Programming Model

Programming Model#

oneDNN Graph programming model allows users to pass a computation graph and get partitions. Users then compile partitions, bind tensor data, and execute compiled partitions. Partitions are decided by oneDNN Graph implementation, which is the key concept to satisfy the different needs of AI hardware classes using a unified API.

The programming model assumes that the main usage is to support deep learning (DL) frameworks or inference engines. DL frameworks have their own representation for the computation graph. oneDNN Graph API is used to offload or accelerate graph partitions from a framework graph. In the description below, “graph” refers to the graph built by oneDNN Graph implementation, and “framework graph” refers to the graph built by the DL framework.

A deep learning computation graph consists of deep neural network (DNN) operations. A DNN operation is a function that takes input data and returns output data. The input and output data are multidimensional arrays called tensors. A DNN operation may consume multiple tensors and produce multiple tensors. A tensor must be produced by a single operation and may be consumed by multiple operations.

oneDNN Graph API uses logical tensor, OP, and graph to represent a computation graph. Logical tensor represents tensor’s metadata, like element data type, shape, and layout. OP represents an operation on a computation graph. OP has kind, attribute, and input and output logical tensors. OPs are added to a graph. Both OP and logical tensor contains a unique ID, so that the graph knows how to connect a producer OP to a consumer OP through a logical tensor. The graph constructed is immutable. The purpose of creating the graph object is to get partitions. After partitions are created, the graph object is not useful anymore. Once users get partitions, users should not add OP to the graph.

oneDNN Graph defines operation set. Users should convert their DNN operation definition to oneDNN Graph operation for graph construction. For operation outside oneDNN Graph operation set, users may use wild-card OP. The wild-card OP represents any OP. With its input and output logical tensors, it enables the oneDNN Graph implementation to receive a full graph and conduct a complete analysis. User needs to use a special “End” op to indicate output tensors of the graph. For any tensors needs to be alive after a graph being executed, it needs to be connected to a “End” op which consumes the tensor. Users may have multiple “End” ops for one graph. For each OP users add to the graph, users must describe its input and output logical tensors. Users must describe data type for each logical tensor. If tensor’s shape and layout are known, users must describe them along with the logical tensor.

A partition is a connected subgraph in a graph. oneDNN Graph implementation analyzes a graph and returns a number of partitions. The returned partitions completely cover all the OPs of the graph and follow topological order. A partition typically contains multiple Ops. Sometimes a partition may contain just one OP, like a Wildcard OP or unsupported OP. A partition contains a flag to indicate whether the partition is supported and thus can be compiled and executed. User needs to check the flag before using the partition.

Partition’s input and output is also called as port. The ports record the logical tensor information which was passed during graph construction. With the logical tensor ID, users can track the producer and

consumer relationship between partitions. The ports also record the data type of corresponding logical tensors.

The returned partitions to users must not form a dependence cycle. For example, a graph contains 3 OPs: A, B, and C. If C consumes A's output and produces B's input, oneDNN Graph implementation must not put A and B into one partition. However, if C is not added to the graph, the returned partition may include A and B, since C is not visible to oneDNN Graph implementation. In this case, it is the user's responsibility to detect the dependence cycle. Once users pass a complete graph, users don't need to check the dependence cycle among the partitions returned by oneDNN Graph.

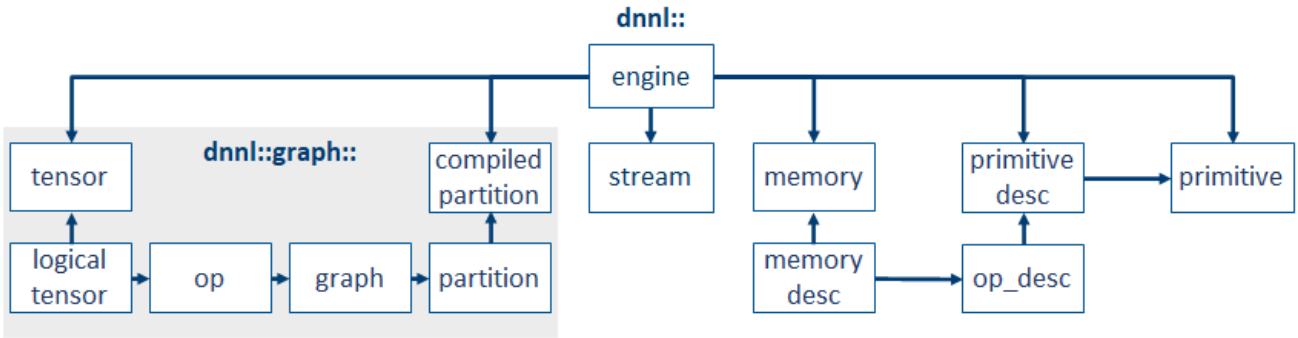
A partition needs to be compiled before execution. The compilation lowers down the compute logic to hardware ISA level and generates binary code. The generated code is specialized for the input and output tensor's metadata. Users must create new logical tensors to pass complete metadata with the compilation API. The logical tensors should fully specify id, data type, shape (can be incomplete for outputs), and layout, the compilation should succeed. The logical tensors passed during compilation time must match IDs with partition's ports. The logical tensors must have same data types with the ports with the port of the same ID.

For the output logical tensors, users must either specify a public layout using size and stride for each tensor dimension or request oneDNN Graph implementation to decide a target-specific layout. For the input logical tensors, users must either specify a public layout or using a target-specific layout produced by predecessor partition compilation. For the logical tensor with target-specific layout, it must be produced by a partition and used only by partitions.

