OpenVINO Custom Layer Development Guide

23-June '21

intel®

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

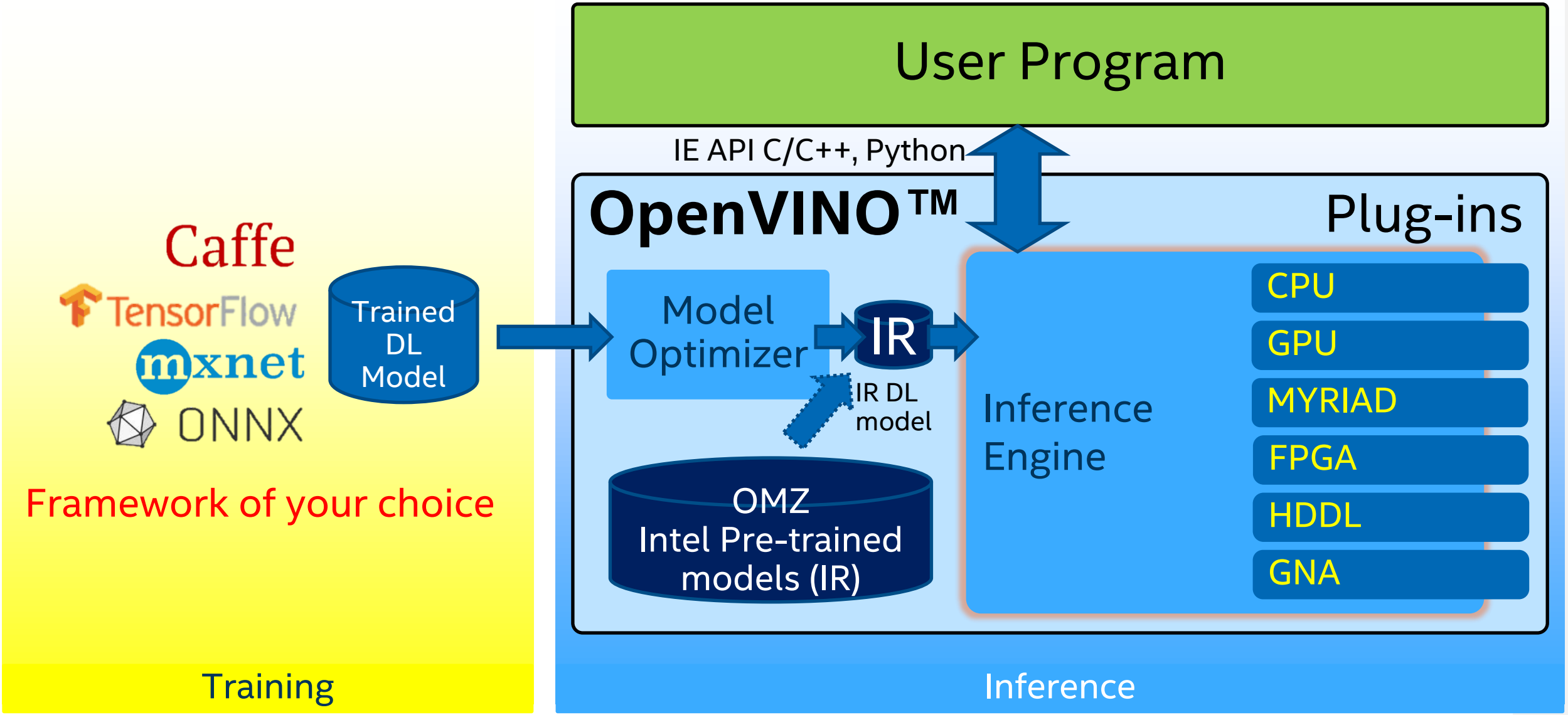
The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Agenda

- OpenVINO Architecture
- Understanding OpenVINO Extension Mechanisms
- MO Extensions
- IE Extensions
- nGraph Op Extension
- OpenVINO sample extension code

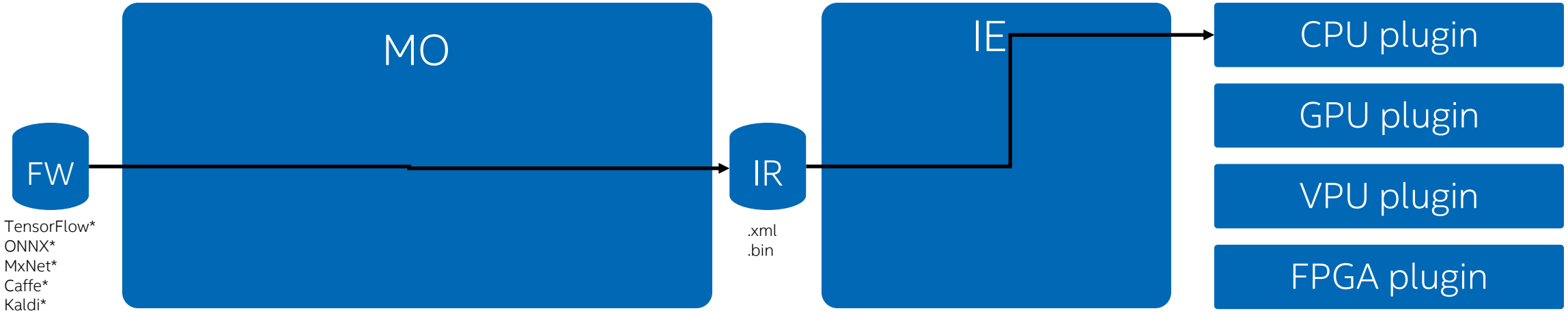
OpenVINO Architecture

OpenVINO Workflow - overview



Myriad: VPU (Vision Processing Unit)
HDDL: High density Deep Learning accelerator
GNA: Gaussian-mixture-model Neural network Accelerator
IR: Intermediate Representation

