

Flash SwiGLU MLP: Full-Pipeline Co-Optimization and Selective Operator Fusion

The **SwiGLU MLP** is one of the core modules in modern large language models, accounting for a large share of both compute and memory overhead. Current optimization approaches mainly fuse only the SwiGLU activation itself, while the matrix-multiplication parts rely on PyTorch's native implementation (which maps to cuBLAS on the backend). Since the SwiGLU activation is an element-wise operation (element-wise multiplication), it easily hits memory-access bottlenecks, so fusing it yields obvious gains. Representative works in this space include:

- PyTorch's native `torch.compile`
- Open-source high-performance operator libraries (e.g., Liger-Kernel's `LigerSwiGLUMLP`)

This article argues that we can go further, and proposes **Flash SwiGLU MLP**. This implementation does not stop at fusing the SwiGLU activation itself — it co-optimizes the complete forward and backward data flow of the SwiGLU MLP, carefully considering different workload scenarios and designing dedicated solutions for each.

Let's look at the performance and correctness results first, then walk through the design.

Performance Evaluation

This implementation is compared against `torch.compile` and `LigerSwiGLUMLP`, measuring **performance** and **memory usage** across different problem sizes.

The **baselines** are constructed as follows:

```
# 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"
)

```

The reason for choosing `mode="max-autotune-no-cudagraphs"` as the baseline is explained in the appendix.

Hardware and software environment:

```

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

```

Test shapes:

Dimension settings (D is the model hidden size, H is the feed-forward size):

D	H
1024	2816
2048	5632
4096	11008

Different batch sizes and sequence lengths S were chosen for **inference** and **training** scenarios:

```

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}

```

Note: in autoregressive inference, each request typically processes 1 token at a time. To improve throughput, Continuous Batching is usually used to concatenate multiple requests along the S dimension, so the " S " in "Inference" here actually refers to the number of concurrent requests (Num of Requests).

Performance results:

The performance of $FP16$ and $BF16$ is similar, so only the $BF16$ results are shown here; the conclusions apply equally to $FP16$. All tests use 10 warmup iterations and report the mean of 50 measured iterations.

The performance plots use line styles to distinguish implementations:

Circle marker: Flash SwiGLU (this implementation);

Triangle marker: Liger Kernel;

Cross (x) marker: `torch.compile`.

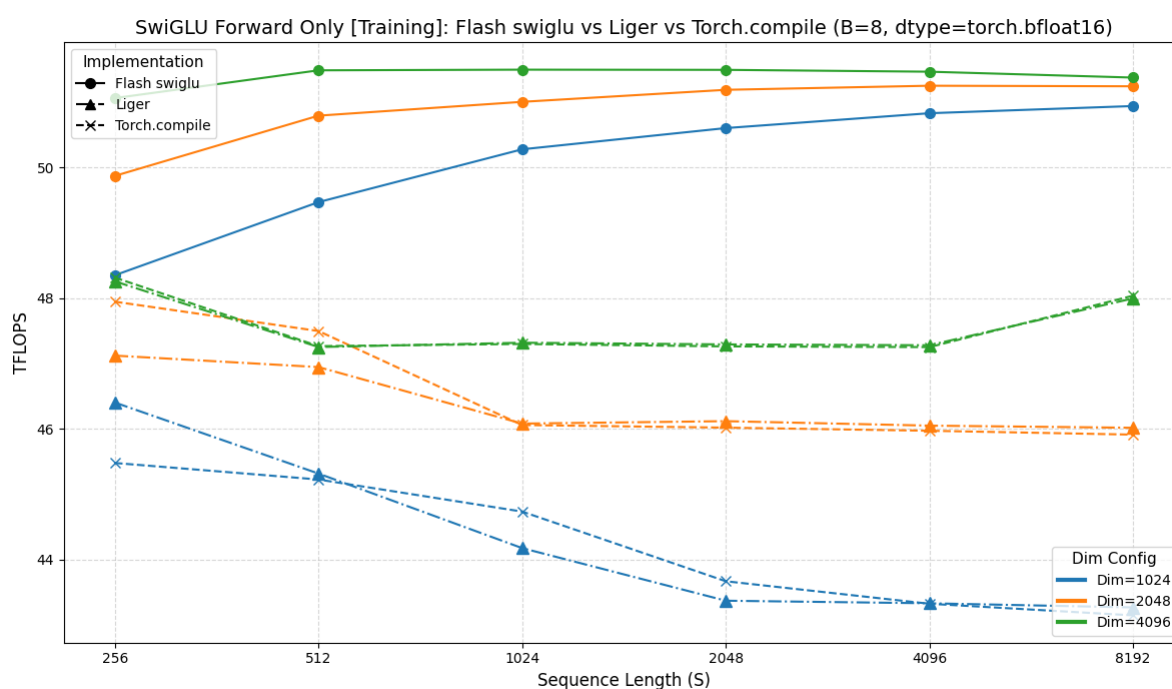
Colors distinguish dimension configurations:

Blue: D=1024, H=2816;

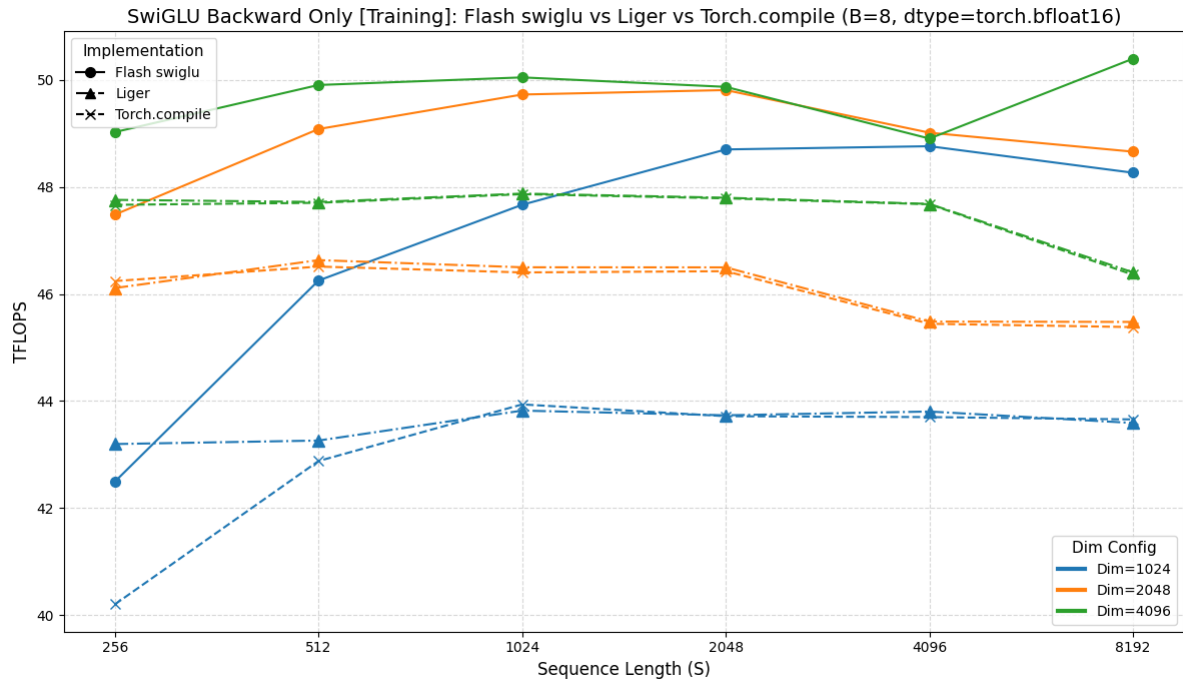
Orange: D=2048, H=5632;