A compiled partition represents the generated code specialized for target hardware and tensor metadata passed with compilation API. Users may cache the compiled partition to amortize the compilation cost among many iterations. If tensor metadata is identical, a compiled partition generated in previous iterations may be reused. Alternatively, implementations may reduce the partition compilation cost by caching the compiled partition internally. This optimization falls outside of the scope of this specification.

To execute a compiled partition, users must pass input and output tensors. Input tensors must bind input data buffers to logical tensors. Users may query the compiled partition for output data buffer sizes. If the sizes are known, users may allocate the output data buffers and bind to output tensors. If the sizes are unknown, users must provide an allocator for oneDNN Graph implementation to allocate the output tensor buffer. The execution API takes a compiled partition, input tensors, and return output tensors with the data buffer updated.

An engine represents a target device and context in the system. It needs to be passed as a parameter for partition compilation. A stream abstracts hardware execution resources of a target device. It is required to execute a compiled partition.



The diagram above summarizes the key programming concepts, and how they interact with each other. The arrow indicates the destination object contains or uses the source object. For example, OP contains logical tensor, and compiled partition uses partition.

Logical Tensor#

Logical tensor describes the metadata of the input or output tensor, like element data type, number of dimensions, size for each dimension, layout.

Besides helping oneDNN Graph implementation to build the graph, Logical tensor plays a critical role to exchange tensor metadata information between users and oneDNN Graph implementation. Users pass input tensor shape information and get the inferred shape for output tensors from a partition. Users pass logical tensors to compilation API for specifying shape and layout information. Users also use a special logical tensor to allow oneDNN Graph implementation to decide the layout for output tensors. After compilation, users can query the compiled partition for output tensors' shape, layout, and sizes.

Each logical tensor has an ID. The tensor metadata may include new shape information in the framework graph as it progresses toward execution. As a logical tensor is not mutable, users must create a new logical tensor with the same ID to pass any new additional information to oneDNN Graph implementation. Users should guarantee that the logical tensor ID is unique within the graph which the logical tensor belongs to.

```
/// Data Type
enum class data_type {
    undef = dnnl_data_type_undef,
    /// 16-bit/half-precision floating point.
    f16,
    /// non-standard 16-bit (bfloat16 w/ 7 bit mantissa) floating point.
    bf16,
    /// 32-bit/single-precision floating point.
    f32,
    /// 32-bit signed integer.
    s32,
    /// 8-bit signed integer.
    s8,
    /// 8-bit unsigned integer.
    u8,
};

/// Layout type
enum class layout_type {
    /// Undefined layout type.
    undef,
    /// Any means to let the library to decide the layout for a tensor
    /// during partition compilation.
    any,
    /// Strided means that the layout of a tensor is determined by the
    /// strides field in the logical tensor.
    strided,
    /// Opaque means that the layout of a tensor is the library specific.
    /// Usually, an opaque layout is generated by a partition which is
    /// compiled with layout type any.
    opaque,
};

/// Tensor property
enum class property_type {
    /// Undefined tensor property.
    undef,
    /// Variable means the tensor may be changed during computation or
```

```
    /// between different iterations.
    variable,
    /// Constant means the tensor will keep unchanged during computation and
    /// between different iterations. It's useful for the library to apply
    /// optimizations for constant tensors or cache constant tensors inside
    /// the library. For example, constant weight tensors in inference
    /// scenarios.
    constant,
};

/// Logical tensor object
class logical_tensor {
public:
    using dims_t = std::vector<dnnl_dim_t>;

    /// default constructor
    /// construct an empty object
    logical_tensor() = default;

    /// Copy
    logical_tensor(const logical_tensor &other) = default;

    /// Assign
    logical_tensor &operator=(const logical_tensor &other) = default;

    /// Constructs a logical tensor object with ID, data type, ndims, layout
    /// type, and property type.
    ///
    /// @param tid Logical tensor ID.
    /// @param dtype Elements data type.
    /// @param ndims Number of dimensions. -1 means unknown and 0 means a scalar
    ///     tensor.
    /// @param ltype Layout type.
    /// @param ptype Property type.
    logical_tensor(size_t tid, data_type dtype, int32_t ndims,
            layout_type ltype, property_type ptype = property_type::undef);

    /// Delegated constructor.
    ///
    /// @param tid Logical tensor ID.
    /// @param dtype Elements data type.
    /// @param ltype Layout type.
    logical_tensor(size_t tid, data_type dtype,
            layout_type ltype = layout_type::undef);

    /// Constructs a logical tensor object with basic information and detailed
    /// dims.
    ///
    /// @param tid Logical tensor ID.
    /// @param dtype Elements data type.
    /// @param adims Logical tensor dimensions. -1 means the size of that
    ///     dimension is unknown. 0 is used to define zero-dimension tensor.
    /// @param ltype Layout type. If it's strided, the strides field in the
    ///     output logical tensor will be deduced accordingly.
    /// @param ptype Property type.
    logical_tensor(size_t tid, data_type dtype, const dims_t &adims,
            layout_type ltype, property_type ptype = property_type::undef);

    /// Constructs a logical tensor object with detailed dims and strides. The
    /// layout_type of the output logical tensor object will always be strided.
    ///
    /// @param tid Logical tensor ID.
    /// @param dtype Elements data type.
    /// @param adims Logical tensor dimensions. -1 means the size of that
    ///     dimension is unknown. 0 is used to define zero-dimension tensor.
```

```
/// @param strides Logical tensor strides. -1 means the stride of the
///     dimension is unknown. The library currently doesn't support other
///     negative stride values.
/// @param ptype Property type.
logical_tensor(size_t tid, data_type dtype, const dims_t &adims,
        const dims_t &strides, property_type ptype = property_type::undef);

/// Constructs a logical tensor object with detailed dims and an opaque
/// layout ID. layout_type of the output logical tensor object will always
/// be opaque.
///
/// @param tid Logical tensor ID.
/// @param dtype Elements data type.
/// @param adims Logical tensor dimensions. -1 means the size of that
///     dimension is unknown. 0 is used to define zero-dimension tensor.
/// @param lid Opaque layout id.
/// @param ptype Property type
logical_tensor(size_t tid, data_type dtype, const dims_t &adims, size_t lid,
        property_type ptype = property_type::undef);

/// Returns dimensions of a logical tensor.
///
/// @returns A vector describing the size of each dimension.
dims_t get_dims() const;

/// Returns the unique id of a logical tensor.
///
/// @returns An integer value describing the ID.
size_t get_id() const;

/// Returns the data type of a logical tensor.
///
/// @returns The data type.
data_type get_data_type() const;

/// Returns the property type of a logical tensor.
///
/// @returns The property type.
property_type get_property_type() const;

/// Returns the layout type of a logical tensor.
///
/// @returns The layout type.
layout_type get_layout_type() const;

/// Returns the layout ID of a logical tensor. The API should be called on a
/// logical tensor with opaque layout type. Otherwise, an exception will be
/// raised.
///
/// @returns Layout ID.
size_t get_layout_id() const;

/// Returns the strides of a logical tensor. The API should be called on a
/// logical tensor with strided layout type. Otherwise, an exception will be
/// raised.
///
/// @returns A vector describing the stride size of each dimension.
dims_t get_strides() const;

/// Returns memory size in bytes required by this logical tensor.
///
/// @returns The memory size in bytes.
size_t get_mem_size() const;

/// Compares if two logical tenors have the same layout.
```

```
    ///
    /// @param lt The input logical tensor to be compared.
    /// @returns @c true if they have the same layout. @c false if they have
    bool has_same_layout(const logical_tensor &lt) const;
};
```

