

RLinf: Flexible and Efficient Large-Scale Reinforcement Learning via Macro-to-Micro Flow Transformation

Chao Yu, **Yuanqing Wang**, Zhen Guo, Hao Lin, Si Xu, Hongzhi Zang, Quanlu Zhang,
Yongji Wu, Chunyang Zhu, Junhao Hu, Zixiao Huang, Mingjie Wei, Yuqing Xie, Ke Yang,
Bo Dai, Zhexuan Xu, Jiakun Du, Xiangyuan Wang, Xu Fu, Letong Shi, Zhihao Liu, Kang Chen,
Weilin Liu, Gang Liu, Boxun Li, Jianlei Yang, Zhi Yang, Guohao Dai, Yu Wang

2026/8/4

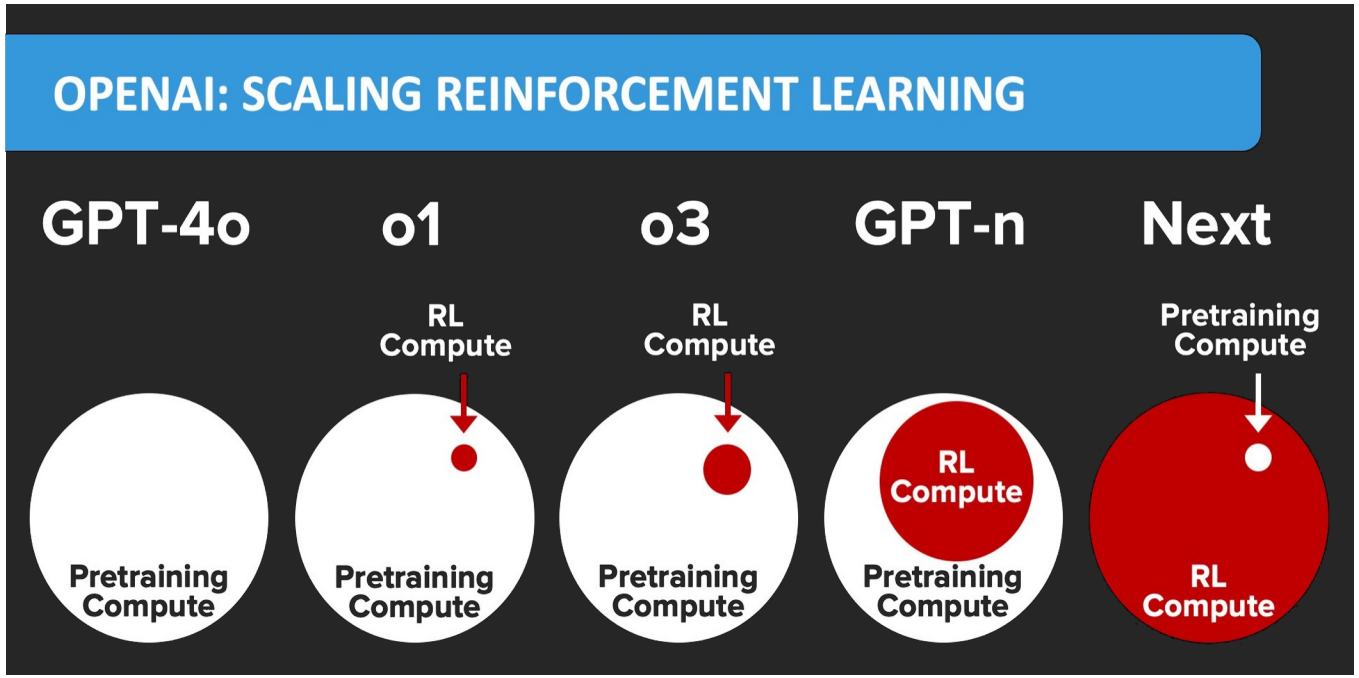
https://github.com/RLinf/RLinf

wangyuanqing@pku.edu.cn

Reinforcement learning: our next step towards AGI

OpenAI o1

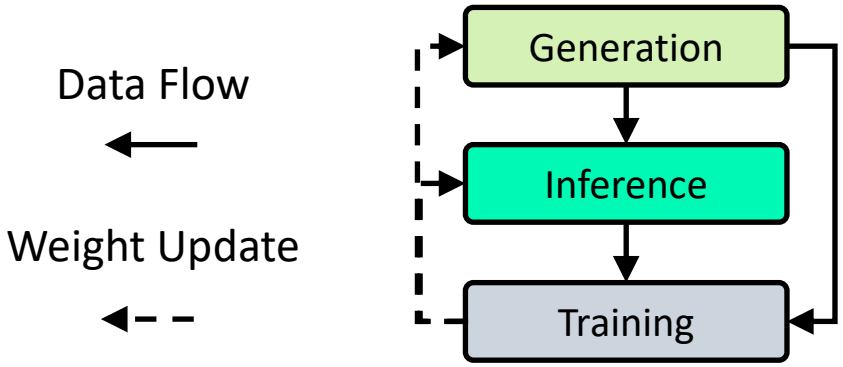
deepseek R1



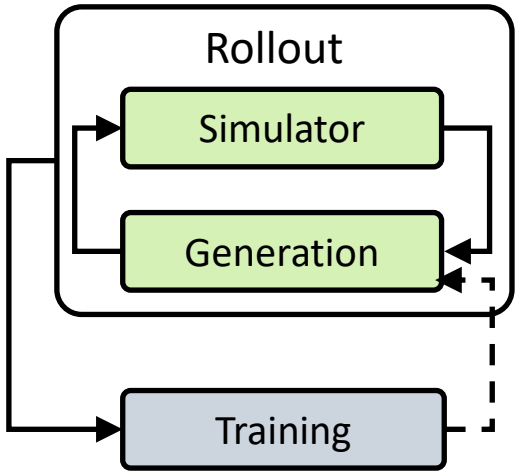
Past	Data-Driven Pre-training / Post-training
Present	Reward-Driven Reinforcement Learning
Future	Scaling RL in Computation beyond Pre-training

