

A Flexible Programmable Pipeline Parallelism Framework for Efficient DNN Training

Lijuan Jiang
jianglijuan@pjlab.org.cn
Shanghai AI Lab
China

Zan Zong
zongz@tsinghua.edu.cn
Tsinghua University
China

Xingjian Qian
qianxingjian@zju.edu.cn
Zhejiang University
China

Hengjie Li
lihengjie@pjlab.org.cn
Shanghai AI Lab
China

Zhenxiang Ma
mazhenxiang@sjtu.edu.cn
Shanghai Jiao Tong University
China

Chao Yang
chao_yang@pku.edu.cn
Peking University
China

Jidong Zhai
zhaijidong@tsinghua.edu.cn
Tsinghua University
China

Abstract

Pipeline parallelism is an essential distributed parallelism method. Increasingly complex and diverse DNN models necessitate meticulously customized pipeline schedules for performance. However, existing practices typically rely on predefined schedules, each with strengths, but fail to adapt automatically to the emerging model architectures. Exploring novel high-efficiency schedules is daunting due to the enormous and varying schedule space. Besides, manually implementing schedules can be challenging due to the onerous coding burdens and constantly changing needs. Unfortunately, existing frameworks have limitations in automated schedule exploration and lack flexibility and controllability.

This paper presents FlexPipe, a programmable pipeline parallelism framework with enhanced productivity, programmability, debuggability, and ease of tuning. FlexPipe has two main components: a succinct domain-specific language (DSL) and an automated scheduler. FlexPipe enables automated schedule exploration for various parallel scenarios within a broad spectrum of schedule types at a small search cost. Besides, users can swiftly develop and customize schedules using the FlexPipe DSL, which embodies flexible controllability in the pipeline order of micro-batch computations over stages. It also provides convenient mechanisms to include new operations in schedules to meet changing demands. Our evaluation results demonstrate that FlexPipe achieves up to 2.28 \times performance speedup compared to the popular large-scale parallel framework Megtron-LM, and gains up to 1.49 \times performance speedup compared to the state-of-the-art automated pipeline parallelism framework.

1 Introduction

Large-scale DNN models have gained prominent performance in various areas such as natural language processing [1, 8], computer vision [6], text-to-video [22], etc. Besides, model sizes [1, 3, 29, 30] increase by leaps and bounds. Distributed parallelism [15] has been the fundamental method for training large-scale models due to the long training time and massive memory consumption brought about by the ever-increasing model sizes. Furthermore, pipeline parallelism is generally used as an indispensable model parallelism method to scale up to larger models across servers [25].

Pipeline parallelism [11] shards the model at the layer level into multiple stages that are placed on different devices. Next, a mini-batch of training samples is split into smaller micro-batches, the execution of which over stages is pipelined to allow devices to work simultaneously. Both the spatial stage placement and the temporal schedule that decides the pipelined execution order of micro-batch computations are critical to the efficiency of pipeline parallelism in terms of device idle time (also referred to as bubbles) and memory consumption.

Extensive research [9, 18, 25, 27] has proposed effective stage placement strategies and schedules. However, current approaches usually rely on predefined schedules, each with strengths, but fail to adapt automatically to the emerging model architectures. Moreover, exploring novel efficient schedules is daunting due to the following two aspects.

First, the schedule space is enormous. It typically requires experts to handcraft ingenious schedules, managing hundreds or even thousands of micro-batches and their intricate dependencies while allowing for multiple in-flight micro-batches for pipeline efficiency. Various stage placement strategies further complicate the schedule space. Multiple stages are placed on the same device, and the execution order of micro-batches over different stages needs to be determined.

Keywords: Pipeline Parallelism, Distributed Training, Schedule Exploration, DSL, Scheduling Language

In addition, the performance bottlenecks of schedules depend not only on the method and the underlying hardware but also on the model’s inputs. As a result, the explored schedule running on the same machine may not be usable or fail to achieve optimal performance given different model inputs. For example, Interleaved 1F1B demonstrates superior performance over 1F1B at small batch sizes. Otherwise, the latter schedule achieves better performance since the influence of bubbles is insignificant, while the former incurs more communications.

Second, manually implementing schedules for various distributed scenarios exposes significant challenges: 1) *Productivity*. Manually customizing and developing schedules requires onerous coding burdens and expertise. The implementation of 1F1B and Interleaved 1F1B contains around 0.7k and 1.4k code lines, respectively, in the popular framework Megatron-LM [4]. 2) *Programmability*. Developers must maintain tangled communications and data dependencies concerning different micro-batches, computation types, and even stages in distributed environments. In addition, diverse model architectures may constantly introduce new operations in schedules. For instance, synchronization is required to exchange the output states of submodules corresponding to different modalities in multi-modal models [28]. 3) *Debuggability*. Manual implementations of schedules are prone to errors and lack intermediate debugging information, making the debugging process very time-consuming and unbearable.

Existing effective frameworks [20] enable automated schedule exploration when the stage placement strategy is specified. However, it shows deficiencies in the diversity of searchable schedules. For example, it cannot support the circular stage placement strategy, with which multiple types of efficient schedules are designed for different parallel scenarios [17, 23, 25, 26]. Besides, the search cost is intolerable when the hardware resources are extensive. Compiler approaches [32] propose a domain-specific language (DSL) to improve productivity in developing schedules. However, implementing 1F1B still requires 0.2k code lines due to the sophisticated two-layer structured syntax. In addition, the compiler approach can neither enable automated exploration of novel schedules nor support new operations in schedules for various model architectures.

In this paper, we present FlexPipe, a flexible programmable pipeline parallelism framework that circumvents limitations in existing frameworks. FlexPipe includes a succinct DSL to express pipeline schedules, enabling automated schedule exploration within a broad spectrum of schedule types at small overheads. With FlexPipe, implementing existing mainstream schedules only requires a few lines of code. Besides, FlexPipe provides mechanisms to control the pipeline order of micro-batch computations over stages flexibly and to support new operations to customize schedules swiftly.

In FlexPipe, we regard the scheduling of a micro-batch computation (i.e., forward or backward pass) as a scheduling

step. We observe that the key to exploring and developing schedules is the flexible controllability in the scheduling order of micro-batch computations. We also observe that, suppose the stage placement strategy is specified, the schedule can be broken down into a series of scheduling steps. The forward or backward passes are constantly selected to be scheduled from the micro-batch computations with resolved dependencies on each device. We can control the schedule by determining the scheduling priorities, which decide the order in which the dependency-free micro-batch computations are selected for scheduling.

Two types of scheduling priorities need to be determined: 1) the priority concerning computation types (we call **computation type traversal priority**), i.e., whether to schedule preferentially forward or backward passes; 2) the priority concerning stages (we call **stage traversal priority**) that decides the micro-batch computations of which stage are preferentially to be scheduled if multiple stages are placed on the same device. Furthermore, based on the concepts of scheduling priorities, we observe that efficient schedules generally employ the same scheduling priorities in different scheduling steps.

FlexPipe is built upon the above essential observations. Besides the FlexPipe DSL, FlexPipe also includes an **automated scheduler** that interacts with the DSL to arrange schedules automatically, and an **auto-tuner** that finds the best configurations for various model inputs. FlexPipe facilitates the schedule exploration with enhanced productivity, programmability, debuggability, and ease of tuning. Compared with state-of-the-art methods, our experiments demonstrate that FlexPipe achieves up to 2.28 \times performance speedup on training language models with large embedding layers and up to 1.29 \times performance speedup in multimodal models. Overall, this work makes the following contributions.