OP#

OP describes a deep neural network operation. OP contains kind, attribute, and input and output logical tensor.

Conv op contains format attributes for both activation and weight tensor, to indicate the semantics of each dimension of tensors. For example, the 2D conv may specify the dimension order is either `NHWC` or `NCHW`. oneDNN Graph uses one letter `X` to generalize all the spatial dimensions so `NXC` or `NCX` are used for the last example. Users should guarantee the OP ID is unique within the graph which the OP is added to.

```
/// Kinds of operations
enum class kind {
    Abs,
    AbsBackprop,
    Add,
    AvgPool,
    AvgPoolBackprop,
    BatchNormForwardTraining,
    BatchNormInference,
    BatchNormTrainingBackprop,
    BiasAdd,
    BiasAddBackprop,
    Clamp,
    ClampBackprop,
    Concat,
    Convolution,
    ConvolutionBackpropData,
    ConvolutionBackpropFilters,
    ConvTranspose,
    ConvTransposeBackpropData,
    ConvTransposeBackpropFilters,
    Dequantize,
    Divide,
    DynamicDequantize,
    DynamicQuantize,
    Elu,
    EluBackprop,
    End,
    Erf,
    Exp,
    GELU,
    GELUBackprop,
    HardSwish,
    HardSwishBackprop,
    Interpolate,
    InterpolateBackprop,
    LayerNorm,
    LayerNormBackprop,
    LeakyReLU,
    Log,
    LogSoftmax,
    LogSoftmaxBackprop,
    MatMul,
    Maximum,
    MaxPool,
```

```
    MaxPoolBackprop,
    Minimum,
    Mish,
    MishBackprop,
    Multiply,
    PReLU,
    PReLUBackprop,
    Quantize,
    Reciprocal,
    ReduceL1,
    ReduceL2,
    ReduceMax,
    ReduceMean,
    ReduceMin,
    ReduceProd,
    ReduceSum,
    ReLU,
    ReLUBackprop,
    Reorder,
    Round,
    Sigmoid,
    SigmoidBackprop,
    SoftMax,
    SoftMaxBackprop,
    SoftPlus,
    SoftPlusBackprop,
    Sqrt,
    SqrtBackprop,
    Square,
    SquaredDifference,
    StaticReshape,
    StaticTranspose,
    Subtract,
    Tanh,
    TanhBackprop,
    TypeCast,
    Wildcard,
    // Sentinel
    LastSymbol,
};

/// Attributes of operations. Different operations support different
/// attributes. Check the document of each operation for what attributes are
/// supported and what are the potential values for them. Missing required
/// attribute or illegal attribute value may lead to failure when adding the
/// operation to a graph.
enum class attr {
    /// Undefined op attribute.
    undef,

    // float32 attributes. The value of these attributes can be any single
    // float32 number.

    /// Specifies an alpha attribute to an op.
    alpha,
    /// Specifies an beta attribute to an op.
    beta,
    /// Specifies an epsilon attribute to an op.
    epsilon,
    /// Specifies a max attribute to an op.
    max,
    /// Specifies a min attribute to an op.
    min,
    /// Specifies a momentum attribute to an op.
    momentum,
```

```
// float32 vector attributes. The value of these attributes can be a
// vector of float32 numbers.

/// Specifies a scales attribute to an op.
scales,

// int64_t attributes. The value of these attributes can be any single
// int64 number.

/// Specifies an axis attribute to an op.
axis,
/// Specifies a begin_norm_axis attribute to an op.
begin_norm_axis,
/// Specifies a groups attribute to an op.
groups,

// int64_t vector attributes. The value of these attributes can be a
// vector of int64 numbers.

/// Specifies an axes attribute to an op.
axes,
/// Specifies a dilations attribute to an op.
dilations,
/// Specifies a filter_shape attribute to an op.
filter_shape,
/// Specifies an input_shape attribute to an op.
input_shape,
/// Specifies a kernel attribute to an op.
kernel,
/// Specifies an order attribute to an op.
order,
/// Specifies an output_padding attribute to an op.
output_padding,
/// Specifies an output_shape attribute to an op.
output_shape,
/// Specifies a pads_begin attribute to an op.
pads_begin,
/// Specifies a pads_end attribute to an op.
pads_end,
/// Specifies a shape attribute to an op.
shape,
/// Specifies a sizes attribute to an op.
sizes,
/// Specifies a strides attribute to an op.
strides,
/// Specifies a zps attribute to an op.
zps,

// bool attributes. The value of these attributes can be any single bool
// value.

/// Specifies an exclude_pad attribute to an op.
exclude_pad,
/// Specifies a keep_dims attribute to an op.
keep_dims,
/// Specifies a keep_stats attribute to an op.
keep_stats,
/// Specifies a per_channel_broadcast attribute to an op.
per_channel_broadcast,
/// Specifies a special_zero attribute to an op.
special_zero,
/// Specifies a transpose_a attribute to an op.
transpose_a,
/// Specifies a transpose_b attribute to an op.
```

```
    transpose_b,
    /// Specifies an use_affine attribute to an op.
    use_affine,
    /// Specifies an use_dst attribute to an op.
    use_dst,

    // string attributes. The value of these attributes can be a string.

    /// Specifies an auto_broadcast attribute to an op. The value can be