Modern RL workflows are highly diverse

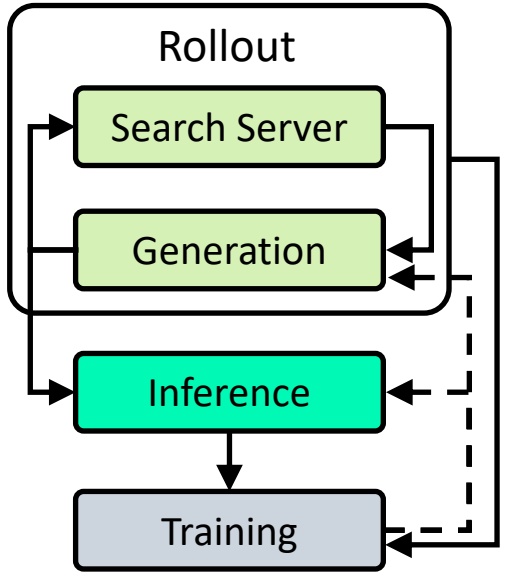
- Diverse scenarios



Reasoning RL

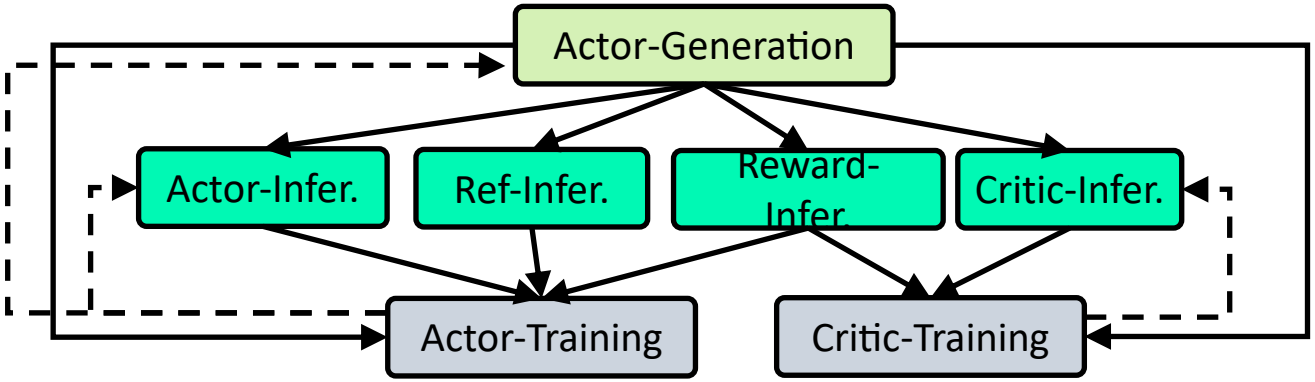


Embodied RL

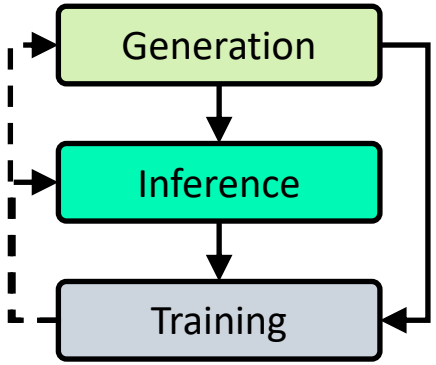


Agentic RL

- Diverse algorithms



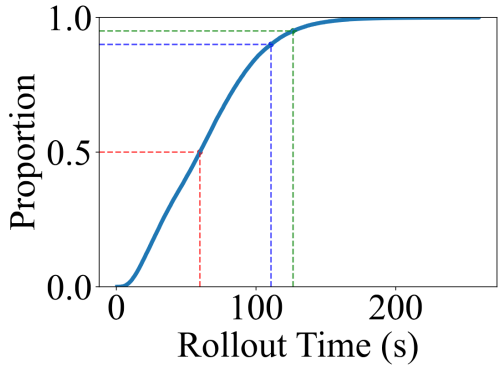
PPO



GRPO

RL components are highly heterogeneous

- ❑ LLM generation, inference, and training

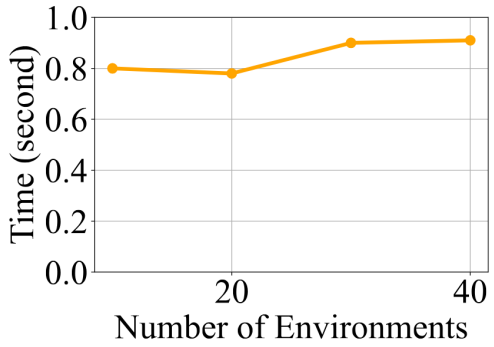


LLM generation: high dynamicity, long tail

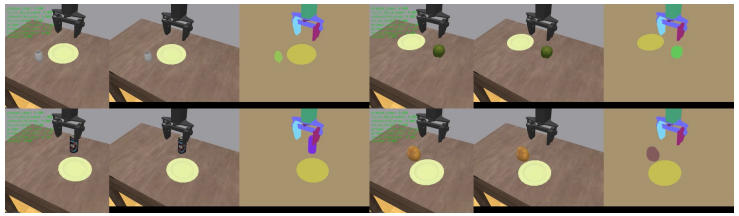
LLM training: more accelerator resources

