
SW support for NPU accelerators for Linux-type operating systems

Bachelor's Thesis by Michal Žáček

Study Programme: Systems and Virtualization

Czech Technical University in Prague

,

Faculty of Information Technology

Prague, 2025-05-26

supervised by
Petr Zemánek

0.1 English

0.1.1 Abstract

This document covers information about the NPU chip included in the NXP i.MX 8MP Evaluation Kit (EVK), how it is used, what the different frontends to it have in common or differ in. We then perform some rudimentary benchmarks in order to determine the practical benefit of utilising the NPU for a common workload, before ending on a description of the components required for PikeOS to claim NPU support, and what porting them could entail.

0.1.2 Keywords

hardware acceleration, embedded, machine learning hardware, linux, platform porting, TQ, NXP i.MX 8MP

0.2 Czech

0.2.1 Abstract

Tento dokument zahrnuje informace o NPU čipu, kterým je vybavena deska NXP i.MX 8MP EVK, jak se používá, jaké k němu existují přístupové cesty, v čem se liší či co mají společného. Provedeme několik základních benchmarků abychom zjistili jaký je skutečný přínos čipu pro některé běžné úkony. Nakonec shrneme co je potřeba aby PikeOS podporoval, pro oficiální podporu tohoto NPU čipu a probereme, co portování těchto knihoven může zahrnovat.

0.2.2 Keywords

hardwarová akcelerace, embed, hardware na strojové učení, linux, porting mezi platformami, TQ, NXP i.MX 8MP

Contents

0.1 English	2
0.1.1 Abstract	2
0.1.2 Keywords	2
0.2 Czech	2
0.2.1 Abstract	2
0.2.2 Keywords	2
1 Declaration	7
2 Introduction	10
3 NPU's (Neural Processing Units)	11
4 File Formats	13
4.1 Keras / HDF5	13
4.2 Pickle	13
4.3 Block map .bmap	14
4.4 System Package Data Exchange .spdx	14
4.5 OpenEmbedded Image Creator .wic	14
4.6 OpenEmbedded kickstart file .wks	14
4.7 Flattened Device Tree .fdt / Device Tree Source .dts	14
5 OpenVX™ & TIM-VX	15
6 Setting up the environment	17
6.1 Kernel Tests	18
6.2 ONNXRuntime	18
6.3 TIM-VX	19
6.4 OpenCV	19
7 The Graph Workflow	22
7.1 Python Subclasses modelling Tensor functions	22
7.1.1 forward (mandatory)	24
7.1.2 __init__	24
7.1.3 Movement functions	24
7.2 Exporting Models to files	24
7.3 Keras	25
7.4 Other Tools	25
7.4.1 Visualization: Netron & Zetane	25
7.4.2 Conversion: Tensor2onnx & Tflite2onnx	26
8 Hardware Specifics	27
8.1 NPU Contents	27
8.1.1 Parallel Processing Unit (PPU)	27
8.1.2 Neural Network Engine	27
8.1.3 Tensor Processor	27
8.2 Configuration Environment Variables read by Tensor Interface Module for OpenVX (TIM-VX)	27
8.3 Power Modes	28

9 Frameworks	29
9.1 LiteRT (Lite RunTime)	29
9.2 Tensor Virtual Machine (TVM)	30
9.3 Open Neural Network Exchange (ONNX)	30
9.4 Unaddressed frameworks	31
9.4.1 The NXP eIQ environment	31
9.4.2 Android NNAPI	31
9.4.3 Paddle Paddle	32
10 Performance	33
10.0.1 Startup	33
10.0.2 Quantization	33
10.0.3 Dynamic loading of Libraries	34
10.0.4 Bus	34
10.0.5 NPU Clogging	34
10.1 Benchmarking Practical Results	34
11 Suggesting PikeOS integration	36
12 Conclusion	37
13 Build configuration	38
14 Glossary	39
Bibliography	40

List of Figures

Figure 1 The NXP i.MX 8MP EVK we worked on	11
Figure 2 A diagram showing TIM-VX's role in running models on NPUs (1)	16
Figure 3 Example of a Netron graph render	25

List of Tables

Table 1	Tensor Processor supported operations	27
Table 2	Configuration Environment Variables read by TIM-VX	28
Table 3	Power Modes	28
Table 4	mobilenet_v1_1.0_224_quant Performance Metrics	34
Table 5	MobileNetV3Large Quantization tests	34
Table 6	Various Models performance	35

Chapter 1

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I declare that I have not used any AI tools during the preparation and writing of the thesis. I am aware of the consequences of apparently unacknowledged use of these tools in the production of any part of my thesis. Specifically the sole use was for occasional primary source reconnaissance, akin to a search engine, before processing and utilising the primary sources on their own merit exclusively using my own eyes and words.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No.,121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 16, 2025:



Assignment of bachelor's thesis

Title: SW support for NPU accelerators for Linux-type operating systems
Student: Michal Žáček
Supervisor: doc. RNDr. Ing. Petr Zemánek, CSc.
Study program: Informatics
Branch / specialization: Computer Systems and Virtualization 2021
Department: Department of Computer Systems
Validity: until the end of summer semester 2026/2027

Instructions

NPU (Neural Processing Unit) accelerators in PCs or notebooks are becoming more and more important these days. The target platform considered in this thesis is the board TQMa8MPxL [1] but results should apply to any device with the NPU accelerator running under Linux and/or PikeOS. Neither the Linux kernel nor PikeOS (real-time embedded OS) currently support AI/ML acceleration using NPUs. The objective of the thesis is to conduct research of open source frameworks with the aim to integrate a reasonably good SW support for utilizing this accelerator to enhance the performance of machine learning (ML) algorithms on these target platforms.

- 1/ Analyze and evaluate implementation of the SW support for utilizing NPU accelerators in representative open-source frameworks.
- 2/ Investigate modules or libraries used in these frameworks for communication with NPU accelerators.
- 3/ Review ML algorithms implemented in these frameworks and select most appropriate frameworks for testing.
- 4/ Design benchmark data sets for testing the ML performance of the selected frameworks on the given Linux platform.
- 5/ Test the performance of selected frameworks on the given Linux platform.
- 6/ Based on these findings, select the most appropriate framework for the integration into the Linux platform and elaborate an integration procedure based on e.g. Yocto packaging system.



7/ Document problems encountered during the integration process.

8/ Propose a procedure for the integration of the SW support for the NPU accelerator on the PikeOS platform.

[1] <https://www.tq-group.com/en/products/tq-embedded/arm-architecture/tqma8mpxl>



Chapter 2

Introduction

Our computing devices have come a long way in terms of speed, and of course in terms of operating complexity. This complexity can hold back very general purpose processing units in performance when it comes to highly specialized tasks. For a long time, GPUs were used where a large number of specialized identical operations needed to be performed on a vast array of differing data inputs, as their specialized design for this use case made them incomparably faster than using a common general-purpose CPU. Though most often associated with their namesake of graphics, they excel in many applications and have become a valuable tool when working with neural networks.

Recently, even more specialized hardware has become available to meet the rising demand to have access to these technologies in more portable devices or for use in embedded situations. Neural Processing Units¹, are becoming more common, however these don't often come with universal drivers and the most widespread standard today has no freely adoptable implementation.

Our goals in this thesis are to initially lay out the defined and known terms used in the context of neural nets, such as hardware, alternate names of known concepts, or file formats. Following this we shall go through the existing solutions implementing both the communication with actually hardware in the form of modules followed by frontend APIs for NPU work. We will then go through the advantages that this hardware should in theory provide followed by a practical test of the actual results on real hardware. The code will be made available so that it may be run to test and/or verify these results. Finally, we discuss what this means for PikeOS integration of NPU support.

As most of our sources are proprietary documentation, this work will sadly be light on detailed images and diagrams available in the primary sources.

