Note

Go to the end to download the full example code.

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
from triton.runtime import driver

DEVICE = triton.runtime.driver.active.get_active_torch_device()


def is_hip():
    return triton.runtime.driver.active.get_current_target().backend ==


def is_cdna():
    return is_hip() and triton.runtime.driver.active.get_current_targe


def naive_softmax(x):
    """Compute row-wise softmax of X using native pytorch

    We subtract the maximum element in order to avoid overflows. Softma
    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 `y = naive_softmax(x)` for $x \in R^{M \times N}$ requires reading $5MN + 2M$ elements from DRAM and writing back $3MN + 2M$ 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 MN bytes, so we could expect a theoretical speed-up of ~4x (i.e., $(8MN + 4M) / 2MN$). 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 set of rows of the input matrix X strided by number of programs, 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_
                   num_stages: tl.constexpr):
    # starting row of the program
    row_start = tl.program_id(0)
    row_step = tl.num_programs(0)
    for row_idx in tl.range(row_start, n_rows, row_step, num_stages=nu
        # The stride represents how much we need to increase the pointe
        row_start_ptr = input_ptr + row_idx * input_row_stride
        # The block size is the next power of two greater than n_cols,
        # 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
        mask = col_offsets < n_cols
        row = tl.load(input_ptrs, mask=mask, 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 (
        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=mask)
```

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

```
properties = driver.active.utils.get_device_properties(DEVICE.index)
NUM_SM = properties["multiprocessor_count"]
NUM_REGS = properties["max_num_regs"]
SIZE_SMEM = properties["max_shared_mem"]
WARP_SIZE = properties["warpSize"]
target = triton.runtime.driver.active.get_current_target()
kernels = {}


def softmax(x):
    n_rows, n_cols = x.shape

    # The block size of each loop iteration is the smallest power of tw
    BLOCK_SIZE = triton.next_power_of_2(n_cols)

    # Another trick we can use is to ask the compiler to use more threa
    # increasing the number of warps (`num_warps`) over which each row
    # You will see in the next tutorial how to auto-tune this value in
    # way so you don't have to come up with manual heuristics yourself.
    num_warps = 8

    # Number of software pipelining stages.
    num_stages = 4 if SIZE_SMEM > 200000 else 2

    # Allocate output
    y = torch.empty_like(x)

    # pre-compile kernel to get register usage and compute thread occup
    kernel = softmax_kernel.warmup(y, x, x.stride(0), y.stride(0), n_ro
                                   num_stages=num_stages, num_warps=nu
    kernel._init_handles()
    n_regs = kernel.n_regs
    size_smem = kernel.metadata.shared
    if is_hip():
        # NUM_REGS represents the number of regular purpose registers.
        # However, this is not always the case. In most cases all regis
        # ISA SECTION (3.6.4 for CDNA3)
        # VGPRs are allocated out of two pools: regular VGPRs and accu
        # with matrix VALU instructions, and can also be loaded direct
        # VGPRs, 256 of each type. When a wave has fewer than 512 tota
        # not required to be equal numbers of both types.
        if is_cdna():
            NUM_GPRS = NUM_REGS * 2

        # MAX_NUM_THREADS represents maximum number of resident threads
        # When we divide this number with WARP_SIZE we get maximum numb
        # execute on a CU (multi-processor)  in parallel.
        MAX_NUM_THREADS = properties["max_threads_per_sm"]
        max_num_waves = MAX_NUM_THREADS // WARP_SIZE
        occupancy = min(NUM_GPRS // WARP_SIZE // n_regs, max_num_waves)
    else:
        occupancy = NUM_REGS // (n_regs * WARP_SIZE * num_warps)
    occupancy = min(occupancy, SIZE_SMEM // size_smem)
    num_programs = NUM_SM * occupancy

    num_programs = min(num_programs, n_rows)

