Confidential

DNN Parser

Design

Table of Contents

[2 References 2](#_Toc75440663)

[3 Purpose 3](#_Toc75440664)

[4 Background 3](#_Toc75440665)

[5 Overview 3](#_Toc75440666)

[6 Configuration File 4](#_Toc75440667)

[6.1 Command Line Override 6](#_Toc75440668)

[7 Input Reader 6](#_Toc75440669)

[7.1 TFLite 6](#_Toc75440670)

[7.2 PyTorch 7](#_Toc75440671)

[7.3 Output Dictionary 7](#_Toc75440672)

[8 DNN Processing 9](#_Toc75440673)

[8.1 Add Post-Processing 10](#_Toc75440674)

[8.2 Reserve RAM for Padding 11](#_Toc75440675)

[8.3 Finalize Operation Parameters 12](#_Toc75440676)

[8.4 Split Multi-layer Convolutions 12](#_Toc75440677)

[8.5 Allocate Scratch Buffers 14](#_Toc75440678)

[8.6 Create Memory Structures 15](#_Toc75440679)

[9 Output Writer 16](#_Toc75440680)

[9.1 ROM Mode 17](#_Toc75440681)

[9.2 Sim Mode 17](#_Toc75440682)

[9.3 I2C Mode 18](#_Toc75440683)

[9.4 Summary File 18](#_Toc75440684)

[10 Operations Interface 19](#_Toc75440685)

[11 Internal vs. External Use 19](#_Toc75440686)

[12 Revision History 20](#_Toc75440687)

# References

|  |  |  |
| --- | --- | --- |
| Ref Num | Description | Link |
| 1 | DNN Firmware Design | <https://github.am.sony.com/Syseng/IMX634FW/blob/master/doc/design/imx681/imx681_dnn.docx> |
| 2 | DNN Firmware Specification | <https://dsp.semicon.sony.co.jp/sc/104/06/400-Deliverable/200-Design/500-Logic%20design/000-Logic_Specification/IMX681_DNN_Specification.pptx> |
| 3 | BMC Memory Configuration | <https://dsp.semicon.sony.co.jp/sc/104/06/400-Deliverable/200-Design/500-Logic%20design/400-Design/710-BMCTOP/300-FunctionSpecification/IMX681_BMC_Memory_Config.xlsx> |
| 4 | DNN Verification Environment | <https://github.am.sony.com/Syseng/IMX634FW/blob/master/doc/design/imx681/imx681_dnn_verification.docx> |
| 5 | DNN Post Processing Spec | <https://github.am.sony.com/Syseng/IMX634FW/blob/master/doc/design/imx681/imx681_dnn_postprocessing.pptx> |

# Purpose

This document describes the design and implementation of the DNN Compiler tool that was developed by the firmware team as part of the IMX681 project. The purpose is to provide an overview on the structure and functionality of this tool.

# Background

The IMX681 image sensor introduced a feature in which a Deep Neural Network (DNN) can be run on-chip to perform object detection, object classification, or other similar imaging algorithms. A default detection algorithm developed by Sony is stored in the chip’s ROM, but it also supports a Reconfigurable DNN mode in which a customer can load their own DNN into RAM and override the default. The intended workflow is that the customer would develop and train their DNN in a deep learning framework (Tensorflow or PyTorch), then convert it to a format that the firmware can interpret and load it to RAM over I2C.

In order to support this functionality, the firmware team developed the DNN Compiler tool. This Python-based tool reads in a saved DNN model from either a .tflite or PyTorch save file, extracts all information that is required to recreate the DNN, and converts it to a format that is compatible with the IMX681 firmware. This tool was initially developed for internal use during the development and testing of the firmware, but it has evolved into a customer facing tool. By providing the customer with this tool, Sony enables the customer to develop their own DNN and load it to IMX681’s memory without sharing any details about the DNN.

For more information on the DNN firmware and the structures that are used to represent a DNN, see reference documents [1] and [2].

# Overview

The block diagram in Figure 1 gives an overview of the DNN Compiler’s flow & components.



Figure - DNN Compiler Block Diagram

The DNN Compiler is invoked from the command line using the following syntax:

python dnn\_compiler.py [config-filename]

The [config-filename] argument specifies the path to a text-based configuration file. This configuration file defines information like input/output file paths, image sensor details (e.g. memory addresses & sizes, endianness, etc), and user settings (e.g. output mode, memory allocation scheme, etc). Section 6 describes the file format in more detail.

One of the parameters in the configuration file is the path to a saved DNN model, stored as either a Tensorflow .tflite file or a PyTorch save file. This file includes all information about the trained DNN, such as its structure, layers, weights, and biases.

