Qwen 3.5 优化设计文档

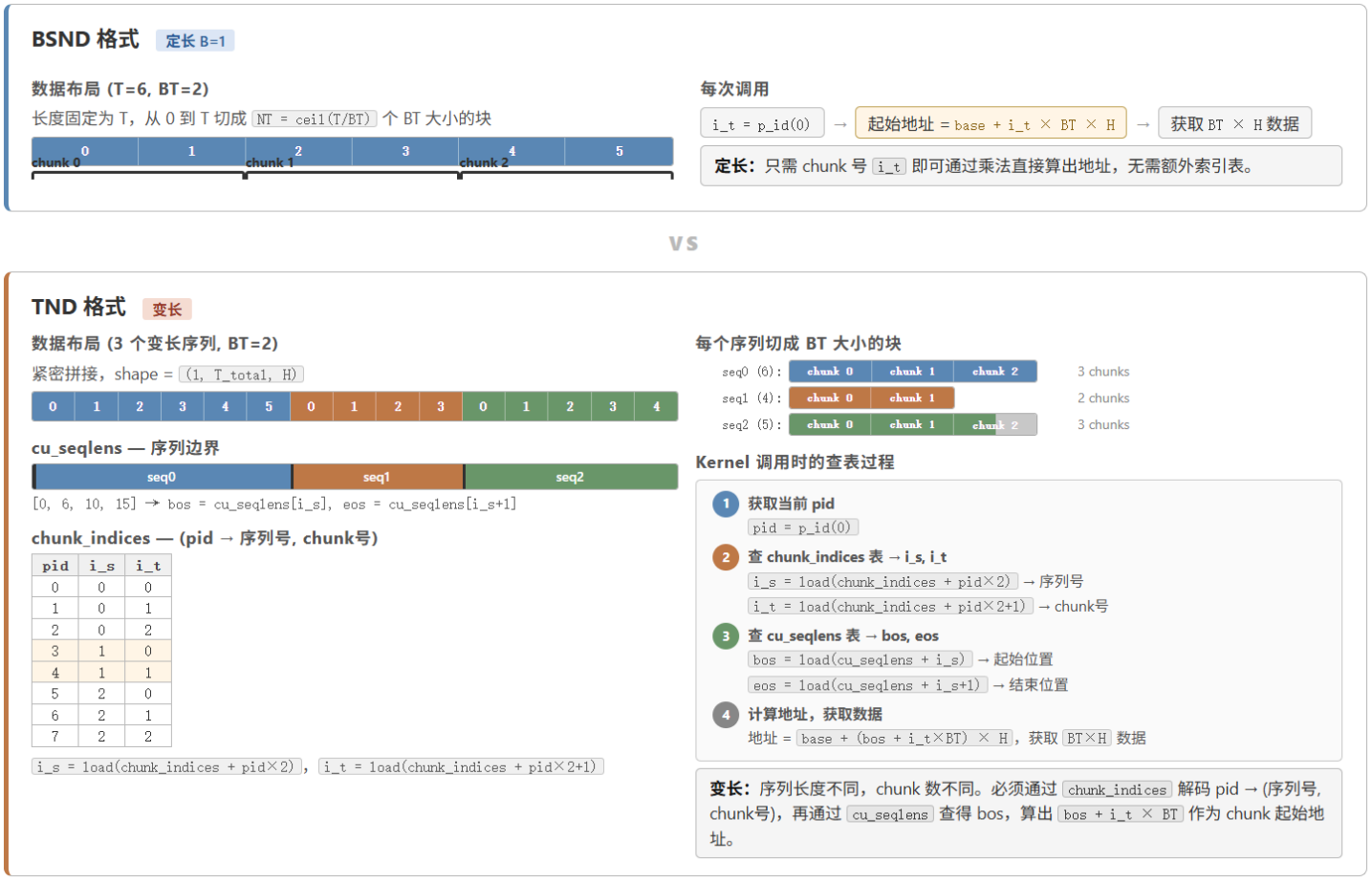
作者：谭邵杰等

1. 项目背景

下一代浦江模型基于 Qwen3.5 开发，为支撑国产大模型在国产算力的高效训练，项目组成功接入并落地了基于A3超节点的XTuner SFT极致性能优化方案和XTuner+vLLM ascend RL方案，本文对适配过程中的性能优化和精度对齐经验进行了总结，以供后续参考。

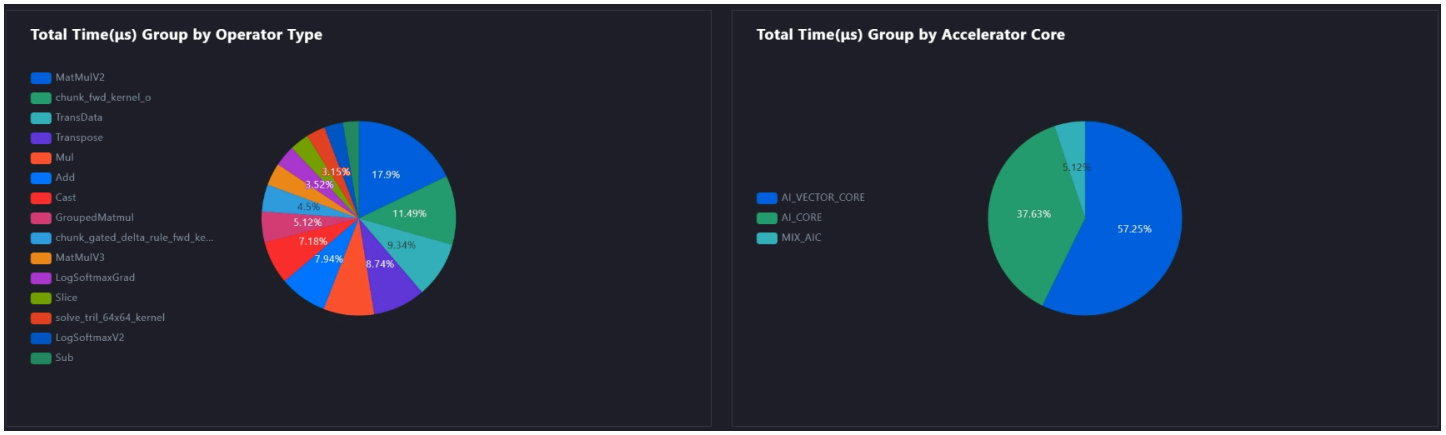
浦江现场的Qwen3.5 SFT相比家里或者其他局点，最不同的地方在于其使用的Xtuner训练框架原生支持TND格式（变长），即Xtuner会将多条数据合并为一条序列作为输入。而家里训练Qwen3.5使用的是BSND格式的输入，因此各种能力储备大多和BSND格式相关，这也导致现场难以复用家里优化经验。

如下图，数据格式的不同最重要的影响便是算子侧的开发，由于算子不能将输入视为一条完整的序列，因此需要在其内部增加额外的分块处理逻辑，如下图所示。而家里已有的算子在TND格式下的性能和精度都需要重新验证，甚至部分算子不支持TND格式，需要重新开发。

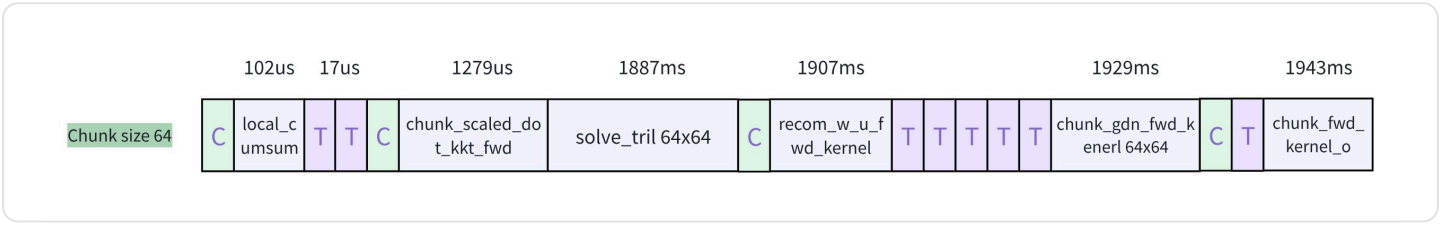


2. GDN性能优化

性能开箱时GDN模块的若干算子占了70%+的时间，为此优先优化GDN模块。



如下图，通过对GDN前向进行拆解：

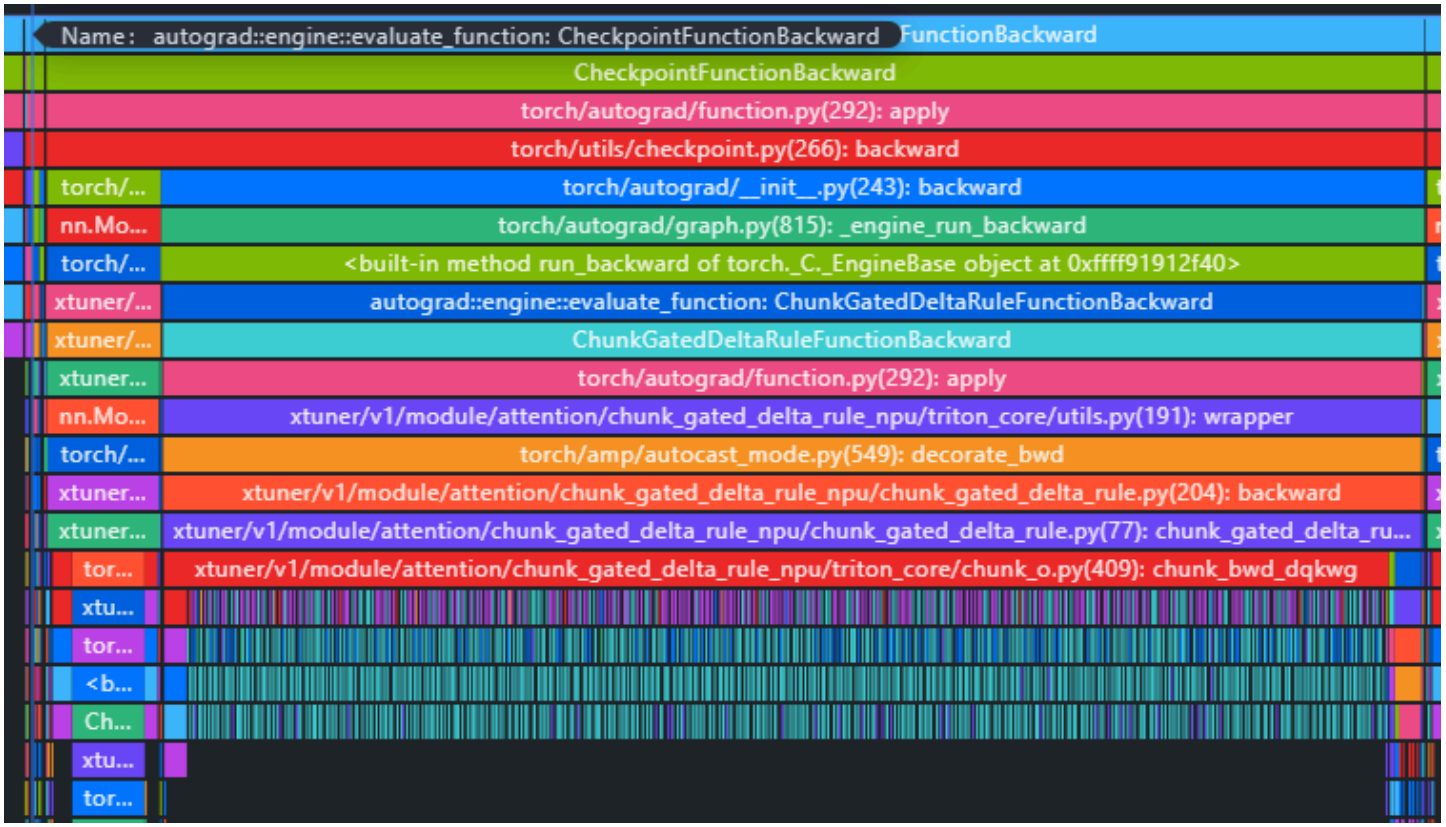


图中的T是transpose，C是chunk indices的CPU侧操作。

从图中可以看出若干优化点：Transpose操作需要消除；chunk indices操作能不能抽取，低效的triton实现以及占比重缺无ascend c实现的kkt和solve tril算子能不能重点优化。后续优化点围绕分析展开：

2.1 Triton劣化问题

不同于上图的分析，最初版本还有额外的问题。其GDN实现来自于mindspeed-core仓库中2.3.0_core_r0.12.1分支，通过采集到的profiling可以看到该版本存在严重的反向性能劣化，在反向过程中存在大量重复的小算子。



通过查看堆栈可知这堆小算子来自于chunk_bwd_dqkwg函数中的一段循环计算g_diff的代码，该段代码中包含大量小算子操作，且kernel_details.csv中显示小算子为MTE bound，从而导致反向性能急剧劣化。查阅与该函数相关的过往PR可知，这段代码的目的是为了提前计算g_diff矩阵，然后在Triton kernel中通过 `tl.load(p_gdiff)` 直接加载预计算的g_diff值来减少计算，这原本是为了减少计算而进行的空间换时间的优化，但是作者没有考虑在varlen场景下，需要根据chunk_indices的长度多次计算g_diff（在PJ场景下，需要进行上百次循环），从而引入大量小算子。将这段代码回退后性能由0.12x->0.25x。

代码块

```
for i in range(len(chunk_indices)):
    i_n = chunk_indices[i, 0]
    i_t = chunk_indices[i, 1]
    bos = cu_seqlens[i_n]
    eos = cu_seqlens[i_n + 1]
    cur_T = eos - bos

    o_t = i_t * BT + bias
    m_t = o_t < cur_T
    m_A = ((o_t[:, None] >= o_t[None, :]) & (m_t[:, None] & m_t))

    if bos + (i_t + 1) * BT > T:
        cur_g = torch.zeros((B, H, BT)).to(g.device)
        cur_g[:, :, :-(bos + (i_t + 1) * BT - T)] = g[:, :, bos + i_t * BT:bos + (i_t + 1) * BT]
    else:
        cur_g = g[:, :, bos + i_t * BT:bos + (i_t + 1) * BT]
```

```
        cur_gdiff = cur_g[:, :, :, None] - cur_g[:, :, None, :]
        cur_gdiff = cur_gdiff.clamp(-60, 60).exp()
        cur_gdiff = cur_gdiff * m_A
        g_diff.append(cur_gdiff)

    g_diff = torch.vstack(g_diff)
```

2.2 Ascend C GDN算子接入

昇腾Triton实现一般作为临时方案，Ascend C调优后才能释放硬件性能，为此选择接入GDN的Ascend C实现。

Ascend C GDN接入的软件依赖如下：CANN 8.5.0的商发和对应PTA包（2.7.1 251113之后）即可，但是需要额外编译flash-linear-attention的run包支持

代码块

```
yum install patch
git clone https://github.com/flashserve/flash-linear-attention-npu.git
cd flash-linear-attention-npu

# 编译大约耗时20mins，其中有个长达8mins的无输出且低CPU占用的编译时段，请耐心等待。
bash build.sh --soc=ascend910_93 --pkg --
ops=chunk_bwd_dv_local,chunk_bwd_dqkwg,chunk_fwd_o,chunk_gated_delta_rule_bwd_d
hu,chunk_gated_delta_rule_fwd_h,prepare_wy_repr_bwd_da,prepare_wy_repr_bwd_full
```

编译结果如下

代码块

```
[2026-04-03 03:02:47] CPack: - package:
/mnt/huawei/tanshaojie/ascendcGDN/flash-linear-attention-npu/build/cann-ops-
transformer-custom_linux-aarch64.run.json generated.
[2026-04-03 03:02:47] CPack: - package:
/mnt/huawei/tanshaojie/ascendcGDN/flash-linear-attention-npu/build/cann-ops-
transformer-custom_linux-aarch64.run generated.
```

安装对应run包

代码块

```
bash /mnt/huawei/tanshaojie/ascendcGDN/flash-linear-attention-npu/build/cann-
ops-transformer-custom_linux-aarch64.run
```