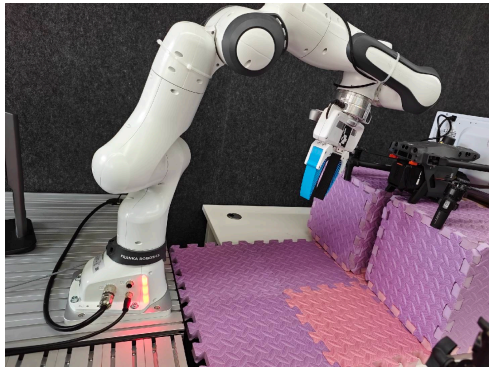
SGL vLLM NVIDIA MEGATRON-LM PyTorch

- ❑ Embodied simulator and physical robots



Simulator: 3D rendering pipeline



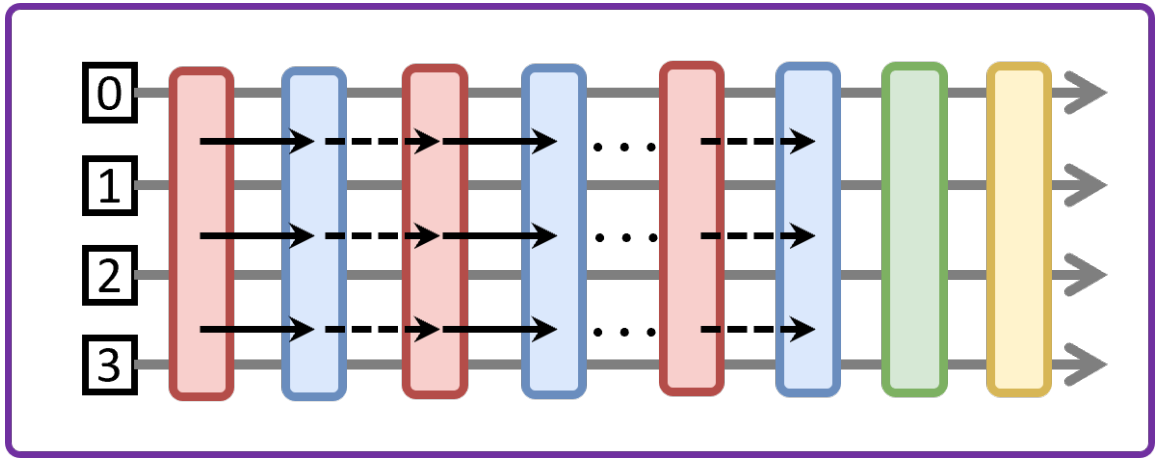


Physical robots: real-time constraint

Real-Time LINUX

- ❑ Agentic tool using

There is no one-fits-all execution plan

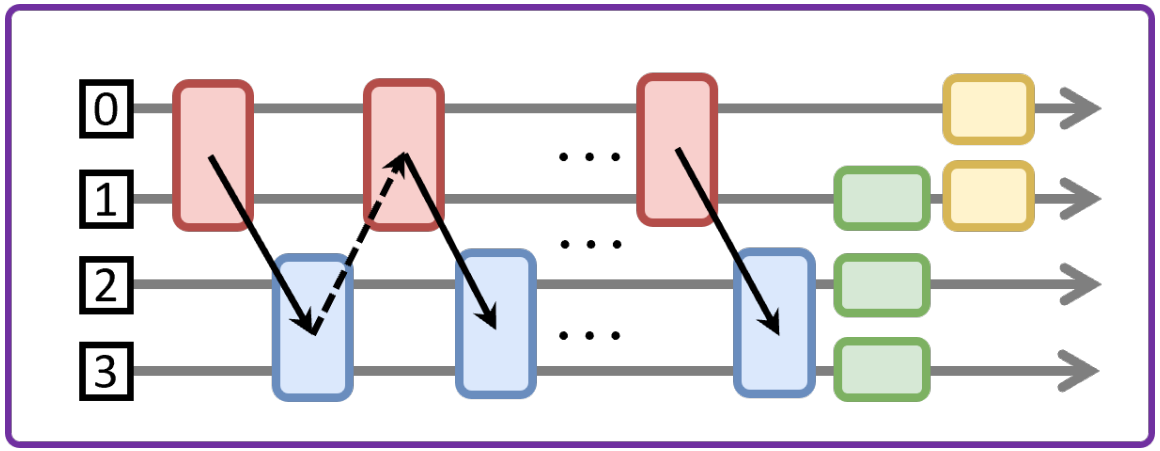


❑ Collocated mode: *veRL, Slime, ROLL*

+ Components can use all resources

- Underutilized during long-tail generation

- High component switching overhead



❑ Disaggregated mode: *AReal, AsyncFlow*

+ Mitigate long-tail computation waste

- High resource requirement

- Resource idling if without async algorithm

Our design philosophy

Flexibility as the key to efficiency

Flexibility == Complexity?

Contributions

- **M2Flow**: A novel RL system design and programming paradigm
  - Macro-to-micro flow transformation that **decouples programming and optimization**
  - Easy-to-program **macro workflow** is recomposed into an optimized **micro execution flow**

- **RLinf**: A highly flexible and scalable RL framework based on M2Flow
  - **Elastic pipelining** and **auto context switching** for realizing M2Flow transformation
  - **Profiling-guided scheduling** for generating optimized execution plan
  - **Adaptive communication** for efficient interactions among components
  - A **Worker abstraction** that enables all the above through one-line class inheritance
  - Rich support for reasoning/embodied/agentic RL workflows, models and algorithms
  - Fully open-sourced at https://github.com/RLinf/RLinf

Macro workflow construction with RLinf

❑ Typical reasoning RL workflow

```python
def run(self, batch_iterator):
    for batch in batch_iterator:
        self._update_rollout_weights()

        self.data_ch.put(batch)

        self.rollout_group.generate(
            in_channel=self.data_ch,
            out_channel=self.rollout_ch,
        )

        self.inference_group.inference(
            in_channel=self.rollout_ch,
            out_channel=self.inference_ch
        )

        self.actor_group.train(
            in_channel=self.inference_ch
        ).wait()
```

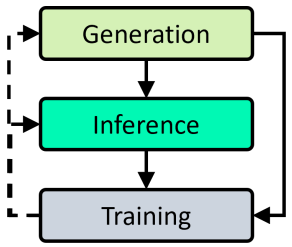
❑ Typical embodied RL workflow

```python
def run(self, num_iters, num_steps):
    for _ in range(num_iters):
        self._update_rollout_weights()
        self.observation_ch.put(None)

        for _ in range(num_steps):
            self.rollout_group.generate(
                in_channel=self.observation_ch,
                out_channel=self.action_ch,
                train_channel=self.train_ch,
            )

            handle = self.env_group.step(
                in_channel=self.action_ch,
                out_channel=self.observation_ch,
                train_channel=self.train_ch,
            )

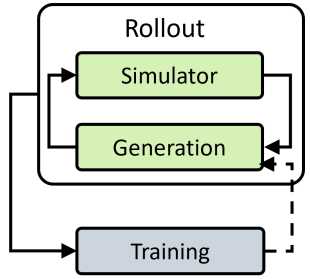
        handle.wait()
        self.actor_group.train(self.train_ch).wait()
```



Data flow expressed via distributed data channels

self.rollout_group Collective management knob of component processes

handle.wait() Synchronization point in the inherently async workflow



Worker and *WorkerGroup* construction

❑ Easy *Worker* main logic construction

```
class RolloutWorker(Worker):
    def generate(self, in_channel: Channel, out_channel: Channel):
        with out_channel.device_lock:
            batch_size = 0
            while batch_size < self.total_batch_size:
                batch = in_channel.get()
                res = self.model.gen(batch)
                out_channel.put(res)
                batch_size += batch.size
```

