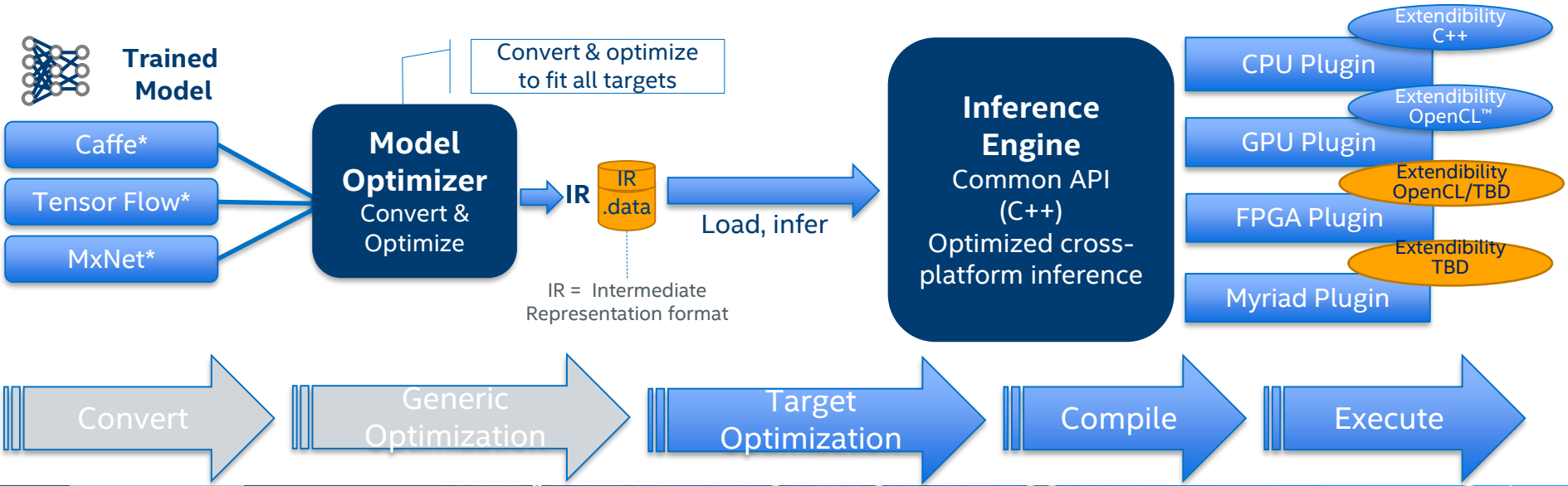
OpenVINO: InferenceEngine

Jan 2019
Kai-Feng Chou/ Maple Chou
Specialist FAE Intel PSG

DLDK

Inference Engine

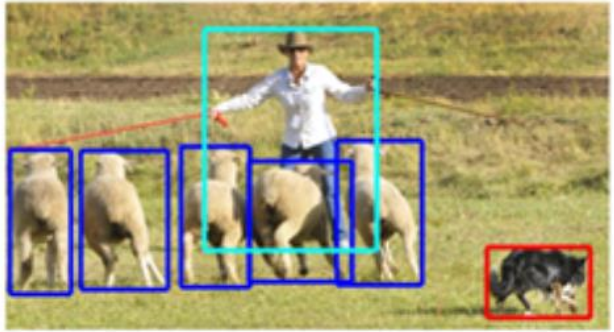
- **What it is**: High-level inference API
- **Why important**: Interface is implemented as dynamically loaded plugins for each hardware type. Delivers best performance for each type without requiring users to implement and maintain multiple code pathways.



Inference on an Intel® Hardware

Many deep learning networks are available—choose the one you need.

(a) classification



(b) detection

(c) segmentation

The complexity of the problem (data set) dictates the network structure. The more complex the problem, the more ‘features’ required, the deeper the network.

