Assignment 1: Implementing FlashAttention and its Variants in Triton

August 29, 2025

Introduction

The goal of this assignment is to develop a deep, practical understanding of modern, memory-efficient attention mechanisms. We will explore FlashAttention and several of its state-of-the-art variants, implementing them from the ground up using the Triton language for high-performance GPU programming.

To build the necessary foundation, we will first analyze the FlashAttention algorithm by implementing its core logic in a PyTorch `autograd.Function` (Problem 1), followed by an introduction to Triton's programming model (Problem 2). With these fundamentals, you will then implement the complete FlashAttention forward pass in Triton, first without (Problem 3) and then with (Problem 4) causal masking.

Starting from Problem 5, we will investigate standard state-of-the-art variants of attention, including Grouped-Query Attention, Sliding Window Attention, and Attention Sinks. We will analyze their standard implementations in PyTorch and then write our own optimized versions in Triton. Finally, an *optional* extra credit problem 7 challenges you to implement the memory-efficient backward pass.

Contents

1 Problem 1: Tiled Attention Forward Pass in PyTorch (200 points)

Your goal is to implement the logic of **Algorithm 1** from the FlashAttention-2 paper within a `torch.autograd.Function`. This will serve as your conceptual model before you write any Triton code.

Background

Before we build our tiled version, let's look at a standard, "naive" implementation of multi-head attention that you might find inside a library like Hugging Face's `transformers`. This code is clear and directly follows the logic of the attention paper, but it contains a well-known performance bottleneck.

The implementation below takes Query, Key, and Value tensors and processes them through multiple attention heads in parallel. Notice the explicit reshaping and transposing to handle the head dimension.

```python
import torch
import torch.nn.functional as F
import math

def standard_multi_head_attention(Q, K, V, mask=None):
    """
    A standard, non-optimized multi-head attention implementation.
    Shapes:
    - Q, K, V: (batch_size, seq_len, embed_dim)
    - mask: (batch_size, 1, 1, seq_len) for causal or padding
    """
    # Assume embed_dim is divisible by num_heads
    batch_size, seq_len, embed_dim = Q.shape
    num_heads = 8
    head_dim = embed_dim // num_heads

    # 1. Project and reshape Q, K, V for multi-head processing
    # (batch, seq_len, embed_dim) -> (batch, num_heads, seq_len, head_dim)
    def reshape_for_heads(x):
        return x.view(batch_size, seq_len, num_heads, head_dim).transpose(1, 2)

    Q_heads = reshape_for_heads(Q)
    K_heads = reshape_for_heads(K)
    V_heads = reshape_for_heads(V)

    # 2. Compute raw attention scores (the bottleneck)
    # K_heads is (batch, num_heads, seq_len, head_dim)
    # Transposed K_heads is (batch, num_heads, head_dim, seq_len)
```

```
    # Result `scores` is (batch, num_heads, seq_len, seq_len) -> THE BIG MATRIX!
    scores = torch.matmul(Q_heads, K_heads.transpose(-1, -2))

    # 3. Scale the scores
    scores = scores / math.sqrt(head_dim)

    # 4. Apply mask (e.g., for causality in decoders)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    # 5. Normalize with softmax
    # This reads the entire big matrix and writes a new one.
    attention_weights = F.softmax(scores, dim=-1)

    # 6. Compute the final output
    # `attention_weights` @ `V_heads`
    output = torch.matmul(attention_weights, V_heads)

    # 7. Reshape output back to the original shape
    # (batch, num_heads, seq_len, head_dim) -> (batch, seq_len, embed_dim)
    output = output.transpose(1, 2).contiguous().view(batch_size, seq_len, embed_dim)

    return output, attention_weights
```

The problem lies in step 2 and 5. The code materializes the full `(batch, num_heads, seq_len, seq_len)` `scores` matrix in memory. This is the memory bottleneck that FlashAttention is designed to eliminate.

The FlashAttention Difference

The Core Problem: Memory Bandwidth

Modern GPUs are incredibly fast at math (FLOPs), but they can be surprisingly slow at moving data around. The bottleneck in standard attention isn't the matrix multiplication itself; it's the time spent waiting for data to travel between the GPU's two types of memory:

- **High-Bandwidth Memory (HBM):** Very large (e.g., 40-80GB), but "slow" by GPU core standards. This is where your Q, K, and V tensors live.
- **SRAM:** Extremely small (a few MBs), but orders of magnitude faster. This is the on-chip memory available to the GPU cores.

Let's trace the data flow of the standard algorithm:

1. **Read**: The GPU loads the entire Q and K matrices from slow **HBM** into its compute units.
2. **Write**: It computes the huge $S = QK^T$ matrix and writes it back to slow **HBM**.
3. **Read**: It then loads that same huge S matrix from **HBM** again to perform the softmax.
4. **Write**: It writes the resulting probability matrix P back to **HBM**.
5. **Read**: It loads P and the V matrix from **HBM** to compute the final output.

The GPU is spending most of its time waiting for these massive reads and writes to and from HBM, making the process **memory-bound**.

The FlashAttention Solution: Tiling and Kernel Fusion

FlashAttention restructures the computation to eliminate these expensive round trips to HBM.

1. **Load Tiles**: The kernel loads a small **block** of Q from HBM into the ultra-fast **SRAM**.
2. **Inner Loop**: It then loops through the K and V matrices, loading them one small **block** at a time into **SRAM**.
3. **Compute in SRAM**: For each pair of Q and K blocks, it computes the score sub-matrix S_{ij} **entirely in SRAM**. It immediately uses this score to update the running softmax statistics and the output accumulator, also stored in **SRAM**.
4. **Discard**: After the update, the intermediate score block S_{ij} is simply discarded. It is never written to HBM.
5. **Final Write**: Only after looping through all K/V blocks is the final, correct output block for the initial Q block written from SRAM to HBM.

This approach, called **kernel fusion**, combines all the steps of attention (score calculation, masking, softmax, value-weighting) into a single operation. Because the huge intermediate matrices are never stored in HBM, the memory bottleneck is removed, and the GPU can spend its time doing what it does best: math.

Your Task: Reimplement FlashAttention-2 using Pytorch

Your task is to implement the FlashAttention-2 forward pass algorithm in PyTorch in the `problem_1.py` file. The key challenge is to avoid creating the full $N \times N$ attention score matrix. This algorithm achieves this through two main techniques: **tiling** and **online softmax**.
Your implementation must produce the correct attention output without materializing the full $N \times N$ attention score matrix. This is achieved by combining two key techniques:

- **Tiling:** Processing the Query, Key, and Value matrices in small, manageable blocks (or "tiles").
- **Online Softmax:** Instead of a standard softmax, you will implement a version that iterates through tiles of keys and values, updating three key accumulators: the output vector, the running maximum, and the softmax normalization factor.

Your final implementation will be slow due to the explicit Python loops, but it should be numerically identical to a standard, optimized attention function.

Understanding the Online Accumulation

The core of the algorithm is updating three statistics for each query block $\mathbf{Q}_i$ as we loop through key/value blocks $(\mathbf{K}_j, \mathbf{V}_j)$:

- $\mathbf{m}_i$: The running maximum of the dot-product scores seen so far. This is crucial for numerical stability.
- $\boldsymbol{\ell}_i$: The running sum of the exponentials of the scores. This will be the denominator in the final softmax calculation.
- $\mathbf{O}_i$: The running output, which is a weighted sum of the value vectors $\mathbf{V}_j$.