Green: D=4096, H=11008.

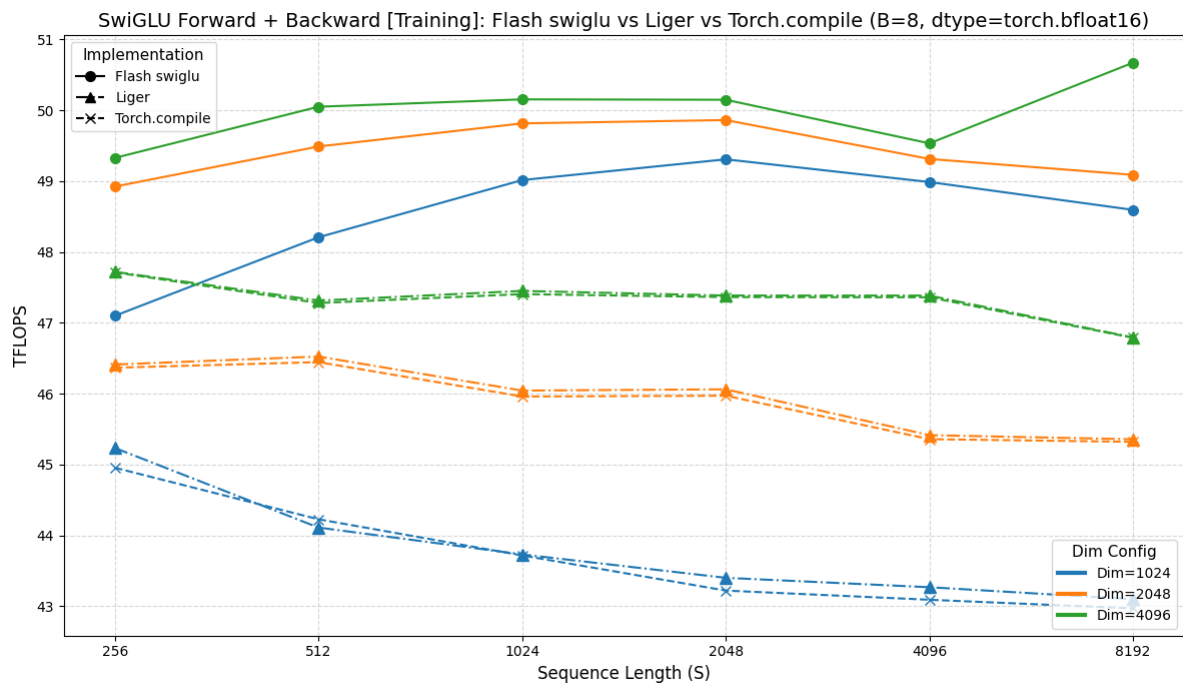
- Training forward pass



- Training backward pass

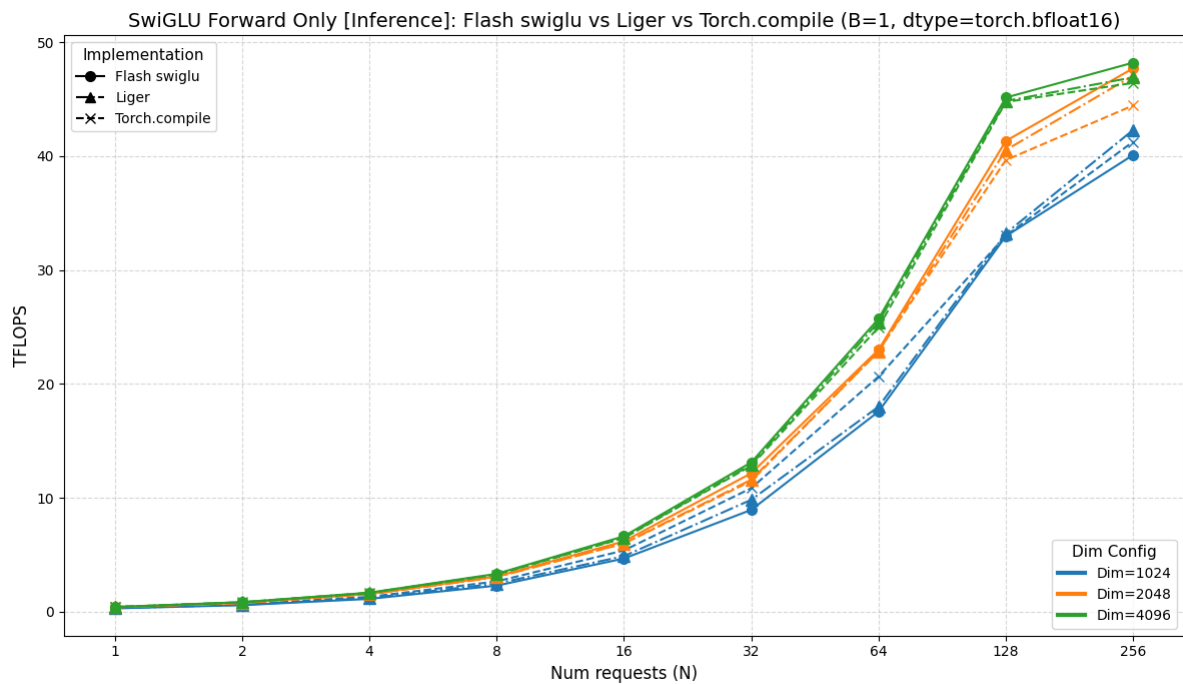


- Training end-to-end



Overall, **in the current test environment**, this implementation outperforms the other two across the training forward, backward, and end-to-end tests, with improvements concentrated in the 5%–15% range. In particular, in large-dimension (Dim=4096) and long-sequence (Seq Len > 2048) scenarios, the TFLOPS of this implementation approach the practical compute ceiling of the device at this precision.

