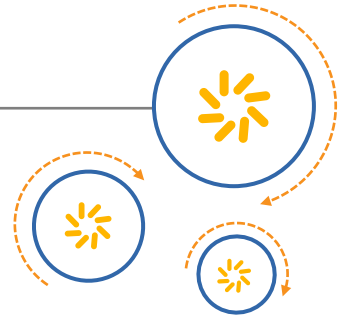




Qualcomm Technologies, Inc.



QNN Weight Sharing

Application Note

March 17, 2025

Confidential and Proprietary – Qualcomm Technologies, Inc.

NO PUBLIC DISCLOSURE PERMITTED: Please report postings of this document on public servers or websites to:
DocCtrlAgent@qualcomm.com.

Restricted Distribution: Not to be distributed to anyone who is not an employee of either Qualcomm Technologies, Inc. or its affiliated companies without the express approval of Qualcomm Configuration Management.

Not to be used, copied, reproduced, or modified in whole or in part, nor its contents revealed in any manner to others without the express written permission of Qualcomm Technologies, Inc.

Snapdragon Neural Processing Engine SDK is a product of Qualcomm Technologies, Inc. Other Qualcomm products referenced herein are products of Qualcomm Technologies, Inc. or its subsidiaries.

Qualcomm is a trademark of Qualcomm Incorporated, registered in the United States and other countries. Snapdragon Neural Processing Engine is a trademark of Qualcomm Incorporated. > Other product and brand names may be trademarks or registered trademarks of their respective owners.

This technical data may be subject to U.S. and international export, re-export, or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

© 2025 Qualcomm Technologies, Inc. All rights reserved.

Contents

1 Introduction..... 3

2 Weight Sharing 3

3 Model Preparation & Execution..... 3

4 HTP Configuration files 6

5 Weight Sharing recommendations 11

6 Limitations..... 15

7 Technical assistance 15

1 Introduction

This document provides guidelines and details on how to enable and achieve optimal weight sharing with models using QNN SDK tools.

It is assumed that the readers are familiar with the Qualcomm QNN SDK tools and the workflow for preparing and executing the models on Qualcomm Snapdragon platforms.

Please refer to the Qualcomm AI runtime SDK documentation for the QNN workflow.

2 Weight Sharing

The QNN weight sharing feature can be used while preparing a context binary that includes multiple graphs with shared weights. It enables combining and storage of common weights across up to 64 graphs within a single QNN context binary. Utilizing this feature helps reduce overall RAM & ROM memory usage by sharing common weights across multiple QNN graphs, especially for large models.

3 Model Preparation & Execution

The QNN weight sharing feature can be applied on models finetuned for various AI usecase using a common base model. The feature leverages the common weights across the fine tuned model variants and packages into the QNN context binary as shown in the below diagram.

