs and batch sizes of the same micro-batch.

At runtime, the dynamic Python scheduler uses the `execute` API to dispatch these preprocessed subgraphs based on the micro-batch size. This approach preserves the performance benefits of other techniques for operator graph execution, while retaining the flexibility of dynamic scheduling.

4 Implementation

We built DynaFlow as a `torch.compile` backend with 4.1K lines of Python code. It can be plugged into any PyTorch-based systems by intercepting the model forward call with this backend. Although some of its features (e.g., the `dynaflow.mark` wrapper) depend on more recent PyTorch versions, all the examples shown above are com-

Category	System	Core	Model	Attn
Serve	vLLM	75	4	0
	SGLang	144	0	45
Train/Infer	Megatron-LM	75	4	0
	Transformers	0	0	0
Visual	xDiT	51	10	0
	FastVideo	30	0	0

Table 1. Engineering cost (measured in lines of code) to integrate DynaFlow into different ML systems.

Category	Strategy	Partition	Scheduler
Split	NanoFlow	8	22
	DBO	16	67
Overlap	SBO	12	24
Fuse	TokenWeave	15	28
	Flux	5	25
	Comet	9	47

Table 2. Engineering cost (measured in lines of code) of using DynaFlow to implement different intra-device parallelism strategies.

patible with PyTorch 2.6+.

Frontend. The DynaFlow frontend acquires the model's computational graph by using the newly introduced TorchDynamo in PyTorch 2 (Ansel et al., 2024). It captures an operator-level computational graph from Python bytecode. This approach ensures broad compatibility with existing PyTorch-based systems like vLLM and SGLang that already support this interface.

Backend. The execution backend is built on an asynchronous, callback-driven architecture to manage the complexities of decoupled scheduling. The backend execution engine maintains the state of the dependency graph. When the frontend scheduler calls `execute`, it enqueues dispatch requests to the engine. The engine consumes these requests, updates the dependency state, and determines which subsequent subgraphs have their logical prerequisites met. This list of ready subgraphs is then enqueued back to the frontend scheduler, which retrieves it via the `get_ready_ops` call. This asynchronous interaction allows the engine to manage complex control-flow dependencies internally, abstracting this complexity from the user's scheduling logic.

5 Evaluation

In this section, we demonstrate the effectiveness of DynaFlow APIs in integrating various intra-device parallelism strategies into state-of-the-art ML systems with minimal code changes and quantify the resulting performance improvements.

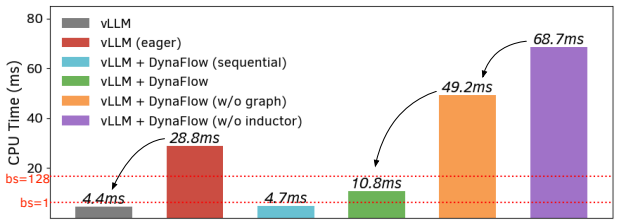


Figure 8. CPU execution time for a single forward pass in vLLM with different DynaFlow configurations.

5.1 Evaluation Setup

Testbeds. We evaluate DynaFlow on (1) a DGX B200 system with 8 NVIDIA B200 GPUs connected by NVLink and (2) an H100 system with 4 NVIDIA H100 GPUs connected by NVLink. We use (2) for experiments related to xDiT, FastVideo, HuggingFace Transformers, Flux, and Comet, due to compatibility issues with Blackwell GPUs. Comet-related experiments use PyTorch 2.6.0 with CUDA 12.4. All other experiments use PyTorch 2.8.0, CUDA 12.8, and NVIDIA driver version 580.

Target Strategies. We implement representative intra-device parallelism strategies across all categories in §2.1. In the evaluation, we measure the performance of split-based micro-batching strategies including NanoFlow (Zhu et al., 2025) and dual-batch overlap (DeepSeek-AI, 2025), communication-overlap strategies, and communication fusion strategies including TokenWeave (Gond et al., 2025), Comet (Chang et al., 2024), and Comet (Zhang et al., 2025).

Target Systems and Baselines. We evaluate DynaFlow on all 6 systems listed in Table 1. Our primary baseline is the unmodified version of each system. We also compare against existing implementations of intra-device parallelism, including (a) the dual-batch overlap implementation in vLLM (vLLM, 2025); and (b) the original TokenWeave (Gond et al., 2025) framework, as well as some naive implementations, including (c) our pull request that implements NanoFlow on top of vLLM collaborated with both teams, which exhibits a fixed threshold for batch splitting (Team, 2025a); (d) our previous effort of integrating NanoFlow into SGLang, which splits the batch for all inputs.

