Inference Engine

Intel® Deep Learning Deployment Toolkit

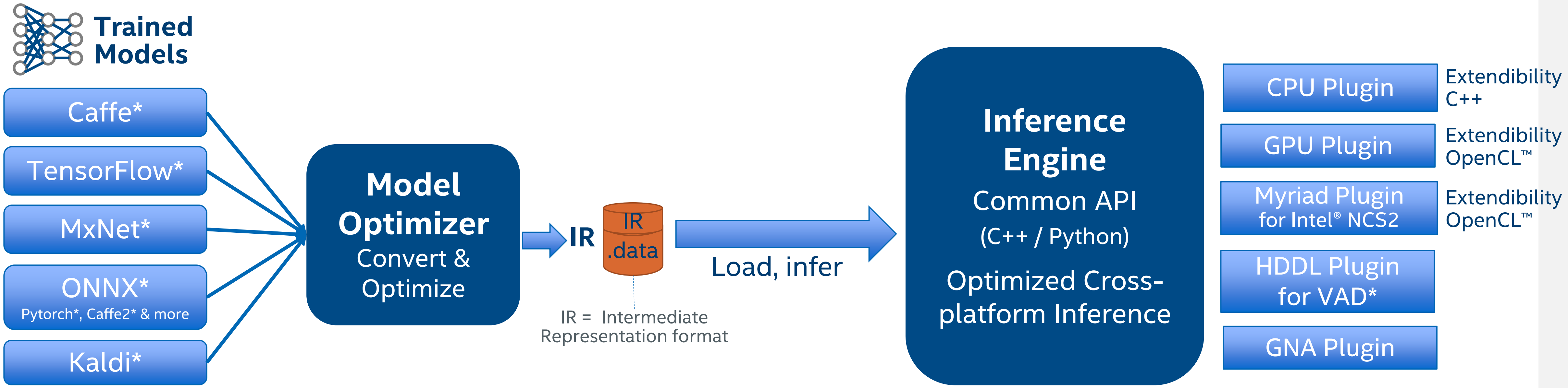
For Deep Learning Inference

Model Optimizer

- A Python* based tool to **import** trained models and **convert** them to Intermediate Representation
- **Optimizes for performance** or space with conservative topology transformations
- **Hardware-agnostic** optimizations

Inference Engine

- High-level, C/C++ and Python, inference **runtime API**
- Interface is implemented as **dynamically loaded plugins** for each hardware type
- Delivers advanced performance for each type **without requiring users to implement and maintain multiple code pathways**



GPU = Intel® CPU with integrated GPU/Intel® Processor Graphics, Intel® NCS = Intel® Neural Compute Stick (VPU)
*VAD = Intel® Vision Accelerator Design Products (HDDL-R)
OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos

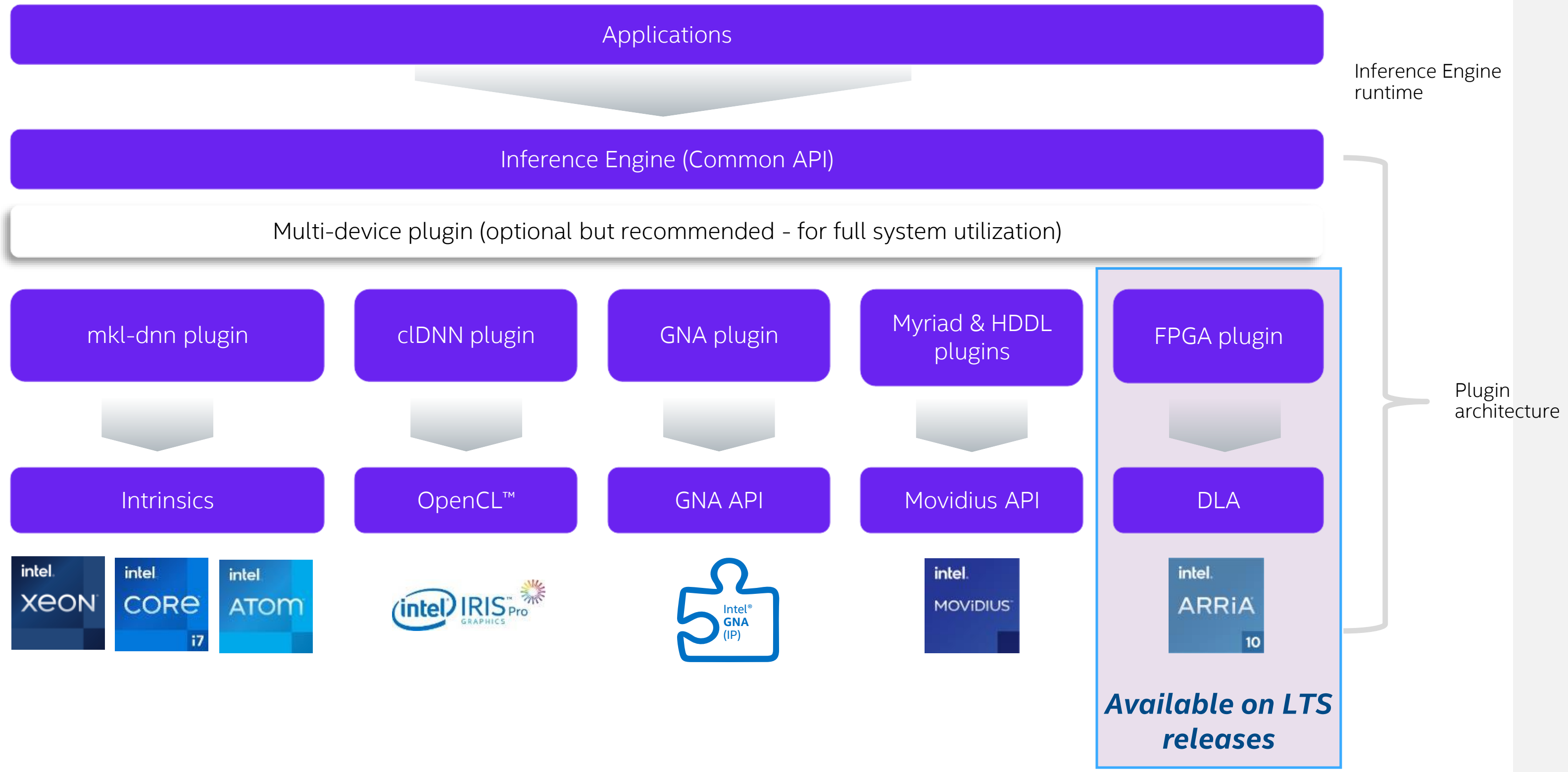
Optimal Model Performance Using the Inference Engine

Core Inference Engine Libraries

- Create Inference Engine Core object to work with devices
- Read the network
- Manipulate network information
- Execute and pass inputs and outputs

