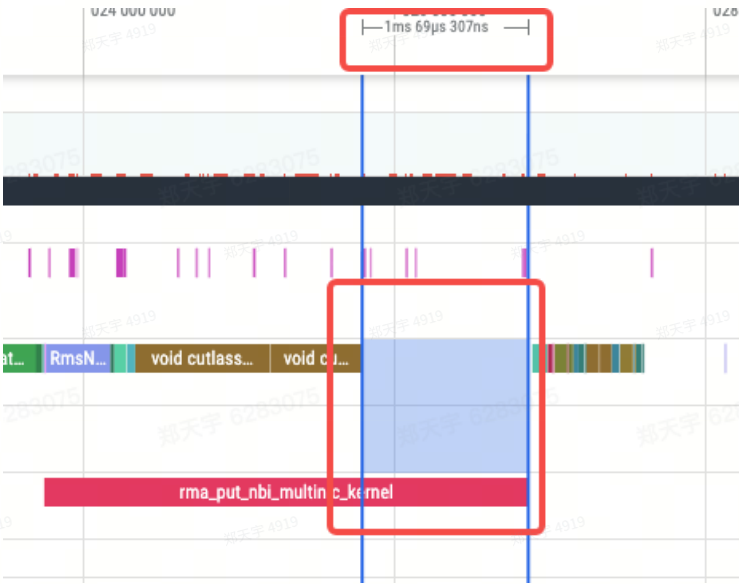


pp的单边put通信支持与 FLUX Manager优化

1. Only put模式下的通信优化



当前的两种通信逻辑：

- 跨层传输

跨层传输的时候，保证对应pp_stage，即pp_rank put的信息，在对应 consumer rank的同一轮rank执行完后就wait即可，这里put不需要发 signal，consumer不需要感知对方什么时候发完，因为根据这个逻辑，已经保证consumer在消费时已经发完了，如果是consumer是本pp rank上的，则不需要跨层通信，直接取本地的数据即可，这个由计算的buffer保证对应数据不会被覆盖。