Models and Datasets. We perform experiments using representative open-sourced dense and MoE models, including Llama-3 series, Qwen-2 series, DeepSeek series, and Mixtral. For real dataset evaluations, we use ShareGPT (HuggingFace, 2023), LMSYS-Chat-1M (Zheng et al., 2023), and Splitwise (Patel et al., 2024) following previous works (Zhu et al., 2025; Gond et al., 2025).

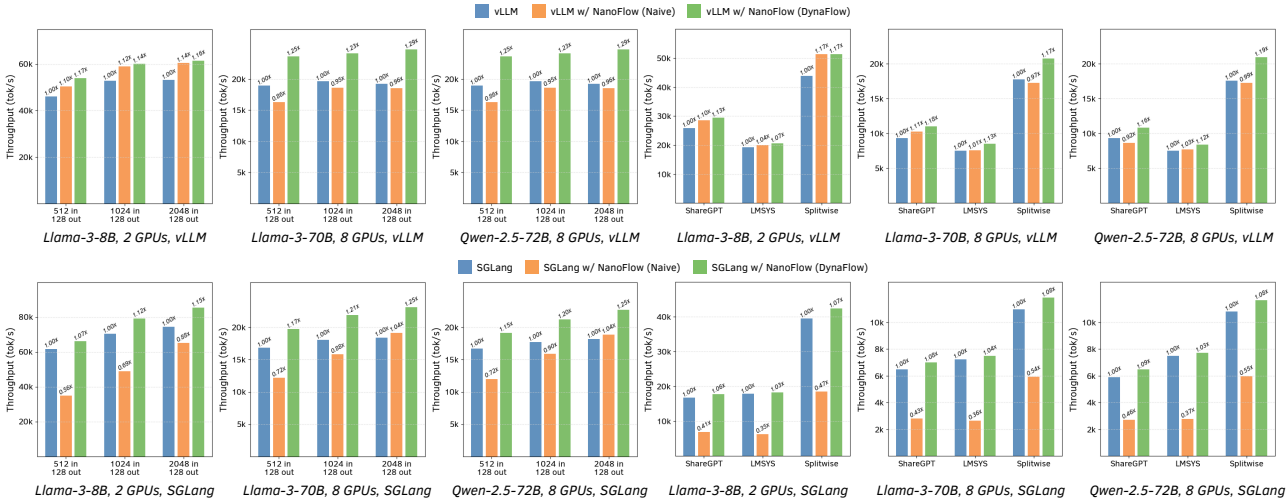


Figure 9. Serving throughput of DynaFlow-enabled NanoFlow integration.

5.2 Microbenchmarks

5.2.1 *Frontend Effectiveness*

Transparency We first evaluate transparency by quantifying the engineering cost, in lines of code (LoC), of integrating DynaFlow into existing ML systems. In vLLM, integration was minimal, requiring only 75 LoC in the GPUModelRunner to handle attention metadata for micro-batches. For MoE models, DynaFlow requires extra but minor annotations (averaging 8 lines per model) to expose operators for fine-grained scheduling. The SGLang integration was slightly more complex due to its backend design and partial `torch.compile` support (v0.5.3.post3). The changes required for other systems are also usually made for `torch.compile` support and are less than 100 lines of code.

Flexibility Beyond low-cost integration, an effective frontend must be flexible enough to express a wide range of parallelism strategies with minimal effort. To quantify this, Table 2 presents the lines of code (LoC) required to implement 6 different intra-device parallelism strategies using DynaFlow. The results show that a diverse set of strategies—spanning overlapping, fusion, and splitting—can be implemented concisely. On average, each strategy requires only 11 lines of code for graph partitioning rules and 31 lines for the dynamic scheduling logic.

