NTNU
Norwegian University of
Science and Technology

High speed color 3D scanner

using NVIDIA Jetson

Rubin Ingwer Gerritsen

Master of Science in Cybernetics and Robotics
Submission date: June 2017
Supervisor: Morten Alver, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem statement

Many industrial applications of 3D vision involve the use of laser scanning of objects on a conveyor and imaging with 2D camera, with image processing that extracts the laser line 3D information and other reflectance properties. Processing of images from these 2D cameras must often be done at 500-1000 fps, and consequently require fast processing. Some custom 3D cameras use FPGA, and some software packages use GPU processing to this end. The goal of this Master's thesis is to implement an embedded 3D vision system, based on laser scanning, with image processing done on an NVIDIA Jetson GPU+CPU board. Such an embedded 3D vision system enables low cost integration into many industrial applications. The implemented system is intended to enable 3D vision in color, with multiple imaging modes.

This master's thesis builds on the project work "Embedded 3D vision for industrial applications using NVIDIA Jetson" where a basic implementation of the scanning system has been established. In this thesis, the task is to integrate the basic implementation into a full experimental setup at SINTEF Sealab and improve functionality, with particular focus on making the system reliable and robust when faced with real-world variability in image quality and processing times.

Tasks included in the project:

- Integrate the NVIDIA Jetson scanner implementation into the experimental setup at SINTEF Sealab
- Development of extended functionality:
  - Handling of color and 12 bit images
  - Adapt system to allow external trigging of camera
  - Review architecture options for multiple camera setups
- System performance and reliability
  - Handling variations in image quality. Analyze accuracy and repeatability of scans.
  - Handling variability in processing time. Ensure graceful degradation if processing is unable to keep up with camera frame rate.

Abstract

Automatic quality grading of food products is often performed by machine vision systems. The maximum performance of such a system is determined by the amount and quality of the data used. A 3D color scanner is able to capture more data than conventional machine vision systems; in addition to color and shape, it also extracts height profiles. However, 3D line scanners available on the market today are expensive because they use advanced customized components. Therefore, SINTEF Ocean has developed a 3D line scanner for only a third of the price by using standardized off-the-shelf components. This price reduction makes the solution applicable to a wider range of problems.

This thesis presents a distributed 3D line scanner using a GPU enabled NVIDIA Jetson TK1 and a NVIDIA Jetson TX1, which reduces the hardware and deployment cost even more compared to SINTEF's solution. The developed scanner is able to operate in multiple modes and is highly configurable over TCP/IP. Each of the modes extracts different object parameters such as height, scatter, and color. It presents how the scanner is benchmarked to find its maximum performance, and how the scan quality is examined by performing repeatability tests and by performing scans of salmon.

The accuracy of the scanner is determined by the scan rate and camera image size. Increasing the image size increases the cross section accuracy but decreases the scan rate. When both color and height is extracted from an image with size 1280x192 pixels, it was found that the Jetson TK1 and Jetson TX1 were able to obtain a scan rate of respectively 220 and 620 frames per second. This is much slower than the image processing module, which is able to obtain a theoretical maximum frame rate of 4600 and 7300 frames per second. The Jetson TK1 is limited by its slow memory transfer speed, while the Jetson TX1 is limited by the maximum frame rate of the camera.

The developed scanner utilizing the Jetson TX1 has the same performance as the previous system developed by SINTEF. The image processing implementation is portable to other GPUs and much faster than the camera can handle. Consequently, there is a large potential performance increase by using another camera solution. Therefore, this master thesis is a valuable contribution to further development of high speed, low cost, 3D line scanners.

Sammendrag

Automatisk kvalitetskontroll av matprodukter blir ofte utført av maskinsynsystemer. Den maksimale ytelsen til slike systemer avhenger av mengden tilgjengelig data og dataens kvalitet. En 3D-fargeskanner innhenter mer data enn konvensjonelle maskinsynsystemer. Grunnen til det er at det i tillegg til farge og form også leses ut høydeprofiler. Men 3D-skannere tilgjengelige i dag er dyre fordi de baserer seg på avanserte skreddersydde komponenter. Derfor har SINTEF Ocean utviklet en 3D-skanner til kun en tredjedel av prisen. Det har de klart ved å bruke standardiserte hyllevarer. Prisreduksjonen gjør løsningen anvendelig til et større spekter av problemer.

Denne masteroppgaven presenterer en distribuert 3D-linjeskanner som bruker en NVIDIA Jetson TK1 og NVIDIA Jetson TX1 som har innebygde GPUer. Dette senker både komponent- og installasjonskostnadene mer sammenlignet med SINTEF sin løsning. Skanneren kan brukes i forskjellige modi, og er konfigurerbar over TCP/IP. Hver av modusene henter ut forskjellige egenskaper til objekter, slik som for eksempel høyde, lysspredning og farge. Denne rapporten presenterer hvordan skanneren er testet for å finne maksimal ytelse. Skannekvaliteten er undersøkt ved repeterbarhetstester og ved å skanne laks.

Nøyaktigheten til skanneren avhenger av skannehastigheten og kameraets bildestørrelse. Ved å øke bildestørrelsen, øker tverrsnittsnøyaktigheten, men da blir skannehastigheten redusert. Når både farge og høydeprofil blir hentet fra en bildestørrelse på 1280x192 piksler, viste testresultater at Jetson TK1 og Jetson TX1 oppnådde en maksimal skannehastighet på respektive 220 og 620 bilder i sekundet. Dette er mye tregere enn bildeprosesseringsmodulen som har en teoretisk maksimal hastighet på 4600 og 7300 bilder i sekundet. Jetson TK1 er begrenset fordi den har begrenset minnehastighet, mens Jetson TX1 er begrenset av kameraets maksimale opptaksfrekvens.

Skanneren som benytter seg av Jetson TX1 har samme ytelse som systemet utviklet av SINTEF. Bildeprosesseringsimplementasjonen er kompatibel med andre GPUer, og er mye raskere enn kameraets begrensninger. Derfor finnes det et stort ytelsespotensiale som kan bli utnyttet ved å bytte kameraløsning, noe som gjør denne masteroppgaven et verdifullt bidrag til videre utvikling av 3D-skannere med høy skannehastighet og lav kostnad.

Preface

This master thesis is a continuation of the project ”Embedded 3D vision for industrial applications using NVIDIA Jetson” and was written at the department of Engineering Cybernetics at NTNU. The thesis' problem statement was created by SINTEF Ocean in order to research if the cost and size requirements of their existing color 3D scanner could be reduced.

The work done in this thesis is a completely new implementation than rather than an extension to the previous scanner. The previous project work uncovered a number of weaknesses, and together with new requirements, this required a full architecture revision. Therefore, a lot of time has been used to redesign functionality that was already present in the previous scanner. The new architecture has led to a system with drastically improved performance, a range of new features, and enhanced flexibility and robustness.

Many problems during the development were resulting from the used PointGrey camera. Luckily, PointGrey's support team has been very helpful and provided test programs and suggestions to alternative implementation strategies. The received test programs made it possible to confirm that the camera design had bugs and limitations that PointGrey were unaware of. To clarify these uncertainties, contact was kept with PointGrey during major parts of this thesis work. In addition, it became necessary to redesing the error handling implemention. Therefore, the camera problems have consumed time that else would have been used to develop other features.

I would like to thank my supervisor and co-supervisor for all the constructive input given throughout both my specialization project and master thesis. Although the many answers from Vijay Venkatraman from PointGrey's support team in some cases have frustrated me, also he deserves credit for this thesis. At last, I also want to thank my family and girlfriend for motivating me throughout the whole period.

Table of Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Introduction

This chapter gives an overview of the goals and motivation of this master thesis. In order to understand the goals and the rest of this thesis, it is highly recommended to read section 1.2, which present the principles of the scanner. Section 1.3 describes some similar available solutions available on market. Next follows the project outline, which presents the interpretation and priority of the goals of this thesis in detail. Finally, in section 1.6, the structure of this thesis is given.

1.1 Motivation

In many industries, inspection and quality grading is a task performed by computer systems. Compared to humans, computer systems are often cheaper, faster, more accurate, and give results that are more consistent across different production lines. Therefore, the manufacturer might not only experience decreased production costs, but also increased consistancy.

The quality of a product is commonly determined by a number of acceptance rules. These can for example limit the size of an object within some boundaries. However, using acceptance rules becomes impractical when the number of parameters gets large and the parameters are interdependent. This is often the case for the food industry. The quality of food is not only determined by its shape, but also by its color. In addition, the shape of food is often irregular. Therefore, machine learning systems are more often used for this purpose, because they can handle a large number of parameters.

The maximum performance of the machine learning system is determined by the amount, quality, and type of the received data. For the case of the food industry, objects are often captured as 2D images. Although 2D images do not express as much information as a human can obtain, it is many cases sufficient (Brosnan and Sun 2004). SINTEF Fisheries and Aquaculture found that by using colored 3D models in combination with a machine learning system, the performance of a system quality sorting Atlantic salmon increased (Sture et al. 2016). The idea is that this approach mimics how humans performs this task more closely. Because of the generality of this solution, this approach can be used in many other application areas.

When SINTEF quality graded Atlantic salmon, a line scanner obtained the 3D models. The scanner works by projecting a laser line around the entire salmon, and observing this line by multiple cameras. The laser line is extracted from the camera images, and interpreted as a cross-section of the object. By moving the salmon through the scanner, cross sections are merged together to a 3D model.

The process of extracting laser lines from images and merging the cross sections to complete 3D models requires lots of computational power. Similar products on the market therefore perform this with customized electronics. This increases the performance, however, due to increased hardware and development costs, the price increases likewise.

The scanner designed by SINTEF is a centralized solution. All cameras are connected to a high-end computer performing both image analysis and quality classification. Therefore, it requires the computer to be placed in the field, in vicinity to both dust, water, and biomass. To prevent the system from being damaged, the system is extensively shielded. Because of the size and shielding requirements, the deployment cost increases. However, the scanner uses off-the-shelf hardware, and is therefore much cheaper than similar products on the market. For a centralized setup with three cameras, the scanner hardware costs approximately 6500 USD. This approximately one third of the price of similar products on the market. Of course, the hardware cost alone cannot justify such a solution. However, it indicates that it is worth optimizing this scanner further.

SINTEF suggested designing a distributed 3D scanner to decrease the deployment and hardware costs even more. The image processing should be performed by multiple NVIDIA Jetsons, which are GPU-enabled embedded computers. The specialization project investigated how the previous scanner can be distributed using multiple NVIDIA Jetson TK1s (Gerritsen 2016). Its implementation had multiple functional and architectural issues that should be addressed before the system could be put into use. In addition, the developed solution was too slow. It was therefore suggested to use the newer Jetson TX1 for further development. The major tasks of this master thesis are therefore to fix the existing issues, integrate it into an experimental lab setup at SINTEF Sealab, port the implementation to the TX1, in addition to develop new features as described in the problem statement.

1.2 Scanner principles

This section describes the general principles of how 3D models are generated using a setup developed by SINTEF and the setup used in this master thesis. The explanations are meant to be brief, with its purpose to make it easier to define the project outline and describe the previous work. Refer to chapter 3 and chapter 4 for a detailed description of the design and implementation.

The scanner creates 3D models of objects moving over a conveyor belt through the scanner, or alternatively, the scanner passes over the object. The scanner obtains cross sections, and merges them together to a complete model. The system developed by SINTEF performs all these operations on the same computer. In this thesis, the operation of obtaining cross sections is distributed among multiple embedded computers, called line scanner modules, and a centralized computer which merges the cross sections together.

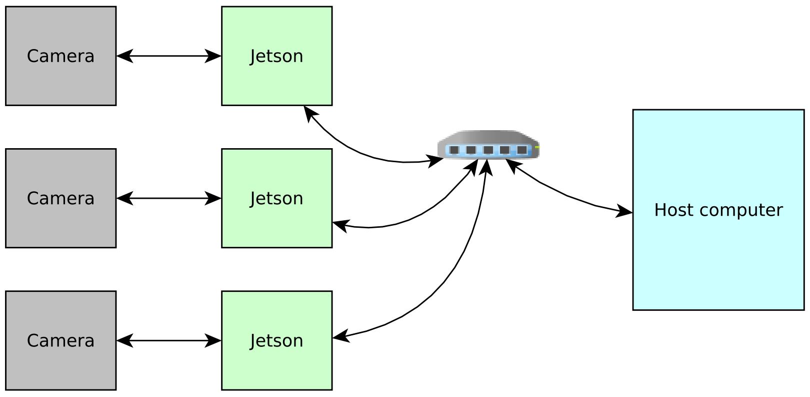


Figure 1.1: Overall system architecture used in this master thesis. Each Jetson is connected to a single camera and to a host computer over a network. SINTEF's centralized solution uses one centralized computer to perform all tasks.

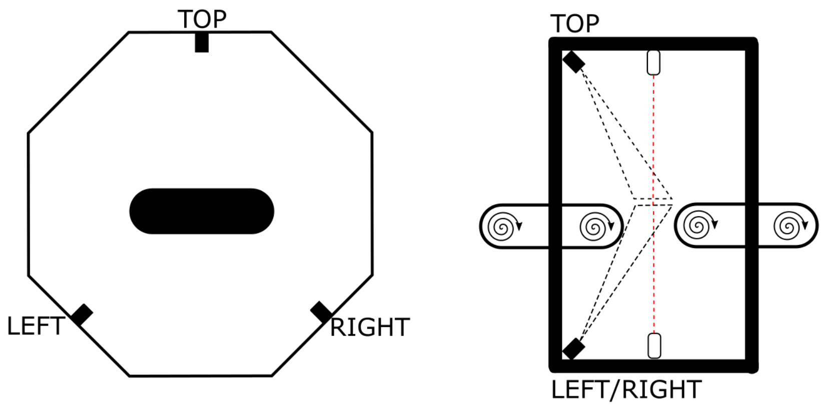


Figure 1.2: Equipment setup when multiple cameras and lasers are included. This illustration is taken from (Sture 2015).

The cross sections are obtained by using multiple cameras. The cameras are mounted at different angles around the conveyor belt to give a full view of the object as shown in Figure 1.2. A laser line is drawn around the object, which is observed by the cameras. Because the cameras are mounted at a fixed angle relative to the laser, the displacement of the laser line determines the height of the object. Therefore, the laser line extractions represent cross sections of the observed object. Color information can also be added to the cross sections, but this is only possible if the laser line is turned off. This can be achieved when the laser is turned off every second image.

A number of factors determine the accuracy of the 3D model. The accuracy in the z-axis (see Figure 1.3) is determined by the image acquisition rate and the speed of the

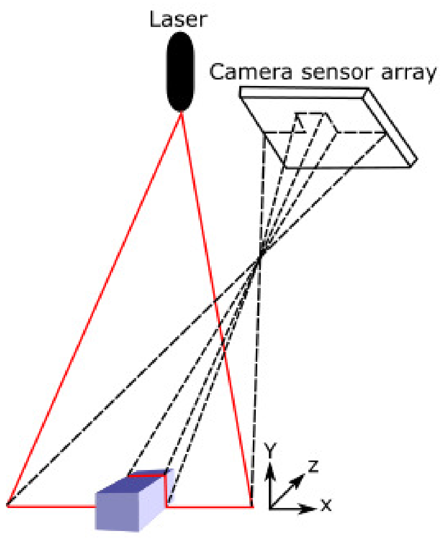


Figure 1.3: A object moving through the laser line. This illustration is taken from (Sture 2015).

conveyor only. The accuracy of the cross section is determined not only by the image resolution, but also by the line extraction algorithm used. When the resolution in the x-axis is increased, it is possible to scan wider objects, or it is possible to move the camera closer to the object. And when the resolution in the y axis is increased, it both possible the scan taller objects, and to increase the accuracy by increasing the angle between the camera and laser. In general, it is therefore possible to say that to increase the accuracy of the 3D model, the image processing workload must be increased.

1.3 Similar products

On the market today, there are multiple products similar to the scanner described in the previous section. In this section, three products from different manufactures are presented and compared. Unfortunately, it was not possible to retrieve the pricing of these products, because pricing is only given out to companies considering buying such products, and not to students. However, SINTEF estimated the pricing of the ColorRanger, Photon Focus, and Cognex cameras to be respectively around 8250, 4250, and 4250 USD. These estimated guesses were made from previous experiences with similar products from these companies.

The included software functionality used for 3D model processing is not compared here, because this master thesis focuses on image feature extraction. Such functionality might be calibration methods, volume calculation, quality assurance, etc. It should be kept in mind that software development and configuration cost are a large contribution to the final installation cost. Therefore, these available features should also be considered before buying a scanner.

The ColorRanger E55434 is an advanced line scanner module produced by SICK. It is interfaced through Gigabit Ethernet and can provide a maximum scan rate of 35000 3D profiles per second. The MultiScan technology allows the camera to be configured to simultaneously extract up to 10 different parameters from each image. These parameters are for example color, height, scatter, intensity, greyscale data, etc. This is achieved by using a custom image sensor where different regions are assigned to the specified parameters. The maximum image width is 1536 pixels with a bit depth of 8 bits per pixel. The product information guide explains additional functionality and specifications (SICK 2011).

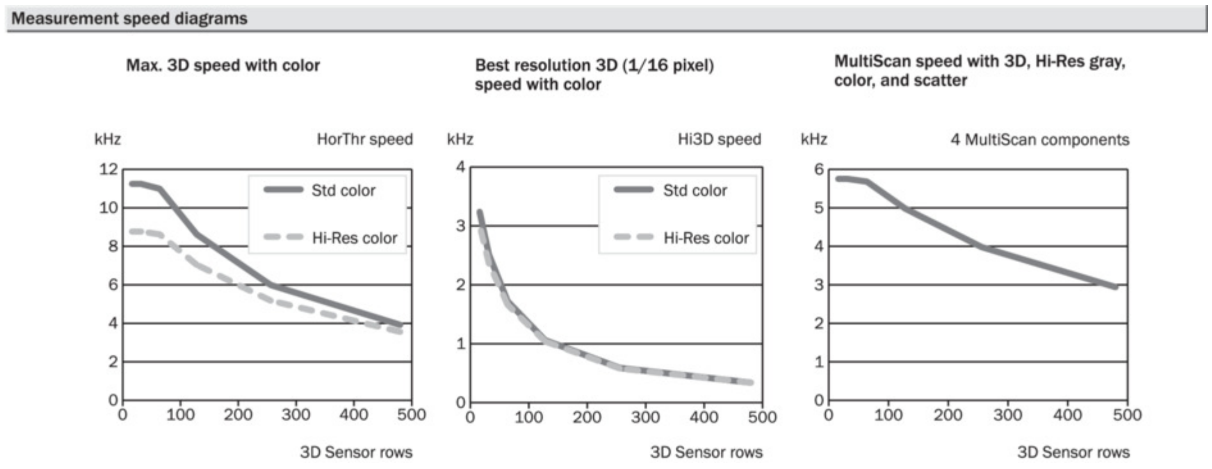


Figure 1.4: Scan rate characteristics for the ColorRanger E55434. This image is retrieved from its product information (SICK 2011).

The MV1 is a camera produced by Photon Focus. It can operate in camera mode, line scanner mode or combined mode where it outputs both image and height information. In the line scanner mode, it outputs both height, width, and intensity of the laser line. It is possible to include color information by using the combined mode. However, it is stated that this mode should be used for debugging purposes. The camera supports a wide range of camera and laser arrangements. The different arrangements are designed to increase the scan quality for objects with high reflectance, high absorption, transparent objects etc. For more information about this camera, refer to its manual (Focus 2016).

The Cognex DS1000 is an IP65 classified line scanner which allows scan rates up to 10 kHz. A special feature of the DS1000 is that is allows 3D image stitching. Multiple scanners will then be able to scan a wide production line, or to scan objects with increased accuracy. The system is distributed, and needs a VC5 controller to process the incoming data.

1.4 Previous work

The TTK4550 specialization project (Gerritsen 2016) done at NTNU in fall 2016 is the basis of this master thesis. The previous implementation by SINTEF has not been

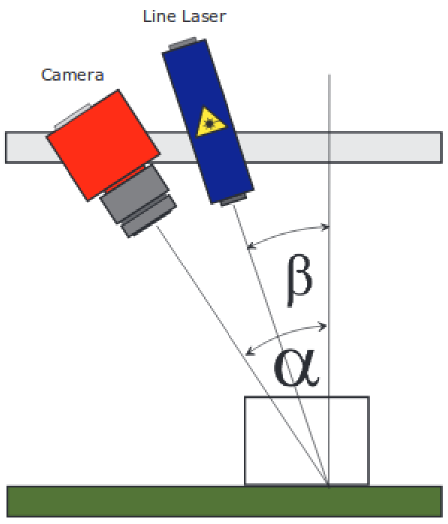
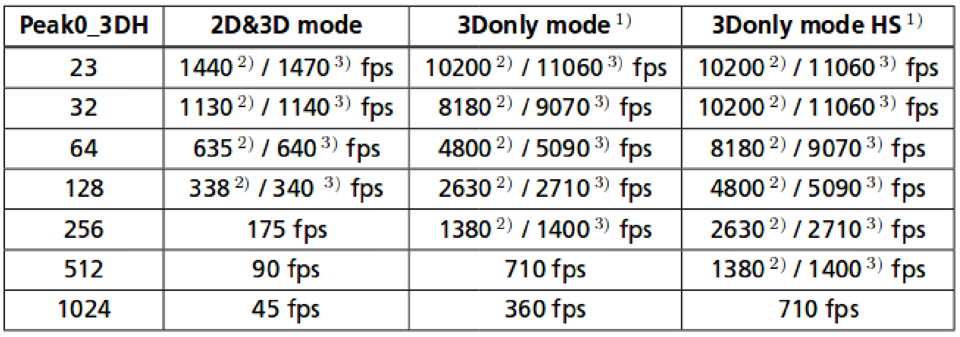


Figure 1.5: Triangulation setup 4 of the MV1 camera is used for high scatter and reflectance materials. This image is taken from its product manual (Focus 2016).



Peak0_3DH	2D&3D mode	3Donly mode 1)	3Donly mode HS 1)
23	1440 2) / 1470 3) fps	10200 2) / 11060 3) fps	10200 2) / 11060 3) fps
32	1130 2) / 1140 3) fps	8180 2) / 9070 3) fps	10200 2) / 11060 3) fps
64	635 2) / 640 3) fps	4800 2) / 5090 3) fps	8180 2) / 9070 3) fps
128	338 2) / 340 3) fps	2630 2) / 2710 3) fps	4800 2) / 5090 3) fps
256	175 fps	1380 2) / 1400 3) fps	2630 2) / 2710 3) fps
512	90 fps	710 fps	1380 2) / 1400 3) fps
1024	45 fps	360 fps	710 fps

Figure 1.6: The maximum framerate for the MV1 camera with given image heights. It is assumed that the image width is 2048 for all these measures. This table is Table 5.3 in its user manual (Focus 2016).

available and has therefore not been used. The specialization project implemented an experimental line scanner module on the NVIDIA Jetson TK1. The developed system was functional, however, it was too slow, and had functional and architectural issues that needed to be fixed to be able to continue the development. Many of these issues are to be addressed in this thesis, and are presented in the next section. The following list gives a summary of the tasks performed in the specialization project:

- A flexible cross platform development environment was set up. This allows both debugging and profiling GPU execution.
- A configuration interface for online configuration of the camera and image processing was developed.
- Simple benchmarking utilities were developed.

- Line extraction of monochrome images was implemented and benchmarked. However, this was done at a low frame rate of 100 FPS, because the system was unstable at higher frame rates.

The following list gives a summary of functional issues:

- The system requires a too high network bandwidth. This issue can only be fixed by revising the communication interface.
- The line scanner module architecture does not allow color extraction. The architecture must be revised to add this functionality.
- The benchmarking approach is not able to distinguish between image processing time and network overhead. This requires the benchmark framework to be revised.
- The system does not allow synchronized triggering of multiple cameras. In order to achieve this, a synchronization primitive must be developed.
- The implementation does not include error-correcting behavior.

1.5 Project outline

The problem statement give the overall goals of this thesis. The purpose of this section is to elaborate how these are interpreted, and clarify on which parts are seen as the most important. In addition to the problem statement, there have been set additional goals and assumptions by SINTEF during the development of the system, which will be described in subsection 1.5.4.

1.5.1 Integration into experimental setup at SINTEF Sealab

This task is interpreted as documenting how the different components are interconnected and how the components are synchronized. For this project, it includes one camera, a laser line, a LED strip, and a conveyor belt, in addition to equipment that allows external control of these components. This task is closely related to improved error handling and system level synchronization.

1.5.2 Development of extended functionality

Development of extended functionality is seen as the most important task of this thesis. The system implemented in fall 2016 was lacking both performance and functionality compared to the system previously developed by SINTEF. The goal should therefore be to reach the performance and functionality of the system developed by SINTEF. The next paragraphs explain the most important parts of the new extended functionality.

Handling of color and 12 bit images

The previous implementation on the Jetson TK1 extracted the laser line from 8-bit monochrome images. The switch to colored images requires a new line extraction algorithm. Handling of color information requires the camera to switch to another mode. Because the camera cannot output RGB images at a high frame rate, the line extraction must use RAW images. As presented in the previous project report, extracting color information requires the architecture of the line scanner module to be revised.

Adapt system to allow external trigging of camera

The suggested system architecture uses multiple line scanner modules. The correctness of the scanner requires these modules to be synchronized. This is achieved through some kind of synchronization mechanisms that involve synchronized triggering of the equipment. This will require an extra centralized module in the system, extra wiring, and extra configuration options for the camera.

Review architecture options for multiple camera setups

Even though the previously presented system architecture will be used for this system as well, the system architecture options should be reviewed. The previous project report presented challenges related to network bandwidth and synchronization, and it is expected that these will be solved for this project. Other system architectures with its strengths and weakness should also be considered.

1.5.3 System performance and reliability

Besides functionality, system performance and reliability are two major concerns. The focus of the specialization projects was setting up a test environment, and not on system reliability. It is therefore natural to continue with that now.

Performance is interpreted as the accuracy and repeatability of the measurements. In terms of a scanner module, the accuracy is translated to the product of image resolution and scan rate. The upper limits of accuracy of should be found, both in terms of the individual components, and of the entire system. The bottlenecks of the system should be found.

The repeatability is a factor of image quality and camera calibration, which requires a lot of testing with different image parameters. Because this task is thought to be very time consuming without improving the quality of the scanner much, this will not be strongly emphasized in this thesis. It is expected that the camera must be re-calibrated when put into operation anyways. However, the repeatability should be tested by comparing an object scanned multiple times, and a discussion should be included to point out techniques that can be used to cope with variable image quality.

Reliability is defined as *the ability of an item to perform a required function, under given environmental and operational conditions and for a stated period of time (ISO 8402)* [ch 1.4] (Rausand and Høyland 2004). One example of such a condition is handling variability in processing time. This might mean including safety margins, or returning suboptimal results. It is expected to define under which the conditions the system should be operable.

1.5.4 Additional requirements and assumptions set by SINTEF during the development

To simplify the system architecture, it was stated that the system should expect to always handle 16-bit images. This would simplify memory management. Adding the possibility of handling 8-bit images would double the number of possible configurations of the system, and make memory management more complicated. This is not directly an issue, but it would make it more challenging and time consuming to implement and test all combinations.

Although not stated explicitly, it is expected that the host control system should be implemented in LabVIEW. In fact, the functionality provided by the host control system could have been obtained using any programming environment. Nevertheless, SINTEF's previous scanner uses LabVIEW, and therefore by implementing the host control system in LabVIEW, this would make future system integration easier.

The system should also be able to operate in different scan modes. This would make the system more flexible, and simplify the future development: Instead of modifying the image processing implementation, a new scan mode can be added.

1.5.5 Scan mode definitions

This section describes the scan modes the system should support. A scan mode is defined by its image sequence and image processing algorithm. For the current system, three image types were defined. However, it should be kept in mind that more image types may be added in the future. The state of the laser and LED strip for each of these image types are given in Table 1.1.

Image type	**LED strip state**	**Laser line state**
Laser (L)	OFF	ON
Color (C)	ON	OFF
Black (B)	OFF	OFF

Table 1.1: State of the laser line and LED strip for the defined image types.

Depending on which frame types that are included, some requirements are set:

- For modes including a laser frame, the height, intensity, scatter and reflectance profile should be calculated. The used definitions of these parameters are found in section 3.2.

- For modes including a color frame, the color should be calculated.

- For modes including a black frame, the height should be calculated from the image resulting from subtracting the black frame from the laser frame.

Mode	**Image sequence**
Mode 0	L L L L ...
Mode 1	L C L C ...
Mode 2	C C C C C ...
Mode 3	L B L B ...
Mode 4	L B C L B C ...

Table 1.2: Definitions of the image sequences for each scan mode.

From the scan mode definitions it is clear that the maximum scan rate is determined by both the camera frame rate, and the image sequence length. Therefore, when the camera operates at a fixed frequency, the scan rate of mode 1 is faster than scan mode 4. Choosing between those modes is therefore a trade-off between scan rate and scan quality.

1.6 Structure of the thesis

- **Chapter 2**. Describes the functionality and limitations of the used equipment. It is described how the different components are interconnected and how they are synchronized.
- **Chapter 3**. This chapter describes message interface between the host control system and line scanner module, and the used image parameter definitions.
- **Chapter 4**. In chapter 4, some of the most important implementation details of the line scanner module are given. It is explained how error detection and correction is achieved, and how the image processing speed is increased.
- **Chapter 5**. In this chapter, a simple LabVIEW host implementation is described. It is described how data is retrieved, extracted, and how it is visualized.
- **Chapter 6**. The scanner is subjected to a range of tests. This chapter describes the test setups used to find limitations, performance, and repeatability of the scanner.
- **Chapter 7**. The results of the scanner tests and benchmarks are given in chapter 7.
- **Chapter 8**. This chapter discusses the validity and implications of the obtained results.
- **Chapter 9**. The work of this master thesis is concluded in chapter 9.

- **Chapter 10**. Further development of new features and alternative architectures is discussed in this chapter.

This report includes multiple appendices:

- **Appendix A**. Many readers might be unfamiliar with GPU programming and CUDA. This chapter gives the necessary background information to understand the implementation details given in chapter 4.
- **Appendix B**. This Appendix describes which and how an operating system was flashed onto the Jetsons. A step-by-step installation guide is given describing how the necessary tools were installed, and how the Jetsons were configured for maximum performance.
- **Appendix C**. In Appendix C, a proposal of a development environment is given, and it is described how this is achieved using NVIDIA Nsight Eclipse Edition.
- **Appendix D**. Multiple applications and tools were developed for different purposes for this master thesis. In this chapter it is explained how to launch and use the line scanner module, LabVIEW host, configuration tools and benchmarker.
- **Appendix E**. The code snippets referred to in this report are placed in Appendix E.
- **Appendix F**. Appendix E gives detailed test results omitted from chapter 7. These results might be used as reference for further development.

Chapter 2

Equipment and setup

For this master thesis, the following components were used:

- A *GS3-U3-23S6C-C* Point Grey Grasshopper3 USB3.0 color camera.
- A NVIDIA Jetson TK1 and a NVIDIA Jetson TX1 for image processing.
- A Lenovo Yoga 500 with an Intel Core i7-6500U CPU and 8GB memory as host control system computer.
- A *Z40M18SF660LP30* line laser from Z-LASER GmbH.
- A 24V LED strip.
- A conveyor belt.
- An Arduino Mega.

First, this chapter gives a description on the camera and Jetsons with focus on its features and limitations. Next follows an explanation on how these are interconnected, and how synchronization is achieved.

2.1 The PointGrey Camera

This section introduces the PointGrey camera used for this project. First, an overview of its features and interface capabilities is given. Next, the output formats of the camera, and the Bayer filter format are described. Then comes an overview of the maximum frame rate for various image sizes. The last sections present limitations and anomalies of the camera found during the development of this system.

2.1.1 Feature overview

GS3-U3-23S6C-C is a color camera from the Grasshopper3 series manufactured by Point-Grey. The camera can be interfaced through either USB2.0 or USB3.0. It has a wide range of functionality ranging configurable exposure time, contrast, capture modes, pixel formats, selecting a region of interest, etc. A complete list of camera features can be found in its Technical Reference Manual (Point Grey 2016a).

Camera control is achieved through one of the available Software Development Kits (SDKs) provided by Point Grey. When this project was started fall 2016, there were two available SDKs, the *FlyCapture SDK* and *FlyCapture2 SDK*. Later on the *Spinnaker SDK* also appeared. Because the camera host computers used for this project, NVIDIA Jetsons, run Ubuntu ARM, and the fact that the *Spinnaker SDK* does not support this operating system, the *FlyCapture2 SDK* is used.

Later it is described that the Jetson TK1 runs Ubuntu 14.04, and the Jetson TX1 runs Ubuntu 16.04. Therefore, two different versions of the *FlyCapture2 SDK* must be used. The TK1 uses version 2.9.3.43, while the TX1 uses version 2.10.3.266. There is a minor API incompatibility between these versions. The newest version includes a field *highPerformanceRetrieveBuffer* in *fc2Config* structure, while the older version the does not.

The SDK provides two methods for retrieving images. Images can be obtained synchronously by calling `fc2RetrieveBuffer()`, or asynchronously through a callback by first calling `fc2SetCallback()`. Image retrieval is only possible after the camera is configured and `fc2StartCapture()` is called. The user is responsible for retrieving and processing images in time. If images are not handled in time, the behavior it specified by the selected buffer policy, `fc2GrabMode`. This can be either `FC2_BUFFER_FRAMES` or `FC2_DROP_FRAMES`.

The camera can operate using two modes, free-running and in asynchronous mode. The first mode sets the camera to capture images at a given frame rate, while the second mode waits for an incoming trigger. The incoming trigger can be either a software trigger, or an external trigger through one of the GPIO ports. Depending on which trigger mode the camera is configured to use, the camera can overlap image retrieval and data transfer (Point Grey 2015b).

It is possible to embed extra information to the images. This information is the encoded into the upper left pixels of an image. It is for example possible to detect if frames are dropped by embedding the time stamp and/or frame counter can be embedded to each image. By embedding the GPIO state into each image, the image can be tagged with a type. For more information about embedded image information, refer to PointGrey's Technical Application Note 10536 (Point Grey 2016b).

2.1.2 Limitations in the FlyCapture2 SDK

Browsing the header files `FlyCapture_C.h` and `FlyCaptureDefs_C.h` found in the include folder reveal that most of the functionality of the camera described by its Reference manual is exposed through the SDK. In addition to the functionality provided through the

SDK functions, it is possible to configure the camera registers directly through functions similar to `fc2WriteRegister()`. Camera register information can be found in the Register Reference (Point Grey 2015a).

However, one function not provided by the SDK, is the possibility of retrieving multiple images at a time. The Register Reference explains how the multi-shot mode can be used to send multiple images from the onboard image frame buffer [sec. 3.3.3](Point Grey 2015a). Point Grey's support team has confirmed that this functionality is not supported for either of the provided SDK's.

Another feature not available to the SDK is the ability to buffer images directly to GPU memory. There exists a function `fc2SetUserBuffers()`, but when this is used to point to GPU memory, images are not received. PointGrey's support team has confirmed the lack of this feature.

When images are retrieved, only a pointer to the image data is provided. This sets a limitation on image buffering in the user application. If the user wants to buffer images, it is required to copy the retrieved images into its own buffer. This because the buffer layout of the buffer provided by `fc2SetUserBuffers()` is unknown. The "extra" required copy operation limits the maximum performance. The SDK could have used an alternative approach where it provided an interface to its own image buffers. This would have removed the need for the extra copy operation, however would probably also increase the probability of memory corruption.

2.1.3 Image output format and the Bayer filter

The camera is able to output images encoded as either `MONO`, `RBG`, `YUV`, or `RAW`, where each pixel format may have multiple pixel depth choices. The project goal states images should have a pixel depth of minimum 12 bits per pixel, and that color images should be supported. This limits us to choose either `YUV` or `RAW`. The `YUV` format transforms the image into brightness and color components, while the `RAW` is the image retrieved directly from the imaging sensors. Because the `RAW` format is easier to interpret, and the camera is able achieve a higher frame rate using this format, the `RAW` format will be used for this project.

The `RAW` format outputs the image with a Bayer tile format. The Bayer tile format is obtained when light is filtered through a Bayer filter. This filter is one of the most commonly used color filter array for arranging RGB pixels on a set of photo resistive sensors. The use of color filter arrays makes it is cheaper to produce image sensors, because the same photo sensors can be used for the entire image sensor. The Bayer filter consists of twice as many green filters as red and blue, which mimics the higher receptiveness to green of the human eye. A typical Bayer filter, called RGGB, is shown in Figure 2.1.

In fact the camera will typically always capture images of the type `RAW`, and convert it to the given format before it is sent to the host. This process is called debayering. When a `RAW` image is converted to an `RGB` image, this is done by interpolating. Depending on which requirements of the converted image, there exist a large number of debayering techniques.

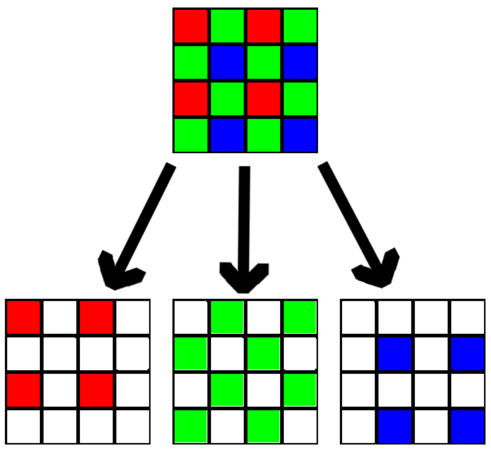


Figure 2.1: This figure shows the orientation of the pixels of an image with the Bayer filter format RGGB.

2.1.4 Documented maximum frame rate

An important property of the camera is its maximum achievable frame rate. This depends on the interface used, the packet size, and the region of interest. Table 8.4.4.2 in the Technical Reference Manual (Point Grey 2016a) summarizes the maximum frame rates for some given pixel formats and image sizes. However, this table is rather limited and does not give a good view on its limitations.

Using the *FlyCapture2 SDK* it is possible to retrieve the maximum frame rate for given configurations. For the pixel formats `RAW12` and `RAW16`, the maximum frame rate varies with the size of the region of interest. From Figure 2.2 it is understood that `RAW12` is the fastest pixel format of these.

The camera technical reference manual (Point Grey 2016a) states that the camera only has a 12-bit ADC. This means that it is unnecessary to use the 16-bit pixel format, as this only adds padding. It is verified that the received data for both 12 and 16 bit images only use 12 bits. Therefore, the reduced frame rate for `RAW16` can only be explained by the necessary padding operation. However, because the difference in maximum frame rate is small, and because `RAW12` requires padding on the host computer, `RAW16` will be used.

2.1.5 Anomalies in the FlyCapture2 SDK

During the development process, several bugs and undocumented features of the Point-Grey Grasshopper3 camera were discovered. These discoveries drastically change the maximum speed of the system and how error correction and error detection can be implemented. Investigation of these bugs consumed a lot of time during the project. The next paragraphs summarize the most important findings. These findings are crucial points to consider for any further development using PointGrey cameras.

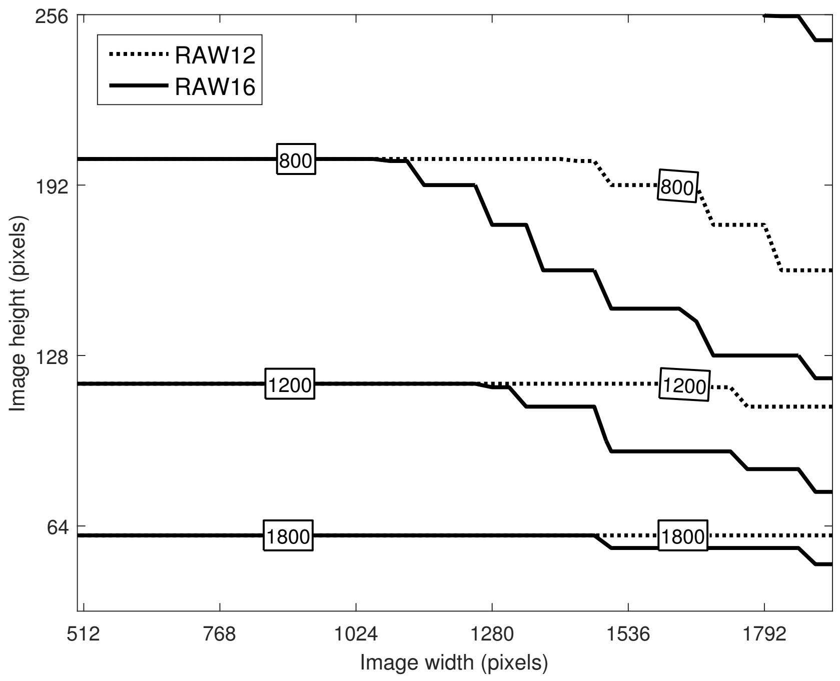


Figure 2.2: Contour plot of the maximum available frame rate for the pixel formats `RAW12` and `RAW16`. The values are retrieved using the *FlyCapture2 SDK*.