共涉及如下若干 GDN triton实现替换成Ascend C实现

变更模块名	triton耗时倍数（相对等价H200）	Ascend C耗时倍数（相对于等价H200）	接入Ascend C加速比
ChunkGatedDeltaRuleFwdH	3.735	1.725	2.165217391
ChunkFwdO	15.935	2.145	7.428904429
ChunkBwdDvLocal	4.68	2.77	1.689530686
ChunkGatedDeltaRuleBwdDhu	3.71	1.3	2.853846154
ChunkBwdDqkwg	5.615	2.62	2.143129771
prepare_wy_repr_bwd	4.075	3.87	1.052971576

注意

1. 相对等价H200代表着考虑了2die A3和H200比较。
2. prepare_wy_repr_bwd 的一个 triton 算子变成 Ascend C算子时被拆解成两个算子：PrepareWyReprBwdDa+PrepareWyReprBwdFull

看上表可见接入Ascend C实现平均快1倍，但是相对于H200还有2倍加的差距。

由于这些Ascend C算子实现比较匆忙，导致有两个遗留问题（后续版本会修复）：

1. cu_len 需要 int64 输入，不然会功能报错。
2. 输入要设置dh0=None，不然输出会和原本实现不一致，导致精度问题。

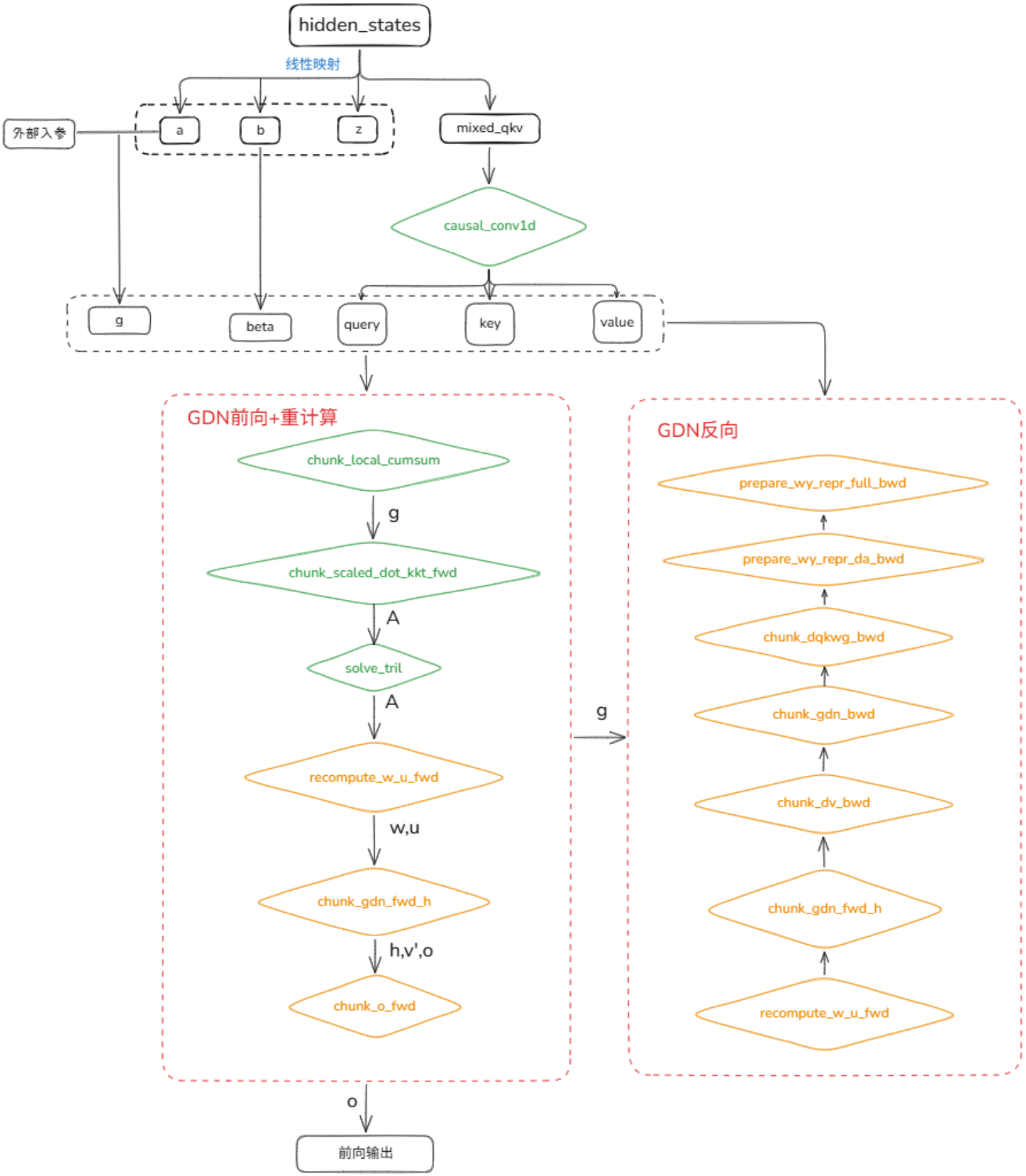
2.3 transpose消除

Transpose作为AiVector算子，对于较大shape的Tensor，天生具有性能劣势。当前配置下，qkv的shape为(B,T,H,D)=(1,64*1024,128,64)，Transpose单算子耗时即相当可观，总耗时在e2e中占比达10.95%

Transpose主要出现在GDN层的Triton算子和AscendC算子交接转换时刻，根本原因在于triton算子来自于mindspeed仓已有实现、惯例接受BTHD格式入参；AscendC算子惯例接受BHTD格式的入参。

具体地，一个典型的GDN结构如下，（以绿色标记triton算子、黄色标记AscendC算子），经过Triton格式的causal_conv1d输出q、k、v送入MTP结构进行计算；MTP前向为Triton算子和AscendC算子混合，反向全为AscendC算子，且MTP前反向复用同样的qkv、反向接受前向输出的g。

总体来看，前向中的Triton算子较为连续、AscendC算子性能更优，因此修改causal_conv1d的出参排布和chunk_scaled_dot_kkt的入参、出参排布是最优的Transpose消除方案，就能够规避入参q、k、v及其梯度转置、中间变量w、u及其梯度转置和部分冗余的g、beta、A转置。



我们对上述triton算子的kernel做了如下修改，具体代码参见上海大模型特战队仓库提交

1. 修改causal_conv1d算子

如上图所示，xtuner框架中对hidden_states做线性映射得到shape为(B,T,H*D)的mixed_qkv，其中H为query、key、value的注意力头数之和，而query、key和value共用的注意力头大小D。

原始版本的causal_conv1d算子实现中，输出qkv均为(B,T, H_{sub} *D)格式，并在算子外部通过reshape对最后一维拆分、对q和k的注意力头数进行repeat_interleave等。

我们最终采用的方案是：令triton算子接受注意力头数H作为入参，并在kernel内部将输出参数内存排布改为(B, H_{sub} ,T,D)格式。虽会导致单算子性能轻微劣化，但考虑transpose规避，仍有性能净收益。

比较tricky的一个点是，我们在重计算阶段仍使用原始causal_conv1d算子前向接口，以尽可能减少内存排布修改带来的算子性能损失。

2. 修改chunk_scaled_dot_kkt算子

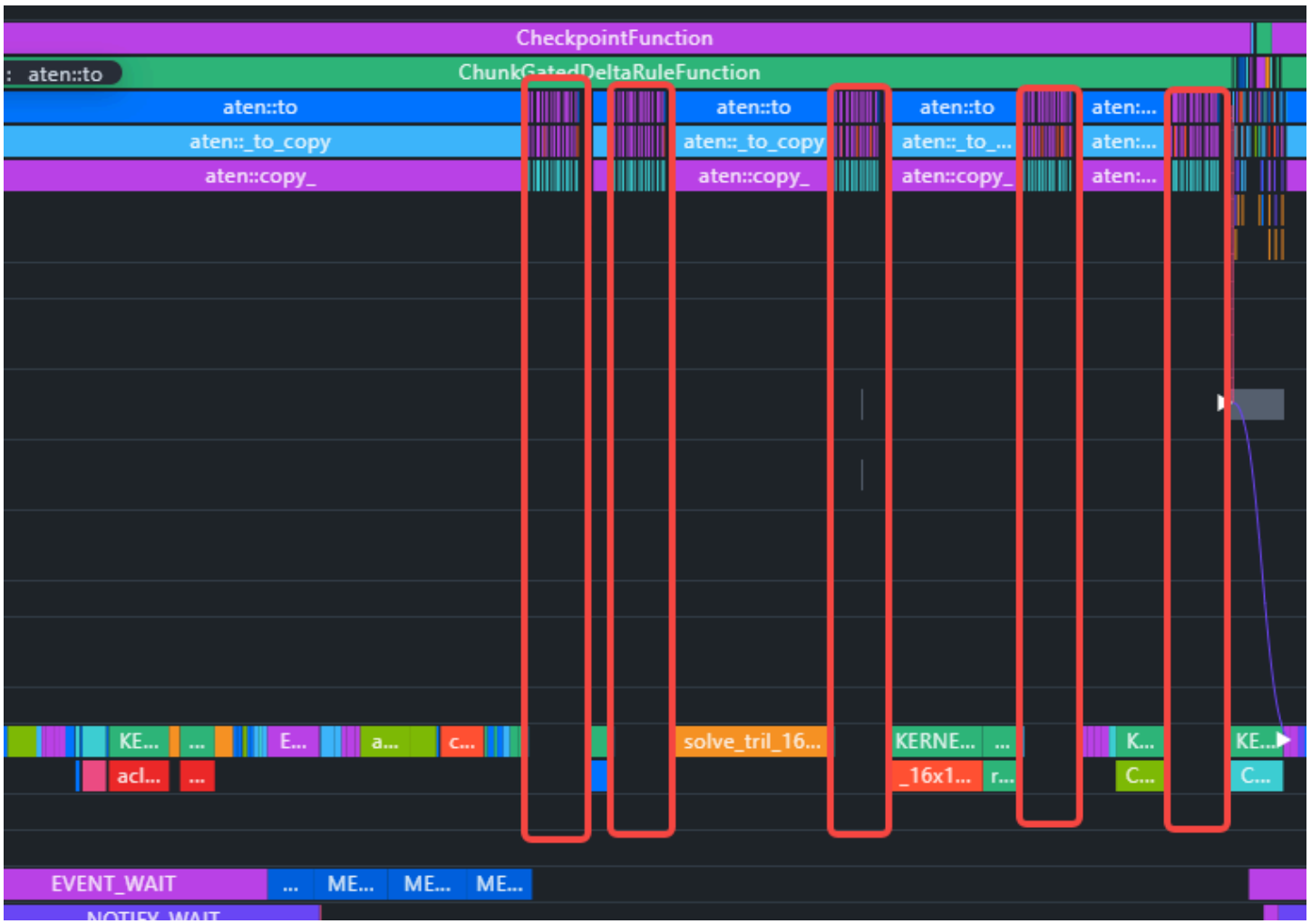
只需修改kernel内部A的内存分配即可，涉及到基址和stride。该算子不涉及反向、无明显性能劣化。

最终，transpose消除的性能优化非常好验证，采集profiling通过kernel_details观察（1）transpose算子本身个数和总耗时是否减少（2）经修改后的triton算子性能本身是否劣化

修改后，单步transpose算子总个数从1689降至1069，总耗时由639.3ms下降至145ms；causal_conv1d算子单步劣化60ms。即，该操作单步净收益400+ms，考虑到e2e耗时本身在15-20s左右，性能收益2.5%

2.4 GDN内chunk indices优化

如下图，GDN接入后通过profiling可以看到GDN算子内部有大量在CPU侧计算的小算子，造成NPU侧的空泡，定位到该空泡由prepare_chunk_indices函数中的tolist引入，在昇腾上tolist会自动触发同步。prepare_chunk_indices函数在GDN的前反向中会被多次调用，该函数会基于cu_seqlens计算chunk_indices，而cu_seqlens在每一个step的数据处理阶段就已经确定，因此chunk_indices完全可以在每次step中只计算一次。我们将GDN内部的prepare_chunk_indices函数提取到模型前向之前，从而消除同步造成的npu空泡。



2.5 GDN 支持chunk size 128

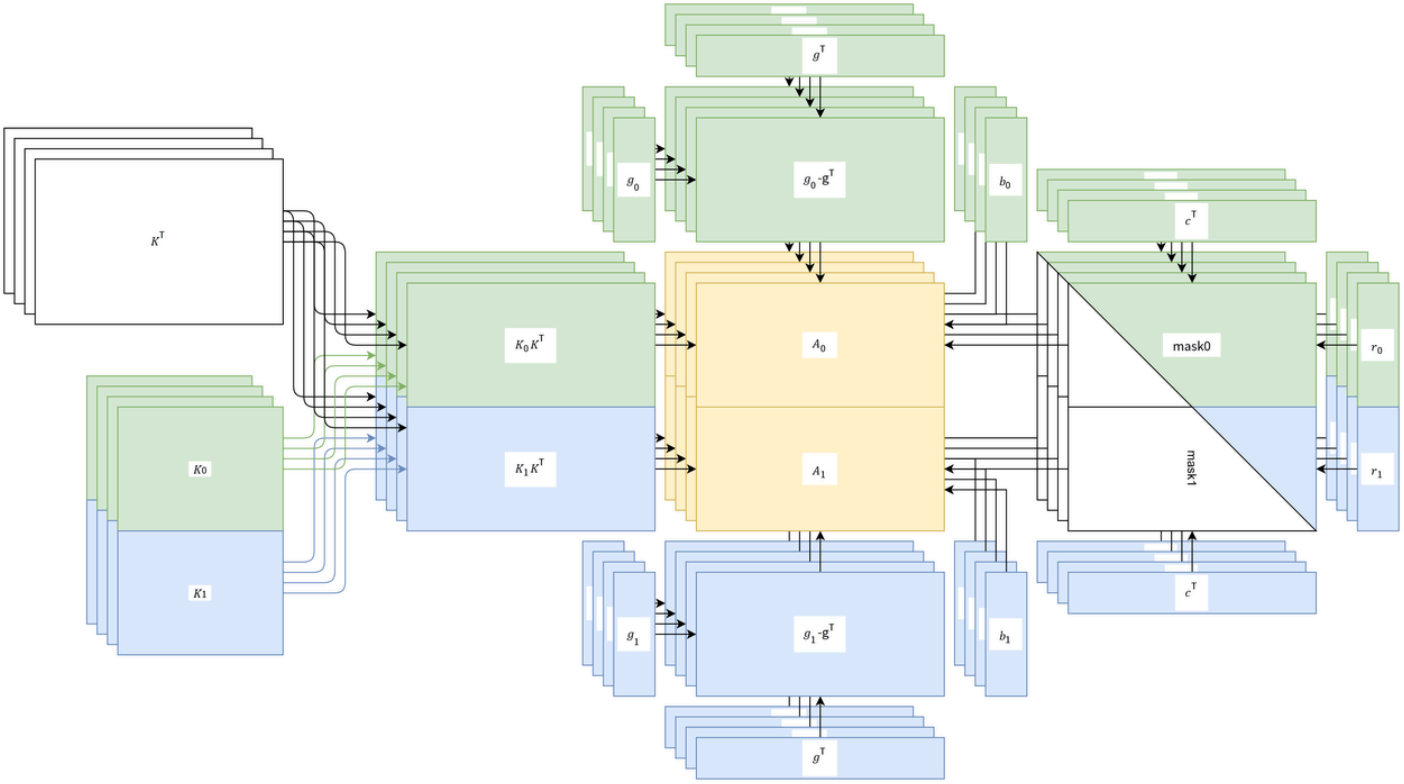
GDN算子的设计参考了GPU实现，在GPU上chunk size默认值是64；但是在NPU上核心少，但是单核性能强。NPU理论上chunk size 增大会更亲和硬件效率。由于Ascend C的GDN实现最多支持chunk size 128，而其余几个triton （recompute、kkt、solve tril）甚至不支持chunk size 128，为此现场开发算子来使其支持chunk size 128，以支持GDN模块的chunk size 128功能。

