NeMo RL

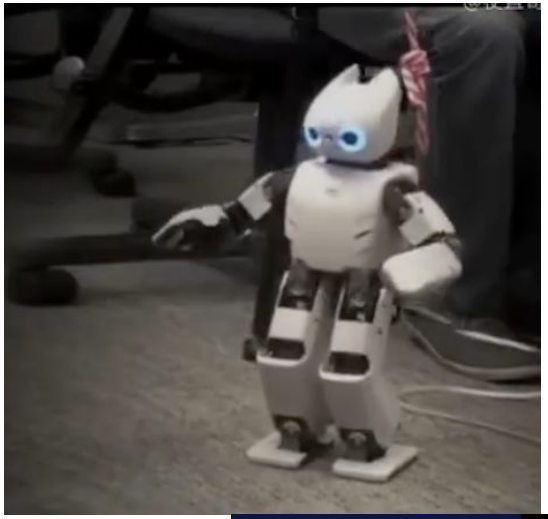
Efficient Reinforcement Learning from single to thousands of GPUs

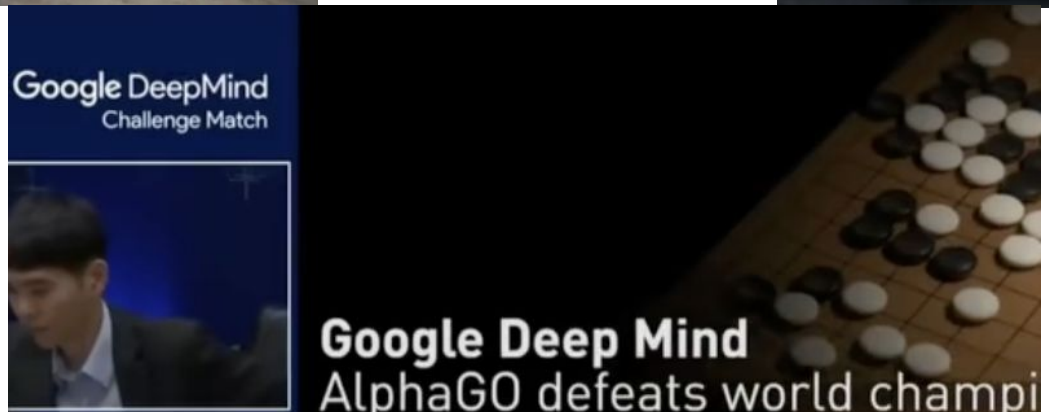
Zhiyu Li
Wenwen Gao

Agenda

- What is Reinforcement Learning and NeMo RL Overview
- Refit Optimizations for large MoEs
- Other on-going work in NeMo RL
  - FP8
  - AsyncRL
  - NeMo Gym
- Future Roadmaps

Reinforcement Learning Everywhere

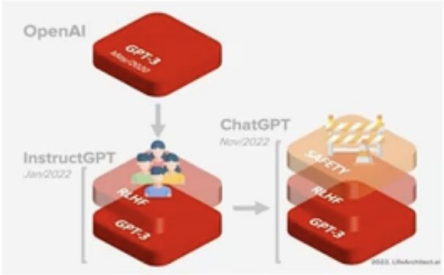




Reinforcement Learning in LLM

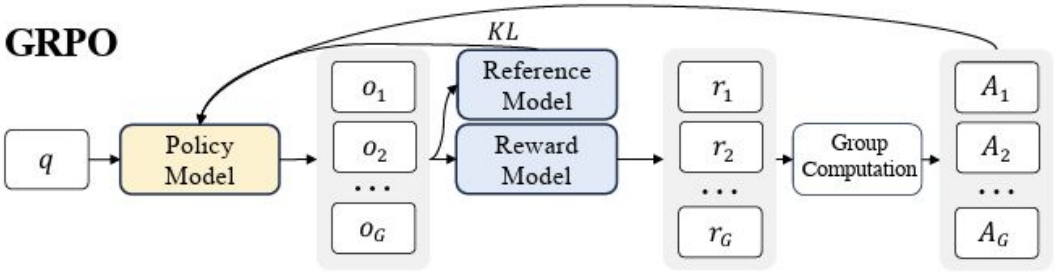
ChatGPT

- Used Reinforcement Learning with Human Feedback (RLHF) PPO



Deepseek-R1

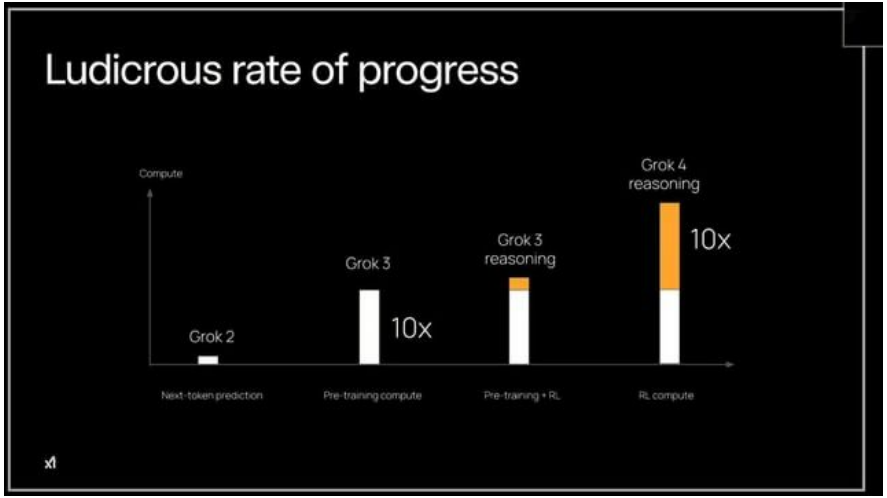
- Used GRPO
- Achieved SOTA accuracy with very limited compute resource

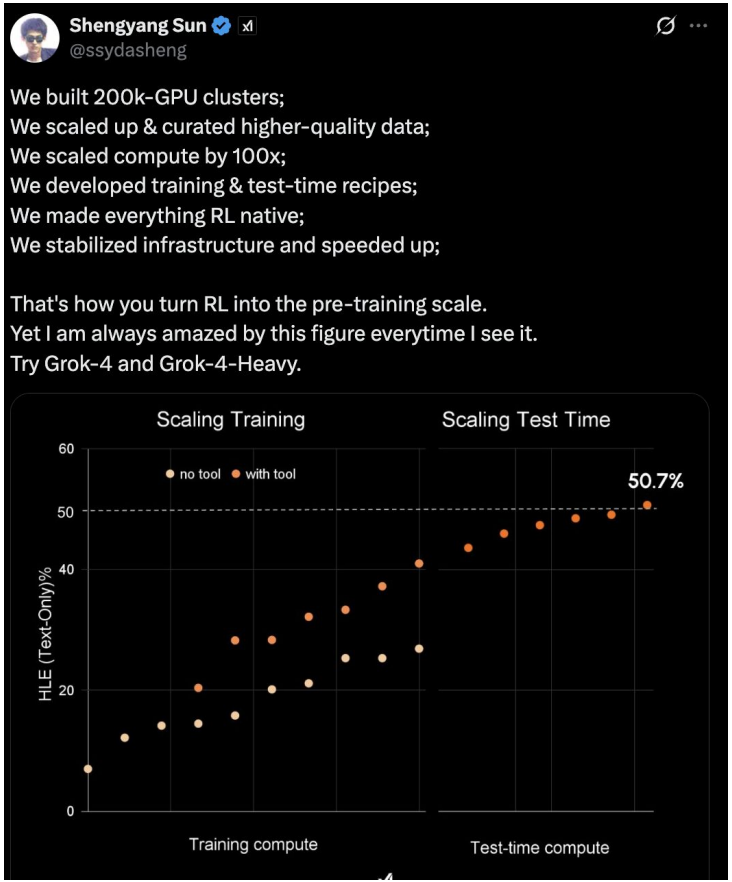


Qwen3-32B

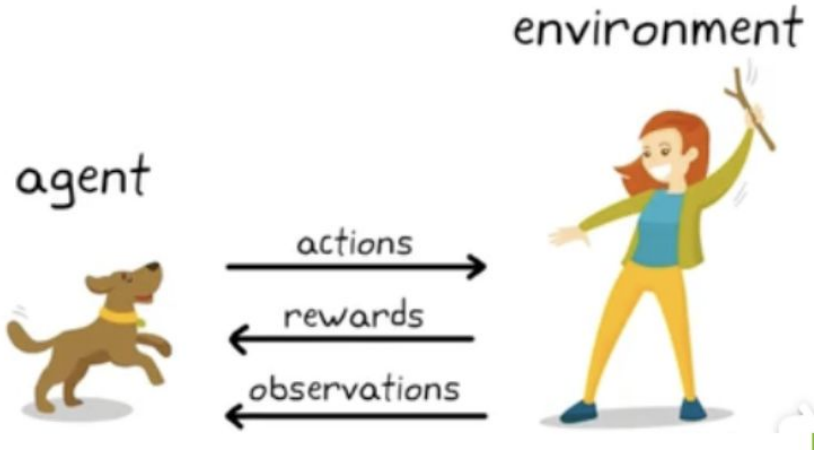
Reinforcement Learning in LLM - Grok 4

- Half of the compute (200k-GPU clusters) is spent on RL!
  - Another use for the AI Factory
- Note the author of the post used to work for NVIDIA!