- It introduces a succinct DSL that allows flexible controllability in the pipeline order of micro-batch computations to express various schedules.
- It enables automated schedule exploration with diverse searchable schedule types at small overheads for various stage placement strategies.
- It provides programmable mechanisms to customize schedules for the emerging model architectures without worrying about tanglesome data dependencies, communications, etc.
- It proposes FlexPipe, an end-to-end system, which instantiates explored schedules for efficient runtime execution. Besides, it can also tune the best configuration of schedule types and hyperparameters for various model inputs and hardware resources.

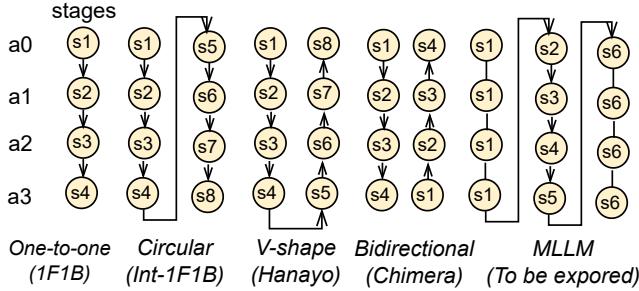


Figure 1. Effective stage placement strategies. A0-a4 represent devices, and s1-s8 represent stages.

2 Background and Motivation

2.1 Pipeline Parallelism.

Prior research has proposed effective stage placement strategies. Figure 1(i) shows that 1F1B uses the one-to-one stage placement strategy. Interleaved 1F1B [25], Hanayo [23] and Chimera [18] propose the circular, V-shape, and bidirectional stage placement strategies, respectively, where bubbles are reduced due to the smaller computational time of a single micro-batch, or more devices working simultaneously. Subsequent research further evolves based on the above stage placement strategies. BitPipe [34] fuses interleaved pipelines with bidirectional pipelines. Furthermore, ZB-H1 [27] splits the gradient computation and fills the weight gradient computations in bubbles of 1F1B to improve efficiency.

2.2 Motivation

Diverse model architectures pose challenges to existing predefined pipeline schedules.

Schedule exploration given various stage placement strategies. Exploring efficient schedules for different stage placement strategies is necessary but prohibitive due to the enormous schedule space. For instance, large embedding layers [33, 36] in multilingual models are introduced to cover the vocabulary of multiple languages. However, using the existing stage placement strategies, the performance degrades due to the imbalanced workloads of the stages with and without embedding layers. Figure 2 profiles a GPT model to compare the imbalanced computations and memory consumption, which become increasingly imbalanced with the increasing vocabulary size. The slowest stage is 5.63× slower than the fastest for the GPT model with the 1M vocabulary size, and costs 4.87× more memory. New stage placement strategies (“MLLM” in Figure 1) that distribute the embedding layers over multiple devices to balance memory and computations are proposed. However, extra bubbles are produced due to data dependencies when the specified stage placement strategy is applied to the predefined schedules.

Schedule customization for diverse model architectures. There is a demand for customizing schedules for

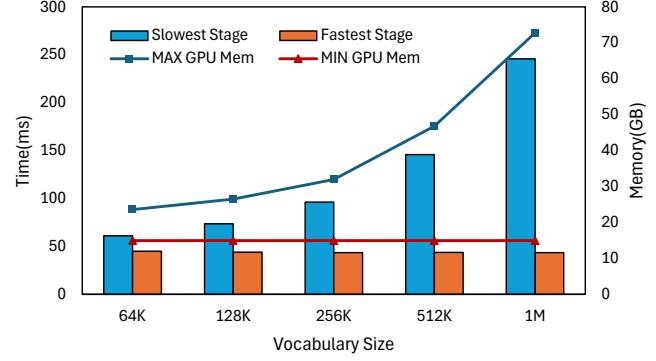


Figure 2. Imbalanced workloads of a 5B GPT model with varying vocabulary sizes.

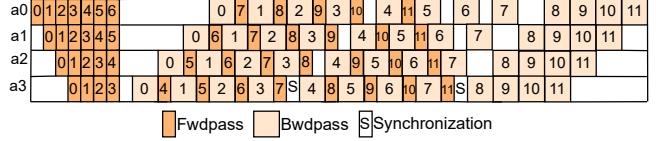
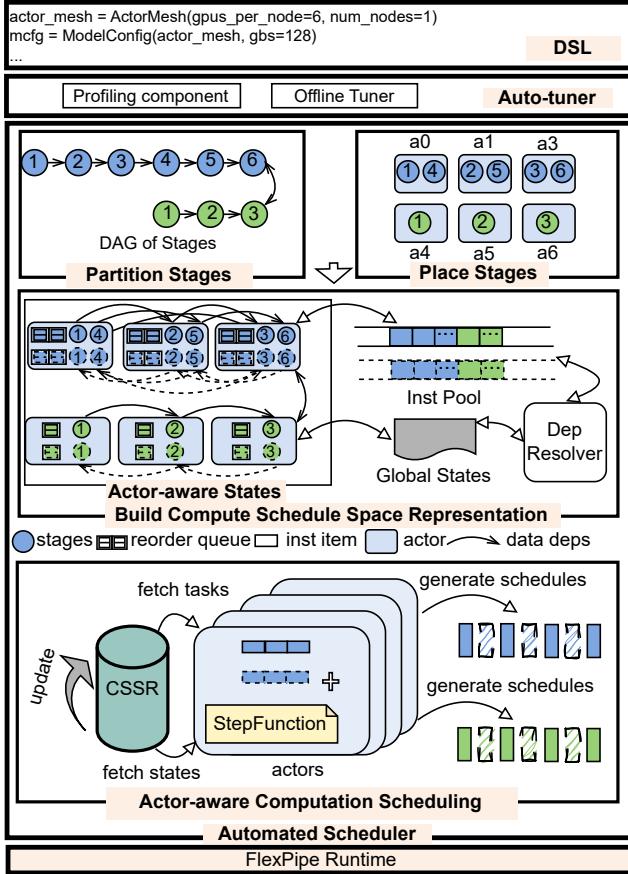


Figure 3. DistMM-Pipe.

various model architectures, which arrange micro-batch computations differently from existing predefined schedules. Besides, new operations may be required. For instance, DistMM-Pipe [10] is designed for multi-modal models [7, 35] that contain multiple submodules to process different modalities such as image and text. DistMM-Pipe distributes different submodules onto different devices for parallel computation. Since submodules need to synchronize the output states, DistMM-Pipe launches more forward passes in the warm-up phase and adds a synchronization operation every few micro-batches to avoid frequent communications, as shown in Figure 3. The synchronization of submodules is supported as a new operation not included in existing schedules. In addition, further analysis shows that synchronization costs around 17%. Asynchronous communications can be leveraged to optimize bubbles. We can also employ mixed schedule types for different submodules since submodules are configured with varying model configurations and may be fit for different schedule types. However, efficient methods to flexibly tune schedules and support new operations are lacking.

3 Overview of FlexPipe

We present the pipeline parallelism framework FlexPipe to screen users from the complexities of exploring and customizing efficient schedules for various parallel scenarios. This section provides an overview of FlexPipe, as shown in Figure 4, which consists of four components: a DSL, an automated scheduler, an auto-tuner, and a runtime.

**Figure 4.** FlexPipe Overview.

FlexPipe DSL (§ 4) offers a simple API to express various schedules. Users only need to specify the corresponding parameters in the API for different schedule types. To include new operations in schedules, users only need to write a function for the new operation. All the other complex scheduling is done by the framework. Furthermore, we provide more mechanisms for users to precisely control the scheduling of micro-batch computations as detailed in Section 4.2.