1. recompute算子 ascend c

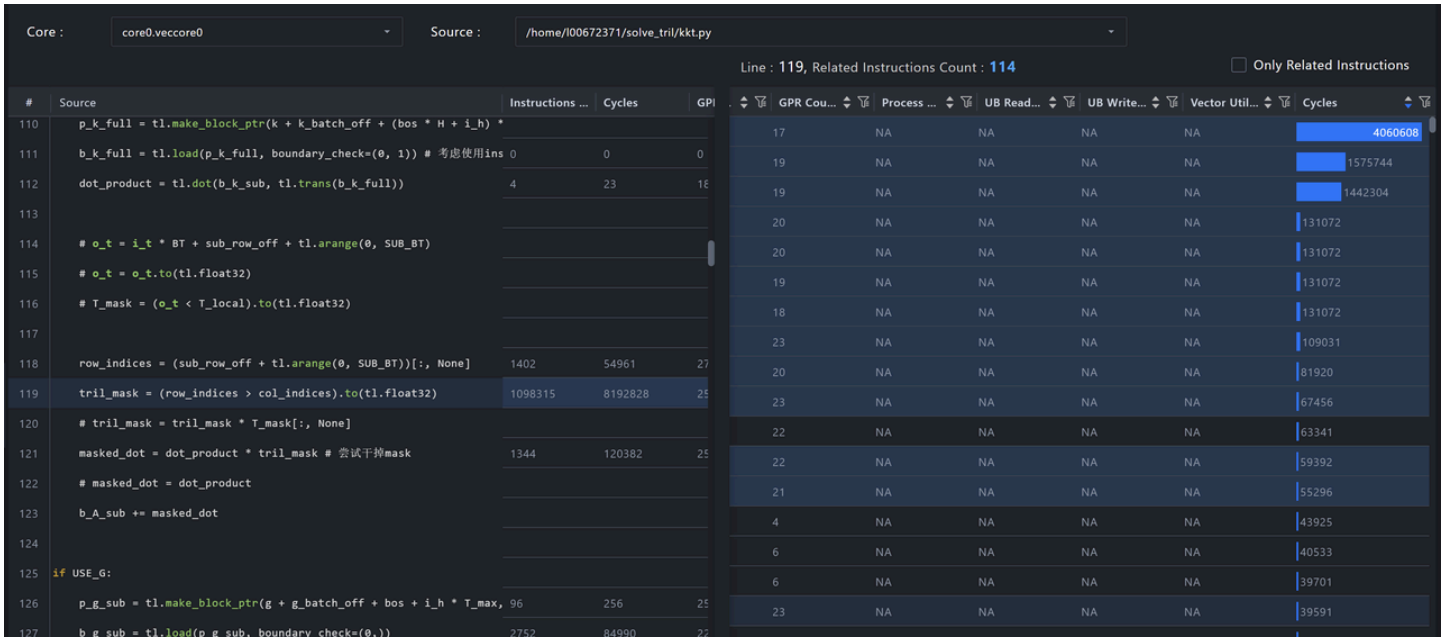
recompute_wu_fwd算子当前triton实现并不支持chunk size 128，分析发现这个算子当前是triton实现，而且逻辑不复杂，当前各个流水线利用率不佳。直接选择手写AscendC实现，兼顾chunk size 128和性能优化。实测整网可以优化0.1s已归档于https://github.com/flashserve/flash-linear-attention-npu仓库

2. Kkt 支持 chunk size 128 并优化

KKT计算分为两个步骤，第一步是核心的K*KT，得到一系列的chunksize*chunksize的tensor，第二个步骤是对这个tensor进行一些element wise的计算以及causal mask处理。前者是在cube上计算，后者则是在vector上计算。



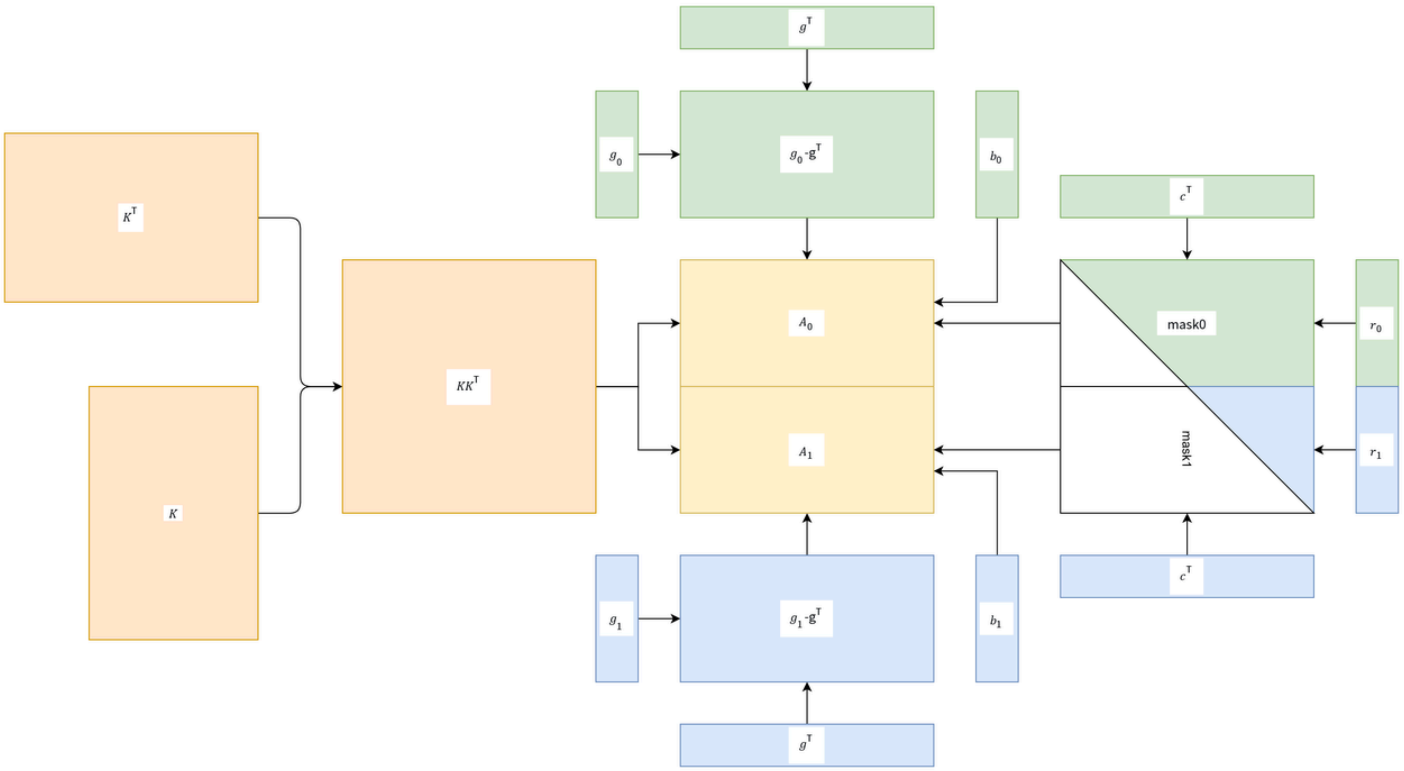
图示将计算过程可视化了，并且展示了如何将计算拆分成两个子任务，一次处理64*128的大小。



开发完成128算子后，发现性能劣化数倍，定位发现mask构造过程中，存在int64compare计算操作，昇腾硬件不支持，退化为scalar计算，耗时显著增加。通过修改计算dtype绕过此问题。性能从原先的完全不可用，优化至4.0ms。然而，和chunksize64的2.3ms差距仍然较大。

chunksize128的另一个挑战是片上内存的利用。chunksize128下，算子需要一次处理128*128个元素，容易导致UB overflow。而我们在两个版本上发现了不同的现象。在8.5.0.B030的CANN下，可以一次处理64*128个元素，而8.5.2的CANN只能处理32*128个元素，这导致硬件利用率急剧下降，性能劣化1倍。与编译器同事共同定位，本质确定为clamp计算中，maximum、minimum对nan处理的差异。triton原生对此操作的定义为默认忽略nan数据，这在NPU上会导致更大的片上内存开销。原生社区提供了propagate_nan参数，用于定义reduce过程中对nan数值的处理。在8.5.0.B030下，编译器错误的忽略了这个参数，不会执行propagate_nan=0的逻辑，后面的版本修复了，因此造成了观察到的UB占用的区别。通过配置propagate_nan变量为1，我们在8.5.2版本也拿到了UB占用的收益，且精度符合标准。性能实现4.0ms到2.6ms的优化。

到此，执行一个完整的128*128block，只需要分两次计算，还能再减少吗？答案是肯定的。在当前计算流程中，cube计算和vector计算存在一些gap。cube核心其实一次就能完成128*128的完整运算，但是vector受限于UB大小，后续对这些数据的求下三角mask、exp、偏置等，无法一次执行，必须分两次，这是当前的瓶颈。因此，我们可以利用A2芯片的cube vector 1：2的硬件结构，利用满两个vector核心，每个核心都只用一次循环即可完成任务。



如图，为首的矩阵乘在cube一次完成，计算结果拆分为两部分由两个vector核心（绿色和蓝色）分别执行后续计算。

通过升级到3.4.0版本的triton ascend，就可以利用sub vector id的接口，自定义vector核心的计算任务。利用这个特点我们实现了性能从2.6ms到2.0ms的优化。由于3.4.0版本未正式发布，客户暂时无法使用。

3. Solve tril支持 chunk size 128 并优化

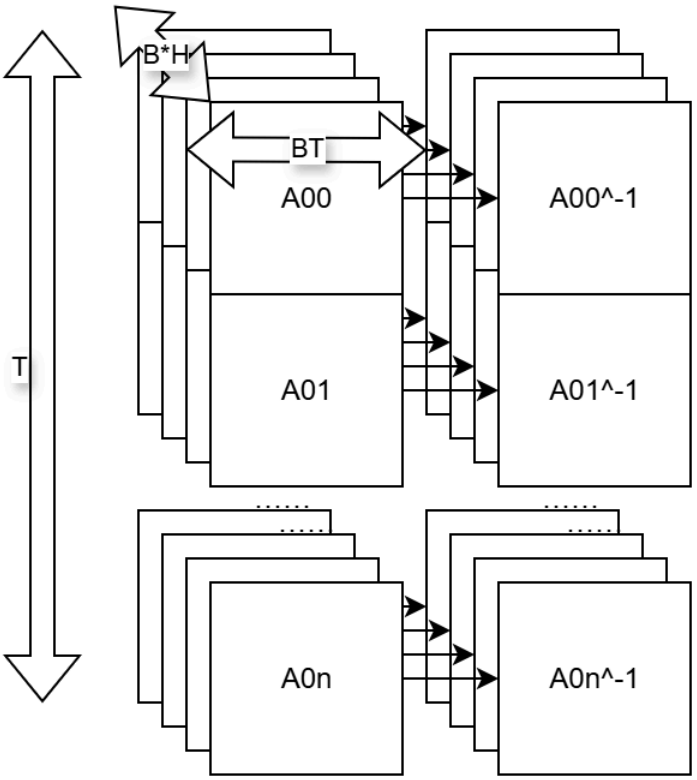
和2.6.1相同的方式，设计中间变量，并改动32*32to64*64算子使其兼容更大chunksize。另外再类似的再开发64*64to128*128即可。性能上，由于增加了64to128的kernel，耗时增加到了15ms。

2.6 Ascend C/Triton算子极致优化

GDN内若干算子（solve tril，kkt，casual_conv1d）性能差，为此针对这些个例进行了极致性能调优。

1. Solve tril ，

Solve tril算子，功能为求一串下三角矩阵的逆，这些下三角块的大小为chunksize x chunksize。输入的shape为B S H BT，其中BT为chunksize，为kkt算子输出。



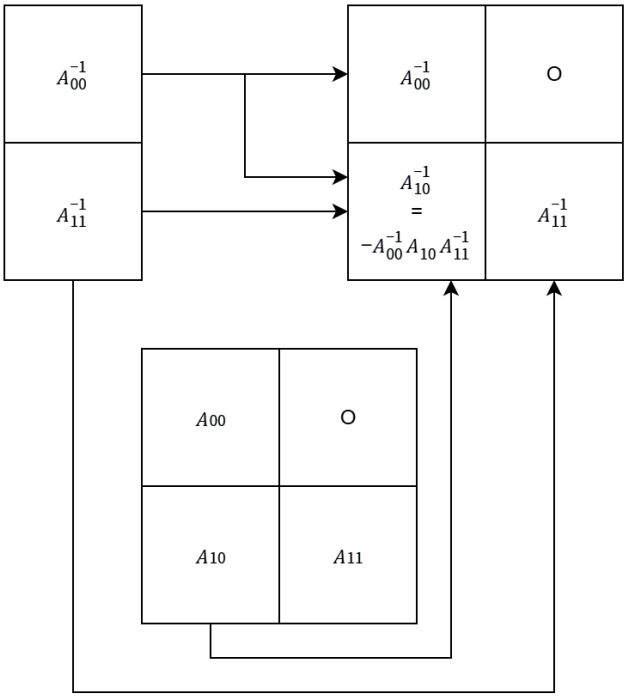
其中求逆有比较多的算法，代表性的有三个算法，前向消元法、Cayley–Hamilton 定理、Bunch–Hopcroft 分块三角求逆公式。

参考文章如下

https://github.com/huawei-csl/gdn-tri-inverse/blob/main/markdown/fast_inver···

gdn-tri-inverse/markdown/fast_inverse_blog/fast_inverse_zh.md at main · huawei-csl/gdn-tri-inverse···

当前mindspeed代码中采用的是使用前向消元法直算64*64的逆，无法利用cube算力。从利用cube算力的角度出发，参考SGlang，实现类似FLA源仓的二阶段计算，第二阶段使用分块三角求逆，利用cube单元。计算逻辑如下，需要先有对角块的逆的中间结果，再迭代计算，每一次迭代都是一个矩阵运算，就可以利用cube完成。



SGlang的实现参考了FLA源仓的实现，由16*16纯V + 16*16 to 64*64两部分组成。增加了中间变量的写入和读取，计算上使用cube加速计算，收益不确定，依赖实验验证。构造golden标准和单算子用例，接入如下实现

https://github.com/huawei-csl/gdn-tri-inverse/blob/main/src/gdn_tri_inverse/t···

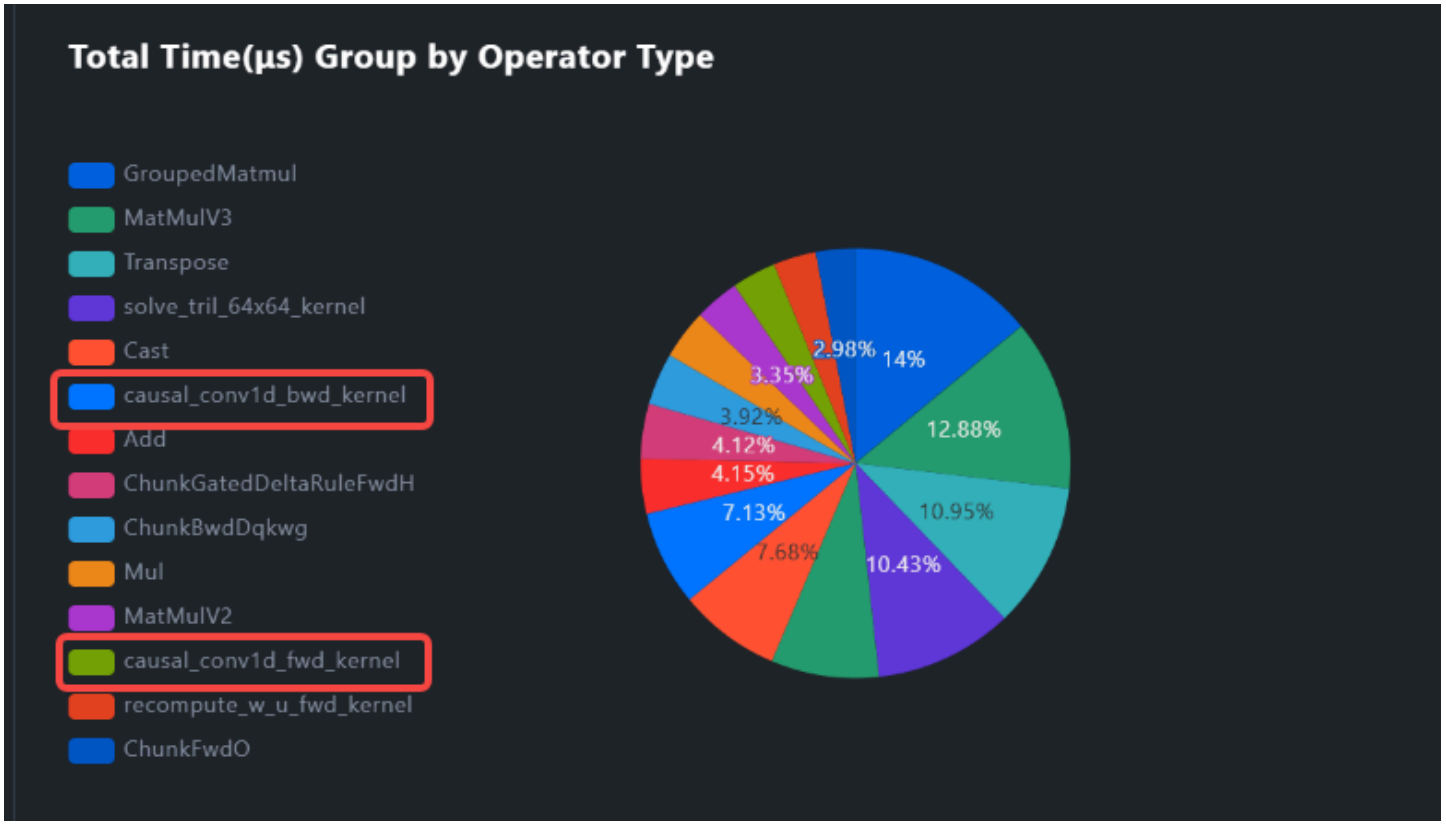
gdn-tri-inverse/src/gdn_tri_inverse/triton_chunk_gdn/fla/solve_tril.py at main · huawei-csl/gdn-tri-···