What is Reinforcement Learning

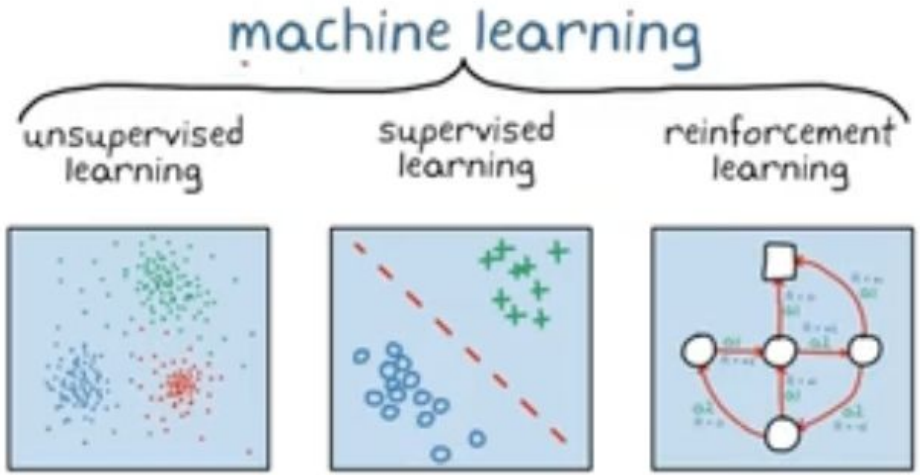


Translating that to the world of LLM

- Agent: model
- Action: give a response based on a prompt
- Rewards:
  - Verifiers - Math, Coding
  - Reward models - any LLMs as a judge
- Environment
  - The whole system of prompts and rewards

Why is Reinforcement Learning so important for LLM

- We are approaching limit to data
  - Hence unsupervised learning (pretraining), and supervised learning (SFT) will approach flatline in its efficacy
- RL is less dependent on data
  - LLM can generate data, including thought process i.e. Chain-of-thought reasoning
- RL enables exploration learning



Nemo RL

Implement SOTA Techniques with high throughput and ease of use

SOTA Techniques

- SFT
- DPO
- GRPO
- DAPO
- PRIME (upcoming)

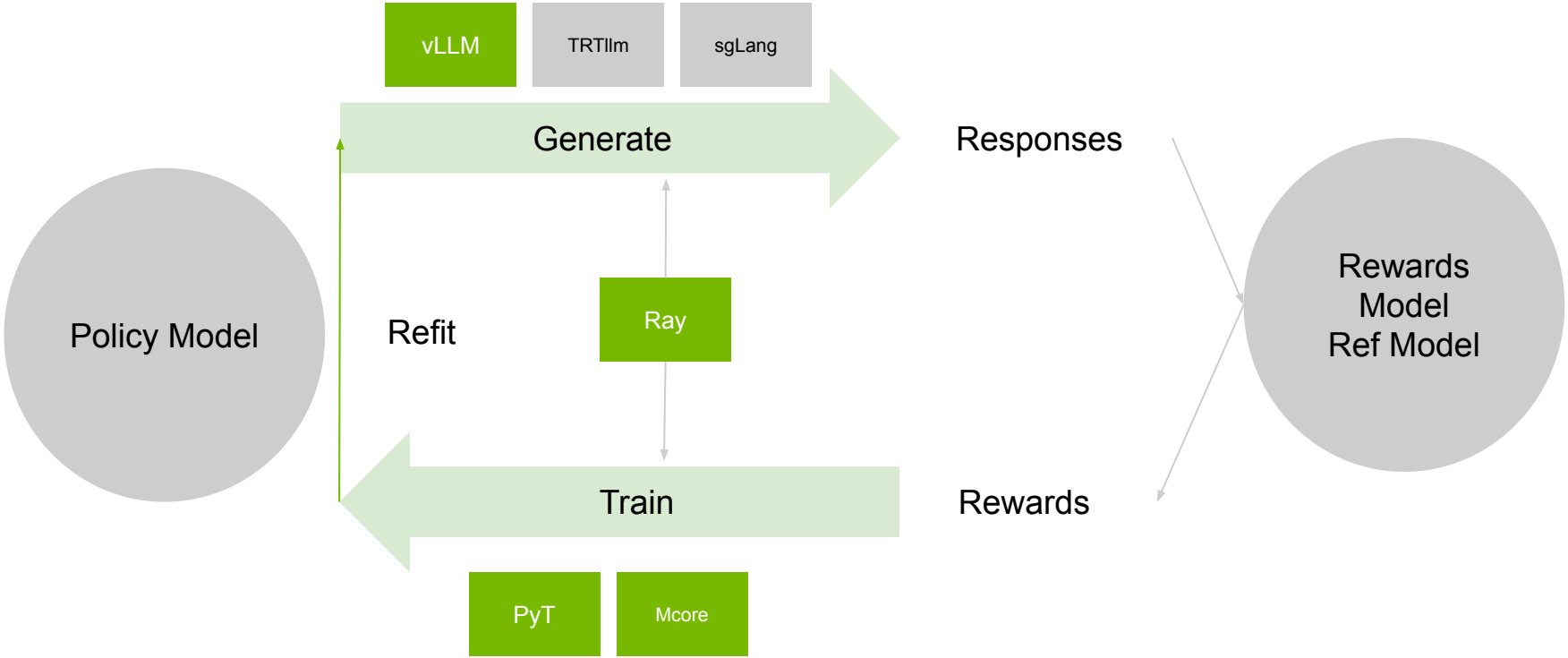
Performance & Scalability

- Low precision generation - FP8 and FP4 upcoming
- 4D parallelism
- Long sequence support via Context Parallelism and Sequence packing

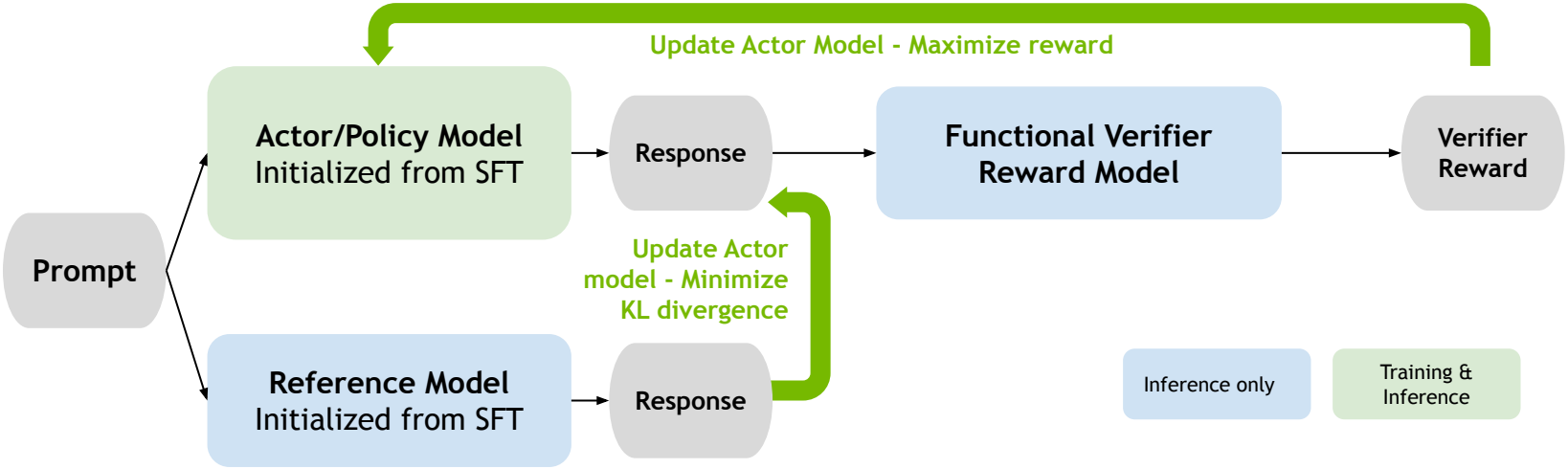
Ease of Use

- Natively support HF ecosystem with PyT FSDP2 and vLLM
- Ray for orchestration
- uv to create isolated virtual environments despite potentially conflicting dependencies

Nemo RL - Multiple training and generation backends for Usability + Scalability



Algorithm Support
GRPO with Functional Verifier



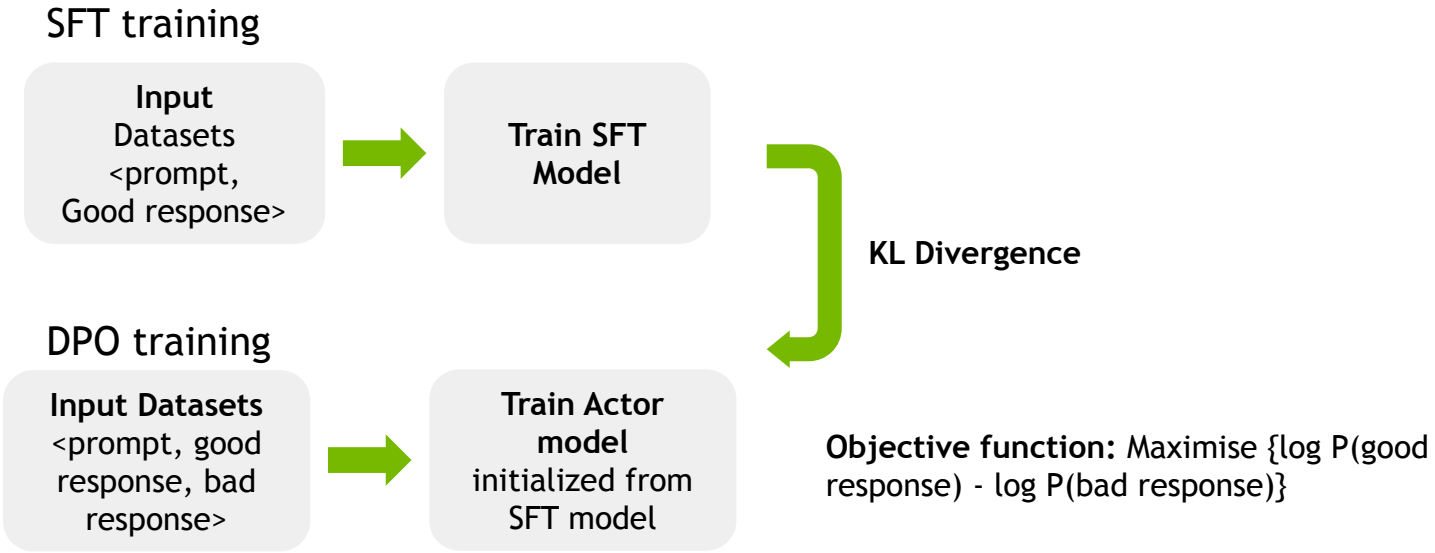
1. Sample N prompts and generate one response for each from the Policy Network
2. For each (prompt, response) pair, compute (a) the response likelihood under the initial policy, (b) the verifier reward of the response
3. Update the Actor to increase the response likelihood when the verifier gives yes, without straying too far away from the initial policy
4. Go back to step 1

Algorithm Support
DPO

Better response in summarization and single-turn dialogue with simpler implementation