The frame counter does not work correctly in asynchronous mode

When the camera was set to run 1000 FPS, and a debug message was printed out to the console every 500 frames, it turned out that the messages were printed out at a rate of one message a second. This indicated that the actual frame rate was 500 FPS. When the camera was set to free running mode, the frame counter indicated that 500 frames were dropped every second. However, when the camera was configured for asynchronous mode, the frame counter indicated that no frames were dropped. Therefore, it was concluded that the embedded frame counter does not work in asynchronous trigger mode. PointGrey's support team suggested using embedded time stamps to be able to see if frames are dropped when the camera is triggered externally.

The time between image time stamps is sometimes negative

After PointGrey's support team was informed with that the embedded frame counter as not working in asynchronous mode, they provided some code to extract the time difference between embedded timestamps. However, this code contained a bug. Occasionally a negative time difference was returned. PointGrey's support team did not answer what the cause of this problem was. I assumed the bug was related to the second counter overflowing. However, the maximum value of this counter was unknown. By trial and error it was found out that this value was 128. It was therefore possible to provide a bugfix as shown in Listing E.8. PointGrey confirmed that this bugfix would work.

There exists no reliable method for retrieving the number of dropped frames

PointGrey provided an extensive test application to determine what the actual maximum frame rate is. It uses the three available methods to determine if frames are dropped: The embedded frame counter, the dropped frames register, and the embedded image timestamps. However, Table 2.1 shows a test result from this code, and shows that all three methods return a different number of dropped frames. Therefore, it was concluded with that at maximum one of the methods was reliable. PointGrey's support team were not able to answer which, but pointed out that the embedded time stamp method should be used until they found a better solution.

	RAW8 1920 x 50	**RAW16 1920 x 50**	**RAW8 1920 x 200**	**RAW16 1920 x 200**
Maximum available frame rate	1961	1961	803	500
The whole grab loop used (s)	0.531	1.016	1.247	3.993
Average FPS based on grab loop time	1883	984	801	250
Dropped frames based on register	0	921	0	997
Dropped frames based on frame counter	0	999	0	999
Dropped frames based on timestamp	0	999	0	963
Average grab time (ms)	0.304	0.784	0.984	3.639
Average FPS based on RetrieveBuffer()	3287	1275	1016	274.76
Minimum grab time (ms)	0.201	0.188	0.165	0.161
Maximum grab time(ms)	0.602	3.82	2.983	4.384

Table 2.1: Achieved frame rate and dropped frames for various pixel formats. The results were obtained using the test program in submitted together with this thesis. The achieved frame rate is lower than the maximum. The number of dropped frames is unreliable.

The calculated maximum frame rate does not depend on the shutter time

This bug was discovered when the maximum frame rate was always 250 FPS, regardless of the set image height. It turned out that the SDK does not take the shutter time into account when the maximum frame rate is calculated. It is therefore allowed to set a shutter time of one second and a frame rate of two FPS. In that case, the camera will take images at 1 FPS. The solution to this bug is to turn the auto shutter functionality off, and set the shutter time to a low value.

The reported maximum frame rate is not correct

It was discovered that the actual maximum frame rate usually is lower that the reported maximum frame rate. This is especially the case when the frame height is low and the pixel format is `RAW16`. PointGrey's support team suggested that this is related to the USB driver. Therefore I tried running the camera on multiple computers, running different versions of Ubuntu and Windows, however, this gave similar results. At last, they did also confirm they were not able to reach the reported maximum frame rate themselves.

A test was performed to measure the actual maximum frame rate. This was done by triggering the camera at with a duty cycle of 400 μs and using PointGrey's test application. This should have given resulting frame rates in the sequence defined by

$$y = \frac{10^6 \cdot n}{400} \quad \forall n = 1, 2, 3, ... \tag{2.1}$$

It was found that the actual maximum frame rate was lower than the specification. Figure 2.3 show the actual maximum frame rate, where the results are interpolated. This because the test revealed another strange behavior: When the image with was set to 1920, and height to 160, 192, 224, and 256 pixels, the maximum frame rate was measured to be 501, 417, 208, and 357 FPS. The expected result would decrease the frame rate as the height increases. Because the results were reproducible for multiple computers, I therefore concluded with that this is a bug and that the maximum frame rate should be determined by trial and error.

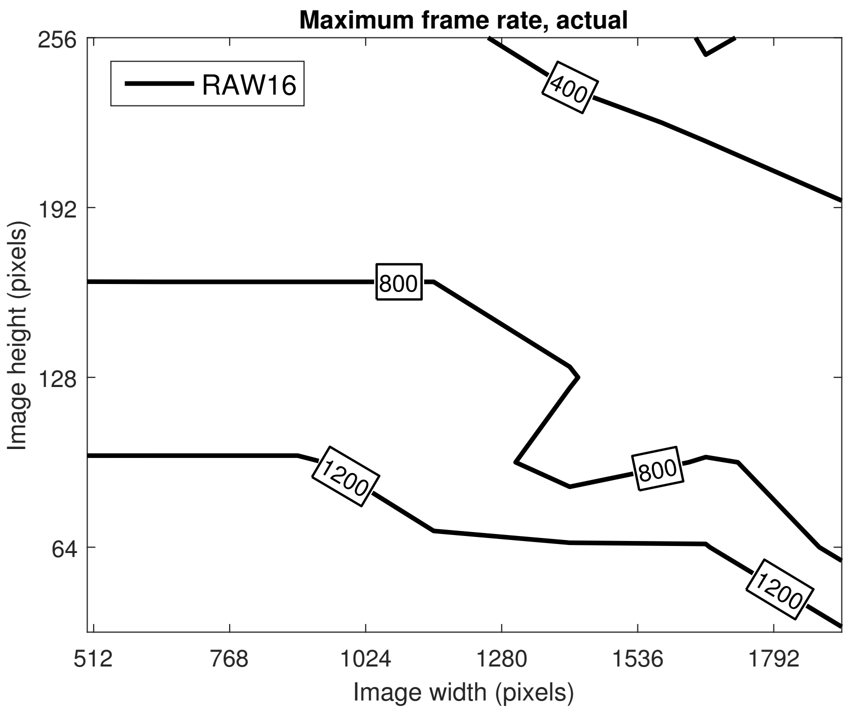


Figure 2.3: Interpolated values of the measured maximum frame rate. It is lower than the specified maximum frame rate given in Figure 2.2.

2.2 The NVIDIA Jetson TK1 and TX1

The NVIDIA Jetson TK1 and TX1 are low cost embedded computers with a built in GPU. They are built to be able to deliver high performance as well as having a low power consumption. The fact that they have a small form factor, a GPU, and a USB3 interface make them useful for this project.

The 3D scanner should be as cheap as possible while delivering the required performance. The specialization project suggested that the cheaper TK1 wasn't powerful enough. Therefore, the implementation software will run on both the TK1 and the TX1 for comparison reasons. Table 2.2 compares the specifications of those two computers.

	Jetson TK1	**Jetson TX1**
Release	June 2014	November 2015
Price (Mar. 2017)	192 USD	499 USD
Dimensions	127 mm x 127 mm	224 mm x 208 mm
GPU architecture	Kepler, 192 CUDA cores	Maxwell, 256 CUDA cores
CPU	2.32GHz ARM quad-core Cortex-A15 CPU	1.73 GHz ARM quad-core Cortex-A57
DRAM	2GB DDR3L	4GB LPDDR4
Max GPU performance	326 GFLOPS	1 TFLOPS
Support concurrent kernels	yes	yes
Async engine count	1	1
GPU Compute capability	3.2	5.3
Multiprocessor count	1	2

Table 2.2: A comparison between the NVIDIA Jetson TK1 and TX1. Some of the values are retrieved from the CUDA function `cudaGetDeviceProperties()`.

NVIDIA provides several tools and packages for simplifying the development process. Linux For Tegra (L4T) is a package containing a bootloader, kernel, and a sample file system. By installing this, the device can run a full-fledged Linux OS. Because the Jetson TX1 is newer, it only support Ubuntu 16.04, whilst the TK1 only supports Ubuntu 14.04. In addition, NVIDIA provides an variant to the Eclipse IDE, called *NSight Eclipse Edition*. This supports debugging GPU code, profiling GPU execution, and remote development. An extended guide on how to use the profiler and debugger can be found in the Profiler User's Guide (NVIDIA 2016c). Appendix B describes how the Jetsons are configured and how libraries and applications are installed.

2.3 Scanner module synchronization

In order to merge the line scans from multiple scanner modules together correctly, the system requires the components and line scanner modules to be synchronized. The lab setup used for this master thesis only consists of one scanner module, and the need for synchronization is therefore limited. Despite this, this section gives a general solution on how multiple cameras, lasers, and LEDs can be synchronized using external triggering. First, it is described which control signals are used, and the purpose of those. Next, their timing characteristics are given for some of the scanning modes defined in the project outline. Finally it is described how this is implemented using an Arduino Mega. Synchronization of the retrieved scan lines at the host controller is handled in software, and is therefore not described until chapter 5.

2.3.1 Control signals and timing

A scanner module consists of one laser, a LED strip, one camera, and one Jetson. These are interconnected to a central trigger module using the signals described Table 2.3 as shown in Figure 2.7. In general, each scanner module has its own set of control wires. However, the LED strip and laser might be shared amongst multiple scanner modules, and might therefore use shared control lines. In addition, the GPIO and trigger signals can be shared if overlapped image capture is allowed.

Name	**Description**
Camera trigger	When this signal is set to low for at least 500μs, an image is captured and sent to the host. If the camera was busy at this time, no image is taken.
Camera GPIO	Indicates indicate the type of image taken. It consists of two wires that allows four image type definitions.
Jetson ready	This signal is set to low when the Jetson is processing an incoming image. This value is set to high when it is ready to retrieve a new image.
Laser	The value of this signal represents the laser being on or off.
LED	This signal is used to turn the LEDs on and off.

Table 2.3: Description of the control signals used for synchronization of line scanner modules.

The timing characteristics depend on the camera frame rate and the current scan mode. Figure 2.4 and Figure 2.5 show timing diagrams for scan mode 0 and scan mode 4 for a single scanner module. When multiple scanner modules are used, it might not be practical to overlap laser frames because they might interfere. This can be solved by obtaining laser frames sequentially as shown in Figure 2.6.

When multiple line scanner modules are used, not only is it necessary that the images are taken sequentially, it is also required that each line scanner module is has retrieved the same amount of images. Therefore, the triggering sequence should only start when all line scanner modules are ready to receive images. For this case, it is therefore necessary that the triggering module and line scanner modules communicate.

It was pointed out in the previous project (Gerritsen 2016), that it was hard to log the time used to process an incoming image when the frame rate was high. Now it will

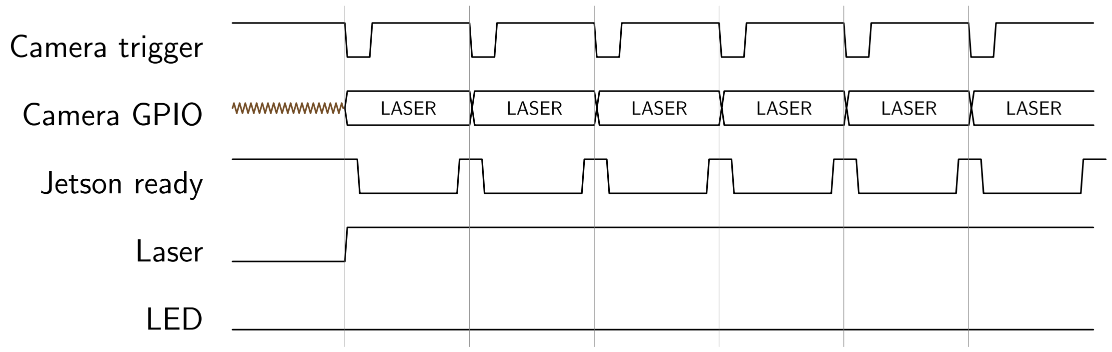


Figure 2.4: Timing diagram for scan mode 0. Scan mode 0 uses only the laser image type. When multiple cameras are used, there are multiple instances of each signal.

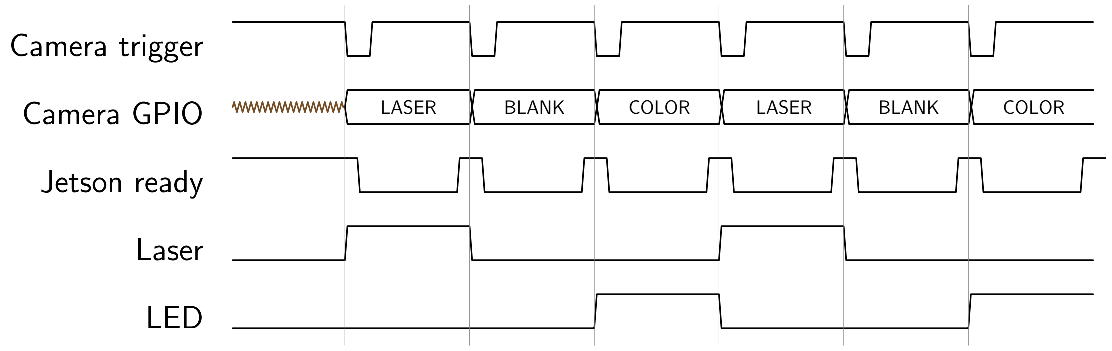


Figure 2.5: Timing diagram for scan mode 4. The state of the LED and laser determine the image type.

be possible to obtain these values by attaching an oscilloscope to the *Jetson ready* signal on the Jetson. In addition, the *Jetson ready* signal can be used for error detection.

2.3.2 External triggering and synchronization using an Arduino Mega

For this master thesis, an Arduino Mega was used for external triggering and synchronization. The Mega is a general-purpose electronic prototyping development kit built around the *ATmega2560* microcontroller. It is well suited for this thesis because it is easy to setup, and has an IDE which as provides a wide range of high abstraction programming interfaces. In addition, the Mega has a large number of general purpose input output connectors (GPIOs), hardware timers, and communication modules. For more information about the Mega, refer to the product overview (Arduino 2017).

Figure 2.7 shows how one single line scanner module in addition to a LED strip is connected to the Arduino. The Arduino ports used are chosen arbitrary from the available non-reserved ports. The LED strip requires external power, and is therefore connected through a relay. When the system consists of multiple line scanner modules, these must be connected in a similar fashion.

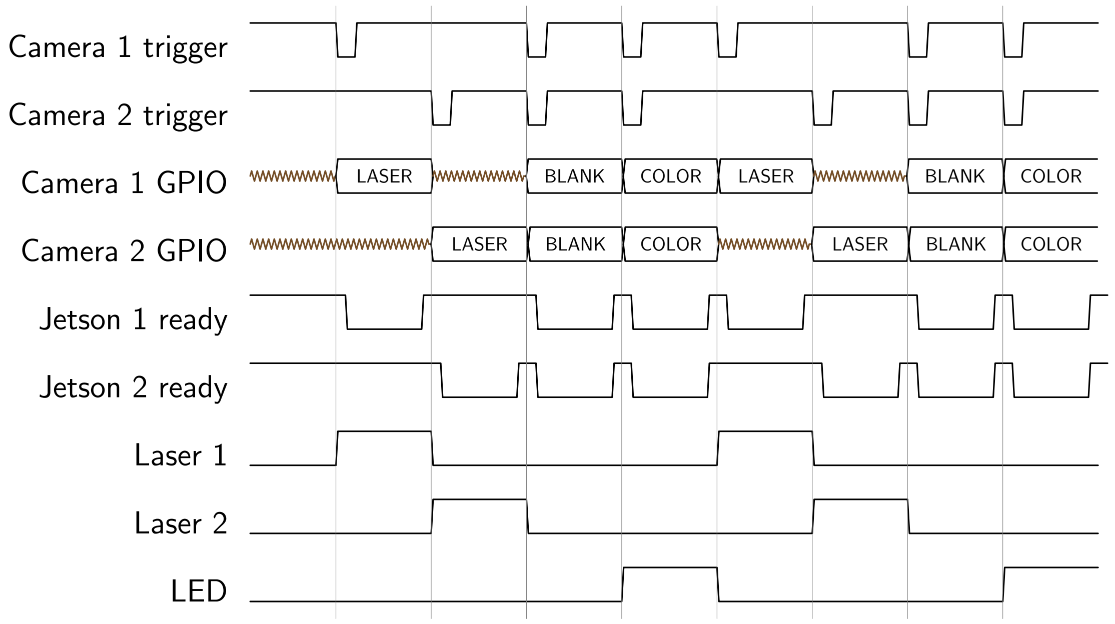


Figure 2.6: A possible timing diagram for scan mode 4 when two cameras and lasers are used. The laser frames are obtained separately at different times, while the blank and color frames are obtained at the same time.

For this master thesis, only one line scanner module was used. It was therefore not necessary to implement communication between the line scanner module and Arduino as described earlier. However, this is not difficult to implement. A suggested approach is to use one GPIO line for each line scanner module, where the value indicates if the line scanner module is ready to start scanning.

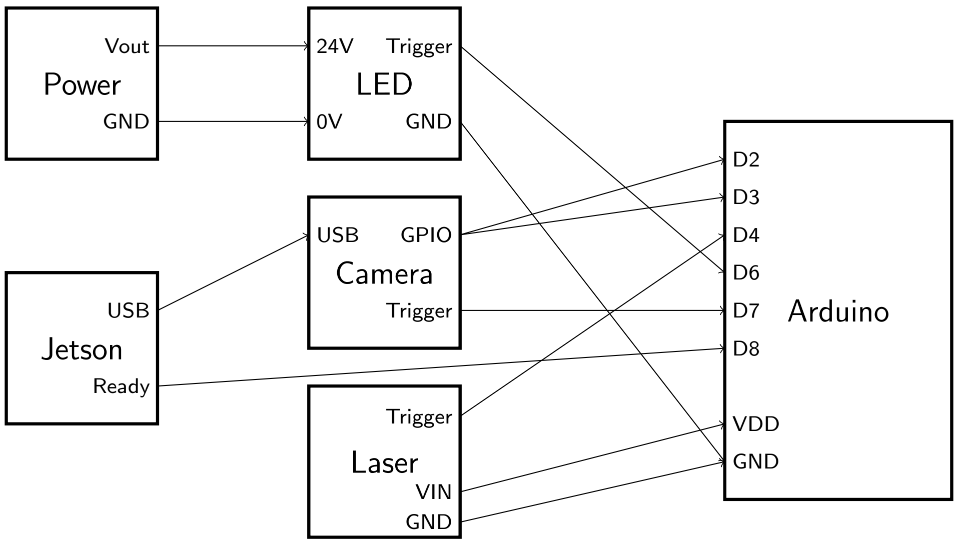


Figure 2.7: Conceptional wiring diagram. The Arduino controls all components of a line scanner module. When multiple line scanner modules are used, some signals might be shared, as shown in Figure 2.6.

Chapter 3

Scanner design

This chapter describes the message interface and definitions of the line parameters used for this project. These descriptions are implementation independent, and can therefore be regarded as specification documents for the scanner. For implementation specific details refer to chapter 4 and chapter 5.

3.1 Message interface

Any type of high bandwidth communication protocol is applicable to communicate between the line scanner modules and host control system. For industrial setup, high speed industrial control networks such as Foundation Fieldbus HSE could have been used. For this thesis, this would have required extra equipment and drivers and therefore increase the development time. Therefore, Ethernet with TCP was used as communication interface.

An alternative to TCP is UDP. UDP is able to utilize more of the network bandwidth, and achieve lower latency. However, it removes the guarantee of the message integrity, arrival, and order of arrival of messages [ch 2.1](Kurose and Ross 2012). Therefore, using UDP would require extra error detection and correction. For this reason, TCP was chosen as transport layer.

The line scanner modules are configured as TCP servers. This has some benefits during setup, configuration and testing. Because the line scanner modules are TCP servers, the hosts must connect to them, which simplifies the situation when multiple hosts are available. In addition, this system architecture allows one application to configure the scanner, and another application use it.

A flexible generic message interface has been designed which makes the development of host and client simple and generic. All messages are defined by the format <*Type, Length, Value*> as shown in Figure 3.1. This message definition allows up to 256 configurable message types, which makes it easy to expand the system functionality.

The generic communication pattern is that the host sends a request to a line scanner module, which then sends a response back. When the line scanner module is the state `SCANNING` (see Table 3.2), the normal communication pattern is broken. In this state the

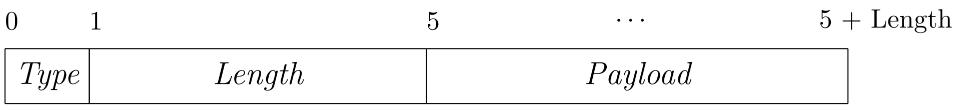


Figure 3.1: The general definition of a message. The defined types are given in Table 3.1. The length is encoded in little endian format.

Value	Name	Description
0x01	`GOTO_STATE`	Controls the state of a line scanner module.
0x02	`CONFIG_CAMERA`	Configures the camera.
0x03	`CONFIG_SCAN`	Configures the scanning module and the image processing.
0x04	`GET_IMAGE`	Retrieve an image from the camera.
0x05	`SCAN_DATA`	Scan data for one or multiple frames.
0x06	`SCAN_ERROR`	An unrecoverable error has occurred during scanning.
0x80	`RESPONSE`	The message is a response to a previous request.

Table 3.1: This table presents the message types used by the 3D scanner. The syntax and semantics of these message types is given in subsection 3.1.1.

line scanner module only accepts the message `GOTO_STATE`, and only sends messages of the type `SCAN_DATA` and `SCAN_ERROR`. Instead of sending messages and expecting a response, the line scanner modules send messages to the host without waiting for a response. This makes it possible to increase the throughput.

State name	Value	Description
`IDLE`	0x01	The device returns to this state when the connection state in the TCP server changes.
`SCANNING`	0x02	The device is scanning and returns scan data. This state can only be entered through the state `IDLE`.
`CONFIG`	0x03	The system can be configured. This state is only reached through `IDLE`.

Table 3.2: The line scanner module states with their main purpose and behavior.

3.1.1 Message definitions

This subsection briefly describes the content and structure of messages defined in Table 3.1. For additional details, refer to the implementations.

Message `RESPONSE`

The response message type is sent from the line scanner module to the host when it has received a message. Its structure is given in Figure 3.2. The encoding and value of the response message is generic, and is defined by its request.

0	1	5	6	7
Type	***Length***	***Payload***		
RESPONSE	*2 + response payload length*	*Request code*	*Response value*	*response payload*

Figure 3.2: The format of the RESPONSE message. The numbers above indicate the byte number. The request code indicates to which request the response belongs. The response value is one of the values defined in Table 3.3.

Response name	Value	Description
SUCCESS	0x00	The request was handled successfully.
ERROR_INVALID_STATE	0x01	The request is invalid because of the current state of the system.
ERROR_NO_MEM	0x02	Memory allocation failed. Memory allocation is often performed during configuration.
ERROR_OTHER	0x03	Unspecified error.
ERROR_INVALID_LENGTH	0x04	The configuration data had a wrong length.
ERROR_NOT_IMPLEMENTED	0x05	The requested functionality is not implemented.
ERROR_NULL	0x06	Null pointer exception.
ERROR_INVALID_PARAMETER	0x07	A given parameter is invalid.
ERROR_INVALID_CONFIG	0x08	The given configuration is invalid.
ERROR_BUSY	0x09	The system is busy. Request failed.
ERROR_NOT_INITIALIZED	0x0A	The given operation is invalid because the module is not initialized.

Table 3.3: The response values defined by the system.

Message GOTO_STATE

The GOTO_STATE message is sent from the host to a line scanner module and dictates the next state of the line scanner module. The defined states are found in Table 3.2. The valid state transitions are IDLE $\leftrightarrow$ SCANNING and IDLE $\leftrightarrow$ CONFIG. The line scanner module might reject this request even though the transition is valid, for example when the scanner is not initialized. The structure of the message is given in Figure 3.3.

Type	***Length***	***Payload***
GOTO_STATE	*4*	*New state*

Figure 3.3: The format of the GOTO_STATE message. The new state is one of the states defined in Table 3.2, and is encoded with four bytes.

3.1.2 Message CONFIG_CAMERA

The message type CONFIG_CAMERA makes it possible to configure the camera over TCP. Because the camera configuration calls are both for input and output, the response payload is also utilized. In this section, the lengths of the payloads for the different calls are not described. However, these follow straight forward from the API. The general format of such a message is given in Figure 3.4.

Type	***Length***	***Payload***	
CONFIG_CAMERA	*4 + config data length*	*Config function*	*Config data*

Figure 3.4: The format of the CONFIG_CAMERA message. The config function is defined in the type camera_config_type_t in Listing E.1. The config data is defined as the data in the type camera_config_t.

3.1.3 Message CONFIG_SCAN

The message type CONFIG_SCAN configures the image processing and scanner. It is therefore possible to set scan mode, image parameters, GPU execution properties etc. over TCP. The message is sent from the host to the line scanner. Currently the response message does not include any payload; however, for future use an extended error can be added. The structure of the scan config message is found in Figure 3.5.

Type	***Length***	***Payload***
CONFIG_SCAN	*1*	*Config data*

Figure 3.5: The format of the CONFIG_SCAN message. The config data is defined in the type scan_config_t in Listing E.2.

Message GET_IMAGE

The message type GET_IMAGE is sent from host to the line scanner module. The line scanner module sends a response with a camera image as payload. The returned image is encoded with the PNG format. This feature is not a critical component of the scanner, but very useful during setup and configuration. Using this feature, it is possible to verify that the image size and region of interest captures the laser line.

Message `SCAN_DATA`

`SCAN_DATA` is sent from the line scanner to the host when the line scanner is in the `SCANNING` state. It does not expect a response from the host. The structure of the message has been kept simple: For all scan modes, the definition is the same as shown in Figure 3.6. Because some scan modes do not utilize all fields, there might be a performance gain in removing unused fields for some of the defined scan modes. However, a better approach is to define new message types.

The line scanner module is allowed to buffer scan data, and is therefore allowed to send image parameters from multiple frames in the same message. The number of frames included in a message is determined by the image buffer size, residing in the field *scan config*. The length of the message does also depend the width of each image. Because only each second column is used in each image, the number of data points per image parameter is $buffer\_size \cdot width/2$. Each parameter image parameter is encoded with a 16 bit value. For the case of the RGB data, each data point therefore contains a 16-bit triplet. The GPU and total execution represent execution time in milliseconds and are benchmark parameters encoded as 4 byte floats each. Because the scan configuration contains 46 bytes, the total payload length is $L = 46 + 7 \cdot 2 \cdot buffer\_size \cdot width/2 + 2 \cdot 4$.

Type	***Length***	***Payload***							
SCAN_DATA	*L*	*Scan config*	*Height*	*Intensity*	*Reflectance*	*Scatter*	*RGB*	*GPU execution*	*Total execution*

Figure 3.6: The format of the `SCAN_DATA` message. The length of the payload depends on the image width and image buffer size. The scan configuration is defined in the type `scan_config_t` in Listing E.2. The values of image parameters are defined in section 3.2. The values of the benchmark properties are defined in subsection 4.4.2.

The larger the image buffer size, the smaller impact the message header and scan configuration data has on the bandwidth usage. Therefore, the highest bandwidth is required when only one image is included per message. For this case, when the image is 1280 pixels wide, the message length is 9019 bytes. Because a Gigabit Ethernet link has a maximum bandwidth of approximately 120 MB/s, this message type definition allows up almost 14000 scan data messages per second. When three line scanner modules share share this link, each camera is then able to output above 4000 samples each second.

Message `SCAN_ERROR`

A `SCAN_ERROR` message is sent from the line scanner module to the host when an error occurs during scanning. The message does not expect a response. When this message is sent, the line scanner module returns to the `IDLE` state.

3.2 Definitions of image parameters

Four parameters should be extracted from each image containing a laser line. The parameters reflect some property of the line at every column. Images where the LED is turned on should be used for color extraction. For some of the parameters there exist many different ways to calculate these.

In general, the simplest (assumed to be fastest) solutions are chosen, except for the case of calculating the height. The described definitions correspond to the definitions used by SINTEF's 3D scanners. The purpose of this section is describe some of the strengths and weaknesses of these definitions, and suggest how the parameters can be described in a more precise way. Because the objective of this thesis is not to optimize the quality of the features, this section should be kept as a reference for further development.

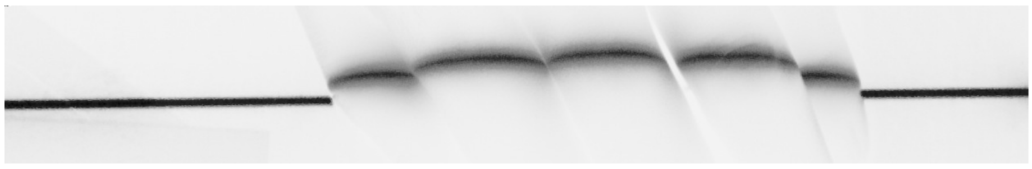


Figure 3.7: An inverted monochrome image of 5 fingers placed in the camera's line of sight.

3.2.1 Height

The height parameter reflects where the laser line is found in the image. The suggested definition of the height parameter is given as

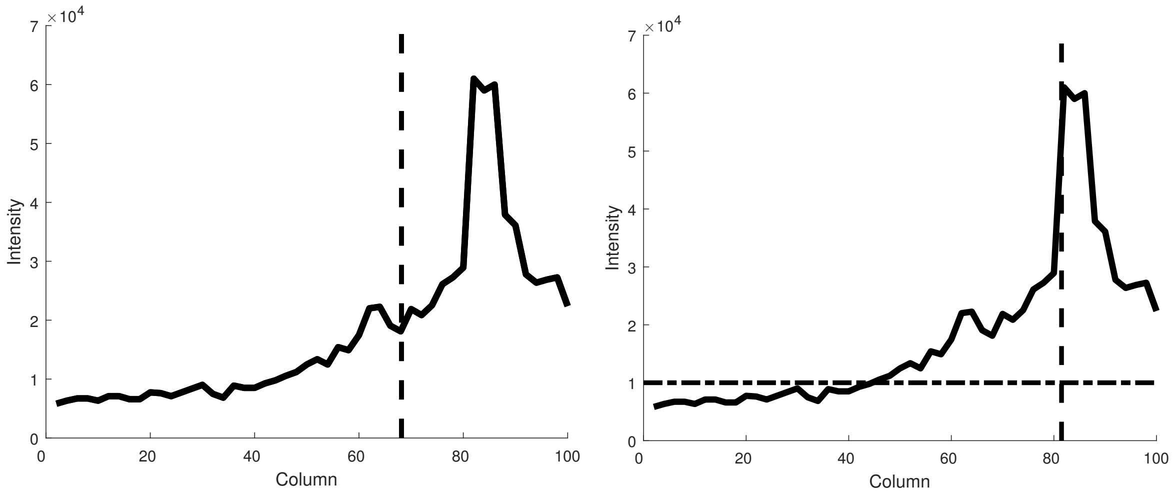
$$\text{Height} = \frac{\Sigma(x_i - \text{Threshold})^{\text{Power}} \cdot i}{\Sigma(x_i - \text{Threshold})^{\text{Power}}}, \qquad \forall(x_i - \text{Threshold}) \geq 0 \tag{3.1}$$

where x_i is the brightness of pixel i in a given scan line. For the case when a black frame is used (see subsection 1.5.5), x_i is the brightness of the laser frame subtracted with the brightness of the black frame. *Threshold* is a fixed integer parameter, and *Power* is a fixed floating point constant.

The height definition is a modified form of a weighted average. Because the definition requires many multiplications and summations, the calculation is quite costly. However, because the height profile is the most important parameter constructing the 3D model, it is important that is that the parameter is calculated accurately. A faster approach is to use the pixel with the maximum intensity, but this approach is more susceptible to salt and pepper noise [p. 338](Gonzalez and Woods 2007). The center of mass is a much more robust approach as this considers all pixels when finding the height. Because the distribution of salt and pepper noise is uniform, its effect will be canceled out. Another advantage with this approach is that it will give sub-pixel resolution.

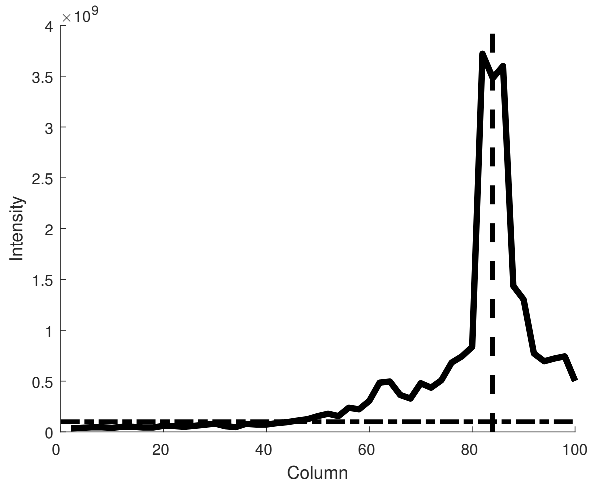
Two modifications are made to the general center of mass algorithm to find a more optimal result. First, a threshold is added which removes background noise. The power parameter decreases the effect of background noise, because the highest value will be weighted more.

Although it the center of mass approach is robust against uniform salt and pepper noise, the calculation will be affected by non-uniform background noise. There are two main approaches that can be used to solve this problem. First, the image region can be shielded for exterior light. Secondly, more advanced image processing techniques can be used. A possible solution is to estimate the background noise and subtract this from the measurement. This is the approach used in scan mode 4 given in Table 1.2.



(a) Height calculated using mass center.

(b) Height calculated using mass center with a threshold value.



(c) Height calculated with the squared intensity profile using a mass center with a threshold value.

Figure 3.8: These plots show the resulting height parameter when different approaches are used.

3.2.2 Intensity

$$\text{Intensity} = x_{\text{Height}} \tag{3.2}$$

The intensity reflects the intensity of the laser line. The current approach uses the brightness of the pixel located at the height. This makes the parameter susceptible to both uniform and non-uniform noise. A more robust approach is an average of multiple pixels, in addition to use the ratio between the intensity of the line and background.

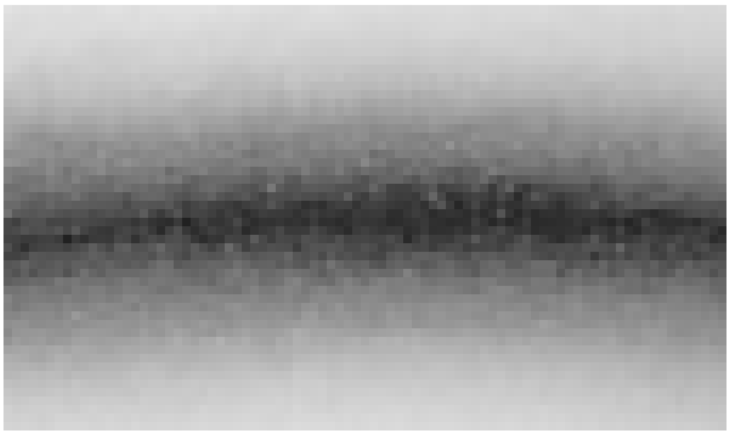


Figure 3.9: A close-up of the image shown in Figure 3.7. At this detail level, salt and pepper noise is revealed.

3.2.3 Reflectance

$$\text{Reflectance} = \frac{\Sigma x_i}{\text{Image height}} \qquad \forall(x_i - \text{Threshold}) \geq 0. \tag{3.3}$$

The reflectance is defined as the sum of the intensities of the pixels for the entire column. The value returned will therefore depend on the light conditions. This dependency can be removed by subtracting the background noise using a black frame. The parameter is made invariant to image height by returning the average intensity.

3.2.4 Scatter

$$\text{Scatter} = x_{(\text{Height}+\text{Scatter offset})} \tag{3.4}$$

The scatter should reflect how wide the laser line is perceived by the camera. This is affected by the light conditions and the reflectance of the object measured. When only one pixel some distance from the calculated height is used, the calculated value will be very susceptible to salt and pepper noise. A better approach would therefore to use an average of some number of pixels around the height.

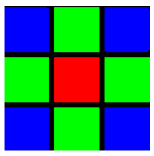
The parameter can also be made invariant to light conditions by returning a ratio between the line intensity and the intensity some distance from the line. This approach will therefore be similar to the more advanced approach of calculating the intensity.

3.2.5 Color

The color of the line is determined at the height of the line. Because only even columns are used for this analysis (see subsection 4.2.3), there exist only two definitions of the color extraction, depending if the height is even or odd.

For denotational simplicity, we define a stencil that is a 3 by 3 pixels region around the pixel of interest. The pixels are numbered 1-9 from top left to bottom right. This allows us to define the different color components as following:

$$R_{even} = x_5 \tag{3.5a}$$
$$G_{even} = \frac{x_2 + x_4 + x_6 + x_8}{4} \tag{3.5b}$$
$$B_{even} = \frac{x_1 + x_3 + x_7 + x_9}{4} \tag{3.5c}$$
$$R_{odd} = \frac{x_2 + x_8}{2} \tag{3.5d}$$
$$G_{odd} = x_5 \tag{3.5e}$$
$$B_{odd} = \frac{x_4 + x_6}{2} \tag{3.5f}$$
$$\tag{3.5g}$$


(a) Even stencil.

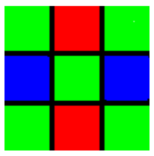

(b) Odd stencil.

Figure 3.10: The stencils used to extract color. The center pixel is located at the given height for the cases when it is even and when it is odd. The debayering filter used is given in Equation 3.5.

Chapter 4

Line scanner module implementation

This chapter describes the implementation of a line scanner module. Each of the most important parts of the implementation is devoted a section: The application architecture, image processing, error detection and recovery, and the performance measurement framework. The language used in this chapter has in some cases a high technical detail level. Therefore, it is expected that the reader has some general programming experience and CUDA GPU programming knowledge. If the reader has no previous experience with CUDA GPU programming, it is highly recommended first reading Appendix A. This appendix presents all the required GPU programming vocabulary necessary to understand the details in this chapter.

The presented implementation is able to run on both the Jetson TK1 and Jetson TX1. Appendix B describes how to install the prerequisite software, and Appendix D describes how the line scanner module application can be launched, configured, and tested. For additional implementation details, refer to the implementation itself.

4.1 Application architecture

The line scanner module application is an event driven application. That is, operations are executed when either a TCP event or camera event arrives. A simplified communication diagram is shown in Figure 4.1. The events, numbered 1-11 are described in Table 4.1.

The *TCP Server* launches a thread that continuously polls for data in the background. When a message is received, the TCP event is forwarded to the state machine. Depending on the message type and state (see Table 3.1 and Table 3.2), the events are forwarded to other modules. This might either start or stop scanning, or configure the image processing and camera.

When the system is in the `CONFIG` state, the messages are forwarded to the *Config event broker*, which decodes and validates the lengths of the messages, before the events are forwarded to the camera or scanner. When a configuration message arrives at either the *Scanner* or the *Camera*, the configuration is applied (or rejected), and a response is sent back. In section 3.1 it is discribed how to configure the Scanner and Camera over TCP.

The TCP server is written with performance in mind. There are no requirements of incoming messages to be handled fast, and when the system is in the `SCANNING` state, it is desirable that the TCP server affects the scanner as little as possible. Therefore the TCP server calls `pthread_yield()` after each time some data is received, as shown in Listing E.9. This transfers execution back to the scanner after each TCP receive.

When the system is enters the `SCANNING` state, a thread is launched in the Scanner module. This thread continuously polls the Camera module for images. When an image arrives, this is denoted as a camera event. The Scanner module buffers the images before they are sent to the Image processing module. When images are finished processing, the Image processing module sends the results to the TCP Server. Alternatively, the asynchronous image retrieval functionality of the FlyCapture2 SDK could have been used. However, it was found that this resulted in a slower maximum frame rate.

The performance of the application is also increased by modifying the scheduler options. Listing E.10 shows how the scheduling priority of the application is changed to maximum. This will minimize the influence of other applications on the line scanner module application. Note that in order to launch the application with scheduler options enabled, the application must be launched with super user rights.