The scheduler (§ 5) arranges schedules automatically. It is worth noting that FlexPipe uses actors as the internal representation for devices. The scheduler primarily constructs the intermediate representation of the computation schedule space (CSSR), inspired by conventional CPU instruction pipeline problems [5]. CSSR leverages a set of instructions to represent various computations and communications. Next, the actor-aware schedule is performed, which interacts with CSSR to automatically generate schedules represented as sequences of instructions according to the scheduling information specified in the DSL. Besides, optimization passes such as gradient separation and asynchronous communications are included to optimize bubbles.

The auto-tuner (§ 6) searches and tunes efficient schedules under different model inputs and hardware resources. The auto-tuner consists of a profiling component and an offline tuner. The profiling component collects the timing metrics for computations and communications. Besides, the collected data is stored for future analysis, which avoids repeated data collection. The offline tuner finds the most suitable schedules enumerated in the schedule space.

4 The FlexPipe DSL

The FlexPipe DSL includes constructs that express stages and a scheduling language to compose pipeline schedules. The FlexPipe DSL is embedded in Python. Users can specify schedules manually. Additionally, the auto-tuner can be invoked to find efficient schedules automatically.

4.1 Data Model

The primary constructs concerning stages consist of Stage, Actor, StageSet, and ActorMesh. Stage and StageSet describe the stage data and the DAG of stages, respectively. Actor and ActorMesh express the device data and the device topology, respectively. These constructs are operated through the construct ModelConfig. Users first define the device topology and model configuration (Lines 1-3 of Figure 5). Besides, users can set the configuration of different submodules by invoking the interface `init_cfg` multiple times with various settings of parameter “modality” (Line 3 of Figure 5).

```

1 actor_mesh = ActorMesh(gpus_per_node=8, num_nodes=4)
2 mcfg = ModelConfig(actor_mesh, gbs=128)
3 mcfg.init_cfg(layer_specs, mbs=4, modality='default')
4 mcfg.parition(num_stages=4)
5 mcfg.place('gpu:0-3','one-to-one')
6 ctp = CompTypeTraversal('bwpass-first')
7 stp = StageTraversal(fstp='breadth-first', bstp='breadth-first')
8 mcfg.set_priority(ctp, stp)
9 mcfg.build_graph()
10 cssr = CSSSR(mcfg)
11 sched = SchedGenerator(mcfg, cssr)
12 sched.gen()
```

Figure 5. 1F1B using the FlexPipe DSL.

FlexPipe uses the method `partition` and `place` (Lines 4-5 of Figure 5) to split model layers into stages and build mappings from stages to actors. By default, we evenly distribute the transformer layers into each stage. Besides, we compare the strengths and weaknesses of different stage placement strategies in Figure 8, and find that each stage placement strategy has strengths and weaknesses so that no single approach can outperform the other methods in all metrics. Hence, we have built in the one-to-one, circular, V-shape, and bidirectional stage placement strategies. In addition, a

stage can be labelled as a shared stage among several actors. A user must rewrite the *partition* method when the default layer split rule fails to meet the demand. The partitioned stages are organized as a DAG (Line 9 of Figure 5).

FlexPipe’s Instructions FlexPipe represents pipeline schedules as sequences of instructions mapped to the corresponding operations at runtime. Each instruction is featured with (stage ID, micro-batch ID). Instructions in FlexPipe can be classified as (i) computations, including forward and backward pass, and the gradient calculation of weights and inputs, and (ii) cross-rank communications, including P2P send and receive for activations and gradients, and synchronizations for multimodal models. The instructions supported by FlexPipe are listed in Table 1. Besides, FlexPipe supports new operations registered as new instructions, which will be detailed in the following section.

Table 1. Instructions of FlexPipe for various schedules.

Computation	FwdPass, BwdPass, CompWeightGrad, CompInputGrad
Communication	SendAct, SendGrad, RecvAct, RecvGrad, SyncWithAllGather, SyncWithGather

4.2 Scheduling Language

Users can specify pipeline schedules using FlexPipe’s scheduling language. At the very least, users can specify schedule types by setting two parameters corresponding to two scheduling priorities: the computation type traversal priority and the stage traversal priority. Table 2 lists the built-in values for the two priorities. Lines 6-8 of Figure 5 illustrate how the priorities are set. *Set_priority* sets the same priorities for actors assigned to a modality. Different priorities can be set for actors using *set_ctp_for_actor* and *set_stp_for_actor* (Line 1,2 of Figure 7(iii)) for the two priorities, respectively. We detail how we abstract the built-in values for the priorities and the scheduling mechanisms based on the priorities in § 5.

Moreover, the constructs CSSR and SchedGenerator (Line 10,11 of Figure 5) describe the two main components of the automated scheduler (§ 5). A user can control the scheduling of micro-batch computations through the methods and programmable mechanisms in terms of the two constructs.

Registration of new instructions. Users can support new operations that FlexPipe has not included yet. Figure 6 presents the registration of synchronization for multimodal models to illustrate how to support and bind new operations with instructions for scheduling. First, *register_new_inst* registers a new instruction and returns the corresponding instruction type (Line 11 of Figure 6). Users can configure the rules to schedule the registered instructions by specifying the attribute “*sched_attr*”. Lines 1-5 of Figure 6 specify how many prior dependent instructions must have been scheduled before scheduling the registered instruction with the

micro-batch ID “*cur_microid*”. Lines 6-10 of Figure 6 specify when to schedule the subsequent dependent instructions with the micro-batch ID “*nxt_microid*”. The scheduling unit “*sched_unit*” decides how many instructions are scheduled at a time. The scheduler (§ 5) calls the “*check_prev_dep*” and “*check_nxt_dep*” automatically to resolve data dependencies during the scheduling process. Users can also configure the attributes “*inst_attr*” for the corresponding operations to use at runtime. We set the ranks of the communication group for synchronization. Second, the data dependency of instructions can be added. Besides, the data dependency of instructions is generally related to stages. For instance, synchronization among submodules must be scheduled after the forward passes and before the backward passes corresponding to the final stages of each submodule in multi-modal models discussed above (Lines 14-19 of Figure 6). Therefore, a new stage must be registered where the new registered instruction is attached through *register_new_stage* (Lines 12,13 of Figure 6). The data dependency of instructions is set through the method *set_cssr_deps* (Line 19 of Figure 6). The set consisting of pairs ((*inst_type1*, *stage_i*), (*inst_type2*, *stage_j*)) needs to be passed, which means the *inst_type1* corresponding to *stage_i* is scheduled before the *inst_type2* corresponding to *stage_j*. Third, *map_inst_to_operation* maps a user-defined

```

1 def check_prev_dep( prev_stageid, prev_microid,
                     cur_stageid, cur_microid, sched_unit=1):
2     s = get_stage(global_stage_id=prev_stageid)
3     dsize = get_dp_size(s.modality)
4     mbs = get_micro_batch_size(s.modality)
5     return prev_microid *dsize*mbs >= cur_microid + sched_unit
6 def check_nxt_dep(cur_stageid, cur_microid,
                   nxt_stageid, nxt_microid, sched_unit=1):
7     s = get_stage(global_stage_id=nxt_stageid)
8     dsize = get_dp_size(s.modality)
9     mbs = get_micro_batch_size(s.modality)
10    return (nxt_microid+sched_unit)*dsize*mbs <= cur_microid
11 new_inst_type = register_new_inst(inst_type='SyncWithGather',
                                    sched_attr={'cprev': check_prev_dep, 'cnxt': check_nxt_dep},
                                    inst_attr={'group':[4,7]}, sched_unit=1)
12 s1 = register_new_stage(mcfg, 'SyncWithGather', modality='img')
13 s2 = register_new_stage(mcfg, 'SyncWithGather', modality='txt')
14 cssr_deps = {}
15 s = mcfg.get_stage(modality='txt', local_stage_id=-1)
16 cssr_deps.update({
17     (s2.stage_idx, new_inst_type):(s.stage_idx, VPipeInstType.BwdPass),
18     (s.stage_idx, VPipeInstType.FwdPass):(s1.stage_idx,new_inst_type)})
17 s = mcfg.get_stage(modality='txt', local_stage_id=-1)
18 cssr_deps.update({
19     (s1.stage_idx,new_inst_type):(s.stage_idx, VPipeInstType.BwdPass),
20     (s.stage_idx, VPipeInstType.FwdPass):(s2.stage_idx,new_inst_type)})
19 set_cssr_deps(cssr, cssr_deps)
#"_exec_sync_with_gather" is a user-defined function
20 register_new_function(new_inst_type, _exec_sync_with_gather)