5.2.2 *Backend Efficiency*

We evaluate the benefits brought by its backend design of low-level optimization compatibility by measuring the CPU execution time of different DynaFlow configurations. This can become a bottleneck and cause GPU idleness if it is not overlapped by GPU execution. To isolate the CPU time, we record the time to launch all operations for a forward pass of the Llama-3-8B model in vLLM, using a batch size of 1 to prevent being bottlenecked by the GPU command buffer (NVIDIA, 2021). The results in Figure 8 demonstrate the effectiveness of our design choices. Enabling low-level optimizations reduces the CPU execution time by 6.4x compared to a non-optimized variant. While the default dynamic scheduling mode (10.8ms) has a higher CPU cost than vLLM's static execution (4.4ms), DynaFlow also supports a sequential fallback mode which exhibits a CPU time of 4.7ms, nearly identical to the vLLM baseline. For reference, we also compare it with the average GPU execution time on the ShareGPT dataset under the same settings, using the two dotted lines in the figure, for batch sizes of 1 and 128. This shows that the CPU time of the sequential fallback mode is sufficiently low to be overlapped by GPU execution even for small batches. The dynamic mode's CPU time is also shorter than the GPU execution time for larger batches, which are the primary scenarios where intra-device parallelism is beneficial.

5.3 End-to-end Throughput

We now evaluate the end-to-end performance gain from using DynaFlow to integrate these intra-device parallelism strategies.

5.3.1 *NanoFlow*

We integrated NanoFlow into vLLM and SGLang, overlapped the compute-bound, network-bound, and memory-bound operators, and evaluated the offline inference throughput. The results are shown in Figure 9.

In vLLM, we compare the throughput of DynaFlow-based integration with vLLM and our previous effort of implementing NanoFlow on vLLM (Baseline (c) in §5.1) as the naive implementation. We disabled TorchInductor as the naive implementation doesn't support it. DynaFlow-based NanoFlow integration achieves up to 1.17x, 1.29x, and 1.29x

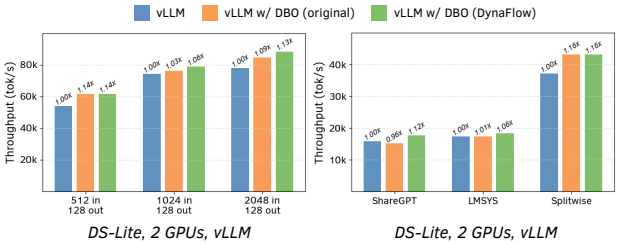


Figure 10. Serving throughput of DynaFlow-based DBO integration in vLLM.

throughput improvement over baseline vLLM on Llama-3-8B, Llama-3-70B, and Qwen-2.5-72B. The speedup mainly comes from the network-bound and memory-bound operations being overlapped when we split the batch. As the number of GPUs increased, the overlapped communication and straggler effects take a higher ratio in the end-to-end time, so the speedup becomes more pronounced. Notably, the speedup on some real workloads like ShareGPT is a bit lower. This is because of the different dataset characteristics and the resulting batch size, which is a condition for batch splitting. In the synthetic evaluation (512 in, 128 out), the average number of tokens per iteration is 8,985, allowing us to use batch splitting for 91.71% of iterations. In contrast, ShareGPT has a higher output-to-input ratio (246 in, 322 out), leading to a lower number of average tokens per iteration of 2,991. Consequently, we only use batch splitting for 46.21% of iterations, which limits the speedup.

The overall trend of SGLang results is similar. The DynaFlow-based integration shows up to 1.17x, 1.17x, and 1.19x speedup across different models. The naive implementation here is straightforward: it splits the batch for every input. In the 2048-in-128-out workload, as the token batch size is large enough, the naive implementation can still result in 1.04x speedup in 8 GPU settings. However, for lightweight workloads, this results in significant performance degradation. As shown in the figure, its throughput can decrease to 0.56x and 0.35x for synthetic and real workloads. This confirms the necessity of dynamically controlling whether to enable the optimization based on the execution context.

