CoreVectorBlox SDK Programmer's Guide v 2.0.2

Introduction

This programmer's guide provides details about Microchip CoreVectorBlox Tool Flow environment and it's use for converting neural networks and generating graphs, to accelerate inference on an FPGA using CoreVectorBlox IP.

The intended audience are software developers using Microchip CoreVectorBlox IP to accelerate neural network applications.

Table of Contents

Introduction.....1

1. Overview.....3
    1.1. Supported Operating Systems.....3
    1.2. Flow Diagram—SDK.....3

2. SDK.....4
    2.1. Setting up the Installation Environment.....4
    2.2. SDK Toolkits.....4
    2.3. VBX Graph Generation, vnnx_compile.....7
    2.4. TFLite Layers.....7
    2.5. VBX Inference Simulation.....7
    2.6. API for Interacting with CoreVectorBlox.....8
    2.7. C Postprocessing.....16

3. Tutorials.....17

4. Application Notes.....18

5. References.....19

6. Revision History.....20

Microchip Information.....23
    Trademarks.....23
    Legal Notice.....23
    Microchip Devices Code Protection Feature.....23

1. Overview

VectorBlox™ programmer’s guide covers both the installation and use of the VectorBlox Accelerator SDK. It includes the steps needed to convert a high-level model description to graph that runs on the CoreVectorBlox IP or on the bit-accurate simulator. This guide also covers the use of the underlying C code and API, needed to run the CoreVectorBlox IP on hardware.

1.1. Supported Operating Systems

Ubuntu 20.04, 22.04 and 24.04 are the current Linux® distributions that are supported.

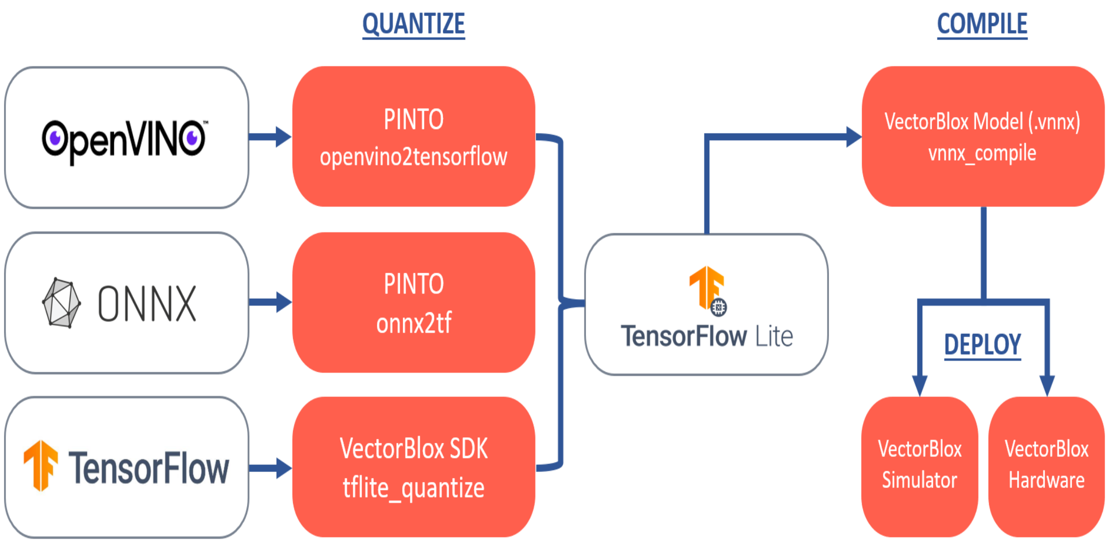
Users who are required to use Microsoft Windows® are recommended to try out Windows Subsystem for Linux: https://docs.microsoft.com/en-us/windows/wsl/install-win10. The user must be aware of few things while using Windows Subsystem for Linux:

- DNS maybe broken when connected to a VPN. The VPN should be down while installing or running the VectorBlox Accelerator SDK.
- The SDK must be installed in the Linux filesystem, and not in the Windows filesystem available in `/mnt/c`

1.2. Flow Diagram—SDK

The following flow diagram shows the steps of a Convolutional Neural Network (CNN) model that goes through the SDK to convert to a CoreVectorBlox graph. There are four paths in this flow depending on the model source, indicated by the white boxes in the following figure. If the model was downloaded with OpenVINO, it will go through OpenVINO's model optimizer and PINTO's openvino2tensorflow tool to generate a TF Lite. If the model is an ONNX or PyTorch format, it can be converted to TF Lite using PINTO's onnx2tf tool. If the model is in TensorFlow format, it can be converted to TF Lite directly. If the model is already in quantized INT8 TF Lite format, it will enter the flow directly at the `vnnx_compile` step.