Resource protection

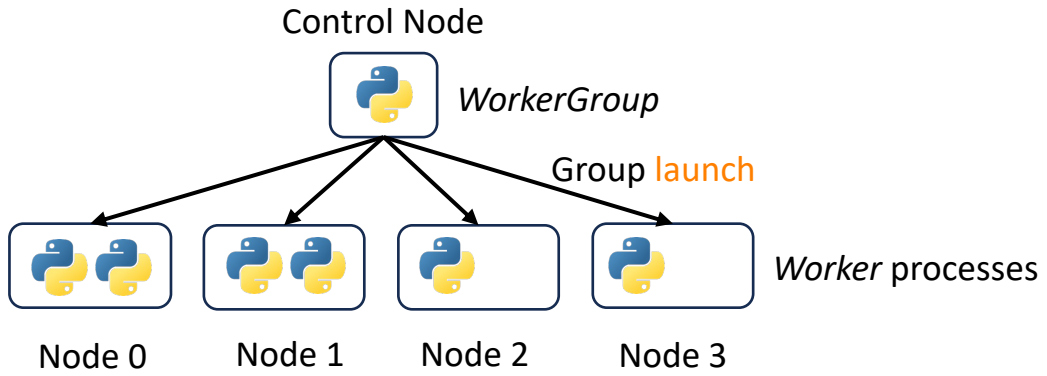
1. Get input data
2. Operate the data
3. Put output data

Almost all components can be modeled in this I/O pattern

❑ Collective *Worker* process management with *WorkerGroup*

```
class RLWorkflowRunner:
    def __init__(self, placement):
        cluster = Cluster(num_nodes=4)
        self.rollout_group = RolloutWorker.create_group().launch(cluster, placement)
```

Remotely launch worker processes in the cluster



Worker and *WorkerGroup* construction

- Easy *Worker* main logic construction

```
class RolloutWorker(Worker):
    def generate(self, in_channel: Channel, out_channel: Channel):
        with out_channel.device_lock:
            batch_size = 0
            while batch_size < self.total_batch_size:
                batch = in_channel.get()
                res = self.model.gen(batch)
                out_channel.put(res)
                batch_size += batch.size
```

Resource protection

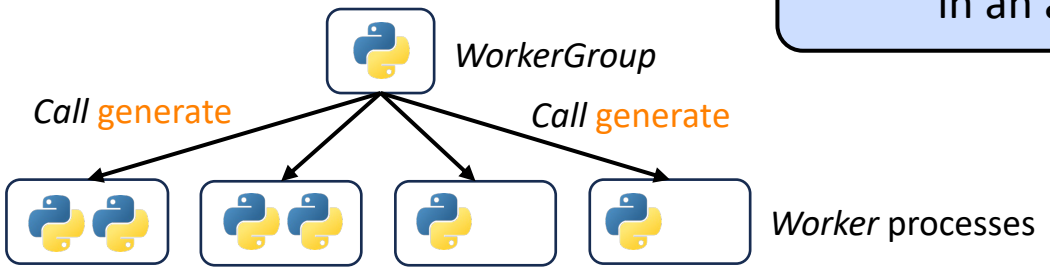
1. Get input data
2. Operate the data
3. Put output data

Almost all components can be modeled in this I/O pattern

- Collective *Worker* process management with *WorkerGroup*

```
class RLWorkflowRunner:
    def __init__(self, placement):
        cluster = Cluster(num_nodes=4)
        self.rollout_group = RolloutWorker.create_group().launch(cluster, placement)
        handle = self.rollout_group.generate(...)
        handle.wait() # [ret_val_rank_0, ret_val_rank_1, ...]
```

Execute worker functions via *WorkerGroup* in an async and SPMD manner



Worker and *WorkerGroup* construction

❑ Easy *Worker* main logic construction

```
class RolloutWorker(Worker):
    def generate(self, in_channel: Channel, out_channel: Channel):
        with out_channel.device_lock:
            batch_size = 0
            while batch_size < self.total_batch_size:
                batch = in_channel.get()
                res = self.model.gen(batch)
                out_channel.put(res)
                batch_size += batch.size
```

Resource protection

1. Get input data
2. Operate the data
3. Put output data

Almost all components can be modeled in this I/O pattern

❑ Collective *Worker* process management with *WorkerGroup*

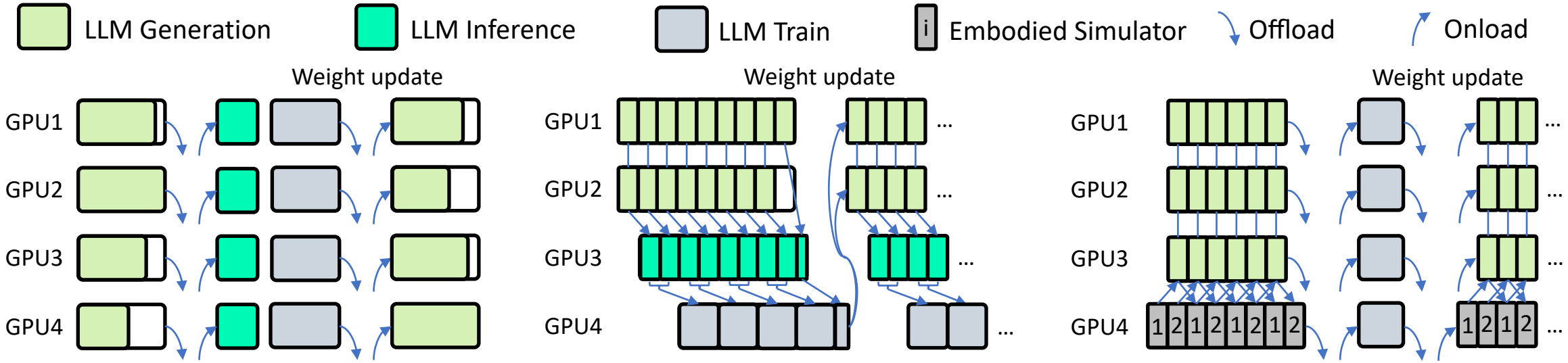
```
class RLWorkflowRunner:
    def __init__(self, placement):
        cluster = Cluster(num_nodes=4)
        self.rollout_group = RolloutWorker.create_group().launch(cluster, placement)
        handle = self.rollout_group.generate(...)
        handle.wait() # [ret_val_rank_0, ret_val_rank_1, ...]
```

❑ Flexible placement and heterogenous cluster support

- ➢ Each worker process can be assigned **any accelerator(s) of any node** across cluster
- ➢ Accelerator can be **heterogeneous** (e.g., mixing CPU, GPU and NPU nodes) across cluster

M2Flow transformation

❑ How to achieve fine-grained scheduling with ***the same workflow code***?



