Tartu University

Faculty of Science and Technology

Institute of Technology

Remi Raugme

# Development of a testing framework for monocular depth estimation models on targeted hardware

Bachelor's thesis (12 ECTS)
Computer Engineering

Supervisors:

MSc Quazi Saimoon Islam
MSc Ric Dengel

Tartu 2025

# Abstract/Resümee

**Development of a testing framework for monocular depth estimation models on targeted hardware**

This thesis presents the development of a testing framework designed to evaluate monocular depth estimation (MDE) models on targeted embedded hardware. The work is motivated by the KuupKulgur lunar microrover project, where traditional depth sensing methods are impractical due to size, weight, and power limitations. MDE models have demonstrated strong accuracy. However, their real-time performance on targeted hardware requires validation.

A modular pipeline was implemented to benchmark various MDE models on the NVIDIA Jetson Orin Nano platform. The framework supports multiple inference backends and weight precisions, enabling comparison between PyTorch and Open Neural Network Exchange (ONNX) Runtime using Floating Point (FP)32 and FP16 precision weights. The framework gathers time and hardware usage data during benchmarking.

The developed testing framework provides a valuable tool for evaluating MDE models in robotic systems, including future lunar rovers.

Four MDE models were tested: Depth Anything V2, Metric3Dv2, Depth Pro and SPIdepth. Once optimisation with NVIDIA TensorRT (TRT) was applied, all models showed improved inference times over PyTorch when using FP16 precision weights. When using FP32 values, PyTorch outperformed TRT, with a few exceptions. The best-performing model was Depth Anything V2's smallest relative model when using TRT optimisation level 2 and FP16 precision weights.

**CERCS:** T120 Systems engineering, computer technology; T125 Automation, robotics, control engineering; P176 Artificial intelligence [1]

**Keywords:** monocular depth estimation, robotics, NVIDIA Jetson, testing framework

**Monokulaarsete sügavushinnangumudelite testimisraamistiku arendamine sihtotstarbelisel riistvaral**

Käesolev töö tutvustab testimisraamistiku väljatöötamist, mille eesmärk on hinnata monokulaarseid sügavushinnangu (MSH) mudeleid sihtotstarbelisel manussüsteemi riistvaral. Töö on tehtud KuupKulguri, kuu-mikrokulgur, projekti raames, kus traditsioonilised sügavus andmete leidmise võimalused ei ole praktilised piiratud suuruse, kaalu ja energiatarbimise tõttu. MSH mudelid on tõestanud oma täpsust, kuid nende reaalajas toimivus sihtotstarbelisel riistvaral vajab valideerimist.

Loodi testimisraamistik, et võrrelda erinevaid MSH mudeleid NVIDIA Jetson Orin Nano

platvormil. Raamistik toetab mitmeid inferentsi võimaldavaid Pythoni teeke, võimaldades võrrelda PyTorchi ja Open Neural Network Exchange (ONNX) Runtime'i, kasutades nii Floating Point (FP) 32 kui ka FP16 andmetüübiga kaalusid. Raamistik kogub testimise käigus prottsessidele kulunud aegu ja riistvarakasutuse andmeid.

Arendatud testimisraamistik võimaldab efektiivsemalt hinnata MHS mudeleid ja nende kasutuselevõtu kõlblikust robotisüsteemides, sealhulgas tulevastes kuukulgurites.

Testiti nelja MSH mudelit: Depth Anything V2, Metric3Dv2, Depth Pro ja SPIdepth. Pärast optimeerimist kasutades NVIDIA TensorRT'd (TRT), paranes kõigi mudelite inferentsi aegu võrreldes PyTorchi kasutamisega, kui kasutati FP16 kaalusid. FP32 väärtuste kasutamisel oli PyTorchi jõudlus parem kui TRT puhul, mõningate eranditega. Parima ajaliste tulemustega mudel oli Depth Anything V2 väikseim suhteline mudel, kasutades TRT optimeerimistasets 2 koos FP16 kaaludega.

**CERCS:** T120 Süsteemitehnoloogia, arvutitehnoloogia; T125 Automatiseerimine, robootika, control engineering; P176 Tehisintellekt [1]

**Märksõnad:** monokulaarne sügavushinnang, robootika, NVIDIA Jetson, testimisraamistik

# Contents

**List of Figures** **6**

**List of Tables** **8**

**1 Introduction** **11**
1.1 Problem Statement . . . 12
1.2 Objectives and Overview . . . 12

**2 Literature Review** **14**
2.1 Lunar Rover environmental perception . . . 14
2.2 Monocular depth estimation . . . 14
2.2.1 Monocular depth estimation on Earth . . . 15
2.2.2 Monocular depth estimation in planetary rovers . . . 15
2.3 KuupKulgur's computational hardware . . . 15
2.4 NVIDIA Jetson . . . 15
2.5 Machine Learning frameworks . . . 16
2.5.1 TensorFlow . . . 16
2.5.2 PyTorch . . . 16
2.6 ONNX and ONNX Runtime . . . 16
2.6.1 ONNX . . . 16
2.6.2 ONNX Runtime . . . 17
2.7 Hardware acceleration libraries . . . 17
2.7.1 NVIDIA CUDA and cuDNN . . . 17
2.7.2 NVIDIA TensorRT . . . 17

**3 Methodology** **18**
3.1 Model selection . . . 18
3.2 Model Overview . . . 19
3.2.1 Model sizes . . . 19
3.2.2 Model input . . . 19
3.3 Initial testing . . . 20
3.3.1 Testing on workstation computer . . . 20
3.3.2 Testing on Jetson . . . 20
3.4 Converting to ONNX . . . 21
3.5 Testing framework . . . 21
3.5.1 Structure . . . 22
3.5.2 PyTorch compatibility . . . 24
3.5.3 ONNX Runtime compatibility . . . 24
3.6 Benchmarking metrics . . . 25

3.7 Testing with pipeline . . . 26

**4 Results and Analysis** **27**
4.1 Time related results . . . 27
4.1.1 Total times overview . . . 27
4.1.2 Optimisation time . . . 28
4.1.3 Pre-processing time . . . 28
4.1.4 Inference time . . . 29
4.2 ONNX Runtime hardware usage results . . . 29
4.2.1 CPU usage . . . 30
4.2.2 GPU usage . . . 30
4.2.3 RAM usage . . . 30
4.2.4 Power consumption . . . 31
4.3 PyTorch hardware usage results . . . 31
4.3.1 CPU usage . . . 32
4.3.2 GPU usage . . . 32
4.3.3 RAM usage . . . 32
4.3.4 Power consumption . . . 32

**5 Discussion** **34**
5.1 Testing anomalies . . . 34
5.1.1 PyTorch . . . 34
5.1.2 ONNX Runtime . . . 35
5.2 Hardware utilisation . . . 35

**6 Conclusion** **37**

**A Configuration file** **43**

**B Visualised time results** **47**

**C Visualised hardware utilisation results** **52**

**Non-exclusive license** **54**

# List of Figures


1.1 Current version of KuupKulgur . . . . . 11
1.2 Image from KuupKulgur's camera with depth estimation from Depth Anything V2 . . . . . 12

2.1 Photo by David Köhler on Unsplash and its depth estimation [15] . . . . . 15

3.1 Flowchart of general workflow . . . . . 22
3.2 Flowchart of the pipeline . . . . . 23

4.1 Time spent on optimisation for Depth Anything V2's smallest NoP relative model 28
4.2 Total time spent on pre-processing for Depth Anything V2's smallest NoP relative model . . . . . 29
4.3 Total time spent on inference for Depth Anything V2's smallest NoP relative model . . . . . 29
4.4 CPU usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's smallest NoP relative model . . . . . 30
4.5 GPU usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's smallest NoP relative model . . . . . 30
4.6 RAM usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's smallest NoP relative model . . . . . 31
4.7 Power consumption during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's smallest NoP relative model . . . . . 31
4.8 CPU usage during PyTorch testing using FP16 weights for Depth Anything V2's smallest NoP relative model . . . . . 32
4.9 GPU usage during PyTorch testing using FP16 weights for Depth Anything V2's smallest NoP relative model . . . . . 32
4.10 Power consumption during PyTorch testing using FP16 weights for Depth Anything V2's smallest NoP relative model . . . . . 33

B.1 Time spent on optimisation for Depth Anything V2's medium NoP relative model 47
B.2 Total time spent on pre-processing for Depth Anything V2's medium NoP relative model . . . . . 47
B.3 Total time spent on inference for Depth Anything V2's medium NoP relative model . . . . . 48
B.4 Time spent on optimisation for Depth Anything V2's smallest NoP metric model 48
B.5 Total time spent on pre-processing for Depth Anything V2's smallest NoP metric model . . . . . 49
B.6 Total time spent on inference for Depth Anything V2's smallest NoP metric model 49
B.7 Time spent on optimisation for Depth Anything V2's medium NoP metric model 50

B.8 Total time spent on pre-processing for Depth Anything V2's medium NoP metric model . . . . . 50
B.9 Total time spent on inference for Depth Anything V2's medium NoP metric model 50
B.10 Total time spent on pre-processing for Metric3Dv2's smallest NoP model . . . 51
B.11 Total time spent on inference for Metric3Dv2's smallest NoP model . . . . . 51

C.1 CPU usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's medium NoP relative model . . . . . 52
C.2 GPU usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's medium NoP relative model . . . . . 52
C.3 CPU usage during PyTorch testing using FP16 weights for Depth Anything V2's medium NoP relative model . . . . . 53
C.4 GPU usage during PyTorch testing using FP16 weights for Depth Anything V2's medium NoP relative model . . . . . 53

# List of Tables

3.1 Overview of selected models with accuracy based on the KITTI Eigen split. . . 19

4.1 Average time spent on pre-processes and inference per image with FP32 precision weights . . . . . 27
4.2 Average time spent on pre-processes and inference per image with FP16 precision weights . . . . . 28

5.1 Overview of combinations that could be tested using PyTorch . . . . . 34
5.2 Combinations of precision and optimisation level tested using ONNX Runtime 35

# Acronyms

**AI** Artificial Intelligence. 14–16

**API** Application Programming Interface. 16, 17

**ARM** Advanced RISC Machine. 16

**CPU** Central Processing Unit. 15–17, 24, 25, 30–32, 35–37, 52

**CUDA** Compute Unified Device Architecture. 17

**cuDNN** CUDA Deep Neural Network. 17

**DL** Deep Learning. 14, 16–18

**ESA** European Space Agency. 15

**FP** Floating Point. 2, 3, 17, 19, 24, 26–29, 31, 34, 35, 37, 52

**GPS** Global Positioning System. 18

**GPU** Graphics Processing Unit. 15–17, 20, 24, 25, 30–32, 35–37, 52

**IMU** Inertial Measurement Unit. 18

**INT** Integer. 17, 24, 37

**KITTI** Karlsruhe Institute of Technology and Toyota Technological Institute. 18, 19

**LIDAR** light detection and ranging. 12

**MDE** monocular depth estimation. 2, 12, 14, 15, 18, 21, 37

**ML** Machine Learning. 14, 16–18

**NASA** National Aeronautics and Space Administration. 15

**NN** neural network. 12, 14, 17

**NoP** number of parameters. 27–29, 31, 34, 35, 37, 47–49, 51, 52

**oneDNN** oneAPI Deep Neural Network Library. 17

**ONNX** Open Neural Network Exchange. 2, 16, 17, 21, 22, 24, 26, 29, 34–37, 47

**PCIe** Peripheral Component Interconnect Express. 16

**PWC** Papers With Code. 18, 19

**RAE** relative absolute error. 18, 19

**RAM** Random Access Memory. 16, 25, 32, 36, 37

**RMSE** root mean squared error. 18, 19

**SDK** Software Development Kit. 16, 17

**TPU** Tensor Processing Unit. 16

**TRT** NVIDIA TensorRT. 2, 3, 17, 21, 26–29, 31, 32, 35–37, 47–49, 52

**USB** Universal Serial Bus. 16

# 1 Introduction

In recent years, there has been a renewed global effort towards space exploration [2]. This resurgence is evident in initiatives such as NASA's Artemis programme, which aims to establish a sustained human presence on the Moon [3]. In parallel, the European Space Agency (ESA) is advancing its own exploration efforts through the Terrae Novae programme, in which lunar exploration plays a central role [4].

With these programs comes a growing demand for autonomous systems across various missions. The autonomy of rovers is one such demand [5]. Consequently, numerous new projects have emerged to develop autonomous rovers for both lunar and planetary exploration. One such project is KuupKulgur, shown in Figure 1.1, Estonia's first lunar rover project, which aims to send a microrover to the surface of the Moon [6]. At present, KuupKulgur lacks autonomous capability. To enable autonomous driving, many different systems and functionalities are required. One component is the ability to perceive the depth of the surrounding environment.

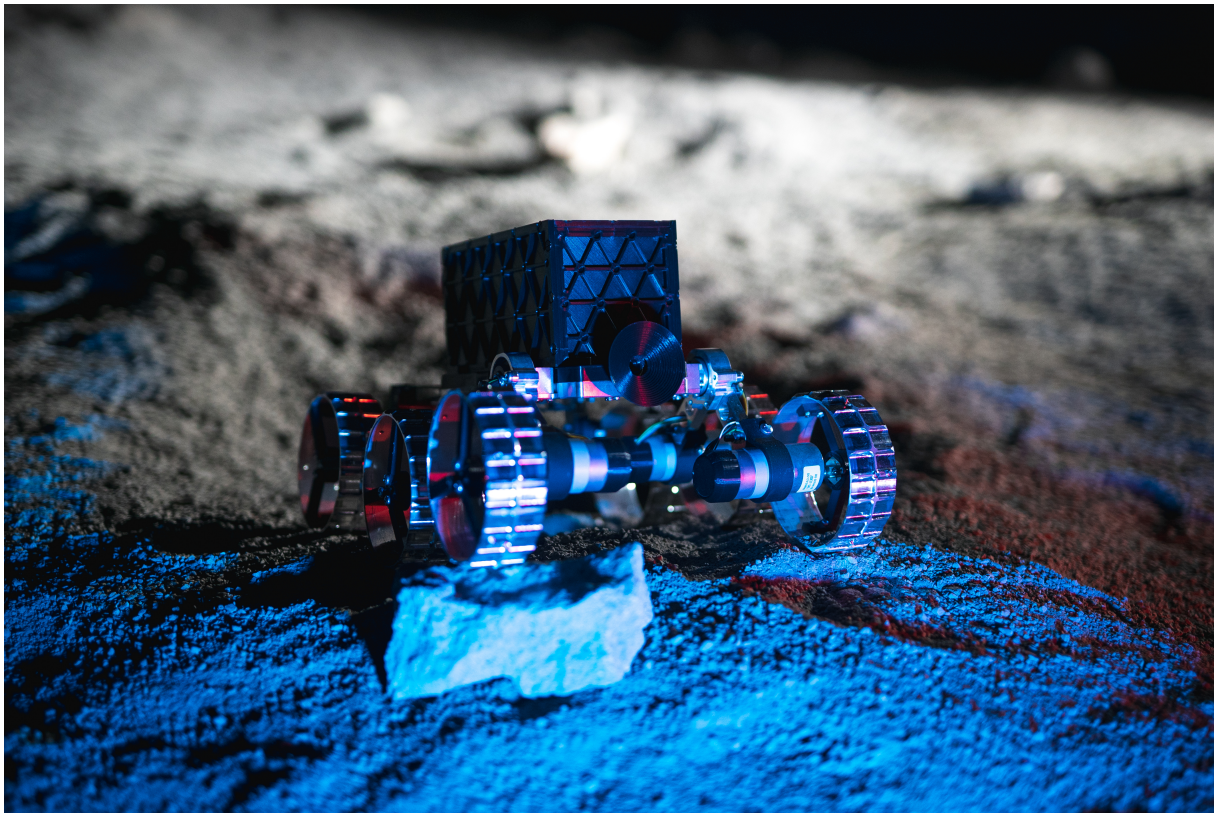

Figure 1.1: Current version of KuupKulgur

Access to accurate depth information is fundamental for many systems that make autonomous driving possible. Among the most prevalent is hazard detection and avoidance. While it is pos-

sible to detect obstacles without depth information, effective avoidance requires knowledge of both the location and size of potential hazards. Without depth perception, estimating the scale and distance of obstacles is unreliable [7].