Figure 1-1. VectorBlox SDK Flow Diagram



2. SDK

The SDK is provided as a single zip file, containing everything needed to produce and test neural networks. The neural networks produced are ready to deploy on any PolarFire® FPGA with a CoreVectorBlox IP instantiated. The layout of the archive is as follows:

```
├── docs
├── app_notes
├── drivers
│   └── vectorblox
├── example
│   ├── sim-c
│   ├── soc-c
│   ├── soc-video-c
│   ├── postprocess
│   ├── python
├── fw
├── install_dependencies.sh
├── lib
│   └── libvbx_cnn_sim.so
├── python
├── README.md
├── requirements.txt
├── setup_vars.sh
└── tutorials
```

2.1. Setting up the Installation Environment

Before using the SDK, dependencies must be installed, the python virtual environment (venv) must be set and several environment variables need to be initialized.

To set the python virtual environment (venv) and initialize several environment variables, perform the following steps:

1. Ensure that the user is working in an Ubuntu 20.04, 22.04 and 24.04 environment with at least Python 3.10.
2. To download the SDK, navigate to the GitHub page and click Tags to view the releases. Download the zip folder for the latest tag release and unzip in a Linux directory.
3. To install the dependencies, navigate to the root folder of the SDK and run the following command:

   ```
   bash install_dependencies.sh
   ```

   Note: sudo access is required to install packages.
4. Create and activate the python3 virtual environment and set the environment variables with the following command:

   ```
   source setup_vars.sh
   ```

   Note: When the user is in the VBX python environment, the shell prompt is prefixed with `(vbx_env)`.
5. To exit or deactivate the environment, run the following command:

   ```
   deactivate
   ```

   Note: The `README.md` file describes these steps and examples of activating and deactivating the virtual environment.

2.2. SDK Toolkits

The following are the four tools used in the VectorBlox flow:

- OpenVINO's model zoo and optimizer

- PINTO's openvino2tensorflow
- PINTO's onnx2tf
- VectorBlox SDK tflite_quantize, tflite_preprocess, tflite_postprocess and tflite_cut

Depending on the source of user's network, the user may either use few of these tools, go directly to TF Lite conversion, or go directly to VNNX graph generation.

For converting to TF Lite, users need their own NumPy array files for TF Lite calibration. Tutorials that are provided in the SDK will download the necessary NumPy calibration data. Users can leverage VectorBlox SDK generate_npy tool to format their data into a NumPy array file.

Each of these VectorBlox flow tools are briefly described in the following sections along with links to their documentation. The tools are available from the command-line within the VBX Python environment.

2.2.1. OpenVINO Model Zoo and Optimizer

The OpenVINO model zoo provides access to many popular neural networks, along with key information about preprocessing and performance. It also provides a tool to download models. The following is a short description of the command usage. The full documentation is available in OpenVINO Toolkit.

- Print all models available in the model zoo.

```
~/sdk (vbx_env)
$ omz_downloader --print_all
```

- Download a model.

```
~/sdk (vbx_env)
$ omz_downloader --name MODEL
```

The OpenVINO `mo` command is a tool to convert and optimize models for inference. Models from various source frameworks are converted to the OpenVINO Intermediate Representation (IR). The networks are further optimized for inference by removing training-time layers like dropout and applying layer fusion. The following are the examples of the command's usage. The full documentation is available in OpenVINO Toolkit.

- Convert and optimize a Caffe model without scale and mean values.

```
~/sdk (vbx_env)
$ mo –framework caffe --input_model MODEL.caffe
```

- Convert and optimize a Caffe model with scale and mean values.

```
~/sdk (vbx_env)
$ mo –framework caffe --input_model MODEL.caffe \
--scale_values [127., 127., 127] --mean_values [64., 64., 64.]
```

Note: The documentation of the model must be reviewed to determine if there is pre-processing or if scale and mean values need to be included with this convert command.