¹Historically also called Versatile/Tensor/Intelligence Processing Units (VPU/TPU/IPU), however for simplicity's sake we will use the term NPU and only deviate if a specific function or product uses one of the alternatives

Chapter 3

NPU (Neural Processing Units)

Neural Processing Units (further shortened to NPU) are in general a type of hardware accelerator which is optimized to efficiently handle Machine Learning workloads. They offer both faster inference and often a lower energy footprint on edge devices (2). Our NPU in question is composed of a frontend that is used for communication between the NPU and the rest of the board, a parallel processing unit, a neural network engine along with a backing storage. The device works with 4-element vectors of 16/32-bit IEEE floating-point or 8/16/32-bit integers either signed or unsigned. The device implements the OpenVX™ hardware API along with some extensions and VeriSilicon provides two separate official libraries through which to call this API. The open-source TIM-VX (1) which contains C++ bindings, and Ovxlib which contains C bindings. TIM-VX calls into one of two SDKs, the VeriSilicon Unified OpenVX™ SDK supporting both compiling and running using either the GPU or NPU units, and the (as far as I can tell proprietary, since the traces of this SDK online are close to nonexistent) VIP-Lite SDK that only contains the runtime for the NPU. TIM-VX is used as the target for third-party frameworks targeting the board's NPU. Of these we used specifically the former VeriSilicon SDK also often listed as Vivante SDK or directly `imx-gpu-viv`² in recipes, as this was the one used by all of the NXP recipes.

The SDK includes the libraries listed in Listing 1. Of these only the ones marked with * are linked against by OpenVX™.

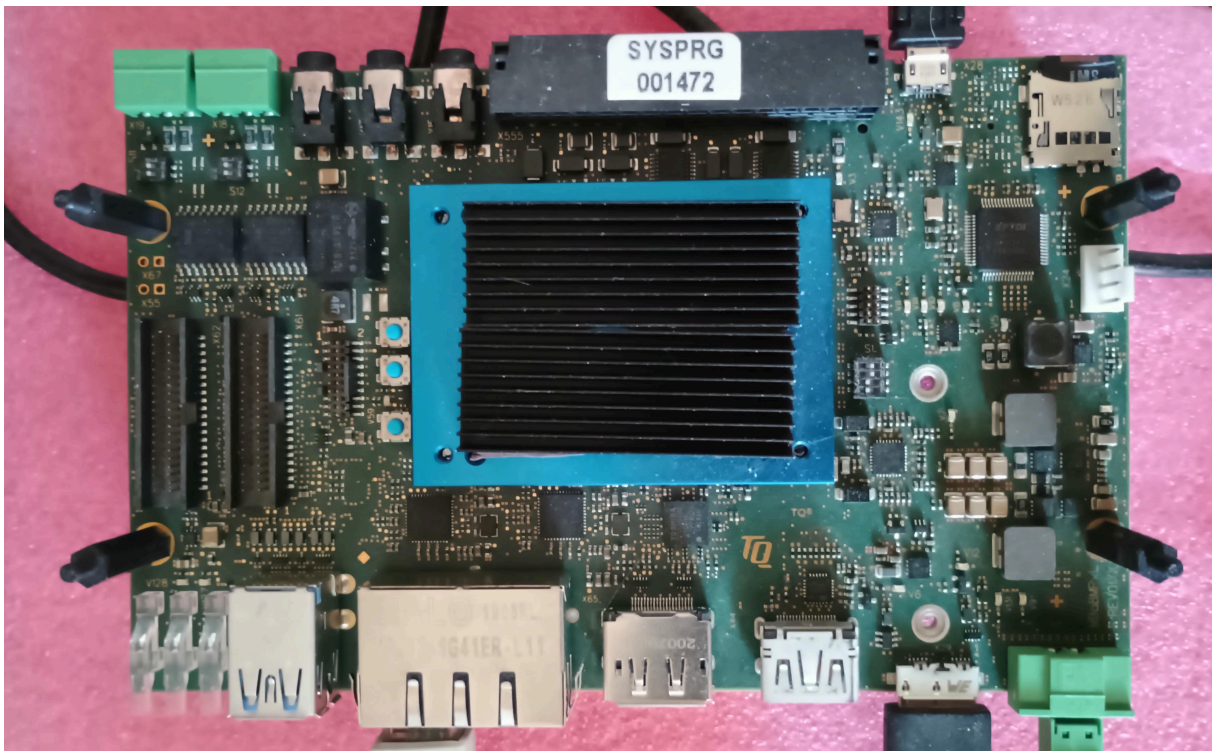


Figure 1: The NXP i.MX 8MP EVK we worked on

²The name is GPU, however it contains shared drivers for both NPU and GPU.

3 NPU's (Neural Processing Units)

- libArchModelSw.so*
- libCLC.so
- libGAL.so*
- libNNArchPerf.so*
- libOpenVX.so*
- libOpenVXC.so
- libOpenVXU.so
- libVSC.so*

Listing 1: List of SDK libraries

Chapter 4

File Formats

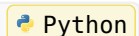
First, we present a rundown of the various different formats encountered throughout this work, commonly used in the machine learning space and during yocto builds and flashes.

4.1 Keras / HDF5

Keras is a library released in 2010 as part of a research effort, originally at Alphabet (3). Its relation to TensorFlow will be apparent later. Keras' currently used filetype `.keras` is a wrapper format consisting of a ZIP-archive containing an `.h5` file holding the layers and weights of a model, and some json metadata about Keras' configuration, the creation date, version information, etc.

HDF stands for Hierarchical Data Format; it contains a POSIX-filesystem style hierarchy of groups akin to directories, even supporting hard and soft links. Datasets take the role of files and contain tensors with associated shapes and dtypes and can be read and written to, treating them as numpy arrays (4),(5).

```
1  import h5py
2
3  data = h5py.File("model.weights.h5")
4
5  # All valid
6  data['/layers/flatten/vars']
7  data['layers/flatten/vars']
8  subdata = data['layers']['flatten']
9  subdata['vars']
10
11 data['/layers/dense/vars/0']
12 #=> <HDF5 dataset "0": shape (784, 128), type "<f4">
13
14 data['/layers/dense/vars/0'][0]
15 #=> array([ 0.01777279,  0.07076738,
16 #          -0.05744237,  0.06602553,
17 #          ...])
```



Listing 2: HDF5 structure access

As compression is done by Keras in-memory saving and loading are transparent to whether one is working with a ZIP archive or a directory of files.

4.2 Pickle

This is dangerous to use.

Some formats, especially those that allow a wide array of Python datatypes, may allow unwanted code to execute as well when loaded. As a representative of these, we mention Pickle. Since pickling allows arbitrary instances of classes, malicious code may be inserted as well into any model downloaded.

An example taken from (6) can be seen in Listing 3.

```

1  import pickle
2  import os
3
4  class RCE:
5      def __reduce__(self):
6          cmd = ('echo GET HACKED')
7          return os.system, (cmd,)
8
9
10 with open("tensor.bin", "wb") as f:
11     pickle.dump(RCE(),f)

```

Listing 3: Class Exploiting Pickle deserialization

4.3 Block map **.bmap**

Relates to Yocto project's (formerly Intel's) bmaptool from (7), which is used to flash data similarly to dd. Unlike dd, this tool verifies the integrity of flashed data, supports more complex arrangements such as sourcing the image from a remote server, supports sparse definitions, and includes protections from accidentally destroying data on disks that seem like regular mounted block devices.

4.4 System Package Data Exchange **.spdx**

A Bill of Materials of all the included packages used to build an image including all the versions and metadata (8).

4.5 OpenEmbedded Image Creator **.wic**

Should your device require multiple partitions (9). Use if bmap is not supported as it does not support sparseness, etc.

4.6 OpenEmbedded kickstart file **.wks**

Contains build commands for the wic command (9).

4.7 Flattened Device Tree **.fdt** / Device Tree Source **.dts**

Device tree source describes hardware (10), and is subsequently compiled into **.dtb**. Determines how some things are flashed and written based on the target.

Chapter 5

OpenVX™ & TIM-VX

OpenVX™ is a standard hardware API that can be implemented by hardware vendors of hardware accelerators and exposed as a C API to users (11). The Khronos group that manages the OpenVX™ specification only describes an abstract machine and certain operators with defined semantics. The implementation of the operators in terms of hardware is completely up to the vendor, and the specification aims to be written in a way as to allow as much optimization as possible. When we spoke so far of `libOpenVX.so`, that is VeriSilicon's pre-compiled implementation of the OpenVX™ API. OpenVX™ officially aims at vision processing specifically, however there are Neural Network extensions to allow using the same pipeline to also accelerate machine learning operations utilizing the tensor structure from the full OpenVX™ v1.2 Spec, and extending it with new operators such as `vxActivationLayer`, `vxConvolutionLayer`, `vxFullyConnectedLayer`, `vxSoftmaxLayer` and more (12).

