Bachelor thesis

Accelerating a Mixed Reality Application on the Edge using a Jetson TX2

Department of Computer and Information Science

Author:
Pablo Oteo de Prado

Supervisors:
Simin Nadjm-Tehrani, LiU
Juan José Costa Prats, UPC

January 2021

Abstract

Mixed reality (MR) applications or Augmented Reality (AR) applications, as their names imply, are meant to unify the virtual world within the real world. This type of application demands high computer capability and also high power demand, which occurs to confront with the devices where they are usually run, the smartphones.

Edge computing brings a solution to it. Offloading the high-demanding parts to an external device seems to solve the issue, but it carries another problem: a high end-to-end latency.

This thesis is focused on reducing the end-to-end latency of an existing MR application. In order to achieve it, the actual edge device, which is a computer, will be swapped to a Jetson TX device. Two main capabilities of the Jetson were used: Dedicated hardware was used to encode and decode the video and also, offloading work from the CPU to the GPU using the Compute Unified Device Architecture (CUDA). These changes lead to reduce the end-to-end latency to half of the initial and also boosted the overall performance of the application. Which almost doubled the initial frames per second.

Resum

Les aplicacions de Realitat Mixta (MR) o Realitat Augmentada (AR), com els seus noms indiquen, tenen com a objectiu unificar el món virtual amb el món real. Aquest tipus d'aplicacions exigeixen una gran capacitat de computació i també una gran demanda d'energia, la qual cosa difereix amb els dispositius on acostumen a funcionar, els smartphones.

El Edge Computing aporta una solució. Traspassant les parts més exigents de l'aplicació a un dispositiu extern sembla resoldre wel problema, però afegeix un de nou: una alta latència d'extrem a extrem.

En aquesta tesi ens centrem a reduir aquesta latència d'extrem a extrem a una aplicació de ja MR existent. Per aconseguir-ho, l'edge device existent, el qual és un ordinador, serà substituït per una Jetson TX. S'utilitzen dues principals característiques dels Jetson per a aconseguir-ho: utilitzarem hardware específic per codificar i descodificar el vídeo i, també, mitigarem la càrrega de la CPU a la GPU mitjnçant l'Arquitectura de Dispositius Unificats de Computació (CUDA). Aquests canvis han permès reduir la latència d'extrem a extrem a la meitat de la inicial i també augmentar el rendiment de l'aplicació. Amb el qual, gairebé s'han duplicat els frames per segon inicials de l'aplicació.

Resumen

Las aplicaciones de Realidad Mixta (MR) o Realidad Aumentada (AR), como sus nombres indican, tienen como objetivo unificar el mundo virtual con el mundo real. Este tipo de aplicaciones exigen una gran capacidad de computación y también una gran demanda de energía, lo cual difiere con los dispositivos donde suelen funcionar, los smartphones.

El Edge Computing aporta una solución a ello. Traspasar las partes más exigentes de la aplicación a un dispositivo externo parece resolver este problema, pero añade otro: una alta latencia de extremo a extremo.

En esta tesis nos centramos en reducir la latencia de extremo a extremo a una aplicació de MR existente. Para conseguirlo, el edge device existente, el cual es un ordenador, será sustituido por una Jetson TX. Se utilizan dos principales características de los Jetson para conseguirlo: utilizaremos hardware específico para codificar y descodificar el vídeo y, también, mitigar la Carga de la CPU a la GPU a través la Arquitectura de Dispositivos Unificados de Computación (CUDA). Estos cambios han permitido reducir la latencia de extremo a extremo a la mitad de la inicial y también aumentar el rendimiento de la aplicación. Lo cual casi duplicó los frames por segundo iniciales de la aplicación.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

The popularity of Mixed reality (MR) or Augmented Reality (AR) applications is increasing over the years. This type of application merges the reality within the virtual world, as an example, we could refer to all the applications within the Google AR and VR projects [1].

As Lindqvist commented in his thesis [2] "Google's ARCore and Apple's ARKit only support a limited number of devices in their respective mobile eco-systems". These limitations are caused by the high capacity and power needed to run this type of application, making the smartphones running hot and out of battery really fast. As a solution to this problem, Lindqvist developed an Edge Computing application, in this software he offloads all high computations to an external device. This was a good solution to the problem, but it caused the following issue: a high end-to-end latency.

The Jetson TX is an embedded low-power system with a Compute Unified Device Architecture (CUDA) capable GPU. Running the Jetson TX as an Edge Device over a computer adds more mobility and flexibility to an Edge Device. Replacing the edge device to a Jetson TX was achieved by Eriksson and Akouri on their thesis [3]. Because of a lower CPU capacity from the Jetson TX compared to the original computer, the end-to-end latency got doubled and the fps decayed to less than half of the originals ones, which were running in the computer. Eriksson and Akouri tried to reduce this latency by using CUDA on the Jetson TX, but it was not successful. As a result, their latency increased over a performance upgrade, the latency got doubled and tripled from the initial one in the Jetson TX in order to achieve an fps increase on the application.

Accelerating the performance of the application running on the Jetson TX to reduce the end-to-end latency would make it viable to run the Jetson TX as an Edge Device.

1.2 Aim

The main goal of this thesis is to reduce the end-to-end latency in the application developed by Lindqvist [2], the Mixed Reality Linköping Edge Offloading (MR Leo) running in the Jetson TX2.

Only the server, in other words, the Edge software, is going to be modified, neither the Android application nor the transport protocol is going to be studied or discussed. To achieve it, it will be needed to learn how the MR Leo application works, how is the application behaving internally,

which are the most time-consuming parts and how can we accelerate them. To sum up, we need to adapt the MR Leo software running in Jetson TX to be fast enough so it is viable as an Edge Device.

1.3 Problem definition

This thesis aims to answer the following points:

- How does the MR Leo software works internally? In short, how is the video getting received, processed, and transmitted back?
- What are the most time-consuming parts of the code? Which ones are up to be accelerated using the Jetson TX device?
- What are the effects of the acceleration? Testing and comparing.

1.4 Delimitations

This thesis does not aim to study or discuss any other characteristic of MR Leo apart from the edge software. Neither the Android application running on the smartphone, the transport protocol used, or the encoding type and bit-rate are going to be studied or discussed. The configuration used for this thesis will be a video encoding/decoding H.264 with a bit-rate of 2000 Kbits/s sent via TCP.

The impact of these configurations in the image quality and the end-to-end delay has been already discussed in previous works as the Lindqvist thesis "Edge Computing for Mixed Reality" [2], the Elgh and Thor thesis "Speeding up a mixed reality application: A study of two encoding algorithms" [4] and the Performance Study of Mixed Reality for Edge Computing paper [5].

1.5 Approach

The approach taken to develop the thesis is the following:

1. Check and study the background and technology used in the MR Leo software.
2. Study the code behind MR Leo, how are the images getting received, processed, and sent back.
3. Discover the critical points of the process. Which are the most timing and capacity demanding parts of the code.
4. Accelerate the code.
5. Test the upgrades, gather the statistics and get conclusions.

1.6 Thesis Outline

The thesis structure is described next:

- Chapter 1 introduces the thesis.
- Chapter 2 describes the main concepts of the thesis.
- Chapter 3 describes the methodology followed.
- Chapter 4 exposes all the results.
- Chapter 5 presents a budget of the thesis.
- Chapter 6 expose the conclusions of the thesis and future work.