The DNN Compiler is broken up into three main layers:

1. Input Reader (Tflite Reader & PyTorch Reader)
2. DNN Processing
3. Output Writer

The Input Reader is responsible for extracting all relevant information from the saved DNN model and loading it into a python dictionary. All Tensorflow or PyTorch specific code is contained within this layer. The Tflite Reader and Pytorch Reader will both output a dictionary in the same exact format, and the remainder of the DNN Compiler is agnostic to which format the saved model was stored in. This layer is described in more detail in Section 7.

The DNN Processor performs a series of tasks that modify or augment the DNN dictionary to make it compatible with the firmware. This includes optimizations, memory allocation, and adding a post-processing layer. This layer is described in more detail in Section 8.

Finally, the Output Writer takes the final DNN dictionary, formats it based on the output mode, and writes it to the final file format. The final output can be in one of three formats:

* Auto-generated C code that can be compiled into the firmware for the ROM model
* System and VPU memory images that can be loaded to memory in an RTL simulation environment
* I2C write sequence to load the DNN to RAM on silicon

In addition to this, a summary text file is always generated that describes the DNN. The Output Writer and file formats are described in more detail in Section 9.

# Configuration File

The configuration file follows a simple human-readable text format. Each configuration parameter is defined with a simple key-value pair as follows:

PARAMETER\_NAME value

Any line that begins with # is interpreted as a comment and ignored by the DNN Compiler. A base configuration file can be included in another configuration file using the following syntax:

INC\_CONFIG relative\_path/filename.inc

In this case, all values from the base configuration are effectively copied to the configuration file that includes it.

The DNN Compiler comes with sample configuration files in the following directory structure:

configs

* imx681\_detection\_rom.cfg
* imx681\_detection\_sim.cfg
* imx681\_detection\_i2c.cfg
* imx681\_classification\_sim.cfg
* imx681\_classification\_i2c.cfg
* base
  + sensor
    - imx681.inc
  + dnn
    - sony\_detection.inc
    - sony\_classification.inc
* test
  + test\_net\_0\_sim.cfg
  + …
  + test\_net\_7\_sim.cfg

The base directory contains base configurations that are included by other configurations. base/sensor/imx681.inc defines common configuration settings that are sensor-specific and always apply when the target platform is IMX681. base/dnn/sony\_detection.inc and base/dnn/sony\_classification.inc contain common configuration items to load DNN models that Sony has developed. The first is a detection algorithm that supports face, hand, and human detection (the default ROM model for IMX681) and the second is an object classification algorithm.

At the top level of the configs directory, there are full configuration files that tie these base configurations together. The naming convention used for these configuration files is:

<sensor>\_<dnn>\_<output mode>.cfg

For example, imx681\_classification\_i2c.cfg contains a full configuration to generate I2C load files to run the sony\_classification model on IMX681.

In addition to these files, the test directory contains more full configurations to load various test networks that are used for verifying the DNN Compiler and firmware (e.g. for Layer Parameter Tests). For more information on these, see reference document [4].

For more information on all the individual configuration parameters available, see the comments in each of these sample files. This document does not provide a full list, but some of the more complicated parameters are explained throughout Sections 7 - 9.

## Command Line Override

Individual parameters from a configuration file can be overridden from the command line using the following syntax:

python dnn\_compiler.py <config\_file> PARAMETER0=value0 PARAMETER1=value1 …

Any number of parameters can be overridden using this method. In addition, if INC\_CONFIG=value is provided, the specified configuration file will be included and any values in it will override existing values.

For example, the following command would use the imx681\_classification\_sim.cfg configuration, but change the OUTPUT\_MODE parameter from “sim” to “rom”:

python dnn\_compiler.py imx681\_classification\_sim.cfg OUTPUT\_MODE=rom

# Input Reader

The TfliteReader and PytorchReader are responsible for reading DNN models from a Tensorflow or PyTorch save file, respectively. The output of these two modules is a python dictionary that includes the following information:

* weights & biases
* quantization parameters
* buffer sizing information for all layer inputs & outputs in the DNN
* details about each layer in the DNN

Subsections 7.1 and 7.2 provide details about each component, then Subsection 7.3 provides details about the format of the common output dictionary.

## TFLite

The TFLiteReader uses the FlatBuffers open source library from Google to read the .tflite file. For information on the one-time setup that was performed for this, see Section 11.1.