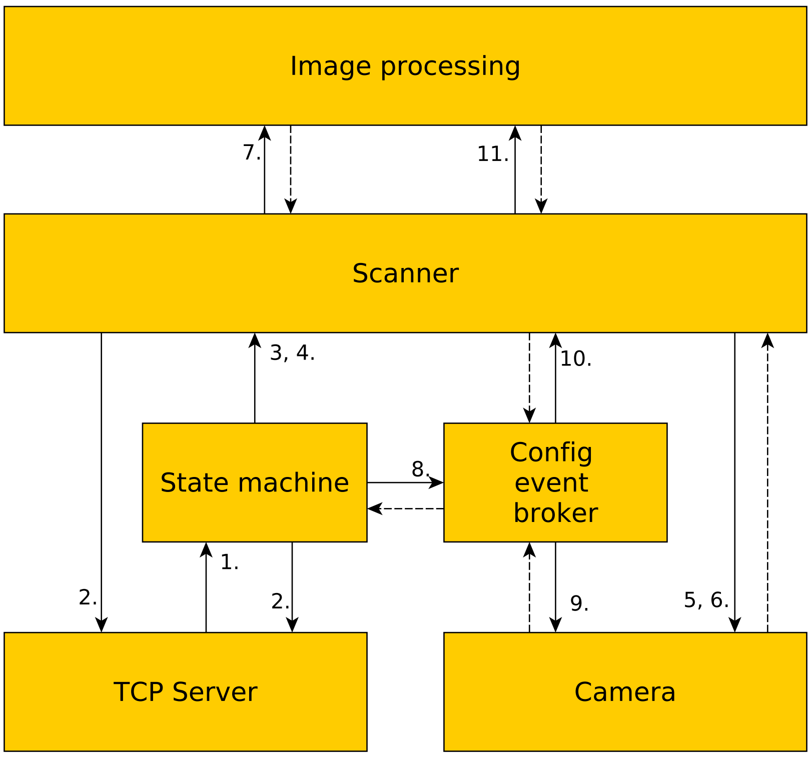


Figure 4.1: Simplified application architecture. The numbered events are explained in Table 4.1.

Event	Name	Description
1	`state_machine_on_tcp_evt`	Called each time data is received. The state machine forwards the event to the correct module.
2	`tcp_server_send_message`	Called when either a command result or scan data should be returned to the client.
3	`scanner_start`	Called when the scanner should start. This starts the camera polling loop.
4	`scanner_stop`	Called when the scanner should stop. This terminates the camera polling loop.
5	`camera_start_capture`	Called when the camera should start capturing images. It is now possible to retrieve images using `camera_get_buffer()`.
6	`camera_stop_capture`	Called when the camera should stop capturing images.
7	`img_processing_start`	Starts an image processing task. The call is nonblocking, and the task will therefore be performed in the background.
8	`on_config_evt`	Called when a config event is received. The event is decoded and forwarded to the correct module. Returns success value.
9	`camera_set_config`	Called when the camera is configured. Returns success value.
10	`scanner_set_config`	Called when the scanner is configured. Returns success value.
11	`img_processing_set_config`	Called when the image processing module is configured. Returns success value.

Table 4.1: Description of the events shown in the application architecture diagram in Figure 4.1.

4.2 Image processing

The system architecture is designed to allow image processing tasks to be performed both asynchronous and in parallel. This allows the system to run with a high frame rate and to be scalable to more advanced GPUs. This section describes how this is achieved using CUDA with the onboard GPU. The last part of this section describes how the different features are extracted from the images.

4.2.1 Image buffering and memory management

The incoming images are stored to subbuffers corresponding to the image type defined by a scan mode before they are processed by the GPU. Therefore, for scan mode 4, there exist three subbuffers: One buffer for laser images, one for blackframe images and one for color images. Because GPU kernels then process multiple images at the time, this results in fewer kernel launches and less and faster memory transfers. This reduces the fraction of sequential code, and by strong and weak scaling (Equation A.1, Equation A.2), this increases the throughput.

The tradeoff with a large image buffer size is an increased scan result latency. Because scan results are available only when all images are received and processed, the maximum latency depends not only on the buffer size, but also on the camera frame rate. When a scan mode is used with a image sequence length of N, the camera framerate is f_c, the buffer size is B, and the image processing time is t_i, the maximum latency t_l is expressed as:

$$t_l = \frac{B \cdot N}{f_c} + t_i. \tag{4.1}$$

When it is assumed that C image processing tasks can execute in parallel, the image processing tasks can last at maximum C times the image buffering time. Therefore, the latency can be expressed as:

$$t_l < (1 + C)\left(\frac{B \cdot N}{f_c}\right). \tag{4.2}$$

Images are buffered either synchronously or asynchronously. When the images are buffered synchronously, the application is blocked until the memory transfer is completed. Therefore, the asynchronous configuration is preferred because it allows the CPU to continue with other tasks in the meanwhile. For benchmarking purposes explained later, both variants are available through compile time flags `SCANNER_BUFFER_SYNC` or `SCANNER_BUFFER_ASYNC`.

The image buffering framework sets a limitation on the maximum frame rate of the system. When images are buffered synchronously, the maximum frame rate is given by the framework processing time t_f, only. When images are buffered asynchronously, the image buffering time must be considered. When it is assumed that there is only one memory transfer at the time, then the maximum framework frequency f_f cannot exceed the image buffering time t_b. Therefore, when the image buffering framework time is considered as the bottleneck of the system, the maximum frame rate is given as

$$f_f = \frac{1}{\max(t_f, t_b)} \tag{4.3}$$

The image buffers and scan results can be stored in either host or device memory. In section A.5 it is described that the capabilities of the GPU affects which configuration is the best. Therefore, the application is designed to allow different configuration options for increased portability. The location of the image buffers and output buffers can be set at compile time with the compile time flags given in Table 4.2.

When images are buffered, pixel index calculations become more complicated. In addition to the row and column, also the image index must be used. Two different approaches are shown in Figure 4.2 where the second approach is more effective when

Flag	Description
FRAME_BUFFER_STORE_DEVICE	Images are buffered in device memory. This does not require an additional copy operation because the images must be buffered anyways. This flag cannot be set when FRAME_BUFFER_STORE_HOST is set.
FRAME_BUFFER_STORE_HOST	Images are buffered to host memory. The host memory is pinned to allow the GPU kernel to access the images. This flag cannot be set when FRAME_BUFFER_STORE_DEVICE is set.
OUTPUT_BUFFER_DEVICE	The scan results are stored in device memory. This requires an additional copy operation when the image processing is complete. This flag cannot be set when OUTPUT_BUFFER_HOST is set.
OUTPUT_BUFFER_HOST	The output buffer is stored in pinned host memory. This flag cannot be set when OUTPUT_BUFFER_DEVICE is set.

Table 4.2: Compile time flags configuring where images and output buffers are stored.

this function is used often. Accessing the pixels is also often done in a predefined pattern, which is seen in Figure 4.3. Later in subsection 4.2.3 it described how this can be used to simplify pixel indexing even more.

```
#define GET_INDEX(row, column, image) (WIDTH*HEIGHT*image + row*WIDTH + column)
```

(a) A naive way of calculating the index of a pixel. The multiplication WIDTH*HEIGHT must be performed every time.

```
#define GET_INDEX(row, column, offset) (offset + column + WIDTH*row)
```

(b) A more optimized way of finding the index of a pixel. The image offset must now only be calculated once.

Figure 4.2: Indexing pixels in an image buffer can be done in different ways. The approach will affect the number of required multiplications.

Allocation and deallocation of image and output buffers is a source of unclear and hard to maintain code. Later is explained how image processing tasks is performed in multiple streams, which results in an even larger amount of buffers. In order to keep control of all buffers, those have been placed in a structure called `stream_data_t`. This does not only increase cohesion, it also simplifies code maintenance and memory management.

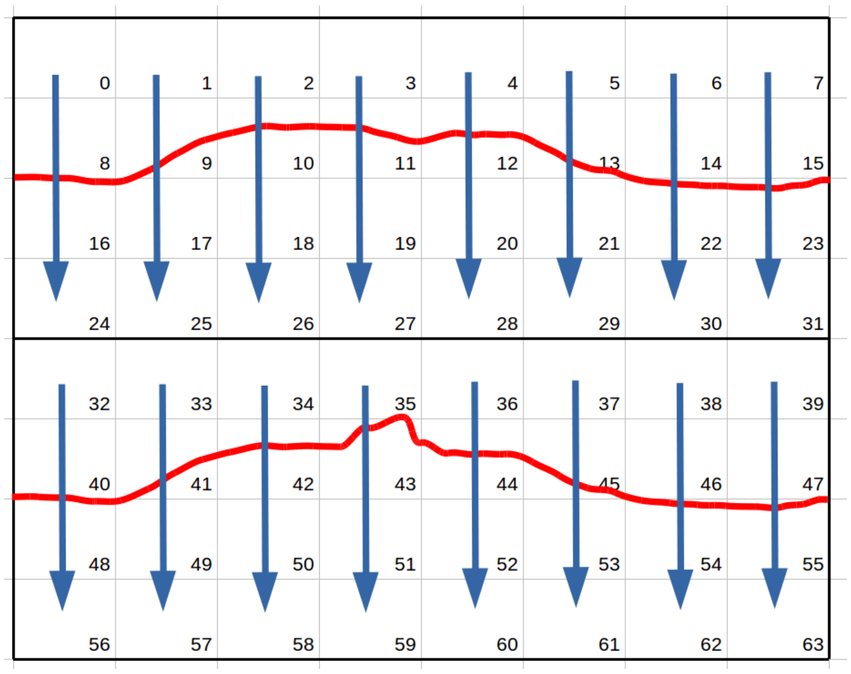


Figure 4.3: When image parameters, such as height, is extracted from images, the pixels are often visited in a predefined pattern. This figure shows how height extraction iterates over the pixels columns when frames are buffered.

4.2.2 Parallel asynchronous image processing

The application performs all image processing procedures asynchronously and in parallel. Different GPU threads handle each of the columns of the received images. Dependent on the used GPU and image processing parameters, multiple image processing tasks might execute in parallel. This is achieved by committing all operations into CUDA streams and keeping separate memory buffers for each stream. For each stream, the following steps are performed in order:

1. Images are buffered. This occurs each time an image arrives. Depending on the configuration images are buffered either to host or to device memory.
2. The image processing kernel is executed. The execution parameters, that is, the block and thread sizes, are retrieved from the image processing configuration.
3. The output data is transferred back to host memory. This task is only executed if `STREAM_DATA_OUTPUT_BUFFER_DEVICE` is set.
4. A CUDA callback is launched where the TCP server sends the output data back to the host.

The maximum available frame rate with this pattern depends on the image buffering time, TCP send time, and image processing time. The theoretical maximum frame rate, only considering TCP send time, is achieved when the camera duty cycle is equal to the

TCP send time. Therefore, when the TCP send time is expressed as t_{TCP}, the maximum frame rate is

$$f_{TCP} = \frac{1}{t_{TCP}}. \tag{4.4}$$

The maximum frame rate only considering image processing depends on the capabilities of the GPU and the length of the image sequence for the selected scan mode. The highest performance is achieved when image processing executes concurrently with image buffering (see section A.5). If this is true, and the selected scan mode has an image sequence length L, the image processing may execute during this entire length. When the maximum number of concurrent kernel executions is expressed as C, and the kernel execution time per image is t_k, the maximum frame rate of the image processing is expressed as

$$f_i = \frac{L \cdot C}{t_k}. \tag{4.5}$$

Because the image processing framework is asynchronous, there must be a synchronization primitive present to prevent images to be buffered to an ongoing image processing task. This is handled by the Scanner module. After an image processing task is launched, `cudaStreamSynchronize()` is called on the next stream. This will block the CPU until it is possible to start buffering images to that stream. A detailed view of the buffering of images and stream synchronization is given in Figure 4.4.

1. Start the scanner. Initialize stream data. Set the image sequence index to 0. Set current stream to 0.
2. An image arrives. The image type is retrieved.
3. Add image to the frame buffer corresponding to the current stream and image type. Images are buffered asynchronously. Jump to the next image sequence index.
4. Check if image processing should start. If the image buffer is full, and the image type is equal to the last image type of the sequence, jump to the next point. Else, jump back to point 2.
5. Start image processing.
6. Switch to the next stream and synchronize. Jump back to point 2.

Figure 4.4: The sequence of events used to buffer images and synchronize CUDA streams. Error detection and handling is omitted here, but is explained in Figure 4.6.

4.2.3 Image parameter extraction

This subsection describes how the different parameters described in section 3.2 are extracted from images. The code examples given are only extractions from the implementations of the different scan modes. The actual code is therefore a combination of these code segments with slight modifications. Nevertheless, the code segments give enough insight into the parameter extraction.

To reduce the computational load, only every second column of the image is used. Because the image is received as a raw image, this means that that only the columns where red pixels are present will be used (see Figure 2.1). Therefore, there is no need to interpolate the red color, and retrieve the line parameters of the interpolated columns. This simplification is therefore valid.

Another applied optimization is precalculating pointers to sub-buffers in the scan data message. The output buffer has a fixed structure, and therefore pointers to the fields of the scan data buffer are fixed. By precalculating these pointers at initialization time, execution time is saved. The pointers are stored in the structure called `scan_output_buffer_t`, and are initialized in `stream_data_init()`. The scan output buffer structure is provided as a p̊arameter to the image processing kernels.

Image processing configuration and parameter retrieval

Each image processing kernel requires some predefined configuration to be able to extract the image parameters. This configuration includes image sizes, image buffer size, scatter distance, etc. This configuration must be accessible by the GPU and is therefore sent as a parameter to the GPU kernel function. In addition, each thread in the kernel must also know on which memory it should operate. This is done often, and it is therefore important that this is done effectively. That is, with the least amount of costly arithmetic operations such as division and modulo operations.

Listing E.4 shows how these parameters are found for scan mode 4. Because multiplication with powers of 2 are substituted with logical shifts, only the calculation of `scan_line`, `local_scan_line`, and `image_offset` use the multiplication engine. Even though they use costly operations, this has been improved drastically compared to the previous implementation. Previously the image index was calculated using an modulo operation as shown in Figure 4.5a. Now this is done cheaper by utilizing multidimensional kernel launches. In Figure 4.5b it is shown that the image index is retrieved from the y-dimension.

Height extraction

The height is defined in Equation 3.1. Calculating the height is the most computational expensive feature extraction. Not only is it necessary to use the entire column, it also requires a large amount of multiplications and exponentiation. Therefore, the implementation has applied a number of optimization techniques to speed up this calculation.

- Pixels are visited only once. Both the numerator and denominator of the definition are calculated in the same loop.

```
const uint32_t scan_line       = blockIdx.x*blockDim.x + threadIdx.x;
const uint32_t image_index     = scan_line / NUM_SCAN_LINES_PER_IMAGE;
const uint32_t local_scan_line = scan_line % NUM_SCAN_LINES_PER_IMAGE;
```

(a) The local scan line and image index are found using expensive arithmetic operations. The kernel is launched in only one dimension, that is blockDim.y = 1.

```
const uint32_t image_index = blockIdx.y;
const uint32_t local_scan_line = blockIdx.x*blockDim.x + threadIdx.x;
const uint32_t scan_line = image_index * NUM_SCAN_LINES_PER_IMAGE + local_scan_line;
```

(b) When using multidimentional kernel launches, the calculation of the local scan line and image index can be simplified. Now blockDim.y = image_buf_size.

Figure 4.5: Two ways of finding `scan_line`, `local_scan_line`, and `image_index`. `local_scan_line` represents the column of the image divided by two, while `scan_line` is the global column number divided by two.

- Index calculation is simplified because pixels are visited in a fixed pattern. This reduces the number of multiplications needed for index calculations.
- Instead of performing exponentiation, the intensity values are retrieved from a lookup table. Because all intensity values are retrieved from a lookup table, the threshold is a builtin feature of this table.

The lookup table is an array of size 2^{12}, which represents 12 bit of pixel depth. The lookup table is placed in CUDA texture memory which gives fast memory access (see section A.3). Alternatives to using texture memory are constant memory and shared memory. These were found to be less efficient: Constant memory serializes accesses and therefore decreases execution speed, and shared memory requires a lot of resources which limits kernel and thread execution parallelism. An example of the texture initialization is shown in Listing E.3.

The height value is returned as a 16 bit unsigned integer (see section 3.1). To utilize the entire bitfield, the height is scaled from `MAX_HEIGHT` to 2^{16}, which is seen on line 32 in Listing E.5.

Reflectance calculation

The reflectance is defined in Equation 3.3. Because it iterates over the same pixels as the height extraction, the same loop can be used for this purpose. In Listing E.5 the computation of the reflectance is shown.

Intensity and scatter extraction

The intensity and scatter have rather simple definitions described in Equation 3.2 and Equation 3.4. Because the calculated height or scatter height might be right between two red pixels in the Bayer filter, the value might need to be interpolated. Alternatively, the height could have been rounded to the nearest even-numbered value. The implementation of intensity and scatter extraction is shown in Listing E.6.

Color extraction

The color extraction is defined precisely in its definition in Equation 3.5. Therefore, the implementation matches the definition quite closely. As seen in Listing E.7, the border pixels are ignored which reduces the number of execution paths. To optimize pixel indexing, only the first index of the used stencil is indexed using the macro `GET_INDEX()`.

4.3 Error detection and correction

In subsection 1.5.3 it is described that one of the major goals of this master thesis is to increase the reliability of the system. One of the factors influencing the reliability is its fault tolerance. This section will first describe how and which errors the line scanner module is able to detect. Next, it is described which mechanisms are used to correct these errors. Because fault tolerance in general decreases the performance, some of the detected errors are defined as unrecoverable. Which of the failures, and why they are defined as unrecoverable is given in the last part of this section.

4.3.1 Detectable errors

The fault tolerance of the line scanner module is based on error detection. The following list describes which errors that are detected, and how:

- Image validity. When an image is received, it must be validated against the image processing configuration. This means that the pixel format and dimensions must be correct. In addition the image type must match the next expected image type from the image sequence corresponding to the current scan mode.
- An image is not received, or received at an unexpected time. The camera is configured to enable embedded time stamps. Therefore, it is possible to compare the timestamp of an incoming image to the previous received image. In this way, the time difference between the two last images is measured.
- Internal GPU errors. CUDA kernels may fail internally, both synchronously and asynchronously. These types of errors are detected after each CUDA call.
- Configuration errors. Each module that can be configured, that is either the scanner or camera validates the configuration parameters. Therefore, a configuration error will be detected.
- Messages arriving when the line scanner module is in an invalid state. The valid state transitions, and which messages that are accepted in each state is described in Table 3.2.

4.3.2 Non-detectable errors

A configuration mismatch between the camera and image processing is not detected at the time the system is configured. However, when images start to arrive, this results in an image validation failure. Therefore, a configuration mismatch is not entirely a non-detectable error.

4.3.3 Recoverable errors

Some errors are recoverable in some degree. This might mean that a suboptimal solution is used, or that the request is denied. The list of recoverable errors and how they are recovered are is found below:

- An image is not received, or it is received late. When an image is received, it is checked if frames are dropped. If this is the case, the previous received image is used for image processing instead. This however, will affect the correctness and give a suboptimal output.
- Configuration errors. Configuration errors are always recoverable, because the configuration is denied in case of an invalid configuration.
- Messages arriving when the line scanner module is in an invalid state. When the system tries to transition to a new invalid state, this operation is rejected, and therefore recoverable.

4.3.4 Unrecoverable errors

When an unrecoverable error is detected, and error message is returned (see Table 3.1), and the system resets itself to the IDLE state. The list of unrecoverable errors is found below:

- Image validity. A configuration mismatch between scanner and image processing is only recoverable by reconfiguration, which must be done by the host.
- An image is not received, or it is received late. Previously it was described that this type of error could be recovered into some degree. However when too many images are dropped, it is unlikely that the system will catch up. Therefore, this is seen as an unrecoverable error.
- Internal GPU errors. The nature of an internal GPU error is often unknown. It might be a memory error, a lack of resources, etc. Because of this and the fact that internal GPU errors will stall the image processing, it is seen as an unrecoverable error.

4.3.5 Error detection and recovery in the scanner module

The sequence of events presented in Figure 4.6 present how error detection and recovery is done. For the case that an unrecoverable error occurs, the system sends an error message resets itself as described earlier. Notice that this sequence is similar to the buffering sequence presented in Figure 4.4. The actual implementation is a combination of these two.

1. An image arrives.
2. Image format validation. If the format of the retrieved image does not match the format of the image processing configuration, this is an unrecoverable error.
3. The time difference between the current and previous image is retrieved. If it fails to retrieve the time difference, this is an unrecoverable error.
4. If the time difference is larger that 50 frames, this is defined as an unrecoverable error.
5. If the time difference between the current and previous frame is correct, that is the time minus the cycle time is less than a threshold, go to point 6. Else jump to point 9.
6. Retrieve the image type. If it fails, this failure is unrecoverable.
7. If the image type was different from the expected image type, this is an unrecoverable error. Notice that it is not possible that a frame was dropped because the time difference was checked earlier.
8. Backup the received image.
9. Buffer images. Use the last received image. For each image added, subtract the cycle time from the frame time difference. Continue until the frame time difference is below the threshold or negative.

Figure 4.6: This sequence of events show the error detection and recovery mechanisms in the Scanner module. Buffering details are left out, but are presented in Figure 4.4.

4.4 Performance measurement framework

The performance of the system is determined by the latency and throughput of the scanner framework and image processing algorithms. The throughput is the most interesting measure, because it affects the maximum accuracy of the scanner. The maximum throughput is determined by the camera, the scanner framework latency, the GPU execution time, and the TCP sending time. This section describes how a measurement framework is set up to allow retrieval of these values.

4.4.1 Measuring scanner framework execution time

The scanner framework determines how fast incoming images are validated and buffered. Because every image first arrives in the scanner framework, it is impractical to measure and store the time used in the framework for every image because this would lead to a lot of data. In addition, because the TCP connection handle is used asynchronously, the scanner framework cannot send benchmark results back over TCP without risking network data corruption. Therefore, another mechanism is used to measure the latency of the scanner framework.

Using a GPIO pin of the board (referred to as *Jetson Ready* in section 2.3), the Jetson can signalize when the scanner framework is entered and when it is left. The framework processing time can therefore be measured by monitoring this pin. The GPIO pin is interfaced through the GPIO driver located at `/sys/class/gpio`. Before it is used a string value must be written to the `<driver_path>/export` file descriptor. Setting the value is done by writing the strings "0" or "1" to the corresponding `<driver_path>/gpio<pin>/value` file. For more details about GPIO handling, refer to the GPIO implementation files.

Previously it was stated that asynchronous image buffering is the the preferred buffering implementation. However, described in Equation 4.3, the image transfer time must be known in order to find the maximum frame rate of the system. The asynchronous memory transfer time could have been measured similar to how GPU execution is measured. However, this approach is complicated and adds some overhead. Therefore, a better approach is to estimate the memory transfer time by comparing the synchronous and asynchronous buffering implementation.

4.4.2 Measuring GPU execution time and TCP sending time

The GPU execution time and TCP sending time are measured with CUDAs event framework. These functions are named `cudaEventRecord()` and `cudaEventElapsedTime()`. The event recording function is appended to a CUDA stream, and are therefore executed when the queue has executed the previous elements. At a later time step the `cudaEventElapsedTime(t1,t2)` can be called to determine the time difference between those events. This mechanism for performance measurement is advantageous over CPU time measurement because it is executed by the GPU itself, and does therefore not interfere with CPU execution. In addition, this mechanism allows independent timers for each CUDA stream, which simplifies timer management.

1. Call `cudaEventElapsedTime(prev_startEvent, prev_kernelFinishedEvent)`. This returns the previous GPU processing time. The alternative is to place this function call as the last element of this sequence. However, this blocks the system until image processing is complete, reducing the maximum throughput drastically.

2. Call `cudaEventElapsedTime(prev_startEvent, prev_tcpFinishedEvent)`. This returns the previous total processing time. The TCP processing time is found by subtracting the GPU processing time.

3. Call `cudaEventRecord(new_startEvent)` to store the start time of GPU processing.

4. Launch the GPU kernel.

5. Call `cudaEventRecord(new_kernelFinishedEvent)`. This function will be called when the kernel is finishes executing.

6. Call `cudaStreamAddCallback()` to add the TCP sending function to the stream.

7. Call `cudaEventRecord(new_tcpFinishedEvent)`. This function will be called when the callback is finished executing.

Chapter 5

Host control system implementation

This chapter describes an implementation of a simple host control system developed in LabVIEW. First, the architecture is described. Next, it is described how the scan data is encoded. Finally, it is described how the scan results are visualized in real-time using the IMAQ package. The LabVIEW host implementation is rather a proof of concept than an application with all functionality included. Therefore, line scanner module configuration is not available through LabVIEW. However, this can be done through for example the configuration tool (see section D.2. Refer to section D.3 for a description on how to use the application.

5.1 Application architecture

The host control system was developed in LabVIEW (*Laboratory Virtual Instrument Engineering Workbench*), which is a development environment for creating applications for instrumentation control and data acquisition. Applications in LabVIEW are developed using the graphical programming language ”G” and the application development is based on data flow. The host application could have been developed in any type of programming environment. LabVIEW was chosen because it was used for the previous scanner developed by SINTEF. In addition, LabVIEW provides packages for simple visualization.

The application is designed to use a producer/consumer architecture. This architecture implementation is achieved based on a queue. The producer loop acquires data and enqueues it into the common queue, while the consumer dequeues elements and uses it for i.e. visualization. Because both to enqueue and to dequeue are independent operations, this system architecture allows concurrent data retrieval and visualization (National Instruments 2016). An example implementation of the producer/consumer architecture is shown in Figure 5.1.

The LabVIEW Host implementation uses multiple queues, one for each type of scan result. That is, one queue for height, one for intensity, reflectance, scatter, and one for color. By decoupling the parameters, it allows each parameter to be processed independently and at different rates. This allows the application to run with an even higher concurrency.

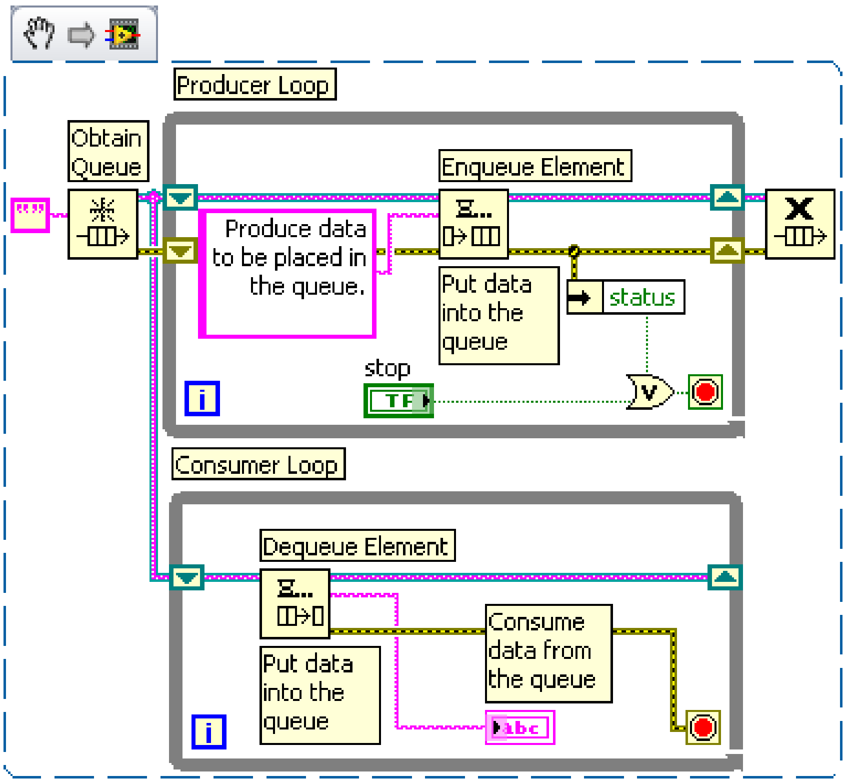


Figure 5.1: The producer/consumer architecture is dependent on a common queue. The two loops may run concurrently. This figure is taken from (National Instruments 2016).

5.2 Data extraction

Reading and decoding data is done in the producer loop. Because the `SCAN_DATA` message has a fixed structure (see section 3.1.3), the extraction of the image parameters is straightforward. First, it is necessary to read the configuration to determine the number of scan lines to retrieve. Next, the data is divided and decoded to the predefined parameters. Finally, the decoded data is put into their respective queues.

LabVIEW uses the less common byte order called big endian. This means that most significant byte is stored at the lowest address. Data received from the line scanner are encoded in little endian byte order, which is the opposite. Therefore, every multibyte value received from a line scanner module must be decoded. For the case of integer values, this means that the bytes must be reversed. For the case of floating-point values, this means that the value must first be interpreted as an integer, then the bytes must be reversed, and finally reinterpreted as a floating-point value. This operation is shown in Figure 5.2.

In order to speed up the data extraction some optimizations are applied:

- The TCP buffer is initialized outside of the producer loop. One-time initialization removes the necessity of reinitializing the buffer every loop iteration.
- Only one buffer is used for incoming data, and all data operation are done on parts

of this buffer. Therefore, there is no need for copying out data and/or additional memory. To achieve this, all image parameters are extracted as sub-arrays of this buffer. This optimization reduces both the memory consumption and prevents repeated memory allocation and deallocation.

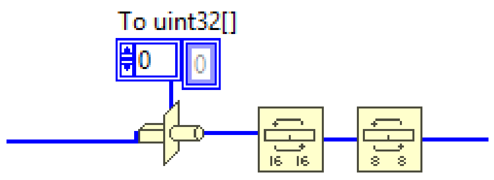


Figure 5.2: This figure shows how the endianness of the retrieved data is changed. The input data is a byte array. The bytes are reversed by first rotating two and two bytes, and then by rotating the individual bytes.

5.3 Data visualization

Data visualization is done in the consumer loop. Each loop iteration the image parameter queues are read and placed into the visualization buffers. The visualization buffers are used as input to the IMAQ library to visualize the incoming data.

The visualization buffers are arrays modified to work as first-in-first-out (FIFO) buffers. Therefore, for some given size, these represent the last received data. To achieve the FIFO buffer functionality, shift register functionality is used. The shift register functionality in LabVIEW passes values between each loop iteration (National Instruments 2015). Therefore, it is possible to store internal state, which in this case is the state of the buffer. Each loop iteration the oldest elements are removed, and new elements of the same size are inserted. An FIFO buffer example is shown in Figure 5.3.

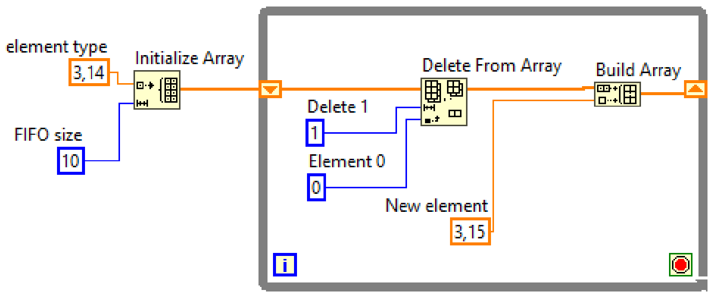


Figure 5.3: An example of a FIFO buffer implemented in LabVIEW. This FIFO is 10 elements long, and inputs the value 3.15 every loop iteration.

The IMAQ library provides simple functionality for visualizing arrays. By using the *ArrayToImage* block, this is done automatically. Even though the IMAQ library is easy to use, it restricts the LabVIEW Host to run on Windows only. LabVIEW is a cross platform solution, but the IMAQ library is not yet available for other operating systems.

The experimental LabVIEW host is implemented to visualize scan data from one line scanner module only, however it is easily expanded to support multiple data sources. In order to assemble data from multiple data sources, the retrieved data must be synchronized. In a graphical programming language, such as ”G”, this is done by inputting both data sources to the same VI as shown in Figure 5.4. The data sources are implemented as separate queues in the same fashion as done for the different scan parameters.

Building 3D models with data from multiple sources is harder. However such a task will depend on the requirements set by the system using the models. There exist a large number of 3D formats, and each model must be built in different ways. Common for all approaches is that each scan sample must be translated to some coordinate and stored in a point cloud, where the translation is obtained by a calibration routine.

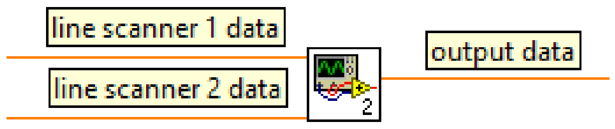


Figure 5.4: This figure shows how scanner data from multiple line scanner modules is synchronized. The VI depends on that both data sources are present before it executes.

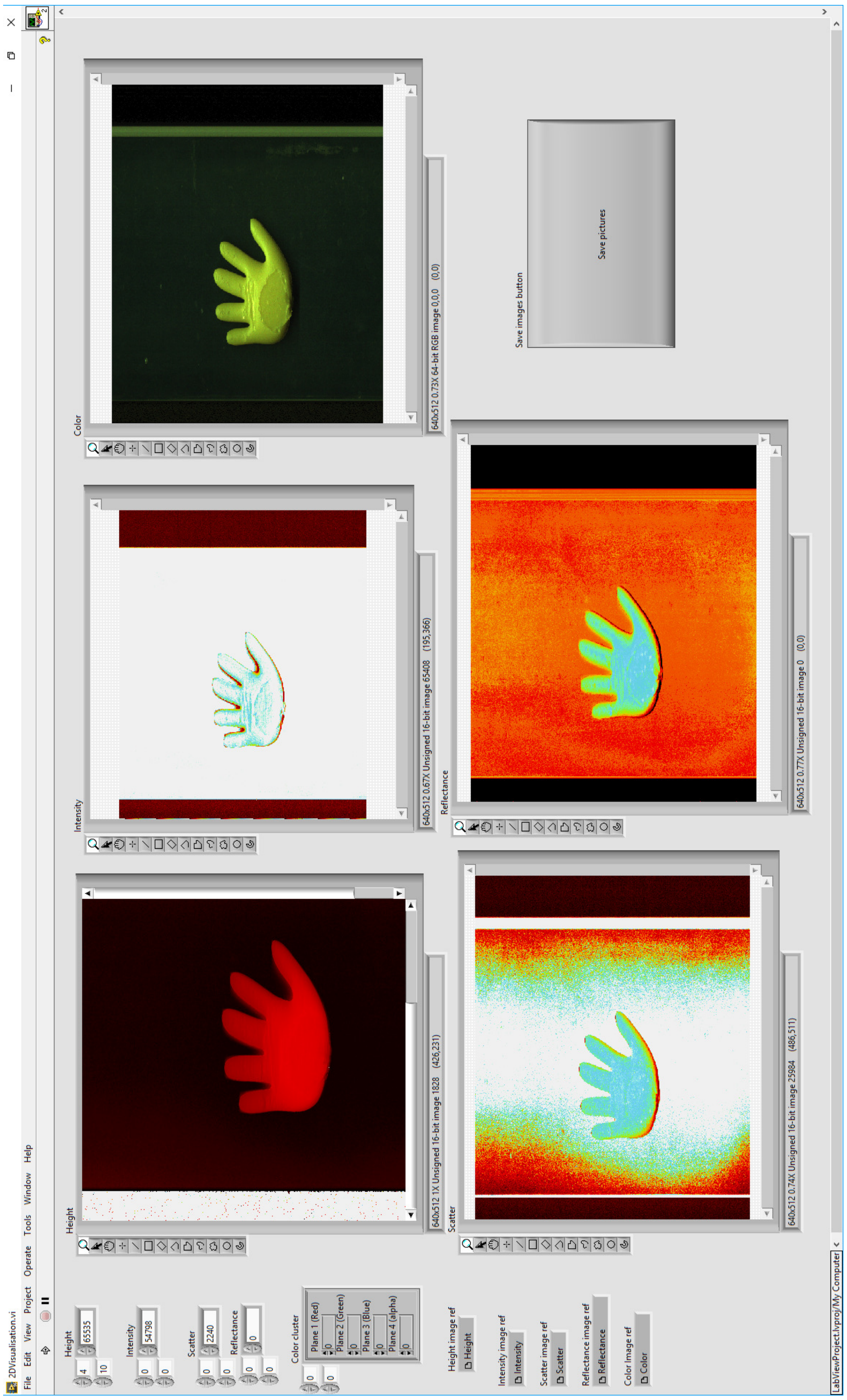


Figure 5.5: A sample of the visual output of *2DVisualization.vi*. The scan data is presented in the order height, intensity, color, scatter, and reflectance. The big button is used to save the samples.

Chapter 6

Test descriptions

This chapter is devoted to describe what features of the system are tested, which test parameters were used, and how the tests were performed. First, a description of the tests performed on the line scanner module is given. Next, it is described how the repeatability is tested. Finally, the last two sections of this chapter describe how the host control system was tested and how scans of salmon were performed.

6.1 Line scanner module tests

The line scanner module is tested to determine the maximum throughput of the system. There are four factors that limit the throughput: The camera, the image buffering framework, the TCP send time, and the image processing. Therefore, when combining the equations 4.4, 4.3, and 4.5, and expressing the maximum camera frame rate as f_c, the maximum frame rate of the line scanner module is expressed as:

$$f = \min(f_c, f_{TCP}, f_f, f_i) = \min\left(f_c, \frac{1}{t_{TCP}}, \frac{1}{\max(t_f, t_b)}, \frac{L \cdot C}{t_k}\right) \tag{6.1}$$

In order to find the bottleneck of the system, the image buffering framework, TCP send time, and image processing are benchmarked. Because all these test result depend on the memory configuration of the line scanner module, the optimum memory configuration is found first. Next, the image buffering framework limitations are found. At last the scan mode implementations are benchmarked to find the TCP send time and maximum rate of the image processing implementation.

6.1.1 Memory configuration

The memory configurations described in Table 4.2 affect how images are buffered, processed, and how the output data is stored. The concurrency of these operations depend on the capabilities of the GPU. Because there does not exist a tool to determine the concurrency at compile time, this property is checked manually using NVIDIAs Visual Profiler. All four possible memory configurations are tested on both the Jetson TK1 and

Jetson TX1. Because the concurrency depends on the size of the kernel, the tests are performed with frame buffer sizes of 1, 2, 4, and 8 images.

Ideally, the concurrency should be verified by running the image processing tasks at a very high frame rate, which will force the system to run at maximum concurrency. However, it turns out that the camera is the bottleneck of the system. Therefore a more computational expensive image processing kernel is used, which is shown in Listing E.11.

6.1.2 Scanner framework benchmarks

The scanner framework is benchmarked by attaching an oscilloscope to the *Jetson Ready* pin. By measuring the time this signal is 0 volts, the framework time is found. The test is performed 20 times. The scanner is configured as described in Table 6.1.

Even though the scanner framework uses asynchronous buffering, the image buffer time must still be known as described in Equation 4.3. A simple conservative approximation is to use the synchronous processing time. Therefore, the framework processing time is approximated to be the the maximum of the synchronous and asynchronous processing time.

Parameter	**Value**
Frame rate	200 FPS
Height	64, 128, and 192 pixels
Width	256, 512, 768, 1024, 1280, 1536, 1792, and 1920 pixels
Image buffer size	1

Table 6.1: Test setup parameters for scanner framework benchmarks. The image buffer size is set to 1 to include kernel launches in every measurement.

6.1.3 Scan mode implementation benchmarks

The project goals stated that the accuracy of the scanner should be measured. In terms of the line scanner module, this means the throughput and scan resolution. The scan mode implementation tests benchmark the image processing rate and TCP send time. For each scan mode, the optimum thread block size should be found, and it should found how the image buffer size affects the image processing speed on both the Jetson TK1 and Jetson TX1.

The tests are performed by the benchmarker described in section D.4. In order to minize network overhead, all test are performed with the line scanner module connected directly to the host computer. For each of the implemented scan modes, 300 samples of the total processing time are collected. The test setup is described in Table 6.2. Scan mode 4 is the most advanced implementation and therefore this implementation is subjected to more tests. It is found how the image size affects the image processing time and TCP send time. The test parameters used for these tests are given in Table 6.3.

Parameter	Value
Frame rate	400 FPS
Height	192 pixels
Width	1280 pixels
Threads per block	32, 64, 128, 256, and 512
Image buffer size	1, 2, 4, 8, 16, 32, 64, and 128
Number of CUDA streams	4

Table 6.2: Test setup parameters for line extraction implementation comparison. The image size parameters are similar to those used in a SINTEF's previous solution (Sture et al. 2016).

Parameter	Value
Frame rate	200 FPS
Height	64, 128, and 192 pixels
Width	256, 512, 768, 1024, 1280, 1536, 1792, and 1920 pixels
Threads per block	256
Image buffer size	8
Number of CUDA streams	4

Table 6.3: Test setup parameters for testing scan mode 4. The tests are performed at 200 FPS instead of 400 FPS due to camera limitations.

6.2 Repeatability tests

A high repeatability means that the same 3D model is obtained every time the object scanned. However, because it is difficult to compare 3D models quantitative, it is difficult to define repeatability in hard numbers. If direct comparison is used, it is required that the objects are perfectly aligned on the conveyor. Because of human error, and elasticity and unregular movements of the conveyor, this is hard to achieve. Calibration techniques are able to place scanned objects in a common reference frame, and therefore create close to perfect alignment. However, using direct comparison, the repeatability tests will still measure the quality of the calibration method, instead of measuring repeatability. In addition, the result will also depend on the nature of the scanned object.

Because the calibration techniques are not implemented and direct comparison is difficult, the repeatability is tested using a simple method. Instead of scanning a object multiple times, an object is only scanned once. The scanned object is a Rexroth bar, which an object with a very even height. The repeatability can therefore be defined as the measured height deviation of this bar. Because the Rexroth has a high and uneven reflectance, and therefore is harder to scan, it is well suited for repeatability tests.