    /// "none" or "numpy".
    auto_broadcast,
    /// Specifies an auto_pad attribute to an op. The value can be "none",
    /// "same_upper", "same_lower", or "valid".
    auto_pad,
    /// Specifies an coordinate_transformation_mode attribute to an op. The
    /// value can be "half_pixel" or "align_corners". The attribute is
    /// defined for Interpolate operations.
    coordinate_transformation_mode
   ,
    /// Specifies a data_format of an op. The value can be "NCX" or "NXC".
    data_format,
    /// Specifies a filter_format of an op. The value can be "OIX" or "XIO".
    filter_format,
    /// Specifies a mode attribute of an op. The value can be "nearest",
    /// "linear", "bilinear", or "trilinear". The attribute is defined for
    /// Interpolate operations.
    mode,
    /// Specifies a qtype attribute to an op. The value can be "per_channel"
    /// or "per_tensor". The attribute is defined for quantization
    /// operations.
    qtype,
    /// Specifies a rounding_type attribute to an op. The value can be
    /// "ceil" or "floor".
    rounding_type,
};

/// An op object.
class op {
public:
    /// Constructs an op object with an unique ID, an operation kind, and a name
    /// string.
    ///
    /// @param id The unique ID of the op.
    /// @param akind The op kind specifies which computation is represented by
    ///     the op, such as Convolution or ReLU.
    /// @param verbose_name The string added as the op name.
    op(size_t id, kind akind, const std::string &verbose_name);

    /// Constructs an op object with an unique ID, an operation kind, and
    /// input/output logical tensors.
    ///
    /// @param id The unique ID of this op.
    /// @param akind The op kind specifies which computation is represented by
    ///     this op, such as Convolution or ReLU.
    /// @param inputs Input logical tensor to be bound to this op.
    /// @param outputs Output logical tensor to be bound to this op.
    /// @param verbose_name The string added as the op name.
    op(size_t id, kind akind, const std::vector<logical_tensor> &inputs,
            const std::vector<logical_tensor> &outputs,
            const std::string &verbose_name);

    /// Adds an input logical tensor to the op.
    ///
    /// @param t Input logical tensor.
    void add_input(const logical_tensor &t);
```

```
/// Adds a vector of input logical tensors to the op.
///
/// @param ts The list of input logical tensors.
void add_inputs(const std::vector<logical_tensor> &ts);

/// Adds an output logical tensor to the op.
///
/// @param t Output logical tensor.
void add_output(const logical_tensor &t);

/// Adds a vector of output logical tensors to the op.
///
/// @param ts The list of output logical tensors.
void add_outputs(const std::vector<logical_tensor> &ts);

/// Sets the attribute according to the name and type (int64_t).
///
/// @tparam Type Attribute's type.
/// @param name Attribute's name.
/// @param value The attribute's value.
/// @returns The Op self.
template <typename Type,
        requires<std::is_same<Type, int64_t>::value> = true>
op &set_attr(attr name, const Type &value);

/// Sets the attribute according to the name and type (float).
///
/// @tparam Type Attribute's type.
/// @param name Attribute's name.
/// @param value The attribute's value.
/// @returns The Op self.
template <typename Type, requires<std::is_same<Type, float>::value> = true>
op &set_attr(attr name, const Type &value);
/// Sets the attribute according to the name and type (bool).
///
/// @tparam Type Attribute's type.
/// @param name Attribute's name.
/// @param value The attribute's value.
/// @returns The Op self.
template <typename Type, requires<std::is_same<Type, bool>::value> = true>
op &set_attr(attr name, const Type &value);

/// Sets the attribute according to the name and type (string).
///
/// @tparam Type Attribute's type.
/// @param name Attribute's name.
/// @param value The attribute's value.
/// @returns The Op self.
template <typename Type,
        requires<std::is_same<Type, std::string>::value> = true>
op &set_attr(attr name, const Type &value);

/// Sets the attribute according to the name and type
/// (std::vector<int64_t>).
///
/// @tparam Type Attribute's type.
/// @param name Attribute's name.
/// @param value The attribute's value.
/// @returns The Op self.
template <typename Type,
        requires<std::is_same<Type, std::vector<int64_t>>::value> = true>
op &set_attr(attr name, const Type &value);

/// Sets the attribute according to the name and type (std::vector<float>).
```

```
    ///
    /// @tparam Type Attribute's type.
    /// @param name Attribute's name.
    /// @param value The attribute's value.
    /// @returns The Op self.
    template <typename Type,
            requires<std::is_same<Type, std::vector<float>>::value> = true>
    op &set_attr(attr name, const Type &value);
};
```