Algorithm 1 FLASHATTENTION-2 forward pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, block sizes B_c, B_r.

1: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ into $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
2: Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ into T_r blocks $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp $\mathbf{L}$ into T_r blocks $\mathbf{L}_1, \dots, \mathbf{L}_{T_r}$ of size B_r each.
3: **for** $1 \le i \le T_r$ **do**
4: Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.
5: On chip, initialize $\mathbf{O}_i^{(0)} = (\mathbf{0})_{B_r \times d} \in \mathbb{R}^{B_r \times d}$, $\boldsymbol{\ell}_i^{(0)} = (\mathbf{0})_{B_r} \in \mathbb{R}^{B_r}$, $\mathbf{m}_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.
6: **for** $1 \le j \le T_c$ **do**
7: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
8: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
9: On chip, compute $\mathbf{m}_i^{(j)} = \max(\mathbf{m}_i^{(j-1)}, \text{rowmax}(\mathbf{S}_{ij})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \mathbf{m}_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\boldsymbol{\ell}_i^{(j)} = e^{\mathbf{m}_i^{(j-1)} - \mathbf{m}_i^{(j)}} \boldsymbol{\ell}_i^{(j-1)} + \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
10: On chip, compute $\mathbf{O}_i^{(j)} = \text{diag}(e^{\mathbf{m}_i^{(j-1)} - \mathbf{m}_i^{(j)}}) \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_{ij} \mathbf{V}_j$.
11: **end for**
12: On chip, compute $\mathbf{O}_i = \text{diag}((\boldsymbol{\ell}_i^{(T_c)})^{-1}) \mathbf{O}_i^{(T_c)}$.
13: On chip, compute $\mathbf{L}_i = \mathbf{m}_i^{(T_c)} + \log(\boldsymbol{\ell}_i^{(T_c)})$.
14: Write $\mathbf{O}_i$ to HBM as the i-th block of $\mathbf{O}$.
15: Write $\mathbf{L}_i$ to HBM as the i-th block of $\mathbf{L}$.
16: **end for**
17: **return** the output $\mathbf{O}$ and the logsumexp $\mathbf{L}$.

The key insight is that when a new, larger maximum score ($\mathbf{m}_{ij}$) is found, we must rescale our previous running statistics ($\boldsymbol{\ell}_i$ and $\mathbf{O}_i$) to be consistent with this new maximum before adding the new values. This is what the $e^{\mathbf{m}_i^{(j-1)} - \mathbf{m}_i^{(j)}}$ term does in the algorithm.

Your Implementation Steps

Inside the inner loop, you need to perform the following steps, which correspond to lines 8, 9, and 10 of the algorithm:

1. **Compute Scores**: Calculate the score matrix $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T$.
2. **Find New Maximum**: Find the maximum score for the current block, $\mathbf{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij})$. Then, update the overall running maximum: $\mathbf{m}_i^{\text{new}} = \max(\mathbf{m}_i^{\text{old}}, \mathbf{m}_{ij})$.
3. **Calculate Scaled Probabilities**: Compute the unnormalized probabilities for the current block, scaled by the *new* maximum: $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \mathbf{m}_i^{\text{new}})$.
4. **Compute Rescaling Factor**: Calculate the factor to adjust the old running statistics: $\text{scale\_factor} = \exp(\mathbf{m}_i^{\text{old}} - \mathbf{m}_i^{\text{new}})$.
5. **Update Statistics**:
   - Update the running sum: $\boldsymbol{\ell}_i^{\text{new}} = \text{scale\_factor} \times \boldsymbol{\ell}_i^{\text{old}} + \text{rowsum}(\tilde{\mathbf{P}}_{ij})$.
   - Update the output accumulator: $\mathbf{O}_i^{\text{new}} = \text{scale\_factor} \times \mathbf{O}_i^{\text{old}} + \tilde{\mathbf{P}}_{ij} \mathbf{V}_j$.
6. **Update Maximum**: Don't forget to set $\mathbf{m}_i = \mathbf{m}_i^{\text{new}}$ for the next iteration.

2 Problem 2: Triton Basics (100 Points)

In Problem 1, the attention logic was implemented in pure PyTorch. While functionally correct, this implementation is slow because standard Python control flow does not efficiently parallelize on the GPU. This section introduces Triton, a language for writing high-performance GPU kernels to overcome this limitation.

Core Concepts of a Triton Kernel

To write a Triton kernel, it is necessary to understand three fundamental concepts: assigning work to parallel program instances, loading the data for that work, and storing the results.

`tl.program_id` (Work Assignment). A Triton kernel is executed by many **program instances** in parallel, organized in a **grid**. The function `tl.program_id(axis)` provides each program instance with its unique ID (coordinate) within the grid. This ID is used to map the instance to a specific portion of the input data, such as a row of a matrix. For a 1D grid of size `M`, `tl.program_id(axis=0)` will return a unique integer from 0 to `M-1` for each instance. (Example in appendix B.1)

`tl.load` (Reading Data). The `tl.load` function moves data from the GPU's main memory (HBM) to the fast on-chip cache (SRAM), where it can be used in computations. It reads data in blocks, or **tiles**, for efficiency. It also takes in a `mask` argument to ensure memory safety by preventing out-of-bounds reads, while the `other` parameter provides a padding value for elements read outside the valid mask region (more details in Appendix B.3).

`tl.store` (Writing Data) The `tl.store` function is the counterpart to `tl.load`. It writes computed results from the fast SRAM back to the main HBM. This is the final step in a kernel, used to save the output. It can also be masked to prevent out-of-bounds writes, which is critical for avoiding memory corruption (more details in Appendix B.2).

Example

The following example demonstrates how these concepts work together in a simple kernel that computes the sum of each row in a matrix. Study this example carefully, as you will adapt its structure to solve your exercise. Refer to Appendix for more advanced examples on Dropout (Appendix C) and Matrix Multiplication (Appendix D) in Triton.

Goal

Given a 2D matrix `X` of shape `(M, D)`, compute an output vector `output` of shape `(M)` where:

$$\text{output}[i] = \sum_{j=0}^{D-1} X[i, j]$$

Implementation and Verification

```

import torch
import triton
import triton.language as tl

@triton.jit
def row_sum_kernel(
    X_ptr, Output_ptr,
    M, D,
    BLOCK_SIZE_D: tl.constexpr
):
    # Get the unique row index for this program instance.
    pid_m = tl.program_id(axis=0)

    # Calculate the starting pointer for the assigned row in X.
    row_start_ptr = X_ptr + pid_m * D

    # Initialize an accumulator in SRAM for the sum.
    accumulator = 0.0

    # Loop over the row in blocks (tiles).
    for d_offset in range(0, tl.cdiv(D, BLOCK_SIZE_D)):
        # Create offsets and a mask for the current tile.
        d_offsets = d_offset * BLOCK_SIZE_D + tl.arange(0, BLOCK_SIZE_D)
        d_mask = d_offsets < D

        # Load a tile of the X row from HBM into SRAM.
        x_tile = tl.load(row_start_ptr + d_offsets, mask=d_mask, other=0.0)

        # Perform computation on the tile in SRAM.
        accumulator += tl.sum(x_tile)

    # Store the final accumulated value from SRAM back to HBM.
    tl.store(Output_ptr + pid_m, accumulator)

def row_sum_triton(X):
    """The Python host code (launcher) for our kernel."""
    M, D = X.shape
    output = torch.empty(M, device=X.device, dtype=X.dtype)
    grid = (M, )
    row_sum_kernel[grid](X, output, M, D, BLOCK_SIZE_D=1024)
    return output

def row_sum_pytorch(X):
    """
    Calculates the sum of each row using a sequential Python loop. (similar to
    ↪  torch.sum(X, dim=1)
    This is intentionally inefficient on a GPU to contrast with the Triton kernel.
    """
    M = X.shape[0]
    output = torch.empty(M, device=X.device, dtype=X.dtype)
    # This `for` loop runs sequentially, one row at a time.
    for i in range(M):
        output[i] = torch.sum(X[i])
    return output
```

```python

# --- Verification Script ---
X_test = torch.randn(256, 3000, device='cuda', dtype=torch.float32)
triton_result = row_sum_triton(X_test)
pytorch_result = row_sum_pytorch(X_test, dim=1)
print(f"Results are close: {torch.allclose(triton_result, pytorch_result)}")
```

Your Task - Tiny Kernel Exercise: Weighted Row-Sum

Goal

To put these concepts into practice, you will implement a simple Triton kernel that performs a weighted sum. Given a 2D matrix `X` of shape `(M, D)` and a 1D weight vector `w` of shape `(D)`, you will compute an output vector `output` of shape `(M)` where:

$$\text{output}[i] = \sum_{j=0}^{D-1} X[i, j] \times w[j]$$

Your kernel will be launched with a 1D grid of size `M`. Each program instance will be responsible for computing the weighted sum for a single row of `X`.

Now, complete the code in `problem_2.py`

3 Problem 3: Triton Kernel for FlashAttention (200 points)

Now, we'll translate the memory-efficient logic from the PyTorch implementation into a high-performance Triton kernel. This is where we achieve significant hardware acceleration by writing code that compiles directly to the GPU's instruction set.

Your Task

Your task is to **implement the core online softmax update logic** within the provided Triton kernel skeleton in the `problem_3.py` file. The kernel already handles the complex parts like setting up the parallel grid, calculating memory pointers, and loading data blocks. You will focus on the mathematical heart of the algorithm.

From PyTorch Loops to a Triton Grid

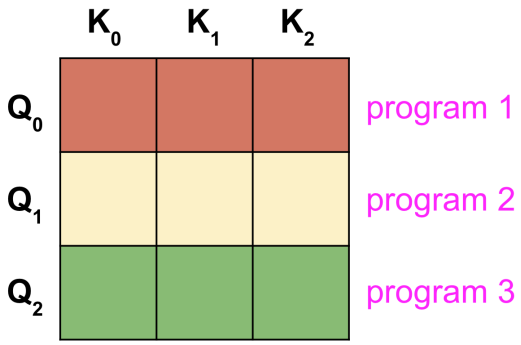

Figure 1: The FlashAttention kernel parallelizes the computation by assigning each program to a specific query block (Q_0, Q_1, Q_2). Each program then serially processes the key and value blocks, accumulating the attention output for its assigned query block.

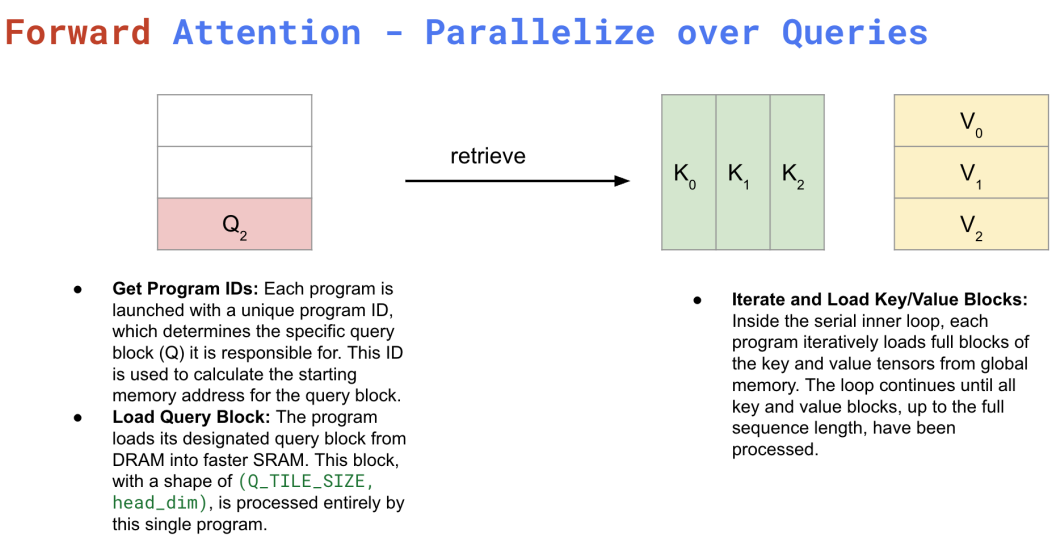

Figure 2: An example of a single parallel program of the FlashAttention kernel. Each program is assigned a specific query block (Q_2), which it loads into SRAM. Within the program's inner loop, it then serially retrieves and processes full blocks of the Key and Value tensors (K_0, K_1, K_2 and V_0, V_1, V_2) from global memory.

The fundamental shift from PyTorch to Triton is how we handle parallelism.

- **PyTorch's Outer Loop**: In Python, we used a `for` loop to iterate through blocks of the Query matrix. This is executed sequentially.
- **Triton's Parallel Grid**: In Triton, this outer loop is replaced by a grid of parallel programs. We launch thousands of program instances simultaneously, each identified by a unique ID `tl.program_id`. This ID determines the specific block of the Query matrix assigned to the instance. As shown in Figure Figure 1, every query block across all heads and batches is processed in parallel.
- **Triton's Inner Loop**: The inner loop, which iterates through blocks of Keys (K) and Values (V), remains a standard Python `for` loop inside the Triton kernel. Each program instance executes this loop independently to compute the final attention output for its assigned query block (Figure (Figure 3)).

Your Implementation Steps

Your task is to fill in the section marked: `--- STUDENT IMPLEMENTATION REQUIRED HERE ---` inside the kernel `_flash_attention_forward_kernel`. The code for loading `q_block`, `k_block`, `v_block` and computing the scores `s_ij` is already provided. Use these tensors to implement the online softmax update logic. The goal is to correctly update the running accumulators (acc, `m_i`, `l_i`) for each new block of keys and values.

4 Problem 4: Adding Causal Masking to the Triton Kernel (100 Points)

In this problem, you will implement a Triton kernel for *causal* FlashAttention. The main challenge is to prevent tokens from attending to future tokens. For maximum efficiency, we will split the inner loop over the key/value blocks into two distinct phases, as shown in Figure 3.