Both scan mode 0 and scan mode 4 are used to test repeatability. These modes use different height extraction algorithms: Scan mode 4 uses black frame subtraction, while scan mode 0 does not. Scan mode 4 is therefore expected to yield a better quality height profile. The other scan modes are only variations of these two scan modes, and therefore there is no need in testing them as well. The number of samples included in the comparison should be as many as possible, and include the entire length of the Rexroth bar. Because scan mode 4 has a lower scan rate than scan mode 0 for the same camera frame rate, this implementation should be used to determine the number of samples.

For this test, simple linear calibration is used to be able to express the repeatability in millimeters. This linear calibration utilizes two measure points: The minimum value is taken as a average of a column beside the Rexroth, and the maximum value is the average of the test column on the Rexroth. The height of the Rexroth is known to be 3 cm.

6.3 Host control system tests

The LabVIEW implementation is an experimental implementation, and is therefore subjected to fewer tests than the line scanner module. However, the bottlenecks of the implementation should still be found.

The bottlenecks of the implementation are found using LabVIEWs profiler. This profiler is able to compare the execution time and memory usage of each VI. It is also able to distinguish between application logic execution and application visualization execution. The test is to be executed with the configuration described in Table 6.4. The test results should not be used for other than indicating which parts of the application that consume the most time and memory.

The faster the LabVIEW host works, the faster TCP packets are read, and the faster the line scanner module is able to send them. Therefore, monitoring the TCP send time from the line scanner module gives an indication of the speed of the LabVIEW implementation. These results can be compared with the results obtained from the benchmarker.

6.4 System functionality tests

The system functionality tests should verify that 3D models are built as expected. For this purpose a salmon obtained from SINTEF Ocean is used. The salmon is scanned with all the implemented scan modes at a rate of 400 frames per second. The test parameters described in Table 6.4.

Parameter	**Value**
Frame rate	400 FPS
Height	192 pixels
Width	1280 pixels
Threads per block	256
Image buffer size	32
Number of CUDA streams	4
Pixel power value	2
Pixel threshold	2300
Scatter distance	10
Camera shutter time	0.5 ms
Camera gain	30

Table 6.4: Configuration parameters used for repeatability and system functionality tests.

Chapter 7

Results

This chapter presents the results of the tests described in the previous chapter. The results are given in the same order as the test are described. That is, first benchmarks of the line scanner, next repeatability test result, then benchmarks of the host control system and finally the test results illustrating the functionality of the system.

For clarity, this chapter only presents the most important benchmark results. These results reflect the overall performance and limitations of the developed system. The more comprehensive results are presented in Appendix F.

7.1 Line scanner module test results

For an image size of 1280x192 it was found that the frame rate was determined by image buffering framework and the camera only. The maximum frame rate for an image size 1280x192 given Equation 6.1 for both the Jetson TK1 and Jetson TX1 are presented in Table 7.1 and Table 7.2. The next subsections give a more detailed view on the test results and show the limitations of each component.

Scan mode	Camera	Framework	TCP send time	Image processing
Mode 0	620	**220**	5806	2020
Mode 1	620	**220**	11000	3860
Mode 2	620	**220**	26660	29240
Mode 3	620	**220**	11530	3200
Mode 4	620	**220**	16990	4660

Table 7.1: For the Jetson TK1 line scanner module, the image buffering framework limits the maximum frame rate. These test result correspond to an image size of 1280x192 and image buffer size of 128.

Scan mode	Camera	Framework	TCP send time	Image processing
Mode 0	**620**	1020	10740	3480
Mode 1	**620**	1020	15980	6470
Mode 2	**620**	1020	10930	29240
Mode 3	**620**	1020	16420	5160
Mode 4	**620**	1020	20190	7300

Table 7.2: For the Jetson TX1 line scanner module, the camera limits the maximum frame rate. These test result correspond to an image size of 1280x192 and image buffer size of 128.

7.1.1 Memory configuration

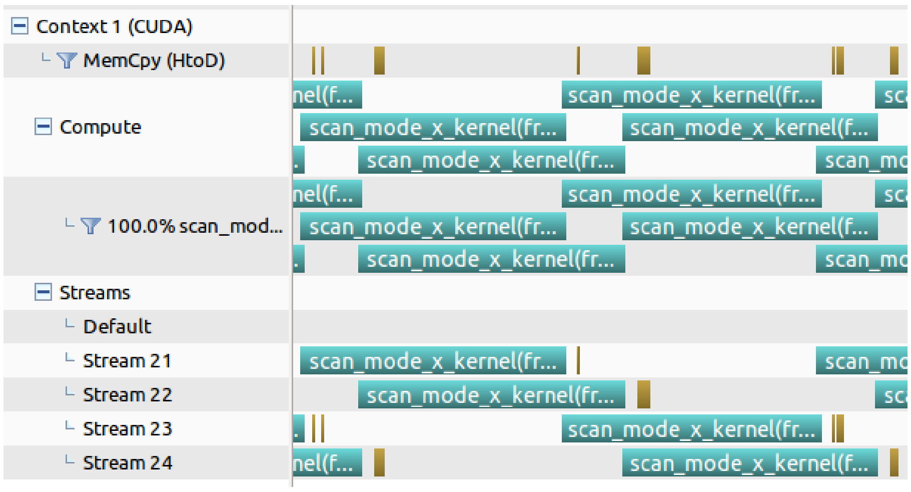
Table 7.3 summarizes the memory configuration tests. It turns out that the configuration <*Host, Host*> and <*Device, Host*> gives the desired behavior: Concurrent memory transfers and kernel execution both on the Jetson TK1 and Jetson TX1. Because <*Device, Host*> gives faster memory access during image processing (see section A.3), this is the preferred configuration. Some of the obtained execution profiles for the configuration <*Device, Host*> are shown in Figure 7.1.

Frame buffer	Output buffer	Concurrent kernels	Overlapping kernel and memory transfer
Host	Host	*Limited*	N/A
Host	Device	No	No
Device	Host	*Limited*	Yes
Device	Device	No	No

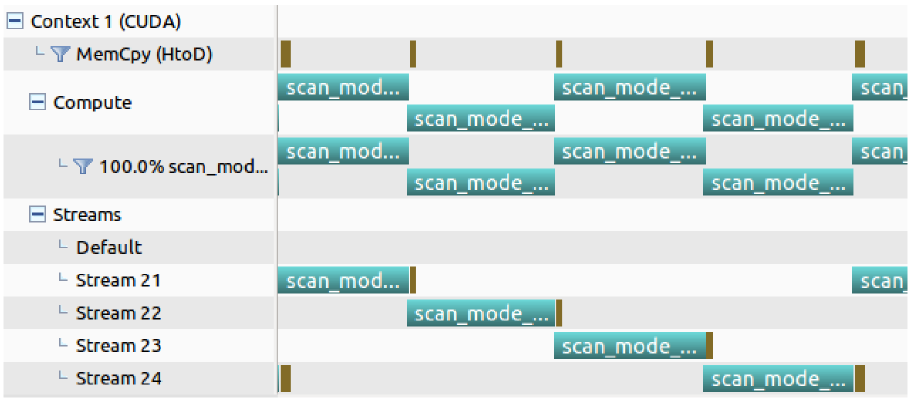
Table 7.3: Concurrency of the tested memory configurations. *Limited* in this context means that execution is concurrent when the image buffer size is less than or equal to 4 when executed on the Jetson TX1. The Jetson TK1 does not run image processing kernels concurrently.

7.1.2 Scanner framework benchmarks

The scanner framework was tested as described in subsection 6.1.2. Assuming that the framework was the limiting factor, the resulting frame rate for a variety of image sizes is shown in Figure 7.2. For all image sizes, the TX1 is multiple times faster than the Jetson TK1. For both the Jetson TK1 and Jetson TX1 the framework overhead increases as the image size increases. The detailed test results presented in subsection F.1.1 reveal that the synchronous execution time is the limiting factor for almost all image sizes.



(a) When the image buffer size is 2, kernel execution is concurrent.



(b) Image buffer size is 8 gives sequential kernel execution.

Figure 7.1: Execution profiles of the <*Device, Host*> configuration running on the Jetson TX1. Memory transfers are concurrent with kernel execution for both image buffer sizes.

7.1.3 Scan mode implementation benchmarks

The section describes the benchmark results of the implementation of the image processing kernels for scan mode 0 and scan mode 4 described in section 4.2. The performance of the additional implementations is presented in subsection F.1.2. Here it is assumed that the maximum frame rate is limited by the image processing time, and the maximum frame rate is therefore calculated using Equation 4.5 with the concurrency set to 1.

The TK1 image processing test results are not directly affected by the image buffering limitation of 220 frames per second. However, it was observed that frames were dropped, and therefore the suboptimal backup images were used. It was found that when the image buffer size and image sequence increased, more frames were dropped. For scan mode 4, an image buffer size of 128, and thread block size of 128 or larger, it was not possible to perform the benchmarks because too many frames were dropped.

Figure 7.3 illustrates that the Jetson TK1 has a faster image processing rate than

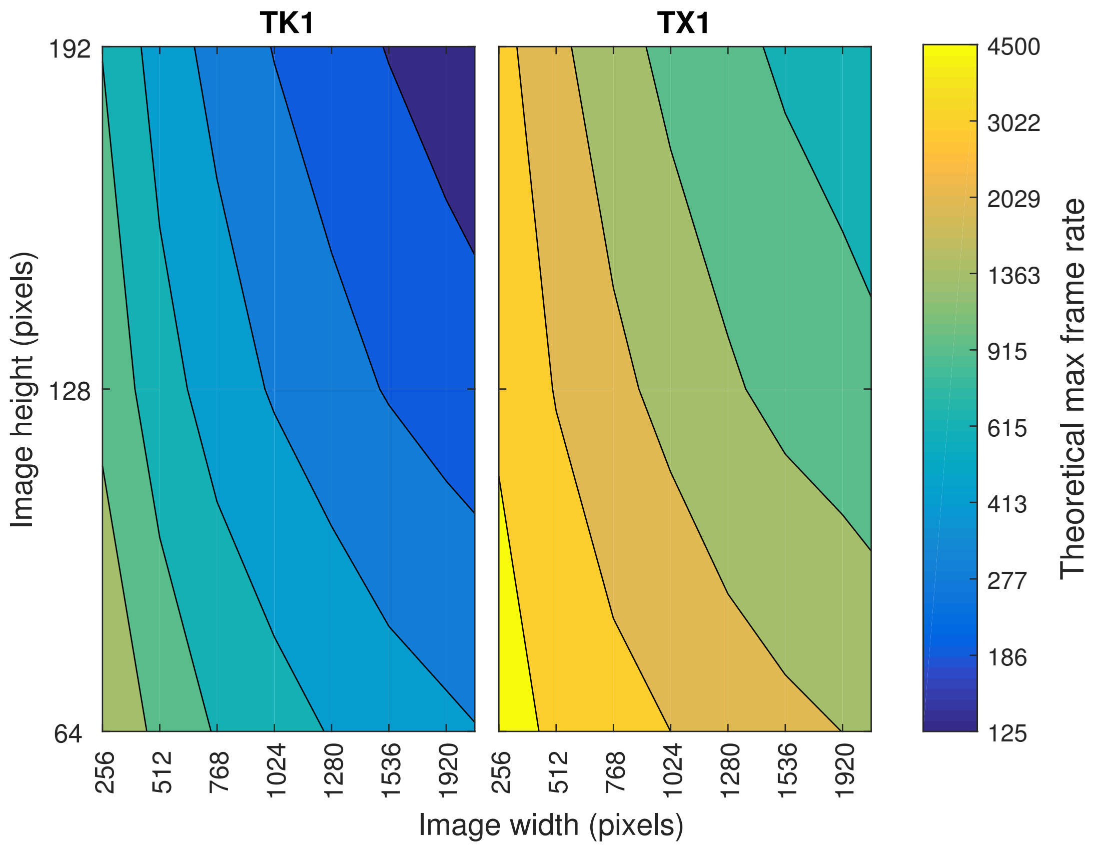


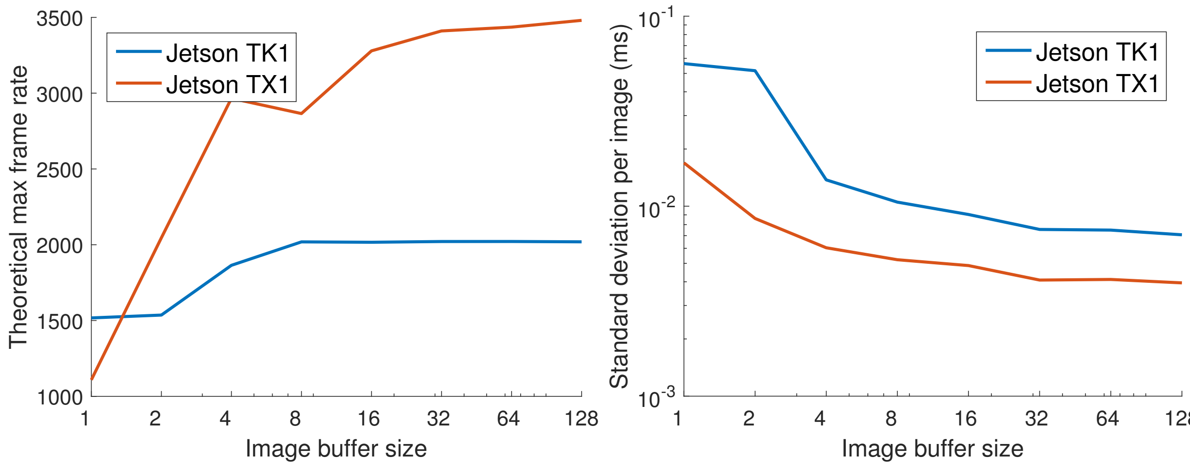
Figure 7.2: Image buffering framework time for the Jetson TK1 and Jetson TX1. The maximum framerate was calculated using Equation 4.3.

the TX1 when the image buffer size 1. When the image buffer size increases, the image processing rate increases. For a buffer size of 128, the TX1 performs almost twice as fast as the TK1. The TX1 has a smaller standard deviation in image processing time. As for the execution time, the standard deviation decreases as the image buffer size increases.

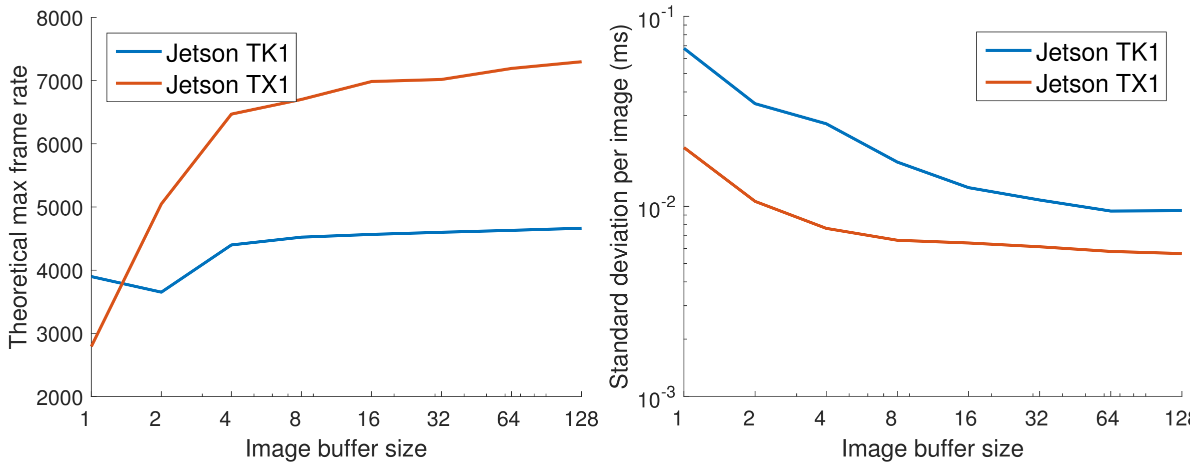
Figure 7.4 illustrates that the TX1 performs worse than the TK1 in terms of TCP send time when the image buffer size is small. When the image buffer size is 1, the average send time per image is as much as 2-3 time larger. However, when the image buffer size increases, the TX1 and TK1 obtain the same results.

The maximum image processing rate for scan mode 4 with varying block sizes and image buffer sizes for the TX1 and TK1 are shown in Figure 7.5. This figure shows that the Jetson TK1 and Jetson TX1 have different optimum thread block sizes. In general, the Jetson TK1 has the highest image processing rate when the block size is 64, while the results of the Jetson TX1 indicate that 128 is the optimal block size for the TX1. However, the detailed results presented subsection F.1.2 indicate that the optimimum block size may vary across the different scan mode implementations.

Figure F.13 plots the frame rate constraints for scan mode 4 for various widths and an image height of 192 pixels. Additional plots for image heights of 64 and 128 pixels are given in subsection F.1.2. It is clear that image processing is much faster than both the



(a) Theoretical maximum frame rate and standard deviation for scan mode 0.



(b) Theoretical maximum frame rate and standard deviation for scan mode 4.

Figure 7.3: Total image processing time benchmark results for scan mode 0 and 4. The test setup parameters were those given in Table 6.2. For each image buffer size, the thread block size with the maximum theoretical frame rate was chosen.

framework and camera. For the TK1, the image buffering framework is the bottleneck, while the camera is the bottleneck for the TX1. For the TX1 and large image widths, the framework overhead is almost the same as the camera limitation.

7.2 Repeatability test results

The test results presented in Table 7.4 show that scan mode 4 has a better repeatability than scan mode 0. The test was performed with an image height of 192, where the Rexroth utilized 1/6 of the image height. For each test 150 height samples were used. Figure 7.7 shows the captured height profile for scan mode 4.

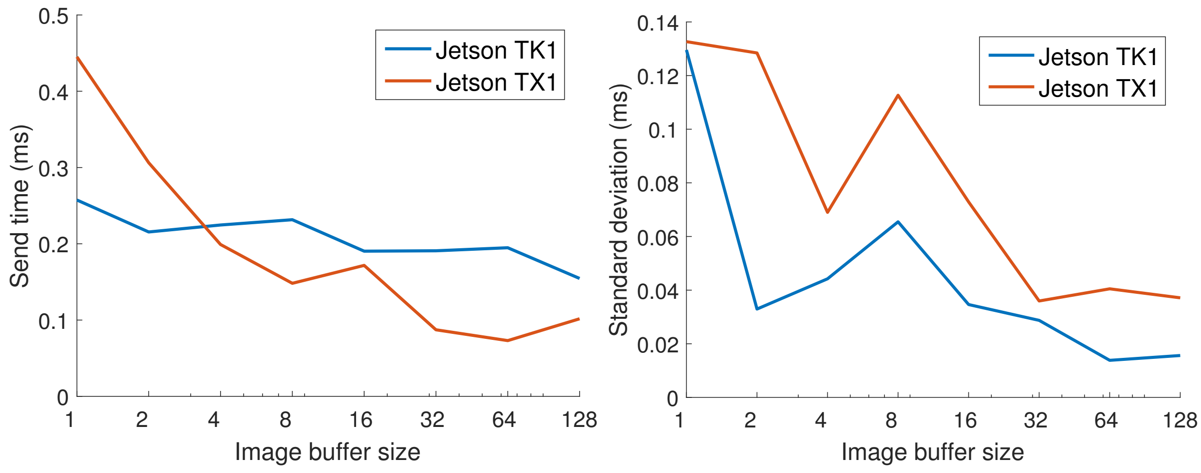


Figure 7.4: TCP send time of the output buffer divided by the number of frames this output buffer includes. The samples were obtained with scan mode 0.

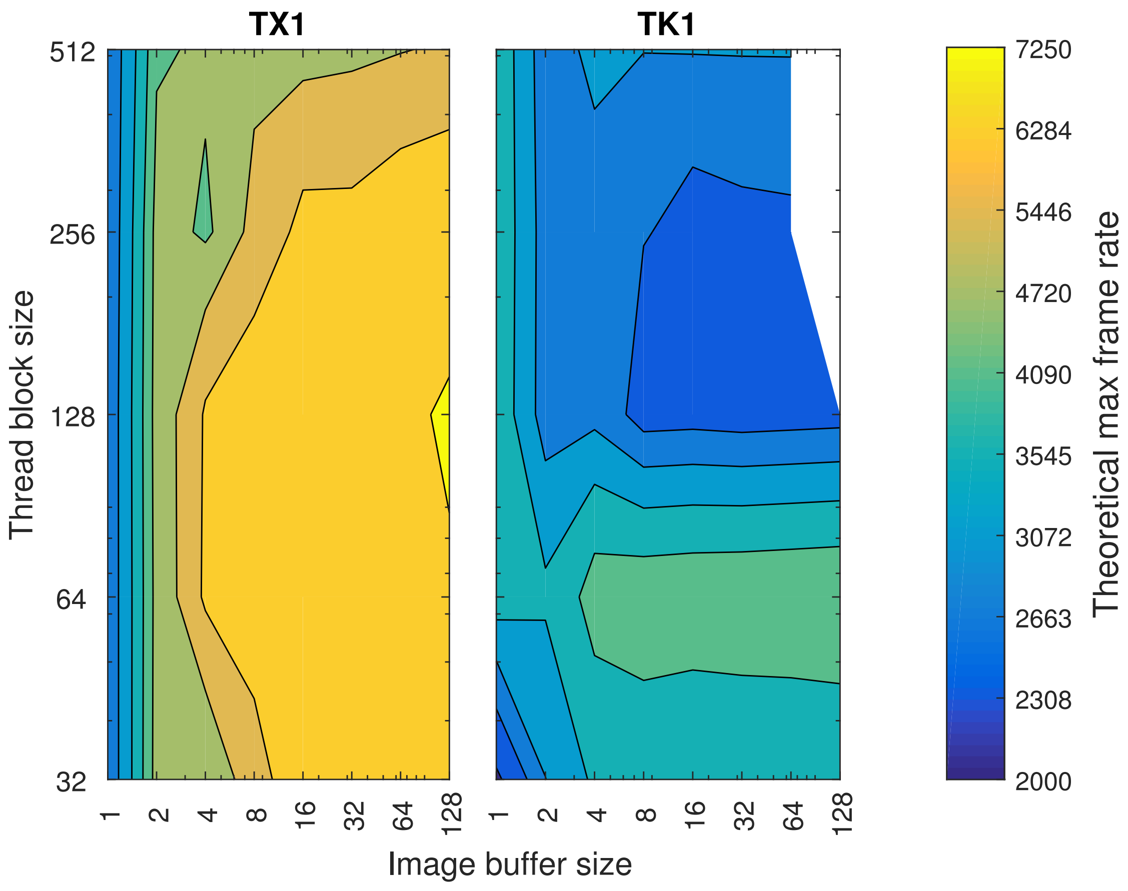


Figure 7.5: Maximum image processing rate for various buffer and thread block sizes for scan mode 4. The results are obtained using the test parameters given in Table 6.2.

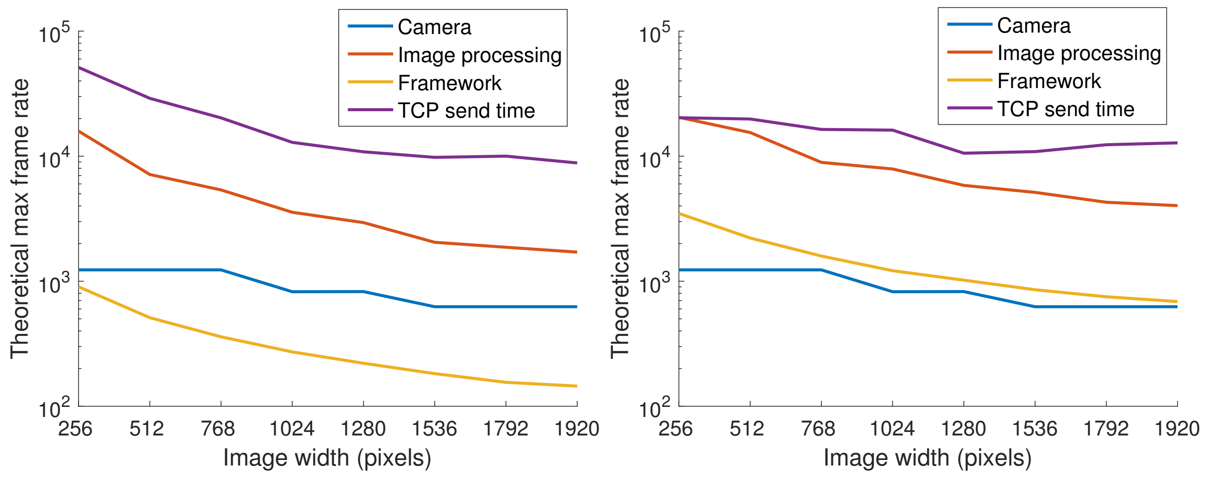


Figure 7.6: Plot of frame rate constraints for scan mode 4 on the Jetson TK1 (left) and TX1 (right) for an image height of 192 pixels. The results were obtained using the configuration parameters given in Table 6.3.

	Minimum (mm)	Maximum (mm)	Standard deviation (mm)
Scan mode 0	27.92	31.06	0.70
Scan mode 4	28.66	31.14	0.41

Table 7.4: The height measurements of a rexroth bar represent the repeatability of the scanner.

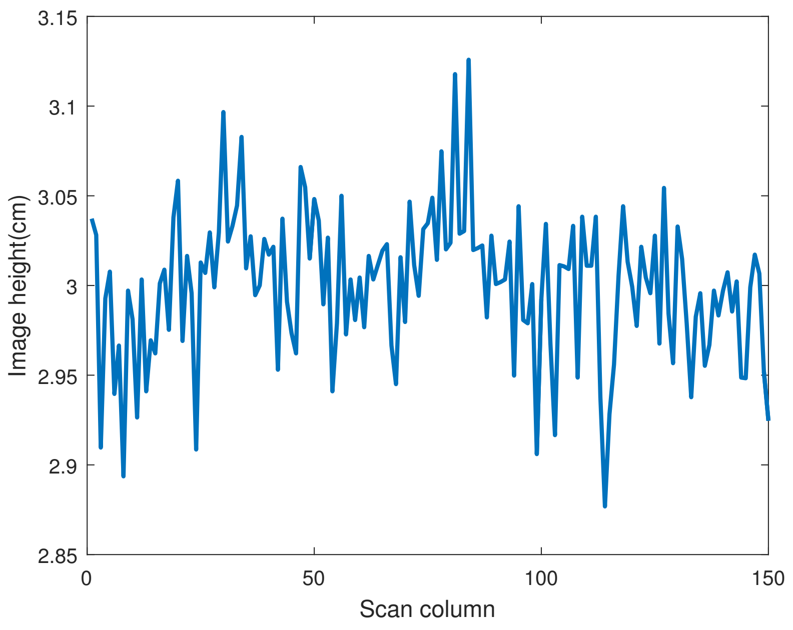
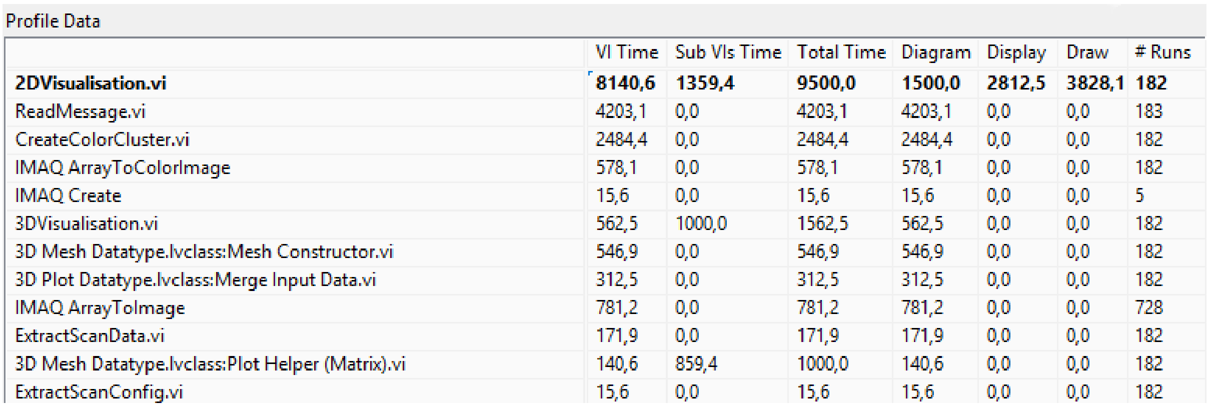


Figure 7.7: The captured height profile of the Rexroth bar used for repeatability testing. The height was obtained using scan mode 4.

7.3 Host control system test results

Figure 7.8 shows an extraction of the LabVIEW profiler results. It is clear that *2DVisualization.vi* is most time expensive VI. This VI uses *IMAQ ArrayToImage*, which is both time consuming and is executed more often. Reading TCP messages is time consuming, while decoding the scan data, done in *ExtractScanData.vi*, is not.



Profile Data

	VI Time	Sub VIs Time	Total Time	Diagram	Display	Draw	# Runs
2DVisualisation.vi	**8140,6**	**1359,4**	**9500,0**	**1500,0**	**2812,5**	**3828,1**	**182**
ReadMessage.vi	4203,1	0,0	4203,1	4203,1	0,0	0,0	183
CreateColorCluster.vi	2484,4	0,0	2484,4	2484,4	0,0	0,0	182
IMAQ ArrayToColorImage	578,1	0,0	578,1	578,1	0,0	0,0	182
IMAQ Create	15,6	0,0	15,6	15,6	0,0	0,0	5
3DVisualisation.vi	562,5	1000,0	1562,5	562,5	0,0	0,0	182
3D Mesh Datatype.lvclass:Mesh Constructor.vi	546,9	0,0	546,9	546,9	0,0	0,0	182
3D Plot Datatype.lvclass:Merge Input Data.vi	312,5	0,0	312,5	312,5	0,0	0,0	182
IMAQ ArrayToImage	781,2	0,0	781,2	781,2	0,0	0,0	728
ExtractScanData.vi	171,9	0,0	171,9	171,9	0,0	0,0	182
3D Mesh Datatype.lvclass:Plot Helper (Matrix).vi	140,6	859,4	1000,0	140,6	0,0	0,0	182
ExtractScanConfig.vi	15,6	0,0	15,6	15,6	0,0	0,0	182

Figure 7.8: Profiler results of the LabVIEW host implementation.

Figure 7.9 shows that the average send time per image is high when the TCP packet size is small. When the image buffer size was 4 and smaller, it was necessary to reduce the scan rate. This because the TCP read time occasionally returned very large values, resulting in too many dropped frames. For image buffer sizes 8 and larger, it was possible to run the camera at 400 frames per second.

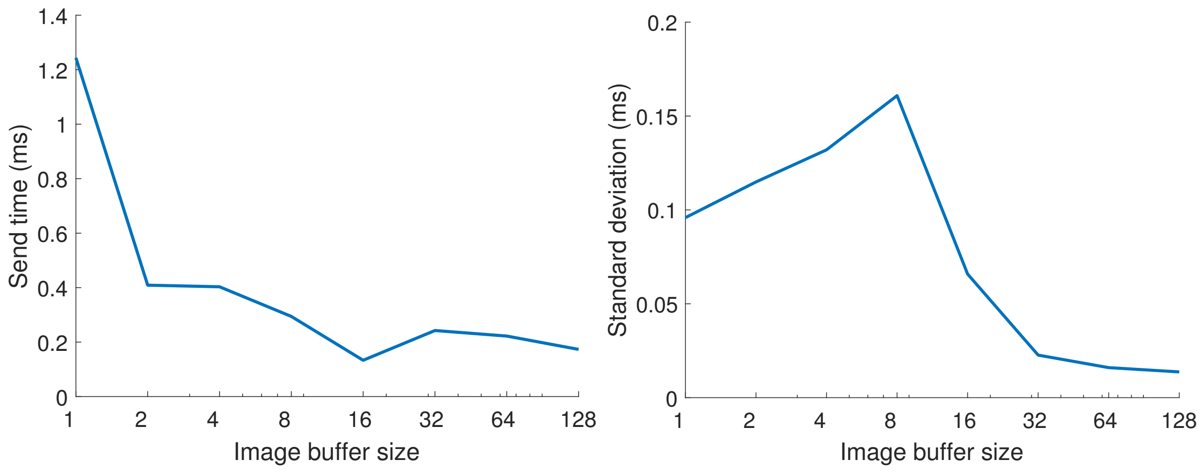
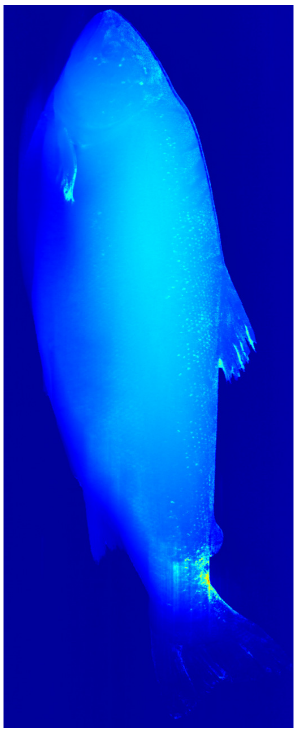
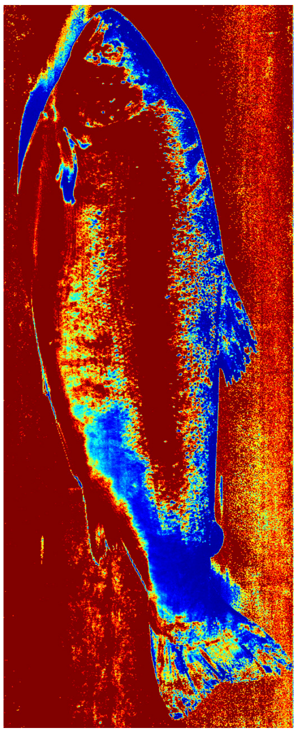


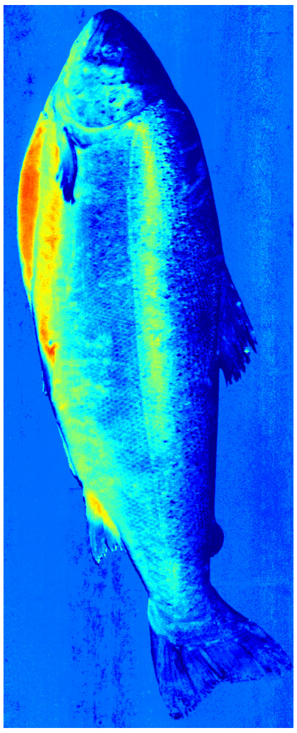
Figure 7.9: TCP send time for different image buffer sizes measured with the LabVIEW host control system.

7.4 System functionality test results

Figure 7.10 and Figure 7.11 present scan results of a salmon obtained with scan mode 0 and scan mode 4. The colors indicate the values: The value increases from blue to green to red. The height profile obtained using scan mode 4 is more even than the profile obtained using scan mode 0. This is visualized more clearly when the height profile is viewed from the side as shown in Figure 7.12. Combining height profile and color results in a colored 3D model is shown in Figure 7.13. Additional salmon scan results are presented in section F.2.


(a) Height profile.


(b) Intensity profile.


(c) Reflectance profile.

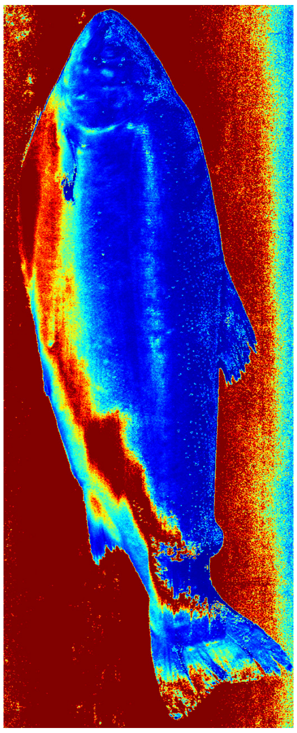
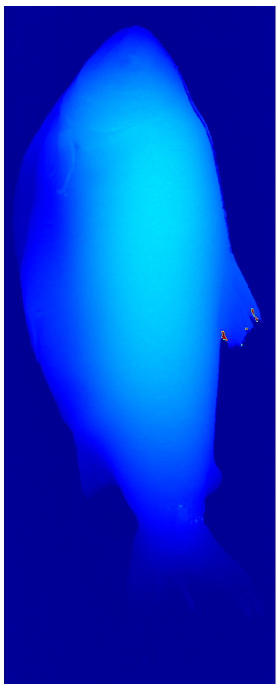
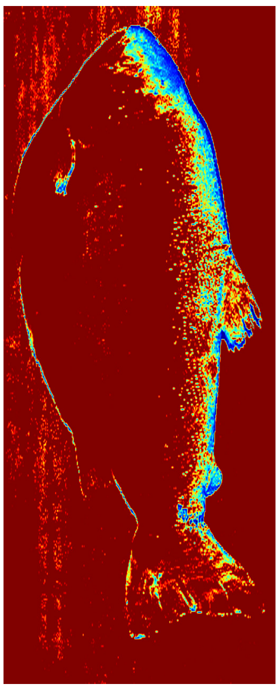
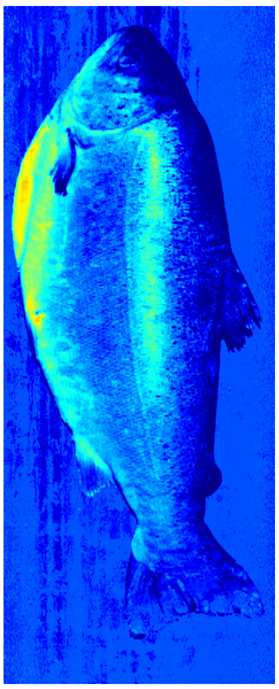
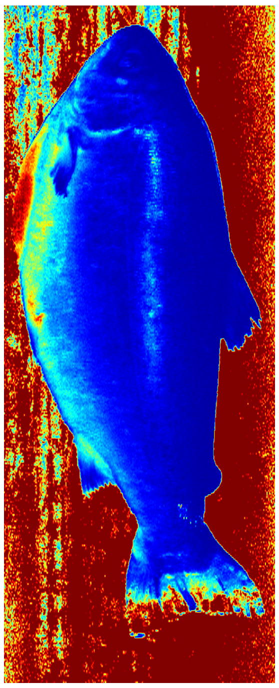

(d) Scatter profile.

Figure 7.10: Scan results of a salmon obtained with scan mode 0 and the test parameters given in Table 6.4.


(a) Height profile.


(b) Intensity profile.


(c) Reflectance profile.


(d) Scatter profile.

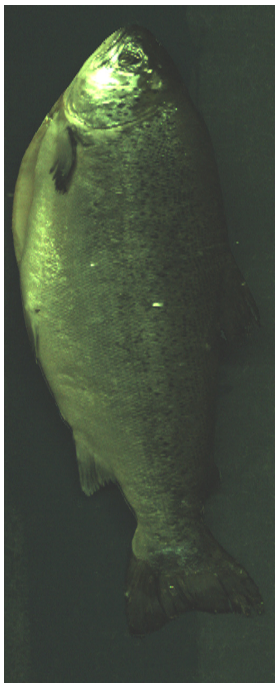
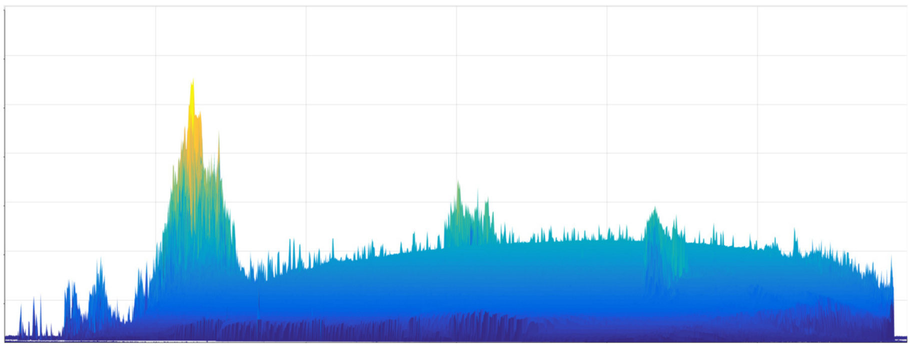

(e) Color image.

Figure 7.11: Scan results of a salmon obtained with scan mode 4 and the test parameters given in Table 6.4.


(a) Height profile obtained with scan mode 0.

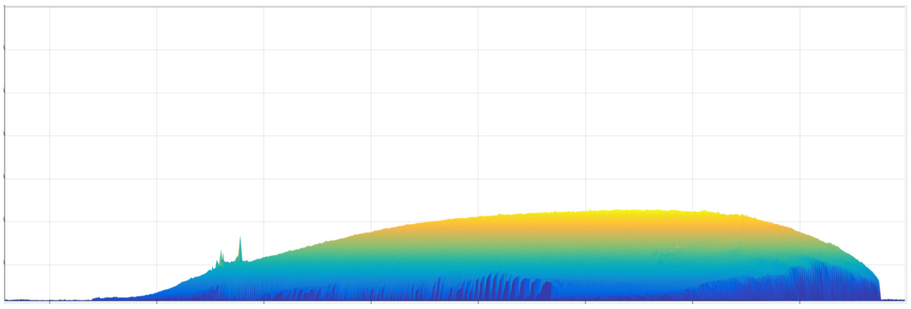

(b) Height profile obtained with scan mode 4.