精度pass，且**性能从25ms优化至14ms**。

16*16 to 64*64的CV算子计算逻辑较为复杂，在片上一次性完成两次Bunch–Hopcroft 分块三角求逆公式计算，考虑拆分为两个独立的kernel分别进行两次Bunch–Hopcroft 分块三角求逆公式计算，简化计算逻辑，让编译器更容易做优化。具体来说就是把16*16 to 64*64的CV算子拆分为16*16 to 32*32+32*32 to 64*64两个CV算子。这个过程中主要要做的是，16*16 to 32*32算子需要兼容大chunksize下的分步运算，并设计中间变量的内存排布，有序执行计算，细节不再赘述。精度pass，且**性能从14ms优化至11ms**。到此累积优化了14ms。

2. Casual conv1d反向

在使用Triton版本的casual conv1d后，采集profiling发现该算子占比如下：



其前反向共占计算时间的3.33%+7.68%=10.01%，另外下表展示了causal_conv1d的前反向算子在NPU与GPU上的性能差距，由于该算子是vector算子，而A3上的vector单元性能是H200的1/6，因此causal_conv1d_forward算子的优化空间不大，但是causal_conv1d_backward具有较大的可优化空间。

算子	NPU（A3）	GPU（H200）	差距
causal_conv1d_backward	33ms	1.5ms	22x
causal_conv1d_forward	5.8ms	1.37ms	4.23x

调整BT和BD的大小，BT：64->4，BD：64->512。这是一个典型的 tiling 参数调优 ，通过增大特征维度的 block 大小（BD）来提高内存带宽利用率和计算强度，同时减小BT来保持合理的并行度，从而在 NPU 上获得更好的性能。

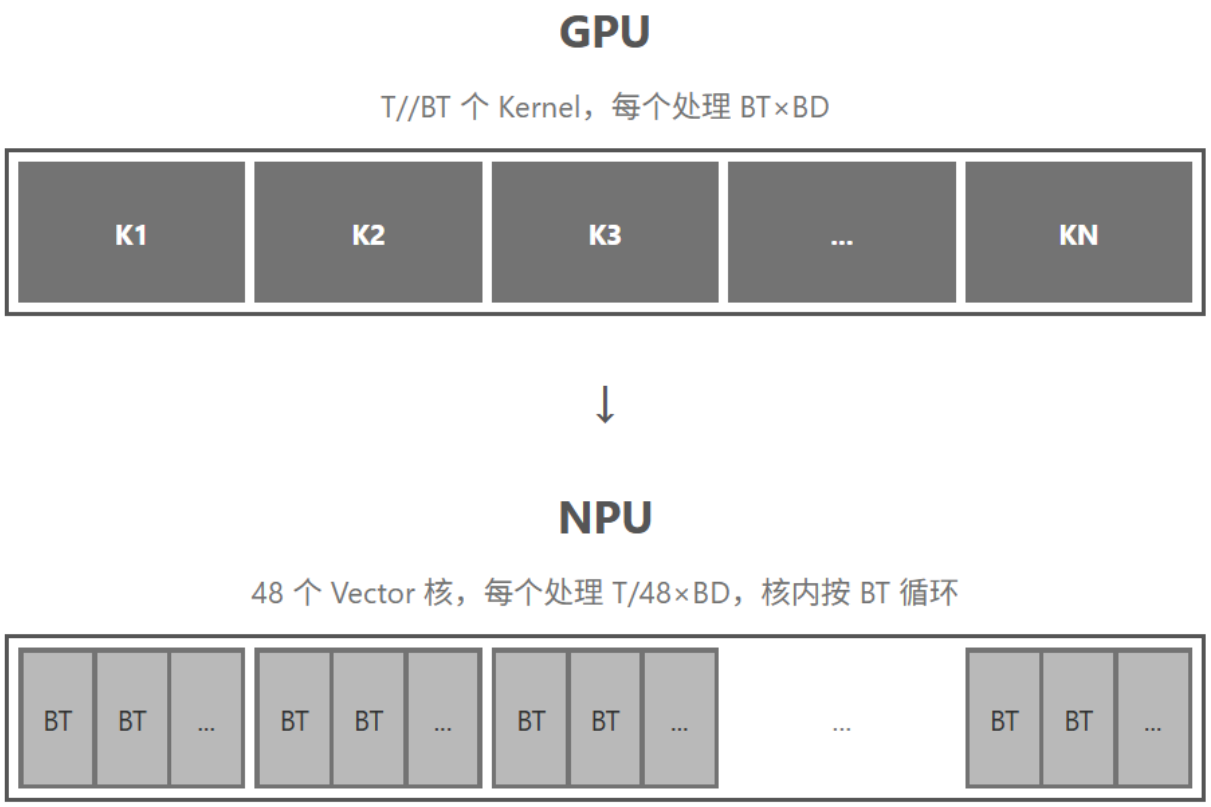
```
475     BT = min(4, triton.next_power_of_2(triton.cdiv(max(16, B * T), NUM_CORES)))
476     # BT = min(64, triton.next_power_of_2(triton.cdiv(max(16, B * T), NUM_CORES)))
477
478     BD = 512
479     # BD = 64
```

调整后的causal_conv1d_backward耗时由33ms下降为18ms。由于 A3 单 Die 的vector性能仅为 H200 的 1/6，在实际对比中通常采用 A3 双 Die 配置对标单张 GPU。若以 18ms 为基准耗时，经架构性能折算（÷6）及双芯并行加速（÷2）后，理论耗时约为 1.5ms。这表明 causal_conv1d_backward在 A3 上性能已基本与 GPU 持平。

3. RMSNormGated triton

Xtuner使用fla库中的RmsNormGated融合算子，虽然该算子可以直接在NPU上运行，需要进行针对NPU的性能优化。

前向逻辑修改：通常GPU开发及优化的Triton算子，数据分片小，Kernel调用次数多，在迁移到NPU上运行时需要减少Kernel调用次数，增加单kernel内处理的数据量，并在Kernel内做Tiling，实现性能最优。而fla库中的前向kernel是一个纯vector算子，Kernel调用次数为T//BT，然后在在每一个Kernel内计算BT*BD数量的数据，通常BT为16,32,64。为了充分利用NPU设备的性能，我们修改前向逻辑，将Kernel调用次数减少为NPU的vector物理核数（A3上是48），相当于每个核处理数据T//48*BD，然后在核内进行循环，每个小循环内处理数据BT*BD。



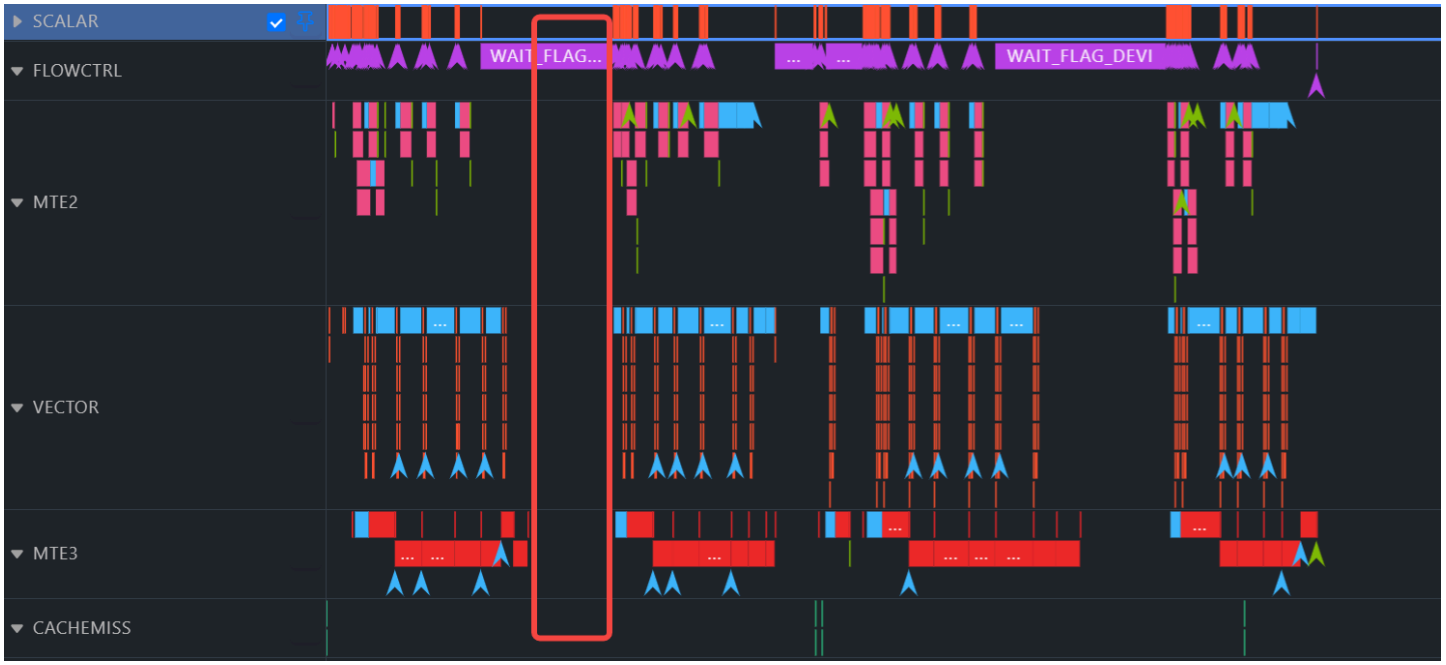
T：序列总长度 | **BT**：每个 block 处理的 token 数（通常为 16/32/64） | **BD**：head 维度

K1~KN：GPU 上每次 launch 的 kernel | **Core**：NPU Vector 物理核（A3 上共 48 个） | **BT（核内）**：NPU 每个核内循环处理的数据粒度

参数调优：由于算子可用的UB空间是有限的，因此为了每一次尽可能处理更多的数据，减少循环次数，我们可以修改BT的大小，使其每次循环处理的数据量尽可能的逼近UB空间大小。当前经过验证前向kernel的BT设置为64，反向kernel的BT设置为24时，在不触发UB overflow的情况下可以取得最优性能。Triton版本的RmsNormGated算子相比npu_rms_norm的ascendC算子+silu小算子，每层有2ms的性能收益。

4. Ascend C 算子同步优化

使用mindstudio工具查看AscendC算子流水图，容易发现其中Vector部分有很长一段空等待时间。



深入阅读AscendC GDN算子相关代码，不难发现其中有4个算子均采取相同的CV核间同步方式进行实现，即单个任务块之间均包含一次CrossCoreWaitFlag和CrossCoreSetFlag，这大大影响到了算子执行的流水，遂修改内部CV同步方式，使用Catlass中的Arch::CrossCoreWaitFlagWithReverse实现，达成每16个任务块一次反向流水，大大降低了冗余同步数量，实测4个算子整网可以优化0.1s。已归档于https://github.com/flashserve/flash-linear-attention-npu仓库

3. 框架流程优化

【框架流程，可优化点一览】

下图是初步开箱采集的profiling，配置为单机16die，模型减到8层，使用SP2，序列长度为64k，通过如下profiling，可以初步分析到以下几个可优化点：

1.vit部分当前使用的是fake input，是一个[4, 1536]的短序列，由于序列短导致计算少，从而导致通讯无法掩盖；

2.profiling中有一段异常通信，该段通信怀疑是由下发异常导致快慢卡问题；

3.SP会带来额外的通信开销，而GPU中不开启SP；

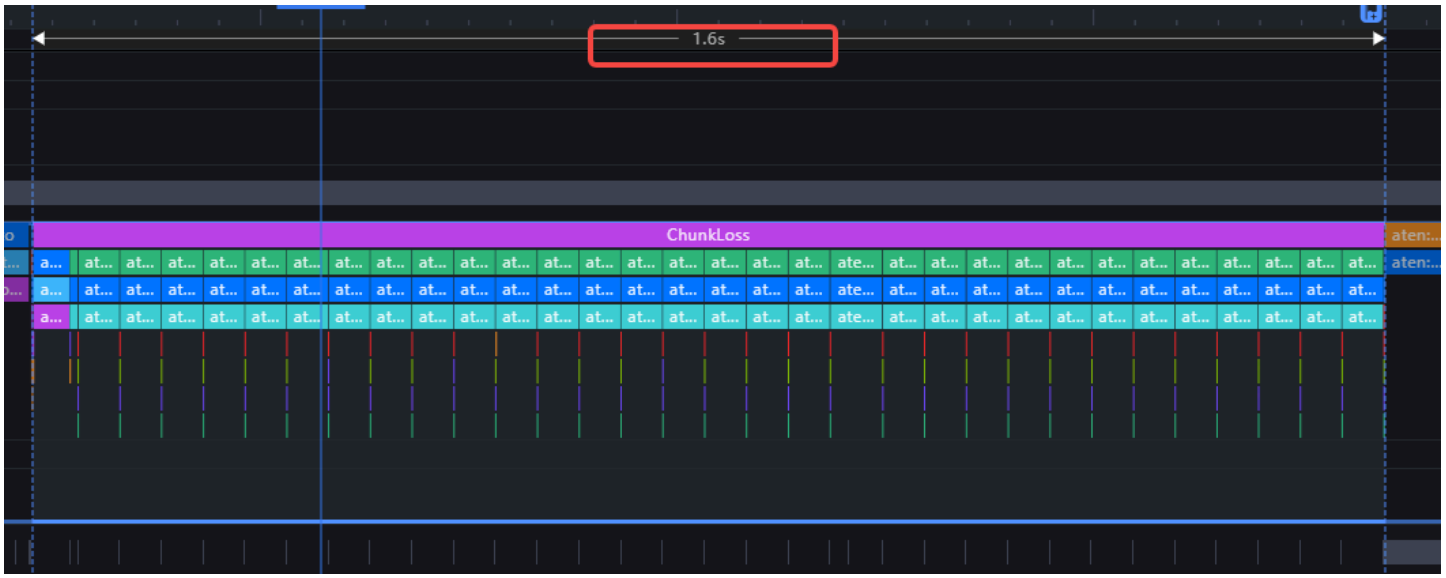
4.chunkloss中小算子过多，可以通过替换融合算子来加速；

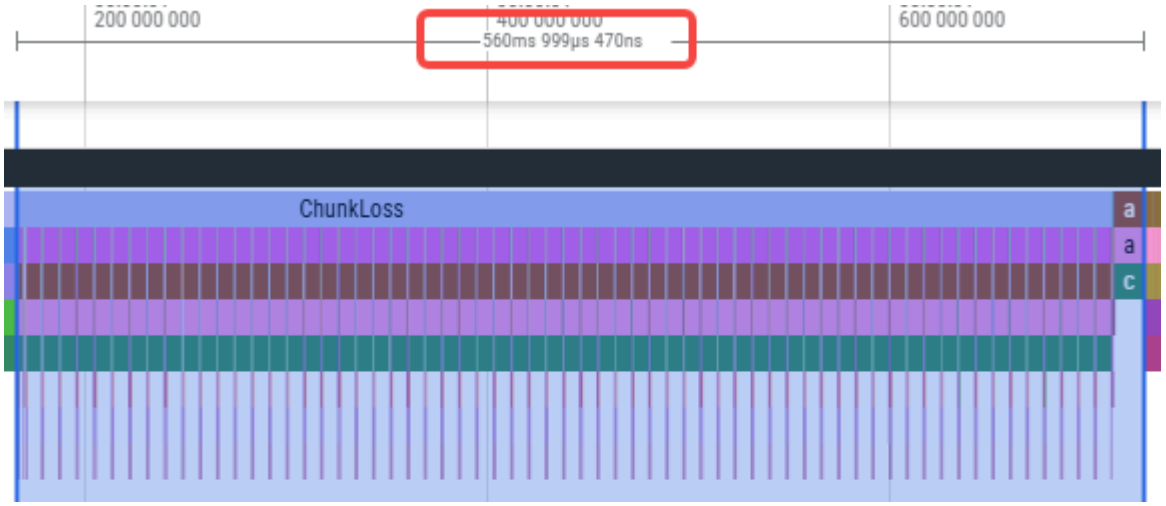
3.1 基础优化

在基础优化方面，首先通过绑核解决部分NPU设备任务下发不均衡引发的快慢卡问题；其次，针对NPU为跑通单机64k长序列而开启SP2所带来的额外通信开销（对比显存达144G无需开启SP的H200 GPU），后续使用双机A3架构后可彻底去除该开销；算子层面，采用 `npu_rms_norm` 融合算子替换仓内原有小算子，且因该融合算子具备输入自动升精特性，可省去显式的float cast操作；此外，将CANN版本由8.5.0升级至8.5.2后，依托毕昇编译器的底层优化进一步提升了算子执行效率，整体带来约0.1s的性能增益。

