Flash Attention Homework

GPU Programming for LLMs

Contents

1 Part 1: Fused Softmax-Matmul in Triton [8 points]

In this first part, you will implement a **fused softmax-matmul** kernel in Triton. This operation computes `softmax(X) @ V` without materializing the full softmax output matrix, which is a key building block for Flash Attention.

1.1 Objectives

- Extend the online softmax algorithm to include matrix multiplication
- Implement a memory-efficient fused kernel in Triton
- Benchmark your implementation against the naive PyTorch version

1.2 Background

1.2.1 From Online Softmax to Fused Softmax-Matmul

During the course, we implemented the **online softmax** algorithm in Triton. The key insight was that we can compute softmax row by row using running statistics:

```
For each block j:
    m_j = max(m_{j-1}, max(X_j))           # running max
    l_j = l_{j-1} * exp(m_{j-1} - m_j) + sum(exp(X_j - m_j))   # running sum
```

Now we extend this to **simultaneously** compute the matrix multiplication with V:

`output = softmax(X) @ V`

where: - `X` has shape `(batch, d1, d2)` — the attention scores - `V` has shape `(batch, d2, d3)` — the values - `output` has shape `(batch, d1, d3)`

1.2.2 The Fused Algorithm

The key observation is that we can accumulate the output incrementally as we process blocks of X and V:

```
For each block j:
    # Update running max and sum (same as online softmax)
    m_j = max(m_{j-1}, max(X_j))
    exp_X_j = exp(X_j - m_j)
    l_j = l_{j-1} * exp(m_{j-1} - m_j) + sum(exp_X_j)

    # Update output accumulator with rescaling
    scale = (l_{j-1} / l_j) * exp(m_{j-1} - m_j)
    normalized_j = exp_X_j / l_j
    O_j = O_{j-1} * scale + normalized_j @ V_j
```

This avoids storing the full `(d1, d2)` softmax matrix in memory.

1.3 Your Task

1.3.1 File Structure

The implementation of the **online softmax** (seen during the course) is given in the following files:

```
online_softmax/
   online_softmax.py
```

with the associated tests provided in:

```
tests
   test_online_softmax.py
```

You can run the tests for the **online softmax** with:

```
pytest tests/test_online_softmax.py -v
```

To complete this part 1, you will need to complete the following files:

```
softmax_matmul/
   softmax_matmul.py

benchmarking/
   bench_softmax_matmul.py
```

The file `sofmmax_matmul/softmax_matmul.py` will contain your Fused Softmax-Matmul Implementation (Part 1.A) and the file `benchmarking/bench_softmax_matmul.py` will contain the code for benchmarking your implementation (Part 1.B).

1.3.2 Part 1.A: Fused Softmax-Matmul Implementation [4 points]

The `softmax_matmul.py` already contains:

1.3.2.1 1. Reference Implementation

```python
def softmax_mult(x, V, dim=-1):
    """
    Reference implementation using PyTorch.

    Args:
        x: Input tensor of shape (batch, d1, d2)
        V: Value tensor of shape (batch, d2, d3)
        dim: Dimension for softmax (default: -1)

    Returns:
        Output tensor of shape (batch, d1, d3)
    """
    return F.softmax(x, dim=dim) @ V
```

Your `softmax_matmul.py` must implement:

1.3.2.2 2. Triton Kernel

```python
@triton.jit
def fused_softmax_kernel(
    x_ptr,
    V_ptr,
    output_ptr,
    # ... strides and dimensions
    d1: tl.constexpr,
    d2: tl.constexpr,
    d3: tl.constexpr,
    BLOCK_1: tl.constexpr,
    BLOCK_2: tl.constexpr,
):
    """
    Fused softmax-matmul kernel.

    Computes softmax(X) @ V for a block of rows.
    """
    # Your code here
```

1.3.2.3 3. Wrapper Function

```
def fused_softmax(x, V, BLOCK_1=16, BLOCK_2=16):
    """
    Compute fused softmax(x) @ V using Triton.

    Args:
        x: Input tensor of shape (batch, d1, d2)
        V: Value tensor of shape (batch, d2, d3)
        BLOCK_1: Block size for d1 dimension
        BLOCK_2: Block size for d2 dimension

    Returns:
        Output tensor of shape (batch, d1, d3)
    """
    # Your code here
```

1.4 Implementation Hints

If you already implemented this during the practicals, you can just copy and paste your code from the Jupyter Notebook to the file.

1.4.1 Starting Point

Use the `online_softmax` implementation from the FlashAttention_empty.ipynb notebook as a template. The main differences are:

1. **Additional input**: You now have a second tensor `V`
2. **Output shape**: Output is `(batch, d1, d3)` instead of `(batch, d1, d2)`
3. **Accumulator**: You need to maintain an output accumulator of shape `(BLOCK_1, d3)`

1.4.2 Block Pointer Setup

You'll need three block pointers: - `x_block`: iterates over blocks of X in the d2 dimension - `V_block`: iterates over corresponding blocks of V in the d2 dimension - `output_block`: stores the final result (no iteration needed)

1.4.3 Numerical Stability

Use `float32` accumulators for: - `m_prev`: running max - `l_prev`: running sum - `out_prev`: output accumulator

1.4.4 GPU Compatibility Note

There's a known bug with Triton on Turing GPUs (T4, RTX 8000) that requires explicit casting for dot products. If you're on Turing:

```
# Cast to float16 for dot product
... tl.dot(a.to(tl.float16), b.to(tl.float16)).to(tl.float32)
```

On Hopper (H100), you can skip the casting.