Known Implementors of this API are:

VeriSilicon The `libOpenVX.so` object all our libraries link against

KhronosGroup/OpenVX-sample-impl Only truly open-source implementation, however, it is supposedly very ad-hoc, slowly implemented, and emulates the given operators on CPU or related using OpenCL (13).

TexasInstruments/tiovx Source available but only authorized for use on TI hardware.

AMD MIVisionX Part of the ROCm environment, it implements OpenVX™ over AMD's GPUs and CPUs (14).

Intel OpenVINO

Nvidia VisionWorks

However, only the relevant vendor makes sense to use as it is tightly coupled to the given device that is to be controlled with it. The `x86_64` OpenVX™ version available from the TIM-VX repository only emulates the behaviour of the API using the machine's CPU.

OpenVX™ still uses a graph abstraction; however, the operations are far too low-level to be convenient and useful for the purposes of general-purpose machine learning jobs. Especially since they do not interoperate with existing well-known formats from the public machine learning ecosystem. Which is why we opted to rather focus on relevant and convenient libraries that wrap OpenVX™ instead of calling it directly.

Further information about what the library handles is given in the chapter on Startup (see Section 10.0.1)).

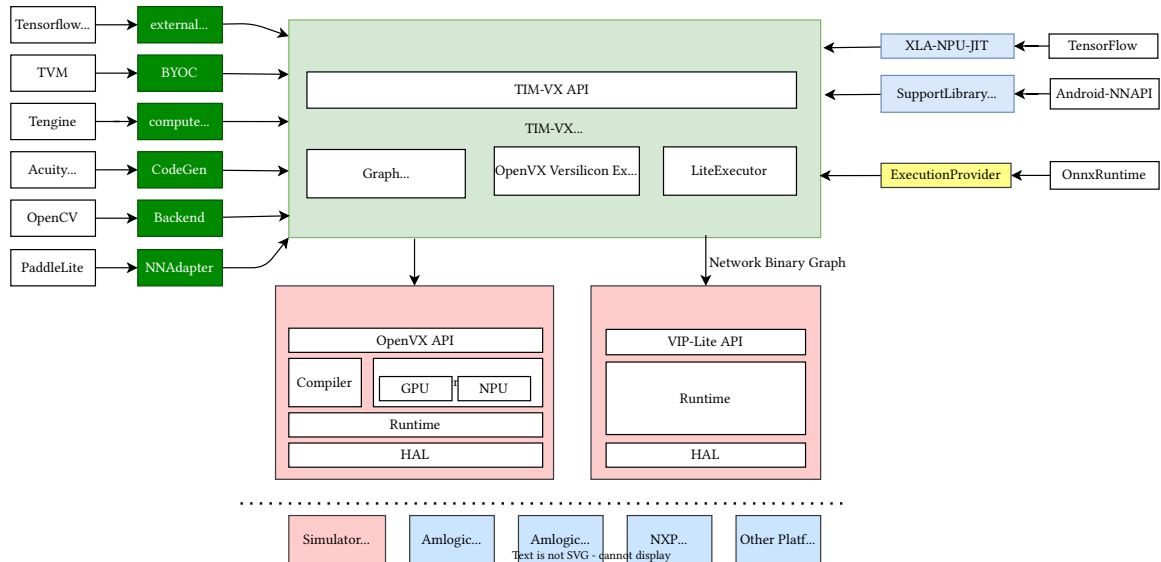


Figure 2: A diagram showing TIM-VX's role in running models on NPUs (1)

TIM-VX is a C++ wrapper library meant for external software vendors to link against instead of OpenVX™ for a more high-level and universal API (1). As can be seen in Figure 2) TIM-VX is linked against by all our frameworks and many more, including the Android soon to be deprecated NNAPI. Some of these are officially supported by the frameworks themselves, and some have official NXP forks or third-party forks. Beyond the higher-abstractions specifically tailored to machine learning, it also includes many utilities for debugging. The library additionally supports boards other than our own.

Chapter 6

Setting up the environment

Our testing environment consists of a Yocto distribution, running on our aforementioned chip.

meta-overlay will further be used to refer to our custom layer for the purposes of this thesis.

We use both the BSP and related layers directly from the TQ website, followed by the NXP meta-imx layer, cloned via the repo (15) utility from their manifest repository (16).

Repo is a utility developed by Alphabet to work with multiple, completely separate as far as Git is concerned, repositories and treat them as Git would treat submodules. The repositories are initialized from an eXtensible Markup Language (XML) file hosted in an arbitrary repository, which contains a set of remotes and “projects” (repositories) that will be fetched into the final checkout.

```
1 repo init \\  
2 -u https://github.com/nxp-imx/imx-manifest.git \\  
3 -b imx-linux-scarthgap \\  
4 -m imx-6.6.52-2.2.0.xml
```

Shell

Listing 4: IMX repo initialization

Afterwards a bit of guesswork was required to get all the compatible versions of all the layers and packages. A combination of some layers from the original BSP and the NXP-IMX manifest was used in the end. Mostly due to mismatches in gstreamer versions as meta-imx requires a version not below 1.24.0, but the bsp provides 1.22.5. The result of this is found in Chapter 13.

Following this we configure a build directory with the Machine: tqma8mpxl-mba8mpx and Distro: fsl-imx-wayland.

```
1 source setup-environment tqma8mpxl_build
```

Shell

Next we need an environment with all the programs that bitbake expects and needs to function. In our case we opted for an Ubuntu:22.04 Docker container, with some extra packages installed as determined by consulting (17) initialized by Listing 5.

```
1 FROM ubuntu:22.04  
2 RUN apt -o APT::Sandbox::User=root update  
3 RUN DEBIAN_FRONTEND=noninteractive TZ=Etc/UTC \\  
4 apt -o APT::Sandbox::User=root \\  
5 install -y gawk wget git diffstat unzip texinfo gcc \\  
6 build-essential chrpath socat cpio python3 python3-pip \\  
7 python3-pexpect xz-utils debianutils iputils-ping \\  
8 python3-git python3-jinja2 python3-subunit zstd \\  
9 liblz4-tool file locales libacl1  
10 RUN locale-gen en_US.UTF-8
```

Dockerfile

Listing 5: Build Image Dockerfile

After building we may enter the environment with Listing 6.

```
1 docker run --rm -it \\  
2 -v .../scarthgap.TQ.ARM.BSP.0001:/src \\  
3 --userns=keep-id bitbake-env
```

Shell

Listing 6: Entering Build Image

And we may now build the project:

```
1 bitbake imx-image-full
```

BitBake

The output will be available under:

```
1 ./tmp/work/imx8mpcvk-poky-linux/imx-image-full/...  
2 .../1.0/deploy-imx-image-full-image-complete
```

This may take many hours. Once successful, flashing is done with the command in Listing 7.

```
1 sudo uuu -v -b sd_all \\  
2 ./images/imx-boot-tqma8mpxl-mba8mpxl-mfgtool.bin-flash_spl_uboot \\  
3 ./images/imx-image-full-tqma8mpxl-mba8mpxl.rootfs.wic
```

Shell

Listing 7: Flashing image to device

6.1 Kernel Tests

The VSOCK Makefile has no install goal, yet the recipe tries to call it, so we just disable that part of the recipe with the following:

```
1 PACKAGECONFIG:kernel-tools = ""
```

BitBake

6.2 ONNXRuntime

We change the original ONNXRuntime recipe's source from NXP to the official repo as it now supports our NPU through the use of the TIM-VX library (Listing 8).

```
1 ONNXRUNTIME_SRC ?= "git://github.com/microsoft/onnxruntime.git"  
2 SRC_URI = "${ONNXRUNTIME_SRC};nobranch=1;protocol=https  
3 # Rel-1.21.0  
4 SRCREV = "e0b66cad282043d4377cea5269083f17771b6dfc"
```

BitBake

Listing 8: ONNXRuntime Build recipe src patch

ONNXRuntime's configuration script can't by default find our installation of TIM-VX, so we must assist it a bit, in addition to adding it to (R)DEPENDS (Listing 9).

```
1 DEPENDS = "libpng zlib tim-vx tvm"  
2 RDEPENDS:${PN} = "tim-vx tvm"  
3  
4 do_configure:prepend () {  
5     export TIM_VX_INSTALL="/usr"  
6 }
```

BitBake

Listing 9: ONNXRuntime Build recipe dependency patch

Next we must also actually enable the use of this library by adding the associated configuration flag (Listing 10).

```
1 EXTRA_OECMAKE += "\\
2     -Donnxruntime_USE_VSiNPU=ON \\
3     -Donnxruntime_USE_TVM=ON \\
4 "
```

Listing 10: ONNXRuntime Build recipe configure flags patch

As of writing the main 1.21.0 version of ONNXRuntime is the first to support VeriSilicon™ NPU (VsiNPU), however the release version had broken support for targets with no fp16 support, therefore a pr's commit with the fix needs to be used.

