CUDA support and development in pytorch

Pytorch provides CUDA implementations for all its native functions.

How to implement cuda support for a new operation?

- Do I need to write a CUDA kernel?

  Likely not, see below

- If operation is expressed through existing pytorch ops, you don't need to, it will automatically be supported on cuda.

- If it's a pointwise operation or reduction, you just need to define a functor to apply to each tensor element, and reuse existing TensorIterator kernels

- Why not write my own kernel? It seems so easy!

  Getting pointwise operation right is surprisingly non-trivial. Pytorch supports many features such as non-contiguous tensors, implicit broadcasting, type promotion. Pytorch is expected to handle tensors with more than INT_MAX elements, and it's easy to run afoul of that writing naive kernels with integer indexing. Your kernel would need to handle all this, at which point it won't be so easy. Besides, existing kernels apply performance optimizations such as unrolling and vectorization.

- Even if it's an irregular operation such as indexing/scatter/gather, try to express it through iteration over tensor elements, in this case, indexing tensor elements and use TI, see examples in the existing indexing/scatter/gather implementations, it will likely be faster and will spare you painful debugging.

- If it's compute intensive operation, such as matrix multiply or convolution, try using existing libraries - cublas and cudnn. Writing compute intensive code requires a lot of expertise

- For common operations such as sort, inclusive/exclusive scan, unique elements use cub library. Don't use thrust!

Common gotchas for writing CUDA code

- If you are writing your kernel, try to use existing utilities to calculate the number of blocks, to perform atomic operations in the kernel, to perform reductions in the block. Additionally, cub also provides block-wide primitives that can be useful.
- Avoid using raw cuda APIs, pytorch typically provides wrappers for those. NEVER allocate memory with cudaMalloc/cudaFree, use only caching allocator
- Avoid host-device synchronizations (can happen if you are copying data from cpu to gpu and back, or call `.item()` on a tensor)

- In pytorch core, codegen takes care of making sure that current device is the same as the device on which tensors are located, and that all arguments are on the same device. If you are writing out-of-core operations, you will need to take care of this yourself

Debugging and profiling tips

- Cuda execution is asynchronous, so backtrace you are getting from cuda error is likely pointing to the wrong place. Error message would typically suggest running with `CUDA_LAUNCH_BLOCKING=1`, do that!
- Use `cuda-memcheck` and `cuda-gdb` to get more detailed information
- you can use `torch.cuda.set_sync_debug_mode` to warn or error out on cuda synchronizations, if you are trying to understand where synchronizations are coming from in your workload or if you are accidentally synchronizing in your operations
- Use pytorch built-in profiler (kineto) or nsys to get information on GPU utilization and most time-consuming kernels

See https://www.dropbox.com/s/3350s3qfy8rpm5a/CUDA_presentation.key?dl=0 for high level overview of CUDA programming and useful links

Goto N1023015 to do TensorIterator cuda perf lab