启发式算法优化remainder*2<pp的bubble

1.问题背景

在 Min_Memory_Any1F1B 调度中，当 micro_batches (mb) 不能被 pp_size (pp) 整除时，会产生余数部分 (rem = mb % pp)，**当rem*2<pp时**，在1F1B阶段由于无法完全填充满流水线，会导致流水线利用率骤降，产生大量额外 bubble。

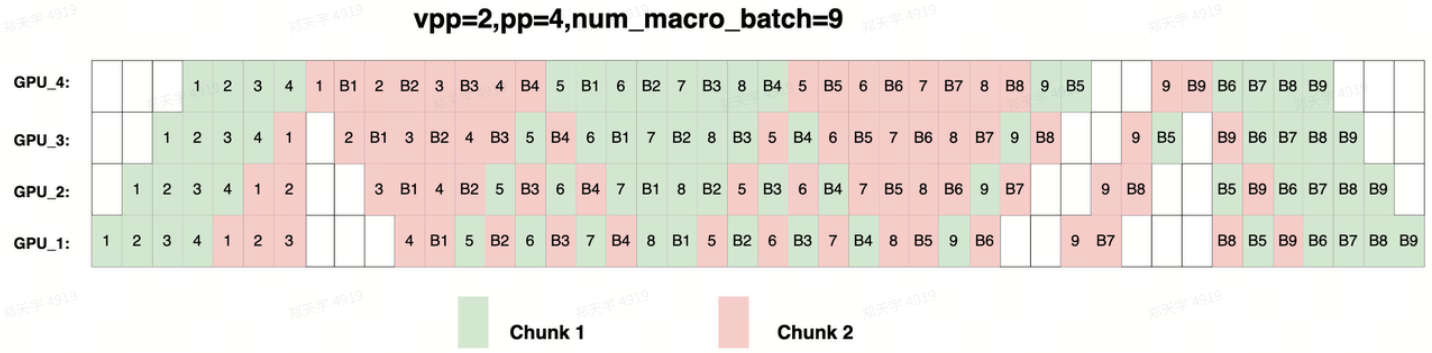
例如 pp=6, vpp=3, mb=14:

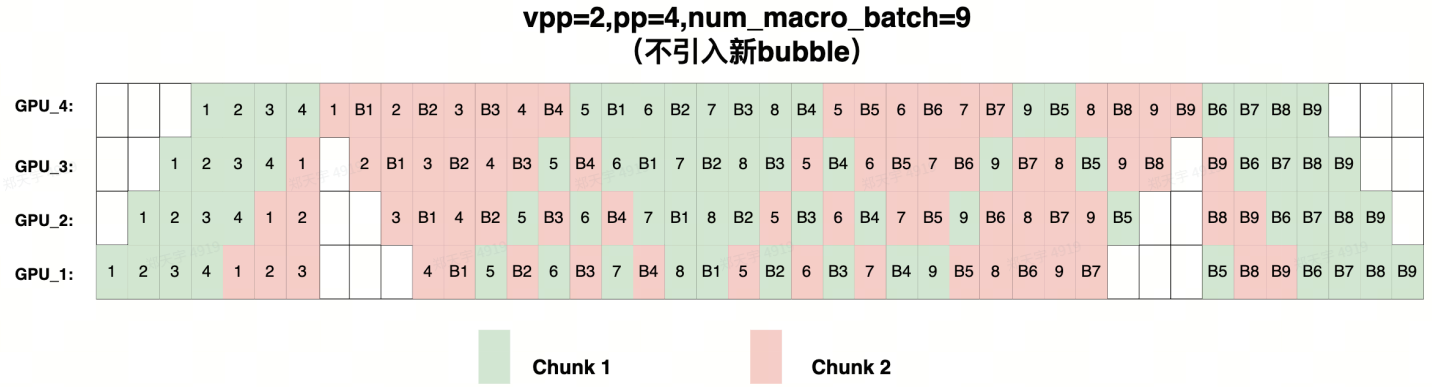
- base_group = 2, rem = 2
- 标准编排: group_batches = [6, 6, 2]
- 扩展后: forward [6,6,6,6,6,6,2,2,2] backward[6,6,6,6,2,2,2,6,6]
- 从第二组(size=6)过渡到第三组(size=2)时，流水线利用率从100%骤降到33%

如上图所示的流水并行图，这里主要是rank0在等待rank5的F12计算的结果，而此时没有新的micro_batch继续做forward了，所以就产生了bubble。

新增加的bubble时间为: (pp//2-remain)*(vpp-1) 个 T(forward+backward)

2.启发式算法分析





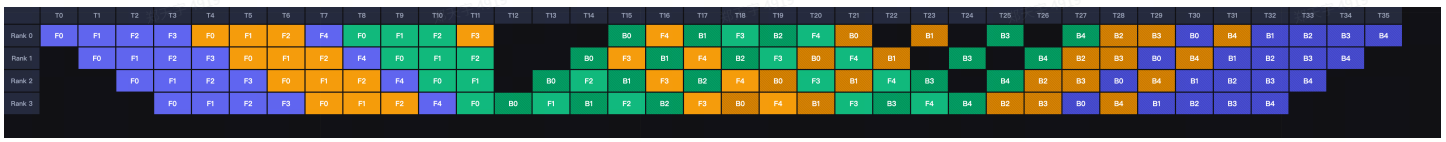
我们以一个数量相对较小的例子来分析一下对于编排的bubble优化，可以发现这里消除bubble的主要方法在于将余数的forward提前一步处理。因为余数的forward也是关键路径的一环，其执行时间，严重影响着整个流水并行的时间，并影响bubble率，因此首先想到的就是对低chunk的forward尽早做处理。

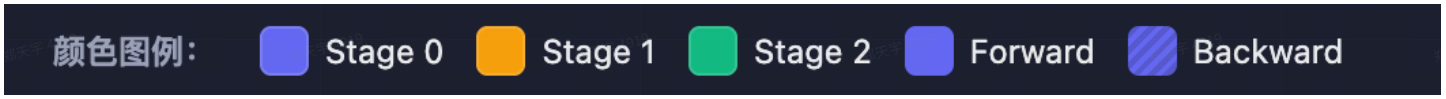
上述事例比较单一，那么接下来分析一些相对复杂的remainder*2<pp的bubble情况：

1.pp=4，vpp=3，micro_batch=5

存在bubble：

不存在新增bubble（提前F4）：

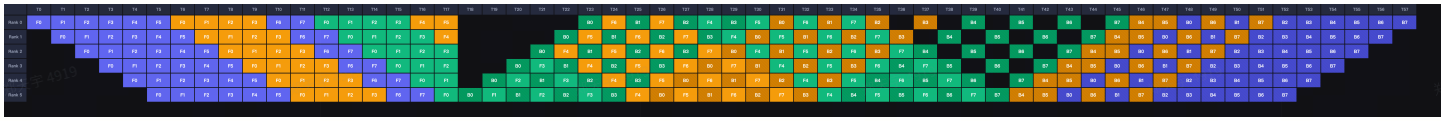


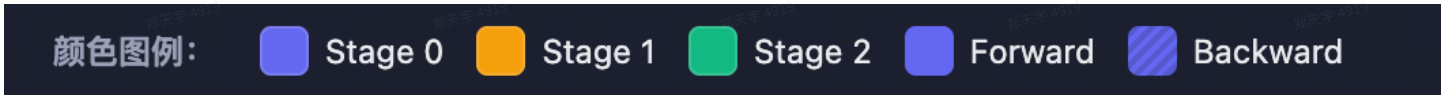


2.pp=6,vpp=3,micro_batch=8（增大pp）

存在bubble：

不存在新增bubble（提前F6，F7）：





3.pp=6,vpp=4,micro_batch=8(增大vpp)

存在bubble:

不存在新增bubble（提前F6，F7）：

4.pp=4,vpp=3,micro_batch=9（增大micro_batch，即增加group数）

存在bubble:

不存在新增bubble（提前F8）：

如上面四个例子所示，我们较小规模的示例开始，逐渐增加pp，vpp，以及micro_batches的数量，可以发现，通过提前处理forward，我们均可以找到一种编排来减少bubble，使得bubble的个数与均匀Min_Memory_Any1f1b(即micro_batches%pp_size==0)或标准Any1F1B时的bubble个数一致，即不新增bubble，从而实现最小bubble率。

然而对于不同的pp、vpp、micro_batches的组合，这里没有一个固定的编排方案，尤其是backward是根据具体forward编排进行调整的，但是存在一定的编排规律，因此我们采用启发式算

法，来解决这个问题。

上面的示例我们不难看出来，主要是需要将余数部分vpp_stage的forward均提前一步到vpp_stage-1处理（最多提前到vpp_stage_1），看起来只要将余数组提前到前一组的尾部处理，将前一组尾部的forward延后，即可实现不新增bubble，然而我们再看一些特殊的情况：

1.边缘插入法与中值插入法

边缘插入法：

`pp_size=4,vpp_size=2,micro_batches=5`

我们把上述在尾部提前余数组的forward处理的方法称为**边缘插入法**，在此场景下，bubble=24，即每个rank 6个bubble，负责常规vpp编排的bubble。micro_batch_4的forward提前，在rank间传递时，传递时间刚好被B0(stage1)，F3(stage1)，B1（stage1）所覆盖，不会新增bubble。

`pp_size=8,vpp_size=2,micro_batches=9`

可以看到这里的bubble是144，即每个rank 有 18个bubble，然而我们知道常规的vpp编排，bubble为2*(pp-1)，即14个bubble（这里我们假设forward和backward的计算时间一致）。

这里多出了4个bubble，其实可以发现原因在于，当pp增大后，意味着rank0从计算完发送forward数据到接收到rank pp-1 的数据时间延长了，即使提前了余数组的forward的处理，后续没有更多的micro_batch来填充1F1B，因此导致了在1F1B间出现了bubble。

那么我们可以想到的一个比较直观的优化就是：将remainder(余数组)的vpp_stage的forward提前的处理，放在round(前一个group)的vpp_stage-1的forward部分的1/2的位置，即pp_size//2的位

置，这样round一定剩余pp_size//2个vpp_stage-1 的 forward还没有处理，而在1F1B阶段，就能填充pp_size个单位时间，从而确保，remainder的forward在rank0到rank pp-1的传递过程中一定有后续的forward进行填充，来降低bubble。

中值插入法

我们讲上述优化方法称为**中值插入法**，如图所示，经过优化，bubble变为112，符合每个rank14个bubble。

2.warmup_step的分析

我们可以发现，中值插入法是可以在数学理论的基础上确保1F1B阶段不会出现backward没有forward匹配的情况，从而避免在复杂情况下出现bubble，然而我们需要思考一个问题，即forward的插入可能会出现在warmup阶段，我们需要分组讨论，以在实际编排时，确定warm_up的step和1F1B的step。这里直接给出具体分组，可划分为两组不同情况：

1. base_group_size==1且vpp_size>2:

如图所示，在此情景下，forward提前在warmup阶段，因此warmup_step的计算方式为：

$$warmup\_step = (vpp\_size - 1) * pp\_size + pp\_size - rank - 1 + (vpp\_size - 2) * remainder$$

2. 其它情况

此情景下，因为forward提前在1F1B阶段(或vpp_size=2),因此不会影响到warmup阶段的step数，warmup_step的计算方式为：

$$warmup\_step = (vpp\_size - 1) * pp\_size + pp\_size - rank - 1$$

base_group_size==1,vpp_size=2:

base_group_size>1:

可以发现，在情景1下，warmup_step会有一些增长，也意味着显存峰值会存在一些增长，但仍然比当前的Any1F1B的显存要小，因为当前Any1F1B在此情景下的warmup_step，是把情景1公式中的 `(vpp_size-2)*remainder` 替换为 `(vpp_size-1)*remainder`，且 `pp_size-rank-1` 的系数为2，而不是1。

并且我们可以发现，其实情景1在某些情况下仍然可以优化：

我们在此处 启发式算法优化remainder*2<pp的bubble，讨论了 `边缘插入法` 和 `中值插入法`，其实可以发现，边缘插入法，不会引入warmup_step的增加，那么我们可以在满足一定条件下，使用边缘插入法，而非中值插入法，

因此我们可以计算得到一个公式(其中r=reminder，pp=pp_size)：

$$(r-1)*2+r*2+1>=pp-1$$

这个公式中左边：(r-1)表示余数组中除去当前处理的这个forward，剩余余数，r表示插入r个余数的forward后，会有对应的r个forward延后执行，*2是因为在1F1B阶段，+1则是当前处理的这个forward后面会有一个backward。右边：pp-1表示当前forward经过pp-1步后即可接收并计算下一个chunk的forward。