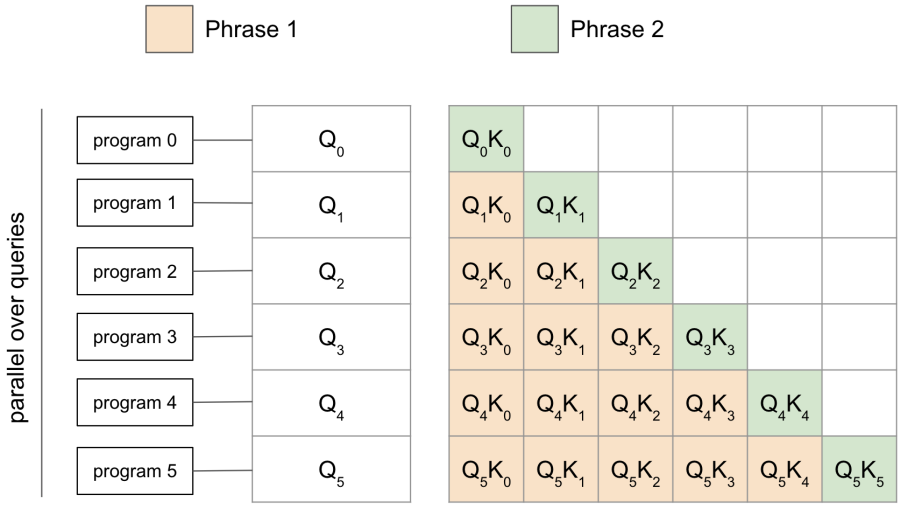

Figure 3: Illustration of a two-phase strategy for causal masking in attention mechanisms. This approach optimizes performance by handling off-diagonal blocks (Phase 1), which require no masking, separately from the diagonal block (Phase 2)

Your task is to fill in the implementation for both phases within the provided `_flash_attention_forward_causal_kern` skeleton.

Phase 1: Off-Diagonal Blocks (No Masking)

The first for loop in the kernel handles the off-diagonal blocks. It iterates through all key/value blocks that are entirely in the past relative to the current query block (i.e., where the key block index is less than the query block index). Since every key in these blocks is guaranteed to have a sequence position less than every query, no causal masking is required.

Phase 2: Diagonal Blocks (With Causal Masking)

The second for loop handles the diagonal blocks, where query and key indices can overlap. For these blocks, you must apply a causal mask to the score matrix $\mathbf{S}_{ij}$ *before* performing the online softmax update. This ensures that a query at position i cannot attend to a key at position j if $j > i$.

1. **Phase 1: Off-Diagonal Blocks (No Masking Needed):**
   - Modify the inner loop to iterate only up to the start of the diagonal block. The loop bound will be `q_block_idx * Q_TILE_SIZE`.
   - Inside this loop, the logic remains the same as in Problem 3. Since every key in these blocks is guaranteed to be in the past relative to the queries, no masking is required. This is a significant optimization.
2. **Phase 2: The Diagonal Block (Masking Required):**
   - After the first loop, handle the single diagonal block where `k_block_start == q_block_idx * Q_TILE_SIZE`.

- **Apply Mask:** Create the causal mask `mask = q_indices[:, None] >= k_indices[None, :]` and apply it to the score matrix `S_ij` before performing the online softmax update for this block.

Section 2: State-of-the-Art Variants

5 Problem 5: Extending for Grouped-Query Attention (GQA) (50 points)

The problem 5 is to adapt your causal FlashAttention kernel to support Grouped-Query Attention (GQA).

Background

In large language models, one of the biggest memory bottlenecks during inference is the **KV Cache**. In standard Multi-Head Attention (MHA), every Query head has its own corresponding K and V head, making the KV Cache very large.

Grouped-Query Attention (GQA) is a memory-saving optimization where multiple Query heads are organized into groups, and each group **shares** a single Key and Value head. For example, you might have 32 Query heads but only 4 K/V heads, meaning each group of 8 Query heads uses the same K and V data.

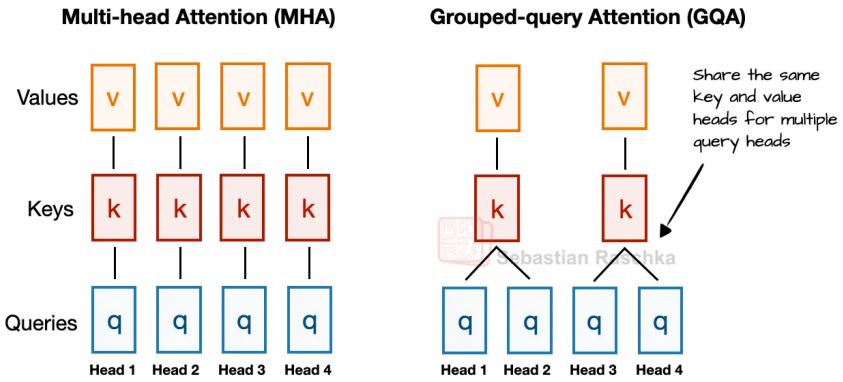

Figure 4: An example comparing Multi-head Attention Grouped-query Attention (GQA). In this figure, the group size for GQA is 2, where a key and value pair is shared among 2 queries. Image from: From GPT-2 to GPT-OSS.

The GptOssForCausalLM (The Hugging Face implementation of GPT-OSS) shows a standard way to handle GQA. The core logic is in the `repeat_kv` function, which explicitly expands the Key and Value tensors to match the number of Query heads before the main attention calculation. .

```
import torch
from torch import nn

class GptOssAttention(nn.Module):
    def __init__(self, config):
        # --- Step 1: Define the GQA Structure ---
        # The key parameter is `num_key_value_groups`, the ratio of Q heads to K/V heads.
        self.num_attention_heads = config.num_attention_heads
        self.num_key_value_heads = config.num_key_value_heads
        self.num_key_value_groups = self.num_attention_heads // self.num_key_value_heads

        # The Query projection is standard for all heads.
        self.q_proj = nn.Linear(config.hidden_size, self.num_attention_heads * config.head_dim)

```

```
        # IMPORTANT: K and V projections are smaller for GQA (num_key_value_heads).
        # In standard MHA, this would be `self.num_attention_heads`.
        self.k_proj = nn.Linear(config.hidden_size, self.num_key_value_heads *
        ↪  config.head_dim)
        self.v_proj = nn.Linear(config.hidden_size, self.num_key_value_heads *
        ↪  config.head_dim)

        self.o_proj = nn.Linear(self.num_attention_heads * config.head_dim,
        ↪  config.hidden_size)

    def forward(self, hidden_states: torch.Tensor, ...):
        # --- Step 2: Project inputs and call the attention function ---
        # Note that K and V tensors will have fewer heads than Q.
        query_states = self.q_proj(hidden_states).view(...)
        key_states = self.k_proj(hidden_states).view(...)
        value_states = self.v_proj(hidden_states).view(...)

        # The `eager_attention_forward` function is called to handle the core logic.
        attn_output, _ = eager_attention_forward(
            self, query_states, key_states, value_states, ...
        )

        attn_output = self.o_proj(attn_output)
        return attn_output, None

def eager_attention_forward(module: nn.Module, query: torch.Tensor, key:
↪  torch.Tensor, ...):
    # --- Step 3: Expand K and V Tensors ---
    # The `repeat_kv` function explicitly duplicates the K/V heads to match the Q
    ↪  heads.
    key_states = repeat_kv(key, module.num_key_value_groups)
    value_states = repeat_kv(value, module.num_key_value_groups)

    # Standard attention calculation follows...
    attn_weights = torch.matmul(query, key_states.transpose(2, 3))
    # ...
    return torch.matmul(attn_weights, value_states), None

def repeat_kv(hidden_states: torch.Tensor, n_rep: int) -> torch.Tensor:
    """
    Expands K/V heads to match the number of Q heads.
    (batch, num_kv_heads, seqlen, head_dim) -> (batch, num_q_heads, seqlen,
    ↪  head_dim)
    """
    batch, num_key_value_heads, slen, head_dim = hidden_states.shape
    if n_rep == 1:
        return hidden_states
    hidden_states = hidden_states[:, :, None, :, :].expand(
        batch, num_key_value_heads, n_rep, slen, head_dim
    )
    return hidden_states.reshape(batch, num_key_value_heads * n_rep, slen, head_dim)
```