1.5 Testing Your Implementation

Run the tests to verify correctness:

```
pytest tests/test_softmax_matmul.py -v
```

The tests check: - Correctness against the reference implementation - Output shape - Numerical stability with large values - Various tensor dimensions

---

1.5.1 Part B: Benchmarking [4 points]

Once your implementation passes the tests, write a benchmark to compare performance.

1.5.1.1 File to Create Create `benchmarking/bench_softmax_matmul.py` that compares your Triton implementation `fused_softmax` against the PyTorch version `softmax_mult`.

1.5.1.2 Benchmark Requirements Your benchmark should:

1. **Compare both implementations**: `softmax_mult` (PyTorch) vs `fused_softmax` (Triton)
2. **Vary the sequence length** (d2): Test with increasing values, `[64, 128, 256, 512, 1024, 2048, 4096, 8192]`
3. **Test different block sizes**: Try `B=BLOCK_1=BLOCK_2` values `[16, 32, 64]`
4. **Measure execution time**: Use CUDA events for accurate GPU timing:

```
start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)

torch.cuda.synchronize()
start.record()
# ... run your function ...
end.record()
torch.cuda.synchronize()

elapsed_ms = start.elapsed_time(end)
```

5. **Include warmup iterations**: Run a few iterations before timing to allow for JIT compilation
6. **Report results**: Print a summary table and save your results to CSV in `outputs/softmax_matmul_benchmark.csv`

The format of your CSV should have the following columns:

Column	Type	Description
`batch_size`	int	Batch size used (16)
`d1`	int	First dimension (2048)
`d2`	int	Sequence length being varied
`d3`	int	Third dimension (512)
`triton`	bool	`True` for Triton implementation, `False` for PyTorch
`BLOCK`	Int64 (nullable)	Block size used (`16`, `32`, `64`), or `<NA>` for PyTorch
`forward_ms_mean`	float (nullable)	Mean forward pass time in milliseconds
`forward_ms_std`	float (nullable)	Standard deviation of forward pass time
`forward_peak_MiB`	float (nullable)	Peak GPU memory usage in MiB

Note: Nullable columns will contain `None`/`<NA>` values for configurations that failed due to OOM or incompatible block sizes.

1.5.1.3 Required Parameters

```
device = "cuda"
dtype = torch.float32
batch_size = 16
nb_warmup = 10
nb_passes = 100
d1 = 2048
d2 = [64, 128, 256, 512, 1024, 2048, 4096, 8192]
d3 = 512
B = [16, 32, 64]
```

1.5.1.4 Running Your Benchmark

```
python -m benchmarking.bench_softmax_matmul
```

1.5.1.5 Expected Results Your fused kernel should: - Be **faster** than the naive implementation for large d2 - Use **less memory** (no intermediate softmax matrix stored)

1.5.1.6 Handling Errors Some configurations will fail due to resource limits or incompatible parameters. Your benchmark should:

1. **Run all configurations** — Don't stop at the first error
2. **Catch and log errors** — Record which configs failed and why
3. **Clean up GPU memory** — Call `torch.cuda.empty_cache()` after OOM errors

Here's a pattern for robust benchmarking:

```
from triton.compiler.errors import CompileTimeAssertionFailure
from triton.runtime.errors import OutOfResources

results = []

for d2_val in d2_values:
    for BLOCK in block_sizes:
        try:
            # Run benchmark for this configuration
            result = run_benchmark_config(d2_val, BLOCK, ...)
            results.append(result)

        except (RuntimeError, OutOfResources) as e:
            err_str = str(e).lower()
            if "out of memory" in err_str:
                print(f"OOM for d2={d2_val}, BLOCK={BLOCK}")
                torch.cuda.empty_cache()  # Free memory before next config
                # Record as failed result with None values
                results.append({
                    "d2": d2_val,
                    "BLOCK": BLOCK,
                    "forward_ms": None,  # Mark as failed
                })
            else:
                raise  # Re-raise unexpected errors

        except CompileTimeAssertionFailure:
            # Block size incompatible with dimensions
```

```
            print(f"Skipping: BLOCK={BLOCK} incompatible with d2={d2_val}")
            results.append({
                "d2": d2_val,
                "BLOCK": BLOCK,
                "forward_ms": None,
            })
```

Common errors you might encounter:

Error	Cause	Solution
`RuntimeError: CUDA out of memory`	Tensor too large for GPU	Catch and continue, log as failed
`OutOfResources`	Triton kernel needs too many registers	Catch and continue
`CompileTimeAssertionFailure`	`d1 % BLOCK_1 != 0` or `d2 % BLOCK_2 != 0`	Skip this config

Tip: Use `pandas.DataFrame` to collect results and display them nicely:

```
import pandas as pd

df = pd.DataFrame(results)
df["BLOCK"] = df["BLOCK"].astype("Int64")  # Nullable int for None values
print(df.to_string())
df.to_csv("outputs/softmax_matmul_benchmark.csv", index=False)
```

2 Part 2: Flash Attention in PyTorch [8 points]

In this part, you will implement **Flash Attention** in pure PyTorch as a custom `torch.autograd.Function` with explicit forward and backward passes. This prepares you for the Triton implementation by understanding the algorithm deeply.