因此，整个整理后得到：

$$pp//4<=r<pp//2$$

在此情景下，我们可以用边缘插入法，从而实现更低的显存，下面也给出了一些例子对比：

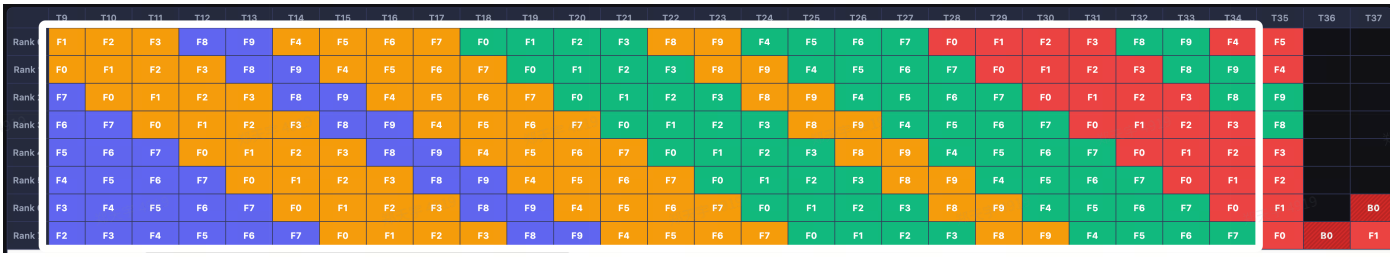
a. pp=4，vpp=4, micro_batch=5

中值插入法(rank3: warm_up_step=14)：

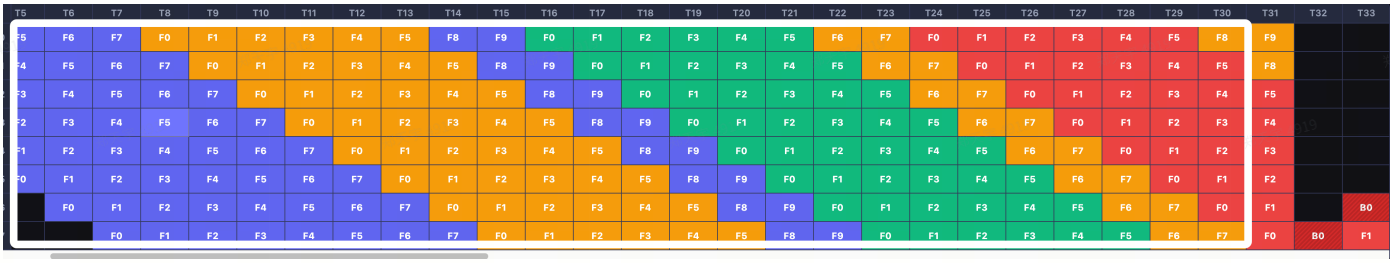
边缘插入法(rank3: warm_up_step=12)：

b. pp=8，vpp=4, micro_batch=10

中值插入法(rank3: warm_up_step=28)：



边缘插入法(rank3: warm_up_step=24)：



3.中值插入法与边缘插入法的使用选择

上面我们讨论了在 `pp//4<=r<pp//2` 的范围内我们可以使用边缘插入法，来降低显存，低于这个范围就不能用，否则会新增bubble，那么我们同样需要讨论一下，中值插入法的适用性:

考虑一下，中值插入法，会带来一个什么问题：非余数group的最后一个group，有pp//2个forward被延迟执行了，我们需要注意一个问题，一个micro_batch的backward可执行的一个准则是它的forward全部执行完毕，而forward的延迟也代表着backward的开始执行的时刻被延迟了，那么当vpp_size>2，且rounds>=2(即rounds=micro_batches//pp_size)，此时，当最后一个非余数group，和余数group交互时，有以下问题：

以rank0为例（每个rank都会有同样的问题）：

Forward: chunk_0. chunk_1. chunk2. ....

个数： [pp_size] [pp_size/2, reminder, pp_size/2] [pp_size/2, reminder, pp_size/2]

Backward: chunk_n-1. chunk_n-2. chunkn-3. ....

个数： [pp_size] [pp_size.] [pp_size.]

即forward和backward会不断错位，最终导致forwrad还在推进，但此时没有可以推进的backward了，导致1F1B过程断开：

~~因此当前使用的结论如下：~~

1. ~~当r>=pp//2时，无需使用启发式算法。~~
2. ~~当 `pp//4<=r<pp//2` 时，使用边缘插入法~~
3. ~~当r<pp//4时，当vpp=2 或 micro_batches//pp =1，使用中值插入法~~
4. ~~若r<pp//4，且vpp>2,且micro_batches//pp>1，则暂时不用启发式算法~~

~~结论暂定，感觉还是可以通过手动编排实现不增加bubble~~

新结论见：启发式算法优化remainder*2<pp的bubble

补充说明：

1. 理论说明vpp=2或micro_batches//pp=1时，使用中值插入法不会新增bubble
   a. micro_batches//pp=1

   以pp_size=8,vpp_size=5,micro_batches=9为例(即1个完整group+1个余数group)：

   证明如下：

   我们在启发式算法优化remainder*2<pp的bubble可知，当vpp>=2的时候，余数组的forward提前在warmup阶段，**仅仅在前一组的最后一个pp_stage出现在1F1B阶段，此时remainder<pp/2，而前pp/2的forward已经完成了全部chunk的forward，因此可以利用其最多pp/2个backward和余数forward组成1F1B**，又因为remainder<pp/2，所以一定不会有断开的1F1B，因此不会增加bubble。

b. vpp=2

以pp_size=8,vpp_size=2,micro_batches=25为例(即3个完整group+1个余数group)：

即16~23与最后一个第24个micro_batches做穿插

证明如下：

因为我们知道，余数group只和最后一个完整的group做交互，且穿插在完整group的chunk1~chunkn之间，chunk0是不受影响的，因此，**当vpp=2时，即只在最后一个chunk做交互，此时完整group的前pp/2个forward一定已经完成了全部的forward并开始做backward，因此最多可以用pp/2个bakcward与余数的forward组成1F1B**，而余数又小于pp/2，所以一定不会有断开的1F1B，因此不会增加bubble。

2. 手动优化中值插入法的编排

pp_size=8,vpp_size=3,micro_batches=17

优化前：

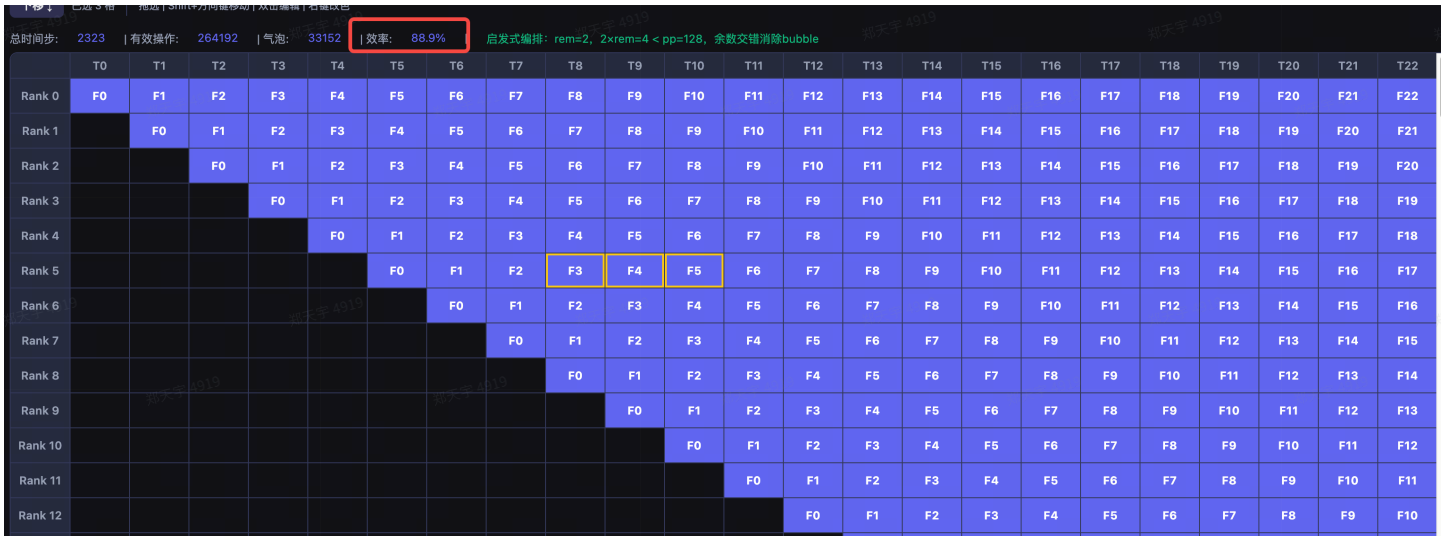
优化后：

可以看到，仍然可以将新增的bubble优化掉，核心思想在于让当前chunk的前pp//2能够先执行，因此为了进一步扩展启发式算法的可用范围，在此又提出了一版级联算法进行优化。启发式算法优化remainder*2<pp的bubble

3. 当前中值插入法引入的bubble情况：

虽然中值插入法，让1F1B断开，且会引入新的bubble，但是我们注意到，在pp，vpp，r相同的情况下，bubble不会随着micro_batches的增大而增大，只会在pp，micro_batches相同的情况下，随着vpp增加，但是增加的bubble其实并不多，这里展示一个相对大一点的例子：

pp_size=128,vpp_size=4,micro_batches=258，此时效率仍然能维持在90%左右。



4.级联算法(中值插入法扩展版)

首先我们可以直观看到中心插入法的问题在于**(余数group用r_group表示，最后一个完整group用b_group表示)**，当 `vpp>2` 且 `micro_batches//pp >1` r_group与b_group会在1F1B间交互，此时，由于每次插入了 r 个r_group的forward，因此后续整体的forward其实被延时处理了，每经过一轮stage，后续的所有stage的forward就会叠加延迟r个step，最终导致forward和backward在数量上无法对齐。

那么我们就需要思考在1F1B阶段该如何让forward和backward能够对齐，或者说让编排的forward都能有某个backawrd与之凑成一对FB。我们观察整除下的1F1B，可以发现其确保了每pp_size为一组变化F和B，从而确保FB在1F1B阶段的紧凑组合，因此我们可以想到一种方法，在r_group在b_group的forward中插入时，限制每pp_size个单位处理一次，并且，对于b_group的每组的前pp//2，保持为当前stage的forward，其余未处理的forward，累计到后续处理，这样就能保证，b_group每个stage的前pp//2的forward一定不会延迟，从而确保至少每轮有pp//2个backward可调度。

级联算法的动机

让 1F1B 阶段，每pp_size个F和B进行紧密组合，找到一种pp_size为一组编排forward的方案。

如上图所示，因为 1F1B 要求一个 F 一个 B 交替。当b_group和r_group交互时，此时填充的backward，为b_group前一组完整group的backward，而每个stage是标准的pp_size个backward，因此我们需要每pp_size个forward与之对应。从而完美配对，不产生bubble。**(余数group用r_group表示，最后一个完整group用b_group表示，用rem表示余数)**

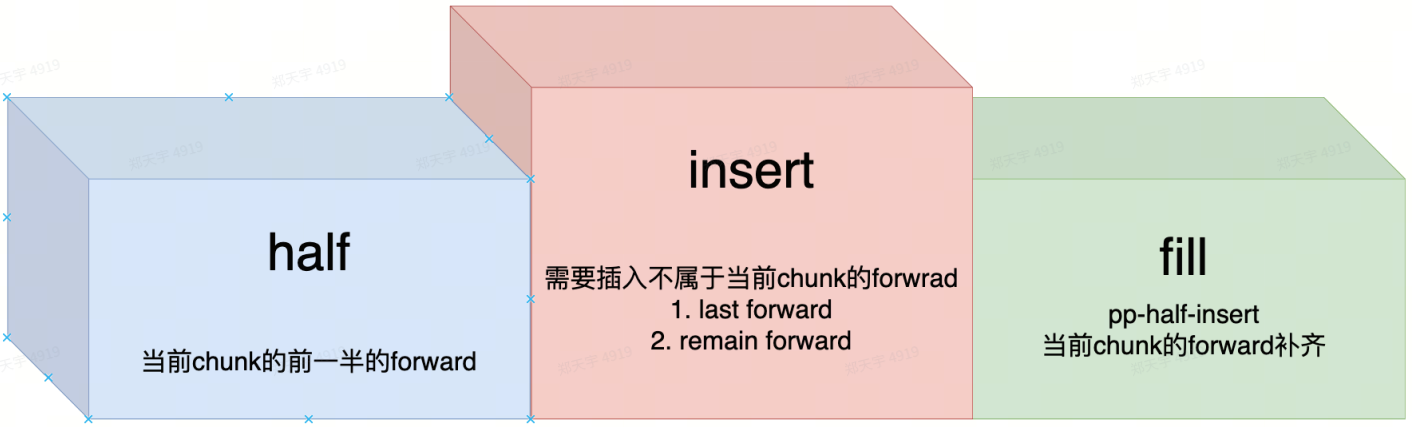
但问题是：最后一个 b_group 里，每个 chunk 实际有 `pp + rem` 个 forward（`pp` 个 当前stage的forward + `rem` 个余数的forward）。直接排就会出现大小为 `pp + rem` 的 block，比可用的 `pp` 个 backward 多出 `rem` 个，产生 `rem` 个 bubble。