Workflow for Inference engine

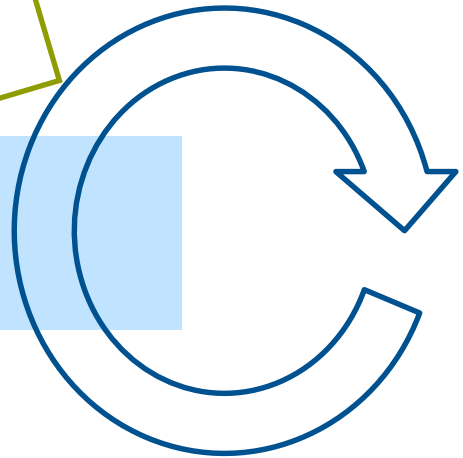
Initialize

Load model and weights
Set batch size (if needed)
Load Inference Plugin (CPU, GPU, FPGA, etc)
Load network to the plugin
Create infer requests

Main loop

- Fill Input
- Infer
- Interpret

Fill request's input buffer with data
Run inference
Interpret output results



Initialize - Load model/ Weights

```
//-----------------------------------------------------------------------------
// Read network information from XML file
//-----------------------------------------------------------------------------

InferenceEngine::CNNNetReader network;
network.ReadNetwork(FLAGS_m);
```

- Every SDK sample has “-m” command-line option

```
//-----------------------------------------------------------------------------
// Read network parameters from BIN file
//-----------------------------------------------------------------------------

std::string binFileName = fileNameNoExt(FLAGS_m) + ".bin";
network.ReadWeights(binFileName.c_str());

//-----------------------------------------------------------------------------
// Set batch size
//-----------------------------------------------------------------------------
```

- Classification sample support batches as “-b” (or via number of input images)

```
network.getNetwork().setBatchSize(FLAGS_batch);
```

Initialize - Load the network to the plugin and get the ExecutableNetwork

```
// ---------------------------  Loading model to the plugin -----------------------------------------

        ExecutableNetwork executable_network = plugin.LoadNetwork(network, {});
        network = {}; //release the network data
        networkReader = {}; //release the network reader
    …
// ---------------------------  Create infer request ------------------------------------------------
        InferRequest infer_request = executable_network.CreateInferRequest();
```

Mainloop - Populate input data/pre-process

```
// --------------------------- Prepare input -------/
for (const auto & item : inputInfo) {
    /** getting input blob (by name) **/
    Blob::Ptr input = infer_request.GetBlob(item.first);
     auto data = input->buffer().as<PrecisionTrait<Precision::U8>::value_type*>();
```

…

Infer

```
infer_request.Infer();
```

Post-Process

```
const Blob::Ptr output_blob = infer_request.GetBlob(firstOutputName);
auto output_data = output_blob->buffer().as<PrecisionTrait<Precision::FP32>::value_type*>();
```

…

Many output formats. Some examples:

- Simple classification: an array of float confidence scores, # of elements=# of classes in the model
- SSD: many “boxes” with a confidence score, label #, xmin,ymin, xmax,ymax

HETERO PLUGIN - FPGA

The rule of auto fallback

Applying device affinities to layers: Automatically, using the fallback *policy*, 1

$ object_detection_sample_ssd **-d HETERO:FPGA,CPU**
-m ssd.xml -i snake.bmp

All IE samples support that

You can load CPU and GPU extensions as usual (“-l” and “-c”)

Regular “-pc” (perf counters) works and gives nice per-subgraph statistics

The “priorities” just defines a greedy behavior

- Keeps all layers that can be executed on the device (FPGA)

Hetero dichotomies

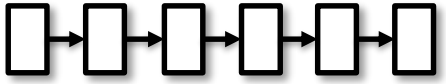
Inference Engine Hetero support is inherently **node-level**

Static user-defined work split (no load-balancing)

In fact, for us the "**Heterogeneity" is almost == "Fallback"**

(enabling and then performance)