The following table summarizes which Tensorflow layer types are supported by the DNN Compiler, and which DNN Operation IDs they are each mapped to in the firmware:

|  |  |  |
| --- | --- | --- |
| Tensorflow Layer Type | Firmware Operation ID | Supported Parameter Values |
| CONV\_2D | CONV\_2D | stride: [1x1, 2x2, 3x3]  filter size: [1x1, 3x3, 5x5]  padding: [same, valid]  activation: [none, relu, relu6]  dilation factor: [1]  groups: [1] |
| DEPTHWISE\_CONV\_2D | DEPTHWISE\_CONV\_2D | stride: [1x1, 2x2, 3x3]  filter size: [1x1, 3x3, 5x5]  padding: [same, valid]  activation: [none, relu, relu6]  dilation factor: [1] |
| FULLY\_CONNECTED | FULLY\_CONNECTED | keep\_dims: [false]  weight\_format: [default]  activation: [none, relu, relu6] |
| RELU  RELU6 | RELU | clip\_max: [0 – 127] |
| ADD  SUB | ADDSUB | modes:  [matrix + matrix,  matrix + scalar,  matrix + per channel scalars]  activation: [none, relu, relu6] |
| MUL | MULTIPLY | mode:  [matrix \* matrix,  matrix \* scalar]  activation: [none, relu, relu6] |
| CONCATENATION  QUANTIZE\* | CONCATENATE | num inputs: [1 – 6]  axis: [1, 2, 3]  activation: [none, relu, relu6] |
| RESIZE\_NEAREST\_NEIGHBOR | INTERPOLATE | h\_scale: [1 - 255]  w\_scale: [1 – 255]  align corners: [false]  half pixel centers: [false] |
| LOGISTIC | SIGMOID | - |
| SOFTMAX | SOFTMAX | axis: [0 – 3]  beta: [1.0] |
| MAX\_POOL\_2D | MAX\_POOL | stride: [1x1 - 6x6]  filter size: [2x2]  padding: [same, valid]  activation: [none, relu, relu6] |
| RESHAPE | RESHAPE | - |
| TRANSPOSE | TRANSPOSE | 2D input only (1 batch, 1 channel) |

\* QUANTIZE is implemented as a CONCATENATE with only one input. This effectively just copies the input buffer to the output buffer while re-quantizing with a different scale and zeropoint

In addition to this, the following layer types can be included in a .tflite file, but they will be ignored by the Compiler:

|  |  |
| --- | --- |
| Tensorflow Layer Type | Reason For Ignoring |
| SHAPE  STRIDED\_SLICE  PACK | When a RESHAPE layer is included in a Tensorflow model, a path of SHAPE 🡪 STRIDED\_SLICE 🡪 PACK is used to get the image’s dimensions. But, the firmware stores the dimensions as a layer parameter instead and does not need this path |
| CUSTOM  DEQUANTIZE | CUSTOM layers are generally used to implement some post-processing at the end of the DNN. However, the DNN Compiler adds its own post-processing and needs to strip any existing post-processing from the DNN.  In some cases, these CUSTOM layers may take in a floating point input, so a DEQUANTIZE layer comes before it. This, too, can be ignored in the firmware implementation. |

## PyTorch

TODO

## Output Dictionary

The output of the TfliteReader or PyTorchReader is a single dictionary with the following fields:

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| static\_data | list of uint8 | Values of all weights & biases for all layers, represented as uint8’s |
| quant\_params | list of 2 float pairs: [scale zeropoint] | Quantization parameters for all layers, represented as Python floats |
| misc\_data | empty list | Reserved for future use in DNN Processor (see Section 9) |
| buffers | list of dictionaries | Details about each input, output, and intermediate result buffer in the DNN (see below) |
| operations | list of dictionaries | Details about each layer in the DNN, in sequential order (see below) |

Each buffer in the DNN is represented by a Python dictionary with the following fields:

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| data\_signed | boolean | True if the data is signed, false if it is unsigned |
| data\_type | DataType enum | Data type (char, short, or long) |
| bpp\_shift | integer | Number of bytes per pixet is (1 << bpp\_shift). Example: a SHORT is 2 bytes (1 << 1), so bpp\_shift is 1. |
| quant\_type | QuantType enum | Indicates if quantization is PER\_TENSOR, PER\_CHANNEL, or PER\_BATCH |
| dimensions | list of 4 integers | Dimensions of this buffer [num\_batches, num\_rows, num\_cols, num\_channels] |
| data\_start\_row | integer | Start row of this buffer’s data if only a portion of the buffer is used (always 0 at this point) |
| data\_start\_col | integer | Start column of this buffer’s data if only a portion of the buffer is used (always 0 at this point) |
| data\_num\_rows | integer | Number of rows of this buffer’s data if only a portion of the buffer is used (always equal to dims[1] at this point) |
| data\_num\_cols | integer | Number of columns of this buffer’s data if only a portion of the buffer is used (always equal to dims[2] at this point) |
| quant\_start\_idx | integer | Start index of this buffer’s quantization parameters in the quant\_params list |
| buffer\_id | integer | Unique ID to identify this buffer |
| num\_connections | integer | Number of layers this buffer is connected to |
| parent | integer | If this buffer is a cropped portion of some larger buffer, this indicates the larger buffer’s index. Otherwise, it is -1. (always -1 at this point) |
| prepended\_dims | integer | If this buffer was less than 4 dimensions, this is the number of dimensions that were prepended to make it a 4D buffer |
| buffer\_type | BufferType enum | Indicates if this buffer is:  MODEL\_INPUT: the primary input to the DNN  SCRATCH\_RAM: an intermediate layer’s output  STATIC\_DATA: read-only input to one layer (e.g. weights and biases) |
| start\_idx | integer | Start index of this buffer’s data in its memory section (at this point, there is no overlap and these indices are sequential) |

Each layer in the DNN is represented by a Python dictionary with the following fields:

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| op\_id | DNNOperationID enum | ID that indicates which layer type this is |
| inputs | list of integers | For each input to this buffer, the index of the buffer in the buffers list |
| outputs | list of integers | For each output from this buffer, the index of the buffer in the buffers list |
| working\_mem\_addr | integer | Address of this layer’s working memory (always 0 at this point) |
| parameters | dictionary | Dictionary of all relevant layer parameters (fields vary based on layer type) |

As noted in these tables, some of the fields in these dictionaries are unused or always at a constant value at the output of the Input Reader. However, these fields are created here so that they can be used in the DNN Processor.

# DNN Processing

While the output of the Input Reader contains all information necessary to reconstruct a DNN, there are many adjustments and optimizations that need to be made to prepare it to run in the image sensor firmware. So, the DNN Processing modules read the DNN dictionary that was generated by the Input Reader, performs a series of tasks that augment or modify the dictionary, then generates a final DNN dictionary and memory dictionaries that contains all information that the firmware will need. Figure 2 summarizes all these tasks, and each one is described in more detail in the following subsections.



Figure - DNN Processor Flow

## Add Post-Processing

When a DNN is run on-chip in the IMX681 firmware, some final post processing is necessary to convert the DNN’s output to its final format that is written to registers and determine when to notify the host that an object is present. The DNN firmware supports two different types of post-processing:

1. Anchor Box Post-Processing
2. Threshold Post-Processing

It is the responsibility of the DNN Compiler to add a layer for one of these types of post-processing to the end of the DNN. In order to do this, the DNN Compiler’s configuration file has a series of parameters that tell it how each output of the DNN maps to an input of the post-processing layer:

* POSTPROC\_DNN\_DATA\_OUT\_IDX
* POSTPROC\_DNN\_DATA\_START\_ROW
* POSTPROC\_DNN\_DATA\_START\_COL
* POSTPROC\_DNN\_DATA\_NUM\_ROWS
* POSTPROC\_DNN\_DATA\_NUM\_COLS
* POSTPROC\_COMPARE\_VALS\_OUT\_IDX
* POSTPROC\_COMPARE\_VALS\_START\_ROW
* POSTPROC\_COMPARE\_VALS\_START\_COL
* POSTPROC\_COMPARE\_VALS\_NUM\_ROWS
* POSTPROC\_COMPARE\_VALS\_NUM\_COLS

It also has a single POSTPROCESSING parameter that defines the type of post-processing (ANCHOR\_BOXES or THRESHOLD), as well as some layer-specific parameter information. These configuration parameters and details on both post-processing types are described in reference document [5].

Based on these configuration settings, the DNN Compiler will do the following:

* Create new buffers for each post-processing input
  + These buffers may be a cropped region of the existing output buffers
* Add the new buffers to the dnn dictionary’s buffer list.
* Add the post processing layer to the end of the dnn dictionary’s operation list.

Note: For testing purposes, this mechanism can also be used to add a layer to the end of the DNN. For example, test modes have been added that allow a ROI\_POOL or GENERATE\_PROPOSALS layer to be appended to the end of the DNN so that the firmware team could test these layers before the firmware team had a DNN containing them.

