The quantized folder holds the implementation of the low-level quantized kernel. The kernels are registered in `torch::_ops` namespace, and operate on the quantized `at::Tensor` data type. You can learn more about the quantized tensors in the quantized tensor API wiki page.

This document serves as an entry point for quantized kernel implementation.

Implementing native quantized ops

The new quantized ops are almost always located under the `ATen/native/quantized/cpu` folder. For the sake of an example, let us implement an element-wise quantized logical XAND operation under `ATen/native/quantized/cpu/qxand.cpp`.

Step 0. Implement the quantized function

Before writing the quantized kernel and registering it, let us implement a quantized function. That would assist in any further discussion. The snippet below shows the implementation of a quantized XAND operator, with the support of all implemented quantized types.

```
Tensor quantized_xand(Tensor qa, Tensor qb) {
  // Some type checks for qa and qb should be here...
  Tensor qc;
  double scale = qa.q_scale();
  int64_t zero_point = qa.q_zero_point();

  auto iter = TensorIterator::binary_op(qc, qa, qb);

  AT_DISPATCH_QINT_TYPES(qa.scalar_type(), "quantized_xand", [&]() {
    Tensor qc = at::_empty_affine_quantized(
        qa.sizes(), at::device(kCPU).dtype(SCALAR_TYPE), scale, zero_point);
    cpu_kernel(iter, [&](scalar_t a_value, scalar_t b_value) -> scalar_t {
      return scalar_t(a_value.val_ & b_value.val_);
    });
  });
  return qc;
}
```

The code above is fairly straight-forward: It takes two quantized tensors `qa` and `qb`, and uses `binary_kernel` to produce a quantized tensor `qc`. We also use the `TensorIterator` in this example. The only part that that requires explicit explanation is the `AT_DISPATCH_QINT_TYPES`. This macro makes sure that the underlying code works with all quantized types. It provides several useful "aliases":

- `SCALAR_TYPE` – `ScalarType` of the quantized tensor (e.g. `kQInt8`)
- `scalar_t` – quantized data type (dtype, e.g. `qint8`)
- `underlying_t` – underlying POD data type (dtype, e.g. `int8_t`)

The macro takes three arguments:

1. Quantized data type. This will define what the "aliases" are. In the example above, the resulting tensor will be the same as the `qa.scalar_type()`.
2. Function name. This argument is currently used for error reporting.
3. Implementation lambda. The main implementation should sit in the body of this lambda. it should also use the aliases for the quantized data types instead of the explicit data types.

Step 1. Define the schema

Update `aten/src/ATen/native/quantized/library.cpp` and add a `def` for your new operator:

```
TORCH_LIBRARY(quantized, m) {
  // ... the existing definitions ...
  m.def("quantized::xand(Tensor qa, Tensor qb) -> Tensor");
}
```

Def takes a **function schema string**: This schema describes the usage of the op. In the example above the schema is `"quantized::xand(Tensor qa, Tensor qb) -> Tensor"`. This translates to `torch._ops.ops.quantized.xand` function in Python of the appropriate signature.

Step 2. Register the implementation

The registration is done using `TORCH_LIBRARY_IMPL`.

```
TORCH_LIBRARY_IMPL(quantized, QuantizedCPU, m) {
  m.impl("xand", TORCH_FN(quantized_xand));
}
```

Step 2b. [Optional] Registering the operation with the `native_functions.yaml`

In some cases, if the signature of the quantized function and its non-quantized counterpart are the same, it is worth adding it to the `ATen/native/native_functions.yaml`. A detailed explanation on this file can be found here.

If adding a new entry to the `native_functions.yaml`:

```
- func: quantized_xand(Tensor qa, Tensor qb) -> Tensor
  dispatch:
    QuantizedCPU: quantized_xand
```

If adding to an existing entry in the `native_functions.yaml`:

If you find an entry in the yaml file, and would like to add a quantized kernel to it, you can just add a new dispatch entry for it. For example, let's assume there existed a `xand` function in the YAML file. In that case, modification would look as:

```yaml
- func: xand(Tensor a, Tensor b) -> Tensor
  dispatch:
    CPU: _xand_cpu     # Assume this existed
    CUDA: _xand_cuda   # Assume this existed
    QuantizedCPU: quantized_xand
```

Putting it all together

The final file `ATen/native/quantized/cpu/qxand.cpp` would look as follows

```cpp
#include <ATen/ATen.h>
#include <ATen/NativeFunctions.h> // Need that for the `native_functions.yaml`
#include <ATen/core/Type.h>
#include <torch/library.h>
#include <ATen/native/TensorIterator.h>
#include <ATen/native/cpu/Loops.h>

namespace at {
  namespace native {
  Tensor quantized_xand(Tensor qa, Tensor qb) {
    // The awesome op implementation...
    return qc;
  }

  TORCH_LIBRARY_IMPL(quantized, QuantizedCPU, m) {
    m.impl("xand", TORCH_FN(quantized_xand));
  }
}}  // namespace at::native
```

Step 3. Administrative stuff

Before the op can be used, it needs to be compiled. If the op is placed under `native/quantized/cpu`, this already done for you. However, if the location is changed, two files must be notified:

- *`caffe2/aten/TARGETS`* – You can follow the same example, and add your path in somewhere in that file. Notice in this file we places the path to the quantized source files:

```
ATEN_NATIVE_CPP = glob([
#...
  "src/ATen/native/quantized/**/*.cpp",
])
```

- *`caffe2/aten/src/ATen/CMakeLists.txt`* – Again, following the example, you must add your paths. The current quantization paths are added as

```
FILE(GLOB native_quantized_cpp
          "native/quantized/*.cpp"
```

```
        "native/quantized/cpu/*.cpp")
```

Using quantized ops

Python

Usage in Python is pretty easy. To implement the python quantized function using our kernel, you can do the following

```python
from torch._ops import ops

def quantized_xand(qa, qb):
#Notice the schema changed from `quantized::xand` to `quantized.xand`
  return ops.quantized.xand(qa, qb)
```

Note: If writing new pytorch functions that use quantized kernels, it is strongly encouraged to place them in the `torch/nn/quantized/functional.py`.

C++

You should not need to use the registered kernels in C++. Although **officially not supported**, you can use the following

```cpp
  Tensor quantized_xand(Tensor qa, Tensor qb) {
    static const c10::OperatorHandle op = c10::Dispatcher::singleton().findSchema({"quantize
    return op.call<Tensor, Tensor, Tensor>(qa, qb);
  }
```