Temporal Scheduling
via **Auto Context Switching**

Temporal multiplexing by putting workers on the same devices

Spatial Scheduling
via **Elastic Pipelining**

Spatially disjoint workers form pipelines with varying data granularity

Hybrid Spatial and
Temporal Scheduling

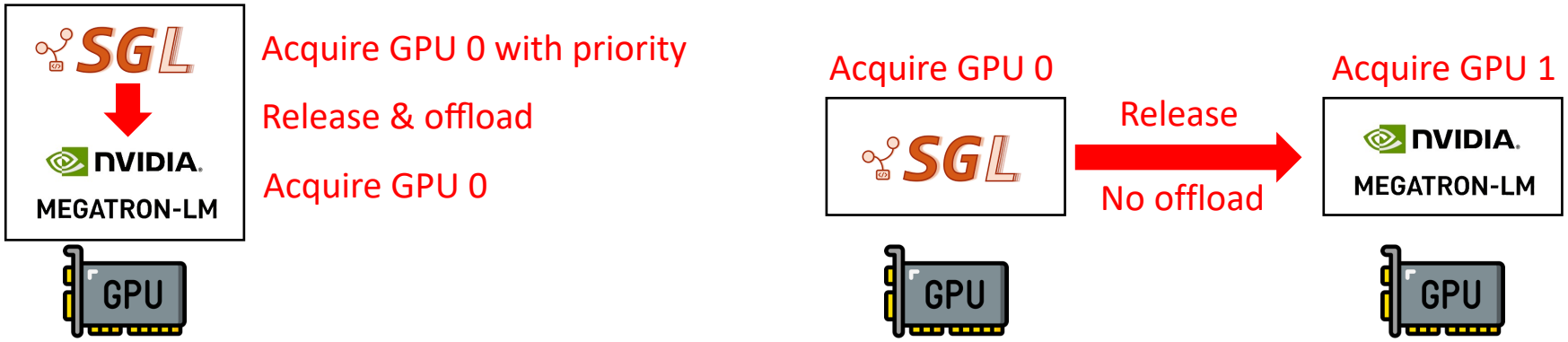
Arbitrary combination of spatial and temporal scheduling of workers

M2Flow transformation: automatic context switching

❑ **Global device lock** for managing concurrent accelerator resource accesses

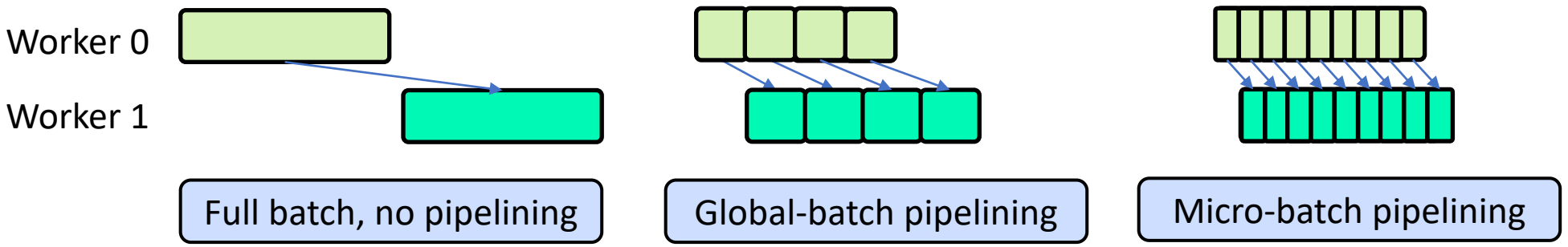
```
class RolloutWorker(Worker):
    def generate(self, in_channel: Channel, out_channel: Channel):
        with out_channel.device_lock:
            batch_size = 0
            while batch_size < self.total_batch_size:
```

- A distributed lock primitive for acquiring and releasing accelerators
- Workers should implement their **onload** and **offload** interfaces
- A **priority lock** based on the data flow direction of the associated channel



M2Flow transformation: elastic pipelining

❑ Data processing granularity as the pipelining granularity



➢ The flow-like interface naturally enables pipelining at varying data granularity

```
prompts = [
    "Write a short, neutral self-introduction for a fictional character. Hello, my name is",
    "Provide a concise factual statement about France’s capital city. The capital of France is",
    "Explain possible future trends in artificial intelligence. The future of AI is",
]
llm.async_generate(prompts, sampling_params)
```

SGLang’s support for batch generation

➢ **Performance factor**: larger batch can better utilize accelerators, but leads to more bubbles

➢ **Semantic factor**: algorithm semantic like batch-wise normalization requires larger batch

Profiling-guided scheduling policy

```
for G_s, G_t in TraverseStCuts (G) do
    /* G_s and G_t share the same gpus */
    T_s, S_s ←FindSchedule (G_s, N, D_table);
    T_t, S_t ←FindSchedule (G_t, N, D_table);
    D_table[(G_s,N)]←(T_s, S_s);
    D_table[(G_t,N)]←(T_t, S_t);
    if T_best > T_s + T_t then
        T_best ←T_s+T_t; S_best ←shared(S_s, S_t);
    end
    /* G_s and G_t use different devices */
    for N_s, N_t in TraverseGpuNum (N) do
        /* N_s+N_t equals N */
        T_s, S_s ←FindSchedule (G_s, N_s, D_table);
        T_t, S_t ←FindSchedule (G_t, N_t, D_table);
        D_table[(G_s,N_s)]←(T_s, S_s);
        D_table[(G_t,N_t)]←(T_t, S_t);
        if T_best >PipeliningTime (T_s, T_t) then
            T_best ←PipeliningTime (T_s, T_t);
            S_best ←pipeline(S_s, S_t);
        end
    end
end
```

Traverse the sub-graph of the workflow graph

Profiling-guided scheduling policy

```
for G_s, G_t in TraverseStCuts (G) do
    /* G_s and G_t share the same gpus */
    T_s, S_s ←FindSchedule (G_s, N, D_table);
    T_t, S_t ←FindSchedule (G_t, N, D_table);
    D_table[(G_s,N)]←(T_s, S_s);
    D_table[(G_t,N)]←(T_t, S_t);
    if T_best > T_s + T_t then
        T_best ←T_s+T_t; S_best ←shared(S_s, S_t);
    end
    /* G_s and G_t use different devices */
    for N_s, N_t in TraverseGpuNum (N) do
        /* N_s+N_t equals N */
        T_s, S_s ←FindSchedule (G_s, N_s, D_table);
        T_t, S_t ←FindSchedule (G_t, N_t, D_table);
        D_table[(G_s,N_s)]←(T_s, S_s);
        D_table[(G_t,N_t)]←(T_t, S_t);
        if T_best >PipeliningTime (T_s, T_t) then
            T_best ←PipeliningTime (T_s, T_t);
            S_best ←pipeline(S_s, S_t);
        end
    end
end
```

