2.3.1的主要的范例

vector add

```
"""
Vector Addition
===============

In this tutorial, you will write a simple vector addition using Triton.

In doing so, you will learn about:

* The basic programming model of Triton.

* The `triton.jit` decorator, which is used to define Triton kernels.

* The best practices for validating and benchmarking your custom ops
against native reference implementations.

"""

# %%
# Compute Kernel
# --------------

import torch

import triton
import triton.language as tl


@triton.jit
def add_kernel(x_ptr,  # *Pointer* to first input vector.
               y_ptr,  # *Pointer* to second input vector.
               output_ptr,  # *Pointer* to output vector.
               n_elements,  # Size of the vector.
               BLOCK_SIZE: tl.constexpr,  # Number of elements each program
should process.
               # NOTE: `constexpr` so it can be used as a shape value.
               ):
    # There are multiple 'programs' processing different data. We identify
which program
    # we are here:
    pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.
    # This program will process inputs that are offset from the initial
data.
    # For instance, if you had a vector of length 256 and block_size of 64,
the programs
    # would each access the elements [0:64, 64:128, 128:192, 192:256].
    # Note that offsets is a list of pointers:
```

```
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Create a mask to guard memory operations against out-of-bounds
accesses.
    mask = offsets < n_elements
    # Load x and y from DRAM, masking out any extra elements in case the
input is not a
    # multiple of the block size.
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    # Write x + y back to DRAM.
    tl.store(output_ptr + offsets, output, mask=mask)


# %%
# Let's also declare a helper function to (1) allocate the `z` tensor
# and (2) enqueue the above kernel with appropriate grid/block sizes:


def add(x: torch.Tensor, y: torch.Tensor):
    # We need to preallocate the output.
    output = torch.empty_like(x)
    assert x.is_cuda and y.is_cuda and output.is_cuda
    n_elements = output.numel()
    # The SPMD launch grid denotes the number of kernel instances that run
in parallel.
    # It is analogous to CUDA launch grids. It can be either Tuple[int], or
Callable(metaparameters) -> Tuple[int].
    # In this case, we use a 1D grid where the size is the number of
blocks:
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
    # NOTE:
    #  - Each torch.tensor object is implicitly converted into a pointer to
its first element.
    #  - `triton.jit`'ed functions can be indexed with a launch grid to
obtain a callable GPU kernel.
    #  - Don't forget to pass meta-parameters as keywords arguments.
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
    # We return a handle to z but, since `torch.cuda.synchronize()` hasn't
been called, the kernel is still
    # running asynchronously at this point.
    return output


# %%
# We can now use the above function to compute the element-wise sum of two
`torch.tensor` objects and test its correctness:

torch.manual_seed(0)
size = 98432
x = torch.rand(size, device='cuda')
y = torch.rand(size, device='cuda')
output_torch = x + y
```

```
output_triton = add(x, y)
print(output_torch)
print(output_triton)
print(f'The maximum difference between torch and triton is '
      f'{torch.max(torch.abs(output_torch - output_triton))}')

# %%
# Seems like we're good to go!

# %%
# Benchmark
# ---------
#
# We can now benchmark our custom op on vectors of increasing sizes to get
a sense of how it does relative to PyTorch.
# To make things easier, Triton has a set of built-in utilities that allow
us to concisely plot the performance of our custom ops.
# for different problem sizes.


@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=['size'],  # Argument names to use as an x-axis for the
plot.
        x_vals=[2**i for i in range(12, 28, 1)],  # Different possible
values for `x_name`.
        x_log=True,  # x axis is logarithmic.
        line_arg='provider',  # Argument name whose value corresponds to a
different line in the plot.
        line_vals=['triton', 'torch'],  # Possible values for `line_arg`.
        line_names=['Triton', 'Torch'],  # Label name for the lines.
        styles=[('blue', '-'), ('green', '-')],  # Line styles.
        ylabel='GB/s',  # Label name for the y-axis.
        plot_name='vector-add-performance',  # Name for the plot. Used also
as a file name for saving the plot.
        args={},  # Values for function arguments not in `x_names` and
`y_name`.
    ))
def benchmark(size, provider):
    x = torch.rand(size, device='cuda', dtype=torch.float32)
    y = torch.rand(size, device='cuda', dtype=torch.float32)
    quantiles = [0.5, 0.2, 0.8]
    if provider == 'torch':
        ms, min_ms, max_ms = triton.testing.do_bench(lambda: x + y,
quantiles=quantiles)
    if provider == 'triton':
        ms, min_ms, max_ms = triton.testing.do_bench(lambda: add(x, y),
quantiles=quantiles)
    gbps = lambda ms: 12 * size / ms * 1e-6
    return gbps(ms), gbps(max_ms), gbps(min_ms)


# %%
# We can now run the decorated function above. Pass `print_data=True` to
```

```
see the performance number, `show_plots=True` to plot them, and/or
# `save_path='/path/to/results/' to save them to disk along with raw CSV
data:
benchmark.run(print_data=True, show_plots=True)
```

fused-softmax

```
"""
Fused Softmax
=============

In this tutorial, you will write a fused softmax operation that is
significantly faster
than PyTorch's native op for a particular class of matrices: those whose
rows can fit in
the GPU's SRAM.

In doing so, you will learn about:

* The benefits of kernel fusion for bandwidth-bound operations.

* Reduction operators in Triton.

"""

# %%
# Motivations
# -----------
#
# Custom GPU kernels for elementwise additions are educationally valuable
but won't get you very far in practice.
# Let us consider instead the case of a simple (numerically stabilized)
softmax operation:

import torch

import triton
import triton.language as tl


@torch.jit.script
def naive_softmax(x):
    """Compute row-wise softmax of X using native pytorch

    We subtract the maximum element in order to avoid overflows. Softmax is
invariant to
    this shift.
    """
    # read  MN elements ; write M  elements
    x_max = x.max(dim=1)[0]
```

```
    # read MN + M elements ; write MN elements
    z = x - x_max[:, None]
    # read  MN elements ; write MN elements
    numerator = torch.exp(z)
    # read  MN elements ; write M  elements
    denominator = numerator.sum(dim=1)
    # read MN + M elements ; write MN elements
    ret = numerator / denominator[:, None]
    # in total: read 5MN + 2M elements ; wrote 3MN + 2M elements
    return ret


# %%
# When implemented naively in PyTorch, computing :code:`y =
naive_softmax(x)` for :math:`x \in R^{M \times N}`
# requires reading :math:`5MN + 2M` elements from DRAM and writing back
:math:`3MN + 2M` elements.
# This is obviously wasteful; we'd prefer to have a custom "fused" kernel
that only reads
# X once and does all the necessary computations on-chip.
# Doing so would require reading and writing back only :math:`MN` bytes, so
we could
# expect a theoretical speed-up of ~4x (i.e., :math:`(8MN + 4M) / 2MN`).
# The `torch.jit.script` flags aims to perform this kind of "kernel fusion"
automatically
# but, as we will see later, it is still far from ideal.

# %%
# Compute Kernel
# --------------
#
# Our softmax kernel works as follows: each program loads a row of the
input matrix X,
# normalizes it and writes back the result to the output Y.
#
# Note that one important limitation of Triton is that each block must have
a
# power-of-two number of elements, so we need to internally "pad" each row
and guard the
# memory operations properly if we want to handle any possible input
shapes:


@triton.jit
def softmax_kernel(output_ptr, input_ptr, input_row_stride,
output_row_stride, n_cols, BLOCK_SIZE: tl.constexpr):
    # The rows of the softmax are independent, so we parallelize across
those
    row_idx = tl.program_id(0)
    # The stride represents how much we need to increase the pointer to
advance 1 row
    row_start_ptr = input_ptr + row_idx * input_row_stride
    # The block size is the next power of two greater than n_cols, so we
can fit each
```

```
    # row in a single block
    col_offsets = tl.arange(0, BLOCK_SIZE)
    input_ptrs = row_start_ptr + col_offsets
    # Load the row into SRAM, using a mask since BLOCK_SIZE may be > than
n_cols
    row = tl.load(input_ptrs, mask=col_offsets < n_cols, other=-
float('inf'))
    # Subtract maximum for numerical stability
    row_minus_max = row - tl.max(row, axis=0)
    # Note that exponentiation in Triton is fast but approximate (i.e.,
think __expf in CUDA)
    numerator = tl.exp(row_minus_max)
    denominator = tl.sum(numerator, axis=0)
    softmax_output = numerator / denominator
    # Write back output to DRAM
    output_row_start_ptr = output_ptr + row_idx * output_row_stride
    output_ptrs = output_row_start_ptr + col_offsets
    tl.store(output_ptrs, softmax_output, mask=col_offsets < n_cols)


# %%
# We can create a helper function that enqueues the kernel and its
(meta-)arguments for any given input tensor.


def softmax(x):
    n_rows, n_cols = x.shape
    # The block size is the smallest power of two greater than the number
of columns in `x`
    BLOCK_SIZE = triton.next_power_of_2(n_cols)
    # Another trick we can use is to ask the compiler to use more threads
per row by
    # increasing the number of warps (`num_warps`) over which each row is
distributed.
    # You will see in the next tutorial how to auto-tune this value in a
more natural
    # way so you don't have to come up with manual heuristics yourself.
    num_warps = 4
    if BLOCK_SIZE >= 2048:
        num_warps = 8
    if BLOCK_SIZE >= 4096:
        num_warps = 16
    # Allocate output
    y = torch.empty_like(x)
    # Enqueue kernel. The 1D launch grid is simple: we have one kernel
instance per row o
    # f the input matrix
    softmax_kernel[(n_rows, )](
        y,
        x,
        x.stride(0),
        y.stride(0),
        n_cols,
        num_warps=num_warps,
```

```
        BLOCK_SIZE=BLOCK_SIZE,
    )
    return y


# %%
# Unit Test
# ---------

# %%
# We make sure that we test our kernel on a matrix with an irregular number
of rows and columns.