2.2.2. PINTO openvino2tensorflow

PINTO's openvino2tensorflow tool converts models in the OpenVINO IR into a TF Lite model. We can use it to keep inputs as batch Number, Channels, Height, Width (NCHW) dimension order or make direct changes to parameters and layers alongside the TF Lite conversion. The following is an example of the tool's usage. More information can be found on the tool's Github page.

```
~/sdk (vbx_env)
$ openvino2tensorflow --model-path MODEL.xml --output_full_integer_quant_tflite \
--load_dest_file_path_for_the_calib_npy CALIB.npy
```

2.2.3. PINTO onnx2tf

PINTO's onnx2tf tool converts models in ONNX format into a TF Lite model. Similar to PINTO's openvino2tensorflow tool, it supports different arguments and allows users flexibility in the TF Lite conversion. We can use it to maintain static input shapes or cut the ONNX graphs at designated nodes when converting to TF Lite. The following is an example of the tool's usage. For more information, see the tool's Github page.

```
~/sdk (vbx_env)
$ onnx2tf --input_onnx_file_path MODEL.onnx --output_integer_quantized_tflite \
--output_signaturedefs \
--custom_input_op_name_np_data_path INPUT_OP NUMPY.npy [MEAN] [STD]
```

2.2.4. VectorBlox SDK generate_npy

`generate_npy` generates a numpy calibration file using sample images, for tflite calibration. The image directory needs to be specified. If using dimensions other than 244 x 244 in height and width, shape needs to be specified. The following is an example of the tool's usage.

```
~/sdk (vbx env)
$ generate_npy  sample_images/ --output_name OUTPUT_NAME --shape HEIGHT WIDTH --count 20
```

2.2.5. VectorBlox SDK tflite_quantize

If the model is based on TensorFlow, then users can utilize the SDK provided tflite_quantize tool to convert it to TF Lite directly. The tool can also allow for normalization based off of mean and scale parameters, which are applied to the calibration data. The following is an example of the tool's usage.

```
~/sdk (vbx_env)
$ tflite_quantize saved_model/ MODEL.tflite --data NUMPY.npy --mean 127.5 --scale 127.5
```

After generating a TF Lite model, the tflite_cut tool can be used to re-assign starting and ending nodes for the TF Lite graph. The following is an example of the tool's usage.

```
~/sdk (vbx_env)
$ tflite_cut MODEL.tflite -i 3 -o 5
```

2.2.6. VectorBlox SDK tflite_preprocess

`tflite_preprocess` adds a preprocessing layer on top of a TF Lite model. This is required for running networks on hardware and in C, as the inputs obtained are uint8 rather than int8, and values would also need to be scaled.

```
~/sdk (vbx env)
$ tflite_preprocess MODEL.tflite -s 255. -m 127
```

2.2.7. VectorBlox SDK tflite_postprocess

`tflite_postprocess` adds an injection layer for our demo, resizes the outputs and turns category outputs to pixel values.

```
~/sdk (vbx_env)
$ tflite_postprocess MODEL.tflite --dataset VOC --opacity 0.8 --height 1080 width 1920
```

2.2.8. VectorBlox SDK tflite_cut

If a model needs to be cut at specific nodes, the `tflite_cut` function can be used to cut the graph at those specific nodes. The output will then be printed out with a `*.0.tflite, *.1.tflite, *.2.tflite` and so on. The following is an example of the tool's usage.

```
~/sdk (vbx_env)
$ tflite_cut MODEL.tflite -c
```

2.3. VBX Graph Generation, vnnx_compile

The TF Lite model is created in the previous step. This model runs through the VectorBlox graph generation tool which produces a quantized VNNX file that runs both on hardware and with the simulator. The user must provide the input TF Lite model, the output file name, and the hardware configuration (V250, V500 and V1000). There will be an updated CoreVectorBlox IP Handbook with more information on V250, V500 and V1000 in a future release. The default configuration in tutorials is V1000.

Additionally, a test image may be included in the graph, allowing inference and post-processing to be easily tested. Users can also specify starting and ending nodes for the VNNX graph if they have pre-processing or post-processing code that replaces those operations.

