VPUX NN Compiler Software Architecture Specification

Vladislav Vinogradov

October 20, 2021

1 Introduction

The **VPUX NN Compiler** is a new NN compiler for VPU platforms. It is based on MLIR framework and utilizes its API and features.

1.1 Goals

The **VPUX NN Compiler** is designed to achieve the following goals:

- Improve network coverage for current VPU generation.
- Make new network enablement process lighter and faster.
- Extend compilation features set:
- Tensors with arbitrary number of dimensions.
- Dynamic shapes.
- Control flow.
- Improve compilation performance.
- Integrate existing solutions from different projects in single code base.
- Design future-proof architecture with extensibility to the next VPU generations.
- Improve developer experience (debuggability, self-validation techniques, testing approach).

1.2 MLIR Framework

The **VPUX NN Compiler** utilizes the following feature from MLIR to improve developer experience:

- IR manipulations.
- Transformations and pass management.
- IR self-validation.
- Unit testing.
- Debugging.

2 Design Principles

The **VPUX NN Compiler** architecture and its implementation is based on the following principles:

1. Explicit notion for IR validity and invariants.
2. Enforced architectural stability and self-validation during compilation pipeline.

3. IR splitting onto separate stages with different level of details.
4. Operation interfaces for generic passes.
5. Atomic passes and pipelines.

The first principle is achieved by MLIR architecture - validation hooks. Each operation/attribute/type has its own validation hook, which checks all invariants of the entity. Each pass/transformation takes a valid IR as input and produces a valid IR as output.

The second principle is described in details in separate section.

The third principle is achieved by MLIR architecture - Dialects concept. The **VPUX NN Compiler** consists of several Dialects with different level of details. The IR is lowered from high level abstractions to more detailed representation step-by-step during compilation pipeline.

The fourth principle encourages using such MLIR concepts as Operation Traits and Interfaces. They allow to reduce code duplication and group similar Operations under a single API. Operation Interfaces also allows to write more generic passes, which are not bound to particular operation set.

The fifth principle declares that each Pass in compilation pipeline must represent one single transformation to reach one particular goal (either IR adaptation or IR optimization). Such "atomic" pass is easier to be covered by unit testing. The "atomic" passes can be joined together in the compilation chain inside pipeline. The pipeline doesn't perform IR transformation on its own, instead it represents a sequence of other passes. The goal of pipeline is to establish correct order of underlying passes, while keeping actual transformation logic inside them.

2.1 Architectural Stability of NN Compiler

2.1.1 Introduction

The stability and debuggability of NN compiler architecture is quite important feature. It improves developer experience and allow to satisfy the following scalability requirements in faster way:

- New networks support.
- Current supported networks improvements (accuracy and/or performance).
- New compiler features (might be imposed by previous items).

Since the NN compiler contains a set of complex optimization passes with dependencies on each other, any change in it might introduce new issues. This increases the time for new features/network enablement, sometimes dramatically. This document discusses some ways to reduce this complexity on architectural level and speed up development process.

Each NN compiler has its own intermediate representation of the model, which it works on. The compiler passes modify this model in place and share some information threw it. The IR itself, just like any other object, should have so called valid state, or some combination of internal invariants. Each compiler pass shouldn't break this valid state. The violation of this rule is one of the most cause of issues with new networks/features.

The compiler architecture stability in this point of view is an API property, which helps developers to keep the IR valid after any transformations done in pass. This also implies that the API should help developers to focus on actual adaptation/optimization task rather than on IR validity checking/keeping. The API should do this automatically and prevent IR transformations that break the combination of internal invariants.

The debuggability in that scope is a set of extra helper methods, which helps to find the IR validity violation. It should point to exact place, where the issue happened and provide as much information as possible to the developer. This will reduce time spent on issue debugging.

2.1.2 IR Valid State

The IR valid state is a combination of different parts:

1. Graph structure validity.
2. Internal information validity.
3. Run-time structure validity.

Each of the parts are covered below in details.

2.1.2.1 Graph Structure Validity

The IR used by NN compiler is represented as some graph structure and this implies some graph structure invariants. This includes:

- Coherency of links between graph objects. For example, for node edge `[src] -> [dst]`, the node `[src]` has a link to node `[dst]` and, vice versa, the node `[dst]` has a link to node `[src]`. In more complex case the edge itself might be represented as separate object, and nodes will have links to that edge object and the edge object will have links to the nodes.
- No cycles in the graph.
- Graph connectedness. There network inputs should be connected with network outputs, in other words, we can reach network inputs from network outputs following backward edges.

2.1.2.2 Internal Information Validity

The NN compiler IR includes not only the connections between internal elements, but some additional information bound to them. In most cases, this information is called *attributes* and are bound to graph nodes and, sometimes, to graph edges. We will use term IR information for the full set of the attributes, bound to its elements.

First, we should divide these attributes onto several sections:

1. **Primary attributes**. Those attributes define the IR, or, in other words, the IR object can't exist without those attributes. Example of such attributes: the type of the IR node (data or operation), shape for tensor, mandatory parameter for NN layer (strides for Convolution, axis for SoftMax).
2. **Compilation attributes**. Those attributes do not exist at the beginning of the compilation process, but are computed and added to the IR during the compilation/adaptation/optimization. For example, memory allocation for tensors. These attributes will be added after corresponding compiler pass or set of passes are executed.
3. **Computable attributes**. Those attributes can be computed from other attributes. For example, full size of tensor can be computed from its shape.

Validity state for those attributes includes their coherency with each other. For example, axis parameter for SoftMax shouldn't exceed the number of dimensions of the operation input tensor. But there are also special rules for different attributes kinds.

Since **primary attributes** defines the IR object, there shouldn't be a way to create the object without those attributes.

Since **compilation attributes** arise at some specific point of compilation, before that point the IR shouldn't contain them at all, and after that point the attributes must be kept by further transformation. For example, once the memory allocation pass is done and some memory information is attached to the data objects, next passes should respect this information, either keep it or re-run the allocation procedure.

As for **computable attributes**, since they are some kind a caching for calculated values, they should be always up to date with the attributes used for the calculation.

2.1.2.3 Run-time Structure Validity

Since NN compiler adopts the model for execution on specific hardware it must consider all limitations of the hardware. For VPU specific case, where we might have different ways to execute NN layers with different restrictions. Some restrictions are coming from the hardware level itself (for NCE module, for example), some restrictions are coming from implementation (for SHAVE kernels). There also might be performance implications, when several ways are possible, but some of them has penalties.

The IR run-time structure validity means that all these restrictions are satisfied.

2.1.3 Proposals

This section will present some architectural proposals to make NN compiler more stable in terms of its evolution.

The generic proposal is that all compiler and IR APIs should perform checks for input parameters and for operation transformation and immediately report in case of errors. This will allow to find errors in pass just in place, where they happened. The checks must be included into the Release build too.

2.1.3.1 Graph Structure Validity Implementation

First, the IR should avoid complicated representation forms and avoid introducing new entities if something can be represented with existing model. This will reduce misunderstanding or misuse of its API in passes. It can consist only of Values and Operations. The computation and execution flow can be fully defined by the dependencies between Values and Operations as well as internal order of the Operations. If the Operation takes the Value as input parameter, the Operation that produces this Value should be executed prior to its consumer. To make additional explicit dependencies between operations (order them to reuse memory, for example), IR might use Barriers as separate kind of Values. In that case each Operation in addition to Tensor outputs also produces single Barrier output. Each Operation, in addition to Tensor inputs, might take extra Barrier inputs. If we'd like to order Operations, that doesn't have Value dependencies over Tensors, we just link them via Value dependency of Barriers. From the graph topology point of view, this connection doesn't differ from connection via Tensors, so passes should only care about Value dependencies.

Second, since in NN all layers can be considered as pure functions without side effects, the compiler might automatically remove operations, which outputs are not used. In that case, to replace the sub-graph, the pass just needs to create new operations and replace the users of one set of Values to other set of Values. The compiler then might remove unused old operations and check that the graph is still connected (there is a path from network inputs to network outputs).

2.1.3.2 Internal Information Validity Implementation

First, **computable attributes** shouldn't be stored as normal attributes. Instead they should be computed on the fly or, for performance reasons, some separate cache mechanism should be used, which will be automatically notified about change of other attributes.

Second, generic types like `string` or `any` should be avoided for IR objects, limiting the supported attributes for each Operation kind. Each Operation (represented as separate class) should explicitly mention supported **primary attributes** (without ability of their modification) and **computation attributes**.

Third, instead of storing some information as attributes of Operations, this information can be represented as separate Operations. For example, explicit `Quantize`, `Dequantize`, `FakeQuantize` operations instead of quantization parameters attribute. The compiler passes can detect patterns like `[Dequantize] -> [Convolution]` and merge them.

2.1.3.3 Run-time Structure Validity Implementation

As already mentioned, each Operation type should be represented as separate class. The class should encapsulate all information about the run-time restrictions for that Operation and check them each time the Operation is modified (eg, its inputs are replaced). The passes should use some API to get the information about the restrictions rather than assuming them based on the Operation type.

Each IR modification done by some pass should be atomic and valid. In other word [Imielick] Such automatic propagation mey be tricky or problematic in case of several interependent attributes/inputs. Assuming we do not allow invalid state intermittently, we may not be able to change an input A without changing the input B if further in the chain these attributes impact the same output, (i.e. each changed independently may lead to invlalid output). In other cases it may lead to unnecessary computations. Perhaps it's not an issue with a careful design, but then it would not be a general rule but rather be defined as a set of more specific invariants. Alternatively "update" and "validate" stages could be separated to explicitly allow objects in intermittent invalid satte. s, it should convert valid IR to another valid IR. The compiler should perform IR validity checks after each pass and/or after each particular modification. The modification can be represented as:

- A single pass, which performs global modification at once.
- A part of pattern match-and-replace logic, where modification affects some part of the IR.

3 Generic Architecture

The **VPUX NN Compiler** consists of the following parts:

- **Core utilities**.
- **FrontEnd**.
- **Dialects**
- **Compilation pipelines**
- **BackEnd**.

3.1 Core Utilities

The **VPUX NN Compiler** core utilities includes various auxiliary classes and functions to simplify IR interpretation and transformations:

- `src/experimental/vpux_compiler/include/vpux/compiler/utils/`
- `src/experimental/vpux_compiler/include/vpux/compiler/core/ops_interfaces.hpp`
- `src/experimental/vpux_compiler/include/vpux/compiler/core/static_allocation.hpp`
- `src/experimental/vpux_compiler/include/vpux/compiler/core/attributes/const_content.hpp`

One part of the core utilities is tensor shape/stride/layout manipulation API:

- `src/experimental/vpux_compiler/include/vpux/compiler/core/attributes/dim.hpp`

- `src/experimental/vpux_compiler/include/vpux/compiler/core/attributes/dim_values.hpp`
- `src/experimental/vpux_compiler/include/vpux/compiler/core/attributes/dims_order.hpp`
- `src/experimental/vpux_compiler/include/vpux/compiler/core/attributes/shape.hpp`
- `src/experimental/vpux_compiler/include/vpux/compiler/core/attributes/strides.hpp`
- `src/experimental/vpux_compiler/include/vpux/compiler/core/attributes/stride_reqs.hpp`

3.1.1 Tensor Descriptor

3.1.1.1 Logical vs Physical dimensions

The **VPUX NN Compiler** uses the following terms for the Tensor dimensions:

- **Logical** dimensions.
- **Physical** dimensions (also mentioned as **Memory** dimensions).

Those terms (**logical** and **physical**) are also applied to tensor shape and strides. For example, the **logical** shape means that the dimensions sizes are assigned to **logical** dimensions.

Logical dimensions are abstracted from actual memory buffer layout. Their order in tensor shape is fixed and matches InferenceEngine, nGraph and MLIR order. The actual meaning of each **logical** dimension is a property concrete Operation. For example, Convolution interprets **logical** shape of activations tensor as `[N, C, H, W]` and **logical** shape of weights tensor as `[O, I, KY, KX]`.

Physical dimensions, in contrast, are bound to actual memory layout and ordered from major (most outer) to minor (most inner). They are used to work with memory buffers in common efficient way.

Both **logical** and **physical** dimensions are represented as separate classes (which internally holds single integer value - dimension index). The `Dim` class represents **logical** dimension, while `MemDim` represents **physical** dimension. These classes don't have implicit casting to integer, only explicit getter method for dimension index. These classes are used as keys to access corresponding shape and strides arrays instead of plain integers. In the same way, shape has two implementations (`Shape` and `MemSpace`) and strides (`Strides` and `MemStrides`). The usage of separate classes (while they have common implementation logic) allows to catch all misuse of those two abstractions at compile time.

3.1.1.2 Memory Layout

The `DimsOrder` class represents memory layout information. It holds permutation array (in packed format) from **logical** dimensions to **physical** dimensions. This class provides API to convert between those two representations in both way. The class also provides API to work with MLIR class (`AffineMap`), which represents more generic layout description.

3.1.1.3 Strides Requirements

The final utility class in this section is `StrideReqs` . It is used to collect various requirements for strides from different places and to calculate the strides based on this information. It supports the following requirements:

- `Any` - means that there is no special requirements for particular dimension.
- `Compact` - the stride for this dimension must not introduce gaps between neighbor elements in this dimension.
- `Aligned` - the byte stride for this dimension must be aligned by particular value.

- `Fixed` - the stride for this dimension must be equal to fixed value.

3.1.2 Quantization

The **VPUX NN Compiler** uses MLIR standard `quant` dialect to represent quantization parameters. The parameters are bound to tensor/memref element type, which allows to distinguish RAW integer types from the quantized types.

```
!qElemType = type !quant.uniform<u8:f16, 1.0:128>
//                               |   |    |   |
//                               |   |    |   zero point
//                               |   |    scale
//                               |   expressed type
//                              storage type

%0 = const.Declare tensor<16x3x3x3x!qElemType> =
    #const.Content<dense<1> : tensor<16x3x3x3xu8>,
[#const.QuantCast<!qElemType>]>

%1 = IE.Dequantize(%0) :
    tensor<16x3x3x3x!qElemType> -> tensor<16x3x3x3xf16>

%2 = VPU.NCE.Conv2D(input : %arg0, filter : %1)
```

3.2 FrontEnd

FrontEnd is used to import external source into MLIR infrastructure. It supports the following sources:

- InferenceEngine `CNNNetwork` object - imported as **IE Dialect**.
- RunTime graph blob - imported as **VPUIP Dialect**.

The **FrontEnd** can be called separately by `vpux-translate` tool. This mode is used for **LLVM LIT** based unit testing, for example.

3.2.1 IE Dialect FrontEnd

The **IE Dialect FrontEnd** imports OpenVINO IR (InferenceEngine `CNNNetwork` object) into MLIR world. It performs 1-to-1 opset mapping from the latest nGraph opset to **IE Dialect** opset. The **IE Dialect FrontEnd** intentionally doesn't include any extra opset conversion logic to make it pure straightforward conversion from one C++ library (nGraph) to other (MLIR). The opset conversion is supposed to be done either on nGraph level as nGraph pass or on MLIR level as MLIR pass.

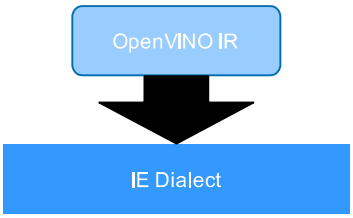

Figure: IE Dialect FrontEnd

3.3 Dialects

The **VPUX NN Compiler** defines the following own Dialects:

- **Const Dialect**
- **IE Dialect**
- **IERT Dialect**
- **VPUIP Dialect**

The Dialect are not used in isolation, but combined with MLIR dialects as well as with each other.

3.3.1 IR Levels

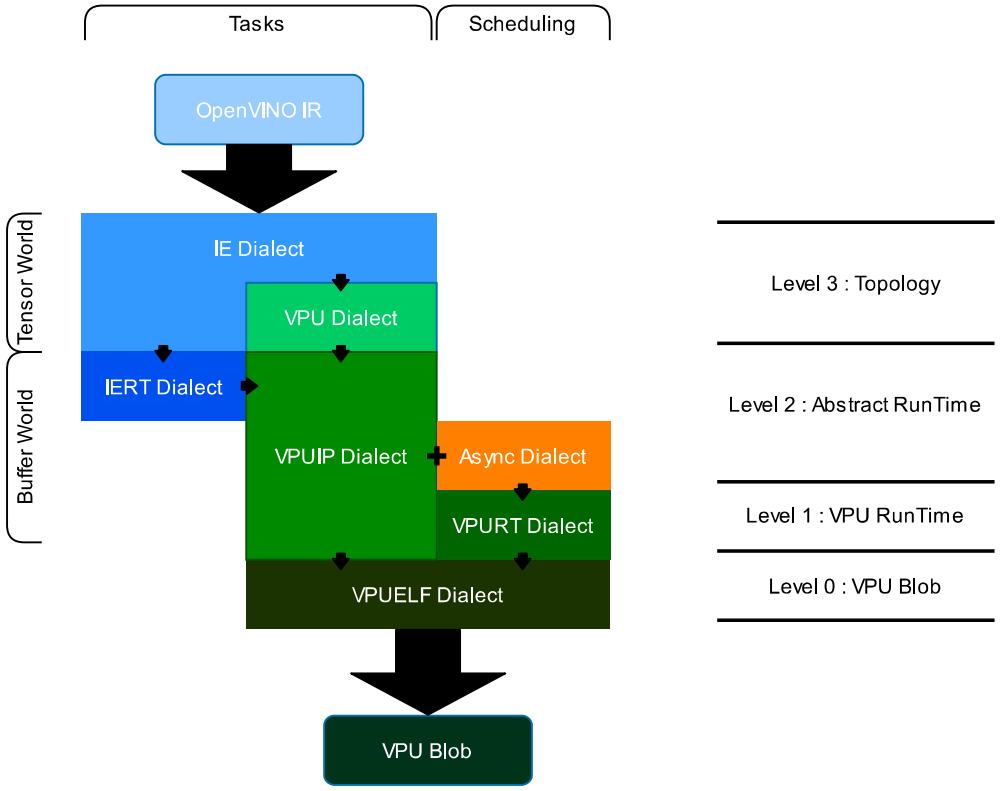

Figure: IR Levels

3.3.1.1 Level 3 : Topology

3.3.1.1.1 HW-agnostic opset

This level represents pure data-flow of the network. It starts from HW-agnostic opset and doesn't contain any predefined scheduling semantic. The dialects on that level operates on **Tensor World** and follows SSA tensor value semantic.

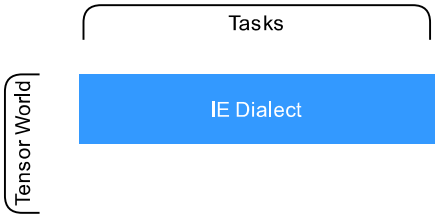

Figure: IE Dialect

Example of the IR on that level:

```
func @main(%arg0: tensor<1x8x4x2xf16>) -> (tensor<1x8x4x2xf16>,
tensor<1x20x8x4xf16>) {
    %0 = const.Declare tensor<1x4x8x20xf16> =
        #const.Content<dense<2.0> : tensor<1x4x8x20xf16>>

    %1 = IE.SoftMax(%arg0) { axis = 1 } :
        tensor<1x8x4x2xf16> -> tensor<1x8x4x2xf16>

    %2 = IE.SoftMax(%0) { axis = 1 } :
        tensor<1x4x8x20xf16> -> tensor<1x4x8x20xf16>

    %3 = IE.Reshape(%2) { static_shape = [1, 20, 8, 4] } :
        tensor<1x4x8x20xf16> -> tensor<1x20x8x4xf16>

    return %1, %3 : tensor<1x8x4x2xf16>, tensor<1x20x8x4xf16>
}
```

The transformations on that stage includes:

- Pre-/post- processing fusing into the main network.
- Canonicalization/simplification of the operations.
- Generic opset adaptation for VPU (preserving HW agnostic specification):
- `AvgPool -> DepthwiseConv`
- `Conv1D -> Conv2D`
- `FullyConnected -> Conv2D`
- etc.

3.3.1.1.2 VPU High ISA

After the initial adaptation on HW agnostic opset, the partial lowering from **IE dialect** to **VPU dialect** as performed. The **VPU dialect** represents VPU HW specific operation in high level view and VPU specific SW kernels. The rest not converted **IE dialect** operations assumed to be executed on SHAVEs. In general it follows the same "pure data flow" semantic as **IE dialect**.

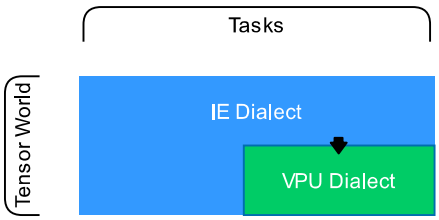

Figure: IE+VPU Dialects

Example of the IR after partial lowering:

```
func @main(%arg0: tensor<1x3x300x300xf16>) -> (tensor<1x16x300x300xf16>) {
    %0 = const.Declare tensor<16x3x3x3xf16> =
        #const.Content<dense<1.000000e+00> : tensor<16x3x3x3xf16>>

    %1 = const.Declare tensor<16xf16> =
        #const.Content<dense<1.000000e+00> : tensor<1x16x1x1xf16>,
[#const.Reshape<[16]>]>

    %2 = VPU.SHAVE.Power(%arg0) { power = 1.0, scale = 0.00390625 } :
        tensor<1x3x300x300xf16> -> tensor<1x3x300x300xf16>

    %3 = VPU.NCE.Conv2D(input : %2, filter : %0, bias : %1)
        { strides = [1, 1], pad = { top = 1, bottom = 1, left = 1, right = 1 }
} :
        tensor<1x3x300x300xf16>, tensor<16x3x3x3xf16>, tensor<16xf16> ->
tensor<1x16x300x300xf16>

    %4 = IE.SoftMax(%3) { axis = 1 } :
        tensor<1x16x300x300xf16> -> tensor<1x16x300x300xf16>

    return %4 : tensor<1x16x300x300xf16>
}
```

The transformations on that stage includes:

- Assign and adjust layouts and memory location for each tensor (based on operations requirements).
- Low precision transformations.
- Tiling (data flow only):
- Introduces Copy operations (will be lowed to DMA).

3.3.1.2 Level 2 : Abstract RunTime

3.3.1.2.1 Bufferization

This level is created by full lowering from previous level as a Bufferization process. This process includes the following steps:

- `Tensor` types are replaced with `MemRef` types.
- Explicit memory allocation operations are added (dynamic allocations as initial state).
- Extra pure-view like operations with aliasing semantic.
- Separate meta-like operation to describe available execution and memory resources.

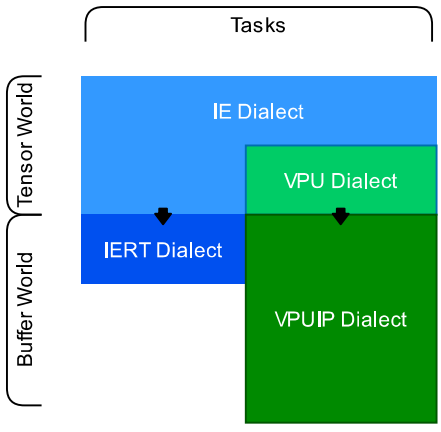

Figure: IERT+VPUIP Dialects

Example of the IR on that level in initial state:

```
#NHWC = affine_map<(d0, d1, d2, d3) -> (d0, d2, d3, d1)>
#OHWI = affine_map<(d0, d1, d2, d3) -> (d0, d2, d3, d1)>

IERT.RunTimeResources @VPU
    availableMemory : {
        IERT.MemoryResource 524288000 bytes of @DDR
            {VPU.bandwidth = 8, VPU.derateFactor = 0.6}
        IERT.MemoryResource 917504 bytes of @CMX_NN [4]
            {VPU.bandwidth = 32, VPU.derateFactor = 1.0}
    }
    executors : {
        IERT.ExecutorResource @DMA_NN
        IERT.ExecutorResource @SHAVE_UPA [16]
        IERT.ExecutorResource @NCE_Cluster [4] {
            IERT.ExecutorResource @DPU [5]
        }
    }

memref.global constant @cst0 : memref<64x16x3x3xf16, #OHWI, #strides0,
@VPU.DDR>
    = dense<[0.0, 1.0, ...]>

func @main(
        %arg0 : memref<1x16x32x32xf16, #NHWC, #strides1, @VPU.DDR> {
IERT.net_input },
        %arg1 : memref<1x64x32x32xf16, #NHWC, #strides2, @VPU.DDR> {
IERT.net_output }) {
    %0 = memref.get_global @cst0 : memref<64x16x3x3xf16, #OHWI, #strides0,
@VPU.DDR>

    // Explicit memory allocation as separate operation
    %1 = memref.alloc() : memref<1x16x32x32xf16, #NHWC, #strides1,
[@VPU.CMX_NN, 0]>
    // Layer operations with memory side-effects:
    // %arg0 has Read memory effect
    // %1 has Write memory effect
    // %2 is an alias for %1 (to preserve data flow information)
    %2 = IERT.Copy inputs(%arg0) outputs(%1)

    // Pure view-like operation
    // %3 is an alias for %0
    %3 = IERT.SubView %0 [0, 0, 0, 0] [32, 16, 3, 3]
        -> memref<32x16x3x3xf16, #OHWI, #strides0, @VPU.DDR>

    %4 = memref.alloc() : memref<32x16x3x3xf16, #NHWC, #strides0, [@VPU.CMX_NN,
0]>
    %5 = IERT.Copy inputs(%3) outputs(%4) // %5 is an alias for %4

    %6 = memref.alloc() : memref<1x32x32x32xf16, #NHWC, #strides0,
[@VPU.CMX_NN, 0]>
    %7 = VPUIP.NCETask inputs(%2, %5) outputs(%6) { strides = [1, 1], MPE_mode
= "VECTOR16" }
        variants : {
            VPUIP.DPUTask { start = [0, 0, 0], end = [31, 15, 31] }
            VPUIP.DPUTask { start = [0, 16, 0], end = [31, 15, 31] }
        }
        PPE : {
```

```
            VPUIP.PPE.RELUX { clamp_min = 0, clamp_max = 6 }
        }

    %8 = IERT.Copy inputs(%7) outputs(%arg1)
    return
}
```

3.3.1.2.2 Opset Finalization

After the Bufferization pipeline is finished few generic optimizations can be applied to that initial form:

- Canonicalization and simplification.
- In-place operations handling.

Next step on that level is final mapping of operations set to **VPUIP dialect**.

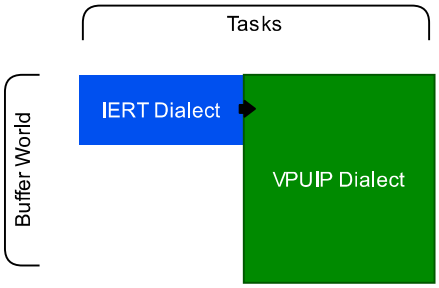

Figure: VPUIP Dialect

3.3.1.2.3 Scheduling

The last part on this level is scheduling planning and optimization:

- Introduce initial asynchronous scheduling:
- Wrap layer operations into asynchronous regions (MLIR standard **async dialect**).
- Establish scheduling dependencies based on data flow.
- Perform advanced scheduling and static memory assignment.

