

Flash SwiGLU MLP：全流程协同优化与选择性算子融合

SwiGLU MLP 是现代大语言模型中计算量和显存开销占比较高的核心模块之一。目前的优化方案主要是对 SwiGLU 激活本身做融合，其余的矩阵乘部分依靠 PyTorch 原生实现（后端会映射到 cuBLAS 上），因为 SwiGLU 激活是 Element-wise 操作（逐元素乘法），容易触及访存瓶颈，融合具有明显收益。这方面的代表工作包括：

- PyTorch 原生的 torch.compile
- 开源高性能算子库（如：Liger kernel 的 LigerSwiGLUMLP）

但本文认为还可以更进一步，故提出 **Flash SwiGLU MLP**，本实现将不止于融合 SwiGLU 激活本身，更对 SwiGLU MLP 前向与反向进行全流程协同优化，充分考虑不同场景负载，设计专用方案。

我们先看性能与正确性测试，再讲具体方案。

性能测试

将本实现与 torch.compile 和 LigerSwiGLUMLP 对比，测试不同数据规模下的性能和显存占用。

对比基线的构建方式如下：

```
# Liger kernel
from liger_kernel.transformers.swiglu import LigerSwiGLUMLP
from transformers import PretrainedConfig

def make_liger_swiglu(dim, hidden_dim, device, dtype, gate_w=None, up_w=None, down_w=None):
    config = PretrainedConfig(
        hidden_size=dim, intermediate_size=hidden_dim, hidden_act="silu"
    )
    mlp = LigerSwiGLUMLP(config).to(device).to(dtype)
    if gate_w is not None:
        mlp.gate_proj.weight.data.copy_(gate_w)
    if up_w is not None:
        mlp.up_proj.weight.data.copy_(up_w)
    if down_w is not None:
        mlp.down_proj.weight.data.copy_(down_w)
    return mlp

# torch.compile
import torch.nn.functional as F

def swiglu_naive_forward(input, gate_weight, up_weight, down_weight):
    G = F.linear(input, gate_weight)
    U = F.linear(input, up_weight)
    A = F.silu(G) * U
    O = F.linear(A, down_weight)
    return O

compiled_swiglu_naive = torch.compile(
    swiglu_naive_forward, mode="max-autotune-no-cudagraphs"
)
```

选择 `mode="max-autotune-no-cudagraphs"` 作为 baseline 的原因见于附录。

硬件与软件环境：

```
GPU: RTX 5060ti 16G
Compute capability: SM120
Driver Version: 580.159.03
CUDA Version: 13.1
PyTorch Version: 2.13.0
Triton Version: 3.7.1
Liger Kernel Version: 0.8.1
```

测试形状：

维度设置 (D 表示模型隐藏层大小, H 表示前馈网络大小)：

D	H
1024	2816
2048	5632
4096	11008

针对推理和训练场景选择了不同的 Batchsize 和 序列长度 S：

```
Inference:
B = 1
S ∈ {1, 2, 4, 8, 16, 32, 64, 128, 256}
dtype ∈ {FP16, BF16}

Training:
B = 8
S ∈ {256, 512, 1024, 2048, 4096, 8192}
dtype ∈ {FP16, BF16}
```

注意：自回归推理中，每个请求一般每次处理 1 个 token，为提高推理效率，通常会使用 Continuous Batching，将多个请求在 S 维度拼接，所以这里的 “Inference” 中的 S 实际指的是同时处理的请求数量 (Num of Requests)

性能测试结果：

FP16, *BF16* 两种数据类型的性能相近，这里只展示 *BF16* 的测试结果，相关结论同样适用于 *FP16* 的情况。所有测试均 warmup 10 次，重复 50 次取均值。

性能对比图中统一用折线形状区分不同的实现：

圆点线：Flash swiglu（本实现）；

三角线：Liger kernel；

叉号线：Torch.compile。

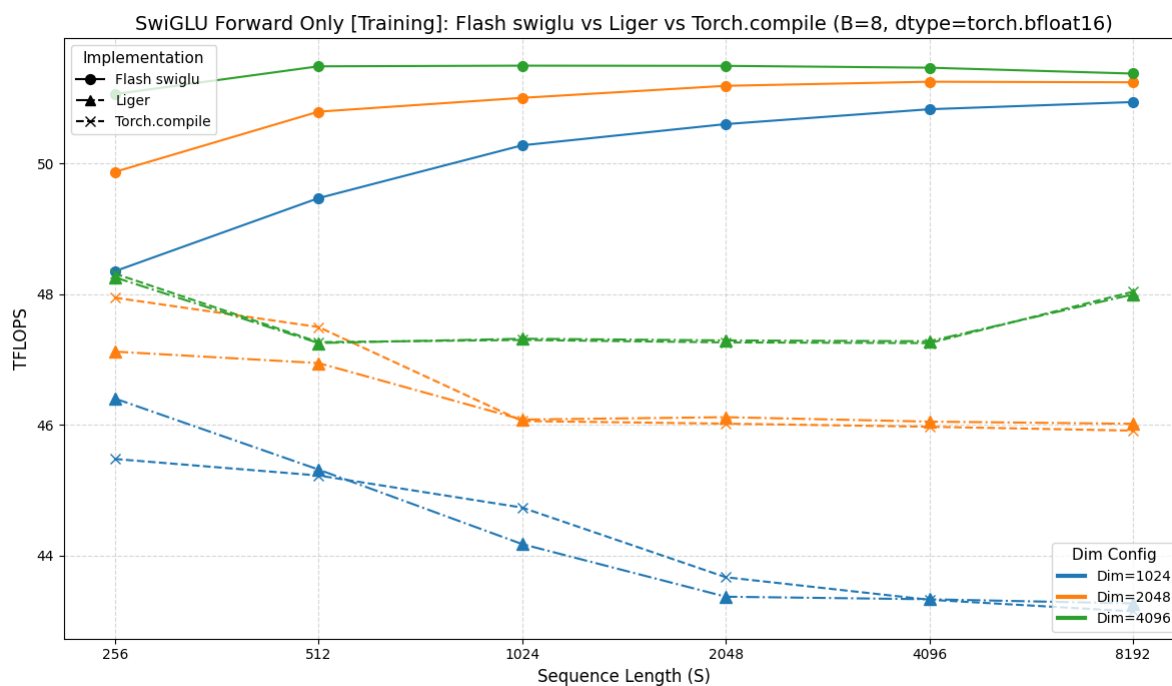
用折线颜色区分维度配置：

蓝色: D=1024, H=2816;

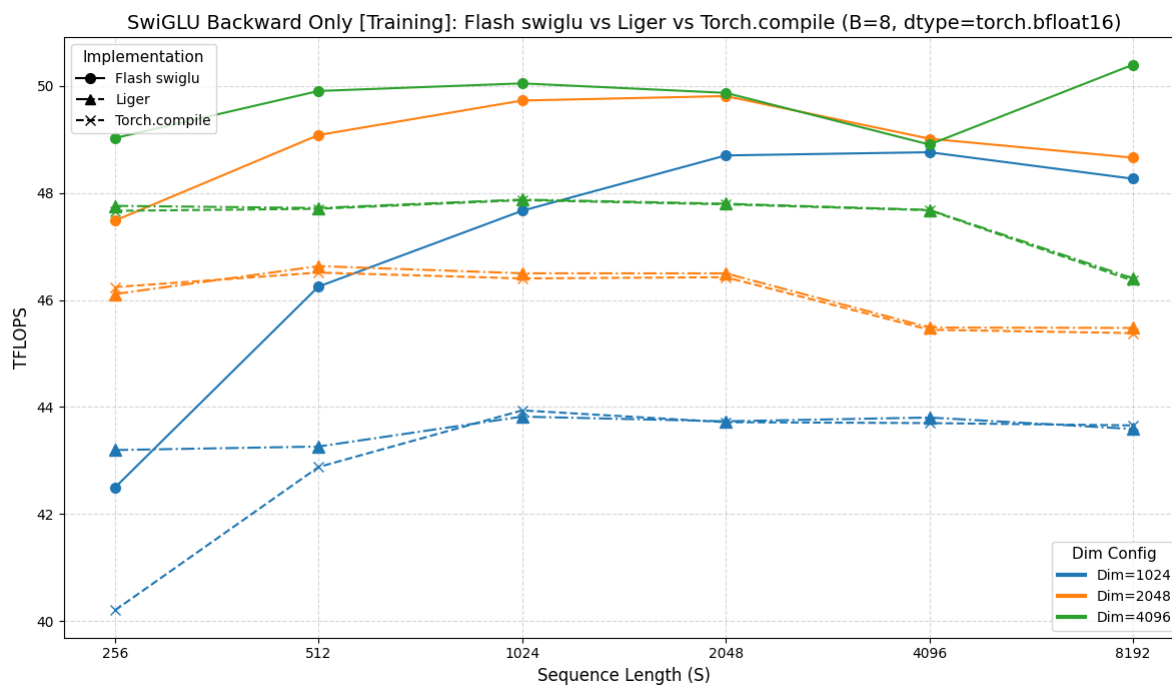
橙色: D=2048, H=5632;