GitHub - Issue: 23957, PR: 23978

It is very fast on track to be merged into main, but now we apply these few commits with a patch.

```
1 SRC_URI:append = " file://fajin-corp_gh_pr_23978.patch"
```

Another patch is required though as the function

`VSiNPUExecutionprovider::GetCapability`

in the file `vsipu_execution_provider.cc` calls a logger, yet omits to declare one so we must add it manually.

```
1 const auto& logger = *GetLogger();
```

The call to `save_build_and_package_info` in `setup.py`, causes an error. It is only a buildinfo log, so simply removing it creates a warning when importing the module, however without hindering further use.

The patch is included with the work.

```
1 SRC_URI:append = " file://fix_logger.patch"
```

6.3 TIM-VX

This was simply needed to be updated to version 1.2.22, from the official VeriSilicon repo Listing 11.

```
1 SRC_URI = "${TIM_VX_SRC};nobranch=1"
2 TIM_VX_SRC ?= "git://github.com/VeriSilicon/TIM-VX.git;protocol=https"
3 SRCREV = "8494275d7608942aa584c9c13bd5e2d77be9906c"
```

Listing 11: TIM-VX's new version bb recipe

6.4 OpenCV

The build recipe for OpenCV is patchable, we add TIM-VX to dependencies, enable it in OpenCV and again point the configuration script at our installation directory with the file `opencv_\\%.bbappend` containing Listing 12.

```

1 EXTRA_OECMAKE:append = "\\
2     -D BUILD_opencv_gapi=OFF \\
3     -D WITH_TIMVX=ON \\
4     -D TIMVX_INSTALL_DIR=/usr/lib \\
5
6
7 DEPENDS:append = " tim-vx"
8 RDEPENDS:${PN}:append = " tim-vx"
9
10 INSANE_SKIP:${PN}-dbg += "libdir file-rdeps"
11 INSANE_SKIP:${PN} += "buildpaths"

```

Listing 12: OpenCV recipe to build with external TIM-VX

Compiling with external TIM-VX lib like this, which is strongly advised against (18), yields a segfaulting library which can be seen in .

```

1 >>> cv.setUseOpenVX(True)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 cv2.error: OpenCV(4.10.0) .../src/ovx.cpp:101:
5   error: (-215:Assertion failed)
6     !flag && "OpenVX support isn't enabled at compile time"
7     in function 'setUseOpenVX'
8
9 >>> cv.useOpenVX()
10 False
11
12 # Segfaults
13 self._model = cv.FaceDetectorYN.create(...)

```

Listing 13: Trying to enable OpenVX in OpenCV with external library

Trying to compile TIM-VX directly in OpenCV requires us to add Fortran to our toolchain due to a new dependence on lapack lest we encounter error Listing 14. We also must re-enable OpenCV downloads since the default recipe tries to prevent random downloads during configuration, by caching the downloads themselves. This can be seen in our new recipe Listing 15.

```

1 libgfortran was skipped:
2   libgfortran needs fortran support to be enabled in the compiler

```

Listing 14: Fortran missing from Toolchain error

```

1 EXTRA_OECMAKE:append = "\\
2     -D BUILD_opencv_gapi=OFF \\
3     -D WITH_TIMVX=ON \\
4     -D OPENCV_ALLOW_DOWNLOADS=ON \\
5 "
6
7 DEPENDS:append = " tim-vx lapack"
8 RDEPENDS:${PN}:append = " tim-vx lapack"

```

BitBake

Listing 15: OpenCV recipe to build with internal TIM-VX

Enabling Fortran breaks the `tpm2-tss-engine` and `isp-imx-dev` packages that is requires by `packagegroup-imx-ml` so we try disabling what is wrong:

```

1 RDEPENDS:packagegroup-imx-ml:remove = " tpm2-tss-engine"
2 INSANE_SKIP:isp-imx-dev += "dev-elf"

```

BitBake

This compiles an image, however OpenCV still throws an error when calling `cv.setUserOpenVX(True)` claiming it is not compiled with the given support and when run anyways with snippet Listing 16, the speeds are identical for both set backends, however the speed seems suspiciously high (in magnitude of hundreds of microseconds).

```

1 import cv2 as cv
2 m = cv.dnn.readNet('model.onnx')
3 m.setPreferable
4 m.setPreferableBackend(cv.dnn.DNN_BACKEND_TIMVX)
5 m.setPreferableTarget(cv.dnn.DNN_TARGET_NPU)
6 import numpy as np
7 m.setInput(np.full(fill_value=[1.0], shape=(1,28,28)))
8 m.forward()

```

Python

Listing 16: Attempt to run OpenCV DNN on NPU

Chapter 7

The Graph Workflow

Let us now describe how graphs are commonly constructed.

7.1 Python Subclasses modelling Tensor functions

All frameworks have Python APIs, and all seem to use a similar form of abstraction. They declare some class in the case of both PyTorch and Open Neural Network Exchange (ONNX) this could be `torch.nn.Module` (19). A model is then defined as a class that inherits from this root, and overrides one or more specific methods.

Once we instantiate such a class we receive an object that takes tensors and returns results. They can be freely composed into graphs of arbitrarily complex modules containing modules which all get compiled to a single graph.

For efficiency's and compatibility's sake these functions are later compiled into other forms especially when serialized into the form of any model file.

In the case of ONNX specifically we may take the example code Listing 17.

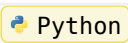
```
1 class OnnxModule(nn.Module):
2     def __init__(self):
3         super().__init__()
4
5     def forward(self, x):
6         return x + 3
```



Listing 17: Python code to be converted to ONNX Intermediate Representation (IR)

Which may be seen by the ONNX exporter³ as Listing 18.

```
1 class GraphModule(torch.nn.Module):
2     def forward(self, x: "f32[100, 128]"):
3         # File: ./onnx.py:48 in forward, code: return x + 3
4         scalar_tensor_default:
5             "f32[]" = torch.ops.aten
6                             .scalar_tensor
7                             .default(3, dtype = torch.float32)
8         add:
9             "f32[100, 128]" = torch.ops.aten
10                                .add.Tensor(x, scalar_tensor_default)
11         return (add,)
```



Listing 18: ONNX Pseudocode

Before being compiled into a low-level representation as shown in Listing 19.

³Shortened by hand for readability

```

1  graph(
2      name=main_graph,
3      inputs=(
4          %"x"<FLOAT,[100,128]>
5      ),
6      outputs=(
7          %"add"<FLOAT,[100,128]>
8      ),
9  ) {
10     0 | # node_Constant_0
11         %"val_0"<?,?> <- ::Constant() {
12             % value=Tensor<INT64,[]>(array(3), name=None)
13             %}
14     1 | # node_Cast_1
15         %"scalar_tensor_default"<FLOAT,[]> <- ::Cast(%"val_0") {
16             % to=FLOAT
17             %}
18     2 | # node_Add_2
19         %"add"<FLOAT,[100,128]> <- ::Add(%"x", %"scalar_tensor_default")
20     return %"add"<FLOAT,[100,128]>
21 }

```

Listing 19: ONNX IR

How this is done from arbitrary Python code is by utilising a FakeTensor class, on which the algorithm actually runs, however instead of the operations happening they are recorded in the background and the result is again always a FakeTensor. After the algorithm finishes, the libraries inspect what operations were performed on what tensors and constructs an equivalent program through some backend, TorchScript, Dynamo, etc. This approach allows quite arbitrary Python programs to be compiled however at the loss of semantics. For example the classic `map(lambda x: x + 2, list)` would just be compiled into a series of completely unrelated additions on some projected content of the tensor.

These models also always have a static type of input and outputs that may be unconstrained by the Python class which implements it but must be specified explicitly when exporting. The part of the type that determines the size of a tensor is usually called their “shape” in this context. We set our input shape to (100, 128) in this example so the exported graph has annotations like `f32[100, 128]`.