OpenVINO workflow - shallow

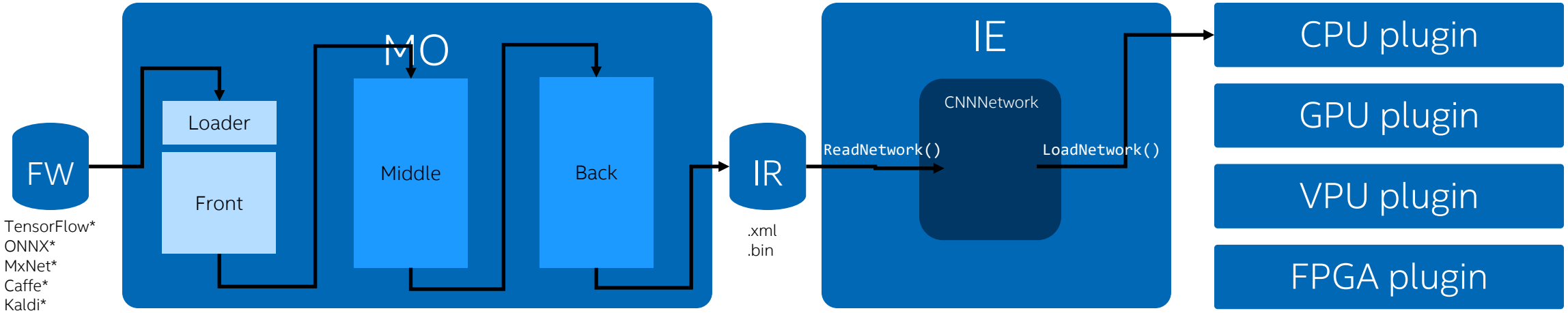


- Load DL model in FW dedicated format
- Extract attributes (parameters) from ops
- Do FW specific optimizations / op fusions (HW agnostic)
- Generate IR model

- Load model in IR (OpenVINO) format
- Schedule and run the model

- Load model to the plugin
- Do HW specific optimizations
- Execute the network / ops

OpenVINO workflow - mid deep

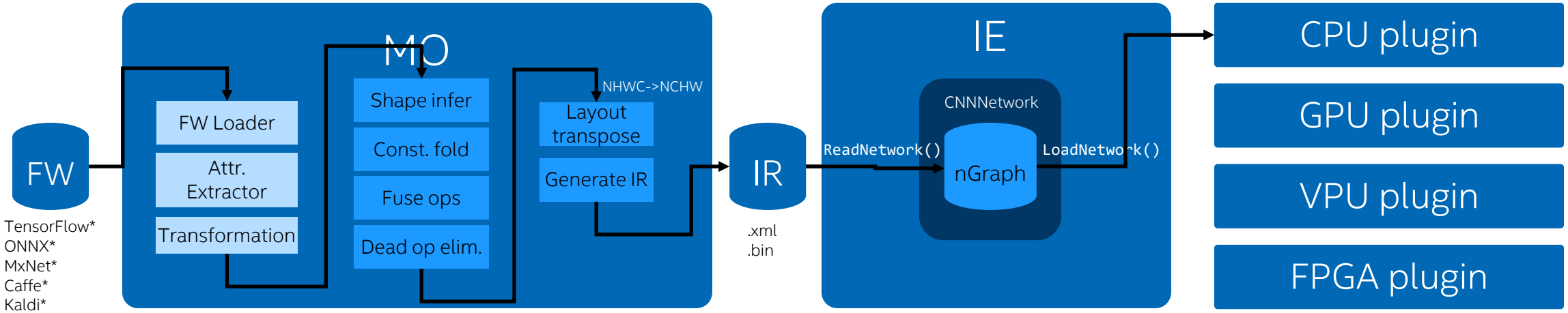


- Loader and Front are FW dependent process
- Middle and Back are FW agnostic / common process

- IR will be loaded and CNNNetwork object will be created

- Plugins receive CNNNetwork, do HW dependent optimization and execute it

OpenVINO workflow - deep

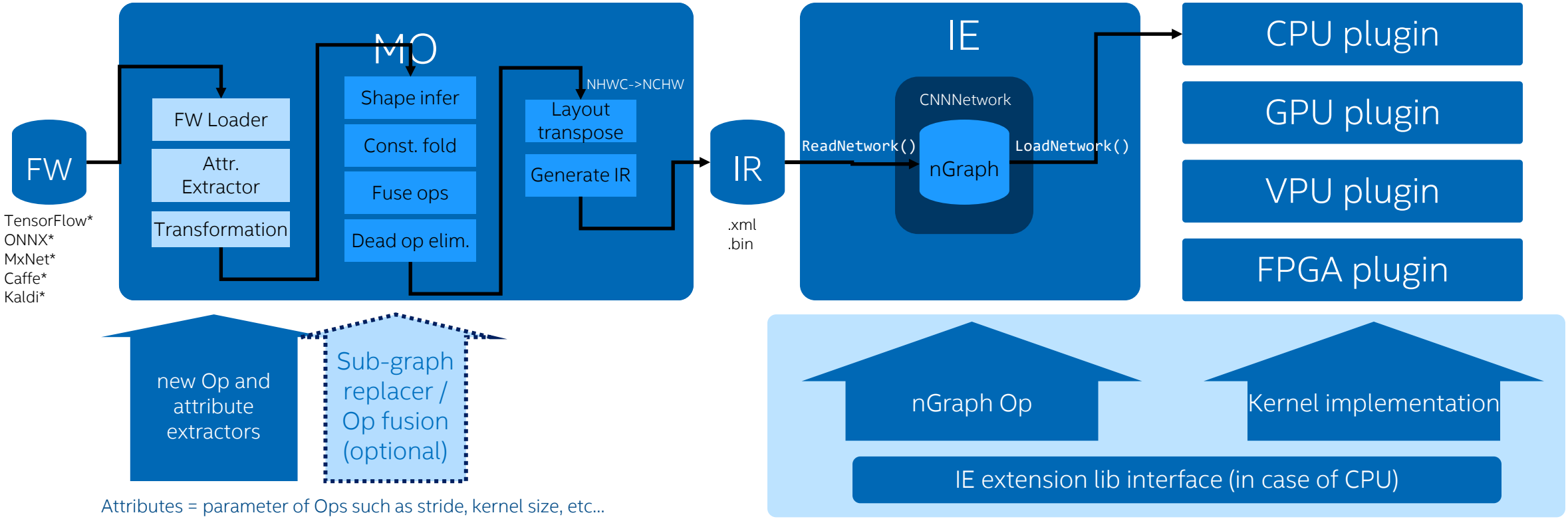


- Load model in IR (OpenVINO) format and build an internal graph object using nGraph