# This will allow us to verify that our padding mechanism works.

torch.manual_seed(0)
x = torch.randn(1823, 781, device='cuda')
y_triton = softmax(x)
y_torch = torch.softmax(x, axis=1)
assert torch.allclose(y_triton, y_torch), (y_triton, y_torch)

# %%
# As expected, the results are identical.

# %%
# Benchmark
# ---------
#
# Here we will benchmark our operation as a function of the number of
columns in the input matrix -- assuming 4096 rows.
# We will then compare its performance against (1) :code:`torch.softmax`
and (2) the :code:`naive_softmax` defined above.


@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=['N'],  # argument names to use as an x-axis for the plot
        x_vals=[128 * i for i in range(2, 100)],  # different possible
values for `x_name`
        line_arg='provider',  # argument name whose value corresponds to a
different line in the plot
        line_vals=[
            'triton',
            'torch-native',
            'torch-jit',
        ],  # possible values for `line_arg``
        line_names=[
            "Triton",
            "Torch (native)",
            "Torch (jit)",
        ],  # label name for the lines
        styles=[('blue', '-'), ('green', '-'), ('green', '--')],  # line
styles
        ylabel="GB/s",  # label name for the y-axis
        plot_name="softmax-performance",  # name for the plot. Used also as
```

```
a file name for saving the plot.
        args={'M': 4096},  # values for function arguments not in `x_names`
and `y_name`
    ))
def benchmark(M, N, provider):
    x = torch.randn(M, N, device='cuda', dtype=torch.float32)
    quantiles = [0.5, 0.2, 0.8]
    if provider == 'torch-native':
        ms, min_ms, max_ms = triton.testing.do_bench(lambda:
torch.softmax(x, axis=-1), quantiles=quantiles)
    if provider == 'triton':
        ms, min_ms, max_ms = triton.testing.do_bench(lambda: softmax(x),
quantiles=quantiles)
    if provider == 'torch-jit':
        ms, min_ms, max_ms = triton.testing.do_bench(lambda:
naive_softmax(x), quantiles=quantiles)
    gbps = lambda ms: 2 * x.nelement() * x.element_size() * 1e-9 / (ms *
1e-3)
    return gbps(ms), gbps(max_ms), gbps(min_ms)


benchmark.run(show_plots=True, print_data=True)

# %%
# In the above plot, we can see that:
#  - Triton is 4x faster than the Torch JIT. This confirms our suspicions
that the Torch JIT does not do any fusion here.
#  - Triton is noticeably faster than :code:`torch.softmax` -- in addition
to being **easier to read, understand and maintain**.
#    Note however that the PyTorch `softmax` operation is more general and
will work on tensors of any shape.
```

matmul

```
"""
Matrix Multiplication
=====================
In this tutorial, you will write a very short high-performance FP16 matrix
multiplication kernel that achieves
performance on parallel with cuBLAS.

You will specifically learn about:

* Block-level matrix multiplications.

* Multi-dimensional pointer arithmetics.

* Program re-ordering for improved L2 cache hit rate.

* Automatic performance tuning.
```

```
"""

# %%
# Motivations
# -----------
#
# Matrix multiplications are a key building block of most modern high-
performance computing systems.
# They are notoriously hard to optimize, hence their implementation is
generally done by
# hardware vendors themselves as part of so-called "kernel libraries"
(e.g., cuBLAS).
# Unfortunately, these libraries are often proprietary and cannot be easily
customized
# to accommodate the needs of modern deep learning workloads (e.g., fused
activation functions).
# In this tutorial, you will learn how to implement efficient matrix
multiplications by
# yourself with Triton, in a way that is easy to customize and extend.
#
# Roughly speaking, the kernel that we will write will implement the
following blocked
# algorithm to multiply a (M, K) by a (K, N) matrix:
#
#  .. code-block:: python
#
#    # Do in parallel
#    for m in range(0, M, BLOCK_SIZE_M):
#      # Do in parallel
#      for n in range(0, N, BLOCK_SIZE_N):
#        acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=float32)
#        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
#
# where each iteration of the doubly-nested for-loop is performed by a
dedicated Triton program instance.

# %%
# Compute Kernel
# --------------
#
# The above algorithm is, actually, fairly straightforward to implement in
Triton.
# The main difficulty comes from the computation of the memory locations at
which blocks
# of :code:`A` and :code:`B` must be read in the inner loop. For that, we
need
# multi-dimensional pointer arithmetics.
#
# Pointer Arithmetics
```

```
# ~~~~~~~~~~~~~~~~~~~
#
# For a row-major 2D tensor :code:`X`, the memory location of :code:`X[i,
j]` is given b
# y :code:`&X[i, j] = X + i*stride_xi + j*stride_xj`.
# Therefore, blocks of pointers for :code:`A[m : m+BLOCK_SIZE_M,
k:k+BLOCK_SIZE_K]` and
# :code:`B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]` can be defined in
pseudo-code as:
#
#  .. code-block:: python
#
#    &A[m : m+BLOCK_SIZE_M, k:k+BLOCK_SIZE_K] =  a_ptr + (m :
m+BLOCK_SIZE_M)[:, None]*A.stride(0) + (k : k+BLOCK_SIZE_K)[None,
:]*A.stride(1);
#    &B[k : k+BLOCK_SIZE_K, n:n+BLOCK_SIZE_N] =  b_ptr + (k :
k+BLOCK_SIZE_K)[:, None]*B.stride(0) + (n : n+BLOCK_SIZE_N)[None,
:]*B.stride(1);
#
# Which means that pointers for blocks of A and B can be initialized (i.e.,
:code:`k=0`) in Triton as the following
# code. Also note that we need an extra modulo to handle the case where
:code:`M` is not a multiple of
# :code:`BLOCK_SIZE_M` or :code:`N` is not a multiple of
:code:`BLOCK_SIZE_N`, in which case we can pad the data with
# some useless values, which will not contribute to the results. For the
:code:`K` dimension, we will handle that later
# using masking load semantics.
#
#  .. code-block:: python
#
#    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
#    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
#    offs_k = tl.arange(0, BLOCK_SIZE_K)
#    a_ptrs = a_ptr + (offs_am[:, None]*stride_am + offs_k [None,
:]*stride_ak)
#    b_ptrs = b_ptr + (offs_k [:, None]*stride_bk + offs_bn[None,
:]*stride_bn)
#
# And then updated in the inner loop as follows:
#
#  .. code-block:: python
#
#    a_ptrs += BLOCK_SIZE_K * stride_ak;
#    b_ptrs += BLOCK_SIZE_K * stride_bk;
#
#
# L2 Cache Optimizations
# ~~~~~~~~~~~~~~~~~~~~~
#
# As mentioned above, each program instance computes a
:code:`[BLOCK_SIZE_M, BLOCK_SIZE_N]`
# block of :code:`C`.
# It is important to remember that the order in which these blocks are
```

```
computed does
# matter, since it affects the L2 cache hit rate of our program. and
unfortunately, a
# a simple row-major ordering
#
#  .. code-block:: Python
#
#    pid = triton.program_id(0);
#    grid_m = (M + BLOCK_SIZE_M - 1) // BLOCK_SIZE_M;
#    grid_n = (N + BLOCK_SIZE_N - 1) // BLOCK_SIZE_N;
#    pid_m = pid / grid_n;
#    pid_n = pid % grid_n;
#
# is just not going to cut it.
#
# One possible solution is to launch blocks in an order that promotes data
reuse.
# This can be done by 'super-grouping' blocks in groups of :code:`GROUP_M`
rows before
# switching to the next column:
#
#  .. code-block:: python
#
#    # Program ID
#    pid = tl.program_id(axis=0)
#    # Number of program ids along the M axis
#    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
#    # Number of programs ids along the N axis
#    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
#    # Number of programs in group
#    num_pid_in_group = GROUP_SIZE_M * num_pid_n
#    # Id of the group this program is in
#    group_id = pid // num_pid_in_group
#    # Row-id of the first program in the group
#    first_pid_m = group_id * GROUP_SIZE_M
#    # If `num_pid_m` isn't divisible by `GROUP_SIZE_M`, the last group is
smaller
#    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
#    # *Within groups*, programs are ordered in a column-major order
#    # Row-id of the program in the *launch grid*
#    pid_m = first_pid_m + (pid % group_size_m)
#    # Col-id of the program in the *launch grid*
#    pid_n = (pid % num_pid_in_group) // group_size_m
#
# For example, in the following matmul where each matrix is 9 blocks by 9
blocks,
# we can see that if we compute the output in row-major ordering, we need
to load 90
# blocks into SRAM to compute the first 9 output blocks, but if we do it in
grouped
# ordering, we only need to load 54 blocks.
#
#   .. image:: grouped_vs_row_major_ordering.png
#
```

```
# In practice, this can improve the performance of our matrix
multiplication kernel by
# more than 10\% on some hardware architecture (e.g., 220 to 245 TFLOPS on
A100).
#

# %%
# Final Result
# ------------

import torch

import triton
import triton.language as tl


# `triton.jit`'ed functions can be auto-tuned by using the
`triton.autotune` decorator, which consumes:
#   - A list of `triton.Config` objects that define different
configurations of
#       meta-parameters (e.g., `BLOCK_SIZE_M`) and compilation options
(e.g., `num_warps`) to try
#   - An auto-tuning *key* whose change in values will trigger evaluation
of all the
#       provided configs
@triton.autotune(
    configs=[
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 256,
'BLOCK_SIZE_K': 64, 'GROUP_SIZE_M': 8}, num_stages=3,
                      num_warps=8),
        triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 256,
'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
                      num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 128,
'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
                      num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 64,
'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
                      num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 128,
'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
                      num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 32,
'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4,
                      num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 32,
'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=5,
                      num_warps=2),
        triton.Config({'BLOCK_SIZE_M': 32, 'BLOCK_SIZE_N': 64,
'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=5,
                      num_warps=2),
    ],
    key=['M', 'N', 'K'],
)
```

```
@triton.jit
def matmul_kernel(
        # Pointers to matrices
        a_ptr, b_ptr, c_ptr,
        # Matrix dimensions
        M, N, K,
        # The stride variables represent how much to increase the ptr by
when moving by 1
        # element in a particular dimension. E.g. `stride_am` is how much
to increase `a_ptr`
        # by to get the element one row down (A has M rows).
        stride_am, stride_ak,  #
        stride_bk, stride_bn,  #
        stride_cm, stride_cn,
        # Meta-parameters
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr,
BLOCK_SIZE_K: tl.constexpr,  #
        GROUP_SIZE_M: tl.constexpr,  #
        ACTIVATION: tl.constexpr  #
):
    """Kernel for computing the matmul C = A x B.