Display usage:

```
usage: vnnx_compile [-h] -t TFLITE -c {V250,V500,V1000} -o OUTPUT [-i [INPUTS ...]] [-s
START_LAYER] [-e END_LAYER]
options:
-h, --help                              show this help message and exit
-t TFLITE, --tflite TFLITE              tflite I8 model description (.tflite)
-c {V250,V500,V1000}, --size-conf
{V250,V500,V1000}
                                        size configuration to build model for
-o OUTPUT, --output OUTPUT              name of vnnx output file
-i [INPUT ...], --input [INPUT ...]     provide test input(image) for model
-s START_LAYER, --start_layer START_LAYER
-e END_LAYER, --end_layer END_LAYER
```

The following is an example usage for generating a VNNX graph from a TF Lite model for the V1000 CoreVectorBlox configuration:

```
~/sdk (vbx_env)
$ vnnx_compile -t MODEL.tflite -c V1000 -o MODEL.vnnx
```

The `vnnx_compile` command is available from the command line within the VBX python environment.

2.4. TFLite Layers

VectorBlox aims to support all TFLITE INT8 layers described in the TFLite INT8 quantization spec.

For the list of currently supported operators and their restrictions, see Github repository.

2.5. VBX Inference Simulation

The VBX graph (VNNX) is generated and ready to run on the functionally accurate VBX simulator. Example scripts which call the simulator and run post-processing are provided for classification and object detection tasks. For more details on simulator use and post-processing that these scripts leverage, and are found in the subsequent sections of this guide.

2.5.1. C Simulation

The following script runs for the C version of the simulator, and prints out the checksum of the network based on either a sample image (.JPG) or the default test_data if a sample image is not passed as an argument.

```
~/sdk (vbx_env)
$VBX_SDK/example/sim-c/ ./sim-run-model MODEL.vnnx IMAGE.jpg POSTPROCESSTYPE
```

2.5.2. Python Simulation

The following scripts run the indicated models with the Python version of the simulator and print the inference results. They also generate an annotated output image.

Arguments passed for the following Python commands are found/explained in their corresponding Python files.

- Run a classifier network on the simulator.

```
~/sdk (vbx_env)
$ python $VBX_SDK/example/python/classifier.py MODEL.vnnx IMAGE.jpg
```

- Run a YOLO network (for example, YOLOv5n) on the simulator.

```
~/sdk (vbx_env)
$ python $VBX_SDK/example/python/yoloInfer.py MODEL.vnnx IMAGE.jpg -j yolov5n.json -l
coco.names -v 5 -t 0.25 --rgb --norm
```

- Run a YOLOv8 network on the simulator.

```
~/sdk (vbx_env)
$ python $VBX_SDK/example/python/yolov8.py MODEL.vnnx IMAGE.jpg -l coco.names --rgb --
scale 255
```

- Run a pose detection network on the simulator.

```
~/sdk (vbx_env)
$ python $VBX_SDK/example/python/posenetInfer.py MODEL.vnnx -i IMAGE.jpg --rgb --mean 128
--scale 128
```

2.6. API for Interacting with CoreVectorBlox

2.6.1. C API (Hardware and Simulator)

The following table lists the hardware and simulator enum types for C API.

Table 2-1. Enum Types

Parameter	Description

`vbx_cnn_calc_type_e`	`VBX_CNN_CALC_TYPE_UINT8,`
	`VBX_CNN_CALC_TYPE_INT8,`
	`VBX_CNN_CALC_TYPE_INT16,`
	`VBX_CNN_CALC_TYPE_INT32,`
	`VBX_CNN_CALC_TYPE_UNKNOWN`

`vbx_cnn_size_conf_e`	`VBX_CNN_SIZE_CONF_V250 = 0,`
	`VBX_CNN_SIZE_CONF_V500 = 1,`
	`VBX_CNN_SIZE_CONF_V1000 = 2`

`vbx_cnn_err_e`	`START_NOT_CLEAR = 3,`
	`OUTPUT_VALID_NOT_SET = 4,`
	`INVALID_NETWORK_ADDRESS = 6,`
	`MODEL_BLOB_INVALID = 7,`
	`MODEL_BLOB_VERSION_MISMATCH = 8,`
	`MODEL_BLOB_SIZE_CONFIGURATION_MISMATCH = 9,`

Table 2-1. Enum Types (continued)

Parameter	Description

`vbx_cnn_state_e`	`READY = 1,`
	`RUNNING = 2,`
	`RUNNING_READY = 3,`
	`FULL = 6,`
	`ERROR = 8`

2.6.1.1. Function Documentation

2.6.1.1.1. int model_check_sanity

```
int model_check_sanity(const model_t * model)
```

Model parsing function checks if the model parameter looks like a valid model.