- Load model in nGraph format
- Do HW specific optimizations
- Execute the executable network / ops

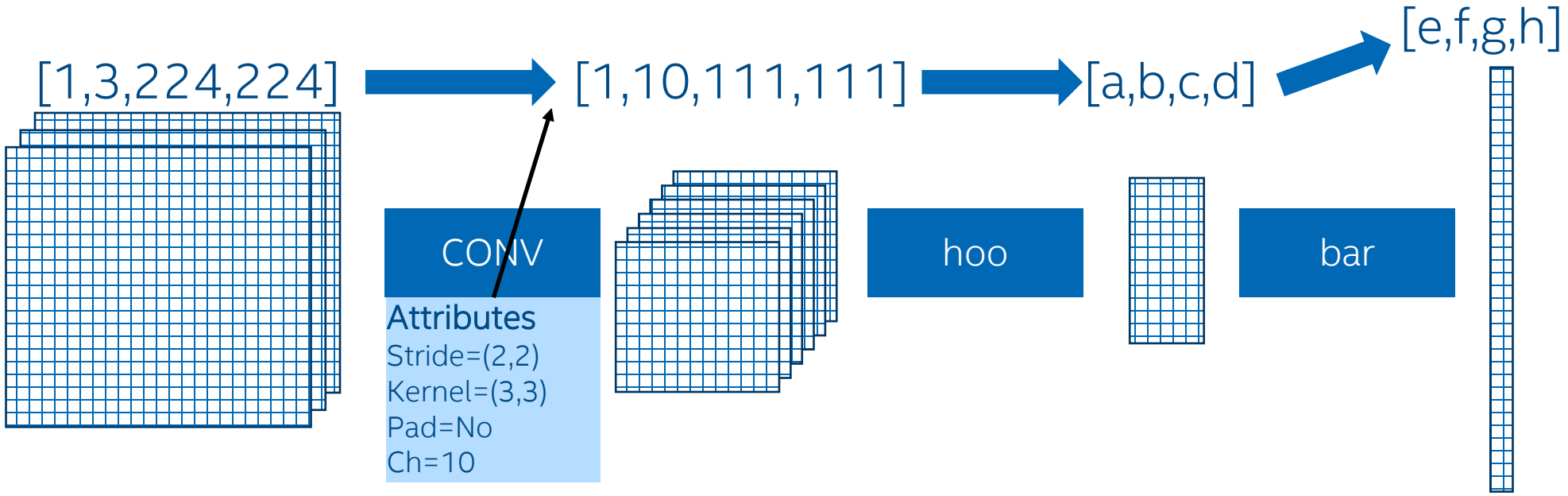
Understanding OpenVINO Extension Mechanisms

What Portion You Need to Develop ?



Attributes = parameter of Ops such as stride, kernel size, etc...

Shape Inference and Attribute Extraction



- Tensor shape will be changed by operation
- Attributes of the operation affects the output shape of the operation
- Shape propagates from input to output
- Custom Ops require to implement shape inference function for correct model conversion
- MO requires attr. extractor so that the generated IR model contains required parameters

Required modules for a new layer support (CPU)

- MO
  - New Op — Value / Shape infer for tensor shape inference / propagation
  - Attr. Extractor — Extract required parameters from Op to generate correct IR model
- IE
  - New nGraph Op — To support the new op in nGraph object in CNNNetwork object
Provides nGraph op skeleton and shape infer function
  - Kernel implementation — Actual tensor processing kernel
  - IE extension library — Implement interface API between IE core and the custom layer
Interface API provides function to register new nGraph Op and query for the kernels in the library

OpenVINO Custom Layer Guide web document

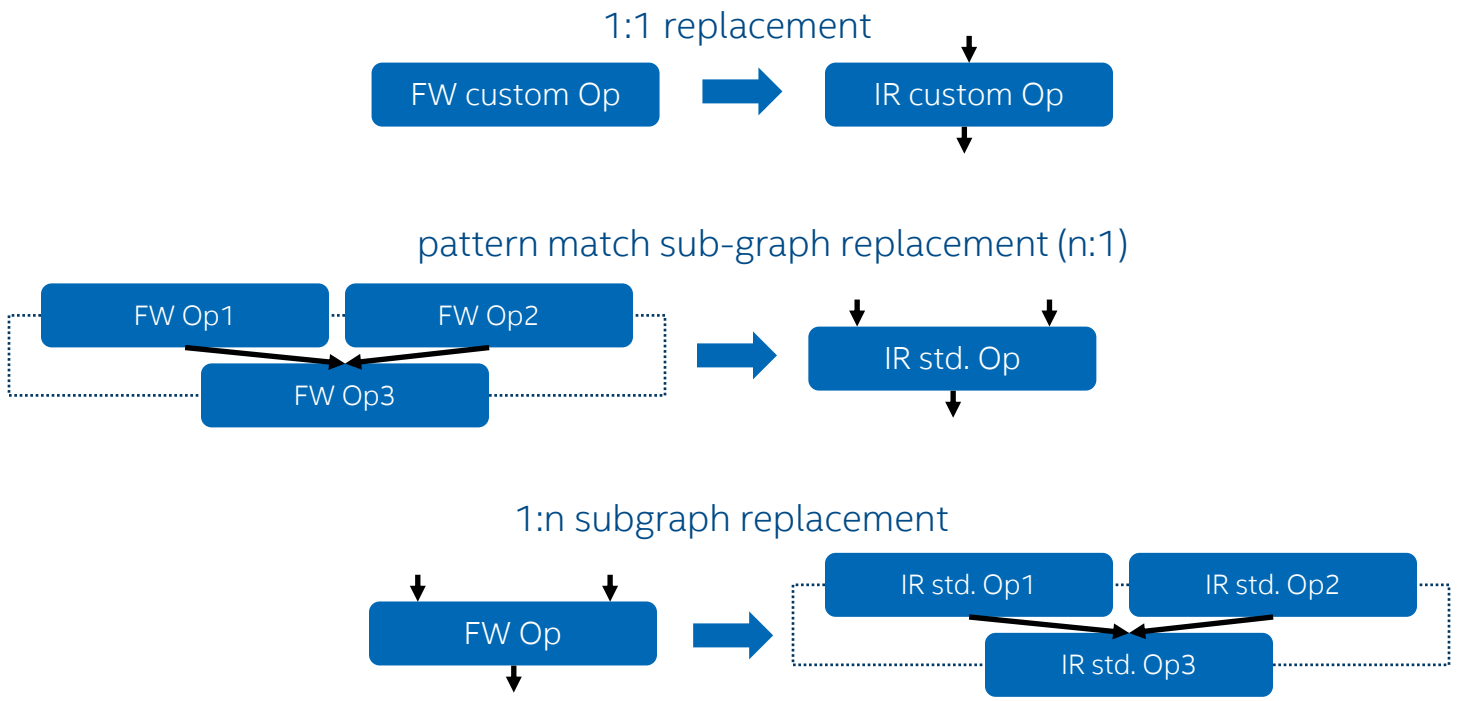
- OpenVINO document page contains holistic "Custom Layer Guide"
  - This uses a model which contains unsupported layer and shows how to develop MO and IE extensions to make it work