3.2 chunk loss优化

在计算chunkloss时发现NPU耗时为1.6s，GPU耗时为0.55s，有3x的性能差距。另外chunkloss由大量的小算子组成，因此考虑在NPU上基于Triton开发一版融合算子来加速。





首先采集整网profiling后，在ASCEND_PROFILER_OUTPUT中找到kernel_details.csv，然后参照代码在文件中找到当前算子的耗时。

58	aclnnMatmul_MatMul(	MatMulV2	dynamic	AI_CORE	3573.4
58	aclnnCast_CastAiCo	Cast	dynamic	AI_VECTOR	1143.02
58	aclnnNeScalar_NotEc	NotEqual	dynamic	AI_VECTOR	3.04
58	aclnnReduceSum_Cast	Cast	dynamic	AI_VECTOR	1.72
58	aclnnReduceSum_Redu	ReduceSum	dynamic	MIX_AIV	5.7
58	aclnnReduceSum_Cast	Cast	dynamic	AI_VECTOR	1.58
58	aclnnEqScalar_Equal	Equal	dynamic	AI_VECTOR	2.26
58	aclnnLogSoftmax_Log	LogSoftmaxV2	dynamic	AI_VECTOR	2320.48
58	aclnnInplaceOne_One	OnesLike	dynamic	AI_VECTOR	8.5
58	aclnnNLLLoss_NLLLos	NLLLoss	dynamic	AI_VECTOR	18.56
58	aclnnMul_MulAiCore_	Mul	dynamic	AI_VECTOR	1.96
58	aclnnReduceSum_Redu	ReduceSum	dynamic	MIX_AIV	5.62

对算子进行梳理，其中MatMul是cube算子，且其性能瓶颈在于mte搬运。一方面MatMul是mte bound，所以写成融合算子后是耗时随输入维度大小线性增长，上线预计取决于MatMul依赖的带宽，所以不需要融合；另一方面triton的cv算子流水特性还有问题，因此暂不考虑cv融合。因此选择对上图黄色算子做vv融合来减耗时。

aic_mte1_ratio	aic_mte2_t	aic_mte2_ratio	aic_fixpi	aic_f
0.655	3468.587	0.973	711.42	
0	0	0	0	

然后找到这一堆vector算子对应的代码，可以较为明显的找到是下图红框中所示的三行代码，这三行代码即我们后续需要依据的小算子基线。在已经获得小算子的基础下，我们需要写相应的triton代码，并最终保证triton代码的输出精度与小算子对齐。

```
    logits = F.linear(hidden_states, head_weight, head_bias)
    logits = logits.float()  # (bs, seq_len, vocab_size)

    shifted_labels = loss_kwargs.shifted_labels  # (bs, seq_len)
    loss_weight = loss_kwargs.loss_weight  # (bs, seq_len)
    assert loss_weight is not None, "loss_weight can not be None"

    logits = logits.reshape(-1, logits.size(-1))  # (bs * seq_len, vocab_size)
    shifted_labels = shifted_labels.flatten()
    loss_weight = loss_weight.flatten()

    rank_grad_tokens = (shifted_labels != self.loss_cfg.ignore_idx).sum()
    if rank_grad_tokens == 0:
        loss = logits.sum() * 0
    else:
        loss = F.cross_entropy(logits, shifted_labels, reduction="none", ignore_index=self.loss_cfg.ignore_idx)
        # Step 2.b in the loss calculation: sum the loss over all tokens
        loss = (loss * loss_weight).sum()

    return loss, (logits, {})
```

在开发完triton代码后需要验证其精度与性能。通常需要写一个单算子脚本，在该脚本中保障小算子和triton融合算子的输入相同，比较输出是否满足双千分之一的标准（相对误差大于千分之一的元素数量的比例小于千分之一）来检验triton版本的融合算子精度是否存在问题。

性能方面则是可以采集小算子和融合算子的profiling，查看kernel_details.csv来分析算子内部的具体耗时。

```
with torch_npu.profiler.profile(
    activities=[torch_npu.profiler.ProfilerActivity.CPU, torch_npu.profiler.ProfilerActivity.NPU],
    schedule=torch_npu.profiler.schedule(wait=0, warmup=0, active=1, repeat=1, skip_first=0),
    experimental_config = torch_npu.profiler._ExperimentalConfig(profiler_level=torch_npu.profiler.ProfilerLevel.Level1),
    record_shapes=True,#采集torch op的input shape和input type的开关
    on_trace_ready=torch_npu.profiler.tensorboard_trace_handler("./perf_part"),
) as prof:
    # Triton 融合算子
    triton_loss = fused_cross_entropy_loss(logits, loss_weight, labels)
    grad_output = torch.ones_like(triton_loss, dtype=torch.float32)
    grad_logits, grad_loss_weight = fused_cross_entropy_loss_back(logits, loss_weight, labels, ignore_index, grad_output)
    # Pytorch 算子
    pt_loss = ChunkLoss.apply(logits, loss_weight, labels)
    pt_loss.backward()
    prof.step()
torch.testing.assert_close(pt_loss, triton_loss, atol=1e-3, rtol=1e-3)
torch.testing.assert_close(grad_logits, logits.grad, atol=1e-3, rtol=1e-3)
torch.testing.assert_close(grad_loss_weight, loss_weight.grad, atol=1e-3, rtol=1e-3)
```

这里有两个优化案例可供大家参考学习：

1.输出为nan

开始时other=float('-inf')，但是输出为nan，改为0.0后精度正常。

```
label_chunk_logits = tl.load(
    logits_ptr + i * C + label_chunk_start + label_chunk_offsets,
    mask=label_chunk_mask,
    other=float('0.0')
)

# 提取目标 logit
target_logit = tl.sum(
    label_chunk_logits * tl.where(label_chunk_offsets == label_offset, 1.0, 0.0),
    0
)
```

-inf时输出为nan

1. `tl.where(label_chunk_offsets == label_offset, 1.0, 0.0)` 产生：

代码块

```
[0.0, 0.0, ..., 1.0, ..., 0.0, 0.0]
# 位置100是1.0，其他位置都是0.0
```

2. 如果 `other=float('-inf')` ， `label_chunk_logits` 中无效位置是 `-inf`
3. 乘法运算：

代码块

```
-inf * 0.0 = nan   ← 问题所在!
-inf * 1.0 = -inf  (如果 label 恰好在无效位置)
valid_value * 0.0 = 0.0
valid_value * 1.0 = valid_value
```

4. `tl.sum([..., nan, ...], 0) = nan`

2.int64导致性能劣化

使用triton版本的chunkLoss融合算子后整网性能劣化3s，分析kernel_details.csv发现反向算子的性能严重劣化。

Type	OP State	Accelerat	Start Tim	Duration(us)
fused_ce_loss_kernel_chunked	static	AI_VECTOR	177565747	3952.94
ReduceSum	dynamic	MIX_AIV	177565747	6.24
Fill	dynamic	AI_VECTOR	177565747	1.4
ZerosLike	dynamic	AI_VECTOR	177565747	1272.34
ZerosLike	dynamic	AI_VECTOR	177565747	1.78
chunk_loss_bw_kernel	static	AI_VECTOR	177565747	118403.96

进一步分析算子的耗时占比，发现aiv_scalar_ratio的数值不正常，vector算子不应该有如此高的scale占比。出现这种原因很可能是因为传入的数据类型中有int64，而vector指令一般不支持int64类型，如果算子中出现int64类型会退化到scalar操作，从而导致性能劣化。

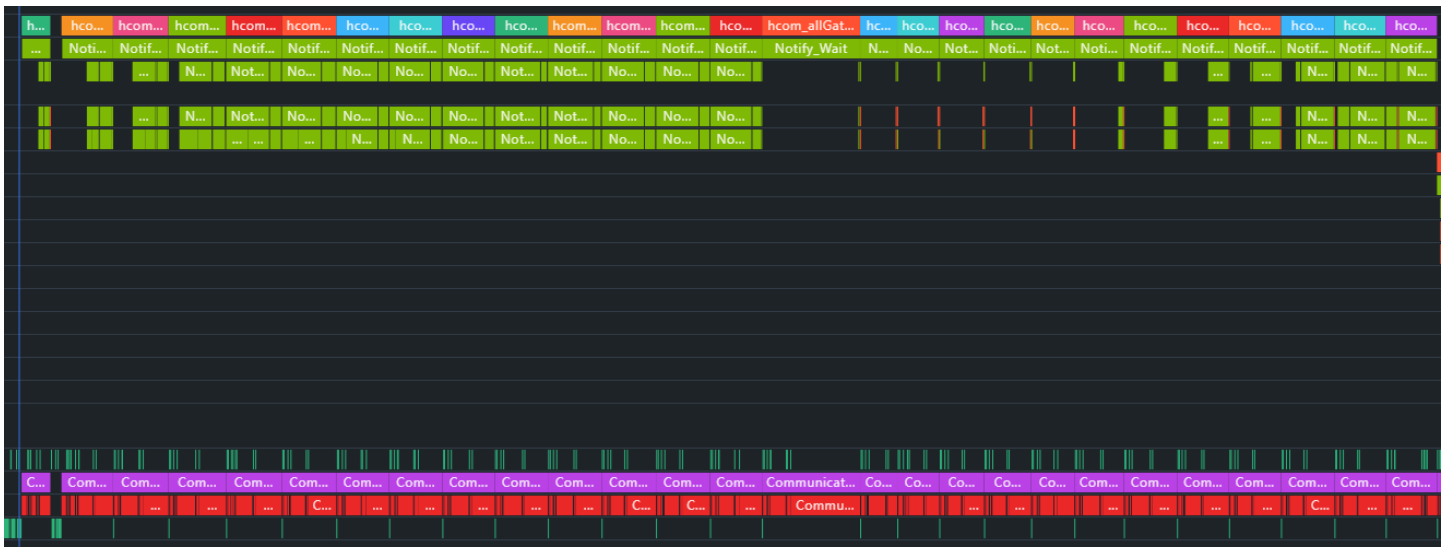
aiv_scalar_ratio
0.579
0.288
0.417
0.015
0.183
0.937

分析代码发现shifted_labels的数据类型为int64，将其转换为int32后获得预期性能收益。

```
loss = fused_cross_entropy_loss(logits, loss_weight, shifted_labels.to(torch.int32), ignore_index=self.l
```

3.3 无ViT对比

由于当前调优过程中所使用的数据集不包含图像或视频数据，Xtuner 框架会构造形状为 `[4, 1536]` 的占位张量作为ViT模块的fake输入。该输入序列较短，导致前向计算的计算负载较低，难以有效掩盖分布式训练中的通信开销，从而引入大量未掩盖通信。然而在实际训练场景中，ViT接收真实的图像或视频输入，经视觉特征提取后将转换为长序列Token。显著增加的计算量能够实现计算与通信的高效重叠，从而有效掩盖通信。



3.4 剩余优化空间分析

最终优化后的版本，free和通信基本消除了，98%都是计算。而计算部分其中 50% 是GMM，MM，45% 是GDN。相对竞品，GMM，MM 性能基本持平H200；但是GDN平均相对于 H200慢一倍，这是导致性能难以达到1.0x的根因。

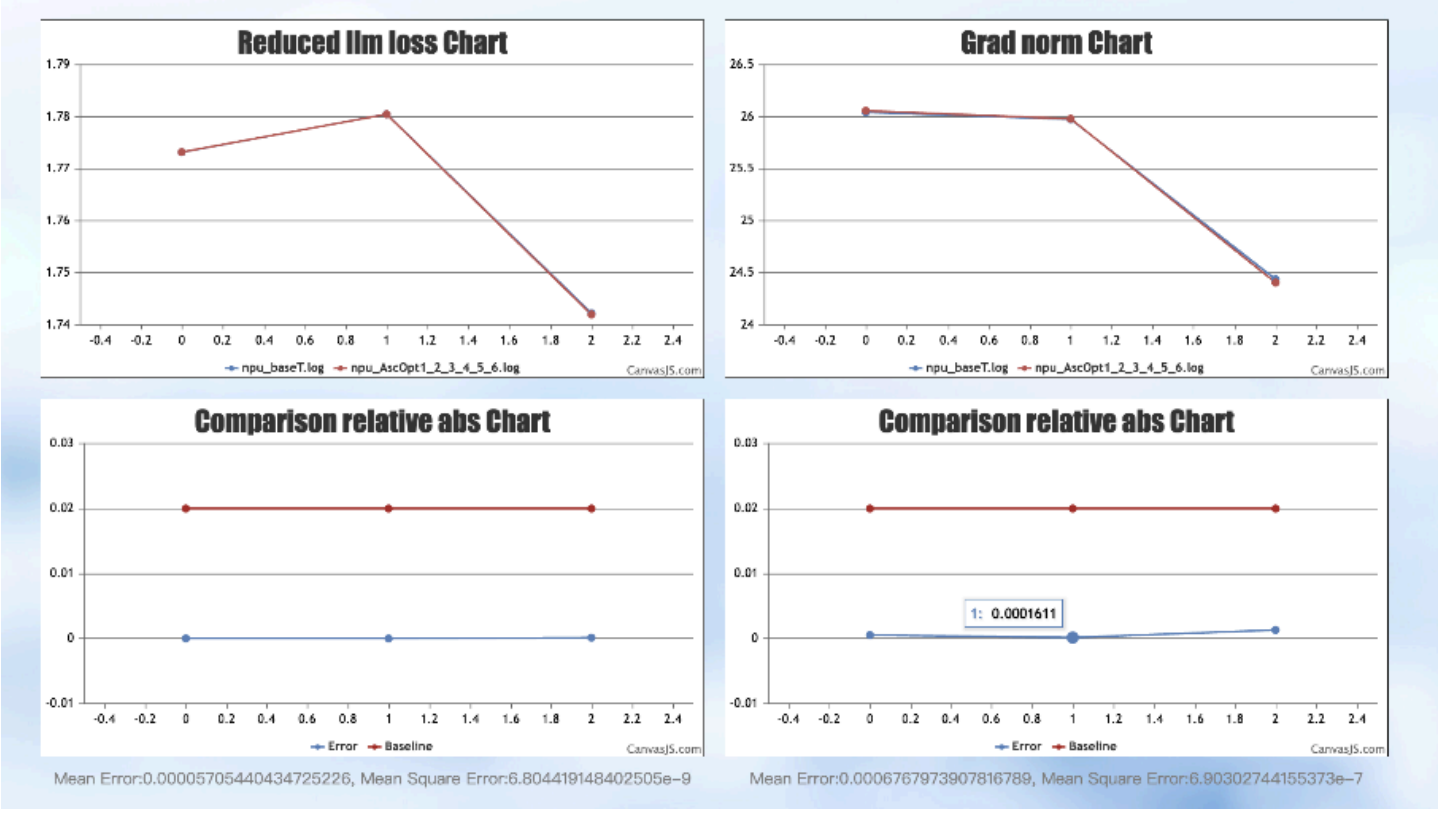
和算子团队沟通后，当前GDN实现基于Catlass的BlockMmad接口浅融合方案实现实现，为快速实现功能版本。后续主要优化思路为使用更底层的tile级接口进行cube部分实现，进一步优化流水及减少搬运量，理论上GDN系列算子应当达成VectorBound。这一部分预计在5.30实现初版