Parameters

`model`: The model to check

Returns

It returns non-zero, if the model does not look valid.

2.6.1.1.2. size_t model_get_allocate_bytes

```
size_t model_get_allocate_bytes(const model_t * model)
```

It is used to get size required to store the entire model, include temporary buffers. The temporary buffers must be contiguous with the model data.

Parameters

`model`: The model to query

Returns

It returns the size required to store entire model in memory.

2.6.1.1.3. size_t model_get_data_bytes

```
size_t model_get_data_bytes(const model_t * model)
```

It is used to get the size required to store only the data part of the model.

Parameters

`model`: The model to query

Returns

It returns the size required to store the data part of the model.

2.6.1.1.4. vbx_cnn_calc_type_e model_get_input_datatype

```
vbx_cnn_calc_type_e model_get_input_datatype(const model_t * model, int
input_index)
```

It is used to get the datatype of an input buffer.

Parameters

The following are the parameters:

- `model`: The model to query
- `input_index`: The index of the input to get the datatype.

Returns

It returns the data type of model input.

2.6.1.1.5. size_t model_get_input_dims

```
size_t model_get_input_dims(const model_t* model, int input_index)
```

It is used to get the dimensions of elements of an input buffer.

Parameters

The following is the list of parameters:

- `model`: The model to query
- `input_index`: Index used to grab the total number of input dimensions.

Returns

It returns the number of dimensions of indexed input buffer.

2.6.1.1.6. size_t model_get_input_length

```
size_t model_get_input_length(const model_t * model, int input_index)
```

It is used to get the length in elements of an input buffer.

Parameters

The following is the list of parameters:

- `model`: The model to query
- `input_index`: The index of the input to get the length of elements.

Returns

It returns the length in elements of input buffer.

2.6.1.1.7. size_t model_get_num_inputs

```
size_t model_get_num_inputs(const model_t * model)
```

It is used to get the number of input buffers for the model.

Parameters

`model`: The model to query

Returns

It returns the number of inputs for the model.

2.6.1.1.8. size_t model_get_num_outputs

```
size_t model_get_num_outputs(const model_t * model)
```

It is used to get the number of output buffers for the model.

Parameters

`model`: The model to query

Returns

It returns the number of outputs for the model.

2.6.1.1.9. vbx_cnn_calc_type_e model_get_output_datatype

```
vbx_cnn_calc_type_e model_get_output_datatype(const model_t * model, int
output_index)
```

It is used to get the datatype of an output buffer.

Parameters

The following is the list of parameters:

- `model`: The model to query
- `output_index`: The index of the output to get the datatype.

Returns
It returns the data type of model output.

2.6.1.1.10. size_t model_get_output_dims

```
size_t model_get_output_dims(const model_t* model,int index)
```

It is used to get the number of dimensions of elements of an output buffer.

Parameters
The following is the list of parameters:
- `model`: The model to query
- `input`: The index of the output to get the length of output dimension index.

Returns
It returns the number of dimensions of indexed input buffer.

2.6.1.1.11. int* model_get_output_shape

```
int* model_get_output_shape(const model_t* model,int index)
```

It is used to get the dimensions of elements of a specific output buffer.

Parameters
The following is the list of parameters:
- `model`: The model to query
- `input`: The index of the input to get the length of output shape index.

Returns
It returns the dimensions in elements of output buffer.

2.6.1.1.12. int* model_get_input_shape

```
int* model_get_input_shape(const model_t* model,int index)
```

It is used to get dimensions of elements of a specific output buffer.

Parameters
The following is the list of parameters:
- `model`: The model to query
- `input_index`: The index of the input to get the length of input shape index.

Returns
It returns the dimensions in elements of output buffer.

