AN12964

i.MX 8M Plus NPU Warmup Time

Rev. 1 — 10/2020 Application Note

Contents

1 Introduction

The i.MX 8M Plus family focuses on Machine Learning (ML) and vision, advanced multimedia, and industrial IoT with high reliability. It is built to meet the needs of Smart Home, Building, City and Industry 4.0 applications.

The i.MX 8M Plus is a powerful quad-core Arm® Cortex®-A53 applications processor running at up to 1.8 GHz with an integrated neural processing unit (NPU) delivering up to 2.3 TOPS. As the first i.MX processor with a machine learning accelerator, the i.MX 8M Plus processor delivers substantially higher performance for ML inference at the edge.

The NXP software development environment for machine learning is eIQ™. It enables the use of ML algorithms on NXP MCUs, i.MX RT crossover MCUs, and i.MX family SoCs. eIQ software includes inference engines, neural network compilers, and optimized libraries.

2 Purpose

When comparing NPU with CPU performance on the i.MX 8M Plus, the perception is that inference time is much longer on the NPU. This is due to the fact that the ML accelerator spends more time performing overall initialization steps. This initialization phase is known as **warmup** and is necessary only once at the beginning of the application. After this step inference is executed in a truly accelerated manner as expected for a dedicated NPU.

The purpose of this document is to clarify the impact of the warmup time on overall performance.

3 Overview

3.1 Software environment

The primary APIs supported by the NPU are OpenVX 1.2. Figure 1 presents the software stack for the two inference engines, eIQ TF Lite and Arm NN, which currently support NPU acceleration for i.MX 8M Plus.

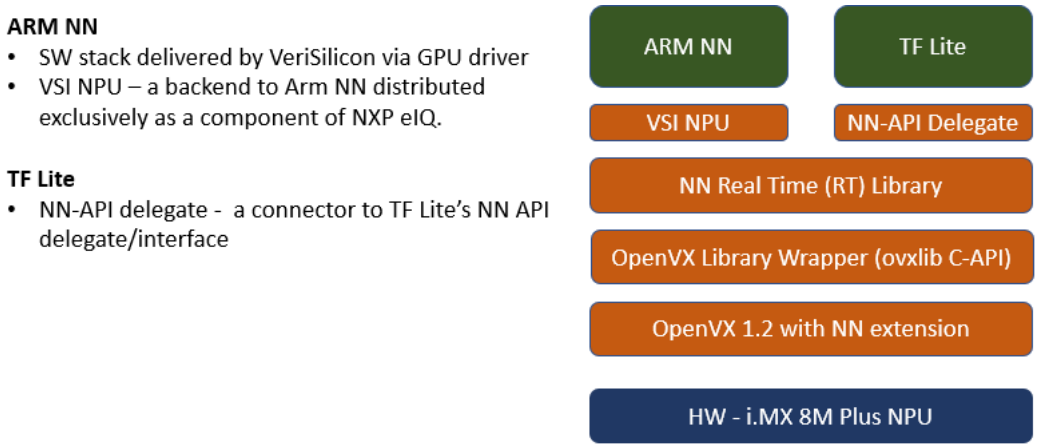

Figure 1. Arm NN and TF Lite software stack

NOTE

- NN RT: Common library which connects ovxlib.
- ovxlib: A wrapper around OpenVX driver to interface NN functionality.
- OpenVX driver: Khronos defined for acceleration in computer vision and NN functionality.

For the purpose of this application note, the following software environment was used:

- Yocto BSP release: i.MX 8M Plus Beta 1 release L5.4.24_2.1.0_MX8MP
  - For details of eIQ support build image for **imx-image-full**, check *i.MX Yocto Project User's Guide* (document IMXLXYOCTOUG).
  - This Yocto BSP release includes TF Lite 2.1.0. It supports hardware acceleration using Neural Networks API (NNAPI) Delegates.
- eIQ TF Lite applications (pre-installed for Yocto images containing eIQ)
  - TF Lite benchmarking application (*/usr/bin/tensorflow-lite-2.1.0/examples/benchmark_model*)
  - TF Lite image classification example (*/usr/bin/tensorflow-lite-2.1.0/examples/label_image*). This was used as a starting point and modified to demonstrate Warmup Time impact.