2.1 Objectives

- Implement the Flash Attention forward pass using the tiling algorithm
- Implement the backward pass with gradient computation for Q, K, and V
- Wrap your implementation as a reusable PyTorch module
- Understand memory-efficient attention without relying on Triton

---

2.2 Background

2.2.1 Standard Attention vs Flash Attention

Standard Attention computes:

$$X = \frac{QK^\top}{\sqrt{d}}, \qquad P = \text{softmax}(X), \qquad O = PV$$

This requires materializing the full $(N \times N)$ attention matrix, which has $O(N^2)$ memory complexity.

Flash Attention computes the same result without materializing the full score matrix by processing attention in **tiles (blocks)** and computing softmax **online** using a running max and running normalizer.

Memory becomes $O(N)$ instead of $O(N^2)$.

2.2.2 The Algorithm: Almost-Flash-Attention

Here, we recall the algorithm presented in FlashAttention_empty.ipynb computing `softmax(x) @ V` in a fused manner.

Input: $x_1, \dots, x_{d_2} \in \mathbb{R}^{d_1}$ and $v \in \mathbb{R}^{d_2 \times d_3}$

Output: $o_1, \dots, o_{d_2} \in \mathbb{R}^{d_3}$

One pass:

$$m_0 = -\infty \tag{1}$$
$$\ell'_0 = 0 \tag{2}$$
$$o'_0 = 0 \tag{3}$$
$$\text{for } i = 1, \dots, d_2 \tag{4}$$
$$m_i \leftarrow \max(m_{i-1}, x_i) \tag{5}$$
$$\ell'_i \leftarrow \ell'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i} \tag{6}$$
$$o'_i \leftarrow e^{m_{i-1} - m_i} \frac{\ell'_{i-1}}{\ell'_i} \odot o'_{i-1} + \frac{e^{x_i - m_i}}{\ell'_i} \otimes v[i, :] \tag{7}$$

2.3 Flash Attention

2.3.0.1 Setup and notation We use:

- $Q \in \mathbb{R}^{d_1 \times d}$
- $K \in \mathbb{R}^{d_2 \times d}$
- $V \in \mathbb{R}^{d_2 \times d_3}$

Scores:

$$X = \frac{QK^\top}{\sqrt{d}} \quad \in \mathbb{R}^{d_1 \times d_2}$$

Probabilities (row-wise softmax):

$$P = \text{softmax}(X)$$

Output:

$$O = PV \quad \in \mathbb{R}^{d_1 \times d_3}$$

In practice, we have $d_1 = d_2 = d_3$, but we find it easier to present the algorithm where each dimension is identified by its size.

2.3.1 Forward Pass — FlashAttention

We want to apply the above algorithm with $x = \frac{QK^T}{\sqrt{d}}$. We use the following notations, for $j \in [d_1]$ and $i \in [d_2]$, $Q^{(j)} = Q[j, :] \in \mathbb{R}^d$, $K_i = K[i, :], V_i = V[i, :] \in \mathbb{R}^d$ and $x_{ij} = x_i^{(j)} = \frac{Q^{(j)}(K_i)^T}{\sqrt{d}}$ and $L \in \mathbb{R}^{d_1}$, with $L^{(j)} = \sum_{i \le d_2} e^{x_i^{(j)} - \max_{i \le d_2} x_i^{(j)}}$.

$$\text{for } j = 1, \dots, d_1 \text{ (outer loop)} \tag{8}$$
$$m_0 = -\infty \tag{9}$$
$$\ell_0^{(j)} = 0 \tag{10}$$
$$o_0^{(j)} = 0 \tag{11}$$
$$\text{for } i = 1, \dots, d_2 \text{ (inner loop)} \tag{12}$$
$$x_i^{(j)} = \frac{Q^{(j)}(K_i)^T}{\sqrt{d}} \tag{13}$$
$$m_i \leftarrow \max(m_{i-1}, x_i^{(j)}) \tag{14}$$
$$\ell_i^{(j)} \leftarrow \ell_{i-1}^{(j)} e^{m_{i-1} - m_i} + e^{x_i^{(j)} - m_i} \tag{15}$$
$$o_i^{(j)} \leftarrow e^{m_{i-1} - m_i} \frac{\ell_{i-1}^{(j)}}{\ell_i^{(j)}} \odot o_{i-1}^{(j)} + \frac{e^{x_i - m_i}}{\ell_i^{(j)}} \otimes V_i \tag{16}$$

Here, we have $x_i^{(j)}, m_i, \ell_i^{(j)} \in \mathbb{R}^{d_1}$ and $o_i^{(j)} \in \mathbb{R}^{d_3}$. Moreover, we have

$$o_{d_2}^{(j)} = O[j, :] \text{ and, } \ell_{d_2}^{(j)} = \sum_{i \le d_2} e^{x_i^{(j)} - \max_{i \le d_2} x_i^{(j)}} = L^{(j)} \tag{17}$$

Hence, this algorithm computes O and L.