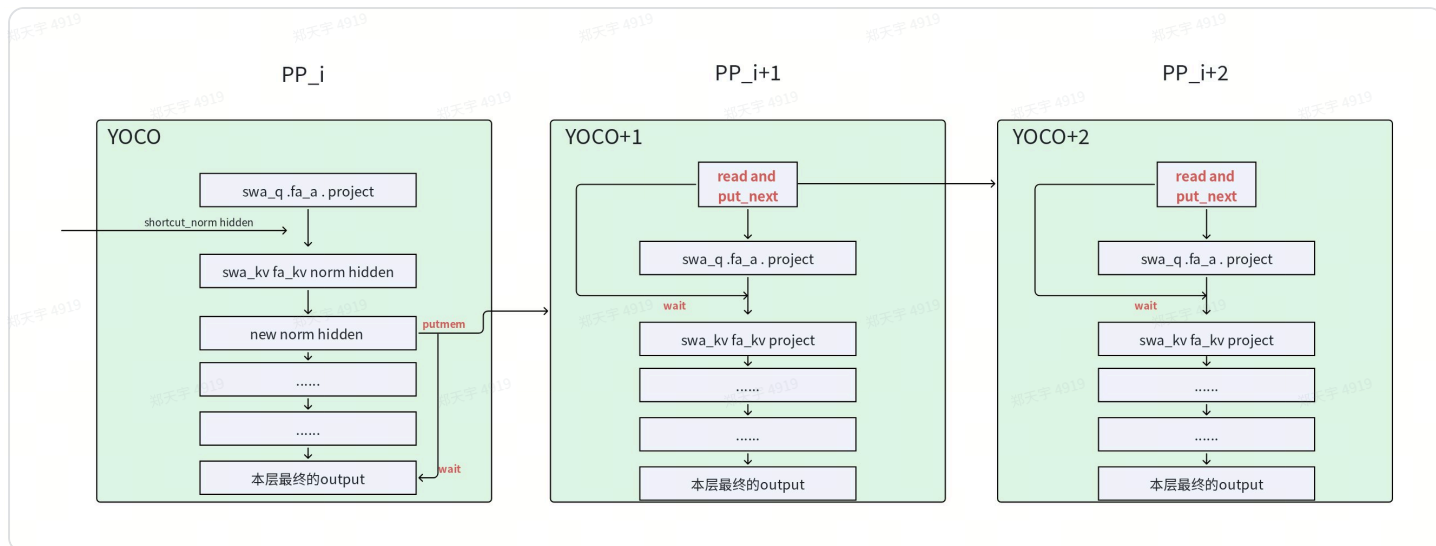
这个不需要修改，因为我们现在的跨层的层数比较大，这里的通信是能被完全盖住的。

- 逐层轮询传输：（通信暴露的点）

当前的逻辑：

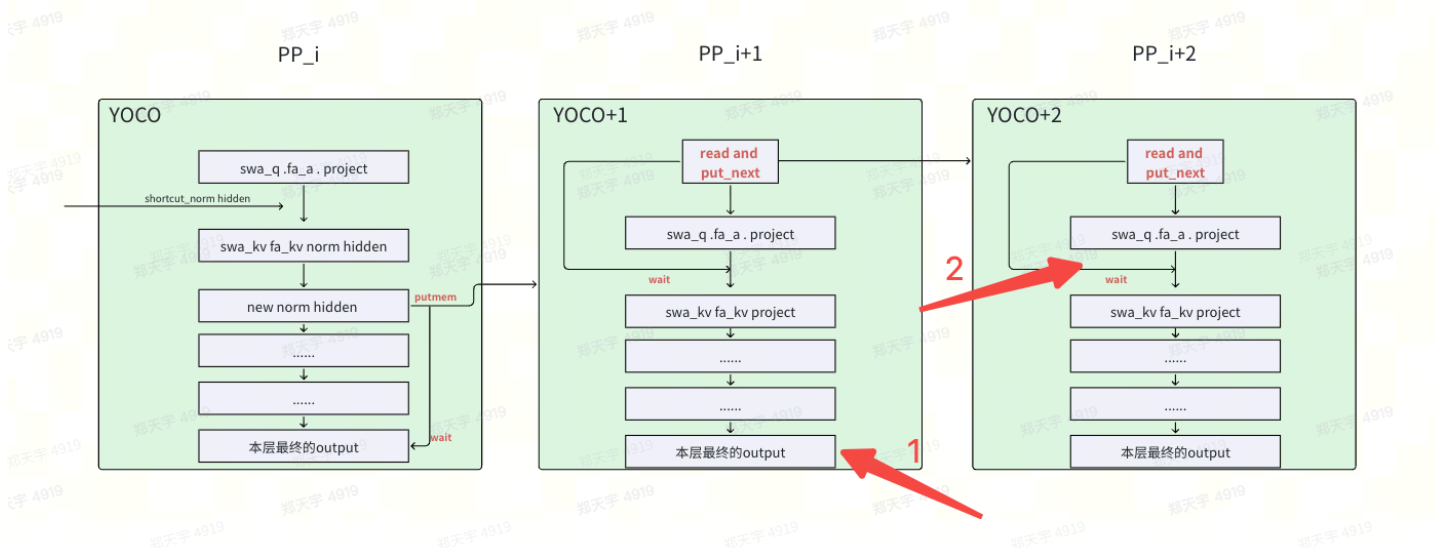
现在的kv hidden state是逐层共享的，所以是逐层传递的逻辑，同样为了少一次get通信，现在也是直接通过putmem，写到下一个stage上，并且在本层保证put传递完成，read的时候就不感知通信，直接取数据。