Traverse the sub-graph of the workflow graph

Compare temporal *vs.* spatial scheduling

Model performance under different accelerator resources with profiling data

Profiling-guided scheduling policy

```
for G_s, G_t in TraverseStCuts (G) do
    /* G_s and G_t share the same gpus */
    T_s, S_s ←FindSchedule (G_s, N, D_table);
    T_t, S_t ←FindSchedule (G_t, N, D_table);
    D_table[(G_s,N)]←(T_s, S_s);
    D_table[(G_t,N)]←(T_t, S_t);
    if T_best > T_s + T_t then
        T_best ←T_s+T_t; S_best ←shared(S_s, S_t);
    end
    /* G_s and G_t use different devices */
    for N_s, N_t in TraverseGpuNum (N) do
        /* N_s+N_t equals N */
        T_s, S_s ←FindSchedule (G_s, N_s, D_table);
        T_t, S_t ←FindSchedule (G_t, N_t, D_table);
        D_table[(G_s,N_s)]←(T_s, S_s);
        D_table[(G_t,N_t)]←(T_t, S_t);
        if T_best >PipeliningTime (T_s, T_t) then
            T_best ←PipeliningTime (T_s, T_t);
            S_best ←pipeline(S_s, S_t);
        end
    end
end
```

Traverse the sub-graph of the workflow graph

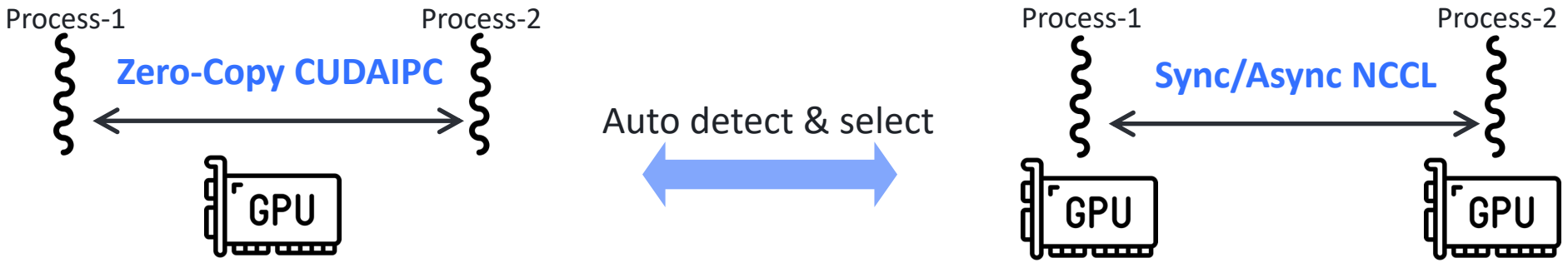
Compare temporal *vs.* spatial scheduling