绿色: D=4096, H=11008。

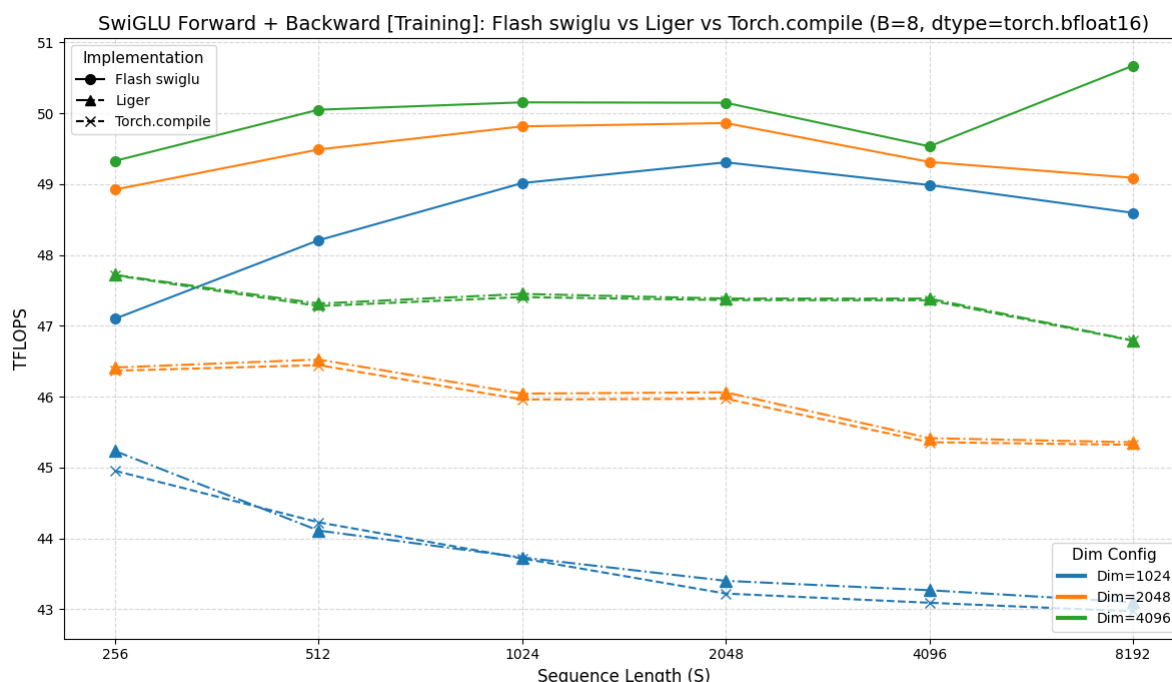
- 训练时的前向传播



- 训练时的反向传播

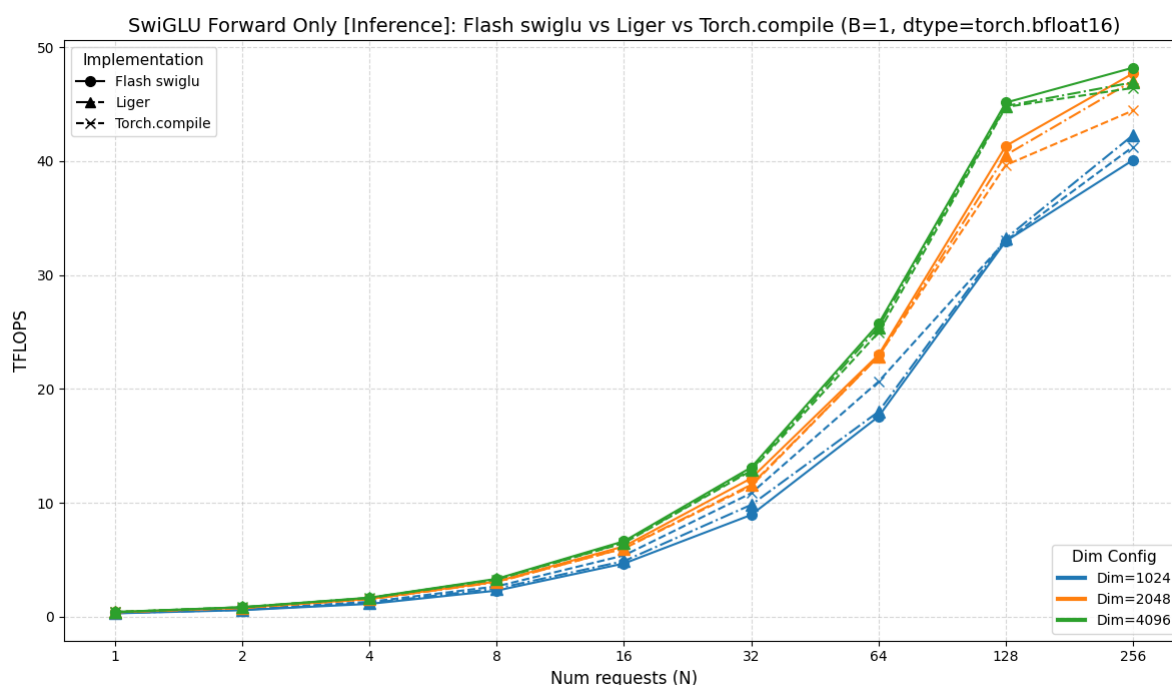


- 训练时端到端



总的来讲，在**当前测试环境**下，本实现在训练阶段的前向、反向以及端到端测试中性能全面优于另外两种实现，提升幅度集中在 5%~15% 之间，特别是在大维度（Dim=4096）和长序列（Seq Len > 2048）场景下，本实现的 TFLOPS 甚至逼近了当前设备在该数据精度下的实际算力上限。