## Reserve RAM for Padding

Some layer types, such as CONV2D, DEPTHWISE\_CONV2D, and MAX\_POOL may require rows and columns of zero padding to be added around an input buffer during processing. In order to implement this in firmware, the size of the input buffer needs to be increased to reserve space for padding. In this case, the previous layer will only write to the data region of the buffer, and the layer that requires padding will initialize the padding region before processing the entire buffer.

In general, the amount of padding needed on each axis by a convolution that uses “SAME” padding mode can be calculated as follows:

* total\_padding = max(filter\_size, stride) – stride

Then, the output size is calculated as follows:

* out\_size = (in\_size – filt\_size + total\_padding)/stride

If the output size is not an integer, additional padding will be added to ensure that it is. Finally, the total padding is split between the beginning and end of the image, where the end of the image gets extra rows/columns if there is an odd number:

* begin\_padding = floor(total\_padding/2)
* end\_padding = total\_padding – begin\_padding

For all buffers that require padding, the DNN Compiler will make the following modifications to the buffer’s dictionary:

* Increase dimensions[1] and dimensions[2] to include padding rows and columns
* Set data\_start\_row, data\_start\_col, data\_num\_rows, and data\_num\_cols to point to the data region of the buffer (excluding padding)

For example, if a 2x4x5x3 buffer requires one row and column on each side of the buffer, the following modifications would be made:

* dims[1] = 6 (+2 for padding rows)
* dims[2] = 7 (+2 for padding columns)
* data\_start\_row = 1 (skip top padding row)
* data\_start\_col = 1 (skip left padding row)
* data\_num\_rows = 4 (original buffer size)
* data\_num\_cols = 5 (original buffer size)

In some specific cases, the DNN Compiler may add padding to other buffers to allow optimizations to be performed in the firmware. For example, if a convolution with a 1x1 filter is included in a DNN, a major optimization can be done in firmware if the input and output buffers contain the same amount of padding. So, padding may be added to the input or output buffer to allow this optimization to be performed.

## Finalize Operation Parameters

The Input Reader parses through a DNN one layer at a time and extracts all parameters associated with each layer as it goes. However, there are some additional parameters that the firmware expects to be stored for each layer type that need to be computed after the entire DNN has been loaded. The DNN Processor will compute these parameters at this point and add their values to the layers’ operation dictionaries.

Many of the parameters that need to be computed are pre-calculated constants. In the firmware, divide operations are very slow and can be costly to the performance of a layer. So, whenever possible, divides of two constants are avoided in firmware by performing the divide ahead of time in the Compiler and storing the result in memory. Other complex computations are also pre-computed to avoid run-time calculations that are required on every frame. As an example, it is very common for a layer’s processing to need to multiply results by the ratio of the input quantization scale factor, Sa, and the output quantization scale factor, Sb. Rather than performing a divide to calculate Sa/Sb in firmware, the DNN Compiler will calculate that result here, and store a single scale\_ratio as a layer parameter.

## Split Multi-layer Convolutions

Whenever a convolution (CONV2D or DEPTHWISE\_CONV2D) is performed in the firmware, the following buffers need to be stored in scratch RAM simultaneously:

* Input buffer (B x Ri x Ci x Ni bytes)
* Output buffer (B x Ro x Co x No bytes)
* Temporary result storage for one channel (Ri x Ci 4-byte words)

In the case of some large convolution operations, this can require a very large amount of memory to be used at a given time. For example, a convolution with a 1 x 120 x 160 x 8 input and 1 x 60 x 80 x 8 output requires the following amount of memory:

* Input Buffer: 120\*160\*8 =153,600 bytes
* Output Buffer: 60\*80\*8 = 38,400 bytes
* Temporary Results: 120\*160\*4 = 76,800 bytes
* **Total: 268,800 bytes**

This exceeds the total amount of scratch RAM available on IMX681. To solve this issue, the DNN Compiler looks for convolutions with large output buffers. When it finds them, it tries to split the image up into smaller sections (e.g. quadrants) and operate on each one individually until it reaches an operation with a smaller buffer size. Then, the results are stitched together into a single image.

As an example, consider the DNN below:

Diagram

Description automatically generated

The Compiler will convert these 4 layers into a 16 layer convolution as follows:

Graphical user interface, PowerPoint

Description automatically generated

In this case, the maximum amount of memory needed at a given time would be during the second convolution on the yellow path. At this time, the following amount of memory would be necessary:

* Result of the blue path: 15 \* 20 \* 8 = 2,400 bytes
* Result of the green path: 15 \* 20 \* 8 = 2,400 bytes
* Result of the orange path: 15 \* 20 \* 8 = 2,400 bytes
* Input buffer: 60 \* 80 \* 8 = 38,400 bytes
* Output buffer: 30 \* 40 \* 8 = 9,600 bytes
* Temporary Results: 60 \* 80 \* 4 = 19,200 bytes
* **Total: 74,400 bytes**

This memory requirement of 74,400 bytes is almost ¼ as large as the original requirement of 268,800 bytes.