- The rest of this presentation uses the same example with a bit detailed explanation and diagrams

- https://docs.openvinotoolkit.org/latest/openvino_docs_HOWTO_Custom_Layers_Guide.html

MO Extensions

Examples of Model Optimizer Extension Forms



- MO supports multiple ways to convert FW specific Ops into IR layers
- Refer to MO extensibility document for more detailed information

MO Extension Directory Structure

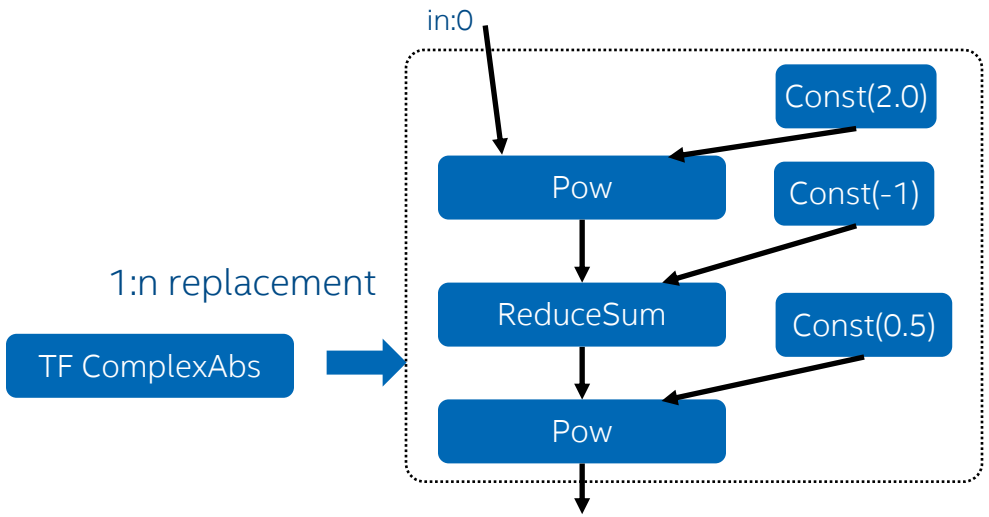
MO Extension files must be placed in a specific directory structure

```
+ extensions        extensions=extension ops
  + load            FW dependent load modules
  + front           FW dependent extractors and transformers
  | | + caffe
  | | + kaldi
  | | + tf          TF dependent attribute extractors and transformers
  | | + mxnet
  | | + onnx
  + middle          FW agnostic transformers
  + back            FW agnostic IR generators
  + ops             New Op for MO
```

You can specify the extension with '**`--extensions`**' option in MO

Example: `python mo.py --input_model model.pb --batch 1 --extensions ./extensions`

Extension example 1 - Op replacement (1:n)



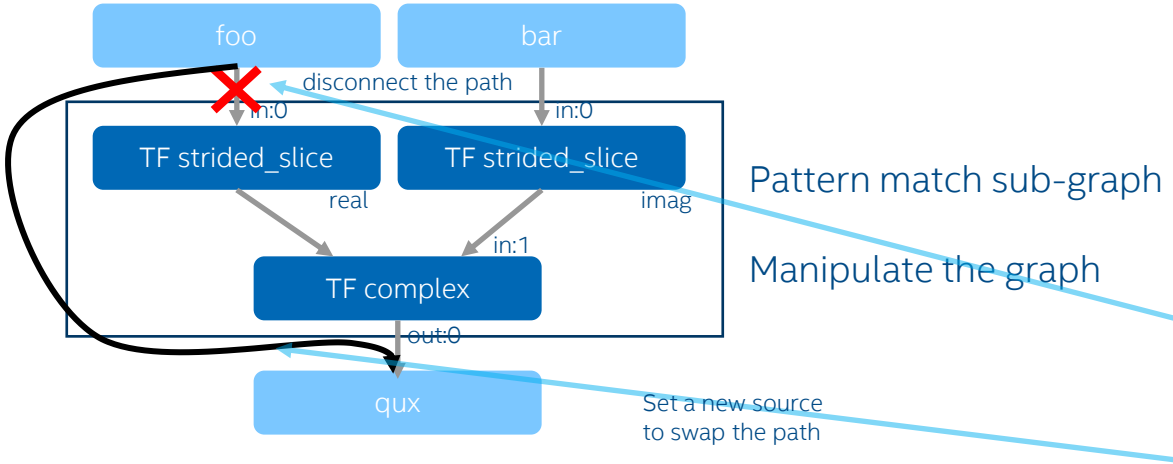
ComplexAbs.py

```
class ComplexAbs(FrontReplacementOp):
    op = "ComplexAbs"
    enabled = True

    def replace_op(self, graph: Graph, node: Node):
        pow_2 = Const(graph, {'value': np.float32(2.0)}).create_node()
        reduce_axis = Const(graph, {'value': np.int32(-1)}).create_node()
        pow_0_5 = Const(graph, {'value': np.float32(0.5)}).create_node()

        sq = Pow(graph, dict(name=node.in_node(0).name + '/sq', power=2.0)).create_node([node.in_node(0), pow_2])
        sum = ReduceSum(graph, dict(name=sq.name + '/sum')).create_node([sq, reduce_axis])
        sqrt = Pow(graph, dict(name=sum.name + '/sqrt', power=0.5)).create_node([sum, pow_0_5])
        return [sqrt.id]
```