Figure 7.12: Height profile of a salmon seen from the side. Scan mode 0 has more uneven results than scan mode 4. The results were obtained with the test parameters given in Table 6.4.

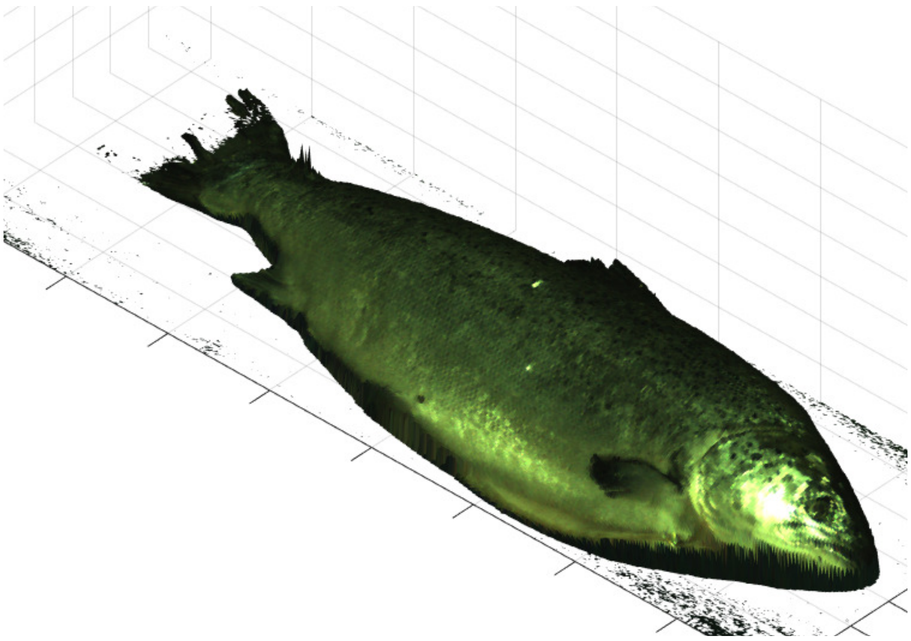


Figure 7.13: Combining the height and color image from Figure 7.11 results in a colored 3D model. The lowest values of the height profile are removed to accentuate the salmon.

Chapter 8

Discussion

This chapter reflects on the results presented in the previous chapter. First, the line scanner module test results are discussed. Next follows the discussions on the repeatability, host control system, and finally the system level test results. Missing features and future development are discussed in chapter 10.

8.1 Line scanner module result interpretation

The line scanner module benchmarks found that the implementation on the Jetson TK1 is limited by its slow memory transfer speeds, while the Jetson TX1 is limited by the maximum frame rate of the camera. For both line scanner modules, the maximum image processing rate and TCP send time is much faster than both the scanner framework and the camera.

8.1.1 Memory configuration results

The memory configuration tests found that both the <*Host, Host*> and <*Device, Host*> configurations allowed limited concurrent kernel execution, and stated that the <*Device, Host*> is the preferred configuration. This assumption is valid because the camera SDK requires the images to be copied out from its own buffer. If the camera library had provided direct access to the image buffers, this extra copy operation would have been unecessary for the <*Host, Host*> configuration, and therefore increased the scanner framework performance. However, this configuration requires all memory to be accessed as pinned memory, which is slower than direct access to GPU memory, which reduces the image processing rate. Therefore, choosing between these two configurations is a trade-off between image processing speed and scanner framework performance.

The validity of the observation that image processing kernels execute concurrently, is discussable. First, a fake image processing kernel was used instead of the actual scan mode implementations. This was necessary because the maximum camera frame rate was much lower than the maximum image processing rate. Even though the kernel launch parameters were correct, the fake kernel resource usage might differ from the image processing kernel resource usage. Therefore, this test is only valid to show that the GPU is able to run multiple kernels concurrently, and should not be used to find the concurrency factor. The property of concurrent kernel execution and memory transfers is not affected by changing the image processing kernel to a fake kernel, and these results are therefore valid.

The maximum number of concurrent kernel executions was not obtained with the memory configuration tests. To find the number of maximum concurrent kernels, a faster camera must be used in combination with the scan mode implementations. Instead of using a faster camera, an alternative approach is to use a camera mock. A camera mock can simulate the arrival of images at any rate. Such an approach would therefore push the image processing to its limits. However, the image buffering framework is still slower than the image processing implementations. Therefore, this approach is only useful if the image buffering framework is optimized further.

All memory configuration test results were obtained using NVIDIAs Visual Profiler. Because the profiler was unable to launch the line scanner module application with superuser rights, which is necessary to change the execution priority of the application (see section 4.1), this feature was turned off. This might have changed CPU execution, however, not the blocking and concurrency properties of the GPU. Therefore, the profiler has not affected the validity of the memory configuration test results.

8.1.2 Scanner framework benchmark results

The scanner framework benchmark results were obtained by measuring both the synchronous and asynchronous framework execution speed. Instead of determining the actual image buffering time, the synchronous execution speed was used as a conservative estimate. Because the synchronous execution speed was the limiting factor for most of the image sizes, it is expected that the framework is able to operate at a somewhat higher frame rate than stated here. The actual image buffering time could have been determined by using CUDA events. However, this would have created extra overhead.

It is assumed that only one memory transfer may be active at the time. For the Jetson TK1 and TX1, which both have only one copy engine, this is most certain the case. However, high end GPUs might be able to run multiple concurrent memory transfers. If a GPU allows high memory copy concurrency, this might result in the asynchronous memory framework being the bottleneck of the image buffering framework. Figure F.1 shows that the asynchronous framework is very fast, and therefore it is unlikely that the asynchronous image buffering framework will be the bottleneck of the line scanner module.

The framework benchmark results indicate that the Jetson TK1 performs worse than the TX1 in terms of memory transfer speed. An entry in NVIDIAs forums suggests that the Jetson TK1 is not able to copy pinned memory fast[1]. The observation of that the TK1 drops more frames at large image buffer sizes and image sequence lengths, which requires more pinned memory, strengthens this theory.

[1] `https://devtalk.nvidia.com/default/topic/948258/performance-of-v4l2_memory_mmap-buffer-memcpy/`

The framework overhead was measured with an image buffer size of 1, which does only require a litte amount of pinned memory. Because it seems like more pinned memory reduces the memory transfer rate, the image buffering framework overhead might be larger when large image buffer sizes are used. On the other hand, when the image buffer size were small, the TK1 did not drop frames as expected from the framework benchmarkmark results. It is therefore likely that this estimate is conservative for small buffer sizes. In order to find the actual image buffering overhead, this can be verified by trial and error, observing if frames are dropped or not.

The problem statement required the scanner to use 12-bit images. This increases the accuracy of scan results. However, if 8-bit images are found to give high enough accuracy, this will increase the performance of the scanner framework because 8-bit images are only half the size of 16-bit images. Therefore, the memory transfer would probably only consume half of the time, and thereby doubling the performance of the image buffering framework.

It is also possible to increase scanner framework performance by removing failsafe functionality. Figure 4.6 describes how each image first is stored as a backup, before it is copied to the image processing buffers. This failsafe feature can be made optional, thereby removing one memory transfer for each image, increasing the image buffering speed. Therefore, the performance of the image buffering framework is a trade-off between maximum image buffering rate rate, and failsafe functionality.

As mentioned earlier, the scanner framework processing time can be increased drastically by using another camera SDK that does not require images to be copied out. For this case, it is assumed that the framework will perform similar to the asynchronous framework, and therefore allow a frame rate of thousands of images per second as shown in Figure F.1.

8.1.3 Scan mode implementation benchmarks

Previously it was shown that the TX1 executed image processing kernels concurrently when the image buffer size was 4 or less. However, as stated, this is not included in the image processing benchmark results. Therefore, if the concurrency results were included, then the TX1 would have performed better for small image buffer sizes. Then the performance gap between the TX1 and TK1 would be even larger.

For all scan mode implementations, the Jetson TK1 performs better at image buffer size 1. This is true even though the Jetson TX1 has a three times higher available computational power than the TK1. However, GPU performance is measured in throughput, and not latency. These result are therefore explained by the TX1 not utilizing its full power for this buffer size. When the buffer size increases, more of its power can be utilized, and thereby the image processing speed increases. On the other hand, the TK1 reaches its maximum performance at even the smallest buffer sizes, and does therefore not benefit from larger image buffer sizes.

Not only the image processing speed benefits from increased image buffer sizes. Both benchmarks of the TCP send time and results from the host control system results indicate that the average send time per scan result decreases as the image buffer size increases.

The trade-off for a large image buffer size is increased latency. However, the latency is only equal to sum of the capture time of the image buffer and image processing time, which at high frames is still only a fraction of a second. Therefore, if the scan results are needed a given time or location on the production line, it might be easier to move the scanner further up.

For all scan modes on both the Jetson TK1 and TX1, the standard deviation in image processing time is small. This is indicates that memory is fetched with few cache misses, or that cache is missed in a regular manner. A small standard deviation reduces the requirement of safety margins in calculating the maximum scan rate, and is therefore desirable.

From Table 7.1 and Table 7.2 it is clear that scan mode 2 processes images much faster than all other scan modes. However, this scan mode does only obtain color information, which is a fast operation. Instead of using this scanner for obtaining a stream of color data, it is probably cheaper and less complex to use and an ordinary color camera.

The bottleneck plots of scan mode 4 are retrieved with an image buffer size of 8. Benchmarks done with larger image buffer sizes indicate an approximate 20% performance gain when the image buffer size increases from 8 to 128. Therefore, the actual image processing speed is even faster than shown in these plots. However, because of the limitations of the camera and image buffering framework, this does not affect the maximum scan rate.

8.2 Repeatability results interpretation

The repeatability test results is a measure on the quality of the scan results. A large number of factors affect the repeatability. In order to compare repeatability, all these factor must be kept constant. Some of these are given below:

- Light conditions. The darker the scan area, the less likely it is for light disturbance to occur. Therefore, a shielded scan area will increase the repeatability. Refer to section 3.2 for a discussion on how external disturbances can be minimized.
- The object itself. For this test, a Rexroth bar was used. Because this object has a high reflectance, the expected measured height deviation is large. A non-reflecting surface would probabily yield a higher repeatability.
- Physical setup. The repeatability results are presented in millimeters. The maximum accuracy, and therefore the repeatability, is determined by the angle between the laser and camera, and the distance between the object and camera.
- Camera parameters. The camera has a large number of parameters that can be altered to increase the image quality. Such parameters are for example white balance, contrast, shutter time, gain, etc.
- Image processing parameters. For this test only one set of image processing parameters was used. Different parameters will probably yield different results.

Table 7.4 presents results for only one given configuration, and therefore this result does only present a small aspect of the scan quality. Because the same test parameters were used for both testing scan mode 0 and scan mode 4, it is valid to compare those results. As expected, scan mode 4 utilizing black frame subtraction, yields better repeatability. However, using black frame subtraction reduces the scan rate because the corresponding image sequence length is longer. Therefore, there is a trade-off between increased repeatability and increased scan rate. In addition it might be possible to achieve the same repeatability by modifying camera and image processing parameters.

Finding an optimum configuration for maximizing the repeatability is a research project itself. The most obvious parameter changes are the image processing parameters, and especially the pixel power parameter (see section 3.2). A large value might give more accurate results, but might be more susceptible to noise. In addition, it is possible to use another set of values in the pixel power lookup table (see section 4.2.3).

8.3 Host control system result interpretation

The host control system benchmark results indicated that visualization and TCP message reception are the two most computational expensive tasks. These are therefore the two first components to optimize when the performance of the host application must be increased. The current implementation is not able to handle small image buffer sizes well, but has sufficient performance for larger sizes. Therefore there is no need in optimizing the application for the current usage. However, when more features are added, and multiple line scanner modules are used, this might not be the case.

Because real-time visualization is not a strictly necessary task to perform, this can be removed in order to increase performance. It will still be required to build the 3D models, but because this can occur in background, it is assumed to take less processing power. Including less computational expensive visualization is possible by reducing the update rate. It is for example possible to visualize a complete 3D model only every time an object has passed.

When Figure 7.4 and Figure 7.9 it is clear that LabVIEW uses more time to read TCP messages than the benchmarker when the image buffer size is small. When the buffer size increases, the average send time per image is close to the results obtained with the benchmarker. Therefore, the LabVIEW host should not be used when the image buffer size is small.

Reading TCP messages is done through stardardized LabVIEW blocks for reading a TCP port. Therefore, it is not possible to speed up this process. However, performance may be increased by increasing the scheduling priority of the application. This might also affect the hosts ability to receive smaller image buffer size scan data.

8.4 System functionality result interpretation

The system functionality test results verified that the scanner is able to scan salmon and reproduce a relative accurate 3D model of it. It is clear that scan mode 4 reproduces the salmon more accurate, with a slower scan rate as its extra cost.

Comparing all the individual salmon scan results indicates that scan mode 4 in overall gives gives better scan results than scan mode 0. The height profile is much more even for scan mode 4 than scan mode 0. This is emphasized by the intensity profile, which has larger red areas, which indicate a higher hit rate of the laser line. The uneven scatter profile for scan mode 0 indicates the same result. However, there is no big difference in reflectance between scan mode 0 and scan mode 4. This is probably because the reflectance definition is robust against noise.

It is unexpected that the scatter value is higher on the conveyor than the salmon. This might be caused by the the definition of scatter and its test configuration value. For the case when the height is slightly miscalculated, the laser line will be some distance away from the calculated height. Because the definition of the scatter is the intensity some distance from the height, the scatter will in this case report the intensity of the laser line. Therefore, it is suggested to improve this definition, to include an intensity on both sides of the calculated height.

The color image retrieved from scan mode 4 is quite dark and green tinted. This is not caused by the line scanner module, but is a result of the camera parameters. The camera is set to a low shutter period, and high gain to be able run at a high frame rate. This effect is easily removed by post processing the color image.

Chapter 9

Conclusion

In this thesis a distributed color 3D scanner was developed and integrated into an experimental setup at SINTEF Sealab. The scanner is able to operate in multiple modes in order to extract different object parameters such as height, intensity, reflectance, scatter and color. Each of the modes have corresponding image sequences defined by the state of a laser line and LED. Black frame subtraction subtracts the background noise and is used to increase the quality of the measurements. The background noise is estimated by using frames where both the laser and LED is turned off.

Image parameter extraction is implemented on both the NVIDIA Jetson TK1 and the NVIDIA Jetson TX1. The implementation allows both asynchronous and concurrent execution on their GPUs. Several techniques are used to increase the image parameter extraction speed: first, images are buffered, which decreases the average image processing time per image. Next, pre-calculated lookup tables replace repeated calculations needed for pixel intensity transformations. Finally, images are traversed in a way where image pixels are visited as few times and with as few time-expensive operations as possible.

The maximum scan rate is determined by the maximum performance of the camera, the image buffering framework, and image processing implementation. For all scan modes, the maximum image processing rate is large. For the scan mode retrieving color and applying black frame subtraction, the TX1 is able to process images at a rate of 7300 frames per second while the TK1 is able to process 4600 frames per second. However, these frame rates are not achievable for full system due to limitations in system components. It was found that respectively the memory transfer speed and camera frame rate were the limiting factors for the Jetson TK1 and the Jetson TX1 line scanner modules. Therefore, for an image size of 1280x192, the maximum camera frame rate is 220 for the Jetson TK1 and 620 for the Jetson TX1.

The scan quality and repeatability of the scanner were tested on a Rexroth bar and on a salmon. For a given test setup, it was found that a height measurement of the Rexroth had a standard deviation of respectively 0.41 mm and 0.70 mm when black frame subtraction was enabled and disabled. A qualitative comparison of the scans performed on salmon confirmed that black frame subtraction increases the overall scan quality.

The camera is the major bottleneck of the developed scanner, both in terms of maximum available frame rate and in terms of SDK limitations. Therefore, if a higher scan rate is required, it is suggested to port the implementation to use another camera type. However, it should also be noticed that the scanner accuracy can be changed to obtain a higher scan rate. This can be done by either improving feature extraction, image quality, or by changing the physical arrangement of the components of the scanner.

The distributed color 3D scanner developed in this thesis is able to reach the performance of the scanner developed by SINTEF, given that the TX1 is used for image processing. The image processing implementations handles 12-bit images, is portable to more advanced GPUs, and is able to extract all image parameters at a high rate. The bottlenecks and limitations of the system are found, and error detection and correction is implemented. Therefore, the goals of thesis this are achieved and makes it a valuable contribution to SINTEF's further development of high-speed color 3D scanners.

Chapter 10

Future work

This chapter presents ideas for further development of the scanner presented in this thesis. The chapter is divided into three parts. The first section provides ideas on how the scan quality can be increased. Next, there is a discussion on suggestions to new functionality. The last section of this chapter investigates architecture alternatives that can reduce cost, increase performance, and add functionality to the scanner.

10.1 Optimize scan quality

In section 8.2 it is described that the repeatability of the scanner is determined by a large number of factors. By changing these parameters, it is might be possible to achieve a sufficient scan quality without black frame subtraction. By disabling black frame subtraction, the image sequence length decreases, which increases the maximum scan rate. Finding an optimum parameter set may be time consuming, and does also dependend on the type of object scanned. However, an increased scan rate increases the accuracy, and is therefore valuable.

Redefining the image parameters is another way of increasing scan quality. Especially the lookup table used for height extraction has many configurations. Changing the value range to be piecewise linear, instead of exponential might give other repeatability results. Section 3.2 discusses how some of the current line parameters can be made more noise robust.

It is discussed that the current definition of scatter is not accurate. This definition should therefore be updated. The definition might include an average pixel intensity of an area, or some fraction between the intensity and surrounding reflectance.

The retrieved color images of the salmon revealed that it might be necessary to post-process the color images. This may also be applied to the other scan result parameters. For example, the height profile can be processed by simple interpolation, removing outliers, or by using more advanced techniques combining all parameters. However, it should be kept in mind that post-processing might be time consuming. Care should therefore be taken to prevent post-processing becoming the bottleneck of the system.

10.2 Implement new features

There are several other features that will increase the number of use cases, and improve the overall quality and performance of the 3D scanner. The features list is ordered with the taks assumed to have the highest priority listed first. Each of the tasks are presented with their problem, and with some ideas on how they can be implemented:

- Calibration. The current scanner implementation lacks calibration, and therefore all scan data is returned without unit. Because most application areas require some sort of reference frame, the scanner is not of value before calibration is implemented. Because there exist a large variety of calibration methods, this task can be both small or large. Due to the spare computational power on the Jetsons, coordinate transformations may be performed directly in the line scanner modules.

- Allow the system to use multiple line scanner modules. There are only two minor modifications required to achieve this functionality. First, the LabVIEW host implementation must be modified to retrieve data from multiple sources. In section 5.3 there are already some suggestions on how data from multiple scanners can be combined. Next, it is required to synchronize the Arduino and line scanner modules to provide synchronized scan data. A suggested approach is described in section 2.3.

- Object detection. The current scanner outputs a continuous stream of data, which the host control system visualizes. In order to analyze objects, the input data stream must be split into objects. This is a task easily done by the host control system. The boundaries can be found by monitoring the scan height.

- Allow to use 8-bit images. The current line scanner module is only able to use 16-bit images. When the pixel depth is halved, the image size is halved Therefore it is likely that the performance of the image buffering framework will double. Because the image buffering framework is the bottleneck for Jetson TK1, this improvement will increase the performance of this line scanner module. Reducing the pixel depth reduces scan accuracy, and it should be investigated how this affects the scan quality.

- Allow a user configurable lookup table. The current lookup table used for height extraction has a fixed exponential shape. Other transformations might give better scan quality results. If this table is modifiable over TCP, it will be easier to test different configuration types. In addition, the number of image processing parameters is reduced.

- More advanced and configurable error handling. By making error handling and failsafe features optional, the framework overhead can be decreased. It should be user configurable to set a maximum number of dropped frames, if a backup frame should be used, etc. Removing the extra memory transfer needed for the backup image is especially valuable for the TK1 where memory transfer speeds is the bottleneck. In addition the error responses should be more intuitive. The current error handling returns an error message when the scanner fails. This should be expanded to include a message informing what went wrong.

- Multiple scan data message definitions. The current line scanner module defines only one scan data message format. Because some of the fields are unused for some of the scan mode definitions, new scan data message definitions can reduce the network overhead. In addition, this will require smaller memory buffers, possibly increasing both image processing speed and framework overhead.

10.3 Consider system architecture alternatives

Some of the limitations of the developed scanner are related to its architecture and used components. It is therefore natural to investigate other solutions. This section first considers other alternative synchronization mechanisms. Next comes a discussion on other camera alternatives.

10.3.1 System synchronization

In section 2.3 it is described how the components of the system are synchronized using an Arduino Mega. Using the Arduino makes synchronization simple, however, the extra component increases both the equipment and installation cost. It is described how usage of an external synchronization component makes it more difficult to synchronize multiple line scanner modules. In addition, it is more difficult to reconfigure the scan mode and frame rate: Either the Arduino must be reprogrammed every time, or through a message interface. Using a message interface requires the Arduino to be connected to either one of the line scanner modules, or to the host computer. It is clear that this will increase the complexity.

Instead of using the Arduino for synchronization, one of the PointGrey cameras can be used. The cameras have GPIO connectors that can be used for output signals. Therefore one camera can be configured to output a pulse signal with a given frequency on one or multiple of its output pins to synchronize the other cameras. Such a solution simplifies the synchronization between multiple line scanner modules and it simplifies scanner configuration. With this system architecture, it is possible to configure the synchronization through the camera SDK.

10.3.2 Camera alternatives

The major scan rate limitation of this scanner is the camera. For an image size of 1280x192 using 16-bit quality, the camera is only able to reach a rate of 620 frames per second. This corresponds to a bandwidth usage of approximately 290 MB/s. The theoretical maximum USB3.0 bandwidth is 625 MB/s. Therefore, it highly suggested to investigate if other camera providers are able to reach a higher frame rate. This will increase the maximum scan rate drastically.

The camera is interfaced through its SDK. Using an SDK is convinient, however it does also come with performance trade-offs. As described in subsection 2.1.2, the SDK requires images to be copied in order to store them in a buffer. In addition, the SDK does not provide the functionality of retrieving multiple images at a time. A more flexible SDK would provide direct access to its buffers, which would eliminate the need for this extra copy operation, and therefore increase performance. In order to obtain such a feature, it is necessary to switch camera provider.

Using USB cameras adds an extra limitation on the system architecture: Each Jetson has only one USB3 port, and it is therefore not possible to let each Jetson handle multiple USB cameras. Requiring one Jetson per camera increases the total hardware and installation cost.

Even though the Jetson TK1 and TX1 only have one USB3 interface, there are some possibilities to connect more cameras to the Jetsons. The Jetson TX1 has a four lane PCI Express 2.0 interface, and the Jetson TK1 is assumed to have one lane PCI Express 2.0 interface[1]. These connectors can be used for USB expansion cards. Because each PCI 2.0 express lane allows transfer speeds up to 500 MB/s, it is possible to add one extra USB port to the TK1 and four to the TX1.

Another approach is to use a different type of camera. Both the Jetson TK1 and Jetson TX1 are equipped with Camera Serial Interface (CSI) connectors. These are low level serial interfaces, where each lane is able to provide a bandwidth of 312.5 MB/s. Using two- or four-lane CSI cameras, a bandwidth of 625 MB/s or 1.25 GB/s is achieved. The Jetson TK1 has four CSI lanes, while the Jetson TX1 has twelve. Therefore it is possible to connect multiple CSI cameras to each Jetson. In addition, the maximum camera bandwidth is increased.

CSI provides high throughput, however this interface has some major trade-offs. First, developing low-level solutions add more complexity which increases the development cost. Next, CSI connectors only allow very short cables, typically an absolute maximum of 50 cm. Therefore, it might not be practically possible to connect more than one camera per Jetson. Finally, there are only a few configurable high-speed cameras available, which in addition only seem to support large image sizes. The *e-CAM40_CUTK1* is such a camera providing only 330 fps at an image size of 672x380 (e-con Systems 2015).

[1]https://devtalk.nvidia.com/default/topic/763037/-jetson-tk1-mini-pci-express-version-/

Bibliography

Amdahl, Dr. Gene M. (1967). “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities”. In: *AFIPS Conference Proceedings* 30, 483–48.

Arduino (2017). *Arduino MEGA 2560 & Genuino MEGA 2560*. URL: https://www.arduino.cc/en/Main/arduinoBoardMega2560 (visited on 04/18/2017).

Brosnan, Tadhg and Da-Wen Sun (2004). “Improving quality inspection of food products by computer vision—a review”. In: *Journal of Food Engineering* 61.1. Applications of computer vision in the food industry, pp. 3 –16. ISSN: 0260-8774.

e-con Systems (2015). *e-CAM40_CUTK1 Datasheet*. URL: https://www.e-consystems.com/downloadecamdoc.asp?file=mb161013jEcAm40CuTk1DaShmIpi01 (visited on 05/28/2017).

Focus, Photon (2016). *MV1-D2048-3D03/3D04 Camera Series user manual*. URL: http://www.photonfocus.com/fileadmin/web/manuals/MAN052_e_V3_1_MV1_D2048_3D03_3D04.pdf (visited on 05/18/2017).

Gerritsen, Rubin Ingwer (2016). *Embedded 3D vision for industrial applications using the NVIDIA Jetson TK1*.

Gonzalez, Rafael C. and Richard E. Woods (2007). *Digital Image Processing*. 3rd ed. Pearson. ISBN: 978-0131687288.

Gustafson, John L. (1988). “Reevaluating Amdahl’s law”. In: *Communications of the ACM* 31, pp. 532–533.

Hennessy, John L. and David A. Patterson (2006). *Computer Architecture: A Quantitative Approach*. 4th ed. Morgan Kaufmann. ISBN: 0123704901.

Kurose, James F. and Keith W. Ross (2012). *Computer Networking: A Top-Down Approach*. 6th ed. Pearson. ISBN: 978-0132856201.

National Instruments (2015). *Tutorial: Timing, Shift Registers, and Case Structures*. URL: http://www.ni.com/white-paper/7592/en/ (visited on 05/19/2017).

— (2016). *Application Design Patterns: Producer/Consumer*. URL: http://www.ni.com/white-paper/3023/en/ (visited on 04/03/2017).

NVIDIA (2012). *How to Optimize Data Transfers in CUDA C/C++*. URL: https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/ (visited on 12/11/2016).

NVIDIA (2014). *Remote application development using NVIDIA® Nsight^{TM} Eclipse Edition*. URL: https://devblogs.nvidia.com/parallelforall/remote-application-development-nvidia-nsight-eclipse-edition/ (visited on 12/11/2016).

— (2016a). *CUDA C Best Practices Guide*. DG-05603-001_v8.0. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (visited on 12/04/2016).

— (2016b). *CUDA C Programming Guide*. PG-02829-001_v8.0. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 12/04/2016).

— (2016c). *Profiler User's Guide*. DU-05982-001_v8.0. URL: http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf (visited on 12/04/2016).

Point Grey (2015a). *Register Reference for Point Grey Digital Cameras*. Version 3.2. URL: http://www.ptgrey.com/support/downloads/10130/ (visited on 12/04/2016).

— (2015b). *What external IIDC trigger modes are supported by my camera*. URL: https://www.ptgrey.com/KB/10250 (visited on 03/15/2017).

— (2016a). *Grasshopper3 U3 USB 3.0 Camera Technical Reference*. Revision 17. URL: http://www.ptgrey.com/support/downloads/10125/ (visited on 12/04/2016).

— (2016b). *Technical Application Note TAN10563, Working with Embedded Image Information*. URL: https://www.ptgrey.com/support/downloads/10563 (visited on 04/09/2017).

— (2016c). *Technical Application Note TAN2009003, Getting Started with FlyCapture 2.x and Linux*. URL: https://www.ptgrey.com/support/downloads/10385 (visited on 12/04/2016).

Rausand, Marvin and Arnljot Høyland (2004). *System Reliability Theory. Models, Statistical Methods, and Application*. 2nd ed. John Wiley & Sons, Inc. ISBN: 978-0-471-47133-2.

SICK (2011). *ColorRanger E 3D Cameras Product Information*. URL: https://www.sick.com/media/docs/8/08/408/Product_information_ColorRanger_E_3D_Cameras_en_IM0035408.PDF (visited on 05/12/2017).

Sture, Øystein (2015). *Automatic Quality Control of Salmon - Using Machine Learning Algorithms based on Input from a 3D Machine Vision System*.

Sture, Øystein et al. (2016). "A 3D machine vision system for quality grading of Atlantic salmon". In: *Computers and Electronics in Agriculture* 123, pp. 142 –148. ISSN: 0168-1699. URL: http://www.sciencedirect.com/science/article/pii/S0168169916300576.

Appendices

Appendix A

Introduction to GPU programming using CUDA

CUDA, which is an abbreviation for *Compute Unified Device Architecture*, is a programming model and abstraction layer used programming GPUs from NVIDIA. This appendix gives a short introduction to CUDA and GPU programming. Despite the overview is rather limited, it should give enough insight to be able to understand the architecture and implementation design descisions discussed in this report. For readers unfamiliar to CUDA, it is therefore strongly advised to read this chapter to learn the basic terms and ideas. To learn more about CUDA, refer to NVIDIAs Best Practices Guide (NVIDIA 2016a) and the Programming Guide (NVIDIA 2016b).

The first section of this chapter presents the general concepts of why and how parallel programming can speed up execution. Next follows sections describing CUDA specific details. It will be described how parallel processes execute on a GPU, how memory is arranged, and how to create an application using a CUDA enabled GPU. At last follows a more detailed description on how the utilize CUDA streams; a powerful tool to be able to run tasks in parallel and asynchronously.

A.1 Parallel programming and processing speed

Gene M. Amdahl stated (Amdahl 1967) that to achieve higher performance, a large portion as possible should be executed in parallel. His text is often rephrased as Amdahl's law, and can be written down as:

$$Speedup = \frac{1}{s + \frac{p}{N}} \tag{A.1}$$

Here s and p is the fraction of respectively sequential and parallel execution, and N is the number of parallel execution lines. From this model it is possible to derive that as N and the fraction of parallel operations and increases, the speedup reaches infinity. Amdahl did also see that this equation had a limited validity range, because with increased parallelism comes increased complexity. John L. Gustafson later stated that *the problem size scales with the number of processors*(Gustafson 1988) and came up with his own model:

$$Speedup = s + p \cdot N \tag{A.2}$$

The two principles these two equations illustrate are often referred to as strong and weak scaling (NVIDIA 2016a). They teach us that in order to increase performance, the parallelism should be increased while keeping the problem complexity low.

A.2 The CUDA execution model

CPUs are designed to minimize the latency, while GPUs are designed is to maximize the throughput. This results in two different hardware implementations. High end CPUs often have only a few pipelines, but is often equipped with instruction level parallelism, advanced prediction schemes, and speculative execution [ch. 2](Hennessy and Patterson 2006). A GPU has the capability of launching thousands of threads in parallel, but let groups of threads, in CUDA called warps, share the same, simpler control logic.

The difference in control logic affects how branching (executing non-linear code) is executed. An if-clause has a condition C, which result executing code A or code B. An advanced CPU with speculative execution might execute both A and B in parallel, before the value of C is finished computing, and commit either A or B when C is present. Therefore, the cost of branching is low. However, on a GPU, this type of control logic is not present. In addition, warps, which are groups of threads, share the same control logic. Therefore, the GPU must first calculate C, and if different threads in the same warp result in different condition values, first A, then B will be executed because the control logic does not allow simultaneous execution of diverging threads.

CUDA does also group threads into blocks, where the size of a block is typically much larger than the warp size. A number of blocks can be executed in parallel on the same or multiple streaming multiprocessors (SMs). Due to this abstraction, CUDA code is portable over a wide range of GPUs, having capabilities of running only a few hundred threads to many thousand threads in parallel. Because newer GPUs might have a newer instruction set, this is expressed by a version number called the compute capability.

A.3 The CUDA memory model

The memories of a GPU and a CPU are physically separated, by CUDA called respectively device and host memory. In general, this means that a GPU cannot operate on host memory, and a CPU cannot operate on device memory. Therefore, to be able to process data from the CPU on the GPU, the data must first be transferred to the device (abbreviated as H2D), and when the operation is complete, be sent back to the host (abbreviated as D2H). Memory allocation is done using `cudaMalloc()`, and memory transfers are done using `cudaMemcpy()`. At a lower level of abstraction, such memory transfers are usually done using a Direct Memory Access controller (DMA), which allows

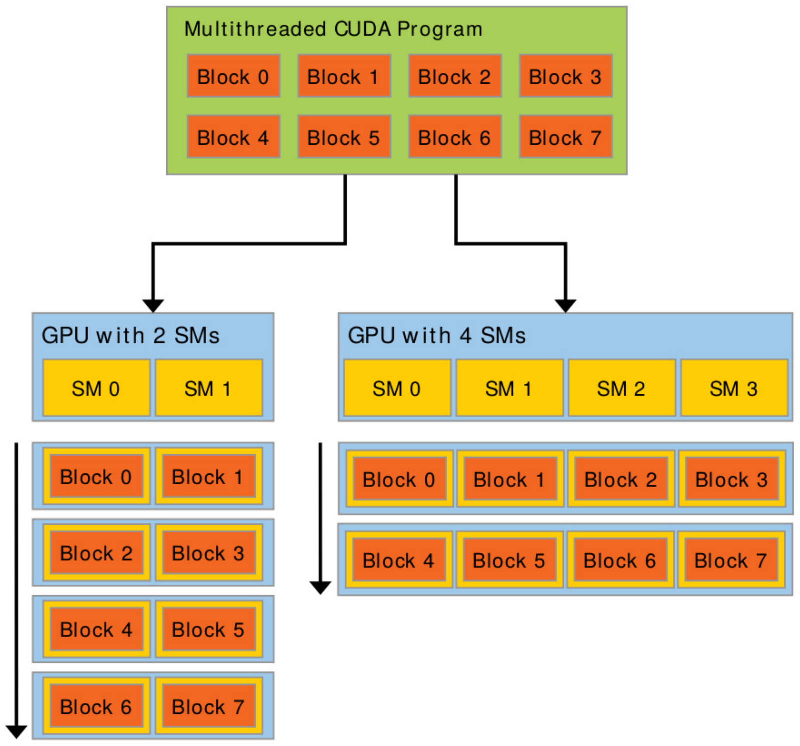


Figure A.1: The abstraction layers using SMs, blocks and threads makes CUDA programs portable over a large range of GPUs. This is Figure 5 taken from the Cuda C Programming Guide (NVIDIA 2016b).

the CPU to continue execution while the memory transfer is in progress. This capability allows for asynchronous memory transfers. This means that when a memory transfer starts, the execution is transferred back to main application. The memory transfer then continues to operate in the background. Asynchronous memory transfers is implemented by the function `cudaMemcpyAsync()`. The number of concurrent transfers between host and device is determined by the direction of transfers, other ongoing operations, and the number of DMA engines, which are called asynchronous engines in CUDA.

Previously it was explained that speedup is achieved when parallelism is increased while time spent on serial execution is kept low. In terms of GPU programming, this often means that the fraction between memory transfers and GPU execution should be low. This can be achieved by reducing the number of memory transfers and by speeding them up. A limiting factor of memory transfer speed is often the virtual memory system. Most operating systems today use a concept of virtual memory. Virtual memory allows us to define a larger memory area than physically available. This allows more applications to be present in memory at once, and simplifies memory isolation between applications. However, this design slows down memory transfers between the CPU and GPU. Before a memory transfer between pages is started, it must be verified if the pages are present in physical memory, and if necessary swapped in.

CUDA allows CPU memory to be pinned, also called to lock pages, of the virtual memory system. This bypasses the availability checking and swapping process, which allows faster memory transfers as shown in Figure A.2. When memory is pinned, CUDA is even able to let the GPU access CPU memory directly, although this might be slow. Memory can be pinned by calling `cudaHostRegister()` (NVIDIA 2012).

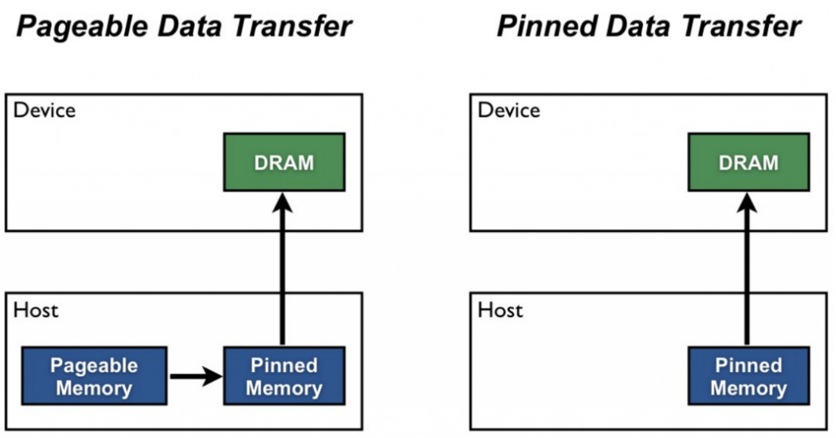


Figure A.2: Pinned memory transfers data faster between host and device.

CPUs normally have to types of memory, global memory and registers, in addition to caches to speed up accesses to global memory. Global memory is large, but slow, while registers fast, but a limited resource. CUDA GPUs also have global memory and register memory. When calling `cudaMalloc()`, global memory is allocated. In addition to these two types of memory, GPUs also have three other types of memory:

- *Shared memory* is memory that is shared between all threads in a block. It is faster than global memory, but slower than registers. Using shared memory it is possible to speed up i.e. matrix multiplication drastically. The shared memory has more size restrictions and affect how many thread blocks than can run in parallel.
- *Constant memory* acts as a kind of cache memory. It is as fast as reading registers, given that all threads in a warp read the same address. However, when different addresses are read, the read operations are serialized, which increases latency. Constant memory is also limited in size, and cannot be changed during execution.
- *Texture memory* is another type of read-only cached memory. When texture memory is read, nearby memory in a multidimensional memory space is also cached. Therefore, texture memory is often used for multidimensional image processing. Because the caching scheme is more advanced than for constant memory, the minimum latency is larger; however it is more optimized for multi-address access.

A.4 The CUDA programming model

Functions run a GPU are in CUDA named kernels. Kernels are called just the same way as normal sequential functions are called, however the degree of parallelism must be specified.

This is specified as the number of blocks and the number of threads per block that should be launched for this function call. The syntax for this is `function_name<<<blockdim, threaddim>>>(parameter list)`. In addition, the function prototype must be prefixed with `__global__` so that the compiler knows this function should be compiled for GPU execution.

The following sequence of operations summarize how data can be processed on a GPU:

1. Call `cudaMalloc(...)` to allocate memory on the GPU.
2. Call `cudaMemcpy(..., cudaMemcpyHostToDevice)` to transfer data from the CPU to the GPU.
3. Call `function_name<<<blockdim, threaddim>>>(...)` to launch a CUDA kernel.
4. Call `cudaMemcpy(..., cudaMemcpyDeviceToHost)` to transfer data back from the GPU to the CPU.
5. Call `cudaFree(...)` to deallocate the memory on the GPU.

A.5 Asynchronous execution using CUDA streams

One of the key features of CUDA GPU programming is asynchronous execution. This means that memory transfers and kernels can be started, and while they are running, the CPU can continue with other tasks. Using CUDA streams, this will in some cases even allow parallel execution of GPU kernels and memory transfers.

A CUDA stream is simply a kind of first-in-first-out (FIFO) buffer of operations. The CPU commits operations to streams, which are for example memory transfers or kernel executions. The GPU reads this queue, and is responsible for executing them. This means that operations on streams are executed asynchronously from the CPU point of view. Because the operations are asynchronous, CUDA has added synchronization primitives to be able to prevent data races and informing to signal when operations are finished. One such example is the function `cudaStreamSynchronize()`, which waits until all operations on a given stream are finished.

Both memory transfers and kernel execution can be committed to a stream and it is therefore possible to commit a H2D copy, kernel execution and D2H copy twice to two different streams before synchronizing. The resulting execution trace depends on the capabilities of the GPU. It might execute all operations serially, or they might be executed in parallel. If the tasks are executed in parallel depends on a number of properties described in detail in CUDAs C programming guide [sec. 3.2.5](NVIDIA 2016b). In general, it is required to have multiple SMs, pinned host memory, having one or more asynchronous engines etc. Even though a GPU might not be able to operate all operations in parallel, it might be valuable to design the program to use multiple streams in case the program will be ported to a more advanced GPU later.