- 推理场景

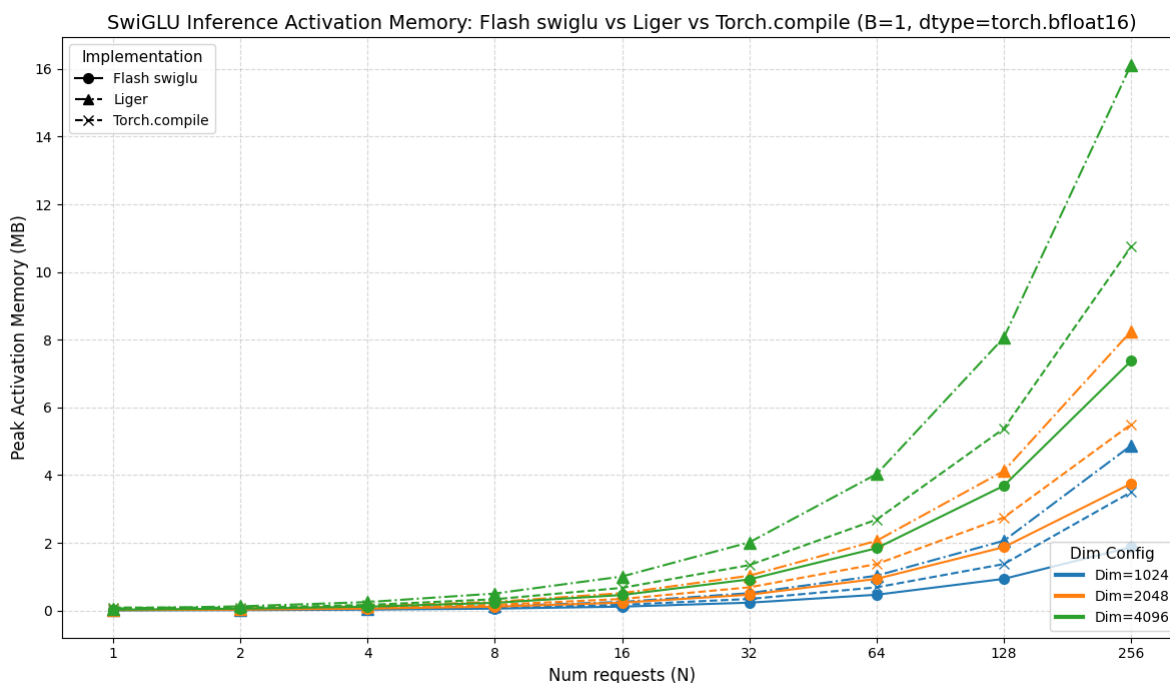


可以发现，在**小规模数据**（D=1024, H=2816）的测试中，本实现性能**略低**于另外两种实现，但在**更大数据规模**的测试中，三种实现性能相近，本实现**略优**。主要是因为本实现使用了 Triton 的 host-side TMA descriptor API，CPU 侧开销更重，**小规模数据下这部分开销无法被摊薄**，导致性能略低。真实的推理场景中，使用 CUDA graph 降低 CPU 侧开销后，本实现的优势才能完全体现。

峰值显存测试结果：

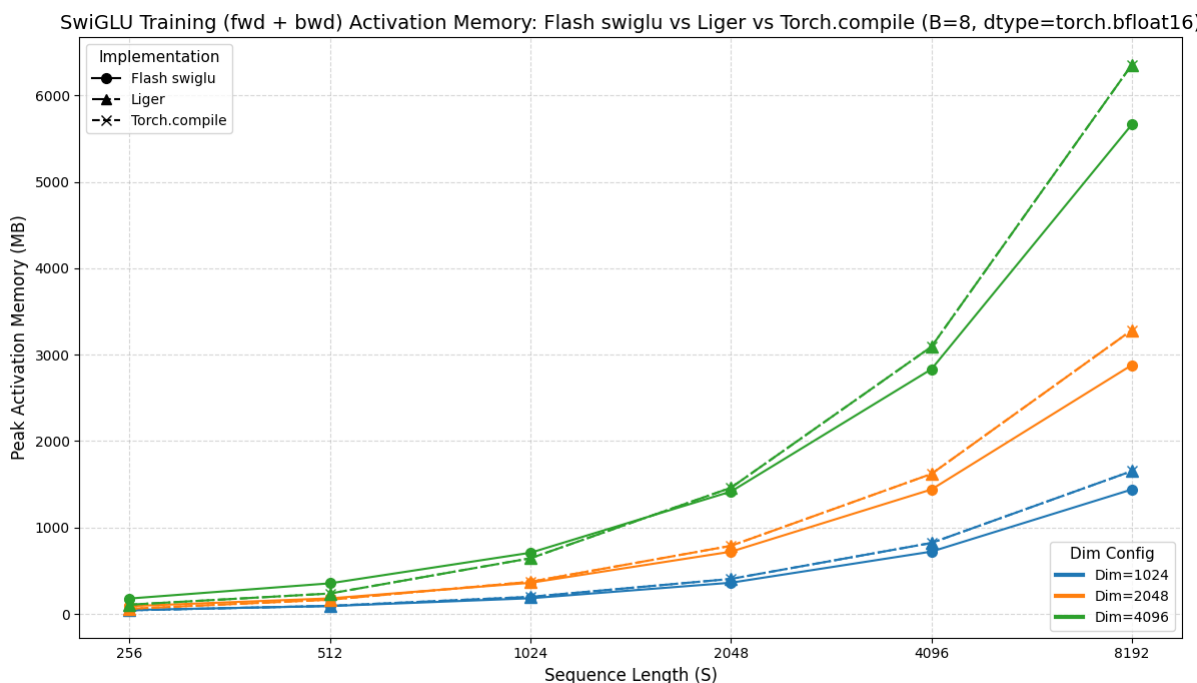
这里测试的是**峰值中间显存**开销（即扣除输入与权重及其梯度之外的峰值中间显存增量）的大小。目的是避免因权重显存占用太大，主导了测试结果，导致测试结果看不出差异。

- 推理时（纯前向传播）的显存占用



推理峰值中间显存在全部测试规模下全面优于 torch.compile 与 Liger Kernel，约为 Liger 的 38.8% ~ 46.9%，torch.compile 的 31.3% ~ 70.9%。

- 训练时（端到端）的显存占用



本实现在长序列场景下，峰值中间显存占用更低，但短序列场景占用反而更高，这主要是因为对比基线在不同数据规模下的行为差异（编译/内存规划策略不同），短序列场景使用了更激进的内存复用优化，显存占用更低。

正确性测试

以 torch.compile 的 naive 实现作为数值参考基准，在训练与推理两种模式、FP16/BF16 两种精度、小规模 (B=1, S=1024, D=128, H=512) 与大规模 (B=1, S=4096, D=4096, H=11008) 两种数据形状下，对前向输出 (O) 以及全部反向梯度 (dI 、 dW_g 、 dW_u 、 dW_d) 分别进行校验，并与 Liger Kernel 的对应结果并列对比。

✓ 表示通过 `torch.allclose(actual, expected, atol=..., rtol=...)`; Max Diff 仅作为附加统计量展示。

• FP16、训练模式：

数据规模	张量	Max Diff (Flash SwiGLU)	Max Diff (Liger)	阈值 (atol/rtol)
小	O	1.53e-05 ✓	1.53e-05 ✓	1e-3 / 1e-4
小	dl	2.44e-04 ✓	6.10e-04 ✓	1e-3 / 1e-4
小	dWg	2.44e-04 ✓	1.22e-04 ✓	1e-3 / 1e-4
小	dWu	2.44e-04 ✓	6.10e-05 ✓	1e-3 / 1e-4
小	dWd	4.88e-04 ✓	4.88e-04 ✓	1e-3 / 1e-4
大	O	4.77e-07 ✓	4.77e-07 ✓	1e-3 / 1e-4
大	dl	3.05e-05 ✓	3.05e-05 ✓	1e-3 / 1e-4
大	dWg	1.53e-05 ✓	1.53e-05 ✓	1e-3 / 1e-4
大	dWu	1.53e-05 ✓	1.53e-05 ✓	1e-3 / 1e-4
大	dWd	3.05e-05 ✓	3.05e-05 ✓	1e-3 / 1e-4

• FP16、推理模式：

数据规模	张量	Max Diff (Flash SwiGLU)	Max Diff (Liger)	阈值 (atol/rtol)
小	O	1.53e-05 ✓	1.53e-05 ✓	1e-3 / 1e-4
大	O	4.77e-04 ✓	4.77e-04 ✓	1e-3 / 1e-4

• BF16、训练模式：

数据规模	张量	Max Diff (Flash SwiGLU)	Max Diff (Liger)	阈值 (atol/rtol)
小	O	1.22e-04 ✓	1.22e-04 ✓	4e-3 / 1e-4
小	dl	1.95e-03 ✓	4.88e-04 ✓	4e-3 / 1e-4
小	dWg	1.95e-03 ✓	9.77e-04 ✓	4e-3 / 1e-4
小	dWu	1.95e-03 ✓	0.00e+00 ✓	4e-3 / 1e-4
小	dWd	3.91e-03 ✓	3.91e-03 ✓	4e-3 / 1e-4
大	O	3.81e-06 ✓	3.81e-06 ✓	4e-3 / 1e-4
大	dl	2.44e-04 ✓	2.44e-04 ✓	4e-3 / 1e-4
大	dWg	1.22e-04 ✓	1.22e-04 ✓	4e-3 / 1e-4
大	dWu	1.22e-04 ✓	0.00e+00 ✓	4e-3 / 1e-4