5.3.2 *Dual-Batch Overlap*

We evaluated the dual-batch overlap strategy in vLLM on the DeepSeek-V2-Lite MoE model on two GPUs with expert parallel, with results shown in Figure 10. Our DynaFlow-enabled DBO integration achieves up to a 1.14x throughput improvement and performs comparably to, and in some workloads up to 1.1x better than, vLLM's hand-built DBO implementation. Notably, on real-world datasets like ShareGPT and LMSYS-Chat, the performance gains for both implementations are more modest, and vLLM's DBO can suffer from performance degradation. This behavior is due to the lightweight nature of these workloads (e.g., ShareGPT has a median of only 638 tokens per iteration and an average context length of 407), where aggressive batch splitting offers limited benefit and can be detrimental due to overhead. vLLM's native implementation employs a static, low batch-size threshold that does not account for context length, a policy we hypothesize is optimized for different conditions like multi-node parallelism with longer contexts. While DynaFlow's opportunity for overlap is also constrained by the workload, its ability to schedule dynamically prevents this degradation. Conversely, on the more compute-intensive Splitwise dataset, where batch splitting is beneficial, our integration achieves a more pronounced 1.16x speedup. For a detailed analysis of performance in communication-constrained environments, which simulate a multi-node low-bandwidth scenario, we provide an additional experiment using PCIe as the interconnect in Appendix B.

5.3.3 *Communication Overlap*

We implement a naive communication-overlapping strategy by splitting the input batch into two for HuggingFace Transformers, Megatron-LM, xDiT, and FastVideo using DynaFlow and overlapping their tensor-parallel and context-parallel communications. The results are shown in Figure 11. For most cases, DynaFlow effectively overlaps communication in the target systems, resulting in up to a 1.15x speedup. We identified two cases that warrant further analysis. First, the overlap in the backward pass of Megatron-LM training was less effective due to a custom fused TP backward operator in Megatron-LM, which could not be captured by TorchDynamo's limited compiled autograd support. Second, for video generation on FastVideo with the Wan-14B model, the speedup was limited with longer video sequences. This is because the workload is primarily attention-bound; as video length increases, the quadratic scaling of attention dominates the linear scaling of communication, reducing the performance impact of communications.

5.3.4 *TokenWeave*

We integrated the TokenWeave into vLLM and HuggingFace Transformers and evaluated it by serving and training a Llama-3-8B model on two GPUs with tensor parallel. As shown in Figure 12, the TokenWeave integration achieves up to 1.21x throughput improvement over vLLM and 1.22x improvement over HuggingFace Transformers. When compared to the original TokenWeave framework, our performance is similar on optimal workloads. However, the Python-native frontend of DynaFlow enables more flexible adaptation to different workloads during runtime. We use it to select the number of CTAs of the fused kernel based on the batch size, which provides up to 12% throughput improvement over the original framework.

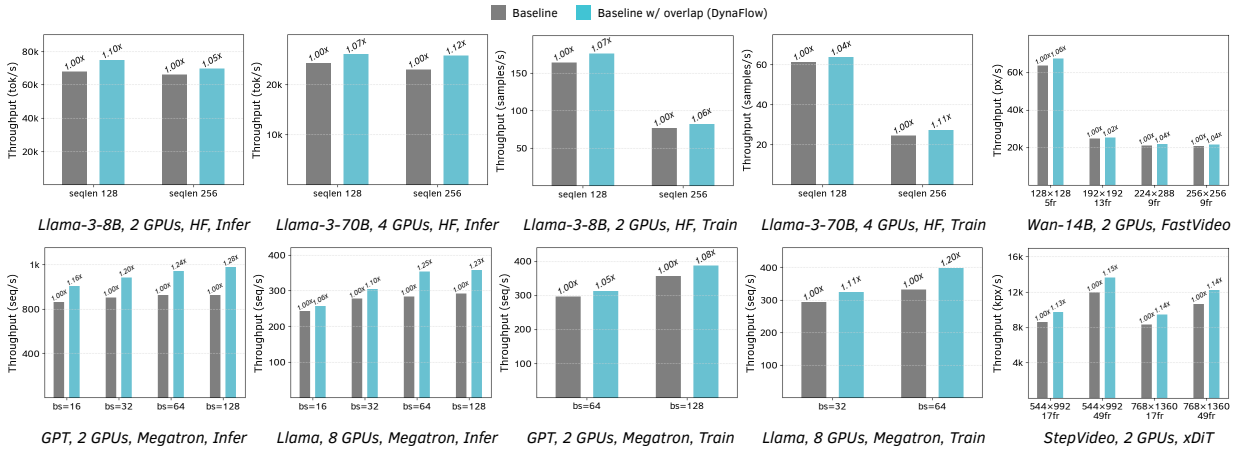


Figure 11. End-to-end throughput of DynaFlow-enabled communication overlap.

