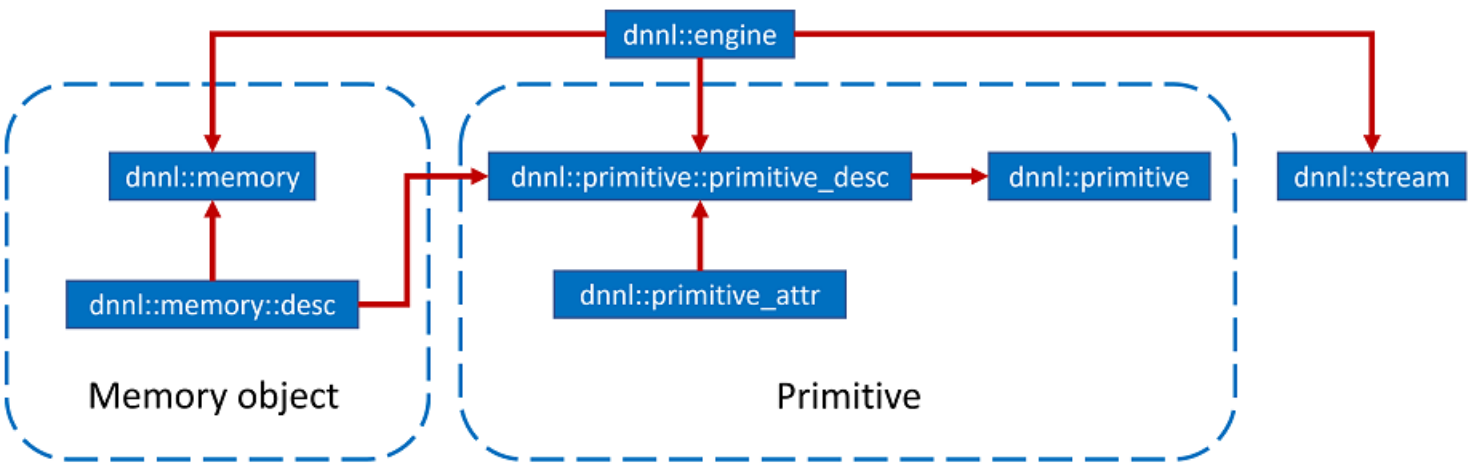
Basic Concepts

Introduction

In this page, an outline of the oneDNN programming model is presented, and the key concepts are discussed, including Primitives, Engines, Streams, and Memory Objects. In essence, the oneDNN programming model consists in executing one or several primitives to process data in one or several memory objects. The execution is performed on an engine in the context of a stream. The relationship between these entities is briefly presented in Figure 1, which also includes additional concepts relevant to the oneDNN programming model, such as primitive attributes and descriptors. These concepts are described below in much more details.



Primitives

oneDNN is built around the notion of a primitive (dnnl::primitive). A primitive is an object that encapsulates a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation. Additionally, using primitive attributes (dnnl::primitive_attr) certain primitives can represent more complex fused computations such as a forward convolution followed by a ReLU.

The most important difference between a primitive and a pure function is that a primitive can store state.

One part of the primitive's state is immutable. For example, convolution primitives store parameters like tensor shapes and can pre-compute other dependent parameters like cache blocking. This approach allows oneDNN primitives to pre-generate code specifically tailored for the operation to be performed. The oneDNN programming model assumes that the time it takes to perform the pre-computations is amortized by reusing the same primitive to perform computations multiple times.

The mutable part of the primitive's state is referred to as a scratchpad. It is a memory buffer that a primitive may use for temporary storage only during computations. The scratchpad can either be owned by a primitive object (which makes that object non-thread safe) or be an execution-time parameter.

Engines

Engines (dnnl::engine) is an abstraction of a computational device: a CPU, a specific GPU card in the system, etc. Most primitives are created to execute computations on one specific engine. The only exceptions are reorder primitives that transfer data between two different engines.

Streams

Streams (dnnl::stream) encapsulate execution context tied to a particular engine. For example, they can correspond to OpenCL command queues.

Memory Objects

Memory objects (dnnl::memory) encapsulate handles to memory allocated on a specific engine, tensor dimensions, data type, and memory format – the way tensor indices map to offsets in linear memory space. Memory objects are passed to primitives during execution.

Levels of Abstraction

Conceptually, oneDNN has multiple levels of abstractions for primitives and memory objects in order to expose maximum flexibility to its users.

- Memory descriptors (dnnl_memory_desc_t, dnnl::memory::desc) define a tensor's logical dimensions, data type, and the format in which the data is laid out in memory. The special format any (dnnl::memory::format_tag::any) indicates that the actual format will be defined later (see Memory Format Propagation).

- Primitives descriptors fully define an operations's computation using the memory descriptors (dnnl_memory_desc_t, dnnl::memory::desc) passed at construction, as well as the attributes. They also dispatch specific implementation based on the engine. Primitive descriptors can be used to query various primitive implementation details and, for example, to implement Memory Format Propagation by inspecting expected memory formats via queries without having to fully instantiate a primitive. oneDNN may contain multiple implementations for the same primitive that can be used to perform the same particular computation. Primitive descriptors allow one-way iteration which allows inspecting multiple implementations. The library is expected to order the implementations from the most to least preferred, so it should always be safe to use the one that is chosen by default.

- Primitives, which are the most concrete, and embody the actual executable code that will be run to perform the primitive computation.

Creating Memory Objects and Primitives

Memory Objects

Memory objects are created from the memory descriptors. It is not possible to create a memory object from a memory descriptor that has memory format set to dnnl::memory::format_tag::any.

There are two common ways for initializing memory descriptors:

- By using dnnl::memory::desc constructors or by extracting a descriptor for a part of a tensor via dnnl::memory::desc::submemory_desc
- By querying an existing primitive descriptor for a memory descriptor corresponding to one of the primitive's parameters (for example, dnnl::convolution_forward::primitive_desc::src_desc).

Memory objects can be created with a user-provided handle (a `void *` on CPU), or without one, in which case the library will allocate storage space on its own.

Primitives

The sequence of actions to create a primitive is:

1. Create a primitive descriptor via, for example, dnnl::convolution_forward::primitive_desc. The primitive descriptor can contain memory descriptors with placeholder format_tag::any memory formats if the primitive supports it.
2. Create a primitive based on the primitive descriptor obtained in step 1.

Graph Extension

Graph extension is a high level abstraction in oneDNN that allows you to work with a computation graph instead of individual primitives. This approach allows you to make an operation fusion:

- Transparent: the integration efforts are reduced by abstracting backend-aware fusion logic.
- Scalable: no integration code change is necessary to benefit from new fusion patterns enabled in oneDNN.

The programming model for the graph extension is detailed in the graph basic concepts section.