Due to size, weight, and power constraints, conventional methods for acquiring depth information, such as light detection and ranging (LIDAR) or stereo vision systems, are unsuitable for KuupKulgur. Currently, KuupKulgur uses a single camera and is likely to do so in the final implementation. As a result, monocular depth perception neural networks (NNs) are considered to be a promising solution. Using NNs for depth perception has proven to be accurate. However, it has been using comparatively more computationally powerful and not radiation-resistant hardware, which is unsuitable for space exploration [8]. Therefore, these models need to be evaluated using hardware suitable for space exploration. An example image from KuupKulgur's camera (left) with relative depth estimation from Depth Anything V2 (right) can be seen in Figure 1.2.

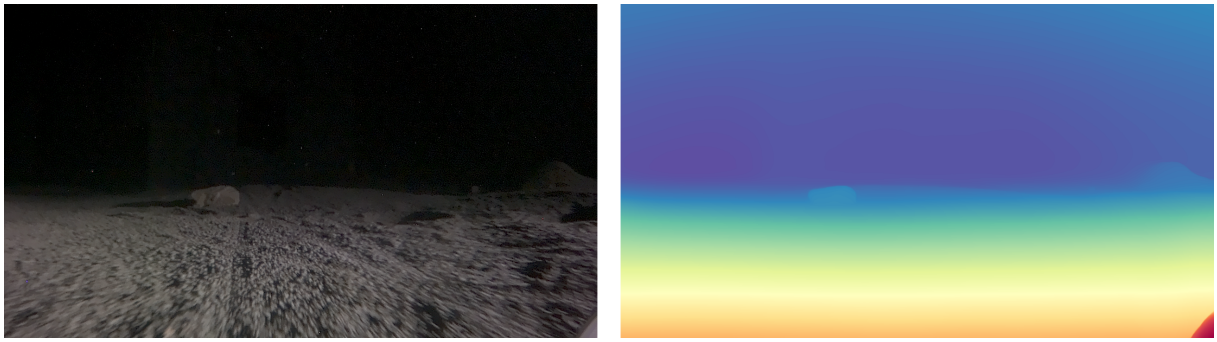

Figure 1.2: Image from KuupKulgur's camera with depth estimation from Depth Anything V2

## 1.1 Problem Statement

Deploying monocular depth perception models on targeted hardware is a non-trivial task. In addition, comprehensive validation is required to determine whether the inference runtimes and hardware demands of current MDE models fall within acceptable thresholds. This validation must be efficient and repeatable, since the field of MDE is evolving rapidly, with new models and optimisations being introduced regularly. Furthermore, given the computational and time cost associated with retraining and adapting models for a specific use case. Development of a reliable and streamlined process for evaluating model suitability before integration is essential.

## 1.2 Objectives and Overview

The main objectives of this thesis are to develop a testing framework for evaluating MDE models and evaluate them for use on KuupKulgur. This requires the completion of the following:

1. Research different depth estimation models that provide promising results and could be usable on KuupKulgur.
2. Develop a framework to test the models in the default configuration.
3. Study and implement optimisation strategies to improve model performance on targeted hardware.

4. Provide a comparative evaluation and analysis of the performance of the models on relevant hardware.

The text in this thesis was proofread using artificial intelligence to streamline the author's original writing.

# 2 Literature Review

## 2.1 Lunar Rover environmental perception

Lunar and planetary rovers that have been deployed have used different sensors to help them perceive the surrounding environment. These include stereo cameras, monocular and multi-spectral cameras, as well as other sensor modalities [9]. In practice, environmental perception systems almost always integrate a combination of sensors to compensate for the limitations of any individual sensor. However, as efforts continue to make new rovers smaller and more power-efficient, integrating and powering all of these traditional sensor systems becomes increasingly challenging. To address this, extracting equivalent or sufficient environmental information using fewer and simpler sensors is required.

A promising direction is the utilisation of Machine Learning (ML), and more broadly Artificial Intelligence (AI) techniques [10]. In particular, the ability to infer depth and other environmental data from a single RGB camera would offer significant advantages. Achieving this would eliminate the need for stereo cameras and other sensors used for environmental perception, reducing weight, cost and size. One such technique is MDE, a computer vision technique powered by Deep Learning (DL), that can extract depth information from a single RGB image. However, applying AI-based techniques such as MDE introduces additional computational costs, compared to dedicated sensors. Historically, planetary rovers have had rather limited computational capabilities, which has restricted the deployment of such ML-based solutions [11].

Although ML has not been used much in planetary rovers historically, recent missions have begun integrating different ML techniques. The prime examples are the Curiosity and Perseverance Mars rovers [12]. The rovers used ML in a few key areas, firstly for aiding in the use of instruments [12]. Secondly, Perseverance used ML for autonomous navigation, mainly path planning [13].

## 2.2 Monocular depth estimation

Monocular depth estimation has been proven to be an effective solution for gathering depth information [8]. The methods used to achieve results differ, but almost all are some sort of deep NN [14]. Mainly DL using encoder-decoder networks. This means using different-sized convolution layers on the encoder side and deconvolution layers on the decoder side. Concatenation is used on corresponding-sized convolution and deconvolution layers to retain information gathered from each layer [14].

### 2.2.1 Monocular depth estimation on Earth

MDE models have proven to provide accurate results. However, time has not been a top priority for most of the solutions, so using them on systems with limited computational hardware requires testing and optimisation. An example output from a relative MDE model, with the original image can be seen in Figure 2.1

Figure 2.1: Photo by David Köhler on Unsplash and its depth estimation [15]

### 2.2.2 Monocular depth estimation in planetary rovers

Although no current or previous planetary rovers have used MDE models. Research and testing are being done by National Aeronautics and Space Administration (NASA), European Space Agency (ESA) and many others to see if it would be possible [16] [17].

## 2.3 KuupKulgur's computational hardware

KuupKulgur is designed to be a lunar microrover that can carry payloads with a CubeSat standard unit size of 2U ( U is a cube with dimensions of 10 x 10 x 10 cm) [6] [18]. This size limitation restricts the amount and type of hardware that can be used. Additionally power consumption of hardware must be taken into consideration. Options are further reduced because the hardware needs to be able to handle radiation, since the goal is for KuupKulgur to be deployed on the Moon. With all these limitations, the list of available options becomes rather short. Suitable candidates for the primary computing hardware include the NVIDIA Jetson and AMD Versal product families. Currently, KuupKulgur uses the NVIDIA Jetson Orin Nano, which was chosen out of all the Jetsons because it is one of the newest and the most power-efficient models available [19].

## 2.4 NVIDIA Jetson

NVIDIA Jetson is a series of System-on-Module boards meant for use in robotics and using AI in embedded applications. Although at the time of writing no commercially available radiation-hardened version of the Jetsons exists, various testing for radiation tolerance has been done [20] [21] [22]. The modules themselves differ in specification, but the core components are the same. The modules consist of a Graphics Processing Unit (GPU), Central Processing Unit

(CPU), Random Access Memory (RAM) and different types of connectors, usually including Universal Serial Bus (USB), Peripheral Component Interconnect Express (PCIe) and Ethernet. The processors used are Advanced RISC Machine (ARM)-based. NVIDIA has designed a Linux-based Software Development Kit (SDK) for the Jetson modules called NVIDIA JetPack. This SDK includes some commonly used libraries needed to get the most out of the hardware on the Jetson board. Other widely used libraries in robotics and AI applications, that have been modified to work on the different Jetson boards, are also available.

## 2.5 Machine Learning frameworks

ML frameworks are collections of tools, libraries and resources designed to simplify and accelerate the development and deployment of ML algorithms and models. These frameworks abstract much of the mathematical complexity involved in implementation. Among the most popular ML frameworks are PyTorch, TensorFlow, Theano and Caffe. Each framework has its advantages and disadvantages, such as the programming language it supports. Most of them are designed to work with Python, which is the most popular language used for ML. The two most popular frameworks for Python, PyTorch and TensorFlow, were considered for use in this thesis.

### 2.5.1 TensorFlow

TensorFlow is an open-source ML framework developed by Google. It supports execution on CPUs, GPUs and Tensor Processing Units (TPUs). It is available in multiple languages, including C++ and JavaScript, although the Python Application Programming Interface (API) is the most widely used and actively maintained. TensorFlow is popular in both research and industry, especially for DL applications. It originally relied on static computational graphs, meaning the model structure had to be fully defined before execution. With the introduction of Eager Execution, it also supports dynamic graph construction, which allows for the graphs to be modified dynamically [23].

### 2.5.2 PyTorch

PyTorch is an open-source ML framework that also supports execution on CPUs, GPUs and TPUs. While C++ support is available, the Python interface is more complete and widely used. The framework is particularly popular in academic research, due to its user-friendly interface and support for dynamic computation graphs, which allow model architecture to be defined and modified during runtime[24]. Since all the models selected in this thesis were implemented using PyTorch, it was chosen as the primary framework for model testing.

## 2.6 ONNX and ONNX Runtime

### 2.6.1 ONNX

ONNX is an open-source format for representing ML models, developed to make switching between different frameworks easy. It defines a standardised representation of commonly used operations in ML and DL models [25].

### 2.6.2 ONNX Runtime

ONNX Runtime is an inference engine developed for optimising and running ONNX models. Although primarily used for inference, training is possible if using PyTorch-based models. ONNX Runtime provides optimisation options for different use cases and hardware. These options include methods to highly optimise a model to run on a GPU or a CPU [26], [27].

Furthermore, ONNX Runtime supports more hardware acceleration libraries than any other ML inference backend. Examples include NVIDIA Compute Unified Device Architecture (CUDA), TRT, DirectML and Intel oneAPI Deep Neural Network Library (oneDNN), with more being added [28].

From the available options that ONNX Runtime provides, two options were considered for use: NVIDIA CUDA and TRT. The other options were discarded because they were meant for use either with CPUs, hardware made by other manufacturers, other operating systems or a combination of the mentioned factors.

## 2.7 Hardware acceleration libraries

Hardware acceleration libraries are software libraries that allow for optimised use of specific hardware. Mostly designed for devices that aren't a CPU. These libraries implement mathematical operations by tailoring to the architecture of the target hardware.

### 2.7.1 NVIDIA CUDA and cuDNN

NVIDIA CUDA is a parallel computing platform and API that enables NVIDIA GPUs to be used for general-purpose computing. CUDA provides access to a virtual instruction set to run calculations on the GPU, greatly increasing the number of similar calculations that can be done compared to a CPU [29].

CUDA is the base for many libraries developed by NVIDIA. CUDA Deep Neural Network (cuDNN) is one such library [30]. The library is designed for use with NNs, providing access to GPU-accelerated operations commonly used in the field [31].

### 2.7.2 NVIDIA TensorRT

TRT is a SDK used for inference of DL models. It uses aspects of CUDA and therefore is confined to use with NVIDIA GPUs. Designed for inference, it has many tools available to optimise the process, several of which aren't available in cuDNN. Because of this, it provides the best inference performance possible from an NVIDIA GPU. Some key optimisations it provides include:

1. Quantisation - converts the weights of a model between different types and sizes of FP and Integer (INT) numbers. For example, from FP32 to FP16 or from FP32 to INT8.

2. Fusion - combines different operations performed during inference.

These optimisations can result in a significant speed-up of inference by possibly orders of magnitude. Additionally, they allow for less memory usage to achieve similar results. However, quantisation does reduce the accuracy of a model, which depends on the difference between the type of original weights being used and the type they were converted into [32]–[34]. TRT was selected as the execution provider for ONNX Runtime, since reducing the time spent for inference is a primary goal of this thesis.

# 3 Methodology

## 3.1 Model selection

The field of MDE using ML is highly active, with new models being released frequently. Consequently, selecting appropriate models proved more time-consuming than initially anticipated, as it required a thorough investigation to determine which models were suitable for evaluation. The criteria for selecting models were as follows

1. **Accuracy** - How closely does the model's output approximate the actual depth? This was primarily evaluated using metrics such as the relative absolute error (RAE) and root mean squared error (RMSE) on relevant datasets.

2. **Framework** – Which framework is used to train and run the model? Ideally, all selected models would utilise the same framework to facilitate streamlined testing and integration.

3. **Output type** - Does the model provide metric or relative depth estimates, or both? Metric output refers to predictions in standard units of length, while relative output conveys depth in relation to other parts of the input. Models capable of producing metric depth were preferred.

4. **Intended use case** – What is the primary application domain of the model? For example, is it designed for indoor or outdoor MDE, and is it tailored for use with still images or video sequences?

The availability of standardised datasets for training and evaluation was critical for assessing the first criterion. One such dataset, and one that is close to the use case of KuupKulgur, was the Karlsruhe Institute of Technology and Toyota Technological Institute (KITTI) dataset. The specific subset used for benchmarking MDE models is the KITTI Eigen split. This dataset comprises driving data collected from a car equipped with four video cameras (two colour and two grayscale), a rotating 3D laser scanner and a combined Global Positioning System (GPS)-Inertial Measurement Unit (IMU) inertial navigation system [35].