In your implementation, you will work with tiles corresponding to blocks in the dimensions of j and i (for simplicity take the same block dimension in each direction).

2.3.2 Backward Pass Derivation for Flash Attention

2.3.2.1 Goal Given upstream gradient

$$dO = \frac{\partial \mathcal{L}}{\partial O},$$

compute dQ, dK, dV.

2.3.2.2 Computational graph We differentiate **backwards** through:

$$Q, K \to X \to P \to O$$

So we apply the chain rule in three steps:

1. $O = PV$
2. $P = \text{softmax}(X)$
3. $X = \frac{QK^\top}{\sqrt{d}}$

2.3.2.3 Step A — Gradient through $O = PV$ Forward definition:

$$O_{i\ell} = \sum_{j=1}^{d_2} P_{ij} V_{j\ell}$$

A.1 — gradient w.r.t. V:

$$\frac{\partial O_{i\ell}}{\partial V_{j\ell}} = P_{ij}$$

Thus:

$$dV_{j\ell} = \sum_{i=1}^{d_1} P_{ij}\, dO_{i\ell}$$

Matrix form:

$$\boxed{dV = P^\top dO}$$

A.2 — gradient w.r.t. P:

$$\frac{\partial O_{i\ell}}{\partial P_{ij}} = V_{j\ell}$$

Thus:

$$dP_{ij} = \sum_{\ell=1}^{d_3} dO_{i\ell}\, V_{j\ell}$$

Matrix form:

$$\boxed{dP = dO\, V^\top}$$

2.3.2.4 Step B — Gradient through softmax Softmax is applied **row-wise**.

For each row i:

$$P_{ij} = \frac{e^{X_{ij}}}{\sum_k e^{X_{ik}}}$$

Let $Z_i = \sum_k e^{X_{ik}}$.

B.1 — softmax derivative:

Two cases:

If $j = k$:

$$\frac{\partial P_{ij}}{\partial X_{ij}} = P_{ij}(1 - P_{ij})$$

If $j \neq k$:

$$\frac{\partial P_{ij}}{\partial X_{ik}} = -P_{ij}P_{ik}$$

B.2 — compact Jacobian form:

$$\frac{\partial P_{ij}}{\partial X_{ik}} = P_{ij}(\delta_{jk} - P_{ik})$$

B.3 — apply chain rule:

$$dX_{ik} = \sum_j dP_{ij} \frac{\partial P_{ij}}{\partial X_{ik}}$$

Substitute:

$$dX_{ik} = \sum_j dP_{ij} P_{ij}(\delta_{jk} - P_{ik})$$

Split:

$$dX_{ik} = P_{ik}\, dP_{ik} - P_{ik} \sum_j P_{ij}\, dP_{ij}$$

Define row scalar:

$$D_i = \sum_j P_{ij}\, dP_{ij}$$

So:

$$\boxed{dX_{ik} = P_{ik}(dP_{ik} - D_i)}$$

Matrix form:

$$\boxed{dX = P \odot (dP - D)}$$

where $D \in \mathbb{R}^{d_1 \times 1}$ is broadcast across columns.

2.3.2.5 Useful simplification (used in FlashAttention) We can avoid explicitly computing $P \odot dP$ by deriving an equivalent expression for D.

Recall that: - $D_i = \sum_j P_{ij}\, dP_{ij}$ (from the softmax backward) - $dP = dO\, V^\top$, so $dP_{ij} = \sum_\ell dO_{i\ell}\, V_{j\ell}$ - $O = PV$, so $O_{i\ell} = \sum_j P_{ij}\, V_{j\ell}$

Substituting the expression for dP_{ij} into D_i:

$$D_i = \sum_j P_{ij}\, dP_{ij} = \sum_j P_{ij} \sum_\ell dO_{i\ell}\, V_{j\ell}$$

Rearranging the sums:

$$D_i = \sum_\ell dO_{i\ell} \sum_j P_{ij}\, V_{j\ell}$$

But $\sum_j P_{ij}\, V_{j\ell} = O_{i\ell}$ (by definition of O), so:

$$\boxed{D_i = \sum_\ell O_{i\ell}\, dO_{i\ell}}$$

In matrix form:

$$D = \text{rowsum}(O \odot dO)$$

Why this matters for FlashAttention: This identity allows us to compute D directly from the output O and its gradient dO, without needing to explicitly compute $P \odot dP$ first.

