

分析这个在npu上运行的triton算子中间结果IR文件，分析哪里可以减小内存使用
算子代码：

```

import os
from typing import Optional, Tuple
from inspect import signature
import torch
import torch.nn.functional as F
import triton
import triton.language as tl

_CONDITIONS = ("seq7168,")

def _chunk_gated_delta_rule_fwd_h_cond(
    k: torch.Tensor,
    w: torch.Tensor,
    u: torch.Tensor,
    g: Optional[torch.Tensor] = None,
    initial_state: Optional[torch.Tensor] = None,
    output_final_state: bool = False,
    chunk_size: int = 64, # SY: remove this argument and force chunk size 64?
    save_new_value: bool = True,
    cu_seqlens: Optional[torch.LongTensor] = None,
):
    seq = k.shape[1]
    return f"seq{seq}" in _CONDITIONS, f"seq{seq}"

if os.environ.get("FLA_USE_FAST_OPS", "0") == "1":
    exp = tl.device.fast_expf
else:
    exp = tl.exp

@triton.jit
def safe_exp(x):
    # return exp(tl.where(x <= 0, x, float("-inf")))
    return exp(x)

@triton.heuristics(
{
    "USE_G": lambda args: args["g"] is not None,
    "USE_INITIAL_STATE": lambda args: args["h0"] is not None,
    "STORE_FINAL_STATE": lambda args: args["ht"] is not None,
    "SAVE_NEW_VALUE": lambda args: args["v_new"] is not None,
    "IS_VARLEN": lambda args: args["cu_seqlens"] is not None,
}
)

@triton.jit(do_not_specialize=["T"])
def chunk_gated_delta_rule_fwd_kernel_h_blockdim64(
    k,
    v,
    w,
    v_new,
    g,
    h,
    h0,
)

```

```

ht,
cu_seqlens,
chunk_offsets,
T,
H: tl.constexpr,
Hg: tl.constexpr,
K: tl.constexpr,
V: tl.constexpr,
BT: tl.constexpr,
BV: tl.constexpr,
USE_G: tl.constexpr,
USE_INITIAL_STATE: tl.constexpr,
STORE_FINAL_STATE: tl.constexpr,
SAVE_NEW_VALUE: tl.constexpr,
IS_VARLEN: tl.constexpr,
):
i_nh = tl.program_id(1)
i_n, i_h = i_nh // H, i_nh % H
T_max = T
if IS_VARLEN:
    bos, eos = tl.load(cu_seqlens + i_n).to(tl.int32), tl.load(
        cu_seqlens + i_n + 1
    ).to(tl.int32)
    T = eos - bos
    NT = tl.cdiv(T, BT)
    boh = tl.load(chunk_offsets + i_n).to(tl.int32)
else:
    bos, eos = i_n * T, i_n * T + T
    NT = tl.cdiv(T, BT)
    boh = i_n * NT

stride_v = H * V
stride_h = H * K * V
stride_k = Hg * K
stride_w = H * K

b_h1_bv1 = tl.zeros([128, BV], dtype=tl.float32)
b_h1_bv2 = tl.zeros([128, BV], dtype=tl.float32)

v_start1 = 0
v_start2 = BV

if USE_INITIAL_STATE:
    h0_ptr = h0 + i_nh * K * V
    p_h0_1_bv1 = tl.make_block_ptr(h0_ptr, (K, V), (V, 1), (0, v_start1), (128, BV), (1, 0))
    b_h1_bv1 += tl.load(p_h0_1_bv1, boundary_check=(0, 1)).to(tl.float32)

    p_h0_1_bv2 = tl.make_block_ptr(h0_ptr, (K, V), (V, 1), (0, v_start2), (128, BV), (1, 0))
    b_h1_bv2 += tl.load(p_h0_1_bv2, boundary_check=(0, 1)).to(tl.float32)

for i_t in range(NT):
    h_base = h + (boh + i_t) * H * K * V + i_h * K * V

    p_h1_bv1 = tl.make_block_ptr(h_base, (K, V), (V, 1), (0, v_start1), (128, BV), (1, 0))
    tl.store(p_h1_bv1, b_h1_bv1.to(p_h1_bv1.dtype.element_ty), boundary_check=(0, 1))

    p_h1_bv2 = tl.make_block_ptr(h_base, (K, V), (V, 1), (0, v_start2), (128, BV), (1, 0))
    tl.store(p_h1_bv2, b_h1_bv2.to(p_h1_bv2.dtype.element_ty), boundary_check=(0, 1))

    v_base = v + bos * H * V + i_h * V
    p_v1 = tl.make_block_ptr(v_base, (T, V), (stride_v, 1), (i_t * BT, v_start1), (BT, BV), (1, 0))
    b_v1 = tl.load(p_v1, boundary_check=(0, 1))
    b_v_new1 = b_v1.to(tl.float32)

```

```

p_v2 = tl.make_block_ptr(v_base, (T, V), (stride_v, 1), (i_t * BT, v_start2), (BT, BV), (1, 0))
b_v2 = tl.load(p_v2, boundary_check=(0, 1))
b_v_new2 = b_v2.to(tl.float32)

w_base = w + bos * H * K + i_h * K
p_w = tl.make_block_ptr(w_base, (T, K), (stride_w, 1), (i_t * BT, 0), (BT, 128), (1, 0))
b_w = tl.load(p_w, boundary_check=(0, 1))
b_v_new1 -= tl.dot(b_w, b_h1_bv1.to(b_w.dtype))
b_v_new2 -= tl.dot(b_w, b_h1_bv2.to(b_w.dtype))

if USE_G:
    last_idx = min((i_t + 1) * BT, T) - 1
    b_g_last = tl.load(g + bos + i_h * T_max + last_idx)
    p_g = tl.make_block_ptr(
        g + bos + i_h * T_max, (T,), (1,), (i_t * BT,), (BT,), (0,))
    b_g = tl.load(p_g, boundary_check=(0,))

    b_v_new1 = b_v_new1 * safe_exp(b_g_last - b_g)[;, None]
    b_v_new2 = b_v_new2 * safe_exp(b_g_last - b_g)[;, None]
    b_g_last = exp(b_g_last)
    b_h1_bv1 = b_h1_bv1 * b_g_last
    b_h1_bv2 = b_h1_bv2 * b_g_last

if SAVE_NEW_VALUE:
    v_new_base = v_new + bos * H * V + i_h * V
    p_v_new1 = tl.make_block_ptr(v_new_base, (T, V), (stride_v, 1), (i_t * BT, v_start1), (BT, BV), (1, 0))
    p_v_new2 = tl.make_block_ptr(v_new_base, (T, V), (stride_v, 1), (i_t * BT, v_start2), (BT, BV), (1, 0))
    tl.store(p_v_new1, b_v_new1.to(p_v_new1.dtype.element_ty), boundary_check=(0, 1))
    tl.store(p_v_new2, b_v_new2.to(p_v_new2.dtype.element_ty), boundary_check=(0, 1))

    b_v_new1 = b_v_new1.to(k.dtype.element_ty)
    b_v_new2 = b_v_new2.to(k.dtype.element_ty)

    k_base = k + bos * Hg * K + (i_h // (H // Hg)) * K
    p_k = tl.make_block_ptr(k_base, (K, T), (1, stride_k), (0, i_t * BT), (128, BT), (0, 1))
    b_k = tl.load(p_k, boundary_check=(0, 1))
    b_h1_bv1 += tl.dot(b_k, b_v_new1)
    b_h1_bv2 += tl.dot(b_k, b_v_new2)

if STORE_FINAL_STATE:
    ht_ptr = ht + i_nh * K * V
    p_ht1_bv1 = tl.make_block_ptr(ht_ptr, (K, V), (V, 1), (0, v_start1), (128, BV), (1, 0))
    tl.store(p_ht1_bv1, b_h1_bv1.to(p_ht1_bv1.dtype.element_ty), boundary_check=(0, 1))

    p_ht1_bv2 = tl.make_block_ptr(ht_ptr, (K, V), (V, 1), (0, v_start2), (128, BV), (1, 0))
    tl.store(p_ht1_bv2, b_h1_bv2.to(p_ht1_bv2.dtype.element_ty), boundary_check=(0, 1))

def prepare_lens(cu_seqlens: torch.LongTensor) -> torch.LongTensor:
    return cu_seqlens[1:] - cu_seqlens[:-1]

def prepare_chunk_indices(
    cu_seqlens: torch.LongTensor, chunk_size: int
) -> torch.LongTensor:
    indices = torch.cat([
        [
            torch.arange(n)
            for n in triton.cdiv(prepare_lens(cu_seqlens), chunk_size).tolist()
        ]
    ])
    return torch.stack([indices.eq(0).cumsum(0) - 1, indices], 1).to(cu_seqlens)