Additionally, the Papers With Code (PWC) platform was instrumental in this selection process. It provides information on various ML and DL domains and associated research publications [36]. The platform also includes benchmarking leaderboards for commonly used datasets, which informed part of the model evaluation process in this work. However, not all models are benchmarked across all standard datasets, necessitating comparisons across multiple datasets.

Despite platforms like PWC offering valuable summaries, detailed insights such as dataset performance, framework compatibility, and specific use cases often requires reviewing the original research papers and available implementation code. Ultimately, the selected models for evaluation included Depth Anything V2 [37], SPIdepth [38], Metric3Dv2 [39], Depth Pro [40], and initially, Video Depth Anything [41].

However, Video Depth Anything was excluded from full evaluation. Preliminary testing showed that its performance on static images was comparable to Depth Anything V2, upon which it is based. Moreover, its video-specific features were deemed irrelevant for the real-time image-based inference targeted in this study.

## 3.2 Model Overview

A brief overview of the selected models is presented in Table 3.1. The values reported for RAE and RMSE are based on the KITTI Eigen split. These values were sourced from either PWC or the respective model's publication. For SPIdepth, the unsupervised version's values are presented. Depth Pro did not have results available for the KITTI dataset at the time of writing. In cases where multiple trained weights were available for a model, the best-performing variant is shown.

Table 3.1: Overview of selected models with accuracy based on the KITTI Eigen split.

| **Model** | **Available Output Types** | **Framework** | **Input Type** | **RAE** | **RMSE** |
|---|---|---|---|---|---|
| Depth Anything V2 | Metric<br>Relative | PyTorch | Monocular Image | 0.045 | 1.861 |
| Depth Pro | Metric | PyTorch | Monocular Image | N/A | N/A |
| Metric3Dv2 | Metric<br>Relative | PyTorch | Monocular Image | 0.039 | 1.766 |
| SPIdepth | Relative | PyTorch | Monocular Image | 0.071 | 3.662 |

### 3.2.1 Model sizes

A diverse range of model sizes was selected to investigate whether there is a correlation between model size and performance. All sizes reported are using FP32 values for weights. Depth Anything V2 offered three variants at the time of testing, for both the metric and relative versions. These sizes were: 100 MB, 390 MB, and 1.34 GB. Metric3Dv2 also provided three size options. However, only the two smaller variants, 151 MB and 1.65 GB, were tested, as the largest (5.51 GB) would exceed the memory limits of the Jetson device. SPIdepth and Depth Pro each had only a single available model size: 2.89 GB and 1.91 GB, respectively.

### 3.2.2 Model input

Each model expects inputs of different default dimensions. The required input resolutions, width $\times$ height, are as follows:

- **Depth Anything V2**: 518 $\times$ 518
- **SPIdepth**: 1024 $\times$ 320
- **Metric3Dv2**: 1064 $\times$ 616
- **Depth Pro**: 1536 $\times$ 1536

Input resolution flexibility varied across models. For SPIdepth and Depth Pro, the input resolution was fixed and could not be modified. Metric3Dv2 accepted all input dimensions, while Depth Anything V2 required dimensions that were multiples of 14. These constraints were identified during model testing and conversion attempts. To facilitate a fair comparison, it was decided to normalise model inputs as closely as possible to a resolution of $1280 \times 720$, also known as 720p. This resolution was selected since it is a widely used standard.

## 3.3 Initial testing

Initial testing involved first verifying the functionality of each model on a workstation computer, followed by the same on the target hardware. Additionally, determining compatible versions of Python, PyTorch, and other required libraries that would work for all models. All initial testing was conducted using PyTorch as the framework, with a GPU as the device used for inference. Additionally, functionality was implemented to measure the inference time for a single forward pass across all models.

### 3.3.1 Testing on workstation computer

Preparing each model for testing involved the following three key steps:

1. Cloning the model's official repository.
2. Downloading the pre-trained weights.
3. Creating a virtual environment and downloading the necessary Python libraries.

The first two steps were straightforward for all models. However, the third step presented challenges, as some projects did not include a comprehensive list of required Python dependencies. Once the mentioned steps were completed, running the models was relatively straightforward, as each repository included examples or instructions for inference.

Once all the models could be successfully run, the next task was to identify versions of Python and the necessary libraries that were compatible across all selected models. Several suitable combinations were identified, but the following combination was ultimately selected:

- Python version 3.10.16
- PyTorch version 2.3
- Torchvision version 0.18

These were the selected versions, since these were the newest PyTorch and Torchvision versions that were supported by the Jetpack version used on the Jetson.

### 3.3.2 Testing on Jetson

Getting the models working on the Jetson followed a similar general process. However, library installation posed greater difficulty than on the workstation computer, because the Jetson is a more specific type of hardware. Although PyTorch and Torchvision were installable via `pip`, these versions caused errors. One option was to compile the libraries from source, but a more efficient and reliable solution involved using a pre-built Python wheel provided by NVIDIA [42].

Another issue was memory availability. Some models initially failed to run due to insufficient memory on the Jetson. With the removal of some unnecessary overhead in the code and terminating non-essential background processes running on the Jetson, all the models except SPIdepth were successfully run.

## 3.4 Converting to ONNX

In principle, converting the models to ONNX format is a relatively straightforward process, comprising of these steps:

1. Load and initialise the model.
2. Set it to evaluation mode.
3. Create a dummy input tensor or use a real input.
4. Call the `torch.onnx.export` function to perform the conversion and save the converted model.

In practice, however, this process required some additional effort, particularly for Depth Anything V2, SPIdepth, and Depth Pro. In these cases, it was necessary to inspect and modify the model code. To locate the point of inference and to set up the export procedure.

In contrast, Metric3Dv2 included a dedicated ONNX export script, simplifying the process significantly.

Adjusting the input resolution of all the models to match $1280 \times 720$ was the main challenge in the conversion process. This step was critical because while both ONNX and TRT support dynamic input resolutions, certain optimisation steps in TRT require fixed input and output dimensions to achieve optimal performance.

For SPIdepth and Depth Pro, the input resolution was discovered to be fixed. So they were exported with their default resolutions. Depth Anything V2 was exported with a resolution of $1274 \times 714$, as it is the closest multiple of 14 to the target resolution. Since Metric3Dv2 had no resolution limitations, it was exported with the desired target resolution.

## 3.5 Testing framework

One of the goals of this thesis is to develop a framework for testing MDE models. To achieve this goal, a pipeline-based approach was adopted. The pipeline was designed to support the following functionality:

- Load a model from a specified path.
- Retrieve images from all subfolders from a given path.
- Pre-process images before inference.
- Perform inference on the pre-processed images.
- Post-process the output into human-interpretable results.
- Log the time taken for inference and pre-processing.

- Record hardware utilisation metrics (e.g., usage percentage, power consumption).

The specifics of pre-processing and post-processing vary between PyTorch and ONNX Runtime, and are therefore described in subsections 3.5.2 and 3.5.3. The pipeline was implemented to support both backends, enabling users to run inference using either PyTorch or ONNX Runtime seamlessly. Using the pipeline is the overall process of preparing and evaluating a model. The complete workflow leading up to using a model with the pipeline is visualised in Figure 3.1.

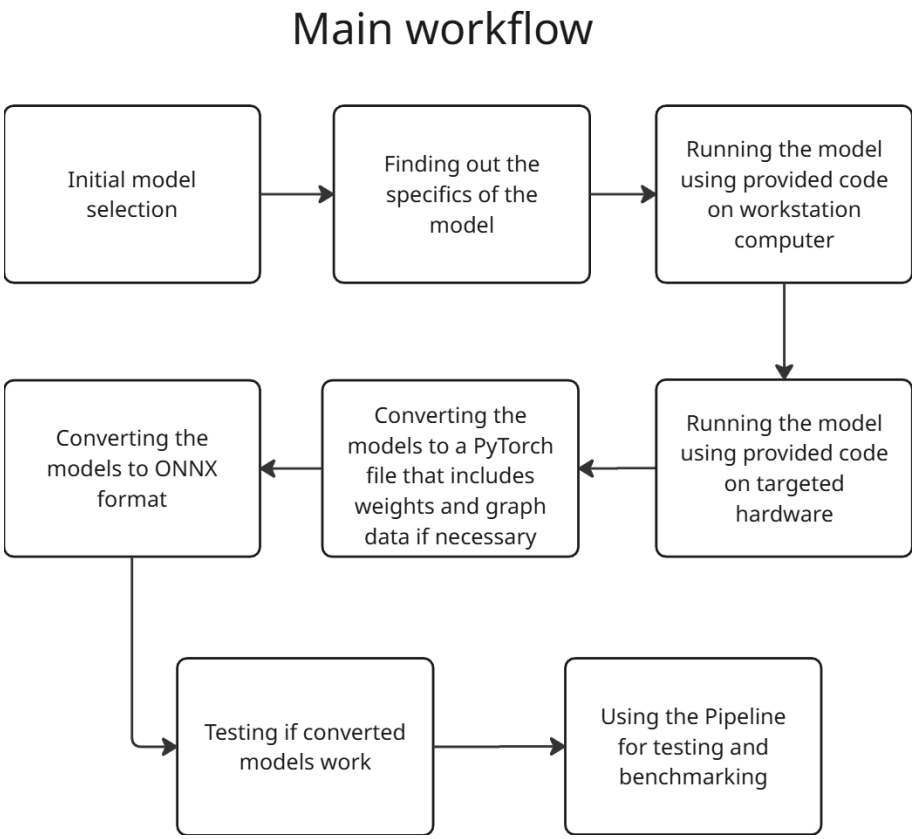

Figure 3.1: Flowchart of general workflow

### 3.5.1 Structure

Before implementation, a general structure for the pipeline was designed, based on the requirements established above. This structure is illustrated in Figure 3.2.

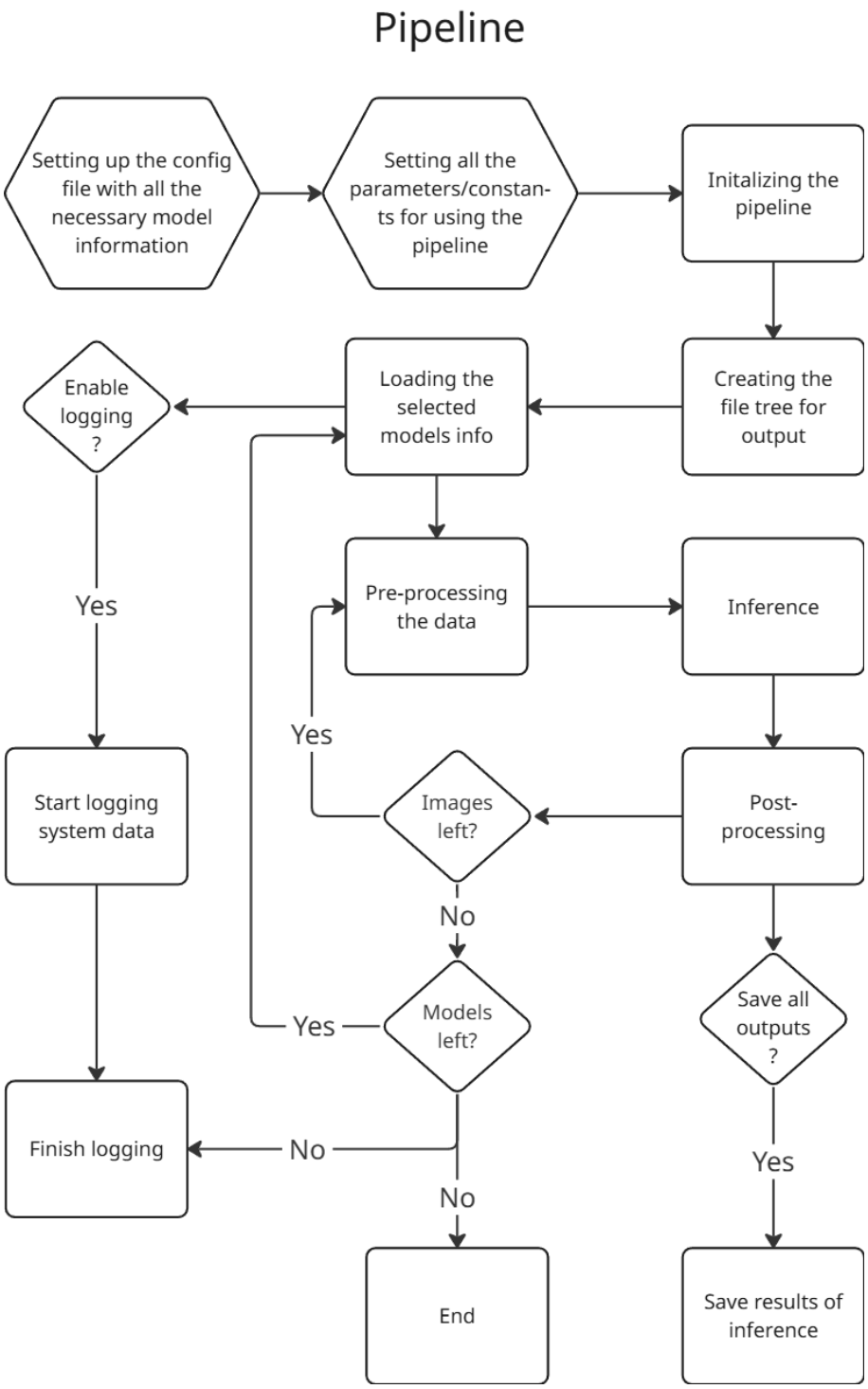


Figure 3.2: Flowchart of the pipeline

To facilitate the addition of future models, a general configuration file format was utilised. This file stores all the necessary metadata required to use each model with the pipeline, including:

- File paths to all available weights for each model.
- Input resolution and whether it is variable, only applies to PyTorch.
- Output type, metric or relative.

- Whether output inversion is required during post-processing.

The pipeline is initialised by creating a `Pipeline` object, with the path to the configuration file as the sole required argument. Subsequently, paths to the input images and output directory are specified. If the output folders do not exist, they are created automatically. A model can then be loaded by providing its name and weight to be used, after which testing may begin. A method, `test_all`, for testing all the available models in the configuration file was also implemented.

During testing, inference and pre-processing times were consistently logged. Hardware monitoring can be optionally enabled through the testing method. Additionally, the option to save all output results in a human-readable format is available. If this option is disabled, only every fifth output is saved. After testing for the selected model is finished, logging is stopped, and the data is saved. Upon completion of testing for a model, logs are saved. When using the `test_all` method, new logs are automatically created for the next model, if there is one.