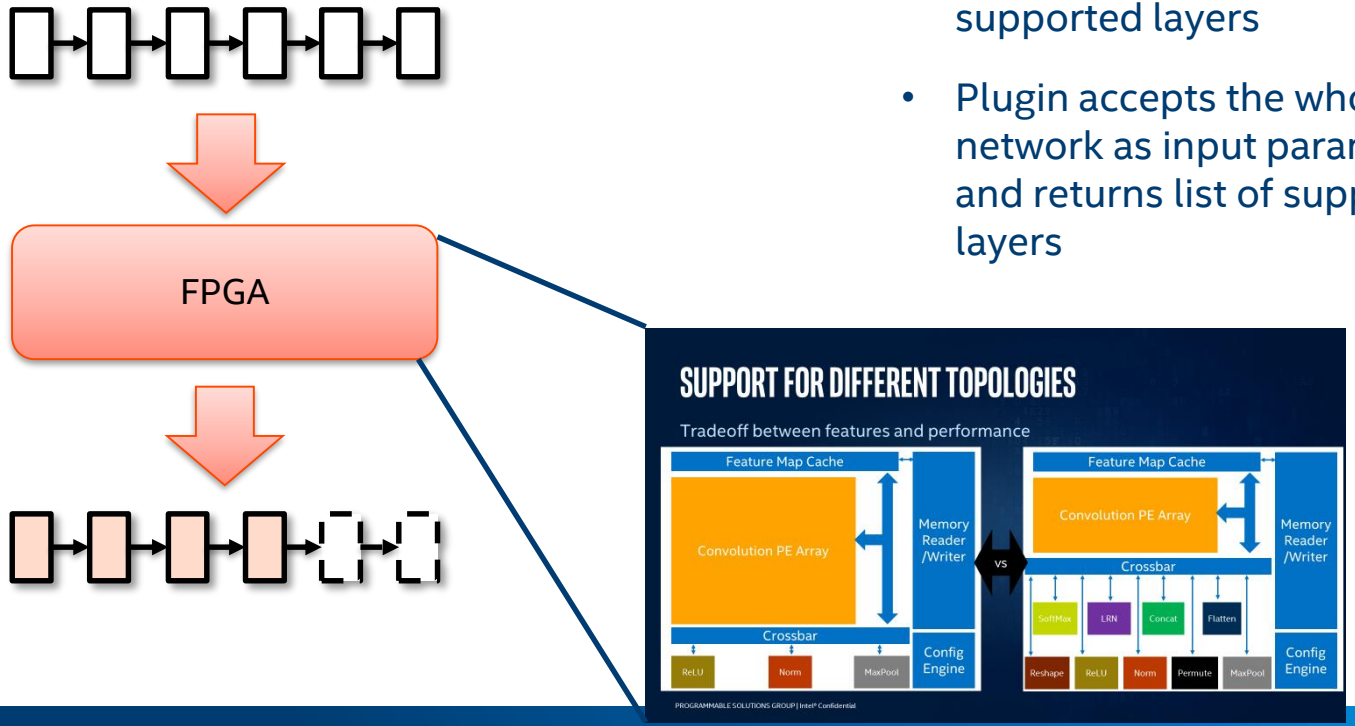
Hetero plugin: fallback policy



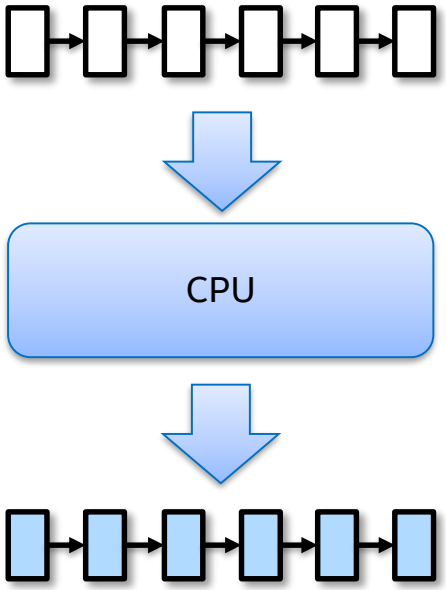
- Specify targets and it's priorities: "HETERO:FPGA,CPU"
- Query each plugin about supported layers
- Plugin accepts the whole network as input parameter and returns list of supported layers