数据规模	张量	Max Diff (Flash SwiGLU)	Max Diff (Liger)	阈值 (atol/rtol)
大	dWd	2.44e-04	2.44e-04	4e-3 / 1e-4

• **BF16、推理模式：**

数据规模	张量	Max Diff (Flash SwiGLU)	Max Diff (Liger)	阈值 (atol/rtol)
小	O	1.22e-04	1.22e-04	4e-3 / 1e-4
大	O	3.81e-06	3.81e-06	4e-3 / 1e-4

结果显示：FP16 和 BF16 下所有对比在全部规模、全部场景下均通过。

BF16 整体精度余量变窄是因为：BF16 尾数只有 7 位（FP16 是 10 位），同样量级的数值在 BF16 下舍入误差天然更大，这是精度格式本身的特性而非实现缺陷。

接下来介绍具体实现方案。

1. 背景

一个标准的 SwiGLU MLP 包含三个线性投影：门控投影 `gate_proj`、上投影 `up_proj` 和下投影 `down_proj`。

其**前向传播**可以表示为：

$$G = I @ W_g^T$$

$$U = I @ W_u^T$$

$$A = \text{SiLU}(G) \odot U$$

$$O = A @ W_d^T$$

其中，输入 $I \in \mathbb{R}^{B \times S \times D}$ ，中间维度为 H ，输出 $O \in \mathbb{R}^{B \times S \times D}$ 。三个权重矩阵的形状分别为：

$$W_g, W_u \in \mathbb{R}^{H \times D}, \quad W_d \in \mathbb{R}^{D \times H}.$$

其**反向传播**可以表示为：

$$dW_d = dO^T @ A$$

$$dA = dO @ W_d$$

$$dU = dA \odot \text{SiLU}(G)$$

$$dG = dA \odot U \odot \text{SiLU}'(G) = dA \odot U \odot (\text{Sigmoid}(G) + \text{SiLU}(G) - \text{Sigmoid}(G) \odot \text{SiLU}(G))$$

$$dW_u = dU^T @ I$$

$$dW_g = dG^T @ I$$

$$dI = dG @ W_g + dU @ W_u$$

其中，输入 $dO \in \mathbb{R}^{B \times S \times D}$ ，输出 dI 、 dW_g 、 dW_u 和 dW_d 。这四个梯度的形状分别为：

$$dI \in \mathbb{R}^{B \times S \times D}, \quad dW_g, dW_u \in \mathbb{R}^{H \times D}, \quad dW_d \in \mathbb{R}^{D \times H}.$$

原始 PyTorch 实现依次执行两个输入投影、SiLU 与逐元素门控，以及一个输出投影。前向传播共包含三个 GEMM，反向传播则包含六个 GEMM，并产生多个尺寸为 $B \times S \times H$ 的中间张量。

对于输入形状

$$I : [B, S, D],$$

以及中间维度 H ，标准 SwiGLU 的前向 FLOPs 为：

$$3 \times (2 \times B \times S \times H \times D),$$

反向 FLOPs 为：

$$6 \times (2 \times B \times S \times H \times D).$$

因此，一次完整训练前向与反向的主要计算量来自九个等价规模的矩阵乘。

这意味着，SwiGLU MLP 的优化不能只关注 SiLU 或逐元素乘法。真正需要解决的是：

- 如何减少 GEMM 周围的中间张量读写 —— kernel 融合；
- 如何降低 kernel launch 数量 —— kernel 融合；
- 如何提高较小 GEMM 的计算效率 —— 几个小的拼成大的；
- 如何避免为了融合而引入过高的寄存器压力 —— 改变累加器使用方式；
- 如何在训练、prefill 和 decoding 三种负载下选择不同策略。

本文向大家介绍 **Flash SwiGLU MLP**：该方案并不试图用单个超大 kernel 替代全部计算，而是以完整的前向和反向数据流为优化对象，重点优化以下环节：

- 前向传播融合 G 、 U 和 A 的计算；
- 将 G 和 U 存入同一块连续显存（物理上拼接成一个张量，为反向传播的大 GEMM 做准备）；
- 反向阶段原地使用 dG, dU 覆写 G, U ，降低显存占用；
- 基于合并的 G 和 U ，将 dW_g 和 dW_u 的计算合并为单次更大的 GEMM；
- 融合 dI 的两个矩阵乘法与逐元素加法，在寄存器中只使用单个累加器，降低寄存器开销；
- 为纯推理路径提供不保存 G, U 的专用前向 kernel。

Flash SwiGLU MLP 的核心思想是：

不盲目融合所有操作，而是充分权衡融合成本，围绕张量生命周期、计算数据流与显存开销进行全流程协同优化。

2. 参考实现

原始前向传播和反向传播实现为：

```
import torch.nn.functional as F

def swiglu_naive_forward(input, gate_weight, up_weight, down_weight):
    G = F.linear(input, gate_weight) # [B, S, H]
    U = F.linear(input, up_weight)   # [B, S, H]
    A = F.silu(G) * U                 # [B, S, H]
    O = F.linear(A, down_weight)      # [B, S, D]
```

```

return 0

def swiglu_naive_backward(dO, input, gate_weight, up_weight, down_weight, A, G, U):
    D = input.shape[-1]
    H = gate_weight.shape[0]
    sigmoid_G = F.sigmoid(G)
    silu_G = F.silu(G)

    # 1. [B*S, D].T @ [B*S, H] = [D, H]
    dwd = dO.reshape(-1, D).T @ A.reshape(-1, H)
    # 2. [B, S, D] @ [D, H] = [B, S, H]
    dA = dO @ down_weight
    # 3. [B, S, H] * [B, S, H] = [B, S, H]
    dU = dA * silu_G
    # 4. [B, S, H] * ... = [B, S, H]
    dG = dA * U * (sigmoid_G + silu_G - sigmoid_G * silu_G)
    # 5. [B*S, H].T @ [B*S, D] = [H, D]
    dwu = dU.reshape(-1, H).T @ input.reshape(-1, D)
    # 6. [B*S, H].T @ [B*S, D] = [H, D]
    dwg = dG.reshape(-1, H).T @ input.reshape(-1, D)
    # 7. [B, S, H] @ [H, D] + ... @ ... = [B, S, D]
    dI = dG @ gate_weight + dU @ up_weight

    return dI, dwg, dwu, dwd

```

可以将反向传播划分为四个逻辑阶段：

- 计算 dW_d
- 计算 dA, dG, dU
- 计算 dW_g, dW_u
- 最后计算 dI 。

如果每一个公式都独立执行，则反向传播除了六次 GEMM 外，还需要：

- 为 dA 分配一个 $B \times S \times H$ 张量；
- 分别为 dG 和 dU 分配两个同尺寸张量；
- 多次读写 (G,U,dA,dG,dU)；
- 执行逐元素操作；
- 在最后阶段对生成的两个矩阵乘结果执行一次加法。