Note that we still need P in the backward pass (for $dV = P^\top dO$ and $dX = P \odot (dP - D)$). The key insight of FlashAttention is that instead of **storing** the full $O(N^2)$ matrix P from the forward pass, we **recompute** it on-the-fly during the backward pass using the saved log-sum-exp values L:

$$P = e^{S-L}, \quad \text{where } S = \frac{QK^\top}{\sqrt{d}}$$

This recomputation is done block by block, so we never materialize the full P matrix in memory.

2.3.2.6 Step C — Gradient through $X = \frac{QK^\top}{\sqrt{d}}$ Forward definition:

$$X_{ij} = \frac{1}{\sqrt{d}} \sum_{r=1}^{d} Q_{ir} K_{jr}$$

C.1 — gradient w.r.t. Q:

$$\frac{\partial X_{ij}}{\partial Q_{ir}} = \frac{K_{jr}}{\sqrt{d}}$$

Thus:

$$dQ_{ir} = \frac{1}{\sqrt{d}} \sum_{j} dX_{ij} K_{jr}$$

Matrix form:

$$\boxed{dQ = \frac{dX\, K}{\sqrt{d}}}$$

C.2 — gradient w.r.t. K:

$$\frac{\partial X_{ij}}{\partial K_{jr}} = \frac{Q_{ir}}{\sqrt{d}}$$

Thus:

$$dK_{jr} = \frac{1}{\sqrt{d}} \sum_{i} dX_{ij} Q_{ir}$$

Matrix form:

$$\boxed{dK = \frac{dX^\top Q}{\sqrt{d}}}$$

2.3.2.7 Final backward algorithm Given dO:

1. Matmul backward:

$$dV = P^\top dO$$

$$dP = dO\, V^\top$$

2. Softmax backward:

$$D = \text{rowsum}(O \odot dO)$$

$$dX = P \odot (dP - D)$$

3. Score backward:

$$dQ = \frac{dX\,K}{\sqrt{d}}$$

$$dK = \frac{dX^\top Q}{\sqrt{d}}$$

2.4 Your Task

2.4.1 File Structure

Create the following file:

```
flash_attention/
    flash_attention.py
```

2.4.2 Implementation Requirements

Your `flash_attention/flash_attention.py` must implement:

2.4.2.1 1. Custom Autograd Function (Forward Pass)

```python
class FlashAttentionPytorch(torch.autograd.Function):
    @staticmethod
    def forward(ctx, Q, K, V, is_causal=False):
        """
        Flash Attention forward pass using tiled online softmax.

        Args:
            ctx: Context object for saving tensors for backward
            Q: Query tensor of shape (batch, seq_len, head_dim)
            K: Key tensor of shape (batch, seq_len, head_dim)
            V: Value tensor of shape (batch, seq_len, head_dim)
            is_causal: Whether to apply causal masking (default: False)

        Returns:
            O: Output tensor of shape (batch, seq_len, head_dim)
        """
        # Your code here

        # Use tiled computation with online softmax:
        # - Outer loop over query blocks
        # - Inner loop over key/value blocks
        # - Maintain running (m, l, O) accumulators
        # - Handle causal masking when is_causal=True

        # IMPORTANT: Save tensors needed for backward
        # ctx.save_for_backward(Q, K, V, L, O)
        # ctx.is_causal = is_causal
        # ctx.sqrt_d = sqrt_d
```

```
    @staticmethod
    def backward(ctx, dO):
        """
        Attention backward pass.

        Args:
            ctx: Context object with saved tensors
            dO: Gradient of loss w.r.t. output

        Returns:
            dQ: Gradient w.r.t. Q
            dK: Gradient w.r.t. K
            dV: Gradient w.r.t. V
            None: No gradient for is_causal
        """
        Q, K, V, L, O = ctx.saved_tensors
        dQ, dK, dV, _ = attention_backward_impl(
            Q, K, V, L, O, dO, ctx.sqrt_d, ctx.is_causal
        )
        return dQ, dK, dV, None
```

2.4.2.2 2. Backward Pass Implementation

```
def attention_backward_impl(Q, K, V, L, O, dO, sqrt_d, is_causal):
    """
    Backward pass implementation for Flash Attention.

    Uses standard attention gradient formulas with recomputation of P
    from saved log-sum-exp values L.

    Args:
        Q: Query tensor of shape (batch, seq_q, d)
        K: Key tensor of shape (batch, seq_k, d)
        V: Value tensor of shape (batch, seq_k, d)
        L: Log-sum-exp values from forward pass, shape (batch, seq_q)
        O: Output from forward pass, shape (batch, seq_q, d)
        dO: Gradient of loss w.r.t. output, shape (batch, seq_q, d)
        sqrt_d: Square root of head dimension (for scaling)
        is_causal: Whether to apply causal masking

    Returns:
        dQ: Gradient w.r.t. Q
        dK: Gradient w.r.t. K
        dV: Gradient w.r.t. V
        None: Placeholder for compatibility
    """
    # Your code here
    #
    # Steps:
    # 1. Compute D = rowsum(O   dO)
    # 2. Recompute S = Q @ K^T / sqrt(d)
    # 3. Apply causal mask if is_causal
    # 4. Recompute P = exp(S - L)
```

```
    # 5. Compute dV = P^T @ dO
    # 6. Compute dP = dO @ V^T
    # 7. Compute dS = P   (dP - D)
    # 8. Compute dQ = dS @ K / sqrt(d)
    # 9. Compute dK = dS^T @ Q / sqrt(d)
```

2.4.2.3 3. Causal Masking Your implementation must support **causal attention** when `is_causal=True`. In causal attention, position i can only attend to positions $j \leq i$.