- Inference scenario

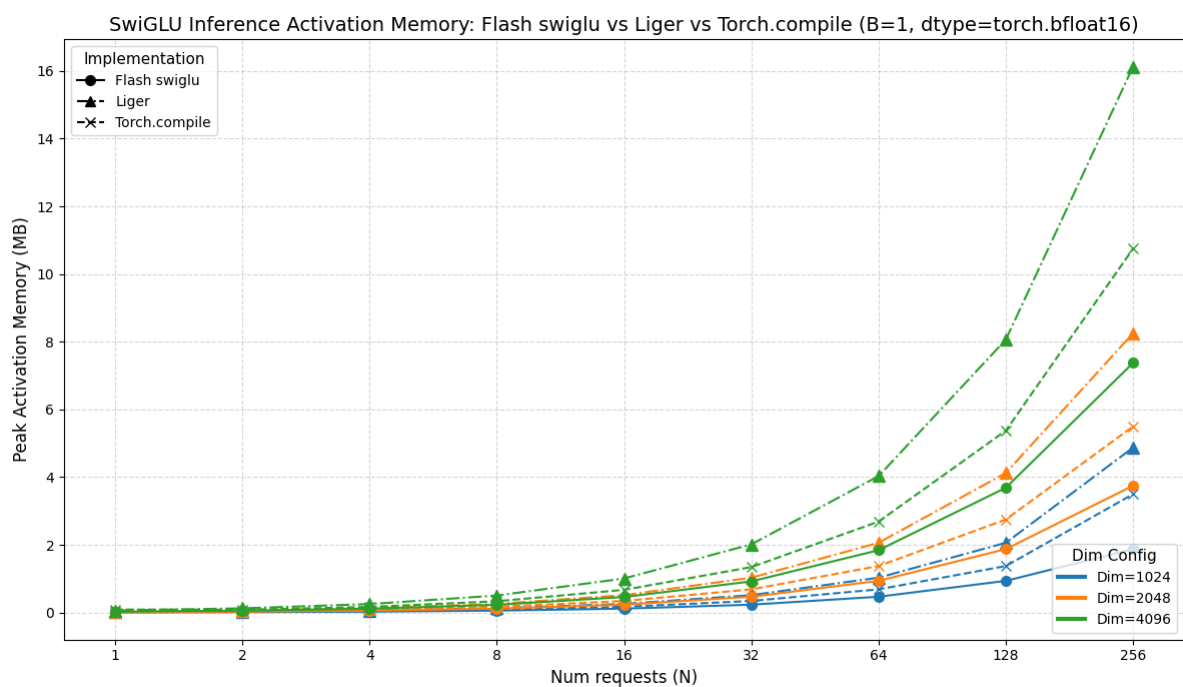


In the **smallest problem size** (D=1024, H=2816) tests, this implementation is *slightly slower* than the other two, but at **larger problem sizes** the three implementations perform similarly, with this one **slightly ahead**. The main reason is that this implementation uses Triton's host-side TMA descriptor API, which carries heavier CPU-side overhead; **at small problem sizes this overhead cannot be amortized**, resulting in slightly lower performance. In real inference deployments, the advantage of this implementation can be more fully realized after using CUDA graphs to reduce CPU-side overhead.

Peak memory results:

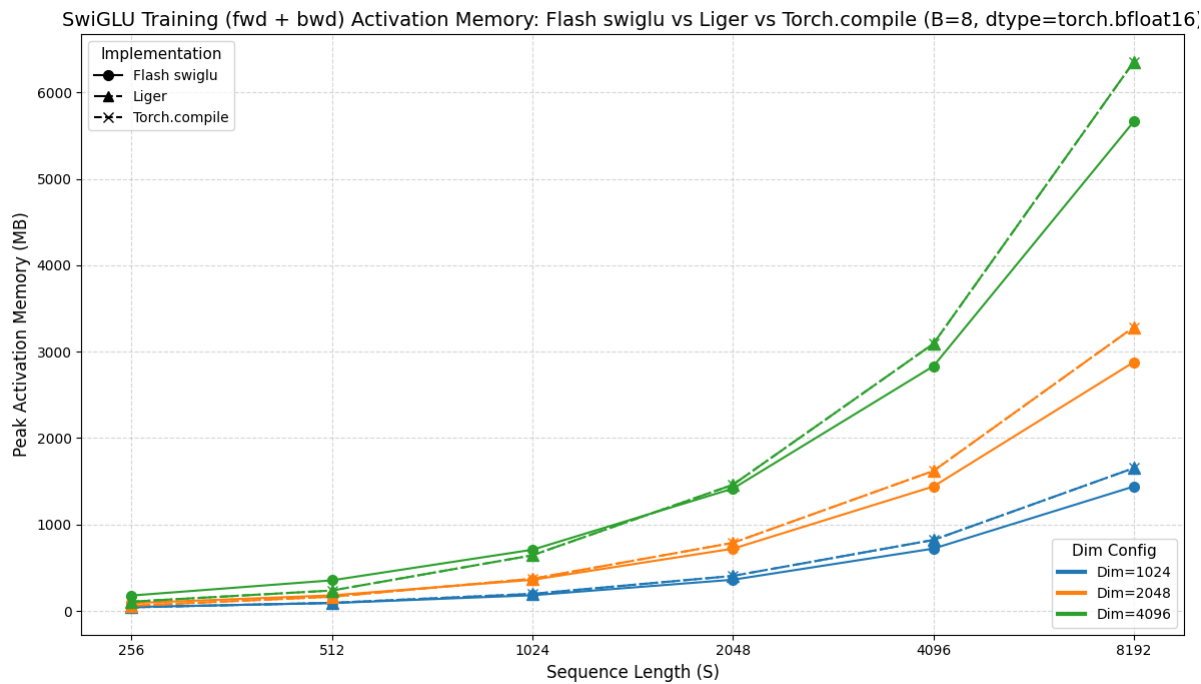
What is measured here is the **peak intermediate memory** overhead (i.e., the peak memory increase after subtracting the fixed footprint of inputs, weights, and weight gradients). The purpose is to avoid weight memory — which is huge — dominating the results and masking the differences we actually want to see.

- Inference (pure forward pass) memory usage



The inference peak intermediate memory consistently outperforms both torch.compile and Liger Kernel across all tested scales, accounting for approximately 38.8%–46.9% of that of Liger Kernel and 31.3%–70.9% of that of torch.compile.

- Training (end-to-end) memory usage



Our implementation achieves lower peak intermediate memory usage in long-sequence scenarios, but higher usage in short-sequence ones. This is primarily attributed to the differing behaviors of the baseline across various data scales—specifically, differences in compilation and memory planning strategies. In short-sequence cases, the baseline employs more aggressive memory reuse optimizations, leading to lower memory footprints.