    # Create a number of persistent programs.
    kernel[(num_programs, 1, 1)](y, x, x.stride(0), y.stride(0), n_rows
    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=DEVICE)
y_triton = softmax(x)
y_torch = torch.softmax(x, axis=1)
assert torch.allclose(y_triton, y_torch), (y_triton, y_torch)
```

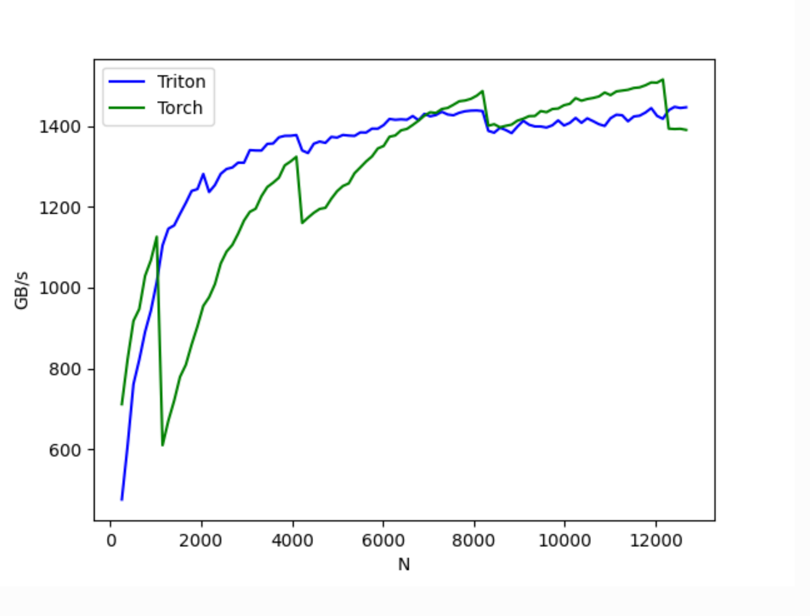
As expected, the results are identical.

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) `torch.softmax` and (2) the `naive_softmax` defined above.

```
@triton.testing.perf_report(
    triton.testing.Benchmark(
        x_names=['N'],  # argument names to use as an x-axis for the p
        x_vals=[128 * i for i in range(2, 100)],  # different possible
        line_arg='provider',  # argument name whose value corresponds
        line_vals=['triton', 'torch'],  # possible values for `line_ar
        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="softmax-performance",  # name for the plot. Used al
        args={'M': 4096},  # values for function arguments not in `x_na
    ))
def benchmark(M, N, provider):
    x = torch.randn(M, N, device=DEVICE, dtype=torch.float32)
    stream = getattr(torch, DEVICE.type).Stream()
    getattr(torch, DEVICE.type).set_stream(stream)
    if provider == 'torch':
        ms = triton.testing.do_bench(lambda: torch.softmax(x, axis=-1))
    if provider == 'triton':
        ms = triton.testing.do_bench(lambda: softmax(x))
    gbps = lambda ms: 2 * x.numel() * x.element_size() * 1e-9 / (ms * 1
    return gbps(ms)


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



Out:

```
softmax-performance:
          N       Triton        Torch
0     256.0   476.046479   711.660224
1     384.0   610.687785   823.622132
2     512.0   761.237735   917.785454
3     640.0   821.846366   947.269530
4     768.0   890.854554  1029.304918
5     896.0   943.567321  1067.973810
6    1024.0  1014.107391  1125.902759
7    1152.0  1104.585451   610.186819
8    1280.0  1145.869035   671.067134
9    1408.0  1153.978342   720.621485
10   1536.0  1182.505924   779.266633
11   1664.0  1209.646945   809.136563
12   1792.0  1239.199090   859.420998
13   1920.0  1243.674859   904.431122
14   2048.0  1281.174058   954.815550
15   2176.0  1236.451253   976.218919
16   2304.0  1253.746038  1008.666748
17   2432.0  1281.514801  1059.845594
18   2560.0  1293.520644  1089.159442
```

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 `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.

Total running time of the script: (0 minutes 23.153 seconds)

Download Jupyter notebook: 02-fused-softmax.ipynb

Download Python source code: 02-fused-softmax.py

Download zipped: 02-fused-softmax.zip

Gallery generated by Sphinx-Gallery