C++ Python

Heterogeneous Plugin

Introducing the Heterogeneous Plugin (Python)

The heterogeneous plugin enables computing the inference of one network on several devices. The purposes of executing networks in heterogeneous mode are to:

- Utilize the power of accelerators to process the heaviest parts of the network and to execute unsupported layers on fallback devices like the CPU
- Utilize all available hardware more efficiently during one inference

The execution through heterogeneous plugin can be divided into two independent steps:

1. Setting of hardware affinity to layers
2. Loading a network to the Heterogeneous plugin, splitting the network to parts, and executing them through the plugin

These steps are decoupled. The setting of affinity can be done automatically using the fallback policy or in manual mode.

The fallback automatic policy causes “greedy” behavior and assigns all layers that can be executed on certain device according to the priorities you specify (for example, HETERO:GPU,CPU). Automatic policy does not take into account plugin peculiarities such as the inability to infer some layers without other special layers placed before or after that layer. The plugin is responsible for solving such cases. If the device plugin does not support the subgraph topology constructed by the HETERO plugin, then you should set affinity manually.

Some of the topologies are not well-supported for heterogeneous execution on some devices or cannot be executed in this mode at all. Examples of such networks are those having activation layers which are not supported on the primary device. If transmitting data from one part of a network to another part in heterogeneous mode takes more time than in normal mode, it may not make sense to execute them in heterogeneous mode. In this case, you can define the most compute intense part manually and set the affinity to avoid sending data back and forth many times during one inference.

Use Default Layer Affinities

To use the default affinities, call `load_network` with the “HETERO” device, with an optional list of devices to consider.

```
from openvino.inference_engine import IECore

ie = IECore()
net = ie.read_network(model=path_to_model)
exec_net = ie.load_network(network=net, device_name='HETERO:GPU,CPU')
```

Annotation of Layers per Device and Default Fallback Policy

Default fallback policy decides which layer goes to which device automatically according to the support in dedicated plugins (GPU, CPU, MYRIAD).

Another way to annotate a network is to set affinity manually using code.

Set Affinity of All Layers to CPU

```
import ngraph as ng
from openvino.inference_engine import IECore

ie = IECore()
# Read a network in IR or ONNX format
net = ie.read_network(path_to_model)
# Create an Ngraph (graph) function from the network
ng_func = ng.function_from_cnn(net)
for node in ng_func.get_ordered_ops():
    rt_info = node.get_rt_info()
    rt_info["affinity"] = "CPU"
```

The fallback policy does not work if even one layer has an initialized affinity. The sequence should be calling the default affinity settings and then setting the layers manually.

Note

If you set affinity manually, be aware that currently Inference Engine plugins do not support constant (*Constant* -> *Result*) and empty (*Parameter* -> *Result*) networks. Please avoid these subgraphs when you set affinity manually.

Example - Manually Setting Layer Affinities

```
import ngraph as ng
from openvino.inference_engine import IECore

ie = IECore()
# Read a network in IR or ONNX format
net = ie.read_network(path_to_model)
ng_func = ng.function_from_cnn(net)

for node in ng_func.get_ordered_ops():
    rt_info = node.get_rt_info()
    rt_info["affinity"] = "CPU"

# Load the network on the target device
exec_net = ie.load_network(network=net, device_name='HETERO:FPGA,CPU')
```

Note

`ie.query_network` does not depend on affinities set by a user, but queries for layer support based on device capabilities.

Details of Splitting Network and Execution

During the loading of the network to the heterogeneous plugin, the network is divided into separate parts and loaded to dedicated plugins. Intermediate blobs between these sub graphs are allocated automatically in the most efficient way.

Execution Precision

The precision for inference in the heterogeneous plugin is defined by:

- Precision of IR
- Ability of final plugins to execute in precision defined in IR

Example:

- If you want to execute GPU with CPU fallback with FP16 on GPU, you need to use only FP16 IR.

OpenVINO samples can be used with the following command:

```
./object_detection_sample_ssd -m  <path_to_model>/ModelSSD.xml -i
<path_to_pictures>/picture.jpg -d HETERO:MYRIAD,CPU
```

where HETERO stands for the heterogeneous plugin

You can point to more than two devices, for example: `-d HETERO:MYRIAD,GPU,CPU`

Analyzing Heterogeneous Execution

After enabling the KEY_HETERO_DUMP_GRAPH_DOT config key, you can dump GraphViz* .dot files with annotations of devices per layer.

The heterogeneous plugin can generate two files:

- `hetero_affinity_<network name>.dot` - annotation of affinities per layer. This file is written to the disk only if the default fallback policy was executed
- `hetero_subgraphs_<network name>.dot` - annotation of affinities per graph. This file is written to the disk during execution of `ICNNNetwork::LoadNetwork()` for the heterogeneous plugin

To Generate the .dot Files

```
ie = IECore()
ie.set_config( config={'HETERO_DUMP_GRAPH_DOT' : 'YES'}, device_name='HETERO')
```

You can use the GraphViz* utility or a file converter to view the images. On the Ubuntu* operating system, you can use xdot:

- `sudo apt-get install xdot`
- `xdot hetero_subgraphs.dot`

You can use performance data (in sample applications, it is the option `-pc`) to get the performance data on each subgraph.

Here is an example of the output for Googlenet v1 running on HDDL with fallback to CPU:

```
subgraph1: 1. input preprocessing (mean data/HDDL):EXECUTED layerType:
realTime: 129   cpu: 129  execType:
subgraph1: 2. input transfer to DDR:EXECUTED                layerType:
realTime: 201   cpu: 0    execType:
subgraph1: 3. HDDL execute time:EXECUTED                    layerType:
realTime: 3808  cpu: 0    execType:
subgraph1: 4. output transfer from DDR:EXECUTED             layerType:
realTime: 55    cpu: 0    execType:
subgraph1: 5. HDDL output postprocessing:EXECUTED           layerType:
realTime: 7     cpu: 7    execType:
subgraph1: 6. copy to IE blob:EXECUTED                      layerType:
realTime: 2     cpu: 2    execType:
subgraph2: out_prob:          NOT_RUN                       layerType: Output
realTime: 0     cpu: 0    execType: unknown
subgraph2: prob:              EXECUTED                      layerType: SoftMax
realTime: 10    cpu: 10   execType: ref
Total time: 4212 microseconds
```

See Also

Supported Devices