Graph#

Graph contains a set of OPs. `add_op()` adds an OP and its logical tensors to a graph. oneDNN Graph implementation accumulates the OPs and logical tensors and constructs and validates the graph as internal state. During `add_op()`, the target OP will be validated against its schema. Once the validation fails, an exception will be thrown out from the API. When `allow_exception=false` is specified, `add_op()` call returns a status. It is the user's responsibility to handle the error either by checking the return value of the API or handling the exception.

A same logical tensor may appear more than twice in `add_op()` call, since it is passed with the producer OP and consumer OPs. oneDNN Graph validates logical tensors with the same id should be identical at the graph construction time.

At the end of graph construction, users need call `finalize()` API to indicate that the graph is ready for partitioning. Then users call `get_partitions()` which returns a set of partitions. After `get_partitions()`, users shall not add ops to the graph. The graph doesn't hold any meaning to the user after partitioning. Users should free the graph.

All the OPs added to the graph will be contained in one of the returned partitions. If an OP is not supported by the oneDNN Graph API implementation, the corresponding partition will be marked as "not supported". Users can check the supporting status of a partition via the API `is_supported()`. Partitions should not form cyclic dependence within the graph. If user doesn't pass a complete graph, it is the user's responsibility to detect any dependence cycle between the partitions and operations not passing to oneDNN Graph implementation.

The logical tensor passed at the graph construction stage might contain incomplete information, for example, dimension and shape information are spatially known. Complete information is not required but helps the oneDNN Graph to form better partition decisions. Adding op to a graph is not thread-safe. Users must create a graph, add op, and get partition in the same thread.

```
/// A graph object.
class graph {
public:
    /// Constructs a graph with an engine kind.
    ///
    /// @param engine_kind Engine kind.
    graph(dnnl::engine::kind engine_kind);

    /// Creates a new empty graph with an engine kind and a floating-point math
    /// mode. All partitions returned from the graph will inherit the engine
    /// kind and floating-point math mode.
    ///
    /// @param engine_kind Engine kind.
    /// @param mode Floating-point math mode.
    graph(dnnl::engine::kind engine_kind, dnnl::fpmath_mode mode);

    /// Adds an op into the graph to construct a computational DAG. The API will
    /// return failure if the operator has already been added to the graph or
```

```
    /// the operation cannot pass the schema check in the library (eg. input and
    /// output numbers and data types, the attributes of the operation, etc.).
    ///
    /// @param op An operation to be added.
    /// @param allow_exception A flag indicating whether the method is allowed
    ///     to throw an exception if it fails to add the op to the graph.
    /// @returns #status::success or a status describing the error otherwise.
    status add_op(const op &op, bool allow_exception = true);

    /// Finalizes a graph. It means users have finished adding operations into
    /// the graph and the graph is ready for partitioning. Adding a new
    /// operation into a finalized graph will return failures. Similarly,
    /// partitioning on a un-finalized graph will also return failures.
    void finalize();

    /// Checks if a graph is finalized.
    ///
    /// @return True if the graph is finalized or false if the graph is not
    /// finalized.
    bool is_finalized() const;

    /// Gets filtered partitions from a graph. Partitions will be claimed
    /// internally according to the capability of the library, the engine kind
    /// of the graph, and the policy.
    ///
    /// @param policy Partition policy, defaults to policy
    ///     #dnnl::graph::partition::policy::fusion.
    /// @return A vector storing the partitions.
    std::vector<partition> get_partitions(
            partition::policy policy = partition::policy::fusion);
};
```

Partition#

Partition represents a collection of OPs identified by oneDNN Graph implementation as the basic unit for compilation and execution. It contains a list of OP, input ports, output ports, and a flag indicating whether the partition is supported. When a partition is created, it's assigned with an ID. oneDNN Graph implementation should guarantee the partition ID is globally unique.

Users can pass the output logical tensors with incomplete shape information (containing -1) to partition compilation API. oneDNN Graph implementation needs calculate the output shapes according to the given input shapes and schema of the OP. After compilation finished, a compiled partition will be generated with full shape information for the input and output logical tensors. Users can query the compiled partition for the output logical tensors and get the shapes.

Partition can be compiled to generate hardware ISA level binary code specialized for input and output tensors' metadata. Users must pass as much tensor metadata as possible to get the best performant compiled code. When users pass partition shape information, it is implementation-dependent to decide whether to support the compilation.

Users must create an input logical tensor list and an output logical tensor list to pass the additional tensor metadata as parameters to the compilation API. The input and output logical tensors must match the id of partitions' ports, which captures the logical tensors information during graph partitioning.