### 3.5.2 PyTorch compatibility

Ensuring PyTorch compatibility required that the pipeline could load and run models using provided weight files and a defined input resolution. To eliminate the need for multiple files to load a model, the weights and model structure were stored in a single file, which could be loaded using PyTorch's native functions rather than the custom functions found in the original model repositories.

This approach generally succeeded. However, since the models were built using functions created by the creators of the models. Trying to load the models using PyTorch's native functions led to errors. These errors were caused by missing functions and/or variables that were defined in the provided code. To address this, the required files were moved into separate folders for each of the models, and these folders were placed into the same folder as the pipeline itself.

Pre-processing for PyTorch involved resizing the input image to a defined resolution. If the model supported variable input, the resolution could be specified by the user. Otherwise, the default resolution from the configuration file was used. The image was then converted to FP32 or FP16 datatype, depending on what datatype the model was set to use. After which it was transformed into a PyTorch tensor, and moved to the same device as the model, usually the GPU.

Post-processing began by moving the output tensor to the CPU and converting it to a NumPy array. This was followed by converting the array to INT8 format and applying post-processing functions to generate a human-interpretable output, which was then saved.

### 3.5.3 ONNX Runtime compatibility

Achieving compatibility with ONNX Runtime, while retaining support for PyTorch, presented a challenge. Since the two frameworks share little functional overlap, it was necessary to implement distinct code paths for operations depending on the selected backend.

Unlike PyTorch, loading the models using ONNX Runtime did not require any of the functions made by the models creators. However, after using the converted ONNX versions of the models, it became apparent that certain models exhibited discrepancies in their outputs compared to their PyTorch counterparts. This indicated that parts of the original post-processing functions were not captured during conversion. To resolve this, the original codebases were examined to identify the missing operations. In most cases, a small set of standardised functions could be applied to restore the expected behaviour.

One notable exception was Metric3Dv2, whose post-processing differed substantially from the other models. A dedicated exception was implemented in the pipeline to handle this and produce appropriate output.

Pre-processing and post-processing for ONNX Runtime was largely similar to that of PyTorch. The main difference was that inputs and outputs did not need to be converted to or from PyTorch tensors. NumPy arrays were sufficient inputs for inference.

## 3.6 Benchmarking metrics

It was previously established that the pipeline should be capable of logging the time taken for inference and pre-processing, as well as hardware utilisation. The following metrics were measured:

- **Pre-processing time per image (in seconds)**: This includes operations discussed in subsection 3.5.3 and subsection 3.5.2.
- **Inference time per image (in seconds)**: This refers to how long it takes to complete inference for the respective backend.
- **Hardware metrics**:
  - CPU utilisation (percentage)
  - GPU utilisation (percentage)
  - RAM utilisation (in MB)
  - power consumption of the entire Jetson (in mW)
- **Timestamps** associated with each of the above measurements

Recording the time taken for pre-processing and inference helps estimate the computational cost, thereby aiding in model selection. Since the speed of inference and preceding processes are a key factor in the selection process.

CPU and GPU usage provide insight into understanding the computational load the system is under during the different processes. RAM usage provides insight into the model's memory utilisation and was a strong indicator for out-of-memory errors when testing larger models.

Power consumption information is crucial because KuupKulgur is currently a battery-powered device. Although solar panels will be added in the future, power generation will still be limited.

Timestamps for each measurement are crucial in helping synchronise information obtained from different measurements. All hardware information gathered aids in the understanding of benefits, disadvantages and limitations when developing or selecting models in the future.

All the timing information was collected using Python's built-in `time` module. A snapshot of the current time was taken immediately before starting a process and after finishing that process. The time elapsed during the process was calculated by subtracting the first time from the second. This was performed independently for each process requiring timing. Hardware information as listed above, with timestamps, was gathered using NVIDIA's tegrastats utility every 100ms. For timing measurements, the timestamp was taken when timing was started, to aid in synchronising with hardware information collected during a process.

## 3.7 Testing with pipeline

All selected models and configurations were evaluated using the developed benchmarking framework. Tests were conducted using a dataset consisting of 100 images, all of which were of a resolution of $1280 \times 720$. The configurations for each model tested were:

- PyTorch as the inference backend, using:
  - FP32 weights.
  - FP16 weights.
- ONNX Runtime with TRT as the execution provider, using both FP32 and FP16 weights with these optimisation levels:
  - Level 0
  - Level 1
  - Level 2
  - Level 3
  - Level 4
  - Level 5

Using FP16 weights for PyTorch required loading the model using `.half()` method and applying the same method to the input tensor. For ONNX Runtime, FP16 weights were enabled by setting the `trt_fp16_enable` variable. Optimisation level was controlled using the `trt_builder_optimization_level` variable. The higher the optimisation level, the more time- and memory-consuming strategies were used. It is unknown what optimisation strategies were used with each level.

All tests were conducted on the Jetson device mentioned in section 2.4. However, not all of the configurations could be tested. In a majority of the cases, the Python process was killed by the system. In the rest of the cases, some other error occurred. A full overview and possible explanations are given in chapter 5 and in tables 5.1 and 5.2.

Changing the configuration being tested required making a change in the configuration file, with the rest of the code remaining unchanged, when using the implemented `test_all` method. The configuration file used during testing is observable in Appendix A.

# 4 Results and Analysis

## 4.1 Time related results

### 4.1.1 Total times overview

An overview of the combined time spent on pre-processing and inference across all tested configurations using FP32 precision weights is presented in Table 4.1 and for FP16 in Table 4.2. In both tables, number of parameters (NoP) signifies how many parameters a given version of the model has, the M stands for million, and the number after TRT means the optimisation level used. If a model is missing from the table or shows N/A, it indicates that the model or configuration could not be tested. The best times for each model version have been highlighted in green, and the worst in red. If a model only has one time, it is treated as the worst. The overall best and worst times have a brighter colouring and bold text.

Table 4.1: Average time spent on pre-processes and inference per image with FP32 precision weights

| Model | NoP | TRT 0 | TRT 1 | TRT 2 | TRT 3 | TRT 4 | TRT 5 | PyTorch |
|---|---|---|---|---|---|---|---|---|
| Depth Anything V2 relative | 24.8M | 2.3138 | 1.5826 | 1.5816 | 1.5821 | 1.5828 | **1.0188** | 1.2207 |
| | 97.5M | 5.2420 | N/A | N/A | N/A | N/A | N/A | 2.7936 |
| | 335.3M | N/A | N/A | N/A | N/A | N/A | N/A | 8.2659 |
| Depth Anything V2 metric | 24.8M | 2.3128 | 1.5768 | 1.5799 | 1.5781 | 1.5817 | 1.0199 | 1.2166 |
| | 97.5M | 5.2456 | N/A | N/A | N/A | N/A | N/A | 2.7992 |
| | 335.3M | N/A | N/A | N/A | N/A | N/A | N/A | **8.2748** |
| Metric3Dv2 | 37.5M | 3.0457 | 2.1484 | 2.1437 | 2.1349 | 2.5873 | 2.5874 | 2.0871 |
| SPIdepth | 722.9M | N/A | N/A | N/A | N/A | N/A | N/A | 2.3549 |

When using FP32 precision weights, average PyTorch times were, in most cases, the best times out of all the tested configurations for that specific model. Except Depth Anything V2's smallest NoP version for both metric and relative. Even then, TRT times were better only when using the highest optimisation level.

When using FP16 precision weights for the models, the trend changes, as can be observed in Table 4.2. In this case, PyTorch is the worst on average across all the different models. The best times using TRT weren't when using the highest optimisation level, but rather when using a level between 1 and 3.

Table 4.2: Average time spent on pre-processes and inference per image with FP16 precision weights

| Model | NoP | TRT 0 | TRT 1 | TRT 2 | TRT 3 | TRT 4 | TRT 5 | PyTorch |
|---|---|---|---|---|---|---|---|---|
| Depth Anything V2 relative | 24.8M | 0.3461 | 0.2815 | **0.2756** | 0.2813 | 0.2862 | 0.2869 | 0.6331 |
| | 97.5M | 0.9026 | 0.5127 | 0.5148 | 0.5152 | 0.5134 | N/A | 1.2902 |
| | 335.3M | N/A | N/A | N/A | N/A | N/A | N/A | 3.4433 |
| Depth Anything V2 metric | 24.8M | 0.3575 | 0.2895 | 0.2898 | 0.2769 | 0.2848 | 0.2828 | 0.6330 |
| | 97.5M | 0.8965 | 0.5201 | 0.5240 | 0.5172 | 0.5140 | N/A | 1.2930 |
| | 335.3M | N/A | N/A | N/A | N/A | N/A | N/A | **3.4469** |
| Metric3Dv2 | 37.5M | 0.5481 | 0.4308 | 0.4340 | 0.4261 | 0.4924 | 0.4977 | N/A |
| SPIdepth | 722.9M | N/A | N/A | N/A | N/A | N/A | N/A | 0.5780 |

### 4.1.2 Optimisation time

Optimisation times for Depth Anything V2's smallest NoP relative model can be observed in Figure 4.1. Increasing the optimisation level caused a varying increase in time, and was similar for some optimisation levels. All of the FP16 precision weight versions took longer than their FP32 counterparts, and interestingly, it was around double the time taken. The longest optimisation (FP16 level 5) took 24 minutes and 17 seconds. Results for other models can be observed in Appendix B.

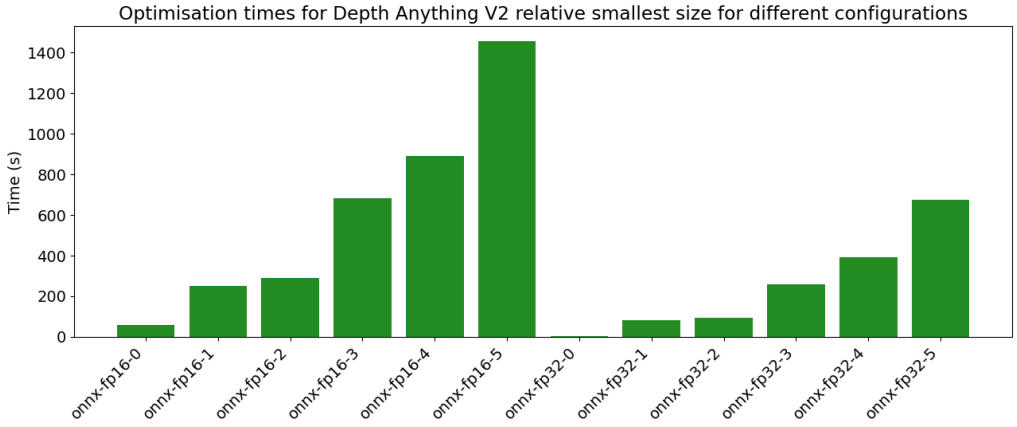


Figure 4.1: Time spent on optimisation for Depth Anything V2's smallest NoP relative model

### 4.1.3 Pre-processing time

Average pre-processing times for all tested configurations of Depth Anything V2's smallest NoP relative model with standard deviation can be observed in Figure 4.2. TRT times were consistent across all the tests. However, PyTorch times were significantly longer and more inconsistent. Results for other models can be observed in **??**.

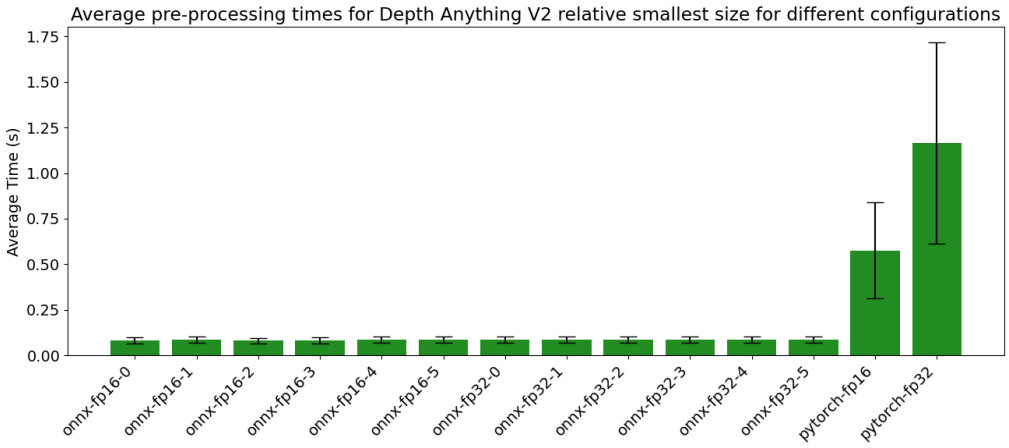


Figure 4.2: Total time spent on pre-processing for Depth Anything V2's smallest NoP relative model

### 4.1.4 Inference time

Inference times for the same model were very consistent for each configuration, as can be observed from the standard deviation in Figure 4.3. For inference, PyTorch had smaller times than TRT. The average inference times across all optimisation levels, when using FP16 precision weights, were significantly more similar compared to FP32. A large improvement can be observed when using level 1 compared to level 0 and level 5 compared to level 4. Results for other models can be observed in **??**.

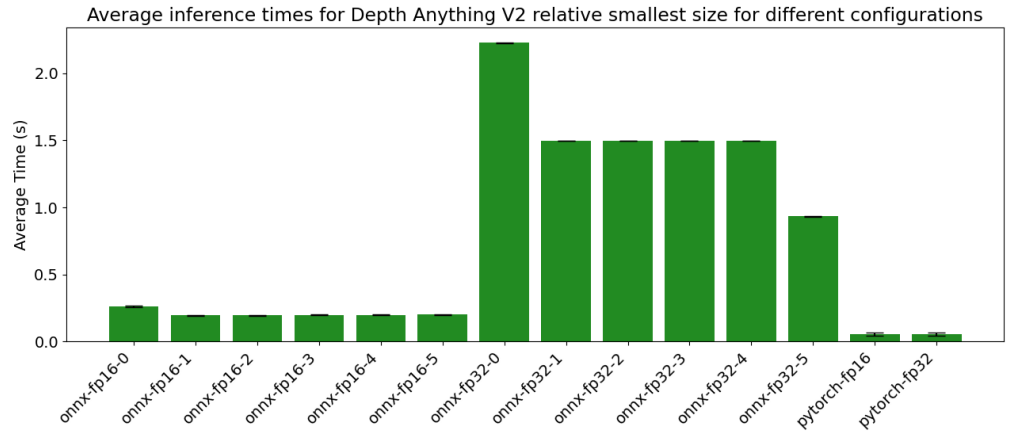


Figure 4.3: Total time spent on inference for Depth Anything V2's smallest NoP relative model

## 4.2 ONNX Runtime hardware usage results

The results presented under this section are for the best-performing configuration across all models, time-wise: Depth Anything V2's relative model with the smallest NoP weight using optimisation level 2 and FP16 weight values. The trends described in each subsection apply to all ONNX models that were tested. Differences compared to PyTorch are presented in chapter 5.