From https://docs.openvinotoolkit.org/latest/openvino_docs_HOWTO_Custom_Layers_Guide.html

- This extension replaces a TF Op with a set of IR standard Ops. No new Op is required.

Extension example 2 - Pattern Match Replacement (Graph manipulation)



Pattern match sub-graph

Manipulate the graph

Complex.py

```python
class Complex(FrontReplacementSubgraph):
    enabled = True

    def pattern(self):
        return dict(
            nodes=[
                ('strided_slice_real', dict(op='StridedSlice')),
                ('strided_slice_imag', dict(op='StridedSlice')),
                ('complex', dict(op='Complex')),
            ],
            edges=[
                ('strided_slice_real', 'complex', {'in': 0}),
                ('strided_slice_imag', 'complex', {'in': 1}),
            ])

    @staticmethod
    def replace_sub_graph(graph: Graph, match: dict):
        strided_slice_real = match['strided_slice_real']
        strided_slice_imag = match['strided_slice_imag']
        complex_node = match['complex']

        # make sure that both strided slice operations get the same data as input
        assert strided_slice_real.in_port(0).get_source() == strided_slice_imag.in_port(0).g
et_source()

        # identify the output port of the operation producing datat for strided slice nodes
        input_node_output_port = strided_slice_real.in_port(0).get_source()
        input_node_output_port.disconnect()

        # change the connection so now all consumers of "complex_node" get data from input n
ode of strided slice nodes
        complex_node.out_port(0).get_connection().set_source(input_node_output_port)
```

From https://docs.openvinotoolkit.org/latest/openvino_docs_HOWTO_Custom_Layers_Guide.html

- This expansion just manipulates the original graph. No new Op is required.

Extension example 3 - Defining a new Op to MO (FFT2D, IFFT2D)

Error message from MO

```
[ ERROR ]  List of operations that cannot be converted to Inference Engine IR:
[ ERROR ]      Complex (1)
[ ERROR ]          lambda_2/Complex
[ ERROR ]      IFFT2D (1)
[ ERROR ]          lambda_2/IFFT2D
[ ERROR ]      ComplexAbs (1)
[ ERROR ]          lambda_2/Abs
[ ERROR ]  Part of the nodes was not converted to IR. Stopped.
```

ops/FFT.py

```
from mo.ops.op import Op

class FFT(Op):
    op = 'FFT'
    enabled = False

    def __init__(self, graph: Graph, attrs: dict):
        super().__init__(graph, {
            'type': self.op,
            'op': self.op,
            'version': 'custom_opset',
            'inverse': None,
            'in_ports_count': 1,
            'out_ports_count': 1,
            'infer': copy_shape_infer
        }, attrs)

    def backend_attrs(self):
        return ['inverse']
```

Attributes of the new Op

front/tf/FFT_ext.py

```
from mo.front.extractor import FrontExtractorOp
from ...ops.FFT import FFT

class IFFT2DFrontExtractor(FrontExtractorOp):
    op = 'IFFT2D'
    enabled = True

    @classmethod
    def extract(cls, node):
        attrs = {
            'inverse': 1
        }
        FFT.update_node_stat(node, attrs)
        return cls.enabled
```

<- Updating an attribute

In this example, FFT Op supports both forward and inverse FFT. 'inverse' flag to be used to determine the FFT direction.

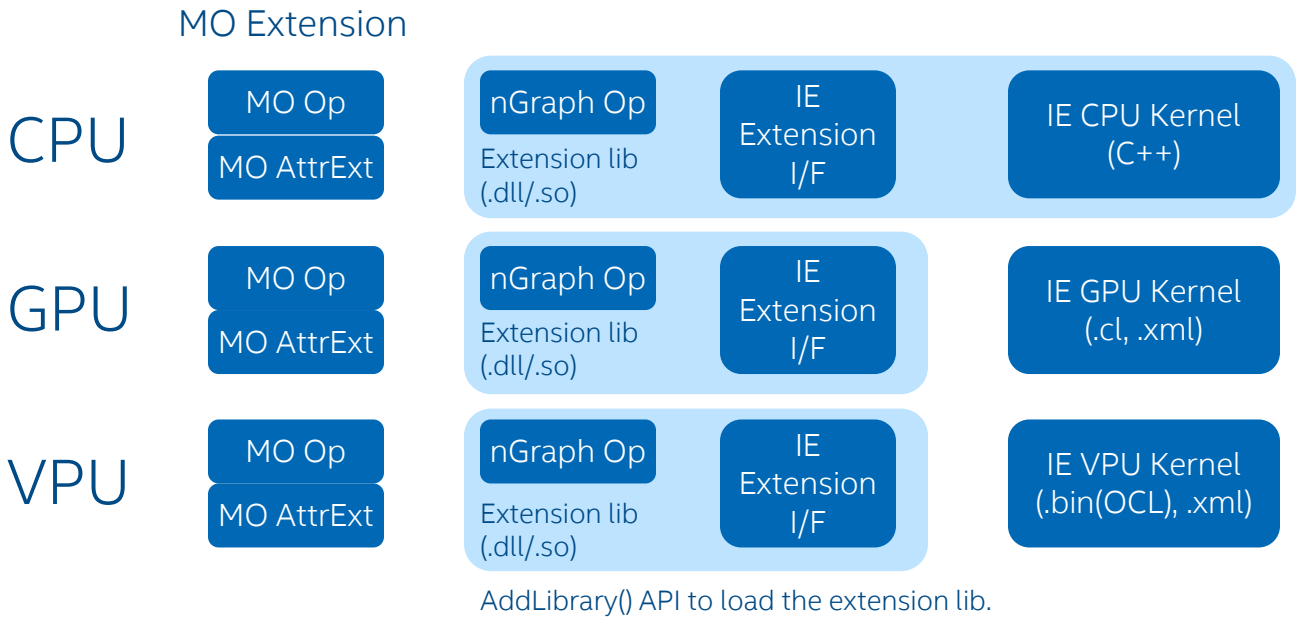
From https://docs.openvinotoolkit.org/latest/openvino_docs_HOWTO_Custom_Layers_Guide.html