级联的核心思想：

延迟处理forward与确保1F1B阶段延迟的forward能backward被覆盖

把每组编排想象成一个固定大小为 `pp` 的箱子。每个 chunk 有 `pp + rem` 个 forward 要放，但箱子只能装 `pp` 个。因此需要**把装不下的 forward 延迟到下一个 chunk 的箱子里处理**。

具体来说，chunk c 的箱子里装三类东西：



1. `first = half = pp//2`：先放当前 chunk c 的前 `pp//2` 个 forward
2. `insert = last + remain`：然后需要插入当前chunk的 forward
   - `last`：上一个 chunk 装不下的 base forward
   - `remain`：当前 chunk 需要插入的 `rem` 个余数 forward
3. `fill`：最后用当前 chunk 剩余的后一半 forward 把箱子填满到恰好 `pp`

延迟处理forwrad：insert和fill用于确保延迟的forwrad能及时被处理

确保延迟的forward被覆盖：first部分确保了，b_group至少有pp/2的forward是没有被延迟处理的，因此backward可以如期执行，从而覆盖后半部分延迟处理，还不能进行backward部分。

级联算法的示例

pp=8, vpp=5, mb=17 (rem=1)

half = 4，每个 chunk 有 17 个 forward 要排。

c=0 的：直接放 8 个，满了。没有上一个chunk需要处理，没有余数forward需要处理。

c=1 的箱子（必须恰好 8 个）：

槽位	来源	数量
`first`	c1 的前半	4
`last`	c0 延迟处理的 base	0
`remain`	c1 需要处理的余数	1
`fill`	c1 补齐	8 - 4 - 0 - 1 = **3**
总计		**8**

c1 自己用了 4 + 3 = 7 个 base forward。剩余：8 - 7 = **1** 个 base 寄存 → 给 c2。

c=2 的箱子：

槽位	来源	数量
`first`	c2 的前半	4
`last`	c1 延迟处理的 base	1

`remain`	c2 需要处理的余数	1
`fill`	c2 补齐	8 - 4 - 1 - 1 = **2**
总计		**8**

c2 用了 4 + 2 = 6 个。剩余：8 - 6 = **2** 个 base → 给 c3。

c=3 的箱子：

槽位	来源	数量
`first`	c3 的前半	4
`last`	c2 延迟处理的 base	2
`remain`	c3 需要处理的余数	1
`fill`	c3 补齐	8 - 4 - 2 - 1 = **1**
总计		**8**

c3 用了 4 + 1 = 5 个。剩余 8-5=3 个 base -> 给c4

c=4 的箱子：

槽位	来源	数量
`first`	c4 的前半	4
`last`	c3 延迟处理的 base	3
`remain`	c4 需要处理的余数	1
`fill`	c4 补齐	8 - 4 - 3 - 1 = **0**
总计		**8**

c3 用了 4 + 0 = 4 个。剩余 8-4=4 个 base

我们考虑一下，如果继续增加chunk，即vpp_size,此时fill将变为负数，相当于我们要从箱子里取走forward，而实际我们只能做插入操作，因此vpp_size无法再增加了。所以我们要讨论一下该算法的适用范围。

级联算法的适用范围

我们观察到：**每过一个 chunk，`last` 增加 `rem`**（因为每个 chunk 都有 `rem` 个余数放不下）。所以 `insert` 逐 chunk 增长：

代码块

```
c=1: insert = rem         → fill = half - rem
c=2: insert = 2*rem       → fill = half - 2*rem
c=3: insert = 3*rem       → fill = half - 3*rem
...
c=vpp-1: insert = (vpp-1)*rem → fill = half - (vpp-1)*rem
```

`fill` 不能为负（不能从箱子里拿走东西，或者换句话说，fill<0表示，half+insert会超过pp_size，此时就无法保证1F1B的紧密编排了），所以：

代码块

```
half - (vpp-1)*rem >= 0
(vpp-1)*rem <= half = pp//2
```

一旦违反，最后一个 chunk 的箱子里要塞的东西超过 `pp` 个，对齐就不可能了。因此，级联算法的最终条件为：(vpp-1)*rem <= pp//2

级联算法最大限度地确保了1F1B的紧密交替排列，一旦打破则1F1B必定断开，同时在此条件下尽可能地去提前余数forward的处理，而这也进一步证明了，在不满足三种方法（边缘插入，中值插入，级联）的范围内，一定无法找到一种编排使得bubble不增加。

注意：级联算法有vpp的限制，但是当micro_batches//p=1 或 vpp=2时，采用中心插入法不会新增bubble，且没有vpp的限制，因此在这两种情况下仍然适用中值插入法。

5.结论

1. 当r>=pp//2时，无需使用启发式算法。
2. 当 `pp//4<=r<pp//2` 时，使用边缘插入法（**若pp不整除4，则必须有r*4>=pp**）
3. 当 `r<pp//4` 时，且 `micro_batches//p=1 或 vpp=2` 时，使用中心插入法
4. 当 `r<pp//4` 时，且 `micro_batches//pp >1 && vpp>2 && (vpp-1)*r <= pp/2)` ，使用级联算法
5. 否则，无法使用启发式算法，即一定无法找到一种编排不新增bubble(证明见 启发式算法优化remainder*2<pp的bubble)。

3.启发式插入算法总结

这里以中值插入法为例，边缘插入法与之区别仅仅在于插入位置不同，级联算法上文已经解释比较清楚

一、启发式条件

当且仅当同时满足以下条件时，启用启发式算法：

1. rem > 0　　(存在余数)
2. rem * 2 < pp　　(余数足够小，可以被"塞入"间隙)
3. rounds > 0　　(至少有一个完整的 round)
4. vpp > 1　　(存在多个 virtual pipeline chunk)

二、Forward 编排算法

核心思想: 先按整除方式编排 forward，然后将余数 forward 插入到

最后一个 round 的各 stage 中间，使流水线保持满载。（实际开发时，可以直接交替插入remainder和上一轮与remainder交换的本该编排的forward，最后检查剩余未编排的forward，从低chunk开始从小往大编排。）

前置定义:

base = (mb // pp) * pp

rem = mb - base

rounds = base // pp

half = pp // 2

步骤 1 — 前 rounds-1 个 round 标准编排:

for rd in [0, rounds-2]:

  for c in [0, vpp-1]:

    fwdGroups.append(chunk=c, size=pp)

步骤 2 — 最后一个 round 的 stage_0 全排:

fwdGroups.append(chunk=0, size=pp)

步骤 3 — stages 1..vpp-1: 在 pp//2 处插入 stage c-1 的余数:

for c in [1, vpp-1]:

  fwdGroups.append(chunk=c, size=half)   ← 当前 stage 的前半

  fwdGroups.append(chunk=c-1, size=rem)   ← 上一个 stage 的余数

  fwdGroups.append(chunk=c, size=pp-half)   ← 当前 stage 的后半

含义: 每个 stage 的 pp 个 forward 被分成前半和后半，

    在中间插入上一个 stage 的 rem 个余数 forward。

    余数只插入 vpp-1 次 (stages 1..vpp-1)。

步骤 4 — 尾部: 最后一个 stage 的余数:

fwdGroups.append(chunk=vpp-1, size=rem)

含义: 最后一个 stage 自身的余数 forward 放在最末尾。

示例 (pp=4, vpp=4, mb=5):

一开始整除编排: [4, 4, 4, 4]

插入 remainder 后:

stage_0: [4] → F0 F1 F2 F3

stage_1: [2, 1, 2] → F0 F1 | F4_s0 | F2 F3

stage_2: [2, 1, 2] → F0 F1 | F4_s1 | F2 F3

stage_3: [2, 1, 2] → F0 F1 | F4_s2 | F2 F3

tail: [1] → F4_s3

合并后 sizes: [4, 2,1,2, 2,1,2, 2,1,2, 1]

示例 (pp=6, vpp=3, mb=14):

早期 round: [6, 6, 6] → mb 0~5

stage_0: [6] → mb 6~11

stage_1: [3, 2, 3] → mb 6~8 | mb 12~13_s0 | mb 9~11

stage_2: [3, 2, 3] → mb 6~8 | mb 12~13_s1 | mb 9~11

tail: [2] → mb 12~13_s2

合并后 sizes: [6,6,6, 6, 3,2,3, 3,2,3, 2]

三、Backward 编排算法

Backward 列表初始时按照 "最大 chunk 优先, 同 chunk 内 mb 从小到大" 排列:

```
for c in [vpp-1, vpp-2, ..., 0]:
  for mb_id in [0, 1, ..., mb-1]:
    bwdList.append(B(mb_id, c))
```

原理:

- 高 chunk (大 stage) 的 backward 优先调度，来加速低stage的backward的开启
- 同 chunk 内按 micro-batch ID 从小到大，主要是与Forward的micro-batch ID处理顺序对应，按序释放activation。

在模拟调度时，backward列表的执行顺序会动态调整。

四、Warmup 计算

启发式模式下，warmup 需要区分两种情况:

Case 1 — rounds == 1 且 vpp > 2:

余数 forward 插入在 warmup 阶段:

warmup = (vpp - 1) * pp + (pp - rank - 1) + (vpp - 2) * rem

Case 2 — 其他情况:

vpp=1或余数 forward 插入在 1F1B 阶段:

warmup = (vpp - 1) * pp + (pp - rank - 1)

五、模拟调度

标准 interleaved 1F1B 调度:

1. Warmup 阶段: 只执行 forward
2. 1F1B 阶段: 1F1B 阶段严格一个 forward 一个 backward 交替
3. Cooldown 阶段: 只做 backward

依赖检查:

- F(mb, ck) 依赖 F(mb, ck-1) 完成 (前一个 chunk 的 forward) 此算法下一定满足
- B(mb, ck) 依赖 F(mb, ck_max-1) 完成 (该 micro-batch 所有 chunk 完成 forward) 需要检验
- B(mb, ck) 依赖 B(mb, ck+1) 完成 (后一个 chunk 的 backward) 需要检验

backaward的特殊情况处理 (调度时动态检测):

1. 检查backward时序合理性：

当准备调度 B(mb, ck) 且 ck < vpp-1 时，假设当前在第t步，检查开区间 (t - pp_size, t) 步内是否存在 B(mb, ck+1)。注意区间内1个forward或1个backward均算1步，注意空的必要bubble也算1步。

若存在，说明 backward 链尚未通过足够的中间 rank 传播完成，此时跳过该 op，优先调度其他可用的 backward。待 B(mb, ck+1) 退出该窗口后再调度 B(mb, ck)。

注意，实际在Schedule构造编排时，不需要构造模拟全部rank的调度，只模拟单个rank的调度，从而提高效率，理由如下：

1. 插入算法的处理，使得forward天然遵循F(mb, ck) 依赖 F(mb, ck-1)
2. 确保B(mb, ck) 依赖 B(mb, ck+1) 完成，建一个本rank的bwd_done_list即可
3. 确保B(mb, ck) 依赖 F(mb, ck_max-1) 完成，建立一个本rank的fwd_done_list即可
4. backward的特殊情况需要检验，也可以根据公式计算出检验窗口为：(t - pp_size, t) ，t为当前backward所在位置。
5. 每个rank执行forward/backward的顺序一致（单看forward或单看backward），因此都以最后一个rank来模拟，得到forward和backward两个表的执行顺序，而F和B交替执行的顺序，由实际调度过程确定。（warmup、1F1B阶段的step数）

4.bubble优化效果

1. bubble公式总结

以下 `T` 表示1个forward+1个backawrd的时间， `*pp` 表示计算所有rank的bubble。

1. 优化前的bubble公式：

$$(pp-1+(\frac{pp}{2}-remain)*(vpp-1))*T$$ *pp

2. 优化后的bubble公式：

$$(pp-1)*T$$ *pp

所以优化后相比优化前减少的bubble率为：

$$\frac{(\frac{pp}{2}-remain)*(vpp-1)}{pp-1+(\frac{pp}{2}-remain)*(vpp-1)}$$

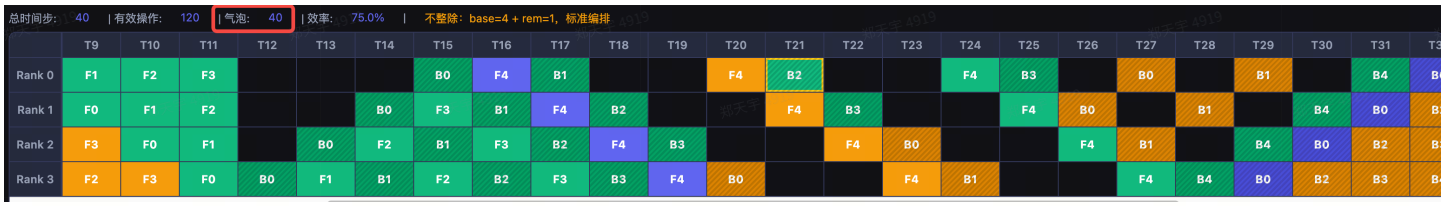
可以发现，随着vpp的增大，优化前后减少的bubble率趋近于100 %

2. 示例展示：

以下假设F和B的时间都为1个bubble，即T=2.