Device-specific Plugin Libraries

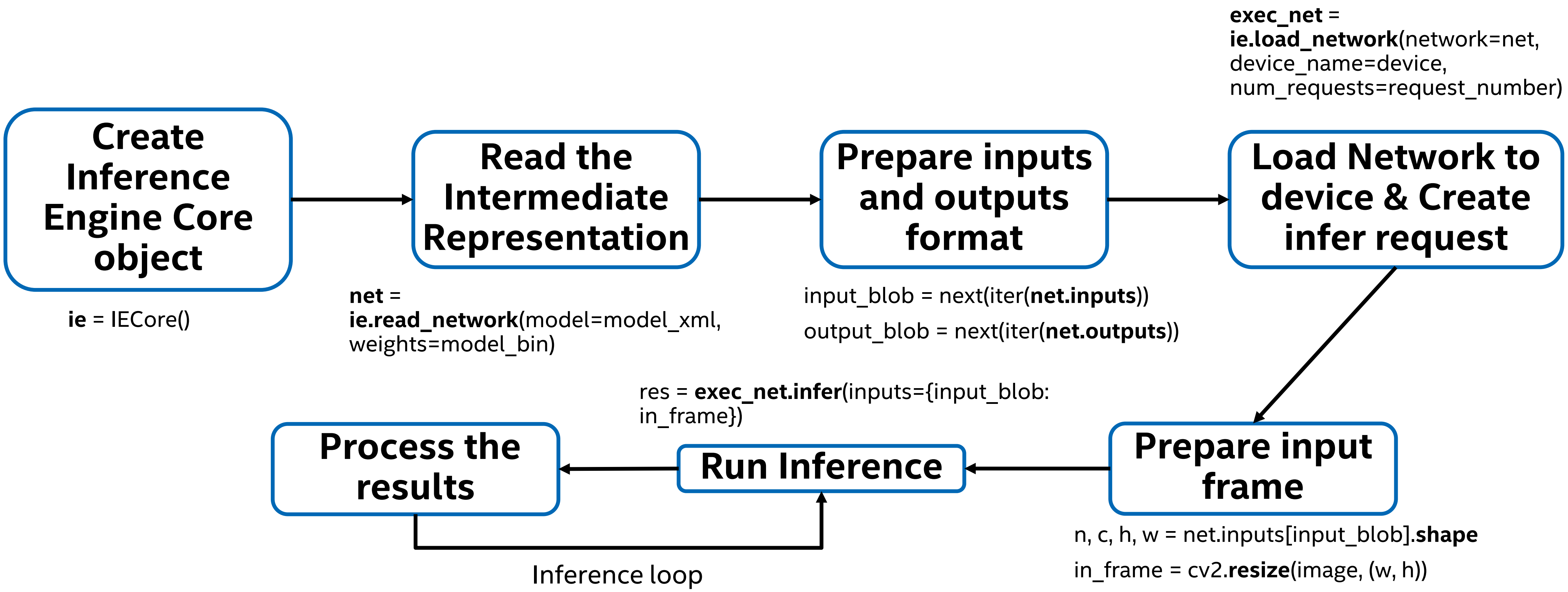
- For each supported target device, Inference Engine provides a plugin — a DLL/shared library that contains complete implementation for inference on this device.



GPU = Intel CPU with integrated graphics/Intel® Processor Graphics/GEN

GNA = Gaussian mixture model and Neural Network Accelerator

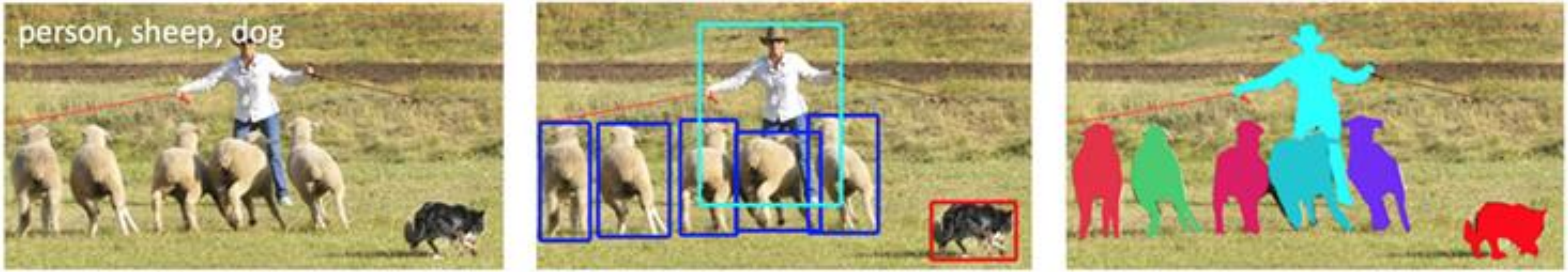
Common Workflow for Using the Inference Engine API



http://docs.openvinotoolkit.org/latest/_docs_IE_DG_Integrate_with_customer_application_new_API.html

Inference on an Intel® Edge System

- 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.