Model performance under different accelerator resources with profiling data

Combine the best policy of each sub-graph

Adaptive communication

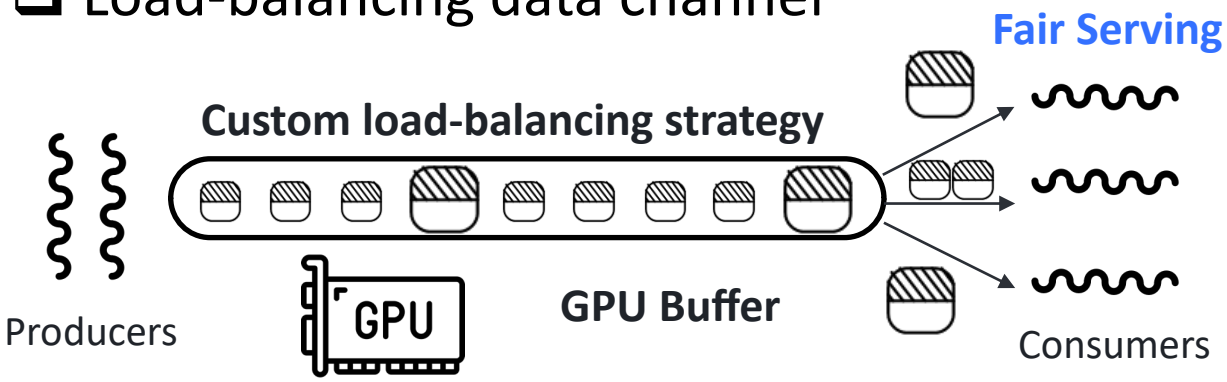
❑ Adaptive cudaIPC/NCCL communication



Zero-copy cudaIPC when sharing GPUs

NCCL-based sync/async NVLINK/GDR communications

❑ Load-balancing data channel



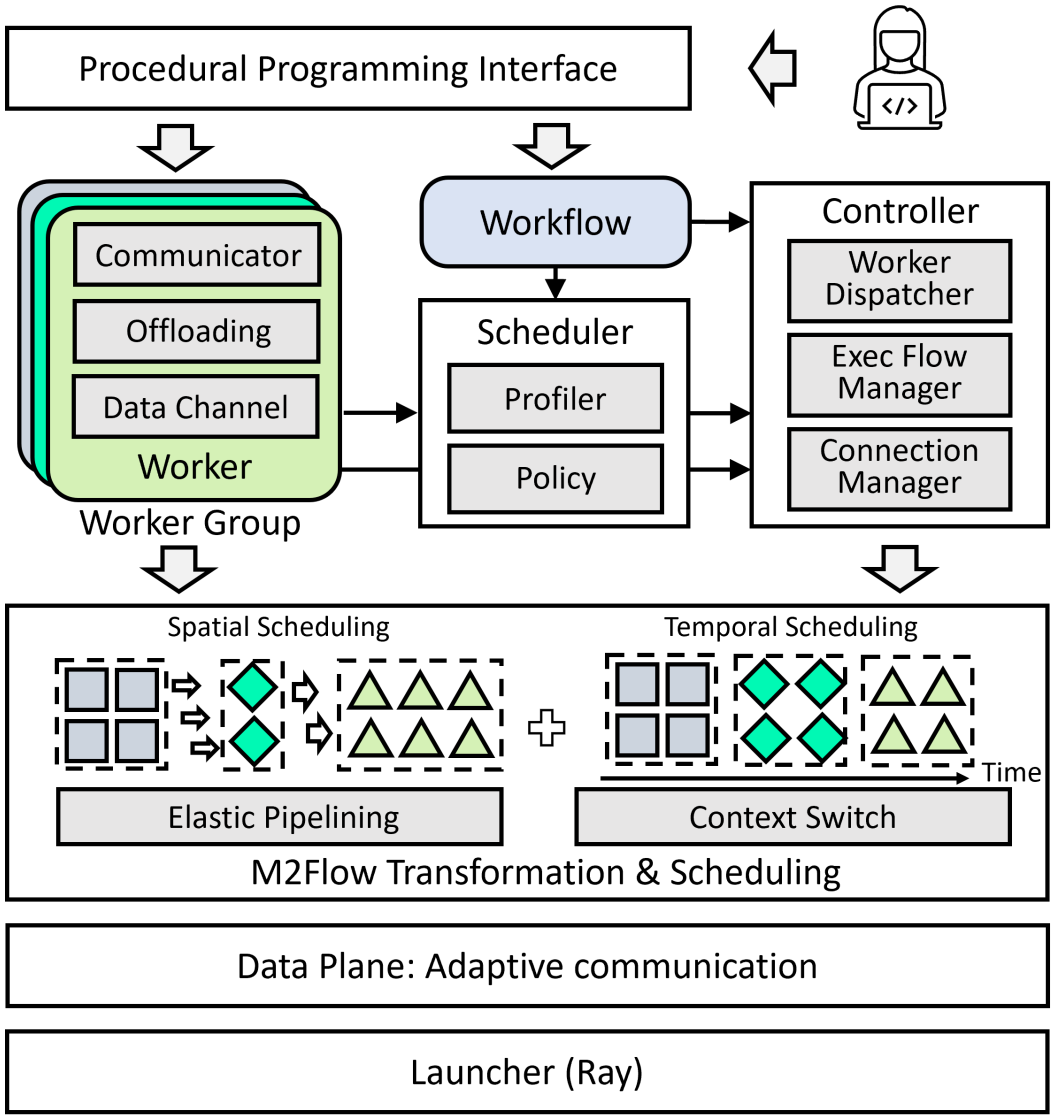
Flexible inter-worker data exchange utility

❑ Auto connection management

No need to create connection

```python
class MyWorker(Worker):
    def hello(self):
        if self.rank == 0:
            msg = "Hello from rank 0"
            self.send(msg, dst_group_name="MyWorker", dst_rank=1)
        elif self.rank == 1:
            msg = self.recv(src_group_name="MyWorker", src_rank=0)
            print(msg)  # Hello from rank 0
```

RLinf architecture



Worker programming

```python
class RolloutWorker(Worker):
    def generate(self, in_channel: Channel, out_channel: Channel):
        with out_channel.device_lock:
            batch_size = 0
            while batch_size < self.total_batch_size:
                batch = in_channel.get()
                res = self.model.gen(batch)
                out_channel.put(res)
                batch_size += batch.size
```

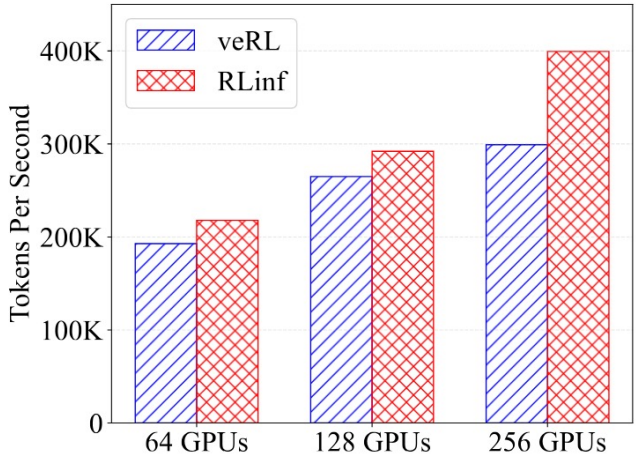
Workflow programming

```python
class RLWorkflowRunner:
    def __init__(self, placement):
        cluster = Cluster(num_nodes=4)
        self.rollout_group = RolloutWorker.create_group().launch(cluster, placement)
        self.inference_group = InferenceWorker.create_group().launch(cluster)
        self.actor_group = ActorWorker.create_group().launch(cluster)
        self.data_ch = Channel.create("Data")
        self.rollout_ch = Channel.create("Rollout")
        self.inference_ch = Channel.create("Inference")

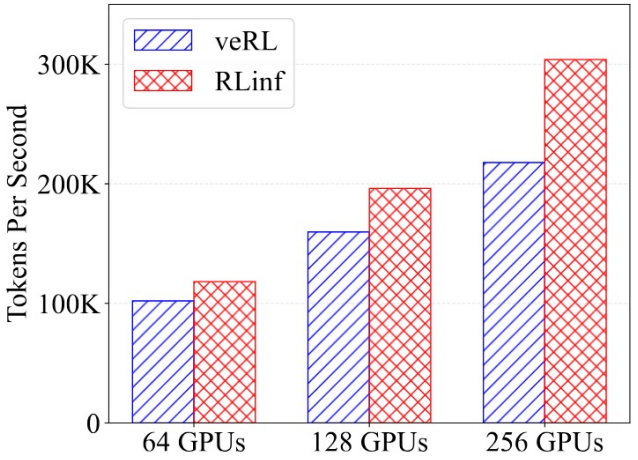
    def run(self, batch_iterator):
        for batch in batch_iterator:
            self._update_rollout_weights()
            self.data_ch.put(batch)
            self.rollout_group.generate(self.data_ch, self.rollout_ch)
            self.inference_group.inference(self.rollout_ch, self.inference_ch)
            self.actor_group.train(in_channel=self.inference_ch).wait()
```

Evaluation: system performance

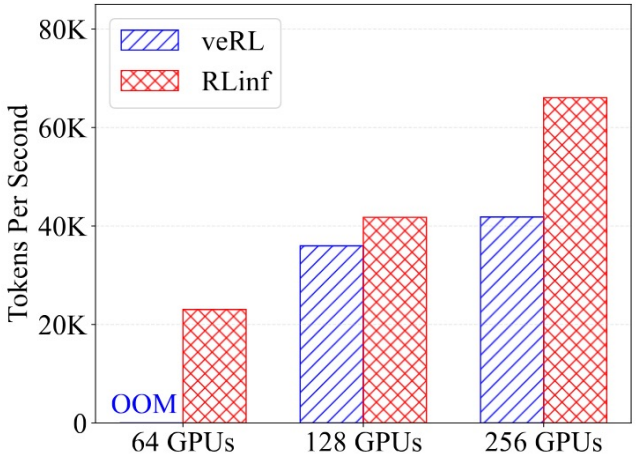
- RLinf consistently outperforms state-of-the-art RL systems by **1.1×~2.13×** on a variety of **reasoning and embodied models** under different cluster scales.