```

Figure 6. A multimodal example for registering instructions.

```

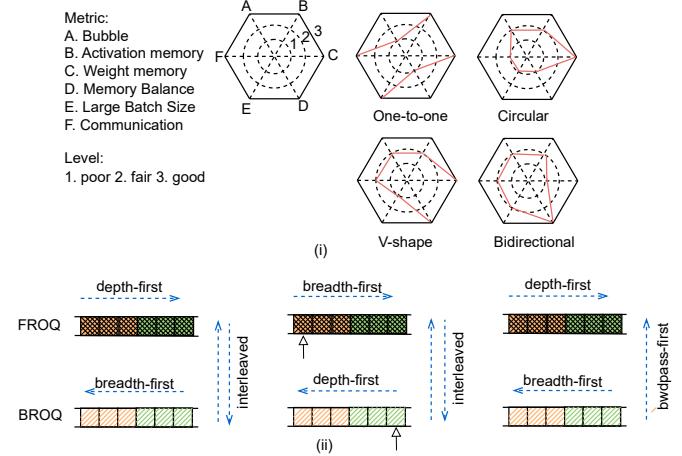
1 def check_loading(pipeid, stageid=None, cur_inflight_mics=None):
2     self.inflight_mics = self.inflight_mics[pipeid]
3     inflight_mics = 0
4     for s in get_actormesh().dev(pipeid).stages:
5         inflight_mics += cur_inflight_mics[s.stageid]
6     return sum(self.inflight_mics) > inflight_mics
7 config_inflight_mics(sched, [[3,3],[3,3]], cfunc=check_loading)
    (i)Configure in-flight micro-batches
1 def _valid_schedule(pipeid, stageid, mroids, insttype):
2     if (insttype == FwdPass and get_stage(stageid).is_first_pipestage)
        or (insttype == BwdPass and get_stage(stageid).is_last_pipestage):
3         return True
4     nstep = -1
5     if insttype == FwdPass:
6         prev_pipeid = get_stage(stageid-1).actors[0]
7         nstep = get_stepnum(prev_pipeid, stageid-1, mroids, insttype)
8     elif insttype == BwdPass:
9         prev_pipeid = get_stage(stageid+1).actors[0]
10    nstep = get_stepnum(prev_pipeid, stageid+1, mroids, insttype)
11    if nstep == -1:
12        return False
13    if get_stepnum(pipeid) <= nstep:
14        return False
15    return True
16 register_new_checkfunc(sched, [[3,3],[3,3]], cfunc=_valid_schedule)
    (ii)Configure user-defined rules for scheduling micro-batches
1 set_ctp_for_actor(actor_id=None, ctp=None)
2 set_stp_for_actor(actor_id=None, stp=None)
3 register_new_priority(sched: SchedGenerator,
                       priority:str, step_func:callable)
    (iii)Configure scheduling priorities for actors

```

Figure 7. DSL Examples of controlling micro-batch orders.

callable operation to the registered instructions at runtime (Line 20 of Figure 6).

Controllability in scheduling micro-batch computations. We further provide multiple methods to control the scheduling order of micro-batch computations besides the scheduling priorities discussed above. Figure 7 gives examples. *Config_inflight_mics* controls the schedule’s maximum number of in-flight micro-batches. Users can specify in-flight micro-batches for each stage via a list through the parameter “inflight-mics”. Besides, Users can also set rules by defining callable functions (Lines 1-6 of Figure 7(i)), which will be invoked by the scheduler (§ 5) automatically. *Register_new_checkfunc* sets rules during the scheduling of micro-batch computations by the scheduler (Line 16 of Figure 7(ii)). The example (Lines 1-15 of Figure 7(ii)) checks whether the instruction of type *insttype* on actor *pipeid* to be scheduled meets the data dependency. The corresponding instruction is to be scheduled if *true* is returned. *Register_new_priority* enables users to register new priorities when the built-in values in Table 2 for the scheduling priorities can not express the user-defined schedules (Line 3 of Figure 7(iii)). We will

**Figure 8.** (i)Characteristics of each stage placement strategy. (ii)Illustrations on the traversal order of instruction items in reorder queues.**Table 2.** Built-in values for the stage placement strategy and the scheduling priorities.

	Name	Values
Stage Placement Strategy		one-to-one, circular, V-shape, bidirectional
Controllability in the Order of Micro-batch Computations Over Stages	Computation Type Traversal Priority	(bwdfirst, unit1, unit2), (fwdfirst, unit1, unit2), (interleaved, unit1, unit2)
	Stage Traversal Priority	breadth-first, (breadth-first, interval), depth-first, (depth-first, interval)

detail how the *step_func* is defined in § 5.2. Users can invoke one or more methods according to their needs.

5 Scheduler

We break down the scheduling process into three phases as shown in Figure 4. First, the model is partitioned into stages and then distributed over actors based on the stage placement strategy. Besides, the stages’ DAG is maintained to track data dependencies. This phase is executed once the stage data is specified as mentioned in Section 4.1. Second, the computation schedule space representation (CSSR) is built. Third, the actor-aware schedule is performed to arrange schedules based on the scheduling settings concerning micro-batch computations described in Section 4.2. We leverage the optimizations on multi-modal models as an example to illustrate the scheduling process in Figure 9.

5.1 Computation Schedule Space

We use the computation schedule space representation (CSSR) to enable various pipeline schedules. As is known, once the stage placement strategy is specified, each actor’s scheduled micro-batch computations are determined. Each actor

must constantly select the micro-batch computations whose data dependency has been resolved for scheduling until all assigned computations have been scheduled. Hence, CSSR is used to dynamically acquire the dependency-free micro-batch computations for each actor at each scheduling step.

The main components of CSSR are listed below.

Instruction Item. Instruction items are instruction instances used to represent the micro-batch computation or communication. An instruction item is regarded as the basic unit scheduled in the scheduler. An instruction item is labeled with (instruction type, micro-batch ID, stage ID).

Instruction pool. The instruction pool holds all the instruction items of all actors to be scheduled. Multiple instruction queues are created. Each instruction queue contains instruction items corresponding to all micro-batch computations over all stages of a specific instruction type. Instruction items are stored in the queues by micro-batch ID and stage ID. Instruction queues corresponding to the instruction type *FwdPass* and *BwdPass* are created by default. Besides, instruction queues corresponding to instruction types that users register using the scheduling language are also created.