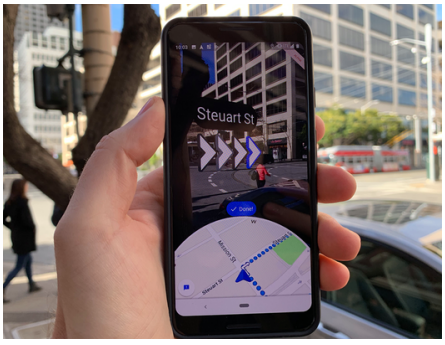
2. Background

This chapter describes the main theoretical concepts used in this thesis:

2.1 Mixed reality

Mixed reality (MR), also called Augmented Reality (AR) is the merging of the real and virtual world. Physical and virtual objects are unified and interact in real-time. This type of applications are getting more and more popular over the years and they are starting to be used in different fields, such as the National Football League (NFL) on TV or Google Maps in smartphone applications.

(a) Augmented Reality in the NFL TV.



(b) Augmented Reality in Google Maps.

Figure 2.1: Augmented Reality applications examples.

2.2 Parallel Computing

The Central Processing Unit (CPU) and Graphics Processing Unit (GPU) have different strengths and weaknesses. The CPU is faster and is better at performing fewer sequential operations fast e.g. workloads like handling the operating system where low latency is preferred. The GPU however is better at computing large amounts of data at the same time [3].

The main objective of using Parallel Computing is to achieve the called Heterogeneous Computing. The key point is that the CPU is good for a certain fraction of code that is latency bound, while

GPU is good at running the Single Instruction Multiple Data (SIMD) part of the code in parallel. It is required that both of the processors, when used optimally, give maximum benefit in terms of performance. This approach of essentially offloading certain types of operations from the processor onto a GPU is called heterogeneous computing [6].

2.3 Point Cloud

Reconstructing a surface out of a three-dimensional set of points, which is obtained by sampling an object's boundary, is done by generating an arbitrary triangular mesh [7].

The purpose of a cloud point is to create a virtual representation of the environment recorded. In MR Leo, the cloud point is generated in the Edge by the library ORB-SLAM2 [8]. Which will be substituted with the library ORB-SLAM2 CUDA [9] to achieve parallel computing and accelerate the image processing. The ORB SLAM will be also responsible for, using the Point Cloud information, generate 3D objects to be shown in the AR application.

2.4 ORB-SLAM2

Simultaneous localization and mapping (SLAM) main goal is to create and update a map of the sensor device's surroundings and tracking the sensor's position relative to this map when it moves around. The resulting three-dimensional map is often modeled as a three-dimensional point cloud. SLAM is often a cornerstone of mixed reality, as to be able to insert content into an environment one has to both create a map of the surroundings to see where these new attributes should be placed, as well as localize oneself within this map [2].

ORB-SLAM2 is a real-time SLAM library developed by Raul Mur-Artal and Juan D. Tardos at the University of Zaragoza. It works for Monocular, Stereo, and RGB-D cameras that compute the camera trajectory and a sparse 3D reconstruction. It can detect loops and re-localize the camera in real-time [8].

2.4.1 ORB-SLAM2 CUDA

The ORB-SLAM2 CUDA is a modified version of ORB-SLAM2 with GPU enhancement. As already commented, this library will substitute the old ORB-SLAM be used to boost the performance of MR Leo.

2.5 Gstreamer

GStreamer is a library for constructing graphs of media-handling components. The applications it supports range from simple playback, audio/video streaming to complex audio and video processing [10].

GStreamer is based on a pipeline multimedia framework. It makes it possible to link media processing systems to complete complex workflows. For example, it can be used to build a system that reads files in one format, processes them, and exports them in another. In MR Leo, the Gstreamer library is used to send, receive, encode and decode the video.

2.6 Pangolin and OpenCV

Pangolin is a lightweight portable rapid development library for managing OpenGL display/interaction and abstracting video input [11].

OpenCV (Open Source Computer Vision Library) is a library of programming functions mainly aimed at real-time computer vision [12].

Both libraries will be used in MR Leo to define and process all the video images. OpenCV is mainly used by the ORB-SLAM2 to generate the Cloud Points and the 3d Objects while the Pangolin library will be used to treat and display the images.

2.7 Related work

Multiple studies have been done about reducing the end-to-end delay in an Edge Computing application. Starting with Lindqvist in his thesis Edge Computing for Mixed Reality [2], where he studied how the transport protocol and the encoding bit-rate affected in the image processing times, fps, and, in general, the performance of the software was affected. Also, Jesper Elgh and Ludvig Thor in their thesis Speeding up a mixed reality application: A study of two algorithms, different types of encoding were tested.

On the other hand, different research has been done about using a Jetson TX module as a tool for running real-time software and ORB SLAMS. Starting with An Evaluation of the NVIDIA TX1 for Supporting Real-time Computer-Vision Workloads [13], in CUDA-Accelerated ORB-SLAM for UAVs [14], and finally Improving Performance of a Mixed Reality Application on the Edge with Hardware Acceleration [9].

3. Method

In this chapter, the method used to develop the thesis, the experiment methodology, and the environment is presented. First of all, an explanation of MR Leo and its code will be exposed. Explaining how it behaves internally and which are the most time-consuming parts, the ones to be accelerated. Then, the main changes done in the code. Finally, the metrics for the performance tests will be described.

3.1 Jetson TX2

The Jetson TX2 will replace the computer as the Edge device on this project. The TX2 model is, among the Jetson TX devices, the fastest, most power-efficient embedded AI computing device. This 7.5-watt supercomputer on a module brings true AI computing at the edge. It's built around an NVIDIA Pascal™-family GPU [15]. In short, it is a low-power consumption embedded system designed to work as an Edge device.

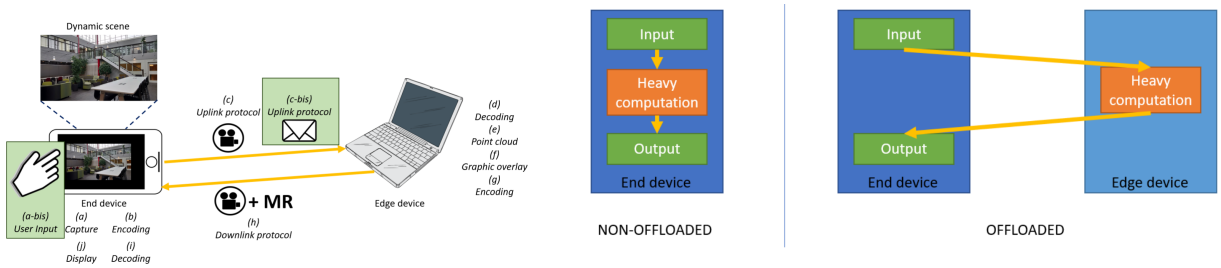
All the specifications of it are shown in the Appendix.

Figure 3.1: The Jetson TX2.

3.2 MR Leo

The Mixed Reality Linköping Edge Offloading or MR Leo is a prototype software developed by Lindqvist on his thesis “Edge Computing for Mixed Reality" [2]. It is based on a server/client model.

The client, a smartphone, would record a real-time video using the camera, then compress it and send it via WIFI to the server. Then, the server would process the video, create the MR environment and send it back to the client. Following this structure, all the high computation parts of the MR application would be offloaded to the edge.



(a) MR Leo software representation.

(b) Edge Computing model.

Figure 3.2: MR Leo Edge Computing representation [5]

3.2.1 Edge Computing

Cloud computing is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user [16].