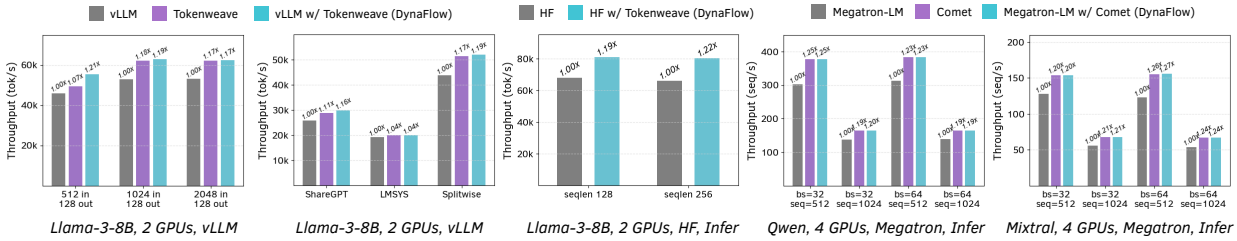


Figure 12. End-to-end throughput of DynaFlow-enabled communication fusion.

5.3.5 *Flux*

We used DynaFlow's `replace_func` API to integrate fused compute-communication kernels from Triton-distributed[1] (Zheng et al., 2025a;b) into vLLM, targeting the `Linear` and `AllReduce` subgraphs. This integration, however, resulted in a performance degradation of up to 20% compared to the original baseline. Profiling analysis indicated that the bottleneck was the fused kernel itself, which exhibited 1.6x higher latency than the separate GEMM and `AllReduce` operations. This highlights the importance of a flexible framework like DynaFlow for rapidly prototyping and validating such optimization techniques.

5.3.6 *Comet*

We also used DynaFlow to integrate the high-performance fused expert-parallel communication kernels from Comet (Zhang et al., 2025) into Megatron-LM. The results are also shown in Figure 12. The integration yielded significant throughput improvements, achieving up to a 1.25x speedup on Qwen2-MoE models and a 1.27x speedup on Mixtral models. This performance level exactly matches that of the Comet team's original Megatron-LM fork (when its other orthogonal optimizations are disabled). This result validates DynaFlow's capability as a transparent integration layer, enabling complex systems like Megatron-LM to adopt state-of-the-art kernels without the substantial engineering overhead of maintaining a custom fork.

[1]The PyTorch dependencies of the original Flux library and vLLM are incompatible.

5.4 Overhead Analysis

We quantified initialization costs using a Llama-3 8B model with a maximum CUDA graph capture batch size of 512. The results are shown in Figure 13. DynaFlow incurs 0.2 s for static analysis, 4.3 s for CUDA graph capture (vs. 2.4 s for vLLM), and a CUDA graph memory footprint of 1.80 GiB (vs. 0.98 GiB for vLLM). These increased results are from capturing separate subgraphs for micro-batches and computational graph analysis. These costs are acceptable compared to the original vLLM initialization cost (24s under the same setup) and modern GPU memory capacity (less than 0.5% of the total available GPU memory on our testbed), and can be further reduced by advanced techniques like Medusa (Zeng et al., 2025).

5.5 Ablation Study

We performed an ablation study on the Llama-3 8B model with two B200 GPUs and the ShareGPT workload, using vLLM with CUDA graph enabled as the baseline (1.00x). As shown in Figure 14, DynaFlow with all optimizations enabled achieves a total speedup of 1.14x. When CUDA graph capture and reply are disabled, the throughput suffers from a significant degradation to 0.96x (0.84x than that of DynaFlow). A similar effect was observed in the baseline,

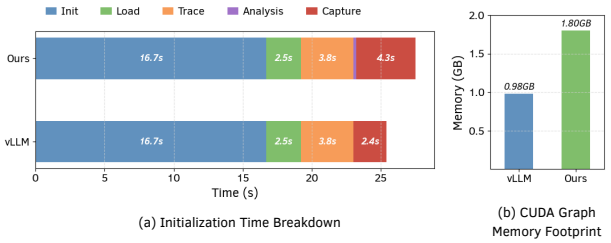


Figure 13. Overhead analysis. `Init`, `Load`, `Trace`, `Analysis`, `Capture` refer to engine initialization, model weight loading, model tracing using TorchDynamo, static analysis in DynaFlow, and CUDA graph capture.