### 4.2.1 CPU usage

Utilisation for each CPU core during the test can be observed in Figure 4.4. During the test, there was only one core at a time that had a significantly higher utilisation, compared to the rest. The core with the higher utilisation changed during the test. The core with the higher utilisation percentage was between 40-100% utilisation during the test. The other cores had a utilisation of around 10-20%. With occasional spikes to around 40% from one core, that wasn't the highest usage core at that given moment.

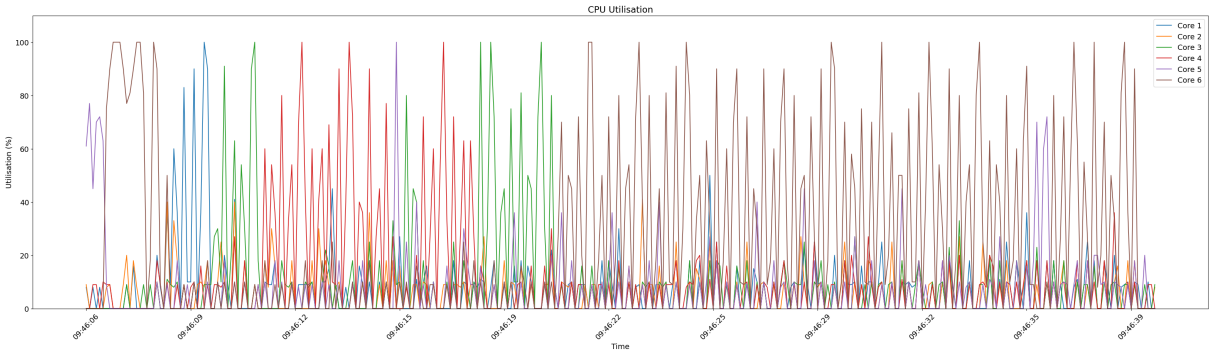


Figure 4.4: CPU usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's smallest NoP relative model

### 4.2.2 GPU usage

GPU utilisation percentage, which can be observed in Figure 4.5, fluctuated between 0% and 100% throughout the test, with the 100% peaks indicating when inference was happening and 0% the downtime between GPU utilising operations. The first 100% peak indicates the start of the first inference during the test.

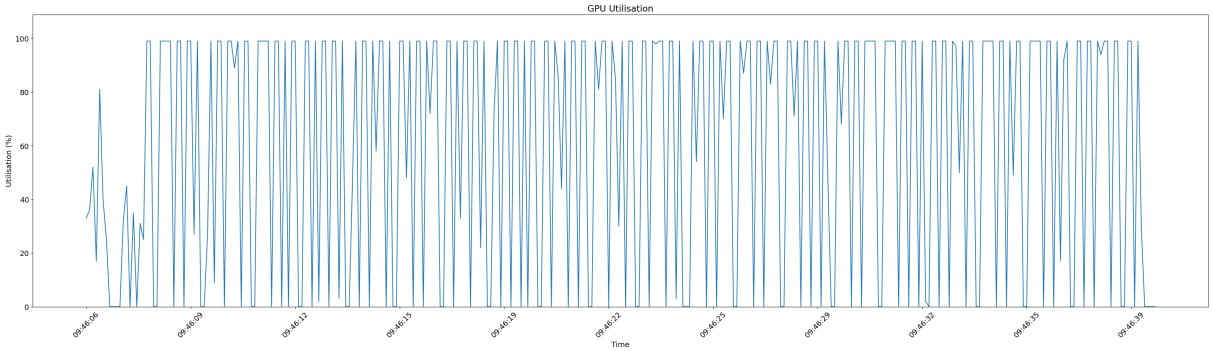


Figure 4.5: GPU usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's smallest NoP relative model

### 4.2.3 RAM usage

During the test, there were small changes before the first inference had started, which can be observed in Figure 4.6. However, once the first inference had started, the amount of memory used was almost unchanged, with the value staying at around 5050 MB.

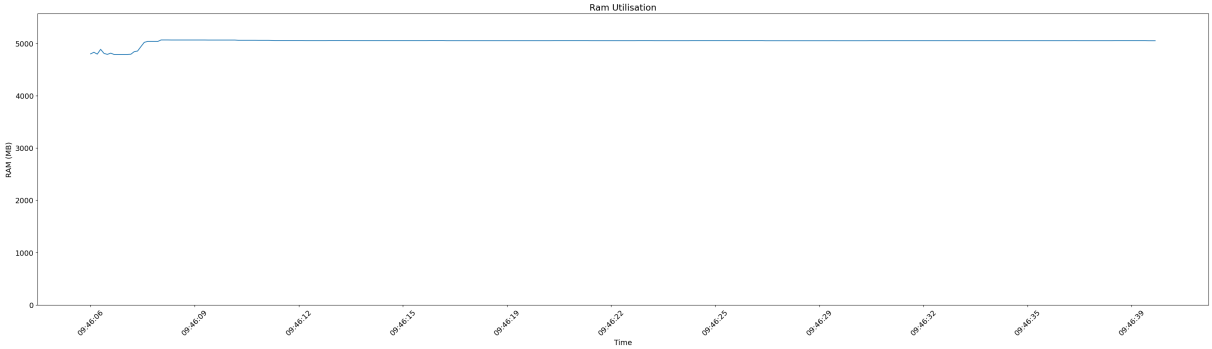


Figure 4.6: RAM usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's smallest NoP relative model

### 4.2.4 Power consumption

Power usage by the Jetson during the test can be observed in Figure 4.7. Power usage peaked at 10430 mW. The CPU and GPU combined used 4160 mW of that peak. Power draw during testing is inconsistent and changes semi-periodically. Each dip signifies the completion of inference on an image. The larger dips happen because every fifth image was saved. The value during inference is between 10430 mW and 8540 mW.

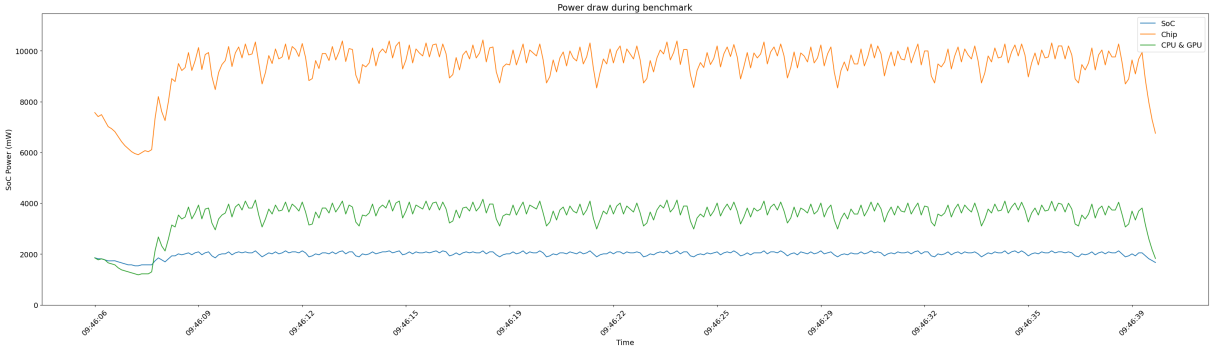


Figure 4.7: Power consumption during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's smallest NoP relative model

## 4.3 PyTorch hardware usage results

The results presented in this section are for Depth Anything V2 relative versions smallest NoP model using FP16 weights with PyTorch as the inference backend. Although this configuration achieved the second-lowest inference time among the PyTorch setups, it was chosen for presentation to ensure consistency with the TRT results shown in section 4.2, which are based on the same base model and weight values. This allows for a more meaningful comparison between PyTorch and TRT performance. The trends observed here are representative of other PyTorch configurations tested.

### 4.3.1 CPU usage

Utilisation for each CPU core during the test can be observed in Figure 4.8. The results follow the same patterns that are described in subsection 4.2.1.

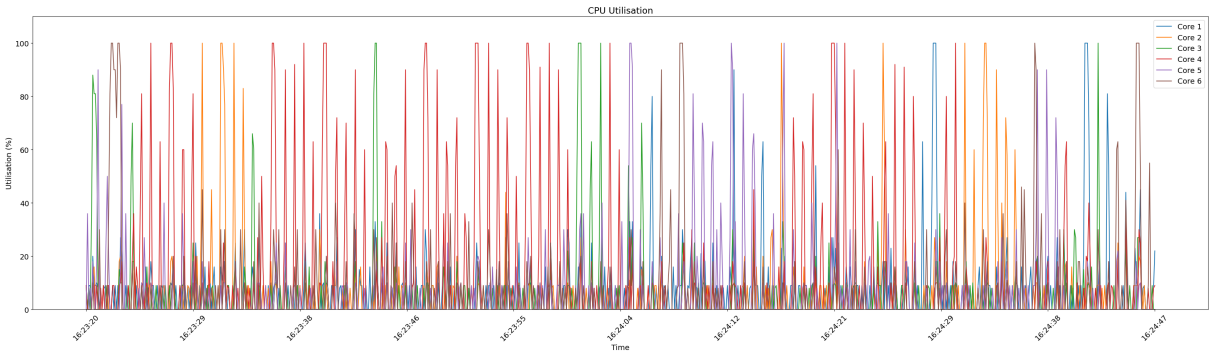


Figure 4.8: CPU usage during PyTorch testing using FP16 weights for Depth Anything V2's smallest NoP relative model

### 4.3.2 GPU usage

GPU utilisation during testing is shown in Figure 4.9. The value jumps between 0% and 100%, with the occasional dip from 100% to around 90%. "Unlike the pattern observed in Figure 4.5, dips in utilisation were considerably less frequent. The dips happen when saving a result occurs, instead of every inference.

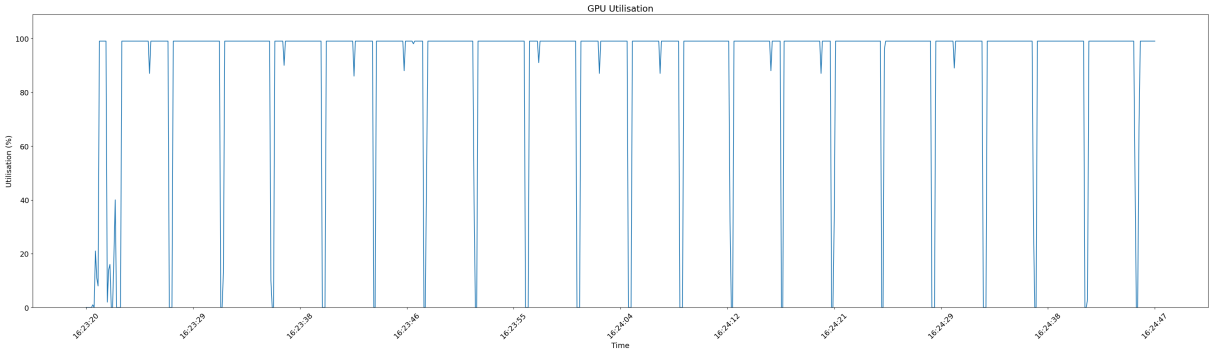


Figure 4.9: GPU usage during PyTorch testing using FP16 weights for Depth Anything V2's smallest NoP relative model

### 4.3.3 RAM usage

RAM usage followed the same trend as described in subsection 4.2.3. However, the value that it stayed around was 3300 MB, which is 1750 MB less than the value for TRT.

### 4.3.4 Power consumption

Power usage data collected during the benchmark is visualised in Figure 4.10. During inference, power consumption peaks at around 11000 mW. The combined usage of the CPU and GPU accounts for approximately 4280 mW of that peak. Unlike the TRT tests, PyTorch power

consumption doesn't dip as often. A significant dip occurred only when saving an image, rather than after every inference. When the dip does happen, it is lower when compared to the one in subsection 4.2.4, the value being 8240 mW compared to 8540 mW.

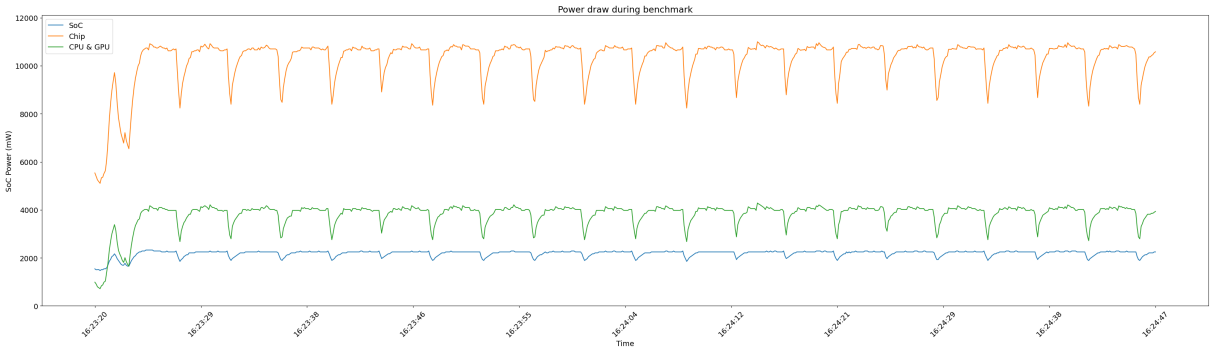


Figure 4.10: Power consumption during PyTorch testing using FP16 weights for Depth Anything V2's smallest NoP relative model

# 5 Discussion

## 5.1 Testing anomalies

### 5.1.1 PyTorch

An overview of combinations that could be tested using PyTorch as the backend is shown in Table 5.1.