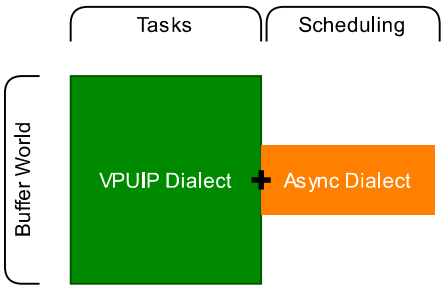

Figure: VPUIP+Async Dialects

Initial scheduling example:

```
%1 = memref.alloc() : memref<1x16x32x32xf16, #NHWC, #strides1, [@VPU.CMX_NN,
0]>
%2 = async.execute { IERT.executor = @VPU.DMA_NN } {
    VPUIP.DMA inputs(%arg0) outputs(%1)
    async.yield
}

%4 = memref.alloc() : memref<32x16x3x3xf16, #NHWC, #strides0, [@VPU.CMX_NN, 0]>
%5 = async.execute { IERT.executor = @VPU.DMA_NN } {
    %0 = memref.get_global @cst0 : memref<64x16x3x3xf16, #OHWI, #strides0,
@VPU.DDR>
    %3 = IERT.SubView %0 [0, 0, 0, 0] [32, 16, 3, 3]
        -> memref<32x16x3x3xf16, #OHWI, #strides0, @VPU.DDR>

    VPUIP.DMA inputs(%3) outputs(%4)
    async.yield
}

%6 = memref.alloc() : memref<1x32x32x32xf16, #NHWC, #strides0, [@VPU.CMX_NN,
0]>
%7 = async.execute [%2, %5] { IERT.executor = [@VPU.NCE_Cluster, 0] } {
    VPUIP.NCETask inputs(%1, %4) outputs(%6)
    async.yield
}
```

After optimizations:

```
%1 = IERT.StaticAlloc <0> : memref<1x16x32x32xf16, #NHWC, #strides1,
[@VPU.CMX_NN, 0]>
%4 = IERT.StaticAlloc <1000> : memref<32x16x3x3xf16, #NHWC, #strides0,
[@VPU.CMX_NN, 0]>
%6 = IERT.StaticAlloc <2000> : memref<1x32x32x32xf16, #NHWC, #strides0,
[@VPU.CMX_NN, 0]>

%2 = async.execute { IERT.executor = @VPU.DMA_NN } {
    %0 = memref.get_global @cst0 : memref<64x16x3x3xf16, #OHWI, #strides0,
@VPU.DDR>
    %3 = IERT.SubView %0 [0, 0, 0, 0] [32, 16, 3, 3]
        -> memref<32x16x3x3xf16, #OHWI, #strides0, @VPU.DDR>

    VPUIP.DMA inputs(%arg0) outputs(%1)
    VPUIP.DMA inputs(%3) outputs(%4)

    async.yield
}

%7 = async.execute [%2] { IERT.executor = [@VPU.NCE_Cluster, 0] } {
    VPUIP.NCETask inputs(%1, %4) outputs(%6)
    async.yield
}
```

3.3.1.3 Level 1 : VPU RunTime

This levels represents VPU run-time logic in terms of MLIR. It is a medium level representation, which reveals VPU run-time scheduling logic and HW configuration without extra low level details like register mappings. This level is created by full lowering from **async dialect** to **VPURT dialect**. The level starts its representation from using virtual barriers, which are then replaced with physical barriers.

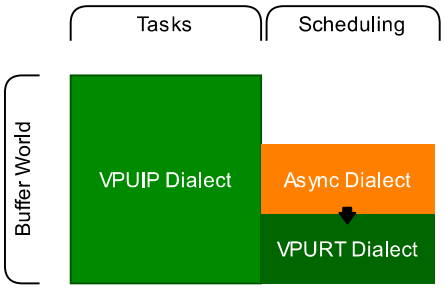

Figure: VPUIP+VPURT Dialects

Example of IR on that level:

```
%1 = VPURT.DeclareBuffer CMX_NN[0] <0> : memref<1x16x32x32xf16, #NHWC,
#strides1, [@VPU.CMX_NN, 0]>
%4 = VPURT.DeclareBuffer CMX_NN[0] <1000> : memref<32x16x3x3xf16, #NHWC,
#strides0, [@VPU.CMX_NN, 0]>
%6 = VPURT.DeclareBuffer CMX_NN[0] <2000> : memref<1x32x32x32xf16, #NHWC,
#strides0, [@VPU.CMX_NN, 0]>
%3 = VPURT.DeclareBuffer BinaryData[0] <0> : memref<32x16x3x3xf16, #OHWI,
#strides0, @VPU.DDR>

%2 = VPURT.DeclareVirtualBarrier
%7 = VPURT.DeclareVirtualBarrier

VPURT.TaskList for @VPU.DMA_NN {
    VPURT.Task updates(%2) {
        VPUIP.DMA inputs(%arg0) outputs(%1)
    }
    VPURT.Task updates(%2) {
        VPUIP.DMA inputs(%3) outputs(%4)
    }
    VPURT.Task waits(%7) {
        VPUIP.DMA inputs(%6) outputs(%arg1)
    }
}

VPURT.TaskList for [@VPU.NCE_Cluster, 0] {
    VPURT.Task waits(%2) updates(%7) {
        VPUIP.NCETask inputs(%1, %4) outputs(%6)
    }
}
```

3.3.1.4 Level 0 : VPU Blob

TBD: finalize design of this level.

This level represents the VPU blob in terms of MLIR and fully matches with its content.

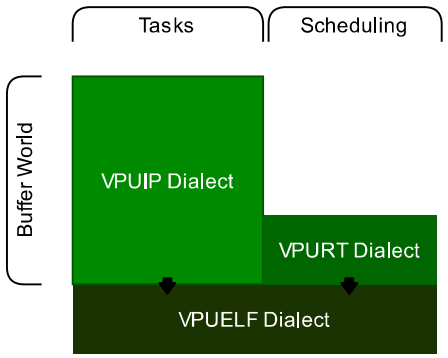

Figure: VPUELF Dialect

3.4 Compilation pipelines

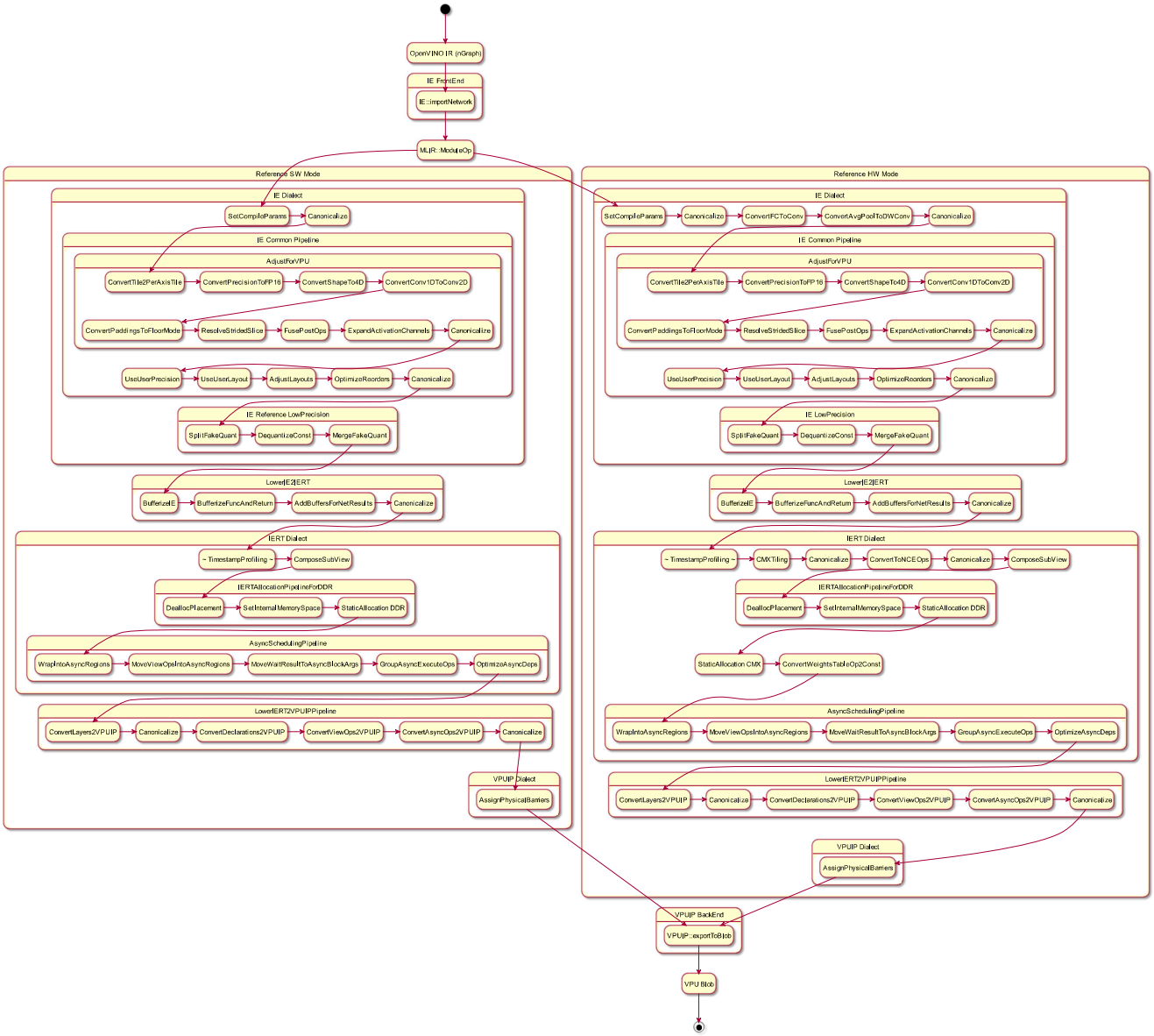

Figure: Compilation pipelines

The **VPUX NN Compiler** defines the compilation process as a set of top-level pass pipelines for various scenarios:

- Reference mode.
- Simple HW mode.
- **TBD:** Advanced HW mode.
- **TBD:** Throughput mode.
- **TBD:** Latency mode.

Pipeline for each mode calls the passes for each Dialect as well as conversion passes to lower from one Dialect to another.

3.5 BackEnd

BackEnd is used to export **VPUX NN Compiler** IR into external output. It supports the following modes:

- **VPUIP Dialect** serialization to runtime blob format.
- **TBD: IE Dialect** convertion to nGraph.

The **BackEnd** can be called separately by `vpux-translate` tool. This mode is used for **LLVM LIT** based unit testing.

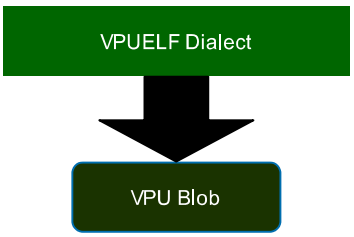

Figure: VPUIP Dialect BackEnd

4 Features Architecture

4.1 Scheduling

This chapter describes the approach used in **VPUX NN Compiler** to produce valid schedule for the run-time, which satisfies both original model data flow and HW resources restrictions. This task is solved step-by-step as a sequence of various passes on different IR levels.

4.1.1 Level 3 : Topology

On this level (**IE Dialect** + **VPU Dialect**) the following parts of scheduling are done:

- Data tiling.
- Memory space adjustment for tensors.
- Copy operation insertion and optimization.

The IR is represented as pure data-flow with special operations for tiles extraction and concatenation:

```
func @main(%in : tensor<1x16x32x32xf16>) -> tensor<1x32x16x16xf16> {
    %weights = const.Declare tensor<32x16x3x3xf16> = !const.Content<...>

    %weights_0 = IE.Slice %weights [0, 0, 0, 0] [16, 16, 3, 3] ->
tensor<16x16x3x3xf16>
    %weights_1 = IE.Slice %weights [16, 0, 0, 0] [16, 16, 3, 3] ->
tensor<16x16x3x3xf16>

    %0 = IE.Copy(%in) -> tensor<1x16x32x32xf16, [@CMX, 0]>
    %1 = IE.Copy(%weights_0) -> tensor<16x16x3x3xf16, [@CMX, 0]>
    %2 = VPU.NCE.Conv(%0, %1) { cluster_id = 0 } -> tensor<1x16x16x16xf16, [@CMX,
0]>
    %3 = IE.Copy(%2) -> tensor<1x16x16x16xf16>

    %4 = IE.Copy(%in) -> tensor<1x16x32x32xf16, [@CMX, 1]>
    %5 = IE.Copy(%weights_1) -> tensor<16x16x3x3xf16, [@CMX, 1]>
    %6 = VPU.NCE.Conv(%4, %5) { cluster_id = 1 } -> tensor<1x16x16x16xf16, [@CMX,
1]>
    %7 = IE.Copy(%2) -> tensor<1x16x16x16xf16>

    %8 = IE.Concat(%3, %7) { axis = 1 } -> tensor<1x32x16x16xf16>
    return %8
}
```

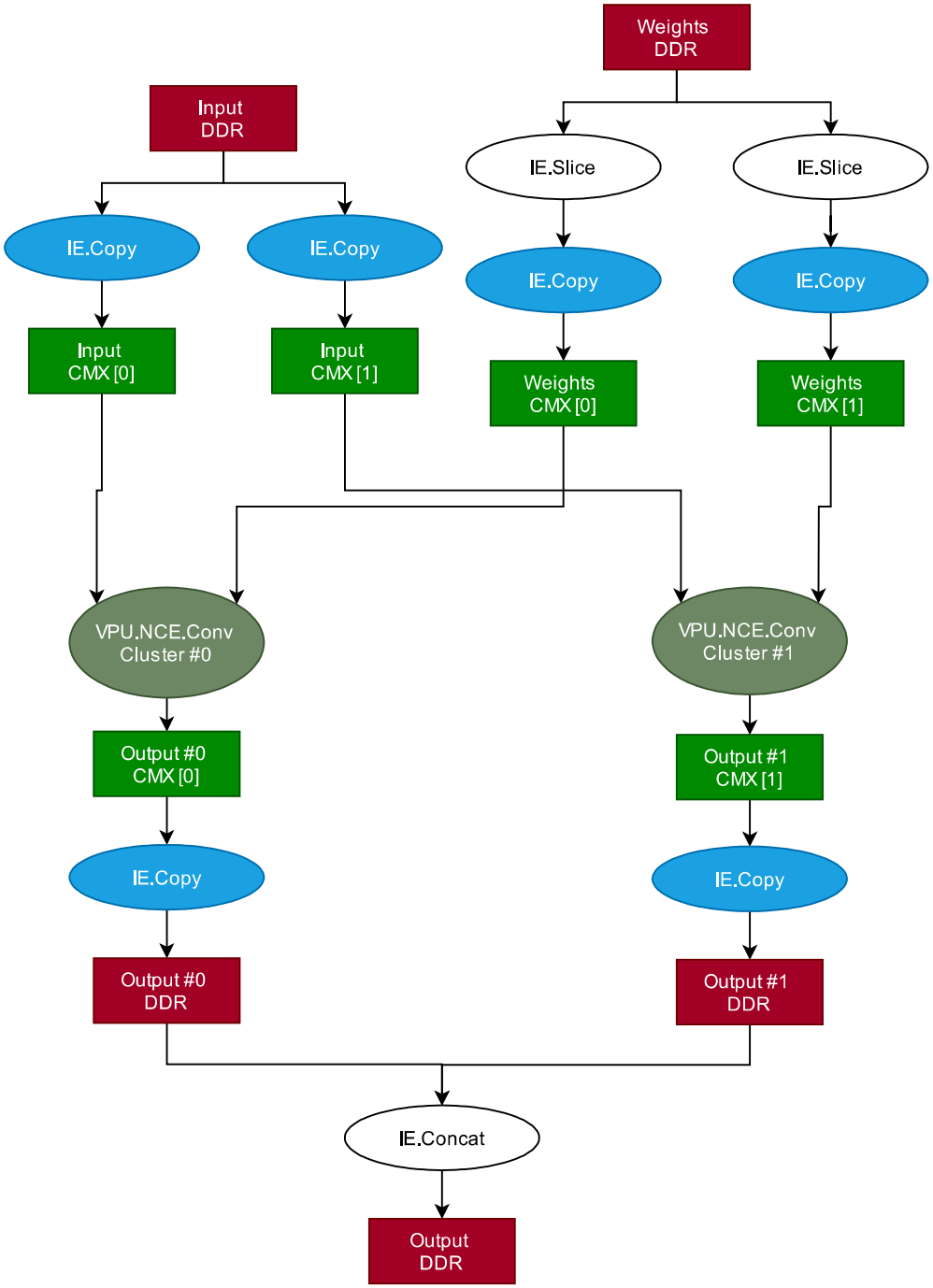

Figure: Topology Level Schedule

Note: despite the fact that this level is pure data flow, the IR is always stored in ordered form. The passes on that level might reorder the operations in the IR (preserving their data flow) for more suitable form for the next levels.

4.1.2 Level 2 : Abstract RunTime

4.1.2.1 Bufferization

The IR on previous level is lowered to this level via special "Bufferization" process. This process will introduce separate `memref.alloc()` operations for explicit Buffer allocations. The run-time layer operations will take the output buffer as their operand and will return an alias to it to preserve data-flow.

```
func @main(%in : memref<1x16x32x32xf16>, %out : memref<1x32x16x16xf16>) ->
memref<1x32x16x16xf16> {
    %weights = const.Declare memref<32x16x3x3xf16> = !const.Content<...>

    %weights_0 = IERT.SubView %weights [0, 0, 0, 0] [16, 16, 3, 3] ->
memref<16x16x3x3xf16>
    %weights_1 = IERT.SubView %weights [16, 0, 0, 0] [16, 16, 3, 3] ->
memref<16x16x3x3xf16>

    %out_0 = IERT.SubView %out [0, 0, 0, 0][1, 16, 16, 16] ->
memref<1x16x16x16xf16>
    %out_1 = IERT.SubView %out [0, 16, 0, 0][1, 16, 16, 16] ->
memref<1x16x16x16xf16>

    %buf0 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>
    %buf1 = memref.alloc() : memref<16x16x3x3xf16, [@CMX, 0]>
    %buf2 = memref.alloc() : memref<16x4xsi32, [@CMX, 0]>
    %buf3 = memref.alloc() : memref<1x16x16x16xf16, [@CMX, 0]>

    %buf4 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 1]>
    %buf5 = memref.alloc() : memref<16x16x3x3xf16, [@CMX, 1]>
    %buf6 = memref.alloc() : memref<16x4xsi32, [@CMX, 1]>
    %buf7 = memref.alloc() : memref<1x16x16x16xf16, [@CMX, 1]>

    %wt_0 = VPUIP.DeclareWeightsTable input(%buf0) weights(%buf1) output(%buf3)
-> memref<16x4xsi32>
    %wt_1 = VPUIP.DeclareWeightsTable input(%buf4) weights(%buf5) output(%buf7)
-> memref<16x4xsi32>

    %0 = IERT.Copy inputs(%in) outputs(%buf0)
    %1 = IERT.Copy inputs(%weights_0) outputs(%buf1)
    %2 = IERT.Copy inputs(%wt_0) outputs(%buf2)
    %3 = VPUIP.NCE.ClusterTask inputs(%0, %1, %2) outputs(%buf3) { cluster_id =
0 }
    %4 = IERT.Copy inputs(%3) outputs(%out_0)

    %5 = IERT.Copy inputs(%in) outputs(%buf4)
    %6 = IERT.Copy inputs(%weights_1) outputs(%buf5)
    %7 = IERT.Copy inputs(%wt_1) outputs(%buf6)
    %8 = VPUIP.NCE.ClusterTask inputs(%5, %6, %7) outputs(%buf7) { cluster_id =
1 }
    %9 = IERT.Copy inputs(%8) outputs(%out_1)

    %10 = IERT.ConcatSubView(%4, %9) as %out
    return %10
}
```

4.1.2.2 Async Regions

The task-level scheduling on that level is represented via MLIR standard Async Dialect. Individual run-time layer operations are wrapped inside `async.execute` operation regions. The pure view-like operations are copied to the inner region near to its users. The `async.execute` operations are connected via `!async.token` dependencies (analogue of virtual barriers). Initially the dependencies are based on data flow only, but can be extended with extra dependencies due to resource limitations. This transformation is done step-by-step by the following passes:

- WrapIntoAsyncRegions

- MoveViewOpsIntoAsyncRegions
- MoveWaitResultToAsyncBlockArgs

Example of the IR after this transformations:

```
func @main(%in : memref<1x16x32x32xf16>, %out : memref<1x32x16x16xf16>) {
    //
    // Declarations
    //

    %weights = const.Declare memref<32x16x3x3xf16> = !const.Content<...>

    %buf0 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>
    %buf1 = memref.alloc() : memref<16x16x3x3xf16, [@CMX, 0]>
    %buf2 = memref.alloc() : memref<16x4xsi32, [@CMX, 0]>
    %buf3 = memref.alloc() : memref<1x16x16x16xf16, [@CMX, 0]>

    %buf4 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 1]>
    %buf5 = memref.alloc() : memref<16x16x3x3xf16, [@CMX, 1]>
    %buf6 = memref.alloc() : memref<16x4xsi32, [@CMX, 1]>
    %buf7 = memref.alloc() : memref<1x16x16x16xf16, [@CMX, 1]>

    %wt_0 = VPUIP.DeclareWeightsTable input(%buf0) weights(%buf1) output(%buf3)
-> memref<16x4xsi32>
    %wt_1 = VPUIP.DeclareWeightsTable input(%buf4) weights(%buf5) output(%buf7)
-> memref<16x4xsi32>

    //
    // 1st tile
    //

    %t0, %f0 = async.execute attributes { IERT.Executor = @NNDMA } {
        %0 = IERT.Copy inputs(%in) outputs(%buf0)
        async.yeild %0
    }
    %t1, %f1 = async.execute attributes { IERT.Executor = @NNDMA } {
        %weights_0 = IERT.SubView %weights [0, 0, 0, 0] [16, 16, 3, 3] ->
memref<16x16x3x3xf16>
        %1 = IERT.Copy inputs(%weights_0) outputs(%buf1)
        async.yeild %1
    }
    %t2, %f2 = async.execute attributes { IERT.Executor = @NNDMA } {
        %2 = IERT.Copy inputs(%wt_0) outputs(%buf2)
        async.yeild %2
    }
    %t3, %f3 = async.execute [%t0, %t1, %t2] (%f0 as %0, %f1 as %1, %f2 as %2)
attributes { IERT.Executor = [@NCE_Cluster, 0] } {
        %3 = VPUIP.NCE.ClusterTask inputs(%0, %1, %2) outputs(%buf3) {
cluster_id = 0 }
        async.yeild %3
    }
    %t4, %f4 = async.execute [%t3] (%f3 as %3) attributes { IERT.Executor =
@NNDMA } {
        %out_0 = IERT.SubView %out [0, 0, 0, 0][1, 16, 16, 16] ->
memref<1x16x16x16xf16>
        %4 = IERT.Copy inputs(%3) outputs(%out_0)
        async.yeild %4
    }
```

```
    //
    // 2nd tile
    //

    %t5, %f5 = async.execute attributes { IERT.Executor = @NNDMA } {
        %5 = IERT.Copy inputs(%in) outputs(%buf4)
        async.yeild %5
    }
    %t6, %f6 = async.execute attributes { IERT.Executor = @NNDMA } {
        %weights_1 = IERT.SubView %weights [16, 0, 0, 0] [16, 16, 3, 3] ->
memref<16x16x3x3xf16>
        %6 = IERT.Copy inputs(%weights_1) outputs(%buf5)
        async.yeild %6
    }
    %t7, %f7 = async.execute attributes { IERT.Executor = @NNDMA } {
        %7 = IERT.Copy inputs(%wt_1) outputs(%buf6)
        async.yeild %7
    }
    %t8, %f8 = async.execute [%t5, %t6, %t7] (%f5 as %5, %f6 as %6, %f7 as %7)
attributes { IERT.Executor = [@NCE_Cluster, 1] } {
        %8 = VPUIP.NCE.ClusterTask inputs(%5, %6, %7) outputs(%buf7) {
cluster_id = 1 }
        async.yeild %8
    }
    %t9, %f9 = async.execute [%t8] (%f8 as %8) attributes { IERT.Executor =
@NNDMA } {
        %out_1 = IERT.SubView %out [0, 16, 0, 0][1, 16, 16, 16] ->
memref<1x16x16x16xf16>
        %9 = IERT.Copy inputs(%8) outputs(%out_1)
        async.yeild %9
    }

    %4 = async.await %f4
    %9 = async.await %f9
    %10 = IERT.ConcatSubView(%4, %9) as %out
    return %10
}
```

4.1.2.3 Generic Optimization

After the introduction of the asynchronous regions the following generic optimizations are performed:

- `OptimizeAsyncDeps`

This optimization tries to remove extra explicit `!async.token` based dependencies, if they are represented implicitly (as a result of transitive dependencies).

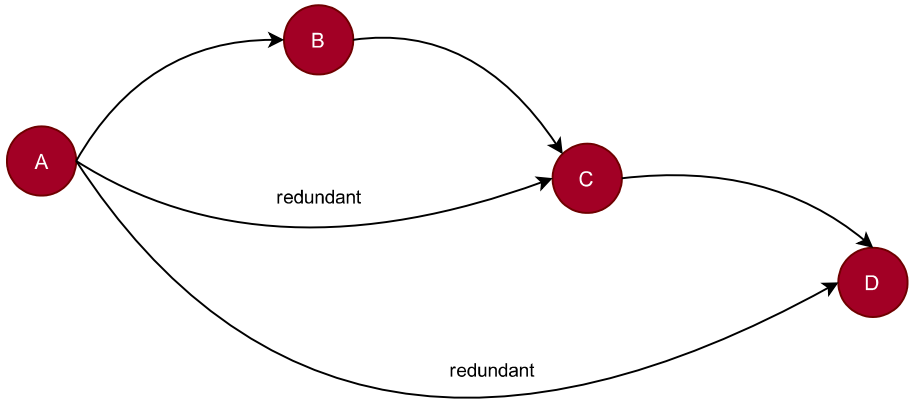

Figure: Before the Pass

The pass will convert such dependencies graph into:

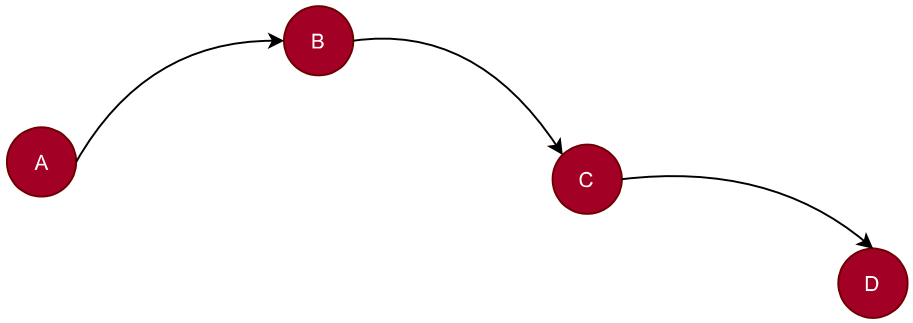

Figure: Before the Pass

4.1.2.4 Feasible Allocation (NNCMX)

Next step on that level is a CMX allocation using feasible memory scheduler. It performs full replacing of dynamic NNCMX `memref.alloc()` operations with `IERT.StaticAlloc` and fixed offset. The pass on that level can introduce extra dependencies between `async.execute` operations due to memory resource limitations.

```
// Prior to the static memory allocation
%buf0 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>
%buf1 = memref.alloc() : memref<16x16x3x3xf16, [@CMX, 0]>
%buf2 = memref.alloc() : memref<16x4xsi32, [@CMX, 0]>
%buf3 = memref.alloc() : memref<1x16x16x16xf16, [@CMX, 0]>

// After
%buf0 = IERT.StaticAlloc <0> -> memref<1x16x32x32xf16, [@CMX, 0]>
%buf1 = IERT.StaticAlloc <1024> -> memref<16x16x3x3xf16, [@CMX, 0]>
%buf2 = IERT.StaticAlloc <2048> -> memref<16x4xsi32, [@CMX, 0]>
%buf3 = IERT.StaticAlloc <4096> -> memref<1x16x16x16xf16, [@CMX, 0]>
```

The feasible memory scheduler was ported from MCM, details off the allocator can be found MCM: Scheduling Assigning Memory Addresses

Briefly, the feasible allocator:

1. Creates an intial schedule using feasible memory scheduler
   - The `async.execute` operations are categorized into two groups:
     - Data operations: operations moving data to NNCMX.
     - Compute operations: all other operations.
   - This is required to prevent scheduling only data operations (filling CMX) and not being able to schedule the corresponding compute operation which frees CMX space. To ensure CMX availability data operations will only be scheduled when required by the corresponding compute operation.
   - Following this categorization, using a topological iteration of the graph based on dependencies from the `AsyncDepsInfo` class, the compute operations from input to output are allocated to NNCMX along with their dependencies using the `LinearScan` class.
     - An operation can have three states
       - ACTIVE: currently in "execution" and scheduling.