Oftentimes a model is made to accept a large “batch” of inputs to parallelize inference. This is implemented as simply setting the first dimension as dynamic, resulting in shapes such as `(?, 100, 128)`. Models must be explicitly resized before using this dynamic size as it will default to 1 otherwise.

```

1  shape = (BATCH_SIZE, *shape[1:])
2  model.resize_tensor_input(0, shape)
3  model.allocate_tensors()
4  model.set_tensor(0, input_data)

```

Listing 20: Setting input tensors with batches

```

1  [ sum(n) for n in x ]

```


Listing 21: List comprehension in Python

The subset of python supported is quite large, even code such as Listing 21, will successfully compile into a graph, the efficiency of which depends heavily on the type of code, for example the above generates a massive array of round-robin addition nodes. Of course calling external functions is also supported and these are inlined and compiled as if written directly inside forward.

If one wishes to have control over what exactly their code compiles to, it is better to directly use modules from the given library.

These can be initialized under `__init__` as instance variables and then utilized in the forward function like Listing 22.

```
1 def __init__(self):
2     super(..., self).__init__()
3     self.relu = nn.ReLU()
4
5 def forward(self, x):
6     x = self.relu(x)
7     return x
```

 Python

Listing 22: Python model module methods

It is also useful to note that this code defines string names for the inputs and outputs of a graph, these are then used to retrieve many outputs or set inputs in a human-readable way.

7.1.1 forward (mandatory)

This is the crux of the pipeline, it defines the actual operations performed on the inputs and returns what the Graph node would have as outputs.

If in need of multiple inputs and outputs the function is allowed to take more than 1 argument, and they will be matched according to name. If on the other hand I want to return multiple outputs we can wrap the return in a dictionary with keys being the names of our outputs and values being the given output tensors.

7.1.2 __init__

Some frameworks, namely ONNXRuntime want their modules to be attributes of the given object, and so we usually then initialise our submodules here and refer to them in forward. For example including `self.conv1 = ...` inside the constructor and then calling as `x = self.relu(x)` in our forward function.

7.1.3 Movement functions

Methods such as `.to().cpu().ipu()`, cause our model to be run on the given target device. However, this may also hint to the given framework what the target is which is also taken into account in cases of optimization, our chosen ones do not do this and use other ways to specify both where and how to run.

7.2 Exporting Models to files

This step is extremely important as it can make or break the model's performance. First and foremost, TensorFlow has the function `tf.saved_model.save()` which simply dumps a bunch of files into a directory that represent the model. This isn't as interesting as the `tf.lite.TFLiteConverter` module. The converter object is constructed from one of a keras model, a `saved_model` path generated by the above save function, or a concrete function. After creation we can set a vast number of options such as `.optimizations`, `.inference_input_type` or `.target_spec.supported_ops`. Most importantly we

should set optimizations to the value `[tf.lite.Optimize.DEFAULT]`, as without this the model won't quantize, nor perform very well. Once this is done setting inference in/out-put types is optional and useful for quantization. It must be stated however that mixing types in the graph leads to something called hybrid data types, which are not supported by the NPU and LiteRT will simply signal an error and fall back onto the CPU delegate.

When quantizing the concept of a representative dataset comes into play. In Python this takes the form of a generator that returns a random assortment of inputs from a dataset, so that the model may be inspected for patterns, minimum and maximum values that occur at different points, in order to minimize the effect of quantization on accuracy.

7.3 Keras

TensorFlow interoperates with and includes a complete interface to the Keras library, which includes its own additional ways to define models. For instance `.api.applications` under keras contains prebuilt models for various uses such as image recognition that can then be freely used and manipulated for our needs. Another way is the `.models.Sequential` class, which takes a list of layers and concatenates them one after another into a model. Keras layers are much like TensorFlow modules.

7.4 Other Tools

7.4.1 Visualization: Netron & Zetane

Netron (20) is an incredibly useful open-source tool for inspecting, visualizing and generally debugging machine learning model graphs. Of relevant formats it supports ONNX, TensorFlow and Keras, along with support to display all metadata contained in a given layer. It runs either as a web app or an offline local program. Netron provides an interactive graph GUI which allows searching, layouting, and exporting to PNG or SVG. An example of such an export in Figure 3.

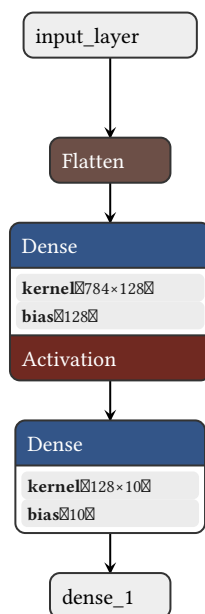


Figure 3: Example of a Netron graph render

Zetane (21) deserves an honourable mention, it is a closed-source batteries included environment for designing ML software and includes a viewer very similar to Netron. This viewer however is much more capable and includes 3D visualizations of convolutions on images, heatmaps, graphs of values,

running the model with weights and inspecting propagation. It is also callable directly via a Python API, so unlike Netron it can be integrated as a frontend into other projects.

7.4.2 Conversion: Tensor2onnx & Tflite2onnx

All manner of programs exist to convert between model formats, which is not a trivial task however. Very generally models can be converted but some programs lose details, such as weights, names, structures that are concisely represented may unwrap into larger forms, and more. For a given application, care should be given to verify that all required information is intact.

Chapter 8

Hardware Specifics

This entire section references mainly (22), VivanteVIP. Most importantly the NPU implements the OpenVX™ API, and so can be controlled using the VeriSilicon libOpenVX implementation.

8.1 NPU Contents

Let us now walk through the contents of our NPU chip. Very little information is available publicly.

8.1.1 Parallel Processing Unit (PPU)

SIMD4, with 4 units with 256 threads each. General purpose highly-parallel unit for standard arithmetic operations.

8.1.2 Neural Network Engine

Specialized support for convolutions. Performs 1152 Multiply–Accumulate Operation (MAC) operations per clock cycle.

8.1.3 Tensor Processor

Supported operations are listed in Table 1. Handles operations other than convolutions. Does not seem to support 32 bit floats.

Table 1: Tensor Processor supported operations

Pooling	Max, average
Unpooling	Yes
Activation	ReLU, Leaky ReLU (LUT for other types)
Normalization	Yes
Region Proposal Support	Yes

It is perhaps useful to mention that some interfacing is proprietary such as interrupts. The NPU can send CPU interrupts, these are set using the driver and cannot in fact be understood without the driver since the NPU and driver both assign arbitrary meaning to the given bits of the interrupt information register.

8.2 Configuration Environment Variables read by TIM-VX

In the i.MX Machine Learning User’s Guide (23) section 6.1.2, we can read about configuration variables.

Table 2: Configuration Environment Variables read by TIM-VX

USE_GPU_INFERENCE	As the NPU and GPU share this driver, TIM-VX uses the value of this variable to determine which to use, "1" for GPU or "0" for NPU
CNN_PERF	Prints how long operations take. Requires VIV_VX_DEBUG_LEVEL=1 and implied by VIV_VX_PROFILE
NN_EXT_SHOW_PERF	Shows the details on how the compiler determines performance
VIV_VX_PROFILE	Enables creation of vprofiler_XXX.vpd files which may be examined using the Vivante vAnalyzer tool from the Vivante VDK. Information is either per-node (value: "1") or per-graph (value: "2")
VIV_VX_DEBUG_LEVEL	Prints extra debug information
VIV_MEMORY_PROFILE	Only applies to CPU/GPU
VIV_VX_ENABLE_CACHE_GRAPH_BINARY	Enables saving of the compiled graph to disk with a hash so that the next time the same graph would be compiled, this *.nb file will be loaded instead. In addition to this the documentations claims that warmup time may take more than one inference
VIV_VX_CACHE_BINARY_GRAPH_DIR	Directory to save the cache files to

8.3 Power Modes

The NPU can be set to 4 different states,

Table 3: Power Modes

On	Standard full-power
Off	Can be powered off, since after leaving this state the device is reinitialized
Idle	Clock speed lowered to $\frac{1}{64^{\text{th}}}$
Suspend	Idle clock speed and requires some time to reach idle

Chapter 9

Frameworks

We will now cover the most popular and best supported frameworks for the NPU.

9.1 LiteRT (Lite RunTime)

This project is closely related to TensorFlow and acts as its ambassador for resource-constrained devices. Tensorflow is used to train, prepare, and debug the model, before it's deployed to a device where only a stripped down runtime is installed. As such it does not contain much beyond what is needed to run inference. Models need to be converted into the FlatBuffers tflite format before use (24).