```

```

def prepare_chunk_offsets(
    cu_seqlens: torch.LongTensor, chunk_size: int
) -> torch.LongTensor:
    return torch.cat(
        [cu_seqlens.new_tensor([0]), triton.cdiv(prepare_lens(cu_seqlens), chunk_size)]
    ).cumsum(-1)

def chunk_gated_delta_rule_fwd_h(
    k: torch.Tensor,
    w: torch.Tensor,
    u: torch.Tensor,
    g: Optional[torch.Tensor] = None,
    initial_state: Optional[torch.Tensor] = None,
    output_final_state: bool = False,
    chunk_size: int = 64, # SY: remove this argument and force chunk size 64?
    save_new_value: bool = True,
    cu_seqlens: Optional[torch.LongTensor] = None,
) -> Tuple[torch.Tensor, torch.Tensor]:
    B, T, Hg, K, V = *k.shape, u.shape[-1]
    H = u.shape[-2]
    BT = chunk_size

    chunk_indices = (
        prepare_chunk_indices(cu_seqlens, chunk_size)
        if cu_seqlens is not None
        else None
    )
    # N: the actual number of sequences in the batch with either equal or variable lengths
    if cu_seqlens is None:
        N, NT, chunk_offsets = B, triton.cdiv(T, BT), None
    else:
        N, NT, chunk_offsets = (
            len(cu_seqlens) - 1,
            len(chunk_indices),
            prepare_chunk_offsets(cu_seqlens, BT),
        )
    assert K <= 256, "current kernel does not support head dimension larger than 256."

    h = k.new_empty(B, NT, H, K, V)
    final_state = (
        k.new_empty(N, H, K, V, dtype=torch.float32) if output_final_state else None
    )

    v_new = torch.empty_like(u) if save_new_value else None
    g = g.transpose(1, 2).contiguous()
    def grid(meta):
        return (1, N * H)

    chunk_gated_delta_rule_fwd_kernel_h_blockdim64[grid](
        k=k,
        v=u,
        w=w,
        v_new=v_new,
        g=g,
        h=h,
        h0=initial_state,
        ht=final_state,
        cu_seqlens=cu_seqlens,
        chunk_offsets=chunk_offsets,
        T=T,
    )

```

```

H=H,
Hg=Hg,
K=K,
V=V,
BT=BT,
BV=64,
num_warp=4,
num_stages=2,
)
return h, v_new, final_state

if __name__ == "__main__":
    if not torch.cuda.is_available():
        print("cuda is not available, use npu")
        from torch_npu.contrib import transfer_to_npu
    for cond in _CONDITIONS:
        args, kwargs = torch.load(f"./chunk_gated_delta_rule_fwd_h@{cond}_input.pt",
map_location="cuda")
        output = torch.load(f"./chunk_gated_delta_rule_fwd_h@{cond}_output.pt",
map_location="cuda")

        print(signature(chunk_gated_delta_rule_fwd_h).bind(*args, **kwargs))

        result = chunk_gated_delta_rule_fwd_h(*args, **kwargs)
        torch.testing.assert_close(result, output, atol=1e-3, rtol=1e-3)

```

已思考 7m 39s >

下面我分成四块来说：

这个 kernel 在干嘛 (把 Triton 代码和 IR 对上，方便我们知道数据流)

IR 里哪些 buffer /内存块实际在吃内存

哪些地方在“无形中放大了内存”

我们可以怎么减

我会一直用 BT=64 (时间块大小) , BV=64 (特征块大小的一半) , K=128 (state 维度) 这些当前 block 形状来描述。所有引用具体内存块形状、offset、大小、同步 flag 等的结论，都来自你给的 IR 片段。 hivim_IR hivim_IR

1. 算子逻辑 / 数据流回顾 (结合 Triton 代码)

内核在做每个序列分块 (chunk) 里的循环状态更新，流程按你的 Triton 代码大概是这样 (每个 head 独立，`i_h` 是当前 head) :

对第 `i_t` 个 chunk (长度 BT=64) :

把当前状态 `b_h1_bv1 / b_h1_bv2` (形状 [K=128, BV=64], float32 累积) 写到输出张量 `h[boh + i_t, i_h, :, :]`，也就是保存这一时刻的隐状态快照。

`python`

```
t1.store(p_h1_bv1, b_h1_bv1.to(dtype_of_h))
t1.store(p_h1_bv2, b_h1_bv2.to(dtype_of_h))
```

取本 chunk 的输入 `v` (原始值 `u`)，做

`b_v_new = v - w @ h_prev`

其中 `w` 是 (`BT x K`)，`h_prev` 是 (`K x BV`)。注意我们要对 `V` 的两个 64-d 子块分别做这件事，所以有 `bv1 / bv2` 两份。结果是 [`BT=64, BV=64`]，先保留成 float32。

`python`

```
b_v_new1 -= t1.dot(b_w, b_h1_bv1.to(b_w.dtype))
b_v_new2 -= t1.dot(b_w, b_h1_bv2.to(b_w.dtype))
```

如果用 gating (USE_G) :

计算 $scale_t = \exp(b_g_last - b_g)$, 其中 b_g 是长度 BT 的门值序列, b_g_last 是当前 chunk 最后一个 token 对应的标量。

把 b_v_new 逐行乘上这个 scale (需要把 [BT] broadcast 成 [BT, BV])。

把隐状态 $b_h1_bv^*$ 整体再乘一个 $\exp(b_g_last)$ 。

如果 SAVE_NEW_VALUE, 把 gated 后的 b_v_new 存到 v_new (这个是给上游/下游用的中间值)。

把 $b_v_new1/2$ cast 成和 k 一样的低精度 (通常 bf16), 然后做

```
b_h1_bv* += k^T @ b_v_new*
```

这里 k tile 形状大概是 ($K=128$, $BT=64$), b_v_new 是 ($BT=64$, $BV=64$), 结果又是 [$K=128$, $BV=64$], 累加到隐状态 (float32 累积)。

最后如果 STORE_FINAL_STATE, 在最后一个 chunk 把最终的 $b_h1_bv^*$ 写到 ht.