1. 示例1：pp=4，vpp=3，mb=5

优化前：



优化后：

Bubble: 40->24 降低 40 %

2. 示例2：pp=4，vpp=3，mb=5

优化前：

优化后：

Bubble: 52->24 降低 54 %

3. 示例3：pp=6，vpp=3，mb=7

优化前：

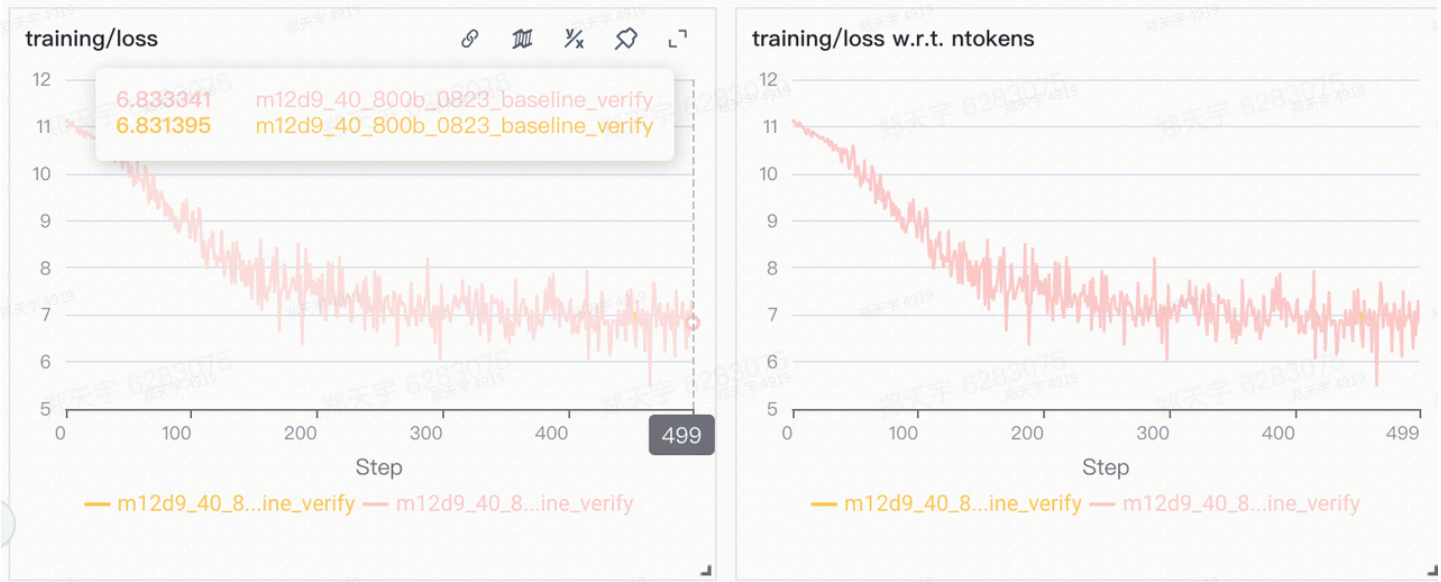
优化后：

Bubble: 108->60 降低 44 %

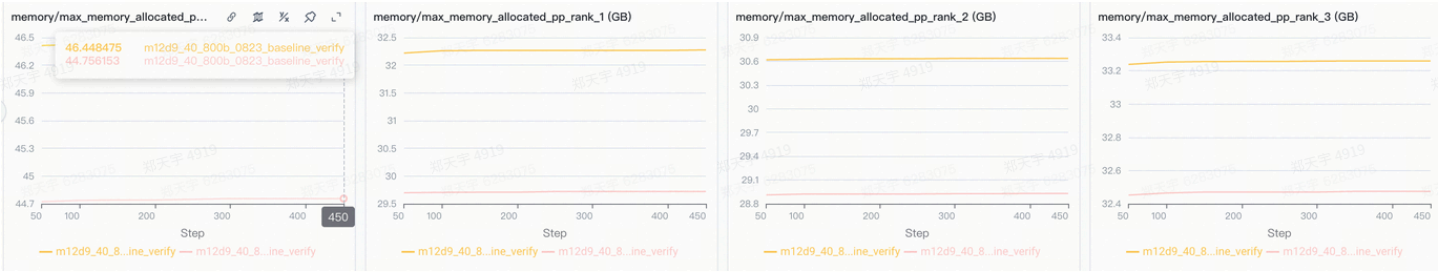
5.训练效果

1.pp_size=4,vpp_size=2,micro_batch=5(边缘插入)

loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_79af170f

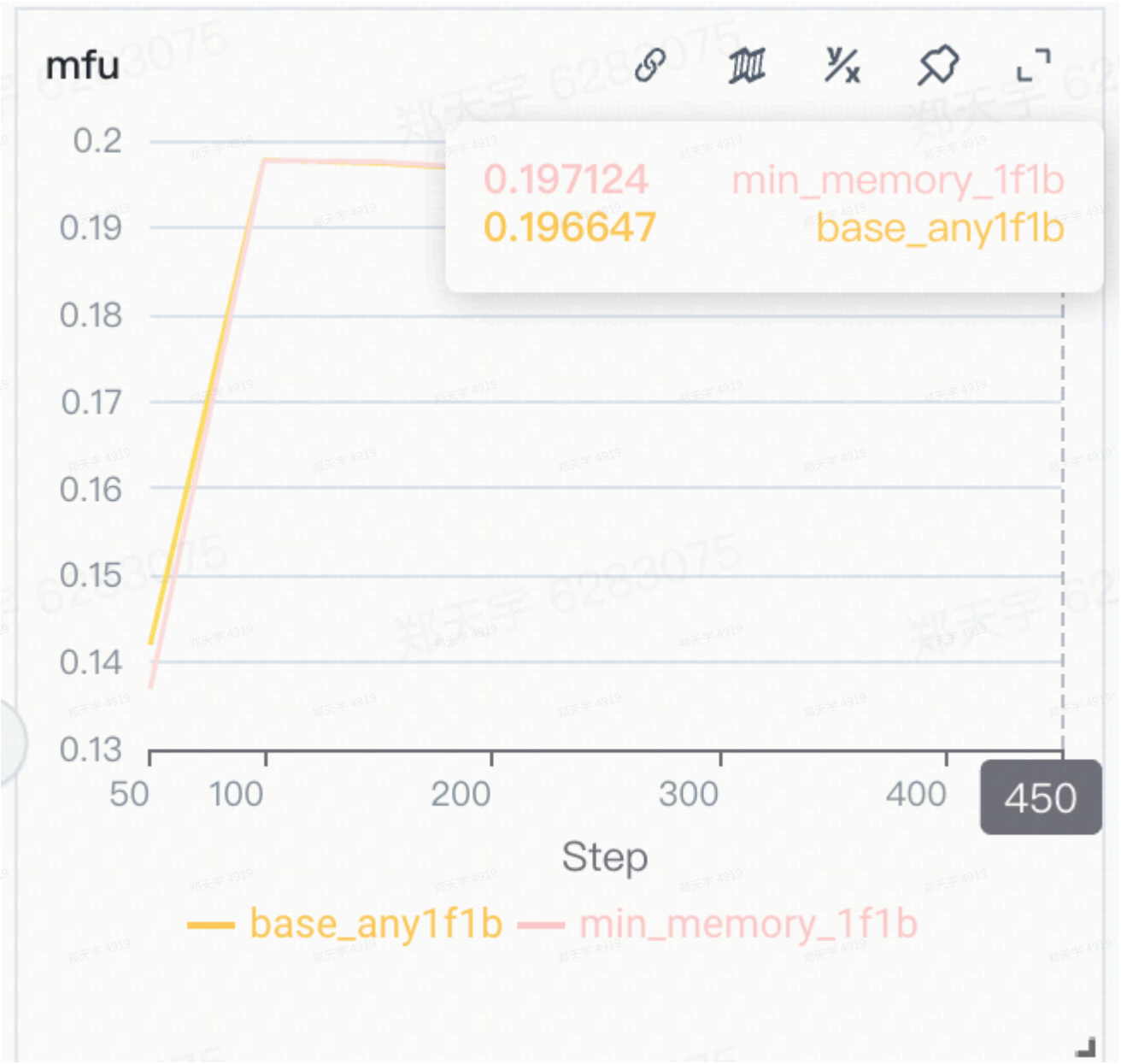


峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约1.8个G。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_6e382304



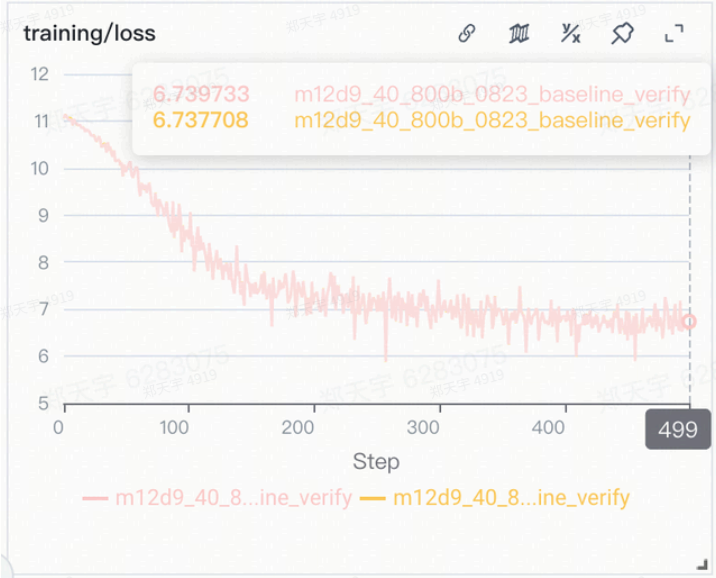
性能：（两者均只开1F1B的overlap）高于base_any1f1b,**约0.001。**

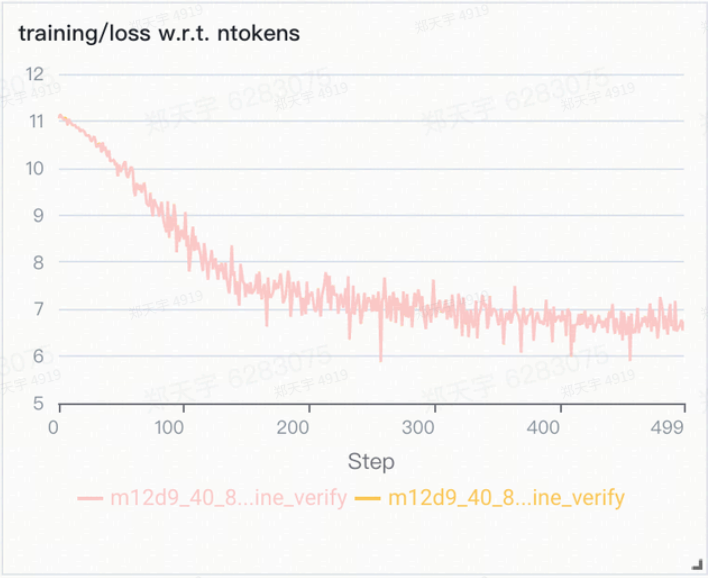
https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_a4d5a5e9