        - CONSUMED: used by consumers and no longer needed.
        - SPILLED: due to fragmentation, op moved back to DDR.
    - The following tables and lists are used to produce a schedule:
        - operation indegree table: number of incoming edges per op
        - opertaion outdegree table: number of outgoing edges per op
        - ready data list: data ops with 0 indegree
        - ready compute list: compute ops with 0 indegree
        - ready active compute list: compute ops with ACTIVE input.
    - Initially input which is always considered a compute operation will be ready (0 indegree) and thus scheduled at time 1, with NNCMX memory allocated.
    - The tables will be updated accordingly reducing consumer in-degree, making consumers of input ACTIVE and adding them to the ready lists. Note: intermediate data ops are propagated.
    - This will result in new ready compute operations which will be checked if they fit in NNCMX and if so scheduled along with their dependencies.
    - If more than one compute operation fits in NNCMX at a particular time, these operations will be scheduled in parallel.
    - If no operation fits in NNCMX at a particular time, fragmentation occured and a dynamic spill is required.
  - The result is a schedule: list of operations executed at a particular time stamp (multiple operations can execute at the same time stamp), along with the resoures used by the operation.
2. Optimizes spills
  - Future feature
  - Remove redundant spill operations
3. Re-orders the IR
  - Update all `AsyncExecOp` position in the block so that their order is aligned with order generated by the feasible memory scheduler. All operations will appear in non-descending order of start time. Such reordering is needed as execution order has more constraints than topological order that IR is aligned with. Without such sorting insertion of token dependency might hit an error.
4. Inserts spill/pull-back DMAs
  - Future feature
  - The spill/pull-back are handled as `IERT.Copy` operation placement (wrapped into `async.execute` region). The new spill operation should be placed right after the buffer producer in the case of fragmentation OR if no operation can fit in NNCMX and some parallel buffers are alive the spill will occur before that operation, allowing it to fit in NNCMX. The pull-back operation should be placed right before its first user.

```
%buf0 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>

// Produces %buf0
%t0, %f0 = async.execute { ... }

// Current operation, not enough memory
%t1, %f1 = async.execute { ... }

// Consumes %buf0
%t2, %f2 = async.execute { ... }
```

    This IR will be replaced with:

```
%buf0 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>
%buf1 = memref.alloc() : memref<1x16x32x32xf16, @DDR>
%buf2 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>

// Produces %buf0
%t0, %f0 = async.execute { ... }

// Spill %buf0
%t01, %f01 = async.execute [%t0] (%f0 as %0) { IERT.Copy inputs(%0)
outputs(%buf1) }

// Add explicit dependency on %t01 for current operation
%t1, %f1 = async.execute [%t01] { ... }

// Pull-back %buf0
%t02, %f02 = async.execute [%t1] (%f01 as %0) { IERT.Copy inputs(%0)
outputs(%buf2) }

// Redirect user to pull-back buffer
%t2, %f2 = async.execute [%t02] (%f02 as 50) { ... }
```

5. Updates dependencies
   - Update all `AsyncExecOp` token dependencies so that resulting execution is aligned with order generated by the feasible memory scheduler.
   - The pass can extend `dependencies` operands of any `async.execute` with tokens produced by previous `async.execute` operation:

```
// Feasible memory scheduler address for %buf0: 0-32767
// Feasible memory scheduler address for %buf1: 32768-37376
// Note: no overlapping addresses
%buf0 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>
%buf1 = memref.alloc() : memref<16x16x3x3xf16, [@CMX, 0]>

// This task is an end-of-life for %buf0
%t0, %f0 = async.execute { ... }

// This task is a start-of-life for %buf1
%t1, %f1 = async.execute [%t] { ... }
```

Since initially the `%t1` task doesn't depend on `%t0` , they can be run in parallel. If the memory restrictions doesn't allow to place both `%buf0` and `%buf1` at the same time stamp, the pass can add explicit dependency between the tasks to ensure that `%buf0` memory is freed and safe for reuse prior to `%t1` execution.

```
// Feasible memory scheduler address for %buf0: 0-32768
// Feasible memory scheduler address for %buf1: 0-4608
// Note: addresses are overlapping
%buf0 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>

%buf1 = memref.alloc() : memref<16x16x3x3xf16, [@CMX, 0]>

%t0, %f0 = async.execute { ... }

// Add %t0 to dependencies list, preserve the original dependencies
%t1, %f1 = async.execute [%t, %t0] { ... }
```

6. Converts `memref.alloc()` to `IERT.StaticAlloc`
    - Replace dynamic NNCMX `memref.alloc()` operations with `IERT.StaticAlloc` and fixed offset.

4.1.2.5 Post NNCMX Allocation Optimization

After memory allocation for NNCMX buffers following optimizations are performed:

- `GroupAsyncExecuteOps`

This optimization tries to merge several `async.execute` operations into single one for special cases (like DMA) to reduce the amount of `async.execute` operations and `!async.token` values.

```
func @main(%in : memref<1x16x32x32xf16>, %out : memref<1x32x16x16xf16>) {
    //
    // Declarations
    //

    %weights = const.Declare memref<32x16x3x3xf16> = !const.Content<...>

    %buf0 = IERT.StaticAlloc <0> -> memref<1x16x32x32xf16, [@CMX, 0]>
    %buf1 = IERT.StaticAlloc <32768> -> memref<16x16x3x3xf16, [@CMX, 0]>
    %buf2 = IERT.StaticAlloc <37376> -> memref<16x4xsi32, [@CMX, 0]>
    %buf3 = IERT.StaticAlloc <37632> -> memref<1x16x16x16xf16, [@CMX, 0]>

    %buf4 = IERT.StaticAlloc <0> -> memref<1x16x32x32xf16, [@CMX, 1]>
    %buf5 = IERT.StaticAlloc <32768> -> memref<16x16x3x3xf16, [@CMX, 1]>
    %buf6 = IERT.StaticAlloc <37376> -> memref<16x4xsi32, [@CMX, 1]>
    %buf7 = IERT.StaticAlloc <37632> -> memref<1x16x16x16xf16, [@CMX, 1]>

    %wt_0 = VPUIP.DeclareWeightsTable input(%buf0) weights(%buf1) output(%buf3)
-> memref<16x4xsi32>
    %wt_1 = VPUIP.DeclareWeightsTable input(%buf4) weights(%buf5) output(%buf7)
-> memref<16x4xsi32>

    //
    // 1st tile
    //

    %t0, %f0:3 = async.execute attributes { IERT.Executor = @NNDMA } {
        %0 = IERT.Copy inputs(%in) outputs(%buf0)

        %weights_0 = IERT.SubView %weights [0, 0, 0, 0] [16, 16, 3, 3] ->
memref<16x16x3x3xf16>
        %1 = IERT.Copy inputs(%weights_0) outputs(%buf1)

        %2 = IERT.Copy inputs(%wt_0) outputs(%buf2)

        async.yeild %0, %1, %2
    }
    %t3, %f3 = async.execute [%t0] (%f0#0 as %0, %f0#1 as %1, %f0#2 as %2)
attributes { IERT.Executor = [@NCE_Cluster, 0] } {
        %3 = VPUIP.NCE.ClusterTask inputs(%0, %1, %2) outputs(%buf3) {
cluster_id = 0 }
        async.yeild %3
    }
    %t4, %f4 = async.execute [%t3] (%f3 as %3) attributes { IERT.Executor =
@NNDMA } {
        %out_0 = IERT.SubView %out [0, 0, 0, 0][1, 16, 16, 16] ->
```

```
memref<1x16x16x16xf16>
        %4 = IERT.Copy inputs(%3) outputs(%out_0)
        async.yeild %4
    }

    //
    // 2nd tile
    //

    %t5, %f5:3 = async.execute attributes { IERT.Executor = @NNDMA } {
        %5 = IERT.Copy inputs(%in) outputs(%buf4)

        %weights_1 = IERT.SubView %weights [16, 0, 0, 0] [16, 16, 3, 3] ->
memref<16x16x3x3xf16>
        %6 = IERT.Copy inputs(%weights_1) outputs(%buf5)

        %7 = IERT.Copy inputs(%wt_1) outputs(%buf6)

        async.yeild %5, %6, %7
    }
    %t8, %f8 = async.execute [%t5] (%f5#0 as %5, %f5#1 as %6, %f5#2 as %7)
attributes { IERT.Executor = [@NCE_Cluster, 1] } {
        %8 = VPUIP.NCE.ClusterTask inputs(%5, %6, %7) outputs(%buf7) {
cluster_id = 1 }
        async.yeild %8
    }
    %t9, %f9 = async.execute [%t8] (%f8 as %8) attributes { IERT.Executor =
@NNDMA } {
        %out_1 = IERT.SubView %out [0, 16, 0, 0][1, 16, 16, 16] ->
memref<1x16x16x16xf16>
        %9 = IERT.Copy inputs(%8) outputs(%out_1)
        async.yeild %9
    }

    %4 = async.await %f4
    %9 = async.await %f9
    %10 = IERT.ConcatSubView(%4, %9) as %out
    return %10
}
```

4.1.2.6 Memory Static Allocation (DDR)

Note: DDR allocation is simpler and follows LinearScan based memory allocation. Currently it forces a fully sequential order due to the Barrier Scheduler.

After NNCMX bufffer were allocated the static memory allocation is used to allocate DDR bufffers. It performs full replacing of dynamic DDR `memref.alloc()` operations with `IERT.StaticAlloc` and fixed offset.

```
// Prior to the static memory allocation
%buf0 = memref.alloc() : memref<1x16x32x32xf16, [@DDR, 0]>
%buf1 = memref.alloc() : memref<16x16x3x3xf16, [@DDR, 0]>
%buf2 = memref.alloc() : memref<16x4xsi32, [@DDR, 0]>
%buf3 = memref.alloc() : memref<1x16x16x16xf16, [@DDR, 0]>

// After
%buf0 = IERT.StaticAlloc <0> -> memref<1x16x32x32xf16, [@DDR, 0]>
%buf1 = IERT.StaticAlloc <1024> -> memref<16x16x3x3xf16, [@DDR, 0]>
%buf2 = IERT.StaticAlloc <2048> -> memref<16x4xsi32, [@DDR, 0]>
%buf3 = IERT.StaticAlloc <4096> -> memref<1x16x16x16xf16, [@DDR, 0]>
```

The pass can extract various information from the input IR. Taking the following IR:

```
func @main(%in : memref<1x16x32x32xf16>, %out : memref<1x32x16x16xf16>) {
    %weights = const.Declare memref<32x16x3x3xf16> = !const.Content<...>

    %buf0 = memref.alloc() : memref<1x16x32x32xf16, [@CMX, 0]>
    %buf1 = memref.alloc() : memref<16x16x3x3xf16, [@CMX, 0]>
    %buf2 = memref.alloc() : memref<16x4xsi32, [@CMX, 0]>
    %buf3 = memref.alloc() : memref<1x16x16x16xf16, [@CMX, 0]>

    // ...

    %t0, %f0:3 = async.execute attributes { IERT.Executor = @NNDMA } {
        %0 = IERT.Copy inputs(%in) outputs(%buf0)

        %weights_0 = IERT.SubView %weights [0, 0, 0, 0] [16, 16, 3, 3] ->
memref<16x16x3x3xf16>
        %1 = IERT.Copy inputs(%weights_0) outputs(%buf1)

        %2 = IERT.Copy inputs(%wt_0) outputs(%buf2)

        async.yeild %0, %1, %2
    }
    %t3, %f3 = async.execute [%t0] (%f0#0 as %0, %f0#1 as %1, %f0#2 as %2)
attributes { IERT.Executor = [@NCE_Cluster, 0] } {
        %3 = VPUIP.NCE.ClusterTask inputs(%0, %1, %2) outputs(%buf3) {
cluster_id = 0 }
        async.yeild %3
    }
    %t4, %f4 = async.execute [%t3] (%f3 as %3) attributes { IERT.Executor =
@NNDMA } {
        %out_0 = IERT.SubView %out [0, 0, 0, 0][1, 16, 16, 16] ->
memref<1x16x16x16xf16>
        %4 = IERT.Copy inputs(%3) outputs(%out_0)
        async.yeild %4
    }

    // ...
}
```

the pass can extract the following information:

- 4 buffers (`%buf0` , `%buf1` , `%buf2` , `%buf3`) to process.
- 3 tasks to schedule (`%t0` , `%t3` , `%t4`).
- Initial tasks dependencies : `%to -> %t3 -> %t4` .

- Use range for each buffer:
- `%buf0 : [%t0, %t3]`
- `%buf1 : [%t0, %t3]`
- `%buf2 : [%t0, %t3]`
- `%buf3 : [%t3, %t4]`

The information can be obtained as IR Analysis. The pass can take an `async.execute` operations in the IR order.

4.1.3 Level 1 : VPU RunTime

After the memory allocation and scheduling finalization on previous level the IR is lowered to VPU specific run-time representation. `async.execute` operations are converted to `VPURT.Task` and `!async.token` values are converted to virtual barriers.

```
func @main(%in : memref<1x16x32x32xf16>, %out : memref<1x32x16x16xf16>) {
    %weights_0 = const.Declare memref<16x16x3x3xf16> = !const.Content<...>
    %weights_1 = const.Declare memref<16x16x3x3xf16> = !const.Content<...>

    %wt_0 = const.Declare memref<16x4xsi32> = !const.Content<...>
    %wt_1 = const.Declare memref<16x4xsi32> = !const.Content<...>

    %buf0 = VPURT.DeclareTensor "CMX"[0] <0> : memref<1x16x32x32xf16, [@CMX, 0]>
    %buf1 = VPURT.DeclareTensor "CMX"[0] <1024> : memref<16x16x3x3xf16, [@CMX,
0]>
    %buf2 = VPURT.DeclareTensor "CMX"[0] <2048> : memref<16x4xsi32, [@CMX, 0]>
    %buf3 = VPURT.DeclareTensor "CMX"[0] <4096> : memref<1x16x16x16xf16, [@CMX,
0]>

    %out_0 = VPURT.DeclareTensor "ProgrammableOutput"[0] <0> :
memref<1x16x16x16xf16>
    %out_1 = VPURT.DeclareTensor "ProgrammableOutput"[0] <2048> :
memref<1x16x16x16xf16>

    %b0 = VPURT.DeclareVirtualBarrier
    %b1 = VPURT.DeclareVirtualBarrier
    %b2 = VPURT.DeclareVirtualBarrier

    VPURT.TasksList for @NNDMA {
        VPURT.Task posts(%b0) {
            VPUIP.DMA inputs(%in) outputs(%buf0)
        }
        VPURT.Task posts(%b0) {
            VPUIP.DMA inputs(%weights_0) outputs(%buf1)
        }
        VPURT.Task posts(%b0) {
            VPUIP.DMA inputs(%wt_0) outputs(%buf2)
        }
    }

    VPURT.TasksList for [@NCE_Cluster, 0] {
        VPURT.Task waits(%b0) posts(%b1) {
            VPUIP.NCE.ClusterTask inputs(%buf0, %buf1, %buf2) outputs(%buf3)
        }
    }

    VPURT.TasksList for @NNDMA {
        VPURT.Task waits(%b1) posts(%b2) {
            VPUIP.DMA inputs(%buf3) outputs(%out_0)
        }
    }

    VPURT.Leon.Wait (%b2)
    return
}
```

The final part of the scheduler is performed on that level - static barrier assignment. The goal of this part is to replace virtual barriers with physical barriers and assign IDs for them.

```
func @main(%in : memref<1x16x32x32xf16>, %out : memref<1x32x16x16xf16>) {
    %weights_0 = const.Declare memref<16x16x3x3xf16> = !const.Content<...>
    %weights_1 = const.Declare memref<16x16x3x3xf16> = !const.Content<...>

    %wt_0 = const.Declare memref<16x4xsi32> = !const.Content<...>
    %wt_1 = const.Declare memref<16x4xsi32> = !const.Content<...>

    %buf0 = VPURT.DeclareTensor "CMX"[0] <0> : memref<1x16x32x32xf16, [@CMX, 0]>
    %buf1 = VPURT.DeclareTensor "CMX"[0] <1024> : memref<16x16x3x3xf16, [@CMX,
0]>
    %buf2 = VPURT.DeclareTensor "CMX"[0] <2048> : memref<16x4xsi32, [@CMX, 0]>
    %buf3 = VPURT.DeclareTensor "CMX"[0] <4096> : memref<1x16x16x16xf16, [@CMX,
0]>

    %out_0 = VPURT.DeclareTensor "ProgrammableOutput"[0] <0> :
memref<1x16x16x16xf16>
    %out_1 = VPURT.DeclareTensor "ProgrammableOutput"[0] <2048> :
memref<1x16x16x16xf16>

    %b0 = VPURT.ConfigureBarrier <0>
    %b1 = VPURT.ConfigureBarrier <1>
    %b2 = VPURT.ConfigureBarrier <2>

    VPURT.TasksList for @NNDMA {
        VPURT.Task posts(%b0) {
            VPUIP.DMA inputs(%in) outputs(%buf0)
        }
        VPURT.Task posts(%b0) {
            VPUIP.DMA inputs(%weights_0) outputs(%buf1)
        }
        VPURT.Task posts(%b0) {
            VPUIP.DMA inputs(%wt_0) outputs(%buf2)
        }
    }

    VPURT.TasksList for [@NCE_Cluster, 0] {
        VPURT.Task waits(%b0) posts(%b1) {
            VPUIP.NCE.ClusterTask inputs(%buf0, %buf1, %buf2) outputs(%buf3)
        }
    }

    VPURT.TasksList for @NNDMA {
        VPURT.Task waits(%b1) posts(%b2) {
            VPUIP.DMA inputs(%buf3) outputs(%out_0)
        }
    }

    VPURT.Leon.Wait (%b2)
    return
}
```

4.2 Software Layers

This document describes the proposed design for generic software layers support in **VPUX NN compiler**. The design is supposed to be used for MTL/LNL activation SHAVE kernels, but can also be extended to KMB UPA SHAVEs if needed.

4.2.1 Terminology

- **Kernel** - the entity, which holds the machine code for the kernel and its RO data. In general, it is an ELF file in memory, with `.text` section. Run-time will run the entry point function from `.text` section during inference.
- **Kernel Invocation** - particular call of the **Kernel** with provided arguments and optional RW data.
- **Kernel Task** - entity for scheduling by the runtime. Includes **Kernel Invocation**, barriers configuration and tile selection.
- **Runtime Kernel** - special management **Kernel**, which performs scheduling of the **Kernel Task**. In particular, it takes the tasks from FIFO, waits its barriers, launch its with provided arguments, updates producer barriers and perform other management work. This kernel is also provided by the **VPUX NN compiler** and is stored in the VPU blob.

4.2.2 Memory Configuration

The SW kernels uses windows-based virtual addressing, which means that all pointers inside the **Kernel** belongs to the one of the following window:

- `WIN_D` - **Kernel** code + RO data
- `WIN_E` - **Kernel Invocation** arguments + RW data
- `WIN_F` - CMX slice per SHAVE tile, it will hold the input/output buffers.

Thus, compiler can generate relocatable code and arguments list, fully ready for execution. All addresses will be represented as `window_start_address` (for example, `0x1F000000`) plus relative offset.

In addition, the compiler will provide SHAVE stack configuration for the particular inference. It will include per-tile stack sizes for each SHAVEs.

4.2.3 Kernel Arguments Contract

The format of particular **Kernel** arguments must be fixed between compiler and the kernel and represented as a contract between them. It includes the order of arguments as well as their internal structure for complex cases.

For example, there should be a single definition of the buffer argument:

```
struct __attribute__((packed)) MemRefData {
    uint32_t dataAddr;  // Can't use pointers, since they have platform-dependent
size.
                        // Will be located in WIN_F.

    uint32_t isStatic;  // Boolean flag to indicate static shape vs dynamic
shape.

    uint32_t numDims;
    uint32_t dimsAddr;      // Pointer to the buffer with dimensions (int32_t[]).
    uint32_t stridesAddr;   // Pointer to the buffer with strides in bits
(int64_t[]).
                            // Will be located in WIN_E (static case) or in WIN_F
(dynamic case).
                            // The kernel should infer output dims/strides and
write them only in dynamic case.

    uint32_t dataType;      // An enum, which should be aligned between kernels
and the compiler.
    uint64_t dimsOrder;     // Packed permutation array.
};
```

The compiler will create those structures for all buffer arguments of the **Kernel Invocation**.

Each particular **Kernel** will have it's own parameters structure corresponding to original NN layer:

```
struct __attribute__((packed)) ClampParams {
    MemRefData input;
    MemRefData output;
    f16 minVal;
    f16 maxVal;
};
```

With this fixed definition of the parameters, the **Kernel** itself can be implemented as C++ function, which takes raw pointer to the arguments list and casts it to its parameter structure. The compiler will guarantee the match of the provided arguments buffer with the contract.

```
__attribute__((section(".arg.data")))
__attribute__((used))
ClampParams arg_buffer;

void clamp(uint32_t argsAddr, uint32_t argsSize) {
    assert(argsSize == sizeof(ClampParams)); // Just for debugging, should be
removed from Release code.

    ClampParams* params = reinterpret_cast<ClampParams*>(static_cast<uintptr_t>
(argsAddr));

    for (...) {
        ...
    }
}
```

4.2.4 Kernel Representation in the Compiler

The SW kernel will be represented in the Compiler in several parts:

1. Function declaration, which holds the arguments contract between compiler and the **Kernel**.
2. The information about location of the **Kernel** code or the implementation of the auto-generated **Kernel** function.
3. `VPUIP.SW.Kernel` operation, which will hold the reference to the **Kernel** function and which will provide common operation interfaces.
4. `VPUIP.SW.Kernel.run` operation, which will be used in the `VPUIP.SW.Kernel` inner region and which will perform arguments mapping.
5. `VPURT.Task` operation, which will hold common scheduling information (barriers and executor).

Summarizing the above, the SW kernel will be represented in the following way at the compiler final stages:

```
// Top-parent module, which holds the entire network/blob.
module @network {

// Sub-module, which holds SW kernel declarations and optional implementations.
// Used to group those declarations for faster access.
module @VPU.SW {
    // The declaration should match C++ params structure in decomposed form.
    // `memref` will be translated to `MemRefData`, while raw scalars will be
translated as is.
    func private @builtin_clamp(%input : memref<*xf16,>, %output : memref<*xf16>,
%minVal : f16, %maxVal : f16)
        attributes { VPU.kernel_code = <place where to get the code> }
}

// The network main entry point, which will be translated to MappedInference or
it's analogue.
func @main(...) {
    // Static barriers declarations.
    %b0 = VPURT.ConfigurePhysicalBarrier <0>
    %b1 = VPURT.ConfigurePhysicalBarrier <1>

    // Task list for DMA engine.
    VPURT.TaskList @DMA {
        // The DMA task, which copies the input tile from DDR to CMX prior to SW
kernel invocation.
        VPURT.Task updates(%b0) {
            VPUIP.DMA inputs(%in_tile_0_ddr) outputs(%in_tile_0_cmx)
        }

        // The DMA task, which copies the output tile from CMX to back to DDR
after the SW kernel invocation.
        VPURT.Task waits(%b1) {
            VPUIP.DMA inputs(%out_tile_0_cmx) outputs(%out_tile_0_ddr)
        }
    }

    // Task list for Activation SHAVEs.
    VPURT.TaskList @ACT_SHAVE {
        // Particular Kernel invocation.
        VPURT.Task waits(%b0) updates(%b1) {
            // Genetic Kernel information for the scheduler.
            VPUIP.SW.Kernel
                    @VPU.SW.builtin_clamp               // The reference to the
```

```
Kernel function.
                    on tile 0                           // The tile index to
execute on.
                    inputs(%in_tile_cmx_0 as %arg0)     // Inputs/outputs buffers
for generic operation interface
                    outputs(%out_tile_cmx_0 as %arg1)   // and their mapping to
inner region.
            {
                // Inner region, isolated from above, which holds the information
about arguments mapping.

                // We can use constant scalars/arrays definitions here.
                %minVal = constant 0.0 : f16
                %maxVal = constant 6.0 : f16

                // The arguments mapping, the order must match the kernel
parameter structure.
                VPUIP.SW.Kernel.run(%arg0, %arg1, %minVal, %maxVal)
            }
        }
    }
}

}
```

The **Runtime Kernel** should also be represented as function declaration with information about the machine code location. It should be referenced by `VPURT.Graph` meta-operation, which also should store the SHAVE stacks configuration.

```
module @network {

module @VPU.SW {
    func private @runtime()
        attributes { VPU.kernel_code = <place where to get the code> }
}

VPURT.Graph {
    act_shaves : {
        runtime: @VPU.SW.runtime,
        stack_configuration: [
            128000, // Size in bytes for the SHAVEs in the first tile.
            128000  // Size in bytes for the SHAVEs in the second tile.
        ]
    }
}

}
```

4.2.5 Kernel Serialization

Note: This part is bound to current FlatBuffer-based VPU blob format. It should be adapted for other formats like ELF.

The VPU Blob uses the following structures to hold the information about the SW kernels:

- `KernelData` - raw binary buffer, which holds **Kernel** ELF file and **Kernel Invocation** arguments buffer.

- `KernelDataReference` - sub-view inside `KernelData`, which refers to particular part (ELF `.text` section, for example).
- `ActKernel` - structure, which holds **Kernel** information:
- `KernelDataReference` to `.text` section.
- Entry point offset inside the `.text` section. **TODO:** should be always assume `0` here?
- `ActKernelInvocation` - structure, which holds **Kernel Invocation** information:
- Reference to the corresponding `ActKernel`.
- `KernelDataReference` to the arguments buffer.
- `KernelDataReference` to the SW `.data` section.
- Tile index to launch the kernel on.
- `ActKernelTask` - holds `ActKernelInvocation` and scheduling related information (like barriers configuration).
- `ActKernelRuntime` - holds the information about **Runtime Kernel** and generic run-time configuration:
- Reference to the corresponding `ActKernel`.
- SHAVE stack configuration.

The serialization should be implemented in the following way:

1. The backend should enumerate all functions inside inner `@VPU.SW` module and get their final machine code (`.text` section). It can take them from file or compile on-the-fly. The backend should serialize those code into `KernelData` structure and add to the common `kernelData` section inside VPU blob. It should also create single `ActKernel` descriptor per each function and store it in internal map for future access.
2. For each `VPUIP.SW.Kernel` in the main entry point function body the backend should take serialized `ActKernel` from its inner map, pack the parameters taken from inner `VPUIP.SW.Kernel.run` operation into single buffer (with windows-based virtual addressing) and create `ActKernelInvocation` structure from them. On top of this structure the generic `ActKernelTask` will be created with corresponding barriers configuration description.

4.2.6 Issues with Current DPU Tasks Representation

In the current scheme the DPU tasks are represented as inner parts of single NCE task. This contradicts with run-time logic, since run-time uses NCE task variant as actual entity of scheduling.

Also in contrast to SW kernels, which uses per-invocation barrier configuration, DPU tasks use shared barriers configuration stored in common invariant part.

This might introduces stalls between DPUs and ACT SHAVEs in MTL/LNL cases.

For example, for the tensor in the CMX memory (either the full tensor or a slice of larger buffer from DDR) the compiler will generate per-DPU split of the workloads. After DPU finishes one workload, the ACT SHAVE can immediately start post-processing kernel on it. But it is not possible right now, since ACT SHAVE Kernel Task have to wait common barrier for all DPUs, so it won't start until all workloads are finished.

4.2.7 Changes for Schema and Run-time

- Merge common arguments into per-invocation arguments, it's up to compiler to optimize its memory footprint.
- Make an argument list just a "black box" binary buffer from run-time perspective.
- Provide just a per-tile stack size without content. No need for per-SHAVE configuration, since compiler is not able to select particular SHAVE for execution, only tile.

- Use single invocation per task - this is closer to run-time scheme of scheduling.
- Use per-DPU task (per-variant) barriers configuration to match SW kernels behavior and avoid stalls between them.

4.3 Custom Layers

TBD: finalize design.

Custom layers (like OCL) can be represented on **IE Dialect** as a call to an external Function. This function will hold all requirements (precisions, layouts, memory spaces, etc.). During the lowering to **VPUIP Dialect** this Function call will be replaced by the corresponding custom layer task.