2.6.1.1.13. size_t model_get_output_length

```
size_t model_get_output_length(const model_t * model, int index)
```

It is used to get the length in elements of an output buffer.

Parameters
The following is the list of parameters:
- `model`: The model to query
- `output_index`: The index of the output to get the output length.

Returns
It returns the length in elements of output buffer.

2.6.1.1.14. float model_get_output_scale_value

```
float model_get_output_scale_value(const model_t * model, int index)
```

It is used to get the amount of output values should be scaled to get the true values.

Parameters
The following is the list of parameters:

- `model`: The model to query
- `output_index`: The output for which to get the scale value.

Returns
It returns the scale value for model output.

2.6.1.1.15. int model_get_output_scale_fix16_value

```
int model_get_output_scale+fix16_value(const model_t * model, int index)
```

It is used to get the amount of output values should be scaled to get the true values.

Parameters
The following is the list of parameters:

- `model`: The model to query
- `index`: The output for which to get the scale value.

Returns
It returns the scale value for model output in fix16 format.

2.6.1.1.16. int model_get_input_scale_fix16_value

```
int model_get_input_scale_fix16_value(const model_t * model, int index)
```

It is used to get the amount of output values should be scaled to get the true values.

Parameters
The following is the list of parameters:

- `model`: The model to query
- `index`: The input for which to get the scale value.

Returns
It returns the scale value for model input in fix16 format.

2.6.1.1.17. int model_get_output_zeropoint

```
int model_get_output_zeropoint(const model_t * model, int index)
```

It is used to get the amount of output values should be offset by to get the true values.

Parameters
The following is the list of parameters:

- `model`: The model to query
- `index`: The output for which to get the zero point value.

Returns
It returns the zero point for model output in fix16 format.

2.6.1.1.18. int model_get_input_zeropoint

```
int model_get_input_zeropoint(const model_t * model, int index)
```

It is used to get the amount of input values should be offset by to get the true values.

Parameters
The following is the list of parameters:

- `model`: The model to query

- `index`: The input for which to get the scale value.

Returns
It returns the zero point for model ipnut in fix16 format.

2.6.1.1.19. vbx_cnn_size_conf_e model_get_size_conf

```
vbx_cnn_size_conf_e model_get_size_conf(const model_t * model)
```

It is used to get the size configuration that the model was generated for. It will be one of the following:

- 0 = V250
- 1 = V500
- 2 = V1000

Parameters
`model`: The model to query

Returns
It returns the size configuration the model was generated.

2.6.1.1.20. void* model_get_test_input

```
void* model_get_test_input(const model_t * model, int index)
```

It is used to get a pointer to test input to run through the graph for an input buffer.

Parameters
The following is the list of parameters:

- `model`: The model to query
- `input_index`: The input from which to get the test input.

Returns
It returns the pointer to test input of model.

2.6.1.1.21. vbx_cnn_err_e vbx_cnn_get_error_val

```
vbx_cnn_err_e vbx_cnn_get_error_val(vbx_cnn_t * vbx_cnn)
```

It is used to read error register and return the error.

Parameters
`vbx_cnn`: The `vbx_cnn` object to use

Returns
It returns the current value of error register.

2.6.1.1.22. vbx_cnn_state_e vbx_cnn_get_state

```
vbx_cnn_state_e vbx_cnn_get_state(vbx_cnn_t * vbx_cnn)
```

It is used to query `vbx_cnn` to see if there is a model running.

Parameters
`vbx_cnn`: The `vbx_cnn` object to use

Returns
Current state of the core. It can be one of the following:

- READY = 1 (Can accept model immediately)
- RUNNING = 2 (Model Running, can accept model eventually)

- RUNNING_READY = 3 (Model Running, can accept model immediately)
- FULL = 6 (Cannot accept model)
- ERROR = 8 (IP Core stopped)

2.6.1.1.23. vbx_cnn_t* vbx_cnn_init

`vbx_cnn_state_e vbx_cnn_get_state(volatile void * ctrl_reg_addr)`

It is used to initialize `vbx_cnn` IP Core. After this, the core accepts instructions. If any other function in this file runs without a valid initialized `vbx_cnn_t`, the result is undefined.

Parameters
`ctrl_reg_addr`: The address of the `VBX CNN S_control` port.