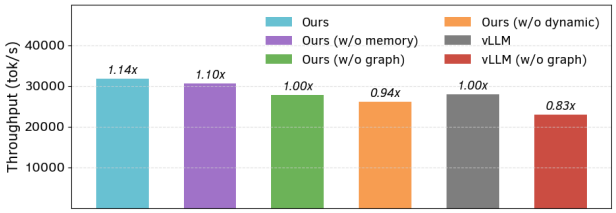


Figure 14. Ablation study. `memory`, `graph`, `dynamic` refer to zero-copy memory pre-allocation, CUDA graph, and dynamic scheduling.

where disabling CUDA Graphs decreased its throughput to 0.83x, indicating the general importance of mitigating CPU overhead for this workload. Next, disabling our zero-copy memory pre-allocation mechanism resulted in a throughput of 1.10x. Finally, we use a static splitting strategy that splits all operators with a fixed batch size threshold, which decreases the throughput to 1.00x.

6 Related Work

Intra-Device Parallelism Recent years have witnessed the wide adoption of intra-device parallelism strategies in both training and inference, as described in Section 2.1. Generic systems for intra-device parallelism, such as NanoFlow (Zhu et al., 2025), have been difficult to integrate into existing training and inference systems due to their workload sensitivity and the need to intercept operator execution. Other automatic systems like Centauri (Chen et al., 2024) aims to the overlapping but are restricted to specific optimization targets, such as computation-communication overlap. Compiler-based systems, such as Comet (Zhang et al., 2025) and CoCoNet (Jangda et al., 2022), focused primarily on kernel fusion techniques. The goal of DynaFlow is complementary; rather than proposing a new fixed strategy, it provides a flexible framework to integrate and dynamically select between or combine different approaches. This flexibility is essential, as recent studies (Zheng et al., 2025a) show that no single strategy, including Comet-style kernel fusion, is universally optimal across all configurations. *Megakernels* are an emerging technique that aims to reduce kernel launch overhead and overlap all operators at fine granularity (Spector et al., 2025; Wu et al., 2025); while such techniques have shown promising latency improvements, improving their dynamicity, such as for conditional computation in MoE, remains an open problem.

General-Purpose ML Runtimes and Compilers. General ML runtimes and compilers such as PyTorch 2 (Ansel et al., 2024) and XLA (Bradbury et al., 2021) offer limited support for introducing arbitrary intra-device parallelism strategies. They typically support custom kernels through graph rewrite rules. However, overlapping and splitting support is limited. XLA does not expose implementation details such as CUDA streams, making it difficult for users to control the execution schedule. Meanwhile, PyTorch takes the opposite approach of exposing much of the CUDA API, forcing users to implement their execution schedules manually and making it difficult to decouple operator execution from model implementation. While both XLA and PyTorch 2 introduce compiler-based overlapping for targeted operators, it is challenging to write rules to cover all possible operators, schedules, and implementation options, especially as the best choice is workload-dependent.

7 Conclusion

The adoption of intra-device parallelism is hindered by a fundamental conflict with the static, sequential programming model, making integration an intractable and inflexible engineering task. We proposed DynaFlow, a framework that resolves this by decoupling the logical model definition from the physical execution schedule. DynaFlow combines a flexible frontend, featuring annotation-based partitioning and a programmable scheduler, with an efficient backend that transparently manages data-flow and preserves optimizations like CUDA Graphs. Our evaluation shows that DynaFlow integrates representative strategies into state-of-the-art ML systems with minimal code, achieving up to 1.29x speedup over original systems and up to 1.1x speedup over existing native implementations.

Acknowledgements

We thank the anonymous MLSys reviewers for their helpful comments. We also thank Kan Zhu, Yilong Zhao, Dedong Xie, Megan Frisella, Zihao Ye, Mat Jacob, and Yuyao Wang, as well as other members and alumni of the UW SyFI Lab, for their insightful discussion during the early stages of this work. Special thanks to Woosuk Kwon, Simon Mo, and the broader vLLM team for their valuable feedback. This work is supported in part by PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, as well as generous donations from NVIDIA, AMD, Intel, and Arm.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., et al. 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*, pp. 929–947, 2024.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., et al. Jax: Autograd and xla. *Astrophysics Source Code Library*, pp. ascl–2111, 2021.