Why This is Inefficient

The `repeat_kv` function is a classic example of a memory-bound operation. It reads the entire K and V tensors from HBM, creates new, larger tensors that are `n_rep` times bigger in the head dimension, and

writes these new tensors back to HBM. This is inefficient for two reasons:

1. **Memory Consumption:** It temporarily allocates a significant amount of extra memory for the expanded K and V tensors.
2. **Memory Bandwidth:** It performs a slow, full read-and-write operation on the entire KV cache just to duplicate data.

A Triton kernel would avoid this entirely. Instead of creating a new tensor, the kernel would use pointer arithmetic. Each program instance, responsible for a specific Query head, would use its ID to calculate the correct pointer to the *shared* K/V head's data in the original, smaller tensor. This is a purely computational change that requires no extra memory allocation or large-scale data movement.

Your Task

Your task is to modify the causal attention kernel from Problem 4 to correctly handle the GQA head mapping. The core attention logic will remain the same. The changes are focused on ensuring each Query head retrieves the correct, shared Key/Value head data.
Your implementation will be done in three small parts:

- Part 1: Calculate the K/V Head Index: Our goal is to map the current query head index (q_head_idx) to its corresponding shared key/value head index (kv_head_idx).
- Part 2: After updating the pointer calculations, you can copy and paste your entire working implementation for the online softmax update from the off-diagonal section of your solution to Problem 4.

6 Problem 6+7: Extending for Sliding Window Attention (SWA) and Attention Sinks (StreamingLLM) (50 points for each)

In these final problems, you'll implement two advanced, memory-efficient attention mechanisms that are crucial for modern Large Language Models: Sliding Window Attention (SWA) and Attention Sinks. You will first build a kernel with SWA in problem_6.py and then extend it with Attention Sinks in problem_7.py to create a complete, state-of-the-art attention implementation.

Background

The quadratic complexity of attention ($O(N^2)$) makes it computationally prohibitive for very long sequences. **Sliding Window Attention (SWA)** is a powerful technique that reduces this to linear complexity by restricting each token's attention to a fixed-size window of its most recent neighbors (e.g., the last 4096 tokens).

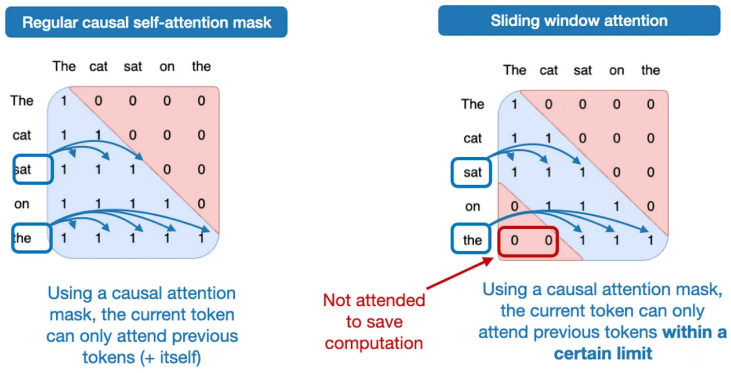

Figure 5: Comparison between regular attention (left) and sliding window attention (right).

While SWA is efficient, it suffers from a critical flaw: as the window slides, the model eventually forgets the very first tokens of a sequence. These initial tokens often contain crucial context or instructions

(like a system prompt). Research on **Attention Sinks** revealed that these first few tokens naturally act as "sinks" that attract attention from the entire sequence, providing stability. When they are evicted from the cache, the model's performance degrades catastrophically. The solution is to augment SWA by forcing the model to *always* attend to the first few "sink" tokens in addition to the local sliding window. This hybrid approach preserves the crucial initial context while still benefiting from the linear complexity of SWA.

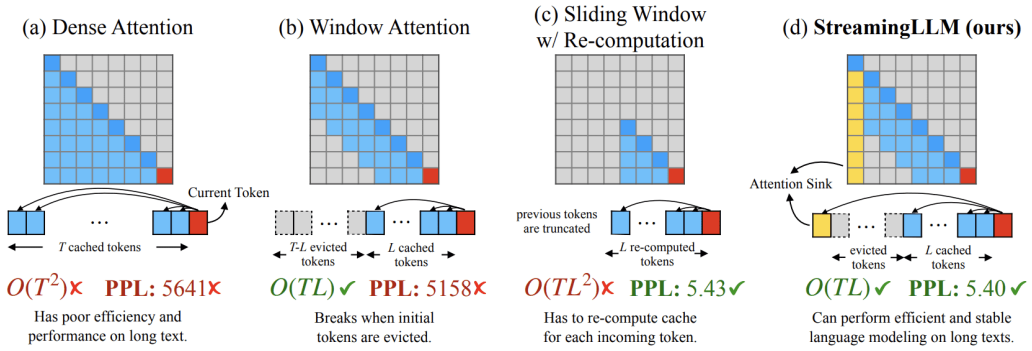

Figure 6: **Comparing Some Attention Methods.** This figure shows how different attention strategies perform when a model, trained on a fixed length, must process a much longer text.**(a) Dense Attention:** Every token attends to every other token. *Problem:* This is very slow ($O(T^2)$) and uses too much memory, and its performance degrades on texts longer than its training length. **(b) Window Attention:** Only keeps the most recent tokens in memory. *Problem:* While efficient, performance collapses once the important starting tokens (the initial context) are pushed out of the memory window. **(c) Sliding Window w/ Re-computation:** Rebuilds the cache for recent tokens at each step. *Problem:* This performs well but is extremely slow, as it constantly re-computes attention for the entire window. **(d) StreamingLLM (Attention Sinks):** The optimal solution. It always keeps the first few "sink" tokens in memory, in addition to the recent tokens. *Result:* It is both fast and stable, maintaining high performance on infinitely long texts by never forgetting the initial context.

The Hugging Face implementation of GPT-OSS handles both SWA and Attention Sinks. The process starts in the main model's forward pass, where the appropriate attention mask is created.