Correctness Evaluation

Using the naive `torch.compile`d implementation as the numerical reference, we validate the forward output (O) and all four backward gradients (dI , dWg , dWu , dWd) under both training and inference modes, both FP16/BF16 precisions, and both small (B=1, S=1024, D=128, H=512) and large (B=1, S=4096, D=4096, H=11008) data shapes, with the Liger Kernel results shown side by side for comparison.

✅ means passing `torch.allclose(actual, expected, atol=..., rtol=...)`; Max Diff is shown only as an additional statistic.

- **FP16, training mode:**

Scale	Tensor	Max Diff (Flash SwiGLU)	Max Diff (Liger)	Threshold (atol/rtol)
Small	O	1.53e-05 ✅	1.53e-05 ✅	1e-3 / 1e-4
Small	dI	2.44e-04 ✅	6.10e-04 ✅	1e-3 / 1e-4
Small	dWg	2.44e-04 ✅	1.22e-04 ✅	1e-3 / 1e-4
Small	dWu	2.44e-04 ✅	6.10e-05 ✅	1e-3 / 1e-4

Scale	Tensor	Max Diff (Flash SwiGLU)	Max Diff (Liger)	Threshold (atol/rtol)
Small	dWd	4.88e-04	4.88e-04	1e-3 / 1e-4
Large	O	4.77e-07	4.77e-07	1e-3 / 1e-4
Large	dl	3.05e-05	3.05e-05	1e-3 / 1e-4
Large	dWg	1.53e-05	1.53e-05	1e-3 / 1e-4
Large	dWu	1.53e-05	1.53e-05	1e-3 / 1e-4
Large	dWd	3.05e-05	3.05e-05	1e-3 / 1e-4

- **FP16, inference mode:**

Scale	Tensor	Max Diff (Flash SwiGLU)	Max Diff (Liger)	Threshold (atol/rtol)
Small	O	1.53e-05	1.53e-05	1e-3 / 1e-4
Large	O	4.77e-04	4.77e-04	1e-3 / 1e-4

- **BF16, training mode:**

Scale	Tensor	Max Diff (Flash SwiGLU)	Max Diff (Liger)	Threshold (atol/rtol)
Small	O	1.22e-04	1.22e-04	4e-3 / 1e-4
Small	dl	1.95e-03	4.88e-04	4e-3 / 1e-4
Small	dWg	1.95e-03	9.77e-04	4e-3 / 1e-4
Small	dWu	1.95e-03	0.00e+00	4e-3 / 1e-4
Small	dWd	3.91e-03	3.91e-03	4e-3 / 1e-4
Large	O	3.81e-06	3.81e-06	4e-3 / 1e-4
Large	dl	2.44e-04	2.44e-04	4e-3 / 1e-4
Large	dWg	1.22e-04	1.22e-04	4e-3 / 1e-4
Large	dWu	1.22e-04	0.00e+00	4e-3 / 1e-4
Large	dWd	2.44e-04	2.44e-04	4e-3 / 1e-4

- **BF16, inference mode:**

Scale	Tensor	Max Diff (Flash SwiGLU)	Max Diff (Liger)	Threshold (atol/rtol)
Small	O	1.22e-04	1.22e-04	4e-3 / 1e-4
Large	O	3.81e-06	3.81e-06	4e-3 / 1e-4

Results: for both FP16 and BF16, all comparisons pass across all scales and all scenarios.

The overall precision margin is narrower for BF16 because: BF16 has only 7 mantissa bits (FP16 has 10), so numbers of the same magnitude inherently incur larger rounding error under BF16. This is a property of the precision format itself, not a defect of the implementation.

Next, let's walk through the design.

1. Background

A standard SwiGLU MLP contains three linear projections: the gate projection `gate_proj`, the up projection `up_proj`, and the down projection `down_proj`.

Its **forward pass** can be written as:

$$\begin{aligned} G &= I @ W_g^T \\ U &= I @ W_u^T \\ A &= \text{SiLU}(G) \odot U \\ O &= A @ W_d^T \end{aligned}$$

where the input is $I \in \mathbb{R}^{B \times S \times D}$, the intermediate dimension is H , and the output is $O \in \mathbb{R}^{B \times S \times D}$. The shapes of the three weight matrices are:

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

Its **backward pass** can be written as:

$$\begin{aligned} dW_d &= dO^T @ A \\ dA &= dO @ W_d \\ dU &= dA \odot \text{SiLU}(G) \\ dG &= dA \odot U \odot \text{SiLU}'(G) = dA \odot U \odot (\text{Sigmoid}(G) + \text{SiLU}(G) - \text{Sigmoid}(G) \odot \text{SiLU}(G)) \\ dW_u &= dU^T @ I \\ dW_g &= dG^T @ I \\ dI &= dG @ W_g + dU @ W_u \end{aligned}$$

where the input is $dO \in \mathbb{R}^{B \times S \times D}$, and the outputs are dI , dW_g , dW_u and dW_d . The shapes of these four gradients are:

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

The original PyTorch implementation executes two input projections, the SiLU and element-wise gating, and one output projection sequentially. The forward pass contains three GEMMs in total, and the backward pass contains six GEMMs, producing several intermediate tensors of size $B \times S \times H$.

For an input of shape

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

and intermediate dimension H , the forward FLOPs of a standard SwiGLU are:

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

and the backward FLOPs are:

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

Therefore, the main compute of a full training forward and backward comes from nine matrix multiplications of equivalent scale.

This means SwiGLU MLP optimization cannot focus only on the SiLU or the element-wise multiplication. What really needs to be solved is:

- How to reduce intermediate tensor reads/writes around the GEMMs — kernel fusion;
- How to reduce the number of kernel launches — kernel fusion;
- How to improve the compute efficiency of smaller GEMMs — merge several small ones into a bigger one;
- How to avoid excessive register pressure introduced for the sake of fusion — change how accumulators are used;
- How to pick different strategies for the three workloads of training, prefill, and decoding.

This article introduces **Flash SwiGLU MLP**: rather than trying to replace all computation with one giant kernel, this approach takes the complete forward and backward data flow as its optimization target, focusing on:

- Forward: fusing the computation of G , U , and A ;
- Storing G and U in the same contiguous memory block (physically concatenated into one tensor, laying the groundwork for a large GEMM in the backward pass);
- Backward: reusing the G , U storage in place for dG , dU to reduce memory usage;
- Based on the concatenated G and U , merging the computation of dW_g and dW_u into a single larger GEMM;
- Fusing the two matrix multiplications and the element-wise addition of dI , using only a single accumulator in registers to reduce register overhead;
- Providing a dedicated forward kernel for the pure inference path that does not save G , U .