When running the library we must set an environment variable which libvx checks to see which device it should use, either NPU or GPU:

```
1 export USE_GPU_INFERENCE=0
```

Shell

Next we must load the external dynamic library which we pass into the LiteRT interpreter in Listing 23.

```
1 import tflite_runtime.interpreter as tflite
2
3 external_delegates = [
4     tflite.load_delegate("/usr/lib/libvx_delegate.so", "")
5 ]
```

Python

Listing 23: Loading LiteRT delegate

Delegates are shims between the Tensorflow python library and the external system library acting as device driver, so even though we want to use libOpenVX.so, we instead load the libvx_delegate.so library which links against it.

```
1 interpreter = tflite.Interpreter(
2     model_path=args.model_file,
3     experimental_delegates=external_delegates,
4 )
```

Python

Listing 24: Create the LiteRT interpreter object

After creating the interpreter object with Listing 24 we can interact with it in one of two ways. Either we have a prepared calling convention called a signature inside the model and call that as in Listing 25 which directly returns an output tensor.


```
1 signature = model.get_signature_runner()
2 signature(x=<tensor>)
```

Python

Listing 25: LiteRT signature runner

Or we go about explicitly setting and reading each input and output. For this it is useful to know their properties, so we may call `model.get_input_details()` or `model.get_output_details()` respectively. These functions return lists of detail objects containing the shape, name and index. This index can be used to read or write tensors with `set_tensor` in Listing 26.

```
1 model.set_tensor(model.get_input_details()[0]['index'], input_data)
2 model.get_tensor(model.get_output_details()[0]['index'])
```

 Python

Listing 26: LiteRT directly setting input tensors

9.2 Tensor Virtual Machine (TVM)

As the full name Apache Tensor Virtual Machine (TVM) (25) might suggest, this is an open-source framework built by Apache in a similar vein to ONNX including their own IR called Relay that is supposed to allow optimizations to be shared between multiple target backends.

It does not have its own format for files, opting to be able to import any of the other frameworks'. It does have the additional capability to compile down models into C++ libraries in the form of shared object files.

According to the recipe's EXTRA_OECMAKE and the fact that it links against tim-vx in the tvm bitbake recipe, support for VsiNPU should already be built in.

The `tvm.relay.frontend` module allows us to load a model, from almost any format user in machine learning, using functions such as `from_keras()`, `from_onnx()`, `from_paddle()` or `from_tflite()`.

However merely trying to import the `tvm.relay` module throws an error concerning python not being able to find the scipy library. Scipy is not packaged by any of the standard NXP layers and didn't have a workable recipe that could be found before this went out of scope for this work.

Another interesting property of TVM is that ONNX can use it as a backend directly. Strangely enough though ONNX even when compiled with `-Donnxruntime_USE_TVM=ON` does not acknowledge TVM as a valid backend.

9.3 Open Neural Network Exchange (ONNX)

Though originally authored by joint efforts of Facebook and Microsoft, this project has flourished into a widely supported open-source ecosystem (26). It comprises a comprehensive specification for models, formats, types, operators and abstract data descriptions. It includes protobuf definitions of their `.onnx` model files. It used to support only inference, but training was added with ONNX IR spec version 7 (27). Models here are represented as either just a stateless inference function in the case of inference-only models, or may be extended with an initialization and a training method which may modify internal stateful variables of the given model. ONNX also encodes a block of operators that encode a more complex task, these may be substituted for builtins by the runtime based off of their name⁴. The internal structure is a list of acyclically dependent topologically sorted nodes, each node providing a name, metadata, i/o and the given operation performed. ONNX however does provide HOF-like operators that are applied to entire subgraphs, thus substituting the need for self-references (28). These nodes are strung together as a pipeline each input being connected to a previously declared output of the same name. Each output name is unique since Single Static Assignment (SSA) is mandatory, verification tools for this and other properties are available from the creators of ONNX. The last argument of some operators is marked as variadic, thus allowing as is traditional with regular languages to pass an arbitrary number of inputs/outputs to it, obviously respecting the minimum arity. These graphs are meant to be assembled programatically, however ONNX does provide a textual form of their files.

Implementing a backend for ONNX is done by providing a Python shim wrapping the functionality as we have compiled in support for VsiNPU, we can simply list it as our inference session provider and run inference as is shown in Listing 27.

⁴Went through changes in IRv9

If ONNXruntime is downloaded from pip, we can use the available provider CPU and Azure. VsiNPU is listed as known but unavailable. Activating it yields an unavailable error so the ONNX we use must be from Yocto.

```

1  import onnxruntime
2  session = onnxruntime.InferenceSession(
3      "model.onnx",
4      providers = [ "VSIINPUExecutionProvider" ]
5  )
6  session.run(
7      input_feed= {
8          "x": np.full(
9              fill_value=[1.0],
10             shape=(1,28,28),
11             dtype=np.float32,
12         )
13     },
14     output_names = ["add"],
15 )
16 outputs = session.run(None, {"input": inputTensor})

```

Listing 27: Running ONNX on NPU

You may convert any torch model into ONNX. Listing 28 shows how given a torch/tensorflow model one can export it into an onnx file.

```

1  torch.onnx.export(
2      model, # model being run
3      torch.randn(1, 28, 28), # model input (or a tuple for multiple inputs)
4      "fashion_mnist_model.onnx", # where to save the model
5      input_names = ['input'], # the model's input names
6      output_names = ['output'], # the model's output names
7  )

```

Listing 28: Exporting Torch to ONNX

9.4 Unaddressed frameworks

9.4.1 The NXP eIQ environment

The NXP eIQ stack officially supports, LiteRT, ONNX, PyTorch and OpenCV. Of these only LiteRT officially supports the included NPU. Due to the fact that it only calls the other libraries listed and that the environment itself is a large and unwieldy project, we opted to skip it in this work.

9.4.2 Android NNAPI

Android provides a direct official API for use in Machine Learning acceleration (29). It is however only exclusively available on Android which requires negotiating with NXP for an image and is to be deprecated in the future in favour of LiteRT. The Linux Kernel also has something called the NNAPI, however that is completely unrelated.

9.4.3 Paddle Paddle

A popular Chinese framework for machine vision, it seems well-documented but much of the community and so documentation is mandarin (30). It is not mentioned by NXP in documentation, however it is marked as having supported for TIM-VX, so may be explored in the future.

Chapter 10

Performance

The entire point of the NPU is that the specialized hardware therein should perform these specific following tasks very swiftly and efficiently. We will now discuss what affects the final performance and then put this claim to a practical test.

10.0.1 Startup

This is apparent when running the C++ example where measuring the entire program's runtime results in 0.2 seconds on CPU and 3.4 seconds when running on NPU even though the internal timer measuring pure inference time shows that NPU speeds up inference from 35ms to 3ms.

After first starting a program you must run at least one so-called “warmup” inference which leads to the pipeline of compilation, optimization, deciding on tiling, and generally preparing the model for a run, before actually running inference on the given model. We state “at least one” because that is what the documentation states, we only encountered warmup requiring one inference run, however documentation advises to first test how warmup behaves for the given model and then determine how to benchmark. This takes time in a magnitude of seconds to tens of seconds depending on model size. After this first inference all subsequent inferences should be a consistent and much higher speed, as long as that model's handle is kept (31). Multiple models can be kept in memory at once, each needs its own warmup phase and then as long as they fit, they can be kept and used at will intermittently.

Some frameworks offer a way to compile multiple models so they efficiently live together in memory (32). This might of course cause some models to have most of the cache memory to themselves, while leaving the rest with very little, however it prioritizes the models based on the user's preference so it is a tradeoff of note having to expulse the cache every time a model is switched and may at times be faster even considering that.

The abstract graph built by the user is analyzed by the implementation and operations may be merged, changed and transformed as to provide the best possible performance (33). This is what takes up the crux of the startup delay and what graph binary caching (see Table 2)) tries to address. It does not affect runs of instance of a program, but rather skips part of the “warmup” inference in subsequent runs.

Input data must be split into smaller fixed-size chunks to fit onto on-chip memory and so that possible offloads into DRAM and other expensive memory operations may be minimized. This is done automatically by the implementation however shows up often in performance logs. The DMA controller then requests and processes tiles as the NPU requires.