When a buffer is partitioned, a new “buffer” is stored for each individual piece of the image. The new buffer’s “dimensions” field will reflect the total size of the large buffer (e.g. 120x160 in the example above), but the “data\_start\_col”, “data\_start\_row”, “data\_num\_cols”, and “data\_num\_rows” fields will be updated to point to the section that this buffer represents. As an example, the bottom-left quadrant’s input buffer would have the following settings:

* dimensions = [1 160 120 1]
* data\_start\_row = 80
* data\_start\_col = 0
* data\_num\_rows = 80
* data\_num\_cols = 60

In addition to this, the “parent” field will be populated with the buffer\_idx of the full large buffer, indicating that this buffer is a portion of a larger buffer.

In order to ensure that the output image produced by a multilayer convolution is identical to that produced by the original DNN, some amount of overlap between partitions may be needed. The DNN processor will determine how much overlap is needed and increase the dimensions of each buffer accordingly.

The configuration file contains two parameters that tell the DNN Compiler when and how to use multi-layer convolutions:

* ML\_CONV\_MAX\_OUT\_SIZE 4096
* ML\_CONV\_NUM\_PARTITIONS 4

With these example settings, any time a convolution layer’s output buffer is larger than 4,096 bytes, the Compiler will try to break it up into 4 quadrants of processing. However, there may be other restrictions that prevent a larger convolution from being broken up. For example, there must be a chain of consecutive convolutions that can be processed in parallel in order to be able to do a multi-layer convolution.

## Allocate Scratch Buffers

As described in the previous section, when each layer is executed in the firmware, the following buffers must be available in scratch RAM:

* Input buffer(s)
* Output buffer
* Temporary working memory

At this step in the DNN Compiler, we know the sizes of each input and output buffer, the number of layers each is used by (the “num\_connections” field), and the total amount of temporary working memory needed by each layer. Using this information, the DNN Compiler can:

* Allocate memory for each buffer, allowing locations of memory to be reused after a buffer is no longer needed
* Store the start address of a buffer’s reserved memory in the buffer’s “start\_idx” field
* Check that enough scratch RAM is available, and print an error if it is not

The size of scratch RAM is stored in the configuration file (e.g. base/sensor/imx681.inc):

1. MAX\_SCRATCH\_MEMORY\_SIZE

In addition to this, the following parameter is available to configure how memory is allocated:

1. SCRATCH\_MEMORY\_REVERSE\_ALLOC

This parameter selects whether memory is allocated starting with lower addresses (e.g. first buffer is at address 0) or if memory is allocated backwards, starting with higher addresses (e.g. first buffer is at address <memory\_size>-<buffer\_size>).

When VPU\_SCRATCH\_MEMORY\_REVERSE\_ALLOC is 0, lower addresses get used far more often than higher addresses. This is because the scratch memory only stretches into higher addresses when it is at its peak usage, but all smaller layers that use less memory end up only using lower addresses.

On IMX681, accessing higher memory addresses uses less power, because the Banked Memory Controller (BMC)’s 2nd and 3rd memory channel are smaller than its 1st memory channel. Setting VPU\_SCRATCH\_MEMORY\_REVERSE\_ALLOC to 1 allows the DNN Compiler to prefer higher addresses that are lower power and use lower addresses as little as possible.

## Create Memory Structures

In the IMX681 firmware, a DNN is represented by 5 memory structures:

* Optable – the operation table containing information about all buffers and operations in the DNN
* Quant Params – the quantization scale and zeropoint for all buffers
* Misc Data – additional parameters for various operations
* Static Data – constant weight and bias buffers
* Scratch RAM – space reserved for temporary results during DNN processing

These structures can be stored in one of four regions of memory based on the different output modes:

|  |  |  |
| --- | --- | --- |
| Memory Region | Use in ROM output mode | Use in other output modes |
| System RAM | - | Optable, Quant Params, Misc Data |
| System ROM | Optable, Quant Params, Misc Data | - |
| BMC RAM | Scratch RAM | Scratch RAM, Static Data |
| BMC ROM | Static Data | - |

During this step of DNN processing, the information from the DNN dictionary and configuration file is used to construct each of the memory structures (as an array of raw byte values) and allocate space for them in the corresponding memory regions. The output is two dictionaries: a memory region dictionary and a memory structure dictionary.

