Getting started

This C++ API example demonstrates the basics of the oneDNN programming model.

Example code: getting_started.cpp

This C++ API example demonstrates the basics of the oneDNN programming model:

- How to create oneDNN memory objects.
    - How to get data from the user's buffer into a oneDNN memory object.
    - How a tensor's logical dimensions and memory object formats relate.
- How to create oneDNN primitives.
- How to execute the primitives.

The example uses the ReLU operation and comprises the following steps:

1. Creating Engine and stream to execute a primitive.
2. Performing Data preparation (code outside of oneDNN).
3. Wrapping data into a oneDNN memory object (using different flavors).
4. Creating a ReLU primitive.
5. Executing the ReLU primitive.
6. Obtaining the result and validation (checking that the resulting image does not contain negative values).

These steps are implemented in the getting_started_tutorial() function, which in turn is called from main() function (which is also responsible for error handling).

Public headers

To start using oneDNN we must first include the `dnnl.hpp` header file in the program. We also include `dnnl_debug.h` in example_utils.hpp, which contains some debugging facilities like returning a string representation for common oneDNN C types.

getting_started_tutorial() function

Engine and stream

All oneDNN primitives and memory objects are attached to a particular dnnl::engine, which is an abstraction of a computational device (see also Basic Concepts). The primitives are created and optimized for the device they are attached to and the memory objects refer to memory residing on the corresponding device. In particular, that means neither memory objects nor primitives that were created for one engine can be used on another.

To create an engine, we should specify the dnnl::engine::kind and the index of the device of the given kind.

```
engine eng(engine_kind, 0);
```

In addition to an engine, all primitives require a dnnl::stream for the execution. The stream encapsulates an execution context and is tied to a particular engine.

The creation is pretty straightforward:

```
stream engine_stream(eng);
```

In the simple cases, when a program works with one device only (e.g. only on CPU), an engine and a stream can be created once and used throughout the program. Some frameworks create singleton objects that hold oneDNN engine and stream and use them throughout the code.

Data preparation (code outside of oneDNN)

Now that the preparation work is done, let's create some data to work with. We will create a 4D tensor in NHWC format, which is quite popular in many frameworks.

Note that even though we work with one image only, the image tensor is still 4D. The extra dimension (here N) corresponds to the batch, and, in case of a single image, is equal to 1. It is pretty typical to have the batch dimension even when working with a single image.

In oneDNN, all CNN primitives assume that tensors have the batch dimension, which is always the first logical dimension (see also Naming Conventions).

```
const int N = 1, H = 13, W = 13, C = 3;

// Compute physical strides for each dimension
const int stride_N = H * W * C;
const int stride_H = W * C;
const int stride_W = C;
const int stride_C = 1;

// An auxiliary function that maps logical index to the physical offset
auto offset = [=](int n, int h, int w, int c) {
    return n * stride_N + h * stride_H + w * stride_W + c * stride_C;
};

// The image size
const int image_size = N * H * W * C;

// Allocate a buffer for the image
std::vector<float> image(image_size);

// Initialize the image with some values
for (int n = 0; n < N; ++n)
    for (int h = 0; h < H; ++h)
        for (int w = 0; w < W; ++w)
            for (int c = 0; c < C; ++c) {
                int off = offset(
                        n, h, w, c); // Get the physical offset of a pixel
                image[off] = -std::cos(off / 10.f);
            }
```

Wrapping data into a oneDNN memory object

Now, having the image ready, let's wrap it in a dnnl::memory object to be able to pass the data to oneDNN primitives.

Creating dnnl::memory comprises two steps:

1. Initializing the dnnl::memory::desc struct (also referred to as a memory descriptor), which only describes the tensor data and doesn't contain the data itself. Memory descriptors are used to create dnnl::memory objects and to initialize primitive descriptors (shown later in the example).

2. Creating the dnnl::memory object itself (also referred to as a memory object), based on the memory descriptor initialized in step 1, an engine, and, optionally, a handle to data. The memory object is used when a primitive is executed.

Thanks to the list initialization introduced in C++11, it is possible to combine these two steps whenever a memory descriptor is not used anywhere else but in creating a dnnl::memory object.

However, for the sake of demonstration, we will show both steps explicitly.

Memory descriptor

To initialize the dnnl::memory::desc, we need to pass:

1. The tensor's dimensions, the semantic order of which is defined by the primitive that will use this memory (descriptor).

   > ⚠ **Warning**
   >
   > Memory descriptors and objects are not aware of any meaning of the data they describe or contain.

2. The data type for the tensor (dnnl::memory::data_type).

3. The memory format tag (dnnl::memory::format_tag) that describes how the data is going to be laid out in the device's memory. The memory format is required for the primitive to correctly handle the data.