Step 1: Creating the Attention Mask In the `GptOssModel.forward` method, the code prepares a dictionary of masks. For layers designated as `sliding_attention`, it calls a special utility to create the appropriate mask.

```python
# --- Step 1: Create the Attention Mask in the Main Model ---
class GptOssModel(nn.Module):
    def forward(self, input_ids: torch.LongTensor, attention_mask:
    ↪  Optional[torch.Tensor] = None, ...):
        inputs_embeds = self.embed_tokens(input_ids)
        hidden_states = inputs_embeds

        # The model prepares a dictionary of masks. For layers designated as
        # "sliding_attention", it calls a special utility to create the mask.
        mask_kwargs = {"config": self.config, "attention_mask": attention_mask, ...}
        causal_mask_mapping = {
            "full_attention": create_causal_mask(**mask_kwargs),
            "sliding_attention": create_sliding_window_causal_mask(**mask_kwargs),
        }

        # The mapping of attention type is then passed down to each decoder layer.
        for decoder_layer in self.layers:
            hidden_states = decoder_layer(
                hidden_states,
                attention_mask=causal_mask_mapping[decoder_layer.attention_type],
                ...
```

```
            )

# --- Step 2: Define the Sink Parameters in the Attention Module ---
class GptOssAttention(nn.Module):
    def __init__(self, config: GptOssConfig, layer_idx: int):
        super().__init__()
        # ... other initializations like q_proj, k_proj, v_proj, ....
        self.sliding_window = config.sliding_window if config.layer_types[layer_idx]
        ↪  == "sliding_attention" else None

        # A learnable Sink Parameter is created for each attention head to act as a
        ↪  sink logit.
        self.sinks = nn.Parameter(torch.empty(config.num_attention_heads))

    def forward(self, hidden_states: torch.Tensor, attention_mask:
    ↪  Optional[torch.Tensor], ...):
        # Project inputs to Q, K, V ...

        # --- Step 3: Call the low-level attention function ---
        # The specific mask and the sink parameters (via `self`) are passed along.
        attn_output, _ = eager_attention_forward(
            self, query_states, key_states, value_states, attention_mask, ...
        )
        # ...


# --- Step 4: Apply the Mask and Sinks in the Low-Level Function ---
def eager_attention_forward(module: nn.Module, query: torch.Tensor, attention_mask:
↪  Optional[torch.Tensor], ...):
    attn_weights = torch.matmul(query, key_states.transpose(2, 3)) * scaling

    # Apply the sliding window mask (if applicable)
    if attention_mask is not None:
        attn_weights = attn_weights + attention_mask

    # Add the Attention Sinks
    # The `module.sinks` parameter defined in Step 2 is used here.
    sinks = module.sinks.reshape(1, -1, 1, 1).expand(
        query.shape[0], -1, query.shape[-2], -1
    )
    combined_logits = torch.cat([attn_weights, sinks], dim=-1)

    # Softmax is applied to the combined logits
    probs = F.softmax(combined_logits, dim=-1)
    # ...
```

Further Reading & Resources

- Efficient Streaming Language Models with Attention Sinks
- The Big LLM Architecture Comparison

Why This is Inefficient

This PyTorch implementation has two main inefficiencies that a fused Triton kernel can solve:

1. **Mask Materialization (SWA):** The `create_sliding_window_causal_mask` function creates a large mask tensor in HBM. For a long sequence, this mask can still be very large and slow to

create and apply. A Triton kernel can implement the sliding window logic directly in its inner loop bounds, never needing to create or load a mask tensor at all.

2. **Logit Concatenation (Sinks):** The `torch.cat` operation is another memory-bound step. It allocates a new, larger tensor in HBM to hold the combined scores and sink logits. A Triton kernel would handle this much more efficiently. It would simply compute the normal attention for the sink blocks and the window blocks sequentially, updating the *same* set of online softmax accumulators (`m_i`, `l_i`, `acc`). This avoids any extra memory allocation.

Your Tasks

Problem 6 (Sliding Window Attention (SWA)): Your task is to implement a causal FlashAttention kernel that incorporates both Grouped-Query Attention (from Problem 5) and Sliding Window Attention.

- Part 1: GQA Head Mapping This problem builds on the previous one. First, copy your working GQA logic from Problem 5 to calculate the correct kv_head_idx from the given q_head_idx.
- Part 2 (SWA Loop Bounds): The core of SWA is restricting the keys that a query block can see. Calculate the starting sequence position of the attention window, window_start, for the current query block.
- Part 3 (SWA Masking and Core Logic): Implement the full logic for both the off-diagonal (Phase 1) and diagonal (Phase 2) loops.

Problem 7 (Adding Attention Sinks: Your final task is to build the ultimate kernel by incorporating GQA, SWA, and Attention Sinks. This will require structuring your kernel's inner loop into three distinct phases.

- Phase 0: Process the Sink Blocks.
- Phase 1: Process Off-Diagonal Blocks in the Window
- Phase 2: Process the Diagonal Block

7 Problem 8: Backward Pass - with Recomputation. (Optional - 200 points)

Background

The **backward pass** is essential for training neural networks, as it computes the gradients needed to update model weights. A major challenge for a memory-efficient attention implementation is that a standard backward pass requires storing the massive intermediate attention score matrix (S). FlashAttention's entire purpose was to avoid storing this matrix.

The solution is **recomputation**. To save memory, we trade a small amount of extra compute. During the backward pass, instead of loading the saved S matrix, we simply recompute it on the fly, one block at a time. For numerical stability, we only need to have saved the small intermediate statistics (m and l) from the forward pass.

Goal

The goal is to implement the memory-efficient backward pass for your FlashAttention kernel, making it fully trainable.

Deliverable

A complete `torch.autograd.Function` with a working `backward` method. This will involve writing a new backward Triton kernel that implements the recomputation strategy and uses atomic operations (`tl.atomic_add`) for safe gradient accumulation. The final deliverable is to verify the correctness of your gradients against PyTorch's standard attention on small shapes using `torch.autograd.gradcheck`.

8 Appendix

A Custom Kernels with `torch.autograd.Function`

What is `autograd.Function`?

PyTorch's magic lies in its automatic differentiation engine, called **Autograd**. When you perform operations on tensors, PyTorch builds a computational graph. During backpropagation (‘loss.backward()‘), it traverses this graph backward to automatically compute gradients for all model parameters.

However, this magic only works for operations that PyTorch already knows about. When we want to introduce a highly optimized, custom operation—like a fused kernel written in Triton or CUDA—PyTorch has no idea what we did. It doesn't know how to run our code or, more importantly, how to calculate its derivative.

`torch.autograd.Function` is the official bridge that connects our custom, low-level code to PyTorch's high-level graph. It's a contract where we tell PyTorch:

- **"Don't track this operation automatically."**
- **"For the forward pass, run the code I provide in the `forward()` method."**
- **"For the backward pass, use the exact mathematical formula I provide in the `backward()` method."**

By defining these two methods, we can integrate any custom computation seamlessly into a larger, trainable PyTorch model.