Table 5.1: Overview of combinations that could be tested using PyTorch

| Model | NoP | FP32 | FP16 |
|---|---|---|---|
| Depth Anything V2 relative | 24.8M | ✓ | ✓ |
| | 97.5M | ✓ | ✓ |
| | 335.3M | ✓ | ✓ |
| Depth Anything V2 metric | 24.8M | ✓ | ✓ |
| | 97.5M | ✓ | ✓ |
| | 335.3M | ✓ | ✓ |
| Depth Pro | 951.9M | **X** | **X** |
| Metric3Dv2 | 37.5M | ✓ | **X** |
| | 441.9M | **X** | **X** |
| SPIdepth | 722.9M | ✓ | ✓ |

As shown in Table 5.1, the models that could be tested differ from those that were runnable during initial testing on the Jetson. Depth Pro could not be tested because the process was killed, due to the system running out of memory. The reason why Metric3Dv2's large NoP version could not run is most likely due to the additional memory required when using a larger input resolution, compared to the initial testing, where the default resolution was used. During initial testing, all the models were tested using the provided default resolutions and resolutions close to them, to assess if the input could be varied. The resolution during initial testing was 512×1088, whereas during testing with the pipeline, it was 720×1280.

Additionally, Table 5.1 shows that Metric3Dv2 could not be tested with FP16 weights. This was caused by the fact that some operations in the structure of the model were defined to output FP32 values. This caused an error because the inputs expected FP16 values.

During analysis, it was discovered that PyTorch's pre-processing took significantly longer compared to ONNX Runtime, which can be seen in Figure 4.2. Although a longer pre-processing time was anticipated, since a conversion from NumPy array to tensor had to be made, the magnitude of the difference was unexpected. A similar observation applies to inference times, but in the opposite direction—PyTorch inference was significantly faster than expected. Additional

testing was done to help understand this behaviour. It was deduced that the likely cause of this is when using PyTorch's `.to()` command, inference starts immediately. Instead of using the appropriate command to start inference. The behaviour described was true for all models except Metric3Dv2. The reason for this was not discovered.

Additionally, Metric3Dv2 TRT optimisation times were not obtained, since logging started after optimisation on that model was finished instead of before. The code used was the same as for all other tests, the reason for this behaviour is undiscovered.

### 5.1.2 ONNX Runtime

An overview of combinations that could be tested using ONNX Runtime as the backend is shown in Table 5.2.

Table 5.2: Combinations of precision and optimisation level tested using ONNX Runtime

| **Model** | **NoP** | **FP32** | | | | | | **FP16** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Opt level | | 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 | 4 | 5 |
| Depth Anything V2 relative | 24.8M | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 97.5M | ✓ | **X** | **X** | **X** | **X** | **X** | ✓ | ✓ | ✓ | ✓ | ✓ | **X** |
| | 335.3M | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** |
| Depth Anything V2 metric | 24.8M | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 97.5M | ✓ | **X** | **X** | **X** | **X** | **X** | ✓ | ✓ | ✓ | ✓ | ✓ | **X** |
| | 335.3M | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** |
| Depth Pro | 951.9M | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** |
| Metric3Dv2 | 37.5M | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 441.9M | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** |
| SPIdepth | 722.9M | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** |

Depth Anything V2's medium NoP version using FP32 precision weights could not be tested beyond optimisation level 0, because it contained a function that TRT wanted to optimise, but didn't have an equivalent to. This applies for both the metric and relative versions. The same error, however, did not occur using FP16 values. Using optimisation level 5 with FP16 values caused the process to get killed, due to a lack of memory.

When using levels 4 or 5 of optimisation with FP16 precision weights, some strategies could not be used due to a lack of memory. The exact number and what strategies could not be used are unknown. This could also be the reason why optimisation levels 1-3 had the best performances.

All other models that could not be tested were due to a lack of memory, which caused the process to be killed.

## 5.2 Hardware utilisation

In general, PyTorch used more resources than ONNX Runtime to achieve comparable results. In almost all aspects, PyTorch exhibited higher CPU and GPU utilisation and consequently, higher power consumption. This is observable when comparing Figure 4.4 and Figure 4.8 for CPU utilisation and Figure 4.5 and Figure 4.9 for GPU utilisation. The difference is especially distinguishable for GPU utilisation. The difference is caused by the fact that PyTorch does

constant optimisation during the test instead of doing it once, as TRT does. This results in the aforementioned resource usage difference. For an enhanced comparison, graphs for CPU and GPU utilisation for the medium version of the same configurations of Depth Anything V2, as shown in chapter 4 are in Appendix C. The only aspect where PyTorch utilised less than ONNX Runtime was RAM. This could be attributed to the configuration used when testing with TRT. However, the reason for this is unknown and would require further testing.

# 6 Conclusion

The goal of this thesis has been to design and implement a testing framework for evaluating the performance of MDE models on targeted hardware, specifically the NVIDIA Jetson product family. This was done in the context of the KuupKulgur lunar rover project.

The developed framework supports inference using PyTorch and ONNX Runtime. It was made with the integration of additional models with minimal effort in mind, allowing streamlined future testing. The benchmarking framework also provides detailed logging of hardware metrics and times taken for different processes, which provides a comprehensive view of model behaviour and performance.

The framework provides access to configurable optimisation using TRT as the execution provider with ONNX Runtime. Additionally, using FP32 and FP16 weights is supported by both backends used.

Once optimisation with TRT was applied, all models showed improved inference times over PyTorch when using FP16 precision weights. When using FP32 values, PyTorch outperformed TRT, with a few exceptions.

PyTorch generally required more system resources than ONNX Runtime to achieve comparable inference results. Specifically, it exhibited higher CPU and GPU utilisation and consequently greater power consumption. This behaviour is attributed to PyTorch performing continuous optimisation during inference, in contrast to TRT, which applies optimisations once during initialisation. The only metric where PyTorch demonstrated lower usage was RAM. However, the cause of this remains unclear and warrants further investigation.

An evaluation of all tested models was conducted, and the most promising candidate for use on KuupKulgur was Depth Anything V2's smallest NoP version using FP16 precision weights. This was the model with the lowest combined time for inference and pre-processing. However, this model was among the least accurate of the evaluated models.

Possible future improvements identified during this thesis include:

- Integration of additional backends to the framework to allow a wider range of models to be tested.
- Addition of methods to allow for INT8 weights to be used.
- Providing support for use with a wider range of targeted hardware besides NVIDIA Jetson.
- Further analysis and implementation of improved memory utilisation metrics to better understand its utilisation.

# Acknowledgements

I would like to express my sincere gratitude to my thesis supervisors, Quazi Saimoon Islam and Ric Dengel, for their continuous support, guidance, and insightful advice throughout the course of this project.

I would also like to thank my friends and fellow KuupKulgur team members for their constructive feedback and encouragement during the writing process.

*/ signed digitally /*

# Bibliography


[1] *Common European Research Classification Scheme (CERCS)*. [Online]. Available: https://tuit.ut.ee/sites/default/files/2023-01/Common%20European%20Research%20Classification%20Scheme.pdf.

[2] R. Pyle, *Space 2.0: How Private Spaceflight, a Resurgent NASA, and International Partners are Creating a New Space Age*, en. BenBella Books, Feb. 2019, Google-Books-ID: IxRbDwAAQBAJ, ISBN: 978-1-946885-74-6.

[3] S. Creech, J. Guidi, and D. Elburn, "Artemis: An Overview of NASA's Activities to Return Humans to the Moon," in *2022 IEEE Aerospace Conference (AERO)*, ISSN: 1095-323X, Mar. 2022, pp. 1–7. DOI: 10.1109/AERO53065.2022.9843277. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9843277 (visited on 05/02/2025).

[4] *Explore2040*. [Online]. Available: https://esamultimedia.esa.int/docs/HRE/Explore_2040.pdf.

[5] I. A. Nesnas, L. M. Fesq, and R. A. Volpe, "Autonomy for Space Robots: Past, Present, and Future," en, *Current Robotics Reports*, vol. 2, no. 3, pp. 251–263, Sep. 2021, ISSN: 2662-4087. DOI: 10.1007/s43154-021-00057-2. [Online]. Available: https://doi.org/10.1007/s43154-021-00057-2 (visited on 05/02/2025).

[6] *KuupKulgur – Tartu Observatory Space Exploration Group*, en-US. [Online]. Available: https://tospexgroup.space/projects/kuupkulgur/ (visited on 05/02/2025).

[7] A. Carrio, S. Vemprala, A. Ripoll, S. Saripalli, and P. Campoy, *Drone Detection Using Depth Maps*, arXiv:1808.00259 [cs], Aug. 2018. DOI: 10.48550/arXiv.1808.00259. [Online]. Available: http://arxiv.org/abs/1808.00259 (visited on 05/02/2025).

[8] A. Masoumian, H. A. Rashwan, J. Cristiano, M. S. Asif, and D. Puig, "Monocular Depth Estimation Using Deep Learning: A Review," en, *Sensors*, vol. 22, no. 14, p. 5353, Jan. 2022, Number: 14 Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 1424-8220. DOI: 10.3390/s22145353. [Online]. Available: https://www.mdpi.com/1424-8220/22/14/5353 (visited on 05/02/2025).

[9] *Perseverance Rover Components - NASA Science*, en-US, Section: Perseverance (Rover), Apr. 2024. [Online]. Available: https://science.nasa.gov/mission/mars-2020-perseverance/rover-components/ (visited on 05/03/2025).

[10] C. Wong, E. Yang, X.-T. Yan, and D. Gu, "Adaptive and intelligent navigation of autonomous planetary rovers — A survey," in *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, ISSN: 2471-769X, Jul. 2017, pp. 237–244. DOI: 10.1109/AHS.2017.8046384. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8046384 (visited on 05/03/2025).

[11] R. Welch, D. Limonadi, and R. Manning, "Systems engineering the Curiosity Rover: A retrospective," in *2013 8th International Conference on System of Systems Engineering*, Jun. 2013, pp. 70–75. DOI: 10.1109/SYSoSE.2013.6575245. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6575245 (visited on 05/03/2025).

[12] *Here's How AI Is Changing NASA's Mars Rover Science - NASA*, en-US, Section: Perseverance (Rover), Jul. 2024. [Online]. Available: https://www.nasa.gov/missions/mars-2020-perseverance/perseverance-rover/heres-how-ai-is-changing-nasas-mars-rover-science/ (visited on 05/03/2025).

[13] N. Abcouwer, S. Daftry, T. del Sesto, *et al.*, "Machine Learning Based Path Planning for Improved Rover Navigation," in *2021 IEEE Aerospace Conference (50100)*, ISSN: 1095-323X, Mar. 2021, pp. 1–9. DOI: 10.1109/AERO50100.2021.9438337. [Online]. Available: https://ieeexplore.ieee.org/document/9438337 (visited on 05/03/2025).

[14] Y. Ming, X. Meng, C. Fan, and H. Yu, "Deep learning for monocular depth estimation: A review," *Neurocomputing*, vol. 438, pp. 14–33, May 2021, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2020.12.089. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231220320014 (visited on 05/03/2025).

[15] Unsplash, *David Köhler (@davidkhlr) — Unsplash Photo Community*, en-US. [Online]. Available: https://unsplash.com/@davidkhlr (visited on 05/20/2025).

[16] Y. Almalioglu, A. Santamaria-Navarro, B. Morrell, and A.-A. Agha-Mohammadi, "Unsupervised Deep Persistent Monocular Visual Odometry and Depth Estimation in Extreme Environments," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, ISSN: 2153-0866, Sep. 2021, pp. 3534–3541. DOI: 10.1109/IROS51168.2021.9636555. [Online]. Available: https://ieeexplore.ieee.org/document/9636555 (visited on 05/03/2025).

[17] K. Van Hecke, G. C. De Croon, D. Hennes, T. P. Setterfield, A. Saenz-Otero, and D. Izzo, "Self-supervised learning as an enabling technology for future space exploration robots: ISS experiments on monocular distance learning," en, *Acta Astronautica*, vol. 140, pp. 1–9, Nov. 2017, ISSN: 00945765. DOI: 10.1016/j.actaastro.2017.07.038. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0094576517302862 (visited on 05/03/2025).

[18] *What are SmallSats and CubeSats? - NASA*, en-US, Section: Ames Research Center. [Online]. Available: https://www.nasa.gov/what-are-smallsats-and-cubesats/ (visited on 05/03/2025).

[19] *Jetson Modules, Support, Ecosystem, and Lineup*, en. [Online]. Available: https://developer.nvidia.com/embedded/jetson-modules (visited on 05/03/2025).

[20] W. S. Slater, N. P. Tiwari, T. M. Lovelly, and J. K. Mee, "Total Ionizing Dose Radiation Testing of NVIDIA Jetson Nano GPUs," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, ISSN: 2643-1971, Sep. 2020, pp. 1–3. DOI: 10.1109/HPEC43674.2020.9286222. [Online]. Available: https://ieeexplore.ieee.org/document/9286222 (visited on 05/03/2025).

[21] M. A. Felix, W. S. Slater, D. C. Landauer, R. E. Pinson, and B. B. Rutherford, "Total Ionizing Dose Radiation Testing of NVIDIA Jetson Orin NX System on Module," in *2024 IEEE Space Computing Conference (SCC)*, Jul. 2024, pp. 116–121. DOI: 10.1109/SCC61854.2024.00019. [Online]. Available: https://ieeexplore.ieee.org/document/10794965 (visited on 05/03/2025).

[22] I. Rodriguez-Ferrandez, L. Kosmidis, M. Tali, D. Steenari, A. Hands, and C. Bélanger-Champagne, "Proton Evaluation of Single Event Effects in the NVIDIA GPU Orin SoM: Understanding Radiation Vulnerabilities Beyond the SoC," en, in *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Rennes, France: IEEE, Jul. 2024, pp. 1–7, ISBN: 9798350370553. DOI: 10.1109/IOLTS60994.2024.10616076. [Online]. Available: https://ieeexplore.ieee.org/document/10616076/ (visited on 05/19/2025).

[23] M. Abadi, A. Agarwal, P. Barham, *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," en,

[24] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in PyTorch," en, Oct. 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ (visited on 05/03/2025).

[25] J. Bai, F. Lu, K. Zhang, *et al.*, *Onnx: Open neural network exchange*, https://github.com/onnx/onnx, 2019.

[26] *ONNX Runtime*, en-US. [Online]. Available: https://onnxruntime.ai/docs/ (visited on 05/03/2025).

[27] *On-Device Training*, en-US. [Online]. Available: https://onnxruntime.ai/docs/get-started/training-on-device.html (visited on 05/03/2025).

[28] *Execution Providers*, en-US. [Online]. Available: https://onnxruntime.ai/docs/execution-providers/ (visited on 05/03/2025).

[29] *CUDA Zone - Library of Resources*, en. [Online]. Available: https://developer.nvidia.com/cuda-zone (visited on 05/04/2025).

[30] *CUDA-X*, en. [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries (visited on 05/04/2025).

[31] *CUDA Deep Neural Network*, en. [Online]. Available: https://developer.nvidia.com/cudnn (visited on 05/04/2025).