The code:

```
auto src_md = memory::desc(
        {N, C, H, W}, // logical dims, the order is defined by a primitive
        memory::data_type::f32, // tensor's data type
        memory::format_tag::nhwc // memory format, NHWC in this case
);
```

The first thing to notice here is that we pass dimensions as `{N, C, H, W}` while it might seem more natural to pass `{N, H, W, C}`, which better corresponds to the user's code. This is because oneDNN CNN primitives like ReLU always expect tensors in the following form:

Spatial dim	Tensor dimensions
0D	$N \times C$
1D	$N \times C \times W$
2D	$N \times C \times H \times W$
3D	$N \times C \times D \times H \times W$

where:

- N is a batch dimension (discussed above),
- C is channel (aka feature maps) dimension, and
- D, H, and W are spatial dimensions.

Now that the logical order of dimension is defined, we need to specify the memory format (the third parameter), which describes how logical indices map to the offset in memory. This is the place where the user's format NHWC comes into play. oneDNN has different dnnl::memory::format_tag values that cover the most popular memory formats like NCHW, NHWC, CHWN, and some others.

The memory descriptor for the image is called `src_md`. The `src` part comes from the fact that the image will be a source for the ReLU primitive (that is, we formulate memory names from the primitive perspective; hence we will use `dst` to name the output memory). The `md` is an initialism for Memory Descriptor.

Alternative way to create a memory descriptor

Before we continue with memory creation, let us show the alternative way to create the same memory descriptor: instead of using the dnnl::memory::format_tag, we can directly specify the strides of each tensor dimension:

```
auto alt_src_md = memory::desc(
        {N, C, H, W}, // logical dims, the order is defined by a primitive
        memory::data_type::f32, // tensor's data type
        {stride_N, stride_C, stride_H, stride_W} // the strides
);

// Sanity check: the memory descriptors should be the same
if (src_md != alt_src_md)
    throw std::logic_error("Memory descriptor initialization mismatch.");
```

Just as before, the tensor's dimensions come in the `N, C, H, W` order as required by CNN primitives. To define the physical memory format, the strides are passed as the third parameter. Note that the order of the strides corresponds to the order of the tensor's dimensions.

Warning

Using the wrong order might lead to incorrect results or even a crash.

Creating a memory object

Having a memory descriptor and an engine prepared, let's create input and output memory objects for a ReLU primitive.

```
// src_mem contains a copy of image after write_to_dnnl_memory function
auto src_mem = memory(src_md, eng);
write_to_dnnl_memory(image.data(), src_mem);

// For dst_mem the library allocates buffer
auto dst_mem = memory(src_md, eng);
```

We already have a memory buffer for the source memory object. We pass it to the dnnl::memory::memory(const dnnl::memory::desc &, const dnnl::engine &, void *) constructor that takes a buffer pointer as its last argument.

Let's use a constructor that instructs the library to allocate a memory buffer for the `dst_mem` for educational purposes.

The key difference between these two are:

1. The library will own the memory for `dst_mem` and will deallocate it when `dst_mem` is destroyed. That means the memory buffer can be used only while `dst_mem` is alive.

2. Library-allocated buffers have good alignment, which typically results in better performance.

Note

Memory allocated outside of the library and passed to oneDNN should have good alignment for better performance.

In the subsequent section we will show how to get the buffer (pointer) from the `dst_mem` memory object.

Creating a ReLU primitive

Let's now create a ReLU primitive.

The library implements ReLU primitive as a particular algorithm of a more general Eltwise primitive, which applies a specified function to each and every element of the source tensor.

Just as in the case of dnnl::memory, a user should always go through (at least) two creation steps (which however, can be sometimes combined thanks to C++11):

1. Create an operation primitive descriptor (here dnnl::eltwise_forward::primitive_desc) that defines operation parameters and is a lightweight descriptor of the actual algorithm that implements the given operation. The user can query different characteristics of the chosen implementation such as memory consumptions and some others that will be covered in the next topic (Memory Format Propagation).

2. Create a primitive (here dnnl::eltwise_forward) that can be executed on memory objects to compute the operation.

oneDNN separates steps 2 and 3 to enable the user to inspect details of a primitive implementation prior to creating the primitive. This may be expensive, because, for example, oneDNN generates the optimized computational code on the fly.

> **Note**
>
> Primitive creation might be a very expensive operation, so consider creating primitive objects once and executing them multiple times.

The code:

```
// ReLU primitive descriptor, which corresponds to a particular
// implementation in the library
auto relu_pd = eltwise_forward::primitive_desc(
        eng, // an engine the primitive will be created for
        prop_kind::forward_inference, algorithm::eltwise_relu,
        src_md, // source memory descriptor for an operation to work on
        src_md, // destination memory descriptor for an operation to work on
        0.f, // alpha parameter means negative slope in case of ReLU
        0.f // beta parameter is ignored in case of ReLU
);

// ReLU primitive
auto relu = eltwise_forward(relu_pd); // !!! this can take quite some time
```

A note about variable names. Similar to the `_md` suffix used for memory descriptor, we use `_d` for the operation descriptor names, `_pd` for the primitive descriptors, and no suffix for primitives themselves.