Actor-aware reorder queues. Reorder queues are used to maintain instruction items whose data dependencies have been resolved for actors. Hence, each actor creates multiple reorder queues that correspond to different instruction types. By default, on an actor, reorder queues corresponding to *FwdPass*, *BwdPass*, and instruction types registered by users are created. Each reorder queue manages dependency-free instruction items corresponding to all micro-batch computations over the stages assigned to the actor to which the reorder queue belongs.

Local buffer. We configure a local buffer for each actor to keep the scheduling states of a scheduling step that mainly tracks the scheduled instruction items. Before the next scheduling step begins, local buffers are cleared. This is primarily intended to simulate a distributed environment in which the scheduling states are inaccessible to remote actors in the same scheduling step.

Data dependency. The scheduling of instruction items adheres to data dependencies, which means scheduling prior dependent instruction items is a prerequisite for scheduling subsequent instruction items. CSSR employs constructs to record data dependencies between instruction items. The data dependencies between instruction items for forward passes are transformed from the DAG of stages, the reverse of which are the data dependencies between instruction items for backward passes. Data dependencies set by *set_cssr_deps* are also built. Besides, the instruction item with a smaller micro-batch ID is preferentially scheduled over a larger one of the same instruction type and stage ID by default.

Algorithm 1: StepFunction

```

Input: actorid, states, cttp, stp, cfuncs
Result: An instruction item or a nop.
1 ordered_inst_type  $\leftarrow$  sort_inst_types(cttp)
2 for insttype in ordered_inst_type do
3   queue  $\leftarrow$  get_reorder_queue(actorid, insttype)
4   direction, interval  $\leftarrow$  parse_stp(stp, insttype)
5   if is_empty(queue) then
6     | continue
7   end
8   if interval  $\neq$  None then
9     | pos, stepped_interval  $\leftarrow$  states
10    | if stepped_interval == interval then
11      |   | pos  $\leftarrow$  move(queue, pos, direction)
12    | end
13    | stageid, microid  $\leftarrow$  fetch_inst1(queue, pos,
14      |   | cfuncs)
15    | return (insttype, stageid, microid)
16  end
17  else
18    | stageid, microid  $\leftarrow$  fetch_inst2(queue,
19      |   | direction, cfuncs)
20    | return (insttype, stageid, microid)
end
end

```

Dynamic data dependency resolver. The data dependency resolver is leveraged to maintain the actor-aware reorder queues dynamically. Once an instruction item is scheduled, the resolver immediately checks if its successors' data dependencies are resolved based on the scheduling states in the corresponding local buffer. Subsequent instruction items with resolved data dependencies are put into the reorder queues of the corresponding actor.

5.2 Actor-aware schedule

We use an actor-aware schedule mechanism that interacts with CSSR to arrange various schedules automatically.

Key Insights. With CSSR, each actor can constantly fetch instruction items for scheduling from the corresponding reorder queues. The correctness of the schedule is ensured through the dynamic data dependency resolver. The key to achieving high efficiency is determining which instruction items to schedule when multiple alternatives are available.

Furthermore, we observe that efficient pipeline schedules generally exhibit repetitive cycles, wherein the same computation types are periodically scheduled over different micro-batches on each actor. The repetitive cycles can be maintained if each actor traverses reorder queues to fetch available instruction items for scheduling in a fixed order.

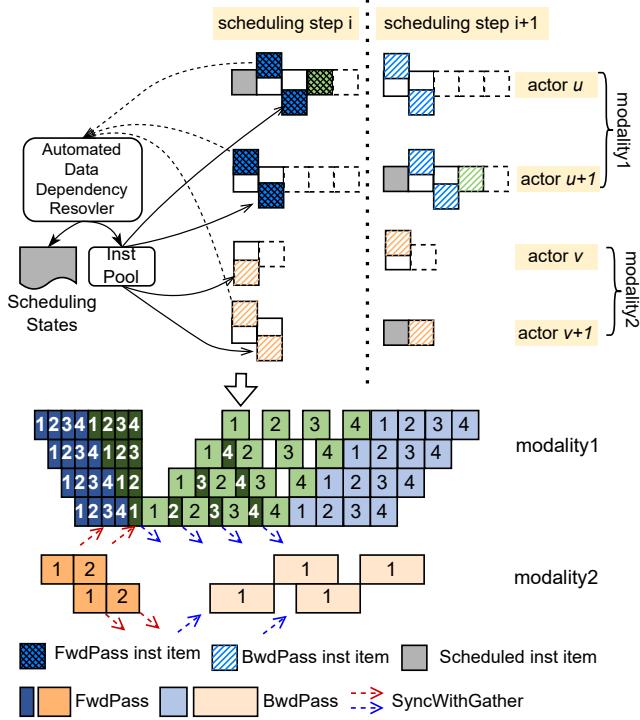


Figure 9. The automated scheduling process. The circular and one-to-one stage placement strategies are applied to “modality1” and “modality2”, respectively. $cttp = \text{“interleaved”}$, $fstp = (\text{“breadth-first”}, 4)$, $bstp = (\text{“depth-first”}, 4)$ for “modality1”. $cttp = \text{“bwdpass-first”}$, $fstp = \text{“breadth-first”}$, $bstp = \text{“breadth-first”}$ for “modality2”.

Various schedules can be arranged by enumerating different traversing orders along reorder queues.

Therefore, we translate the controllability in the scheduling of micro-batch computations into controlling the traversing order of instruction items in the reorder queues of each actor. We abstract two types of priorities to determine various traversing orders of reorder queues as detailed below.

Computation Type Traversal Priority. The computation type traversal priority determines which type of computations (i.e., forward or backward passes) to be prioritized for scheduling. More specifically, it depicts which reorder queue (i.e., the one corresponding to *FwdPass* or *BwdPass*) to be traversed preferentially. *CompTypeTraversal* (Line 6 of Figure 5) allow users to set different priorities. Besides, we build in three kinds of priorities (listed in Table 2), namely *bwdpass-first*, *fwdpass-first*, and *interleaved*, which refers to preferentially traversing the reorder queues corresponding to *BwdPass*, *FwdPass*, and interlaced *BwdPass* and *FwdPass*, respectively. We can also set different scheduling units *unit1* and *unit2* for *FwdPass* and *BwdPass* instruction items, respectively. By default, *unit1* and *unit2* are set to 1.

Stage Traversal Priority. The stage traversal priority determines the micro-batch computations over which stage to be prioritized for scheduling when multiple stages are placed on an actor. Concretely, it determines how to traverse the reorder queue corresponding to a specific instruction type.

We use *StageTraversal* and *interval* (Line 7 of Figure 5) to control the direction and the speed of the traversal to a reorder queue, respectively. We build in two types of traversal directions for *StageTraversal*, including *breadth-first* and *depth-first*. *Breadth-first* traverses the reorder queues from front to end and preferentially schedules instruction items concerning micro-batch computations of stages with model layers in the front end, while *depth-first* traverses in the opposite direction and preferentially schedules instruction items concerning micro-batch computations of stages with latter model layers. Besides, when *interval* is set with an integer, the current traversed instruction items with the same stage ID must be scheduled consecutively *interval* times before the traversal moves forward to the instruction items with a different stage ID in the configured traversal direction. Otherwise, the first traversed instruction item in the traversal direction is scheduled. Besides, the stage traversal priority can be set differently for *FwdPass* and *BwdPass* reorder queues.

After configuring priorities, actors schedule instruction items in the following steps.

Step 1. Initialize reorder queues. Only *FwdPass* instruction items of the first pipeline stage are dependency-free and schedulable, and are put into the reorder queue corresponding to *FwdPass* of the actor with the first stage.

Step 2. In a scheduling step, all actors perform *StepFunction* in Algorithm 1 to fetch schedulable instruction items. If there are no schedulable alternatives, a bubble comes up.