Edge computing equals cloud computing but at the edge of the network, being just at one or two hops of the user. This way, a reliable channel and a lower latency are achieved in the communication between the server and the client.

3.2.2 MR Leo Installation on the Jetson TX2

Some details were to be looked into before being able to run MR Leo in the Jetson TX2 device. Nevertheless, this work was already done by Eriksson and Akouri in their thesis [9]. The full steps to reproduce the installation are described, both for the client and the server.

3.2.3 Code Analysis

A deep analysis of the code in MR Leo is presented. The software runs using Qt, which uses mostly classes and threads. The different classes and threads will be presented and, with it, the way that

the video images are getting processed.

We can divide the code into 2 main parts. The first 4 classes presented manage the communication link, the definition of the user interface, and the main threads running. Secondly, the next five classes that manage the video processing:

- **Main.cpp:** In this class, the user interface is defined.

  This defines a simple executable that can be run from the Linux terminal ./MR-Leo-server. The different options to run can be shown executing ./MR-Leo-server –help.

  Finally, it starts the Mrserver class using the configuration defined in the executable call.

- **Global.h:** Defines global variables, such as the image frame to be processed and the timers used to run the performance tests.

- **Mrserver.cpp:** Define and initialize all the remaining classes and threads. Also, here it is defined all the thread dependencies when they are stared and stopped.

- **Tcpconnection.cpp:** Defines the TCP connection with the smartphone. It looks for a port for the communication and waits for it to connect. Finally, the end of communication to stop the threads.

- **Videoreceiver.cpp:** This class, as the name implies, manages the reception of the video.

  Using Gstreamer the video is:

  1. Demultiplexed.
  2. Parsed.
  3. Decoded.
  4. Video converted from I420 format to RGB, because I420 is not supported by the ORB-SLAM2.

  Then, each time an image frame is ready to be processed a callback to the Imageprocesser thread is called.

- **Imageprocesser.cpp:** The frame is converted to an OpenCV format, which is used by the ORB-SLAM2 to process the image. After that, it uses the Orbslamprocesser class to process. When the cloud point or the 3d object is created the ViewerAR is called where the images get ready to be sent back to the smartphone.

- **Orbslamprocesser.cpp:** In this class different functions are created where the ORB-SLAM2 library is used to process an image. The cloud point and the 3d objects are calculated and passed to the ViewerAR.

- **ViewerAR.cpp:** The library Pangolin is used to get the images ready to be sent. The Pangolin library is used to fuse the cloud point and the 3d object, which was previously calculated, with the image frames. Finally, the video transmitter is called to send them.
- **Videotransmitter.cpp:** This class manages the transmission of the video.

  Using Gstreamer the video is:

  1. Video converted from RGB format to I420.
  2. Encoded.
  3. Multiplexed.

  And finally, it is sent via TCP.

3.2.4 Time Analysis

To find the most demanding parts of the code, after analyzing the code flow and how the frames were getting received, processed, and sent back, some timers were set up.

The expected results were to find that the image processing was the most time-consuming part of the code but the results were a little bit different than expected. While the image processing took, on average, 90 ms to complete, the video receiving and video transmitting took a long time, around 275 to 325 ms. This was caused by the encoding, decoding, and video conversions. The following graph shows the graphic representation of the image flow over the code:

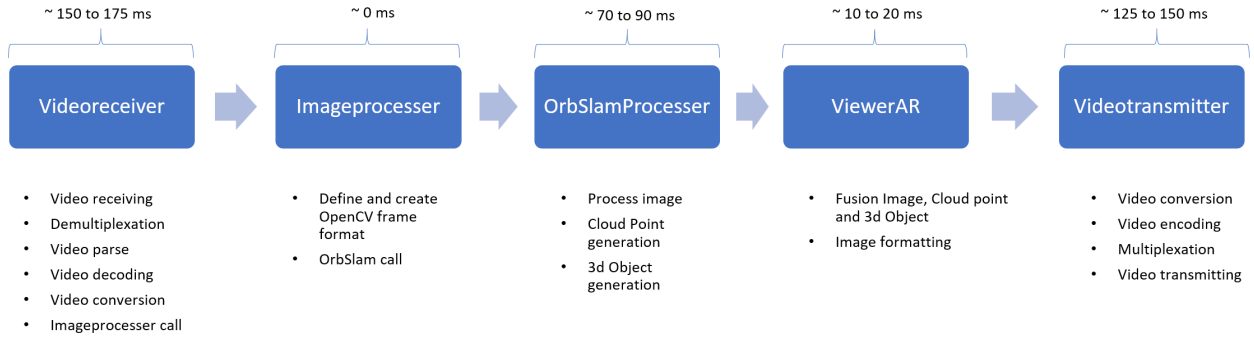


Figure 3.3: Graphical representation of the code.

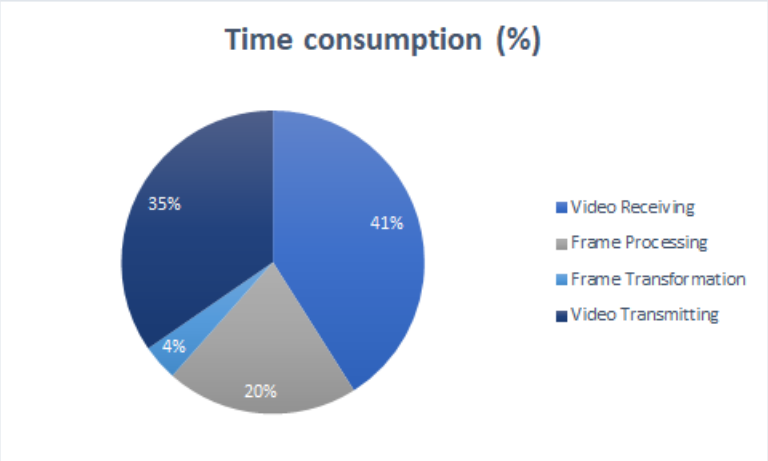


Figure 3.4: Graphical representation of the time consumption (%).

3.3 Accelerating the MR Leo

After investigating the MR Leo code and we can compare its time performance within the software running in the computer and the one in the Jetson TX2 we conclude that the most critical parts of the code are the video receiving and transmitting taking approximately 150 ms each, specifically the encoding and decoding, followed by the ORB SLAM process taking around 90 ms. A graphical representation is shown in the table 3.1.

Looking into the way the images are getting processed, we can also notice that they are getting processed one by one, and, when the frames arrive faster than they are getting processed, the frame is drooped or it has to wait until the last one is processed.

Taking all of that into account, two main changes were done in the code. Firstly, the Jetson TX2 has specific hardware to encode and decode video in an H264 format. Secondly, the library used to process the frames, the ORB-SLAM2, was successfully changed to the ORB-SLAM2 CUDA version, developed by Thien Nguyen [9].

3.3.1 Encode/decode Acceleration

Using the Gstreamer Debugging Tools [17], an exact description of the receiving and transmitting process was obtained. Using that, multiple configurations were tested by following the Accelerated Gstreamer User Guide [18]. Finally, the receiving and transmitting pipelines were changed to use the specific hardware for encoding and decoding.

3.3.2 Gstreamer Pipelines

In this section, a further explanation of the Gstreamer pipelines is presented.