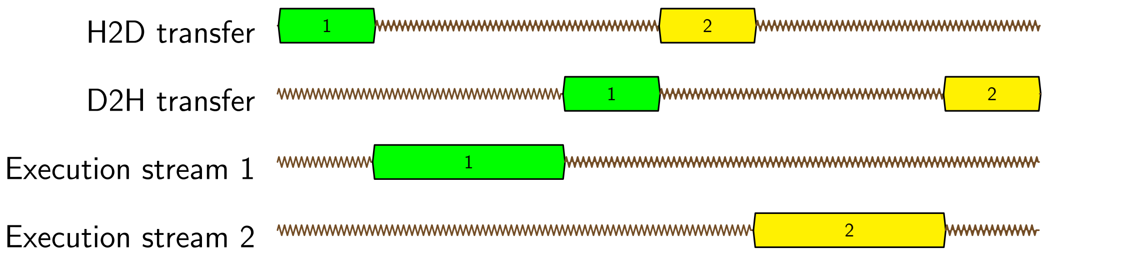
A.5.1 Some execution traces

This subsection presents some possible execution traces for GPUs with different capabilities. The purpose is to give deeper insight how the capabilities of a GPU might limit or increase the throughput of a process. In the general case, there exists a large set of possible execution traces. Therefore, some assumptions are set to limit the number of traces. First, the assumption is made that there is a continuous stream of GPU tasks. Secondly it is assumed that memory transfers and kernel executions is concurrent. The analysis is then only based on two properties of the GPU:

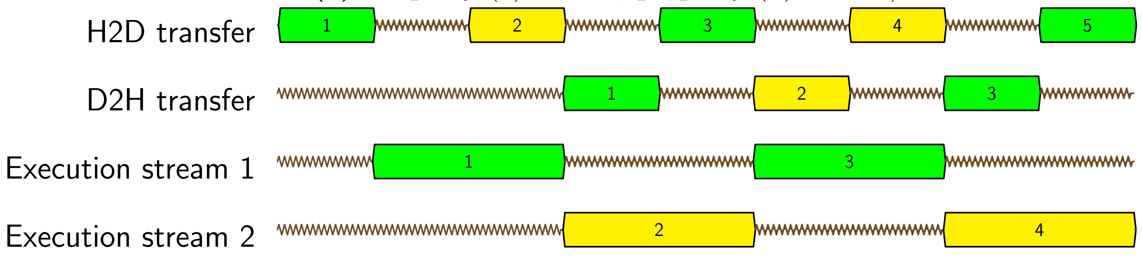
(a) If H2D and D2H transfers are independent. When this property is false, the sequence of H2D (stream 1), D2H (stream 1), H2D (stream 2), will block on H2D (stream 2) until D2H (stream 1) is finished. This requires at least two copy engines.

(b) If kernel executions are concurrent. The maximum concurrency of the GPU, that is the maximum number of kernels that can be executed concurrently, depends on the kernel and GPU itself. This property is therefore more difficult to verify.

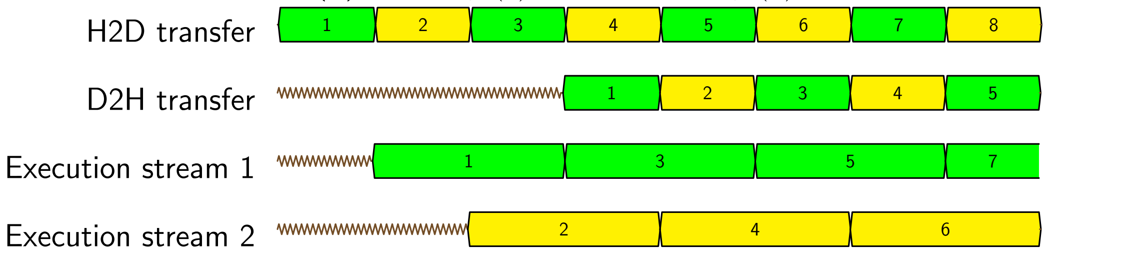
In Figure A.3 traces are given for when both input and output data is stored in device memory. Therefore, it is necessary with both a H2D and a D2H transfer. In Figure A.4 traces are presented when output data is located accessed through pinned host memory. For this case, it is then only necessary with a H2D transfer. It is clear that this configuration does not need property (a) to be true to achieve high throughput. Therefore, for this configuration will likely be favorable for low end GPUs. However, it must be remembered that memory access to output data will be slower, and therefore it should be verified which configuration gives the best performance.



(a) Property (a) is false, property (b) is false)

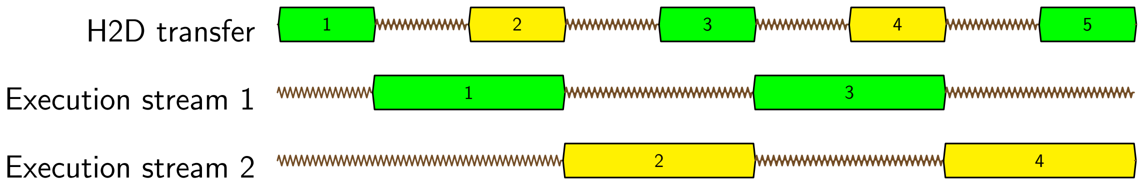


(b) Property (a) is true, property (b) is false.

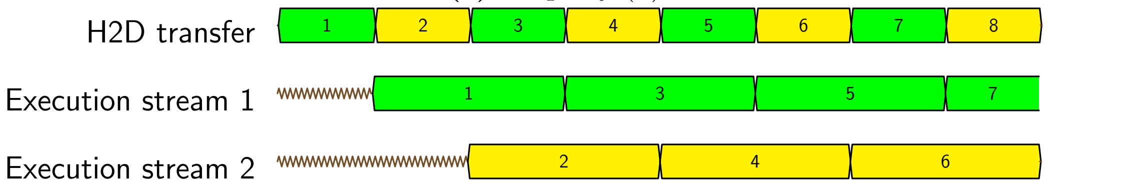


(c) Property (a) is true, property (b) is true.

Figure A.3: Execution traces when both input and output data is stored in device memory.



(a) Property (b) is false.



(b) Property (b) is true.

Figure A.4: Execution traces when only input data is stored in device memory. Note that the traces are almost identical to the traces (b) and (c) in Figure A.3, without requiring property (a) to be true.

Appendix B

Setup of the Jetson TK1 and Jetson TX1

This chapter describes the configuration of the operating system tools and libraries on the NVIDIA Jetson TK1 and Jetson TX1. First the proposed setup is given, next follows a step-by-step guide describing how to configure the Jetsons to the setup used by this thesis.

B.1 Proposed setup

The proposed setup contains a version of Linux for Tegra, a variant of the *Flycapture2 SDK* for the camera, and a CUDA toolchain. This setup was chosen because of limitations of the used equipment, performance reasons, and for increasing the ease of use.

Linux for Tegra (L4T) is a package for the Jetsons created by NVIDIA containing a bootloader and Ubuntu file system. The TK1 uses version 21.5, which contains Ubuntu 14.04, while the TX1 uses version 24.2.1 which contains Ubuntu 16.04. Because Linux for Tegra is the default setup, the number of installation guides and available support is large, which decreases setup time.

The Flycapture2 SDK was chosen because it is the only Linux compatible SDKs for the PointGrey camera. Because the Jetson TK1 and TX1 run two different Ubuntu versions, two different SDK versions must be used: 2.9.3.43 and 2.10.3.266. It is doubtful if the Flycapture2 SDK will work on other Linux distributions, because it depends on many libraries only available through Ubuntu. These are for example `libraw1394-8`, `libgtkmm-2.4-dev`, and `libusb-1.0`. The full set of requirements are found in Technical Application Note 2009003 (Point Grey 2016c).

The CUDA toolchains were installed on the Jetsons because the NVIDIA Nsight IDE was unable to configure projects for cross compilation (see Appendix C). The CUDA toolchains available for the Jetson TK1 and TX1 are for CUDA version 6.5 and 8.0. Unfortunately, these toolchains are not available through direct download. However, these are available through the NVIDIA Jetpack, which is an installation package installing both Linux for Tegra, IDE, and toolchain.

The Jetson TK1 and TX1 are designed to maximize computational power per energy unit. Therefore, these computers contain many configuration options for turning on and off power saving features. For this thesis, power consumption is not an issue. Therefore, the Jetsons can be configured for maximum performance. This includes enabling all CPU cores, use the high performance CPU cores, and maximizing the CPU, GPU, and memory clock rate. The Jetson TX1 sample file system contains a script performing all these operations. During this project, the script has been generalized to work on both the TK1 and TX1. The script is included in the delivery as `jetson_clocks.sh`.

B.2 Step-by-step installation guide

This section is a step-by-step guide on how the software was installed and configured on the NVIDIA Jetson TK1 and Jetson TX1. First, it describes how the file system is set up and flashed. Then it is described how the CUDA Toolkit and the binaries of the *Flycapture2 SDK* are installed.

For clarity: Files and folders are emphasized as `my_folder/my_file`, text to be inserted or modified is emphasized as *some text*. Command line statements are emphasized as `my_command`.

The following list gives the prerequisites for the setup. Software components belonging to the Jetson can be downloaded from https://developer.nvidia.com/embedded/downloads. The *Flycapture2 SDK* can be downloaded from https://www.ptgrey.com/support/downloads:

- A host computer running Ubuntu 14.04. The newest stable release, 16.04, is not supported.
- A NVIDIA Jetson TK1 and/or NVIDIA Jetson TX1.
- A micro USB cable for flashing the boards.
- A keyboard and monitor or a serial cable. This is used to access its terminal.
- A driver package. The Jetson TK1 uses version 21.5 where the downloaded file is named `Tegra124_Linux_R21.5.0_armhf.tbz2`. The Jetson TX1 uses version 24.2.1 where the downloaded file is named `Tegra210_Linux_R24.2.1_aarch64.tbz2`.
- The L4T Sample Root Filesystem. The Jetson TK1 uses version 21.5 where the downloaded file is named `Tegra_Linux_Sample-Root-Filesystem_R21.5.0_armhf.tbz2`. The Jetson TK1 uses version 24.2.1 where the downloaded file is named `Tegra_Linux_Sample-Root-Filesystem_R24.2.1_aarch64.tbz2`.
- The Flycapture2 SDK files. The Jetson TK1 uses version 2.9.3.43 where the downloaded file is named `flycapture.2.9.3.43_armhf.tar.gz`. The Jetson TX1 uses version 2.10.3.266 where the downloaded file is named `flycapture.2.10.3.266_arm64.tar.gz`.

- A CUDA Toolkit for L4T. The Jetson TX1 uses version 6.5 for L4T version 21.5, while the TX1 uses version 8.0 for L4T version 24.2.1. These toolkits can unfortunately not be downloaded directly from the support websites of NVIDIA. However, by reading the configuration files of the NVIDIA Jetpack installer direct links were obtained: `http://developer.download.nvidia.com/devzone/devcenter/mobile/jetpack_l4t/005/linux-x64/cuda-repo-l4t-r21.5-6-5-local_6.5-53_armhf.deb` and `http://developer.download.nvidia.com/devzone/devcenter/mobile/jetpack_l4t/006/linux-x64/cuda-repo-l4t-8-0-local_8.0.34-1_arm64.deb`.

B.2.1 Configure and flash the operating system

The steps in the list below describe how the bootloader, kernel and file system are configured and flashed onto the Jetson. NVIDIA has created a setup, called NVIDIA Jetpack, which does all this, however, it does not configure the USB port to use USB3 for the Jetson TK1. The best way is therefore to follow this guide.

1. Extract the Jetson TK1 Driver Package. The folder should be named `Linux_For_Tegra`.
2. Extract the Sample Root Filesystem in the folder `Linux_For_Tegra/rootfs`.
3. Next the system must be modified to use USB3 instead of USB2, however this step is only necessary for the TK1. First ensure line 38 in `Linux_For_Tegra/jetson-tk1.conf` contains the value *0x6209C000*. Change *usb_port_owner_info=0* to *usb_port_owner_info=2* in the file `Linux_For_Tegra/bootloader/ardbeg/jetson-tk1_extlinux.conf.emmc` to use USB3. Then and add the line *usbcore.usbfs_memory_mb=1000* to the same file to increase the internal USB buffer size.
4. To compile and create the binaries to be transferred, run `sudo ./apply_binaries.sh`.
5. Before the Jetson can be flashed, it must be set in bootloader mode. This is done by holding the recovery button and pressing the reset button. Verify that the device has entered bootloader mode, run the command `lsusb` and check that an NVIDIA device appears on the list.
6. Run `sudo ./flash jetson-tk1 mmcblk0p1` or `sudo ./flash.sh jetson-tx1 mmcblk0p1` to flash the bootloader, kernel, and file system onto the Jetson. This may take a while.

Now that the Jetson is up and running, hook up a keyboard and monitor or connect to it using a serial cable. Log in using the credentials *Username: ubuntu, Password: ubuntu.* If the device is connected to the internet, change its password by using the command `passwd`. After this is done it is possible to connect to the device using ssh. Update the system to use the newest updates using `sudo apt-get update` and `sudo apt-get upgrade`, however do not upgrade to newer versions of Ubuntu.

B.2.2 Install CUDA

These steps assumes the system is accessed through *ssh*. Because the project setup compiles the programs directly on the Jetson, the entire toolchain must be installed. This is done by following the next steps:

1. From the host, copy the CUDA Toolkit to device using `scp <toolchain_name.deb> ubuntu@ip_address:~`.
2. On the target install the package by running `sudo dpkg -i <toolchain_name.deb>`.
3. Now update the system sources, run `sudo apt-get update`.
4. Install the CUDA toolkit, run `sudo apt-get install cuda-core-6-5 cuda-toolkit-6-5` or `sudo apt-get install cuda-core-8-0 cuda-toolkit-8-0`.
5. Now that the `nvcc` compiler is installed, it must be added to the PATH variable of the system. This is done by adding */usr/local/cuda-6.5/bin* or */usr/local/cuda-8.0/bin* to the PATH variable in `/etc/environment`. Run `. /etc/environment` when finished. Verify that the `nvcc` compiler is found by running `nvcc –version`.
6. The last step is to add the CUDA libraries to the libraries search path. Edit the file `etc/ld.so.conf.d/cuda.conf` and add the line *usr/local/cuda-6.5/lib* or *usr/local/cuda-8.0/lib*. Close and save the file and run `sudo ldconfig`.

B.2.3 Install the *Flycapture2 SDK* binaries

The Flycapture2 SDK binaries must be installed on the Jetson in order to use the camera. Because the system is compiled locally on the Jetsons, the header files must also be copied to the correct include folders.

1. First some prerequisite libraries must be installed. This is done by running `sudo apt-get install build-essential libraw1394-11 libgtkmm-2.4-1c2a libusb-1.0-0`. Ignore possible errors.
2. From the host copy the archive to the device: `scp <sdk_files.tar.gz> ubuntu@ip_address:~`. Next extract it using `tar -xvzf <sdk_files.tar.gz>`.
3. Install the SDK by navigating into the flycapture folder and run `sudo ./flycap2-conf`. When prompted with which user to add to some video groups, use *ubuntu*. Reboot the system when complete.

4. Next is to move the library binaries and include folders to the correct folders in the file system. This is done by the following steps:
   ```
   sudo mkdir /usr/include/flycapture
   cd ~/flycapture.<version>/include/
   sudo cp -r * /usr/include/flycapture/
   sudo mkdir /usr/lib/flycapture
   cd ~/flycapture.<version>/lib/
   sudo cp -r * /usr/lib/flycapture/
   ```

5. Finally, the libraries can be added to the library search path. Edit the file `etc/ld.so.conf.d/flycapture.conf` by adding the lines */usr/lib/flycapture* and */usr/lib/flycapture/C*. Run `sudo ldconfig` to complete.

Appendix C

Development environment setup

This chapter describes how the development environment for the Jetson TK1 and Jetson TX1 was configured. It is required that the Jetsons are configured as described in Appendix B. The first section describes the proposed setup, and section C.2 is a step-by-step installation guide.

C.1 Proposed setup

Using an Integrated Development Environment (IDE) simplifies development. An IDE usually provides both code completion, debug functionality, and an integrated toolchain. Both Nsight Visual Studio Edition and Nsight Eclipse Edition and provide advanced CUDA GPU debugging and GPU profiling. The Visual Studio Edition runs on Windows only, and does not support remote development. Therefore, the Eclipse edition is the only available IDE compatible with NVIDIA Jetson development.

Nsight Eclipse Edition allows two different types of remote development projects (NVIDIA 2014). A cross-compiled project compiles the application on the host computer, while a synchronized project transfers the files to the device and compiles them there. It is beneficial to compile the program on the host computer because it speeds up the compilation speed. However, it was found impossible to configure Nsight from cross-compilation because NVIDIA does not longer provide any up to date guide for this configuration. The resulting development environment therefore uses synchronized projects. This creates some overhead in compiling and file transfer, but delivers most of the functionality directly in the IDE. The entire application in this project uses about one (TX1) or two (TK1) minutes to compile when all source files are rebuilt.

The Nsight debugger was found to be very slow. It was therefore often beneficial to edit the program, recompile, and restart the application instead of launching a debug session. This issue will probably be present when using cross-compiled projects as well.

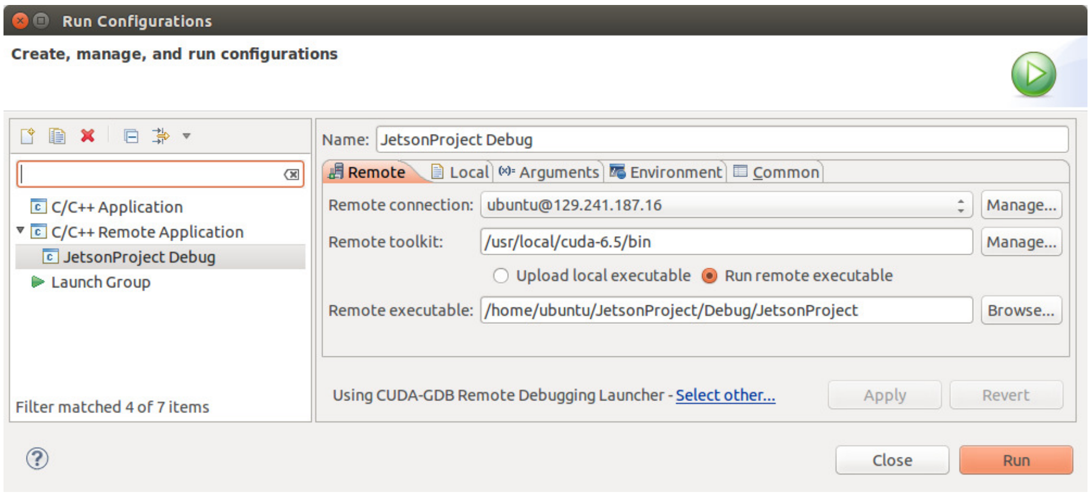
C.2 Step-by-step installation guide

This is a step-by-step guide for installing Nsight Eclipse Edition and how to configure the projects to use synchronized developed projects. The guide is tested with the Jetsons configured as described in Appendix B.

A prerequisite for synchronized remote projects is that the target system is connected to the same network, *ssh* is available, and that *git* is installed on both the host and target system. The host computer must run Ubuntu 14.04. Nsight Eclipse Edition does unfortunately not support the latest stable Ubuntu release, 16.04.

The easiest way to install Nsight Eclipse Edition is to run through the NVIDIA Jetpack installer using the installation guide found at `http://docs.nvidia.com/jetpack-l4t/`. For this project, Jetpack version 2.3.1 is used which supplies Nsight version 8.0. The installation guide will also prompt to install a preconfigured operating system on the Jetson. Feel free to do this, but remember to replace it with the setup described in Appendix B afterwards.

To develop an application for the Jetson, it should be started by using *Empty CUDA Project* as project type in the new project wizard in Nsight. When the source files are added, the build, debug, run, and profile configuration should be edited to use a remote toolchain. These configuration windows should look similar to what is shown in Figure C.1. The build configuration is found in the menu *Project* → *Properties* → *Build* → *Target Systems*. The run, debug, and profile configurations, are found in the *Run* menu bar.


Run Configurations

Create, manage, and run configurations

- C/C++ Application
- C/C++ Remote Application
  - JetsonProject Debug
- Launch Group

Filter matched 4 of 7 items

Name: JetsonProject Debug

Tabs: Remote | Local | Arguments | Environment | Common

Remote connection: ubuntu@129.241.187.16

Remote toolkit: /usr/local/cuda-6.5/bin

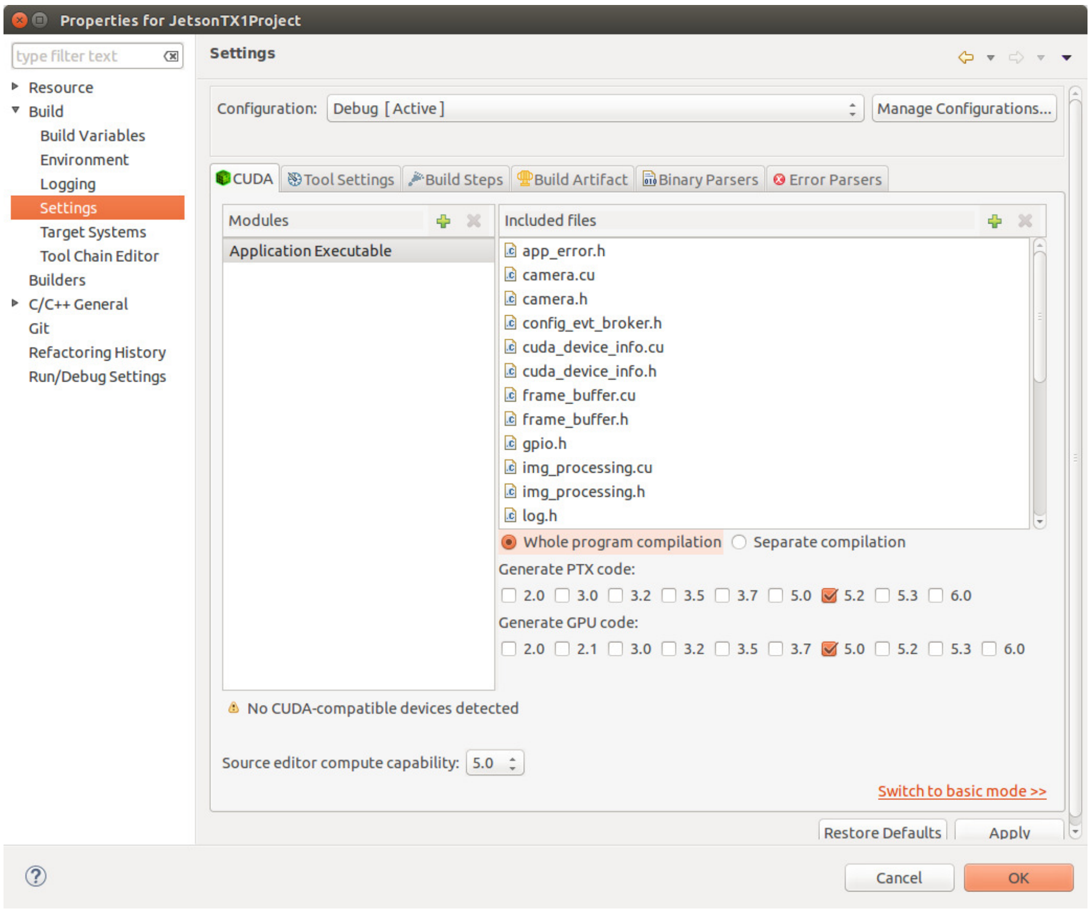
Upload local executable [✓ Run remote executable]

Remote executable: /home/ubuntu/JetsonProject/Debug/JetsonProject

Using CUDA-GDB Remote Debugging Launcher - Select other...

Figure C.1: Configuration windows for remote synchronized projects in NVIDIA Nsight Eclipse Edition. For the Jetson TX1, the remote toolkit path must be modified.

The Jetson TK1 and Jetson TX1 have different CUDA compute capabilities. NVIDIA Nsight must therefore be configured differently for these two boards. The compute capability options are found in *Project* → *Properties* → *Build* → *Settings*. The Jetson TK1 is should use the configuration <3.2, 3.0>, while the TX1 should use <5.2, .0>. Figure C.2 shows the configuration window where the compute capability is selected.



Properties for JetsonTX1Project

type filter text

- Resource
- Build
  - Build Variables
  - Environment
  - Logging
  - Settings
  - Target Systems
  - Tool Chain Editor
- Builders
- C/C++ General
- Git
- Refactoring History
- Run/Debug Settings

Settings

Configuration: Debug [Active]   Manage Configurations...

CUDA | Tool Settings | Build Steps | Build Artifact | Binary Parsers | Error Parsers

Modules: Application Executable

Included files:
- app_error.h
- camera.cu
- camera.h
- config_evt_broker.h
- cuda_device_info.cu
- cuda_device_info.h
- frame_buffer.cu
- frame_buffer.h
- gpio.h
- img_processing.cu
- img_processing.h
- log.h

[✓ Whole program compilation] Separate compilation

Generate PTX code: ☐ 2.0 ☐ 3.0 ☐ 3.2 ☐ 3.5 ☐ 3.7 ☐ 5.0 ☑ 5.2 ☐ 5.3 ☐ 6.0

Generate GPU code: ☐ 2.0 ☐ 2.1 ☐ 3.0 ☐ 3.2 ☐ 3.5 ☐ 3.7 ☑ 5.0 ☐ 5.2 ☐ 5.3 ☐ 6.0

No CUDA-compatible devices detected

Source editor compute capability: 5.0

Switch to basic mode >>

Restore Defaults   Apply

Cancel   OK

Figure C.2: The synchronized projects must be configured to use the right CUDA compute capability. This windows shows how the IDE is configured for the Jetson TX1.

When these steps are performed, the development environment is configured. It is possible to add multiple build targets in order to develop for both the Jetson TK1 and TX1 at the same time. However, this sometimes leads to merge conflicts. These are easiest resolved by removing the entire project directory on the Jetsons. By rebuilding the project, these files are resynchronized and rebuilt.

Appendix D

Using the applications and tools

This appendix chapter describes how to run the developed applications in order to test and benchmark the developed scanner. First, in section D.1, it is presented how the line scanner module is compiled and launched. Next follows a description of two configuration tool, which performs line scanner module configuration in either Windows or Ubuntu. In section D.3 it is explained how the LabVIEW host is tested and configured. The last two sections of this appendix describe the benchmarker and a python tool for scan data visualization in Ubuntu.

The necessary source code and project setups is delivered together with this report. Each of the applications are found in separate folders. It is required that the Jetsons are flashed and programmed as described by Appendix B, and the test setup installed as described in Appendix C.

D.1 The line scanner module

The line scanner module application implementation is described in chapter 4. Using NVIDIA Nsight Eclipse Edition, it is possible to import a project from the folder `JetsonTX1Project`. It might be necessary to reconfigure the project to set the correct IP addresses, CUDA compute capabilities, and build paths for the target system. Appendix C describes how this is done.

Before the application is started, it must be built. Compiling is done by pressing *CTRL + B*. This might take more than two minus on the Jetson TK1 and one minute for the Jetson TX1. The build status is shown in the console at the lower half of the screen. When the application is built, it must be started remotely. The application cannot be launched from NVIDIA Nsight, because it requires super user rights for changing execution priority. Therefore, launch the application through ssh: `ssh ubuntu@<ip_address>`, `sudo ./<application>`. The application logs a message *Application started* to the terminal when the application is started.

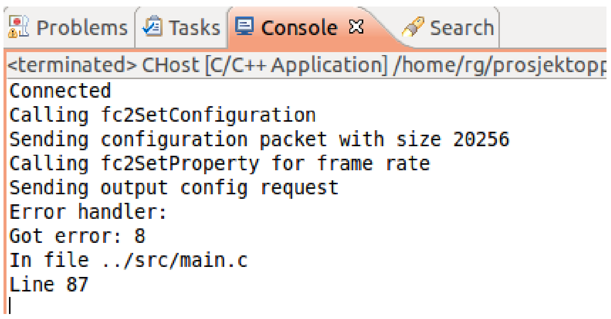
D.2 The configuration tools

The configuration tools are applications that connect to a line scanner module, sends camera and scanner configuration parameters, before they disconnects. A project created in Visual Studio can be extracted from the folder `WindowsCHost`, while a project created in Eclipse can be extracted from the folder `ConfigurationTool`.

The port, IP address, and the given configuration executions of the Jetson must be set in `main.c`. The most important system parameters are found in the two files `scan_config.c` and `camera_config.c`. In the file `scan_config.c` the scanner settings and image processing parameters are set. Using `camera_config.c` it possible to configure camera parameters such as white balance, gain, and shutter time.

Because the configuration tool is applicable for both the Jetson TK1 and Jetson TX1 line scanner module, and because they use different versions of the *FlyCapture2 SDK*, these must be differentiated. When `FLYCAPTURE2_SDK_1404` is defined, the camera configuration will only be accepted on the TK1. When `FLYCAPTURE2_SDK_1604` is defined, the configuration calls are only applicable on the TX1. Because the length of the configuration calls differ, an invalid configuration will be denied by the line scanner module.

The application is started by pressing either *CTRL + F5* on Windows, or by pressing *CTRL + F11* on Ubuntu. When invalid parameters are set, this is shown in the console window as in Figure D.1. The definition of error codes are found in `ret_code.h`.



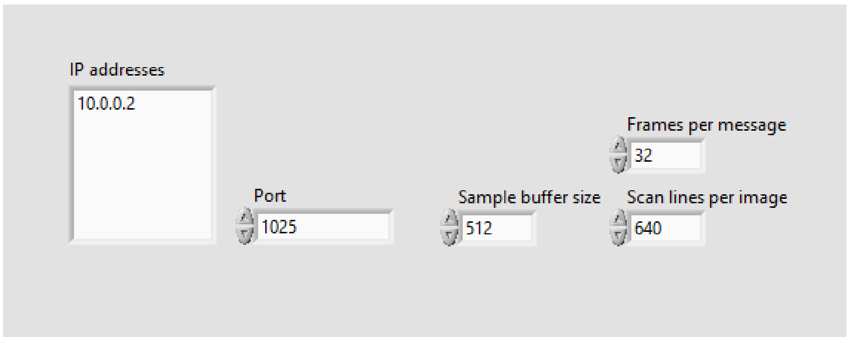
```
<terminated> CHost [C/C++ Application] /home/rg/prosjektopp
Connected
Calling fc2SetConfiguration
Sending configuration packet with size 20256
Calling fc2SetProperty for frame rate
Sending output config request
Error handler:
Got error: 8
In file ../src/main.c
Line 87
```

Figure D.1: Console output of the Ubuntu configuration tool when an error is reported.

D.3 The LabVIEW host

The line scanner module should be started as described in the previous section before the host control system is launched. The LabVIEW host control system retrieves and visualizes data from a line scanner module. The application requires the host computer to run Windows with LabVIEW and the IMAQ package installed. Its implementation is described in chapter 5.

Before the application is started, it must first be configured with some parameters. These are found in *TestScannerPanel.vi*. Fill in the parameters similar to the values shown in Figure D.2. When this is done, press the run button in the top left corner. To include visualization open *2DVisualization.vi* and/or *3DVisualization.vi*. A sample of 2D visualization and 3D visualization is given in Figure 5.5 and Figure D.3



IP addresses: 10.0.0.2
Port: 1025
Sample buffer size: 512
Frames per message: 32
Scan lines per image: 640

Figure D.2: *TestScannerPanel.vi* in the host control system application must be configured similar to the configuration shown here.

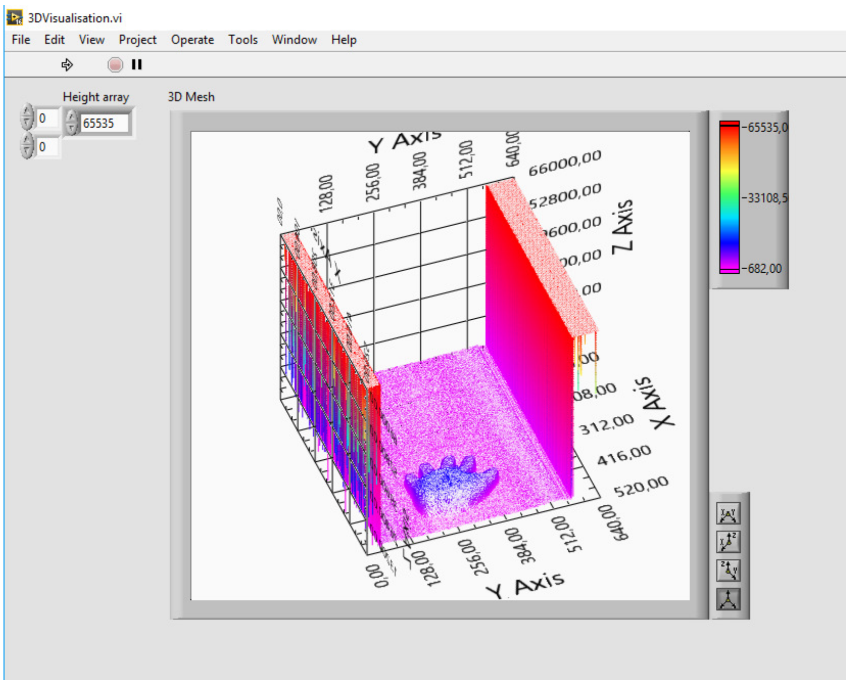


Figure D.3: A sample of the visual output of *3DVisualization.vi*.

D.4 The benchmarker

The benchmarker is a variant of the configuration tool for Ubuntu. Does nothing more that starting the scanner with a given image buffer and block size configuration, decodes the GPU and TCP processing time and stores the results to text files. Because of its simplicity, it is assumed that the benchmarker affects the TCP send time measurements as little as possible.

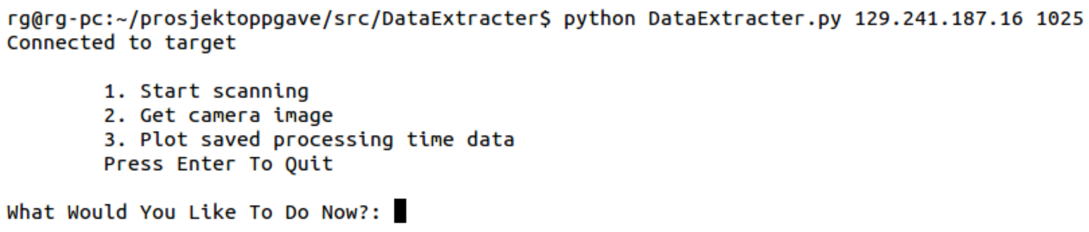
The benchmarker functionality is added by adding the lines `benchmarker_do_all()` or `benchmarker_start()` to `main.c`, , which either starts a range of tests or one test case. The test case parameters are found in `benchmarker.c` and `benchmarker.h`. The benchmarker is started the same way as the configuration tool: By pressing *CTRL + F11*.

The name of the output files, such as *Wed May 3 08_53_43 2017 ImgBuf2BlockSize128.txt*, is a combination of the time the tests started, the image buffer size, and thread block size. The content of the benchmark files is first a summary, then the GPU benchmark values, then the TCP benchmark values, and finally the total execution time results. The summary includes the minimum, maximum, average and variance of the measurements. This predefined format simplifies the extraction and comparison with other test results.

D.5 The data extractor

The data extractor is a Python client made to visualize and record both line data and benchmark parameters in Ubuntu. The tool makes it easy to validate the correctness of the received data, without needing to switch to a Windows computer during development. The data extractor is found in the `DataExtracter` folder. The application runs using python 2.7 and has dependencies to python libraries *numpy* and *matplotlib*. The libraries are installed using: `sudo apt-get install python-pip python-dev` and `sudo pip install numpy matplotlib`.

The start menu gives three options as shown in Figure D.4. The first option enters a new menu and changes the state of the target to scan mode. The second function retrieves an image from the device, which is useful for camera and laser alignment. The last option, provided a file name from a previously recorded benchmark, plots histograms of the latency and throughput measurements.



```
rg@rg-pc:~/prosjektoppgave/src/DataExtracter$ python DataExtracter.py 129.241.187.16 1025
Connected to target

        1. Start scanning
        2. Get camera image
        3. Plot saved processing time data
        Press Enter To Quit

What Would You Like To Do Now?: █
```

Figure D.4: The start menu of the data extractor. The data extractor must be executed with target IP address and port as arguments.

In scanning mode options are provided to either show the sampled height profile, visualize the benchmark results, or save benchmarks to file. The visualization tools plot data in real time, at these tools are therefore very useful for testing.

The file format of the benchmarks is simple. Each line contains one sample. A set of samples is preceded with either "Callback processing time:" or "Image processing time". It is therefore possible to add extra information to the file using a text editor and still be able to plot the data using option 3 in the start menu.

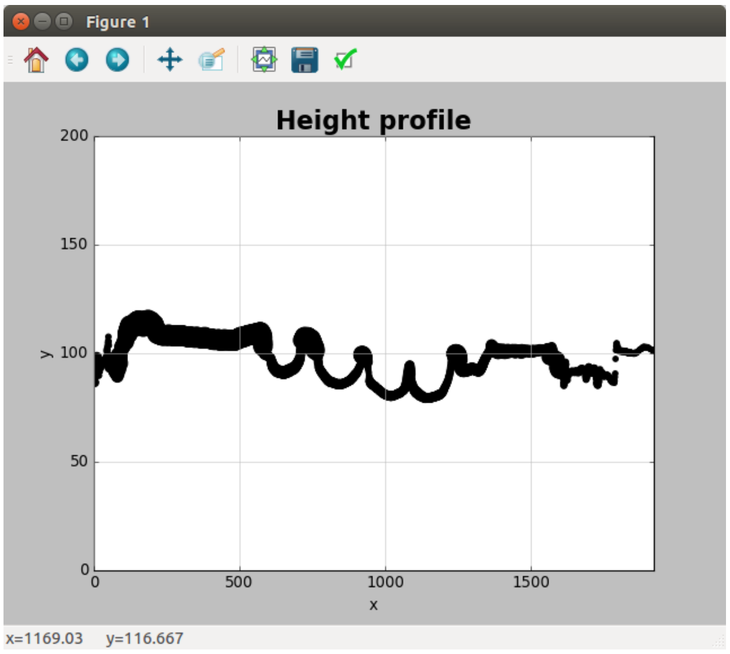


Figure D.5: Visualization of the height profile using the data extractor. The thickness of the line corresponds to the measured scatter. The height profile is updated in real time.

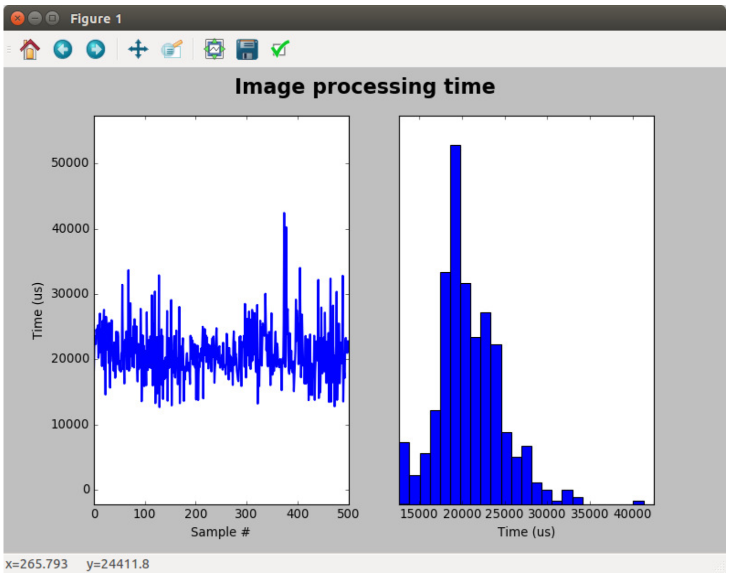


Figure D.6: Visualization of the image processing time obtained by the data extractor. The histogram is updated in real time.

Appendix E

Code snippets

This appendix includes all code snippets referred to from the report. Many of the code snippets here are only extractions from the actual implementation files. Therefore, refer to those when a deeper understanding is needed. The first section of this chapter lists configuration structure definitions. Next the code snippets regarding image parameter extraction is given. The last section of this chapter presents other code snippets not fitting into the two first categories.

E.1 Configuration structure definitions

The configuration structures and enumerations are C code equivalents to message structure definitions defined in subsection 3.1.1. The definitions are presented in C code because the representation is close to the real byte representations.