    A has shape (M, K), B has shape (K, N) and C has shape (M, N)
    """
    # -----------------------------------------------------------
    # Map program ids `pid` to the block of C it should compute.
    # This is done in a grouped ordering to promote L2 data reuse.
    # See above `L2 Cache Optimizations` section for details.
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + (pid % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m

    # -----------------------------------------------------------
    # Create pointers for the first blocks of A and B.
    # We will advance this pointer as we move in the K direction
    # and accumulate
    # `a_ptrs` is a block of [BLOCK_SIZE_M, BLOCK_SIZE_K] pointers
    # `b_ptrs` is a block of [BLOCK_SIZE_K, BLOCK_SIZE_N] pointers
    # See above `Pointer Arithmetics` section for details
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] *
stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] *
stride_bn)

    # -----------------------------------------------------------
    # Iterate to compute a block of the C matrix.
```

```
    # We accumulate into a `[BLOCK_SIZE_M, BLOCK_SIZE_N]` block
    # of fp32 values for higher accuracy.
    # `accumulator` will be converted back to fp16 after the loop.
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        # Load the next block of A and B, generate a mask by checking the K
dimension.
        # If it is out of bounds, set it to 0.
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K,
other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K,
other=0.0)
        # We accumulate along the K dimension.
        accumulator += tl.dot(a, b)
        # Advance the ptrs to the next K block.
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    # You can fuse arbitrary activation functions here
    # while the accumulator is still in FP32!
    if ACTIVATION == "leaky_relu":
        accumulator = leaky_relu(accumulator)
    c = accumulator.to(tl.float16)

    # -----------------------------------------------------------
    # Write back the block of the output matrix C with masks.
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn *
offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)


# We can fuse `leaky_relu` by providing it as an `ACTIVATION` meta-
parameter in `_matmul`.
@triton.jit
def leaky_relu(x):
    x = x + 1
    return tl.where(x >= 0, x, 0.01 * x)


# %%
# We can now create a convenience wrapper function that only takes two
input tensors,
# and (1) checks any shape constraint; (2) allocates the output; (3)
launches the above kernel.


def matmul(a, b, activation=""):
    # Check constraints.
    assert a.shape[1] == b.shape[0], "Incompatible dimensions"
    assert a.is_contiguous(), "Matrix A must be contiguous"
    assert b.is_contiguous(), "Matrix B must be contiguous"
    M, K = a.shape
```

```
    K, N = b.shape
    # Allocates output.
    c = torch.empty((M, N), device=a.device, dtype=a.dtype)
    # 1D launch kernel where each block gets its own program.
    grid = lambda META: (triton.cdiv(M, META['BLOCK_SIZE_M']) *
triton.cdiv(N, META['BLOCK_SIZE_N']), )
    matmul_kernel[grid](
        a, b, c,  #
        M, N, K,  #
        a.stride(0), a.stride(1),  #
        b.stride(0), b.stride(1),  #
        c.stride(0), c.stride(1),  #
        ACTIVATION=activation  #
    )
    return c


# %%
# Unit Test
# ---------
#
# We can test our custom matrix multiplication operation against a native
torch implementation (i.e., cuBLAS).

torch.manual_seed(0)
a = torch.randn((512, 512), device='cuda', dtype=torch.float16)
b = torch.randn((512, 512), device='cuda', dtype=torch.float16)
triton_output = matmul(a, b)
torch_output = torch.matmul(a, b)
print(f"triton_output={triton_output}")
print(f"torch_output={torch_output}")
if torch.allclose(triton_output, torch_output, atol=1e-2, rtol=0):
    print("✅ Triton and Torch match")
else:
    print("❌ Triton and Torch differ")

# %%
# Benchmark
# ---------
#
# Square Matrix Performance
# ~~~~~~~~~~~~~~~~~~~~~~~~~~
#
# We can now compare the performance of our kernel against that of cuBLAS.
Here we focus on square matrices,
# but feel free to arrange this script as you wish to benchmark any other
matrix shape.


@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=['M', 'N', 'K'],  # Argument names to use as an x-axis for
the plot
        x_vals=[128 * i for i in range(2, 33)],  # Different possible
```

```
values for `x_name`
        line_arg='provider',  # Argument name whose value corresponds to a
different line in the plot
        # Possible values for `line_arg`
        line_vals=['cublas', 'triton'],
        # Label name for the lines
        line_names=["cuBLAS", "Triton"],
        # Line styles
        styles=[('green', '-'), ('blue', '-')],
        ylabel="TFLOPS",  # Label name for the y-axis
        plot_name="matmul-performance",  # Name for the plot, used also as
a file name for saving the plot.
        args={},
    ))
def benchmark(M, N, K, provider):
    a = torch.randn((M, K), device='cuda', dtype=torch.float16)
    b = torch.randn((K, N), device='cuda', dtype=torch.float16)
    quantiles = [0.5, 0.2, 0.8]
    if provider == 'cublas':
        ms, min_ms, max_ms = triton.testing.do_bench(lambda:
torch.matmul(a, b), quantiles=quantiles)
    if provider == 'triton':
        ms, min_ms, max_ms = triton.testing.do_bench(lambda: matmul(a, b),
quantiles=quantiles)
    perf = lambda ms: 2 * M * N * K * 1e-12 / (ms * 1e-3)
    return perf(ms), perf(max_ms), perf(min_ms)


benchmark.run(show_plots=True, print_data=True)
```

dropout

```
"""
Low-Memory Dropout
==================

In this tutorial, you will write a memory-efficient implementation of
dropout whose state
will be composed of a single int32 seed. This differs from more traditional
implementations of dropout,
whose state is generally composed of a bit mask tensor of the same shape as
the input.

In doing so, you will learn about:

* The limitations of naive implementations of Dropout with PyTorch.

* Parallel pseudo-random number generation in Triton.

"""
```

```
# %%
# Baseline
# --------
#
# The *dropout* operator was first introduced in [SRIVASTAVA2014]_ as a way
to improve the performance
# of deep neural networks in low-data regime (i.e. regularization).
#
# It takes a vector as input and produces a vector of the same shape as
output. Each scalar in the
# output has a probability :math:`p` of being changed to zero and otherwise
it is copied from the input.
# This forces the network to perform well even when only :math:`1 - p`
scalars from the input are available.
#
# At evaluation time we want to use the full power of the network so we set
:math:`p=0`. Naively this would
# increase the norm of the output (which can be a bad thing, e.g. it can
lead to artificial decrease
# in the output softmax temperature). To prevent this we multiply the
output by :math:`\frac{1}{1 - p}`, which
# keeps the norm consistent regardless of the dropout probability.
#
# Let's first take a look at the baseline implementation.

import tabulate
import torch

import triton
import triton.language as tl


@triton.jit
def _dropout(
    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 that an element of `x` is changed to zero
    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)
    # The line below is the crucial part, described in the paragraph above!
    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[grid](x, x_keep, output, n_elements, p, BLOCK_SIZE=1024)
    return output


# Input tensor
x = torch.randn(size=(10, )).cuda()
# Dropout mask
p = 0.5
x_keep = (torch.rand(size=(10, )) > p).to(torch.int32).cuda()
#
output = dropout(x, x_keep=x_keep, p=p)
print(tabulate.tabulate([
    ["input"] + x.tolist(),
    ["keep mask"] + x_keep.tolist(),
    ["output"] + output.tolist(),
]))

# %%
# Seeded dropout
# --------------
#
# The above implementation of dropout works fine, but it can be a bit
awkward to deal with. Firstly
# we need to store the dropout mask for backpropagation. Secondly, dropout
state management can get
# very tricky when using recompute/checkpointing (e.g. see all the notes
about `preserve_rng_state` in
# https://pytorch.org/docs/1.9.0/checkpoint.html). In this tutorial we'll
describe an alternative implementation
# that (1) has a smaller memory footprint; (2) requires less data movement;
and (3) simplifies the management
# of persisting randomness across multiple invocations of the kernel.
#
# Pseudo-random number generation in Triton is simple! In this tutorial we
will use the
# :code:`triton.language.rand` function which generates a block of
uniformly distributed :code:`float32`
# values in [0, 1), given a seed and a block of :code:`int32` offsets. But
if you need it, Triton also provides
# other :ref:`random number generation strategies <Random Number
Generation>`.
#
# .. note::
#    Triton's implementation of PRNG is based on the Philox algorithm
(described on [SALMON2011]_).
#
# Let's put it all together.
```

```
@triton.jit
def _seeded_dropout(
    x_ptr,
    output_ptr,
    n_elements,
    p,
    seed,
    BLOCK_SIZE: tl.constexpr,
):
    # compute memory offsets of elements handled by this instance
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # load data from x
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    # randomly prune it
    random = tl.rand(seed, offsets)
    x_keep = random > p
    # write-back
    output = tl.where(x_keep, x / (1 - p), 0.0)
    tl.store(output_ptr + offsets, output, mask=mask)


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


x = torch.randn(size=(10, )).cuda()
# Compare this to the baseline - dropout mask is never instantiated!
output = seeded_dropout(x, p=0.5, seed=123)
output2 = seeded_dropout(x, p=0.5, seed=123)
output3 = seeded_dropout(x, p=0.5, seed=512)

print(
    tabulate.tabulate([
        ["input"] + x.tolist(),
        ["output (seed = 123)"] + output.tolist(),
        ["output (seed = 123)"] + output2.tolist(),
        ["output (seed = 512)"] + output3.tolist(),
    ]))

# %%
# Et Voilà! We have a triton kernel that applies the same dropout mask
provided the seed is the same!