In short, in Gstreamer, a pipeline could be defined as a step. In our case, we need to receive the video via TCP, demultiplex it, parse it, decode it and finally, video converts it to RGB. To do that, each step is called via a pipeline. All pipelines are delimited by a "!" and depending on the type, multiple configurations can be used.

An easier way to understand this is by creating a graphical representation. It is possible to obtain it in .dot format for the pipeline, which gives full information about each pipeline. The information about how to get the .dot filed is explained in the Gstreamer Debugging Tools [17]. All the pipelines used in this thesis are included in the Appendix.

3.3.3 Studying a Gstreamer Pipeline

A study of a pipeline will be exposed next. As an example, the initial receiving pipeline will be used.

Initial receiving pipeline:

- **tcpserversrc** name=insrc port=<port> host=<host> ! **tsdemux ! h264parse** ! **avdec h264** output-corrupt=false ! **videoconvert** ! video/x-raw,format=(string)RGB ! **videoconvert** ! **appsink** name=sink emit-signals=true max-buffers=1 drop=true

In bold font are the pipeline calls, the rest are the parameters used. Most of them are self-explained, the tcpserversrc defines the video source, and the app sink emits a signal to the application passing the output of the last pipeline.

To further understand what is happening in each pipeline, we generate the graphical representation, shown in the figure 3.5.

As we can see in the figure, all information about each pipeline is displayed. Each pipeline is connected to the next one by the src-sink structure. The configuration used and the image format is also shown.

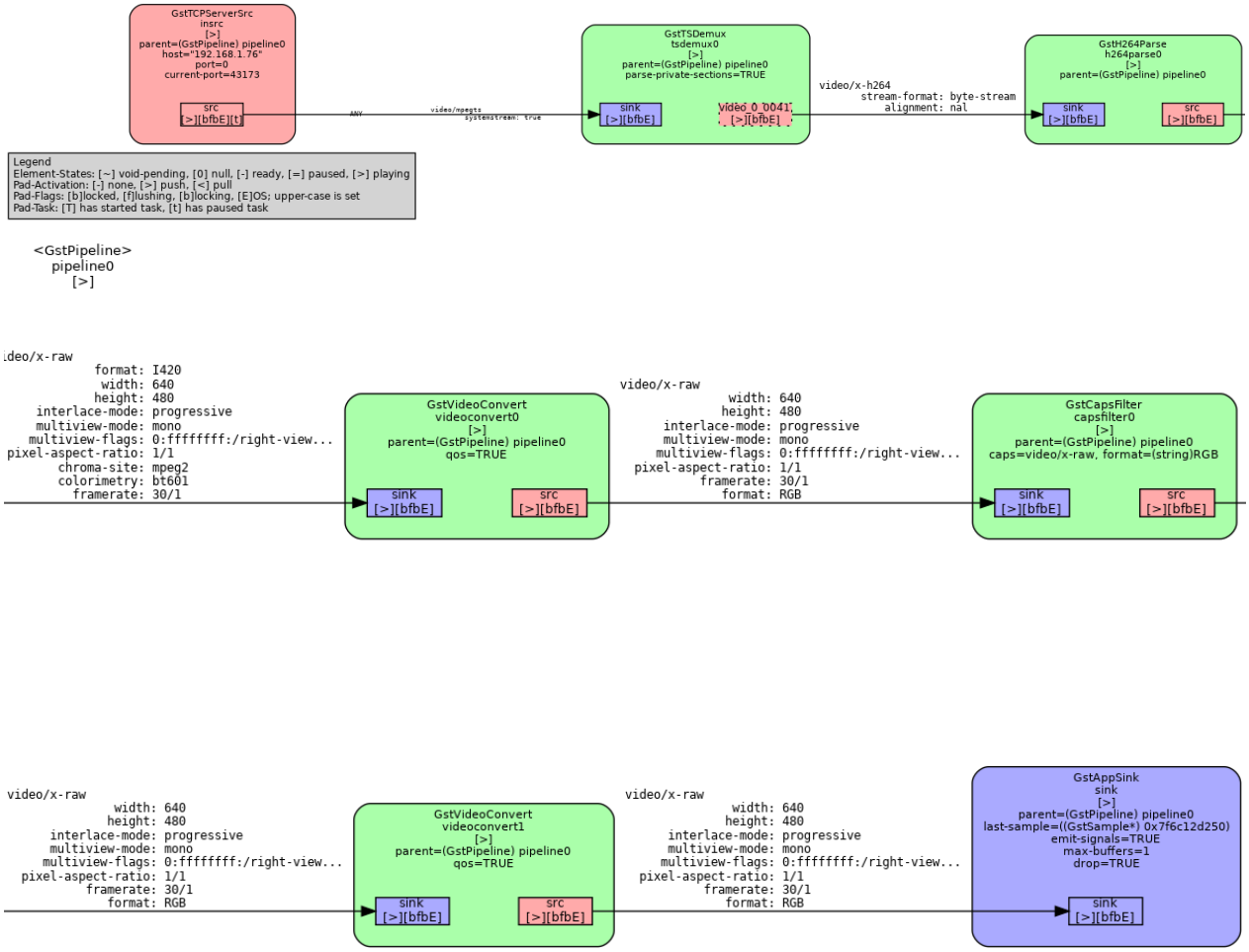


Figure 3.5: Gsteamer video receiving pipeline.

3.3.4 Final Gstreamer Pipelines

Using the tools exposed previously and all the information available on the Accelerated Gstreamer User Guide [18]. The pipelines on the project were changed as next:

- **Video receiving pipelines**

Initial pipeline:

tcpserversrc name=insrc port=<port> host=<host> ! tsdemux ! h264parse ! avdec h264 output-corrupt=false ! videoconvert ! video/x-raw,format=(string)RGB ! videoconvert ! appsink name=sink emit-signals=true max-buffers=1 drop=true

Accelerated pipeline:

tcpserversrc name=insrc port=<port> host=<host> tsdemux ! h264parse ! omxh264dec enable-low-outbuffer=1 disable-dpb=true ! queue ! videoconvert ! video/x-raw,format=(string)RGB ! appsink name=sin emit-signals=true max-buffers=1 drop=true

- **Video transmitting pipelines**

  Initial pipeline:

  appsrc name=appsrc stream-type=0 is-live=true do-timestamp=true format=time caps=video/x-raw,format=(string)RGB,width=(int)640,height=(int)480,framerate=30/1 ! videoconvert ! x264enc speed-preset=1 bitrate=<bitrate> tune=zerolatency ! mpegtsmux ! tcpclientsink host=<host> port=<port> sync=false

  Accelerated pipeline:

  appsrc name=appsrc stream-type=0 is-live=true do-timestamp=true format=time caps=video/x-raw,format=(string)RGB,width=(int)640,height=(int)480,framerate=30/1 ! videoconvert ! omxh264enc bitrate=<bitrate> ! mpegtsmux ! tcpclientsink host=<host>

In short, the encoding and decoding pipelines were changed to use the specific hardware existing in the Jetson TX2. This was achieved by swapping the "avdec h264 !" pipeline to the "omxh264dec !" for the decoding, and "264enc !" for "mxh264enc !" for the encoding. Also, different "queue !" elements were added, which divided the tasks into different threads, the "queue !" elements were really important to upgrade the performance of the pipelines. Finally, a "videoconvert !" pipeline was taken out since it was not needed.

It is interesting to comment that the "videoconvert !" pipelines have an accelerated version using CUDA, called "nvvidconv !", but RGB formatting is not supported for the moment. In the future, this may be implemented, which could boost the performance even more.