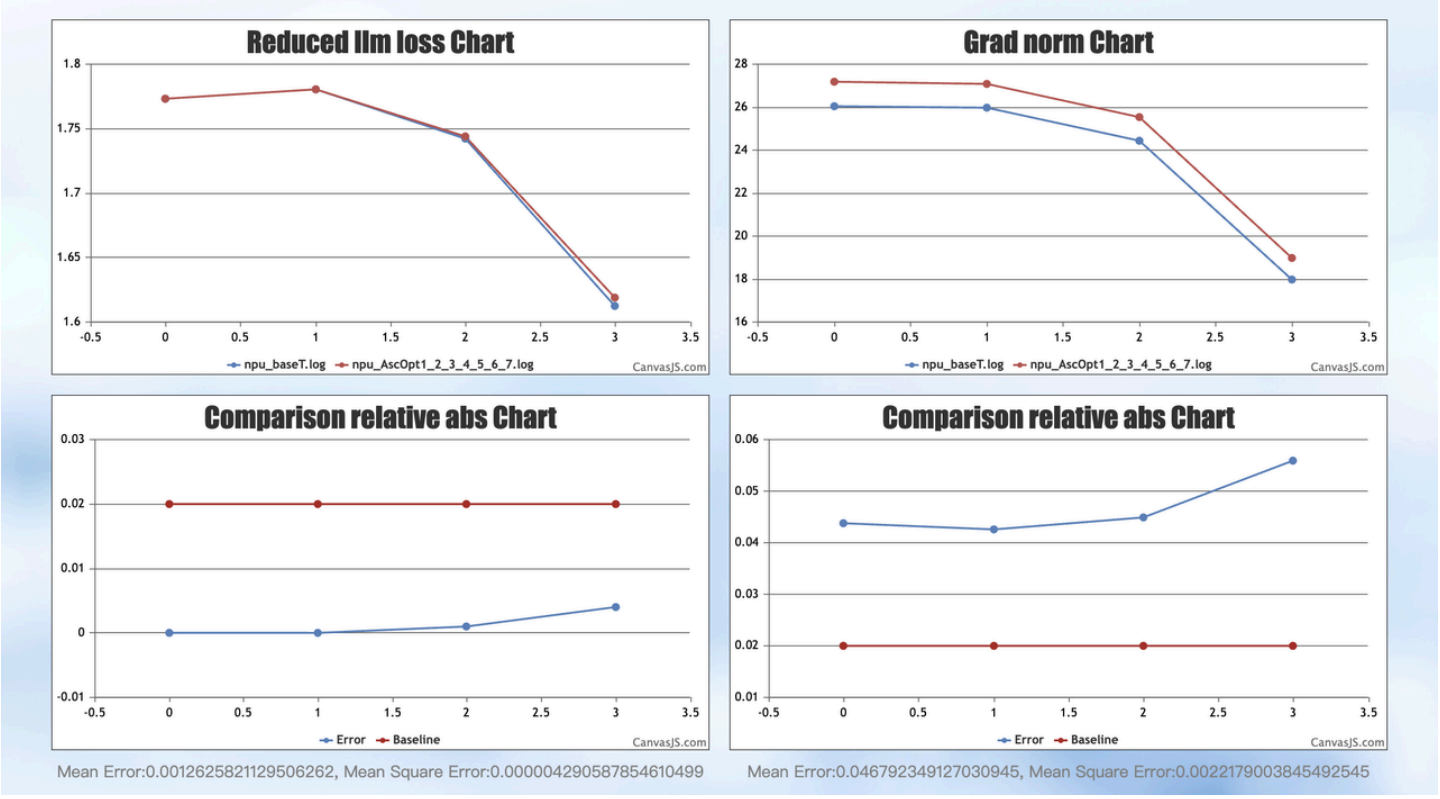
4. 精度

为了避免精度问题，现场采用较严格的首步grad norm 千分位要求。

4.1 Ascend C GDN 算子

7个GDN的Ascend C算子接入后导致首步超过千分位的grad norm差异，为此通过消融实验，定位到最后的第7个算子的问题：





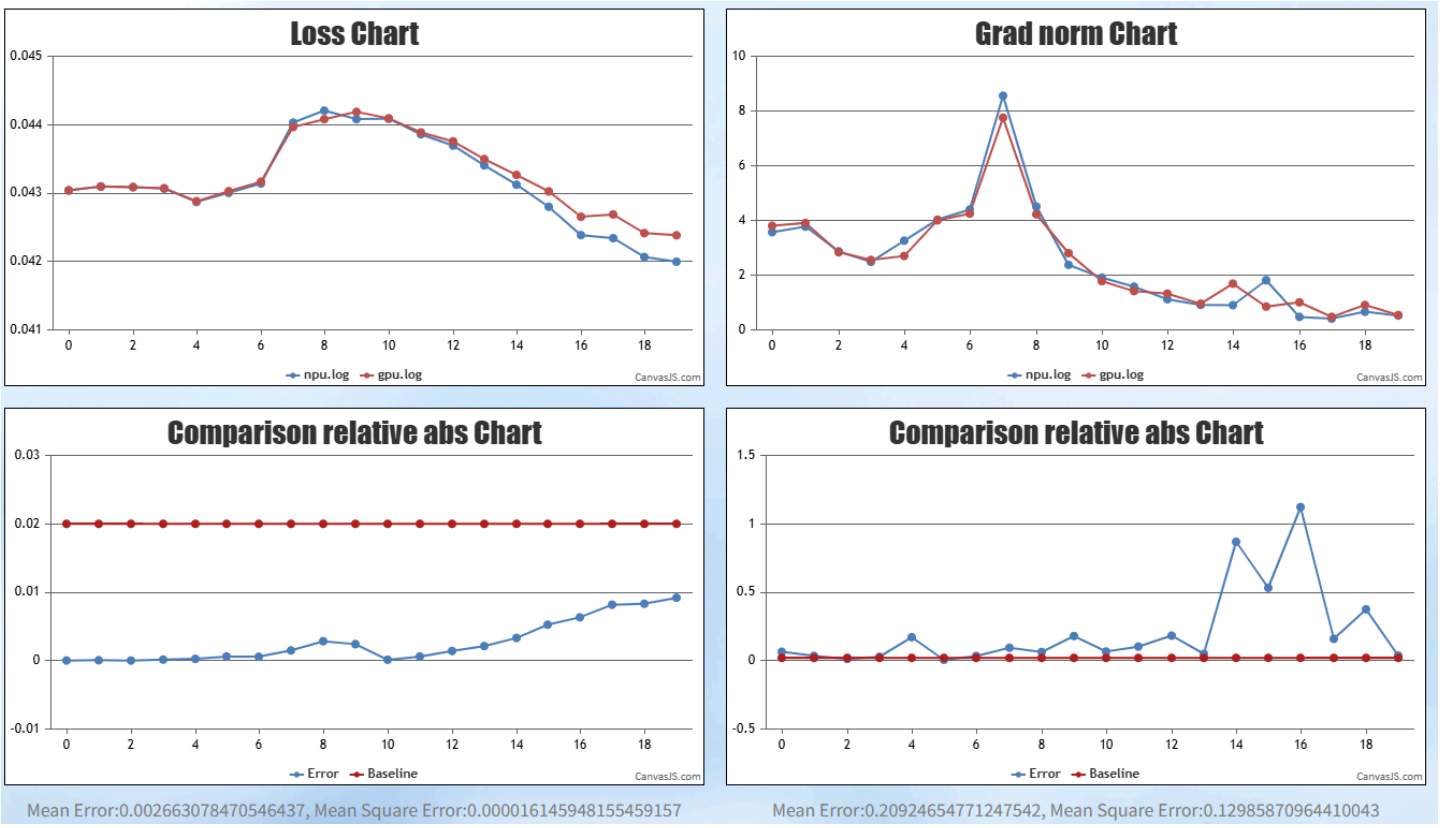
Mean Error:0.0012625821129506262, Mean Square Error:0.000004290587854610499

Mean Error:0.046792349127030945, Mean Square Error:0.0022179003845492545

算子同学分析后修复了不一致实现的问题，相关修改已归档于https://github.com/flashserve/flash-linear-attention-npu仓库

4.2 数据集切换到1k的分析

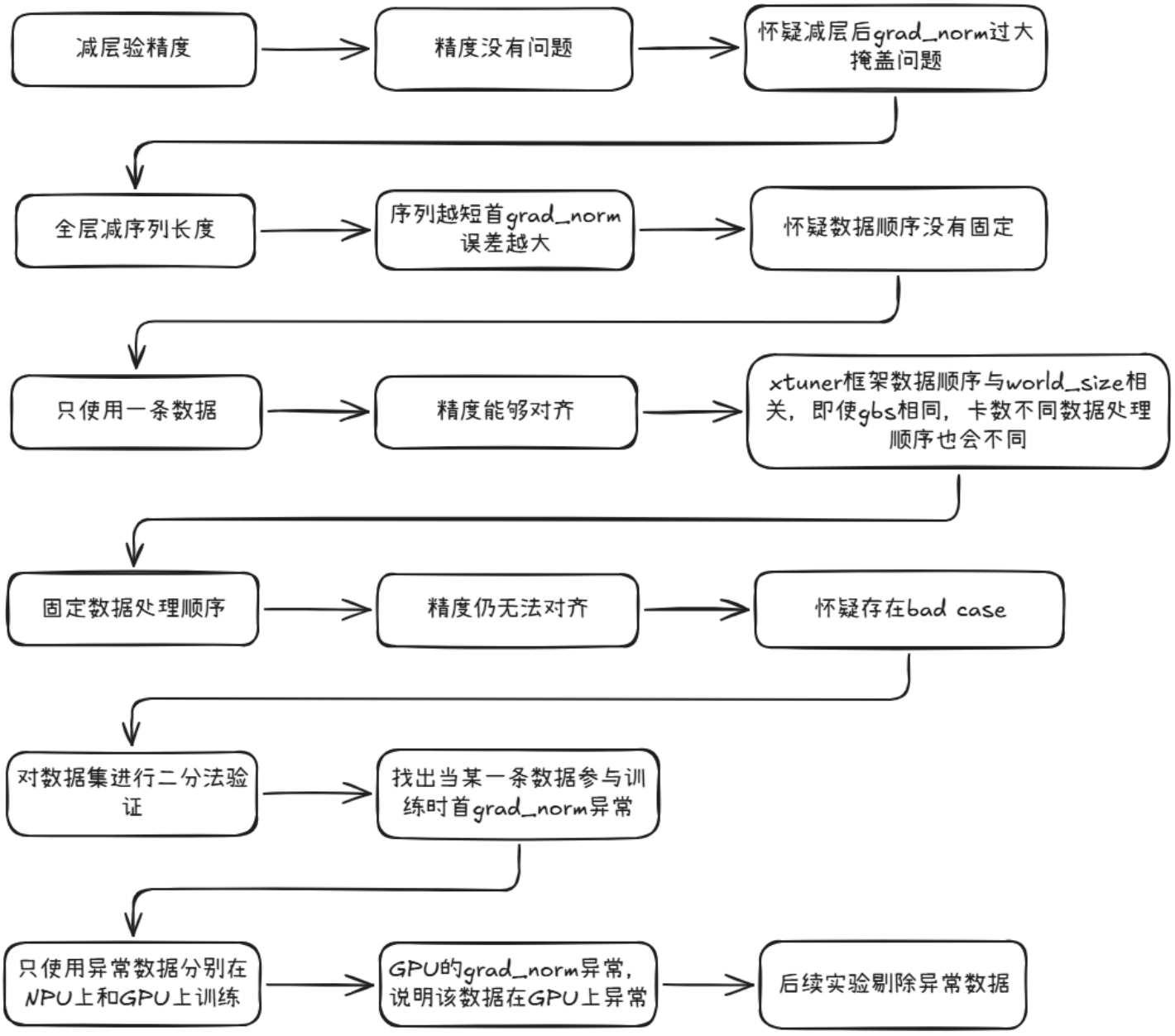
最开始我们在ALPACA开源数据集上进行训练，该数据集的平均序列长度为88，在精度已经对齐后我们切换到客户提供的真实数据集上，新数据集的平均序列长度为1K，但是首grad_norm却出现了较大误差（6.5%），需要定位分析。



Mean Error:0.002663078470546437, Mean Square Error:0.00001614594815545915 7

Mean Error:0.20924654771247542, Mean Square Error:0.12985870964410043

整个定位过程如下图所示：首先为了快速复现精度问题，对模型进行减层，但是发现精度没有问题，怀疑是减层后grad_norm过大导致问题被掩盖。随后通过减序列长度来复现精度问题，发现序列越短grad_norm误差越大，进而怀疑数据顺序未固定；通过单条数据测试确认了xtuner框架的数据处理顺序受world_size影响，但在固定数据顺序后精度仍无法对齐，从而推测存在坏数据（bad case）；随后利用二分法成功定位到导致梯度异常的具体数据，并通过在NPU和GPU上的对比实验确认该数据在GPU上存在异常，最终决定在后续实验中剔除该异常数据以解决问题。



4.3 算子内高精度实现

在高精度要求下，我们发现一类算子高精度实现与小算子低精度差异导致的精度差异。

1. Rope

Qwen3.5中新使用了一种partial rope的结构，我们使用npu_rotary_mul来替换其中的部分小算子，但是发现替换前后精度无法对齐，经过单算子复现发现确实存在精度问题。后咨询该算子接口同学得

知，npu_rotary_mul会自动在内部对输入进行升精操作，从而导致小算子无法与之对齐，将小算子的输入全部cast到float32，输出cast到bfloat16后精度正常。

```
q_embed = (q_rot * cos) + (rotate_half(q_rot) * sin)
k_embed = (k_rot * cos) + (rotate_half(k_rot) * sin)
# import torch_npu
# q_embed = torch_npu.npu_rotary_mul(q_rot, cos, sin)
# k_embed = torch_npu.npu_rotary_mul(k_rot, cos, sin)
```

2. rmsnorm 可以去除激活的cast，但是其余不行

Qwen3.5在RmsNorm上同样进行了创新，使用一种新的形式——zero_centered_rms_norm，在计算逻辑上相较以前的Rmsnorm需要额外对weight+1。我们最初在使用npu上的npu_rms_norm融合算子对其进行替换时，为了保持与小算子输入的格式一致，同样对x和weight进行float转换，但是因为npu_rms_norm会对输入进行自动升精，因此为了节省额外的cast耗时，将x和weight的float去除，结果去除后出现精度异常，经实验发现单独去除x的float不会有精度问题，而weight本身的大小为128，不去除weight的float并不会造成多余的cast耗时。

```
def native_zero_centered_rms_norm(x: torch.Tensor, weight: torch.Tensor, epsilon: float) -> torch.Tensor:
    def _norm(x):
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + epsilon)

    output = _norm(x.float())
    # Llama does x.to(float16) * w whilst Qwen3_5Moe is (x * w).to(float16)
    # See https://github.com/huggingface/transformers/pull/29402
    output = output * (1.0 + weight.float())
    return output.type_as(x)


def zero_centered_rms_norm_npu(x: torch.Tensor, weight: torch.Tensor, epsilon: float) -> torch.Tensor:
    import torch_npu
    output = torch_npu.npu_rms_norm(x, 1.0 + weight.float(), epsilon)[0]
    return output.type_as(x)
```