It is worth mentioning that we specified the exact tensor and its memory format when we were initializing the `relu_d`. That means `relu` primitive would perform computations with memory objects that correspond to this description. This is the one and only one way of creating non-compute-intensive primitives like Eltwise, Batch Normalization, and others.

Compute-intensive primitives (like Convolution) have an ability to define the appropriate memory format on their own. This is one of the key features of the library and will be discussed in detail in the next topic: Memory Format Propagation.

Executing the ReLU primitive

Finally, let's execute the primitive and wait for its completion.

The input and output memory objects are passed to the `execute()` method using a <tag, memory> map. Each tag specifies what kind of tensor each memory object represents. All Eltwise primitives require the map to have two elements: a source memory object (input) and a destination memory (output).

A primitive is executed in a stream (the first parameter of the `execute()` method). Depending on a stream kind, an execution might be blocking or non-blocking. This means that we need to call dnnl::stream::wait before accessing the results.

```
// Execute ReLU (out-of-place)
relu.execute(engine_stream, // The execution stream
        {
                // A map with all inputs and outputs
                {DNNL_ARG_SRC, src_mem}, // Source tag and memory obj
                {DNNL_ARG_DST, dst_mem}, // Destination tag and memory obj
        });

// Wait the stream to complete the execution
engine_stream.wait();
```

The Eltwise is one of the primitives that support in-place operations, meaning that the source and destination memory can be the same. To perform in-place transformation, the user must pass the same memory object for both the `DNNL_ARG_SRC` and `DNNL_ARG_DST` tags:

```
// Execute ReLU (in-place)
// relu.execute(engine_stream,  {
//          {DNNL_ARG_SRC, src_mem},
//          {DNNL_ARG_DST, src_mem},
//         });
```

Obtaining the result and validation

Now that we have the computed result, let's validate that it is actually correct. The result is stored in the `dst_mem` memory object. So we need to obtain the C++ pointer to a buffer with data via dnnl::memory::get_data_handle() and cast it to the proper data type as shown below.

Warning

The dnnl::memory::get_data_handle() returns a raw handle to the buffer, the type of which is engine specific. For the CPU engine the buffer is always a pointer to `void`, which can safely be used. However, for engines other than CPU the handle might be runtime-specific type, such as `cl_mem` in case of GPU/OpenCL.

```
// Obtain a buffer for the `dst_mem` and cast it to `float *`.
// This is safe since we created `dst_mem` as f32 tensor with known
// memory format.
std::vector<float> relu_image(image_size);
read_from_dnnl_memory(relu_image.data(), dst_mem);
/*
// Check the results
for (int n = 0; n < N; ++n)
    for (int h = 0; h < H; ++h)
        for (int w = 0; w < W; ++w)
            for (int c = 0; c < C; ++c) {
                int off = offset(
                        n, h, w, c); // get the physical offset of a pixel
                float expected = image[off] < 0
                        ? 0.f
                        : image[off]; // expected value
                if (relu_image[off] != expected) {
                    std::cout << "At index(" << n << ", " << c << ", " << h
                              << ", " << w << ") expect " << expected
                              << " but got " << relu_image[off]
                              << std::endl;
                    throw std::logic_error("Accuracy check failed.");
                }
            }
```

main() function

We now just call everything we prepared earlier.

Because we are using the oneDNN C++ API, we use exceptions to handle errors (see API). The oneDNN C++ API throws exceptions of type dnnl::error, which contains the error status (of type dnnl_status_t) and a human-readable error message accessible through regular `what()` method.

```
int main(int argc, char **argv) {
    int exit_code = 0;

    engine::kind engine_kind = parse_engine_kind(argc, argv);
    try {
        getting_started_tutorial(engine_kind);
    } catch (dnnl::error &e) {
        std::cout << "oneDNN error caught: " << std::endl
                  << "\tStatus: " << dnnl_status2str(e.status) << std::endl
                  << "\tMessage: " << e.what() << std::endl;
        exit_code = 1;
    } catch (std::string &e) {
        std::cout << "Error in the example: " << e << "." << std::endl;
        exit_code = 2;
    }

    std::cout << "Example " << (exit_code ? "failed" : "passed") << " on "
              << engine_kind2str_upper(engine_kind) << "." << std::endl;
    finalize();
    return exit_code;
}
```

Upon compiling and run the example the output should be just:

```
Example passed.
```

Users are encouraged to experiment with the code to familiarize themselves with the concepts. In particular, one of the changes that might be of interest is to spoil some of the library calls to check how error handling happens. For instance, if we replace

```
relu.execute(engine_stream, {
        {DNNL_ARG_SRC, src_mem},
        {DNNL_ARG_DST, dst_mem},
    });
```

with

```
relu.execute(engine_stream, {
        {DNNL_ARG_SRC, src_mem},
        // {DNNL_ARG_DST, dst_mem}, // Oops, forgot about this one
    });
```

we should get the following output:

```
oneDNN error caught:
        Status: invalid_arguments
        Message: could not execute a primitive
Example failed.
```