(a) 1.5B

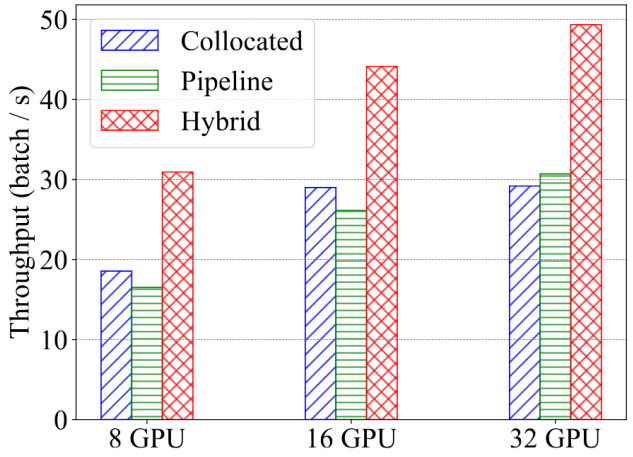


(b) 7B

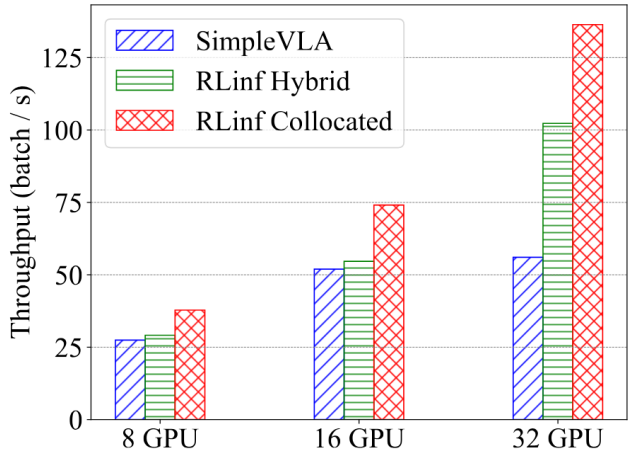


(c) 32B

Model	Environment
Qwen2.5-1.5B	-
Qwen2.5-7B	-
Qwen2.5-32B	-
OpenVLA	ManiSkill
OpenVLA-OFT	LIBERO



(a) ManiSkill



(b) LIBERO

Evaluation: model performance

- Models trained with RLinf can achieve SOTA performance on various datasets

Model	AIME 24	AIME 25	GPQA-diamond	Average
1.5B models				
DeepSeek-R1-Distill-Qwen-1.5B (base model)	28.33	24.90	27.45	26.89
DeepMath-1.5B	37.80	30.42	32.11	33.44
DeepScaleR-1.5B-Preview	40.41	30.93	27.54	32.96
AReaL-1.5B-Preview-Stage-3	40.73	31.56	28.10	33.46
AReaL-1.5B-retrain*	44.42	34.27	33.81	37.50
FastCuRL-1.5B-V3	43.65	32.49	35.00	37.05
RLinf-math-1.5B	**48.44**	**35.63**	**38.46**	**40.84**
7B models				
DeepSeek-R1-Distill-Qwen-7B (base model)	54.90	40.20	45.48	46.86
AReaL-boba-RL-7B	61.66	49.38	46.93	52.66
Skywork-OR1-7B	66.87	52.49	44.43	54.60
Polaris-7B-Preview	**68.55**	51.24	43.88	54.56
AceMath-RL-Nemotron-7B	67.30	**55.00**	45.57	55.96
RLinf-math-7B	68.33	52.19	**48.18**	**56.23**

Math reasoning model performance

Evaluation: model performance

- Models trained with RLinf can achieve SOTA performance on various datasets

Model	Algorithm	Vision	Semantic	Position	Average
gen-robot/openvla-7b-rlvla-rl	PPO	76.6%	75.4%	77.6%	76.1%
RLinf-OpenVLAOFT-GRPO-ManiSkill3-25ood	GRPO	**84.6%**	51.6%	42.9%	61.5%
RLinf-OpenVLAOFT-PPO-ManiSkill3-25ood	PPO	80.5%	56.6%	56.1%	64.5%
RLinf-OpenVLA-GRPO-ManiSkill3-25ood	GRPO	74.7%	74.4%	81.6%	75.5%
RLinf-OpenVLA-PPO-ManiSkill3-25ood	PPO	82.0%	**80.6%**	**89.3%**	**82.2%**

Model	LIBERO				
	Spatial	Object	Goal	Long	Avg.
Octo	78.9	85.7	84.6	51.1	75.1
OpenVLA	84.7	88.4	79.2	53.7	76.5
Nora	92.2	95.4	89.4	74.6	87.9
π_0+FAST	96.4	96.8	88.6	60.2	85.5
π_0	96.8	98.8	95.8	85.2	94.2
UniVLA	96.5	96.8	95.6	92.0	95.2
OpenVLA-OFT One-Trajectory SFT					
Baseline	56.5	25.6	45.6	9.7	34.4
RLinf	99.0	99.0	99.0	94.4	97.9
Δ	+42.5	+73.4	+53.4	+84.7	+63.5

Embodied model performance on ManiSkill3 and LIBERO

Conclusions

- ❑ M2Flow: a flexible RL system paradigm that decouples programming and execution
- ❑ Elastic pipelining and auto context switching mechanisms for M2Flow transformation
- ❑ Profiling-guided scheduling policy to compose optimized execution plan
- ❑ RLinf: a flexible and efficient RL system based on M2Flow
- ❑ https://github.com/RLinf/RLinf. Welcome your issues, PRs and star!