2.pp_size=4,vpp_size=2,micro_batch=9(边缘插入)

loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_ffbb0e96

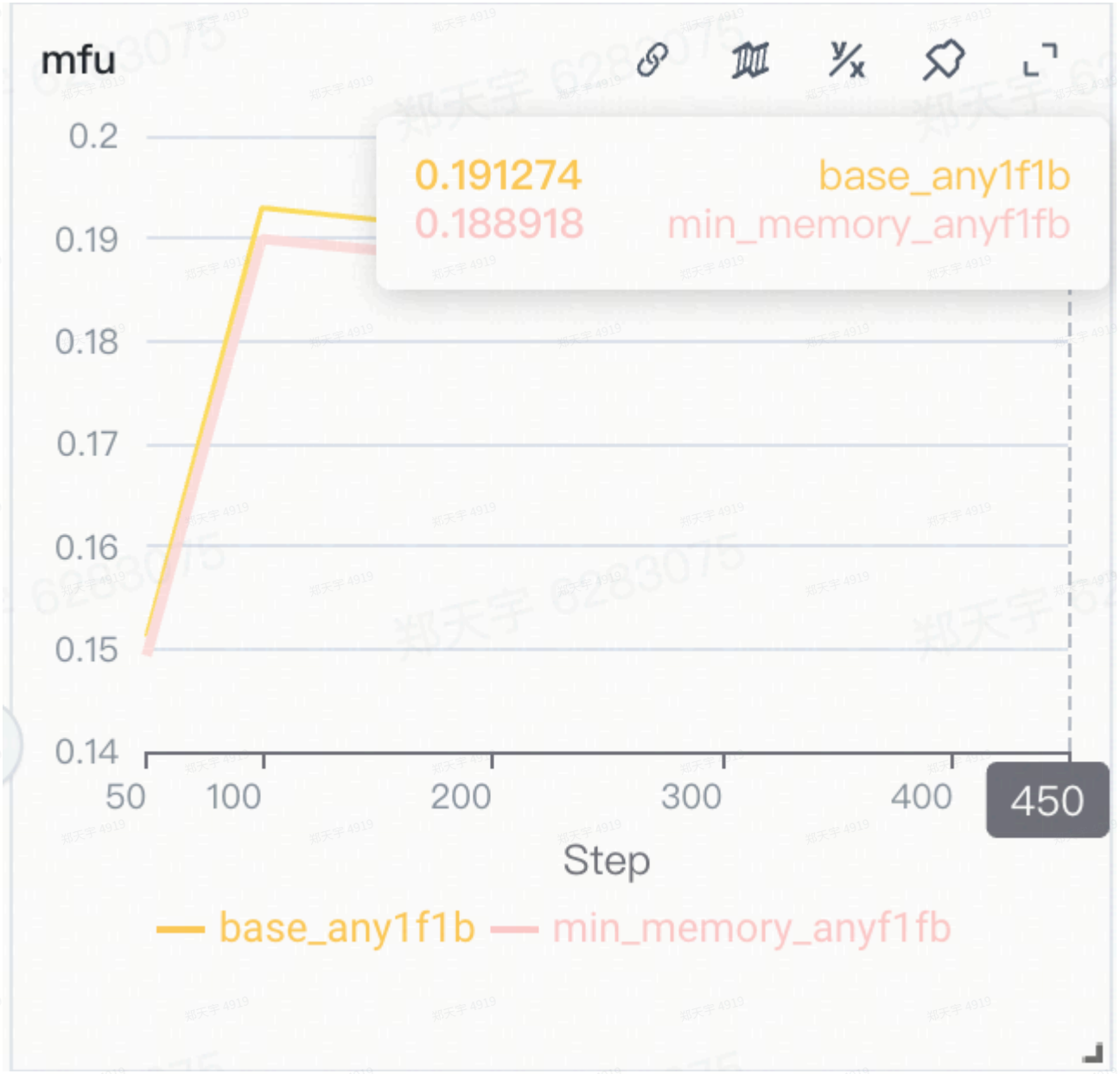




峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约3.5个G。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_40e4e08f

性能：（两者均只开1F1B的overlap）：低于base_any1f1b,**约0.003。**

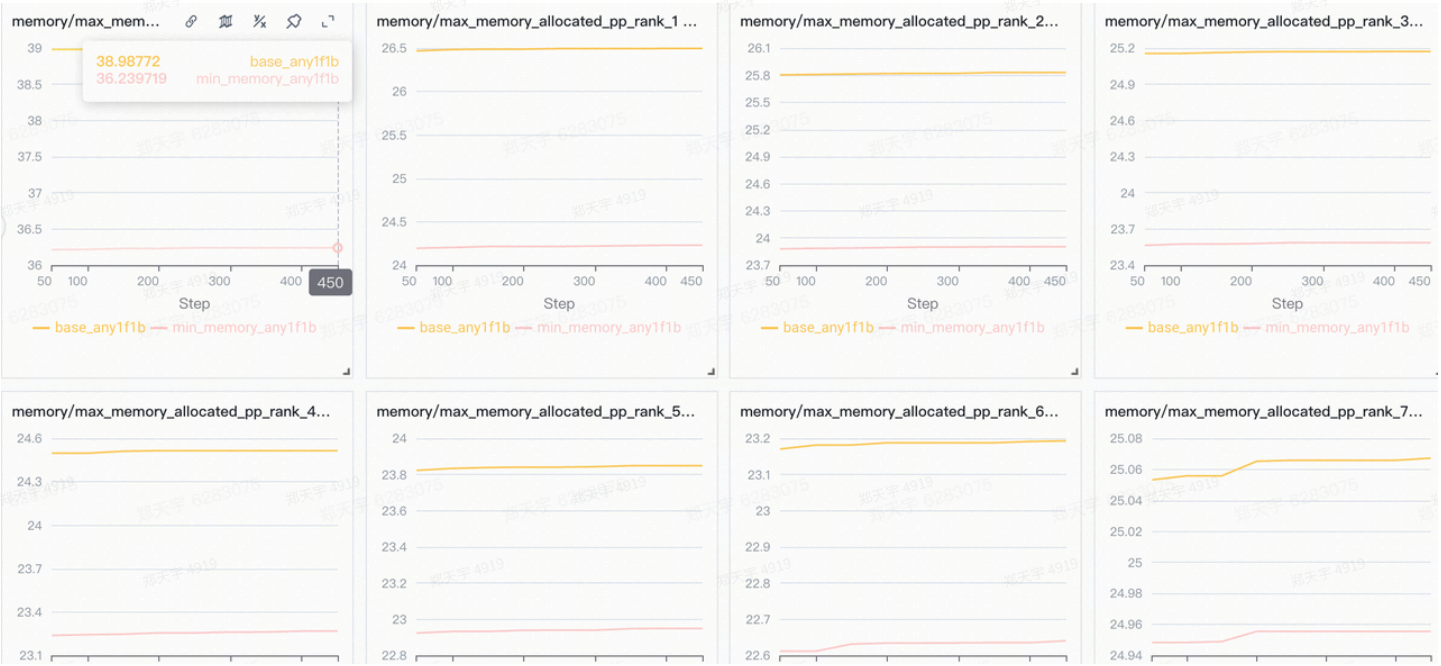
https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_e9303c37



3.pp_size=8,vpp_size=3,micro_batch=18(边缘插入)

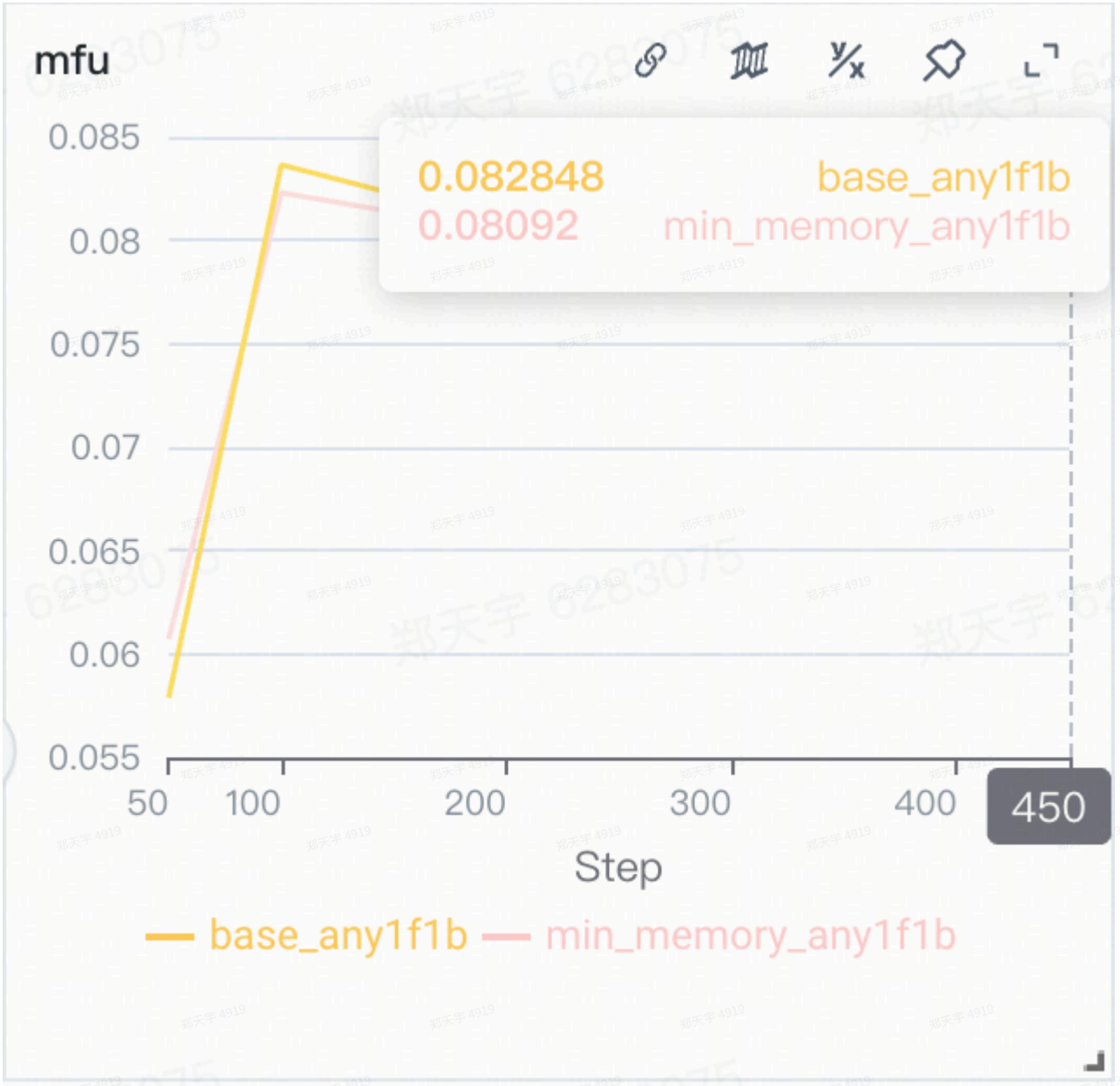
loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_5d757c05

峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约2.8个G。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_5d757c05