Listing E.1: Definition of camera configuration function enumeration and data structure.

```c
/** @brief Camera configuration types.
 *
 *  @details The configuration types correspond to the FlyCapture2 SDK configuration
      calls.
 */
typedef enum
{
    CAMERA_CONFIG_fc2GetConfiguration          = 0x01,
    CAMERA_CONFIG_fc2SetConfiguration          = 0x02,
    CAMERA_CONFIG_fc2GetCameraInfo             = 0x03,
    CAMERA_CONFIG_fc2GetPropertyInfo           = 0x04,
    CAMERA_CONFIG_fc2GetProperty               = 0x05,
    CAMERA_CONFIG_fc2SetProperty               = 0x06,
    CAMERA_CONFIG_fc2GetVideoModeAndFrameRateInfo = 0x07,
    CAMERA_CONFIG_fc2GetVideoModeAndFrameRate   = 0x08,
    CAMERA_CONFIG_fc2SetVideoModeAndFrameRate   = 0x09,
    CAMERA_CONFIG_fc2GetFormat7Info            = 0x0A,
    CAMERA_CONFIG_fc2ValidateFormat7Settings    = 0x0B,
    CAMERA_CONFIG_fc2GetFormat7Configuration    = 0x0C,
    CAMERA_CONFIG_fc2SetFormat7ConfigurationPacket = 0x0D,
    CAMERA_CONFIG_fc2SetFormat7Configuration    = 0x0E,
    CAMERA_CONFIG_fc2GetEmbeddedImageInfo       = 0x0F,
    CAMERA_CONFIG_fc2SetEmbeddedImageInfo       = 0x10,
    CAMERA_CONFIG_fc2GetTriggerModeInfo         = 0x11,
    CAMERA_CONFIG_fc2GetTriggerMode             = 0x12,
    CAMERA_CONFIG_fc2SetTriggerMode             = 0x13,
    CAMERA_CONFIG_fc2GetTriggerDelayInfo        = 0x14,
    CAMERA_CONFIG_fc2GetTriggerDelay            = 0x15,
    CAMERA_CONFIG_fc2SetTriggerDelay            = 0x16,

    CAMERA_CONFIG_FORCE32_BIT                   = 0xFFFFFFFF
} camera_config_type_t;


/** @brief Configuration data for the FlyCapture2 SDK calls. */
typedef struct
{
    camera_config_type_t type;                    /**< Type of configuration. */

    union
    {
        fc2Config         config;                     /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetConfiguration or @ref CAMERA_CONFIG_fc2SetConfiguration. */
        fc2CameraInfo     cameraInfo;                 /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetCameraInfo. */
        fc2PropertyInfo   propertyInfo;               /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetPropertyInfo. */
        fc2Property       property;                   /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetProperty or @ref CAMERA_CONFIG_fc2SetProperty. */
        fc2TriggerModeInfo triggerModeInfo;           /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetTriggerModeInfo. */
        fc2TriggerMode    triggerMode;                /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetTriggerMode or @ref CAMERA_CONFIG_fc2SetTriggerMode. */
        fc2TriggerDelayInfo triggerDelayInfo;         /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetTriggerDelayInfo. */
        fc2TriggerDelay   triggerDelay;               /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetTriggerDelay or @ref CAMERA_CONFIG_fc2SetTriggerDelay. */

        struct
        {
            fc2VideoMode videoMode;
            fc2FrameRate frameRate;
            BOOL         supported;
        } video_framerate_info;                   /**< Config struct used when type is @ref CAMERA_CONFIG_fc2GetVideoModeAndFrameRateInfo, @ref CAMERA_CONFIG_fc2GetVideoModeAndFrameRate, or @ref CAMERA_CONFIG_fc2SetVideoModeAndFrameRate. */

        struct
        {
            fc2Format7Info format7Info;
            BOOL           supported;
        } format7Info;                              /**< Config struct used when type
            is @ref CAMERA_CONFIG_fc2GetFormat7Info. */

        struct
        {
            fc2Format7ImageSettings imageSettings;
            fc2Format7PacketInfo  packetInfo;
            BOOL                  supported;
        } format7Settings;                          /**< Config struct used when type
            is @ref CAMERA_CONFIG_fc2ValidateFormat7Settings.*/

        struct
        {
            fc2Format7ImageSettings imageSettings;
            union
            {
                unsigned int          packetSize;
                float                 percentage;
            };
        } format7Configuration;                     /**< Config struct used when type
            is @ref CAMERA_CONFIG_fc2GetFormat7Configuration,
            CAMERA_CONFIG_fc2SetFormat7ConfigurationPacket, or
            CAMERA_CONFIG_fc2SetFormat7Configuration.*/

        fc2EmbeddedImageInfo  embeddedImageInfo;
    } data;

} camera_config_t;
```

Listing E.2: Definition of scanner configuration enumeration and structures.

```
/** @brief This enum defines the GPIO pin values for the Jetson TK1 and TX1. */
typedef enum
{
    // Jetson TK1 pins
    gpio57  =  57,   // J3A1 - Pin 50
    gpio160 = 160,   // J3A2 - Pin 40
    gpio161 = 161,   // J3A2 - Pin 43
    gpio162 = 162,   // J3A2 - Pin 46
    gpio163 = 163,   // J3A2 - Pin 49
    gpio164 = 164,   // J3A2 - Pin 52
    gpio165 = 165,   // J3A2 - Pin 55
    gpio166 = 166,   // J3A2 - Pin 58

    // Jetson TX1 pins
    gpio36 = 36,     // J21 - Pin 32 - Unused - AO_DMIC_IN_CLK
    gpio37 = 37,     // J21 - Pin 16 - Unused - AO_DMIC_IN_DAT
    gpio38 = 38,     // J21 - Pin 13 - Bidir - GPIO20/AUD_INT
    gpio63 = 63,     // J21 - Pin 33 - Bidir - GPIO11_AP_WAKE_BT
    gpio184 = 184,   // J21 - Pin 18 - Input - GPIO16_MDM_WAKE_AP
    gpio186 = 186,   // J21 - Pin 31 - Input - GPIO9_MOTION_INT
    gpio187 = 187,   // J21 - Pin 37 - Output - GPIO8_ALS_PROX_INT
    gpio219 = 219    // J21 - Pin 29 - Output - GPIO19_AUD_RST
} gpio_pin_t;


/**@brief The defined scan modes.
 *
 * @details Up to 2^32 scan modes can be defined. In the description the frame types are abbreviated as following:
 *
 * L: An image containing the laser line.
 * B: An image where the laser line and LED is turned off.
 * C: An illuminated by LEDs.
 */
typedef enum
{
    SCAN_MODE_0 = 0,  /**< L, L, L, ... */
    SCAN_MODE_1 = 1,  /**< L, C, L, C, ... */
    SCAN_MODE_2 = 2,  /**< C, C, C, ... */
    SCAN_MODE_3 = 3,  /**< L, B, L, B, ... */
    SCAN_MODE_4 = 4,  /**< L, B, C, L, B, C, ... */

    SCAN_MODE_FORCE_32_BIT = 0xFFFFFFFF
} scan_mode_t;

/**@brief The image info structure holds information about image configuration. */
typedef struct image_info_s
{
    uint32_t          frame_height;            /**< The height of the image in pixels. */
    uint32_t          num_scan_lines_per_image; /**< The number of scan lines per image. That is the number of columns divided by two. */
} image_info_t;


/** @brief Image processing configuration. */
typedef struct
{
    uint32_t scatter_distance;   /**< Scatter distance is the intensity of the pixel calculated some distance from the height. Defined in pixels. */
    uint32_t threshold_value;    /**< The threshold value sets a lower limit of pixel intensities. All values below are not used for height and reflectance calculations. */
    float    pixel_power;        /**< The exponent used for height calculation. */

    uint32_t num_streams;        /**< The number of CUDA streams used for image processing. */
    uint32_t threads_per_block;  /**< The threads per block is a kernel launch parameter. */
    uint32_t frames_per_message; /**< Frames per message indicates the size of the
        frame buffer and output message. */
} img_processing_config_t;


/** @brief Scanner configuration. */
typedef struct scan_config_s
{
    scan_mode_t            scan_mode;                 /**< The active scan mode used
        to retrieve the image parameters. */
    image_info_t           image_info;                /**< The image size used for
        parameter extraction. */
    gpio_pin_t             scanner_ready_pin;         /**< The GPIO indicating when
        the scanner framework is busy. */
    float                  fps;                       /**< The expected camera frame
        rate. This is used to see if frames are dropped. */
    float                  cycle_time_diff_threshold_ms; /**< A threshold value
        determining the maximum jitter in image reception time.*/

    img_processing_config_t img_processing_config;    /**< The image processing
        configuration used for parameter extraction. */
} scan_config_t;
```

E.2 Image processing

The code snippets presented in this section are extractions from the image processing implementation files. The actual implementation files might differ slightly from those presented here.

Listing E.3: This code snippet demonstrates how a lookup table is placed in texture memory. The values are created in host memory, before they are transferred to the GPU.

```
ret_code_t scan_mode_0_init(scan_config_t * p_config)
{
    float h_texture[LOOKUP_TABLE_LENGTH];
    cudaError_t cuda_error;
    float * d_texture;
    cudaArray *cuArray;


    for (int i = 0; i < LOOKUP_TABLE_LENGTH; i++)
    {
        // implement threshold
        if ( i > (p_config->img_processing_config.threshold_value >> 4))
        {
            h_texture[i] = pow(i,p_config->img_processing_config.pixel_power);

            // values are in the range 0 .. 2^16
            h_texture[i] = (1<<16)*(h_texture[i] /
                pow(LOOKUP_TABLE_LENGTH,p_config->img_processing_config.pixel_power));
        }
    }

    cuda_error = cudaMalloc( (void **) &d_texture, sizeof(h_texture));
    if (cuda_error != cudaSuccess)
    {
        LOG("Failed allocating texture memory\n");
        return RET_ERROR_NO_MEM;
    }

    // Allocate texture array and copy image data
    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0,
        cudaChannelFormatKindFloat);

    cuda_error = cudaMallocArray(&cuArray,
                                 &channelDesc,
                                 LOOKUP_TABLE_LENGTH * sizeof(float),
                                 1);
    if (cuda_error != cudaSuccess)
    {
        LOG("Failed allocating texture array memory\n");
        return RET_ERROR_NO_MEM;
    }

    cuda_error = cudaMemcpyToArray(cuArray,
                                   0,
                                   0,
                                   h_texture,
                                   LOOKUP_TABLE_LENGTH * sizeof(float),
                                   cudaMemcpyHostToDevice);
    if (cuda_error != cudaSuccess)
    {
        LOG("Failed copying texture array memory\n");
        return RET_ERROR_NO_MEM;
    }

    // Bind the array to the texture
    cuda_error = cudaBindTextureToArray(lookup_table, cuArray, channelDesc);
    if (cuda_error != cudaSuccess)
    {
        LOG("Failed binding texture array memory\n");
        return RET_ERROR_NO_MEM;
    }
    return RET_SUCCESS;
}
```

Listing E.4: This code snippet shows how the image processing parameters are retrieved for scan mode 4. The variables `d_frame_buffer` and `d_output_buffer` are inputs to the image processing kernels and contain the required image processing parameters. The variable `blockIdx` contains threads specific parameters.

```
const uint16_t * p_raw16_data = (uint16_t *)d_frame_buffers[0].d_image_buffer;
const uint16_t * p_black_data = (uint16_t *)d_frame_buffers[1].d_image_buffer;
const uint16_t * p_color_data = (uint16_t *)d_frame_buffers[2].d_image_buffer;

const uint32_t NUM_SCAN_LINES_PER_IMAGE = d_frame_buffers[0].s_width /
    (sizeof(uint16_t) * 2);
const uint32_t MAX_HEIGHT           = d_frame_buffers[0].s_height;
const uint32_t SCATTER_DISTANCE     =
    d_output_buffer->p_config->img_processing_config.scatter_distance;
const uint16_t PITCH                = d_frame_buffers[0].d_pitch / sizeof(uint16_t);

const uint32_t image_index = blockIdx.y;
const uint32_t local_scan_line = blockIdx.x*blockDim.x + threadIdx.x;
const uint32_t scan_line = image_index * NUM_SCAN_LINES_PER_IMAGE + local_scan_line;

const uint32_t column = local_scan_line * 2;
const uint32_t image_offset = image_index * PITCH * MAX_HEIGHT;
const uint32_t scan_line_start_index = image_offset + column;

if (local_scan_line >= NUM_SCAN_LINES_PER_IMAGE)
{
    return;
}
```

Listing E.5: Extraction of the height and reflectance properties of an image. `scan_line_index` represents the first index of the column containing red values. `PITCH` is the width of the image. For the case when a black frame is used, line 11 is modified to subtract `p_black_data[input_index]`.

```
 1 float height = 0;
 2 uint32_t reflectance = 0;
 3
 4 // First calculate the height based only on the red pixels
 5 uint32_t i, input_index;
 6
 7 input_index = scan_line_start_index;
 8
 9 for (i = 0; i < MAX_HEIGHT; i += 2) // only use data from each second line
10 {
11     float val = tex1D(lookup_table, p_raw16_data[input_index] >> 4);
12
13     reflectance += val;
14     height     += i * val;
15
16     input_index += 2*PITCH; // jump 2 lines to the next line containing RED.
17 }
18
19 if (reflectance != 0)
20 {
21     height /= reflectance;
22 }
23 else
24 {
25     height = 0;
26 }
27
28 // normalize the height to its range
29 d_output_buffer->p_height_buffer[scan_line] = (height/MAX_HEIGHT)*(1 << 16);
30 d_output_buffer->p_reflectance_buffer[scan_line] = reflectance/(float)MAX_HEIGHT;
```

Listing E.6: Extraction of the intensity and scatter properties of an image.

```
if ( ((uint16_t)height & 1) == 0)
{
    d_output_buffer->p_intensity_buffer[scan_line]
    = p_raw16_data[GET_INDEX(column, (uint16_t)height, image_offset)];
}
else
{
    d_output_buffer->p_intensity_buffer[scan_line]
    = p_raw16_data[GET_INDEX(column, (uint16_t)height - 1, image_offset)] / 2
    + p_raw16_data[GET_INDEX(column, (uint16_t)height + 1, image_offset)] / 2;
}

const uint32_t scatter_height = MIN((uint16_t)height + SCATTER_DISTANCE, MAX_HEIGHT -
    2);

if ( (scatter_height & 1) == 0)
{
    d_output_buffer->p_scatter_buffer[scan_line]
            = p_raw16_data[GET_INDEX(column, scatter_height, image_offset)];
}
else
{
    d_output_buffer->p_scatter_buffer[scan_line]
          = p_raw16_data[GET_INDEX(column, scatter_height - 1, image_offset)] / 2
          + p_raw16_data[GET_INDEX(column, scatter_height + 1, image_offset)] / 2;
}
```

Listing E.7: Extraction of the color an image. `local_scan_line` represents the column index of a the columns containing red values.

```
#define GET_INDEX(column, height, offset) ((offset) + (column) + PITCH*(height))

const uint32_t r_index = scan_line * 3;
const uint32_t g_index = r_index + 1;
const uint32_t b_index = r_index + 2;

if (local_scan_line > 0 && local_scan_line < NUM_SCAN_LINES_PER_IMAGE - 1
           && (uint16_t)height > 0 && (uint16_t)height < MAX_HEIGHT - 1)
{
    const uint32_t first_index = GET_INDEX(column - 1, (uint16_t)height - 1,
        image_offset);

    if ( ((uint16_t)height & 1) == 0)
    {
        d_output_buffer->p_rgb_buffer[r_index]
              = p_color_data[first_index + PITCH + 1];

        d_output_buffer->p_rgb_buffer[g_index]
              = p_color_data[first_index + PITCH] / 4
              + p_color_data[first_index + PITCH + 2] / 4
              + p_color_data[first_index + 1] / 4
              + p_color_data[first_index + 2 * PITCH + 1] / 4;

        d_output_buffer->p_rgb_buffer[b_index]
              = p_color_data[first_index] / 4
              + p_color_data[first_index + 2] / 4
              + p_color_data[first_index + 2 * PITCH] / 4
              + p_color_data[first_index + 2 * PITCH + 2] / 4;
    }
    else
    {
        d_output_buffer->p_rgb_buffer[r_index]
              = p_color_data[first_index + 1] / 2
              + p_color_data[first_index + 2 * PITCH + 1] / 2;

        d_output_buffer->p_rgb_buffer[g_index]
              = p_color_data[first_index + PITCH + 1];
        d_output_buffer->p_rgb_buffer[b_index]
              = p_color_data[first_index + PITCH] / 2
              + p_color_data[first_index + PITCH + 2] / 2;
    }
}
```

E.3 Other snippets

Listing E.8: Calculating the time difference between two images using the embedded time stamp. Line 21 to 26 indicate the bugfix for the occasional negative time difference.

```
 1 static ret_code_t get_diff_to_previous(fc2Image * pImage, float * p_diff_ms)
 2 {
 3     static fc2TimeStamp prev_time_stamp;
 4 
 5     if (m_scanner.first_image_received == false)
 6     {
 7         // Reset prev_time_stamp
 8         prev_time_stamp = fc2GetImageTimeStamp(pImage);
 9 
10         m_scanner.first_image_received = true;
11 
12         *p_diff_ms = m_scanner.cycle_time_ms;
13         return RET_SUCCESS;
14     }
15 
16     fc2TimeStamp    curr_time_stamp = fc2GetImageTimeStamp(pImage);
17 
18     int secondsDiff = curr_time_stamp.cycleSeconds - prev_time_stamp.cycleSeconds;
19     int countDiff  = curr_time_stamp.cycleCount - prev_time_stamp.cycleCount;
20 
21    // START BUGFIX
22     if (secondsDiff < 0)
23     {
24         secondsDiff = 128 + secondsDiff;
25     }
26    // END BUGFIX
27 
28     if (countDiff < 0)
29     {
30         // There are 8000 cycles in a second
31         countDiff = 8000 + countDiff;
32         secondsDiff--;
33     }
34 
35     *p_diff_ms = double(secondsDiff)*1000 + double(countDiff) / 8;
36 
37     memcpy(&prev_time_stamp, &curr_time_stamp, sizeof(fc2TimeStamp));
38 
39     return RET_SUCCESS;
40 }
```

Listing E.9: The TCP server transfers execution to other threads after receiving a chunk of data.

```c
static ret_code_t try_read_length(uint32_t length)
{
    uint32_t bytes_read;
    int temp_bytes_read;

    bytes_read = 0;
    while ( bytes_read < length)
    {
        temp_bytes_read = recv(client_fd,&recv_buffer[bytes_read],length - bytes_read,
            0);
        if (temp_bytes_read <= 0)
        {
            return RET_ERROR_OTHER;
        }

        bytes_read += temp_bytes_read;

        pthread_yield();

    }
    return RET_SUCCESS;
}
```

Listing E.10: This code snippet demonstrates how the application scheduling policy was set to maximum priority.

```c
ret_code_t set_scheduler_options(void)
{
    int err;

    // set scheduler to be RR with high priority
    struct sched_param param;
    param.sched_priority = sched_get_priority_max(SCHED_RR);
    err = sched_setscheduler(0, SCHED_RR, &param);
    if (err != 0)
    {
        perror("Setting scheduler parameters");
        return RET_ERROR_OTHER;
    }

    return RET_SUCCESS;
}
```

Listing E.11: This code snippet shows the implementation of scan mode x used for memory configuration tests. It uses computationally expensive operations.

```
__global__
void scan_mode_x_kernel(frame_buffer_t * d_frame_buffers,
                      scan_output_buffer_t * d_output_buffer)
{
    uint32_t i,j;
    for (i = 0; i < 200; i++)
    {
        for (j = 0; j < 10; j++)
        {
            const uint32_t scan_lines =
                d_output_buffer->p_config->image_info.num_scan_lines_per_image;
            d_output_buffer->p_height_buffer[i % scan_lines] = j*3.41 + 3.4 * i / (j %
                23 + 1);
        }
    }
}
```

Appendix F

Detailed test results

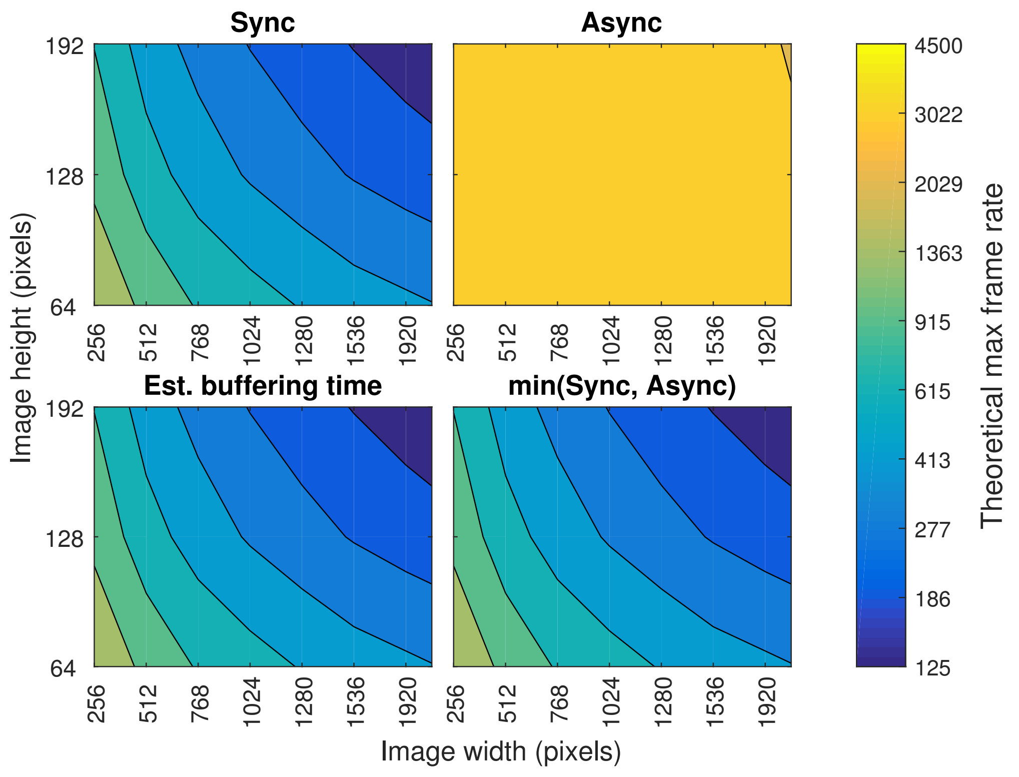
Many tests were performed on the scanner. The result chapter of this thesis is not suited for presenting all these results. This appendix gives the detailed test results, which might be used to obtain deeper understanding or as a reference for future development. The additional test results are only related to line scanner module tests and system functionality tests. Those test results are therefore given in section F.1 and in section F.2.

F.1 Line scanner results

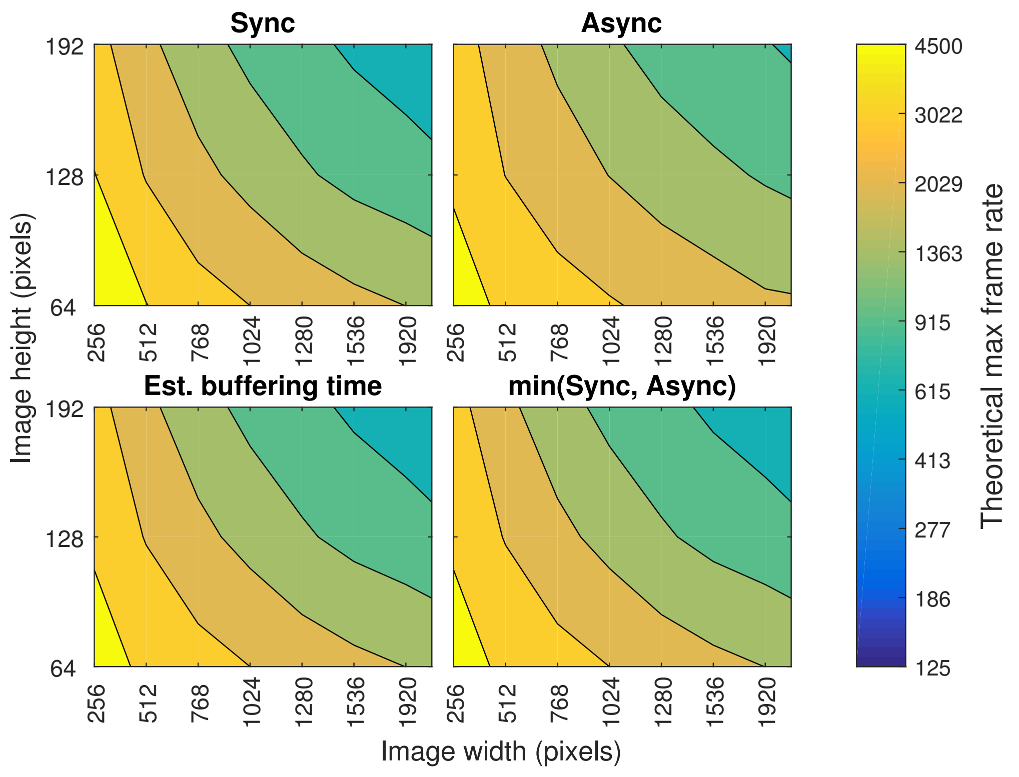
The detailed line scanner results are presented in the same order as done in section 7.1. Therefore, the image buffering framework test results are presented first, and then the scan mode implementation benchmark results.

F.1.1 Detailed scanner framework benchmark results

The detailed framework benchmark results presented in this section include both synchronous, asynchronous, and two estimates of the actual image buffering time. The first estimate subtracts the asynchronous execution speed from the synchronous. The second approach uses the minimum frame rate of the synchronous and asynchronous results. Both estimates give similar results.



(a) TK1 benchmark results.



(b) TX1 benchmark results.

Figure F.1: Detailed framework benchmark results. See subsection F.1.1 for a description of the results.

F.1.2 Detailed scan mode implementation benchmark results

This section first presents a summary of the image processing benchmarks for each of the scan modes for different image buffer sizes. For each image buffer size, the thread block size with the maximum theoretical framerate was chosen. Next the more detailed image processing implementations are shown, where the test results include all image buffer and thread block sizes. The final test results presented here include the line scanner module bottlenecks for some given image sizes.

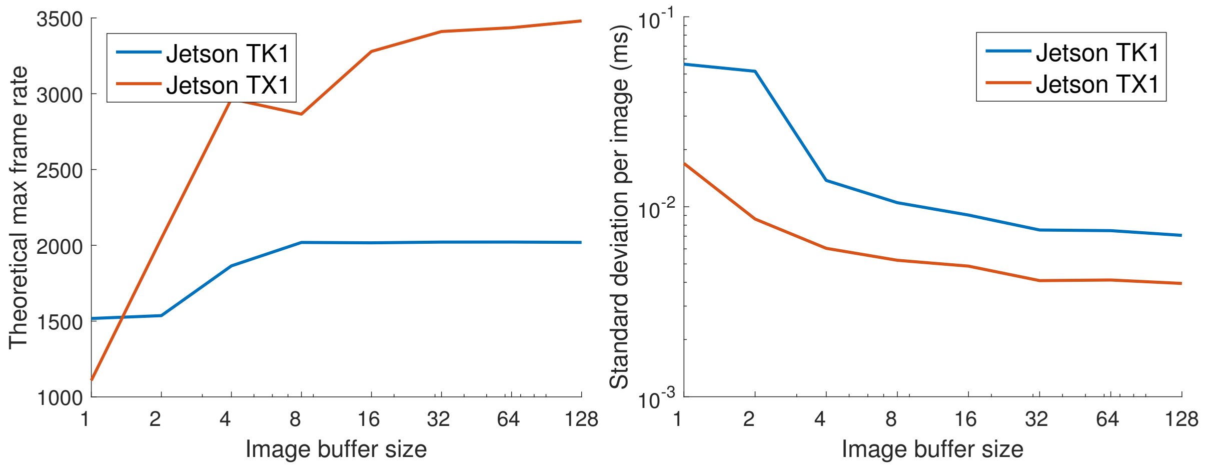


Figure F.2: Theoretical maximum frame rate and standard deviation for scan mode 0.

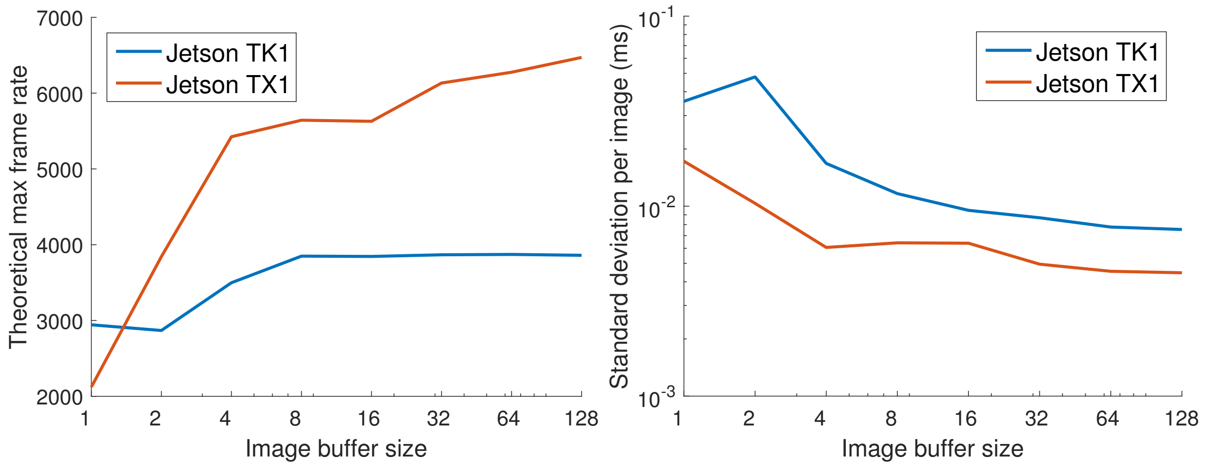


Figure F.3: Theoretical maximum frame rate and standard deviation for scan mode 1.

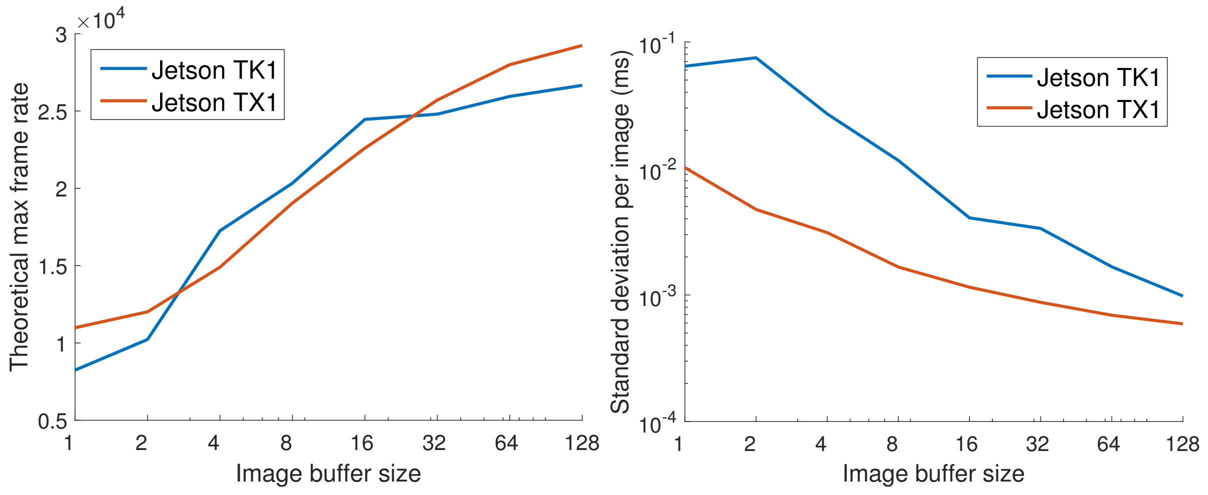


Figure F.4: Theoretical maximum frame rate and standard deviation for scan mode 2.

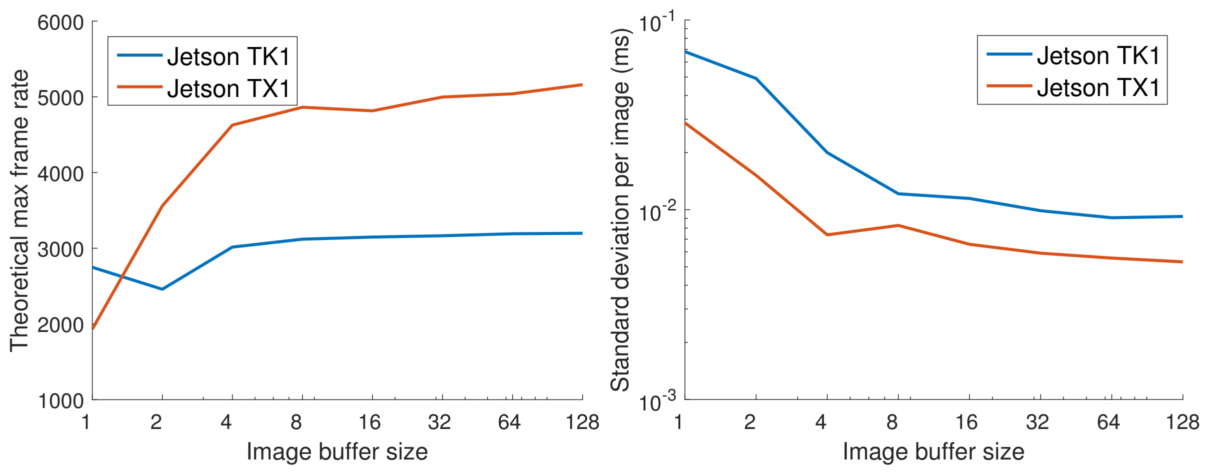


Figure F.5: Theoretical maximum frame rate and standard deviation for scan mode 3.

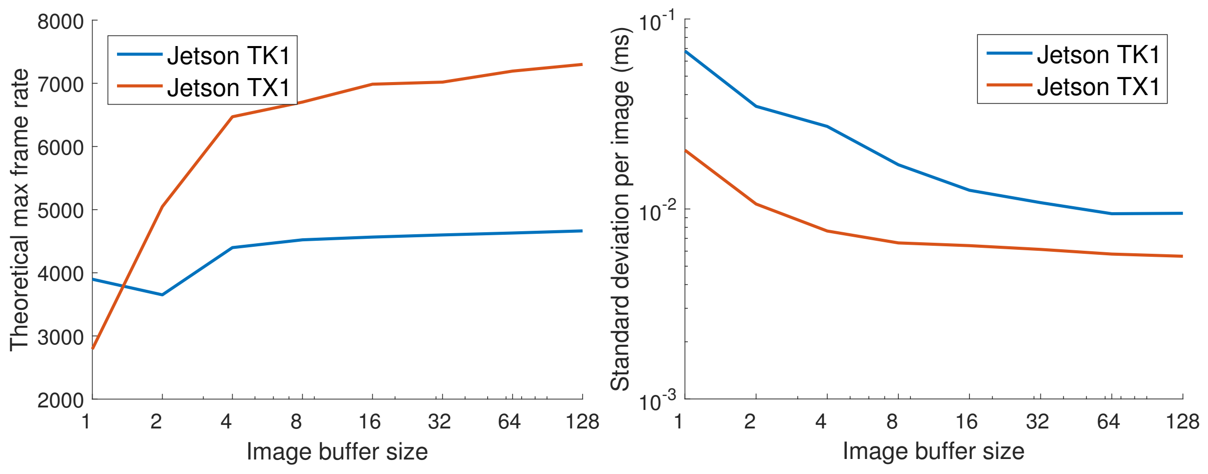


Figure F.6: Theoretical maximum frame rate and standard deviation for scan mode 4.

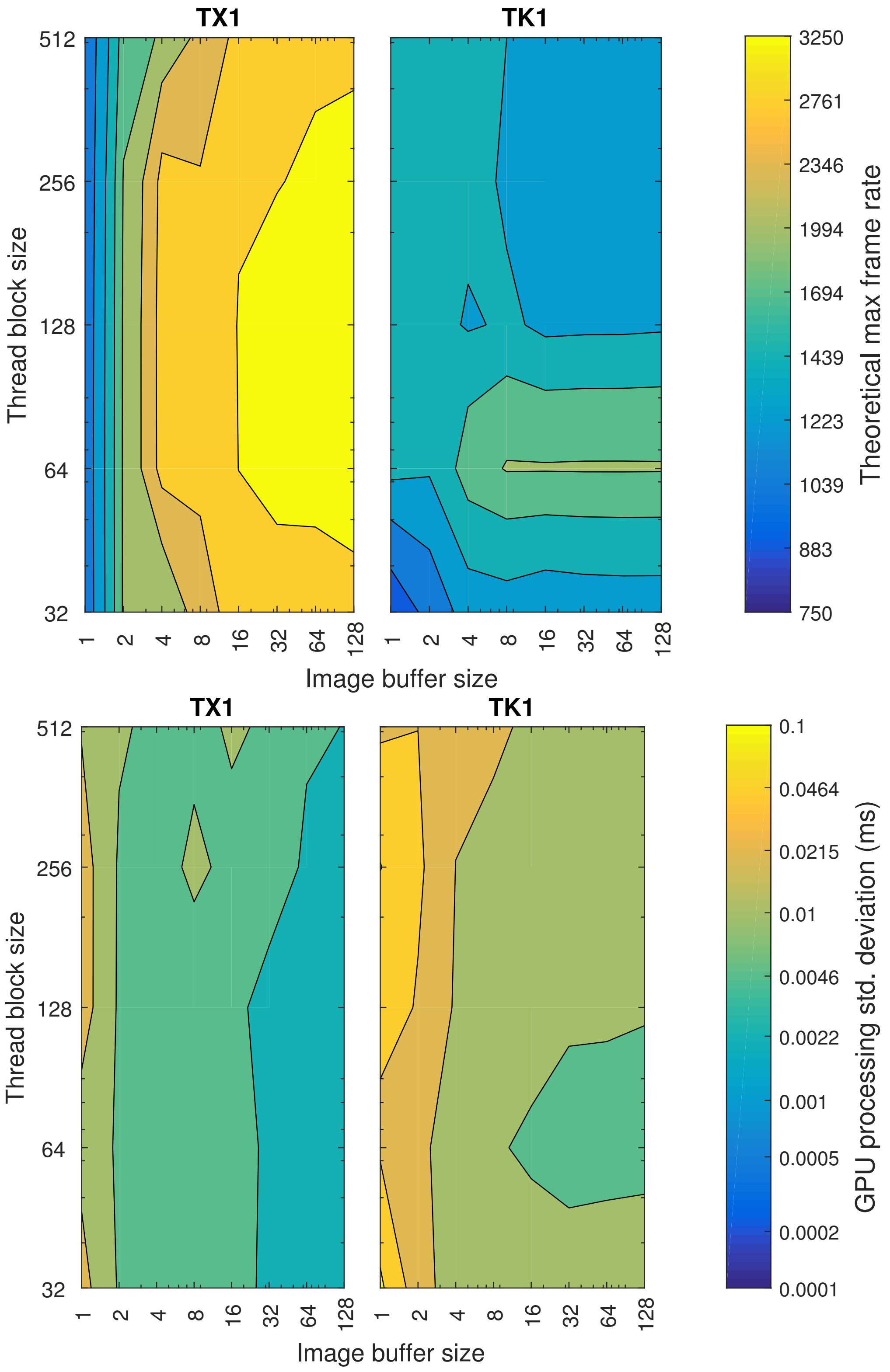


Figure F.7: Scan mode 0 with multiple image buffer and thread block sizes..