- Op defines new 'FFT' Op with default attributes
- In general, the attribute extractor extracts attributes from FW Op. and set the attribute to the nGraph Op.
  - In this extension (extractor), the extension changes the 'inverse' attribute based on the FW Op type
  - 'FFT2D' -> 'FFT' { 'inverse' : 0}
  - 'FFT2D' -> 'FFT' { 'inverse' : 1}
- Extractor extracts necessary attributes from original FW model and set them to the attributes of the new Op

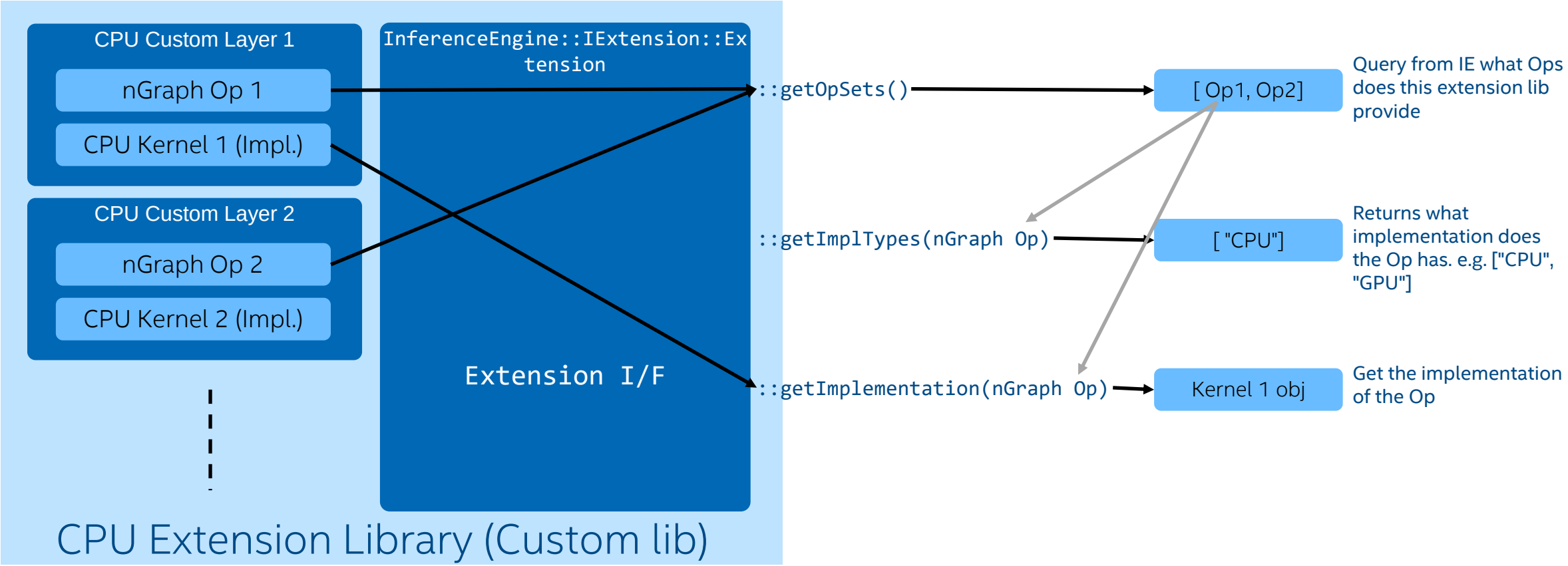
IE Extensions

Difference of Kernel Development by Device

Device	Kernel language	Pre-compile	nGraph Op development	How to develop and use	Guidance doc
CPU	C++	YES	Required	Develop kernel and IE Extension lib I/F in c++ and build shared lib (DLL/.so). Use AddExtension() to load the lib	Here
integrated GPU	OpenCL C	NO	Required	Develop OpenCL source (.cl) and descriptor (.xml). Place files in Inference_engine/bin/intel64/Release or specify the path with SetConfig() API	Here
VPU	OpenCL C	YES	Required	Develop OpenCL code and XML config file. Compile OpenCL code with OpenCL compiler (clc). Use SetConfig() API to load the compiled kernel binary (.bin) and config file (.xml).	Here



Overview of an IE Extension Library for CPU plugin



Template Extension

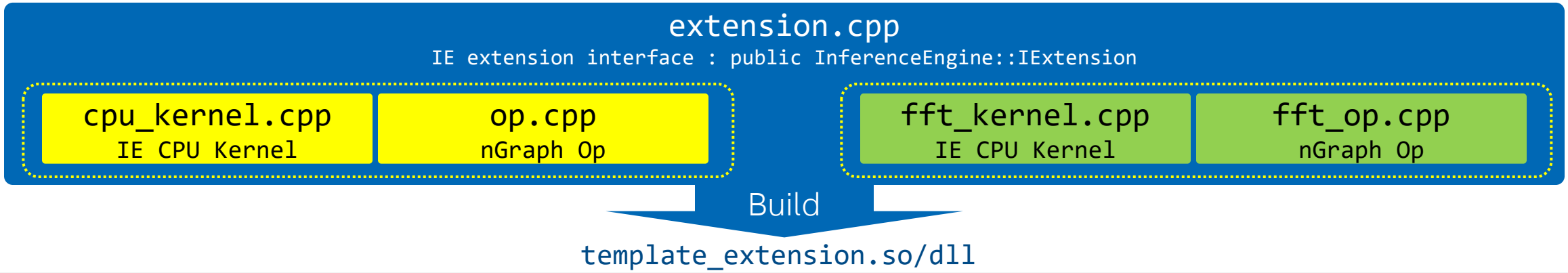
Note: FFT and iFFT ops are going to be supported from OV2021.4. This sample may not work with OV2021.4. Please try this with OV2021.3.