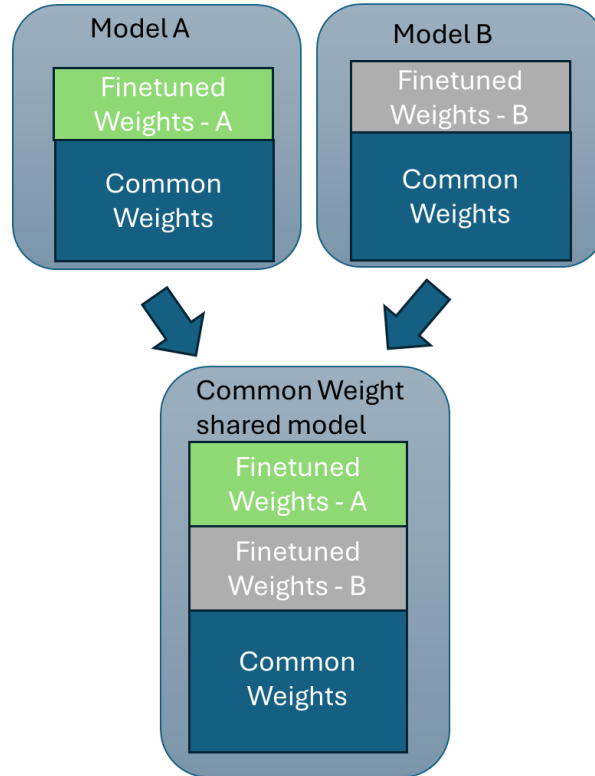


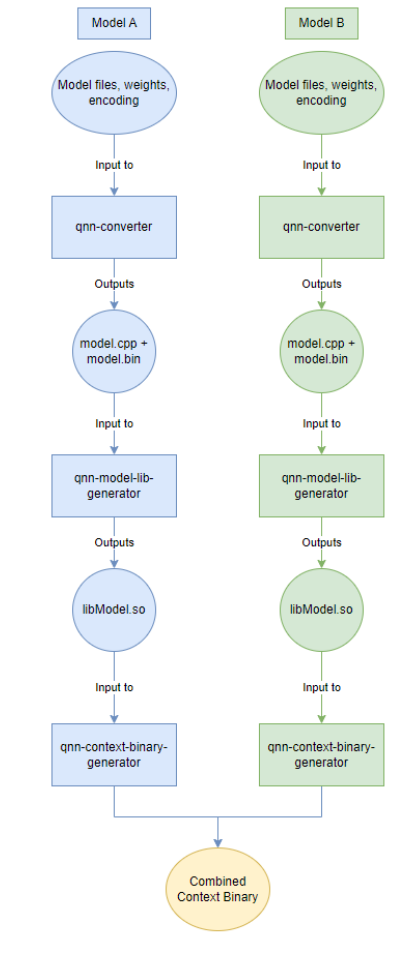
Fig 1: QNN Weight sharing feature using two models A, B finetuned from common base model.

Below is the procedure for using the QNN weight sharing feature using QNN SDK tools during model preparation step.

- Follow the QNN model conversion and model lib generation procedure using the QNN SDK tools.
- Once the model libs for the individual finetuned models are available, use the QNN context binary generation tool to generate the QNN context binary with common weights and finetuned weights as illustrated in the Fig 1 above.

NOTE: The QNN weight sharing feature is currently supported only on Linux host.

- Once we have created the QNN Context binary using above command we can execute the model on Snapdragon platforms normally using the QNN tools or integrate into the AI applications.



• Fig 2: Illustration of combined context binary creation

Below is an example for using the QNN weight sharing feature for generation a context binary with common and finetuned weights for model A, B (on Ubuntu Host)

```

./qnn-context-binary-generator --model<model-A.so>,<model-B.so>

--backend ${QNN_SDK_ROOT}/lib/x86_64-linux-clang/libQnnHtp.so

--binary_file context_binary_combined.bin

--config_file htp_backend_extension.json
  
```

- Below is an example for running the 2 models on Snapdragon X Elite platform (on Windows) using the combined context binaries.

```

.\qnn-net-run.exe --retrieve_context "context_binary_combined.bin"
  
```

```

--input_list "target_1.txt", "target_2.txt"

--backend " ${QNN_SDK_ROOT}\bin\arm64x-windows-
msvc\backend_libs\QnnHtp.dll"

--config_file "htp_backend_extensions.json"

--output_dir "output"

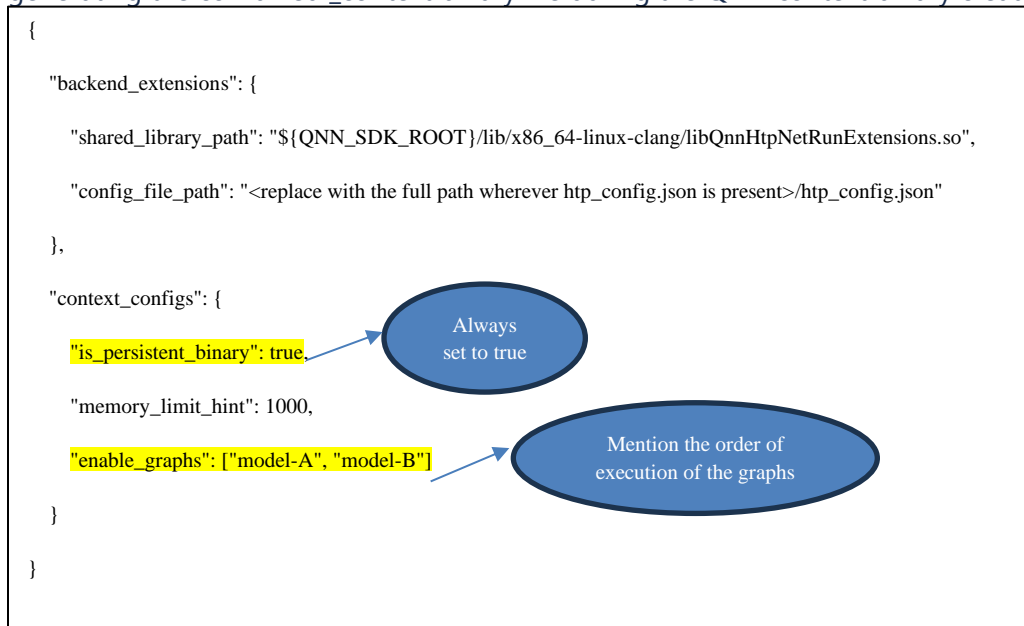
```

Note: target_1.txt and target_2.txt are the path of the text files that contain the paths to the input raw files for Model-A and Model-B respectively. The input files for the models needs to be comma separated like how it was mentioned above when there is more than one model. The paths within each input text file should be listed as input_1:=<path to input 1 raw> input_1:=<path to input 2 raw> input_3:=<path to input 3 raw>

4 HTP Configuration files

The HTP configuration files allow for choosing the available options for exercising the QNN weight sharing features. Let's look at the options that are available and their purpose in customizing the usage of QNN weight sharing feature as the QNN application requirements.