4.3.1 Custom Binary Layer

The **Custom Binary Layer** represents pre-compiled custom layer (eg. user-provided) on all Dialects. The pre-compiled code in binary from is provided as separate constant operand. The Operation for such kind of Layer holds the type of the layer (for example, `OCL`, `CPP`) to be used in lowering passes. Also it holds various attributes to handle layouts/precision/strides requirements in generic passes. In addition to the above attributes extra specific attributes can be attached to hold lower Dialect specific information.

```
#NHWC = affine_map<(n, c, h, w) -> (n, h, w, c)>
#OYXI = affine_map<(o, i, y, x) -> (o, y, x, i)>

%bin_code = IE.Constant tensor<10240xui8> = dense<...>

%output = IE.CustomBinaryLayer VPUIP.OCL from %bin_code
    (%input : tensor<1x16x100x100xf16>, %weights : tensor<32x16x3x3xf16>)
    -> tensor<1x32x100x100xf16>
    {
        input_layouts = [#NHWC, #OYXI],
        output_layouts = [#NHWC],
        VPUIP.max_num_shaves = 8
    }
```

In the lowering pass from **IE Dialect** to **IERT Dialect** the `tensor` types will be bufferized to `memref`, allocation will be added and layouts restriction will be used to adjust other operations. Same passes will be used for allocation and reordering optimizations as for known Layer Operations.

```
#NHWC = affine_map<(n, c, h, w) -> (n, h, w, c)>
#OYXI = affine_map<(o, i, y, x) -> (o, y, x, i)>

%bin_code = IERT.Constant memref<10240xui8> = dense<...>

%input_NHWC = alloc : memref<1x16x100x100xf16, #NHWC>
IERT.Reorder(%input, %input_NHWC)
dealloc %input

%output_NHWC = alloc : memref<1x32x100x100xf16, #NHWC>
IERT.CustomBinaryLayer VPUIP.OCL from %bin_code
    (%input_NHWC, %weights_OYXI, %output_NHWC)
    {
        VPUIP.max_num_shaves = 8
    }
dealloc %input_NHWC

%output = alloc : memref<1x32x100x100xf16>
IERT.Reorder(%output_NHWC, %output)
dealloc %output_NHWC
```

At the end the `IERT.CustomBinaryLayer` Operation will be lowered to corresponding Operation from **VPUIP Dialect** based on its type.

4.4 Dynamic Shapes Support

TBD: finalize design.

The **VPUX NN Compiler** will support dynamic shapes during the whole compilation stack, using different representation ways on different abstraction levels.

It will allow to use 2 modes for dynamic shapes processing:

1. The actual result shape is calculated inside some runtime kernel (**preferable**).
2. The actual result shape is calculated with separate code.

Initial IR in **IE Dialect** will use Tensor types with dynamic shapes as is:

```
// Input batch, height, width are dynamic; channels are static.
// Output is all dynamic.
// Rank is static.
func @main(%input: tensor<?x3x?x?xf16>) -> tensor<?x?xf16> {
    // This layer has some formula to calculate output shape from input and
attributes.
    // The runtime implementation of the layer will calculate the output shape
internally.
    %temp0 = IE.Layer1(%input) : tensor<?x3x?x?xf16> -> tensor<?x16x?x?xf16>

    // This is a simple element-wise layer, output shape is equal to input shape.
    %temp1 = IE.SimpleLayer(%temp0) : tensor<?x16x?x?xf16> -> tensor<?x16x?x?xf16>

    // This is a custom layer, it has provided metadata for output shape
calaculation,
    // but it must be done separately.
    %temp2 = IE.Custom(%temp1) : tensor<?x16x?x?xf16> -> tensor<?x?xf16>

    return %2
}
```

Separate Pass will add explicit notion of shapes into the IR. The shapes will be represented as separate Values (Operation arguments and results). All Operations with dynamic inputs will get additional arguments with actual shape per each dynamic input. Operation that calculates result shape internally (mode 1) will return it as additional result (implicit tuple). Operation that uses separate code for result shape computation (mode 2) will get it as one more additional argument. The shape computation code will be inserted prior to this Operation.

```
func @main(%input: tensor<?x3x?x?xf16>) -> tensor<?x?xf16>, !shape.type {
    // Extract actual shape for network input
    %input_shape = shape.shape_of %input : tensor<?x3x?x?xf16> -> !shape.shape

    // Layer1 takes the shape of its argument as additional parameter.
    // Layer1 will compute the output shape internally and it returns it as
additional result (implicit tuple).
    %temp0:2 = IE.Layer1(%input)[%input_shape] : tensor<?x3x?x?xf16>, !shape.shape
-> tensor<?x16x?x?xf16>, !shape.type

    // SimpleLayer takes the shape of its argument as additional parameter.
    // Since for SimpleLayer output shape == input shape, it doesn't add additonal
value for it,
    // instead it notifies the Pass to reuse existing Shape Value.
    %temp1 = IE.SimpleLayer(%temp0#0)[%temp0#1] : tensor<?x16x?x?xf16>,
!shape.shape -> tensor<?x16x?x?xf16>

    // %temp0#1 (the shape of %temp0 tensor) is used as shape value for %temp1.
    // Custom layer has separate code for output shape calculation, so it is called
prior to the Custom Operation.
    %temp2_shape = call @calc_custom_shape(%temp0#1) : !shape.shape -> !shape.shape

    // Custom takes the shape of its argument as additional parameter.
    // Custom takes the shape of its result as additional parameter (since it is
calculated separately).
    %temp2 = IE.Custom(%temp1#0)[%temp0#1][%temp2_shape] : tensor<?x16x?x?xf16>,
!shape.shape -> tensor<?x?xf16>

    return %temp2, %temp2_shape
}
```

Next Pass will calculate upper bound for the dynamic shapes to allow static memory allocation.

```
func @main(%input: tensor<100x3x300x300xf16>) -> tensor<100x1000xf16>, !shape.type
{
    %input_shape = IE.ActualShapeOf %input : tensor<100x3x300x300xf16> ->
!shape.shape

    %temp0:2 = IE.Layer1(%input)[%input_shape] : tensor<100x3x300x300xf16>,
!shape.shape -> tensor<100x16x200x200xf16>, !shape.type

    %temp1 = IE.SimpleLayer(%temp0#0)[%temp0#1] : tensor<100x16x200x200xf16>,
!shape.shape -> tensor<100x16x200x200xf16>

    %temp2_shape = call @calc_custom_shape(%temp0#1) : !shape.shape -> !shape.shape

    %temp2 = IE.Custom(%temp1#0)[%temp0#1][%temp2_shape] :
tensor<100x16x200x200xf16>, !shape.shape -> tensor<100x1000xf16>

    return %temp2, %temp2_shape
}
```

At the final stage the shape Values will be lowered (bufferized) into `MemRef` types in the same way as tensors.

```
func @main(%input: memref<100x3x300x300xf16>, %output:memref<100x1000xf16>,
%output_shape:memref<2xui32>) {
    // Extract actual shape for network input.
    // The memory for this Shape Value is allocated by caller (InferenceManager).
    %input_shape = IERT.ActualShapeOf %input : memref<100x3x300x300xf16> ->
memref<4xui32>

    // Allocate memory for Layer1 result
    %temp0 = alloc : memref<100x16x200x200xf16>

    // Allocate memory for Layer1 result shape
    %temp0_shape = alloc : memref<4xui32>

    // On this step Operation takes both inputs and outputs as arguments, since
they are allocated outside.
    IERT.Layer1(%input, %temp0)[%input_shape, %temp0_shape]

    // Allocate memory for SimpleLayer result.
    %temp1 = alloc : memref<100x16x200x200xf16>

    IE.SimpleLayer(%temp0, %temp1)[%temp0_shape]

    dealloc %temp0

    call @calc_custom_shape(%temp0_shape, %output_shape)

    // Custom Operation will write directly to %output.
    IE.Custom(%temp1, %output)[%temp0_shape, %output_shape]

    dealloc %temp0_shape

    return
}
```

After that alloc/dealloc pairs can be replaced with statically allocated memory offsets, using DDR for Shapes.

5 Dialects Details

5.1 OpInterface definitions

5.1.1 DotInterface (`DotInterface`)

Base interface for Dot graph generation #### Methods: ##### `getNodeColor`

```
vpux::DotNodeColor getNodeColor();
```

Get node color NOTE: This method *must* be implemented by the user.

5.1.1.0.1 `printAttributes`

```
std::string printAttributes();
```

If non empty overrides the default attributes in the dot graph NOTE: This method *must* be implemented by the user.

5.1.2 MultiViewOpInterface (`MultiViewOpInterface`)

An extended version of view-like operation. It allows to define several resulting views in single operation.

5.1.2.1 Methods:

5.1.2.1.1 `getViewSource`

```
mlir::Value getViewSource(ptrdiff_t resultInd);
```

The source buffer from which the corresponding view is created, NULL if the result is not a view NOTE: This method *must* be implemented by the user.

5.2 'const' Dialect

Const Dialect The **VPUX NN Compiler** uses lazy constant folding approach to reduce memory footprint for large constant values (like dense tensors).

The **Const Dialect** provides utilities for that lazy constant folding support.

It defines a special attribute `Const::ContentAttr` to hold original constant data and the transformations applied to it. The `Const::ContentAttr` provides an API to apply the transformations on-the-fly, when accessing the data.

The **Const Dialect** supports the following transformations:

- Precision conversion
- Quantization types casting from raw storage
- Dequantization
- Reshape
- Reorder
- Padding
- SubView

The transformations are stored as separate attributes, which implemented specific Attribute Interface. The interface allows to extend the set of transformations outside of the **Const Dialect** implementation.

Initial non-transformed constant should be created via `Const::DeclareOp` and `Const::ContentAttr`:

```
mlir::ElementsAttr baseValue = ...;
Const::ConstOp constOp = builder.create<Const::DeclareOp>(loc, resultType,
Const::ContentAttr::get(baseValue));
```

In assembly code it will be represented as:

```
%0 = const.Declare tensor<1x2x3x4xf32> = #const.Content<dense<...> :
tensor<1x2x3x4xf32>>>
```

The content is accessed from `Const::DeclareOp` via `content()` method. The method returns special `Const::Content` object, which allows to access underlying values as range. The `Const::Content` object must be stored in separate variable, since it might contain temporal buffer, which will be freed at object destruction.

```
Const::DeclareOp constOp = ...;
Const::Content content = constOp.content(); // This call will apply all
transformations
const auto valsRange = content.getValues<float>(); // Access the values via range-
like class
for (auto val : valsRange) { ... }
```

Note: the `getValues` allows to specify desired type for elements and will perform conversion from underlying storage type on-the-fly.

To perform tranformations on constant data, the new `Const::ContentAttr` must be created on top of existed with specified transformation:

```
Const::DeclareOp origConstOp = ...;
Const::ConstContentAttr origConstAttr = origConstOp.contentAttr();
Const::ConstContentAttr newConstAttr =
origConstAttr.convertElemType(mlir::Float16Type::get(ctx));
Const::ConstOp newConstOp = builder.create<Const::DeclareOp>(loc,
newConstAttr.getType(), newConstAttr);
```

In assembly code it will be represented as:

```
%0 = const.Declare tensor<1x2x3x4xf16> = #const.Content<dense<...> :
tensor<1x2x3x4xf32>, [#const.ConvertElemType<f16>]>
```

The `Const::ConstContentAttr` might hold a list of transformations, they will be applied in the order:

```
%0 = const.Declare memref<1x2x3x4xf16, #NHWC, #strides> =
    #const.Content<
        dense<...> : tensor<2x3x4xf32>,
        [
            #const.ConvertElemType<f16>,
            #const.Reshape<[1, 2, 3, 4]>,
            #const.Reorder<#NHWC>
        ]
    >
```

Note: the tensor-related type attributes (shape, layout, element type) for the `Const::DeclareOp` result value and for `Const::ConstContentAttr` final infered type must match.

The quantized constant is represented via special `#const.QuantCast` transformation, which casts raw integer storage representation to quantized type with quantization parameters:

```
%0 = const.Declare tensor<1x16x1x1x!quant.uniform<u8:f16, ...>> =
    #const.Content<
        dense<...> : tensor<1x16x1x1xui8>,
        [
            #const.QuantCast<!quant.uniform<u8:f16, ...>>
        ]
    >
```

5.2.1 AttributeInterface definitions

5.2.1.1 TransformAttrInterface (`Const_TransformAttrInterface`)

The interface for Attributes, which holds information about lazy constant folding operation.

5.2.1.1.1 Methods:

5.2.1.1.1.1 `inferOutputType`

```
mlir::ShapedType inferOutputType(mlir::ShapedType input);
```

Infer output type NOTE: This method *must* be implemented by the user.

5.2.1.1.1.2 `transform`

```
vpux::Const::Content transform(vpux::Const::Content&input);
```

Transform the constant content NOTE: This method *must* be implemented by the user.

[TOC]

5.2.2 Attribute definition

5.2.2.1 AddAttr

Add constant content

5.2.2.1.1 Parameters:

Parameter	C++ type	Description
bias	`mlir::FloatAttr`	

5.2.2.2 BroadcastAttr

Broadcast axis by value of constant content

5.2.2.2.1 Parameters:

Parameter	C++ type	Description
axis	`mlir::IntegerAttr`	
value	`mlir::IntegerAttr`	

5.2.2.3 ContentAttr

Lazy folded constant content

This attribute holds base constant and transformation applied to it. It provides and API to get transformed values on the fly.

5.2.2.3.1 Parameters:

Parameter	C++ type	Description
baseContent	`mlir::ElementsAttr`	
transformations	`mlir::ArrayAttr`	
finalType	`mlir::ShapedType`	

5.2.2.4 ConvertElemTypeAttr

Convert constant content element type

5.2.2.4.1 Parameters:

Parameter	C++ type	Description
elemType	`mlir::Type`	

5.2.2.5 DequantizeAttr

Dequantize constant content

5.2.2.6 PadWithZeroAttr

Pad constant content with zeros

5.2.2.6.1 Parameters:

Parameter	C++ type	Description
padBefore	`mlir::ArrayAttr`	
padAfter	`mlir::ArrayAttr`	

5.2.2.7 QuantCastAttr

Cast element type from raw integer to quantized type

5.2.2.7.1 Parameters:

Parameter	C++ type	Description
elemType	`mlir::quant::QuantizedType`	

5.2.2.8 ReorderAttr

Reorder constant content

5.2.2.8.1 Parameters:

Parameter	C++ type	Description
order	`mlir::AffineMapAttr`	

5.2.2.9 RescaleAttr

Rescale constant content

5.2.2.9.1 Parameters:

Parameter	C++ type	Description
scale	`mlir::FloatAttr`	

5.2.2.10 ReshapeAttr

Reshape constant content

5.2.2.10.1 Parameters:

Parameter	C++ type	Description
shape	`mlir::ArrayAttr`	

5.2.2.11 SubViewAttr

Extract subview from constant content

5.2.2.11.1 Parameters:

Parameter	C++ type	Description
offset	`mlir::ArrayAttr`	
shape	`mlir::ArrayAttr`	

5.2.3 Operation definition

5.2.3.1 `const.Declare` (vpux::Const::DeclareOp)

Constant tensor/buffer declaration

Syntax:

```
operation ::= `const.Declare` attr-dict type($output) `=` $content
```

This operation can perform extra lazy constant folding transformations for constant content.

5.2.3.1.1 Attributes:

Attribute	MLIR Type	Description
`content`	vpux::Const::ContentAttr	Lazy folded constant content

5.2.3.1.2 Results:

Result	Description
`output`	statically shaped tensor of any type values or statically shaped memref of any type values

5.3 'IE' Dialect

InferenceEngine IR Dialect The **IE Dialect** represents InferenceEngine/nGraph IR in terms of MLIR framework.

It has the following properties:

- Describes network topology without HW details (memory hierarchy, memory allocation, scheduling).
- Represents the latest nGraph opset and in addition some portion of legacy IE opset (for convenience).
- Works with MLIR Tensor Types as atomic Values (no memory effects), all operations are pure.
- Performs high level transformations/optimizations, that doesn't need low level details (memory buffers, layouts, scheduling).

Some of the layer operations in the **IE Dialect** defines Canonicalization hooks to simplify IR for further optimizations:

- Remove redundant Operations (same type `Reshape` / `Tile` , `Add` with 0, etc.).
- Apply Lazy Constant Folding.
- Replace Constant Values with Attributes (more linear graph).
- Fuse common patterns (for example, `Mul+Add => ScaleShift` , `Convolution+Bias`).
- Use more convinient Operations (for example, `MatMul => FullyConnected`).

Quantization parameters are stored as a part of tensor/buffer element type (`QuantizedType` from **Quant Dialect**).

The network topology (nGraph) is represented as a MLIR Function, which works with `tensor` types.

```
func @main(%input: tensor<1x1000xf32>) -> tensor<1x1000xf32> {
    %output = IE.SoftMax(%input) {axisInd = 1} : tensor<1x1000xf32> ->
tensor<1x1000xf32>
    return %output
}
```

The network inputs and outputs information (names, precision, layout) is held in separate Operation - `IE.CNNNetwork` .

```
IE.CNNNetwork
    entryPoint : @main
    inputsInfo : {
        IE.DataInfo "input" : memref<1x3x400x400xf32>
    }
    outputsInfo : {
        IE.DataInfo "output" : memref<1x1000xf32>
    }
```

5.3.1 OpInterface definitions

5.3.1.1 AlignedChannelsOpInterface (`IE_AlignedChannelsOpInterface`)

Interface for operations that require channel alignment ##### Methods: ###### `verifyChannels`

```
mlir::LogicalResult verifyChannels();
```

Verify channel alignement NOTE: This method *must* be implemented by the user.

5.3.1.1.0.1 `getChannelAlignment`

```
int64_t getChannelAlignment();
```

Get channel alignement factor in elements NOTE: This method *must* be implemented by the user.

5.3.1.2 LayerOpInterface (`IE_LayerOpInterface`)

Base interface for IE Layer Operation ##### Methods: ###### `getInputs`

```
mlir::OperandRange getInputs();
```

Get all layer input tensors NOTE: This method *must* be implemented by the user.

5.3.1.2.0.1 `getOutputs`

```
mlir::ResultRange getOutputs();
```

Get all layer output tensors NOTE: This method *must* be implemented by the user.

5.3.1.3 LayerWithPostOpInterface (`IE_LayerWithPostOpInterface`)

Interface for operations that support post-processing ##### Methods: ###### `getPostOp`

```
llvm::Optional<mlir::OperationName> getPostOp();
```

Get the post-processing operation NOTE: This method *must* be implemented by the user.

5.3.1.3.0.1 `getPostOpAttrs`

```
mlir::DictionaryAttr getPostOpAttrs();
```

Get the post-processing operation attributes NOTE: This method *must* be implemented by the user.

5.3.1.3.0.2 `setPostOp`

```
void setPostOp(mlir::Operation*postOp);
```

Set post-processing operation attribute NOTE: This method *must* be implemented by the user.

5.3.1.3.0.3 `isSupportedPostOp`

```
bool isSupportedPostOp(mlir::Operation*postOp);
```

Set post-processing operation attribute NOTE: This method *must* be implemented by the user.

5.3.1.4 LayoutInfoOpInterface (`IE_LayoutInfoOpInterface`)

Interface for operations to provide information about supported layout for inputs/outputs ##### Methods: ###### `inferLayoutInfo`

```
void inferLayoutInfo(vpux::IE::LayerLayoutInfo&info);
```

Infer supported Data Layouts from inputs to outputs or describe the supported combination NOTE: This method *must* be implemented by the user.

5.3.1.4.0.1 `getLayoutInfo`

```
vpux::IE::LayerLayoutInfo getLayoutInfo();
```

Get information about current layout for Layer inputs and outputs NOTE: This method *must* be implemented by the user.

[TOC]

5.3.2 Operation definition

5.3.2.1 `IE.Add` (vpux::IE::AddOp)

InferenceEngine Add layer

Syntax:

```
operation ::= `IE.Add` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.1.1 Attributes:

Attribute	MLIR Type	Description
`auto_broadcast`	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors
`post_op`	vpux::IE::PostOp	DictionaryAttr with field(s): 'name', 'attrs' (each field having its own constraints)

5.3.2.1.2 Operands:

Operand	Description
`input1`	ranked tensor of 16-bit float or 32-bit float values
`input2`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.1.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.2 `IE.AvgPool` (vpux::IE::AvgPoolOp)

InferenceEngine AvgPool layer

Syntax:

```
operation ::= `IE.AvgPool` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.2.1 Attributes:

Attribute	MLIR Type	Description
`kernel_size`	::mlir::ArrayAttr	64-bit integer array attribute
`strides`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_begin`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end`	::mlir::ArrayAttr	64-bit integer array attribute
`rounding_type`	vpux::IE::RoundingTypeAttr	Rounding type that operations support
`exclude_pads`	::mlir::UnitAttr	unit attribute

5.3.2.2.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.2.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.3 `IE.CNNNetwork` (vpux::IE::CNNNetworkOp)

InferenceEngine CNN Network description

Syntax:

```
operation ::= `IE.CNNNetwork` attr-dict
              `entryPoint` `:` $entryPoint
              `inputsInfo` `:` $inputsInfo
              `outputsInfo` `:` $outputsInfo
```

This operation is bound to MLIR Module and holds extra information about InferenceEngine CNN Network:

- Precision and layout for user-provided inputs.
- Precision and layout for user-provided outputs.
- Entry point (Function name) for the network inference.

5.3.2.3.1 Attributes:

Attribute	MLIR Type	Description
`entryPoint`	::mlir::FlatSymbolRefAttr	flat symbol reference attribute

5.3.2.4 `IE.CTCGreedyDecoder` (vpux::IE::CTCGreedyDecoderOp)

InferenceEngine CTCGreedyDecoder layer

Syntax:

```
operation ::= `IE.CTCGreedyDecoder` `(` operands `)` attr-dict `:`
type(operands) `->` type(results)
```

5.3.2.4.1 Attributes:

Attribute	MLIR Type	Description
`mergeRepeated`	::mlir::UnitAttr	unit attribute

5.3.2.4.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values
`sequenceLengths`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.4.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.5 `IE.CTCGreedyDecoderSeqLen` (vpux::IE::CTCGreedyDecoderSeqLenOp)

InferenceEngine CTCGreedyDecoderSeqLen layer

Syntax:

```
operation ::= `IE.CTCGreedyDecoderSeqLen` `(` operands `)` attr-dict `:`
type(operands) `->` type(results)
```

5.3.2.5.1 Attributes:

Attribute	MLIR Type	Description
`mergeRepeated`	::mlir::UnitAttr	unit attribute

5.3.2.5.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values
`sequenceLength`	ranked tensor of 32-bit signed integer values
`blankIndex`	ranked tensor of 32-bit signed integer values

5.3.2.5.3 Results:

Result	Description
output	ranked tensor of 32-bit signed integer values
outputLength	ranked tensor of 32-bit signed integer values

5.3.2.6 IE.Clamp (vpux::IE::ClampOp)

InferenceEngine Clamp layer

Syntax:

```
operation ::= `IE.Clamp` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.6.1 Attributes:

Attribute	MLIR Type	Description
min	::mlir::FloatAttr	64-bit float attribute
max	::mlir::FloatAttr	64-bit float attribute

5.3.2.6.2 Operands:

Operand	Description
input	ranked tensor of 16-bit float or 32-bit float values

5.3.2.6.3 Results:

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.7 IE.Concat (vpux::IE::ConcatOp)

InferenceEngine Concat layer

Syntax:

```
operation ::= `IE.Concat` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.7.1 Attributes:

Attribute	MLIR Type	Description
axis	mlir::IntegerAttr	Integer attribute
offset	mlir::IntegerAttr	Integer attribute
stride	mlir::IntegerAttr	Integer attribute

5.3.2.7.2 Operands:

Operand	Description
inputs	ranked tensor of any type values

5.3.2.7.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.8 `IE.Convert` (vpux::IE::ConvertOp)

InferenceEngine Convert layer

Syntax:

```
operation ::= `IE.Convert` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.8.1 Attributes:

Attribute	MLIR Type	Description
`dstElemType`	::mlir::TypeAttr	any type attribute

5.3.2.8.2 Operands:

Operand	Description
`input`	ranked tensor of any type values

5.3.2.8.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.9 `IE.Convolution` (vpux::IE::ConvolutionOp)

InferenceEngine Convolution layer

Syntax:

```
operation ::= `IE.Convolution` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.9.1 Attributes:

Attribute	MLIR Type	Description
`strides`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_begin`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end`	::mlir::ArrayAttr	64-bit integer array attribute
`dilations`	::mlir::ArrayAttr	64-bit integer array attribute

Attribute	MLIR Type	Description
`post_op`	vpux::IE::PostOp	DictionaryAttr with field(s): 'name', 'attrs' (each field having its own constraints)

5.3.2.9.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values
`filter`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values
`bias`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values

5.3.2.9.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values

5.3.2.10 `IE.DataInfo` (vpux::IE::DataInfoOp)

Information about InferenceEngine CNN Network input/output Data object

Syntax:

```
operation ::= `IE.DataInfo` $name `:` $userType
              attr-dict
```

This operation is bound to `IE.CNNNetwork` Operation and holds information about Data object:

- Name
- Original shape
- User-defined precision (element type)
- User-defined layout

5.3.2.10.1 Attributes:

Attribute	MLIR Type	Description
`name`	::mlir::StringAttr	string attribute
`userType`	::mlir::TypeAttr	any type attribute

5.3.2.11 `IE.Deconvolution` (vpux::IE::DeconvolutionOp)

InferenceEngine Deconvolution layer

Syntax:

```
operation ::= `IE.Deconvolution` `(` operands `)` attr-dict `:` type(operands)
`->` type(results)
```

5.3.2.11.1 Attributes:

Attribute	MLIR Type	Description
`strides`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_begin`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end`	::mlir::ArrayAttr	64-bit integer array attribute
`dilations`	::mlir::ArrayAttr	64-bit integer array attribute
`output_padding`	::mlir::ArrayAttr	64-bit integer array attribute

5.3.2.11.2 Operands:

Operand	Description
`feature`	ranked tensor of any type values
`filter`	ranked tensor of any type values
`output_shape`	1D tensor of integer values

5.3.2.11.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.12 `IE.Dequantize` (vpux::IE::DequantizeOp)

InferenceEngine Dequantize layer

Syntax:

```
operation ::= `IE.Dequantize` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.12.1 Attributes:

Attribute	MLIR Type	Description
`dstElemType`	::mlir::TypeAttr	any type attribute

5.3.2.12.2 Operands:

Operand	Description
`input`	ranked tensor of QuantizedType values

5.3.2.12.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.13 `IE.DetectionOutput` (vpux::IE::DetectionOutputOp)

InferenceEngine DetectionOutput layer

Syntax:

```
operation ::= `IE.DetectionOutput` `(` operands `)` attr-dict `:`
type(operands) `->` type(results)
```

5.3.2.13.1 Attributes:

Attribute	MLIR Type	Description
`attr`	vpux::IE::DetectionOutputAttr	DictionaryAttr with field(s): 'num_classes', 'background_label_id', 'top_k', 'variance_encoded_in_target', 'keep_top_k', 'code_type', 'share_location', 'nms_threshold', 'confidence_threshold', 'clip_after_nms', 'clip_before_nms', 'decrease_label_id', 'normalized', 'input_height', 'input_width', 'objectness_score' (each field having its own constraints)

5.3.2.13.2 Operands:

Operand	Description
`in_box_logits`	2D tensor of floating-point values
`in_class_preds`	2D tensor of floating-point values
`in_proposals`	3D tensor of floating-point values
`in_additional_preds`	2D tensor of floating-point values
`in_additional_proposals`	2D tensor of floating-point values

5.3.2.13.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.14 `IE.Divide` (vpux::IE::DivideOp)

InferenceEngine Divide layer

Syntax:

```
operation ::= `IE.Divide` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.14.1 Attributes:

Attribute	MLIR Type	Description
`auto_broadcast`	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.14.2 Operands:

Operand	Description
`input1`	ranked tensor of 16-bit float or 32-bit float values
`input2`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.14.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.15 `IE.Elu` (vpux::IE::EluOp)

InferenceEngine Elu layer

Syntax:

```
operation ::= `IE.Elu` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.15.1 Attributes:

Attribute	MLIR Type	Description
x	::mlir::FloatAttr	64-bit float attribute

5.3.2.15.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.15.3 Results:

Result	Description

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.16 `IE.Erf` (vpux::IE::ErfOp)

InferenceEngine Erf layer

Syntax:

```
operation ::= `IE.Erf` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.16.1 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.16.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.17 `IE.Exp` (vpux::IE::ExpOp)