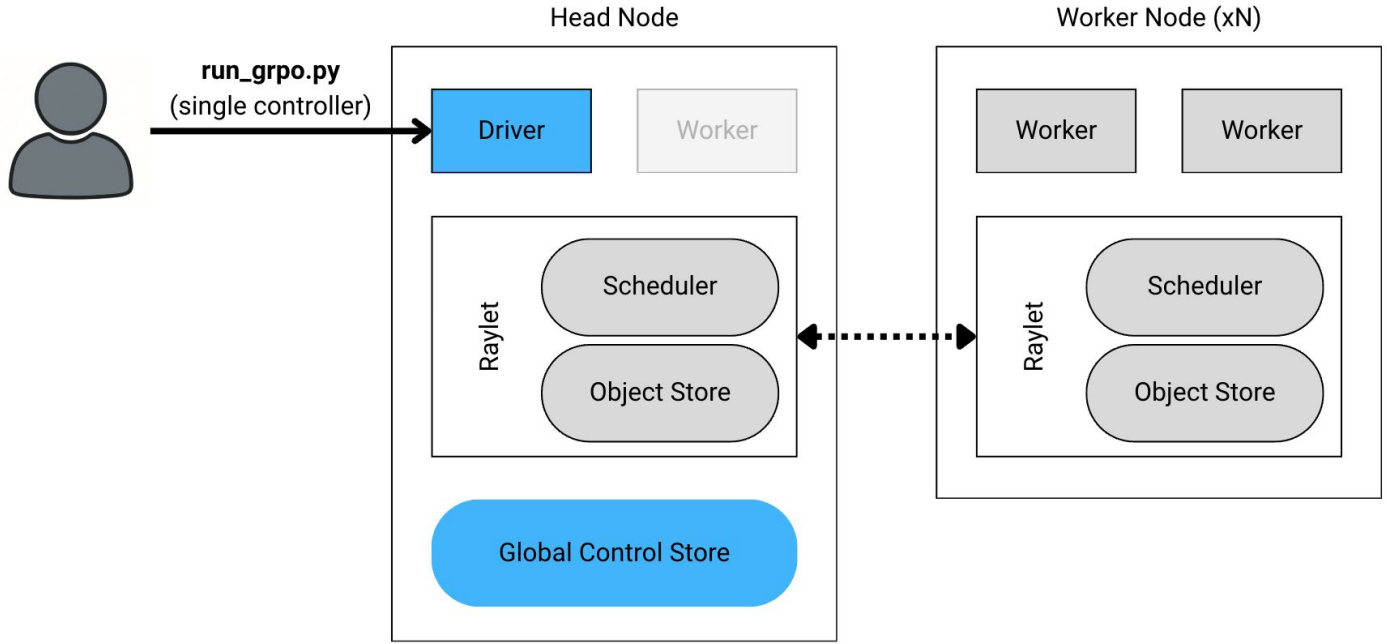
Mapping reward functions and optimal policies directly with a simple classification objective.

Trained on Anthropic dataset.



Engineering Highlights
Ray for Orchestration, Single Controller

- Ray is an abstraction on top of slurm/k8s that we use for scheduling
- It follows the single-controller pattern: driver gives workers instructions



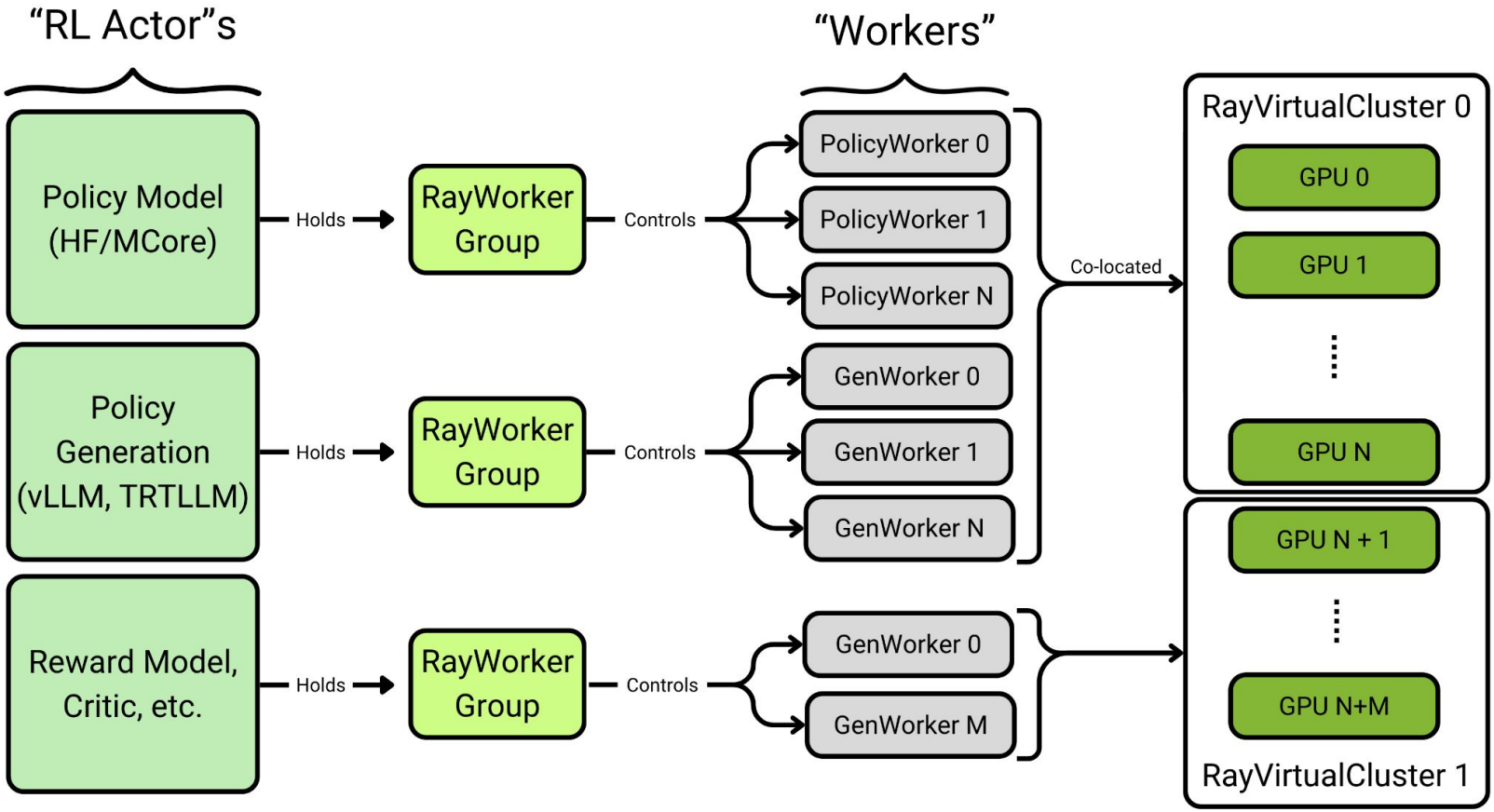
Engineering Highlights
Ray + uv

- uv is a super fast python package manager
- It allows us to quickly create isolated virtual environments for potentially conflicting environments
- Ray natively integrates with uv to make it easy to propagate project dependencies to workers
- uv keeps our dependencies in sync

```
# Local run
uv run examples/run_grpo.py

# Multi-node run (same UX)
uv run examples/run_grpo.py
```

NeMo RL (abstractions)



Motivation for tracking logprob error

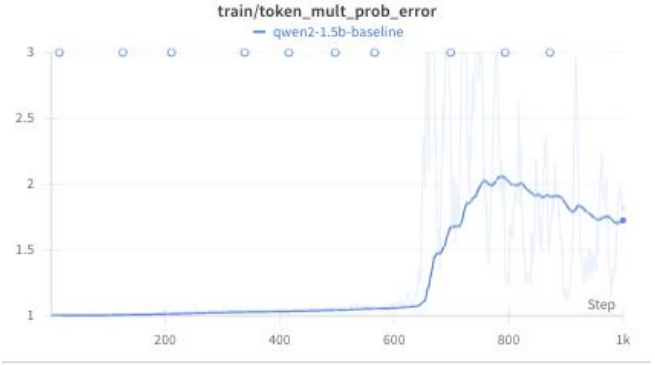
- On-policy RL requires the inference policy implementation to match the training model implementation
- Training can be unstable, or collapse, if this difference is too large
- NeMo RL measures this deviation throughout training by calculating the **log probability error**

$$\frac{1}{n}\sum_{i=1}^{n(\text{tokens})}\exp\left(\left\|\text{logprobs-train-fwk}_i - \text{logprobs-inference-fwk}_i\right\|\right)$$

- Ideally close to **1.0**, anything beyond **1.05** is considered “large” and warrants investigation
- Simple metric has caught many bugs!

Motivation for tracking logprob error

- Qwen2-1.5B divergence caught with our metric
- Training for a few steps did not catch this (needed >600 steps)
- Without metric, just appears as poor choice of hyperparameters
- Root caused to tied-embedding configuration difference between training and inference