3.3.5 ORB SLAM Acceleration

To accelerate the image processing, which creates the could point and the 3d objects, the ORB-SLAM2, was changed to the ORB-SLAM2 CUDA version, developed by Thien Nguyen [9].

By using this version of the library, not only we reduced the image processing time to almost half the initial time, from an average of 72 ms to 40 ms, but the image processing was also paralleled. When using this new library, we could process multiple images at once. This was a big change for reducing the end-to-end delay. A representation of the code flow comparison is presented next:

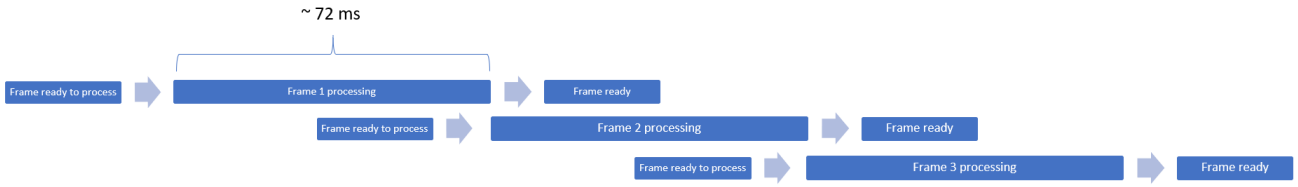


Figure 3.6: Original ORB-SLAM2 image processing representation.

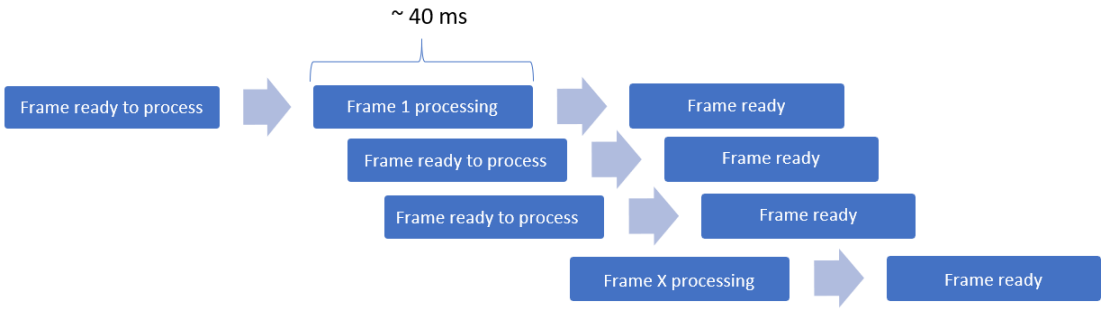


Figure 3.7: Accelerated ORB-SLAM2 CUDA image processing representation.

3.4 Metrics

The performance of MR Leo has been already tested in the work of Lindqvist [2], and also Eriksson and Akouri [3]. Some of their metrics are suitable to test the performance in the end-to-end latency and the performance in our work. Also, using equal metrics allows us to compare their work.

3.4.1 Testing Environment

All tests were taken using the following hardware and configuration:
- Router Asus RTAC51U, IEEE 802.11ac standard.
- Jetson TX2
- Smartphone Xiaomi Mi 8
- H264 encoding and decoding at a 2000 Kbits/s bit-rate.
- TCP communication between the smartphone and the router.
- Ethernet communication between the Jetson TX2 and the router.

For a more detailed description and all the specifications of the Jetson TX2 see the Appendix.

3.4.2 Testing Variables

- **Frame Round Trip Time**

  The frame round trip time (FRTT) is the end-to-end delay. The time elapsed since the frame left the smartphone until it came back to be displayed.

- **Time to Virtual Element**

  The time to virtual element (T2VE) is the time elapsed between the user orders to display a 3d object in the application until it is displayed on the screen. In short, the response time of the software to show a 3d object when it is called.

- **Mixed Reality Processing Time**

  The mixed reality processing time is the time spent to create the cloud point of one frame. It is the time elapsed inside the OBR-SLAM2.

- **Complete Time**

  The complete time is the full time taken from an image to get processed and be ready to be sent. In this metric, the time spent to encode and decode is not taken.

- **Refresh Rate and Skipped Frames**

  The refresh rate and skipped frames metric analyze the performance of the edge. Starting with 30 frames per second (fps) sent from the smartphone, some frames are dropped in the communication or lost due to a lack of resources from the server. This happens because frames are dropped to not oversize the queue and increase the delay.

Figure 3.8: Experiment setup [3].

4. Results

In this section, all results are presented. Each metric will be shown and discussed. The main objective of this chapter is to compare the performance between the initial MR Leo server the accelerated one, both running in the Jetson TX2.

4.1 Frame Round Trip Time

The next graph shows the CDF of the image delay or Frame Round Trip Time (FRTT). It is calculated in milliseconds. The blue lines show the base MR Leo, without the acceleration changes. On the other hand, the grey shows the accelerated times.

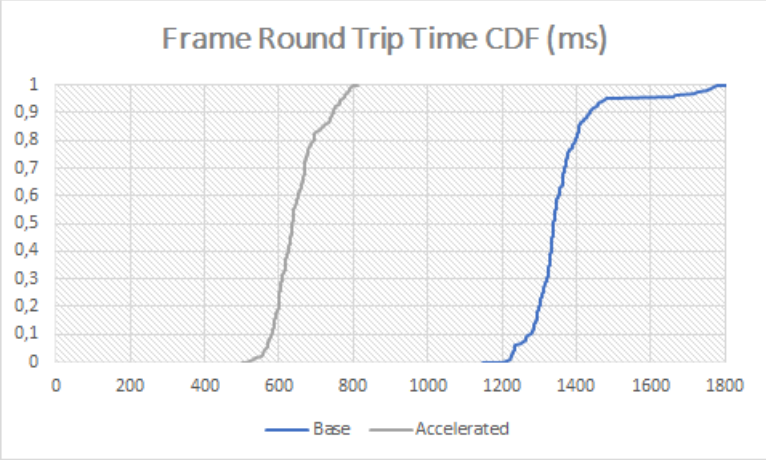


Figure 4.1: CDF of the FRTT in milliseconds.

FRTT	Base	Accelerated
Average	1360	650
Max	1779	798
Min	1215	529

Table 4.1: Average FRTT times.

As we can observe in the figures, the end-to-end delay has been reduced to more than half the initial time, from 1285 to 570 ms on average. It is important to point that in this metric the delay added from the smartphone and the delay from the transmission over the TCP is also included, this means that the delay in the server has been reduced by a greater percent.

4.2 Time to Virtual Element

The CDF of the Time To Virtual element is shown next. As before, the blue line shows the base times and the grey line the accelerated ones.

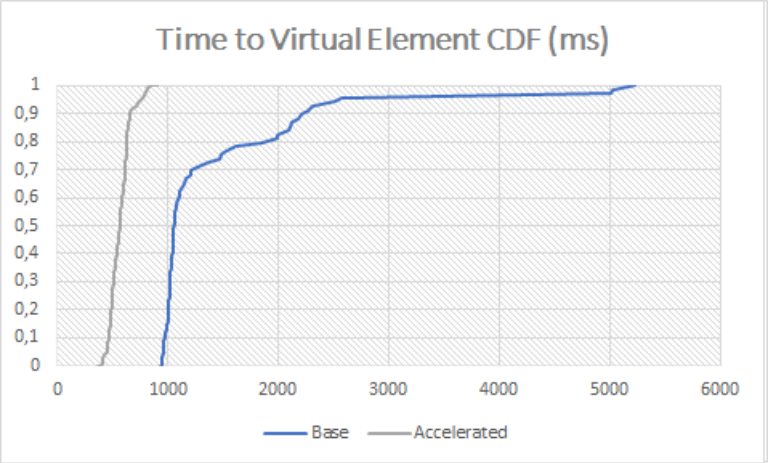


Figure 4.2: CDF of the TTVE in milliseconds.