10.0.2 Quantization

Since the NPU itself is built for all of 8/16/32-bit wide floats (23) and different sizes of integers, we may wish to lessen the load and speed-up the inference by opting to sacrifice the precision of the result and instead of running the entire process with F32, we use INT8, according to (34) this may lead to the operations being implemented 2 to 4 times faster with integers.

10.0.3 Dynamic loading of Libraries

In the case of both the Python and C++ interfaces, using either ONNX or LiteRT, the case is always that the interfacing library must be loaded dynamically. Loading may take up a significant amount of time, in one experiment the Python profiler shows 7\% of the runtime taken up by `do_lookup_x` from `ld-linux.so`.

10.0.4 Bus

Since the NPU acts as a completely separate device from the standard SOC's CPU and memory, even having its own clock, we must transfer the model and input parameters over the Advanced eXtensible Interface (AXI) and Advanced High-performance Bus (AHB) and that takes time.

Behind that Bus lies a Memory Controller, scheduler and more which all work together to slow down the call especially initially.

If a tensor or other piece of our pipeline is marked as being bound to memory on the device it is called a "device tensor" (35). As an example CUDA allows the creation of a batch of memory marked "CudaPinned" which is asynchronously accessible to the device.

10.0.5 NPU Clogging

During tests we often observed that if the input data is too large, or the NPU is otherwise mishandled then all further requests to it simply block forever. I am sure the NPU can be reset, however we opted to just restart the device for now whenever this occurs, sometimes having to hard-kill all processes that are working with the NPU as they even block system restarts.

10.1 Benchmarking Practical Results

Running the same model under LiteRT and ONNX, yields interestingly different results for CPU speed and NPU Warmup, which makes sense as even though the models implement the same structure they have different representations and so may be compiled very differently. Seems that once the model is up and running though the NPU has the same performance under both.

Table 4: mobilenet_v1_1.0_224_quant Performance Metrics

Framework	CPU	NPU Warmup	NPU
ONNX	50ms	16ms	4ms
LiteRT	132ms	380ms	2.3ms

With this we can estimate what this NPU would allow us to do in terms of live image classification. Even though the input image is only 224 by 224 pixels, using the CPU we would only get 7 images identified per second, while using the NPU that goes up to 435 images per second, for example we could classify in real time every single frame of 18 different cameras before dropping below 24 frames per second on each.

Table 5: MobileNetV3Large Quantization tests

Type	CPU	NPU Warmup	NPU
F32	107ms	6.4s	11ms
I8	107ms	6.5s	10ms
F32 (No optimization)	98ms	749ms	345ms

Table 4 shows us that the biggest difference lies in whether or not optimizations are enabled. For the next Table 6 we use the built-in `keras.applications` models. They are ready-made, of various sizes,

well-documented, don't use exotic operators, and accurately depict what a common workload might look like, so we decided they'd make good sources of performance data. Other ML papers on the matter seem to also use some combinations of these model architectures to show results (2).

Table 6: Various Models performance

Model	CPU	NPU Warmup	NPU
VGG19	4s	37s	34ms
MobileNetV3Large	107ms	6.4s	11ms
MobileNetV3Small	36ms	2.6s	4ms
MobileNetV2	82ms	5.1s	9ms
MobileNet	131ms	5.5s	7ms

As the NPU is optimized only for specific tasks so certain operands may end up just being run on the CPU even if the NPU is set as the target, only emitting a Warning notice to standard error.

In addition, an inefficiently built model may even run slower on NPU due to the various overheads if it is not something that gets optimized, bumping a 52ms runtime on the CPU up to a consistent 200ms on the NPU. I reiterate, enabling hardware acceleration slowed down our inference to 25% of what it achieved on just CPU.

Testing with absolutely minimal models like the one suggested for usage on the MNIST dataset, shows that below some threshold of size the CPU is again faster. For example the NPU will not go below 400 μ s while the CPU manages 180 μ s.

Chapter 11

Suggesting PikeOS integration

We see that the most integral part of porting this ecosystem to a given OS, would be to get TIM-VX working as that is targeted by all the other libraries. For that, the following must first be addressed.

Porting libOpenVX This will be the most difficult part as that is closed-source and shipped as a prebuilt binary artefact. This shared object binary can of course be patched and edited to work around some issues; however, there is little that can be done if the library does not work for some other more nuanced reason.

Runtime dependencies Luckily, libOpenVX has very few runtime dependencies. Apart from the standard set of linux libraries libOpenVX requires libVSC, libGAL, libArchModelSw and libNNArchPerf, all of which are also shipped as binary blobs. Only shipping libOpenVX and omitting TIM-VX does not lessen the load substantially, as all of these dependencies are also required by it.

Compile TIM-VX against our libOpenVX library and ship it All our mentioned frameworks have multiple sets of bindings, primarily always Python and C++, along with a mix of other languages. For performance sensitive applications, it would be beneficial to support the C++ API while Python will depend on whether clients wish to prototype their applications on the device or if the workflow of training on external hardware and only running on the device is sufficient.

Wrapper ML frameworks need to be ported Once that is in place TVM and OpenCV support external file types, while ONNX, Keras and TensorFlow provide tooling to convert and often have built-in facilities to work with, import and export all of their respective formats. And so supporting one of these well should suffice for most applications. We would suggest the Lite Runtime as that is the one “blessed” by NXP as their primary supported framework and it has the largest community.

Chapter 12

Conclusion

We set out to perform a reconnaissance of hardware acceleration in the context of machine learning on embedded devices. To determine how the hardware is accessed, what external libraries let us interface into it, and to test that the performance gains are as claimed and suspected.

Much of the authors' time was spent on familiarization with the bitbake Yocto build system, BSPs, flashing to the EVK, navigating hardware vendor documentation, and other themes completely tangential to the problems we set out to address. Despite that, we identified the primary point of ingress for controlling the NPU, several libraries that support it even outside the NXP specification.

We either compiled these libraries with NPU support, recorded what steps were done to do so, and afterwards described how to specify to the library that it should utilize the NPU. Or in the cases where a roadblock was hit, we described what the next step would be in getting it to work. All the code used is included as part of this work.

We identified a good set of models to use for benchmarks and then benchmarked the libraries on CPU and NPU, concluding that in terms of raw NPU inference performance on the same model their differences are negligible, if optimized correctly. And that the the NPU itself achieves tenfold or hundredfold performance compared to the CPU, when configured correctly, depending on model.

Finally, we outlined which libraries are relevant for PikeOS to claim NPU support and briefly discussed from an outside perspective how that might be done.

Chapter 13

Build configuration

```
1  Build Configuration:
2  BB_VERSION           = "2.8.0"
3  BUILD_SYS            = "x86_64-linux"
4  NATIVELSBSTRING      = "universal"
5  TARGET_SYS           = "aarch64-poky-linux"
6  MACHINE              = "tqma8mpxl-mba8mpxl"
7  DISTRO               = "fsl-imx-wayland"
8  DISTRO_VERSION       = "6.6-scarthgap"
9  TUNE_FEATURES        = "aarch64 armv8a crc crypto"
10 TARGET_FPU           = ""
11 meta
12 meta-poky            = "HEAD:200d12b6a58ad961d60a7774ca0f7a9d29498724"
13 meta-oe
14 meta-python
15 meta-multimedia      = "HEAD:72018ca1b1a471226917e8246e8bbf9a374ccf97"
16 meta-freescale       = "HEAD:0627128b341cfb2bef7a0832ce8cac0ce1127f13"
17 meta-qt6             = "HEAD:586a6cb5aec755803a3be3cec359baafe89d6432"
18 meta-tq              = "HEAD:257b8c0b4b6df3bb27fb69bd2312dd254c73fed3"
19 meta-imx-ml
20 meta-imx-sdk
21 meta-imx-bsp         = "HEAD:219f6d04a4c339eb6f2dc626f944bbdf9a716ff5"
22 meta-arm
23 meta-arm-toolchain   = "HEAD:950a4afce46a359def2958bd9ae33fc08ff9bb0d"
24 meta-freescale-distro = "HEAD:b9d6a5d9931922558046d230c1f5f4ef6ee72345"
25 meta-overlay         = "<unknown>:<unknown>"
26 meta-virtualization  = "HEAD:6f3c1d8f90947408a6587be222fec575a1ca5195"
27 meta-file systems
28 meta-networking      = "HEAD:72018ca1b1a471226917e8246e8bbf9a374ccf97"
29 meta-tpm
30 meta-openssl         = "HEAD:459d837338ca230254baa2994f870bf6eb9d0139"
31 meta-clang           = "HEAD:2b7433611d80f6d0ee1b04156fa91fc73d3c2665"
```