The memory region dictionary contains an entry for each of the four memory regions (System RAM, System ROM, BMC RAM, BMC ROM). For each one, the following fields are stored:

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| desc | string | Name used to refer to this memory region in output files |
| start\_addr | integer | The start address of this region in memory |
| max\_size | integer | The maximum number of bytes that fit in this region of memory |
| size | integer | The actual size, in bytes, of the data that is allocated in this region |
| structs | list of MemoryStructures | List of which structures will be stored in this region, in order. |

The memory structures dictionary contains an entry for each of the five memory structures (Optable, Quant Params, Misc Data, Static Data, Scratch RAM). For each one, the following fields are stored:

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| desc | string | Name used to refer to this memory structure in output files |
| start\_addr | integer | The start address of this structure in memory |
| size | integer | The size, in bytes, of the data |
| bytes | list of lists of bytes | Raw byte values of the data in this region. This is a 2D list of bytes, were each element in the structure is represented as a 1D list. For example, each buffer or operation in the optable is represented as a list of bytes, and the overall bytes structure concatenates them all together. |

Using the information in these dictionaries, the Compiler can determine if the DNN fits in memory, what values to write to DNN address registers, and the raw data that needs to be written to output files.

# Output Writer

The Output Writer takes all information that has been added to the DNN dictionary, Memory Region dictionary, and Memory Structure dictionary, and uses it to write files in a format that the firmware understands. The information in the operations dictionary is converted to the S\_DNN\_OPERATION format used in the firmware and the information in the buffers dictionary is converted to the S\_DNN\_BUFFER\_DESCRIPTOR format that is used in the firmware. For each layer type, layer parameters are handled in one of the following two ways:

1. Parameters are packed into a 32-bit integer as bitfields, and stored in the S\_DNN\_OPERATION’s param field
2. Parameters are written to the misc\_data array, and their start index is stored in the S\_DNN\_OPERATION’s param field

The final data can be written to one of three different file formats, based on the setting of the following configuration parameter:

OUTPUT\_MODE

The value of this parameter is a string with one of the following values: rom, sim, i2c. Each of these values is summarized in the following table, and the following subsections provide more details.

|  |  |  |
| --- | --- | --- |
| Mode | Output File Format | Usecase |
| rom | C source (.c and .h) | Auto-generating firmware source code that can be compiled directly into the firmware for a ROM model. |
| sim | Binary memory images (.bin) | Creating raw binary images of the DNN’s system RAM and BMC RAM segments that can be loaded directly to memory in an RTL simulation environment. |
| i2c | Binary I2C sequence (.bin) | Creating an I2C sequence that can be used to load this DNN directly to a physical image sensor. |

## ROM Mode

In ROM mode, the following C source files are generated in the output directory:

|  |  |
| --- | --- |
| Filename | Description |
| dnn\_common\_auto.h | Header that defines all constants and structures that are shared between firmware and the DNN Compiler, and declares objects that are defined in the remaining files. |
| <name>\_misc\_data\_auto.c | Defines the misc data array’s contents, as a int8[] |
| <name>\_optable\_auto.c | Defines the optable’s contents, as a int8[] |
| <name>\_quant\_params\_auto.c | Defines the quant\_params array’s contents, as a int8[] |
| dnn\_op\_func\_table\_auto.c | Defines an operation function table that maps a DNN\_OPERATION\_ID to a function that will implement that operation. |

This mode is only intended to be used by the firmware team. The output files can be copied to the following location in the firmware source code, then compiled directly into the firmware build:

src/category/dnn/auto

Each file that defines a structure is prefixed with a DNN name in case the firmware needs to support multiple ROM models for different imaging modes in the future.

## Sim Mode

For a reconfiguration DNN, the firmware requires some memory structures to be stored in system RAM (e.g. optable, quantization parameters, misc data array), and some memory structures to be stored in BMC RAM (e.g. weights and biases). In the RTL Simulation and FW Simulation environments, tools exist to quickly pre-load regions of these RAMs with constant data during initialization, and this is what is used to test the reconfigurable DNN prior to silicon tape-out.

When the output mode is “sim”, the firmware will generate three output files:

|  |  |
| --- | --- |
| Filename | Description |
| <name>\_system\_memory.bin | The data that is required in system RAM to run this DNN |
| <name>\_vpu\_memory.bin | The data that is required in BMC RAM to run this DNN |
| <name>\_registers.txt | Register setting information for the simulator |

The DNN Compiler also outputs information about the sizes and start addresses of each of these memory segments, which can be used to set up a simulation. For more information about how these files are used, see reference document [4].