TTVE	Base	Accelerated
Average	1285	570
Max	2570	544
Min	941	405

Table 4.2: Average TTVE times.

As expected, the TTVE has also reduced to less than half of the initial one. It should be pointed that, sometimes, the virtual element failed and this caused a delay that could go up to 5000 ms. In order to get a more realistic average and maximum TTVE, these metrics were not used to calculate the table, but it is reflected on the CDF.

4.3 Mixed Reality Processing Time and Complete Time

Next, the Mixed Reality Processing Time (MR Time) and the Complete time are presented. The maximum, minimum, and average time is shown in the tables.

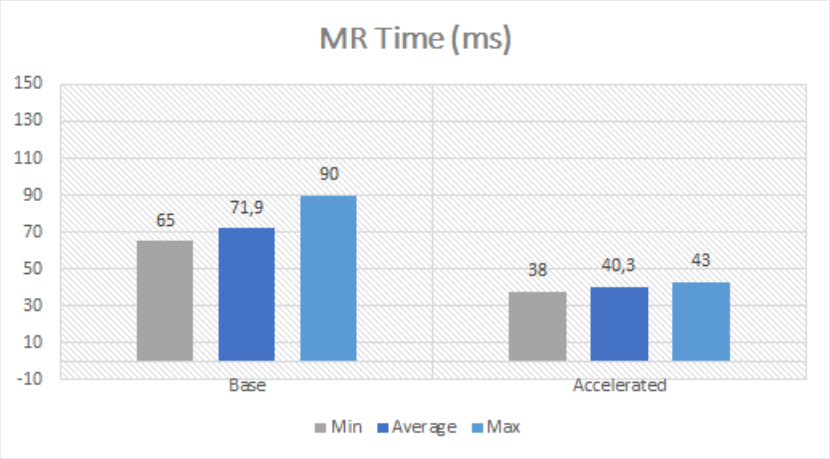


Figure 4.3: Maximum, minimum and average MR Time (ms).

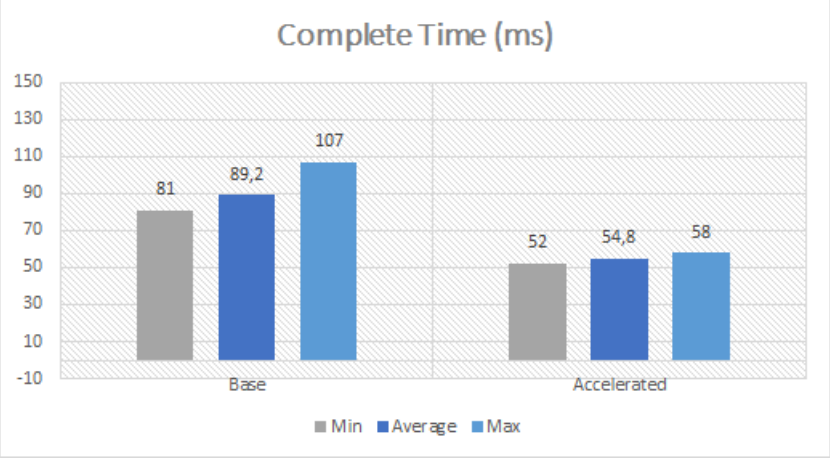


Figure 4.4: Maximum, minimum and average Complete Time (ms).

We can observe that the average time to process each frame has reduced to half of the initial. Then, we could ask that, if the overall processing time it's just reduced to 50 ms, why was the FRTT time reduced for more than 600 ms?

The solution comes with the parallelism achieved when image processing. As shown previously, now the image processing is being done by the GPU, which allows to process multiple images at the same time, see Figures 3.6 and 3.7. Also, with a lower CPU demand, the other tasks are getting faster.

4.4 Refresh Rate and Skipped Frames

The Refresh Rate in frames per second (fps) and the Skipped Frames in percentage are shown next. The maximum Refresh Rate would be obtained by having a 0% of Skipped Frames or, in other words, a 0% of Lost Frames. With this metric, we can see if the Jetson is capable of following the frame demand and also if the communication channel can be trusted. In this case, since TCP is being used, all the frames lost are most likely to be a lack of process power from the Jetson.

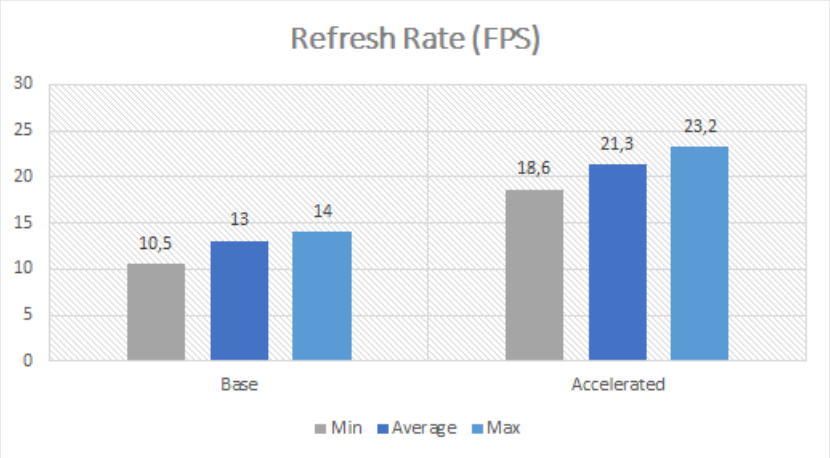


Figure 4.5: Maximum, minimum and average refresh rate (fps).

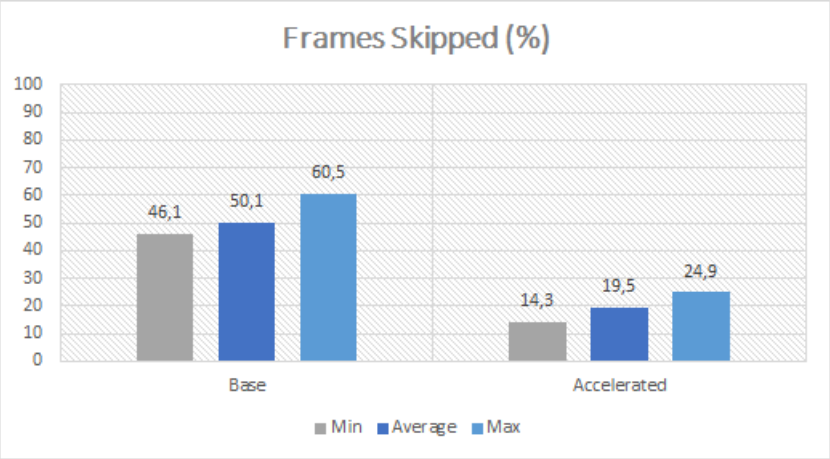


Figure 4.6: Maximum, minimum and average frames skipped (%).

