Note

Go to the end to download the full example code.

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

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


@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 pro
               # NOTE: `constexpr` so it can be used as a shape value.
               ):
    # There are multiple 'programs' processing different data. We iden
    # we are here:
    pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is
    # This program will process inputs that are offset from the initia
    # For instance, if you had a vector of length 256 and block_size o
    # 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 ac
    mask = offsets < n_elements
    # Load x and y from DRAM, masking out any extra elements in case th
    # 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.device == DEVICE and y.device == DEVICE and output.device
    n_elements = output.numel()
    # The SPMD launch grid denotes the number of kernel instances that
    # It is analogous to CUDA launch grids. It can be either Tuple[int
    # In this case, we use a 1D grid where the size is the number of b
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),
    # NOTE:
    #  - Each torch.tensor object is implicitly converted into a pointe
    #  - `triton.jit`'ed functions can be indexed with a launch grid to
    #  - 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()` has
    # 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=DEVICE)
y = torch.rand(size, device=DEVICE)
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))}')
```

Out:

```
tensor([1.3713, 1.3076, 0.4940,  ..., 0.6724, 1.2141, 0.9733], de
tensor([1.3713, 1.3076, 0.4940,  ..., 0.6724, 1.2141, 0.9733], de
The maximum difference between torch and triton is 0.0
```

Seems like we're good to go!

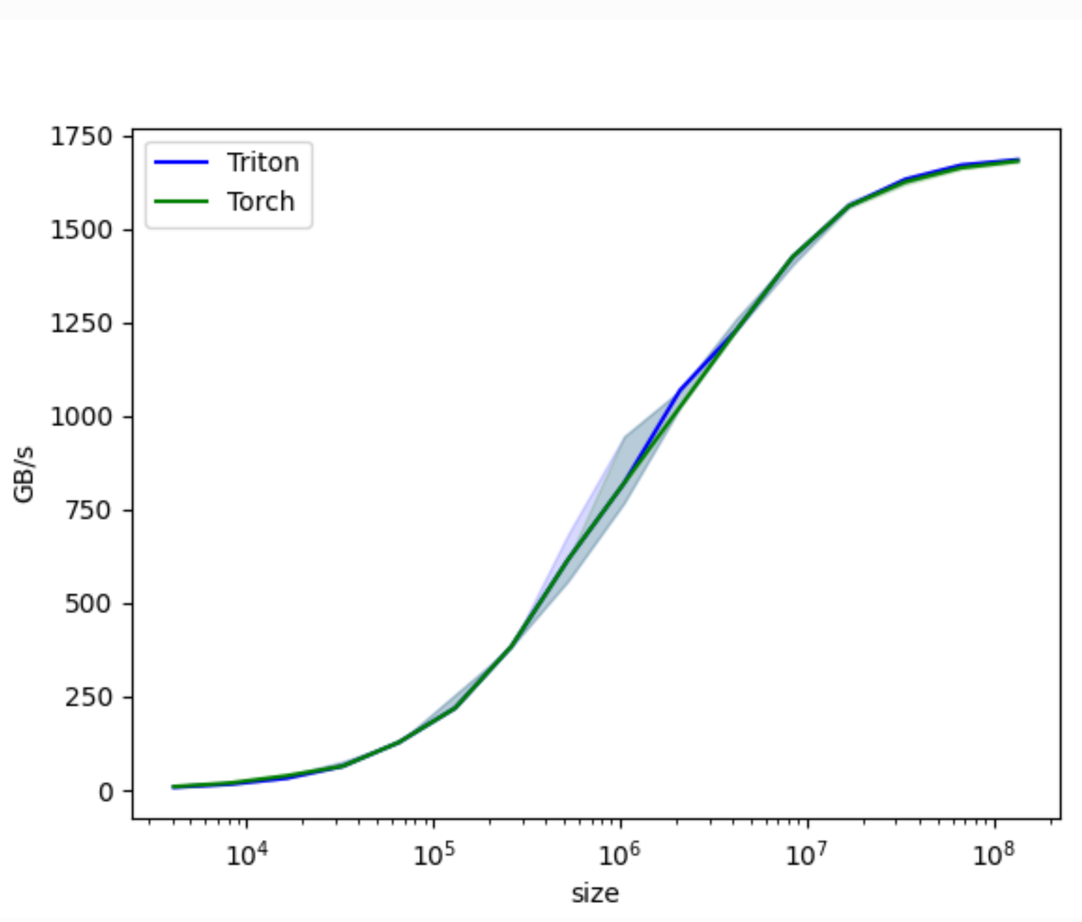
Benchmark

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
        x_vals=[2**i for i in range(12, 28, 1)],  # Different possible
        x_log=True,  # x axis is logarithmic.
        line_arg='provider',  # Argument name whose value corresponds
        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
        args={},  # Values for function arguments not in `x_names` and
    ))
def benchmark(size, provider):
    x = torch.rand(size, device=DEVICE, dtype=torch.float32)
    y = torch.rand(size, device=DEVICE, 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, qua
    if provider == 'triton':
        ms, min_ms, max_ms = triton.testing.do_bench(lambda: add(x, y)
    gbps = lambda ms: 3 * x.numel() * x.element_size() * 1e-9 / (ms * 1
    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)
```



Out:

```
vector-add-performance:
           size       Triton        Torch
0        4096.0     8.000000     9.600000
1        8192.0    15.999999    19.200000
2       16384.0    31.999999    38.400001
3       32768.0    63.999998    63.999998
4       65536.0   127.999995   127.999995
5      131072.0   219.428568   219.428568
6      262144.0   384.000001   384.000001
7      524288.0   614.400016   614.400016
8     1048576.0   819.200021   819.200021
9     2097152.0  1068.521715  1023.999964
10    4194304.0  1228.800031  1228.800031
11    8388608.0  1424.695621  1424.695621
12   16777216.0  1560.380965  1560.380965
13   33554432.0  1631.601649  1624.859540
14   67108864.0  1669.706983  1662.646960
15  134217728.0  1684.008546  1680.410210
```

Total running time of the script: (0 minutes 6.277 seconds)

Download Jupyter notebook: 01-vector-add.ipynb

Download Python source code: 01-vector-add.py

Download zipped: 01-vector-add.zip

Gallery generated by Sphinx-Gallery