也就是说: 我们每个 chunk 里在两个方向上跑两次 GEMM, 再做一些逐元素的 exp / mul / broadcast, 状态不断在 float32 里积累。

IR 里其实正好把这条流水线切成两个硬件子核:

AIC (Cube / matmul 核心) 负责 GEMM (mmadL1) ;

AIV (Vector 核心) 负责逐元素的 exp / mul / broadcast / cast / 写回。

两个子核之间靠一块全局内存 “workspace” 传中间结果。IR 里这个 workspace 是 memref<?xi8, #hivm.address_space<gm>>, 然后被切成很多 view。 □ hivim_IR

2. IR 显示的内存占用在哪

2.1 最大的一块: workspace (~144 KB / block)

在 IR 里我们看到:

ir

```
%0 = memref.view %arg2[%c0_i320] [147456] : memref<?xi8, gm> to memref<?x?xbf16, gm>
%2 = memref.reinterpret_cast ... offset: [0]      sizes: [128, 64] -> memref<128x64xbf16, gm>
%6 = memref.reinterpret_cast ... offset: [16384] sizes: [128, 64] -> memref<128x64xbf16, gm>
%10 = memref.reinterpret_cast ... offset: [32768] sizes: [64, 64] -> memref<64x64xf32, gm>
%14 = memref.reinterpret_cast ... offset: [49152] sizes: [64, 64] -> memref<64x64xf32, gm>
%18 = memref.reinterpret_cast ... offset: [65536] sizes: [64, 64] -> memref<64x64xbf16, gm>
%22 = memref.reinterpret_cast ... offset: [73728] sizes: [64, 64] -> memref<64x64xbf16, gm>
%26 = memref.reinterpret_cast ... offset: [81920] sizes: [128, 64] -> memref<128x64xf32, gm>
%30 = memref.reinterpret_cast ... offset: [114688] sizes: [128, 64] -> memref<128x64xf32, gm>
```

整个 view 占用的 byte range 是 0 ~ 147456 (=144 KiB); 这些 offset 正好对齐 16KB/32KB 步长。 □ hivim_IR

我们可以把它理解成 8 个“信箱”:

$2 \times 128 \times 64 \times \text{bf16}$ (16KB each)

$2 \times 64 \times 64 \times \text{f32}$ (16KB each)

$2 \times 64 \times 64 \times \text{bf16}$ (8KB each)

$2 \times 128 \times 64 \times \text{f32}$ (32KB each)

合计: $16+16+16+8+8+32+32 = 144\text{KB}$ 。

这些信箱分别承载不同阶段的中间结果 (老状态 downcast 后的 $K \times V$ 切片、 $v - w @ h$ 的结果、gating 后的新 v_new 切片、更新过的隐状态切片等), 在 AIC \leftrightarrow AIV 两个子核之间来回传。AIC 的 matmul 会把输出用 fixpipe 直接写到这些 view (写到 GM), AIV 再从这些 view 里用 nd2nz / nz2nd 读回到 on-chip buffer 继续处理。 □ hivim_IR

这一块是当前最核心的中间内存开销来源, 而且是 per-(blockIdx) 的: 也就是说对于每个正在并行跑的逻辑 block (对应一个 (batch_id, head_id)), 都会占一份 144KB 的 scratch。

2.2 UB / CBUF / CC 上的双缓冲

IR 里到处有

ir

```
hivm.hir.annotation.mark %69 ... {multi_buffer = 2 : i64}
```

之类的标记，说明在 on-chip (ub / cbuf / cc) 层面我们还做了 double buffering 来隐藏 global memory latency。每个 tile 典型大小：

128x64xf32 在 UB 是 $128 \times 64 = 32\text{KB}$ ；

64x64xf32 在 UB 是 16KB；

128x64xbf16 在 UB 是 16KB；

等等。 □ hivim_IR

双缓冲意味着同一类 tile 我们可能要同时留两份 (current + next)，进一步推高本地 buffer 用量 (UB/CBUF 占用翻倍)。这不会增加 GM 的峰值占用，但是会增加单核可并发 block 数的内存门槛。

2.3 精度放大：很多中间值被强行升到 float32

看 Triton 源码：

`b_h1_bv*` 是 float32 累积的状态 (这个必须是 float32 做长序列累积，理论上比较难降)。

但有些中间量，比如 `b_v_new1 -= t1.dot(...)`，本来输入是 bf16 (`b_h1_bv1.to(b_w.dtype)`)，计算的 `t1.dot` 结果 naturally 是 float32，然后整个 `b_v_new1` 就变成 float32 [BT=64, BV=64]。你把这玩意儿原样写进 workspace 的时候，它就以 64x64xf32 的形式滞留在 GM (对应上面 offset 32768 / 49152 的 16KB + 16KB)。 □ hivim_IR

同理，更新完状态 `b_h1_bv* += k^T @ b_v_new*`，结果 state 仍然是 float32 (128x64xf32)，被 fixpipe 丢到 workspace 的那两块 32KB 大砖 (offset 81920 / 114688)。 □ hivim_IR

3. 哪里发生了“内存膨胀”

3.1 AIC/AIV 分核导致“GM 当作消息队列”

注意代码并不是一个单核里顺序做完，而是：

AIV (vector) 负责 elementwise exp / broadcast / mul / cast；

AIC (cube) 负责 GEMM。

两个核不能直接共享各自的 on-chip buffer，所以它们约定用一块 GM (上面那 144KB workspace 的各个切片) 作为“邮件箱”互相传阶段产物。

例子 (按 IR)：

AIV 把当前隐状态 `b_h1_bv*` downcast 成 bf16，写进 workspace 的 128x64xbf16 那两块；AIC 读出来做 `w @ h_prev`。 □ hivim_IR

AIC 把 (`v - w@h`) 的结果 (float32, 64x64xf32) 写回 workspace；AIV 读出来乘 `exp(b_g_last - b_g)`，做 gating；再把结果 cast 成 bf16，写到 workspace 的 64x64xbf16 两块，给 AIC 用来做 `k^T @ v_new`。 □ hivim_IR

AIC 做完 `k^T @ v_new` 得到新的隐状态 (float32, 128x64xf32)，再写到 workspace；AIV 再读回来，乘 `exp(b_g_last)` (对整块状态做缩放)，得到下一轮的 `b_h1_bv*` (float32 in UB)。 □ hivim_IR

这就解释了为什么 workspace 需要同时容纳：

bf16 的 [128, 64] (老状态副本给 AIC 用)、

f32 的 [64, 64] (中间 `new_v`, 没有 gating)、

bf16 的 [64, 64] (gating 后的 `new_v`, 下一个 GEMM 要用)、

f32 的 [128, 64] (更新后还没乘 exp 的新状态)。

问题：为了跨核通信，它把所有这些阶段的中间值都落到了 GM，并且是成块存 (64×64 或 128×64)，而不是按列/按行流水式地消费；所以需要那整块 144KB 的 mailbox 常驻。

从内存角度说，这是最贵的一点，也是 NPU hetero-pipeline 的典型代价：GM 被迫当 scratch。

3.2 中间值是 float32，而后面其实会降精度

看刚才那条数据流：

$(v - w@h) \rightarrow \text{float32 tile } [64, 64] \rightarrow \text{AIV 读} \rightarrow \text{gating、乘 exp、再 cast 成 bf16} \rightarrow \text{存回 } [64, 64] \text{ bf16 给下一步 GEMM。}$