The core idea of Flash SwiGLU MLP is:

Do not blindly fuse everything. Instead, carefully weigh the cost of fusion and co-optimize the full pipeline around tensor lifetimes, the compute data flow, and memory overhead.

2. Reference Implementation

The naive forward and backward are implemented as:

```
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

```

The backward pass can be divided into four logical stages:

- Compute dW_d
- Compute dA, dG, dU
- Compute dW_g, dW_u
- Finally compute dI .

If each formula were executed independently, then in addition to the six GEMMs, the backward pass would also need to:

- Allocate one $B \times S \times H$ tensor for dA ;
- Allocate two more tensors of the same size for dG and dU respectively;
- Read and write (G,U,dA,dG,dU) multiple times;
- Execute the element-wise operations;
- Perform one addition on the two matmul results in the final stage.

These intermediate operations contribute relatively few FLOPs compared to the GEMMs, but they introduce extra memory traffic, allocation, and kernel launch overhead.

3. Overall Design of Flash SwiGLU MLP

3.1 Forward Pass

```
Input I
|
├─ fused Triton kernel ─> G
|                       └> U
|                       └> A = SiLU(G) ⊙ U
|
└─ F.linear(A, wd) ───> O
```

The forward Triton kernel fuses these three steps:

$$G = I @ W_g^T$$

$$U = I @ W_u^T$$

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

The output projection still calls `F.linear`:

$$O = A @ W_d^T$$

Here we choose to keep `down_proj` as an independent GEMM while fusing `gate_proj + up_proj + SiLU + gating` into a single Triton kernel.

3.2 Backward Pass

```
Stage 1: dwd = dOT @ A

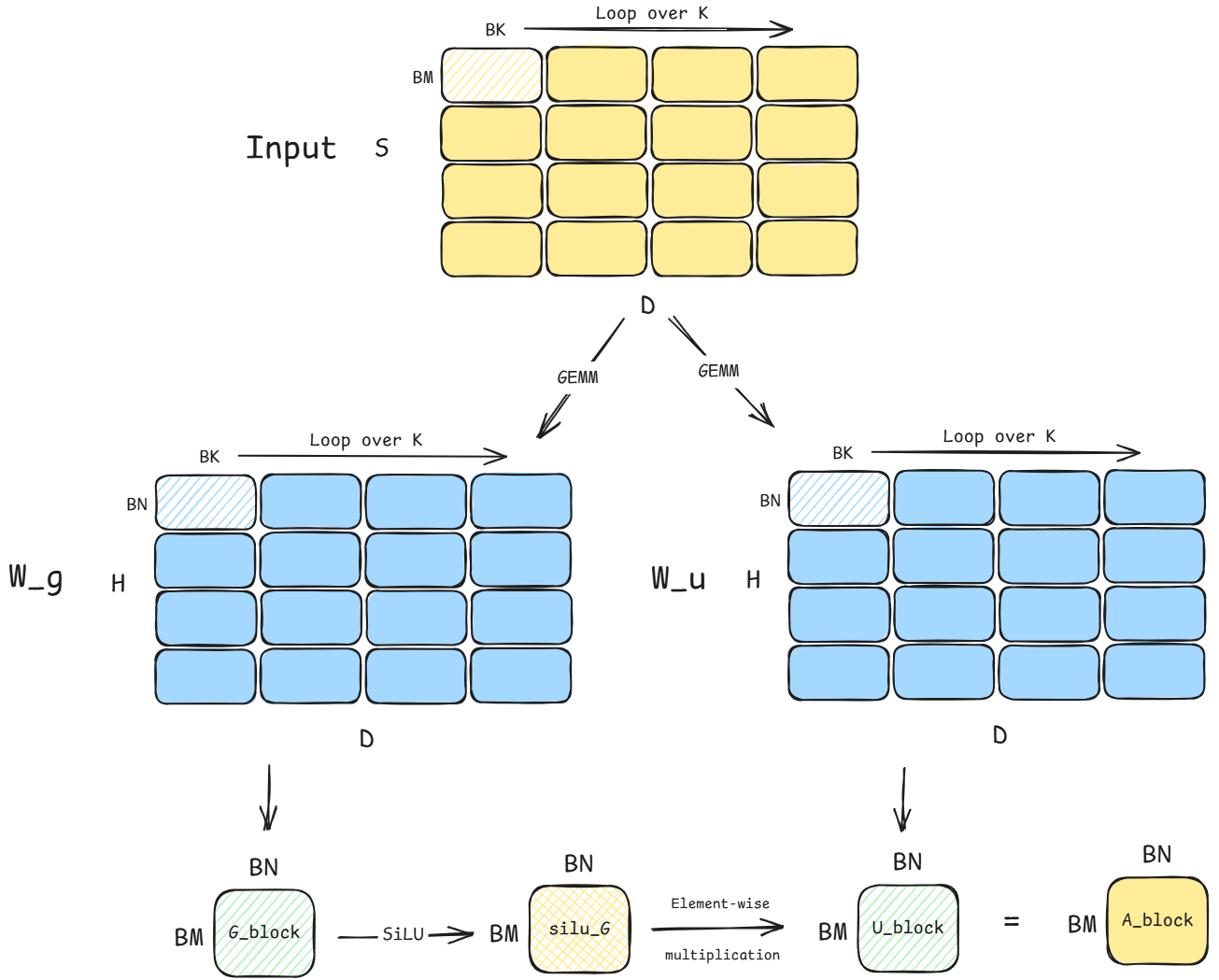
Stage 2: fused kernel
    dA = dO @ wd
    dU = dA ⊙ SiLU(G)
    dG = dA ⊙ U ⊙ SiLU'(G)

Stage 3: dwgu = concat(dG, dU)T @ I

Stage 4: fused kernel
    dI = dG @ wg + dU @ wu
```

This partitioning preserves high-performance implementations for the large GEMMs while fusing the intermediate operations that are most prone to memory-access bottlenecks.

4. Forward Fusion Design



4.1 Fusing the Two Input Projections

In each Triton program, one input tile is loaded:

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

along with the corresponding gate-weight tile and up-weight tile:

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

The kernel maintains two FP32 accumulators (the G_block and U_block in the figure above):

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

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

Within the same K loop it executes:

$$\text{acc}_g + = I_{\text{tile}} @ W_{g,\text{tile}}^T$$

$$\text{acc}_u + = I_{\text{tile}} @ W_{u,\text{tile}}^T$$

This completes both the gate and up projections within the same reduction loop.

The input tile is reused across the two matrix multiplications. Compared to two fully independent GEMMs, this design avoids reloading the input data redundantly and saves one kernel launch.