Hetero plugin: fallback policy

- Specify targets and it's priorities: "HETERO:FPGA,CPU"
- Query each plugin about supported layers
- Plugin accepts the whole network as input parameter and returns list of supported layers

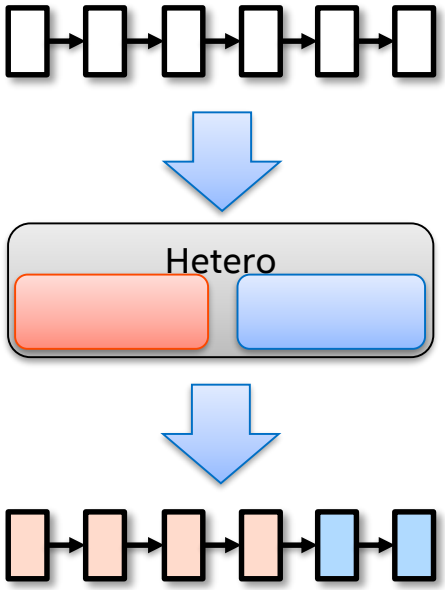


Hetero plugin: fallback policy



- Specify targets and it's priorities: "HETERO:FPGA,CPU"
- Query each plugin about supported layers
- Plugin accepts the whole network as input parameter and returns list of supported layers

Hetero plugin: fallback policy



- Specify targets and it's priorities: "HETERO:FPGA,CPU"
- Query each plugin about supported layers
- Plugin accepts the whole network as input parameter and returns list of supported layers
- Set resulting affinity according to plugins priority

Visualizing the actual Hetero plugin decisions

Code

```
#include "hetero/hetero_plugin_config.hpp“
…
   plugin.SetConfig({
   { HeteroConfigParams::KEY_HETERO_DUMP_GRAPH_DOT, PluginConfigParams::YES }
   });
```

Produces 2 files:

hetero_affinity.dot <- affinities. One color – one device
hetero_subgraphs.dot <- subgraphs. One color – one subgraph

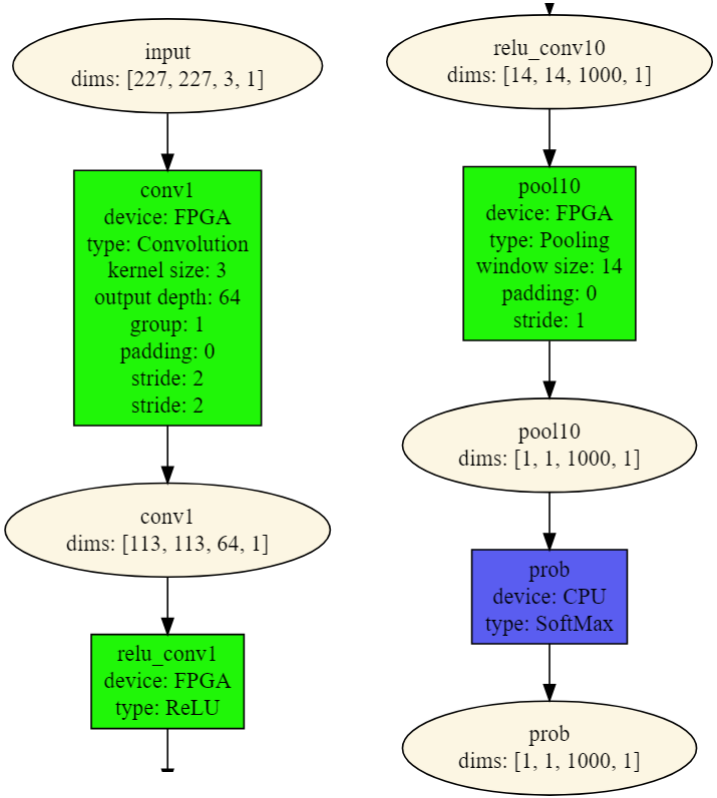
Viewing

```
$sudo apt-get install xdot (Ubuntu)- on Windows install GraphViz
view the file
$xdot split_graph.dot
Or convert to the pdf:
$dot -Tpdf hetero_affinity.dot -O
or copy-paste content of a file to the http://www.webgraphviz.com/
```