- **Model preparation phase:** The `htp_backend_extensions.json` file is used for generating the combined `_context` binary file during the QNN context binary creation phase



- **Model execution phase:** The `http_config.json` is used for choosing the required model for loading the models from the combined context binary into the RAM for execution of the finetuned models.

Currently there are two modes of loading the finetuned models based on tradeoff between model initialization time and the RAM footprint of the QNN application.

The `is_persistent_binary` flag needs to be set as true to enable the graph

switching property while the `enable_graphs` list helps to set the order of priority for graph execution.

- **Graph switching and Lazy loading:**

Graph switching allows all graphs to be enabled initially, keeping each graph in the unloaded state

- Each time, it loads the graph in the order specified in the enable graphs flag.
- This leads to faster execution of all graphs except the first due to the initialization time taken to enable all graphs.

For more information about this feature, refer the [docs](#).

```

{
  "graphs": [
    {
      "vtcm_mb": 8,
      "graph_names": ["model-A", "model-B"],
      "O": 3.0,
      "fp16_relaxed_precision": 1,
      "hvx_threads": 4
    }
  ],
  "devices": [
    {
      "soc_id": 60,
      "dsp_arch": "v73",
      "cores": [
        {
          "core_id": 0,
          "perf_profile": "burst",
          "rpc_control_latency": 100
        }
      ]
    }
  ],
  "context": {
    "weight_sharing_enabled": true
  }
}

```

The **weight sharing flag** is the one that is responsible for the feature hence is always set to be true.

- **Sampe** `htp_backend_extensions.json` while executing the “qnn-net-run” command


```
{
  "backend_extensions": {
    "shared_library_path": "${QNN_SDK_ROOT}/bin/arm64x-windows-msvc/backend_libs/QnnHtpNetRunExtensions.dll",
    "config_file_path": " <replace with the full path wherever http_config.json is present>/http_config.json"
  },
  "context_configs": {
    "is_persistent_binary": true,
    "memory_limit_hint": 1000,
    "enable_graphs": ["model-A", "model-B"]
  }
}
```

- Sample `http_config.json` while executing the “qnn-net-run” command

```

{
  "graphs": [
    {
      "vtcm_mb": 8,
      "graph_names": ["vae_decoder_64", "vae_decoder_128"],
      "O": 3.0,
      "fp16_relaxed_precision": 1,
      "hvx_threads": 4
    }
  ],
  "devices": [
    {
      "soc_id": 60,
      "dsp_arch": "v73",
      "cores": [
        {
          "core_id": 0,
          "perf_profile": "burst",
          "rpc_control_latency": 100
        }
      ]
    }
  ],
  "context": {
    "weight_sharing_enabled": true
  }
}

```

5 Weight Sharing recommendations

While the QNN weight sharing feature is model independent, it would result in optimized model RAM and ROM footprint by following certain guidelines.

- All the source models used with the QNN weight sharing need to be finetuned from a same base model to maximize the weight sharing and obtain best RAM & ROM optimization.

Checking the bin files generated from the "qnn-onnx-converter" command for the individual models by running the following command. The below command extracts the raw files for all the weights in the model into the specified output directory.

```
tar xf /path/to/Model-A.bin -C /output_path/
```

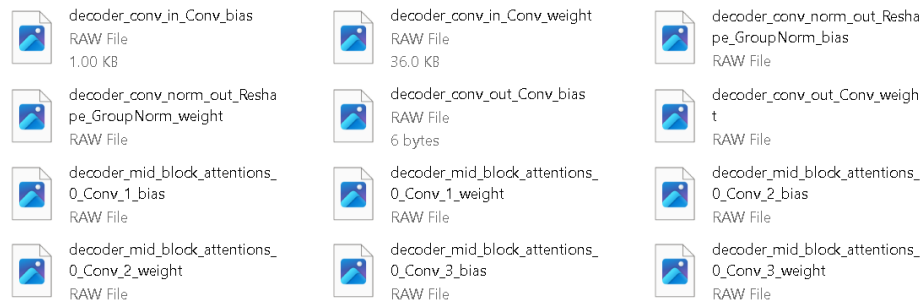


Fig 3: Example of Model-A raw files

As illustrated in Fig 3, the raw files for Model-A correspond to the model architecture's layer names and their respective weights. The tar command extracts the binary file into individual raw files for each layer, with the file names matching the layer names of the model. The contents of these raw files represent the individual model weights.