Process the results

Object Detection SSD example

- Process the results (Post-processing)

The array of detection summary info, name - `detection_out`, shape - `1, 1, N, 7`, where N is the number of detected bounding boxes. For each detection, the description has the format: [`image_id`, `label`, `conf`, `x_min`, `y_min`, `x_max`, `y_max`], where:

- `image_id` - ID of the image in the batch
- `label` - predicted class ID
- `conf` - confidence for the predicted class
- (`x_min`, `y_min`) - coordinates of the top left bounding box corner (coordinates are in normalized format, in range [0, 1])
- (`x_max`, `y_max`) - coordinates of the bottom right bounding box corner (coordinates are in normalized format, in range [0, 1])

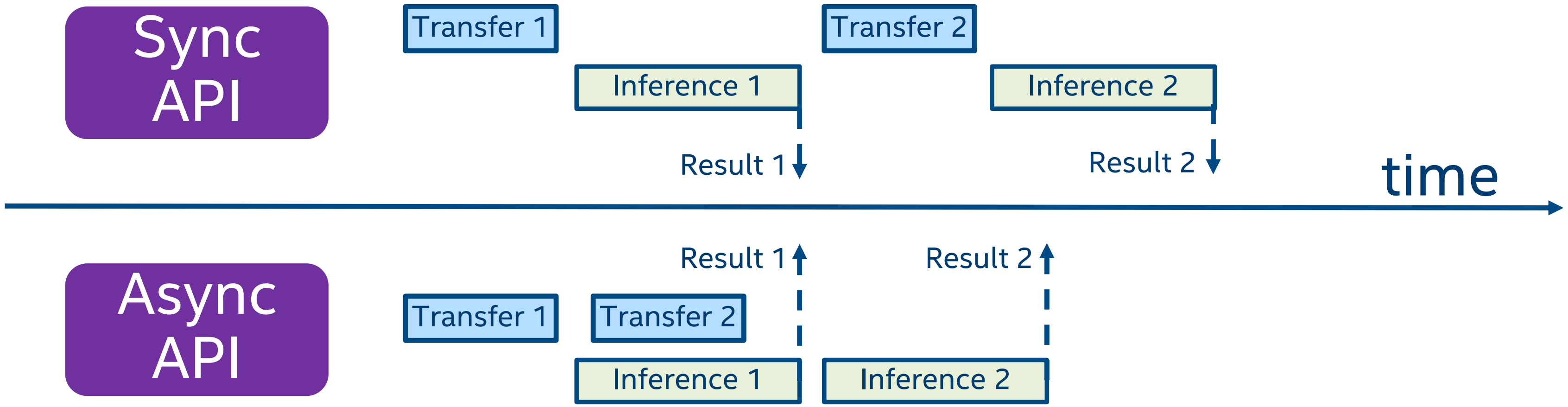
```
res = res[out_blob]
boxes, classes = {}, {}
data = res[0][0]
for number, proposal in enumerate(data):
    if proposal[2] > 0:
        imid = np.int(proposal[0])
        ih, iw = images_hw[imid]
        label = np.int(proposal[1])
        confidence = proposal[2]
        xmin = np.int(iw * proposal[3])
        ymin = np.int(ih * proposal[4])
        xmax = np.int(iw * proposal[5])
        ymax = np.int(ih * proposal[6])
        print("[{},{}] element, prob = {:.6}    ({},{})-({},{}) batch
        id : {}".format(number, label, confidence, xmin, ymin, xmax,
        ymax, imid), end="")
        if proposal[2] > 0.5:
            print(" WILL BE PRINTED!")
            if not imid in boxes.keys():
                boxes[imid] = []
            boxes[imid].append([xmin, ymin, xmax, ymax])
            if not imid in classes.keys():
                classes[imid] = []
            classes[imid].append(label)
    else:
        print()

for imid in classes:
    tmp_image = cv2.imread(args.input[imid])
    for box in boxes[imid]:
        cv2.rectangle(tmp_image, (box[0], box[1]), (box[2], box[3]), (
        232, 35, 244), 2)
    cv2.imwrite("out.bmp", tmp_image)
    log.info("Image out.bmp created!")
```