See example ouput here <LIVE DEMO>

In the future: feedback from OpenVINO, why smth fell out of the FPGA

Hetero plugin: visualize resulting affinity



Applying device affinities to layers: Automatically, using the fallback *policy* , 2

```
HeteroPluginPtr plugin(make_plugin_name("HeteroPlugin"));
CNNNetReader reader;
reader.ReadNetwork(“Model.xml”);
reader.ReadWeights(“Model.bin”);
CNNNetwork network = reader.getNetwork();

plugin->SetConfig({ { "TARGET_FALLBACK", “FPGA,CPU"} });
plugin->LoadNetwork(exeNetwork, network,{}, &response);
```

Applying device affinities to layers: Explicit, using the API

```
HeteroPluginPtr plugin(make_plugin_name("HeteroPlugin"));
CNNNetReader reader;
reader.ReadNetwork(“Model.xml”);
reader.ReadWeights(“Model.bin”);
CNNNetwork network = reader.getNetwork();
plugin->SetConfig({ { "TARGET_FALLBACK", “FPGA,CPU"} }, &response);

plugin->SetAffinity(network,{}, &response);

auto network = netBuilder.getNetwork();
        auto it = network.begin();
        while (it != network.end()) {
            CNNLayer::Ptr layer = *it++;
            layer->affinity = “FPGA";
            if (layer->name == “conv1") || layer->kernel_size >= 15) {
                 layer->affinity = “CPU";
            }
        }
 status = plugin_ptr->LoadNetwork(ie_net.getNetwork(), &dsc);
```

OPTIMIZATIONS

Optimizations Flow Summary

1. (Offline) Model **conversion**
   - Output: framework- and device-agnostic IR
   - Main tool: Model Optimizer
2. Model **execution**: Inference Engine (IE) performs the inference for the give IR and inputs
   - Output: sustained (stand-alone) model perf (specific device)
   - Main tool: IE Samples
3. **Integration** of the Inference Engine code into real application

- Output: actual KPI
- Main tool: VTune

Inference Engine, few general FPGA tips

- First iteration with FPGA is always significantly slower than the rest, make sure you run multiple iterations (“-ni”)
- Use classification_async to hide data transfers
  - (prev Vtune screenshot)
  - Pronounced for lightweight network (or large inputs/outputs)
- If CPU utilization of concern, minimizes the busy wait time when OMP threads loop in between parallel regions:
  - set KMP_BLOCKTIME=0
  - consider playing OMP_NUM_THREADS
- FPGA performance heavily depends on the bitstream

MKL-specific OMP settings

https://software.intel.com/en-us/articles/recommended-settings-for-calling-intel-mkl-routines-from-multi-threaded-applications

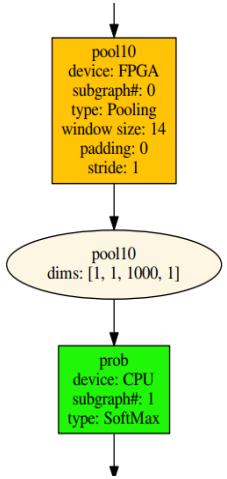
This are better some time that OMP_NUM_THREADS

Inference Engine, hetero with FPGA

- Just like for the MYRIAD, min 2 infer requests in flight are recommended to hide the data transfer costs
  - (notice that 5 is HW-implied limit for the FPGA today)
- All stages are observable in VTune (next foils)
- Perf counters (“-pc”)