Repeating the above steps for the Model-B and comparing the raw files of the two models will reveal how similar their weights are. The weight sharing feature will be useful only if there is a significant similarity in weights

5.1 Matching AIMET encodings:

To allow appropriate weight sharing, the encodings must be matching between the graph for all the nodes. The encodings of the graphs could be found in their ".encodings" files. Here's an example of what they might look like:


```

import os

import hashlib

def get_file_hash(file_path):

    hasher = hashlib.md5()

    with open(file_path, 'rb') as file:

        buf = file.read()

        hasher.update(buf)

    return hasher.hexdigest()

def compare_files(file_path1, file_path2):

    hash1 = get_file_hash(file_path1)

    hash2 = get_file_hash(file_path2)

    return hash1 == hash2

def compare_folders(folder1, folder2):

    folder1_files = set(os.listdir(folder1))

    folder2_files = set(os.listdir(folder2))

    same_name_files = folder1_files.intersection(folder2_files)

    same_files_count = 0

    different_files_count = 0

    for file_name in same_name_files:

        file_path1 = os.path.join(folder1, file_name)

        file_path2 = os.path.join(folder2, file_name)

        if compare_files(file_path1, file_path2):

            same_files_count += 1

        else:

            different_files_count += 1

    unmatched_files_folder1 = folder1_files - folder2_files

    unmatched_files_folder2 = folder2_files - folder1_files

    print(f"Total files with the same names: {len(same_name_files)}")

    print(f"Total files that are identical: {same_files_count}")

    print(f"Total files that are different: {different_files_count}")

    print(f"Total files in folder1 with no matching name in folder2: {len(unmatched_files_folder1)}")

    print(f"Total files in folder2 with no matching name in folder1: {len(unmatched_files_folder2)}")

# Example usage

folder1 = "model-A\\raw_files_model1"

folder2 = "model-A\\raw_files_model2"

compare_folders(folder1, folder2)

```

- The example output of the above code is as follows:

- Models with weights being shared:**

```
Total files with the same names: 140
Total raw files that are identical: 140
Total raw files that are different: 0
Total files in folder1 with no matching name in folder2: 0
Total files in folder2 with no matching name in folder1: 0
```

- Models with no weights being shared:**

```
Total files with the same names: 158
Total raw files that are identical: 4
Total raw files that are different: 154
Total files in folder1 with no matching name in folder2: 0
Total files in folder2 with no matching name in folder1: 0
```

From the above output, we can observe that even if the raw files between the models are the same, the contents of the raw files are not the same which we identify through cosine similarity of the raw files. Only when the weights are similar considerable weight sharing among the models can be established.

- The weight sharing feature can also be verified by checking the ROM of the individual context binaries (Model-A.bin, Model-B.bin in Fig 4) and the single binary that was generated with weight sharing flag enabled(context_binary_combined.bin in Fig 4). It is very clear that the ROM of the single context_binary is much lesser than the multiple context_binaries combined.




 context_binary_combined.bin	03-01-2025 17:19	BIN File	1,61,715 KB
 Model-A.bin	03-01-2025 17:18	BIN File	1,08,844 KB
 Model-B.bin	03-01-2025 17:19	BIN File	1,51,916 KB

Fig 4: Generated context binaries with their respective ROM sizes

- Note that the sizes of the bin files of the models can vary largely even if their input layer has different resolutions even if they are trained from the same base model.

6 Limitations

- Only supports offline prepare on x86_Linux platform. Online prepare and other platforms (ARM/x86_Windows) offline prepare are not supported as of now.
- Feature enabled for NPU's with Hexagon v73 or later architecture.
- Any previously generated binaries will not automatically benefit from Weight Sharing. Users are required to regenerate new serialized binary to benefit from Weight Sharing.

7 Technical assistance

For model details on QNN weight sharing feature, please refer to the [Qualcomm® AI Engine Direct HTP documentation](#)

For assistance or clarification on information in this document, open a technical support case at <https://support.qualcomm.com/>.

You will need to register for a Qualcomm ID account and your company must have support enabled to access our Case system.

Other systems and support resources are listed on <https://qualcomm.com/support>.

If you need further assistance, you can send an email to qualcomm.support@qti.qualcomm.com.