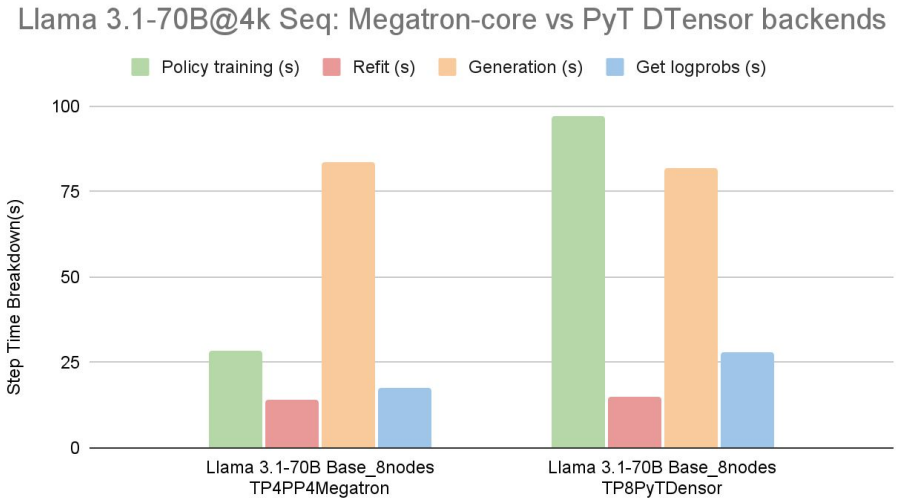


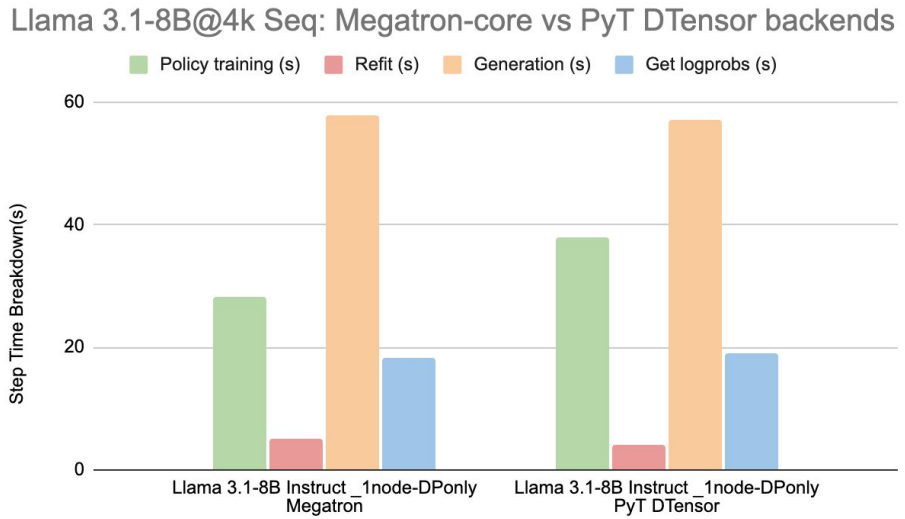
Model/Seq Support Matrix in PyT backend

	8k	16k	24k	32k	64k
4B					
7B/8B					
32B					
30B-moe					

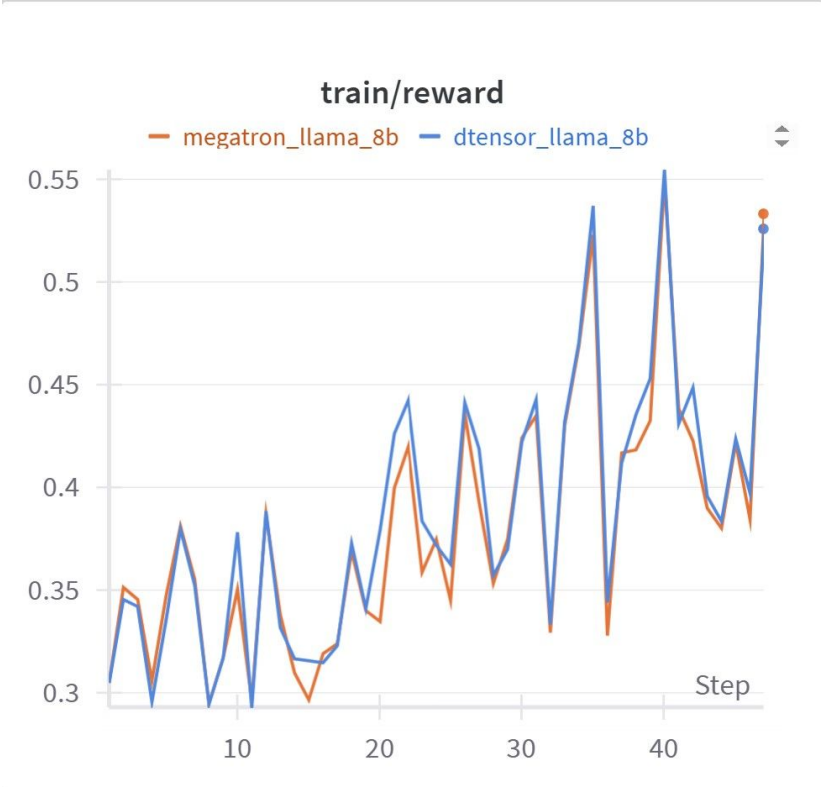
Verified on Nemo-RL PyT/DTensor

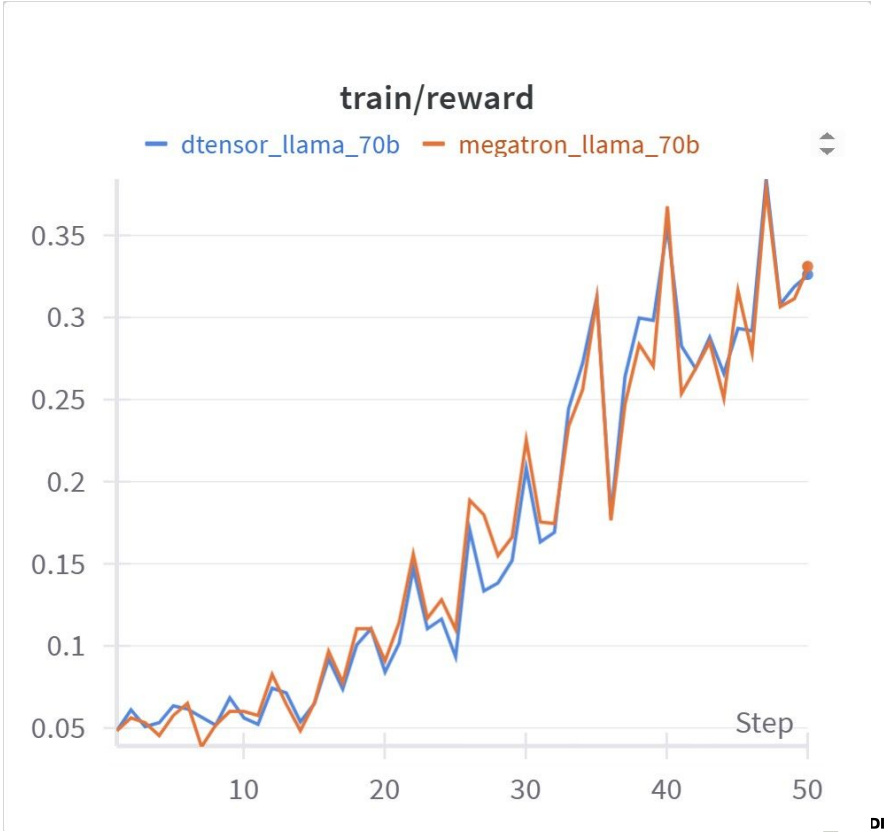
Nemo RL Performance: Megatron-core vs PyT DTensor