NOTE

For more details on the `benchmark_model` and `label_image` applications, refer to *i.MX Linux® User's Guide* (document IMXLUG).

3.2 Hardware overview

The following table lists the hardware features relevant for the use case described in this application note.

CPU	4 × Cortex-A53 1.8 GHz
DDR	16/32-bit LPDDR4/DDR4/DDR3L
AI/ML	NN Accel 2.3 TOPS
L2 cache	512 KB with ECC

NPU

SRAM (256KB) available for the neural network engine. Some of its features are:

- Provide Intelligent caching mechanism for kernel and input tensor.
- Pre-determine the best caching allocation.
- Guarantee no cache thrashing.
- Multiple layers of kernel can be stored at the same time.
- Store intermediate tensors.
- Intermediate tensors are often broken down into smaller tile to reduce memory footprint.

For more details, refer to *i.MX 8M PLUS APPLICATIONS PROCESSOR FAMILY* (document IMX8MPLUSFS).

3.3 CPU vs NPU performance for Mobilenet V1

Run the TF Lite benchmarking application for the default Mobilenet V1 model included in the Yocto image.

```
$: cd /usr/bin/tensorflow-lite-2.1.0/examples
```

Run the example on the **CPU**:

```
$ ./benchmark_model --graph=mobilenet_v1_1.0_224_quant.tflite
```

Run the example on the **NPU** (set the **use_nnapi** argument to enable acceleration):

```
$: ./benchmark_model --graph=mobilenet_v1_1.0_224_quant.tflite --use_nnapi=true
```

From the output printed in the console, the information below is relevant for performance:

CPU:

Average inference timings in us: Warmup: 156891, Init: 80087, no stats: 154844

NPU:

Average inference timings in us: Warmup: 7.44319e+06, Init: 29924, no stats: 3114.78

It is easy to notice that the ~7s warmup time for the NPU is far greater than the rest of the displayed time values. This is the reason why when running this example on the NPU it takes longer then on the CPU. However, inference itself runs much faster on the NPU then on the CPU even if at first sight this might not be obvious.

3.4 Benchmarking output

The below is a brief explanation of the time information displayed by the benchmarking application

- **Init**: Time spent for initialization steps: load the model, initialize flags/parameters, and perform initial validations/ adjustments for the tensors.
- **Warmup**: Time spent for the initial inference run(s). The benchmarking application has an argument that allows performing multiple warm-up runs, `warmup_runs`. In most cases one iteration is enough. Long warmup time is an expected behavior as the GPU/NPU driver needs to convert and transfer necessary data to the GPU/NPU memory and perform neural network optimizations. All of this is executed during the initial model inference. The purpose of warmup runs is to exclude the time of these operations from the final inference times, when everything is properly set up. The iterations following the graph initialization are performed many times faster.
- **No stats**: The average time spent to run inference, the warmup time excluded.

4 Warmup time – in-depth analysis

4.1 Warmup time in the case of multiple models running sequentially

Let's assume we created an application that runs inference on multiple models sequentially. We have Model A, Model B and Model C. The inference for each is run one step at a time, not necessarily in the same order.

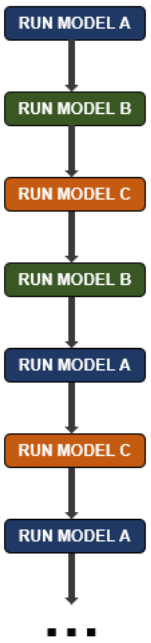

Figure 2. Models running sequentially

Assuming inference for all the models is executed from the same application. Each of the models have associated a TF Lite Interpreter instance. The Init and Warmup phases should be executed in the beginning. Then the appropriate TF Lite Interpreter will be used to run inference as needed on the corresponding model. To detail the previous sequence, the steps are shown in Figure 3.