InferenceEngine Exp layer

Syntax:

```
operation ::= `IE.Exp` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.17.1 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.17.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.18 `IE.Expand` (vpux::IE::ExpandOp)

Expand tensor with uninitialized values

Syntax:

```
operation ::= `IE.Expand` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.18.1 Attributes:

Attribute	MLIR Type	Description
pads_begin	::mlir::ArrayAttr	64-bit integer array attribute
pads_end	::mlir::ArrayAttr	64-bit integer array attribute

5.3.2.18.2 Operands:

Operand	Description
input	ranked tensor of any type values

5.3.2.18.3 Results:

Result	Description
output	ranked tensor of any type values

5.3.2.19 IE.FakeQuantize (vpux::IE::FakeQuantizeOp)

InferenceEngine FakeQuantize layer

Syntax:

```
operation ::= `IE.FakeQuantize` `(` operands `)` attr-dict `:` type(operands)
`->` type(results)
```

5.3.2.19.1 Attributes:

Attribute	MLIR Type	Description
levels	mlir::IntegerAttr	Integer attribute
auto_broadcast	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.19.2 Operands:

Operand	Description
input	ranked tensor of 16-bit float or 32-bit float values
input_low	ranked tensor of 16-bit float or 32-bit float values
input_high	ranked tensor of 16-bit float or 32-bit float values
output_low	ranked tensor of 16-bit float or 32-bit float values
output_high	ranked tensor of 16-bit float or 32-bit float values

5.3.2.19.3 Results:

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.20 IE.FloorMod (vpux::IE::FloorModOp)

InferenceEngine FloorMod layer

Syntax:

```
operation ::= `IE.FloorMod` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.20.1 Attributes:

Attribute	MLIR Type	Description
`auto_broadcast`	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.20.2 Operands:

Operand	Description
`input1`	ranked tensor of 16-bit float or 32-bit float values
`input2`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.20.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.21 `IE.Floor` (vpux::IE::FloorOp)

InferenceEngine Floor layer

Syntax:

```
operation ::= `IE.Floor` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.21.1 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.21.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.22 `IE.FullyConnected` (vpux::IE::FullyConnectedOp)

InferenceEngine FullyConnected layer

Syntax:

```
operation ::= `IE.FullyConnected` `(` operands `)` attr-dict `:` type(operands)
`->` type(results)
```

5.3.2.22.1 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values
`weights`	ranked tensor of 16-bit float or 32-bit float values
`bias`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.22.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.23 `IE.GRN` (vpux::IE::GRNOp)

InferenceEngine GRN layer

Syntax:

```
operation ::= `IE.GRN` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.23.1 Attributes:

Attribute	MLIR Type	Description
`bias`	::mlir::FloatAttr	64-bit float attribute

5.3.2.23.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.23.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.24 `IE.Gather` (vpux::IE::GatherOp)

InferenceEngine Gather layer

Syntax:

```
operation ::= `IE.Gather` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.24.1 Operands:

Operand	Description

Operand	Description
`input`	ranked tensor of any type values
`indices`	ranked tensor of any type values
`axis`	ranked tensor of any type values

5.3.2.24.2 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.25 `IE.GroupConvolution` (vpux::IE::GroupConvolutionOp)

InferenceEngine GroupConvolution layer

Syntax:

```
operation ::= `IE.GroupConvolution` `(` operands `)` attr-dict `:`
type(operands) `->` type(results)
```

5.3.2.25.1 Attributes:

Attribute	MLIR Type	Description
`strides`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_begin`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end`	::mlir::ArrayAttr	64-bit integer array attribute
`dilations`	::mlir::ArrayAttr	64-bit integer array attribute
`groups`	mlir::IntegerAttr	Integer attribute
`post_op`	vpux::IE::PostOp	DictionaryAttr with field(s): 'name', 'attrs' (each field having its own constraints)

5.3.2.25.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values

Operand	Description
`filter`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values
`bias`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values

5.3.2.25.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values

5.3.2.26 `IE.HSwish` (vpux::IE::HSwishOp)

InferenceEngine HSwish layer

Syntax:

```
operation ::= `IE.HSwish` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.26.1 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.26.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.27 `IE.Interpolate` (vpux::IE::InterpolateOp)

InferenceEngine Interpolate layer

Syntax:

```
operation ::= `IE.Interpolate` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.27.1 Attributes:

Attribute	MLIR Type	Description

Attribute	MLIR Type	Description
sizes_attr	::mlir::ArrayAttr	64-bit integer array attribute
scales_attr	::mlir::ArrayAttr	64-bit float array attribute
axes_attr	::mlir::ArrayAttr	64-bit integer array attribute
attr	vpux::IE::InterpolateAttr	DictionaryAttr with field(s): 'mode', 'shape_calc_mode', 'coord_mode', 'nearest_mode', 'antialias', 'pads_begin', 'pads_end', 'cube_coeff' (each field having its own constraints)

5.3.2.27.2 Operands:

Operand	Description
input	ranked tensor of 16-bit float or 32-bit float values
sizes	ranked tensor of integer values
scales	ranked tensor of 16-bit float or 32-bit float values
axes	ranked tensor of integer values

5.3.2.27.3 Results:

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.28 `IE.LRN` (vpux::IE::LRNOp)

InferenceEngine LRN layer

Syntax:

```
operation ::= `IE.LRN` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.28.1 Attributes:

Attribute	MLIR Type	Description
alpha	::mlir::FloatAttr	64-bit float attribute
beta	::mlir::FloatAttr	64-bit float attribute
bias	::mlir::FloatAttr	64-bit float attribute
size	mlir::IntegerAttr	Integer attribute

5.3.2.28.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values
`axis`	ranked tensor of any type values

5.3.2.28.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.29 `IE.LRN_IE` (vpux::IE::LRN_IEOp)

InferenceEngine LRN_IE layer

Syntax:

```
operation ::= `IE.LRN_IE` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.29.1 Attributes:

Attribute	MLIR Type	Description
`alpha`	::mlir::FloatAttr	64-bit float attribute
`beta`	::mlir::FloatAttr	64-bit float attribute
`bias`	::mlir::FloatAttr	64-bit float attribute
`size`	mlir::IntegerAttr	Integer attribute
`region`	vpux::IE::LRN_IERegionAttr	LRN_IE region that operations support

5.3.2.29.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.29.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.30 `IE.LSTMCell` (vpux::IE::LSTMCellOp)

InferenceEngine LSTMCell layer

Syntax:

```
operation ::= `IE.LSTMCell` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.30.1 Attributes:

Attribute	MLIR Type	Description
`hiddenSize`	mlir::IntegerAttr	Integer attribute

5.3.2.30.2 Operands:

Operand	Description
`inputData`	2D tensor of 16-bit float or 32-bit float values
`initialHiddenState`	2D tensor of 16-bit float or 32-bit float values
`initialCellState`	2D tensor of 16-bit float or 32-bit float values
`weights`	2D tensor of 16-bit float or 32-bit float values
`recurrenceWeights`	2D tensor of 16-bit float or 32-bit float values
`biases`	1D tensor of 16-bit float or 32-bit float values

5.3.2.30.3 Results:

Result	Description
`outputHiddenState`	2D tensor of 16-bit float or 32-bit float values
`outputCellState`	2D tensor of 16-bit float or 32-bit float values

5.3.2.31 `IE.LeakyRelu` (vpux::IE::LeakyReluOp)

InferenceEngine LeakyRelu layer

Syntax:

```
operation ::= `IE.LeakyRelu` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.31.1 Attributes:

Attribute	MLIR Type	Description
`negative_slope`	::mlir::FloatAttr	64-bit float attribute

5.3.2.31.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.31.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.32 `IE.MVN` (vpux::IE::MVNOp)

InferenceEngine MVN layer

Syntax:

```
operation ::= `IE.MVN` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.32.1 Attributes:

Attribute	MLIR Type	Description
`across_channels`	::mlir::BoolAttr	bool attribute
`normalize_variance`	::mlir::BoolAttr	bool attribute
`eps`	::mlir::FloatAttr	64-bit float attribute

5.3.2.32.2 Operands:

Operand	Description
`input`	ranked tensor of any type values

5.3.2.32.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.33 `IE.MatMul` (vpux::IE::MatMulOp)

InferenceEngine MatMul layer

Syntax:

```
operation ::= `IE.MatMul` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.33.1 Attributes:

Attribute	MLIR Type	Description
`transpose_a`	::mlir::UnitAttr	unit attribute
`transpose_b`	::mlir::UnitAttr	unit attribute

5.3.2.33.2 Operands:

Operand	Description
`input1`	ranked tensor of 16-bit float or 32-bit float values
`input2`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.33.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.34 `IE.MaxPool` (vpux::IE::MaxPoolOp)

InferenceEngine MaxPool layer

Syntax:

```
operation ::= `IE.MaxPool` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.34.1 Attributes:

Attribute	MLIR Type	Description
`kernel_size`	::mlir::ArrayAttr	64-bit integer array attribute
`strides`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_begin`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end`	::mlir::ArrayAttr	64-bit integer array attribute
`rounding_type`	vpux::IE::RoundingTypeAttr	Rounding type that operations support
`post_op`	vpux::IE::PostOp	DictionaryAttr with field(s): 'name', 'attrs' (each field having its own constraints)

5.3.2.34.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values

5.3.2.34.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float or QuantizedType values

5.3.2.35 `IE.Maximum` (vpux::IE::MaximumOp)

InferenceEngine Maximum layer

Syntax:

```
operation ::= `IE.Maximum` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.35.1 Attributes:

Attribute	MLIR Type	Description

Attribute	MLIR Type	Description
`auto_broadcast`	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.35.2 Operands:

Operand	Description
`input1`	ranked tensor of any type values
`input2`	ranked tensor of any type values

5.3.2.35.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.36 `IE.Minimum` (vpux::IE::MinimumOp)

InferenceEngine Minimum layer

Syntax:

```
operation ::= `IE.Minimum` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.36.1 Attributes:

Attribute	MLIR Type	Description
`auto_broadcast`	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.36.2 Operands:

Operand	Description
`input1`	ranked tensor of any type values
`input2`	ranked tensor of any type values

5.3.2.36.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.37 `IE.Mish` (vpux::IE::MishOp)

InferenceEngine Mish layer

Syntax:

```
operation ::= `IE.Mish` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.37.1 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.37.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.38 `IE.Multiply` (vpux::IE::MultiplyOp)

InferenceEngine Multiply layer

Syntax:

```
operation ::= `IE.Multiply` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.38.1 Attributes:

Attribute	MLIR Type	Description
`auto_broadcast`	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.38.2 Operands:

Operand	Description
`input1`	ranked tensor of 16-bit float or 32-bit float values
`input2`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.38.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.39 `IE.Negative` (vpux::IE::NegativeOp)

InferenceEngine Negative layer

Syntax:

```
operation ::= `IE.Negative` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.39.1 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.39.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.40 `IE.NormalizeL2` (vpux::IE::NormalizeL2Op)

InferenceEngine NormalizeL2 layer

Syntax:

```
operation ::= `IE.NormalizeL2` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.40.1 Attributes:

Attribute	MLIR Type	Description
`eps`	::mlir::FloatAttr	64-bit float attribute
`eps_mod`	vpux::IE::EpsModeAttr	EpsMode that the InferenceEngine supports

5.3.2.40.2 Operands:

Operand	Description
`data`	ranked tensor of any type values
`axes`	ranked tensor of integer values

5.3.2.40.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.41 `IE.PRelu` (vpux::IE::PReluOp)

InferenceEngine PRelu layer

Syntax:

```
operation ::= `IE.PRelu` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.41.1 Operands:

Operand	Description

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values
`negative_slope`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.41.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.42 `IE.Pad` (vpux::IE::PadOp)

InferenceEngine Pad layer

Syntax:

```
operation ::= `IE.Pad` `(` $input `)` (`[` $pads_begin^ `,` $pads_end (`,`
$pad_value^)? `]`)? attr-dict `:` type(operands) `->` type(results)
```

5.3.2.42.1 Attributes:

Attribute	MLIR Type	Description
`pads_begin_attr`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end_attr`	::mlir::ArrayAttr	64-bit integer array attribute
`pad_value_attr`	::mlir::FloatAttr	64-bit float attribute
`mode`	vpux::IE::PadModeAttr	TPadMode that the InferenceEngine supports

5.3.2.42.2 Operands:

Operand	Description
`input`	ranked tensor of any type values
`pads_begin`	ranked tensor of integer values
`pads_end`	ranked tensor of integer values
`pad_value`	ranked tensor of integer or floating-point values

5.3.2.42.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.43 `IE.PerAxisTile` (vpux::IE::PerAxisTileOp)

InferenceEngine per axis Tile layer

Syntax:

```
operation ::= `IE.PerAxisTile` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.43.1 Attributes:

Attribute	MLIR Type	Description

Attribute	MLIR Type	Description
axis	mlir::IntegerAttr	Integer attribute
tiles	mlir::IntegerAttr	Integer attribute

5.3.2.43.2 Operands:

Operand	Description
input	ranked tensor of any type values

5.3.2.43.3 Results:

Result	Description
output	ranked tensor of any type values

5.3.2.44 IE.Power (vpux::IE::PowerOp)

InferenceEngine Power layer

Syntax:

```
operation ::= `IE.Power` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.44.1 Attributes:

Attribute	MLIR Type	Description
auto_broadcast	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.44.2 Operands:

Operand	Description
input1	ranked tensor of 16-bit float or 32-bit float values
input2	ranked tensor of 16-bit float or 32-bit float values

5.3.2.44.3 Results:

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.45 IE.Proposal (vpux::IE::ProposalOp)

InferenceEngine Proposal layer

Syntax:

```
operation ::= `IE.Proposal` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.45.1 Attributes:

Attribute	MLIR Type	Description
`proposal_attrs`	vpux::IE::ProposalAttr	DictionaryAttr with field(s): 'baseSize', 'preNmsTopN', 'postNmsTopN', 'nmsThresh', 'featStride', 'minSize', 'ratio', 'scale', 'clipBeforeNms', 'clipAfterNms', 'normalize', 'boxSizeScale', 'boxCoordinateScale', 'framework', 'inferProbs' (each field having its own constraints)

5.3.2.45.2 Operands:

Operand	Description
`class_probs`	ranked tensor of 16-bit float or 32-bit float values
`bbox_deltas`	ranked tensor of 16-bit float or 32-bit float values
`image_shape`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.45.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.46 `IE.Quantize` (vpux::IE::QuantizeOp)

InferenceEngine Quantize layer

Syntax:

```
operation ::= `IE.Quantize` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.46.1 Attributes:

Attribute	MLIR Type	Description
`dstElemType`	::mlir::TypeAttr	any type attribute

5.3.2.46.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.46.3 Results:

Result	Description
output	ranked tensor of QuantizedType values

5.3.2.47 IE.ROIPooling (vpux::IE::ROIPoolingOp)

InferenceEngine ROIPooling layer

Syntax:

```
operation ::= `IE.ROIPooling` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.47.1 Attributes:

Attribute	MLIR Type	Description
output_size	::mlir::ArrayAttr	64-bit integer array attribute
spatial_scale	::mlir::FloatAttr	64-bit float attribute
method	vpux::IE::ROIPoolingMethodAttr	ROIPoolingMethod that the InferenceEngine supports

5.3.2.47.2 Operands:

Operand	Description
input	ranked tensor of 16-bit float or 32-bit float values
coords	ranked tensor of 16-bit float or 32-bit float values

5.3.2.47.3 Results:

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.48 IE.ReLU (vpux::IE::ReLUOp)

InferenceEngine ReLU layer

Syntax:

```
operation ::= `IE.ReLU` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.48.1 Operands:

Operand	Description
input	ranked tensor of 16-bit float or 32-bit float values

5.3.2.48.2 Results:

Result	Description

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.49 IE.RegionYolo (vpux::IE::RegionYoloOp)

InferenceEngine RegionYolo layer

Syntax:

```
operation ::= `IE.RegionYolo` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.49.1 Attributes:

Attribute	MLIR Type	Description
coords	mlir::IntegerAttr	Integer attribute
classes	mlir::IntegerAttr	Integer attribute
regions	mlir::IntegerAttr	Integer attribute
do_softmax	::mlir::BoolAttr	bool attribute
mask	::mlir::ArrayAttr	64-bit integer array attribute
axis	mlir::IntegerAttr	Integer attribute
end_axis	mlir::IntegerAttr	Integer attribute
anchors	::mlir::ArrayAttr	64-bit float array attribute

5.3.2.49.2 Operands:

Operand	Description
input	4D tensor of floating-point values

5.3.2.49.3 Results:

Result	Description
output	ranked tensor of any type values

5.3.2.50 IE.Reorder (vpux::IE::ReorderOp)

InferenceEngine Reorder layer

Syntax:

```
operation ::= `IE.Reorder` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.50.1 Attributes:

Attribute	MLIR Type	Description
dstOrder	::mlir::AffineMapAttr	AffineMap attribute

5.3.2.50.2 Operands:

Operand	Description

Operand	Description
`input`	ranked tensor of any type values

5.3.2.50.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.51 `IE.ReorgYolo` (vpux::IE::ReorgYoloOp)

InferenceEngine ReorgYolo layer

Syntax:

```
operation ::= `IE.ReorgYolo` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.51.1 Attributes:

Attribute	MLIR Type	Description
`stride`	mlir::IntegerAttr	Integer attribute

5.3.2.51.2 Operands:

Operand	Description
`input`	4D tensor of integer or floating-point values

5.3.2.51.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.52 `IE.Reshape` (vpux::IE::ReshapeOp)

InferenceEngine Reshape layer

Syntax:

```
operation ::= `IE.Reshape` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.52.1 Attributes:

Attribute	MLIR Type	Description
`special_zero`	::mlir::UnitAttr	unit attribute
`shape_value`	::mlir::ArrayAttr	64-bit integer array attribute

5.3.2.52.2 Operands:

Operand	Description
`input`	ranked tensor of any type values

Operand	Description
shape	ranked tensor of integer values

5.3.2.52.3 Results:

Result	Description
output	ranked tensor of any type values

5.3.2.53 `IE.ScaleShift` (vpux::IE::ScaleShiftOp)

InferenceEngine ScaleShift layer

Syntax:

```
operation ::= `IE.ScaleShift` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.53.1 Operands:

Operand	Description
input	ranked tensor of 16-bit float or 32-bit float values
weights	ranked tensor of 16-bit float or 32-bit float values
biases	ranked tensor of 16-bit float or 32-bit float values

5.3.2.53.2 Results:

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.54 `IE.Sigmoid` (vpux::IE::SigmoidOp)

InferenceEngine Sigmoid layer

Syntax:

```
operation ::= `IE.Sigmoid` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.54.1 Operands:

Operand	Description
input	ranked tensor of 16-bit float or 32-bit float values

5.3.2.54.2 Results:

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.55 `IE.Slice` (vpux::IE::SliceOp)

Extract single slice from tensor

Syntax:

```
operation ::= `IE.Slice` $source $static_offsets $static_sizes
              attr-dict `:` type($source) `to` type(results)
```

5.3.2.55.1 Attributes:

Attribute	MLIR Type	Description
`static_offsets`	::mlir::ArrayAttr	64-bit integer array attribute
`static_sizes`	::mlir::ArrayAttr	64-bit integer array attribute

5.3.2.55.2 Operands:

Operand	Description
`source`	ranked tensor of any type values

5.3.2.55.3 Results:

Result	Description
`result`	ranked tensor of any type values

5.3.2.56 `IE.SoftMax` (vpux::IE::SoftMaxOp)

InferenceEngine SoftMax layer

Syntax:

```
operation ::= `IE.SoftMax` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.56.1 Attributes:

Attribute	MLIR Type	Description
`axisInd`	mlir::IntegerAttr	Integer attribute

5.3.2.56.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.56.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.57 `IE.Split` (vpux::IE::SplitOp)

InferenceEngine Split layer

Syntax:

```
operation ::= `IE.Split` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.57.1 Attributes:

Attribute	MLIR Type	Description
num_splits	mlir::IntegerAttr	Integer attribute
axis_value	mlir::IntegerAttr	Integer attribute

5.3.2.57.2 Operands:

Operand	Description
input	ranked tensor of any type values
axis	ranked tensor of any type values

5.3.2.57.3 Results:

Result	Description
outputs	ranked tensor of any type values

5.3.2.58 IE.SquaredDiff (vpux::IE::SquaredDifferenceOp)

InferenceEngine SquaredDiff layer

Syntax:

```
operation ::= `IE.SquaredDiff` `(` operands `)` attr-dict `:` type(operands) `-
>` type(results)
```

5.3.2.58.1 Attributes:

Attribute	MLIR Type	Description
auto_broadcast	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.58.2 Operands:

Operand	Description
input1	ranked tensor of 16-bit float or 32-bit float values
input2	ranked tensor of 16-bit float or 32-bit float values

5.3.2.58.3 Results:

Result	Description
output	ranked tensor of 16-bit float or 32-bit float values

5.3.2.59 IE.Squeeze (vpux::IE::SqueezeOp)

InferenceEngine Squeeze layer

Syntax:

```
operation ::= `IE.Squeeze` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.59.1 Attributes:

Attribute	MLIR Type	Description
axes_value	::mlir::ArrayAttr	64-bit integer array attribute

5.3.2.59.2 Operands:

Operand	Description
input	ranked tensor of any type values
axes	ranked tensor of integer values

5.3.2.59.3 Results:

Result	Description
output	ranked tensor of any type values

5.3.2.60 IE.StridedSlice (vpux::IE::StridedSliceOp)

InferenceEngine StridedSlice layer

Syntax:

```
operation ::= `IE.StridedSlice` `(` operands `)` attr-dict `:` type(operands)
`->` type(results)
```

5.3.2.60.1 Attributes:

Attribute	MLIR Type	Description
begins_attr	::mlir::ArrayAttr	64-bit integer array attribute
ends_attr	::mlir::ArrayAttr	64-bit integer array attribute
strides_attr	::mlir::ArrayAttr	64-bit integer array attribute
begin_mask	::mlir::ArrayAttr	64-bit integer array attribute
end_mask	::mlir::ArrayAttr	64-bit integer array attribute
new_axis_mask	::mlir::ArrayAttr	64-bit integer array attribute
shrink_axis_mask	::mlir::ArrayAttr	64-bit integer array attribute
ellipsis_mask	::mlir::ArrayAttr	64-bit integer array attribute

5.3.2.60.2 Operands:

Operand	Description
input	ranked tensor of any type values
begins	1D tensor of integer values
ends	1D tensor of integer values

Operand	Description
`strides`	1D tensor of integer values

5.3.2.60.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.61 `IE.Subtract` (vpux::IE::SubtractOp)

InferenceEngine Subtract layer

Syntax:

```
operation ::= `IE.Subtract` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.61.1 Attributes:

Attribute	MLIR Type	Description
`auto_broadcast`	vpux::IE::AutoBroadcastTypeAttr	Specifies rules used for auto-broadcasting of input tensors

5.3.2.61.2 Operands:

Operand	Description
`input1`	ranked tensor of 16-bit float or 32-bit float values
`input2`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.61.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.62 `IE.Swish` (vpux::IE::SwishOp)

InferenceEngine Swish layer

Syntax:

```
operation ::= `IE.Swish` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.62.1 Attributes:

Attribute	MLIR Type	Description
`beta_value`	::mlir::FloatAttr	64-bit float attribute

5.3.2.62.2 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values
`beta`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.62.3 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.63 `IE.Tanh` (vpux::IE::TanhOp)

InferenceEngine Tanh layer

Syntax:

```
operation ::= `IE.Tanh` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.63.1 Operands:

Operand	Description
`input`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.63.2 Results:

Result	Description
`output`	ranked tensor of 16-bit float or 32-bit float values

5.3.2.64 `IE.Tile` (vpux::IE::TileOp)

InferenceEngine Tile layer

Syntax:

```
operation ::= `IE.Tile` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.64.1 Operands:

Operand	Description
`input`	ranked tensor of any type values
`repeats`	ranked tensor of 64-bit signed integer values

5.3.2.64.2 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.65 `IE.TopK` (vpux::IE::TopKOp)

InferenceEngine TopK layer

Syntax:

```
operation ::= `IE.TopK` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.65.1 Attributes:

Attribute	MLIR Type	Description
`axis`	mlir::IntegerAttr	Integer attribute
`mode`	vpux::IE::TopKModeAttr	TopKMode that the InferenceEngine supports
`sort`	vpux::IE::TopKSortTypeAttr	TopKSortType that the InferenceEngine supports
`element_type`	::mlir::TypeAttr	any type attribute

5.3.2.65.2 Operands:

Operand	Description
`input`	ranked tensor of any type values
k	0D tensor of integer values

5.3.2.65.3 Results:

Result	Description
`output_values`	ranked tensor of any type values
`target_shape`	ranked tensor of any type values

5.3.2.66 `IE.Transpose` (vpux::IE::TransposeOp)

InferenceEngine Transpose layer

Syntax:

```
operation ::= `IE.Transpose` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.66.1 Attributes:

Attribute	MLIR Type	Description
`order_value`	::mlir::AffineMapAttr	AffineMap attribute

5.3.2.66.2 Operands:

Operand	Description
`input`	ranked tensor of any type values

Operand	Description
`order`	ranked tensor of 64-bit signed integer values

5.3.2.66.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.3.2.67 `IE.Unsqueeze` (vpux::IE::UnsqueezeOp)

InferenceEngine Unsqueeze layer

Syntax:

```
operation ::= `IE.Unsqueeze` `(` operands `)` attr-dict `:` type(operands) `->`
type(results)
```

5.3.2.67.1 Attributes:

Attribute	MLIR Type	Description
`axes_value`	::mlir::ArrayAttr	64-bit integer array attribute

5.3.2.67.2 Operands:

Operand	Description
`input`	ranked tensor of any type values
`axes`	ranked tensor of integer values

5.3.2.67.3 Results:

Result	Description
`output`	ranked tensor of any type values

5.4 'IERT' Dialect

InferenceEngine RunTime Dialect The **IERT Dialect** represents bufferized version of **IE Dialect**.

It has the following properties:

- Works with fixed operation set (like **IE Dialect**).
- Represents execution scheduling and memory allocation.
- Works with `MemRefType` .
- Includes transformations and optimizations closer to HW level (memory re-usage, parallel resources usage, etc.).

TBD: It operates with `MemRefType` , but in contrast to MLIR uses SSA value semantic (inspired by PlaidML approach). It combines both memory effects and buffer aliasing for this:

- Each layer operation takes as its operands both input and output buffers.
- The layer marks input buffer as read-only and output buffer as write-only via memory effects inferface.
- The layer returns new buffer Value, which is an alias for output buffer.

```
#NHWC = affine_map<(n, c, h, w) -> (n, h, w, c)>

func @main(%input: memref<1x3x240x240xf16, #NHWC>, %output: memref<1x3x240x240xf16,
#NHWC>) -> memref<1x3x240x240xf16, #NHWC> {
    %1 = IERT.SoftMax(%input, %output) {axisInd = 1} // %1 is an alias for %output
    return %1
}
```

The memory allocation/deallocation is defined as separate operations (dynamic or static).

The **IERT Dialect** uses the following scheme to represent scheduling information:

- Operations order defines scheduling.
- Each IERT operation is assumed as blocking: next operation will not start until previous is finished.
- Concurrent execution is defined as asynchronous regions (**Async Dialect**).

```
%11_t, %11_f = async.execute { IERT.executor = "NCE_Cluster" }
    [%7_t, %8_9_t](%8_9_f#0 as %8, %8_9_f#1 as %9)
{
    %11_0_t, %11_0_f = async.execute { IERT.executor = "DPU" }
    {
        %11_0 = IERT.Convolution(%7, %8, %9) to %10_0 { strides = [1, 1],
pads_begin = [1, 1], pads_end = [1, 1] }
        async.yield %11_0
    }

    %11_1_t, %11_1_f = async.execute { IERT.executor = "DPU" }
    {
        %11_1 = IERT.Convolution(%7, %8, %9) to %10_1 { strides = [1, 1],
pads_begin = [1, 1], pads_end = [1, 1] }
        async.yield %11_1
    }

    %11:2 = async.await %11_0_f, %11_1_f
    %11 = IERT.FakeConcat(%11#0, %11#1) to %10
    async.yield %11
}
```

The **IERT Dialect** provides separate Operation to describe the available and used run-time resources. It deals with the following resource types:

- Memory space.
- Executor (CPU, HW module, DMA).