Users must specify `strided`, `any`, or `opaque` as the `layout_type` for the parameter logical tensors. When users specify `any` for a logical tensor, the tensor must be an output tensor, and oneDNN Graph implementation decides the best performant layout for the compiled partition. If it is `strided`, it must use the public data layout described by the logical tensor. For `opaque`, the parameter logical tensor

contains a target-specific layout, which must be determined by the compilation of preceding partitions producing the tensor. If the layout is row-major contiguous, the compilation must succeed. If the layout has a stride, it is implementation dependent whether the compilation succeed. If certain dimension of shape or the rank is unknown, it is implementation dependent whether the compilation succeed. If the compilation succeeds for unknown dimension or rank, the compiled partition should be able to handle any value for that dimension or any rank at the execution time.

```
/// Policy specifications for partitioning.
enum class policy {
    /// Max policy is to be defined. The library intends to deliver best
    /// optimization and larger partition with max policy. It also means
    /// users may lose fine-grained control the operations in the partition.
    /// Currently, max policy has the same effect as fusion policy.
    max,
    /// Fusion policy returns partitions with typical post-op fusions, eg.
    /// Convolution + ReLU or other element-wise operations or a chian of
    /// post-ops.
    fusion,
    /// Debug policy doesn't not apply any fusions. It returns partitions
    /// with single operations in each partition. The policy is useful when
    /// users notice any bug or correctness issue in max policy or fusion
    /// policy.
    debug,
};

/// Partition kind. It defines the basic structure of the subgraph contained
/// in a partition. For example, kind
/// #dnnl::graph::partition::kind::convolution_post_ops indicates the
/// partition contains one Convolution and its post-ops. But the operation
/// kind of the post-ops are not specified. Partition's kind is decided by
/// the library internally and can be queried from a partition.
enum class kind {
    /// The partition's kind is not defined.
    undef,
    /// The partition contains a Convolution and its post-ops.
    convolution_post_ops,
    /// The partition contains a ConvTranspose and its post-ops.
    convtranspose_post_ops,
    /// The partition contains an Interpolate and its post-ops.
    interpolate_post_ops,
    /// The partition contains a MatMul and its post-ops.
    matmul_post_ops,
    /// The partition contains a Reduction and its post-ops.
    reduction_post_ops,
    /// The partition contains an Unary op and its post-ops.
    unary_post_ops,
    /// The partition contains a Binary op and its post-ops.
    binary_post_ops,
    /// The partition contains a Pooling op (AvgPool or MaxPool) and its
    /// post-ops.
    pooling_post_ops,
    /// The partition contains a BatchNorm op and its post-ops.
    batch_norm_post_ops,
    /// Other partitions based on post-ops but not specified by above kinds.
    misc_post_ops,
    /// The partition contains a quantized version of Convolution and its
    /// post-ops.
    quantized_convolution_post_ops,
    /// The partition contains a quantized version of ConvTranspose and its
    /// post-ops.
    quantized_convtranspose_post_ops,
    /// The partition contains a quantized version of MatMul and its
    /// post-ops.
    quantized_matmul_post_ops,
    /// The partition contains a quantized version of Unary op and its
```

```
    /// post-ops.
    quantized_unary_post_ops,
    /// The partition contains a quantized version of Pooling op and its
    /// post-ops.
    quantized_pooling_post_ops,
    /// Other partitions based quantization and post-ops but not specified
    /// by above kinds.
    misc_quantized_post_ops,
    /// The partition contains a Convolution backward op and its post-ops.
    convolution_backprop_post_ops,
    /// The partition contains a variant of Multi-head Attention.
    mha,
    /// The partition contains a variant of Multi-layer Perceptron.
    mlp,
    /// The partition contains a variant of quantized MHA.
    quantized_mha,
    /// The partition contains a variant of quantized MLP.
    quantized_mlp,
    /// The partition contains a variant of residual convolutional block.
    residual_conv_blocks,
    /// The partition contains a variant of quantized version of residual
    /// convolutional block.
    quantized_residual_conv_blocks,
};

/// A partition object.
class partition {
public:
    partition() = default;

    /// Constructs a partition object
    ///
    /// @param p A raw pointer to the C API handle
    partition(dnnl_graph_partition_t p);

    /// Creates a new partition with a given operator and engine kind. The API
    /// is used to create a partition from an operation directly without
    /// creating the graph and calling `get_partitions()`. The output partition
    /// contains only one operation.
    ///
    /// @param aop An operation used to create the partition.
    /// @param ekind Engine kind.
    partition(const op &aop, engine::kind ekind);

    /// Returns the number of operations contained in the partition.
    ///
    /// @returns Number of operations.
    size_t get_ops_num() const;

    /// Returns all operation IDs contained in the partition.
    ///
    /// @returns An unordered set of operation IDs.
    std::vector<size_t> get_ops() const;

    /// Returns the unique ID of the partition. Partition ID is generated by the
    /// library internally. The ID can be used for debugging purpose or verbose.
    ///
    /// @returns ID of the partition.
    size_t get_id() const;

    /// Compiles a partition with given input and output logical tensors. The
    /// output logical tensors can contain unknown dimensions. For this case,
    /// the compilation will deduce the output shapes according to input shapes.
    /// The output logical tensors can also have layout type `any`. The
    /// compilation will choose the optimal layout for output tensors. The
```

```
    /// optimal layout will be represented as an opaque layout ID saved in the
    /// output logical tensor.
    ///
    /// @param inputs A list of input logical tensors.
    /// @param outputs A list of output logical tensors.
    /// @param e The engine used to compile the partition.
    /// @returns A compiled partition.
    compiled_partition compile(const std::vector<logical_tensor> &inputs,
            const std::vector<logical_tensor> &outputs, const engine &e) const;

    /// Returns the supporting status of a partition. Some operations may not be
    /// supported by the library under certain circumstances. During
    /// partitioning stage, unsupported partitions will be returned to users
    /// with each containing an unsupported operation. Users should check the
    /// supporting status of a partition before transforming the computation
    /// graph or compiling the partition.
    ///
    /// @returns @c true if this partition is supported or @c false if this
    ///     partition isn't supported by the library
    bool is_supported() const;

    /// Returns a list of input logical tensors from the partition.
    ///
    /// @returns A list of input logical tensors.
    std::vector<logical_tensor> get_in_ports() const;

    /// Returns a list of output logical tensors from the partition.
    ///
    /// @returns A list of output logical tensor.
    std::vector<logical_tensor> get_out_ports() const;

    /// Returns the engine kind of the partition
    ///
    /// @returns The engine kind
    engine::kind get_engine_kind() const;

    /// Returns the kind of the partition.
    ///
    /// @returns The partition kind.
    kind get_kind() const;
};
```