4.2 Further Fusing the SiLU Activation and Element-wise Multiplication

Once both projections are done, the two accumulators (acc_g and acc_u) can be consumed directly in registers:

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

i.e.:

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

After the computation, A is written back. Note that in the training phase we still need to save G and U for the backward pass, so acc_g and acc_u here must be written back to global memory. But during inference, G and U don't matter at all, so this fusion is more memory-friendly for inference.

Fusing the gate and up projections together with the SiLU activation and element-wise multiplication in this way avoids running, separately:

- the gate GEMM;
- the up GEMM;
- the SiLU;
- the element-wise multiplication.

As a result, the forward pass only needs one fused kernel plus one output-projection GEMM. This drastically reduces the memory traffic from repeatedly materializing intermediate values, and also avoids redundant loads of the input.

4.3 Why G and U Are Still Saved During Training

Because the backward pass needs both G and U at the same time:

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

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

Therefore, training forward must choose between "saving G, U " and "recomputing G, U in backward" — the classic compute-vs-memory trade-off.

The current implementation chooses to save G, U , for the following reasons:

- Recomputation in backward would need to re-run the gate and up input projections, adding a relatively large amount of compute (in our tests, the recompute strategy is much slower);
- Unlike the QK score matrix in attention, whose size is $O(S^2)$, G, U don't blow up memory even when stored;
- Additionally, the storage of G, U can later be reused to hold dG, dU , further easing memory pressure.

4.3.1 A Useful Trick

There's a useful trick worth mentioning here: the code uses a single contiguous tensor `AGU` of shape

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

to store the three tensors A , G , and U :

```
# 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 :]
```

This has three main advantages:

- Only one `torch.empty` call is needed, reducing allocation overhead;
- It allocates and frees one large block of memory, which **mitigates memory fragmentation**.

The third advantage is introduced in Section 5 — think about it first.

Hint: it improves compute efficiency in the backward pass

4.4 Why Not Fuse `down_proj` As Well

If you're familiar with FlashAttention (and if not, see this [FlashAttention-from-scratch tutorial series](#)), it's easy to notice: SwiGLU MLP performs two projections, then some element-wise operations, then one more matmul to produce the output. This compute flow closely resembles attention, which does a QK matmul, then softmax, and finally a matmul with V to get the output. So we could borrow FlashAttention's streaming computation idea.

Then the compute flow becomes:

- For each tile of the input $I_{\text{tile}} \in \mathbb{R}^{B_M \times K}$, iterate over W_g and W_u along the N dimension, doing a matmul with each $W_{g,\text{tile}}, W_{u,\text{tile}} \in \mathbb{R}^{B_N \times K}$ to get $G_{\text{block}}, U_{\text{block}} \in \mathbb{R}^{B_M \times B_N}$
- $A_{\text{block}} = \text{SiLU}(G_{\text{block}}) \odot U_{\text{block}}, A_{\text{block}} \in \mathbb{R}^{B_M \times B_N}$
- Each A_{block} is multiplied with the corresponding tile of $W_d, d_{\text{tile}} \in \mathbb{R}^{B_N \times K}$
- Accumulate the result into $acc \in \mathbb{R}^{B_M \times K}$; after iterating over W_g and W_u along N, acc becomes one tile of the final output O

This looks reasonable — it seems we really can compute the entire SwiGLU MLP with a single operator. But in practice, the resulting operator is extremely slow. The reason lies in $acc \in \mathbb{R}^{B_M \times K}$: the K here corresponds to the last dimension D of the input, commonly a few thousand in size. Putting such a large accumulator in registers inevitably causes register spills. So putting the full D there directly is infeasible.

You might think: since the full D doesn't fit, can't I tile D and keep only part of it? Purely from the perspective of GPU register and shared-memory capacity, yes, you can. But think about it: if we split the D dimension into 10 pieces and assign them to 10 thread blocks, each thread block has to compute A_{block} again — 10 times total. What if D is larger and we need 100 pieces? The redundant computation is unacceptable!

Alternatively, instead of spreading the 10 pieces across 10 thread blocks, let a single thread block compute them: put $acc \in \mathbb{R}^{B_M \times K}$ in global memory and only load one $B_M \times B_K$ block into registers at a time. Now registers fit, and there's no redundant computation — perfect? Not at all! Because this requires repeated global memory accesses: we traded redundant computation for redundant memory traffic, which is still not worthwhile.

Therefore, computing the entire SwiGLU MLP in a single kernel is not practical!

Then why does FlashAttention get away with it?

Because FlashAttention's D dimension is the attention head dimension, commonly 128 in size — GPU registers can hold it, so it works there.

5. The Four-Stage Backward Design

5.1 Stage 1: Compute dW_d with a PyTorch GEMM

$$dW_d = dO^T @ A$$

The implementation flattens the batch and sequence dimensions:

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

i.e.:

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

There is no element-wise fusion opportunity in this stage, and the GEMM is usually large enough, so handing it to PyTorch's backend already yields good performance — this leverages cuBLAS or whatever high-performance matmul implementation PyTorch selects on the current platform.

5.2 Stage 2: Fuse dA , dG , and dU

The naive implementation first computes:

$$dA = dO @ W_d$$

and then computes separately:

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

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

Flash SwiGLU MLP completes all three steps in a single Triton kernel:

The kernel first computes dA_{tile} via a matmul:

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

It then loads G_{tile} and U_{tile} saved from the forward pass and computes:

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

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

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

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

The fused kernel completes both the dA GEMM and the element-wise computation of the two gradients dG, dU within the same program. Note that dA_{tile} stays in registers the whole time and is never materialized, so the complete dA tensor is never written to memory. This completely eliminates the allocation, write-back, and re-read overhead of dA in global memory.

5.2.1 Memory Optimization: Overwriting G, U In Place

Observing the backward formulas, we can see that once dG, dU have been computed, the G, U saved from the forward pass are never used again by later stages. Their lifetimes are therefore mutually exclusive with those of dG, dU , and since these operations are element-wise multiplications, there is no interaction between elements. So Flash SwiGLU MLP simply does:

```
dGU = GU
```

i.e., dG, dU overwrite G, U in place. This memory reuse is an explicit design choice in the implementation, not something that relies on compiler behavior.

Compared to reallocating dG and dU separately, this strategy saves:

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

of additional activation memory.

Note that the comments in the `_swiglu_kernel_backward_dGU` function in this project's source file `flash-swiglu-mlp/src/flash_swiglu_mlp/kernel.py` explicitly point out that the repeated trial runs during autotuning can corrupt the data that in-place overwriting depends on. Therefore the backward (dGU) kernel does not autotune independently; it reuses the tile configuration selected by the forward pass.

Why can the forward-selected tile configuration be reused?

