The most important things to know:

Don't add a kernel to this folder unless you want it to be compiled multiple times for different instruction sets. Yes, this folder is named `cpu`, but that doesn't mean put any old CPU kernel it. Only put CPU kernels which need to be compiled multiple times to take advantage of AVX512/AVX2/SSE instructions, but only on processors that support them.

Ensure that all implementations in this folder are put in an anonymous namespace. The files in this folder are compiled multiple times with different headers. It's important that these functions have internal linkage so that kernels for different architectures don't get combined during linking. It's sufficient to label functions "static", but class methods must be an unnamed namespace to have internal linkage (since static means something different in the context of classes).

The basic recipe is to define your kernel, and then register it using DECLARE/REGISTER DISPATCH. Writing a kernel requires three steps:

1. Declare your dispatch in a header file using `DECLARE_DISPATCH(fn_type, fnNameImpl);` where `fn_type` is the function pointer type of the kernel (e.g., defined as `using fn_type = void(*)(Tensor&, const Tensor&)` and `fnNameImpl` is the name of your dispatch registry. (It doesn't really matter where you put this declaration.)
2. Define your dispatch in a C++ file that is NOT in the cpu directory (dispatch must be defined exactly once) using `DEFINE_DISPATCH(fnNameImpl)` (matching the name of your declaration.) Include the header file that declares the dispatch in this C++ file. Conventionally, we define the dispatch in the same file we will define our native function in.
3. Define a native function which calls into the dispatch using `fnNameImpl(kCPU, arguments...)`, where the arguments are the arguments according to the `fn_type` you defined in the declaration.
4. Write your actual kernel (e.g., `your_kernel`) in the cpu directory, and register it to the dispatch using `REGISTER_DISPATCH(fnNameImpl, &your_kernel)`.

There are plenty of existing examples, look at them for more details.

TODO: Clarify and add more documentation all around.

All of the `*.cpp` files in this folder will be compiled under all compiler flags specified by `CPU_CAPABILITY_FLAGS` in `aten/src/ATen/CMakeLists.txt`.

The purpose of this is to allow the compilation with various compiler flags to enable features such as AVX2 or AVX512 instructions, while using runtime dispatch, which makes sure only valid instructions will be used on any given platform.

vec.h provides a generic implementation of vec type that allows the programmer to write code packing various primitives (such as floats) within 256bit & 512bits registers. vec defines various operators such as + and * and provides functions to allow operations such as max, min, etc.

As an example `ReduceOpsKernel.cpp` implements a generic `kernel_` that reduces an entire array using a given associative binary operation such as +.

More explicity, calling `kernel_` with template argument `std::plus` will cause it to sum up the entire array into a single value.

`ReduceOpsKernel.cpp` uses the `CPU_CAPABILITY_*` macros to "know" under which compiler flags it is currently compiled. This allows the programmer to write generic code, which will be compiled under multipled compilation settings.

`../ReduceOps.cpp` now includes the header `ReduceOpsKernel.h`, which contains a generic definition of `sumImplAll`. This function allows the user to reduce over a dimension or all dimensions. The appropiate capability is chosen at runtime using cpuinfo. If the current platform has AVX2, `sumImpl` will be set to `sumImplAll<CPUCapability::AVX2>`.

At runtime, the following environment variables control which codepath is taken:

x64 options: ATEN_CPU_CAPABILITY=avx2 # Force AVX2 codepaths to be used ATEN_CPU_CAPABILITY=avx # Force AVX codepaths to be used ATEN_CPU_CAPABILITY=default # Use oldest supported vector instruction set