这里有一个逻辑绑定在一起，即read+转发操作，



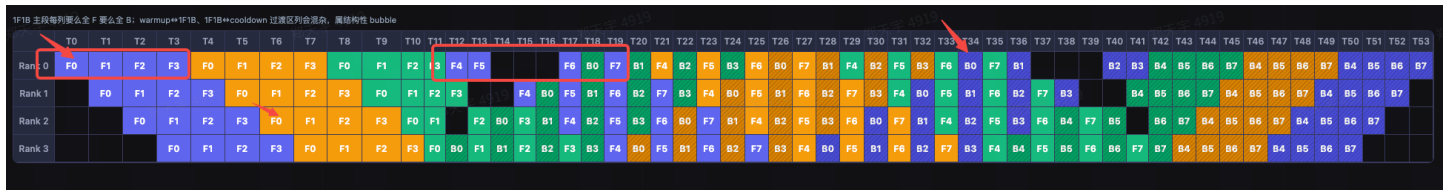
现在每层这里都有putmem的耗时暴露，这个暴露可以完全优化掉，如图，第一个yoco的put + wait没问题，是在最后这个fwd结束的时候wait的，实际上可以再晚一些。

后续的消费层，除去最后一层，会read+put转发，这里wait的用法，是跟双端put get对应的，但是现在我们已经确保put在上一层做完了，所以这里read没有实际通信wait，并且，转发到下一组的put由于跟read绑定了操作，导致wait的时间其实早了，应该在本层的output计算完后，再wait即可。这里用的是staging临时区，在当前层wait是合理的，防止被覆盖，而实际上，如果putmem带上了signal，其实wait时间可以更晚。



其实可以在2的地方wait，overlap的更充分，而我们可以用共享的slot buffer区，因为我们创建的时候其实已经保证了这个关系。

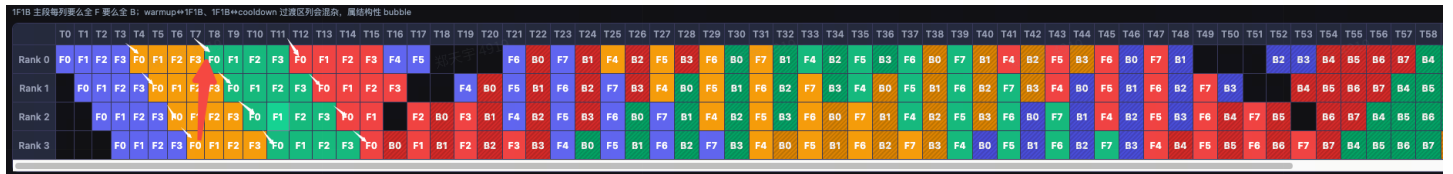
这个是我们留的buffer，主要是保证，当前这个F0的 kv hidden state 在bwd 消费前不被覆盖，所以是 $x1+x2$ 。当然这个显存是可以优化的，对应一个公式，具体看我文档（现在 [王曦](#) 的最新mr 应该是也做了这部分显存优化的）：[FluxManager](#)



而上面这个关系就保证了，我们其实可以利用共享的slot，来putmem，并且在read端 wait，那如何保证当前putmem完成前，数据不被其它micro_batch覆盖？

其实主要就下面两种情况：

1



第一种情况，同一组micro_batch覆盖，这个可以直接覆盖，因为共享的hidden_state完全一样，并且，当前这个pp_stage的同一个micro_batch要做下一个layer的fwd的时候，那上一层的fwd的put一定已经发完并被使用了。

2



第二种是，这个pp_stage的最后一层的fwd，没有新的fwd能证明它put通信做完了，但是bwd可以，而这时候，第二个group用到的buffer，是我们预留的x2部分，也不会覆盖。

优化前：

trace (run27) :

<https://seed.byteintl.net/development/profiling/preview/eac13c47ce2a3e345a6452c9a315adac4b263b0bb4347b8fc7f6ff5cc5cfc92a?>

