This example using existing Linear Interpolation (aka lerp) operator, but same guidelines apply for other operators (new and existing ones).

As all changes going to impact performance significantly, we can use the simplest benchmark to measure operator speed before updates and establish the baseline.

```
y = torch.randn(1000000)
x = torch.randn(1000000)

timeit torch.lerp(x, y, 0.5)
2.4 ms ± 25.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Simple implementation

As the first step, we are naively introducing `TensorIterator` to replace `CPU_tensor_apply3` (it is considered the lousy pattern to use cpu/Loops.h anywhere but in cpu/ subfolder, but we will return to it later in this doc).

[[images/tensor_iterator/change0.png]]

Update: `binary_kernel` recently was replaced by more universal `cpu_kernel`, they have exact same API.

code: https://github.com/pytorch/pytorch/pull/21025/commits/c5593192e1f21dd5eb1062dbacfdf7431ab1d47f

In compare to TH_APPLY_* and CPU_tensor_apply* and solutions, TensorIterator usage is separated by two steps.

1. Defining iterator configuration with the TensorIterator::Builder. Under the hood, builder calculates tensors shapes and types to find the most performant way to traverse them (https://github.com/pytorch/pytorch/blob/dee11a92c1f1c423020b965837432924289e0417/aten/src/ATen/native/TensorIterator.h#L285)
2. Loop implementation. There are multiple different kernels in Loops.h depending on the number of inputs, they dispatch automatically by `cpu_kernel`, the ability to do parallel calculations, vectorized version availability (cpu_kernel_vec), type of operation (vectorized_inner_reduction). In our case, we have one output and two inputs, in this type of scenario we can use cpu_kernel. TensorIterator automatically picks the best way to traverse tensors (such as taking into account contiguous layout) as well as using parallelization for bigger tensors.

As a result, we have a 24x performance gain.

```
In [*2*]: x = torch.randn(1000000)
In [*3*]: y = torch.randn(1000000)

In [*4*]: timeit torch.lerp(x,y,0.5)
106 µs ± 3.4 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Most of the gain comes from the parallelization, so it is worth checking with OMP_NUM_THREADS=1.

Here we can see 2x improvement in comparison to the CPU_tensor_apply3 version of the code.

```
In [*2*]: x = torch.randn(1000000)
In [*3*]: y = torch.randn(1000000)

In [*4*]: timeit torch.lerp(x,y,0.5)
1.15 ms ± 65 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Using native/cpu

However, as noted above, it is sub-optimal (and NOT recommended solution) as we are not using vectorization and various compile-time optimizations. We can fix it by moving code into the native/cpu folder.

// Content of the aten/src/ATen/native/cpu/LerpKernel.cpp
(https://github.com/pytorch/pytorch/pull/21025/commits/bcf9b1f30331fc897fcc94661e84d47e91bf7290#diff-8d5b3146a020ca873b987cb2aced7fa0)

```
#include <ATen/ATen.h>
#include <ATen/Dispatch.h>
```

```
#include <ATen/native/Lerp.h>
#include <ATen/native/TensorIterator.h>
#include <ATen/native/cpu/Loops.h>

namespace at {
namespace native {
namespace {

static void lerp_kernel(
    Tensor& ret,
    const Tensor& self,
    const Tensor& end,
    Scalar weight) {
  auto builder = at::TensorIterator::Builder();
  builder.add_output(ret);
  builder.add_input(self);
  builder.add_input(end);
  auto iter = builder.build();

   AT_DISPATCH_FLOATING_TYPES(ret.scalar_type(), "lerp_kernel", [&] {
    scalar_t weight_val = weight.to<scalar_t>();
    at::native::binary_kernel(*iter, [=](scalar_t self_val, scalar_t end_val) {
      return (weight_val < 0.5)
          ? self_val + weight_val * (end_val - self_val)
          : end_val - (end_val - self_val) * (1 - weight_val);
    });
  });
}

} // anonymous namespace

REGISTER_DISPATCH(lerp_stub, &lerp_kernel);

} // namespace native
} // namespace at
```

code: https://github.com/pytorch/pytorch/pull/21025/commits/bcf9b1f30331fc897fcc94661e84d47e91bf7290

This code using the same type dispatch pattern `AT_DISPATCH_FLOATING_TYPES`, but moving everything into the kernel code for better optimization.

[[images/tensor_iterator/change1.png]]

Such code organization gives us 2x performance boost in comparison to the previous naive implementation. This optimization is archived by compiling AVX2 kernel versions (only applies to native/cpu folder) and dispatching supported code based on cpuinfo.

Multithreaded

```
In [*2*]: x = torch.randn(1000000)
In [*3*]: y = torch.randn(1000000)

In [*4*]: timeit torch.lerp(x,y,0.5)
51 µs ± 934 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Single thread

```
In [*2*]: x = torch.randn(1000000)
In [*3*]: y = torch.randn(1000000)

In [*4*]: timeit torch.lerp(x,y,0.5)
440 µs ± 2.13 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Vectorization with the cpu_kernel_vec

In many cases, we can also benefit from the explicit vectorization (provided by Vec256 library). TensorIterator provides the easy way to do it by using _vec loops.

[[images/tensor_iterator/change2.png]]

code: https://github.com/pytorch/pytorch/pull/21025/commits/83a23e745e839e8db81cf58ee00a5755d7332a43

We are doing so by replacing the cpu_kernel with the cpu_kernel_vec. At this particular case, weight_val check was omitted (to simplify example code), and performance benchmark show no significant gain.