性能：（两者均只开1F1B的overlap）：低于base_any1f1b,**约0.002。**

https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_505efc70

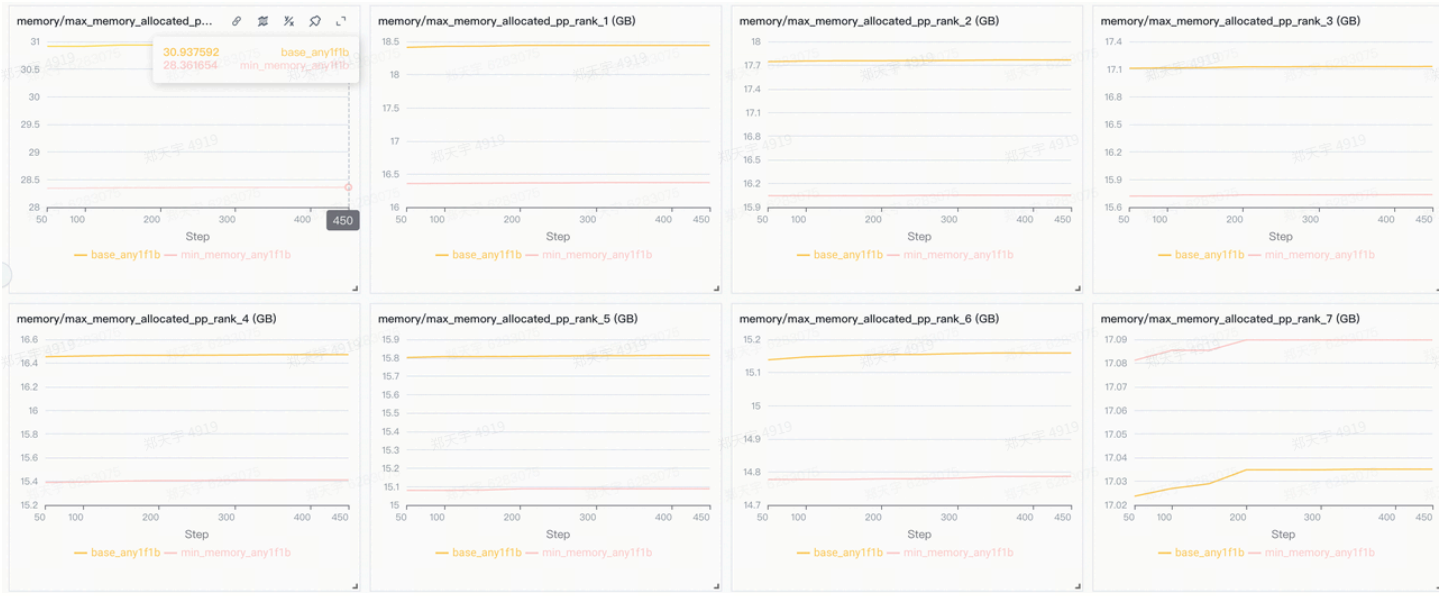


4.pp_size=8,vpp_size=2,micro_batch=18(中值插入)

loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_88023061

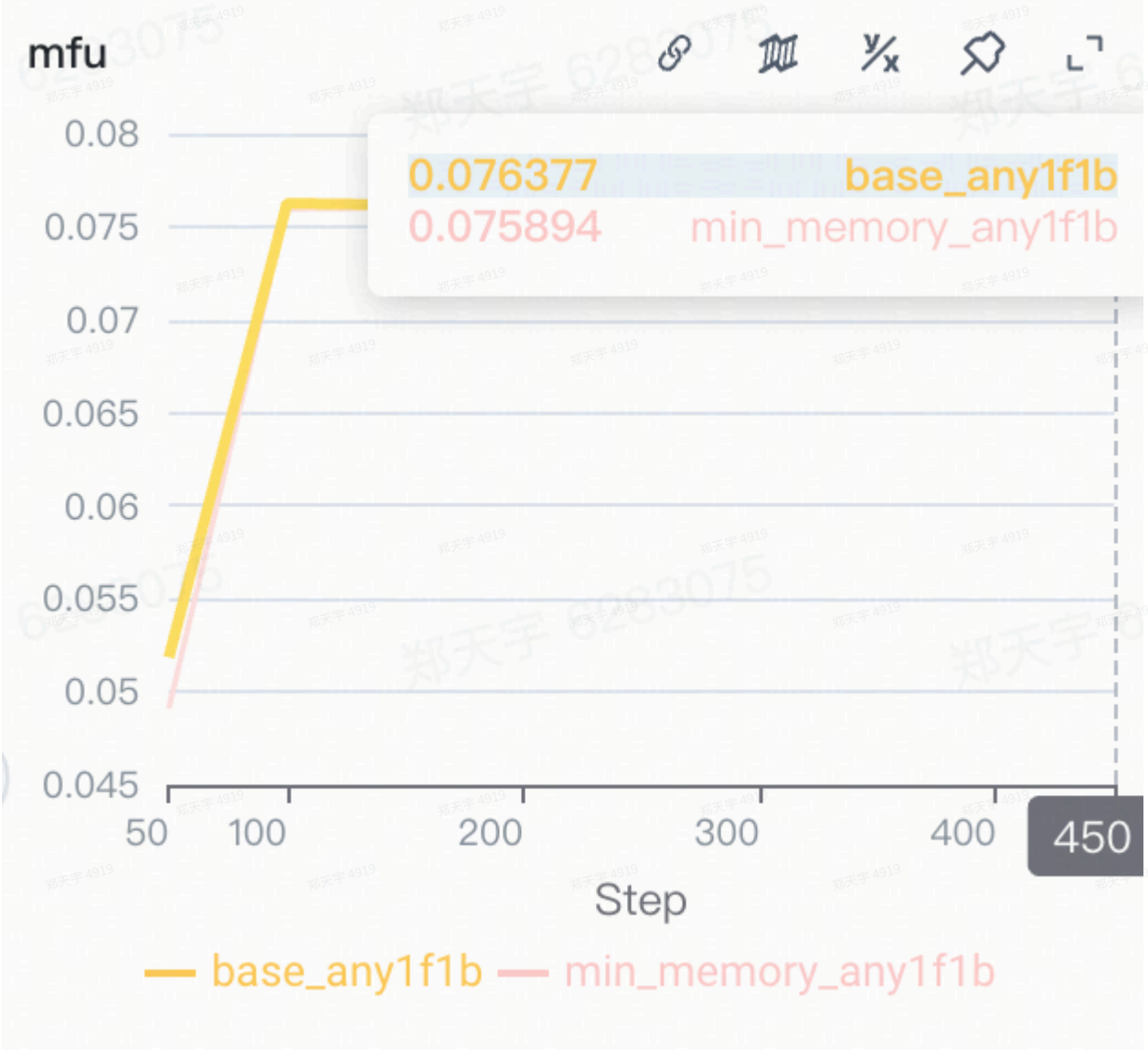
峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约2.6个G。

https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_ecd0b4ca



性能：（两者均只开1F1B的overlap）：低于base_any1f1b,**约0.0005。**

https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_95712385

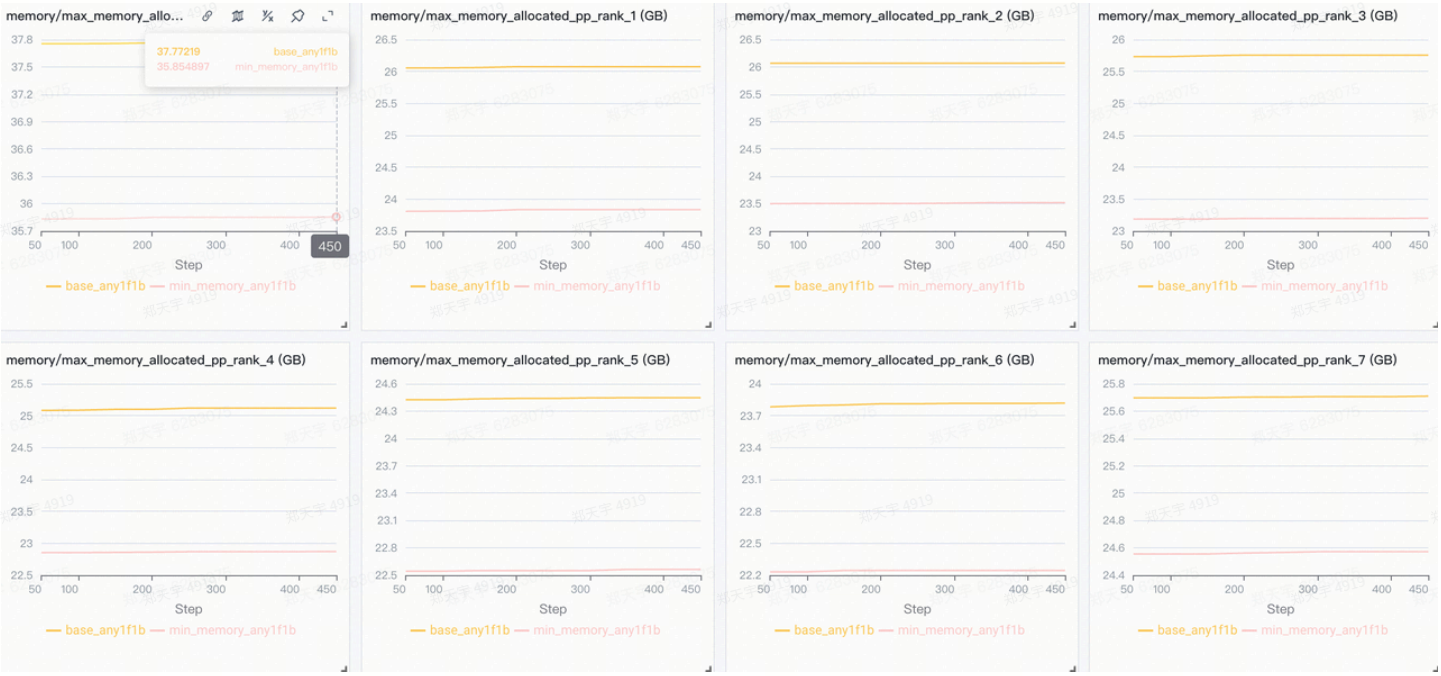


5.pp_size=8,vpp_size=3,micro_batch=10(中值插入)

loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。

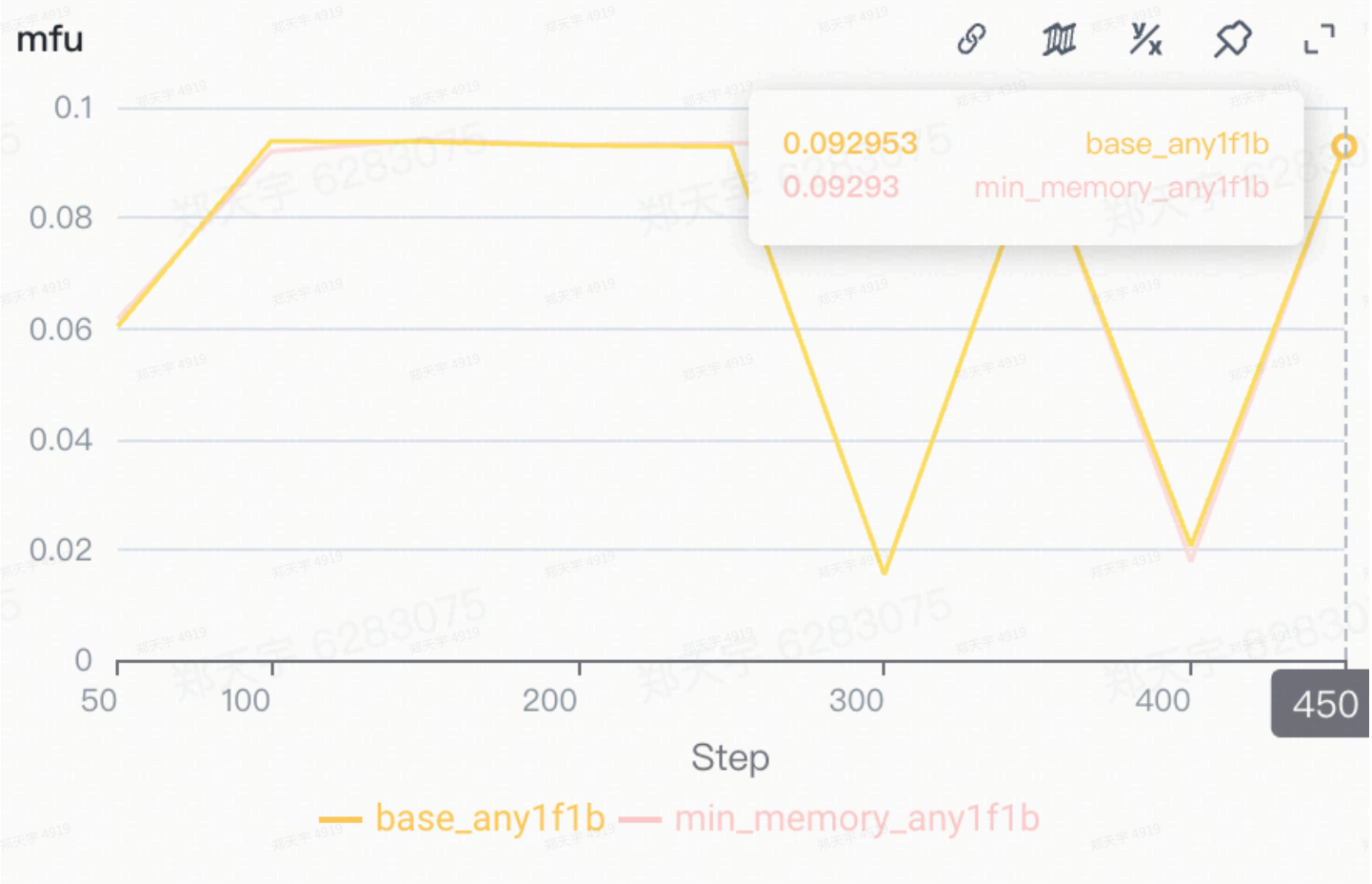
https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_18eb0dcc

峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约2个G。https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_e834f0b3



性能：（两者均只开1F1B的overlap）：低于base_any1f1b,**约0.00002。**

https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_16e8a5cd

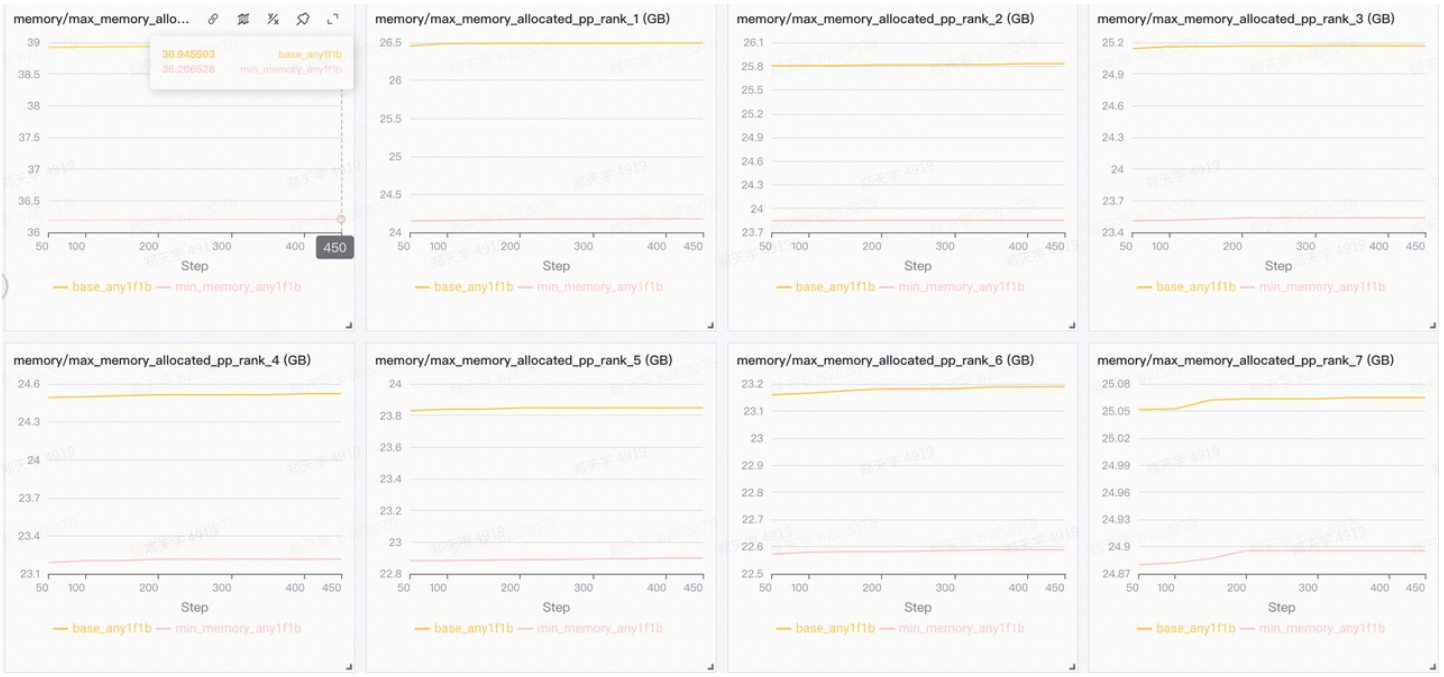


6.pp_size=8,vpp_size=3,micro_batch=17(级联插入)

loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。
https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_19193820

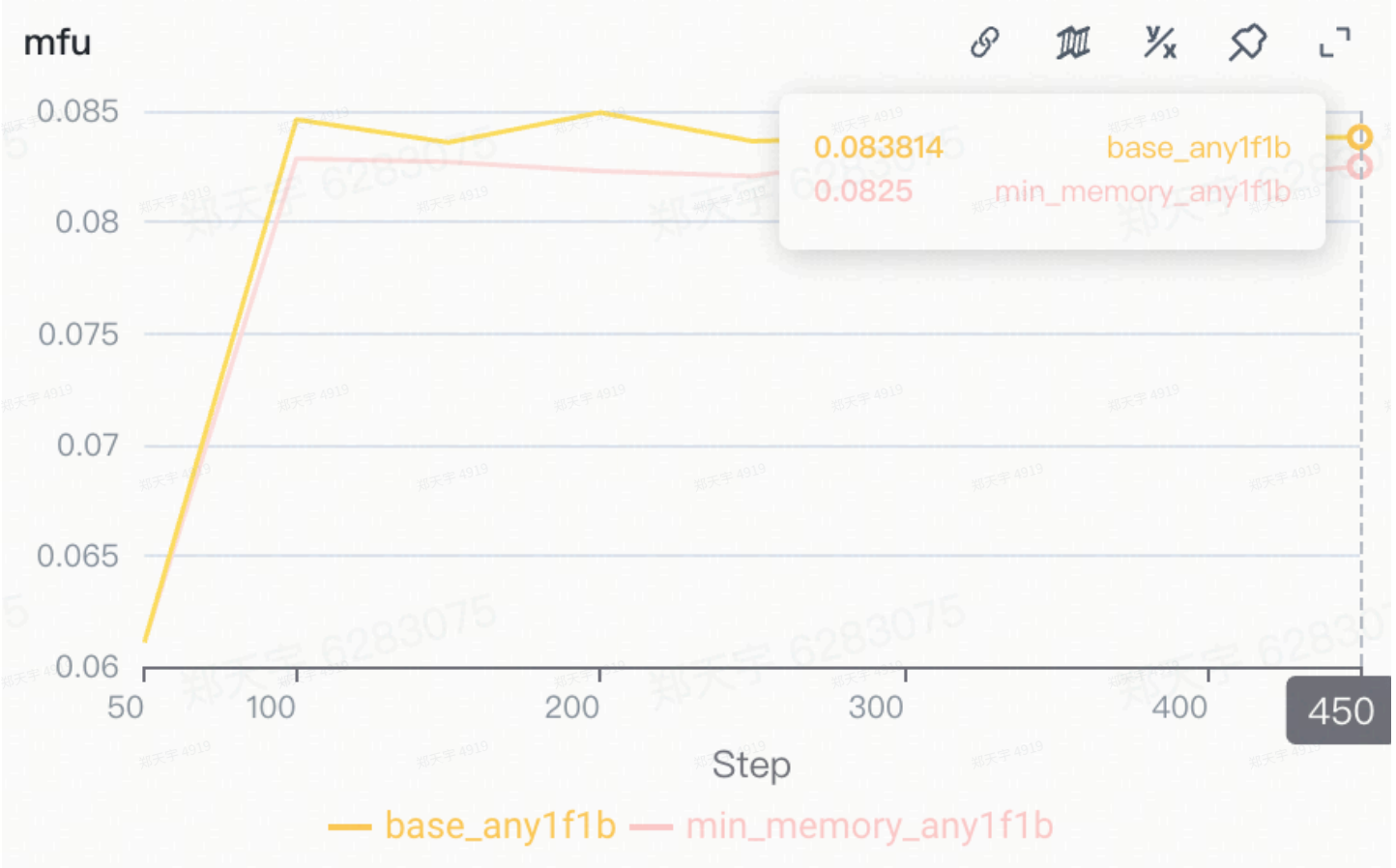
峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约2.7个G。

https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_2946b553



性能：（两者均只开1F1B的overlap）：低于base_any1f1b,**约0.001。**

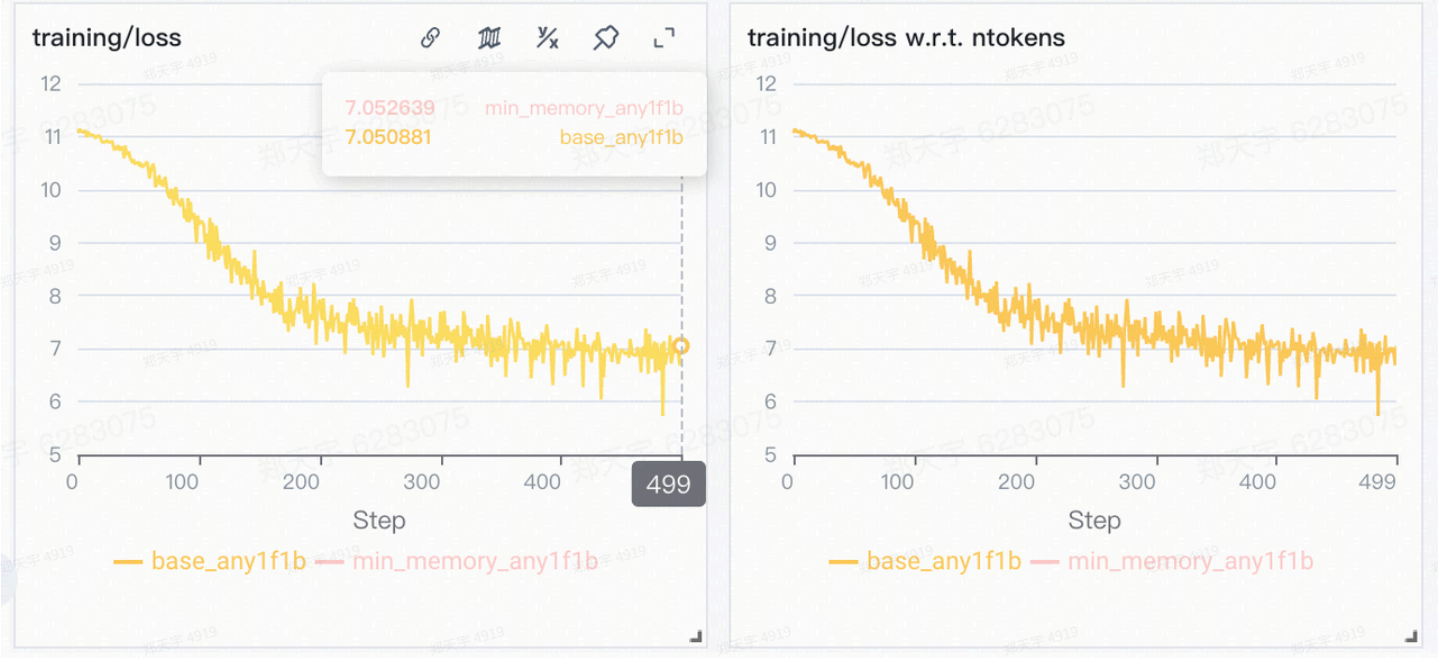
https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_d0a9849f



7.pp_size=8,vpp_size=4,micro_batch=17(级联插入)

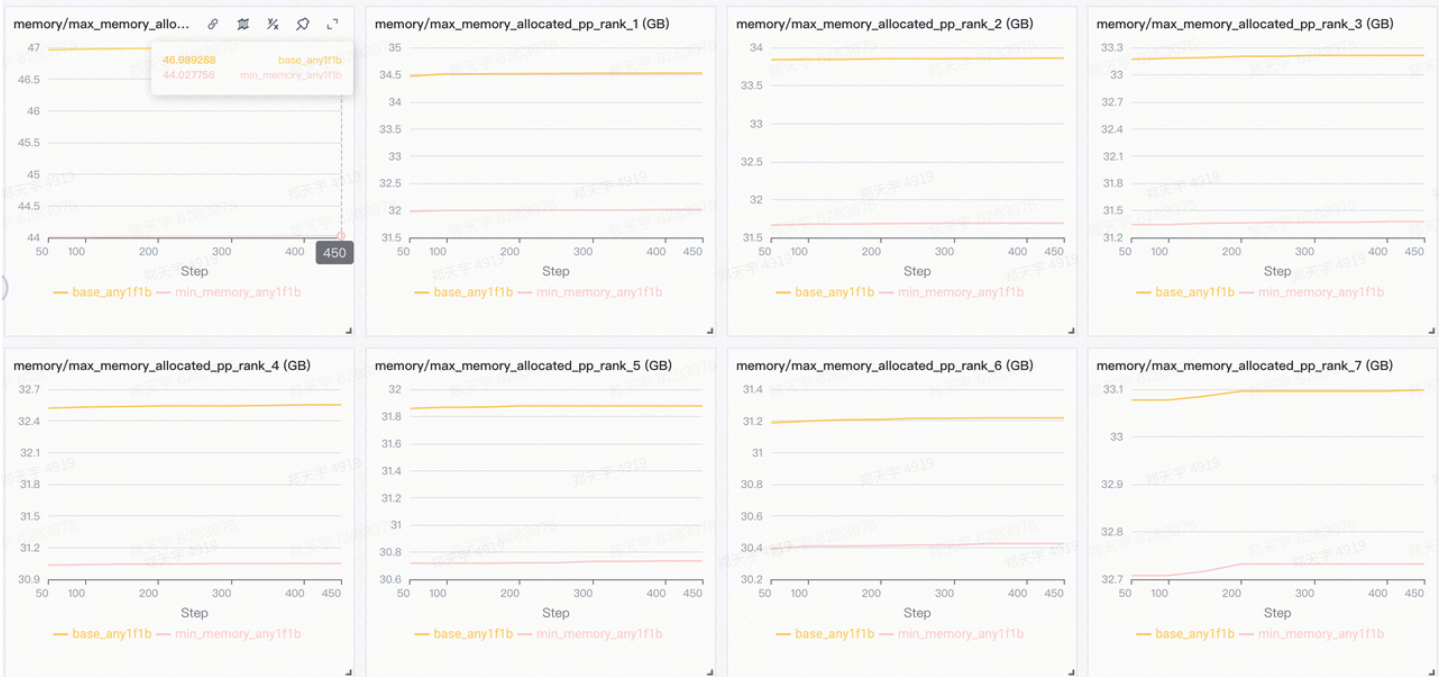
loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。