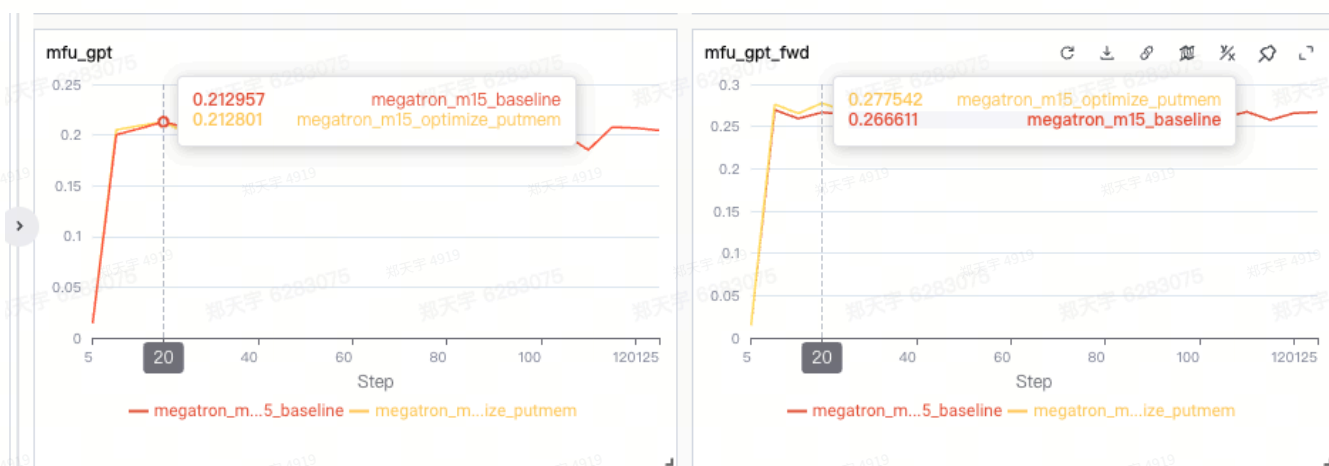
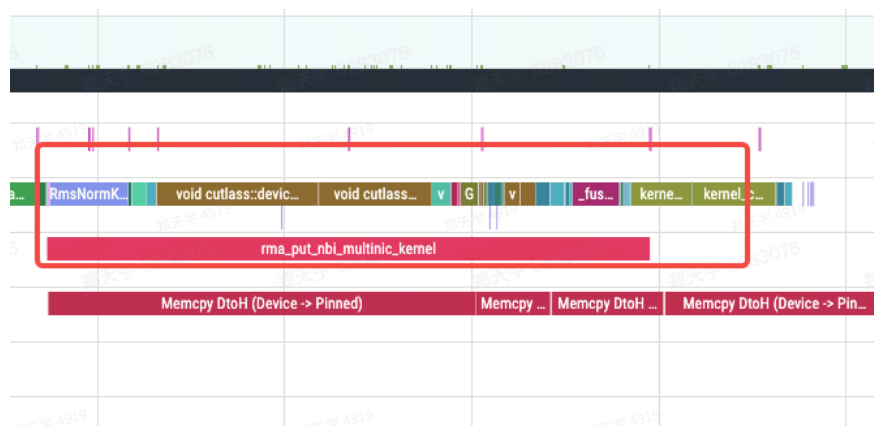
trace_source_info=%7B%22title%22%3A%22M15_150B_megatron_test_64gpus_copy_3%22%2C%22url%22%3A%22https%3A%2F%2Fseed.byteintl.net%2Fdevelopment%2Finstance%2Fjobs%2F626307df4ac0fabf%3FtabState%3Drun_info%26trialId%3D302249786%22%2C%22subTitle%22%3A%22Trial%20302249786%20Run%2027%22%7D

- 优化后：（现在是挪到fwd结束wait，已经完全能盖住了）

trace (run28) :