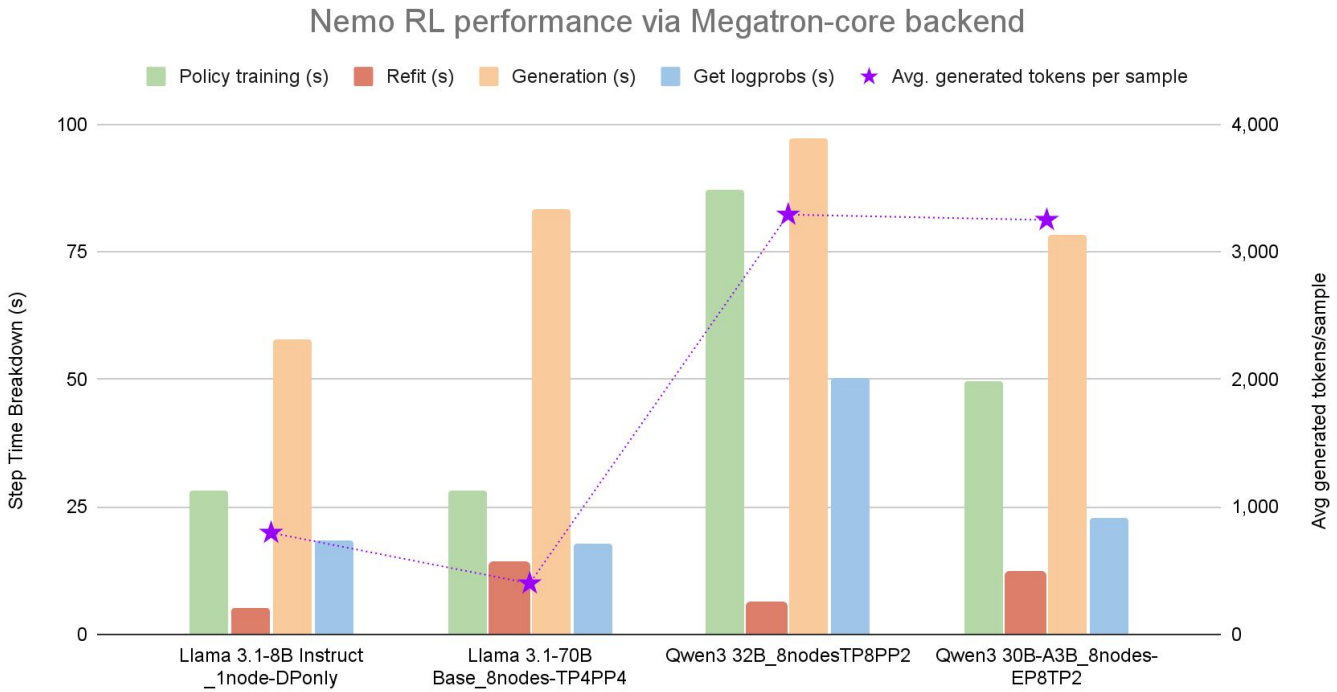


Accuracy Curves - Megatron-core backend matching PyT Dtensor backends





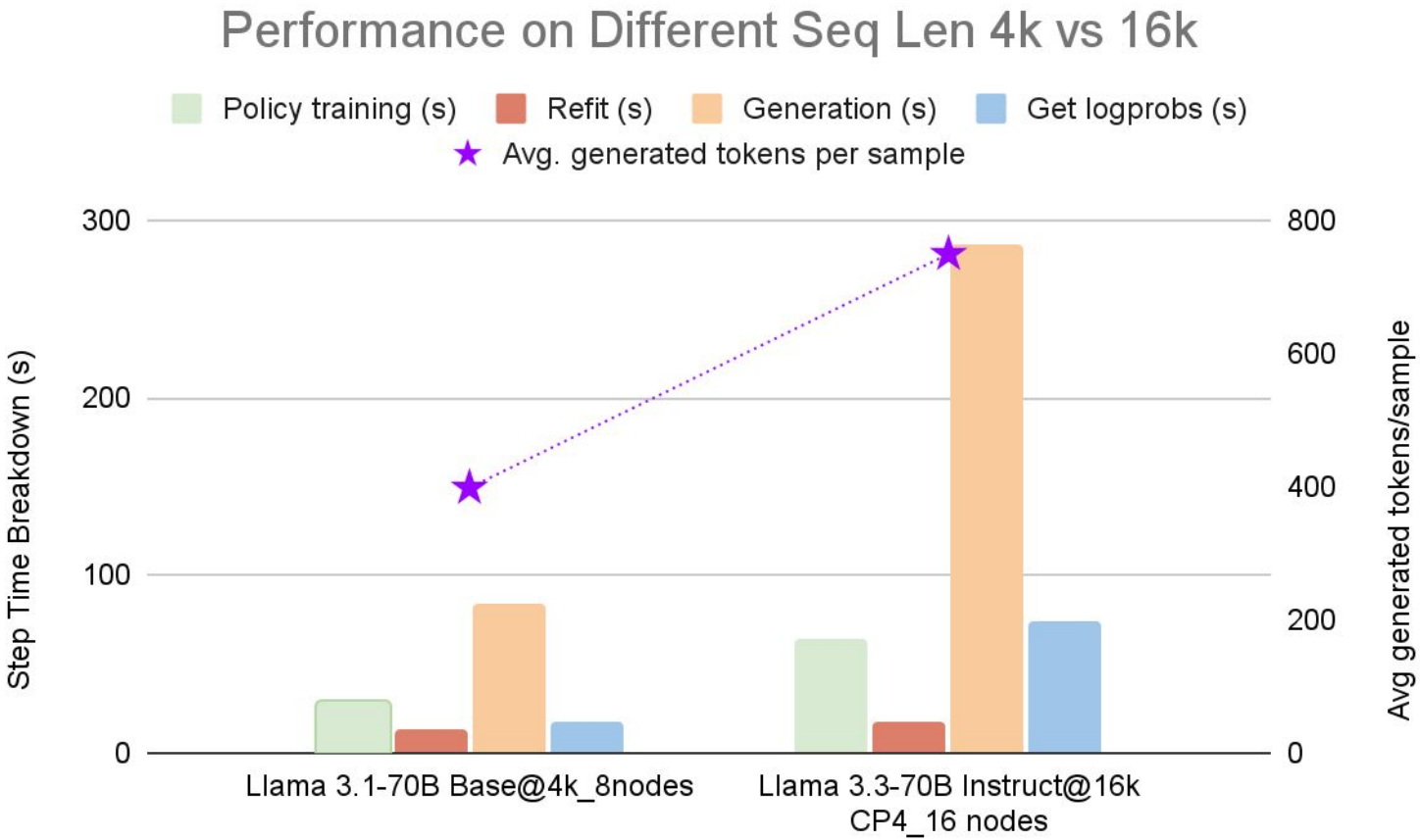
Nemo RL Performance by Model



Config

- max sequence length 4096
- rollout batch size 2048
- global batch size 512

Nemo RL Performance: Long Context Support



Agenda

- What is Reinforcement Learning and NeMo RL Overview
- Refit Optimizations for large MoEs
- Other on-going work in NeMo RL
  - FP8
  - AsyncRL
  - NeMo Gym
- Future Roadmaps