```
IERT.RunTimeResources
    availableMemory : {
        IERT.MemoryResource 1073741824 bytes
        IERT.MemoryResource 31457280 bytes of "DDR" {VPUIP.bandwidth = 8 : i64,
VPUIP.derateFactor = 6.000000e-01 : f64}
        IERT.MemoryResource 4194304 bytes of "CMX_UPA" {VPUIP.bandwidth = 16 : i64,
VPUIP.derateFactor = 8.500000e-01 : f64}
        IERT.MemoryResource 1048576 bytes of "CMX_NN" {VPUIP.bandwidth = 32 : i64,
VPUIP.derateFactor = 1.000000e+00 : f64}
    }
    usedMemory : {
        IERT.MemoryResource 2048 bytes of "DDR"
        IERT.MemoryResource 1048576 bytes of "CMX_NN"
    }
    executors : {
        IERT.ExecutorResource 1 of "Leon_RT"
        IERT.ExecutorResource 1 of "Leon_NN"
        IERT.ExecutorResource 16 of "SHAVE_UPA"
        IERT.ExecutorResource 20 of "SHAVE_NN"
        IERT.ExecutorResource 4 of "NCE_Cluster" {
            IERT.ExecutorResource 5 of "NCE_PerClusterDPU"
        }
        IERT.ExecutorResource 1 of "DMA_UPA"
        IERT.ExecutorResource 1 of "DMA_NN"
    }
```

The `IERT.RunTimeResources` is filled by underlying low-level dialect to provide information about HW-specific resources.

5.4.1 OpInterface definitions

5.4.1.1 AsyncLayerOpInterface (`IERT_AsyncLayerOpInterface`)

Interface for layers that will be executed asynchronously in separate Executor ##### Methods: ###### `getExecutor`

```
mlir::Attribute getExecutor(uint32_t&numUnits);
```

Get Executor for the asynchronous launch NOTE: This method *must* be implemented by the user.

5.4.1.2 LayerOpInterface (`IERT_LayerOpInterface`)

Base interface for IERT Layer Operation ##### Methods: ###### `getInputs`

```
mlir::OperandRange getInputs();
```

Get all layer input memory buffers NOTE: This method *must* be implemented by the user.

5.4.1.2.0.1 `getOutputs`

```
mlir::OperandRange getOutputs();
```

Get all layer output memory buffers NOTE: This method *must* be implemented by the user.

5.4.1.2.0.2 `getOpOperands`

```
llvm::detail::concat_range<mlir::OpOperand,
llvm::MutableArrayRef<mlir::OpOperand>, llvm::MutableArrayRef<mlir::OpOperand>>
getOpOperands();
```

Get all layer memory buffers NOTE: This method *must* be implemented by the user.

5.4.1.2.0.3 `getInOpOperands`

```
llvm::MutableArrayRef<mlir::OpOperand> getInOpOperands();
```

Get all layer input memory buffers NOTE: This method *must* be implemented by the user.

5.4.1.2.0.4 `getOutOpOperands`

```
llvm::MutableArrayRef<mlir::OpOperand> getOutOpOperands();
```

Get all layer output memory buffers NOTE: This method *must* be implemented by the user.

[TOC]

5.4.2 Operation definition

5.4.2.1 `IERT.Add` (vpux::IERT::AddOp)

InferenceEngine run-time Add layer

Syntax:

```
operation ::= `IERT.Add` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.1.1 Attributes:

Attribute	MLIR Type	Description
`post_op`	vpux::IE::PostOp	DictionaryAttr with field(s): 'name', 'attrs' (each field having its own constraints)

5.4.2.1.2 Operands:

Operand	Description
`input1`	memref of 16-bit float or 32-bit float values
`input2`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.1.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.2 `IERT.AvgPool` (vpux::IERT::AvgPoolOp)

InferenceEngine run-time AvgPool layer

Syntax:

```
operation ::= `IERT.AvgPool` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.2.1 Attributes:

Attribute	MLIR Type	Description
`kernel_size`	::mlir::ArrayAttr	64-bit integer array attribute
`strides`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_begin`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end`	::mlir::ArrayAttr	64-bit integer array attribute
`exclude_pads`	::mlir::UnitAttr	unit attribute

5.4.2.2.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.2.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.3 `IERT.CTCGreedyDecoder` (vpux::IERT::CTCGreedyDecoderOp)

InferenceEngine run-time CTCGreedyDecoder layer

Syntax:

```
operation ::= `IERT.CTCGreedyDecoder` attr-dict
              `inputs` `(` $input `:` type($input) `,` $sequenceLengths `:`
type($sequenceLengths) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.3.1 Attributes:

Attribute	MLIR Type	Description
mergeRepeated	::mlir::UnitAttr	unit attribute

5.4.2.3.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
sequenceLengths	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.3.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.4 IERT.CTCGreedyDecoderSeqLen (vpux::IERT::CTCGreedyDecoderSeqLenOp)

InferenceEngine run-time CTCGreedyDecoderSeqLen layer

Syntax:

```
operation ::= `IERT.CTCGreedyDecoderSeqLen` attr-dict
              `inputs` `(` $input `:` type($input) `,` $sequenceLength `:`
type($sequenceLength) (`,` $blankIndex^ `:` type($blankIndex))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `,`
$outputLength_buff `:` type($outputLength_buff) `)`
              `->` type(results)
```

5.4.2.4.1 Attributes:

Attribute	MLIR Type	Description
mergeRepeated	::mlir::UnitAttr	unit attribute

5.4.2.4.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
sequenceLength	memref of 32-bit signed integer values
blankIndex	memref of 32-bit signed integer values
output_buff	memref of 32-bit signed integer values
outputLength_buff	memref of 32-bit signed integer values

5.4.2.4.3 Results:

Result	Description
output	memref of 32-bit signed integer values
outputLength	memref of 32-bit signed integer values

5.4.2.5 IERT.Clamp (vpux::IERT::ClampOp)

InferenceEngine run-time Clamp layer

Syntax:

```
operation ::= `IERT.Clamp` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.5.1 Attributes:

Attribute	MLIR Type	Description
min	::mlir::FloatAttr	64-bit float attribute
max	::mlir::FloatAttr	64-bit float attribute

5.4.2.5.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.5.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.6 IERT.ConcatView (vpux::IERT::ConcatViewOp)

InferenceEngine run-time ConcatView layer. Dummy operation to maintain use-def chains.

Syntax:

```
operation ::= `IERT.ConcatView` attr-dict
              `inputs` `(` $inputs `:` type($inputs) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.6.1 Operands:

Operand	Description
inputs	memref of any type values
output_buff	memref of any type values

5.4.2.6.2 Results:

Result	Description
output	memref of any type values

5.4.2.7 `IERT.Convert` (vpux::IERT::ConvertOp)

InferenceEngine run-time Convert layer

Syntax:

```
operation ::= `IERT.Convert` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.7.1 Operands:

Operand	Description
input	memref of any type values
output_buff	memref of any type values

5.4.2.7.2 Results:

Result	Description
output	memref of any type values

5.4.2.8 `IERT.Convolution` (vpux::IERT::ConvolutionOp)

InferenceEngine run-time Convolution layer

Syntax:

```
operation ::= `IERT.Convolution` attr-dict
              `inputs` `(` $input `:` type($input) `,` $filter `:`
type($filter) (`,` $bias^ `:` type($bias))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.8.1 Attributes:

Attribute	MLIR Type	Description
strides	::mlir::ArrayAttr	64-bit integer array attribute
pads_begin	::mlir::ArrayAttr	64-bit integer array attribute
pads_end	::mlir::ArrayAttr	64-bit integer array attribute
dilations	::mlir::ArrayAttr	64-bit integer array attribute

Attribute	MLIR Type	Description
`post_op`	vpux::IE::PostOp	DictionaryAttr with field(s): 'name', 'attrs' (each field having its own constraints)

5.4.2.8.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float or QuantizedType values
`filter`	memref of 16-bit float or 32-bit float or QuantizedType values
`bias`	memref of 16-bit float or 32-bit float or QuantizedType values
`output_buff`	memref of 16-bit float or 32-bit float or QuantizedType values

5.4.2.8.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float or QuantizedType values

5.4.2.9 `IERT.Copy` (vpux::IERT::CopyOp)

InferenceEngine run-time Copy layer

Syntax:

```
operation ::= `IERT.Copy` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.9.1 Operands:

Operand	Description
`input`	memref of any type values

Operand	Description
`output_buff`	memref of any type values

5.4.2.9.2 Results:

Result	Description
`output`	memref of any type values

5.4.2.10 `IERT.Dequantize` (vpux::IERT::DequantizeOp)

InferenceEngine run-time Dequantize layer

Syntax:

```
operation ::= `IERT.Dequantize` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.10.1 Operands:

Operand	Description
`input`	memref of QuantizedType values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.10.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.11 `IERT.DetectionOutput` (vpux::IERT::DetectionOutputOp)

InferenceEngine run-time DetectionOutput layer

Syntax:

```
operation ::= `IERT.DetectionOutput` attr-dict
              `inputs` `(` $in_box_logits `:` type($in_box_logits) `,`
$in_class_preds `:` type($in_class_preds) `,` $in_proposals `:`
type($in_proposals) (`,` $in_additional_preds^ `:` type($in_additional_preds))?
(`,` $in_additional_proposals^ `:` type($in_additional_proposals))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.11.1 Attributes:

Attribute	MLIR Type	Description

Attribute	MLIR Type	Description
`attr`	vpux::IE::DetectionOutputAttr	DictionaryAttr with field(s): 'num_classes', 'background_label_id', 'top_k', 'variance_encoded_in_target', 'keep_top_k', 'code_type', 'share_location', 'nms_threshold', 'confidence_threshold', 'clip_after_nms', 'clip_before_nms', 'decrease_label_id', 'normalized', 'input_height', 'input_width', 'objectness_score' (each field having its own constraints)

5.4.2.11.2 Operands:

Operand	Description
`in_box_logits`	memref of floating-point values
`in_class_preds`	memref of floating-point values
`in_proposals`	memref of floating-point values
`in_additional_preds`	memref of floating-point values
`in_additional_proposals`	memref of floating-point values
`output_buff`	memref of floating-point values

5.4.2.11.3 Results:

Result	Description
`output`	memref of floating-point values

5.4.2.12 `IERT.Divide` (vpux::IERT::DivideOp)

InferenceEngine run-time Divide layer

Syntax:

```
operation ::= `IERT.Divide` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.12.1 Operands:

Operand	Description
`input1`	memref of 16-bit float or 32-bit float values
`input2`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.12.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.13 IERT.Elu (vpux::IERT::EluOp)

InferenceEngine run-time Elu layer

Syntax:

```
operation ::= `IERT.Elu` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.13.1 Attributes:

Attribute	MLIR Type	Description
x	::mlir::FloatAttr	64-bit float attribute

5.4.2.13.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.13.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.14 IERT.Erf (vpux::IERT::ErfOp)

InferenceEngie run-time Erf layer

Syntax:

```
operation ::= `IERT.Erf` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.14.1 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.14.2 Results:

Result	Description

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.15 IERT.ExecutorResource (vpux::IERT::ExecutorResourceOp)

Information about executor resource

Syntax:

```
operation ::= `IERT.ExecutorResource` attr-dict
              $count `of` $kind
              $subExecutors
```

The executor resource is defined by the following attributes:

- Kind - optional kind of the executor.
- Count - number of executor units.

5.4.2.15.1 Attributes:

Attribute	MLIR Type	Description
kind	::mlir::Attribute	any attribute
count	mlir::IntegerAttr	Integer attribute

5.4.2.16 IERT.Exp (vpux::IERT::ExpOp)

InferenceEngine run-time Exp layer

Syntax:

```
operation ::= `IERT.Exp` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.16.1 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.16.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.17 IERT.FakeQuantize (vpux::IERT::FakeQuantizeOp)

InferenceEngine FakeQuantize layer

Syntax:

```
operation ::= `IERT.FakeQuantize` attr-dict
              `inputs` `(` $input `:` type($input) `,` $input_low `:`
type($input_low) `,` $input_high `:` type($input_high) `,` $output_low `:`
type($output_low) `,` $output_high `:` type($output_high) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.17.1 Attributes:

Attribute	MLIR Type	Description
`levels`	mlir::IntegerAttr	Integer attribute

5.4.2.17.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`input_low`	memref of 16-bit float or 32-bit float values
`input_high`	memref of 16-bit float or 32-bit float values
`output_low`	memref of 16-bit float or 32-bit float values
`output_high`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.17.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.18 `IERT.FloorMod` (vpux::IERT::FloorModOp)

InferenceEngine run-time FloorMod layer

Syntax:

```
operation ::= `IERT.FloorMod` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.18.1 Operands:

Operand	Description
`input1`	memref of 16-bit float or 32-bit float values
`input2`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.18.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.19 `IERT.Floor` (vpux::IERT::FloorOp)

InferenceEngine run-time Floor layer

Syntax:

```
operation ::= `IERT.Floor` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.19.1 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.19.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.20 `IERT.FullyConnected` (vpux::IERT::FullyConnectedOp)

InferenceEngine run-time FullyConnected layer

Syntax:

```
operation ::= `IERT.FullyConnected` attr-dict
              `inputs` `(` $input `:` type($input) `,` $weights `:`
type($weights) (`,` $bias^ `:` type($bias))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.20.1 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`weights`	memref of 16-bit float or 32-bit float values
`bias`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.20.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.21 `IERT.GRN` (vpux::IERT::GRNOp)

InferenceEngine run-time GRN layer

Syntax:

```
operation ::= `IERT.GRN` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.21.1 Attributes:

Attribute	MLIR Type	Description
bias	::mlir::FloatAttr	64-bit float attribute

5.4.2.21.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.21.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.22 IERT.GenericReshape (vpux::IERT::GenericReshapeOp)

InferenceEngine run-time generic Reshape layer

Syntax:

```
operation ::= `IERT.GenericReshape` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `->` type(results)
```

5.4.2.22.1 Operands:

Operand	Description
input	strided memref of any type values

5.4.2.22.2 Results:

Result	Description
output	strided memref of any type values

5.4.2.23 IERT.GroupConvolution (vpux::IERT::GroupConvolutionOp)

InferenceEngine run-time GroupConvolution layer

Syntax:

```
operation ::= `IERT.GroupConvolution` attr-dict
              `inputs` `(` $input `:` type($input) `,` $filter `:`
type($filter) (`,` $bias^ `:` type($bias))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.23.1 Attributes:

Attribute	MLIR Type	Description
`strides`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_begin`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end`	::mlir::ArrayAttr	64-bit integer array attribute
`dilations`	::mlir::ArrayAttr	64-bit integer array attribute
`groups`	mlir::IntegerAttr	Integer attribute
`post_op`	vpux::IE::PostOp	DictionaryAttr with field(s): 'name', 'attrs' (each field having its own constraints)

5.4.2.23.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`filter`	memref of 16-bit float or 32-bit float values
`bias`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.23.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.24 `IERT.HSwish` (vpux::IERT::HSwishOp)

InferenceEngine run-time HSwish layer

Syntax:

```
operation ::= `IERT.HSwish` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.24.1 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.24.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.25 `IERT.ImplicitReorder` (vpux::IERT::ImplicitReorderOp)

Compile-time reorder layer

Syntax:

```
operation ::= `IERT.ImplicitReorder` attr-dict
              `inputs` `(` $source `:` type($source) `)`
              `->` type(results)
```

5.4.2.25.1 Attributes:

Attribute	MLIR Type	Description
dstOrder	::mlir::AffineMapAttr	AffineMap attribute

5.4.2.25.2 Operands:

Operand	Description
source	memref of any type values

5.4.2.25.3 Results:

Result	Description
result	memref of any type values

5.4.2.26 `IERT.Interpolate` (vpux::IERT::InterpolateOp)

InferenceEngine run-time Interpolate layer

Syntax:

```
operation ::= `IERT.Interpolate` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.26.1 Attributes:

Attribute	MLIR Type	Description
mode	vpux::IE::InterpolateModeAttr	Specifies type of interpolation
coord_mode	vpux::IE::InterpolateCoordModeAttr	coordinate_transformation_mode specifies how to transform the coordinate.
nearest_mode	vpux::IE::InterpolateNearestModeAttr	specifies round mode when mode == nearest
antialias	::mlir::UnitAttr	unit attribute

5.4.2.26.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.26.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.27 `IERT.LRN_IE` (vpux::IERT::LRN_IEOp)

InferenceEngine run-time LRN_IE layer

Syntax:

```
operation ::= `IERT.LRN_IE` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `ouputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.27.1 Attributes:

Attribute	MLIR Type	Description
`alpha`	::mlir::FloatAttr	64-bit float attribute
`beta`	::mlir::FloatAttr	64-bit float attribute
`bias`	::mlir::FloatAttr	64-bit float attribute
`size`	mlir::IntegerAttr	Integer attribute
`region`	vpux::IE::LRN_IERegionAttr	LRN_IE region that operations support

5.4.2.27.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.27.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.28 `IERT.LSTMCell` (vpux::IERT::LSTMCellOp)

InferenceEngine run-time LSTMCell layer

Syntax:

```
operation ::= `IERT.LSTMCell` attr-dict
              `inputs` `(` $inputData `:` type($inputData) `,`
$initialHiddenState `:` type($initialHiddenState) `,` $initialCellState `:`
type($initialCellState)
              `,` $weights `:` type($weights) `,` $biases `:` type($biases) `)`
              `outputs` `(` $outputHiddenState_buff `:`
type($outputHiddenState_buff) `,` $outputCellState_buff `:`
type($outputCellState_buff) `)`
              `->` type(results)
```

5.4.2.28.1 Attributes:

Attribute	MLIR Type	Description
`hiddenSize`	mlir::IntegerAttr	Integer attribute

5.4.2.28.2 Operands:

Operand	Description
`inputData`	memref of 16-bit float or 32-bit float values
`initialHiddenState`	memref of 16-bit float or 32-bit float values
`initialCellState`	memref of 16-bit float or 32-bit float values
`weights`	memref of 16-bit float or 32-bit float values
`biases`	memref of 16-bit float or 32-bit float values
`outputHiddenState_buff`	memref of 16-bit float or 32-bit float values
`outputCellState_buff`	memref of 16-bit float or 32-bit float values

5.4.2.28.3 Results:

Result	Description
`outputHiddenState`	memref of 16-bit float or 32-bit float values
`outputCellState`	memref of 16-bit float or 32-bit float values

5.4.2.29 `IERT.LeakyRelu` (vpux::IERT::LeakyReluOp)

InferenceEngine run-time LeakyRelu layer

Syntax:

```
operation ::= `IERT.LeakyRelu` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.29.1 Attributes:

Attribute	MLIR Type	Description
`negative_slope`	::mlir::FloatAttr	64-bit float attribute

5.4.2.29.2 Operands:

Operand	Description

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.29.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.30 `IERT.MVN` (vpux::IERT::MVNOp)

InferenceEngine Run-Time MVN layer

Syntax:

```
operation ::= `IERT.MVN` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.30.1 Attributes:

Attribute	MLIR Type	Description
`across_channels`	::mlir::BoolAttr	bool attribute
`normalize_variance`	::mlir::BoolAttr	bool attribute
`eps`	::mlir::FloatAttr	64-bit float attribute

5.4.2.30.2 Operands:

Operand	Description
`input`	memref of any type values
`output_buff`	memref of any type values

5.4.2.30.3 Results:

Result	Description
`output`	memref of any type values

5.4.2.31 `IERT.MaxPool` (vpux::IERT::MaxPoolOp)

InferenceEngine run-time MaxPool layer

Syntax:

```
operation ::= `IERT.MaxPool` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.31.1 Attributes:

Attribute	MLIR Type	Description

Attribute	MLIR Type	Description
kernel_size	::mlir::ArrayAttr	64-bit integer array attribute
strides	::mlir::ArrayAttr	64-bit integer array attribute
pads_begin	::mlir::ArrayAttr	64-bit integer array attribute
pads_end	::mlir::ArrayAttr	64-bit integer array attribute
post_op	vpux::IE::PostOp	DictionaryAttr with field(s): 'name', 'attrs' (each field having its own constraints)

5.4.2.31.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float or QuantizedType values
output_buff	memref of 16-bit float or 32-bit float or QuantizedType values

5.4.2.31.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float or QuantizedType values

5.4.2.32 `IERT.Maximum` (vpux::IERT::MaximumOp)

InferenceEngine run-time Maximum layer

Syntax:

```
operation ::= `IERT.Maximum` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.32.1 Operands:

Operand	Description
input1	memref of 16-bit float or 32-bit float values

Operand	Description
input2	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.32.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.33 `IERT.MemoryResource` (vpux::IERT::MemoryResourceOp)

Information about memory resource

Syntax:

```
operation ::= `IERT.MemoryResource` $byteSize `bytes` (`of` $kind^)?
              attr-dict
```

The memory resource is defined by the following attributes:

- Kind - optional kind of memory space.
- Size - size in bytes of memory space.

5.4.2.33.1 Attributes:

Attribute	MLIR Type	Description
kind	::mlir::Attribute	any attribute
byteSize	mlir::IntegerAttr	Integer attribute

5.4.2.34 `IERT.Minimum` (vpux::IERT::MinimumOp)

InferenceEngine run-time Minimum layer

Syntax:

```
operation ::= `IERT.Minimum` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.34.1 Operands:

Operand	Description
input1	memref of 16-bit float or 32-bit float values
input2	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.34.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.35 `IERT.Mish` (vpux::IERT::MishOp)

InferenceEngine run-time Mish layer

Syntax:

```
operation ::= `IERT.Mish` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.35.1 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.35.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.36 `IERT.Multiply` (vpux::IERT::MultiplyOp)

InferenceEngine run-time Multiply layer

Syntax:

```
operation ::= `IERT.Multiply` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.36.1 Operands:

Operand	Description
`input1`	memref of 16-bit float or 32-bit float values
`input2`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.36.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.37 `IERT.Negative` (vpux::IERT::NegativeOp)

InferenceEngine run-time Negative layer

Syntax:

```
operation ::= `IERT.Negative` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.37.1 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.37.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.38 IERT.PRelu (vpux::IERT::PReluOp)

InferenceEngine run-time PRelu layer

Syntax:

```
operation ::= `IERT.PRelu` attr-dict
              `inputs` `(` $input `:` type($input) `,` $negative_slope `:`
type($negative_slope) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.38.1 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
negative_slope	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.38.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.39 IERT.Pad (vpux::IERT::PadOp)

InferenceEngine run-time Pad layer

Syntax:

```
operation ::= `IERT.Pad` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.39.1 Attributes:

Attribute	MLIR Type	Description
`pads_begin`	::mlir::ArrayAttr	64-bit integer array attribute
`pads_end`	::mlir::ArrayAttr	64-bit integer array attribute
`pad_value`	::mlir::FloatAttr	64-bit float attribute
`mode`	vpux::IE::PadModeAttr	TPadMode that the InferenceEngine supports

5.4.2.39.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.39.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.40 `IERT.PerAxisTile` (vpux::IERT::PerAxisTileOp)

InferenceEngine run-time per axis Tile layer

Syntax:

```
operation ::= `IERT.PerAxisTile` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.40.1 Attributes:

Attribute	MLIR Type	Description
`axis`	mlir::IntegerAttr	Integer attribute
`tiles`	mlir::IntegerAttr	Integer attribute

5.4.2.40.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.40.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.41 `IERT.Power` (vpux::IERT::PowerOp)

InferenceEngine run-time Power layer

Syntax:

```
operation ::= `IERT.Power` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.41.1 Operands:

Operand	Description
input1	memref of 16-bit float or 32-bit float values
input2	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.41.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.42 IERT.Proposal (vpux::IERT::ProposalOp)

InferenceEngine run-time Proposal layer

Syntax:

```
operation ::= `IERT.Proposal` attr-dict
              `inputs` `(` $class_probs `:` type($class_probs) `,` $bbox_deltas
`:` type($bbox_deltas) `,` $image_shape `:` type($image_shape) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.42.1 Attributes:

Attribute	MLIR Type	Description
proposal_attrs	vpux::IE::ProposalAttr	DictionaryAttr with field(s): 'baseSize', 'preNmsTopN', 'postNmsTopN', 'nmsThresh', 'featStride', 'minSize', 'ratio', 'scale', 'clipBeforeNms', 'clipAfterNms', 'normalize', 'boxSizeScale', 'boxCoordinateScale', 'framework', 'inferProbs' (each field having its own constraints)

5.4.2.42.2 Operands:

Operand	Description

Operand	Description
class_probs	memref of 16-bit float or 32-bit float values
bbox_deltas	memref of 16-bit float or 32-bit float values
image_shape	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.42.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.43 IERT.Quantize (vpux::IERT::QuantizeOp)

InferenceEngine run-time Quantize layer

Syntax:

```
operation ::= `IERT.Quantize` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.43.1 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of QuantizedType values

5.4.2.43.2 Results:

Result	Description
output	memref of QuantizedType values

5.4.2.44 IERT.ROIPooling (vpux::IERT::ROIPoolingOp)

InferenceEngine run-time ROIPooling layer

Syntax:

```
operation ::= `IERT.ROIPooling` attr-dict
              `inputs` `(` $input `:` type($input) `,` $coords `:`
type($coords) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.44.1 Attributes:

Attribute	MLIR Type	Description
output_size	::mlir::ArrayAttr	64-bit integer array attribute
spatial_scale	::mlir::FloatAttr	64-bit float attribute

Attribute	MLIR Type	Description
`method`	vpux::IE::ROIPoolingMethodAttr	ROIPoolingMethod that the InferenceEngine supports

5.4.2.44.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`coords`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.44.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.45 `IERT.ReLU` (vpux::IERT::ReLUOp)

InferenceEngine run-time ReLU layer

Syntax:

```
operation ::= `IERT.ReLU` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.45.1 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.45.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.46 `IERT.RegionYolo` (vpux::IERT::RegionYoloOp)

InferenceEngine run-time RegionYolo layer

Syntax:

```
operation ::= `IERT.RegionYolo` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.46.1 Attributes:

Attribute	MLIR Type	Description
coords	mlir::IntegerAttr	Integer attribute
classes	mlir::IntegerAttr	Integer attribute
regions	mlir::IntegerAttr	Integer attribute
do_softmax	::mlir::BoolAttr	bool attribute
mask	::mlir::ArrayAttr	64-bit integer array attribute
axis	mlir::IntegerAttr	Integer attribute
end_axis	mlir::IntegerAttr	Integer attribute
anchors	::mlir::ArrayAttr	64-bit float array attribute

5.4.2.46.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.46.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.47 IERT.Reorder (vpux::IERT::ReorderOp)

InferenceEngine run-time Reorder layer

Syntax:

```
operation ::= `IERT.Reorder` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.47.1 Operands:

Operand	Description
input	memref of any type values
output_buff	memref of any type values

5.4.2.47.2 Results:

Result	Description
output	memref of any type values

5.4.2.48 IERT.RunTimeResources (vpux::IERT::RunTimeResourcesOp)

Definition of run-time resources

Syntax:

```
operation ::= `IERT.RunTimeResources` attr-dict
              `availableMemory` `:` $availableMemory
              `usedMemory` `:` $usedMemory
              `executors` `:` $executors
```

This operation defines various resources consumed at run-time:

- Available memory spaces for interal buffers.
- Used memory spaces for interal buffers.
- Executors for asynchronous calls.