也就是说，这个 $[64, 64]$ float32 tile 的“最终命运”就是被乘一堆标量/逐元素缩放后再 cast 成 bf16，然后下游 GEMM 其实用的就是 bf16 版本。

但在当前 IR，我们还是把 float32 版本整块写进 GM，再从 GM 把 float32 整块读回来做 gating。 □ hivim_IR

同样地，对新的隐状态 ($b_h1_bv^* += k^T @ v_new$)：

AIC 产生 float32 $[128, 64]$ ，写 GM；

AIV 再读回来乘 $\exp(b_g_last)$ ，得到下一轮的隐状态（仍然 float32 保存在 UB 的 loop-carried $\%arg18/\%arg19$ ），然后又把 bf16 版写回 workspace 作为“老状态 downcast 版”。

这里我们确实需要 float32 版本的隐状态在循环里继续累加（长序列数值稳定性）。但我们是否需要把 *pre-scale* 的那一版 float32 全量落到 GM，而不是直接在 AIC 端先 downcast 再写 GM？这个就是一个可优化点（后面会详细说）。

3.3 广播 [:, None] 带来的 tile 级扩维

```
b_v_new1 = b_v_new1 * safe_exp(b_g_last - b_g)[: , None]
```

语义上， $(b_g_last - b_g)$ 是长度 BT=64 的向量，把它 [:, None] 后逻辑 shape 是 $[64, 1]$ ，然后要对 b_v_new1 的 $[64, 64]$ 每列都乘同一列向量。

IR 里可以看到：

它先把 $b_g_last - b_g$ 算成一个 $[64]/[64, 1]$ 的 UB buffer；

再用 expand_shape / collapse_shape 之类的 op，把这个 $[64, 1]$ 按列广播成 $[64, 64]$ 的效果，然后用 vmul 去乘已有的 $[64, 64]$ tile。中间还分配了一个小的 memref<512xf32, ub> 临时空间做广播/对齐。 □ hivim_IR

这个广播是在 UB（向量核本地）里发生的，所以不会额外占 GM 空间。但是它强制要求我们必须整块留住那两个 $[64, 64]$ float32 tile (16KB each) 在 UB，否则没法一次性做逐元素乘。换句话说，这一步也推高了 tile 的粒度 (64x64 一下子就 16KB)，让我们必须把那 16KB float32 tile 先写到 GM 再读回来处理。

4. 可以怎么降内存 (按优先级/现实度分)

4.1 优先级最高：压缩“跨核信箱”(workspace) 的精度

现在 workspace 里最贵的是那几块 float32：

两块 $64 \times 64 \times f32$ ($16\text{KB} \times 2 = 32\text{KB}$)

两块 $128 \times 64 \times f32$ ($32\text{KB} \times 2 = 64\text{KB}$)

合计 96KB，占整个 144KB workspace 的 2/3。 □ hivim_IR

观察它们的用途：

$64 \times 64 \times f32$ 是 $(v - w@h)$ 还没乘 gating 的结果；AIV 读回来后第一件事就是按行乘一个 scale（由 $\exp(b_g_last - b_g)$ 得到），之后会再把它 cast 成 bf16，既要写到 v_new （可选）也要把 bf16 版本喂给下一步 $k^T @ v_new$ 。

结论：**AIC 其实可以在写 GM 之前就先把 tile downcast 成 bf16 存 GM。**

AIV 读 bf16 → 立刻 upcast 回 f32 (UB 内做一条 vcast) → 乘 gating。

损失：gating 乘法是在 f32 里做的，输入从 bf16 还原，会有一点额外量化误差。但注意后面马上还是会再 cast 回 bf16 才给后续 GEMM，所以我们本来就要丢精度。大概率可接受。

收益：这两块 mailbox 从 16KB 变 8KB，各省一半，总共省 ~16KB。

$128 \times 64 \times f32$ 是 $k^T @ v_new$ 的结果，也就是“下一时刻的隐状态（还没乘 $\exp(b_g_last)$ ）”。AIV 读回来会做一件事：先把它乘上 $\exp(b_g_last)$ （标量广播到整块），然后这变成新的 loop-carried 隐状态 (float32) 留在 UB 的 $\%arg18/\%arg19$ ，用于下一轮循环。 □ hivim_IR

这里 tricky：loop-carried 的隐状态我们的确希望保留成 float32（长序列累积稳定性）。

但是，**AIC 可以不必把 float32 直接落 GM。**

方案 A (保守 + 省 GM 带宽 / 容量)：AIC 把 float32 结果先 downcast bf16 再写 GM。AIV 读回 bf16 → upcast 到 f32 → 乘 $\exp(b_g_last)$ → 得到下一轮的 loop-carried 隐状态 (float32 in UB)。

方案 B (更激进)：让 AIC 根本不把这一步的结果落 GM，而是把 AIC 的这一块 GEMM 合并到 AIV 这一核（也就是让“乘 $\exp(b_g_last)$ ”直接发生在 GEMM 之后，仍在 on-chip float32），然后只把 downcast 之后的 bf16 版本写回 workspace 作为“给下一轮 w@h 的输入”。这等价于把“GEMM2 ($k^T @ v_new$) + $\exp(b_g_last)$ 缩放”看成同一阶段，从而少一次 GM 往返。这个需要更重的核间融合/调度调整。

即使只做保守的“所有跨核 mailbox 全部以 bf16 形式落 GM，再到对端 upcast 回 f32”，理论上 workspace 里那两块 32KB 也可以压到 $16\text{KB} \times 2 = 32\text{KB}$ ，而不是 64KB。加上上面 16KB 的节约，总共可从 $96\text{KB} \rightarrow 48\text{KB}$ ，workspace 峰值从 ~144KB 掉到 ~96KB / block，直接少三分之一以上。

这是在不改变数学公式、只改变“中间缓存的精度”的前提下做到的（核心思想：AIC 不再把 float32 中间值直接写 GM，而是先本地 cast→bf16，再写；AIV 负责 upcast 回 float32 做后续逐元素运算 / 累加）。

这类改动对模型数值会有一点影响，但注意我们后续反正都在 bf16 上做 GEMM 和存档，所以大部分分支本来就不是全程保持 float32 精度，通常是可以接受/可调的。

4.2 选更细的 tile (缩小 BV)

当前我们一次性处理 $\text{BV}=64$ 的列块并把 $[64, 64]$ 或 $[128, 64]$ 整块缓存下来。

如果我们把 BV 切半，比如 $\text{BV}=32$ ，一次只处理 32 列：

mailbox 里 $[64, 64]$ f32/bf16 tile 会降成 $[64, 32]$ ，直接减半；

$[128, 64]$ tile 也跟着变 $[128, 32]$ ，再减半；

workspace 的常驻容量线性下降。

代价：

kernel loop 次数翻倍，调度/同步/flag (SET_FLAG/WAIT_FLAG) 更多；

GEMM tile 变小，AIC 矩阵乘效率可能明显下降 (cube unit 通常喜欢至少 64×64 或 128×64 这种 tile 才能喂满)。

也就是说：**BV 变小 = 用算力换内存。**

如果你在 NPU 上最卡的是显存/临时 GM 而不是算力，这个是可以尝试的方向。

4.3 减少“必须存出来的中间结果”：按列/按行即时消费，避免整块落 GM

看 gating 这段：

AIV 把 $(v - w@h)$ 这块 $[64, 64]$ tile 读回来 (现在是 float32)，然后做广播乘以 $\exp(b_g_last - b_g)$ (逐行的 scale)。