这些中间操作的 FLOPs 相对 GEMM 较少，但会引入额外的访存、显存分配和 kernel launch 开销。

3. Flash SwiGLU MLP 总体设计

3.1 前向传播



前向 Triton kernel 融合这三步：

$$G = I @ W_g^T$$

$$U = I @ W_u^T$$

$$A = \text{SiLU}(G) \odot U$$

输出投影仍调用 `F.linear`：

$$O = A @ W_d^T$$

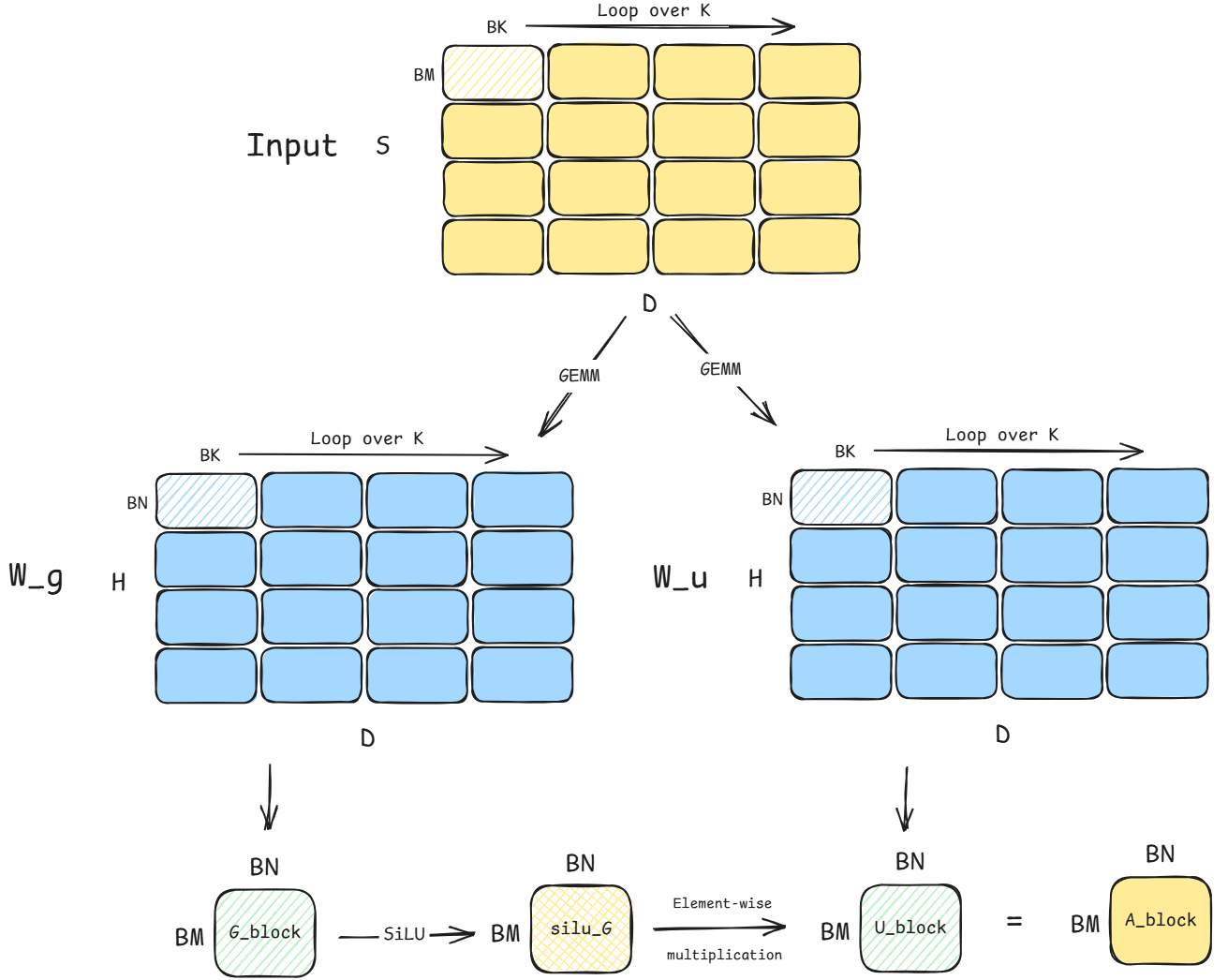
这里选择保留 `down_proj` 为独立 GEMM，而将 `gate_proj + up_proj + SiLU + gating` 融合进一个 Triton kernel。

3.2 反向传播



这种划分保留了大规模 GEMM 的高性能实现，同时融合了最容易产生访存瓶颈的中间操作。

4. 前向融合设计



4.1 融合两个输入投影

在每个 Triton program 中，加载一个输入块：

$$I_{tile} \in \mathbb{R}^{B_M \times B_K}$$

并分别加载对应的门控权重块和上投影权重块：

$$W_{g,tile}, W_{u,tile} \in \mathbb{R}^{B_N \times B_K}$$

内核维护两个 FP32 累加器（即上图中的 G_block 和 U_block ）：

$$acc_g \in \mathbb{R}^{B_M \times B_N}$$

$$acc_u \in \mathbb{R}^{B_M \times B_N}$$

在同一个 K 循环中依次执行：

$$acc_g += I_{tile} @ W_{g,tile}^T$$

$$acc_u += I_{tile} @ W_{u,tile}^T$$

如此，就在同一个归约循环中完成 gate 和 up 两个投影。

输入块在两个矩阵乘之间被复用。与两个完全独立的 GEMM 相比，该设计可以减少输入数据的重复加载，并减少一次矩阵乘的 kernel launch。

4.2 进一步融合 SiLU 激活与逐元素乘法

两个投影完成后，两个累加器（ acc_g 和 acc_u ）可以直接在寄存器中计算：

$$A_{\text{tile}} = \text{SiLU}(\text{acc}_g) \odot \text{acc}_u$$

即：

$$A = \text{SiLU}(G) \odot U$$

计算完成后写回 A 。需要注意的是，在训练阶段，我们还需要保存 G 和 U 供反向传播使用，所以这里的 acc_g 和 acc_u 需要写回全局内存。但在推理阶段，则完全不用管 G 和 U ，所以这种融合方式在推理时，对显存更友好。

将 gate 和 up 两个投影与 SiLU 激活和逐元素乘法融合的这种方式，避免了独立执行：

- gate GEMM；
- up GEMM；
- SiLU；
- 逐元素乘法。

使得前向传播只需一个融合 kernel 和一个输出投影 GEMM。大幅减少中间变量反复物化带来的访存压力，也避免了输入数据反复加载带来的重复访存。

4.3 训练阶段为什么仍然保存 G 和 U

因为反向传播需要同时使用 G 和 U ：

$$dU = dA \odot \text{SiLU}(G)$$

$$dG = dA \odot U \odot \text{SiLU}'(G) = dA \odot U \odot (\text{Sigmoid}(G) + \text{SiLU}(G) - \text{Sigmoid}(G) \odot \text{SiLU}(G))$$

因此，训练前向需要在“保存 G, U ”与“反向时重新计算 G, U ”之间做出选择，也就是经典的“计算与显存” trade-off。

当前实现选择保存 G, U ，原因包括：

- 反向重计算需要重新执行 gate 和 up 两次输入投影，增加的计算量相对比较大（经过测试，重计算的策略会慢很多）；
- G, U 的大小不像 attention 计算中的 QK 分数矩阵那样是 $O(S^2)$ 的，即使存储 G, U ，也不容易显存爆炸；
- 另外，后续还可以复用 G, U 的存储空间保存 dG, dU ，显存压力就更小了。

4.3.1 小巧思

这里有一个值得一提的小巧思：代码中使用一个形状为

$$[B, S, 3H]$$

的连续张量 `AGU` 来保存 A 和 G, U 这三个张量：

```
# Optimization: merge 3 torch.empty calls into 1 to reduce allocator overhead.
AGU = torch.empty(B, S, 3 * hidden_dim, device=input.device, dtype=input.dtype)
# Slice out 3 independent 3D views.
A = AGU[..., :hidden_dim]
G = AGU[..., hidden_dim : 2 * hidden_dim]
U = AGU[..., 2 * hidden_dim :]
```

主要有三个优势：

- 只需要调用一次 `torch.empty`，可以减少显存分配的开销；
- 这是大块显存空间的分配与释放，可以缓解显存碎片化。

第三点放到第五节介绍，大家可以先思考一下。

提示：反向传播时能提高计算效率

4.4 为什么不把 `down_proj` 也融合进来

如果大家了解 FlashAttention（不了解的话可以参考这个[从零实现 FlashAttention 系列教程](#)），就很容易想到：SwiGLU MLP 在两个投影矩阵之后进行一些逐元素操作，再接一个矩阵乘得到输出，这个计算流程就很像 attention 计算中先 QK 矩阵乘，再 softmax，最后再和 V 做矩阵乘法得到输出。所以也就可以借鉴 FlashAttention 的流式计算思路。