5.4.2.49 `IERT.ScaleShift` (vpux::IERT::ScaleShiftOp)

InferenceEngine run-time ScaleShift layer

Syntax:

```
operation ::= `IERT.ScaleShift` attr-dict
              `inputs` `(` $input `:` type($input) (`,` $weights^ `:`
type($weights))? (`,` $biases^ `:` type($biases))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.49.1 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`weights`	memref of 16-bit float or 32-bit float values
`biases`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.49.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.50 `IERT.Sigmoid` (vpux::IERT::SigmoidOp)

InferenceEngine run-time Sigmoid layer

Syntax:

```
operation ::= `IERT.Sigmoid` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.50.1 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.50.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.51 IERT.SoftMax (vpux::IERT::SoftMaxOp)

InferenceEngine run-time SoftMax layer

Syntax:

```
operation ::= `IERT.SoftMax` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.51.1 Attributes:

Attribute	MLIR Type	Description
axisInd	mlir::IntegerAttr	Integer attribute

5.4.2.51.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.51.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.52 IERT.SquaredDifference (vpux::IERT::SquaredDifferenceOp)

InferenceEngine run-time SquaredDifference layer

Syntax:

```
operation ::= `IERT.SquaredDifference` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.52.1 Operands:

Operand	Description
input1	memref of 16-bit float or 32-bit float values
input2	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.52.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.53 `IERT.StaticAlloc` (vpux::IERT::StaticAllocOp)

InferenceEngine run-time static buffer allocation

Syntax:

```
operation ::= `IERT.StaticAlloc` `<` $offset `>` attr-dict `->` type(results)
```

5.4.2.53.1 Attributes:

Attribute	MLIR Type	Description
`offset`	mlir::IntegerAttr	Integer attribute

5.4.2.53.2 Results:

Result	Description
`memory`	memref of any type values

5.4.2.54 `IERT.StridedSlice` (vpux::IERT::StridedSliceOp)

InferenceEngine run-time StridedSlice layer

Syntax:

```
operation ::= `IERT.StridedSlice` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.54.1 Attributes:

Attribute	MLIR Type	Description
`begins`	::mlir::ArrayAttr	64-bit integer array attribute
`ends`	::mlir::ArrayAttr	64-bit integer array attribute
`strides`	::mlir::ArrayAttr	64-bit integer array attribute

5.4.2.54.2 Operands:

Operand	Description
`input`	memref of any type values
`output_buff`	memref of any type values

5.4.2.54.3 Results:

Result	Description
`output`	memref of any type values

5.4.2.55 `IERT.SubView` (vpux::IERT::SubViewOp)

Extract single subview from buffer

Syntax:

```
operation ::= `IERT.SubView` $source $static_offsets $static_sizes
($static_strides^)?
              attr-dict `:` type($source) `to` type(results)
```

5.4.2.55.1 Attributes:

Attribute	MLIR Type	Description
`static_offsets`	::mlir::ArrayAttr	64-bit integer array attribute
`static_sizes`	::mlir::ArrayAttr	64-bit integer array attribute
`static_strides`	::mlir::ArrayAttr	64-bit integer array attribute

5.4.2.55.2 Operands:

Operand	Description
`source`	memref of any type values

5.4.2.55.3 Results:

Result	Description
`result`	memref of any type values

5.4.2.56 `IERT.Subtract` (vpux::IERT::SubtractOp)

InferenceEngine run-time Subtract layer

Syntax:

```
operation ::= `IERT.Subtract` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.56.1 Operands:

Operand	Description
`input1`	memref of 16-bit float or 32-bit float values
`input2`	memref of 16-bit float or 32-bit float values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.56.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.57 `IERT.Swish` (vpux::IERT::SwishOp)

InferenceEngine run-time Swish layer

Syntax:

```
operation ::= `IERT.Swish` attr-dict
              `inputs` `(` $input `:` type($input) (`,` $beta^ `:`
type($beta))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.57.1 Attributes:

Attribute	MLIR Type	Description
beta_value	::mlir::FloatAttr	64-bit float attribute

5.4.2.57.2 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
beta	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.57.3 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.58 IERT.Tanh (vpux::IERT::TanhOp)

InferenceEngine run-time Tanh layer

Syntax:

```
operation ::= `IERT.Tanh` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.58.1 Operands:

Operand	Description
input	memref of 16-bit float or 32-bit float values
output_buff	memref of 16-bit float or 32-bit float values

5.4.2.58.2 Results:

Result	Description
output	memref of 16-bit float or 32-bit float values

5.4.2.59 IERT.Tile (vpux::IERT::TileOp)

InferenceEngine run-time Tile layer

Syntax:

```
operation ::= `IERT.Tile` attr-dict
              `inputs` `(` $input `:` type($input) `,` $repeats `:`
type($repeats) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.59.1 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`repeats`	memref of 64-bit signed integer values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.59.2 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.4.2.60 `IERT.Timestamp` (vpux::IERT::TimestampOp)

Get timer timestamp operation

Syntax:

```
operation ::= `IERT.Timestamp` attr-dict `->` type(results)
```

Get timer timestamp operation

5.4.2.60.1 Results:

Result	Description
`output`	statically shaped memref of any type values

5.4.2.61 `IERT.Transpose` (vpux::IERT::TransposeOp)

InferenceEngine run-time Transpose layer

Syntax:

```
operation ::= `IERT.Transpose` attr-dict
              `inputs` `(` $input `:` type($input) (`,` $order^ `:`
type($order))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              `->` type(results)
```

5.4.2.61.1 Attributes:

Attribute	MLIR Type	Description
`order_value`	::mlir::AffineMapAttr	AffineMap attribute

5.4.2.61.2 Operands:

Operand	Description
`input`	memref of 16-bit float or 32-bit float values
`order`	memref of 64-bit signed integer values
`output_buff`	memref of 16-bit float or 32-bit float values

5.4.2.61.3 Results:

Result	Description
`output`	memref of 16-bit float or 32-bit float values

5.5 'VPUIP' Dialect

VPU NN RunTime Dialect The **VPUIP Dialect** represents NN RunTime IR in terms of MLIR framework.

It allows to work with the graph schema inside MLIR framework:

- Validate it.
- Perform additional low level transformations/optimizations.

It handles such VPU-specifics as:

- Memory/executors hierarchy.
- HW barriers notion.
- Supported operation set.

5.5.1 OpInterface definitions

5.5.1.1 TaskOpInterface (`VPUIP_TaskOpInterface`)

Interface for VPUIP Task ##### Methods: ###### `waitBarriers`

```
mlir::ValueRange waitBarriers();
```

Barriers that will free this task to run#### `waitBarriersMutable`

```
mlir::MutableOperandRange waitBarriersMutable();
```

Barriers that will free this task to run#### `updateBarriers`

```
mlir::ValueRange updateBarriers();
```

Barriers that will be at least partially unlocked when this task is complete#### `updateBarriersMutable`

```
mlir::MutableOperandRange updateBarriersMutable();
```

Barriers that will be at least partially unlocked when this task is complete#### `serialize`

```
vpux::VPUIP::BlobWriter::SpecificTask
serialize(vpux::VPUIP::BlobWriter&writer);
```

Serialize the Task to BLOB format NOTE: This method *must* be implemented by the user.

5.5.1.1.0.1 `getTaskType`

```
static vpux::VPUIP::TaskType getTaskType();
```

Get the VPUIP TaskType for the Operation## UPATaskOpInterface (`VPUIP_UPATaskOpInterface`)

Interface for VPUIP UPA Task ##### Methods: ###### `maxShaves`

```
vpux::Optional<int64_t> maxShaves();
```

Get maximal number of UPA SHAVEs to use#### `setMaxShaves`

```
void setMaxShaves(int64_t maxShaves);
```

Update maximal number of UPA SHAVEs to use NOTE: This method *must* be implemented by the user.

5.5.1.1.0.2 `isTrailingSWLayer`

```
bool isTrailingSWLayer();
```

Is current task the trailing SW layer#### `markAsTrailingSWLayer`

```
void markAsTrailingSWLayer();
```

Mark current task as trailing SW layer NOTE: This method *must* be implemented by the user.

[TOC]

5.5.2 Type constraint definition

5.5.2.1 VPUIP Barrier Type

This object represents closely a Barrier in the device ### Operation definition

5.5.2.2 `VPUIP.CTCGreedyDecoderSeqLenUPA` (vpux::VPUIP::CTCGreedyDecoderSeqLenUPAOp)

CTCGreedyDecoderSeqLen UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.CTCGreedyDecoderSeqLenUPA` attr-dict
              `inputs` `(` $input `:` type($input) `,` $sequenceLength `:`
type($sequenceLength) `,` ($blankIndex^ `:` type($blankIndex))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `,`
$outputLength_buff `:` type($outputLength_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.2.1 Attributes:

Attribute	MLIR Type	Description
mergeRepeated	::mlir::UnitAttr	unit attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.2.2 Operands:

Operand	Description
input	memref of 16-bit float values
sequenceLength	memref of 32-bit signed integer values
blankIndex	memref of 32-bit signed integer values
output_buff	memref of 32-bit signed integer values
outputLength_buff	memref of 32-bit signed integer values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.2.3 Results:

Result	Description
output	memref of 32-bit signed integer values
outputLength	memref of 32-bit signed integer values

5.5.2.3 VPUIP.CTCGreedyDecoderUPA (vpux::VPUIP::CTCGreedyDecoderUPAOp)

CTCGreedyDecoder UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.CTCGreedyDecoderUPA` attr-dict
              `inputs` `(` $input `:` type($input) `,` $sequenceLengths `:`
type($sequenceLengths) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.3.1 Attributes:

Attribute	MLIR Type	Description

Attribute	MLIR Type	Description
mergeRepeated	::mlir::UnitAttr	unit attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.3.2 Operands:

Operand	Description
input	memref of 16-bit float values
sequenceLengths	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.3.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.4 VPUIP.ClampUPA (vpux::VPUIP::ClampUPAOp)

Clamp UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.ClampUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.4.1 Attributes:

Attribute	MLIR Type	Description
min	::mlir::FloatAttr	64-bit float attribute
max	::mlir::FloatAttr	64-bit float attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.4.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.4.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.5 `VPUIP.ConfigureBarrier` (vpux::VPUIP::ConfigureBarrierOp)

A task to configure the setup for a barrier

Syntax:

```
operation ::= `VPUIP.ConfigureBarrier` attr-dict
              `<` $id `>`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.5.1 Attributes:

Attribute	MLIR Type	Description
id	mlir::IntegerAttr	Integer attribute

5.5.2.5.2 Operands:

Operand	Description
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.5.3 Results:

Result	Description
barrier	VPUIP Barrier Type

5.5.2.6 `VPUIP.ConvertUPA` (vpux::VPUIP::ConvertUPAOp)

Convert UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.ConvertUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.6.1 Attributes:

Attribute	MLIR Type	Description
scale	::mlir::FloatAttr	64-bit float attribute
bias	::mlir::FloatAttr	64-bit float attribute
fromDetectionOutput	::mlir::UnitAttr	unit attribute
haveBatch	::mlir::UnitAttr	unit attribute
batchID	mlir::IntegerAttr	Integer attribute

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.6.2 Operands:

Operand	Description
input	memref of any type values
output_buff	memref of any type values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.6.3 Results:

Result	Description
output	memref of any type values

5.5.2.7 `VPUIP.ConvolutionUPA` (vpux::VPUIP::ConvolutionUPAOp)

Convolution UPA SHAVE kernel (reference implementation)

Syntax:

```
operation ::= `VPUIP.ConvolutionUPA` attr-dict
              `inputs` `(` $input `:` type($input) `,` $filter `:`
type($filter) (`,` $bias^ `:` type($bias))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.7.1 Attributes:

Attribute	MLIR Type	Description
strides	::mlir::ArrayAttr	64-bit integer array attribute
dilations	::mlir::ArrayAttr	64-bit integer array attribute
padsBegin	::mlir::ArrayAttr	64-bit integer array attribute
padsEnd	::mlir::ArrayAttr	64-bit integer array attribute
groups	mlir::IntegerAttr	Integer attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.7.2 Operands:

Operand	Description
input	memref of 16-bit float values
filter	memref of 16-bit float values
bias	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type

Operand	Description
updateBarriers	VPUIP Barrier Type

5.5.2.7.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.8 VPUIP.DPUTask (vpux::VPUIP::DPUTaskOp)

This object represents workload for a single DPU tile

Syntax:

```
operation ::= `VPUIP.DPUTask` attr-dict
```

5.5.2.8.1 Attributes:

Attribute	MLIR Type	Description
start	::mlir::ArrayAttr	64-bit integer array attribute
end	::mlir::ArrayAttr	64-bit integer array attribute
pad	vpux::VPUIP::PaddingAttr	DictionaryAttr with field(s): 'left', 'right', 'top', 'bottom' (each field having its own constraints)
mpe_mode	vpux::VPUIP::MPEModeAttr	MPE Mode

5.5.2.9 VPUIP.DeclareTensor (vpux::VPUIP::DeclareTensorOp)

TensorReference value declaration

Syntax:

```
operation ::= `VPUIP.DeclareTensor` $locale custom<LocaleIndex>($localeIndex)
`<` $dataIndex `>` attr-dict `->` type(results)
```

5.5.2.9.1 Attributes:

Attribute	MLIR Type	Description
locale	vpux::VPUIP::MemoryLocationAttr	Values indicating which type of memory a tensor resides in

Attribute	MLIR Type	Description
localeIndex	::mlir::ArrayAttr	64-bit integer array attribute
dataIndex	mlir::IntegerAttr	Integer attribute
sparsityIndex	mlir::IntegerAttr	Integer attribute
storageElementIndex	mlir::IntegerAttr	Integer attribute
storageElementSize	mlir::IntegerAttr	Integer attribute
leadingOffset	mlir::IntegerAttr	Integer attribute
trailingOffset	mlir::IntegerAttr	Integer attribute

5.5.2.9.2 Results:

Result	Description
memory	memref of any type values

5.5.2.10 VPUIP.DeclareVirtualBarrier (vpux::VPUIP::DeclareVirtualBarrierOp)

VPUIP virtual Barrier declaration

Syntax:

```
operation ::= `VPUIP.DeclareVirtualBarrier` attr-dict `->` type(results)
```

5.5.2.10.1 Results:

Result	Description
barrier	VPUIP Barrier Type

5.5.2.11 VPUIP.DetectionOutputUPA (vpux::VPUIP::DetectionOutputUPAOp)

DetectionOutput UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.DetectionOutputUPA` attr-dict
              `inputs` `(` $in_box_logits `:` type($in_box_logits) `,`
$in_class_preds `:` type($in_class_preds) `,` $in_proposals `:`
type($in_proposals) (`,` $in_additional_preds^ `:` type($in_additional_preds))?
(`,` $in_additional_proposals^ `:` type($in_additional_proposals))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.11.1 Attributes:

Attribute	MLIR Type	Description
`attr`	vpux::IE::DetectionOutputAttr	DictionaryAttr with field(s): 'num_classes', 'background_label_id', 'top_k', 'variance_encoded_in_target', 'keep_top_k', 'code_type', 'share_location', 'nms_threshold', 'confidence_threshold', 'clip_after_nms', 'clip_before_nms', 'decrease_label_id', 'normalized', 'input_height', 'input_width', 'objectness_score' (each field having its own constraints)
`maxShaves`	mlir::IntegerAttr	Integer attribute
`isTrailingSWLayer`	::mlir::UnitAttr	unit attribute

5.5.2.11.2 Operands:

Operand	Description
`in_box_logits`	memref of 16-bit float values
`in_class_preds`	memref of 16-bit float values
`in_proposals`	memref of 16-bit float values
`in_additional_preds`	memref of 16-bit float values
`in_additional_proposals`	memref of 16-bit float values
`output_buff`	memref of 16-bit float values
`waitBarriers`	VPUIP Barrier Type
`updateBarriers`	VPUIP Barrier Type

5.5.2.11.3 Results:

Result	Description
`output`	memref of 16-bit float values

5.5.2.12 `VPUIP.EltwiseUPA` (vpux::VPUIP::EltwiseUPAOp)

Eltwise UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.EltwiseUPA` attr-dict
              `inputs` `(` $input1 `:` type($input1) `,` $input2 `:`
type($input2) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.12.1 Attributes:

Attribute	MLIR Type	Description
type	vpux::VPUIP::EltwiseLayerTypeAttr	Type of Eltwise layer
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.12.2 Operands:

Operand	Description
input1	memref of 16-bit float values
input2	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.12.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.13 VPUIP.EluUPA (vpux::VPUIP::EluUPAOp)

Elu UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.EluUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.13.1 Attributes:

Attribute	MLIR Type	Description
x	::mlir::FloatAttr	64-bit float attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.13.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.13.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.14 VPUIP.Empty (vpux::VPUIP::EmptyOp)

Empty management task

Syntax:

```
operation ::= `VPUIP.Empty` attr-dict
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
```

5.5.2.14.1 Operands:

Operand	Description
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.15 VPUIP.ErfUPA (vpux::VPUIP::ErfUPAOp)

Erf UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.ErfUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.15.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.15.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values

Operand	Description
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.15.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.16 VPUIP.ExpUPA (vpux::VPUIP::ExpUPAOp)

Exp UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.ExpUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.16.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.16.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.16.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.17 VPUIP.FakeQuantizeUPA (vpux::VPUIP::FakeQuantizeUPAOp)

FakeQuantize UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.FakeQuantizeUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.17.1 Attributes:

Attribute	MLIR Type	Description
`levels`	mlir::IntegerAttr	Integer attribute
`input_low`	vpux::Const::ContentAttr	Lazy folded constant content
`input_high`	vpux::Const::ContentAttr	Lazy folded constant content
`output_low`	vpux::Const::ContentAttr	Lazy folded constant content
`output_high`	vpux::Const::ContentAttr	Lazy folded constant content
`maxShaves`	mlir::IntegerAttr	Integer attribute
`isTrailingSWLayer`	::mlir::UnitAttr	unit attribute

5.5.2.17.2 Operands:

Operand	Description
`input`	memref of 16-bit float values
`output_buff`	memref of 16-bit float values
`waitBarriers`	VPUIP Barrier Type
`updateBarriers`	VPUIP Barrier Type

5.5.2.17.3 Results:

Result	Description
`output`	memref of 16-bit float values

5.5.2.18 `VPUIP.FloorUPA` (vpux::VPUIP::FloorUPAOp)

Floor UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.FloorUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.18.1 Attributes:

Attribute	MLIR Type	Description
`maxShaves`	mlir::IntegerAttr	Integer attribute
`isTrailingSWLayer`	::mlir::UnitAttr	unit attribute

5.5.2.18.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.18.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.19 VPUIP.FullyConnectedUPA (vpux::VPUIP::FullyConnectedUPAOp)

FullyConnected UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.FullyConnectedUPA` attr-dict
              `inputs` `(` $input `:` type($input) `,` $weights `:`
type($weights) (`,` $bias^ `:` type($bias))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.19.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.19.2 Operands:

Operand	Description
input	memref of 16-bit float values
weights	memref of 16-bit float values
bias	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.19.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.20 VPUIP.GRNUPA (vpux::VPUIP::GRNUPAOp)

GRN UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.GRNUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.20.1 Attributes:

Attribute	MLIR Type	Description
`bias`	::mlir::FloatAttr	64-bit float attribute
`maxShaves`	mlir::IntegerAttr	Integer attribute
`isTrailingSWLayer`	::mlir::UnitAttr	unit attribute

5.5.2.20.2 Operands:

Operand	Description
`input`	memref of 16-bit float values
`output_buff`	memref of 16-bit float values
`waitBarriers`	VPUIP Barrier Type
`updateBarriers`	VPUIP Barrier Type

5.5.2.20.3 Results:

Result	Description
`output`	memref of 16-bit float values

5.5.2.21 `VPUIP.Graph` (vpux::VPUIP::GraphOp)

The root object for the VPUIP Execution Graph

Syntax:

```
operation ::= `VPUIP.Graph` attr-dict
              `options` `:` $options
              `version` `:` $version
```

5.5.2.21.1 Attributes:

Attribute	MLIR Type	Description
`options`	vpux::VPUIP::ExecutionFlagAttr	Each of these enums' presence informs how the current schedule is configured

Attribute	MLIR Type	Description
`version`	vpux::VPUIP::VersionAttr	DictionaryAttr with field(s): 'majorV', 'minorV', 'patchV', 'hash', 'contextStr' (each field having its own constraints)

5.5.2.22 `VPUIP.HSwishUPA` (vpux::VPUIP::HSwishUPAOp)

HSwish UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.HSwishUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.22.1 Attributes:

Attribute	MLIR Type	Description
`maxShaves`	mlir::IntegerAttr	Integer attribute
`isTrailingSWLayer`	::mlir::UnitAttr	unit attribute

5.5.2.22.2 Operands:

Operand	Description
`input`	memref of 16-bit float values
`output_buff`	memref of 16-bit float values
`waitBarriers`	VPUIP Barrier Type
`updateBarriers`	VPUIP Barrier Type

5.5.2.22.3 Results:

Result	Description
`output`	memref of 16-bit float values

5.5.2.23 `VPUIP.Interpolate` (vpux::VPUIP::InterpolateUPAOp)

Interpolate UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.Interpolate` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.23.1 Attributes:

Attribute	MLIR Type	Description
mode	vpux::IE::InterpolateModeAttr	Specifies type of interpolation
coord_mode	vpux::IE::InterpolateCoordModeAttr	coordinate_transformation_mode specifies how to transform the coordinate.
nearest_mode	vpux::IE::InterpolateNearestModeAttr	specifies round mode when mode == nearest
antialias	::mlir::UnitAttr	unit attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.23.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.23.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.24 VPUIP.LSTMCellUPA (vpux::VPUIP::LSTMCellUPAOp)

LSTMCell UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.LSTMCellUPA` attr-dict
              `inputs` `(` $inputData `:` type($inputData) `,`
$initialHiddenState `:` type($initialHiddenState)
              `,` $initialCellState `:` type($initialCellState) `,` $weights
`:` type($weights) `,` $biases `:` type($biases) `)`
              `outputs` `(` $outputHiddenState_buff `:`
type($outputHiddenState_buff)
              `,` $outputCellState_buff `:` type($outputCellState_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.24.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.24.2 Operands:

Operand	Description
inputData	memref of 16-bit float values
initialHiddenState	memref of 16-bit float values
initialCellState	memref of 16-bit float values
weights	memref of 16-bit float values
biases	memref of 16-bit float values
outputHiddenState_buff	memref of 16-bit float values
outputCellState_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.24.3 Results:

Result	Description
outputHiddenState	memref of 16-bit float values
outputCellState	memref of 16-bit float values

5.5.2.25 VPUIP.LeakyReluUPA (vpux::VPUIP::LeakyReluUPAOp)

LeakyRelu UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.LeakyReluUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.25.1 Attributes:

Attribute	MLIR Type	Description
negative_slope	::mlir::FloatAttr	64-bit float attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.25.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.25.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.26 VPUIP.MVNUPA (vpux::VPUIP::MVNUPAOp)

MVN UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.MVNUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.26.1 Attributes:

Attribute	MLIR Type	Description
across_channels	::mlir::BoolAttr	bool attribute
normalize_variance	::mlir::BoolAttr	bool attribute
eps	::mlir::FloatAttr	64-bit float attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.26.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.26.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.27 VPUIP.MishUPA (vpux::VPUIP::MishUPAOp)

Mish UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.MishUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.27.1 Attributes:

Attribute	MLIR Type	Description
`maxShaves`	mlir::IntegerAttr	Integer attribute
`isTrailingSWLayer`	::mlir::UnitAttr	unit attribute

5.5.2.27.2 Operands:

Operand	Description
`input`	memref of 16-bit float values
`output_buff`	memref of 16-bit float values
`waitBarriers`	VPUIP Barrier Type
`updateBarriers`	VPUIP Barrier Type

5.5.2.27.3 Results:

Result	Description
`output`	memref of 16-bit float values

5.5.2.28 `VPUIP.NCEClusterTask` (vpux::VPUIP::NCEClusterTaskOp)

NCE Cluster Task Operation

Syntax:

```
operation ::= `VPUIP.NCEClusterTask` attr-dict
              `input` `(` $input  `:` type($input) `)`
              (`weights` `(` $weights^  `:` type($weights) `)`)?
              (`weight_table` `(` $weight_table^  `:` type($weight_table) `)`)?
              (`activation_window` `(` $activation_window^  `:`
type($activation_window) `)`)?
              `parent_input` `(` $parent_input `:` type($parent_input) `)`
              `parent_output` `(` $parent_output `:` type($parent_output) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
              `variants` `:` $variants
              `PPE` `:` $ppe
```

This operation defines NCE cluster task which describes single cluster of 5 DPUs. It is comprised of two argument categories:

- Variants - describes the attributes for an individual DPU within the cluster.
- Invariants - describes the collective attributes of the cluster.

The variants argument takes on a region argument and up to 5 DPUTaskOps. The invariants take on a variety of argument types.

The NCEClusterTaskOp also supports fixed PPE functions as well as generic PPE instruction lists. The generic PPE instruction list argument needs to be described as a region of PPE supported ops. Single fixed PPE functions and generic PPE instruciton list usage is mutually exclusive.

5.5.2.28.1 Attributes:

Attribute	MLIR Type	Description
task_type	vpux::VPUIP::NCETaskTypeAttr	NCE task type
kernel_size	::mlir::ArrayAttr	64-bit integer array attribute
kernel_strides	::mlir::ArrayAttr	64-bit integer array attribute
kernel_padding	::mlir::ArrayAttr	64-bit integer array attribute
activation_window_channel_length	mlir::IntegerAttr	Integer attribute
is_continued	::mlir::UnitAttr	unit attribute

5.5.2.28.2 Operands:

Operand	Description
input	memref of 16-bit float or QuantizedType values
weights	memref of 16-bit float or QuantizedType values
weight_table	memref of 32-bit signed integer values
activation_window	memref of 8-bit unsigned integer values
parent_input	memref of any type values
parent_output	memref of any type values
output_buff	memref of 16-bit float or QuantizedType values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.28.3 Results:

Result	Description
output	memref of 16-bit float or QuantizedType values

5.5.2.29 VPUIP.NNDMA (vpux::VPUIP::NNDMAOp)

NN DMA task

Syntax:

```
operation ::= `VPUIP.NNDMA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.29.1 Attributes:

Attribute	MLIR Type	Description
compression	::mlir::UnitAttr	unit attribute
port	mlir::IntegerAttr	Integer attribute

5.5.2.29.2 Operands:

Operand	Description
input	memref of any type values
output_buff	memref of any type values

Operand	Description
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.29.3 Results:

Result	Description
output	memref of any type values

5.5.2.30 VPUIP.NegativeUPA (vpux::VPUIP::NegativeUPAOp)

Negative UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.NegativeUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.30.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.30.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.30.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.31 VPUIP.NormUPA (vpux::VPUIP::NormUPAOp)

Norm UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.NormUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.31.1 Attributes:

Attribute	MLIR Type	Description
alpha	::mlir::FloatAttr	64-bit float attribute
beta	::mlir::FloatAttr	64-bit float attribute
bias	::mlir::FloatAttr	64-bit float attribute
local_size	mlir::IntegerAttr	Integer attribute
region	vpux::IE::LRN_IERegionAttr	LRN_IE region that operations support
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.31.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.31.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.32 VPUIP.PPETask (vpux::VPUIP::PPETaskOp)

PPE Type for NCE Task

Syntax:

```
operation ::= `VPUIP.PPETask` $ppe_layer_type attr-dict
```

5.5.2.32.1 Attributes:

Attribute	MLIR Type	Description
ppe_layer_type	vpux::VPUIP::PPELayerTypeAttr	Post Processing Element Type
clamp_low	mlir::IntegerAttr	Integer attribute
clamp_high	mlir::IntegerAttr	Integer attribute
lrelu_mult	mlir::IntegerAttr	Integer attribute
lrelu_shift	mlir::IntegerAttr	Integer attribute

5.5.2.33 VPUIP.PReluUPA (vpux::VPUIP::PReluUPAOp)

PRelu UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.PReluUPA` attr-dict
              `inputs` `(` $input `:` type($input) `,` $negative_slope `:`
type($negative_slope) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.33.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.33.2 Operands:

Operand	Description
input	memref of 16-bit float values
negative_slope	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.33.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.34 VPUIP.Pad (vpux::VPUIP::PadUPAOp)

Pad UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.Pad` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.34.1 Attributes:

Attribute	MLIR Type	Description
pads_begin	::mlir::ArrayAttr	64-bit integer array attribute
pads_end	::mlir::ArrayAttr	64-bit integer array attribute
pad_value	::mlir::FloatAttr	64-bit float attribute

Attribute	MLIR Type	Description
mode	vpux::IE::PadModeAttr	TPadMode that the InferenceEngine supports
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.34.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.34.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.35 `VPUIP.PerAxisTileUPA` (vpux::VPUIP::PerAxisTileUPAOp)

Tile for per axis case UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.PerAxisTileUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.35.1 Attributes:

Attribute	MLIR Type	Description
axis	mlir::IntegerAttr	Integer attribute
tiles	mlir::IntegerAttr	Integer attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.35.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.35.3 Results:

Result	Description

Result	Description
output	memref of 16-bit float values

5.5.2.36 `VPUIP.PermuteUPA` (vpux::VPUIP::PermuteUPAOp)

Permute UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.PermuteUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.36.1 Attributes:

Attribute	MLIR Type	Description
order_value	::mlir::AffineMapAttr	AffineMap attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.36.2 Operands:

Operand	Description
input	memref of any type values
output_buff	memref of any type values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.36.3 Results:

Result	Description
output	memref of any type values

5.5.2.37 `VPUIP.PoolingUPA` (vpux::VPUIP::PoolingUPAOp)

MAX and AVG Pooling UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.PoolingUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.37.1 Attributes:

Attribute	MLIR Type	Description

Attribute	MLIR Type	Description
type	vpux::VPUIP::PoolLayerTypeAttr	Type of Pooling layer
kernel	::mlir::ArrayAttr	64-bit integer array attribute
strides	::mlir::ArrayAttr	64-bit integer array attribute
padsBegin	::mlir::ArrayAttr	64-bit integer array attribute
padsEnd	::mlir::ArrayAttr	64-bit integer array attribute
excludePad	::mlir::UnitAttr	unit attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.37.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.37.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.38 VPUIP.ProposalUPA (vpux::VPUIP::ProposalUPAOp)

Proposal UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.ProposalUPA` attr-dict
              `inputs` `(` $class_probs `:` type($class_probs) `,` $bbox_deltas
`:` type($bbox_deltas) `,` $image_shape `:` type($image_shape) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.38.1 Attributes:

Attribute	MLIR Type	Description

Attribute	MLIR Type	Description
proposal_attrs	vpux::IE::ProposalAttr	DictionaryAttr with field(s): 'baseSize', 'preNmsTopN', 'postNmsTopN', 'nmsThresh', 'featStride', 'minSize', 'ratio', 'scale', 'clipBeforeNms', 'clipAfterNms', 'normalize', 'boxSizeScale', 'boxCoordinateScale', 'framework', 'inferProbs' (each field having its own constraints)
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.38.2 Operands:

Operand	Description
class_probs	memref of 16-bit float values
bbox_deltas	memref of 16-bit float values
image_shape	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.38.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.39 VPUIP.QuantCastUPA (vpux::VPUIP::QuantCastUPAOp)

FakeQuantize UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.QuantCastUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.39.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute

Attribute	MLIR Type	Description
`isTrailingSWLayer`	::mlir::UnitAttr	unit attribute

5.5.2.39.2 Operands:

Operand	Description
`input`	memref of 16-bit float or QuantizedType values
`output_buff`	memref of 16-bit float or QuantizedType values
`waitBarriers`	VPUIP Barrier Type
`updateBarriers`	VPUIP Barrier Type

5.5.2.39.3 Results:

Result	Description
`output`	memref of 16-bit float or QuantizedType values

5.5.2.40 `VPUIP.ROIPoolingUPA` (vpux::VPUIP::ROIPoolingUPAOp)

ROIPooling UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.ROIPoolingUPA` attr-dict
              `inputs` `(` $input `:` type($input) `,` $coords `:`
type($coords) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.40.1 Attributes:

Attribute	MLIR Type	Description
`output_size`	::mlir::ArrayAttr	64-bit integer array attribute
`spatial_scale`	::mlir::FloatAttr	64-bit float attribute
`method`	vpux::IE::ROIPoolingMethodAttr	ROIPoolingMethod that the InferenceEngine supports
`maxShaves`	mlir::IntegerAttr	Integer attribute
`isTrailingSWLayer`	::mlir::UnitAttr	unit attribute

5.5.2.40.2 Operands:

Operand	Description
`input`	memref of 16-bit float values
`coords`	memref of 16-bit float values
`output_buff`	memref of 16-bit float values
`waitBarriers`	VPUIP Barrier Type

Operand	Description
updateBarriers	VPUIP Barrier Type

5.5.2.40.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.41 VPUIP.ReLUUPA (vpux::VPUIP::ReLUUPAOp)

ReLU UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.ReLUUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.41.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.41.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.41.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.42 VPUIP.RegionYoloUPA (vpux::VPUIP::RegionYoloUPAOp)

RegionYolo UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.RegionYoloUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.42.1 Attributes:

Attribute	MLIR Type	Description
coords	mlir::IntegerAttr	Integer attribute
classes	mlir::IntegerAttr	Integer attribute
regions	mlir::IntegerAttr	Integer attribute
do_softmax	::mlir::BoolAttr	bool attribute
mask	::mlir::ArrayAttr	64-bit integer array attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.42.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.42.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.43 `VPUIP.ScaleShiftUPA` (vpux::VPUIP::ScaleShiftUPAOp)

ScaleShift UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.ScaleShiftUPA` attr-dict
              `inputs` `(` $input `:` type($input) (`,` $weights^ `:`
type($weights))? (`,` $biases^ `:` type($biases))? `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.43.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.43.2 Operands:

Operand	Description
input	memref of 16-bit float values
weights	memref of 16-bit float values
biases	memref of 16-bit float values

Operand	Description
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.43.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.44 VPUIP.SigmoidUPA (vpux::VPUIP::SigmoidUPAOp)

Sigmoid UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.SigmoidUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.44.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.44.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.44.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.45 VPUIP.SoftMaxUPA (vpux::VPUIP::SoftMaxUPAOp)

SoftMax UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.SoftMaxUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.45.1 Attributes:

Attribute	MLIR Type	Description
axisInd	mlir::IntegerAttr	Integer attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.45.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.45.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.46 VPUIP.StridedSlice (vpux::VPUIP::StridedSliceUPAOp)

StridedSlice UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.StridedSlice` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.46.1 Attributes:

Attribute	MLIR Type	Description
begins	::mlir::ArrayAttr	64-bit integer array attribute
ends	::mlir::ArrayAttr	64-bit integer array attribute
strides	::mlir::ArrayAttr	64-bit integer array attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.46.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.46.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.47 VPUIP.SwishUPA (vpux::VPUIP::SwishUPAOp)

Swish UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.SwishUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.47.1 Attributes:

Attribute	MLIR Type	Description
beta_value	::mlir::FloatAttr	64-bit float attribute
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.47.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.47.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.48 VPUIP.TanhUPA (vpux::VPUIP::TanhUPAOp)

Tanh UPA SHAVE kernel

Syntax:

```
operation ::= `VPUIP.TanhUPA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.48.1 Attributes:

Attribute	MLIR Type	Description
maxShaves	mlir::IntegerAttr	Integer attribute
isTrailingSWLayer	::mlir::UnitAttr	unit attribute

5.5.2.48.2 Operands:

Operand	Description
input	memref of 16-bit float values
output_buff	memref of 16-bit float values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.48.3 Results:

Result	Description
output	memref of 16-bit float values

5.5.2.49 VPUIP.UPADMA (vpux::VPUIP::UPADMAOp)

UPA DMA task

Syntax:

```
operation ::= `VPUIP.UPADMA` attr-dict
              `inputs` `(` $input `:` type($input) `)`
              `outputs` `(` $output_buff `:` type($output_buff) `)`
              (`waits` `(` $waitBarriers^ `:` type($waitBarriers) `)`)?
              (`updates` `(` $updateBarriers^ `:` type($updateBarriers) `)`)?
              `->` type(results)
```

5.5.2.49.1 Operands:

Operand	Description
input	memref of any type values
output_buff	memref of any type values
waitBarriers	VPUIP Barrier Type
updateBarriers	VPUIP Barrier Type

5.5.2.49.2 Results:

Result	Description
output	memref of any type values

5.5.2.50 `VPUIP.WeightsTableOp` (vpux::VPUIP::WeightsTableOp)

Intermediate task for creating weights table based on the addresses of CMX buffers

Syntax:

```
operation ::= `VPUIP.WeightsTableOp` attr-dict
              `op_input` `(` $op_input  `:` type($op_input) `)`
              `op_output` `(` $op_output  `:` type($op_output) `)`
              (`weights` `(` $weights^  `:` type($weights) `)`)?
              (`bias` `(` $bias^  `:` type($bias) `)`)?
              (`activation_window` `(` $activation_window^  `:`
type($activation_window) `)`)?
              `->` type(results)
```

5.5.2.50.1 Operands:

Operand	Description
`op_input`	memref of 16-bit float or QuantizedType values
`op_output`	memref of 16-bit float or QuantizedType values
`weights`	memref of 16-bit float or QuantizedType values
`bias`	memref of 16-bit float or 32-bit float values
`activation_window`	memref of 8-bit unsigned integer values

5.5.2.50.2 Results:

Result	Description
`output`	memref of 32-bit signed integer values

5.5.3 Type definition

5.5.3.1 BarrierType

VPUIP Barrier Type

This object represents closely a Barrier in the device

6 Passes Details

6.1 Level 3 Passes

6.1.0.0.1 `-adjust-layouts` : Adjust required layouts for all layers

This pass adds the required layouts instead of the default one depending on the layer specification from underlying Dialect. ##### `-convert-avg-pool-to-dw-conv` : Convert AvgPool op to GroupConvolution op The pass is a part of `AdjustForVPU` pipeline.

This pass replaces suitable `AvgPool` operations with `GroupConvolution` operation. ##### `-convert-conv1d-to-conv2d` : Convert Convolution1D and GroupConvolution1D to its 2D variance The pass is a part of `AdjustForVPU` pipeline.

Extends input, filter and output tensors with height = 1. [N, C, W] -> [N, C, 1, W] strides: {2} -> strides: {1, 2} pads_begin: {2} -> pads_begin: {0, 2} pads_end: {2} -> pads_end: {0, 2} dilations: {2} -> dilations: {1, 2} ##### `-convert-fc-to-conv` : Convert FullyConnected op to Convolution operation The pass is a part of `AdjustForVPU` pipeline.

This pass replaces all `FullyConnected` operations with `Convolution` operation. It inserts extra `Reshape` operations to satisfy `Convolution` specification. ##### `-convert-paddings-to-floor-mode` : Convert Convolution and Pooling layers paddings to FLOOR rouding mode The pass is a part of `AdjustForVPU` pipeline.

This pass updates padding attributes for Convolution and Pooling layers. It switches layer rounding mode to FLOOR and updates paddings to satisfy output shape. ##### `-convert-precision-to-fp16` : Convert tensors precision from FP32 to FP16 The pass is a part of `AdjustForVPU` pipeline.

This pass replaces all FP32 tensors with FP16. It updates both function bodies as well as Function signatures. ##### `-convert-scale-shift-depthwise` : Convert Scale-Shift operation to Depthwise Convolution The pass is a part of `HardwareMode` pipeline.

Convert Scale-Shift operation to Depthwise convolution. ##### `-convert-shape-to-4d` : Convert tensors shapes to 4D The pass is a part of `AdjustForVPU` pipeline.

This pass replaces ND tensor with 4D analogues for layers, which has such limitations on VPUIP level. Also this pass replaces ND network inputs and outputs with 4D analogues to overcome runtime limitations. ##### `-convert-tile-to-per-axis-tiles` : Convert tile op by multiple axes to multiple PerAxisTile operations The pass is a part of `AdjustForVPU` pipeline.

This pass replaces all `Tile` op with a set of `PerAxisTile` operations. ##### `-convert-weights-to-u8` : Shift data from a signed range to an unsigned one The pass is a part of `LowPrecision` pipeline.

Pass detects quantized convolution and shifts weights data from a signed range to an unsigned one ##### `-dequantize-const` : Dequantize constant tensors The pass is a part of `LowPrecision` pipeline.

It performs constant folding for `Constant -> quant.dcast` case. The pass is used as a fallback to FP16 computations for the cases, where quantized types where not used by layers. ##### `-expand-activation-channels` : Allign input tensors shape of DPU operation with hardware requirements The pass is a part of `buildHardwareModePipeline` pipeline.

This pass processes operations, which can be compile as a DPU tasks and expands channels number to number divisible by 16 in case they doesn't satisfy hardware requirements ##### `-fuse-post-ops` : Fuse activation functions with tasks that support post-processing The pass is a part of `AdjustForVPU` pipeline.

Fuse activation functions (e.g. ReLU, leaky ReLU) with tasks that support post-processing depending on the compilation mode ##### `-fuse-quantized-ops` : Update quantize/dequantize ops The pass is a part of `LowPrecision` pipeline.

Pass detects pattern quant.dcast -> op -> quant.qcast and converts it into single quantized Op ##### `-handle-asymmetric-strides` : Handle operations with asymmetric strides The pass is a part of `AdjustForVPU` pipeline.

This pass splits operations so that they are able to be infered with symmetric strides on dpu because of hardware limitation. ##### `-merge-fake-quant` : Merge back to FakeQuantize The pass is a part of `LowPrecision` pipeline.

It merges pair `quant.qcast -> quant.dcast` into single `IE.FakeQuantize` . The pass is used as a fallback to FP16 computations for the cases, where quantized types where not used by layers. ##### `-optimize-reorders` : Optimize extra Reorder operations This pass tries to optimize out Reorder operations

for common cases by propagating them from inputs to outputs and merging into layers. ##### `-resolve-strided-slice` : Decouple strided slice to slice + reshape The pass is a part of `AdjustForVPU` pipeline.

It replaces IE::StridedSlice operation with simple StridedSlice and Reshape. ##### `-split-fake-quant` : Splits FakeQuantize The pass is a part of `LowPrecision` pipeline.

It splits `FakeQuantize` operations to `quant.qcast -> quant.dcast` pair. ##### `-use-user-layout` : Use user layouts for entry point function prototype This pass updates the CNNNetwork entry point function prototype and uses user-provided layouts for its operands and results. The pass inserts Reorder operations from/to topology layout. ##### `-use-user-precision` : Use user precisions for entry point function prototype This pass updates the CNNNetwork entry point function prototype and use user-provided precisions for its operands and results. The pass inserts Convert operations from/to topology precisions.

6.2 Level 2 Passes

6.2.0.0.1 `-cmx-tiling` : Tile Operations to the condition that all their I/O fit into CMX

This pass will replace a set of operations with the pattern OP ==> Subview->Copy->Op->Copy->SubView. The condition for the replacment is that for each op the SUM of their I/O tensors to fit into CMX memory ##### `-group-async-execute-ops` : Reduces number of async.execute operations Groups consecutive operations which utilizes the same executor and max resources into same async.execute region ##### `-move-view-ops-into-async-regions` : Moves view-like Operations inside the asynchronous regions which depends on them ##### `-move-wait-result-to-async-block-args` : Moves 'async.await' result usage from 'async.execute' body to it's operands ##### `-optimize-async-deps` : Optimizes dependencies between 'async.execute' operations The pass removes 'async.await' Operations between two consecutive 'async.execute' regions and establish token-based dependencies between 'async.execute' operations. ##### `-profiling-timestamp` : DMA-Timestamp based network profiling This pass add dma-timestamp based network profiling. ##### `-set-internal-memory-space` : Set specific memory space for all internal memory buffers This pass updates all Types for internal memory buffers and sets the specified memory space for them.

6.2.0.0.1.1 Options

```
-memory-space : Memory space to perform allocation
```

6.2.0.0.2 `-static-allocation` : Replace dynamic allocations with static

This pass replaces all dynamic `alloc` / `dealloc` Operations with `IERT.StaticAlloc` . It uses simple LinearScan algorithm.

6.2.0.0.2.1 Options

```
-memory-space : Memory space to perform allocation
```

6.2.0.0.3 `-wrap-into-async-regions` : Wraps layer operations into asynchronous regions

This pass wraps each IERT layer operation into async region preserving linear execution.

6.3 Level 1 Passes

6.3.0.0.1 `-assign-physical-barriers` : Assign physical barriers

This pass replaces virtual barriers with physical barriers and assign IDs to them.

6.3.0.0.1.1 Options

```
-num-barriers : Number of physical barriers, available for usage
```

6.3.0.0.2 `-convert-wtable-op-to-constant` : Convert WeightsTable Operations to IERT.ConstantOp

This pass fills weights table considering the information about the offset in the memory of the weights or activation window. ##### `-dump-statistics-of-task-ops` : Dump the statistics of used Task operations This pass dumps the statistics of used Task operations and makes a report as warning for operations not converted to DPU. ##### `-set-compile-params` : Set compilation parameters related to VPUIP Dialect This pass attaches compilation parameters related to **VPUIP Dialect** to Module attributes and initializes **IERT Dialect** run-time resources information.

6.3.0.0.2.1 Options

```
-vpu-arch          : VPU architecture to compile for
-compilation-mode  : Set compilation mode as reference (ReferenceSW) or hardware
(ReferenceHW)
-num-of-dpu-groups : Number of DPU groups
```

6.4 Conversion Passes

6.4.0.0.1 `-add-buffers-for-net-results` : Add network results in Function parameters

This pass adds buffers to the function parameters to copy the network result to them. In contrast to MLIR standard analogue pass, this pass preserves the function results to utilize use-def chains on bufferized IR. The `return` operation will take an aliases of output buffers from the function arguments. ##### `-bufferize-IE` : Bufferize the IE dialect into the IERT dialect on Function level This pass bufferizes **IE Dialect** into **IERT Dialect**:

- Updates only Function inner regions.
- Doesn't change Function signatures.
- Replaces Layer Operations with IERT analogues.
- Replaces Quant Dialect Operations with IERT analogues. ##### `-bufferize-func-and-return` : Bufferize func/return ops A bufferize pass that bufferizes std.func ops. In contrast to MLIR standard analogue pass, this pass uses vpux::BufferizeTypeConverter to process encoding attribute in mlir::RankedTensorType ##### `-convert-async-ops-to-VPUIP` : Convert Async Dialect Operations to VPUIP Dialect This pass inlines 'async.execute' body to parent Block and replaces '!async.token' based dependencies with VPUIP virtual barriers. ##### `-convert-declarations-to-VPUIP` : Convert declarations (constants and memory buffers) to VPUIP Dialect ##### `-convert-layers-to-VPUIP` : Convert Layers Operations to VPUIP Dialect (UPA and DMA tasks) ##### `-convert-to-nce-ops` : Convert to NCE2 ops Convert ops which can be executed on NCE to explicit NCE ops. ##### `-convert-view-ops-to-VPUIP` : Convert view-like Operations to VPUIP Dialect

6.5 Common Passes

6.5.0.0.1 `-dealloc-placement` : Insert dealloc operations for dynamically allocated memory buffers

This pass supports multi-view Operations in contrast to the MLIR standard pass. ##### `-move-declarations-to-top` : Move all declaration ops to top of parent block ##### `-print-dot` : Convert current MLIR graph to Dot graph Convert current MLIR graph to Dot graph.

6.5.0.0.1.1 Options

```
-declareOp : Print declare memory operations
-constOp   : Print const declare operations
-output    : Path to the output file
-pass      : Print Dot after the pass
```

7 MLIR Framework Details

7.1 Useful Links

- Main web page for the MLIR project: https://mlir.llvm.org/
- Forum: https://llvm.discourse.group/c/mlir/31
- Presentations and talks: https://mlir.llvm.org/talks/

7.1.1 Get Familiar with Project

- Lexicon
- Toy LLVM Tutorial
- Video & slides
- https://llvm.org/devmtg/2019-04/talks.html#Tutorial_1
- https://docs.google.com/presentation/d/11-VjSNNNJoRhPILxFgvtb909it1WNdxTnQFipryfAPU/edit#slide=id.g7d334b12e5_0_4
- https://llvm.org/devmtg/2019-04/talks.html#Keynote_1
- https://storage.googleapis.com/pub-tools-public-publication-data/pdf/1c082b766d8e14b54e36e37c9fc3ebbe8b4a72dd.pdf
- https://www.youtube.com/watch?v=qzljG6DKgic
- https://www.youtube.com/watch?v=Y4SvqTtOIDk
- What is tensor type
- What is memref type

7.1.2 Architecture Details

- Bufferization
- Dialect conversion
- Operations/Types/Attributes Interfaces
- Operations Traits
- Canonicalization
- Operation definition and code generation
- Pass infrastructure

- Pattern rewrite passes:
- https://mlir.llvm.org/docs/PatternRewriter/
- https://mlir.llvm.org/docs/DeclarativeRewrites/
- https://mlir.llvm.org/docs/Tutorials/QuickstartRewrites/
- Custom Dialect types and attributes
- IR traversal
- Symbols

7.1.3 MLIR Standard Dialects

- Common operations
- Control flow
- Asynchronous execution
- Quantization support
- Dynamic shape support
- High-level representation of tensor linear algebra:
- https://mlir.llvm.org/docs/Rationale/RationaleLinalgDialect/
- https://mlir.llvm.org/docs/Dialects/Linalg/
- Polyhedral model
- TOSA

7.1.4 Infrastructure

- Code generation tool
- Pattern-based IR testing:
- Test runner tool
- Check tool

7.1.5 Source Code Location

- Upstream source code
- Fork for VPUX plugin

7.2 MLIR FAQ

7.2.1 Operation Traits and Interfaces

The MLIR has such concepts as **Traits** and **Interfaces**, which are used to describe common properties for the set of **Operations** (they are also applicable to **Types** and **Attributes**). Those concepts differs from the similar common C++ concepts. This section will give detailed overview of this MLIR concepts.

7.2.1.1 Operation Traits

Traits in MLIR have two purposes.

First, they act like some `Boolean` flag, indicating that **Operation** has some specific properties/invariant/behavior. This property can be checked via the following method:

```
mlir::Operation* someOp = ...;
if (someOp->hasTrait<IsCommutative>()) {
    ...
}
```

Also such **Trait** class can implement additional validation hook to check related invariant and those hook will be called by MLIR during IR validation in addition to operation validation hook.

```
template <typename ConcreteType>
class SameOperandsShape : public TraitBase<ConcreteType, SameOperandsShape> {
public:
  static LogicalResult verifyTrait(Operation *op) {
    return impl::verifySameOperandsShape(op);
  }
};
```

Second, the **Trait** class can be used as mixin to add additional API to the operation. Since in C++ **Operation** class actually inherits the **Trait** class, all the methods (both static and non-static) declared in the **Trait** class will be available in **Operation** class. Moreover, the **Trait** class has access to the **Operation** pointer, so the method implementation can get any IR information. This allows to have common functionality with single implementation, which will be shared across several operations.

```
template <typename ConcreteOp>
class RTLayer : public mlir::OpTrait::TraitBase<ConcreteOp, RTLayer> {
public:
    SmallVector<mlir::Value> getInputs() {
        return getRTLayerInOperand(this->getOperation());
    }
};

class SomeOp : public mlir::Op<SomeOp, RTLayer> { // Internally RTLayer will be
used as parent class
};

SomeOp someOp = ...;
for (auto in : someOp.getInputs()) {  // The method is inherited from the Trait
class
    ...
}
```

Note, that the **Trait** class is declared as template and it accepts the actual type of Operation it is bound to:

```
template <typename ConcreteOp>
class SomeTrait : public mlir::OpTrait::TraitBase<ConcreteOp, SomeTrait> {
    void someMethod() {
        // We han safely case generic `getOperation()` to specific `ConcreteOp`
and
        // access its methods
        auto attrVal = mlir::cast<ConcreteOp>(this->getOperation()).someAttr();
    }
};

class SomeOp : public mlir::Op<SomeOp, SomeTrait> {
};

// Skipping some implementation details, it is equivivalient to:
class SomeOp : SomeTrait<SomeOp>
```

This is a known C++ idiom Curiously recurring template pattern - CRTP. MLIR uses it quite widely.

7.2.1.2 Operation Interfaces

The **Interfaces** in MLIR provides access to operation API without dealing with concrete **Operation** type. They are represented as separate classes and have no inheritance connection with **Operation** classes. This is also a known C++ Concept-Model idiom, it can be illustrated with the following example pseudo-code:

```
class InterfaceConcept {
   virtual void someMethod() = 0;
};

template <class ConcreteOp>
class InterfaceModel : InterfaceConcept {
   void someMethod() override {
       // Redirect the call to ConcreteOp method
       static_cast<ConcreteOp*>(getOperation())->someMethod();
   }
}

class SomeOp {
    // Note: the method is not virtual
    void someMethod() {
        // implementation
    }
};

mlir::Operation* someOp = ...; // Assuming this is SomeOp
InterfaceConcept interface = dyn_cast<InterfaceConcept>(someOp); // Internally
it will create InterfaceModel<SomeOp> and return reference to it
interface.someMethod(); // The will perform virtual call of
InterfaceModel<SomeOp>::someMethod, which will redirect it to
SomeOp::someMethod
```

When the **Operation** in MLIR declares that it supports some **Interface**, it must provide implementation for its methods, so the redirection can be done. This can be done manually by adding the implementation to the **Operation** class directly or it can be done via inheriting the method from some **Trait** (as shown in previous section).

Actually, each **Interface** in MLIR has counter-part **Trait** class. When the **Operation** declares that it supports the **Interface**, it actually inherits this **Trait** companion. So, the default implementation for the **Interface** method can be placed there.

On the other hand, MLIR allows to override the method implementation in the **Interface** Model part. But in that case the method will not be available in the **Operation** at all:

```
class InterfaceConcept {
    virtual int someMethod() = 0;
};

template <class ConcreteOp>
class InterfaceModel : InterfaceConcept {
   int someMethod() override {
       // Own implementation
       return 42;
   }
}

class SomeOp : DeclareInterface<Interface> {
    // Note: the Operation doesn't have someMethod
};

SomeOp someOp = ...;
InterfaceConcept interface = dyn_cast<InterfaceConcept>(someOp);

// Can call the method from Interface
interface.someMethod();

// But it it not available from the Operation class
// someOp.someMethod();
```

When the **Interface** is declared in `tblgen` file, developer can optionally provide both default implementation - for the **Trait** class (so the method will be available in the **Operation** class and the **Interface** Model will just do redirection) or to the **Interface** Model implementation only (in that case the method will be available in the **Interface** only).

```
InterfaceMethod<
    "Desciption",
    "void", "someMethod", (ins),
    [{
        // This is the Interface Model implementation only, method will not be
available in the Operation class
    }],
    [{
        // This is implementation inside Trait class, the method will be
available both in Operation and in Interface
    }]
>
```

7.2.2 ValueRange Misuse

MLIR provides a class named `ValueRange` (and corresponding `OpOperandsRange`, `OpResultsRange`). Those types can be misused, which will lead to undefined behavior. Those classes do not own the memory of the range, they just store raw pointer to the range begin inside. If the base range is destroyed, this classes will

be left with dangling pointer and its usage will lead to its dereference. Those might cause various issues, including segmentation faults. For example:

```
mlir::ValueRange range {val1, val2, val3};
// some code
range[1].setType(...);

// {val1, val2, val3} - is std::initializer_list
// It is created on stack as temporary object
// mlir::ValueRange stores its pointer
// std::initializer_list is destroyed (as tempory object)
// Now mlir::ValueRange contains invalid pointer, its dereference is UB
```

Summarizing this, the `ValueRange` class and its analogues should be used carefully. It is safe to use it as function argument (since those variables are temporary by definition).

7.2.3 Operation Optional Operands and Results

MLIR allows to have optional operands and results in **Operation**. They are declared explicitly via wrapping into `Optional<...>` construction in the `tblgen` file. Underneath MLIR generates extra API to handle them.

Internally, **Operation** stores the list of its operands/results in dense format. Missing optional values are not stored at all in this list. They are handled by special mechanism, called operands/results segments. Based on `tblgen` definition MLIR splits the operands range onto segments. For example, for the following case:

```
let operands = [ required1, required2, optional ];
```

there will be 2 segments. The first segment will start from the begging and always have 2 elements - `[required1, required2]`. The second segment - remaining part. If the optional operand is missing the second segment will be empty. MLIR will use this information to return `NULL` from optional accessor.

```
Value optional() {
    auto segment = getODSOperandsSegment(1);
    return segment.empty() ? nullptr : segment[0];
}
```

If there are several optional operands:

```
let operands = [ required, optional1, optional2 ];
```

MLIR will use special **Attribute** to store the segments sizes. Developer have to explicitly add special **Trait** to `tblgen` definition to enable this - `AttrSizedOperandSegments`. By default, MLIR will not print this attribute in ASM form. It will looks like:

```
operand_segments = [
    1, // [required] segment, always == 1
    0, // [optional1] segment, 0 means that optional1 is missing
    1  // [optional2] segment, 1 means that optional2 is present
]
```

MLIR will use this attribute in the above `getODSOperandsSegment` method implementation.