[32] Y. Zhou and K. Yang, "Exploring TensorRT to Improve Real-Time Inference for Deep Learning," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, Dec. 2022, pp. 2011–2018. DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00299. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10074837 (visited on 05/04/2025).

[33] O. Shafi, C. Rai, R. Sen, and G. Ananthanarayanan, "Demystifying TensorRT: Characterizing Neural Network Inference Engine on Nvidia Edge Devices," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, Nov. 2021, pp. 226–237. DOI: 10.1109/IISWC53511.2021.00030. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9668285 (visited on 05/04/2025).

[34] G. P. B, R. Marion Lincy G, A. Rishekeeshan, and Deekshitha, "Accelerating Native Inference Model Performance in Edge Devices using TensorRT," in *2024 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, ISSN: 2769-5565, May 2024, pp. 1–7. DOI: 10.1109/RAICS61201.2024.10690032. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10690032 (visited on 05/04/2025).

[35] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The KITTI dataset," EN, *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, Sep. 2013, Publisher: SAGE Publications Ltd STM, ISSN: 0278-3649. DOI: 10.1177/0278364913491297. [Online]. Available: https://doi.org/10.1177/0278364913491297 (visited on 04/22/2025).

[36] *Papers with Code - The latest in Machine Learning*, en. [Online]. Available: https://paperswithcode.com/ (visited on 04/22/2025).

[37] L. Yang, B. Kang, Z. Huang, *et al.*, *Depth Anything V2*, arXiv:2406.09414 [cs], Oct. 2024. DOI: 10.48550/arXiv.2406.09414. [Online]. Available: http://arxiv.org/abs/2406.09414 (visited on 05/04/2025).

[38] M. Lavreniuk, *SPIdepth: Strengthened Pose Information for Self-supervised Monocular Depth Estimation*, arXiv:2404.12501 [cs], Sep. 2024. DOI: 10.48550/arXiv.2404.12501. [Online]. Available: http://arxiv.org/abs/2404.12501 (visited on 05/04/2025).

[39] M. Hu, W. Yin, C. Zhang, *et al.*, "Metric3Dv2: A Versatile Monocular Geometric Foundation Model for Zero-shot Metric Depth and Surface Normal Estimation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 12, pp. 10 579–10 596, Dec. 2024, arXiv:2404.15506 [cs], ISSN: 0162-8828, 2160-9292, 1939-3539. DOI: 10.1109/TPAMI.2024.3444912. [Online]. Available: http://arxiv.org/abs/2404.15506 (visited on 05/04/2025).

[40] A. Bochkovskii, A. Delaunoy, H. Germain, *et al.*, *Depth Pro: Sharp Monocular Metric Depth in Less Than a Second*, arXiv:2410.02073 [cs], Apr. 2025. DOI: 10.48550/arXiv.2410.02073. [Online]. Available: http://arxiv.org/abs/2410.02073 (visited on 05/04/2025).

[41] S. Chen, H. Guo, S. Zhu, *et al.*, *Video Depth Anything: Consistent Depth Estimation for Super-Long Videos*, arXiv:2501.12375 [cs], Jan. 2025. DOI: 10.48550/arXiv.2501.12375. [Online]. Available: http://arxiv.org/abs/2501.12375 (visited on 05/04/2025).

[42] *PyTorch for Jetson - Jetson & Embedded Systems / Announcements*, en-GB, Section: Robotics & Edge Computing, Mar. 2019. [Online]. Available: https://forums.developer.nvidia.com/t/pytorch-for-jetson/72048 (visited on 04/22/2025).

# A Configuration file

```yaml
models:
  torch:
    Depth-Anything-V2:
      weights:
        default: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_s.pt
        s: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_s.pt
        b: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_b.pt
        l: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_l.pt
      input_size: [1274, 714] # [width, height]
      veriable_input_size: true
      metric_output: false
      inverse_depth: true
      input_format: "image"


    Metric3D-V2:
      weights:
        default: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/metric_3d_s.pt
        s: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/metric_3d_s.pt
        l: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/metric_3d_l.pt
      input_size: [720, 1280] # !!![height, with]!!!
      veriable_input_size: true
      metric_output: false
      inverse_depth: false
      input_format: "image"

    Depth-Anything-V2-Metric:
      weights:
        default: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_metric_s.pt
        s: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_metric_s.pt
```

```yaml
        b: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_metric_b.pt
        l: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_metric_l.pt
      input_size: [1274, 714] # [width, height]
      veriable_input_size: true
      metric_output: true
      inverse_depth: false
      input_format: "image"

    ml-depth-pro:
      weights:
        default: "/home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_pro.pt"
      input_size: [1536, 1536] # [width, height]
      veriable_input_size: false
      metric_output: true
      inverse_depth: false
      input_format: "image"

    SPIdepth:
      weights:
        default: "/home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/spidepth.pt"
      input_size: [1024, 320] # [width, height]
      veriable_input_size: false
      metric_output: false
      inverse_depth: false
      input_format: "image"


  onnx:
    Depth-Anything-V2:
      weights:
        default: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_s.onnx
        s: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_s.onnx
        b: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_b.onnx
        l: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_l.onnx
      input_size: [1274, 714] # [width, height]
      veriable_input_size: false
      metric_output: false
      inverse_depth: true
      input_format: "image"


```

```yaml
  Metric3D-V2:
    weights:
      default: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/metric_3d_s.onnx
      s: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/metric_3d_s.onnx
      l: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/metric_3d_l.onnx
    input_size: [1280, 720] # [width, height]
    veriable_input_size: false
    metric_output: true
    inverse_depth: false
    input_format: "image"

  Depth-Anything-V2-Metric:
    weights:
      default: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_metric_s.onnx
      s: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_metric_s.onnx
      b: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_metric_b.onnx
      l: /home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/depth_anything_metric_l.onnx
    input_size: [1274, 714] # [width, height]
    veriable_input_size: false
    metric_output: true
    inverse_depth: false
    input_format: "image"

  ml-depth-pro:
    weights:
      default: "/home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/ml_depth_pro.onnx"
    input_size: [1536, 1536] # [width, height]
    veriable_input_size: false
    metric_output: true
    inverse_depth: false
    input_format: "image"

  SPIdepth:
    weights:
      default: "/home/orin/Remi/Monodepth/depth-model-pipeline/pipeline/models/spidepth.onnx"
    input_size: [1024, 320] # [width, height]
    veriable_input_size: false
    metric_output: false
    inverse_depth: false
    input_format: "image"
```

```
112
113  execution_providers:
114      - TensorrtExecutionProvider:
115          device_id: 0
116          #trt_fp16_enable: true
117          #trt_int8_enable: true
118          trt_builder_optimization_level: 5
119          trt_max_workspace_size: 7516192768 #7GB limit
120          trt_engine_cache_enable: true
121          trt_engine_cache_path: /home/orin/Remi/Monodepth/
                depth-model-pipeline/testing_pipeline/model_caches
                /fp32_5/ #rest of the path is added in the code
122          trt_timing_cache_enable: true
123          trt_timing_cache_path: /home/orin/Remi/Monodepth/
                depth-model-pipeline/testing_pipeline/model_caches
                /fp32_5/ #rest of the path is added in the code
124          trt_force_timing_cache: true
```

# B Visualised time results

This appendix contains the results for model configurations tested, that could be used with both PyTorch and ONNX Runtime, which were not shown in chapter 4.

Depth Anything V2's medium NoP relative model TRT optimisation times can be observed in Figure B.1. Average pre-processing times with standard deviation in Figure B.2 and average inference times with standard deviation in Figure B.3.

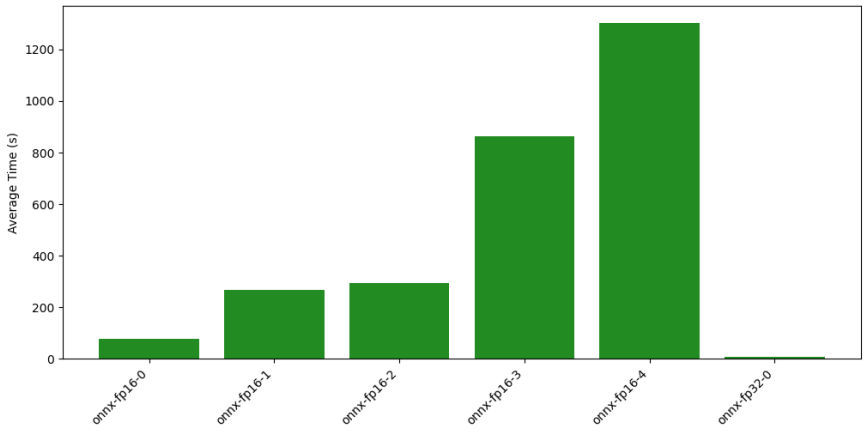


Figure B.1: Time spent on optimisation for Depth Anything V2's medium NoP relative model

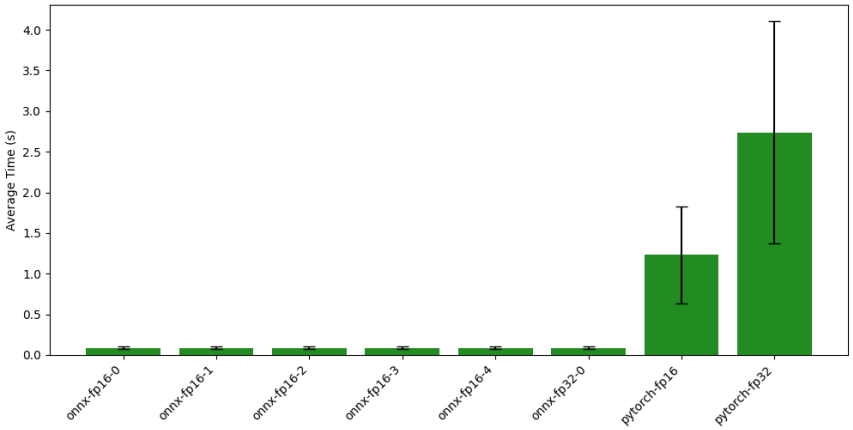


Figure B.2: Total time spent on pre-processing for Depth Anything V2's medium NoP relative model

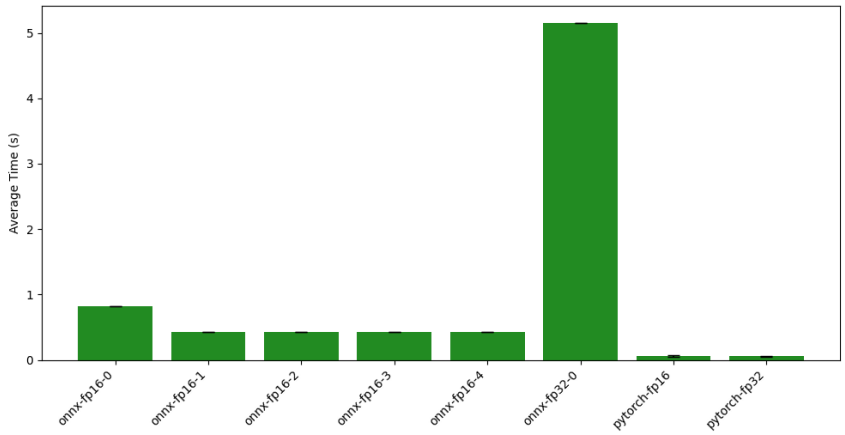


Figure B.3: Total time spent on inference for Depth Anything V2's medium NoP relative model

Depth Anything V2's smallest NoP metric model TRT optimisation times can be observed in Figure B.4. Average pre-processing times with standard deviation in Figure B.5 and average inference times with standard deviation in Figure B.6.

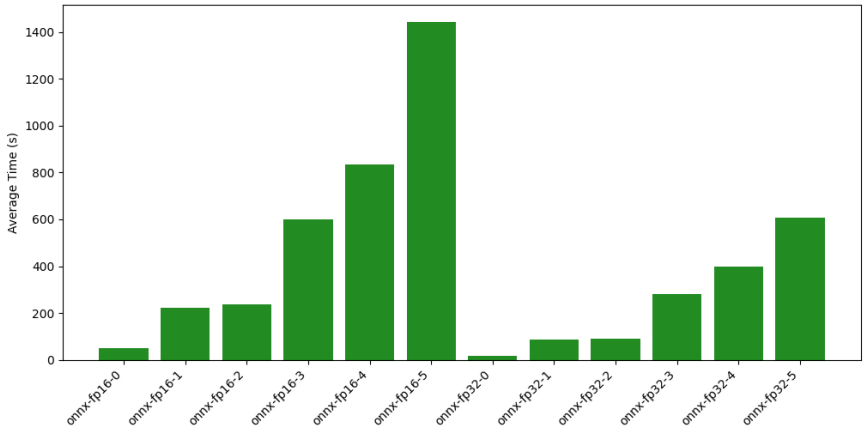


Figure B.4: Time spent on optimisation for Depth Anything V2's smallest NoP metric model

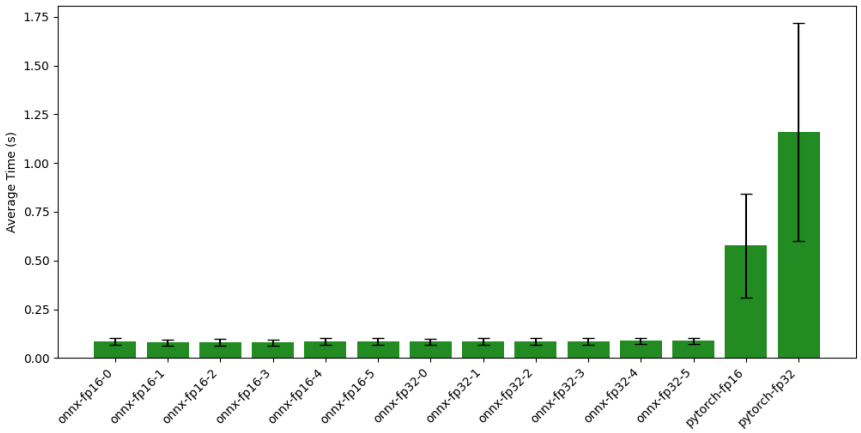


Figure B.5: Total time spent on pre-processing for Depth Anything V2's smallest NoP metric model

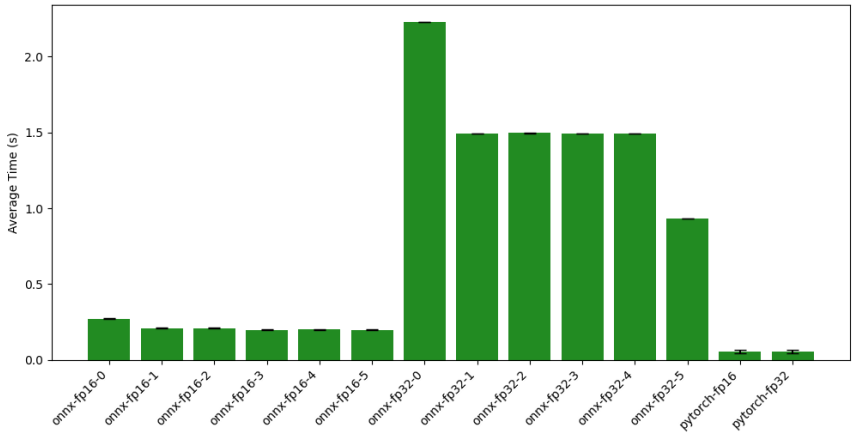


Figure B.6: Total time spent on inference for Depth Anything V2's smallest NoP metric model

Depth Anything V2's medium NoP metric model TRT optimisation times can be observed in Figure B.7. Average pre-processing times with standard deviation in Figure B.8 and average inference times with standard deviation in Figure B.9.