Tensor#

Tensor is an abstraction for multidimensional input and output data needed in the execution of a compiled partition. A tensor contains a logical tensor, an engine and a data handle.

Users are responsible for managing the tensor's lifecycle, e.g. free the resource allocated, when it is not used anymore.

```
/// A tensor object
class tensor {
public:
    using dims_t = std::vector<dnnl_dim_t>;

    /// Default constructor. Constructs an empty object.
    tensor() = default;

    /// Constructs a tensor object according to a given logical tensor, an
    /// engine, and a memory handle.
    ///
    /// @param lt The given logical tensor
    /// @param aengine Engine to store the data on.
```

```
    /// @param handle Handle of memory buffer to use as an underlying storage.
    tensor(const logical_tensor &lt, const engine &aengine, void *handle);

    /// Returns the underlying memory buffer.
    ///
    /// On the CPU engine, or when using USM, this is a pointer to the
    /// allocated memory.
    void *get_data_handle() const;

    /// Sets the underlying memory handle.
    ///
    /// @param handle Memory handle.
    void set_data_handle(void *handle);

    /// Returns the associated engine.
    ///
    /// @returns An engine object
    engine get_engine() const;
};
```

Compiled Partition#

A *compiled partition* represents the generated code specialized for target hardware and meta data described by parameter logical tensors. Compiled partition contains a partition and a handle representing the target specific compiled object.

After the compilation API is invoked, users must query the logical output tensor of the compiled partition to know the output tensor's layout id and size. The layout id is an opaque identifier for the target-specific layout. Users may pass the layout id for the next partition compilation so that it can be optimized to expect a specific input layout. Users may use the size to allocate the memory buffer of the output tensors for execution.

Framework passes the tensors and compiled partition as parameters to execution API. The parameter logical tensors must be in the same order when they are passed in the compilation API, and their IDs must match with the compiled partition's internal logical tensors. The layout type of each tensor must be `strided` or `opaque`.

The compiled partition may support in-place optimization, which reuses the input tensor data buffer for the output tensor for lower memory footprint and better data locality. For each compiled partition, users can get pairs of input and output ports. For the pair of input and output ports, user can use a same memory buffer when passing input and output tensors along with execution API. The in-place optimization is optional, when users use another memory buffer for the output tensor, oneDNN Graph must update the output tensor.

If users place a tensor with data buffer pointer in outputs, the backend shall use the data buffer provided by users.

Users may convert the parameter tensor with public layout to the target specific layout expected by the compiled partition. A common optimization in deep learning inference is that users may prepack the weight in the target-specific layout required by the compiled partition and cache the reordered weight for late use.

```
/// A compiled partition object.
class compiled_partition {
public:
    /// Default constructor. Constructs an empty object.
    compiled_partition() = default;
```

```
    /// Queries an input or output logical tensor according to tensor ID. If the
    /// tensor ID doesn't belong to any input or output of the compiled
    /// partition, an exception will be raised by the API.
    ///
    /// @param tid The unique id of required tensor.
    /// @returns The logical tensor.
    logical_tensor query_logical_tensor(size_t tid) const;

    /// Returns the hint of in-place pairs from a compiled partition. It
    /// indicates that an input and an output of the partition can share the
    /// same memory buffer for computation. In-place computation helps to reduce
    /// the memory footprint and improves cache locality. But since the library
    /// may not have a global view of user's application, it's possible that the
    /// input tensor is used at other places in user's computation graph. In
    /// this case, the user should take the in-place pair as a hint and pass a
    /// different memory buffer for output tensor to avoid overwriting the input
    /// memory buffer which will probably cause unexpected incorrect results.
    ///
    /// @returns A list of pairs of input and output IDs.
    std::vector<std::pair<size_t, size_t>> get_inplace_ports() const;

    /// Execute a compiled partition.
    ///
    /// @param astream Stream object to run over.
    /// @param inputs A list of input tensors.
    /// @param outputs A list of output tensors.
    void execute(dnnl::stream &astream, const std::vector<tensor> &inputs,
            const std::vector<tensor> &outputs) const;
};

/// Executes a compiled partition in a specified stream and returns a SYCL
/// event.
///
/// @param c_partition Compiled partition to execute.
/// @param astream Stream object to run over
/// @param inputs Arguments map.
/// @param outputs Arguments map.
/// @param deps Optional vector with `sycl::event` dependencies.
/// @returns Output event.
inline sycl::event dnnl::graph::sycl_interop::execute(
        compiled_partition &c_partition, stream &astream,
        const std::vector<tensor> &inputs, std::vector<tensor> &outputs,
        const std::vector<sycl::event> &deps = {});
```

Engine#

Engine represents a device and its context. Compiled partitions are associated with engines. A compiled partition should only access the tensor which is associated with the same device and context, no matter the tensor is produced by a compiled partition or created directly by the user.