Returns
A `vbx_cnn_t` structure. `.initialized` is set on success, otherwise it is zero.

2.6.1.1.24. int vbx_cnn_model_poll

`int vbx_cnn_model_poll(vbx_cnn_t * vbx_cnn)`

It waits for model to complete.

Parameters
`vbx_cnn`: The `vbx_cnn` object to use

Returns
The returns can be one of the following:

- 1 = If network is running
- 0 = If network is done
- −1 = If there is an error during network processing
- −2 = No network is running

2.6.1.1.25. int vbx_cnn_model_start

`int vbx_cnn_model_start(vbx_cnn_t * vbx_cnn, model_t * model, vbx_cnn_io_ptr_t * io_buffers)`

It runs the model with the I/O buffers specified. One model can be queued while another model is running to achieve peak throughput. In that case, the calling code looks like:

```
vbx_cnn_model_start(vbx_cnn,model,io_buffers);
 while (input = get_input()){
   io_buffers[0] = input;
   vbx_cnn_model_start(vbx_cnn,model,io_buffers);
   while(vbx_cnn_model_poll(vbx_cnn)>0);
}
while(vbx_cnn_model_poll(vbx_cnn)>0);
```

Parameters
The following is the list of parameters:

- `vbx_cnn`: The `vbx_cnn` object to use
- `model`: The model to query
- `io_buffers`: Array of pointers to the input and output buffers. The pointers to the input buffers are first, and are followed by the pointers to the output buffers.

Returns
Non zero if model does not run. It occurs if `vbx_cnn_get_state()` returns FULL or ERROR.

2.6.1.1.26. int vbx_cnn_model_wfi

```
int vbx_cnn_model_wfi(vbx_cnn_t * vbx_cnn)
```

It waits for interrupt to determine model completion.

Parameters

`vbx_cnn`: The `vbx_cnn` object to use

Returns

The following is the list of possible returns:

- 1 = If network is running
- 0 = If network done or no network is running
- −1 = If error in processing network

2.6.1.1.27. void vbx_cnn_model_isr

```
void vbx_cnn_model_isr(vbx_cnn_t * vbx_cnn)
```

It interrupts service register.

Parameters

`vbx_cnn`: The `vbx_cnn` object to use

Returns

If a model is running, it clears the `output_valid` register once model is done.

2.6.2. Python API (Simulator Only)

2.6.2.1. vbx.sim.Model

Methods

The following is the list of methods:

- `__init__(self,model_bytes)`: Creates the model object from the bytes object passed into method.
- `run(self,inputs)`: Runs the model with inputs passed as a list of numpy arrays. It returns a list of numpy arrays as output.

Attributes

The following is the list of attributes:

- `num_outputs`: The number of outputs of this model.
- `num_inputs`: The number of inputs of this model.
- `output_lengths`: A list of lengths in number of elements for each output buffer of the model.
- `output_dims`: A list of dimensions for each output buffer of the model.
- `input_lengths`: A list of lengths in number of elements for each input buffer of the model.
- `input_dims`: A list of dimensions for each input buffer of the model.
- `output_dtypes`: A list of `numpy.dtype` for output describing the element type of each output buffer of the model.
- `input_dtypes`: A list of `numpy.dtype` for output describing the element type of each input buffer of the model.
- `output_scale_factor`: A list of floats describing how to scale each output buffer of the model.
- `description`: A string that the model is generated with describing the model.
- `test_input`: A list of inputs that can be passed into `Model.run()` as test data.

2.7. C Postprocessing

For a list of currently supported Postprocessing functions, see the Github repository.

3. Tutorials

The SDK contains a set of end-to-end tutorials in the `tutorials` directory.

Tutorials are organized by their source.

For every given tutorial, there is a script for users to run that performs the following steps:

1. Download the corresponding neural network using the respective source.
2. If not already in TF Lite format, transform the downloaded network into a TF Lite format using openvino2tensorflow, onnx2tf or tflite_quantize.
3. Once generated, run `tflite_preprocess` to add a preprocessing layer prior to generating the VNNX blob.
4. Create the VNNX blob by calling VectorBlox's graph generation tool, `vnnx_compile`.
5. Simulate the VNNX blob using a Python inference script from example/python directory with our provided test image.