# If you'd like explore further applications of pseudorandomness in GPU
programming, we encourage you
# to explore the `triton/language/random` folder!
```

```
# %%
# Exercises
# ---------
#
# 1. Extend the kernel to operate over a matrix and use a vector of seeds -
one per row.
# 2. Add support for striding.
# 3. (challenge) Implement a kernel for sparse Johnson-Lindenstrauss
transform which generates the projection matrix one the fly each time using
a seed.

# %%
# References
# ----------
#
# .. [SALMON2011] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E.
Shaw, "Parallel Random Numbers: As Easy as 1, 2, 3", 2011
# .. [SRIVASTAVA2014] Nitish Srivastava and Geoffrey Hinton and Alex
Krizhevsky and Ilya Sutskever and Ruslan Salakhutdinov, "Dropout: A Simple
Way to Prevent Neural Networks from Overfitting", JMLR 2014
```

layer-norm

```
"""
Layer Normalization
====================
In this tutorial, you will write a high-performance layer normalization
kernel that runs faster than the PyTorch implementation.

In doing so, you will learn about:

* Implementing backward pass in Triton.

* Implementing parallel reduction in Triton.

"""

# %%
# Motivations
# -----------
#
# The *LayerNorm* operator was first introduced in [BA2016]_ as a way to
improve the performance
# of sequential models (e.g., Transformers) or neural networks with small
batch size.
# It takes a vector :math:`x` as input and produces a vector :math:`y` of
the same shape as output.
# The normalization is performed by subtracting the mean and dividing by
the standard deviation of :math:`x`.
```

```
# After the normalization, a learnable linear transformation with weights
:math:`w` and biases :math:`b` is applied.
# The forward pass can be expressed as follows:
#
# .. math::
#    y = \frac{ x - \text{E}[x] }{ \sqrt{\text{Var}(x) + \epsilon} } * w +
b
#
# where :math:`\epsilon` is a small constant added to the denominator for
numerical stability.
# Let’s first take a look at the forward pass implementation.

import torch

import triton
import triton.language as tl

try:
    # This is https://github.com/NVIDIA/apex, NOT the apex on PyPi, so it
    # should not be added to extras_require in setup.py.
    import apex
    HAS_APEX = True
except ModuleNotFoundError:
    HAS_APEX = False


@triton.jit
def _layer_norm_fwd_fused(
    X,  # pointer to the input
    Y,  # pointer to the output
    W,  # pointer to the weights
    B,  # pointer to the biases
    Mean,  # pointer to the mean
    Rstd,  # pointer to the 1/std
    stride,  # how much to increase the pointer when moving by 1 row
    N,  # number of columns in X
    eps,  # epsilon to avoid division by zero
    BLOCK_SIZE: tl.constexpr,
):
    # Map the program id to the row of X and Y it should compute.
    row = tl.program_id(0)
    Y += row * stride
    X += row * stride
    # Compute mean
    mean = 0
    _mean = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        a = tl.load(X + cols, mask=cols < N, other=0.).to(tl.float32)
        _mean += a
    mean = tl.sum(_mean, axis=0) / N
    # Compute variance
    _var = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
    for off in range(0, N, BLOCK_SIZE):
```

```
        cols = off + tl.arange(0, BLOCK_SIZE)
        x = tl.load(X + cols, mask=cols < N, other=0.).to(tl.float32)
        x = tl.where(cols < N, x - mean, 0.)
        _var += x * x
    var = tl.sum(_var, axis=0) / N
    rstd = 1 / tl.sqrt(var + eps)
    # Write mean / rstd
    tl.store(Mean + row, mean)
    tl.store(Rstd + row, rstd)
    # Normalize and apply linear transformation
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        mask = cols < N
        w = tl.load(W + cols, mask=mask)
        b = tl.load(B + cols, mask=mask)
        x = tl.load(X + cols, mask=mask, other=0.).to(tl.float32)
        x_hat = (x - mean) * rstd
        y = x_hat * w + b
        # Write output
        tl.store(Y + cols, y, mask=mask)


# %%
# Backward pass
# -------------
#
# The backward pass for the layer normalization operator is a bit more
involved than the forward pass.
# Let :math:`\hat{x}` be the normalized inputs :math:`\frac{ x - \text{E}
[x] }{ \sqrt{\text{Var}(x) + \epsilon} }` before the linear transformation,
# the Vector-Jacobian Products (VJP) :math:`\nabla_{x}` of :math:`x` are
given by:
#
# .. math::
#    \nabla_{x} = \frac{1}{\sigma}\Big( \nabla_{y} \odot w - \underbrace{
\big( \frac{1}{N} \hat{x} \cdot (\nabla_{y} \odot w) \big) }_{c_1} \odot
\hat{x} - \underbrace{ \frac{1}{N} \nabla_{y} \cdot w }_{c_2} \Big)
#
# where :math:`\odot` denotes the element-wise multiplication,
:math:`\cdot` denotes the dot product, and :math:`\sigma` is the standard
deviation.
# :math:`c_1` and :math:`c_2` are intermediate constants that improve the
readability of the following implementation.
#
# For the weights :math:`w` and biases :math:`b`, the VJPs
:math:`\nabla_{w}` and :math:`\nabla_{b}` are more straightforward:
#
# .. math::
#    \nabla_{w} = \nabla_{y} \odot \hat{x} \quad \text{and} \quad
\nabla_{b} = \nabla_{y}
#
# Since the same weights :math:`w` and biases :math:`b` are used for all
rows in the same batch, their gradients need to sum up.
# To perform this step efficiently, we use a parallel reduction strategy:
```

```
    each kernel instance accumulates
    # partial :math:`\nabla_{w}` and :math:`\nabla_{b}` across certain rows
    into one of :math:`\text{GROUP_SIZE_M}` independent buffers.
    # These buffers stay in the L2 cache and then are further reduced by
    another function to compute the actual :math:`\nabla_{w}` and
    :math:`\nabla_{b}`.
    #
    # Let the number of input rows :math:`M = 4` and :math:`\text{GROUP_SIZE_M}
    = 2`,
    # here's a diagram of the parallel reduction strategy for
    :math:`\nabla_{w}` (:math:`\nabla_{b}` is omitted for brevity):
    #
    #   .. image:: parallel_reduction.png
    #
    # In Stage 1, the rows of X that have the same color share the same buffer
    and thus a lock is used to ensure that only one kernel instance writes to
    the buffer at a time.
    # In Stage 2, the buffers are further reduced to compute the final
    :math:`\nabla_{w}` and :math:`\nabla_{b}`.
    # In the following implementation, Stage 1 is implemented by the function
    :code:`_layer_norm_bwd_dx_fused` and Stage 2 is implemented by the function
    :code:`_layer_norm_bwd_dwdb`.


    @triton.jit
    def _layer_norm_bwd_dx_fused(DX,  # pointer to the input gradient
                                 DY,  # pointer to the output gradient
                                 DW,  # pointer to the partial sum of weights
    gradient
                                 DB,  # pointer to the partial sum of biases
    gradient
                                 X,  # pointer to the input
                                 W,  # pointer to the weights
                                 B,  # pointer to the biases
                                 Mean,  # pointer to the mean
                                 Rstd,  # pointer to the 1/std
                                 Lock,  # pointer to the lock
                                 stride,  # how much to increase the pointer
    when moving by 1 row
                                 N,  # number of columns in X
                                 eps,  # epsilon to avoid division by zero
                                 GROUP_SIZE_M: tl.constexpr, BLOCK_SIZE_N:
    tl.constexpr):
        # Map the program id to the elements of X, DX, and DY it should
    compute.
        row = tl.program_id(0)
        cols = tl.arange(0, BLOCK_SIZE_N)
        mask = cols < N
        X += row * stride
        DY += row * stride
        DX += row * stride
        # Offset locks and weights/biases gradient pointer for parallel
    reduction
        lock_id = row % GROUP_SIZE_M
```

```
    Lock += lock_id
    Count = Lock + GROUP_SIZE_M
    DW = DW + lock_id * N + cols
    DB = DB + lock_id * N + cols
    # Load data to SRAM
    x = tl.load(X + cols, mask=mask, other=0).to(tl.float32)
    dy = tl.load(DY + cols, mask=mask, other=0).to(tl.float32)
    w = tl.load(W + cols, mask=mask).to(tl.float32)
    mean = tl.load(Mean + row)
    rstd = tl.load(Rstd + row)
    # Compute dx
    xhat = (x - mean) * rstd
    wdy = w * dy
    xhat = tl.where(mask, xhat, 0.)
    wdy = tl.where(mask, wdy, 0.)
    c1 = tl.sum(xhat * wdy, axis=0) / N
    c2 = tl.sum(wdy, axis=0) / N
    dx = (wdy - (xhat * c1 + c2)) * rstd
    # Write dx
    tl.store(DX + cols, dx, mask=mask)
    # Accumulate partial sums for dw/db
    partial_dw = (dy * xhat).to(w.dtype)
    partial_db = (dy).to(w.dtype)
    while tl.atomic_cas(Lock, 0, 1) == 1:
        pass
    count = tl.load(Count)
    # First store doesn't accumulate
    if count == 0:
        tl.atomic_xchg(Count, 1)
    else:
        partial_dw += tl.load(DW, mask=mask)
        partial_db += tl.load(DB, mask=mask)
    tl.store(DW, partial_dw, mask=mask)
    tl.store(DB, partial_db, mask=mask)
    # Release the lock
    tl.atomic_xchg(Lock, 0)


@triton.jit
def _layer_norm_bwd_dwdb(DW,  # pointer to the partial sum of weights
gradient
                         DB,  # pointer to the partial sum of biases
gradient
                         FINAL_DW,  # pointer to the weights gradient
                         FINAL_DB,  # pointer to the biases gradient
                         M,  # GROUP_SIZE_M
                         N,  # number of columns
                         BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N:
tl.constexpr):
    # Map the program id to the elements of DW and DB it should compute.
    pid = tl.program_id(0)
    cols = pid * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    dw = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    db = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
```

```
    # Iterate through the rows of DW and DB to sum the partial sums.
    for i in range(0, M, BLOCK_SIZE_M):
        rows = i + tl.arange(0, BLOCK_SIZE_M)
        mask = (rows[:, None] < M) & (cols[None, :] < N)
        offs = rows[:, None] * N + cols[None, :]
        dw += tl.load(DW + offs, mask=mask, other=0.)
        db += tl.load(DB + offs, mask=mask, other=0.)