Because the matmul part of this kernel ($dA = dO @ W_d$) has the same shape as the forward matmul, so the optimal configuration found for the forward pass will likely perform well on the dGU kernel too.

However, the truly optimal approach is to tune each Triton kernel separately on the current device, find the optimal configuration for different workloads, and then use `@triton.heuristics` instead of `@triton.autotune`. This guarantees each operator uses its optimal configuration while avoiding the downsides of autotuning.

My implementation uses `@triton.autotune` as a compatibility compromise — it is not the optimal way.

Also worth pointing out: in-place overwriting trades away **higher-order differentiation capability** in exchange for memory optimization. But standard training with a single first-order backward pass (which covers 90% of cases) is unaffected; only scenarios requiring multiple backwards (`retain_graph=True`) or higher-order derivatives (`create_graph=True`) — such as meta-learning, HVP/Hessian computation, gradient penalties, etc. — are impacted.

5.3 Stage 3: Compute dW_g and dW_u with a Single GEMM

$$dW_u = dU^T @ I$$

$$dW_g = dG^T @ I$$

Calling two GEMMs separately incurs two kernel launches, and both operations read the same input I , causing redundant memory traffic.

Remember that trick from Section 4.3? This is where it pays off!

Since we stored G, U in the same contiguous tensor from the very beginning, after the in-place overwrite we can treat `dGU` as:

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

So only a single GEMM call is needed:

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

Then splitting the result with views gives dW_g and dW_u :

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

If the forward pass had not placed GU in one contiguous block of memory, we'd have to manually concatenate here — involving a new allocation and a copy — and performance would likely be worse than just running two GEMMs.

5.4 Stage 4: Compute dI with a Single Accumulator

The input gradient is:

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

A straightforward implementation would be:

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

This approach requires:

- two GEMMs;
- two $B \times S \times D$ intermediate tensors;
- one element-wise addition kernel;
- extra HBM reads/writes of the two intermediate results.

Flash SwiGLU MLP uses one fused Triton kernel and creates only a single FP32 accumulator (*acc*):

In the K loop of the matmul, for each K tile it executes, in sequence:

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

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

In the code, the two `tl.dot`s explicitly share the same `acc`, and finally:

$$dI = \text{acc}$$

5.4.1 Why a Single Accumulator Matters

Suppose we maintained separate accumulators for the two matmuls:

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

then the register demand for accumulators would be twice that of the current approach, noticeably increasing register pressure, and we'd still need $\text{acc}_g + \text{acc}_u$ — one extra element-wise addition.

The current approach maintains only:

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

accumulating both reduction paths into it sequentially. This:

- reduces the number of accumulators;
- lowers register pressure;
- eliminates the $\text{acc}_g + \text{acc}_u$ element-wise addition.

6. Tile Scheduling and L2 Swizzle

Every kernel in this operator can use 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
```

The kernels use 2D logical tiles:

$$[B_M, B_N].$$

The program ID first goes through the L2 swizzle and is transformed into:

$$(pid_m, pid_n).$$

The implementation groups several M tiles into a group and reorders the (M,N) traversal within each group. The goal of this strategy is to make adjacent programs more likely to reuse the same data, improving L2 cache locality.

`GROUP_SIZE_M=1` is equivalent to not doing L2 swizzle. Adding 1 to the autotune configuration choices lets the operator decide for itself whether to use L2 swizzle.

7. Autotune

The autotuning keys in this implementation are: `"dim", "hidden_dim", "bucket_M"`, i.e., `D, H, bucket_M`.

Here `bucket_M` means bucketing the M dimension of the matmul (`batch_size * sequence length`).

```
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)
```

Worth mentioning:

- Since the M dimension of the matmul directly affects the workload size, it must be part of the autotuning keys — but we also can't trigger a fresh round of autotuning for every concrete M , as that cost would be too high. So the `get_bucket_m` function buckets M, where $M = B * S$;
- Also, once the M dimension reaches some threshold, increasing the length further no longer improves GPU utilization — it only increases compute time proportionally, and the optimal configuration stops

changing. So beyond some threshold, autotuning should no longer be triggered. Hence the `threshold` in `get_bucket_m`;

- For scenarios with extremely small M (e.g., the decoding phase of inference, where continuous batching concatenates multiple concurrent requests along the S dimension with $B=1$), `bucket_size=128` is clearly inappropriate — a finer granularity is needed. So `get_bucket_m` adds another branch: when $M < 256$, `bucket_size=32`.
- **Note:** the `threshold=4096` and `bucket_size=32` when $M < 256$ are just **empirical values** for the current device, not universal rules.

8. Separating Training Mode and Inference Mode

8.1 Training Mode

When the inputs require gradients (i.e., backward propagation is needed, which is the training mode), the training path is invoked, and the tensors needed for backward are saved via `ctx.save_for_backward`.

```
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):
            O, 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 O
        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 Inference Mode

No backward propagation is needed during inference, so there's no need to save G, U . For this, a dedicated inference kernel was developed that produces only:

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

followed by:

$$O = A @ W_d^T$$

Since G, U are not written back — only A is — the corresponding call path only needs to allocate one A of shape $[B, S, H]$, instead of the $[B, S, 3H]$ `AGU` used for training.

When the model is in inference mode, this operator automatically selects this inference path.

9. Performance Differences Across Workloads

For the training scenario, the earlier performance tests already showed that this implementation outperforms the other two across the tested configurations, so using Flash SwiGLU MLP for training is strongly recommended.

Let's now discuss in detail how this implementation applies to inference. LLM inference is typically divided into two phases with distinctly different workload characteristics: Prefill and Decoding.

9.1 Prefill

Under common **Chunked Prefill** modes, the prompt processed in one pass usually has a relatively large sequence length, which brings:

- a large M dimension for the matmuls: $M = B \times S$;
- higher GPU parallelism;
- compute dominating the cost, with fixed CPU-side overhead such as kernel launches proportionally small.

In this situation, the forward fusion of Flash SwiGLU MLP can fully leverage the following advantages:

- the input is reused across the gate/up matmuls, avoiding redundant memory traffic;
- the G, U matrices are not materialized — they stay in registers and feed the swiglu activation directly, reducing memory traffic overhead;
- the SiLU activation and gating ($\text{SiLU}(G) \odot U$) don't spawn separate kernels, avoiding extra kernel launch overhead.

Therefore, **prefill is the primary recommended inference scenario for this kernel**.

9.2 Decoding

Autoregressive decoding usually processes only one or a few tokens at a time. Even with continuous batching, constrained by the number of concurrent requests, the effective S remains small.

The small workload size then leads to:

- kernel launch, TensorDescriptor creation, and other CPU-side overheads taking a large share;
- insufficient parallelism and low GPU utilization.