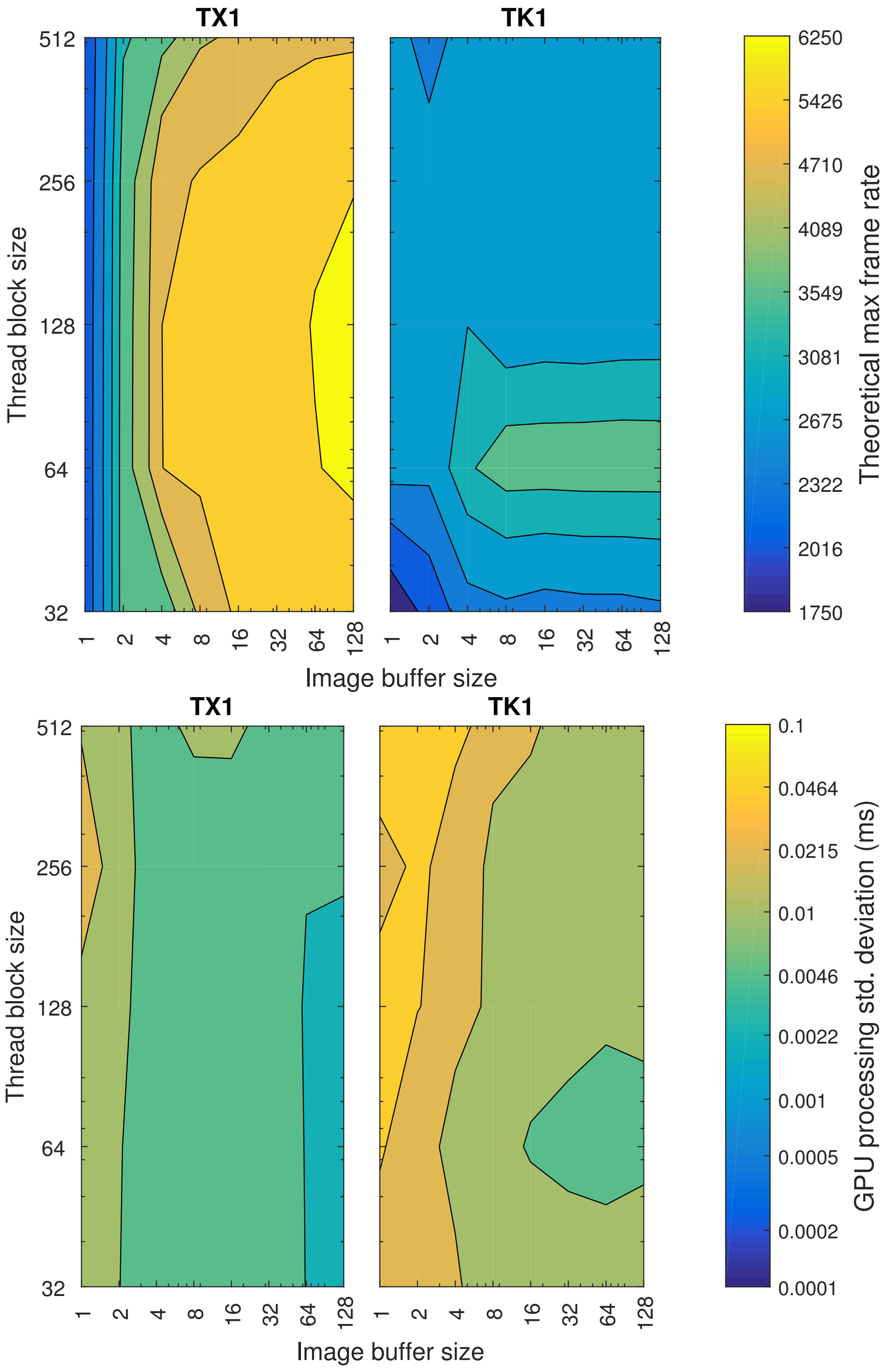


Figure F.8: Scan mode 1 with multiple image buffer and thread block sizes..

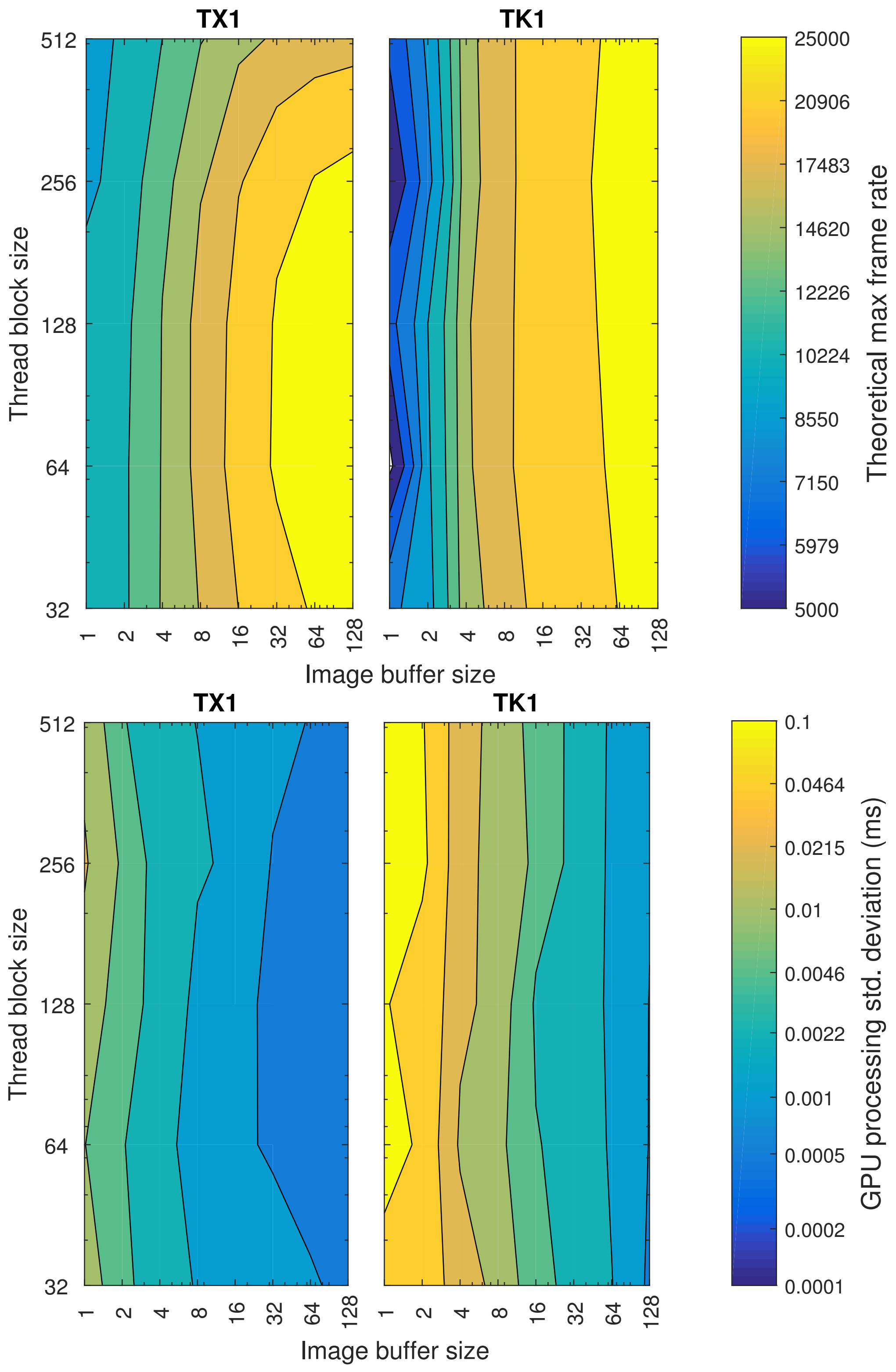


Figure F.9: Scan mode 2 with multiple image buffer and thread block sizes.

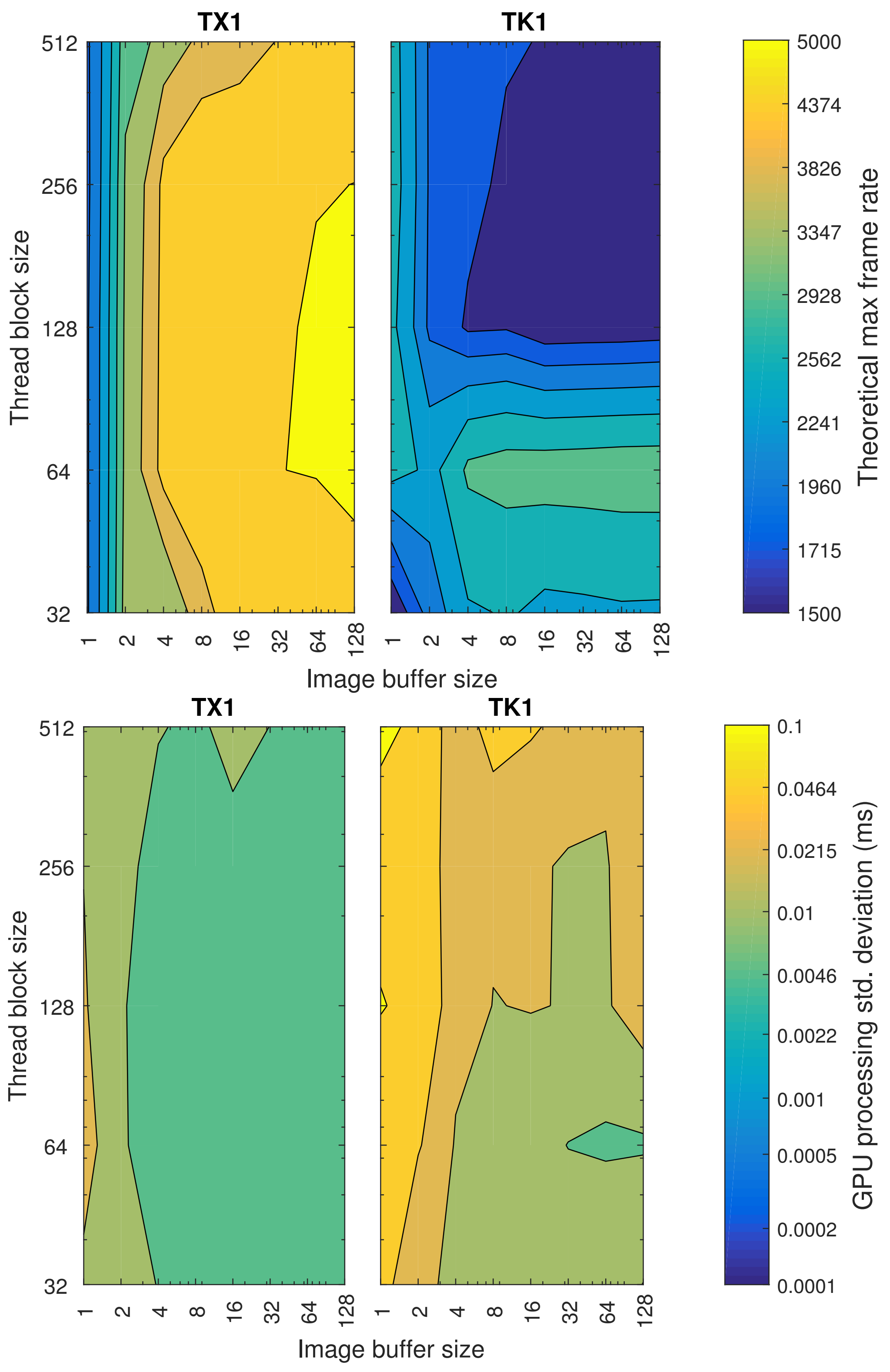


Figure F.10: Scan mode 3 with multiple image buffer and thread block sizes.

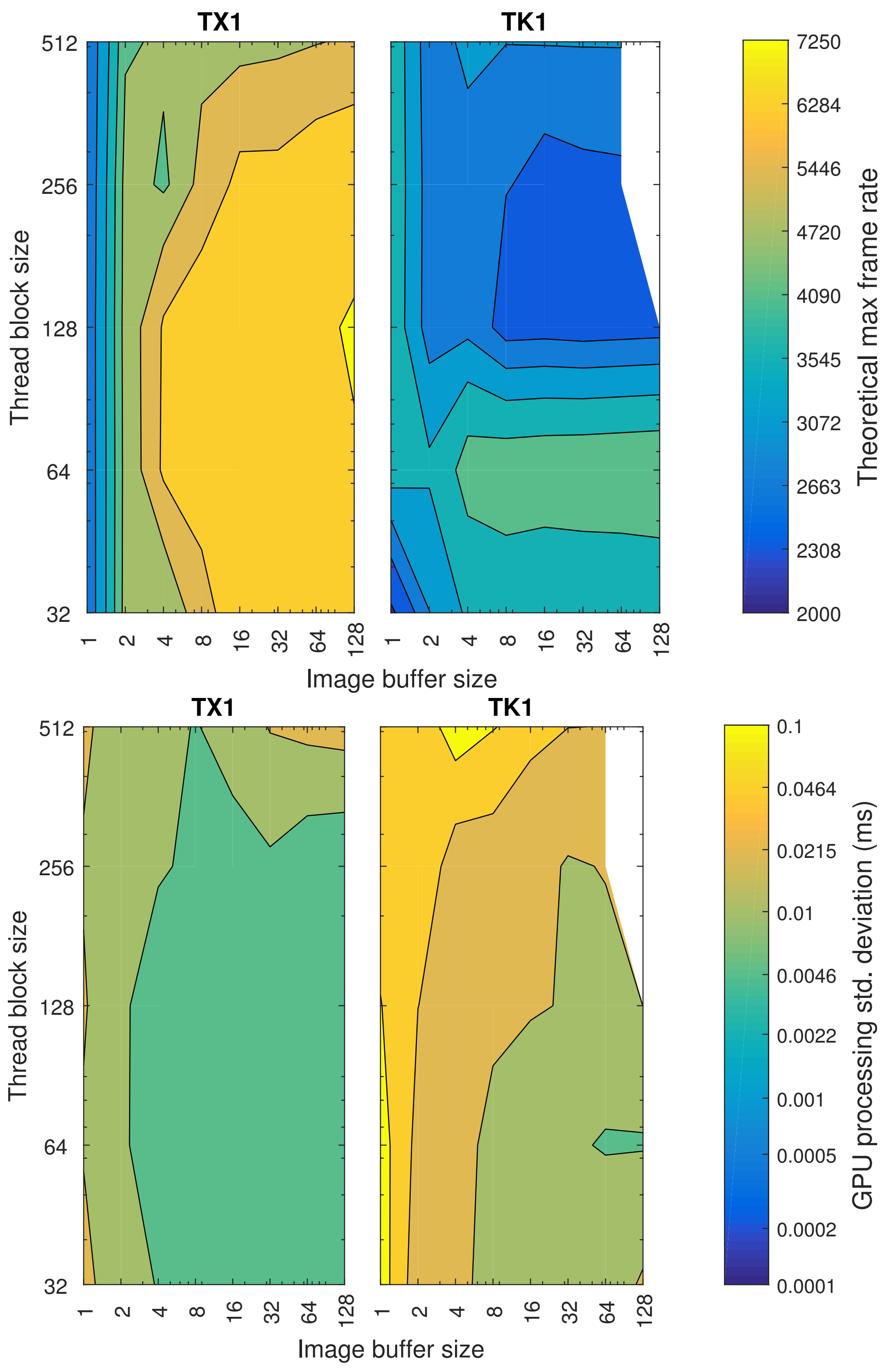
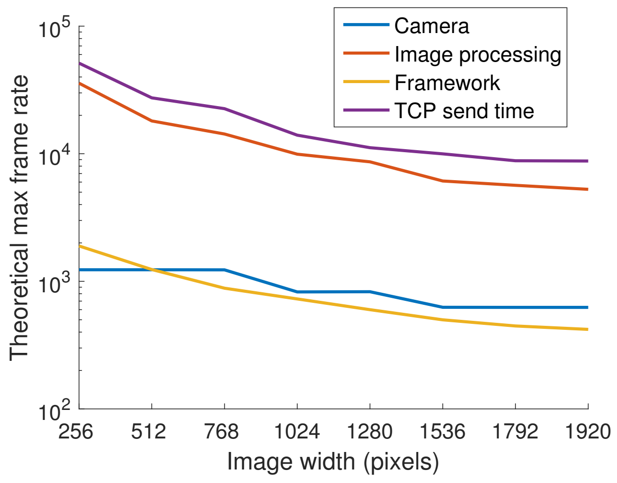
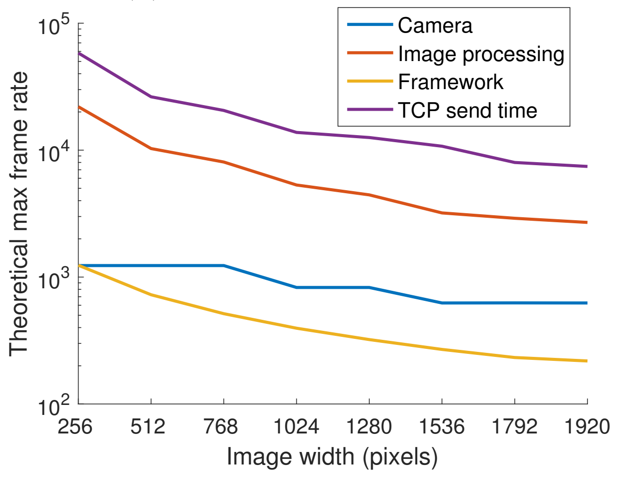


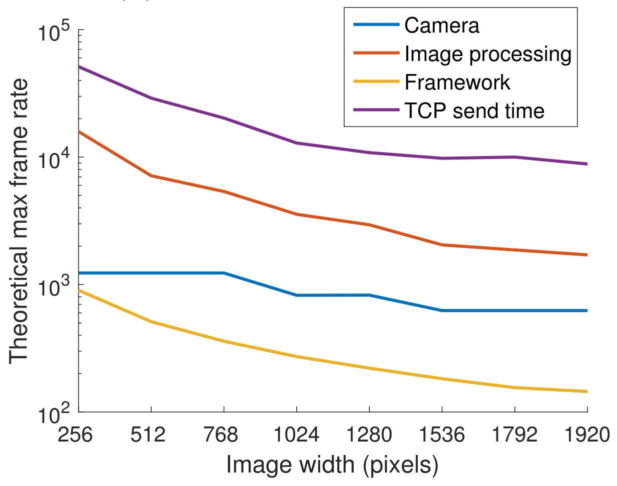
Figure F.11: Scan mode 4 with multiple image buffer and thread block sizes.



(a) Image height is 64 pixels.

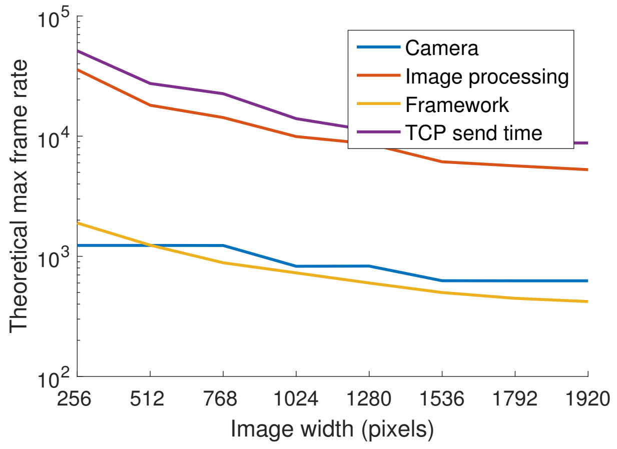


(b) Image height is 128 pixels.

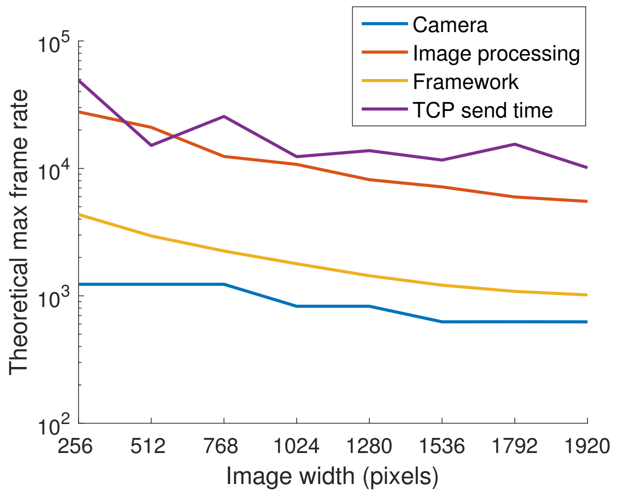


(c) Image height is 192 pixels.

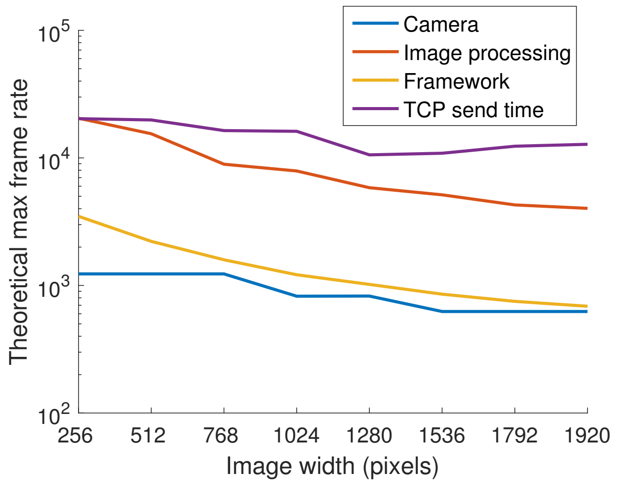
Figure F.12: Plot of frame rate constraints for scan mode 4 on the Jetson TK1.



(a) Image height is 64 pixels.



(b) Image height is 128 pixels.

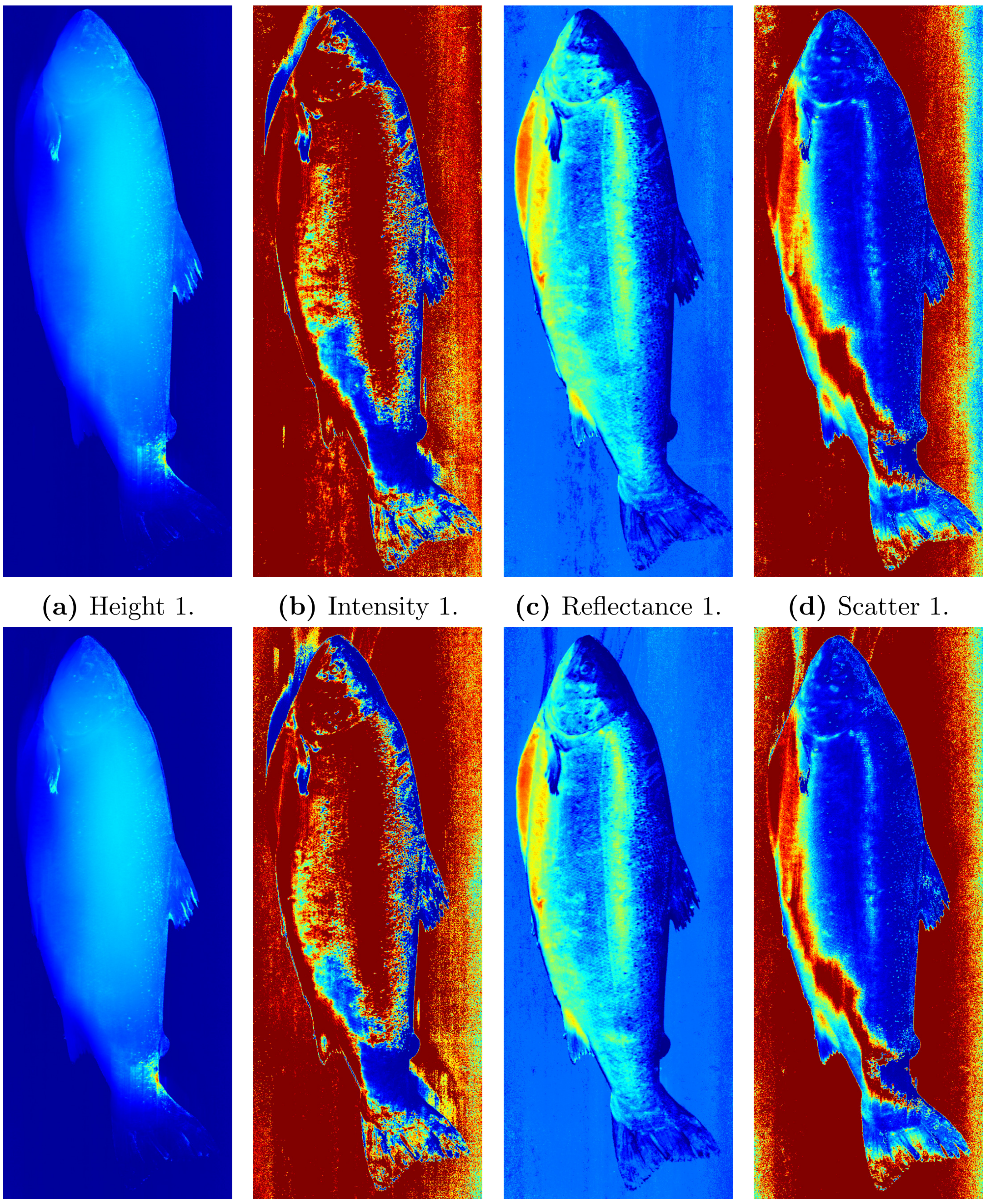


(c) Image height is 192 pixels.

Figure F.13: Plot of frame rate constraints for scan mode 4 on the Jetson TX1.

F.2 System functionality results

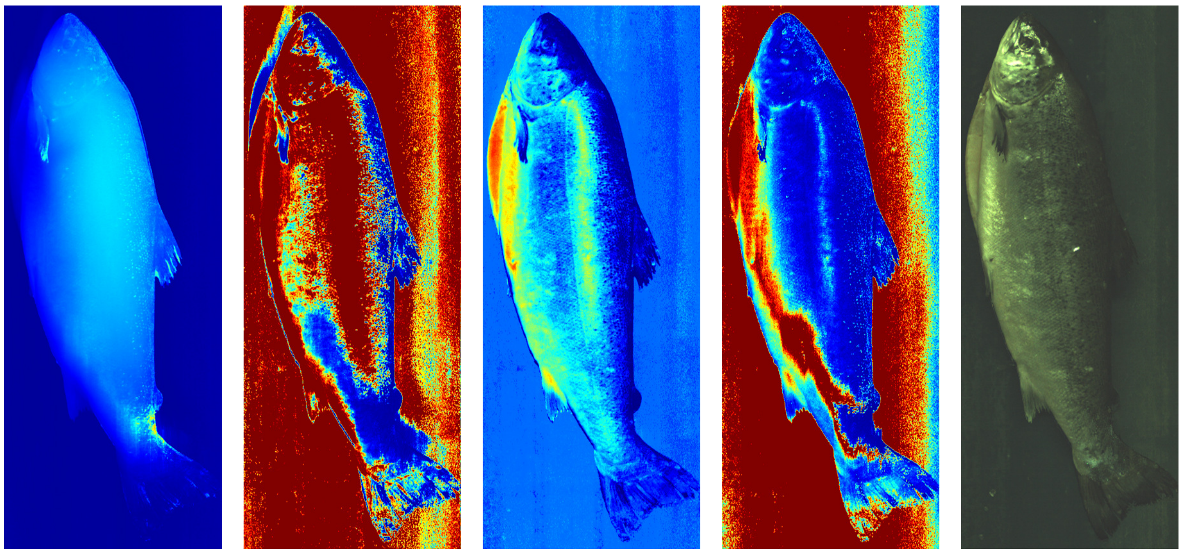
This section present scan results of a salmon. Each scan is performed twice in each mode with the test parameters given in Table 6.4. For the height, intensity, scatter and reflectance, the colors indicate the values: The values increase from blue to green to red. The brightness of the color images is increased to improve the printing quality.



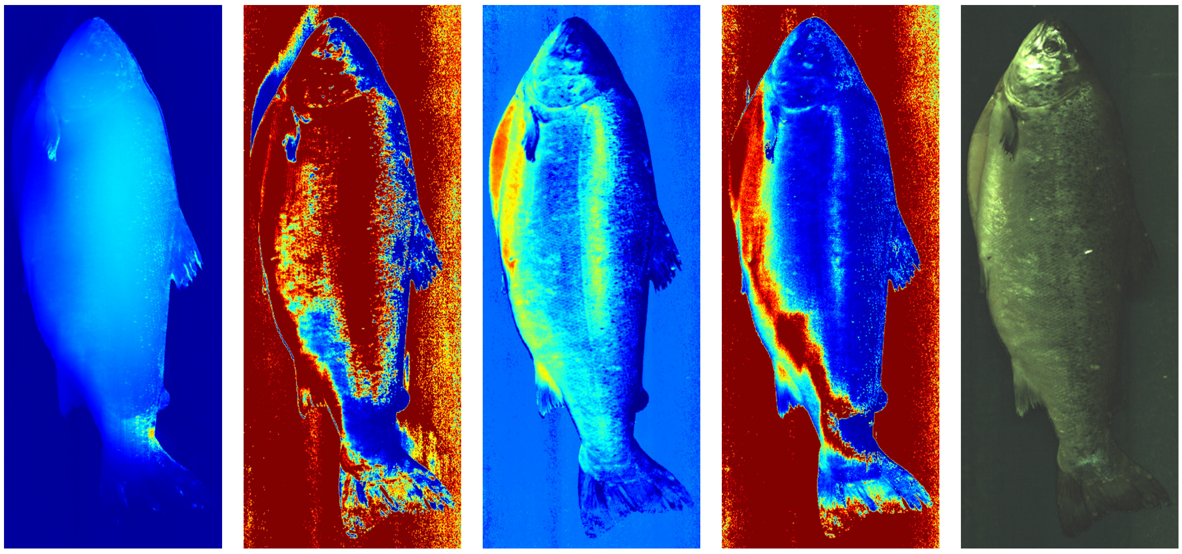
(a) Height 1. **(b)** Intensity 1. **(c)** Reflectance 1. **(d)** Scatter 1.

(e) Height 2. **(f)** Intensity 2. **(g)** Reflectance 2. **(h)** Scatter 2.

Figure F.14: Two different scans of a salmon obtained with scan mode 0.



(a) Height 1. **(b)** Intensity 1. **(c)** Reflectance 1. **(d)** Scatter 1. **(e)** Color image 1.



(f) Height 2. **(g)** Intensity 2. **(h)** Reflectance 2. **(i)** Scatter 2. **(j)** Color image 2.

Figure F.15: Two different scans of a salmon obtained with scan mode 1.

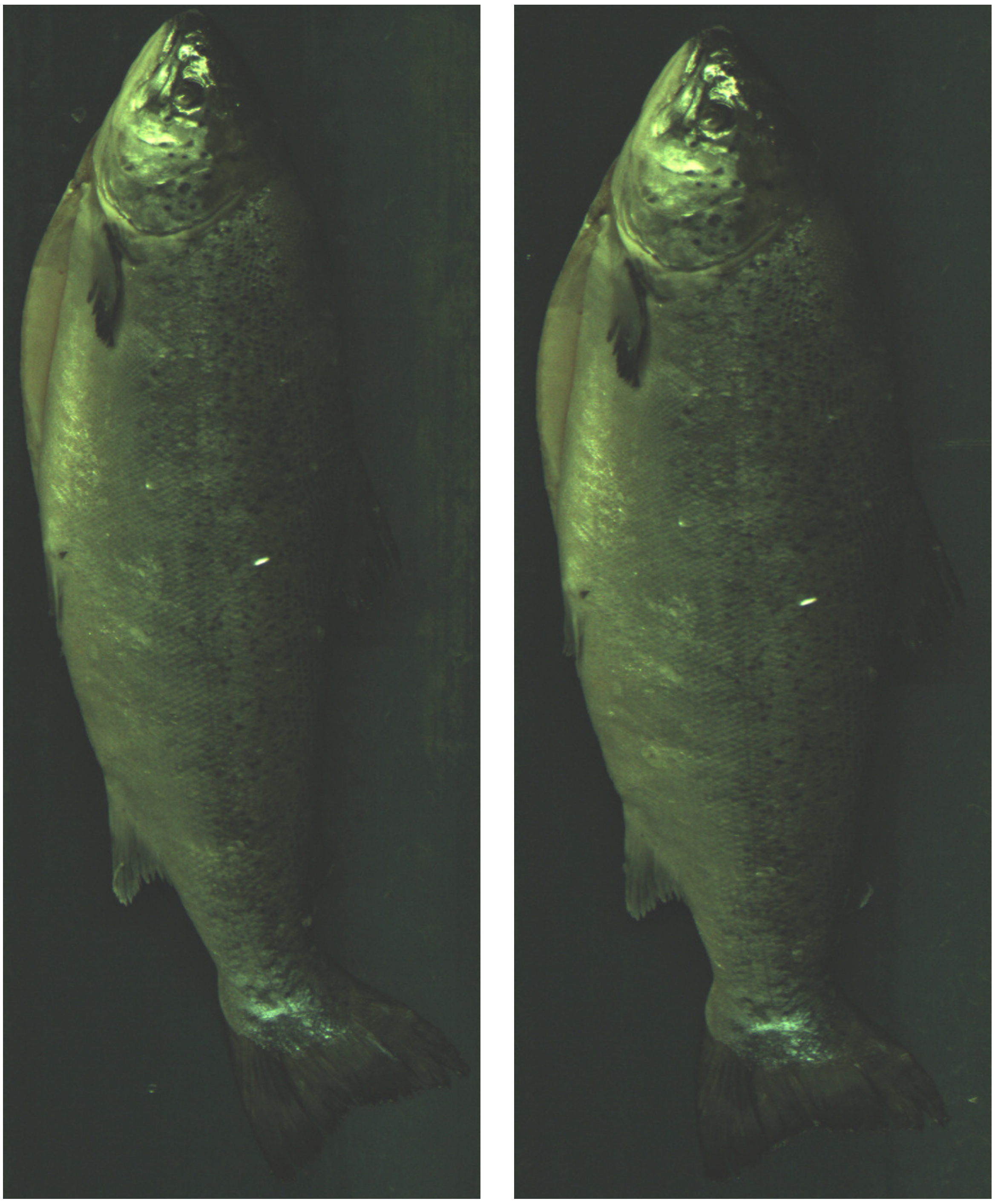
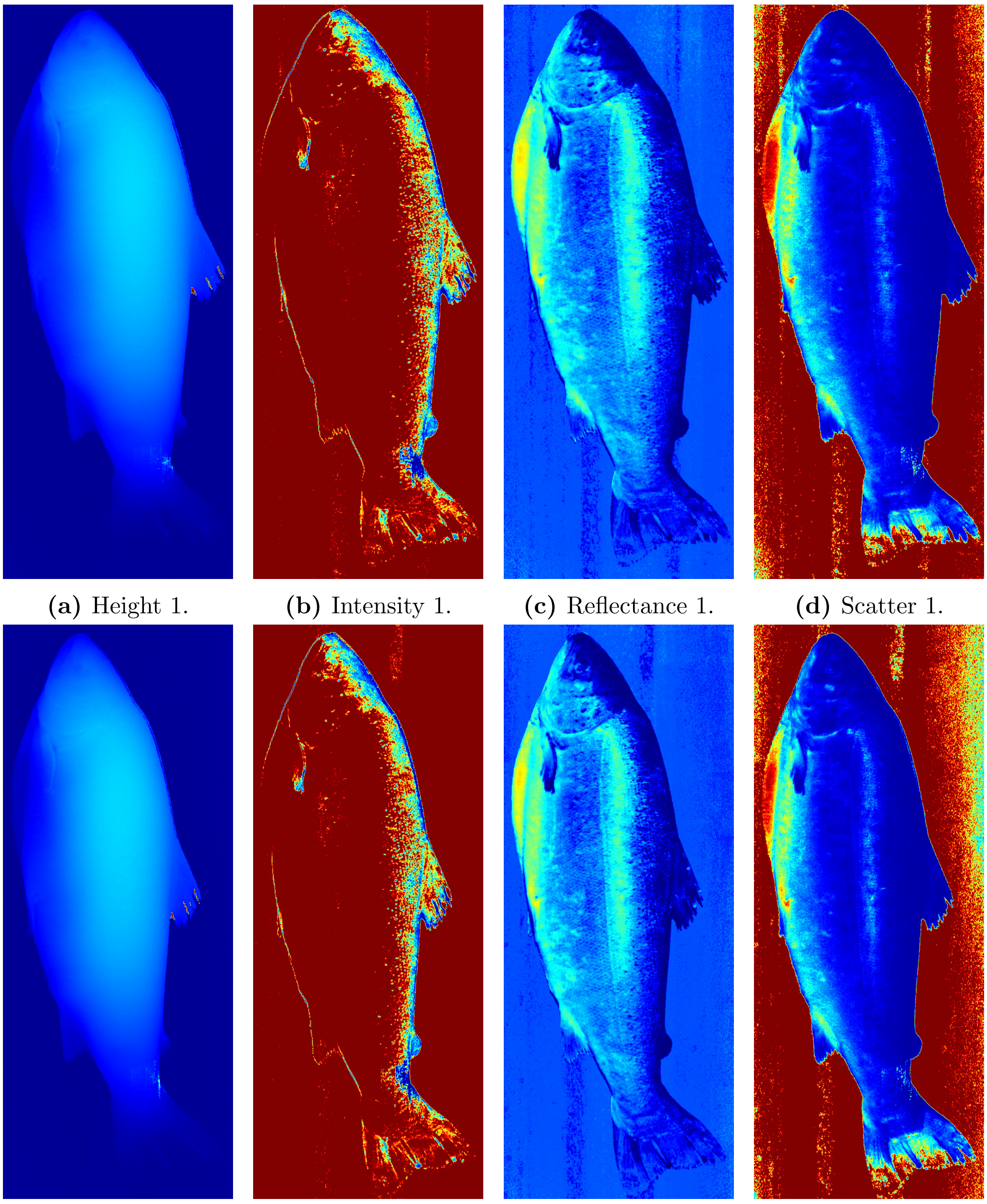

(a) Color image 1. (b) Color image 2.

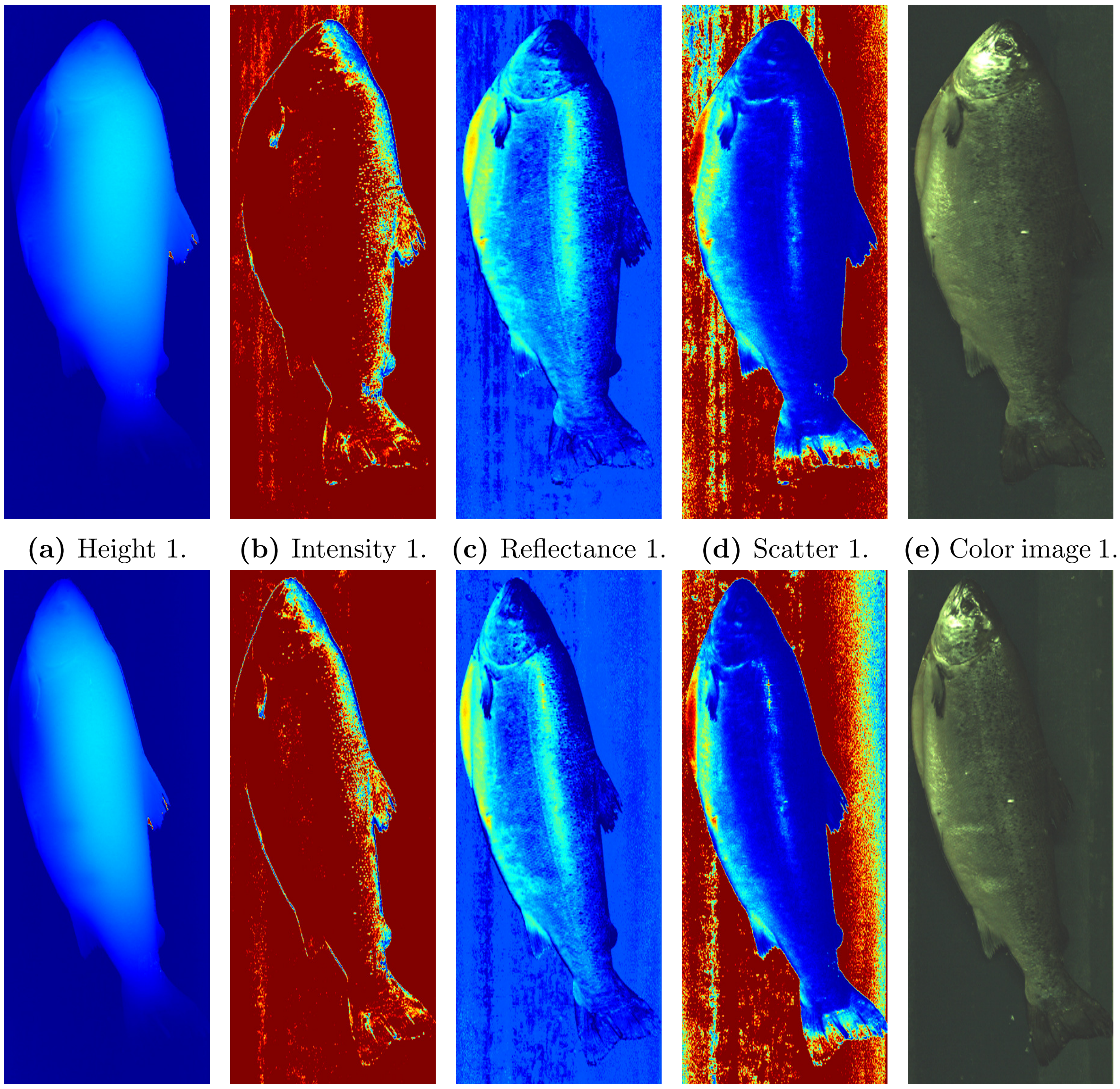
Figure F.16: Two different scans of a salmon obtained with scan mode 2. The brightness is increased to improve printing quality.



(a) Height 1. **(b)** Intensity 1. **(c)** Reflectance 1. **(d)** Scatter 1.

(e) Height 2. **(f)** Intensity 2. **(g)** Reflectance 2. **(h)** Scatter 2.

Figure F.17: Two different scans of a salmon obtained with scan mode 3.



(a) Height 1. **(b)** Intensity 1. **(c)** Reflectance 1. **(d)** Scatter 1. **(e)** Color image 1.

(f) Height 2. **(g)** Intensity 2. **(h)** Reflectance 2. **(i)** Scatter 2. **(j)** Color image 2.

Figure F.18: Two different scans of a salmon obtained with scan mode 4.