We can see that using the GPU to process the images resulted in a performance boost. Not just because the GPU is being faster to process each image, but multiple images are getting processed at once, which decreases the frames lost caused by the lack of process capability. The maximum fps to process is 30 since it is the refresh rate coming from the smartphone, so, even with the acceleration, some frames are still getting dropped.

5. Budget

A brief analysis of the cost of this project is presented in this chapter. Three main aspects are used to estimate the budget.Material, human resources, and indirect costs:

Material: The material used in this project is presented in the next table. For the office material, a 10% depreciation over a life span of seven years was calculated. For the technology material, a four years lifespan was used.

Material	**Cost**	**Depreciation**
Office chair	100€	4.76€
Office table	130€	6.19€
Computer	500€	41.66€
Jetson TX2	393€	32.75€
Computer Peripherals	60€	5€
Total	1.183€	**115€**

Table 5.1: Estimated total cost from the materials.

Human cost: To calculate the human cost we use a telecommunication student with a salary of 11€/h. The thesis takes a work of 18 ECTS thesis, which are equivalent to 450 hours, the final cost will be **4950€**.

Indirect costs: The indirect costs will be calculated from the use of internet and electricity. Assuming a 30€ and 25€ per month respectively, the total cost in 4 months would be **220€**.

To sum up, a full cost is presented in the following table:

Concept	**Cost**
Material	115€
Human Costs	4950€
Indirect costs	220€
Total cost	**5285€**

Table 5.2: Estimated total cost of the project.

6. Conclusion and future work

6.1 Conclusion

Reducing the end-to-end latency was the main objective of this thesis. When we look at all of the test results, it is clear that this objective has been achieved and the acceleration on MR Leo has been successful. In the Jetson TX2, the end-to-end delay has been decreased from an average of 1360 ms to 650 ms, which is less than half the initial time. Besides, the performance of the software was also improved, processing almost doubles the image frames of the initial one.

When comparing its performance with the original one, which runs in a conventional computer [2], we see that it is just slightly better. In that scenario, MR Leo runs with an average of 750 ms using the same configuration. On the other hand, the frames dropped increased in the Jetson version, an average of 28 frames per second when running in a computer went down to 24. In any way, the 24 frames per second obtained are enough for a good experience when using the software but further investigation on the ORB-SLAM2 CUDA can be done to achieve better performance.

We find that the Jetson TX2 can run the software with a similar or even better performance than a conventional computer. Even that it has a lower computer capability, lower power consumption, and smaller size, it can keep up. This leads to an added mobility and makes the Edge device more flexible to different types of use in the future.

More acceleration in the code is possible to achieve in the Jetson device, there are little details that could be upgraded in the Edge software. On the other hand, the transport protocol used is also to be studied. All these topics are discussed in the next section as future work.

Overall, the acceleration changes make it viable to use the Jetson TX2 as an Edge device, but further acceleration is up to be done.

6.2 Future work

In this section future steps to take and future upgrades on the code will be discussed. Mainly next steps for further acceleration on the Jetson TX2 are presented and finally a discussion about the transport protocol:

6.2.1 Gstreamer Acceleration

In the Gstreamer pipelines, multiple transformations on the video are made to get the frames ready to process. One of them is the video conversion from I420 to RGB. This conversion is not supported by the Nvidia Accelerated Gstreamer [18] pipelines and it is, for the moment, getting processed by the CPU. Maybe in the future, this will be supported, in any case, adapting the ORB-SLAM2 to work using the I420 format would also reduce the processing time.

6.2.2 Image Processing Acceleration

Even if the processing time has reduced to almost half the initial time, the other most impacting change that reduced the end-to-end delay was the parallel processing of the images. Further acceleration can be done in the ORB-SLAM2 CUDA. For the moment, only an average of 24 % of the GPU is being used while processing the images, with some peaks of almost 50 %. This indicates that it would be possible to use more cores of the GPU.

6.2.3 Resolution Increase

As just commented, only an average of 24 % of the GPU is being used while processing the images, with some peaks of almost 50 %. This indicates that is highly possible to do an upgrade of the image resolution without sacrificing much end-to-end delay. This would have to be tested.

6.2.4 Memory Allocation Optimization

The use of memory is really important when using acceleration on the GPU. In CUDA programming we need to find which part of the code is slower. This code runs in the GPU and the CPU, but it also has memory work. There are different types of memory, and usually, the memory is not shared between the CPU and the GPU. Because of that if the GPU or the CPU process faster than the memory writing speed, we get to a situation called memory bounded.

As the code is right now, the frames arrive in the CPU memory, then get transformed to an OpenCV format for the ORB-SLAM2 CUDA, and then they are called to process. This means that the images are stored in the CPU just to get copied to the GPU later to get processed. Writing these frames directly to the GPU would also save time and reduce the processing time.

6.2.5 Transport Protocol

Lindqvist showed in his thesis [2] that using UDP over TCP decreased the end-to-end delay from 750 to less than 400 ms. Nevertheless, it also caused issues with the video receiving due to the packets arriving not in order. Using Real-Time Streaming Protocol (RTSP) might be the best solution for this type of communication.

Bibliography

[1] *Google AR VR*. URL: https://arvr.google.com/ar/.

[2] Johan Lindqvist. "Edge Computing for Mixed Reality". In: *MA thesis. Sweden: Linköping University* (2019).

[3] Jesper Eriksson and Christoffer Akouri. "Improving Performance of a Mixed Reality Applicaon on the Edge with Hardware Acceleration". In: *MA thesis. Sweden: Linköping University* (2020).

[4] Jesper Elgh and Ludvig Thor. "Speeding up a mixed reality application: A study of two encoding algorithms". In: *MA thesis. Sweden: Linköping University* (2020).

[5] Klervie Toczé, Johan Lindqvist, and Simin Nadjm-Tehrani. "Performance Study of Mixed Reality for Edge Computing". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing.* (2019).

[6] Jaegeun Han and Bharatkumar Sharma. "Learn CUDA Programming". In: (2019).

[7] Lars Linsen. "Point cloud representation". In: *Universitat Karlsruhe, Germany* (2001).

[8] R. Mur-Artal and J. D. Tardós. "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras". In: *IEEE Transactions on Robotics* (2017).

[9] Thien Nguyen. *ORB-SLAM2-CUDA*. URL: https://github.com/thien94/ORB_SLAM2_CUDA.

[10] *GStreamer: open source multimedia framework*. URL: https://gstreamer.freedesktop.org/.

[11] *Pangolin: a lightweight portable rapid development library for managing OpenGL display / interaction and abstracting video input*. URL: https://github.com/stevenlovegrove/Pangolin.

[12] *The OpenCV library*. URL: https://opencv.org/.

[13] "An Evaluation of the NVIDIA TX1 for Supporting Real-time Computer-Vision Workloads". In: *Department of Computer Science, University of North Carolina at Chapel Hill* (2017).

[14] Donald Bourque. "CUDA-Accelerated ORB-SLAM for UAVs". In: *MA thesis. USA: Worcester Polytechnic Institute* (2017).

[15] *Jetson TX2 Module*. URL: https://developer.nvidia.com/embedded/jetson-tx2.

[16] Ahmadreza Montazerolghaem, Mohammad Hossein Yaghmaee, and Alberto Leon-Garcia. "Green Cloud Multimedia Networking: NFV/SDN Based Energy-Efficient Resource Allocation". In: *IEEE Transactions on Green Communications and Networking* (2020).