    # Write the final sum to the output.
    sum_dw = tl.sum(dw, axis=0)
    sum_db = tl.sum(db, axis=0)
    tl.store(FINAL_DW + cols, sum_dw, mask=cols < N)
    tl.store(FINAL_DB + cols, sum_db, mask=cols < N)


# %%
# Benchmark
# ---------
#
# We can now compare the performance of our kernel against that of PyTorch.
# Here we focus on inputs that have Less than 64KB per feature.
# Specifically, one can set :code:`'mode': 'backward'` to benchmark the
backward pass.


class LayerNorm(torch.autograd.Function):

    @staticmethod
    def forward(ctx, x, normalized_shape, weight, bias, eps):
        # allocate output
        y = torch.empty_like(x)
        # reshape input data into 2D tensor
        x_arg = x.reshape(-1, x.shape[-1])
        M, N = x_arg.shape
        mean = torch.empty((M, ), dtype=torch.float32, device='cuda')
        rstd = torch.empty((M, ), dtype=torch.float32, device='cuda')
        # Less than 64KB per feature: enqueue fused kernel
        MAX_FUSED_SIZE = 65536 // x.element_size()
        BLOCK_SIZE = min(MAX_FUSED_SIZE, triton.next_power_of_2(N))
        if N > BLOCK_SIZE:
            raise RuntimeError("This layer norm doesn't support feature dim
>= 64KB.")
        # heuristics for number of warps
        num_warps = min(max(BLOCK_SIZE // 256, 1), 8)
        # enqueue kernel
        _layer_norm_fwd_fused[(M, )](  #
            x_arg, y, weight, bias, mean, rstd,  #
            x_arg.stride(0), N, eps,  #
            BLOCK_SIZE=BLOCK_SIZE, num_warps=num_warps, num_ctas=1)
        ctx.save_for_backward(x, weight, bias, mean, rstd)
        ctx.BLOCK_SIZE = BLOCK_SIZE
        ctx.num_warps = num_warps
        ctx.eps = eps
        return y
```

```
    @staticmethod
    def backward(ctx, dy):
        x, w, b, m, v = ctx.saved_tensors
        # heuristics for amount of parallel reduction stream for DW/DB
        N = w.shape[0]
        GROUP_SIZE_M = 64
        if N <= 8192: GROUP_SIZE_M = 96
        if N <= 4096: GROUP_SIZE_M = 128
        if N <= 1024: GROUP_SIZE_M = 256
        # allocate output
        locks = torch.zeros(2 * GROUP_SIZE_M, dtype=torch.int32,
device='cuda')
        _dw = torch.empty((GROUP_SIZE_M, w.shape[0]), dtype=x.dtype,
device=w.device)
        _db = torch.empty((GROUP_SIZE_M, w.shape[0]), dtype=x.dtype,
device=w.device)
        dw = torch.empty((w.shape[0], ), dtype=w.dtype, device=w.device)
        db = torch.empty((w.shape[0], ), dtype=w.dtype, device=w.device)
        dx = torch.empty_like(dy)
        # enqueue kernel using forward pass heuristics
        # also compute partial sums for DW and DB
        x_arg = x.reshape(-1, x.shape[-1])
        M, N = x_arg.shape
        _layer_norm_bwd_dx_fused[(M, )](  #
            dx, dy, _dw, _db, x, w, b, m, v, locks,  #
            x_arg.stride(0), N, ctx.eps,  #
            BLOCK_SIZE_N=ctx.BLOCK_SIZE,  #
            GROUP_SIZE_M=GROUP_SIZE_M,  #
            num_warps=ctx.num_warps)
        grid = lambda meta: [triton.cdiv(N, meta['BLOCK_SIZE_N'])]
        # accumulate partial sums in separate kernel
        _layer_norm_bwd_dwdb[grid](
            _dw, _db, dw, db, GROUP_SIZE_M, N,  #
            BLOCK_SIZE_M=32,  #
            BLOCK_SIZE_N=128, num_ctas=1)
        return dx, None, dw, db, None


layer_norm = LayerNorm.apply


def test_layer_norm(M, N, dtype, eps=1e-5, device='cuda'):
    # create data
    x_shape = (M, N)
    w_shape = (x_shape[-1], )
    weight = torch.rand(w_shape, dtype=dtype, device='cuda',
requires_grad=True)
    bias = torch.rand(w_shape, dtype=dtype, device='cuda',
requires_grad=True)
    x = -2.3 + 0.5 * torch.randn(x_shape, dtype=dtype, device='cuda')
    dy = .1 * torch.randn_like(x)
    x.requires_grad_(True)
    # forward pass
    y_tri = layer_norm(x, w_shape, weight, bias, eps)
```

```
    y_ref = torch.nn.functional.layer_norm(x, w_shape, weight, bias,
eps).to(dtype)
    # backward pass (triton)
    y_tri.backward(dy, retain_graph=True)
    dx_tri, dw_tri, db_tri = [_.grad.clone() for _ in [x, weight, bias]]
    x.grad, weight.grad, bias.grad = None, None, None
    # backward pass (torch)
    y_ref.backward(dy, retain_graph=True)
    dx_ref, dw_ref, db_ref = [_.grad.clone() for _ in [x, weight, bias]]
    # compare
    assert torch.allclose(y_tri, y_ref, atol=1e-2, rtol=0)
    assert torch.allclose(dx_tri, dx_ref, atol=1e-2, rtol=0)
    assert torch.allclose(db_tri, db_ref, atol=1e-2, rtol=0)
    assert torch.allclose(dw_tri, dw_ref, atol=1e-2, rtol=0)


@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=['N'],
        x_vals=[512 * i for i in range(2, 32)],
        line_arg='provider',
        line_vals=['triton', 'torch'] + (['apex'] if HAS_APEX else []),
        line_names=['Triton', 'Torch'] + (['Apex'] if HAS_APEX else []),
        styles=[('blue', '-'), ('green', '-'), ('orange', '-')],
        ylabel='GB/s',
        plot_name='layer-norm-backward',
        args={'M': 4096, 'dtype': torch.float16, 'mode': 'backward'},
    ))
def bench_layer_norm(M, N, dtype, provider, mode='backward', eps=1e-5,
device='cuda'):
    # create data
    x_shape = (M, N)
    w_shape = (x_shape[-1], )
    weight = torch.rand(w_shape, dtype=dtype, device='cuda',
requires_grad=True)
    bias = torch.rand(w_shape, dtype=dtype, device='cuda',
requires_grad=True)
    x = -2.3 + 0.5 * torch.randn(x_shape, dtype=dtype, device='cuda')
    dy = .1 * torch.randn_like(x)
    x.requires_grad_(True)
    quantiles = [0.5, 0.2, 0.8]
    # utility functions
    if provider == 'triton':

        def y_fwd():
            return layer_norm(x, w_shape, weight, bias, eps)  # noqa: F811,
E704

    if provider == 'torch':

        def y_fwd():
            return torch.nn.functional.layer_norm(x, w_shape, weight, bias,
eps)  # noqa: F811, E704
```

```
    if provider == 'apex':
        apex_layer_norm =
apex.normalization.FusedLayerNorm(w_shape).to(x.device).to(x.dtype)

        def y_fwd():
            return apex_layer_norm(x)  # noqa: F811, E704

    # forward pass
    if mode == 'forward':
        gbps = lambda ms: 2 * x.numel() * x.element_size() / ms * 1e-6
        ms, min_ms, max_ms = triton.testing.do_bench(y_fwd,
quantiles=quantiles, rep=500)
    # backward pass
    if mode == 'backward':

        def gbps(ms):
            return 3 * x.numel() * x.element_size() / ms * 1e-6  # noqa:
F811, E704

        y = y_fwd()
        ms, min_ms, max_ms = triton.testing.do_bench(lambda: y.backward(dy,
retain_graph=True), quantiles=quantiles,
                                                     grad_to_none=[x],
rep=500)
    return gbps(ms), gbps(max_ms), gbps(min_ms)


test_layer_norm(1151, 8192, torch.float16)
bench_layer_norm.run(save_path='.', print_data=True)

# %%
# References
# ----------
#
# .. [BA2016] Jimmy Lei Ba and Jamie Ryan Kiros and Geoffrey E. Hinton,
"Layer Normalization", Arxiv 2016
```

fused attention

```
"""
Fused Attention
===============

This is a Triton implementation of the Flash Attention v2 algorithm from
Tri Dao (https://tridao.me/publications/flash2/flash2.pdf)
Credits: OpenAI kernel team

Extra Credits:
- Original flash attention paper (https://arxiv.org/abs/2205.14135)
- Rabe and Staats (https://arxiv.org/pdf/2112.05682v2.pdf)
```

```
"""

import pytest
import torch

import triton
import triton.language as tl


@triton.jit
def _attn_fwd_inner(acc, l_i, m_i, q,  #
                    K_block_ptr, V_block_ptr,  #
                    start_m, qk_scale,  #
                    BLOCK_M: tl.constexpr, BLOCK_DMODEL: tl.constexpr,
BLOCK_N: tl.constexpr,  #
                    STAGE: tl.constexpr, offs_m: tl.constexpr, offs_n:
tl.constexpr,  #
                    N_CTX: tl.constexpr):
    # range of values handled by this stage
    if STAGE == 1:
        lo, hi = 0, start_m * BLOCK_M
    elif STAGE == 2:
        lo, hi = start_m * BLOCK_M, (start_m + 1) * BLOCK_M
        lo = tl.multiple_of(lo, BLOCK_M)
    # causal = False
    else:
        lo, hi = 0, N_CTX
    K_block_ptr = tl.advance(K_block_ptr, (0, lo))
    V_block_ptr = tl.advance(V_block_ptr, (lo, 0))
    # loop over k, v and update accumulator
    for start_n in range(lo, hi, BLOCK_N):
        start_n = tl.multiple_of(start_n, BLOCK_N)
        # -- compute qk ----
        k = tl.load(K_block_ptr)
        qk = tl.zeros([BLOCK_M, BLOCK_N], dtype=tl.float32)
        qk += tl.dot(q, k)
        if STAGE == 2:
            mask = offs_m[:, None] >= (start_n + offs_n[None, :])
            qk = qk * qk_scale + tl.where(mask, 0, -1.0e6)
            m_ij = tl.maximum(m_i, tl.max(qk, 1))
            qk -= m_ij[:, None]
        else:
            m_ij = tl.maximum(m_i, tl.max(qk, 1) * qk_scale)
            qk = qk * qk_scale - m_ij[:, None]
        p = tl.math.exp2(qk)
        l_ij = tl.sum(p, 1)
        # -- update m_i and l_i
        alpha = tl.math.exp2(m_i - m_ij)
        l_i = l_i * alpha + l_ij
        # -- update output accumulator --
        acc = acc * alpha[:, None]
        # update acc
        v = tl.load(V_block_ptr)
```

```
            acc += tl.dot(p.to(tl.float16), v)
            # update m_i and l_i
            m_i = m_ij
            V_block_ptr = tl.advance(V_block_ptr, (BLOCK_N, 0))
            K_block_ptr = tl.advance(K_block_ptr, (0, BLOCK_N))
        return acc, l_i, m_i