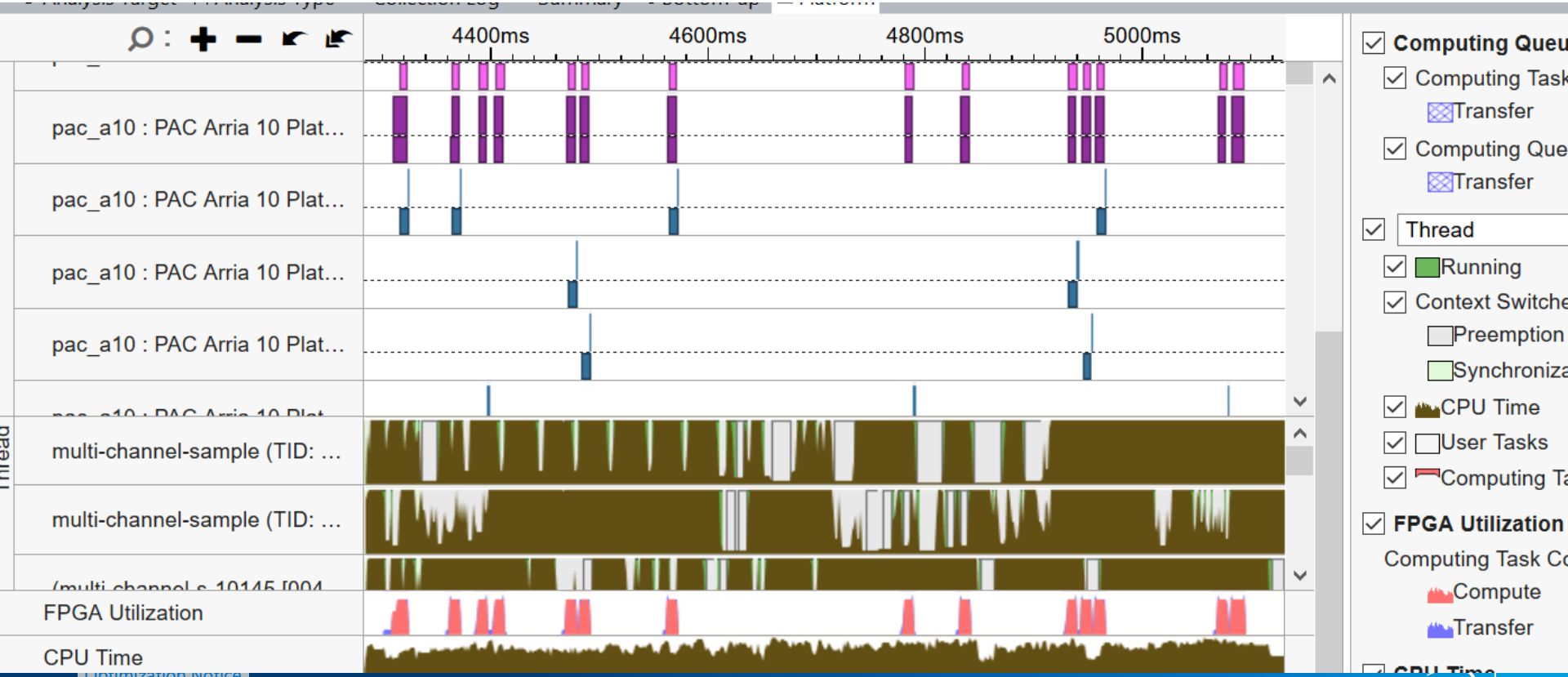
```
subgraph1: 1. input preprocessing (mean data/FPGA):EXECUTED   layerType:          realTime: 129    cpu: 129
subgraph1: 2. input transfer to DDR:EXECUTED                  layerType:          realTime: 201    cpu: 0
subgraph1: 3. FPGA execute time:EXECUTED                      layerType:          realTime: 3808   cpu: 0
subgraph1: 4. output transfer from DDR:EXECUTED               layerType:          realTime: 55     cpu: 0
subgraph1: 5. FPGA output postprocessing:EXECUTED             layerType:          realTime: 7      cpu: 7

subgraph2: EXECUTED                                           layerType: SoftMax  realTime: 10     cpu: 10
Total time: 4212     microseconds
```



- Image “Classification Sample, pipelined”
- More on hetero: link

VTune, FPGA is coming!