In the forward pass: - Skip key/value blocks that are entirely beyond the current query positions (early exit optimization) - For blocks that partially overlap, apply a causal mask: `python # For query positions [offset_i : end_i] and key positions [offset_j : end_j] mask = torch.arange(offset_i, end_i, device=device).unsqueeze(-1) >= \ torch.arange(offset_j, end_j, device=device).unsqueeze S = torch.where(mask, S, torch.tensor(float("-inf"), device=device, dtype=dtype))`

In the backward pass: - Apply the same causal mask when recomputing attention scores: `python if is_causal: causal_mask = torch.triu(torch.ones(seq_q, seq_k, device=Q.device, dtype=torch.bool), diagonal=1) S = S.masked_fill(causal_mask, float("-inf"))`

2.5 Implementation Hints

2.5.1 Forward Pass Tips

1. **Loop structure**: Use nested loops over query blocks (outer) and key/value blocks (inner). Take blocks of sizes $16x16$
2. **Handling non-divisible sequences**: If `seq_len % block_size != 0`, handle the last partial block:

   ```
   for i in range(0, seq_len, block_size):
       block_end = min(i + block_size, seq_len)
       Q_block = Q[:, :, i:block_end, :]
       # ...
   ```

3. **Scaling factor**: Don't forget to divide by `sqrt(d)`:

   ```
   sqrt_d = math.sqrt(head_dim)
   ```

4. **Numerical stability**: Always subtract the max before taking exp.
5. **Save for backward**: Store `L` (log-sum-exp) per row, and store `O` at the end:

   ```
   ctx.save_for_backward(Q, K, V, L, O)
   ```

2.5.2 Backward Pass Tips

1. **Recompute, don't store**: recompute the attention scores X (without tiling, i.e. not like in the forward pass).
2. **D computation**: This term accounts for the normalization in softmax:

   ```
   D = (dO * O).sum(dim=-1, keepdim=True)
   ```

2.6 Testing Your Implementation

Run the tests to verify correctness:

```
pytest tests/test_flash_attention_pytorch.py -v
```

The tests check: - Forward pass correctness against standard attention - Backward pass correctness using `torch.autograd.gradcheck` - Output shapes - Numerical stability - Various sequence lengths and head dimensions

2.6.1 Gradient Checking

You can manually verify gradients with:

```
from torch.autograd import gradcheck

Q = torch.randn(1, 32, 16, dtype=torch.float64, requires_grad=True)
K = torch.randn(1, 32, 16, dtype=torch.float64, requires_grad=True)
V = torch.randn(1, 32, 16, dtype=torch.float64, requires_grad=True)

assert gradcheck(
    lambda q, k, v: FlashAttentionFunction.apply(q, k, v),
    (Q, K, V),
    eps=1e-6,
    atol=1e-4,
    rtol=1e-3
)
```

Note: Use `float64` for gradient checking to get accurate numerical gradients.

2.7 Expected Behavior

Your implementation should:

1. **Match standard attention output** within floating-point tolerance (rtol=1e-3 for float32)
2. **Pass gradient checks** for all inputs Q, K, V
3. **Handle various shapes**:
    - Different batch sizes
    - Different numbers of heads
    - Sequence lengths that are/aren't divisible by block_size
    - Different head dimensions
4. **Be memory efficient**: For large sequences, peak memory should be $O(N \cdot \text{block\_size})$ rather than $O(N^2)$

3 Part 3: Flash Attention in Triton [8 points (+ 4 points)]

In this final part, you will port your Flash Attention implementation to **Triton**. You have already implemented and understood the algorithm in Part 2 using PyTorch. Now you will write a GPU-optimized Triton kernel for the forward pass, leveraging the same online softmax algorithm but with explicit control over memory access patterns and parallelization.

3.1 Objectives

- Port the Flash Attention forward pass to a Triton kernel
- Use block pointers and tiled computation for efficient GPU execution
- Integrate your Triton kernel with PyTorch's autograd system
- Benchmark your implementation against PyTorch's optimized `scaled_dot_product_attention`

3.2 Prerequisites

This part assumes you have completed **Part 2: Flash Attention in PyTorch**. You should be familiar with: - The Flash Attention forward algorithm (online softmax over tiles) - The backward pass formulas (dQ, dK, dV) - Saving the log-sum-exp L for the backward pass - Causal masking implementation

Refer to Part 2 for the algorithm details if needed.