# We don't run auto-tuning everytime to keep the tutorial fast.
Uncommenting
# the code below and commenting out the equivalent parameters is convenient
for
# re-tuning.
# @triton.autotune(
#    configs=[
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 64}, num_stages=4,
num_warps=8),
#        triton.Config({'BLOCK_M': 256, 'BLOCK_N': 64}, num_stages=3,
num_warps=8),
#        triton.Config({'BLOCK_M': 256, 'BLOCK_N': 32}, num_stages=3,
num_warps=8),
#        triton.Config({'BLOCK_M': 256, 'BLOCK_N': 32}, num_stages=3,
num_warps=4),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 32}, num_stages=3,
num_warps=4),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 32}, num_stages=4,
num_warps=4),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 64}, num_stages=3,
num_warps=4),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 64}, num_stages=4,
num_warps=4),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 64}, num_stages=3,
num_warps=8),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 64}, num_stages=7,
num_warps=8),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 32}, num_stages=7,
num_warps=8),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 32}, num_stages=6,
num_warps=8),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 32}, num_stages=5,
num_warps=8),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 32}, num_stages=4,
num_warps=8),
#        triton.Config({'BLOCK_M': 128, 'BLOCK_N': 64}, num_stages=6,
num_warps=4),
#    ],
#    key=['N_CTX'],
# )


@triton.jit
def _attn_fwd(Q, K, V, sm_scale, M, Out,  #
              stride_qz, stride_qh, stride_qm, stride_qk,  #
              stride_kz, stride_kh, stride_kn, stride_kk,  #
```

```
                stride_vz, stride_vh, stride_vk, stride_vn,  #
                stride_oz, stride_oh, stride_om, stride_on,  #
                Z, H,  #
                N_CTX: tl.constexpr,  #
                BLOCK_M: tl.constexpr,  #
                BLOCK_DMODEL: tl.constexpr,  #
                BLOCK_N: tl.constexpr,  #
                STAGE: tl.constexpr  #
                ):
    start_m = tl.program_id(0)
    off_hz = tl.program_id(1)
    off_z = off_hz // H
    off_h = off_hz % H
    qvk_offset = off_z.to(tl.int64) * stride_qz + off_h.to(tl.int64) *
stride_qh

    # block pointers
    Q_block_ptr = tl.make_block_ptr(
        base=Q + qvk_offset,
        shape=(N_CTX, BLOCK_DMODEL),
        strides=(stride_qm, stride_qk),
        offsets=(start_m * BLOCK_M, 0),
        block_shape=(BLOCK_M, BLOCK_DMODEL),
        order=(1, 0),
    )
    V_block_ptr = tl.make_block_ptr(
        base=V + qvk_offset,
        shape=(N_CTX, BLOCK_DMODEL),
        strides=(stride_vk, stride_vn),
        offsets=(0, 0),
        block_shape=(BLOCK_N, BLOCK_DMODEL),
        order=(1, 0),
    )
    K_block_ptr = tl.make_block_ptr(
        base=K + qvk_offset,
        shape=(BLOCK_DMODEL, N_CTX),
        strides=(stride_kk, stride_kn),
        offsets=(0, 0),
        block_shape=(BLOCK_DMODEL, BLOCK_N),
        order=(0, 1),
    )
    O_block_ptr = tl.make_block_ptr(
        base=Out + qvk_offset,
        shape=(N_CTX, BLOCK_DMODEL),
        strides=(stride_om, stride_on),
        offsets=(start_m * BLOCK_M, 0),
        block_shape=(BLOCK_M, BLOCK_DMODEL),
        order=(1, 0),
    )
    # initialize offsets
    offs_m = start_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = tl.arange(0, BLOCK_N)
    # initialize pointer to m and l
    m_i = tl.zeros([BLOCK_M], dtype=tl.float32) - float("inf")
```

```
    l_i = tl.zeros([BLOCK_M], dtype=tl.float32) + 1.0
    acc = tl.zeros([BLOCK_M, BLOCK_DMODEL], dtype=tl.float32)
    # load scales
    qk_scale = sm_scale
    qk_scale *= 1.44269504  # 1/log(2)
    # load q: it will stay in SRAM throughout
    q = tl.load(Q_block_ptr)
    # stage 1: off-band
    # For causal = True, STAGE = 3 and _attn_fwd_inner gets 1 as its STAGE
    # For causal = False, STAGE = 1, and _attn_fwd_inner gets 3 as its
STAGE
    if STAGE & 1:
        acc, l_i, m_i = _attn_fwd_inner(acc, l_i, m_i, q, K_block_ptr,
V_block_ptr,  #
                                        start_m, qk_scale,  #
                                        BLOCK_M, BLOCK_DMODEL, BLOCK_N,  #
                                        4 - STAGE, offs_m, offs_n, N_CTX  #
                                        )
    # stage 2: on-band
    if STAGE & 2:
        # barrier makes it easier for compielr to schedule the
        # two loops independently
        tl.debug_barrier()
        acc, l_i, m_i = _attn_fwd_inner(acc, l_i, m_i, q, K_block_ptr,
V_block_ptr,  #
                                        start_m, qk_scale,  #
                                        BLOCK_M, BLOCK_DMODEL, BLOCK_N,  #
                                        2, offs_m, offs_n, N_CTX  #
                                        )
    # epilogue
    m_i += tl.math.log2(l_i)
    acc = acc / l_i[:, None]
    m_ptrs = M + off_hz * N_CTX + offs_m
    tl.store(m_ptrs, m_i)
    tl.store(O_block_ptr, acc.to(Out.type.element_ty))


@triton.jit
def _attn_bwd_preprocess(O, DO,  #
                         Delta,  #
                         Z, H, N_CTX,  #
                         BLOCK_M: tl.constexpr, D_HEAD: tl.constexpr  #
                         ):
    off_m = tl.program_id(0) * BLOCK_M + tl.arange(0, BLOCK_M)
    off_hz = tl.program_id(1)
    off_n = tl.arange(0, D_HEAD)
    # load
    o = tl.load(O + off_hz * D_HEAD * N_CTX + off_m[:, None] * D_HEAD +
off_n[None, :])
    do = tl.load(DO + off_hz * D_HEAD * N_CTX + off_m[:, None] * D_HEAD +
off_n[None, :]).to(tl.float32)
    delta = tl.sum(o * do, axis=1)
    # write-back
    tl.store(Delta + off_hz * N_CTX + off_m, delta)
```

```
# The main inner-loop logic for computing dK and dV.
@triton.jit
def _attn_bwd_dkdv(dk, dv,  #
                   Q, k, v, sm_scale,  #
                   DO,  #
                   M, D,  #
                   # shared by Q/K/V/DO.
                   stride_tok, stride_d,  #
                   H, N_CTX, BLOCK_M1: tl.constexpr,  #
                   BLOCK_N1: tl.constexpr,  #
                   BLOCK_DMODEL: tl.constexpr,  #
                   # Filled in by the wrapper.
                   start_n, start_m, num_steps,  #
                   MASK: tl.constexpr):
    offs_m = start_m + tl.arange(0, BLOCK_M1)
    offs_n = start_n + tl.arange(0, BLOCK_N1)
    offs_k = tl.arange(0, BLOCK_DMODEL)
    qT_ptrs = Q + offs_m[None, :] * stride_tok + offs_k[:, None] * stride_d
    do_ptrs = DO + offs_m[:, None] * stride_tok + offs_k[None, :] *
stride_d
    # BLOCK_N1 must be a multiple of BLOCK_M1, otherwise the code wouldn't
work.