Example: A Custom Cube Function

Let's create a custom function that computes $y = x^3$. The derivative is $\frac{dy}{dx} = 3x^2$. We'll implement this in an `autograd.Function`.

```python
import torch

# Inherit from torch.autograd.Function
class Cube(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes.
    """
    @staticmethod
    def forward(ctx, x):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        # ctx.save_for_backward is used to save the input tensor `x`
        # We will need it in the backward pass to compute the gradient.
        ctx.save_for_backward(x)
        return x.pow(3)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        # We retrieve the saved input tensor `x` from the context.
```

```
        x, = ctx.saved_tensors

        # The derivative of y = x^3 is dy/dx = 3x^2.
        # According to the chain rule, the gradient w.r.t the input is:
        # (gradient of loss w.r.t output) * (derivative of this function)
        grad_input = grad_output * 3 * x.pow(2)

        # This must return a gradient for each input argument to forward().
        return grad_input

# Create a user-friendly handle for our function
cube_function = Cube.apply

# --- Let's test it ---
x = torch.tensor(2.0, requires_grad=True)
y = cube_function(x) # y = 2^3 = 8
print(f"Output y: {y}")

# Now, let's backpropagate
y.backward()

# The gradient should be 3 * x^2 = 3 * 2^2 = 12
print(f"Gradient at x: {x.grad}")
```

B Triton Memory Operations

Efficiently managing data movement between the GPU's large but slow High-Bandwidth Memory (HBM) and its small but fast on-chip SRAM is the most critical aspect of writing high-performance kernels. The `tl.store` and `tl.load` functions are the primary tools for this task.

B.1 `tl.program_id`: Work Assignment

The `tl.program_id(axis)` function is the primary mechanism for parallelization. A kernel is launched on a 1D, 2D, or 3D **grid** of **program instances**. This function returns the unique coordinate (ID) for the current program instance along a specific axis of that grid. This ID is then used to map the program instance to a specific slice of work (e.g., a specific row or head).

```
import torch
import triton
import triton.language as tl

@triton.jit
def get_program_ids_kernel(output_ptr, M, N):
    # Get the unique 2D coordinate for this program instance
    pid_m = tl.program_id(axis=0)  # The row index in the grid (0 to M-1)
    pid_n = tl.program_id(axis=1)  # The column index in the grid (0 to N-1)

    # Calculate the linear memory offset for this 2D coordinate
    offset = pid_m * N + pid_n

    # Store a unique value based on the ID, e.g., 1002 for (1, 2)
    unique_id = pid_m * 1000 + pid_n
    tl.store(output_ptr + offset, unique_id)

# --- Verification ---
```

```
M, N = 4, 5
output = torch.empty(M * N, device='cuda', dtype=torch.int32)

# Launch the kernel on a 2D grid of size (M, N)
get_program_ids_kernel[(M, N)](output, M, N)

# Reshape the output to see the 2D grid of unique IDs
print(output.reshape(M, N))
# Expected output:
# tensor([[   0,    1,    2,    3,    4],
#         [1000, 1001, 1002, 1003, 1004],
#         [2000, 2001, 2002, 2003, 2004],
#         [3000, 3001, 3002, 3003, 3004]], device='cuda:0', dtype=torch.int32)
```

B.2 `tl.store`: Writing Data to Main Memory

The `tl.store` function writes data from a program instance's fast SRAM (registers) back to the main HBM. This is typically the final step of a kernel, used to save the computed result to the output tensor.

```
# Example: Safely storing a block of computed results
out_offsets = tl.arange(0, BLOCK_SIZE)
# `pid` is the program ID, `N` is the total valid size of the output dimension
mask = (pid * BLOCK_SIZE + out_offsets) < N
tl.store(Output_ptr + out_offsets, final_results, mask=mask)
```

Let's break down the arguments:

- `pointer` **(e.g.,** `Output_ptr + out_offsets`**):** The destination memory address(es) in HBM. This is calculated from a base pointer and a set of offsets, specifying the exact location(s) to write to.
- `value` **(e.g.,** `final_results`**):** The data to be written. This is a variable or block of data currently held in the program's SRAM.
- `mask` **(Optional but recommended):** A boolean tensor of the same shape as `value`. The store operation is only performed for elements where the mask is `True`. Using a mask is essential for safety, as it prevents the kernel from writing data outside the allocated bounds of the output tensor. An out-of-bounds write can silently corrupt other data in memory, leading to extremely difficult-to-debug errors.

B.3 `tl.load`: Reading Data from Main Memory

The `tl.load` function reads data from HBM into SRAM, making it available for the compute units to perform calculations. This is typically the first step in any kernel that processes input data.

```
# Example: Safely loading a block of input data
in_offsets = tl.arange(0, BLOCK_SIZE)
mask = in_offsets < D # `D` is the valid size of the input dimension
x_tile = tl.load(X_ptr + in_offsets, mask=mask, other=0.0)
```

Let's break down its arguments:

- `pointer` **(e.g.,** `X_ptr + in_offsets`**):** The source memory address(es) in HBM from which to read. By providing a block of addresses, we can load an entire tile of data in a single, efficient operation.
- `mask`**:** Similar to its role in `tl.store`, this boolean mask prevents the kernel from reading memory outside the valid bounds of the input tensor. An out-of-bounds read would otherwise cause a GPU crash.

- `other` **(Optional):** This parameter specifies a fallback value to use for elements where the mask is `False`. This "padding" ensures that the loaded tile in SRAM has a full, regular shape (e.g., 64 elements), which is often required for hardware efficiency, even if the source data was smaller. It fills the invalid parts of the tile with a safe, known value like 0.0.

C Dropout in Triton

Adapted from Triton Tutorials: Dropout

Dropout is a regularization technique that randomly sets a fraction of input units to 0 during training to prevent overfitting. The standard implementation, used by frameworks like PyTorch, operates in two stages: first, a random boolean mask tensor is created, and second, this mask is used to zero out elements of the input tensor.

The following script contains the complete implementation in Triton and the equivalent PyTorch.

```python
import torch
import triton
import triton.language as tl

@triton.jit
def _dropout_kernel(
    x_ptr,       # pointer to the input
    x_keep_ptr, # pointer to a mask of 0s and 1s
    output_ptr, # pointer to the output
    n_elements, # number of elements in the `x` tensor
    p,           # probability of an element being zeroed
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    # Load data
    x = tl.load(x_ptr + offsets, mask=mask)
    x_keep = tl.load(x_keep_ptr + offsets, mask=mask)
    # Apply dropout using the pre-computed mask
    # Note: tl.where(condition, val_if_true, val_if_false)
    output = tl.where(x_keep, x / (1 - p), 0.0)
    # Write-back output
    tl.store(output_ptr + offsets, output, mask=mask)

def dropout(x, x_keep, p):
    output = torch.empty_like(x)
    assert x.is_contiguous()
    n_elements = x.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
    _dropout_kernel[grid](x, x_keep, output, n_elements, p, BLOCK_SIZE=1024)
    return output

# --- Verification Script ---
x = torch.randn(size=(10,), device='cuda')
p = 0.5

# 1. Create the random mask, just like standard PyTorch does.
x_keep_mask = (torch.rand_like(x) > p)
x_keep_int = x_keep_mask.to(torch.int32) # Convert bool to int for the kernel

# 2. Run our Triton kernel with the pre-computed mask.
```

```
triton_output = dropout(x, x_keep=x_keep_int, p=p)

# 3. Perform the equivalent operation in pure PyTorch.
# This is the logic of torch.nn.functional.dropout.
pytorch_output = torch.where(x_keep_mask, x / (1 - p), 0.0)

# 4. Verify that the results are identical.
print(f"Results are close: {torch.allclose(triton_output, pytorch_output)}")
```

While the PyTorch verification code looks simple, it involves several distinct operations, each requiring a separate program (kernel) to be launched on the GPU and involving data movement to and from the GPU's main memory (HBM).