实际上这个乘法是逐行相同的系数，对每一列都一样。因此理论上可以按行 streaming：

AIC 把 $(v - w@h)$ 还没 gating 的行块 (1×64 或者小行块) 写出去；

AIV 立刻读这一小块、乘 gating、(可选) 写到 v_new / 喂给下一个 GEMM；

不一定需要把完整的 64×64 tile 先落在 GM 再整块读。

IR 现在是整块 tile 落 GM，再整块读回来，因为这样最容易做高效大矩阵 vmul，也方便后面一次性 vmul 广播 (IR 里可以看到 expand_shape -> broadcast -> vmul 的模式，说明它是“先准备好 64×64 ，再一次性乘”而不是 streaming)。 

要把它改成 streaming，需要：

重排循环，让 AIC/AIV 在 chunk 内更细粒度地交替；

或者在同一核里 fuse 一部分逐元素操作，避免把还没乘 gating 的那份结果写回 GM。

这属于“跨核fuse/调度重写”的量级，工程成本高，但它可以把 “ $64 \times 64 \times f32$ 这个中间产物” 直接消掉 (或者至少极大缩小它的生命周期 / 常驻大小)。

4.4 降低输出需求 (接口级策略)

从 Python 侧：

`h = k.new_empty(B, NT, H, K, V)`：我们无条件分配了整块 h 并且在循环里每个 chunk 都往里写一份快照 (`t1.store(p_h1_bv*, ...)`)。

如果下游并不真的需要所有中间 chunk state (例如纯推理时，你可能只关心最后的 ht)，可以把“保存每个 chunk 的状态”做成可选，或者把 h 换成一个 ring buffer (只留最近几块)，而不是 $[NT, H, K, V]$ 全存。

这样可以从根上砍一大块显存占用，因为 h 是 $[NT, H, K, V]$ 这种 5D 张量，往往非常大。

`v_new = torch.empty_like(u)` 只有在 `save_new_value=True` 时才真的会被用到。已经有 `SAVE_NEW_VALUE` 这个 heuristic，但在调用端可以更激进：如果推理/训练的这一步其实不需要把中间的 `v_new` 留给别人，就关掉它。

这会少掉一次全局写入和潜在的长生命周期缓冲。

`g = g.transpose(1, 2).contiguous()`：这在 host 侧就复制了一份 $[B, H, T]$ 的 gating 向量成新的 layout，占用额外内存。

如果 NPU kernel 能接受原 layout (或者我们能在 kernel 里做更轻量的非拷贝 view / stride 访问，而不是 `.contiguous()` 复制)，可以省掉这份重复的 gating buffer。

这些都是“API / 调用策略层面”的内存优化，不涉及 NPU IR 本身，但对总体占用 (尤其大 batch / 长序列) 其实是最直接的减法。

4.5 降低 on-chip double buffering (multi_buffer=2)

IR 里大量的 `multi_buffer = 2` 意味着我们为吞吐量开了双缓冲，比如：

```
%69 = memref.alloca ... {multi_buffer = 2}
```

然后 `WAIT_FLAG`, `SET_FLAG` 等硬同步，典型的 producer-consumer pipeline。 hivim_IR

如果芯片的 UB/CBUF/L1 容量吃紧，而你宁愿牺牲并行流水深度 / 吞吐，可以把 `multi_buffer` 从 2 降到 1。这样每个正在执行的 block 占用的 UB/CBUF 会直接砍半，允许在极限内塞更多 block 或者在资源受限时跑得下去。

代价：性能会掉（无法 overlap load/compute/store），但峰值 on-chip buffer 占用下降。

总结（重点结论）

(A) 现在最大的内存开销是 workspace，那块 144KB/并发block 的“跨核邮箱”。 它之所以这么大，是因为：

我们把两个半维度 (BV=64 的两半) 都保留成整块 tile；

很多中间结果都在 float32 里整体落到 GM 再读回来；

AIC / AIV 拆核执行，必须靠 GM 传这些 tile。 hivim_IR hivim_IR

(B) 直接的省内存方向：让这些 mailbox 里的中间产物以 bf16（而不是 f32）落到 GM，再由对端核 upcast 回 f32。

这能把 workspace 里那 96KB 的 float32 大块近似砍半，整体 workspace 峰值从 ~144KB/block 降到 ~96KB/block 量级，非常可观，同时改动相对可控（本质是把 fixpipe 写回前多一步 cast）。 hivim_IR

(C) 进一步的方向（代价更高）：

把 AIC/AIV 的阶段进一步 fuse，让一部分逐元素操作 (gating、exp 缩放) 在同一核 / 同一 pipeline 段里完成，不再把“未gating的 f32 tile”整块写 GM 再读回来；

把列块 BV 从 64 改成 32 (甚至 16)，以 tile 更小换更少的 mailbox 常驻空间；

在接口上让 h / v_new 变成可选输出 (推理时很多时候只需要最终状态 ht)，以及避免对 g 做整块 .contiguous() 复制；

降低 `multi_buffer` 深度，牺牲吞吐换 on-chip buffer 峰值下降。 hivim_IR

如果你要一个一句话版重点：把 AIC→AIV / AIV→AIC 的中间交换张量从 float32 改成 bf16（由消费者核再临时上采样回 f32 计算），这是内存-收益比最高的第一刀，直接帮你缩掉 workspace 的三分之一以上峰值占用，而不必马上改算子数学或调度。

来源

□ ↗ ↙ ⌂ ⌂ ...

好的，很好的分析，然后告诉我“压缩“跨核信箱”（workspace）的精度”（要求精度不能下降很多）、把 AIC/AIV 的阶段进一步 fuse，让一部分逐元素操作 (gating、exp 缩放) 在同一核 / 同一 pipeline 段里完成，不再把“未gating的 f32 tile”整块写 GM 再读回来 这两个修改如何准确落实到我给你的算子代码中

已思考 2m 30s >

太好了，这一步我们开始“动刀”代码本身了 ✓

我会分别讲两个改法怎么真正落到你这版 Triton kernel 里，并给出接近可直接替换的代码骨架（不是逐行 diff，而是结构化改写，方便你对比）。

先统一两个概念（重要）

AIV / 向量核：做逐元素操作 (exp, *, broadcast 之类)，喜欢 float32 精度。

AIC / 矩阵核：做大块 matmul (t1.dot)，喜欢 bf16 输入，accumulate 成 float32。

你现在这个 Triton kernel 在高层逻辑上已经是“一整个 for 循环都在一个 kernel 里”，但 NPU 后端在 IR 里会自动把它拆成“向量核”和“矩阵核”的流水线，并用一块全局内存（我们说的 workspace / mailbox）互相传 tile。这个 mailbox 里现在存了很多 float32 的中间 tile，导致 GM 上的峰值内存很大。

我们要做两件事：

压缩 mailbox 精度：保证跨核传输走的是 bf16 (或和 k / w 相同的低精度) , 而不是大块的 float32。

把 gating / exp 缩放 提前融合进同一轮计算里, 让“未 gating 的 float32 大 tile”根本没有理由落盘到 GM。

下面分别说怎么在你的 Triton 代码里明确表达这两个意图。

Part 1. 压缩“跨核信箱”(workspace) 精度

问题回顾

IR 里出现了这些跨核中转块 (举例) :

64×64 的 float32 tile: 这是 ($v - w @ h$) 的结果, 尚未 gating。

128×64 的 float32 tile: 这是更新后隐状态的 float32 版本。

这些 tile 之所以是 float32 被写到 GM, 是因为:

AIC (做 matmul) 产出了 float32 结果;

AIW 需要再对它做逐元素操作;

两个核不能共享寄存器/UB, 所以只能把 float32 全块写到 GM, 然后另一核再读。

我们希望 mailbox 传 bf16, 而不是 float32。意思是: **在写进 GM 之前先 downcast 成 bf16; 对端核从 GM 读到 bf16, 再立刻 upcast 回 float32 继续算。**

为了让编译器/后端更愿意这么做, 你要在 Triton kernel 里“显式存在”这两个版本 (float32主状态 + bf16影子状态), 而不是只有 float32 变量, 让编译器自己在分核时去猜怎么降精度。

也就是说: 我们主动维护一份 **bf16 影子**, 并只把影子那份往外存。

怎么改代码

关键点是两个地方:

(1) 隐状态 b_h1_bv*:

现在代码里你只有:

python

```
b_h1_bv1 = tl.zeros([128, BV], dtype=tl.float32)
b_h1_bv2 = tl.zeros([128, BV], dtype=tl.float32)
```

我们加上影子版本 (低精度, 通常 bf16) :

python

```
b_h1_bv1_f32 = tl.zeros([128, BV], dtype=tl.float32)
b_h1_bv2_f32 = tl.zeros([128, BV], dtype=tl.float32)
```

```
# 影子, 用于喂给 W (w @ h) 这条 GEMM 和跨核通信
state_shadow_bv1 = b_h1_bv1_f32.to(tl.bfloat16) # or k.dtype.element_ty
state_shadow_bv2 = b_h1_bv2_f32.to(tl.bfloat16)
```

注意: tl.bfloat16 这里代表“低精度格式”, 实际上你应该用 k.dtype.element_ty (或者 w.dtype) 来保持一致, 这样编译器知道矩阵核读的是低精度版本。

然后在每一轮循环结束后, 更新这两个版本:

python

```
# 更新 float32 主状态
b_h1_bv1_f32 = new_state_bv1_f32      # float32, 用于数值稳定的累计
```

```
b_h1_bv2_f32 = new_state_bv2_f32

# 重新生成影子, shadow 用于下一轮 w@h
state_shadow_bv1 = b_h1_bv1_f32.to(k.dtype.element_ty)
state_shadow_bv2 = b_h1_bv2_f32.to(k.dtype.element_ty)
```

这样告诉编译器：

“下一轮 $w @ h_{prev}$ 只需要用 `state_shadow_*` (`bf16`)，不必把整块 `float32` 的 `b_h1_bv*_f32` 泄到 GM 再 downcast。”
“跨核如果一定要走 GM，请用 `shadow` (`bf16`)，不是主状态 (`float32`)。”

然后你在算 $w @ h$ 时，显式用 `shadow`，而不要在 `t1.dot` 里直接 `.to(b_w.dtype)`：

旧写法：

python

```
b_v_new1 -= t1.dot(b_w, b_h1_bv1.to(b_w.dtype))
```

新写法：

python

```
# 先准备 shadow (bf16) 版本，见上面
prod1 = t1.dot(b_w, state_shadow_bv1) # prod1 accumulates as fp32
prod2 = t1.dot(b_w, state_shadow_bv2)

b_v_new1_f32 = b_v1.to(t1.float32) - prod1
b_v_new2_f32 = b_v2.to(t1.float32) - prod2
```

这样 AIC 看到的是 `bf16` 输入 (`state_shadow_*`) 而不是 `float32` 主状态，从而不需要把 `float32` 主状态写进 GM 给它。

(2) ($v - w@h$) 结果本身：

我们马上就会把它 `gating`，然后立刻 `cast` 成低精度喂给下一个 GEMM ($k^T @ \dots$)。对外（跨核）其实只需要那个低精度版本。

我们可以显式做两步变量：

python

```
# 还在 float32 的版本，用于 gating 计算
b_v_new1_f32 = ... # float32
b_v_new2_f32 = ...

# gating 后（还是 float32，见 Part 2）
# ...

# SAVE_NEW_VALUE 这一步，我们要写出去的就是 bf16 → 这一版就是 mailbox 走出去的版本
b_v_new1_bf16 = b_v_new1_f32.to(k.dtype.element_ty) # bf16
b_v_new2_bf16 = b_v_new2_f32.to(k.dtype.element_ty)

# 喂第二个 GEMM 用 bf16
delta1_f32 = t1.dot(b_k, b_v_new1_bf16)
delta2_f32 = t1.dot(b_k, b_v_new2_bf16)
```

并且，在 `SAVE_NEW_VALUE` 分支里，不要用 `b_v_new1_f32.to(p_v_new1.dtype.element_ty)` 这种“我随时现转现存”的写法，而是直接把我们刚才显式出来的 `b_v_new*_bf16` 存下去。也就是：

python

```
if SAVE_NEW_VALUE:
    t1.store(p_v_new1, b_v_new1_bf16.to(p_v_new1.dtype.element_ty), boundary_check=(0, 1))
```

```
tl.store(p_v_new2, b_v_new2_bf16.to(p_v_new2.dtype.element_ty), boundary_check=(0, 1))
```

这一步是个信号：从现在起，跨核/持久化用的就是 `_bf16` 版本，不是 `_f32` 版本。

后端看到这个 pattern，更有可能把 mailbox 里的那两块大 tile 用 `bf16` 去分配/回写，而不是 `float32`。

Part 2. 把 gating / exp 缩放 融进同一轮，避免“未 gating 的 f32 tile”落 GM

现在 IR 做了这件很浪费内存的事：

AIC 先算 $(v - w@h)$ ，结果是 $[BT, BV]$ `float32`。

它把这个 `float32` 整块写到 GM (mailbox)。

AIW 读回来，再乘 `safe_exp(b_g_last - b_g)` 做 gating 广播。

乘完 gating 后又 cast 成 `bf16`，写回 GM，AIC 再拿这份 `bf16` 做 $k^T @ \dots$ 。

我们想消掉步骤 2 这份“未 gating 的 f32 tile”的大块写。

思路：在 Triton 里把 gating 计算提到更前面，让 $(v - w@h)$ 立刻乘上 gating，再 cast 成 `bf16` 并立马用掉。

这样跨核出现的中间产物就直接是 `bf16` 的“gated `v_new`”，而不是 ungated 的 `float32` 版本。

具体怎么改？

我们只需要把 gating 的两条缩放因子提前算出来，然后在构造 `b_v_new*_f32` 的时候就直接乘上。

gating 因子有两个：

每个时间步的 `scale_row = exp(b_g_last - b_g)`，形状 $[BT]$ ，后来要 broadcast 成 $[BT, BV]$ 逐行乘到 $(v - w@h)$ 上。

对隐状态的全局缩放 `scale_state = exp(b_g_last)`，是单个标量（对这个 head 的 chunk 全部列都乘一样）。

在你的原始代码里：

python

```
if USE_G:
    last_idx = min((i_t + 1) * BT, T) - 1
    b_g_last = tl.load(g + bos + i_h * T_max + last_idx)
    p_g = tl.make_block_ptr(...)
    b_g = tl.load(p_g, boundary_check=(0,))
    b_v_new1 = b_v_new1 * safe_exp(b_g_last - b_g)[:, None]
    b_v_new2 = b_v_new2 * safe_exp(b_g_last - b_g)[:, None]
    b_g_last = exp(b_g_last)           # reusing the same var name, becomes scalar exp
    b_h1_bv1 = b_h1_bv1 * b_g_last
    b_h1_bv2 = b_h1_bv2 * b_g_last
```