Chapter 14

Glossary

AHB. Advanced High-performance Bus	34
AXI. Advanced eXtensible Interface	34
EVK. Evaluation Kit	2, 5, 11, 37
HOF. Higher Order Function	30
IR. Intermediate Representation	22, 23, 30
MAC. Multiply-Accumulate Operation. $a \leftarrow a + (b \times c)$	27
ONNX. Open Neural Network Exchange	4, 22, 23, 25, 30, 31, 34, 36
SSA. Single Static Assignment	30
TIM-VX. Tensor Interface Module for OpenVX	3, 6, 11, 18, 19, 21, 27, 28, 32
TVM. Tensor Virtual Machine	4, 30, 36
VsiNPU. VeriSilicon™ NPU	19, 30, 31
XML. eXtensible Markup Language	17
tensor. Sufficiently described for purposes of this text as multidimensional arrays	3, 6, 13, 15, 22, 23, 24, 27, 29, 30, 34

Bibliography

1. VeriSilicon/TIM-VX. Online. [Accessed 10 February 2025]. Available from: <https://github.com/VeriSilicon/TIM-VX>
2. MOGAKA, Obed M., FORSBERG, Håkan and DANESHTALAB, Masoud. *Bridging Quantization and Deployment: A Fixed-Point Workflow for FPGA Accelerators*. 2025.
3. Farewell and thank you for the continued partnership, Francois Chollet!- Google Developers Blog. Online. [Accessed 10 May 2025]. Available from: <https://developers.googleblog.com/en/farewell-and-thank-you-for-the-continued-partnership-francois-chollet/>
4. h5py: Manual. Online. [Accessed 29 April 2025]. Available from: <https://docs.h5py.org/en/stable/quick.html>
5. The HDF5 Field Guide | Getting Started. Online. [Accessed 1 May 2025]. Available from: https://support.hdfgroup.org/documentation/hdf5/latest/_intro_h_d_f5.html
6. HAMANN, David. David Hamann. Online. 2020. [Accessed 25 October 2024]. Available from: <https://davidhamann.de/2020/04/05/exploiting-python-pickle/>
7. GitHub - yoctoproject/bmaptool: BMAP Tools. Online. [Accessed 2 May 2025]. Available from: <https://github.com/yoctoproject/bmaptool>
8. The System Package Data Exchange™ - Learn. Online. [Accessed 10 May 2025]. Available from: <https://spdx.dev/learn/overview/>
9. The Yocto Project - Dev Manual. Online. [Accessed 10 May 2025]. Available from: <https://docs.yoctoproject.org/dev/dev-manual/>
10. FreeBSD Wiki - Flattened Device Tree. Online. [Accessed 10 May 2025]. Available from: <https://wiki.freebsd.org/FlattenedDeviceTree>
11. The Khronos Group. Online. 2011. [Accessed 25 October 2024]. Available from: <https://www.khronos.org/opencvx/>
12. The OpenVX™ Neural Network Extension. Online. [Accessed 2 May 2025]. Available from: https://registry.khronos.org/OpenVX/extensions/vx_khr_nn/1.3/html/vx_khr_nn_1_3.html
13. KhronosGroup/OpenVX-sample-impl. Online. [Accessed 25 October 2024]. Available from: <https://github.com/KhronosGroup/OpenVX-sample-impl>
14. AMD OpenVX documentation | ROCm™ Software Future Release. Online. [Accessed 3 May 2025]. Available from: https://rocm.docs.amd.com/projects/MIVisionX/en/develop/how-to/amd_opencvx.html
15. repo - The Multiple Git Repository Tool. Online. [Accessed 2 May 2025]. Available from: <https://gerrit.googlesource.com/git-repo>
16. GitHub - nxp-imx/imx-manifest: i.MX Release Manifest. Online. [Accessed 2 May 2025]. Available from: <https://github.com/nxp-imx/imx-manifest>
17. Yocto Project Reference Manual | Required Packages for the Host Development System. Online. [Accessed 2 May 2025]. Available from: <https://docs.yoctoproject.org/2.4/ref-manual/ref-manual.html#detailed-supported-distros>

Bibliography

18. TIM VX Backend For Running OpenCV On NPU. Online. [Accessed 2 May 2025]. Available from: <https://github.com/opencv/opencv/wiki/TIM-VX-Backend-For-Running-OpenCV-On-NPU>
19. Module — PyTorch 2.6 documentation. Online. [Accessed 17 February 2025]. Available from: <https://pytorch.org/docs/stable/generated/torch.nn.Module.html>
20. Netron App. Online. [Accessed 2 May 2025]. Available from: <https://netron.app/>
21. Zetane | Reliable AI automation for high-risk industries. Online. [Accessed 2 May 2025]. Available from: <https://zetane.com/>
22. i.MX 8M Plus Applications Processor Reference Manual. Online. [Accessed 1 January 2025]. Available from: <https://www.nxp.com/webapp/Download?colCode=IMX8MPRM>
23. i.MX Machine Learning User's Guide. Online. [Accessed 25 October 2024]. Available from: <https://www.nxp.com/docs/en/user-guide/IMX-MACHINE-LEARNING-UG.pdf>
24. LiteRT overview - Google AI Edge. Online. [Accessed 29 April 2025]. Available from: <https://ai.google.dev/edge/litert>
25. WANG, Yanzhao and XIE, Fei. Extending Tensor Virtual Machine to Support Deep-Learning Accelerators with Convolution Cores. In : Online. 2022. p. 189--194. Available from: <https://ieeexplore.ieee.org/document/9763822/?arnumber=9763822>
26. ONNX. Online. [Accessed 2 May 2025]. Available from: <https://onnx.ai/>
27. Open Neural Network Exchange Intermediate Representation (ONNX IR) Specification - ONNX 1.18.0 documentation. Online. [Accessed 17 February 2025]. Available from: <https://onnx.ai/onnx/repo-docs/IR.html>
28. ONNX Concepts - ONNX 1.18.0 documentation. Online. [Accessed 17 February 2025]. Available from: <https://onnx.ai/onnx/intro/concepts.html>
29. Neural Networks API | Android NDK | Android Developers. Online. [Accessed 2 May 2025]. Available from: <https://developer.android.com/ndk/guides/neuralnetworks>
30. GitHub - PaddlePaddle/Paddle: PArallel Distributed Deep LEarning: Machine Learning Framework from Industrial Practice. Online. [Accessed 3 May 2025]. Available from: <https://github.com/PaddlePaddle/Paddle>
31. i.MX 8M Plus NPU Warmup Time. Online. [Accessed 3 May 2025]. Available from: <https://www.mouser.com/pdfDocs/AN12964.pdf>
32. Edge TPU Compiler | Coral. Online. [Accessed 1 May 2025]. Available from: <https://coral.ai/docs/edgetpu/compiler/>
33. ABEYSINGHE, Madushan, VILLARREAL, Jesse, WEAVER, Lucas and BAKOS, Jason. OpenVX Graph Optimization for Visual Processor Units. In : *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Online. IEEE, 2019. p. 123--130. Available from: <https://doi.org/10.1109/asap.2019.00-19>
34. PyTorch documentation - Quantization. Online. [Accessed 2 May 2025]. Available from: <https://pytorch.org/docs/stable/quantization.html>
35. Device tensors. Online. [Accessed 3 May 2025]. Available from: <https://onnxruntime.ai/docs/performance/device-tensor.html>

Bibliography

36. NXP Community. Online. 2020. [Accessed 19 February 2025]. Available from: <https://community.nxp.com/t5/i-MX-Processors-Knowledge-Base/How-to-use-OpenVX-extension-for-NPU-GPU-to-accelerate-machine/ta-p/1113429>
37. MBa8MPxL User's Manual. Online. [Accessed 25 October 2024]. Available from: <https://www.tq-group.com/filedownloads/files/products/embedded/manuals/arm/carrierboard/MBa8MPxL/MBa8MPxL.UM.0103.pdf>
38. i.MX 8M Plus Applications Processor Datasheet for Industrial Products. Online. [Accessed 27 October 2024]. Available from: <https://www.nxp.com/webapp/Download?colCode=IMX8MPIEC>