VTune, CPU

Grouping: Task Domain / Task Type / Function / Call Stack

Task Domain / Task Type / Function / Call Stack	CPU Time ▼
InferenceEngine	8.257s
Task_runNoThrow	2.765s
MKLDNN_INFER	2.765s
style/decode/conv_t2	0.302s
style/decode/ResizeNearestNeighbor_1_nchw_nChw8c	0.145s
style/decode/conv_t3	0.118s
style/decode/conv_t1	0.117s
style/decode/ResizeNearestNeighbor_1	0.103s
style/decode/add_1	0.087s
style/transfer/FusedBatchNorm_3/MVN_275	0.079s

Use Asynchronous execution!

Inference calls are usually among the heaviest parts of an application

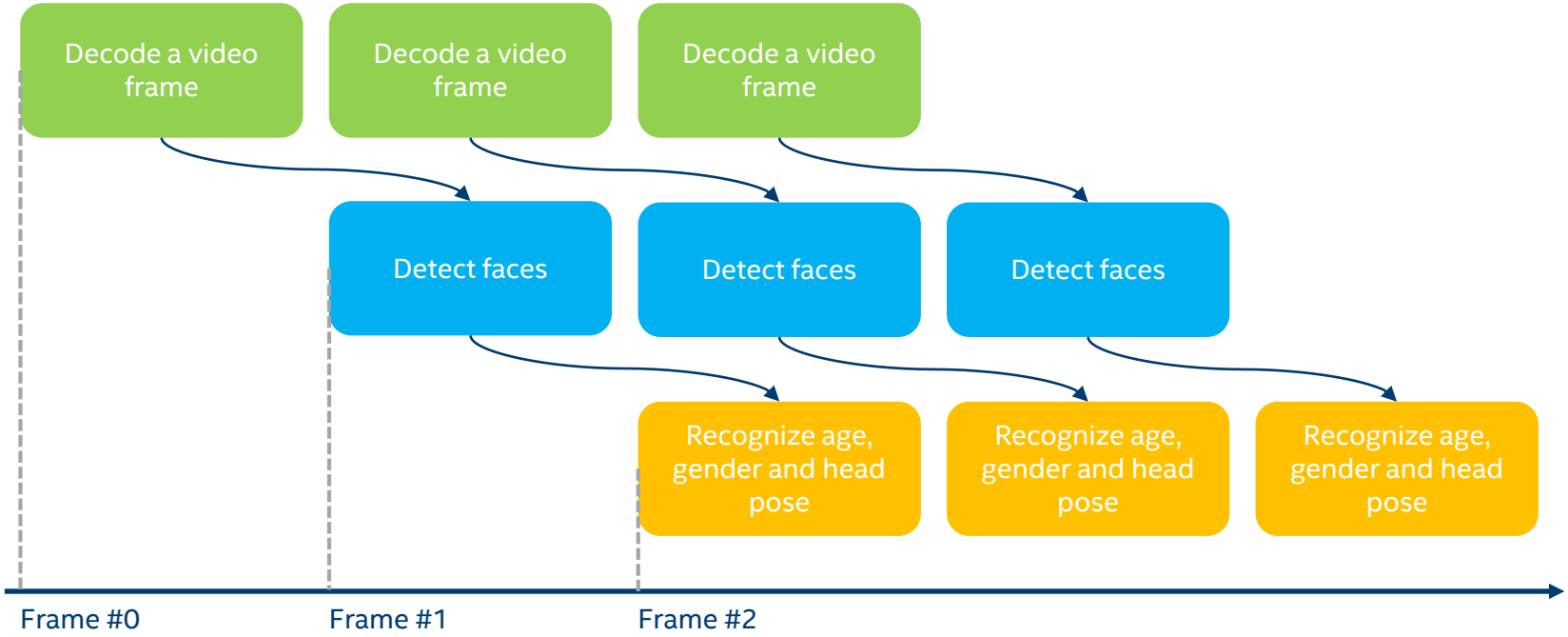
- it is crucial to overlap them with other logic such as data transfer, video encoding-decoding, visualization, inferences of other networks, etc.