1. `torch.rand_like(x)`: Launch a kernel to generate random numbers and write them to HBM.
2. `> p`: Launch a second kernel to read the random numbers, perform the comparison, and write a new boolean mask to HBM.
3. `torch.where(...)`: Launch a third kernel to read the input `x` and the mask from HBM, perform the operation, and write the final output to HBM.

The Triton kernel is more optimal because it performs **kernel fusion**. It combines the core logic into a single operation. In one pass, the kernel loads a block of `x` and a block of the `x_keep` mask into fast on-chip SRAM, performs the conditional logic and scaling, and writes the final result back to HBM. This reduces performance costs from both **kernel launch overhead** and **memory traffic**.

D High-Level Matrix Multiplication in Triton

Adapted from Triton Tutorials: Matrix Multiplication

Matrix multiplication is a fundamental building block for deep learning workloads. In high-performance computing, this operation is often referred to as **GEMM** (General Matrix to Matrix Multiplication), a standard routine from libraries like BLAS and cuBLAS. While these vendor-optimized libraries are fast, Triton allows for the creation of custom, high-performance matrix multiplication kernels that can be fused with other operations (e.g., activation functions).

The Blocked Algorithm

The core idea behind an efficient Triton matrix multiplication is to break the large operation into smaller, manageable blocks. The algorithm can be described with the following pseudo-code, which multiplies a (M, K) by a (K, N) matrix:

```
# Do in parallel
for m in range(0, M, BLOCK_SIZE_M):
  # Do in parallel
  for n in range(0, N, BLOCK_SIZE_N):
    # Each program instance computes one block of C
    acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=float32)
    # The inner loop over the K dimension is sequential
    for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
      acc += dot(a, b)
    C[m : m+BLOCK_SIZE_M, n : n+BLOCK_SIZE_N] = acc
```

In Triton, the two outer loops marked '# Do in parallel' are handled by the grid of program instances. Each iteration of this doubly-nested loop is performed by a dedicated program on the GPU. The inner loop over the 'K' dimension remains sequential within each of those program instances.

Pointer Arithmetic. The main challenge in implementing the blocked algorithm is calculating the correct memory addresses for the tiles of `A` and `B`. The kernel uses its program ID, the block sizes, and the tensor strides to do this. We can define a block of pointers for a tile of matrix `A` as follows:

```
# The kernel signature now includes stride arguments passed from the host
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr,
                  stride_am, stride_ak,
                  stride_bk, stride_bn,
                  ...):
    # Get the program's unique (m, n) coordinate in the 2D grid
    pid_m = tl.program_id(axis=0)
    pid_n = tl.program_id(axis=1)


    # Create the offsets for the initial block of A (where k=0)
    offs_am = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_k = tl.arange(0, BLOCK_SIZE_K)

    # Create a 2D block of pointers using the strides
    # stride_am is the stride for A along the M dimension (row stride)
    # stride_ak is the stride for A along the K dimension (column stride)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)

    # Create the offsets for the initial block of B (where k=0)
    offs_bn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    # Note that B's dimensions are (K, N)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
```

Strides. To calculate memory addresses correctly, a kernel needs to know the tensor's **strides**. A stride is the number of elements one must jump in memory to get to the next element in a given dimension. For a 2D matrix, the address of element `X[i, j]` is `base_ptr + i*stride_row + j*stride_col`. These stride values are passed from the host (Python) to the Triton kernel as arguments. In PyTorch, you can get them using `tensor.stride(dim)`.

Updating Pointers in the Inner Loop. Inside the sequential loop over the `K` dimension, we don't need to recalculate the full pointer block. We can simply advance the existing pointers by the block size along the `K` dimension. This is a very efficient operation.

```
# Inside the `for k in range(...)` loop
for k in range(0, K, BLOCK_SIZE_K):
    # Load tiles using the current a_ptrs and b_ptrs
    a_tile = tl.load(a_ptrs, mask=...)
    b_tile = tl.load(b_ptrs, mask=...)

    # Perform computation
    acc += tl.dot(a_tile, b_tile)

    # Advance the pointers for the next iteration using the K-dimension stride
    a_ptrs += BLOCK_SIZE_K * stride_ak
    b_ptrs += BLOCK_SIZE_K * stride_bk
```

This combination of a parallel grid, tiled loading via block pointers, and sequential accumulation in SRAM is the foundation of high-performance matrix multiplication in Triton.

L2 Cache Optimizations. The order in which the blocks of the output matrix `C` are computed significantly affects performance. A simple row-major ordering of the program grid is often inefficient.

The problem is that this simple ordering leads to poor data reuse in the GPU's L2 cache. For each block of `C`, a program must loop through the entire K-dimension, loading tiles from matrix `B`. In a row-major schedule, by the time the next program instance (for the next row) needs to access the first tile of `B`, it has likely been evicted from the cache, forcing a slow reload from HBM.

A better solution is a **grouped ordering**. This approach computes several blocks of `C` within the same "group" of rows before moving to the next group. This ensures that when a tile of matrix `B` is loaded into the L2 cache, it is reused by multiple program instances before it can be evicted. This increases the L2 cache hit rate and improves performance.

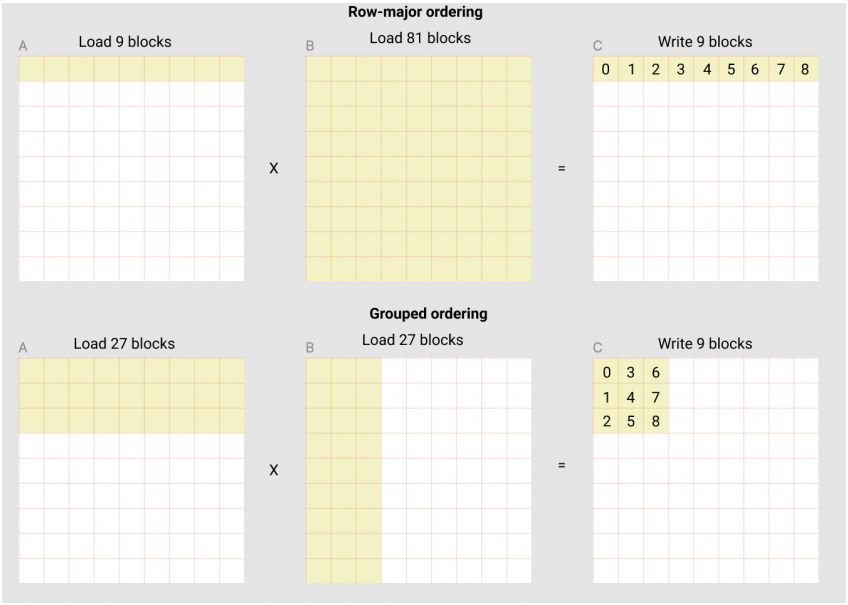

Figure 7: Comparison of memory access patterns for row-major vs. grouped ordering.

For example in Figure 7, in a matmul where each matrix is 9 blocks by 9 blocks, computing the output in row-major ordering requires loading 90 blocks into SRAM to compute the first 9 output blocks. With grouped ordering, only 54 blocks are needed.