At the workload sizes covered by the current tests, although Flash SwiGLU MLP still slightly outperforms the Liger kernel and `torch.compile` baselines in some scenarios, compared with PyTorch's more mature optimizations — especially its C++-based scheduling machinery — `torch.compile()` remains the more robust choice.

So, unless CUDA graphs can be used to cut CPU-side overhead, the native `torch.compile` implementation is recommended for decoding scenarios.

10. Distributed Training Compatibility

The current implementation and validation of Flash SwiGLU MLP are both based on single-GPU scenarios, with no dedicated optimization for distributed training yet. However, the design does not conflict with mainstream distributed parallelism strategies, in two respects:

10.1 Naturally Compatible with Data Parallelism (DDP/ZeRO)

This implementation plugs into PyTorch's autograd system through a standard `torch.autograd.Function`: the `dwg`, `dwu`, `dwd` returned by `backward` accumulate normally onto the `.grad` of `gate_weight`, `up_weight`, and `down_weight` respectively — exactly the same gradient-accumulation behavior as native `nn.Linear`.

10.2 Structural Compatibility with Tensor Parallelism (TP)

Taking Megatron-LM-style tensor parallelism as an example: `gate_proj` and `up_proj` are sharded along the H dimension as `ColumnParallelLinear` (each card holds a weight slice of `H/tp_size`), `down_proj` is symmetrically sharded as `RowParallelLinear`, and the two are stitched together by the `f/g` communication primitives — forward does one all-reduce after the `down_proj` output, and backward does one all-reduce before the gradient flows back to the input `I`. These two communication points are determined by the linear-algebraic structure of TP (the output of a row-parallel layer must first be computed as a local sum on each card, then reduced across cards) and are independent of how operators are fused: no matter how the gate/up projections, SiLU, gating, and `down_proj` are split into kernels, as long as all these computations are card-local and don't access weights or activations across cards, the positions of the communication points don't change.

The fusion scope of this implementation (gate/up projections + SiLU + gating) is entirely card-local computation, so it can be embedded directly into the local computation segment of TP without modifying the kernel internals — just replace the weights with each card's slice.

This compatibility currently remains at the level of structural analysis; this article has not actually validated correctness or performance of DDP or TP scenarios in multi-GPU environments.

11. Conclusion

Flash SwiGLU MLP is not simply writing SiLU and matmuls into one Triton kernel — it is an end-to-end reorganization of the SwiGLU training data flow.

Its **main contributions** include:

1. **Forward dual-projection fusion**

Computes (G, U, A) in a single Triton kernel, reuses the input tile, and eliminates the separate activation and gating kernels.

2. Contiguous AGU memory layout

Uses a single $[B, S, 3H]$ buffer to store A, G, U , lowering allocation overhead, mitigating memory fragmentation, and laying the foundation for the large GEMM in the backward pass.

3. Eliminating dA materialization

Computes dA, dG, dU in the same kernel, avoiding the allocation and HBM round trip of the full dA tensor.

4. In-place gradient overwrite

Overwrites the expired G, U with dG, dU in place, significantly reducing backward peak memory.

5. Weight-gradient GEMM aggregation

Merges two $[H, BS] \times [BS, D]$ GEMMs into one $[2H, BS] \times [BS, D]$ GEMM, improving compute efficiency.

6. Computing dI with a **single accumulator**

Accumulates dI into the same accumulator, lowering register pressure and eliminating intermediate outputs and the element-wise addition.

7. Decoupling training and inference

The training path saves the G, U needed for backward, while the inference path produces only A for lower memory usage — switching automatically.

Within the scope of current tests, Flash SwiGLU MLP demonstrates better training performance than both the Liger Kernel and PyTorch compile implementations, with lower memory usage. More importantly, these gains don't come from reducing the necessary matmul FLOPs, but from more sensible operator fusion, memory layout, intermediate-state lifetime management, and on-chip accumulation design.

Limitations of this study:

1. Multiple-backward scenarios with `retain_graph/create_graph` are not supported (see Section 5.2.1)
2. Using `@triton.autotune` instead of `@triton.heuristics` for autotuning is a compatibility compromise (see Section 5.2.1)
3. The advantage in decoding scenarios has not been validated with CUDA graphs yet (see Section 9.2)
4. So far validated on a single GPU (SM120) only; generalization to other architectures is untested
5. Distributed scenarios (DDP/TP) have only received structural design analysis so far, with no actual multi-GPU testing

Future optimization opportunities:

1. Backward currently computes all four gradients unconditionally. Using the per-item information in `ctx.needs_input_grad`, gradient computation could be skipped for inputs that don't need gradients (e.g., frozen weight scenarios), returning `None` instead.
2. Add a performance mode (like a "turbo mode" in a game): during forward, save $\text{Sigmoid}(G)$ and $\text{SiLU}(G)$ directly for the backward pass instead of saving G , so the backward pass doesn't have to recompute $\text{Sigmoid}(G)$ and $\text{SiLU}(G)$ from G . But then the tensor saved by forward changes from AGU to $A, U, \text{Sigmoid}(G), \text{SiLU}(G)$, increasing memory usage. This is a space-for-time trade-off, suitable for scenarios with higher demands on training speed and relatively ample memory.

That's everything about Flash SwiGLU MLP. Discussion and issues are welcome.

Appendix

Why choose `torch.compile(..., mode="max-autotune-no-cudagraphs")` as the baseline?

This article chooses `torch.compile(..., mode="max-autotune-no-cudagraphs")` as the baseline rather than the default or `reduce-overhead` modes that enable CUDA graphs, mainly for the following reasons:

- On one hand, the compiled object in the test script is shared by the correctness, performance, and memory tests, and needs to work stably across different shapes and different batch/backward scenarios;
- On the other hand, the memory tests rely on `torch.cuda.max_memory_allocated()` to measure peaks, while CUDA graphs introduce separate private memory pools that could distort peak memory readings — conflicting with the core goal of this article's memory comparisons.

Furthermore, the performance tests cover a full grid of 3 dimension configs × 6 sequence lengths (training) / 9 concurrent request counts (inference), and each shape change triggers a re-capture of the CUDA graph, whose one-time overhead would contaminate the timing results. Training scenarios also involve per-iteration `zero_grad` and backward, which is a usage pattern where CUDA graph capture is relatively fragile.

Therefore `max-autotune-no-cudagraphs` was chosen: it retains Inductor's automatic operator tuning and fusion capabilities while avoiding the extra noise and potential compatibility issues CUDA graphs bring in these scenarios. It should be noted that this choice means the baseline does not reflect CUDA graphs' reduction of CPU-side overhead; in real decoding deployments, a `torch.compile` implementation paired with CUDA graphs may perform better, so which approach to use still needs to be decided by local testing.