Chang, L.-W., Bao, W., Hou, Q., Jiang, C., Zheng, N., Zhong, Y., Zhang, X., Song, Z., Yao, C., Jiang, Z., et al. Flux: Fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

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 *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 178–191, 2024.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

DeepSeek-AI. Profiling data in deepseek infra. https://github.com/deepseek-ai/profile-data, 2025.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Ge, S., Zhang, Y., Liu, L., Zhang, M., Han, J., and Gao, J. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*, 2023.

Gond, R., Kwatra, N., and Ramjee, R. Tokenweave: Efficient compute-communication overlap for distributed llm inference. *arXiv preprint arXiv:2505.11329*, 2025.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

HuggingFace. Sharegpt dataset, 2023.

Ishii, A. and Wells, R. The nvlink-network switch: Nvidia's switch chip for high communication-bandwidth superpods. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pp. 1–23. IEEE Computer Society, 2022.

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

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 *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., et al. Torchtitan: One-stop pytorch native solution

for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–15, 2021.

NVIDIA. Advanced api performance: Command buffers. NVIDIA Developer Technical Blog, 2021. URL https://developer.nvidia.com/blog/advanced-api-performance-command-buffers.

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Peng, B., Quesnelle, J., Fan, H., and Shippole, E. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

PyTorch. Graph memory management. PyTorch Documentation, 2021. URL https://docs.pytorch.org/docs/stable/notes/cuda.html#graph-memory-management.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Sergeev, A. and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

Spector, B., Juravsky, J., Sul, S., Dugan, O., Lim, D., Fu, D., Arora, S., and Ré, C. Look ma, no bubbles! designing a low-latency megakernel for llama-1b, 2025. URL https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

Team, M. L., Li, B., Lei, B., Wang, B., Rong, B., Wang, C., Zhang, C., Gao, C., Zhang, C., Sun, C., et al. Longcat-flash technical report. *arXiv preprint arXiv:2509.01322*, 2025.

Team, N. Nanoflow-style computation-communication overlap. Github Pull Request, 2025a. URL https://github.com/vllm-project/vllm/pull/23592.

Team, S. Support overlapping two batches. GitHub Pull Request, 2025b. URL https://github.com/sgl-project/sglang/pull/4068.

vLLM. Fused moe modular kernel. https://docs.vllm.ai/en/latest/design/fused_moe_modular_kernel.html, 2025.

Wang, S., Wei, J., Sabne, A., Davis, A., Ilbeyi, B., Hechtman, B., Chen, D., Murthy, K. S., Maggioni, M., Zhang, Q., et al. 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*, pp. 93–106, 2022.

Wu, M., Cheng, X., Liu, S., Shi, C., Ji, J., Ao, K., Velliengiri, P., Miao, X., Padon, O., and Jia, Z. Mirage: A multi-level superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, Boston, MA, July 2025. USENIX Association.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

Zaheer, M., Guruganesh, G., Dubey, K. A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

Zeng, S., Xie, M., Gao, S., Chen, Y., and Lu, Y. Medusa: Accelerating serverless llm inference with materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 653–668, 2025.

Zhang, S., Zheng, N., Lin, H., Jiang, Z., Bao, W., Jiang, C., Hou, Q., Cui, W., Zheng, S., Chang, L.-W., et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *arXiv preprint arXiv:2502.19811*, 2025.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E. P., Gonzalez, J. E., Stoica, I., and Zhang, H. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

Zheng, L., Yin, L., Xie, Z., Sun, C. L., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Zheng, S., Bao, W., Hou, Q., Zheng, X., Fang, J., Huang, C., Li, T., Duanmu, H., Chen, R., Xu, R., Guo, Y., Zheng, N., Jiang, Z., Di, X., Wang, D., Ye, J., Lin, H., Chang, L.-W., Lu, L., Liang, Y., Zhai, J., and Liu, X. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler, 2025a. URL https://arxiv.org/abs/2504.19442.

Zheng, S., Fang, J., Zheng, X., Hou, Q., Bao, W., Zheng, N., Jiang, Z., Wang, D., Ye, J., Lin, H., et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025b.

Zhu, K., Gao, Y., Zhao, Y., Zhao, L., Zuo, G., Gu, Y., Xie, D., Ye, Z., Kamahori, K., Lin, C.-Y., et al. {NanoFlow}: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 749–765, 2025.