Figure 3. Detailed steps for models running sequentially

To exemplify the impact on performance, we will run a similar sequence on three TF Lite quantized models:

- Mobilenet V1
- Mobilenet V2
- Inception V3

The time values are obtained when running on the NPU.

Table 1. Warmup time

	Warmup (ms)	Inference (ms)
Mobilenet V1	7896.6	3.36
Mobilenet V2	8846.58	3.457
Inception V3	42169	17.359

NOTE

Table 1 lists preliminary results, subject to change.

The average inference time remains the same no matter in what order the models are run.

If the models are loaded and used from the same application, the information will be kept, and warmup time only has an impact in the initialization phase. If the application exits and restarts, it will load the models again and go through the warmup phase.

NOTE

- Refer to C++ sample code for the C++ sample implementation.
- Refer to Python sample code for the python sample implementation.
  - The python TF Lite API currently allows running inference only on the NPU (CPU not supported).
  - With the python API, the Init time is not output separately. Only the warm-up and inference time are shown.
- The results obtained for the C++ and python applications are similar, there is no noticeable overhead in the python version.
- The implementation is based on TF Lite for the purpose of providing a simplified example. The described behavior is the same when using eIQ Arm NN.
- This application is focused on the i.MX 8M Plus NPU but note that the warmup time is specific to ML accelerators with hardware support for the OpenVX API. The warmup time will be significant also for i.MX8 platforms using the GPU as ML accelerator.

4.2 OVX graph caching

There is a way to improve warmup time for subsequent runs of the application by setting the following environment variables:

```
export VIV_VX_CACHE_BINARY_GRAPH_DIR=`pwd`
export VIV_VX_ENABLE_CACHE_GRAPH_BINARY="1"
```

By setting up these variables the result of the OpenVX graph compilation will be stored on disk, as the `network binary` graph files. For example, the files resulted after caching the models used as example in Warmup time in the case of multiple models running sequentially are:

The runtime will do a quick hash check on the network and if it matches the `*.nb` file hash it will load it into the NPU memory directly.

NOTE

The environment variables need to be set persistently, such as, available after reboot. Otherwise, the caching mechanism will be bypassed even if the `*.nb` files are available.

For the previous example, the following values are obtained for warmup times starting with the second run of the application:

Table 2. Warmup time

	Warmup time (ms)	
	NO OVX Graph Caching	**OVX Graph Caching ENABLED**
Mobilenet V1	7896.6	2192.34
Mobilenet V2	8846.58	1828.95
Inception V3	42169	8800.39

NOTE

Table 2 lists preliminary results, subject to change.

The decrease of the warm-up time is explained by the fact that the generated network binary graph requires no further compilation and the instructions will be issued to hardware with minimal preparation.

The main steps of the graph caching process are:

1. Load TF Lite file into CPU DDR, done by the TF Lite runtime.
2. The neural network runtime will build the network using the TF Lite model.

3. The neural network runtime will compile the network and store the NPU instructions in memory owned by the NPU.
4. The NPU instructions will be saved on disk in the format of a `*.nb` file.
5. Next time the same network is loaded in Step 2.
6. The neural network runtime will do a quick hash check on the network. If it matches the Hash, it will load the `*.nb` file into NPU memory directly.
7. Run inference on the NPU.

To analyze the impact on memory usage, the size of used CPU RAM, CPU Cache/Buffers and NPU memory was analyzed in different stages. For details, see Hardware overview. The peak value for memory usage is presented for reference. The same application is described in Warmup time in the case of multiple models running sequentially was used.

1. AT BOOT: Measurement was performed immediately after linux login, without executing anything else.
2. FIRST RUN [GRAPH NOT CACHED]: The first run of the application right after login.
3. FIRST RUN [GRAPH CACHED]: Graph caching was enabled.