4.4 Casual Conv1d算子变长支持

Causal conv1d小算子不支持变长，而GPU上使用融合算子支持变长，这一差距导致最初开箱时精度存在问题。

Causal_conv1d最初在NPU上使用的小算子版本：

代码块

```
1   mixed_qkv =
    F.silu(F.conv1d(mixed_qkv,weight.unsqueeze(1),bias,padding=weight.shape[1]-1,gr
```

```
    oups=weight.shape[0])[:,:,:seq_len])
```

在GPU上使用的causal_conv1d库中的Triton版本的融合算子:

代码块

```
from causal_conv1d import causal_conv1d_fn
mixed_qkv = self.causal_conv1d_fn(
    x=mixed_qkv,  # need non contiguous
    weight=weight,
    bias=bias,
    activation=self.activation,
    seq_idx=seq_idx,
)
```

但是在对齐精度的过程中发现causal_conv1d的小算子版本和融合算子版本在变长(no varlen)情况下数学逻辑上不等价，因此需要设计一个在NPU上可用的支持变长的causal_conv1d算子。

之后发现mojo_opset库中存在可以在NPU上使用且支持变长的Triton版本的causal_conv1d算子，因此使用该算子替换小算子。

5. RL

关于xtuner-rl流程的一些概念和总体流程，可以查看上一篇稼先。

5.1 Partial Rollout

【RL流程在前一篇稼先的基础上变成异步】

1. Partial rollout定义

在RL中的完整一步迭代中，在Rollout出该step所需要的数据量后，就会强制停止所有的推理任务；对于其中被截断的推理请求，会暂存已经生成的token数的数据到**RelayBuffer**中，在下一次推理中作为新的输入继续生成。由于模型权重发生了更新，所以这也相当于是一种异步(**Async**)训练。

Partial Rollout主要意义在于复用推理结果来提高推理的资源利用率，但这也会造成long-tail的数据由不同的模型权重产生。客户训练采用的是“**两步异步**”的方法。

相较于InternS1-RL中的Partial Rollout, xtuner完善并扩充了这部分的逻辑。

2. 数据的状态分类

在单轮训练中，总共需要的推理数量为gbs * prompt_repeat_k。对于每一条推理数据，xtuner对数据状态作了分类与状态检查：

A. INIT： 数据在经过DataSampler采样后的初始化状态。

B. ABORTED： 数据已经 rollout 过，而且已经产生了部分有效轨迹，但是被系统中途截断的状态。

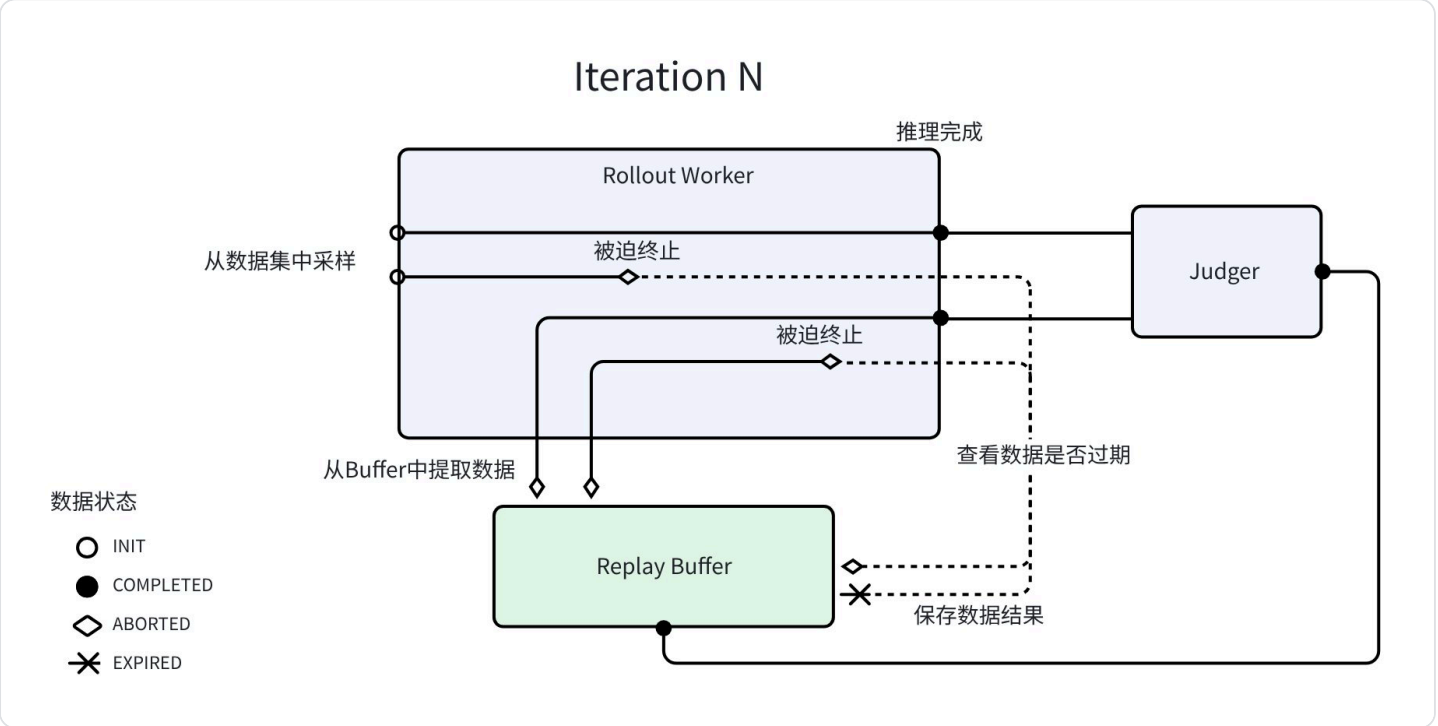
C. EXPIRED： 每条数据会有一个version控制，当version过老（权重已经迭代多轮），COMPLETED或者ABORTED数据不再适用于新的权重，就会转为EXPIRED状态。

D. COMPLETED：数据已经进行完整推理（到达response_len或末尾出现终止符）。xtuner会进一步检查response是否为空来确保数据的可用性。

3. 数据流行为

在单步推理中，对于每一条数据流，主要有下面四种过程：

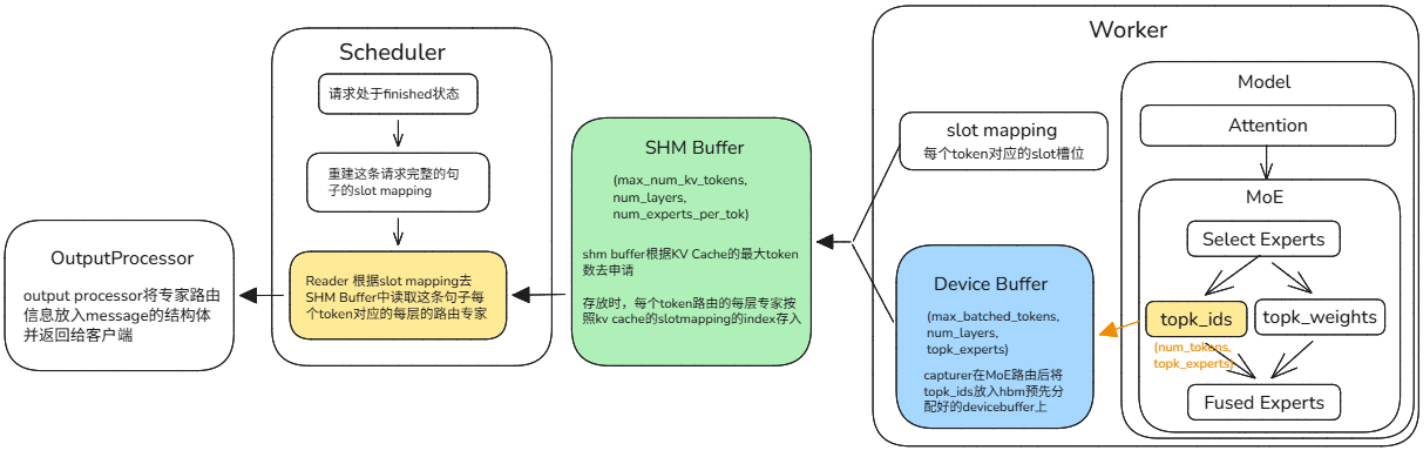
1. 从 INIT 开始且推理完成：
   如果这次推理正常结束，并且 response 完整可用，就会变成 `COMPLETED`，接着进入 reward/judger、replay buffer，最后被 trainer 取出做训练。
2. 从 INIT 开始，但中途被abort：
   会先进行 rollout，但由于被pause，状态会变成 `ABORTED`。
   这时它不会直接进入训练，而是会连同已经生成的那部分 token 一起存进 replay buffer，等待下一轮处理。
3. 一个 response 从 ABORTED 开始继续推理
   会先从 replay buffer 被重新取出，然后把“原始 prompt + 历史已经生成的 response 前缀”拼起来，作为新的prefill继续 rollout。
   如果这次补全后正常结束，就会转成 COMPLETED，再进入 reward、buffer 和训练；如果还是被中断，就继续保持 ABORTED，等待后续再续写。
4. 一个 response 从 ABORTED 开始，但推理次数过多，也就是转成 EXPIRED，不再使用。



5.2 Router Replay

Vllm 推理框架原有的Router Replay流程设计如下：

1. Capturer首先在MoE Select Experts路由专家后，捕获到本次的topk_ids，并按照对应的layer index存入，HBM预先分配好的Device Buffer上，这个Buffer的大小为(max_num_batched_tokens, num_layers, topk_experts)
2. 在模型前向执行结束后，Worker会将device buffer里的token取出，并将这些token的专家按照slot mapping的id存入host侧预先分配好的SHM Buffer上，这个buffer在shm共享内存里，这个buffer的大小和kv cache最大的token数一致（ma_num_kv_tokens, num_layers, num_experts_per_tok）
3. Scheduler里，当一条请求处于结束状态时（遇到eos或到达最大长度），会首先重建整体条句子KV Cache里的slot mapping，并根据这个slot mapping从SHM Buffer里取出这条句子每个token对应的专家
4. 这些专家会被返回到Output Processor，并进行后处理，放入message结构体返回到客户端



vllm上的Router Replay在我们的场景下有两个已知问题：

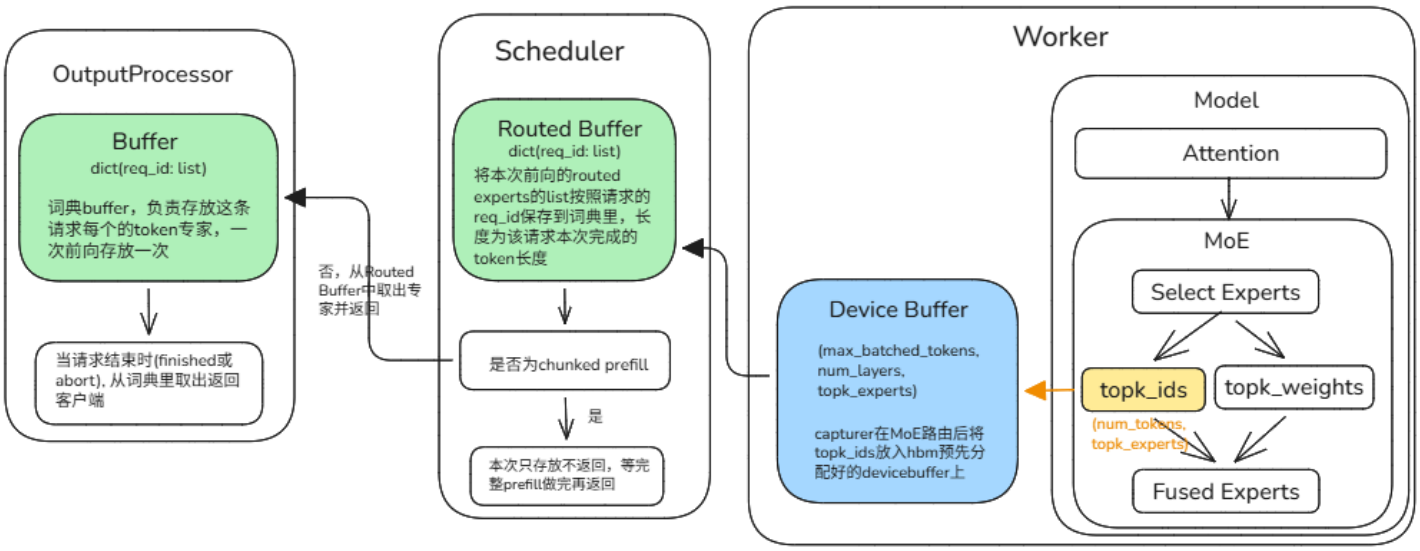
1. 共享内存读取依赖文件锁机制，且 SHM 缓冲区通过 NumPy 数组执行同步阻塞式内存拷贝，存在较高 CPU 开销，另外Reader需要一次性从SHM中读取一条句子的全部专家也存在较大cpu负荷。推理框架调度器Scheduler基于单线程事件循环设计，负责高频请求调度、KV Cache 分配及 Worker 任务下发。若调度流程中混入文件读写、锁等待等阻塞或CPU Heavy的操作，会直接造成调度器卡死。实测环境下，推理服务启动约 20 秒后完全hang住，业务吞吐降为 0，具体issue可以参考vllm官网的测试结果 https://github.com/vllm-project/vllm/issues/38079
2. 当前调用逻辑，只有当请求在finished状态时才能返回路由专家，abort场景下，路由专家无法返回

针对以上问题，我们需要对router replay进行重新设计，设计要点如下

1. 路由专家复用 vLLM Worker 现有通信链路，通过 `ModelRunnerOutput` 直接回传token信息至Scheduler，废弃共享内存缓冲区读写，规避文件锁带来的阻塞问题。

2. Scheduler在单次前向计算完成后，即刻回传当前步路由专家信息至输出OutputProcessor。既缩减单轮通信数据量、降低阻塞风险，又可避免请求abort时，已生成的路由数据无法传出的问题。
3. 将路由专家数据格式由 NumPy 数组调整为List[int]，彻底消除 NumPy 同步内存拷贝引发的阻塞问题

重新设计的流程如下：



1. 从 Device Buffer 直接读取专家数据并拷贝至 CPU，转为 list 格式，依托 vLLM 原生链路上报至 Scheduler。
2. 采用分步推送机制，每次前向计算完成后，将当前步专家信息同步至 OutputProcessor；在其内部维护字典结构，以 request_id 为键、专家信息 list 为值，逐轮追加数据。待请求全量处理完毕后统一返回客户端，同时规避 Scheduler 触发 Abort 时，因Scheduler和OutputProcessor交互中断造成的 router replay 数据丢失问题。
3. 受 chunked prefill 机制限制，未完成全量prefill的请求不会传递至 OutputProcessor。为此在 Scheduler 侧新增字典缓存，临时存储各请求已处理 token 对应的专家数据，待 prefill 完整执行完毕后再统一发出。

相关代码已提交到 Ascend-ShangHai-LLM：

https://github.com/Ascend-ShangHai-LLM/vllm/pull/1
https://github.com/Ascend-ShangHai-LLM/vllm-ascend/pull/5

5.3 训推一致性

LMDeploy 在 A3、H200 环境下首步 KL 稳定为1.7e-4；A3 环境下 vLLM-Ascend 首步 KL 为 5e-4，二者 KL 差值约 3 倍。