Brief Overview of Refit

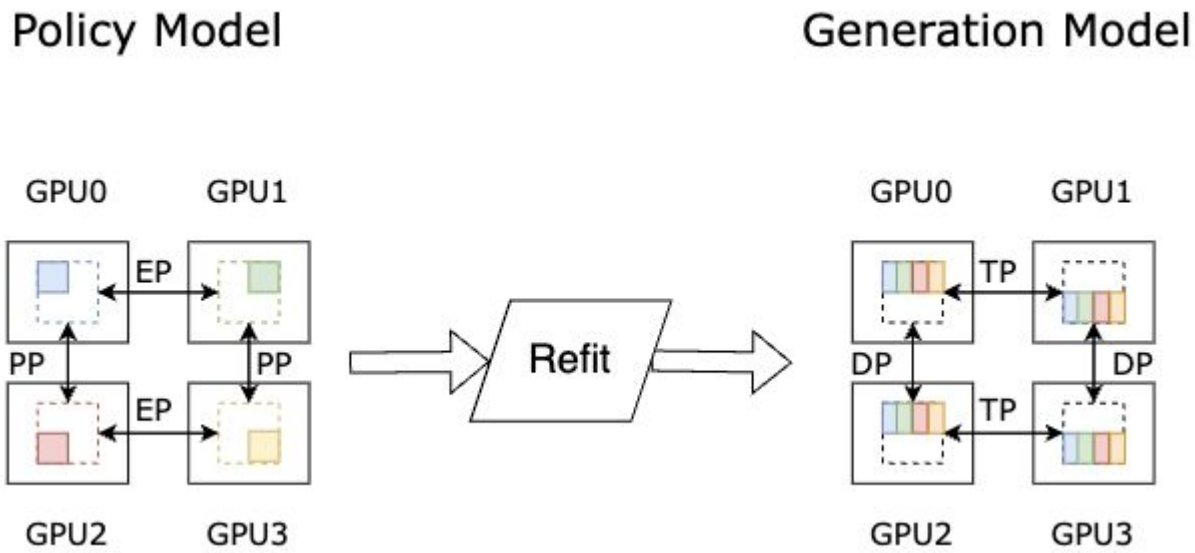
📍 什么是Refit？

在 RL 训练中：

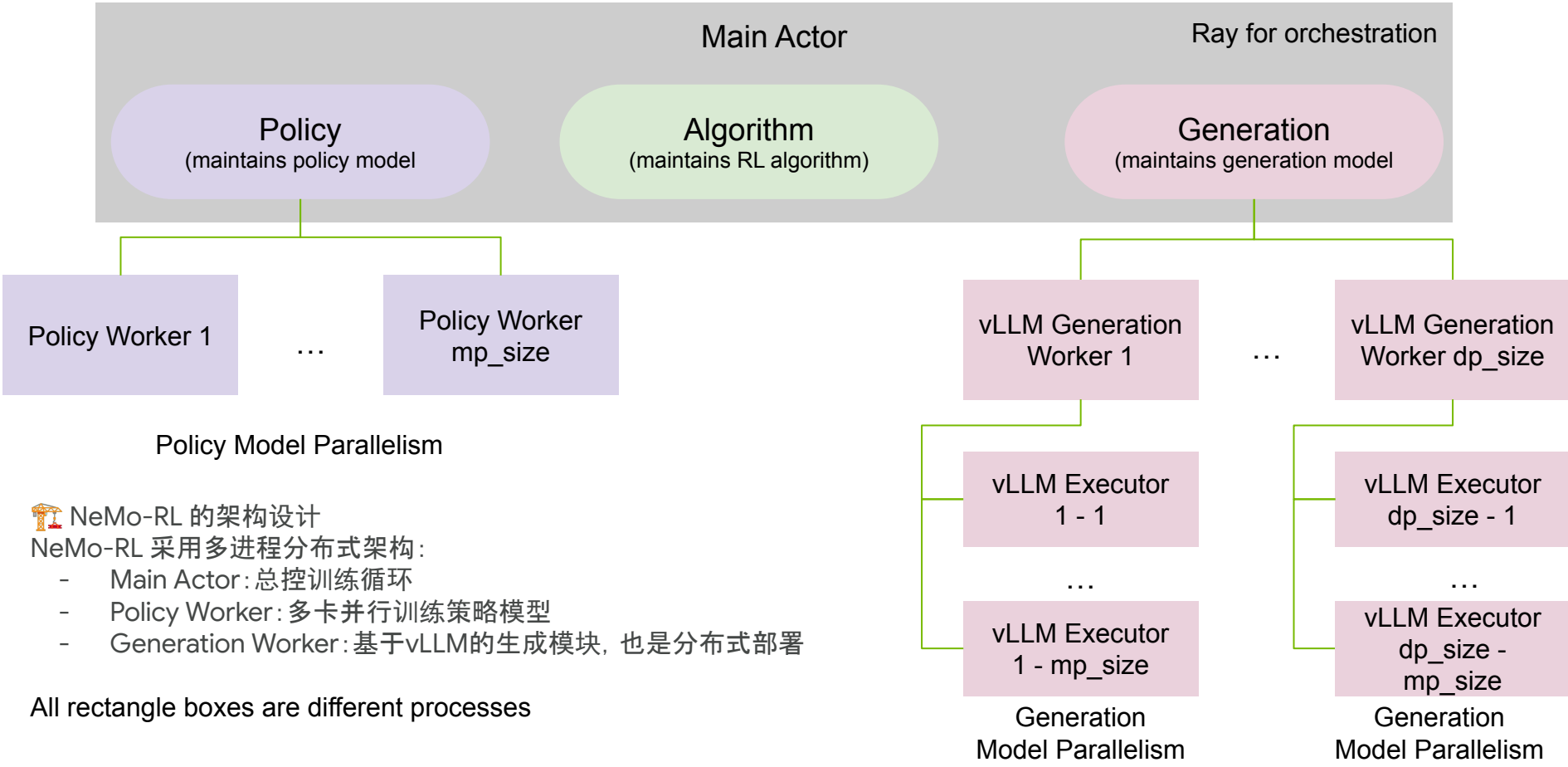
Policy model → 负责训练（策略更新）

Generation model → 负责推理（生成轨迹）

👉 On-policy 同步参数的频率是每一次训练步骤；
👉 两者并行方式不同，导致参数需要跨并行方案同步（reshard）；



NeMo RL's overall software architecture



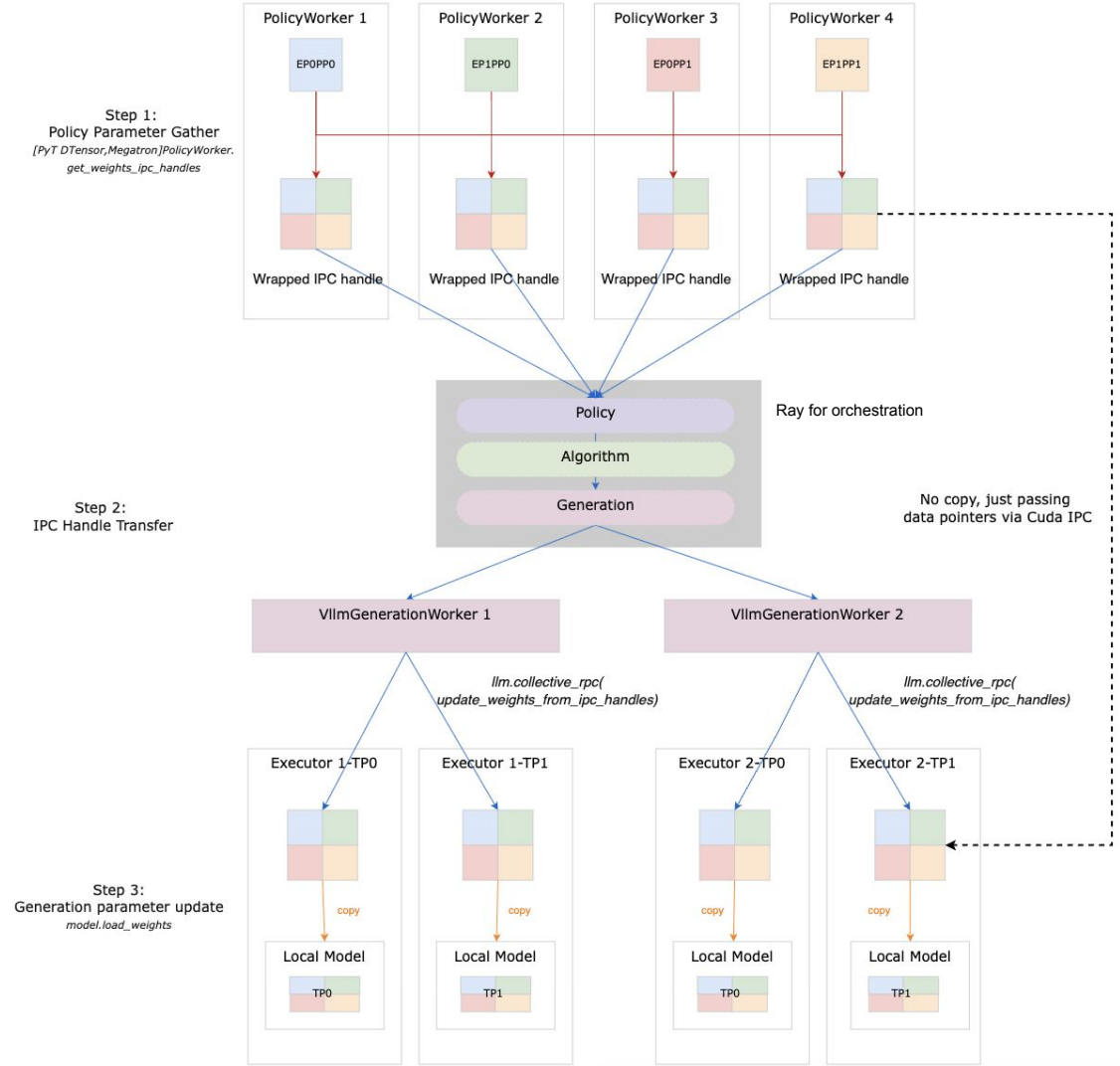
🏗️ NeMo-RL 的架构设计

NeMo-RL 采用多进程分布式架构：

- Main Actor：总控训练循环
- Policy Worker：多卡并行训练策略模型
- Generation Worker：基于vLLM的生成模块，也是分布式部署

All rectangle boxes are different processes

Refit Workflow



Three main steps

1. Policy params gather
2. IPC handle transfer
3. Generation params update

⚡ Refit 三步走

1. 收集Policy参数：从多卡聚合完整参数到单卡
2. 通过ray传递IPC Handle：通过CUDA IPC跨进程传参数（原版最大瓶颈！）
3. Generation model 更新自身分片参数

如果模型太大，可将模型参数分组，每一组进行以上(1)到(3)的操作。

🔑 什么是 IPC handle？

IPC = Inter-Process Communication（进程间通信）

- 在 GPU 上，参数张量本身存储在显存里，不可能复制一份（太大 & 太慢）。
- 所以 CUDA 提供了 IPC handle，其实就是一个 “跨进程共享显存的引用”。
- 就好比：不是搬家把家具（tensor）搬过去，而是发一个钥匙（handle），让别的进程能直接进屋用家具。

📦 在 Refit 里的作用

- Policy Worker 把参数张量生成（all-gather），调用 CUDA IPC → 生成一个 handle。
- 这个 handle 会通过Ray发给 Generation Worker。
- Generation Worker 拿到 handle 后，通过 cudaIpcOpenMemHandle，就能在本地进程里访问同一块 GPU 内存里的 tensor。

我们发现，使用 Ray 在Policy和Generation Worker间传递 IPC handle时存在性能瓶颈。主要问题在于，当参数数量很大时，为每个参数张量传递的成本变得非常高，这在大模型上尤为突出。

以 DeepSeek-V3 为例，其 671B 的总参数量对应着大约 45000 个独立的参数张量，在未优化下，意味着45000次IPC handle传递的开销

Pre and After Optimizations 优化前后对比

	Step 1: Policy parameter gather(s)	Step 2: IPC handle transfer(s)	Step 3: Generation parameter update(s)	Total Refit Time(s)
NeMo-RL w/o optimization	132.9	558.1	1.5	692.5
NeMo-RL v0.3	12.2	33.5	1.5	47.2
Speed-up				**14.7x**
Speed-of-light	3.3	0	0.01	

Table 1: Refit Performance Breakdown and Optimization for DeepSeek V3 - Training sharding EP64PP8 and Generation sharding TP64.

优化1：通过打包张量减少 CUDA IPC 数量

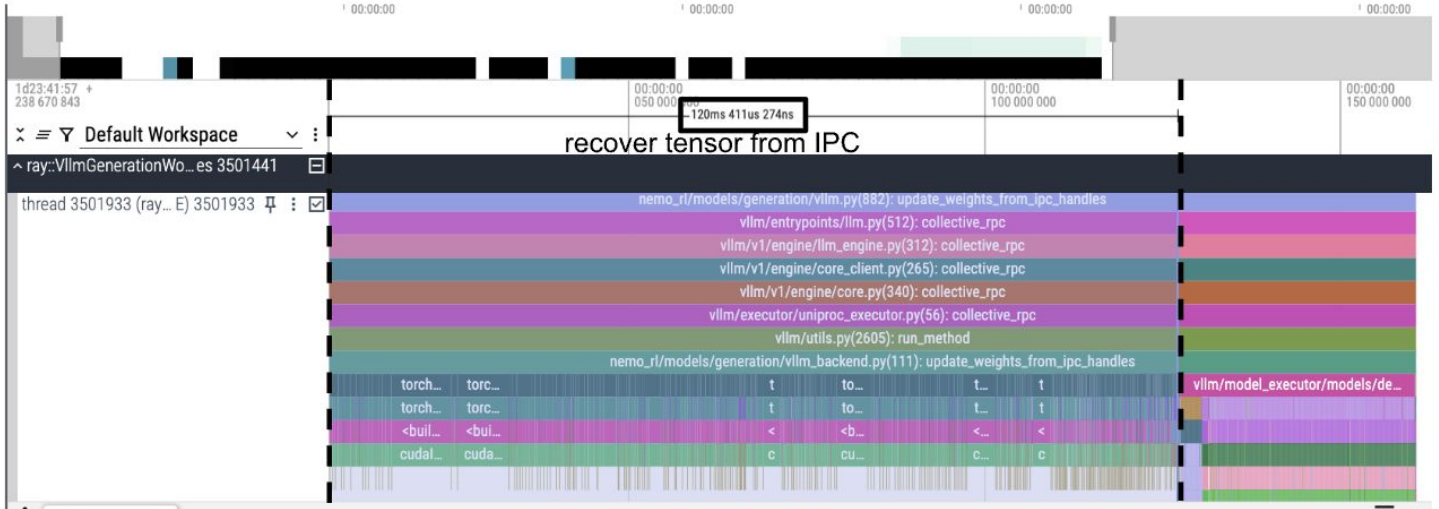
🛠️ 优化一：张量打包（Tensor Packing）—— 从“搬砖”到“集装箱”

❌ 优化前：原始“搬砖”模式
场景：假设模型有4.5万块“砖头”（参数张量）。工人们（进程）需要一块一块地用手搬运（IPC传输），每拿一块砖都要弯腰一次（调用 rebuild_cuda_tensor）。
问题：弯腰4.5万次！虽然每次弯腰（0.18毫秒）很快，但次数太多，总时间长达16秒。效率极低！

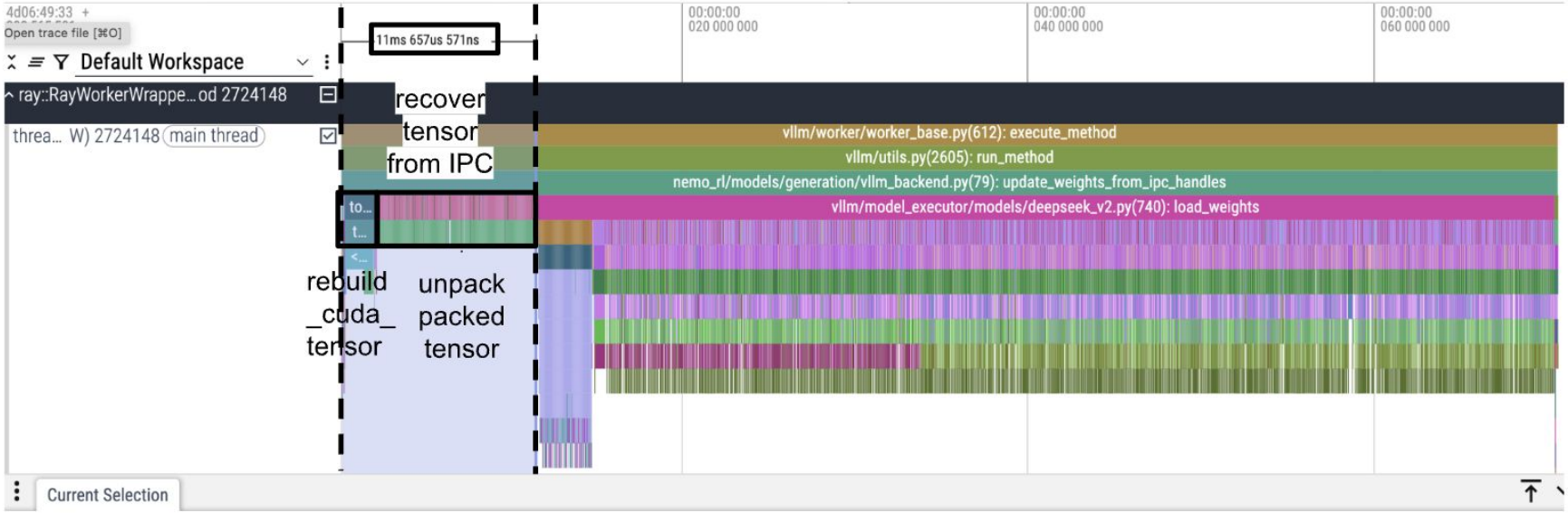
✅ 优化后：“集装箱”模式
场景：我们找来几个巨大的集装箱（IPC Buffer），把所有砖头（参数）整齐地打包进几个集装箱里。同时，我们准备一张“装箱单”（metadata），写明每块砖在集装箱里的位置（offset）、类型（dtype）和形状（shape）。
操作：现在只需要用吊车（IPC）把几个集装箱整体运过去。对方收到后，看着“装箱单”，直接从集装箱里按位置取出砖头即可。
效果：从弯腰4.5万次 → 弯腰几次（吊车装卸集装箱的次数）！耗时从120ms降到12ms，单位操作效率提升10倍！总体提升1.5倍