3.3 Your Task

3.3.1 File Structure

You will complete the implementation in:

```
flash_attention/
  flash_attention.py

benchmarking/
  bench_attention.py
```

Your `flash_attention.py` should contain: - `FlashAttentionPytorch`: Your implementation from Part 2 - `FlashAttentionTriton`: The new Triton-based implementation (this part)

3.3.2 Part 3.A: Triton Forward Kernel [6 points]

3.3.2.1 1. Triton Kernel Implement the forward kernel that parallelizes over query blocks and batches:

```
@triton.jit
def flash_fwd_kernel(
    Q_ptr, K_ptr, V_ptr, O_ptr, L_ptr,
    stride_qb, stride_qq, stride_qd,
    stride_kb, stride_kk, stride_kd,
    stride_vb, stride_vk, stride_vd,
    stride_ob, stride_oq, stride_od,
    stride_lb, stride_lq,
    N_QUERIES, N_KEYS, scale,
    D: tl.constexpr,
    BLOCK_Q: tl.constexpr,
    BLOCK_K: tl.constexpr,
    is_causal: tl.constexpr,
):
    """
```

```
    Flash Attention forward kernel using online softmax algorithm.

    Each program instance processes one query block for one batch element.
    """
    # Your implementation here
```

The kernel should: - Use `tl.program_id(0)` for query block index and `tl.program_id(1)` for batch index - Create block pointers for Q, K, V, O, and L - Implement the same online softmax loop as in Part 2, but using Triton primitives - Store both output `O` and log-sum-exp `L`

3.3.2.2 2. Autograd Function

Wrap your Triton kernel in an autograd function:

```
class FlashAttentionTriton(torch.autograd.Function):
    """Flash Attention using Triton kernel for forward pass."""

    @staticmethod
    def forward(ctx, Q, K, V, is_causal=False):
        """
        Forward pass using Triton kernel.

        Args:
            Q: Query tensor of shape (batch, seq_q, d)
            K: Key tensor of shape (batch, seq_k, d)
            V: Value tensor of shape (batch, seq_k, d)
            is_causal: Whether to apply causal masking

        Returns:
            Output tensor of shape (batch, seq_q, d)
        """
        # Allocate output tensors O and L
        # Choose block sizes (e.g., BLOCK_Q = BLOCK_K = 64)
        # Configure grid: (num_query_blocks, batch_size)
        # Launch flash_fwd_kernel
        # Save tensors for backward
        # Return O

    @staticmethod
    def backward(ctx, dO):
        """
        Backward pass - reuse your PyTorch implementation from Part 2.
        """
        Q, K, V, L, O = ctx.saved_tensors
        dQ, dK, dV, _ = attention_backward_impl(
            Q, K, V, L, O, dO, ctx.sqrt_d, ctx.is_causal
        )
        return dQ, dK, dV, None
```

Note: For the backward pass, you can directly reuse the `attention_backward_impl` function from Part 2. The forward pass produces the same outputs (O and L), so the backward pass is identical.

3.4 Implementation Hints

3.4.1 From PyTorch to Triton

Your Part 2 forward pass has this structure:

```
for i in range(num_query_blocks):      # Outer loop over query blocks
    for j in range(num_key_blocks):    # Inner loop over key/value blocks
        # Online softmax update
```

In Triton, the **outer loop becomes parallelization**: - Each program instance handles one query block (and one batch element) - Only the inner loop over key/value blocks remains as an explicit loop

3.4.2 Block Pointer Setup

Set up block pointers after computing batch and query block offsets:

```
def flash_fwd_kernel(
    Q_ptr, K_ptr, V_ptr,
    O_ptr, L_ptr,
    stride_qb, stride_qq, stride_qd,
    stride_kb, stride_kk, stride_kd,
    stride_vb, stride_vk, stride_vd,
    stride_ob, stride_oq, stride_od,
    stride_lb, stride_lq,
    N_QUERIES,
    N_KEYS,
    scale,
    D: tl.constexpr,
    BLOCK_Q: tl.constexpr,
    BLOCK_K: tl.constexpr,
    is_causal: tl.constexpr,
):
    """Flash Attention forward kernel using online softmax algorithm."""
    # Get program IDs
    pid_q = tl.program_id(0)
    pid_b = tl.program_id(1)

    # Create block pointers for this batch and query tile
    Q_ptr = Q_ptr + pid_b * stride_qb
    # Create block pointer for queries (fixed position)
    Q_block = tl.make_block_ptr(
        Q_ptr,
        shape=(N_QUERIES, D),
        strides=(stride_qq, stride_qd),
        offsets=(pid_q * BLOCK_Q, 0),
        block_shape=(BLOCK_Q, D),
        order=(1, 0),
    )

# ...
```

where `scale` is $\frac{1}{\sqrt{d}}$

3.4.3 Causal Masking in Triton

You can use `tl.arange` and `tl.where`.

3.4.4 Casting

Triton requires matching dtypes for matrix multiplications. You need to use `float32` for accumulators for numerical stability, you need to cast tensors appropriately:

1. **Before `tl.dot`**: Cast P_j (the attention weights) to match V_j's dtype:

   ```
   O_i = O_i * alpha[:, None] + tl.dot(P.to(V_j.dtype), V_j)
   ```

2. **Before storing output**: Cast the final output back to the input dtype:

   ```
   O_i = O_i.to(Q_i.dtype)
   tl.store(O_block, O_i, ...)
   ```

You can access dtypes using: - `tensor.dtype` for Triton tensors - `block_ptr.type.element_ty` for block pointers

3.4.5 Grid Configuration

For grid configuration, use the same strategy as in Part 1: each Triton program instance will load only elements from a single batch index, and only write to a single query tile of Q, O and L.