OpenVINO GitHub version contains a CPU extension library example code. The sample code includes 2 CPU kernels and an IE interface class.

https://github.com/openvinotoolkit/openvino/tree/master/docs/template_extension

```
Directory of ~~~\openvino\docs\template_extension
CMakeLists.txt
extension.cpp     Implementation of extension shared library (includes cpu_kernel and fft_kernel)
extension.hpp
cpu_kernel.cpp    Implementation of IE extension (cpu_kernel, offset function(dst_data = src_data + ofst))
cpu_kernel.hpp
op.cpp            Implementation of nGraph Op of (src + ofst) operation
op.hpp
fft_kernel.cpp    Implementation of IE extension (FFT CPU kernel)
fft_kernel.hpp
fft_op.cpp        Implementation of nGraph Op of FFT
fft_op.hpp
```

This CPU kernel is simple and easy to understand == good reference



Extension I/F class in the template_extension

```
namespace TemplateExtension {

class Extension : public InferenceEngine::IExtension {
public:
    Extension();                                                                    // (Optional) Register nGraph Ops for ONNX importer. Empty otherwise
    ~Extension();
    void GetVersion(const InferenceEngine::Version*& versionInfo) const noexcept override;     // Return extension API version and description
    void Unload() noexcept override {}
    void Release() noexcept override { delete this; }                               // Delete an extension library object
    std::map<std::string, ngraph::OpSet> getOpSets() override;                      // Register custom Ops
    std::vector<std::string> getImplTypes(const std::shared_ptr<ngraph::Node>& node) override; // Return supported implementation types for the Op (CPU/GPU/..)
    InferenceEngine::ILayerImpl::Ptr getImplementation(const std::shared_ptr<ngraph::Node>& node,
                                                       const std::string& implType) override;  // Return corresponding implementation (kernel) of the custom Op.
};

}  // namespace TemplateExtension

// Exported function
INFERENCE_EXTENSION_API(InferenceEngine::StatusCode) InferenceEngine::CreateExtension(InferenceEngine::IExtension *&ext,
                                                                                      InferenceEngine::ResponseDesc *resp) noexcept {
    try {
        ext = new Extension();              // Create an instance of this extension and return it
        return OK;
    } catch (std::exception &ex) {
        if (resp) { }                       // Error handling
        return InferenceEngine::GENERAL_ERROR;
    }
}
```

(Reference) IE Extension (CPU) run-time call order

`[TRACE] Extension::CreateExtension()`	Called on library loading
`[TRACE] Extension::Extension()`	Create Extension I/F class instance
`[TRACE] Extension::getOpSets()`	Register Ops
`[TRACE] Extension::getOpSets()`	
`[TRACE] Extension::getImplTypes(FFT_Op)`	Check supported device of the Op
`[TRACE] Extension::getImplementation(FFT_Op)`	Get actual kernel object of the Op

CPU Kernel example: cpu_kernel

cpu_kernel.hpp

```
class OpImplementation : public InferenceEngine::ILayerExecImpl {
public:
    explicit OpImplementation(const std::shared_ptr<ngraph::Node>& node);
    InferenceEngine::StatusCode getSupportedConfigurations(std::vector<InferenceEngine::LayerConfig> &conf, InferenceEngine::ResponseDesc *resp) noexcept override;
    InferenceEngine::StatusCode init(InferenceEngine::LayerConfig &config, InferenceEngine::ResponseDesc *resp) noexcept override;
    InferenceEngine::StatusCode execute(std::vector<InferenceEngine::Blob::Ptr> &inputs, std::vector<InferenceEngine::Blob::Ptr> &outputs,
                                        InferenceEngine::ResponseDesc *resp) noexcept override;
private:
    int64_t add;
    ngraph::Shape inShape;
    ngraph::Shape outShape;
    std::string error;
};
```

cpu_kernel.cpp

```
OpImplementation::OpImplementation(const std::shared_ptr<ngraph::Node> &node) { ... }

InferenceEngine::StatusCode OpImplementation::getSupportedConfigurations(std::vector<InferenceEngine::LayerConfig> &conf, InferenceEngine::ResponseDesc *resp) noexcept { ... }

InferenceEngine::StatusCode OpImplementation::init(InferenceEngine::LayerConfig &config, InferenceEngine::ResponseDesc *resp) noexcept { ... }

InferenceEngine::StatusCode OpImplementation::execute(std::vector<InferenceEngine::Blob::Ptr> &inputs,
                                                       std::vector<InferenceEngine::Blob::Ptr> &outputs,
                                                       InferenceEngine::ResponseDesc *resp) noexcept {
    const float* src_data = inputs[0]->cbuffer().as<const float *>() + inputs[0]->getTensorDesc().getBlockingDesc().getOffsetPadding();
    float *dst_data = outputs[0]->buffer().as<float *>() + outputs[0]->getTensorDesc().getBlockingDesc().getOffsetPadding();

    for (size_t i = 0; i < inputs[0]->size(); i++) {
        dst_data[i] = src_data[i] + add;
    }
    return InferenceEngine::OK;
}
```