📌 技术要点：
核心动作：torch.cat() 把多个小张量拼接成一个连续的大Buffer。
关键文件：实现见 NeMo-RL PR 589

Overhead - Prior to optimization 1 tensor packing



Overhead - After optimization 1 tensor packing



🛠️ 优化二：绕过 collective_rpc —— 从“全员大会”到“点对点私聊”

❌ 优化前：开“全员广播大会” (broadcast)
场景：老板（Main Actor）有个通知要告诉所有64个分公司（TP=64的Executors）。他先打电话给64个分公司。而且，老板的通知内容包含了给所有64个分公司的信息，电话内容巨长！
问题：电话次数多：64×64=4096个电话。
通话内容冗长：每次电话都要念一遍给64个分公司的信息，信息量随着分公司数量平方级增长（$O(N^2)$）！效率低下。

✅ 优化后：“点对点发微信” (p2p communication)
场景：老板聪明了。他为公司准备各自个不同的通知包，每个包只包含其下属分公司需要的信息。然后直接微信发给他们。
操作：每个通话的内容都变短了，次数也没变多。
效果：总通信数据量从平方级（$O(N^2)$）降为线性级（$O(N)$）！耗时直接降到原来的1/3.8！

📌 技术要点：
核心动作：将 collective_rpc 广播调用，拆分为多个独立的调用 executor 的 `update_weights_from_ipc_handles`，并为每个调用只传递其所需的ipc handles。
关键文件：实现见 NeMo-RL PR 634

Overhead of argument serialization in collective_rpc API call - O(N^2)

🛠️ 优化三：元数据精 简 —— 从“冗长报表”到“极简备忘录”

❌ 优化前：提交一份冗长的“完整报表”
场景：每次传输都要附带一份巨长的元数据（metadata），就像一份完整的报表，里面充满了：
- 是否打包（is_packed: True）（有用）
- 重建函数（rebuild_func）（没用，对方都知道）
- 参数信息字典：每个参数名都对应其dtype，shape，size，offset（大部分是重复和可推算的）

问题：这份“报表”序列化（打包）和反序列化（解包）的成本非常高，尤其里面很多字段不是ython原生类型，处理起来很慢。

✅ 优化后：只写一份“极简备忘录”
场景：我们极简主义！
缓存不变信息：dtype、shape这些每次都不变，提前存好，不用每次传。
动态计算：offset可以通过dtype和shape算出来，不用传。
删除函数：rebuild_cuda_tensor这个函数对方也有，不用传。
结果：最后元数据只剩下：
- 是否打包（is_packed: True）（1个布尔值）
- IPC Handle（1个对象）
- 参数名列表（1个列表）

效果：数据量大幅减少，提高序列化反序列化速度使得refit快了2.5倍！

📌 技术要点：
核心动作：移除所有冗余、可计算、可缓存的字段。
关键文件：实现见 PR 638, PR 686, PR 698

Three Optimizations and their speedup - a total of 14x

	DSV3 Refitting Time (s)	Marginal Improvement	Cumulative Improvement
Baseline	693	1.0x	1.0x
Part 1 - Reducing Cuda IPC Overhead via Tensor Packing	450	1.5x	1.5x
Part 2 - Mitigating the overhead of vLLM's collective_rpc API	120	3.8x	5.8x
Part 3 - Further reducing refit overhead via Ahead-of-time metadata preparation	47	2.5x	14.7x

Table 2: Refit optimization breakdown and their contribution to performance improvements.

💎 总结一下三项优化的精髓：

1. 打包优化：解决的是“次数多”的问题，变多次IPC为少数几次。
2. RPC优化：解决的是“数据量广播信息冗余”的问题，变广播为精准投送。
3. 元数据优化：解决的是“额外开销大”的问题，变冗长报表为极简备忘录。

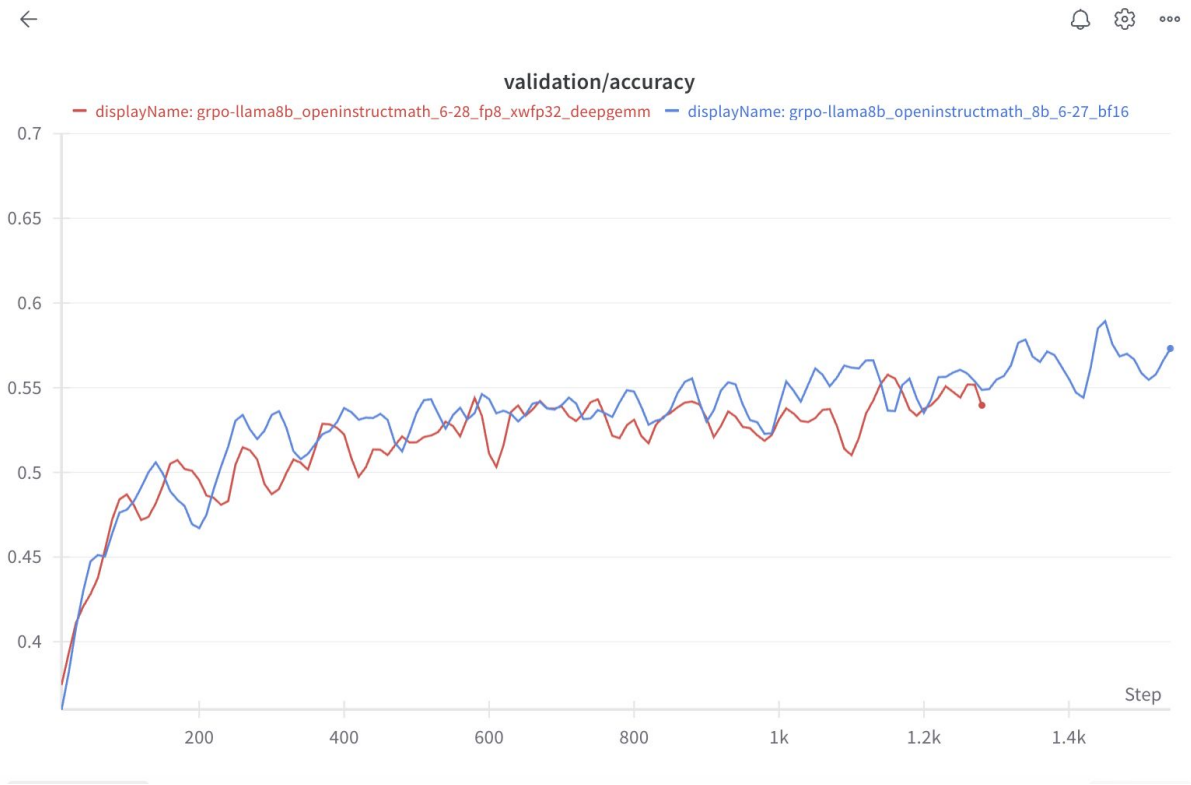
未来计划

- 集成 NeMo Megatron-Bridge，进一步优化训练的时间

Agenda

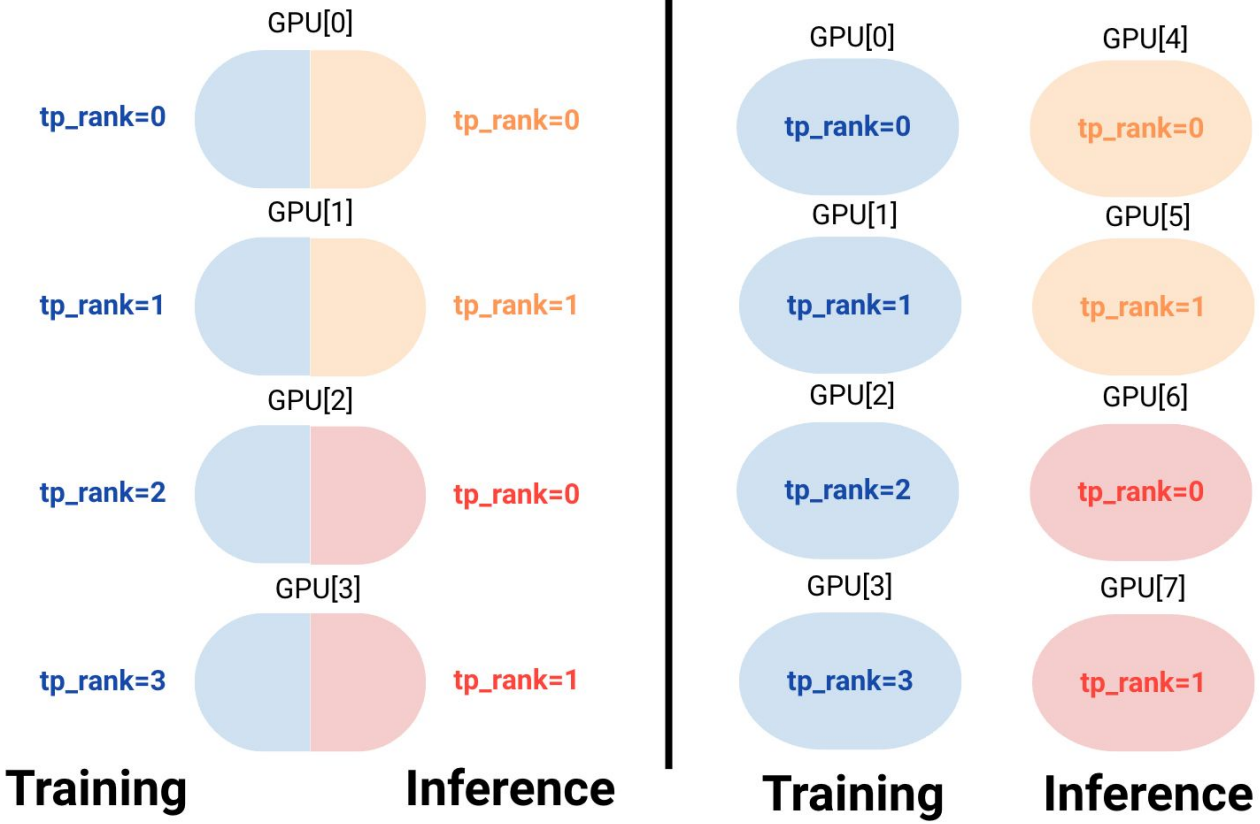
- What is Reinforcement Learning and NeMo RL Overview
- Refit Optimizations for large MoEs
- Other on-going work in NeMo RL
  - FP8
  - AsyncRL
  - NeMo Gym
- Future Roadmaps