3.4.6 GPU Compatibility Note

Triton works best on Ampere (A100) and Hopper (H100) GPUs. On older Turing GPUs (T4, RTX 8000), some kernels may fail to compile or produce incorrect results. If your code works on H100 but fails on T4, this is expected behavior — focus on newer GPU architectures for this assignment.

3.5 Testing Your Implementation

Run the tests to verify correctness:

```
# Test correctness (forward and backward)
pytest tests/test_flash_attention.py -v

# Test memory efficiency
pytest tests/test_flash_memory.py -v
```

The tests check: - **Correctness**: Output matches reference implementation (forward and backward) - **Causal masking**: Proper application of the causal mask - **Output shape**: Correct tensor dimensions - **Memory efficiency**: No quadratic-size tensors saved for backward pass - **Saved tensors**: Correct tensors (Q, K, V, O, L) saved for backward

---

3.5.1 Part 3.B: Benchmarking [2 points]

Once your implementation passes the tests, benchmark it against PyTorch's optimized attention.

3.5.1.1 File to Complete Complete `benchmarking/bench_attention.py` to compare your Triton implementation against PyTorch's `scaled_dot_product_attention`.

3.5.1.2 Benchmark Requirements Your benchmark should:

1. **Compare both implementations**:
   - `pytorch_sdpa`: `torch.nn.functional.scaled_dot_product_attention` (compiled)
   - `flash_triton`: Your `FlashAttentionTriton` implementation (compiled)

2. **Vary the context length**: Test with increasing sequence lengths: `[256, 1024, 4096, 8192, 16384]`
3. **Measure performance**:
   - Forward pass execution time
   - Backward pass execution time
   - Peak GPU memory usage
   - Saved activations memory
4. **Use CUDA events for accurate timing** and **Include warmup iterations**
5. **Report results**: Print a summary table and save to CSV

3.5.1.3 Required Parameters

```
device = "cuda"
dtype = torch.float32
batch_size = 8
nb_warmup = 10
nb_forward_passes = 100
nb_backward_passes = 100

d_models = [64]
context_lengths = [256, 1024, 4096, 8192, 16384]
```

3.5.1.4 CSV Output Format Save results to `outputs/csv/attention_benchmark.csv` with columns:

Column	Type	Description
`implementation`	str	Implementation name (`pytorch_sdpa` or `flash_triton`)
`d_model`	int	Model dimension
`seq_len`	int	Sequence length
`forward_ms`	float	Forward pass time in milliseconds
`forward_peak_MiB`	float	Peak GPU memory during forward pass
`backward_ms`	float (nullable)	Backward pass time in milliseconds
`backward_peak_MiB`	float (nullable)	Peak GPU memory during backward pass
`saved_activations_MiB`	float	Memory for saved activations
`status`	str	`ok`, `OOM`, or `OOM(backward)`
`gpu`	str	GPU name

3.5.1.5 Running Your Benchmark

```
# Run all implementations
python -m benchmarking.bench_attention

# Run only Triton implementation
python -m benchmarking.bench_attention --impl flash_triton

# Run only PyTorch SDPA
python -m benchmarking.bench_attention --impl pytorch_sdpa
```

3.5.1.6 Expected Results Your Flash Attention implementation should: - Match or approach PyTorch SDPA performance (which uses FlashAttention internally on supported hardware) - Use **linear memory** for the forward pass (no N×N attention matrix) - Scale to longer sequences than naive attention

---

3.6 Bonus: Triton Backward Kernel [+4 points]

For bonus points, implement the backward pass in Triton as well. This requires writing a kernel that:

1. Recomputes attention scores tile-by-tile (like the forward pass)
2. Computes gradients dQ, dK, dV using the formulas from Part 2
3. Uses atomic operations or careful accumulation for dK and dV (since multiple query blocks contribute to the same key/value gradients)

3.7 Leaderboard (Optional)

Want to go further? Try to improve the performance of your implementation using any optimization tricks you can think of.

Rules:

- You cannot change the function's input/output signature
- You must use Triton (no CUDA)
- The implementation must be your own work (no pre-existing implementations)

Benchmarking: Measure your performance on an H100 GPU using `benchmarking/submit_leaderboard.py`.

Optimization ideas:

- Tune tile sizes for your kernel (use Triton's autotune feature)
- Experiment with additional Triton configuration parameters
- Implement the backward pass directly in Triton instead of relying on `torch.compile`
- Use two separate passes for the backward computation: one for dQ and another for dK/dV, avoiding atomics or inter-block synchronization
- Exit program instances early during causal masking by skipping tiles that are entirely zeroed out
- Separate non-masked tiles from diagonal tiles: compute the former without index comparisons, and the latter with a single comparison