<https://seed.byteintl.net/development/profiling/preview/7b3040ebca16d3d03cb25afbcdd2a3747dc9ee6d7c578e12d4d7152fd24039c0?>

trace_source_info=%7B%22title%22%3A%22M15_150B_megatron_test_64gpus_copy_3%22%2C%22url%22%3A%22https%3A%2F%2Fseed.byteintl.net%2Fdevelopment%2Finstance%2Fjobs%2F626307df4ac0fabf%3FtabState%3Drun_info%26trialId%3D302249786%22%2C%22subTitle%22%3A%22Trial%20302249786%20Run%2028%22%27D



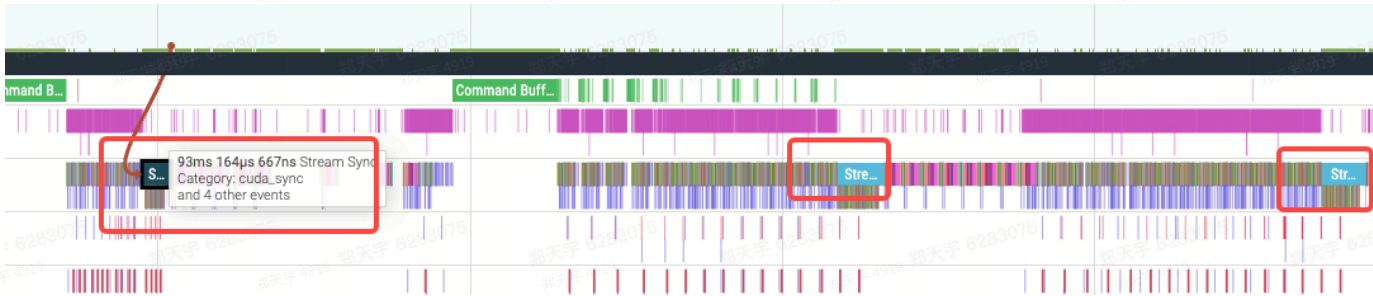
mfu_gpt_fwd稳定高，在0.277左右（差不多符合预期了，如果seq到8k，还能更高，64卡目前开不到，256卡理论能开到，有资源了试一下。这次测试 yoco通信去重，以及bf16版vwn还没接，现在还做了cast，这两块也还能优化一些），mfu_gpt是0.212，等单边put和任意编排micro_batch实现，mfu_gpt理论还能提升不少。



Loss bitwise对齐。

Tracking: https://seed.byteintl.net/experiment/tracking/detail?Id=project_20260623_7e8baa7a&selectedTrial=&tab=CHART&viewId=view_20260823_36175246

4k目前cpu launch还是慢了，导致有很多sync，以及dsp 间的频繁长barrier，其实compute bound 不是瓶颈。



训练任务：

序号：	comfig	配置	任务链接 (共用)	run	说明	
1.	1. byteload r	1. loader_type=byted_local_loader	https://seed.byteintl.net/development/insta	27	baseline	

	2. 开bf16 3. Epkerne l 换环境变量 4. 改oe embedding 的形状，开48个table，让rank0和其它pp stage 尽量平衡 5. Bumi kerne l 优化	file_type=lance shuffle_algo=global use_refactor_transform=true sync_init 去掉 cpu_prefetch.enable 去掉 allow_transform_failure=True enable_client_managed_ckpt=false 2. use_bf16_residual=true 3. NCCL_CGA_CLUSTER_SIZE: 4 FLASH_COMM_CGA_CLUSTER_SIZE: 4 4. over_enc_vocab_size=[225013,225029,225041,225043,225089,225091,225119,225163,225167,225181,225223,225247,225257,225263,225337,225349,225359,225397,225401,225403,225419,225421,225431,225439,225461,225467,225503,225509,225559,2250581,225601,225623,225637,225641,225649,225667,225671,225679,225709,225713,225761,225763,2250769,2250779,225793,225799,225869,225887] nce/jobs/626307d4ac0fabf? tabState=run_info&trialId=302249786				
2.	putmem 优化	无需配置		28		
3.	Onlyput p2p通信	use_pp_hidden_nvshmem=True		40	gpu, oom, 后续测试	