   Set `VIV_VX_CACHE_BINARY_GRAPH_DIR` and `VIV_VX_ENABLE_CACHE_GRAPH_BINARY`. The variables were set persistently in order to remain set after a reboot. Then the application was run again.
4. SECOND RUN [GRAPH CACHED]: The second run of the application. The graph was previously cached.
5. AFTER APPLICATION EXIT

Figure 4 illustrates the memory usage at each stage.

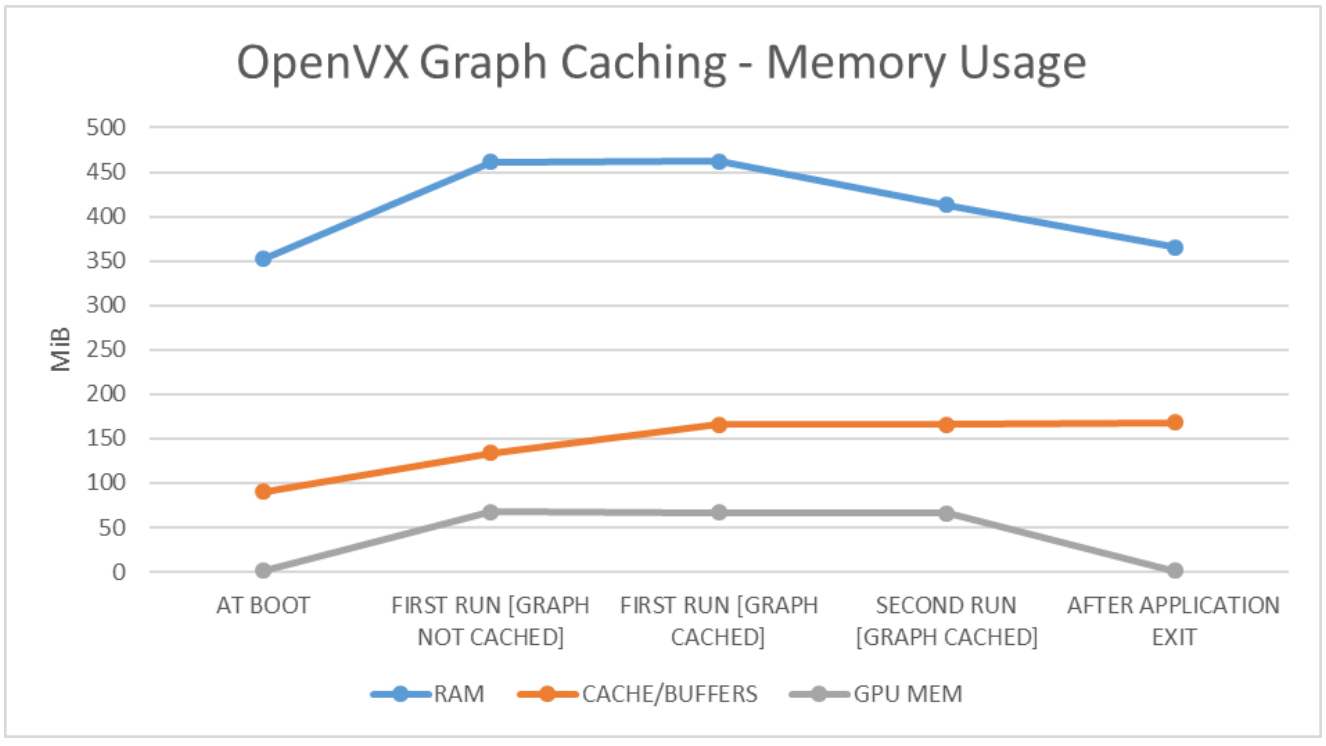

Figure 4. Memory usage

NOTE

Figure 4 lists preliminary results, subject to change.

The memory impact should to be taken into consideration when designing the application, to improve the warm-up time might have a draw-back on overall system performance. The impact should be analyzed for each case in order to decide if it will benefit the overall solution.

NOTE

- CPU RAM and Buffers/Cache usage was measured with the linux command, `free -h`.
- GPU memory usage was measured with the linux command, `gmem_info`.