## I2C Mode

When a customer loads their own DNN on real silicon, they will use I2C to write directly to memory regions and configure DNN registers. When the output mode is “i2c”, the firmware will generate a single file, <name>\_ load\_sequence\_i2c.bin, that contains the exact byte sequence that should be written over I2C to load this DNN. This I2C sequence is generally a sequence of 3 byte write operations, where the bytes are:

* Address MSB
* Address LSB
* Data

The sequence includes all writes that are necessary to load the DNN, including:

* I2C initialization sequence
* Configuring customer registers for the DNN
* Loading structures to system RAM and BMC RAM using Group Remappings
* Resetting group remappings

After a customer sends this raw sequence over I2C byte-by-byte, the firmware will be configured to run their DNN next time the image sensor enters DNN Internal mode.

## Summary File

In addition to outputting one of the file types described in the previous three subsections, the Compiler will *always* output a human-readable text file called <name>\_summary.txt.

This file provides a lot of useful information about the DNN. The header at the top of the file provides the following information:

* Size of each individual memory structure
* Amount of bytes used vs. available in System and BMC RAM
* Settings of customer registers that are necessary to load this DNN

After this, a description of each layer in the final DNN is provided. For each layer, the operation ID, parameters, and working memory address are provided. Then, each input and output buffer is described in detail. An estimate of how long each layer’s processing will take is given as well.

Finally, at the end of the file, the following information is provided:

* An estimate of the total processing time of the DNN
* A breakdown of which VPU instructions will be used, and how much time will be spent on each individual instruction type

# Operations Interface

As the DNN Compiler operates on layers in a DNN one-by-one, there are several cases where inputs and outputs need to be handled differently based on the layer type. For example, the TFLite Reader needs to extract different layer parameters from each layer type, the DNN Processor needs to reserve a different amount of working memory for each layer type, and the Output Writer needs to pack layer parameters differently based on the layer type. In order to keep all layer-specific code together, an Operations interface was developed. Each supported layer type has its own implementation of this interface in the following directory:

internal/operations

Each layer’s implementation has the following functions:

|  |  |  |
| --- | --- | --- |
| Function Name | Called From | Description |
| get\_op\_params\_tflite | InputReader | Extract the layer’s parameters from a .tflite file and store in dictionary |
| get\_op\_params\_pytorch | InputReader | Extract the layer’s parameters from a PyTorch file and store in dictionary |
| finalize\_op\_params | DNNProcessor | Calculate final parameter values for this layer based on the existing dictionary (see Section 8.3) |
| supports\_cropping | DNNProcessor | Check if this layer type supports operating on a cropped region of a larger buffer |
| op\_params\_to\_byte\_array | OutputWriter | Convert this operation’s parameter dictionary to a byte array and append it to the misc\_data structure, or pack the fields into a single 32-bit integer and return its value |
| op\_params\_to\_string | OutputWriter | Convert this operation’s parameter dictionary to a human-readable string (to write to summary file) |
| get\_working\_memory\_size | DNNProcessor | Calculate how much working memory is required by this layer based on buffer sizes |
| get\_processing\_time | OutputWriter | Estimate which VPU instructions will be used by this layer and how much time the processing will take |
| optional: add\_postprocessing | DNNProcessor | If this is a post-processing layer, create the layer and append it to the DNN dictionary |

# Internal vs. External Use

The DNN Compiler is used as both an internal tool for development within Sony, and as a customer facing tool that Sony provides to external customers. There are some features that are necessary for internal development, but should not be exposed to the customer because they either contain proprietary information or are experimental. These features are enabled or disabled using the following constant in the internal/constants.py:

DNN\_COMPILER\_INTERNAL\_USE

If this is set to True, all features are enabled. If it is set the False, the following limitations are set:

* ROM output mode disabled
* VPU instruction counts omitted from summary.txt output file
* The range of supported values for some layer parameters are limited to values that have been fully tested

When the DNN compiler is released to a customer, the following steps are taken to protect the code:

* DNN\_COMPILER\_INTERNAL\_USE should always be set to False
* The python source code is obfuscated using PyArmor

For more information on the release procedure, see README.md in the DNN Compiler GIT repository.

# Revision History

|  |  |  |
| --- | --- | --- |
| Revision | Author | Change |
| 2021-05-10 | Mike Soltiz | Initial revision |
| 2021-06-24 | Mike Soltiz | Updated to include information about I2C output mode, memory dictionaries, and internal vs. external mode. |
|  |  |  |