于是计算流程变成：

- Input 的每个 $I_{\text{tile}} \in \mathbb{R}^{B_M \times K}$ 沿 N 维度遍历 W_g 和 W_u ，每次与 $W_{g,\text{tile}}, W_{u,\text{tile}} \in \mathbb{R}^{B_N \times K}$ 做矩阵乘，得到 $G_{\text{block}}, U_{\text{block}} \in \mathbb{R}^{B_M \times B_N}$
- $A_{\text{block}} = \text{SiLU}(G_{\text{block}}) \odot U_{\text{block}}, A_{\text{block}} \in \mathbb{R}^{B_M \times B_N}$
- 每个 A_{block} 与对应的 W_d 的 $d_{\text{block}} \in \mathbb{R}^{B_N \times K}$ 做矩阵乘
- 将结果累加到 $acc \in \mathbb{R}^{B_M \times K}$ ，当沿 N 维度遍历完 W_g 和 W_u ， acc 即为最终输出 O 的一个分块

看上去很合理，好像真的可以用单个算子计算 SwiGLU MLP，但当你真的这样做后，你会发现算子奇慢无比。原因就在 $acc \in \mathbb{R}^{B_M \times K}$ ，这里的 K 对应的是 Input 维度的最后一维 D，常用规模是几千，如果把这么大的累加器放进寄存器，必然引起寄存器溢出。所以，直接放完整的 D 不可行。

那大家可能会想，既然完整的 D 放不下，我把 D 切块，放一部分不就行了吗？单从 GPU 寄存器和共享内存容量的角度来看，确实可以。但大家想一想，假如我们把 D 维度拆成了 10 份，把 10 份分给 10 个线程块计算，每个线程块都要算一遍 A_{block} ，总共就需要算 10 遍，如果 D 更大一些，需要切 100 份呢？重复计算的量太大，完全不划算！

或者，不把这 10 份分给 10 个线程块，而是让一个线程块算，把 $acc \in \mathbb{R}^{B_M \times K}$ 放到全局内存里，每次只加载 $B_M \times B_K$ 大小的一块进寄存器，这样寄存器容量放得下，也不需要重复计算，是不是就完美了？并没有！因为这样就需要反复的做全局内存的访问，减少了重复计算，换来了重复访存，依然不划算。

所以，想要通过一个 kernel 就算完整整个 SwiGLU MLP，不可行！

那为什么 FlashAttention 可以这样算呢？

因为 FlashAttention 的 D 维度是注意力头维度，常用大小是 128，GPU 的寄存器装得下，所以才可行。

5. 反向传播的四阶段设计

5.1 阶段一：使用 PyTorch GEMM 计算 dW_d

$$dW_d = dO^T @ A$$

实现将 batch 和 sequence 维展平：

```
dwd = dO.reshape(-1, dim).T @ A.reshape(-1, H)
```

即：

$$[D, BS] \times [BS, H] \rightarrow [D, H]$$

这一阶段没有额外的逐元素融合机会，而且 GEMM 通常足够大，所以交给 PyTorch 后端执行就能获得不错的性能，这样可以利用 cuBLAS 或 PyTorch 在当前平台上选择的高性能矩阵乘实现。

5.2 阶段二：融合 dA 、 dG 和 dU

原始实现首先计算：

$$dA = dO @ W_d$$

然后分别计算：

$$dU = dA \odot \text{SiLU}(G)$$

$$dG = dA \odot U \odot (\text{Sigmoid}(G) + \text{SiLU}(G) - \text{Sigmoid}(G) \odot \text{SiLU}(G))$$

Flash SwiGLU MLP 在单个 Triton kernel 内完成这三个步骤：

内核首先通过矩阵乘计算 dA_{tile} ：

$$dA_{\text{tile}} = dO_{\text{tile}} @ W_{d,\text{tile}}$$

然后加载前向保存的 G_{tile} 和 U_{tile} ，计算：

$$\text{Sigmoid}(G_{\text{tile}}) = \sigma(G_{\text{tile}})$$

$$\text{SiLU}(G_{\text{tile}}) = G_{\text{tile}} \odot \text{Sigmoid}(G_{\text{tile}})$$

$$dU_{\text{tile}} = dA_{\text{tile}} \odot \text{SiLU}(G_{\text{tile}})$$

$$dG_{\text{tile}} = dA_{\text{tile}} \odot U_{\text{tile}} \odot (\text{Sigmoid}(G_{\text{tile}}) + \text{SiLU}(G_{\text{tile}}) - \text{Sigmoid}(G_{\text{tile}}) \odot \text{SiLU}(G_{\text{tile}})).$$

融合内核的代码在同一 program 内完成 dA GEMM 和 dG, dU 两个梯度的逐元素计算，需要注意 dA_{tile} 一直位于寄存器中，不需要物化，所以完整的 dA 张量从未写入显存。这就完全消除了 dA 的全局显存分配、写回和重新读取的开销。

5.2.1 内存优化：原地覆写 G, U

观察反向传播公式可以发现，前向传播保存的 G, U 在计算出 dG, dU 后就不再被后续阶段使用。因此，它们与 dG, dU 的生命周期互斥，再加上这部分操作是逐元素乘法，元素之间没有交互。所以 Flash SwiGLU MLP 直接执行：

```
dGU = GU
```

也就是让 dG, dU 原地覆写 G, U 。该内存复用在实现中是显式设计，而非依赖编译器行为。

相较于分别重新分配 dG 和 dU ，该策略减少了：

$$2 \times BSH \times \text{sizeof}(\text{dtype})$$

的额外激活显存。

需要注意的是，本项目的源码：`flash-swiglu-mlp/src/flash_swiglu_mlp/kernel.py` 的 `_swiglu_kernel_backward_dGU` 函数注释明确指出，自动调优过程中的重复运行可能破坏原地覆写所依赖的数据，因此，反向 (dGU) kernel 不独立执行 autotune，而是复用前向选出的 tile 配置。

为什么可以复用前向选出的 tile 配置？

因为该 kernel 的矩阵乘部分 ($dA = dO @ W_d$) 和前向传播的矩阵乘形状是一样的，所以前向的最优配置大概率在 dGU kernel 上也能有好的表现。

但是，真正最优的写法是针对每一个 triton kernel 在当前设备上单独进行调优，找出不同负载下的最优配置，然后用 `@triton.heuristics` 而非 `@triton.autotune`，这样可以保证每一个算子都能使用其最优配置，且避免 autotune 带来了负面影响。

我的实现使用 `@triton.autotune` 是为了兼容性做的妥协，不是最优写法。

另外，需要提醒大家：原地覆写是用**牺牲高级求导能力**来换取内存的优化，但仅做单次一阶反向传播的常规训练（90% 都是这种）对此无感，只是会影响需要多次反向（`retain_graph=True`）或高阶求导（`create_graph=True`）的场景，如元学习、HVP/Hessian 计算、梯度惩罚等。