    tl.static_assert(BLOCK_N1 % BLOCK_M1 == 0)
    curr_m = start_m
    step_m = BLOCK_M1
    for blk_idx in range(num_steps):
        qT = tl.load(qT_ptrs)
        # Load m before computing qk to reduce pipeline stall.
        offs_m = curr_m + tl.arange(0, BLOCK_M1)
        m = tl.load(M + offs_m)
        qkT = tl.dot(k, qT)
        pT = tl.math.exp2(qkT - m[None, :])
        # Autoregressive masking.
        if MASK:
            mask = (offs_m[None, :] >= offs_n[:, None])
            pT = tl.where(mask, pT, 0.0)
        do = tl.load(do_ptrs)
        # Compute dV.
        ppT = pT
        ppT = ppT.to(tl.float16)
        dv += tl.dot(ppT, do)
        # D (= delta) is pre-divided by ds_scale.
        Di = tl.load(D + offs_m)
        # Compute dP and dS.
        dpT = tl.dot(v, tl.trans(do)).to(tl.float32)
        dsT = pT * (dpT - Di[None, :])
        dsT = dsT.to(tl.float16)
        dk += tl.dot(dsT, tl.trans(qT))
        # Increment pointers.
        curr_m += step_m
        qT_ptrs += step_m * stride_tok
        do_ptrs += step_m * stride_tok
```

```
    return dk, dv


# the main inner-loop logic for computing dQ
@triton.jit
def _attn_bwd_dq(dq, q, K, V,  #
                 do, m, D,
                 # shared by Q/K/V/DO.
                 stride_tok, stride_d,  #
                 H, N_CTX,  #
                 BLOCK_M2: tl.constexpr,  #
                 BLOCK_N2: tl.constexpr,  #
                 BLOCK_DMODEL: tl.constexpr,
                 # Filled in by the wrapper.
                 start_m, start_n, num_steps,  #
                 MASK: tl.constexpr):
    offs_m = start_m + tl.arange(0, BLOCK_M2)
    offs_n = start_n + tl.arange(0, BLOCK_N2)
    offs_k = tl.arange(0, BLOCK_DMODEL)
    kT_ptrs = K + offs_n[None, :] * stride_tok + offs_k[:, None] * stride_d
    vT_ptrs = V + offs_n[None, :] * stride_tok + offs_k[:, None] * stride_d
    # D (= delta) is pre-divided by ds_scale.
    Di = tl.load(D + offs_m)
    # BLOCK_M2 must be a multiple of BLOCK_N2, otherwise the code wouldn't
work.
    tl.static_assert(BLOCK_M2 % BLOCK_N2 == 0)
    curr_n = start_n
    step_n = BLOCK_N2
    for blk_idx in range(num_steps):
        kT = tl.load(kT_ptrs)
        vT = tl.load(vT_ptrs)
        qk = tl.dot(q, kT)
        p = tl.math.exp2(qk - m)
        # Autoregressive masking.
        if MASK:
            offs_n = curr_n + tl.arange(0, BLOCK_N2)
            mask = (offs_m[:, None] >= offs_n[None, :])
            p = tl.where(mask, p, 0.0)
        # Compute dP and dS.
        dp = tl.dot(do, vT).to(tl.float32)
        ds = p * (dp - Di[:, None])
        ds = ds.to(tl.float16)
        # Compute dQ.
        # NOTE: We need to de-scale dq in the end, because kT was pre-
scaled.
        dq += tl.dot(ds, tl.trans(kT))
        # Increment pointers.
        curr_n += step_n
        kT_ptrs += step_n * stride_tok
        vT_ptrs += step_n * stride_tok
    return dq


@triton.jit
```

```
def _attn_bwd(Q, K, V, sm_scale,  #
              DO,  #
              DQ, DK, DV,  #
              M, D,
              # shared by Q/K/V/DO.
              stride_z, stride_h, stride_tok, stride_d,  #
              H, N_CTX,  #
              BLOCK_M1: tl.constexpr,  #
              BLOCK_N1: tl.constexpr,  #
              BLOCK_M2: tl.constexpr,  #
              BLOCK_N2: tl.constexpr,  #
              BLK_SLICE_FACTOR: tl.constexpr,  #
              BLOCK_DMODEL: tl.constexpr):
    LN2: tl.constexpr = 0.6931471824645996  # = ln(2)

    bhid = tl.program_id(2)
    off_chz = (bhid * N_CTX).to(tl.int64)
    adj = (stride_h * (bhid % H) + stride_z * (bhid // H)).to(tl.int64)
    pid = tl.program_id(0)

    # offset pointers for batch/head
    Q += adj
    K += adj
    V += adj
    DO += adj
    DQ += adj
    DK += adj
    DV += adj
    M += off_chz
    D += off_chz

    # load scales
    offs_k = tl.arange(0, BLOCK_DMODEL)

    start_n = pid * BLOCK_N1
    start_m = start_n

    MASK_BLOCK_M1: tl.constexpr = BLOCK_M1 // BLK_SLICE_FACTOR
    offs_n = start_n + tl.arange(0, BLOCK_N1)

    dv = tl.zeros([BLOCK_N1, BLOCK_DMODEL], dtype=tl.float32)
    dk = tl.zeros([BLOCK_N1, BLOCK_DMODEL], dtype=tl.float32)

    # load K and V: they stay in SRAM throughout the inner loop.
    k = tl.load(K + offs_n[:, None] * stride_tok + offs_k[None, :] *
stride_d)
    v = tl.load(V + offs_n[:, None] * stride_tok + offs_k[None, :] *
stride_d)

    num_steps = BLOCK_N1 // MASK_BLOCK_M1

    dk, dv = _attn_bwd_dkdv(dk, dv,  #
                            Q, k, v, sm_scale,  #
                            DO,  #
```

```
                                M, D,  #
                                stride_tok, stride_d,  #
                                H, N_CTX,  #
                                MASK_BLOCK_M1, BLOCK_N1, BLOCK_DMODEL,  #
                                start_n, start_m, num_steps,  #
                                MASK=True  #
                                )

    start_m += num_steps * MASK_BLOCK_M1
    num_steps = (N_CTX - start_m) // BLOCK_M1

    # Compute dK and dV for non-masked blocks.
    dk, dv = _attn_bwd_dkdv(  #
        dk, dv,  #
        Q, k, v, sm_scale,  #
        DO,  #
        M, D,  #
        stride_tok, stride_d,  #
        H, N_CTX,  #
        BLOCK_M1, BLOCK_N1, BLOCK_DMODEL,  #
        start_n, start_m, num_steps,  #
        MASK=False  #
    )

    dv_ptrs = DV + offs_n[:, None] * stride_tok + offs_k[None, :] *
stride_d
    tl.store(dv_ptrs, dv)

    # Write back dK.
    dk *= sm_scale
    dk_ptrs = DK + offs_n[:, None] * stride_tok + offs_k[None, :] *
stride_d
    tl.store(dk_ptrs, dk)

    # THIS BLOCK DOES DQ:
    start_m = pid * BLOCK_M2
    end_n = start_m + BLOCK_M2

    MASK_BLOCK_N2: tl.constexpr = BLOCK_N2 // BLK_SLICE_FACTOR
    offs_m = start_m + tl.arange(0, BLOCK_M2)

    q = tl.load(Q + offs_m[:, None] * stride_tok + offs_k[None, :] *
stride_d)
    dq = tl.zeros([BLOCK_M2, BLOCK_DMODEL], dtype=tl.float32)
    do = tl.load(DO + offs_m[:, None] * stride_tok + offs_k[None, :] *
stride_d)

    m = tl.load(M + offs_m)
    m = m[:, None]

    # Compute dQ for masked (diagonal) blocks.
    # NOTE: This code scans each row of QK^T backward (from right to left,
    # but inside each call to _attn_bwd_dq, from left to right), but that's
    # not due to anything important.  I just wanted to reuse the loop
```

```
    # structure for dK & dV above as much as possible.
    num_steps = BLOCK_M2 // MASK_BLOCK_N2
    dq = _attn_bwd_dq(dq, q, K, V,  #
                      do, m, D,  #
                      stride_tok, stride_d,  #
                      H, N_CTX,  #
                      BLOCK_M2, MASK_BLOCK_N2, BLOCK_DMODEL,  #
                      start_m, end_n - num_steps * MASK_BLOCK_N2,
num_steps,  #
                      MASK=True  #
                      )
    end_n -= num_steps * MASK_BLOCK_N2
    # stage 2
    num_steps = end_n // BLOCK_N2
    dq = _attn_bwd_dq(dq, q, K, V,  #
                      do, m, D,  #
                      stride_tok, stride_d,  #
                      H, N_CTX,  #
                      BLOCK_M2, BLOCK_N2, BLOCK_DMODEL,  #
                      start_m, end_n - num_steps * BLOCK_N2, num_steps,  #
                      MASK=False  #
                      )