Step 3. Each actor updates the scheduling states in its local buffer.

Step 4. The dynamic data dependency resolver checks the subsequent instruction items whose data dependency has been resolved and puts the schedulable alternatives into the corresponding reorder queues.

Step 5. Update the scheduling states from the local buffer to the instruction pool and clear the local buffer. If all instruction items of the instruction pool have been scheduled, the scheduling process exits. Otherwise, return to *step 2*.

Algorithm 1 presents how to fetch instruction items according to the built-in values of priorities. Besides, the rules specified by users (Line 7/Line 16 of Figure 7(i)/(ii)) are checked through *cfuncs* when fetching instruction items (Line 14 and 18 of Algorithm 1). Users can define new traversal orders of instruction items in reorder queues by writing a new callable step function and registering it as a new value for the two priorities through Line 3 of Figure 7(iii).

Bubble optimizations. We design and implement a gradient separation algorithm to automatically optimize bubbles

of different schedules. As shown in Figure 10, when a bubble is detected, we first estimate the blocked instruction item, e.g., a *BwdPass* whose micro-batch ID is 2 on actor 0. Then, we backtrace the scheduled *BwdPass* to stash weight gradient computations of actors 1 and 2 to schedule instruction items in the dependency chain as early as possible. With dependent instruction items scheduled, the blocked instruction items can be released to eliminate bubbles. The stashed weight gradients are inserted into the subsequent bubbles on the corresponding actors. We employ asynchronous P2P communication that can be referred to in the work [14], and details are omitted here.

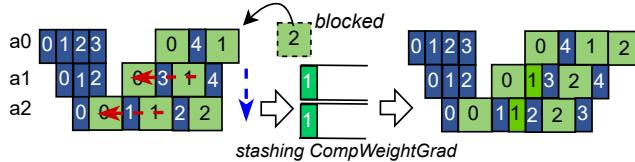


Figure 10. An example for the gradient separation algorithm.

6 Auto-Tuner

We use grid search to find optimal schedules in the parameter space defined by:

$$\underbrace{(pp, dp, mbs, \text{placement strategies}, fstp, bstp, ctp)}_{\text{schedule types}}$$

The parameter values are set according to the following rules.

- The parameters determining the schedule types are taken from Table 2.
- The *interval* is not set unless the circular placement is applied.
- The *ctp* is not set to *forward-first* unless the user specifies.
- *unit1* and *unit2* is set to 1 since we generally apply fine-grained gradient computation to reduce bubbles.

7 Evaluation

We conduct experiments on up to 64 NVIDIA A800 SXM4 80GB GPUs distributed across 8 nodes.

7.1 Automated Exploration of Schedules.

In this section, FlexPipe automatically searches efficient schedules for transformer models with large embedding layers. Two main model configurations with varying vocabulary sizes are tested as shown in Table 3.

Baselines. We compare FlexPipe with three baselines employing different schedules: 1) Megatron-LM, which uses 1F1B; 2) T-1F1B, which adapts 1F1B to incorporate the MLLM stage placement strategy in Figure 1, and the corresponding schedule of T-1F1B is arranged by FlexPipe; 3) Tessel,

Table 3. Model configurations of GPT models.

Model Size without Embedding Layers	5B	16.1B
Total Model Size	5.4B/5.7B/6.3B/7.7B/10.2B	18.2B/20.3B/24.3B/25.1B/25.9B
Vocabulary Size	64K/128K/256K/512K/1M	256K/512K/1M/1.1M/1.2M
Layers	64	80
Attention Heads	2560	4096
Hidden Size	64	32
Sequence Length	512	1024
Global Batch Size	128	128

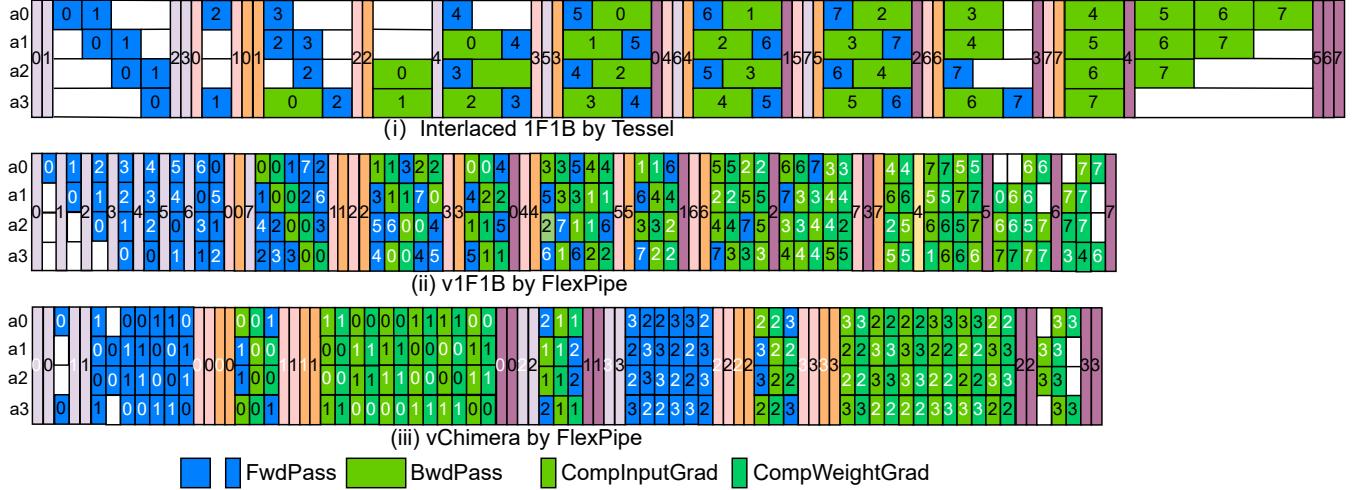
Table 4. Model configurations of CLIP models.

Total Model Size	4.7B	8.2B	9.7B
Submodule	Audio	Text	Video
Submodule Size	3B	1.7B	5.5B
Layers	240	240	440
Attention Heads	16	16	16
Hidden Size	1024	768	1024
Sequence Length	264	77	264
Global Batch Size	128	128	128

which automatically searches schedules for the MLLM stage placement strategy. In the MLLM stage placement strategy, tensor parallelism of embedding layers is applied along the vocabulary dimension and distributed to the GPUs within the same pipeline parallelism group.

We search the space of the parameters (*pp*, *dp*, *mbs*) (for power-of-two) to find the best performance for each baseline. For a specific parameter combination (*pp*, *mbs*), Tessel automatically searches pipeline schedules. We illustrate the schedule searched by Tessel when *pp* = 4 in Figure 11.

FlexPipe searches the space described in Section 6. Parameters *pp*, *dp*, and *mbs* are enumerated in the same way as those of the baseline. Given a parameter combination (*pp*, *mbs*), FlexPipe automatically searches schedules and returns the alternatives with the least bubble ratio for different stage placement strategies. Specifically, the schedule space is defined by (*stage placement strategy*, *fstp*, *bstp*, *ctp*), each of which is enumerated from the corresponding built-in values in Table 2. A combination of V-shape and bidirectional stage placement strategies is also considered. Besides, we use tensor parallelism along the vocabulary dimension for each stage placement strategy to distribute the first and last layer over GPUs within the pipeline parallelism group. The *interval* is set to *pp* for the option (*breadth-first*, *interval*) and (*depth-first*, *interval*) of the stage traversal priority *fstp* and *bstp*. We illustrate the schedule searched by FlexPipe when *pp* = 4 in Figure 11. We evaluate the two types of schedules since the bubble ratios of the two are close. Besides, even though vChimera uses the bidirectional stage placement strategy, which shows fewer bubbles, its efficiency may degrade due to more synchronization between the last stages, especially with large GPUs.