5.3 阶段三：一次 GEMM 同时计算 dW_g 和 dW_u

$$dW_u = dU^T @ I$$

$$dW_g = dG^T @ I$$

如果分别调用两个 GEMM，产生两次 kernel launch 开销的同时，两次操作都会读取同一个输入 I ，造成重复访存。

大家还记得 4.3 节提到的那个巧思吗？就是用在这里的！

我们从一开始就将 G, U 存储在同一个连续张量中，经过原地覆盖后，可以将 `dGU` 视为：

$$dGU = \text{concat}(dG, dU) \in \mathbb{R}^{B \times S \times 2H}$$

于是只需要调用一次 GEMM：

```
dwug = dGU.reshape(-1, 2 * H).T @ input.reshape(-1, D)
```

随后使用视图将结果拆分就得到了 dW_g 和 dW_u ：

```
dwg = dwug[:,H]
dwu = dwug[H,:]
```

如果前向传播中没有将 GU 放在同一块内存里，那么这里就要手动进行一次拼接，会涉及到新内存分配与复制，性能大概率还不如直接两次 GEMM。

5.4 阶段四：单累加器计算 dI

输入梯度为：

$$dI = dG @ W_g + dU @ W_u$$

一种直接实现方式是：

```
dI_g = dG @ wg
dI_u = dU @ wu
dI = dI_g + dI_u
```

该方案需要：

- 两次 GEMM；
- 两个 $B \times S \times D$ 中间张量；
- 一个逐元素加法 kernel；
- 对两个中间结果的额外 HBM 读写。

Flash SwiGLU MLP 使用一个融合 Triton kernel，并只创建一个 FP32 accumulator (acc)：

在矩阵乘的 K 循环中，对每个 K tile 中依次执行：

$$acc += dG_{\text{tile}} @ W_{g,\text{tile}},$$

$$acc += dU_{\text{tile}} @ W_{u,\text{tile}}.$$

代码中两个 `tl.dot` 显式共享同一个 `acc`，最后：

$$dI = acc$$

5.4.1 单累加器的意义

假设分别为两个矩阵乘维护 accumulator：

$$acc_g, acc_u \in \mathbb{R}^{B_M \times B_N},$$

则 accumulator 的寄存器需求是当前方案的两倍，会明显增加寄存器压力，而且最后还需要 $acc_g + acc_u$ ，多一次逐元素加法。

当前方案只维护：

$$acc \in \mathbb{R}^{B_M \times B_N},$$

并将两条归约路径依次累加到其中。这样能够：

- 减少 accumulator 的数量；

- 降低寄存器压力;
- 消除 $\text{acc}_g + \text{acc}_u$ 这一逐元素求和操作。

6. Tile 调度与 L2 Swizzle

本算子的每个 kernel 都可以使用了 L2 Swizzle:

```
@triton.jit
def _l2_swizzle(
    pid,
    M,
    N,
    BLOCK_M: tl.constexpr,
    BLOCK_N: tl.constexpr,
    GROUP_SIZE_M: tl.constexpr
):
    num_pid_m = tl.cdiv(M, BLOCK_M)
    num_pid_n = tl.cdiv(N, BLOCK_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    base_pid_m = group_id * GROUP_SIZE_M
    effective_group_size_m = tl.minimum(num_pid_m - base_pid_m, GROUP_SIZE_M)
    pid_in_group = pid % num_pid_in_group
    pid_m = pid_in_group % effective_group_size_m + base_pid_m
    pid_n = pid_in_group // effective_group_size_m
    return pid_m, pid_n
```

kernel 使用二维逻辑 tile:

$$[B_M, B_N].$$

program ID 首先经过 L2 swizzle, 转换为:

$$(pid_m, pid_n).$$

实现将多个 M tile 组成一个 group, 再在 group 内调整 (M,N) 的遍历顺序。该策略的目标是让相邻 program 更有可能复用同一部分数据, 提升 L2 cache locality。

`GROUP_SIZE_M=1` 等价于不执行 L2 swizzle。将 1 添加到 autotune 的配置选择中, 允许算子自己选择是否使用 L2 swizzle。

7. Autotune

本实现自动调优的 key 包括: "dim", "hidden_dim", "bucket_M", 即 `D, H, bucket_M`。

其中 `bucket_M` 表示对矩阵乘的 M 维度 (`batch_size * sequence length`) 进行分桶。

```
def get_bucket_m(M, threshold=4096, bucket_size=128):
    """
    Returns the globally unique bucket index mapped to  $M = B * S$ .
    - [0, 256) : Fine-grained bucketing with a granularity of 32, occupying buckets 0
    to 7.
```

```

- [256, threshold): Coarse-grained bucketing with a granularity of 128.
- >= threshold    : All fall into the last fixed large bucket.
"""

# The first 256 lengths are bucketed with a granularity of 32, consuming 256 // 32 = 8
buckets.
offset = 256 // 32

if M >= threshold:
    # All extra-long sequences are placed into the final bucket: 8 + 4096//128 = 40
    return offset + threshold // bucket_size
elif M < 256:
    # Fine-grained range.
    return M // 32

# Coarse-grained range, must add the previous offset to avoid index conflicts with the
[0, 256) range.
return offset + M // bucket_size

bucket_M = get_bucket_m(B*S)

```

值得一提的是：

- 由于矩阵乘的 M 维度直接影响参与计算的数据规模，所以必须加入自动调优的 key，但也不能让每一个具体的 M 都触发一轮新的 autotune，否则代价太大，因此我使用 `get_bucket_m` 函数对 M 进行分桶， $M = B * S$ ；
- 另外，当 M 维度达到某个阈值后，继续增加长度并不会再提高 GPU 的利用率，只是成比例的增加计算时间，此时的最优配置不会再变，所以当超过某个阈值后，就不应再触发自动调优。于是，我在 `get_bucket_m` 中加入了 `threshold`；
- 对于 M 维度极小的场景（如推理时的 decoding 阶段，用 continuous batching 把多个并发请求在 S 维度拼接， $B=1$ ），使用 `bucket_size=128` 显然是不合适的，需要更细的粒度，所以 `get_bucket_m` 额外增加一个分支，当 $M < 256$ 时，`bucket_size=32`。
- **注意：**这里的 `threshold=4096` 和当 $M < 256$ 时，`bucket_size=32` 这些数据只是基于当前设备的**经验值**，不是普遍规律。

8. 训练模式与推理模式分离

8.1 训练模式

在输入存在梯度需求（即需要进行反向传播，属于训练模式）时调用训练路径，并通过 `ctx.save_for_backward` 保存反向所需张量。

```

class _SwiGLUFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, gate_weight, up_weight, down_weight):
        assert (
            input.is_cuda
            and gate_weight.is_cuda
            and up_weight.is_cuda
            and down_weight.is_cuda

```

```

    )
    # Note:
    # PyTorch automatically disables global gradient computation during
    # the forward pass, so torch.is_grad_enabled() cannot be used.
    if any(ctx.needs_input_grad):
        0, AGU, fwd_best_config = swiglu_forward(input, gate_weight, up_weight,
down_weight)
        ctx.save_for_backward(input, gate_weight, up_weight, down_weight, AGU)
        ctx.fwd_best_config = fwd_best_config
        return 0
    else:
        # Inference mode: no GU saving.
        return swiglu_forward_inference(input, gate_weight, up_weight, down_weight)

@staticmethod
def backward(ctx, do):
    input, gate_weight, up_weight, down_weight, AGU = ctx.saved_tensors
    dI, dwg, dwu, dwd = swiglu_backward(
        do, input, gate_weight, up_weight, down_weight, AGU, ctx.fwd_best_config
    )
    return dI, dwg, dwu, dwd

```

8.2 推理模式

推理过程中不需要反向传播，所以不需要保存 G, U 。为此，单独开发了一个推理专用 kernel，只生成：

$$A = \text{SiLU}(G) \odot U$$

随后调用：

$$O = A @ W_d^T$$

由于不写回 G, U ，只写回 A ，对应的调用路径只需分配一个形状为 $[B, S, H]$ 的 A ，不再分配训练使用的形状为 $[B, S, 3H]$ 的 AGU 。

当模型处于推理模式时，本算子会自动选择该推理路径。

9. 不同负载场景的性能差异

训练场景在前面的性能测试中已经看到，本实现取得了全面领先的成绩。所以强烈推荐在训练中使用 Flash SwiGLU MLP。

下面详细讨论一下本实现在推理中的应用，大语言模型推理通常可以划分为 Prefill 和 Decoding 两个负载特征明显不同的阶段。

9.1 Prefill

在常见的 **Chunked Prefill** 模式下，一次处理的 prompt 通常具有相对较大的 sequence length，这带来了：

- 较大的矩阵乘 M 维度： $M = B \times S$ ；
- 更高的 GPU 并行度；
- 计算占主要成本，kernel launch 等 CPU 侧固定开销占比较低。