我们把这段逻辑重排成“先得到缩放系数，再把缩放合进 temp 的生成过程”，像这样：

python

```
# 1. 预取 gating 因子
if USE_G:
    last_idx = min((i_t + 1) * BT, T) - 1
    g_last_scalar = tl.load(g + bos + i_h * T_max + last_idx)    # shape: scalar
    p_g = tl.make_block_ptr(g + bos + i_h * T_max, (T,), (1,), (i_t * BT,), (BT,), (0,))
    g_row = tl.load(p_g, boundary_check=(0,))                      # shape: [BT]

    scale_row = safe_exp(g_last_scalar - g_row)                  # [BT]
    scale_state = exp(g_last_scalar)                            # scalar

else:
    # no gating: all-ones
```

```
scale_row = tl.full([BT], 1.0, dtype=tl.float32)
scale_state = tl.full([], 1.0, dtype=tl.float32) # scalar
```

然后我们把后面 ($v - w@h$) 的时候直接融合 scale_row:

python

```
# 2. matmul 用 shadow 的 bf16 状态, 结果是 float32
prod1 = tl.dot(b_w, state_shadow_bv1) # [BT, BV] accum float32
prod2 = tl.dot(b_w, state_shadow_bv2)

# 3. 基础差值:  $v - w@h$  (还是 float32)
tmp1_f32 = b_v1.to(tl.float32) - prod1
tmp2_f32 = b_v2.to(tl.float32) - prod2

# 4. gating 融进来: 逐行乘 scale_row
# scale_row: [BT], 我们需要 [BT, 1] broadcast 到 [BT, BV]
tmp1_f32 = tmp1_f32 * scale_row[:, None]
tmp2_f32 = tmp2_f32 * scale_row[:, None]

# 5. 把“隐状态自身的缩放”先做在主状态上 (float32)
b_h1_bv1_f32 = b_h1_bv1_f32 * scale_state
b_h1_bv2_f32 = b_h1_bv2_f32 * scale_state

# (到这一步为止, 我们已经做完 gating 了, tmp*_f32 现在就是 b_v_new (after gate) 的 float32 版本,
# 再也没有“未 gating 的 f32 tile”。也就是说, AIC 不需要把未 gating tile 写到 GM。)
```

到这一步, 你已经满足“fuse gating”。我们把原来分两阶段的事合并成一个阶段:

先拿到 scale_row / scale_state

立刻把 scale_row 用在 ($v - w@h$) 上

再立刻 scale_state 进 $b_h1_bv*_f32$

这样, 对后端来说, “tmp1_f32 就是最终要喂给 K^T 的 gated 值”, 它可以更放心地在跨核通信处直接下采样成 bf16; 不需要先把 ungated 的 float32 结果丢到 GM 再回来乘 gate。

然后后续部分保持和 Part 1 的影子逻辑一致:

python

```
# 6. SAVE_NEW_VALUE / v_new 写出去的版本: downcast
b_v_new1_bf16 = tmp1_f32.to(k.dtype.element_ty)
b_v_new2_bf16 = tmp2_f32.to(k.dtype.element_ty)

if SAVE_NEW_VALUE:
    tl.store(p_v_new1, b_v_new1_bf16.to(p_v_new1.dtype.element_ty), boundary_check=(0,1))
    tl.store(p_v_new2, b_v_new2_bf16.to(p_v_new2.dtype.element_ty), boundary_check=(0,1))

# 7. 第二个 GEMM:  $K^T @ (gated v\_new)$ 
b_k = tl.load(p_k, boundary_check=(0, 1)) # [K, BT]
delta1_f32 = tl.dot(b_k, b_v_new1_bf16)      # -> [K, BV] float32
delta2_f32 = tl.dot(b_k, b_v_new2_bf16)

# 8. 累加到主状态
b_h1_bv1_f32 = b_h1_bv1_f32 + delta1_f32
b_h1_bv2_f32 = b_h1_bv2_f32 + delta2_f32

# 9. 更新 shadow (低精度) 给下一轮的 w@h 用
```

```
state_shadow_bv1 = b_h1_bv1_f32.to(k.dtype.element_ty)
state_shadow_bv2 = b_h1_bv2_f32.to(k.dtype.element_ty)
```

到这个版本为止，有两件关键的优化效果：

跨核的 mailbox 里，不再需要“未 gating 的 f32 tile”。

我们已经把 gating 在 matmul 后立刻做完了 ($\text{tmp}^*_\text{f32} = (\dots - \text{prod}) * \text{scale_row}[:, \text{None}]$)，所以 mailbox 只会看到“已经 gating 完成并且允许被 downcast 的版本”，即 $\text{b}_\text{v}_\text{new}^*_\text{bf16}$ 。

跨核传递隐状态时，也只需要 bf16 影子，而不是 float32 整块。

因为我们明确维护了 state_shadow_bv^* 作为下一轮 $w @ h$ 的输入；后端可以把这个 shadow 放到 mailbox，而不是 float32 主状态整块。