**Figure 11.** Schedules explored by Tessel and FlexPipe.

We compare the search cost of Tessel and FlexPipe in Table 5, where the “X” mark represents that the search process fails. Tessel searches the repend, warm-up, and cool-down phase based on the Z3-solver. Besides, the search space shows explosive growth with the increased number of devices. The results show that the search cost by Tessel is the poorest among the three search methods, and the search process fails when the number of devices reaches 8. To improve the search cost by Tessel, Tessel-fast uses several algorithms to construct schedules shown in Figure 11. However, the search still fails when the number of devices reaches 16. FlexPipe uses the least search cost of the three search approaches. The search cost increases gently with the increased number of devices. It also searches for multiple efficient alternatives for different model configurations.

Table 5. Comparison of search cost by FlexPipe and Tessel.

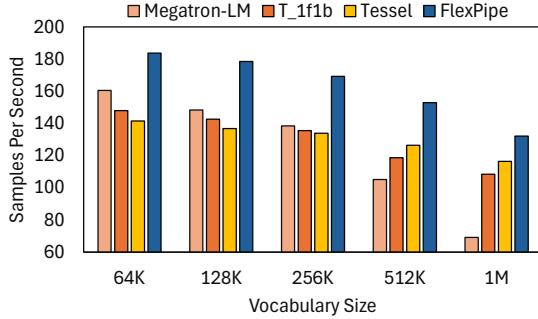
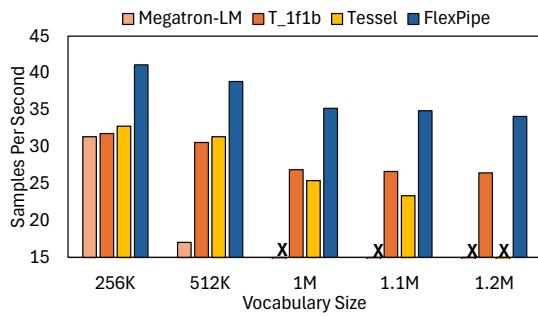
GPUs	Tessel	Tessel-fast	FlexPipe
2	1.29s	0.26s	4.73s
4	163.62s	11.1s	8.28s
8	X	729.74s	71.28s
16	X	X	107.34s
32	X	X	534.78s

End-to-end performance. Figure 12 compares the end-to-end performance results of 5B GPT models with varying vocabulary sizes from 64K to 1M on 32 GPUs. The experimental results demonstrate that FlexPipe gains 1.14×-1.91×, 1.20×-1.28×, and 1.13×-1.30× speedup compared with Megatron-LM, T-1F1B, and Tessel, respectively. The performance of Megatron-LM degrades due to the imbalanced workloads incurred by the large vocabulary size. The larger the vocabulary size, the more the performance declines. T-1F1B introduces bubbles because of data dependency between

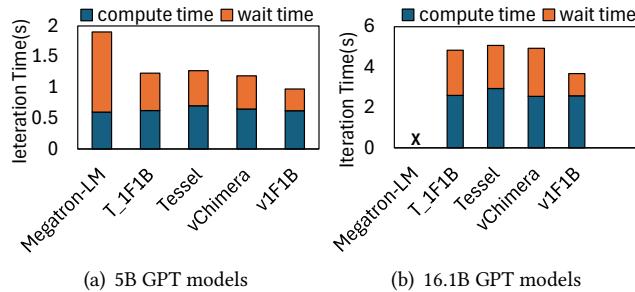
AllReduce operations due to the tensor parallelism of the Embedding layers. Tessel launches more in-flight micro-batches to avoid the bubbles similar to T-1F1B. However, more in-flight micro-batches are launched in the warm-up phase, introducing more bubbles in the warm-up and cool-down phase as shown in Figure 11. The schedules searched by FlexPipe show fewer bubbles and gain a prominent performance improvement compared to the schedules of the three baselines. FlexPipe utilizes a schedule space that ranges from a wider spectrum of schedule types. Besides, the searched alternatives are further optimized by gradient separation.

We also compare the end-to-end performance results of 16.1B GPT models with varying vocabulary sizes from 256K to 1.2M on 32 GPUs in Figure 13. The “X” mark represents that the out-of-memory issue is still encountered even if the recomputation is used. FlexPipe also shows superior performance to that of the other three baselines. More specifically, FlexPipe achieves 1.31×-2.28×, 1.27×-1.31×, and 1.24×-1.49× speedup compared with Megatron-LM, T-1F1B, and Tessel, respectively. Megatron-LM requires more memory consumption due to the large vocabulary size, and encounters the out-of-memory issue when the vocabulary size reaches 1M. It is worth noting that *pp* searches under 16 for Tessel since Tessel cannot support the automated schedule exploration when the number of GPUs reaches 16. Hence, the schedule by Tessel also encounters the out-of-memory issue when the vocabulary size reaches 1.2M.

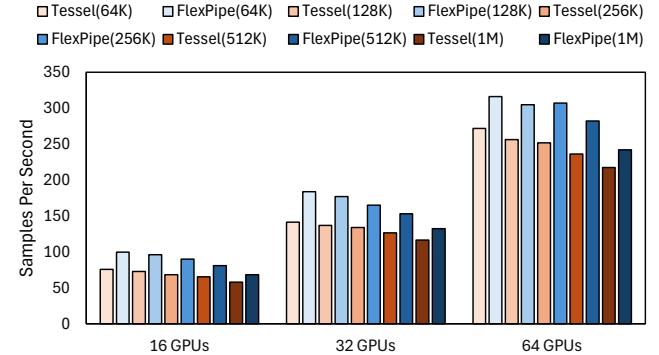
Performance breakdown. Bubbles are the key factor that affects the efficiency of pipeline parallelism. Bubbles can be caused by data dependency and communication operations such as AllReduce by the tensor parallelism and P2P communication to exchange activations and gradients. To accurately determine bubbles in the schedules, we profile the runtime of an iteration for each schedule, break down the execution into

**Figure 12.** End-to-end performance of 5B GPT models.**Figure 13.** End-to-end performance of 16.1B GPT models.

the stage computation time and device waiting time, and use the device waiting time to reflect bubbles. Figure 14 presents the runtime breakdown of the 5B and 16.1B models with a 1M vocabulary size, respectively. The computation time of different schedules does not differ much, since each schedule must compute the same whole model. The slight difference lies in the computation efficiency caused by different micro-batch sizes. A larger micro-batch size generally has higher computation efficiency. The device waiting time exhibits a noticeable difference among different schedules, consistent with the end-to-end performance results. Megatron-LM shows the largest device waiting time. The device waiting time of T-1F1B and Tessel is also larger than that of FlexPipe.

**Figure 14.** Runtime performance breakdown.

Parallel scalability. We present the weak scaling results on 5B GPT models in Figure 15. The number of GPUs increases from 16 to 64. Each schedule employs the same parameter configuration with which the schedule achieves optimal end-to-end performance results, but scales dp with an exact multiple of the number of GPUs. The experimental result shows that FlexPipe achieves superior performance with different hardware resources.