https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_64d42744



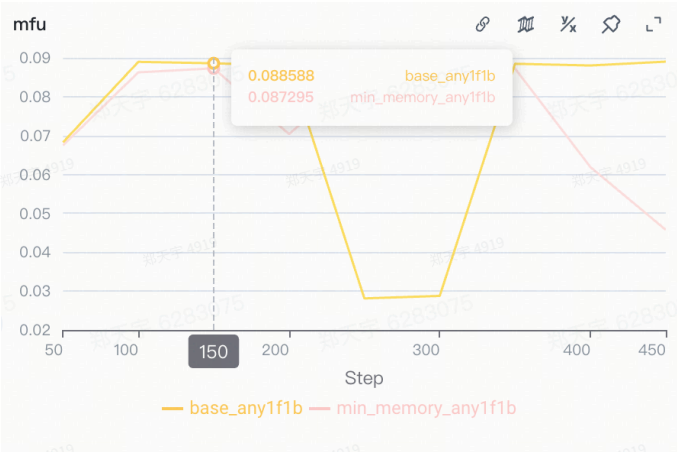
峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约3.0个G。

https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260414_e5046ce9



性能：（两者均只开1F1B的overlap）：比较两者最高点，低于base_any1f1b,**约0.017。**

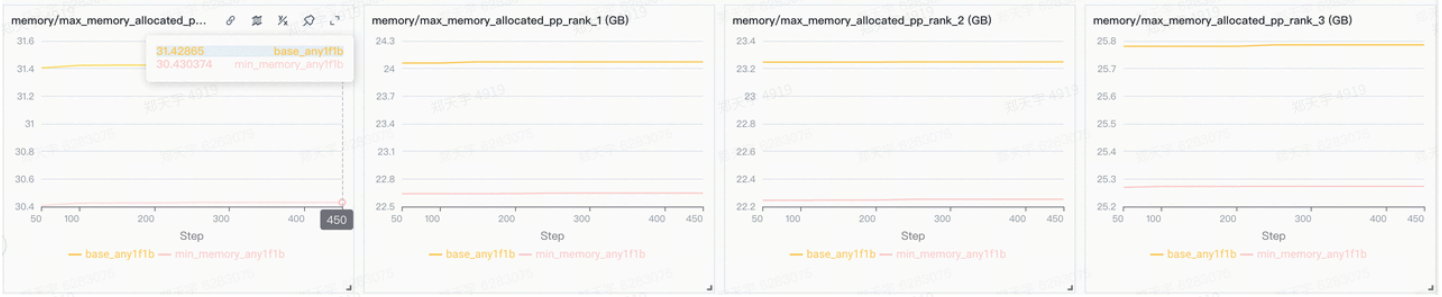
https://seed.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_9e3b2908



8.pp4dp2ep2dsp2oe2测试：pp_size=4,vpp_size=3,micro_batch=5

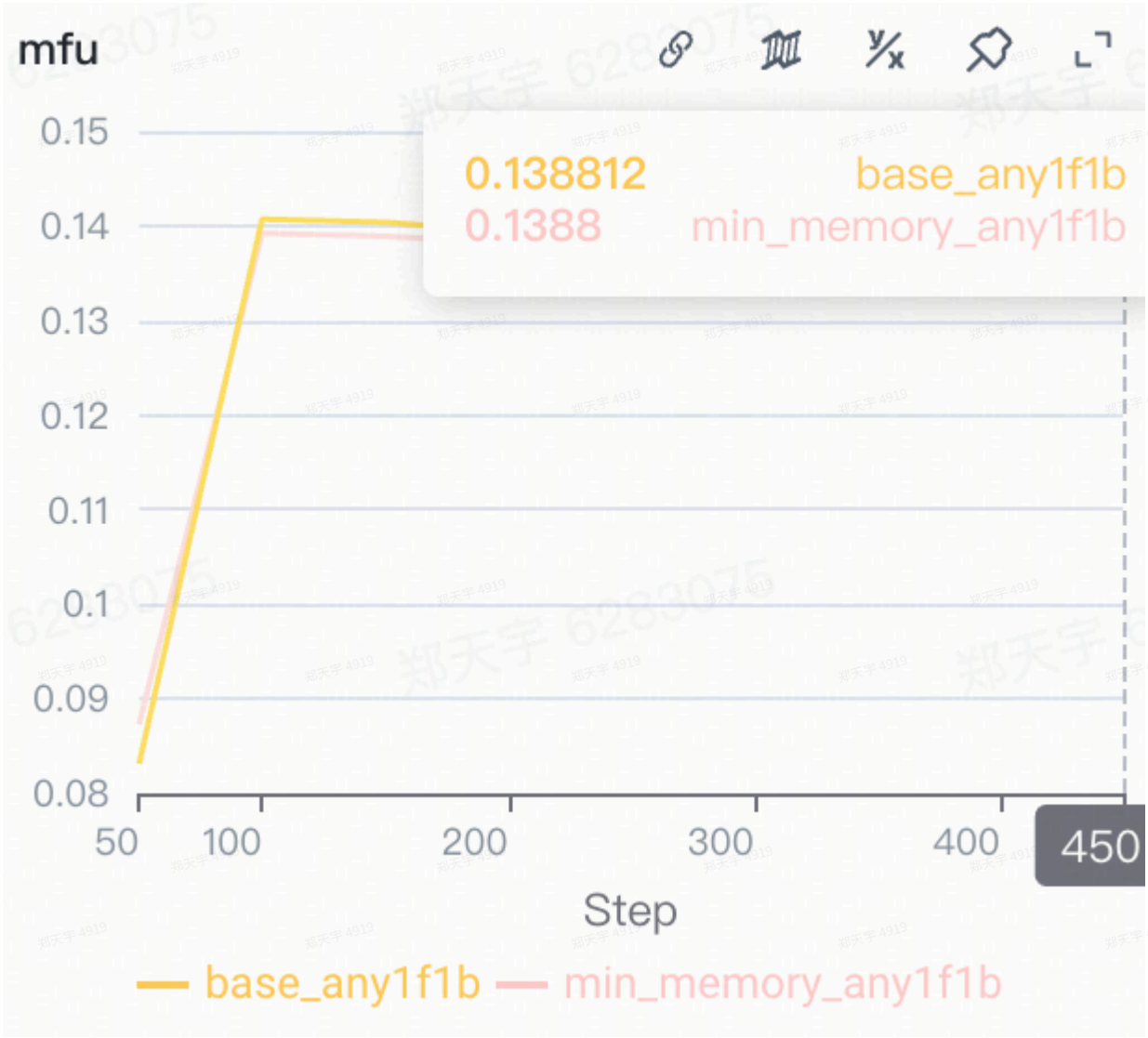
loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_d5ac8d47

峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约1.0个G。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_ea1597fe



性能：（两者均只开1F1B的overlap）：低于base_any1f1b,**约0.000012。**

https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_6016372b



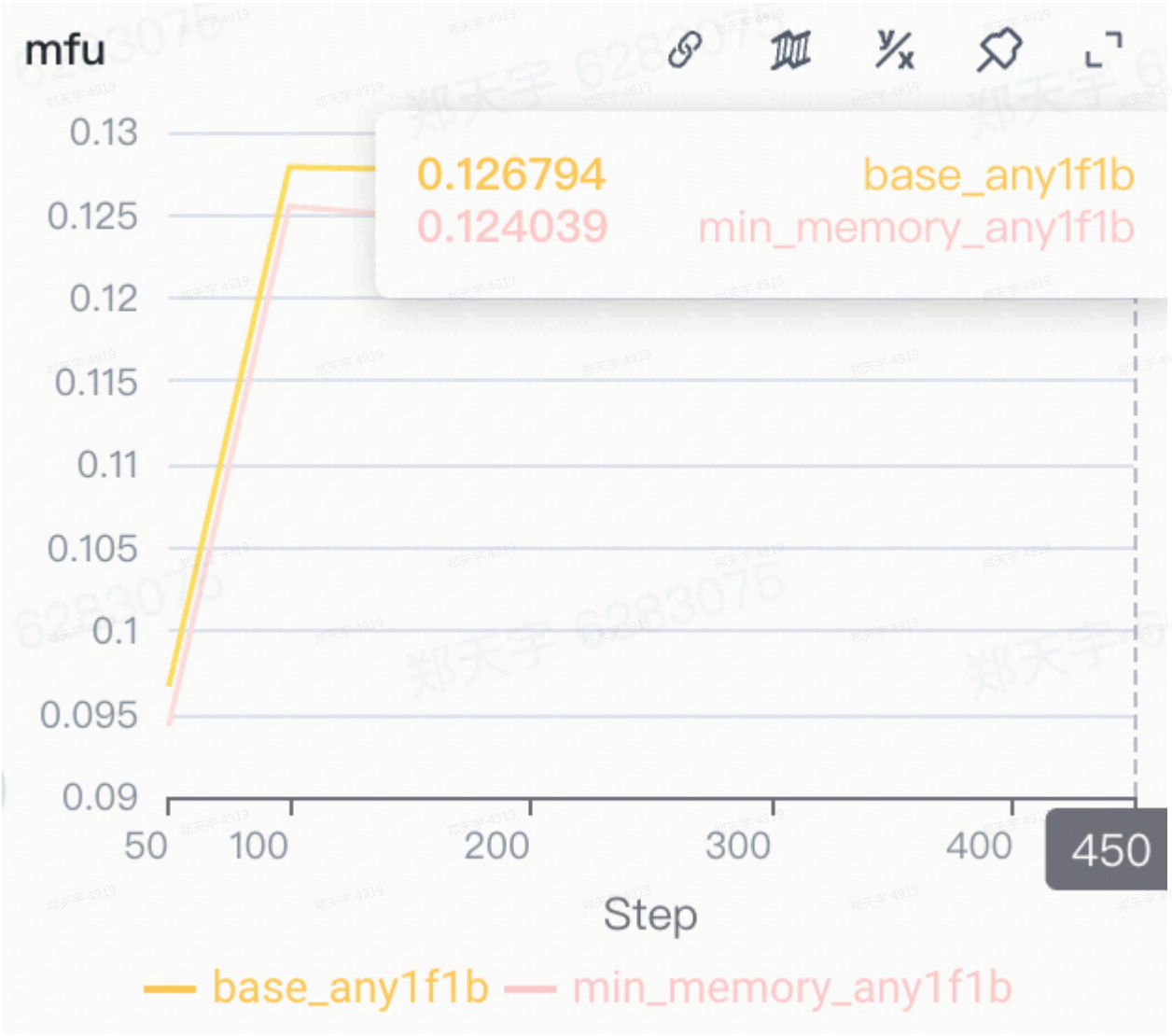
9.pp4dp2ep2dsp2oe2测试：pp_size=4,vpp_size=3,micro_batch=9

loss对齐： 和默认Any1f1b **不能bitwise对齐，符合预期**，因为两者在计算micro_batch时，计算顺序不同，梯度累积时，浮点数存在微小差异。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_81a5905d

峰值显存优化： 和默认Any1f1b相比，pp_0~pp_3的峰值显存均有降低，符合预期，其中pp_0降低最多约2.1个G。https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_3284eef8

性能：（两者均只开1F1B的overlap）：低于base_any1f1b,**约0.002。**

https://ml.bytedance.net/experiment/tracking/detail?Id=project_20250731_ee2582db&selectedTrial=&tab=CHART&viewId=view_20260415_987024f2



6.MR链接

https://code.byted.org/seed/mariana/merge_requests/14648