把这些整合成一个修改后循环骨架

下面是合在一起后的 `for i_t in range(NT):` 伪代码骨架。你可以直接对照你的原函数，把相应部分替换/重排。

python

```
# 事前：初始化
b_h1_bv1_f32 = tl.zeros([128, BV], dtype=tl.float32)
b_h1_bv2_f32 = tl.zeros([128, BV], dtype=tl.float32)

if USE_INITIAL_STATE:
    # Load initial_state -> float32
    # (和你原来做的一样)
    b_h1_bv1_f32 += tl.load(p_h0_1_bv1, boundary_check=(0,1)).to(tl.float32)
    b_h1_bv2_f32 += tl.load(p_h0_1_bv2, boundary_check=(0,1)).to(tl.float32)

# 初始 shadow (bf16 / same dtype as w/k)
state_shadow_bv1 = b_h1_bv1_f32.to(k.dtype.element_ty)
state_shadow_bv2 = b_h1_bv2_f32.to(k.dtype.element_ty)

for i_t in range(NT):
    # 0. 把“chunk起始状态快照”写到 h 里（保持你原语义）
    h_base = h + (boh + i_t) * H * K * V + i_h * K * V
    p_h1_bv1 = tl.make_block_ptr(h_base, (K, V), (V, 1),
                                  (0, v_start1), (128, BV), (1, 0))
    p_h1_bv2 = tl.make_block_ptr(h_base, (K, V), (V, 1),
                                  (0, v_start2), (128, BV), (1, 0))
    tl.store(p_h1_bv1, b_h1_bv1_f32.to(p_h1_bv1.dtype.element_ty), boundary_check=(0,1))
    tl.store(p_h1_bv2, b_h1_bv2_f32.to(p_h1_bv2.dtype.element_ty), boundary_check=(0,1))

    # 1. 读当前 chunk 的 v (两半) 和 w/k tiles
    v_base = v + bos * H * V + i_h * V
    p_v1 = tl.make_block_ptr(v_base, (T, V), (stride_v, 1),
                            (i_t * BT, v_start1), (BT, BV), (1, 0))
    p_v2 = tl.make_block_ptr(v_base, (T, V), (stride_v, 1),
                            (i_t * BT, v_start2), (BT, BV), (1, 0))
    b_v1 = tl.load(p_v1, boundary_check=(0,1))
    b_v2 = tl.load(p_v2, boundary_check=(0,1))

    w_base = w + bos * H * K + i_h * K
    p_w = tl.make_block_ptr(w_base, (T, K), (stride_w, 1),
                           (i_t * BT, 0), (BT, 128), (1, 0))
    b_w = tl.load(p_w, boundary_check=(0,1))
```

```

k_base = k + bos * Hg * K + (i_h // (H // Hg)) * K
p_k = tl.make_block_ptr(k_base, (K, T), (1, stride_k),
                        (0, i_t * BT), (128, BT), (0, 1))
b_k = tl.load(p_k, boundary_check=(0,1))

# 2. 预取 gating 因子
if USE_G:
    last_idx = min((i_t + 1) * BT, T) - 1
    g_last_scalar = tl.load(g + bos + i_h * T_max + last_idx) # scalar
    p_g = tl.make_block_ptr(g + bos + i_h * T_max, (T,), (1,),
                            (i_t * BT,), (BT,), (0,))
    g_row = tl.load(p_g, boundary_check=(0,))

    scale_row = safe_exp(g_last_scalar - g_row) # [BT]
    scale_state = exp(g_last_scalar)           # scalar
else:
    scale_row = tl.full([BT], 1.0, dtype=tl.float32)
    scale_state = tl.full([], 1.0, dtype=tl.float32)

# 3. (v - w @ h_prev) , 用 shadow (bf16) 喂 w@h, 结果是 float32
prod1 = tl.dot(b_w, state_shadow_bv1) # float32 [BT,BV]
prod2 = tl.dot(b_w, state_shadow_bv2)

tmp1_f32 = b_v1.to(tl.float32) - prod1
tmp2_f32 = b_v2.to(tl.float32) - prod2

# 4. 把 gating 融进 temp
tmp1_f32 = tmp1_f32 * scale_row[:, None] # broadcast [BT]->[BT,BV]
tmp2_f32 = tmp2_f32 * scale_row[:, None]

# 5. 状态先乘 scale_state (float32保持, 数值稳定)
b_h1_bv1_f32 = b_h1_bv1_f32 * scale_state
b_h1_bv2_f32 = b_h1_bv2_f32 * scale_state

# 6. SAVE_NEW_VALUE: 我们要写出去 / 跨核传递的版本是 bf16 (或 k.dtype)
tmp1_bf16 = tmp1_f32.to(k.dtype.element_ty)
tmp2_bf16 = tmp2_f32.to(k.dtype.element_ty)

if SAVE_NEW_VALUE:
    v_new_base = v_new + bos * H * V + i_h * V
    p_v_new1 = tl.make_block_ptr(v_new_base, (T, V), (stride_v, 1),
                                (i_t * BT, v_start1), (BT, BV), (1, 0))
    p_v_new2 = tl.make_block_ptr(v_new_base, (T, V), (stride_v, 1),
                                (i_t * BT, v_start2), (BT, BV), (1, 0))
    tl.store(p_v_new1, tmp1_bf16.to(p_v_new1.dtype.element_ty), boundary_check=(0,1))
    tl.store(p_v_new2, tmp2_bf16.to(p_v_new2.dtype.element_ty), boundary_check=(0,1))

# 7. 第二个 GEMM: k^T @ (已经 gating 且 downcast 了的 tmp)
delta1_f32 = tl.dot(b_k, tmp1_bf16) # float32 [K,BV]
delta2_f32 = tl.dot(b_k, tmp2_bf16)

b_h1_bv1_f32 = b_h1_bv1_f32 + delta1_f32
b_h1_bv2_f32 = b_h1_bv2_f32 + delta2_f32

```

```

# 8. 更新 shadow 供下一轮使用 (bf16 / k.dtype)
state_shadow_bv1 = b_h1_bv1_f32.to(k.dtype.element_ty)
state_shadow_bv2 = b_h1_bv2_f32.to(k.dtype.element_ty)

# 循环结束后写 final_state (和你原版逻辑一样)
if STORE_FINAL_STATE:
    ht_ptr = ht + i_nh * K * V
    p_ht1_bv1 = tl.make_block_ptr(ht_ptr, (K, V), (V, 1),
                                  (0, v_start1), (128, BV), (1, 0))
    p_ht1_bv2 = tl.make_block_ptr(ht_ptr, (K, V), (V, 1),
                                  (0, v_start2), (128, BV), (1, 0))
    tl.store(p_ht1_bv1, b_h1_bv1_f32.to(p_ht1_bv1.dtype.element_ty), boundary_check=(0,1))
    tl.store(p_ht1_bv2, b_h1_bv2_f32.to(p_ht1_bv2.dtype.element_ty), boundary_check=(0,1))

```

这两个改动实现了什么

1. 压缩 mailbox 精度

我们明确区分了 `b_h1_bv*_f32` (循环内部 float32 主状态, 只在本核里活) 和 `state_shadow_bv*` (bf16 影子, 用来喂 W@h 和可持久化到 GM)。

我们明确区分了 `tmp*_f32` (gated 之前/之后的 float32 版本, 只在本核活) 和 `tmp*_bf16` (准备跨核/写回的低精度版本)。

因为我们清楚地“只把 `*_bf16` 版本写出去 / 传下去”, 后端会更倾向用 bf16 来布置那块 144KB 的 workspace, 而不是用 float32 大砖头。

2. 融合 gating, 消掉“未 gating 的 f32 tile”外泄

我们把 `scale_row` 和 `scale_state` 提前, 直接在本轮里把 $(v - w@h)$ 乘上 gating。

所以跨核永远看不到“还没 gating 的大 float32 tile”。跨核只会看到 `tmp*_bf16` 这种已经 gating 好、并且降精度过过的版本。

这基本解决了 IR 里那块“ 64×64 float32 未 gating tile 先写 GM 再读回来乘 exp”的大额浪费。

小结

你需要做的核心代码层级改动可以记成三条规则:

显式维护两份隐状态:

`b_h1_bv*_f32`: 真正的累积状态 (float32), 只在 loop 内部继续用。

`state_shadow_bv*` = `b_h1_bv*_f32.to(k.dtype.element_ty)`: 专门准备用来做 $w @ h$ 的 bf16 影子, 也是唯一允许“泄露到 GM / 跨核通信”的版本。

先算 gating 因子, 再把 gating 直接融合进 $(v - w@h)$, 然后立刻 downcast 成 bf16 得到 `tmp*_bf16`, 并用它直接去做 $k^T @ (\dots)$ 。

这样, 中途就不需要把“未 gating 的 float32 大 tile”写出去了。

所有 `tl.store` 到全局内存 (`p_v_new*`, `p_h1_bv*` 等) 都用低精度版本 (bf16 / `k.dtype.element_ty`) , float32 版本只当成“寄存器里的活变量”, 从来不写出。

这样, NPU 后端即使还是必须通过 GM 来连 AIC/AIV, 也只会把 bf16 版本放进 mailbox, mailbox 直接缩小。

这样修改后, 数值路径基本保持原始数学:

$(v - w@h)$ 仍然在 float32 里做减法和 gating, 精度没大幅降;

$h *= \exp(b_g_{last})$ 仍然在 float32 上做;

$h += k^T @ (gated v_{new})$ 仍然是 float32 累加;

唯一真的长久存到 GM 的, 都是 bf16 版本 (影子, 或者中间 gated `v_new` 的副本), 这会把中间 workspace 那块显存/全局内存峰值明显压下去, 同时不牺牲主路径的 float32 稳定性。

Is this conversation helpful so far?   