**Figure 15.** Weak scaling for 5B GPT models.

7.2 Optimizations on Multimodal Models

Compared with DistMM-Pipe, we use asynchronous communications to exchange the output states of submodules corresponding to different modalities. The asynchronous communication is registered as the new instruction *SyncWithGather* as elaborated in Figure 6. The instruction *SyncWithGather* is scheduled following each corresponding forward pass of the last pipeline stage for both modalities. Furthermore, we employ mixed schedules for different submodules to enhance efficiency. As in our experiments, we use interleaved 1F1B to calculate the audio/image modality, while applying 1F1B to the text modality. The schedule by FlexPipe is shown in Figure 9. Figure 16 presents the end-to-end performance of CLIP models on 40 GPUs. Compared with DistMM-Pipe, FlexPipe gains 1.26×, 1.29×, and 1.18× performance speedup for CLIP models with size 4.7B, 8.2B, and 9.7B, respectively. The efficiency of the models with the former two sizes is comparatively low due to the small hidden sizes and attention heads.

8 Related Works

Schedules and Optimizations. Existing research has primarily focused on handcrafting efficient schedules for performance. GPipe [11] leverages batch splitting to enable devices to work simultaneously. 1F1B [24] proposes early backward scheduling to reduce activation memory consumption. Subsequent schedules [17, 18, 23, 25, 27, 34] have built upon the two techniques to reduce bubbles further. Multiple works [2, 12, 21, 31] leverage recomputations in pipeline

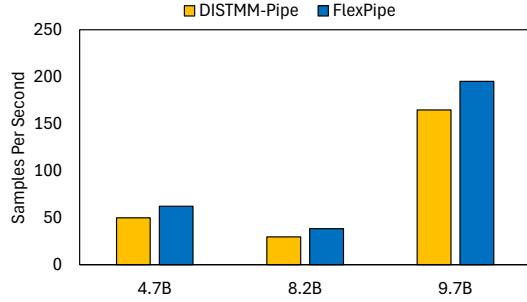


Figure 16. End-to-end performance of CLIP models.

parallelism methods to reduce memory consumption. Optimizations on long sequence scenarios are also proposed [19]. Bpipe [16] and Mpress [37] optimize the unbalanced memory requirements of different devices. Our work supports most existing schedules and can further integrate various optimization approaches mentioned above by implementing optimization passes.

Frameworks. DynaPipe [14] uses a dynamic micro-batching approach to the multi-task training of LLMs. Tessel [20] is a two-phase approach to explore efficient schedules for various stage placement strategies automatically. GraphPipe [13] uses a graph structure to partition DNN models to balance workloads. Our work covers a broader range of searchable schedule types than Tessel. Besides, our work can further use the partition algorithm by GraphPipe to improve efficiency.

9 Conclusion

FlexPipe explores efficient schedules automatically based on a succinct DSL and an automated scheduler with negligible overheads. It also provides programmable mechanisms to customize schedules for diverse model architectures swiftly. Evaluation results demonstrate prominent performance improvement compared with existing practices. We look forward to extending FlexPipe with more optimization approaches to enhance efficiency for various scenarios.

References

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
- [2] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 472–487.
- [3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [4] Megatron-LM Contributors. 2025. Megatron-LM & Megatron-Core. <https://github.com/NVIDIA/Megatron-LM>
- [5] John Crawford. 1990. The execution pipeline of the intel i486 cpu. In *1990 Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*. IEEE Computer Society, 254–255.
- [6] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).
- [7] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, et al. 2023. Palm-e: An embodied multimodal language model. (2023).
- [8] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
- [9] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.
- [10] J. Huang, Z. Zhang, S. Zheng, and et al. 2024. DISTMM: Accelerating Distributed Multimodal Model Training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1157–1171.
- [11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).
- [12] Yuzhou Huang, Yapeng Jiang, Zicong Hong, Wuhui Chen, Bin Wang, Weixi Zhu, Yue Yu, and Zibin Zheng. 2025. Obscura: Concealing Recomputation Overhead in Training of Large Language Models with Bubble-filling Pipeline Transformation. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 665–678.
- [13] Byungsoo Jeon, Mengdi Wu, Shiyi Cao, Sunghyun Kim, Sunghyun Park, Neeraj Aggarwal, Colin Unger, Daiyaan Arfeen, Peiyuan Liao, Xupeng Miao, et al. 2024. GraphPipe: Improving Performance and Scalability of DNN Training with Graph Pipeline Parallelism. *arXiv preprint arXiv:2406.17145* (2024).
- [14] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. 2024. DynaPipe: Optimizing multi-task training through dynamic pipelines. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 542–559.
- [15] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.
- [16] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. 2023. BPIPE: memory-balanced pipeline parallelism for training large language models. In *International Conference on Machine Learning*. PMLR, 16639–16653.
- [17] Joel Lamy-Poirier. 2023. Breadth-First Pipeline Parallelism. *Proceedings of Machine Learning and Systems* 5 (2023).
- [18] Shigang Li and Torsten Hoefer. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
- [19] Junfeng Lin, Ziming Liu, Yang You, Jun Wang, Weihao Zhang, and Rong Zhao. 2025. WeiPipe: Weight Pipeline Parallelism for Communication-Effective Long-Context Large Model Training. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 225–238.

- [20] Zhiqi Lin, Youshan Miao, Guanbin Xu, Cheng Li, Olli Saarikivi, Saeed Maleki, and Fan Yang. 2024. Tessel: Boosting Distributed Execution of Large DNN Models via Flexible Schedule Search. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 803–816.
- [21] Weijian Liu, Mingzhen Li, Guangming Tan, and Weile Jia. 2025. Mario: Near Zero-cost Activation Checkpointing in Pipeline Parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 197–211.
- [22] Yixin Liu, Kai Zhang, Yuan Li, Zhiling Yan, Chujie Gao, Ruoxi Chen, Zhengqing Yuan, Yue Huang, Hanchi Sun, Jianfeng Gao, et al. 2024. Sora: A review on background, technology, limitations, and opportunities of large vision models. *arXiv preprint arXiv:2402.17177* (2024).
- [23] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. 2023. Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.
- [24] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.
- [25] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. 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*. 1–15.
- [26] Penghui Qi, Xinyi Wan, Nyamdavaa Amar, and Min Lin. 2024. Pipeline Parallelism with Controllable Memory. *arXiv preprint arXiv:2405.15362* (2024).
- [27] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2023. Zero Bubble Pipeline Parallelism. *arXiv preprint arXiv:2401.10241* (2023).
- [28] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PMLR, 8748–8763.
- [29] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).
- [30] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multi-task learners. *OpenAI blog* 1, 8 (2019), 9.
- [31] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. 2024. AdaPipe: Optimizing Pipeline Parallelism with Adaptive Recomputation and Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 86–100.
- [32] Yu Tang, Lujia Yin, Qiao Li, Hongyu Zhu, Hengjie Li, Xingcheng Zhang, Linbo Qiao, Dongsheng Li, and Jiaxin Li. 2025. Koala: Efficient Pipeline Training through Automated Schedule Searching on Domain-Specific Language. *ACM Transactions on Architecture and Code Optimization* (2025).
- [33] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussonot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118* (2024).
- [34] Houming Wu, Ling Chen, and Wenjie Yu. 2024. BitPipe: Bidirectional Interleaved Pipeline Parallelism for Accelerating Large Models Training. *arXiv preprint arXiv:2410.19367* (2024).
- [35] Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. 2022. Coca: Contrastive captioners are image-text foundation models. *arXiv preprint arXiv:2205.01917* (2022).
- [36] Bo Zheng, Li Dong, Shaohan Huang, Saksham Singhal, Wanxiang Che, Ting Liu, Xia Song, and Furu Wei. 2021. Allocating large vocabulary capacity for cross-lingual language model pre-training. *arXiv preprint arXiv:2109.07306* (2021).
- [37] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. 2023. Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 556–569.