1. **轻量化场景复现**

原始业务配置：GBS=512、迭代次数 = 8、平均生成长度 11k，经过滤后需完整执行 4096 条请求；

轻量化复现配置：GBS=4、迭代次数 = 4、生成长度 1k，关闭数据过滤。

复现结果：

- LMDeploy 首步 KL：1e-4
- vLLM-Ascend 首步 KL：3.9e-4

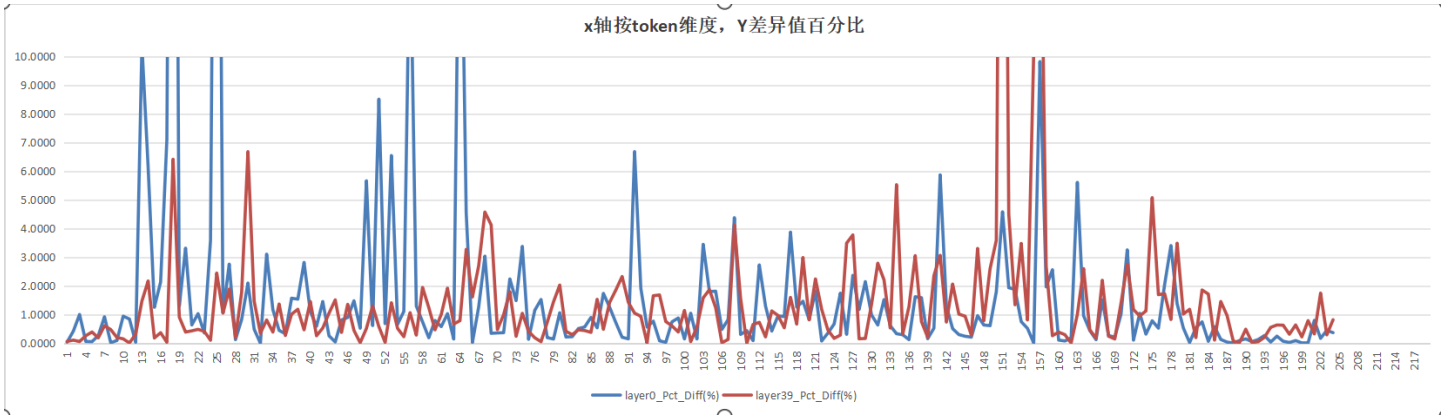
2. **分阶段 KL 拆解与数据分析**

首先分解prefill和decode对KL的贡献：

平均KL	lmdeploy	vllm-ascend
Prefill	1.6744e-06	-2.0489e-08
decode	7.4227e-05	0.0003

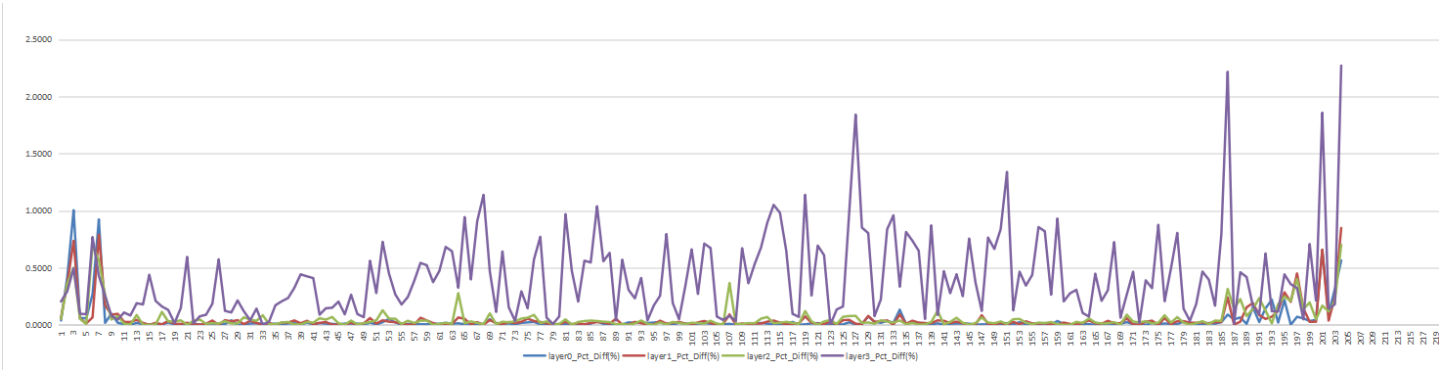
可见 Prefill 的 KL 差异极小， 但由于prefill的logprob只有单token维度，该指标不足以证明prefill的训推一致性效果。为精确定位偏差来源，我们决定逐层导出训练与推理侧各 Token 输出的 L1 Norm 做逐维比对。

数据如下：



逐层比对结果显示：模型首层 VIT 对应区间 Token（9~186）从首层即出现显著偏差，部分 Token L1 Norm 误差超 10%。优先对齐 VIT 模块输出，结合profiling定位，核心差异源于 RoPE 实现不一致；完成 RoPE 对齐后，VIT 输出 L1 Norm 误差由 0.02% 收敛至 0.01%。

VIT 对齐后，模型前三层全量 Token 训推误差收敛至 1% 以内，但第四层偏差再度扩大。结合模型结构差异分析，第四层启用 GQA 注意力机制，判断 GQA 模块为关键差异诱因。

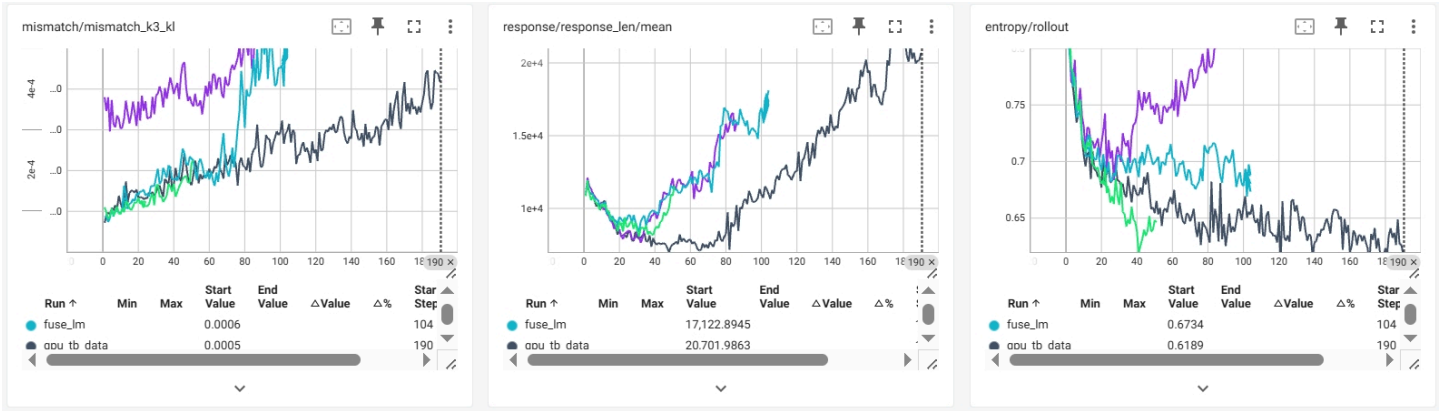


进一步通过算子剖析排查，明确 QKV 矩阵乘、M-RoPE 两处推理实现与训练侧存在差异。完成上述算子对齐优化后，整体平均 KL 仅优化 7%，量级无本质改善，vLLM-Ascend 仍维持在4e-4 量级，与 LMDeploy 基线存在明显差距。

综上，基于逐层 L1 Norm 溯源关键偏差算子的排查方案未达预期，同时验证：单一 LogProb 指标敏感度有限，无法有效量化算子替换对训推一致性的全局影响。

3. 指标选型结论：entropy更适配一致性度量

LogProb 值域波动范围有限，仅能反映单一生成 Token 的局部偏差；而**熵（Entropy）** 可表征全词表分布差异，对模型隐藏层特征、算子实现偏差更敏感，是更优的训推一致性评估指标，以下实验也证明了这点。



客户基于 LMDeploy 开展融合算子消融实验：关闭 MoE 门控、LayerNorm 融合算子后，如图虽然全局 KL 与生成长度无明显波动，但entropy曲线出现显著分化；原生 LMDeploy 融合算子方案熵值下降趋势平缓，消融后熵值明显回落。

另外横向对比：vLLM-Ascend 在 Prefill 阶段启用了 GDN 融合算子，熵值从40步后开始一直上升，与 LMDeploy 基线分布差异显著。可见entropy确实是一个更敏感的指标可以帮助我们度量训推一致性的差异

4. 标准化训推一致性排查思路

结合本次问题复盘，我们梳理了后续训推对齐的可行的标准化流程：

a. 依托 Profiling 全面梳理训练、推理框架的算子实现差异，形成差异算子清单；

b. 基于 Token 级 / 句子级熵值、LM Head 输出 L1Norm 等高敏感指标，开展算子消融对照实验；

c. 找到对训推一致性指标影响权重最高的核心算子进行替换优化。

6. 总结

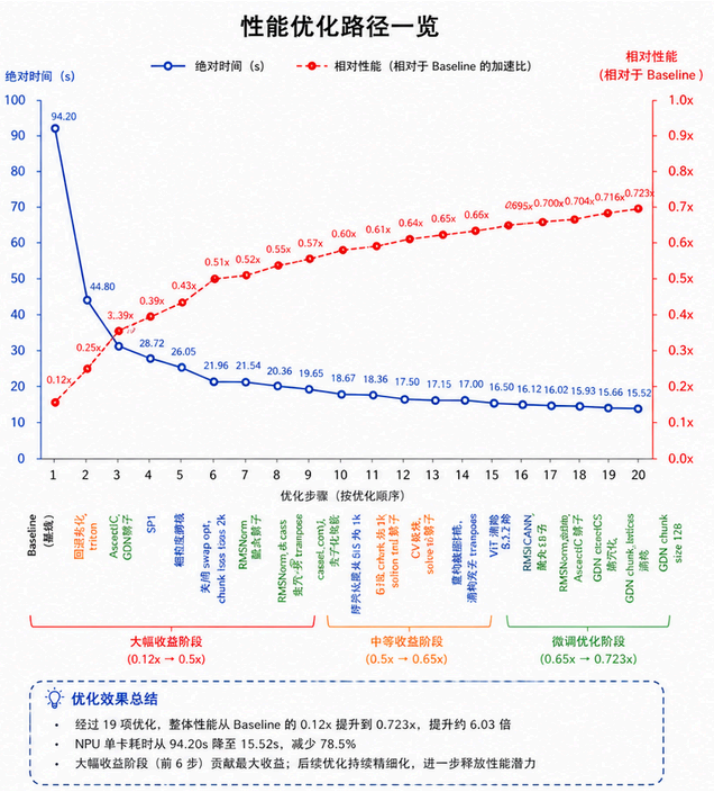
本次在A3超节点上针对Qwen3.5 的 SFT和RL流程进行了深入优化和分析，如下图，通过算子和框架联合优化，SFT性能最终达成 0.723x H200。

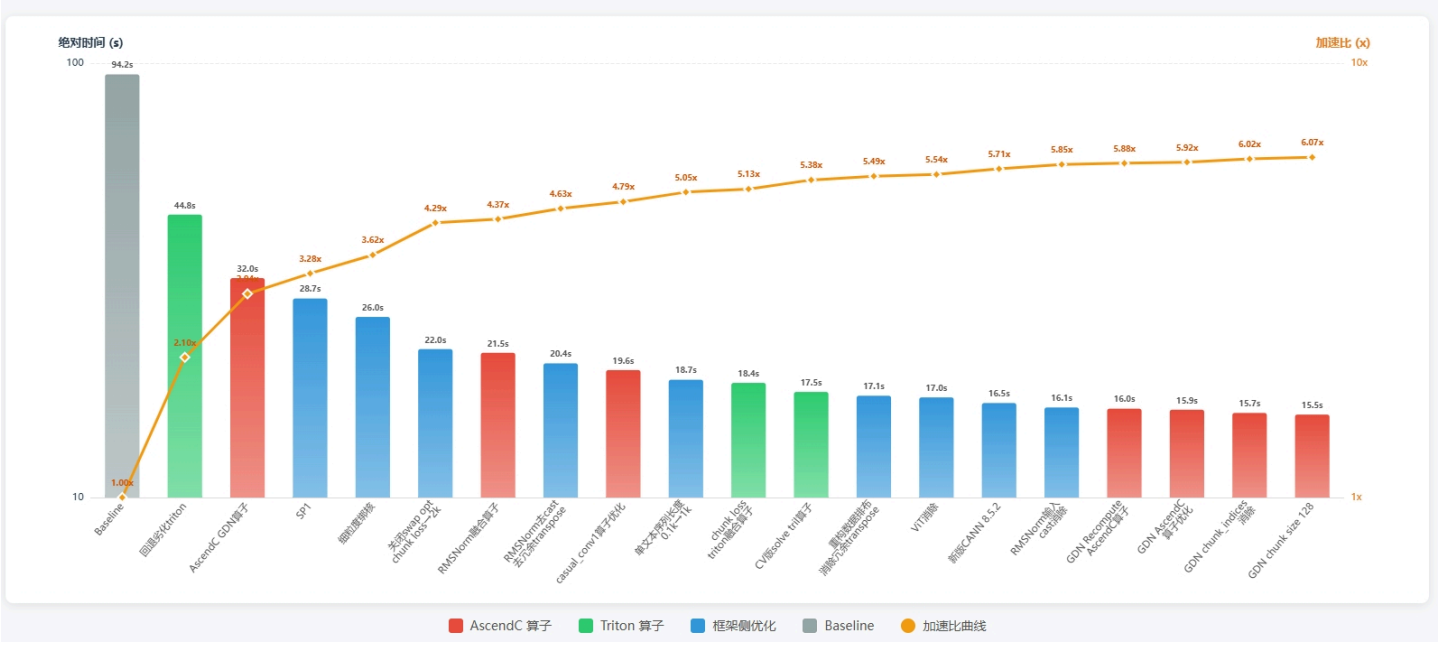
【下图2选1】

编号	优化点	具体描述	优化类型	相对性能 (相对于 Baseline)	绝对时间 (s)
1	Baseline（基线）	-	-	0.12x	94.20
2	回退劣化triton	triton 劣化实现回退	Triton	0.25x	44.80
3	AscendC GDN算子	高性能算子加速 GDN 模块	Ascend C	0.35x	32.00
4	SP1	SP1 减少 all2all 通信	框架侧优化	0.39x	28.72
5	细粒度绑核	缓解快慢卡问题	框架侧优化	0.43x	26.05
6	关闭 swap opt；chunk loss size 增大到 2k	充分利用显存，批量计算加速	框架侧优化	0.51x	21.96
7	RMSNorm 融合算子	融合算子加速	Ascend C	0.52x	21.54
8	RMSNorm 去掉 cast，并且去除冗余 transpose	去除冗余计算	Ascend C	0.55x	20.36
9	casual_conv1 算子优化	手动调整 tiling 规模，寻找硬件亲和划分	Ascend C	0.57x	19.65
10	单文本序列长度从 0.1k 为 1k	提高 NPU 单核计算效率	框架侧优化	0.60x	18.67
11	接入 chunk loss triton 融合算子	高效 triton 实现替代小算子	Triton	0.61x	18.36
12	CV 版本 solve tril 算子	更高效的 triton 实现，最大化硬件利用	Triton	0.64x	17.50
13	重构数据排布，消除冗余transpose	消除冗余计算	框架侧优化	0.65x	17.15
14	ViT 消除	空 ViT 数据性能比不真实，需要去掉评估	框架侧优化	0.66x	17.00
15	新版 CANN 8.5.2	新版毕昇编译器	框架侧优化	0.67x	16.50
15	RMSNorm 输入 cast 消除	去除非 weight 输入的 cast，不影响精度	Ascend C	0.695x	16.12
17	GDN Recompute AscendC算子	高性能算子	Ascend C	0.700x	16.02
18	GDN AscendC 算子优化	简化算子内同步	Ascend C	0.704x	15.93
19	GDN chunk_indices消除	合并冗余计算	Ascend C	0.716x	15.66
20	GDN chunk size 128	极致利用硬件	Ascend C	0.723x	15.52

Ascend C：Ascend C 算子优化　Triton：Triton 算子优化　框架侧优化　框架 / 系统层面优化

注：相对性能为相对于 Baseline（0.12x）的加速比（越大越好）；绝对时间为 NPU 单卡耗时（秒，越小越好）





我们总结了如下两大类经验以供参考：

| GND模块优化方案介绍