FP8 Rollout



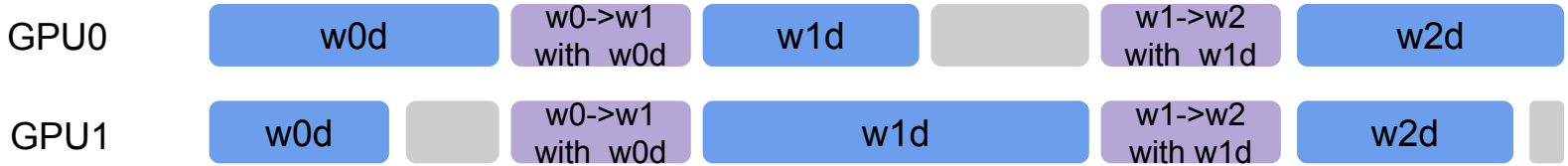
- Model: llama3-8b
- seqlen grows from 600 ->1.5k ish over 3k steps
- FP8 linear layer with 128 block scaling (i.e. deepseek fp8) + **importance sampling** convergence
- Speedup: 30% on the decode (GEMMs account for 70% of it), 10% e2e
- With FP8, 45% of the decoder loop is spent on bf16 attention and kv cache
- Future work:
  - try FP8 attention and kv cache
  - QAT in training

Engineering Highlights
Placement strategy - Collocated vs Split



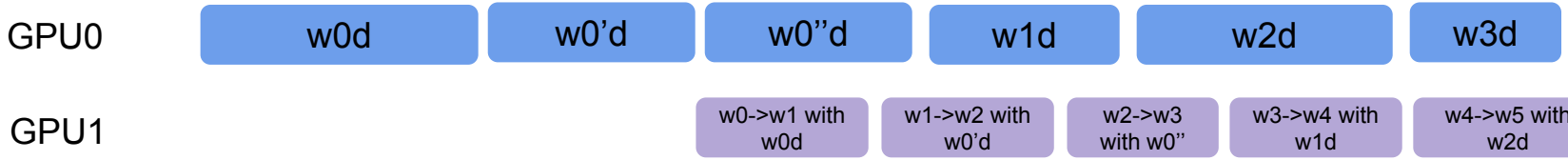
Asynchronous RL - n step off

On-policy/Sync RL - Collocation placement



Off-policy - Split placement (1-step off)

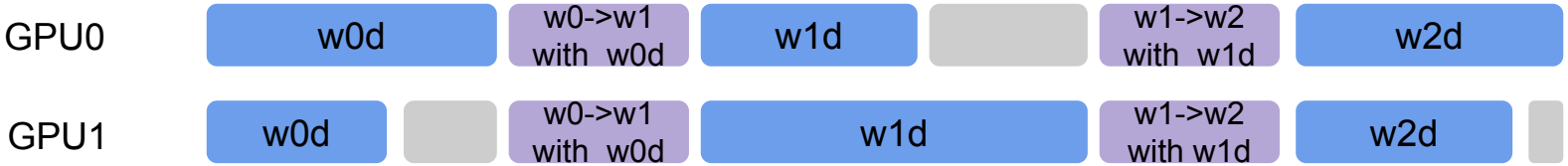
Off-policy - Split placement (2-step off)



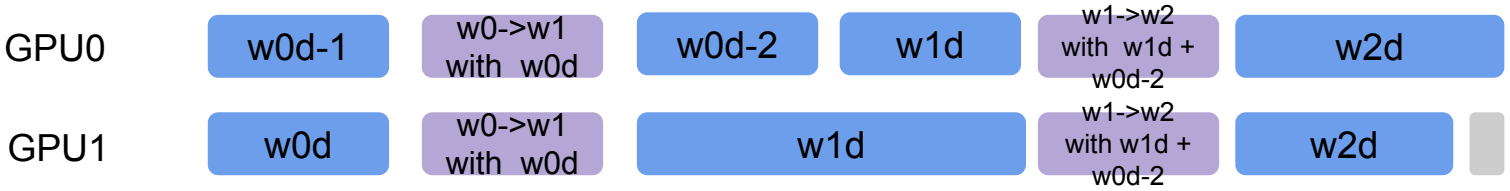
Idle time | w*d Data generation from checkpoint w* | w*->w** Training checkpoint from w* to w**

Asynchronous RL - partial rollout

On-policy/Sync RL - Collocation placement

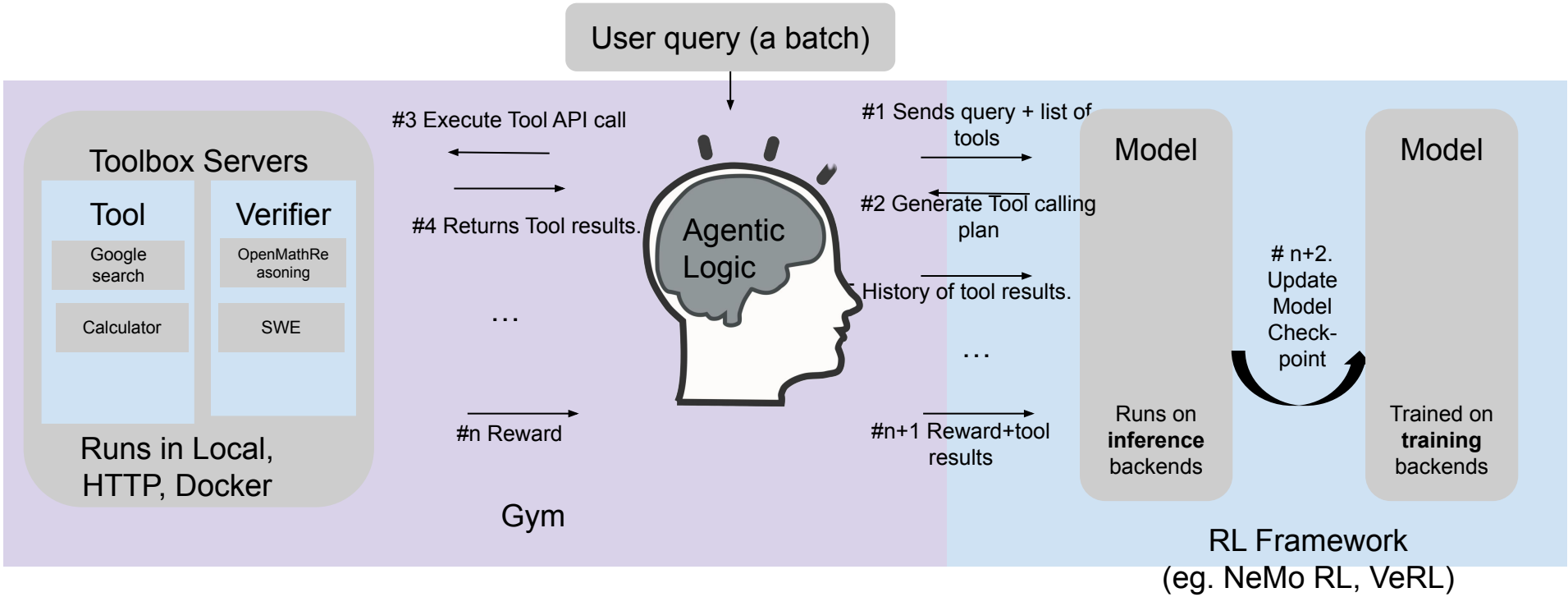


On-policy/Sync RL - Collocation placement with partial rollout



Split placement with partial rollout follows similar

RL System = NeMo RL + NeMo Gym



List of Example Verifier Env by Domains

Math
Open-Math-Reasoning
StackOverflowDump
DAPO-17k

Coding
SWE
SWE + openhands
SWE + mini-swe-agent
Competition
Programming pool
Terminal Bench

Tool
WorkBench
Financial Agent
Search-R1
Text2SQL

Science Knowledge
GPQA
HLE
OpenStem
StackOverflow
Olympiad

Game
Wordle
Puzzle
Arcade

? Domain
Proprietary Dataset Env

Agenda

- What is Reinforcement Learning and NeMo RL Overview
- Refit Optimizations for large MoEs
- Other on-going work in NeMo RL
  - FP8
  - AsyncRL
  - NeMo Gym
- Future Roadmaps, Blogs and Q&A

Roadmap Reinforcement Learning - Nemo RL

	Current (v0.2.1)	V0.3 - July 21st	V0.4 (Sep 2025)	v0.5+
Training Optimization	PyT - FSDP, TP, SP Ray Orchestration	PyT - FSDP2 Mcore - 4D parallelism Sequence packing, CP	FP8 e2e Conversion between HF and mcore backends	GB200 Fault tolerance and in-process restart
Inference Optimization	vLLM	Async vLLM engine for faster multi-turn on-policy FP8 vllm generation	AsyncRL with split placement	AsyncRL-magistral variant FP4 Generation TRTLLM, SGLang
Algorithm	GRPO, DPO, SFT, Multi-turn tool use	LLM as a judge	GSPO, DAPO with dynamic sampling and reward shaping Reward model Search-R1, SWE,Openhand	Gym environment
Model	HF models up to 32B @16k seq	70B with up to 16k, incl Qwen-3 30b/32B	DSV3, Qwen3-235B, GPT-OSS VLM via PyT	VLM via megatron

Blogs

Improve Weight Transfer (i.e. Refit) in Deepseek v3 by more than 10x

Scalable and Performant Post-training with Nemo-RL via Megatron-Core

Reproduce DeepScaleR recipe in GRPO