Actual kernel code in CPU extension

nGraph Op Expansion

nGraph Op example: cpu_kernel

op.hpp

```
#pragma once

#include <ngraph/ngraph.hpp>

namespace TemplateExtension {

class Operation : public ngraph::op::Op {
public:
    NGRAPH_RTTI_DECLARATION;

    Operation() = default;
    Operation(const ngraph::Output<ngraph::Node>& arg, int64_t add);
    void validate_and_infer_types() override;
    std::shared_ptr<ngraph::Node> clone_with_new_inputs(const ngraph::OutputVector& new_args) const override;
    bool visit_attributes(ngraph::AttributeVisitor& visitor) override;
    int64_t getAddAttr() const { return add; }
    bool evaluate(const ngraph::HostTensorVector& outputs,
                  const ngraph::HostTensorVector& inputs) const override;

private:
    int64_t add;
};

}
```

op.cpp

```
#include "op.hpp"

NGRAPH_RTTI_DEFINITION(TemplateExtension::Operation, "Template", 0);

Operation::Operation(const ngraph::Output<ngraph::Node> &arg, int64_t add) : Op({arg}), add(add) {
    constructor_validate_and_infer_types();
}

void Operation::validate_and_infer_types() { ... }

std::shared_ptr<ngraph::Node> Operation::clone_with_new_inputs(const ngraph::OutputVector &new_args) const { ... }

bool Operation::visit_attributes(ngraph::AttributeVisitor &visitor) {
    visitor.on_attribute("add", add);
    return true;
}

template <class T> void implementation(const T* input, T* output, int64_t add, size_t size) {
    for (size_t i = 0; i < size; i++) {
        output[i] = input[i] + add;
    }
}

template <ngraph::element::Type_t ET> bool evaluate_op(const ngraph::HostTensorPtr& arg0, const ngraph::HostTensorPtr& out, int64_t add)
{
    size_t size = ngraph::shape_size(arg0->get_shape());
    implementation(arg0->get_data_ptr<ET>(), out->get_data_ptr<ET>(), add, size);
    return true;
}

bool Operation::evaluate(const ngraph::HostTensorVector& outputs,
                         const ngraph::HostTensorVector& inputs) const {
    switch (inputs[0]->get_element_type())
    {
    case ngraph::element::Type_t::i8:   return evaluate_op<ngraph::element::Type_t::i8>  (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::i16:  return evaluate_op<ngraph::element::Type_t::i16> (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::i32:  return evaluate_op<ngraph::element::Type_t::i32> (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::i64:  return evaluate_op<ngraph::element::Type_t::i64> (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::u8:   return evaluate_op<ngraph::element::Type_t::u8>  (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::u16:  return evaluate_op<ngraph::element::Type_t::u16> (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::u32:  return evaluate_op<ngraph::element::Type_t::u32> (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::u64:  return evaluate_op<ngraph::element::Type_t::u8>  (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::bf16: return evaluate_op<ngraph::element::Type_t::bf16>(inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::f16:  return evaluate_op<ngraph::element::Type_t::f16> (inputs[0], outputs[0], getAddAttr());
    case ngraph::element::Type_t::f32:  return evaluate_op<ngraph::element::Type_t::f32> (inputs[0], outputs[0], getAddAttr());
    default: break;
    }
    return false;
}
```

Actual kernel code in nGraph Op.

OpenVINO sample extension code

template_extension

Steps to Run Sample Custom Layers (template extension, win10)

The procedure described here is using an example from following web page. Please refer to the web page for details.
https://docs.openvinotoolkit.org/latest/openvino_docs_HOWTO_Custom_Layers_Guide.html

*** Prerequisites: TensorFlow 1.15.0, keras 2.2.4, Microsoft Visual Studio 2019, CMake x64, Python 3.7 x64**
*** Open "Developer Command Prompt for VS2019" (not a standard command prompt)**

*** Set OpenVINO environment variables**

```
call "%PROGRAMFILES(X86)%\intel\openvino_2021\bin\setupvars.bat"      # "intel" portion might be "intelswtools"
```

*** Download a Keras model which includes unsupported ops**

```
git clone https://github.com/rmsouza01/Hybrid-CS-Model-MRI
cd Hybrid-CS-Model-MRI
git checkout 2ede2f96161ce70dcdc922371fe6b6b254aafcc8
```

The zip file can be download from my GitHub repo.
https://github.com/yas-sim/openvino-custom-layer-development-guide

*** Extract the contents of "`Hybrid-CS-Model-MRI_extensions_and_test_code.zip`" to the current directory**
The zip file contains MO-extensions (Complex, ComplexAbs, FFT/IFFT), nGraph Ops(FFT), IE-extensions (IFFT/IFFT), and some script for automation.

*** Convert trained model in HDF5 (Keras) into frozen PB (TF)**

```
python convert_hdf5_to_frozen_pb.py
  -> .\wnet_20.pb will be generated
```

*** Use model optimizer to convert the frozen pb model into an OpenVINO IR model**

```
python "%INTEL_OPENVINO_DIR%\deployment_tools\model_optimizer\mo.py" --input_model ./wnet_20.pb -b 1 --extensions ./mo_extensions
  -> .\wnet_20.bin, .\wnet_20.xml will be generated
```

The model contains unsupported Ops (Complex, ComplexAbs, IFFT2D). You'll get errors if you don't use the MO extensions

```
[ ERROR ]  List of operations that cannot be converted to Inference Engine IR:
[ ERROR ]      Complex (1)
[ ERROR ]          lambda_2/Complex
[ ERROR ]      IFFT2D (1)
[ ERROR ]          lambda_2/IFFT2D
[ ERROR ]      ComplexAbs (1)
[ ERROR ]          lambda_2/Abs
[ ERROR ]  Part of the nodes was not converted to IR. Stopped.
```

*** Build C++ CPU extension library (The code is taken from template_extension from https://github.com/openvinotoolkit/openvino/tree/master/docs/template_extension)**

```
cd ie_extensions
mkdir build
build.bat
cd ..
 -> ie_extensions\build\release\template_extension.dll will be generated
```

*** Copy CPU extension library and run a Python test program**

```
run_test.bat
```

Useful Links

- Model Optimizer
  - Model Optimizer Developer Guide
  - Custom Layers Guide
  - Available Operations Sets

- Inference Engine
  - Custom Operations Guide
  - Inference Engine Developer Guide
  - Inference Engine Extensibility Mechanism
  - Extension Library
  - Custom nGraph Operation
  - Sample Template Extension Code

- Others
  - OpenVINO Source Code (MO and IE), GitHub
  - OpenVINO API Reference (IE, nGraph)
  - nGraph
  - NetworkX - A Python based graph construction / manipulation library. Used in MO

intel®