Engine contains device kind, and a device id or device handle. From the device kind, the engine knows how to generate code for the target device and what kind of device object to be expected. The device id ensures that there is a unique engine being created for each device. The device handle passed from framework allows oneDNN Graph implementation to work on the device specified by the framework.

User programs may access the device directly and interoperate with oneDNN Graph to perform a task on the device. Typically user programs manage the device, which create the device handle and use that to create an oneDNN Graph engine. User programs can generate a tensor on a device and pass it to a compiled partition associated with that engine.

Stream#

Stream is the logical abstraction for execution units. It is created on top of oneDNN Graph engine. For SYCL device, it contains an openCL queue. oneDNN Graph engine may have multiple streams. A compiled partition is submitted to a stream for execution.

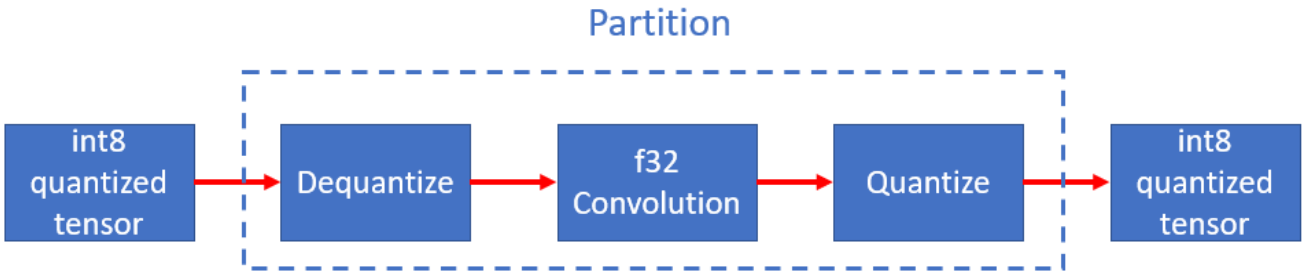
Low Precision Support#

oneDNN Graph provides low precision support including both int8 (signed/unsigned 8-bit integer), bf16, and f16. For int8, oneDNN Graph API supports quantized model with static quantization. For bf16 or f16, oneDNN Graph supports deep learning framework's auto mixed precision mechanism. In both cases, oneDNN Graph API expects users to convert the computation graph to low precision representation and specify the data's precision and quantization parameters. oneDNN Graph API implementation should strictly respect the numeric precision of the computation.

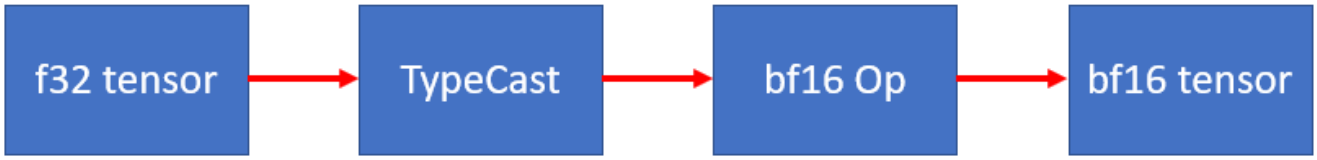
For int8, oneDNN Graph API provides two operations: Dequantize and Quantize. Dequantize takes integer tensor with its associated scale and zero point and returns f32 tensor. Quantize takes f32 tensor, scale, zero point, and returns integer tensor. The scale and zero point are single dimension tensors, which could contain one value for the per-tensor quantization case or multiple values for the per-channel quantization case. The integer tensor could be represented in both unsigned int8 or signed int8. Zero point could be zero for symmetric quantization scheme, and a non-zero value for asymmetric quantization scheme.

Users should insert Dequantize and Quantize in the graph as part of quantization process before passing to oneDNN Graph. oneDNN Graph honors the data type passed from user and faithfully follows the numeric semantics. For example, if the graph has a Quantize followed by Dequantize with exact same scale and zero point, oneDNN Graph implementation should not eliminate them since that implicitly changes the numeric precision.

oneDNN Graph partitioning API may return a partition containing the Dequantize, Quantize, and Convolution operations in the between. Users don't need to recognize the subgraph pattern explicitly and convert to fused op. Depending on oneDNN Graph implementation capability, the partition may include more or fewer operations.



For bf16, oneDNN Graph provides TypeCast operation, which can convert a f32 tensor to bf16 or f16, and vice versa. All oneDNN Graph operations support bf16 and f16. It is user's responsibility to insert TypeCast to clearly indicate the numeric precision. oneDNN Graph implementation fully honors the user-specified numeric precision. If users first typecast from f32 to bf16 and convert back, oneDNN Graph implementation does the exact data type conversions underneath.



General API notes#

There are certain assumptions on how oneDNN Graph objects behave:

- Logical tensor behave similarly to trivial types.
- All other objects behave like shared pointers. Copying is always shallow.

Error Handling#

The C++ API throws exceptions for error handling.