These scripts are examples of the complete pipeline for generating a VNNX blob for a neural network. Users may use or modify these scripts to generate a VNNX for their own custom network to fit their use cases.

To run a tutorial script, navigate to the appropriate network tutorial directory and run the following command:

```
~/sdk/tutorials/[source]/[network_name] (vbx_sdk)
$ bash [network_name].sh
```

4. Application Notes

Contains application notes for tasks outside of the SDK, including creating advanced demos.

5. References

The following documents are referred in this user guide:

- *CoreVectorBlox IP Handbook*
- OpenVINO™ Toolkit

6. Revision History

The revision history describes the changes that were implemented in the document. The changes are listed by revision, starting with the most current publication.

Revision	Date	Description
H	5/2025	The following is a list of changes made in the revision H of the document: • Updated for version 2.0.2 • Updated Supported Operating Systems • Updated SDK • Added VectorBlox SDK tflite_postprocess • Added C Postprocessing
G	12/2024	The following is a list of changes made in the revision G of the document: • Updated for version 2.0 – Certain C functions updated to support int8_t parameters over fix16_t – Updated VNNX generation flow to reflect 2.0 TF Lite-based flow – Updated tutorial directory layout – Update environment requirements – Added more details about new TF Lite-conversion tools – Updated VNNX generation command usage – Added more python inference scripts and example usages • Added VectorBlox SDK generate_npy • Added VectorBlox SDK tflite_preprocess • Added VectorBlox SDK tflite_cut • Added int* model_get_output_shape • Added int* model_get_input_shape • Added int model_get_output_scale_fix16_value • Added int model_get_input_scale_fix16_value • Added int model_get_output_zeropoint • Added int model_get_input_zeropoint • Added section, int_post _process_ultra_nms • Added section, int_post_process_ultra_int8 • Updated TFLite Layers • Updated C Simulation • Updated Python Simulation
F	06/2024	Updated for version 1.4.5 • Updated with pose estimation demo • Added POSNET as post processing option under section C Simulation • Added following for Function Documentation for Post Processing: – int decodeMultiplePoses • Added following in Function Documentation for Python API: – vbx.postprocess.posenetProc – int decodeMultiplePoses

Revision History (continued)

Revision	Date	Description

E	09/2023	Updated for version 1.4.4
		• Updated face_t in Post Processing C API with object_t
		• Added functionality for interrupt based approach with
		  – int vbx_cnn_model_wfi
		• Added following in Function Documentation for Post Processing:
		  – int post_process_lpd
		  – fix16_t post_process_lpr
		• Added following in Function Documentation for Python API:
		  – vbx.postprocess.lpd
		  – vbx.postprocess.lpr
		• Added following in Function Documentation for C API:
		  – int vbx_cnn_model_wfi
		  – void vbx_cnn_model_isr

D	09/2022	Updated for version 1.4 • Updated OpenVINO links • Updated the first step in the Setting up the Installation Environment section • Updated the Supported OpenVINO Layers table with new layers Maximum, Minimum, and Subtract • Updated the Python Simulation and C Simulation sections under the VBX Inference Simulation section • Added the following under the Function Documentation section – int* model_get_input_dims – int* model_get_output_dims • Added the following in the vbx.sim.Model section under the API for Interacting with CoreVectorBlox section – output_dims – input_dims • Added the following in the Function Documentation section under the Post Processing section: – int post_process_scrfd – int post_process_yolo – vbx.postprocess.scrfd • Updated yolo_cfg_t under the C API (Hardware and Simulator) section • Updated the int post_process_yolo section
C	02/2022	• Updated for version 1.3: – Added Tutorials section description and usage – Updated OpenVINO links and version to 2021.4
B	08/2021	• Updated the following in version 1.2: – Added KL Divergence option to model generation in VBX Graph Generation section – Added Blazeface, Retinaface, and SSD post processing functions in Function Documentation section

Revision History (continued)

Revision	Date	Description
A	04/2021	• Updated for version 1.1 (The document has been rebranded per Microchip standards. • Document number updated from 50200927 to DS60001701A per Microchip format.)
1.0	11/2020	Initial Revision

Microchip Information

Trademarks

ISBN: 979-8-3371-1173-5

Legal Notice

Microchip Devices Code Protection Feature