5 Conclusion

When benchmarking an application, it is important to have a good understanding of the data in order to draw correct conclusions.

In the case of the i.MX 8M Plus NPU, the warmup time is considerably longer then the inference time. It has an impact only on the initialization phase of an application. The time should be measured separately for warmup and inference.

Warmup time will usually affect only the first inference run. However, depending on the ML model type it might be noticeable for the first few inference runs. Some preliminary tests must be done to take a decision on what to consider **warmup** time.

Once the warmup phase is well delimited, the subsequent inference runs can be considered as **pure** inference and used to compute an average for the inference phase.

Currently the warmup time can be decreased for subsequent application runs by using a graph caching mechanism. One should measure the impact of this feature on overall system performance, in order to decide is the application will benefit from using this mechanism.

There is work in progress to improve the initial warm-up time, speeding up the graph initialization without using the caching mechanism, so please check future releases for related updates.

6 Revision history

Table 3. Revision history

Revision number	Date	Substantive changes
0	08/2020	Initial release
1	10/2020	Updated OVX graph caching.

A C++ sample code

Use the source code attached to this application note.

Prerequisites for building the C++ app:

1. Yocto build environment for i.MX 8M Plus Beta 1 release, L5.4.24_2.1.0_MX8MP
2. Yocto SDK (a.k.a toolchain)
   - For the SDK to be able to build TF Lite apps, one needs to add the following to `local.conf`:

     ```
     TOOLCHAIN_TARGET_TASK += "tensorflow-lite-dev tensorflow-lite-staticdev"
     ```

   - Build the SDK:

     ```
     bitbake imx-image-full -c populate_sdk
     ```

- Install the SDK:

```
./${YOCTO_BUILD_ROOT}/tmp/deploy/sdk/ fsl-imx-xwayland-glibc-x86_64-imx-image-full-aarch64-imx8mpevk-toolchain-5.4-zeus.sh
```

To build the CPP app:

1. Activate the Yocto toolchain:

```
source ${YOCTO_SDK_LOCATION}/environment-setup-aarch64-poky-linux
```

2. Change directory to *${YOCTO_BUILD_ROOT}/tmp/work/aarch64-poky-linux/tensorflow-lite/2.1.0-r0/git/tensorflow/lite/examples/label_image/*.
3. Overwrite existing C++ `label_image` sources with the ones attached to this Application Note.
4. Build with the following command:

```
$CC -o lbl_img label_image.cc bitmap_helpers.cc ../../tools/evaluation/utils.cc -I=/usr/include/tensorflow/lite/tools/make/downloads/flatbuffers/include -I=/usr/include/tensorflow/lite/tools/make/downloads/absl -ltensorflow-lite -lstdc++ -lpthread -lm -ldl -lrt
```

Deploy the `lbl_image` executable on the board in */usr/bin/tensorflow-lite-2.1.0/examples*.

Deploy the input models in the same location, just the `*.tflite` files, */usr/bin/tensorflow-lite-2.1.0/examples/*:

- Mobilenet V1
- Mobilenet V2
- Inception V3

When running, make sure to set warm-up runs to 1 in order not to break the average value. As the first run seems to take considerably more time, add the `-w 1` option.

- For CPU:

```
./lbl_img -i grace_hopper.bmp -l labels.txt -w 1
```

- For NPU:

```
./lbl_img -i grace_hopper.bmp -l labels.txt -w 1 -a 1
```

B Python sample code

Use the source code attached to this Application Note.

- Deploy `label_image.py` on the board to overwrite existing script, */usr/bin/tensorflow-lite-2.1.0/examples/label_image.py*.
- Deploy the input models in the same location, just the `*.tflite` files, */usr/bin/tensorflow-lite-2.1.0/examples/*:
  - Mobilenet V1
  - Mobilenet V2
  - Inception V3
- Run script: *python3 label_image.py*

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 10/2020

Document identifier: AN12964

arm