4.	先测一下8k	use_pp_hidden_nvshmem=false	43	cpu, oom, 需要调整loader的配置
5.	接着测3	use_pp_hidden_nvshmem=True export NCCL_MAX_NCHANNELS=8 export NCCL_MAX_P2P_NCHANNELS=4 export NCCL_BUFFSIZE=2097152 # 2 MB, down from default 4 MB	run 45	Hang 查原因
6.	测一下6k	use_pp_hidden_nvshmem=false		

2. 支持单边put通信，替换send recv

量	值	说明
max_position_embeddings	32768	
micro_batch_size	32	
dsp_size	8	
seq_len_per_dsp	$(1 * 32768 + 1) // 8 = 4096$	
hidden_size	12288	
VWN	use_vwn=true, n_in/m = 32/8 = 4	
dtype	bf16	
group[0].size	4	只需要按第一个group预留即可
pp_size	8	

现在预留的buffer按最大buffer来设置，且fwd和bwd的buffer分开设置，即：

$$(2 + (\text{group_size}[0] + \text{group_size}[1]) * 2) * 4096 * 12288 * 4 * 2 / 1024^3 = 12.75\text{G}$$

后面如果开 8k + pp4:

gorup_size缩小为4, 4096->8192

需要13.5G

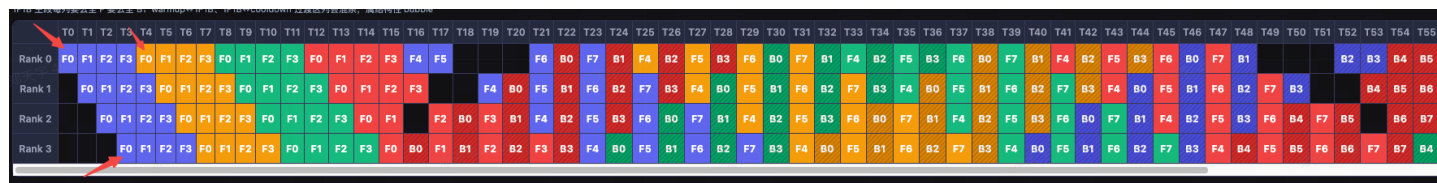
而实际我们可以做显存优化, 见下

显存优化:

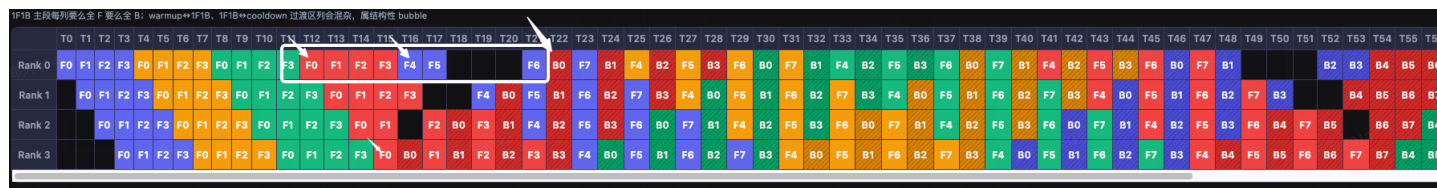
当前预留的buffer显存占比为:

fwd:

1. 情况1:



2. 情况2:



所以, 先不考虑显存优化的问题, 这里的buffer最小得开 $\text{group}[0] + \text{group}[1]$ 的size大小, 一定不会被覆盖, 对应bwd也开了对应大的buffer, 但是从2可以看出来, 这里不需要开到两个group的大小, 开7个就够了。实际上这里就是算一个 交界处, micro_i 的fwd 对应的 bwd 中间最多有几个 fwd 同时存在, 具体可以看公式推导: [FluxManager](#)

bwd:



可以看到这里, 我们思考一个问题, 红色的bwd出现后, 这表示fwd的put已经完成了, 所以这里bwd其实是可以直接覆盖写fwd的buffer的, 因此实际上fwd和bwd可以共用buffer。