Inference Engine

Synchronous vs Asynchronous Execution

- In IE API model can be executed by **Infer Request** which can be:
- **Synchronous** - blocks until inference is completed.
  - exec_net.infer(inputs = {input_blob: in_frame})
- **Asynchronous** – checks the execution status with the wait or specify a completion callback *(recommended way)*.
  - exec_net.start_async(request_id = id, inputs={input_blob: in_frame})
  - If exec_net.requests[id].wait() != 0

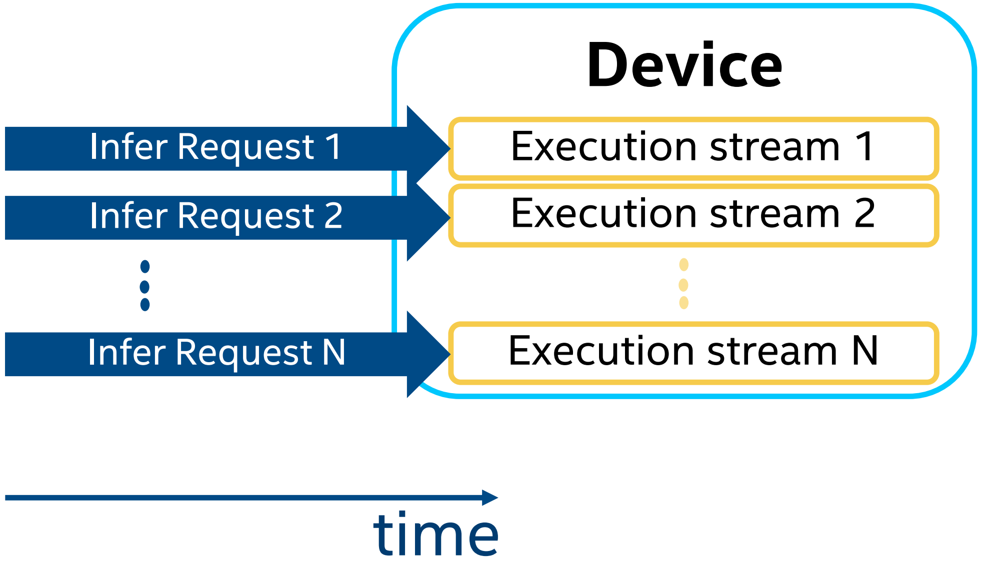
    do something



Inference Engine

Throughput Mode for CPU, iGPU and VPU

- **Latency** – inference time of 1 frame (ms).
- **Throughput** – overall amount of frames inferred per 1 second (FPS)
- **“Throughput"** mode allows the Inference Engine to efficiently run multiple infer requests simultaneously, greatly improving the overall throughput.
- Device resources are divided into execution “**streams**” – parts which runs infer requests in parallel