    # Write back dQ.
    dq_ptrs = DQ + offs_m[:, None] * stride_tok + offs_k[None, :] *
stride_d
    dq *= LN2
    tl.store(dq_ptrs, dq)


empty = torch.empty(128, device="cuda")


class _attention(torch.autograd.Function):

    @staticmethod
    def forward(ctx, q, k, v, causal, sm_scale):
        # shape constraints
        Lq, Lk, Lv = q.shape[-1], k.shape[-1], v.shape[-1]
        assert Lq == Lk and Lk == Lv
        assert Lk in {16, 32, 64, 128}
        o = torch.empty_like(q)
        BLOCK_M = 128
        BLOCK_N = 64 if Lk <= 64 else 32
        num_stages = 4 if Lk <= 64 else 3
        num_warps = 4
        stage = 3 if causal else 1
        # Tuning for H100
        if torch.cuda.get_device_capability()[0] == 9:
            num_warps = 8
            num_stages = 7 if Lk >= 64 else 3
        grid = (triton.cdiv(q.shape[2], BLOCK_M), q.shape[0] * q.shape[1],
1)
        M = torch.empty((q.shape[0], q.shape[1], q.shape[2]),
device=q.device, dtype=torch.float32)
```

```
        _attn_fwd[grid](
            q, k, v, sm_scale, M, o,  #
            q.stride(0), q.stride(1), q.stride(2), q.stride(3),  #
            k.stride(0), k.stride(1), k.stride(2), k.stride(3),  #
            v.stride(0), v.stride(1), v.stride(2), v.stride(3),  #
            o.stride(0), o.stride(1), o.stride(2), o.stride(3),  #
            q.shape[0], q.shape[1],  #
            N_CTX=q.shape[2],  #
            BLOCK_M=BLOCK_M,  #
            BLOCK_N=BLOCK_N,  #
            BLOCK_DMODEL=Lk,  #
            STAGE=stage,  #
            num_warps=num_warps,  #
            num_stages=num_stages  #
        )

        ctx.save_for_backward(q, k, v, o, M)
        ctx.grid = grid
        ctx.sm_scale = sm_scale
        ctx.BLOCK_DMODEL = Lk
        ctx.causal = causal
        return o

    @staticmethod
    def backward(ctx, do):
        q, k, v, o, M = ctx.saved_tensors
        assert do.is_contiguous()
        assert q.stride() == k.stride() == v.stride() == o.stride() ==
do.stride()
        dq = torch.empty_like(q)
        dk = torch.empty_like(k)
        dv = torch.empty_like(v)
        BATCH, N_HEAD, N_CTX = q.shape[:3]
        PRE_BLOCK = 128
        NUM_WARPS, NUM_STAGES = 4, 5
        BLOCK_M1, BLOCK_N1, BLOCK_M2, BLOCK_N2 = 32, 128, 128, 32
        BLK_SLICE_FACTOR = 2
        RCP_LN2 = 1.4426950408889634  # = 1.0 / ln(2)
        arg_k = k
        arg_k = arg_k * (ctx.sm_scale * RCP_LN2)
        PRE_BLOCK = 128
        assert N_CTX % PRE_BLOCK == 0
        pre_grid = (N_CTX // PRE_BLOCK, BATCH * N_HEAD)
        delta = torch.empty_like(M)
        _attn_bwd_preprocess[pre_grid](
            o, do,  #
            delta,  #
            BATCH, N_HEAD, N_CTX,  #
            BLOCK_M=PRE_BLOCK, D_HEAD=ctx.BLOCK_DMODEL  #
        )
        grid = (N_CTX // BLOCK_N1, 1, BATCH * N_HEAD)
        _attn_bwd[grid](
            q, arg_k, v, ctx.sm_scale, do, dq, dk, dv,  #
            M, delta,  #
```

```
            q.stride(0), q.stride(1), q.stride(2), q.stride(3),  #
            N_HEAD, N_CTX,  #
            BLOCK_M1=BLOCK_M1, BLOCK_N1=BLOCK_N1,  #
            BLOCK_M2=BLOCK_M2, BLOCK_N2=BLOCK_N2,  #
            BLK_SLICE_FACTOR=BLK_SLICE_FACTOR,  #
            BLOCK_DMODEL=ctx.BLOCK_DMODEL,  #
            num_warps=NUM_WARPS,  #
            num_stages=NUM_STAGES  #
        )

        return dq, dk, dv, None, None


attention = _attention.apply


@pytest.mark.parametrize("Z, H, N_CTX, D_HEAD", [(1, 2, 1024, 64)])
@pytest.mark.parametrize("causal", [True])
def test_op(Z, H, N_CTX, D_HEAD, causal, dtype=torch.float16):
    torch.manual_seed(20)
    q = (torch.empty((Z, H, N_CTX, D_HEAD), dtype=dtype,
device="cuda").normal_(mean=0.0, std=0.5).requires_grad_())
    k = (torch.empty((Z, H, N_CTX, D_HEAD), dtype=dtype,
device="cuda").normal_(mean=0.0, std=0.5).requires_grad_())
    v = (torch.empty((Z, H, N_CTX, D_HEAD), dtype=dtype,
device="cuda").normal_(mean=0.0, std=0.5).requires_grad_())
    sm_scale = 0.5
    dout = torch.randn_like(q)
    # reference implementation
    M = torch.tril(torch.ones((N_CTX, N_CTX), device="cuda"))
    p = torch.matmul(q, k.transpose(2, 3)) * sm_scale
    if causal:
        p[:, :, M == 0] = float("-inf")
    p = torch.softmax(p.float(), dim=-1).half()
    # p = torch.exp(p)
    ref_out = torch.matmul(p, v)
    ref_out.backward(dout)
    ref_dv, v.grad = v.grad.clone(), None
    ref_dk, k.grad = k.grad.clone(), None
    ref_dq, q.grad = q.grad.clone(), None
    # triton implementation
    tri_out = attention(q, k, v, causal, sm_scale).half()
    tri_out.backward(dout)
    tri_dv, v.grad = v.grad.clone(), None
    tri_dk, k.grad = k.grad.clone(), None
    tri_dq, q.grad = q.grad.clone(), None
    # compare
    assert torch.allclose(ref_out, tri_out, atol=1e-2, rtol=0)
    assert torch.allclose(ref_dv, tri_dv, atol=1e-2, rtol=0)
    assert torch.allclose(ref_dk, tri_dk, atol=1e-2, rtol=0)
    assert torch.allclose(ref_dq, tri_dq, atol=1e-2, rtol=0)


try:
```

```
    from flash_attn.flash_attn_interface import \
        flash_attn_qkvpacked_func as flash_attn_func
    HAS_FLASH = True
except BaseException:
    HAS_FLASH = False

TORCH_HAS_FP8 = hasattr(torch, 'float8_e5m2')
BATCH, N_HEADS, N_CTX, D_HEAD = 4, 48, 4096, 64
# vary seq length for fixed head and batch=4
configs = []
for mode in ["fwd", "bwd"]:
    for causal in [True, False]:
        if mode == "bwd" and not causal:
            continue
        configs.append(
            triton.testing.Benchmark(
                x_names=["N_CTX"],
                x_vals=[2**i for i in range(10, 15)],
                line_arg="provider",
                line_vals=["triton"] + (["flash"] if HAS_FLASH else []),
                line_names=["Triton"] + (["Flash-2"] if HAS_FLASH else []),
                styles=[("red", "-"), ("blue", "-")],
                ylabel="ms",
                plot_name=f"fused-attention-batch{BATCH}-head{N_HEADS}-
d{D_HEAD}-{mode}-causal={causal}",
                args={
                    "H": N_HEADS,
                    "BATCH": BATCH,
                    "D_HEAD": D_HEAD,
                    "dtype": torch.float16,
                    "mode": mode,
                    "causal": causal,
                },
            ))


@triton.testing.perf_report(configs)
def bench_flash_attention(BATCH, H, N_CTX, D_HEAD, causal, mode, provider,
dtype=torch.float16, device="cuda"):
    assert mode in ["fwd", "bwd"]
    warmup = 25
    rep = 100
    if provider == "triton":
        q = torch.randn((BATCH, H, N_CTX, D_HEAD), dtype=dtype,
device="cuda", requires_grad=True)
        k = torch.randn((BATCH, H, N_CTX, D_HEAD), dtype=dtype,
device="cuda", requires_grad=True)
        if mode == "fwd" and TORCH_HAS_FP8:
            q = q.to(torch.float8_e5m2)
            k = k.to(torch.float8_e5m2)
        v = torch.randn((BATCH, H, N_CTX, D_HEAD), dtype=dtype,
device="cuda", requires_grad=True)
        sm_scale = 1.3
        fn = lambda: attention(q, k, v, causal, sm_scale)
```

```
        if mode == "bwd":
            o = fn()
            do = torch.randn_like(o)
            fn = lambda: o.backward(do, retain_graph=True)
        ms = triton.testing.do_bench(fn, warmup=warmup, rep=rep)
    if provider == "flash":
        qkv = torch.randn((BATCH, N_CTX, 3, H, D_HEAD), dtype=dtype,
device=device, requires_grad=True)
        fn = lambda: flash_attn_func(qkv, causal=causal)
        if mode == "bwd":
            o = fn()
            do = torch.randn_like(o)
            fn = lambda: o.backward(do, retain_graph=True)
        ms = triton.testing.do_bench(fn, warmup=warmup, rep=rep)
    flops_per_matmul = 2.0 * BATCH * H * N_CTX * N_CTX * D_HEAD
    total_flops = 2 * flops_per_matmul
    if causal:
        total_flops *= 0.5
    if mode == "bwd":
        total_flops *= 2.5  # 2.0(bwd) + 0.5(recompute)
    return total_flops / ms * 1e-9


# only works on post-Ampere GPUs right now
bench_flash_attention.run(save_path=".", print_data=True)
```

math function

```
"""
Libdevice (`tl.math`) function
==============================
Triton can invoke a custom function from an external library.
In this example, we will use the `libdevice` library (a.k.a `math` in
triton) to apply `asin` on a tensor.
Please refer to https://docs.nvidia.com/cuda/libdevice-users-
guide/index.html regarding the semantics of all available libdevice
functions.
In `triton/language/math.py`, we try to aggregate functions with the same
computation but different data types together.
For example, both `__nv_asin` and `__nvasinf` calculate the principal value
of the arc sine of the input, but `__nv_asin` operates on `double` and
`__nv_asinf` operates on `float`.
Using triton, you can simply call `tl.math.asin`.
Triton automatically selects the correct underlying device function to
invoke based on input and output types.
"""

# %%
#  asin Kernel
# ------------
```

```
import torch

import triton
import triton.language as tl


@triton.jit
def asin_kernel(
    x_ptr,
    y_ptr,
    n_elements,
    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
    x = tl.load(x_ptr + offsets, mask=mask)
    x = tl.math.asin(x)
    tl.store(y_ptr + offsets, x, mask=mask)


# %%
#  Using the default libdevice library path
# -----------------------------------------
# We can use the default libdevice library path encoded in
`triton/language/math.py`

torch.manual_seed(0)
size = 98432
x = torch.rand(size, device='cuda')
output_triton = torch.zeros(size, device='cuda')
output_torch = torch.asin(x)
assert x.is_cuda and output_triton.is_cuda
n_elements = output_torch.numel()
grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
asin_kernel[grid](x, output_triton, n_elements, BLOCK_SIZE=1024)
print(output_torch)
print(output_triton)
print(f'The maximum difference between torch and triton is '
      f'{torch.max(torch.abs(output_torch - output_triton))}')

# %%
#  Customize the libdevice library path
# -------------------------------------
# We can also customize the libdevice library path by passing the path to
the `libdevice` library to the `asin` kernel.

output_triton = torch.empty_like(x)
asin_kernel[grid](x, output_triton, n_elements, BLOCK_SIZE=1024,
                  extern_libs={'libdevice':
'/usr/local/cuda/nvvm/libdevice/libdevice.10.bc'})
print(output_torch)
```

```
print(output_triton)
print(f'The maximum difference between torch and triton is '
      f'{torch.max(torch.abs(output_torch - output_triton))}')
```