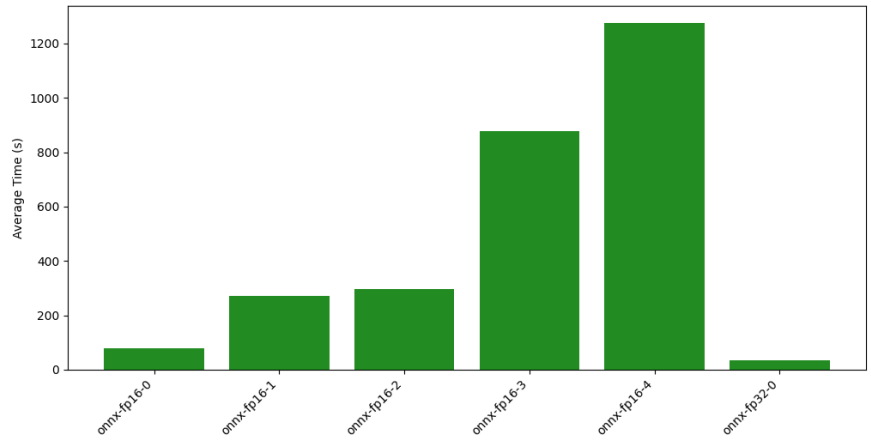


Figure B.7: Time spent on optimisation for Depth Anything V2's medium NoP metric model

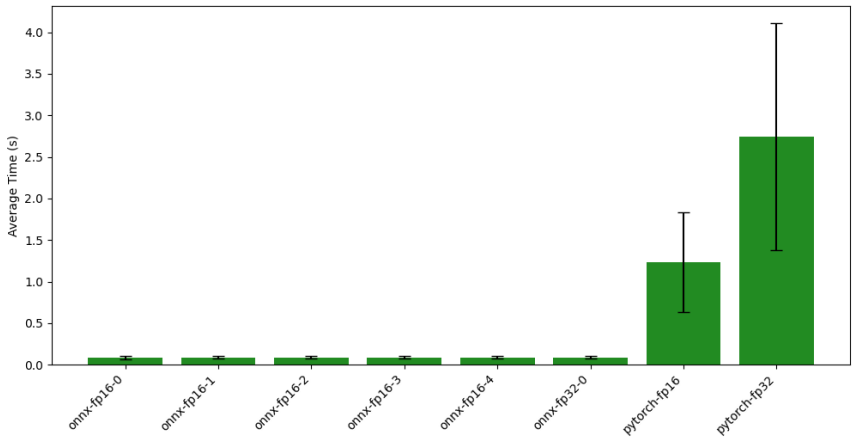


Figure B.8: Total time spent on pre-processing for Depth Anything V2's medium NoP metric model

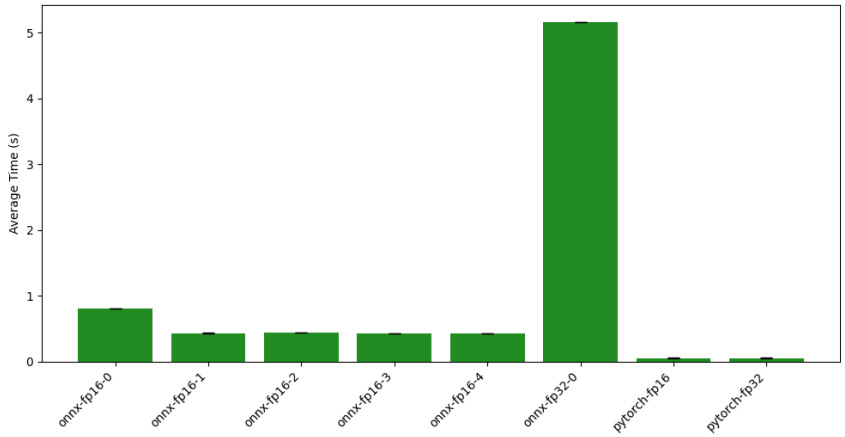


Figure B.9: Total time spent on inference for Depth Anything V2's medium NoP metric model

Metric3Dv2's smallest NoP model average pre-processing times with standard deviation in Figure B.10 and average inference times with standard deviation in Figure B.11. Optmisation times aren't presented for the reasons explained in section 5.1.

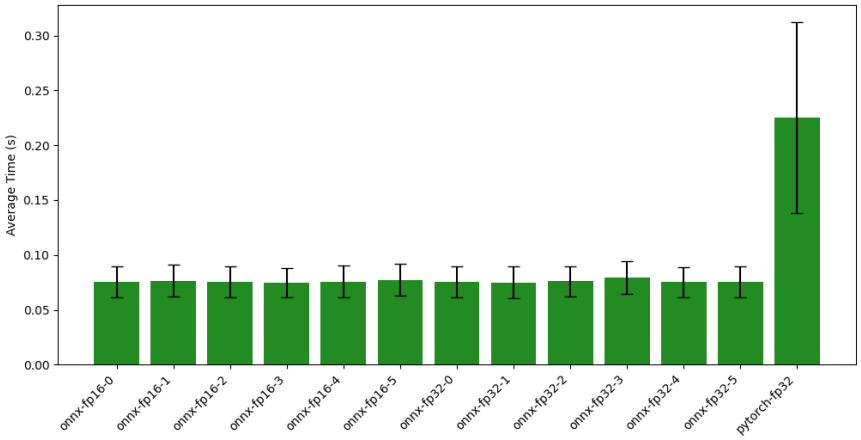


Figure B.10: Total time spent on pre-processing for Metric3Dv2's smallest NoP model

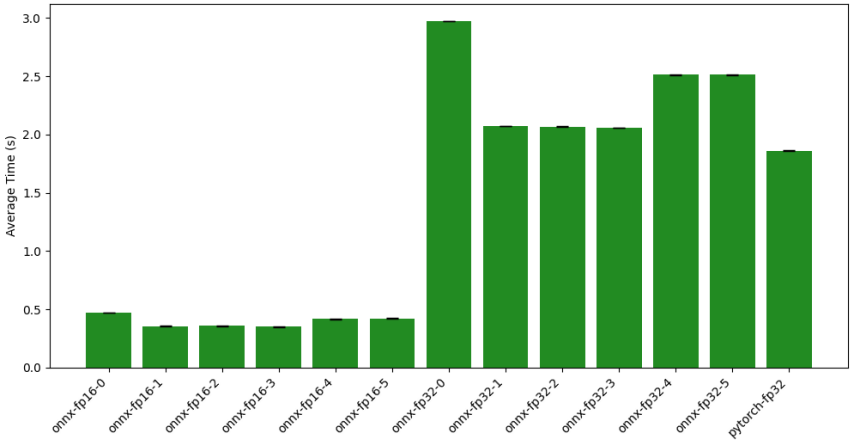


Figure B.11: Total time spent on inference for Metric3Dv2's smallest NoP model

# C Visualised hardware utilisation results

This appendix contains visualisations of CPU and GPU usage during the testing of Depth Anything V2's relative medium NoP model using TRT and optimisation level 2 with FP16 precision weights in Figure C.1 and Figure C.2. Using Pytorch as the backend with FP16 precision weights for the same model CPU utilisation is observable in Figure C.3 and GPU in Figure C.4.

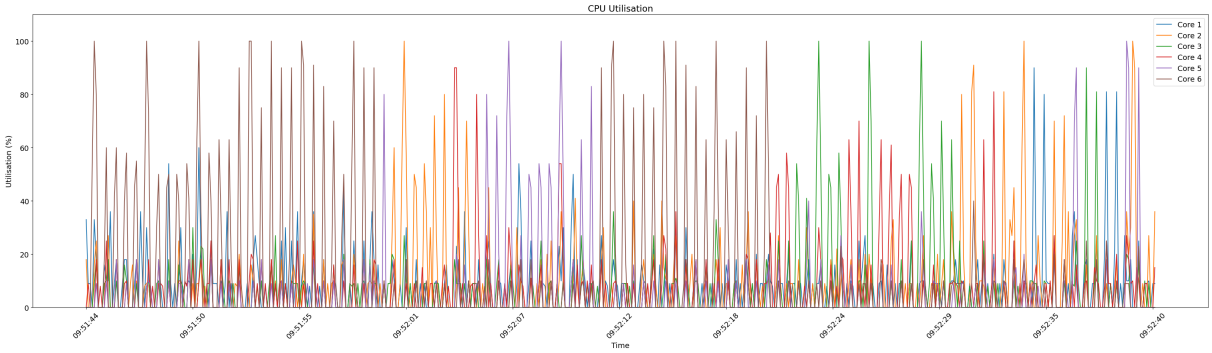


Figure C.1: CPU usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's medium NoP relative model

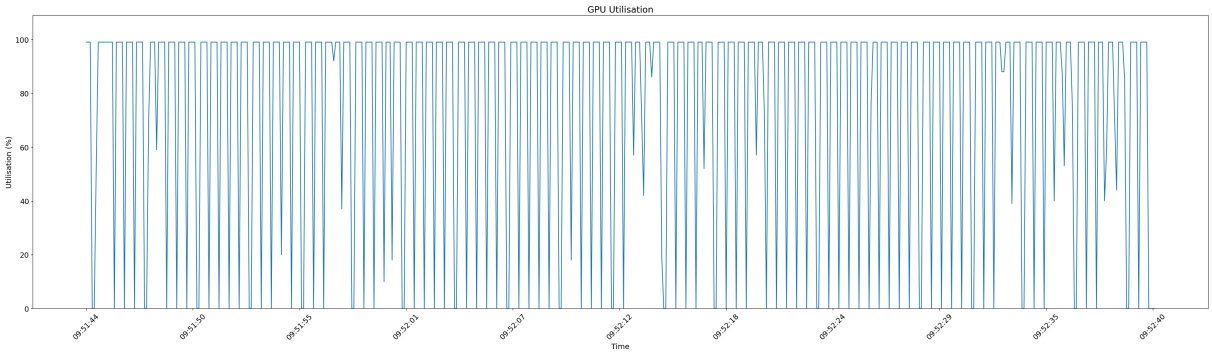


Figure C.2: GPU usage during TRT optimisation level 2 testing using FP16 weights for Depth Anything V2's medium NoP relative model

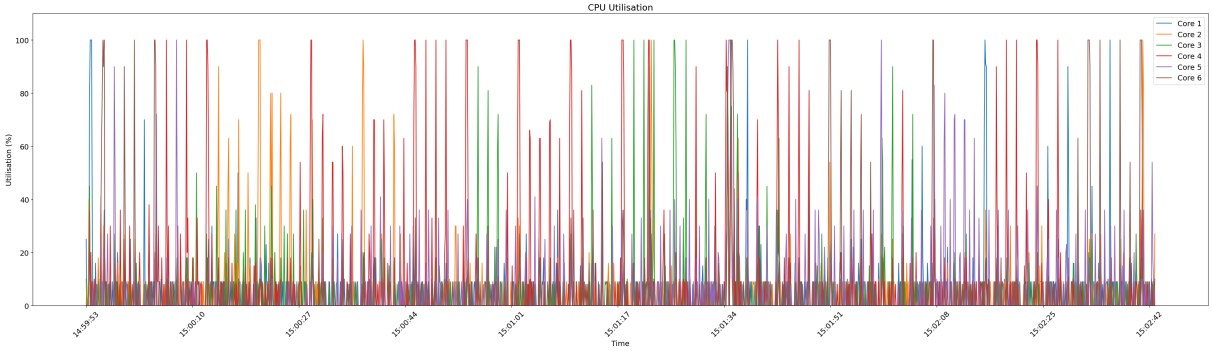


Figure C.3: CPU usage during PyTorch testing using FP16 weights for Depth Anything V2's medium NoP relative model

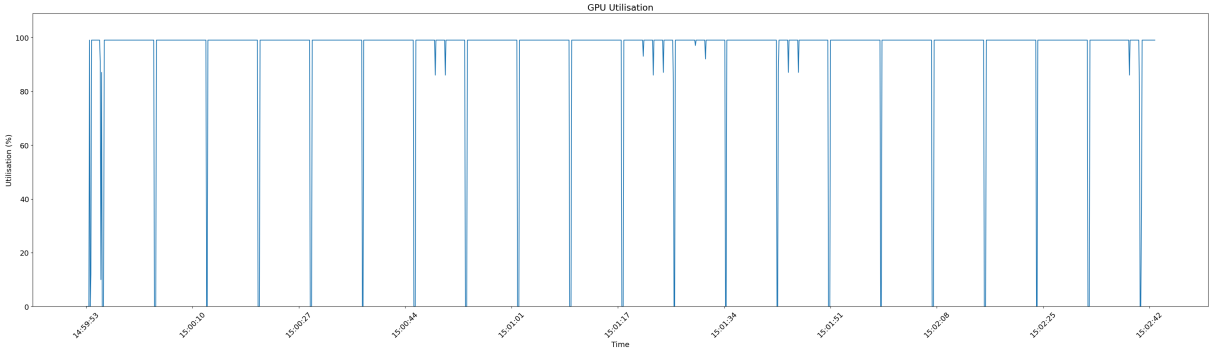


Figure C.4: GPU usage during PyTorch testing using FP16 weights for Depth Anything V2's medium NoP relative model

# Non-exclusive licence to reproduce thesis and make thesis public

I, Remi Raugme

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   **”Development of a testing framework for monocular depth estimation models on targeted hardware”**

   supervised by Quazi Saimoon Islam and Ric Dengel

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

*Remi Raugme*
**20.05.2025**