在这种情况下，Flash SwiGLU MLP 的前向融合能够充分发挥以下优势：

- 输入在 gate/up 两个矩阵乘中复用，避免重复访存；
- G, U 矩阵不物化，留在寄存器中直接做 swiglu 激活，减少访存开销；
- SiLU 激活和门控 ($\text{SiLU}(G) \odot U$) 不产生独立 kernel，避免额外 kernel launch 开销。

因此，**prefill 是当前 kernel 在推理任务中的主要推荐场景。**

9.2 Decoding

自回归 decoding 通常每次只处理一个或少量 token，即使使用 continuous batching，受限于并发请求数，等效的 S 仍然较小。

此时数据规模较小，导致：

- kernel launch、TensorDescriptor 创建等 CPU 侧开销占比较大；
- 并行度不足，GPU 利用率不高。

在当前测试覆盖的数据规模下，虽然 Flash SwiGLU MLP 在部分场景仍略优于 Liger kernel 和 torch.compile 这两个基线，但相比于 PyTorch 更成熟的优化，尤其得益于其底层基于 C++ 开发的调度机制，`torch.compile()` 仍然是更稳健的选择。

所以，除非能使用 CUDA graph 降低 CPU 侧开销，否则在 decoding 场景，更推荐使用原生的 torch.compile 实现。

10. 分布式训练兼容性

Flash SwiGLU MLP 目前的实现与验证均基于单卡场景，尚未针对分布式训练做专门优化，但在设计上并不与主流分布式并行策略冲突，具体体现在两方面。

10.1 与数据并行（DDP/ZeRO）天然兼容

本实现通过标准的 `torch.autograd.Function` 接入 PyTorch autograd 体系：`backward` 返回的 `dwg`、`dwu`、`dwd` 会正常累加到 `gate_weight`、`up_weight`、`down_weight` 各自的 `.grad` 上，与原生 `nn.Linear` 的梯度累加行为完全一致。

10.2 与张量并行（TP）的结构性兼容

以 Megatron-LM 风格的张量并行为例：`gate_proj`、`up_proj` 按 `ColumnParallelLinear` 沿 H 维度切分（每卡持有 H/tp_size 的权重切片），`down_proj` 按 `RowParallelLinear` 对称切分，两者通过 `f/g` 通信原语衔接——前向在 `down_proj` 输出后做一次 all-reduce，反向在梯度传回输入 `I` 之前做一次 all-reduce。这两个通信点由 TP 的线性代数结构决定（row-parallel 层的输出必须先在各卡上算出局部和，再跨卡规约），与算子融合的方式无关：无论 gate/up 投影、SiLU、门控、`down_proj` 被拆成几个 kernel 执行，只要这些计算都是单卡本地的、不跨卡访问权重或激活，通信点的位置就不会改变。

本实现当前的融合范围（gate/up 投影 + SiLU + 门控）全部是单卡本地计算，因此可以直接嵌入 TP 的本地计算段，无需修改 kernel 内部逻辑，只需将权重替换为各卡切片即可。

这一兼容性目前仅停留在结构分析层面，本文尚未在多卡环境下实际验证 DDP 或 TP 场景的正确性与性能表现。

11. 结语

Flash SwiGLU MLP 不是简单地将 SiLU 和矩阵乘法写进一个 Triton kernel，而是对 SwiGLU 训练数据流进行了端到端重组。

其主要贡献包括：

1. 前向双投影融合

在一个 Triton kernel 中计算 (G,U,A) ，复用输入 tile，并消除独立激活和门控 kernel。

2. 连续的 AGU 内存布局

使用单一 $[B, S, 3H]$ 缓冲区存储 A, G, U ，降低显存分配开销，缓解显存碎片化，并为反向传播的大 GEMM 奠定基础。

3. 消除 dA materialization

在同一个 kernel 中计算 dA, dG, dU ，避免完整 dA 张量的分配和 HBM 往返。

4. 原地梯度覆写

使用 dG, dU 原地覆盖生命周期已经结束的 G, U ，显著降低反向峰值显存。

5. 权重梯度 GEMM 聚合

将两个 $[H, BS] \times [BS, D]$ GEMM 合并为一个 $[2H, BS] \times [BS, D]$ GEMM，提高计算效率。

6. 单 accumulator 累加计算 dI

在同一个 accumulator 中累加计算 dI ，降低寄存器压力并消除中间输出与逐元素加法。

7. 训练与推理解耦

训练路径保存反向所需的 G, U ，推理路径则只生成 A ，获得更低显存占用，并能够自动切换。

在当前测试覆盖范围内，Flash SwiGLU MLP 已表现出优于 Liger Kernel 和 PyTorch compile 实现的训练性能，并具有更低的显存占用。更重要的是，这些收益并非来自减少必要的矩阵乘 FLOPs，而是来自更合理的算子融合、内存布局、中间状态生命周期管理和片上累加设计。

本研究局限性总结：

1. 不支持 retain_graph/create_graph 的多次反向场景（见 5.2.1 节）
2. autotune 用 `@triton.autotune` 而非 `@triton.heuristics` 是兼容性妥协（见 5.2.1 节）
3. decoding 场景暂未通过 CUDA graph 验证优势（见 9.2 节）
4. 目前只在单一 GPU（SM120）上验证，未测试其他架构的泛化性
5. 分布式场景（DDP/TP）目前仅完成结构性设计分析，未在多卡环境下实测

未来的优化空间：

1. backward 目前无条件计算全部四个梯度。可以利用 `ctx.needs_input_grad` 的逐项信息，对不需要梯度的输入跳过对应的梯度计算（如 frozen 权重场景），返回 None 即可。
2. 增加一个性能模式（类似于打游戏时开“电竞模式”），前向时直接为反向传播保存 $\text{Sigmoid}(G)$ 和 $\text{SiLU}(G)$ ，而不是保存 G ，这样反向传播时就不用基于 G 重新计算 $\text{Sigmoid}(G)$ 和 $\text{SiLU}(G)$ 。但这样原本前向传播保存的 AGU 张量就变成了 $A, U, \text{Sigmoid}(G), \text{SiLU}(G)$ ，显存占用会增加。这是一个用空间换时间的 trade-off，适合对训练速度有更高要求且显存相对充足的场景。

以上就是 Flash SwiGLU MLP 的全部内容，欢迎大家讨论和提 issue。

附录

为什么选用 `torch.compile(..., mode="max-autotune-no-cudagraphs")` 作为 baseline?

本文选用 `torch.compile(..., mode="max-autotune-no-cudagraphs")` 作为 baseline，而非默认或 `reduce-overhead` 等启用 CUDA graph 的 mode，主要基于以下考虑：

- 一方面，测试脚本中该编译对象被正确性、性能、显存三类测试共用，需要在不同 shape、不同 batch/backward 场景下稳定工作；
- 另一方面，显存测试基于 `torch.cuda.max_memory_allocated()` 统计峰值，而 CUDA graph 会引入独立的私有内存池，可能使显存峰值读数失真，与本文核心的显存对比目标相冲突。

此外，性能测试覆盖 3 组维度配置 × 6 个序列长度（训练）/ 9 档并发请求数（推理）的完整网格，每个 shape 切换都会触发 CUDA graph 重新 capture，其一次性开销会混入计时结果；训练场景还涉及逐次 `zero_grad` 与 backward，属于 CUDA graph capture 相对脆弱的使用模式。

因此选择 `max-autotune-no-cudagraphs`：既保留 Inductor 的算子自动调优与融合能力，又避免上述场景下 CUDA graph 带来的额外噪声与潜在兼容性问题。需要说明的是，这一选择意味着 baseline 未能体现 CUDA graph 对 CPU 侧开销的削减效果，在真实的 decoding 部署场景中，配合 CUDA graph 的 `torch.compile` 实现可能会有更好的表现，所以具体使用哪种方式还需本地实测。