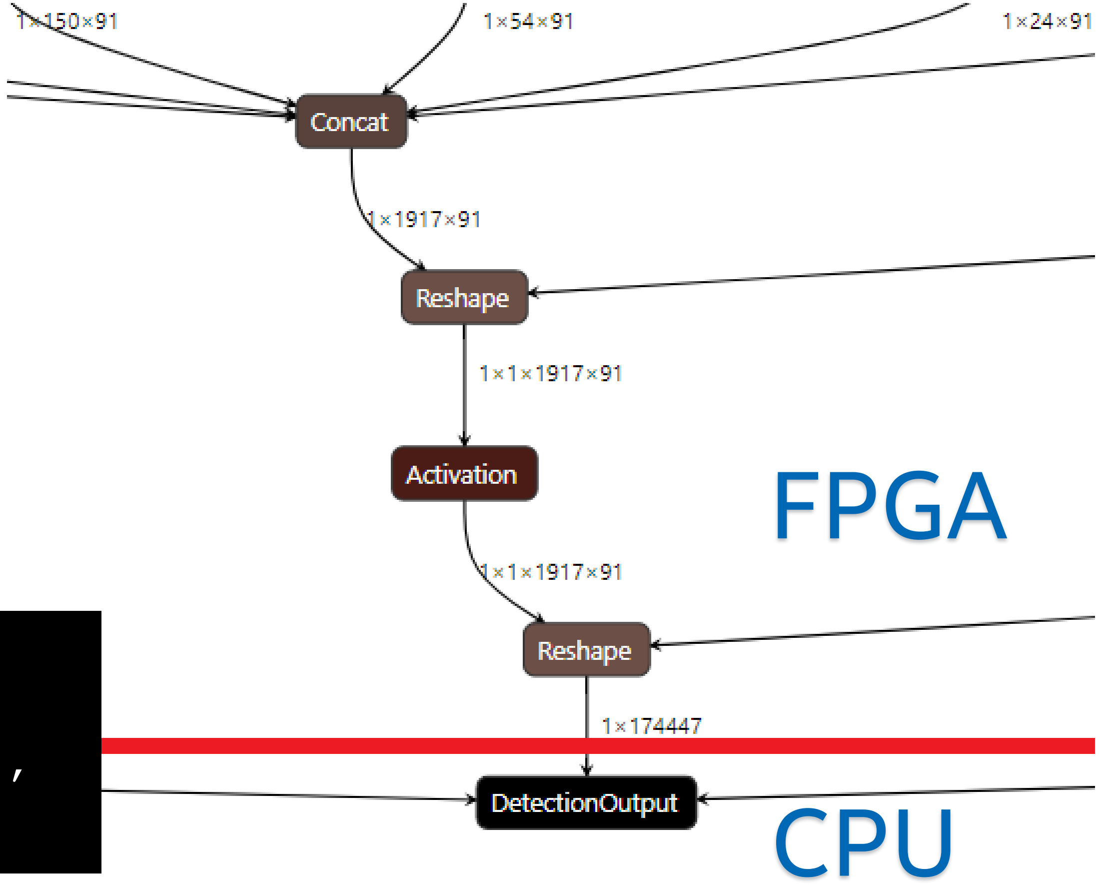
CPU Example:

```
ie = IECore()
ie.GetConfig(CPU, KEY_CPU_THROUGHPUT_STREAMS)
```

Inference Engine

Heterogeneous Support

- You can execute different layers on different HW units
- Offload unsupported layers on fallback devices:
  - Default affinity policy
  - Setting affinity manually (`CNNLayer::affinity`)
- All device combinations are supported (CPU, GPU, FPGA, MYRIAD, HDDL)
- Samples/demos usage “`-d HETERO:FPGA,CPU`”




```
InferenceEngine::Core core;
          auto executable_network =
          core.LoadNetwork(reader.getNetwork(),
          "HETERO:FPGA,CPU");
```

Inference Engine

Multi-device Support

Automatic load-balancing between devices (inference requests level) for full system utilization

- Any combinations of the following devices are supported (CPU, iGPU, VPU, HDDL)
- As easy as “-d MULTI:CPU,GPU” for cmd-line option of your favorite sample/demo
- C++ example (Python is similar)

```
Core ie;
ExecutableNetwork exec =
ie.LoadNetwork(network,{{“DEVICE_PRIORITIES”, “CPU,GPU”}},
“MULTI”)
```