With Async API this is possible

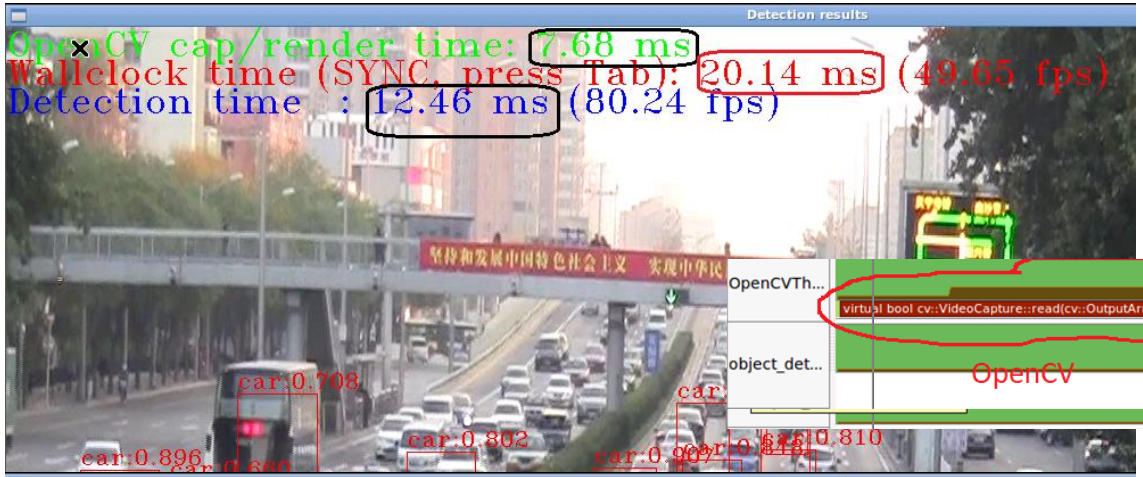
- A request for inference is executed in the background (until you need it's result)
- Your code can continue other tasks in a meantime.

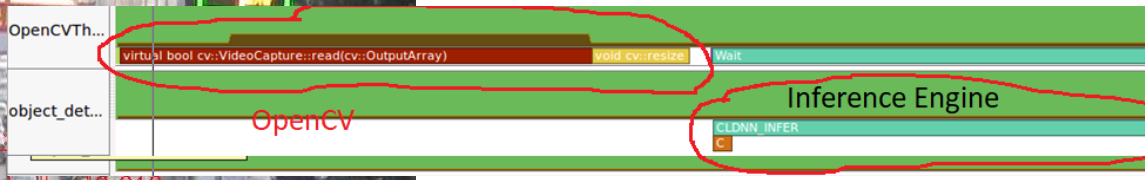
```
// populate inputs etc
auto input = async_infer_request.GetBlob(input_name);
...
// start the async infer request (puts the request to the queue and immediately returns)
async_infer_request.StartAsync();
// here you can continue execution on the host until results of the current request are really
needed
...
async_infer_request.Wait(IInferRequest::WaitMode::RESULT_READY);
auto output = async_infer_request.GetBlob(output_name);
```

Asynchronous execution



Old-style API





Async API

Summary

- Carefully inspect the MO output
- Use CV SDK samples as the perf test-beds (perf counters/batching)
- Understand the implications of the Async/Hetero execution
- Leverage the APIs interop
- User VTune for app-level opt
- etc…etc

READ THE DLDT OPT GUIDE!
https://software.intel.com/en-us/articles/OpenVINO-Inference-Engine-Optimization-Guide

Most important samples

classification_sample (all classification topologies)

classification_sample_**async** (FPGA-friendly perf flavor)

object_detection_samples_ssd (SSD-based topologies)

object_detection_sample (Faster-RCNN)

Yolo (archived)

style_transfer_sample

mask_rcnn_sample

…