[17] *Gstreamer Debugging Tools*. URL: https://gstreamer.freedesktop.org/documentation/tutorials/basic/debugging-tools.html?gi-language=c.

[18] *Accelerated GStreamer User Guide - NVIDIA Developer*. URL: https://developer.download.nvidia.com/embedded/L4T/r32_Release_v1.0/Docs/Accelerated_GStreamer_User_Guide.pdf?_tAXrc3k5egUE-Q1i_lRTXuSOgoCU_-5C2csvHgkw-wrO_2JwmvHrkdx4WQKThg_LqCxwxZg8Ohk-HTBpNxuKfgM-8KKl45TJdZOp9A_Mwq3KEqZyrPI9VCiuMtTIN17HREYqyJXBmVVXp9y35oQYVKdVaZ1Vpcdp-rUsIZ1b5UIeYQ.

7. Appendix

7.1 Jetson TX2 device

Jetson TX2 Technical Specifications:

- NVIDIA Pascal™ Architecture GPU.
- 2 Denver 64-bit CPUs + Quad-Core A57 Complex.
- 8 GB L128 bit DDR4 Memory.
- 32 GB eMMC 5.1 Flash Storage.
- Connectivity to 802.11ac Wi-Fi and Bluetooth-Enabled Devices.
- 10/100/1000BASE-T Ethernet.

Figure 7.1: Jetson developer box.

7.2 Gstreamer Pipelines

The Gsteamer pipelines are shown in the following order:

- Base receiving pipeline

- Accelerated receiving pipeline
- Base transmitting pipeline
- Accelerated receiving pipeline

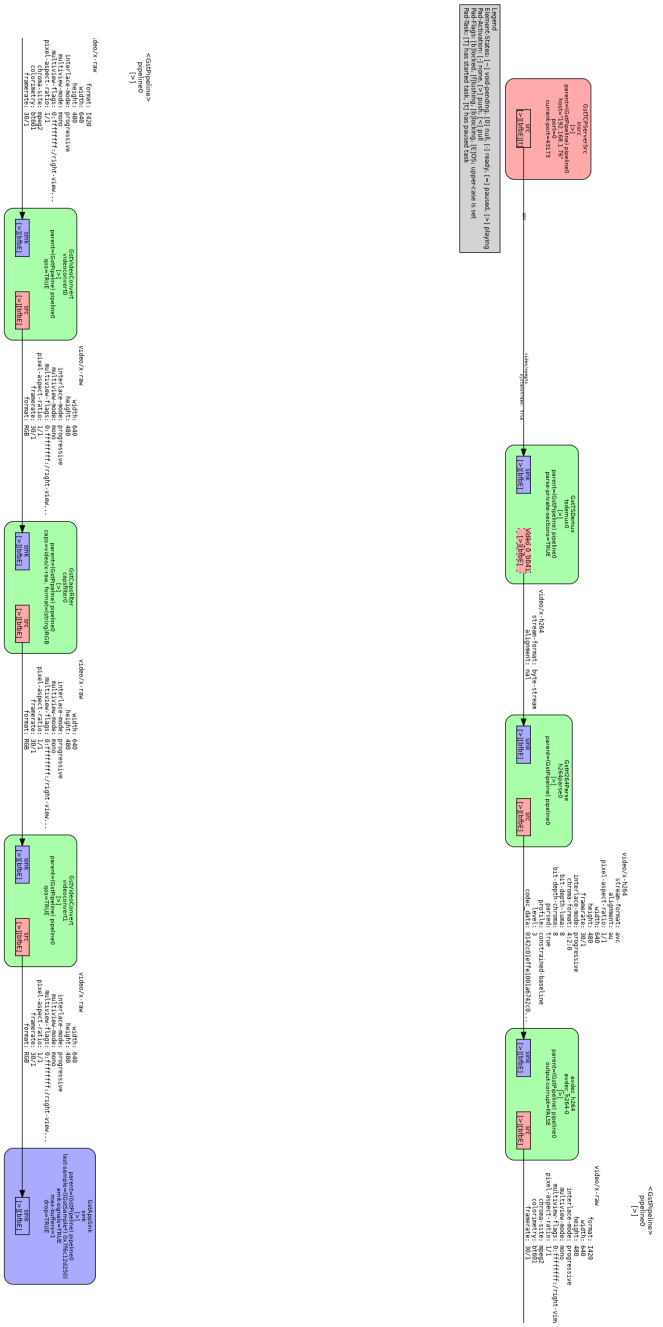


Figure 7.2: Base Gstreamer receiving pipeline.

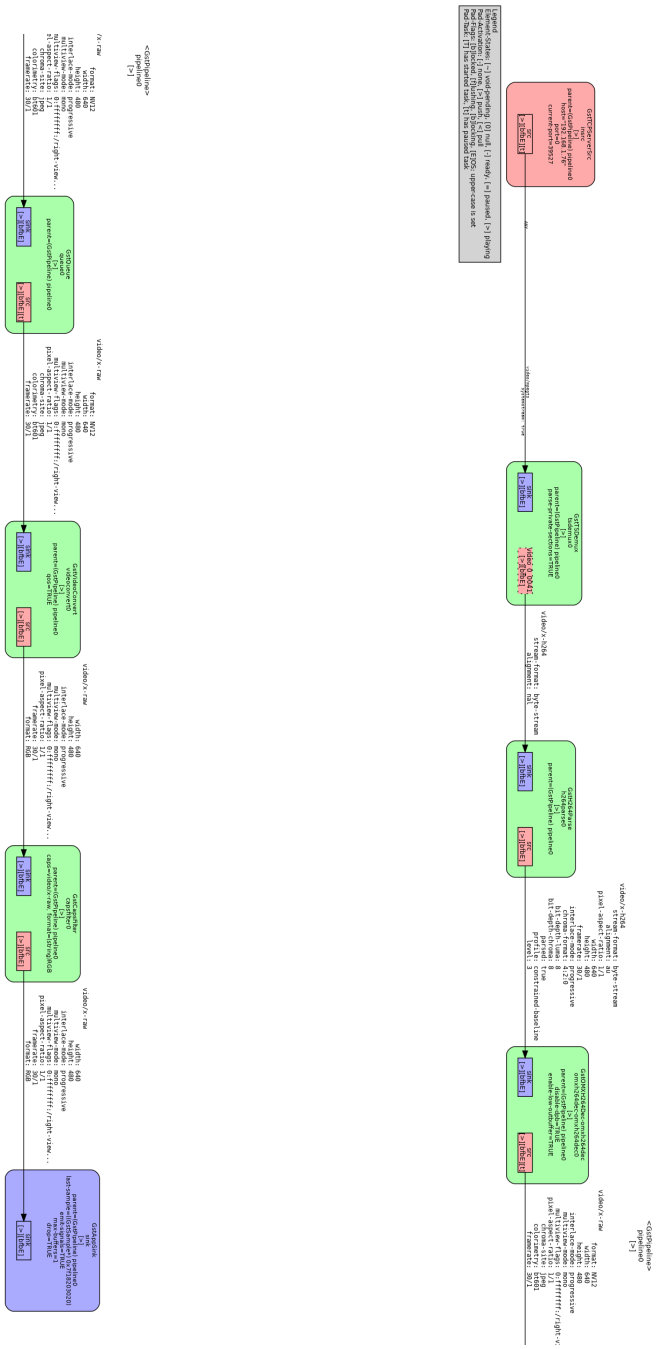


Figure 7.3: CUDA Gstreamer receiving pipeline.

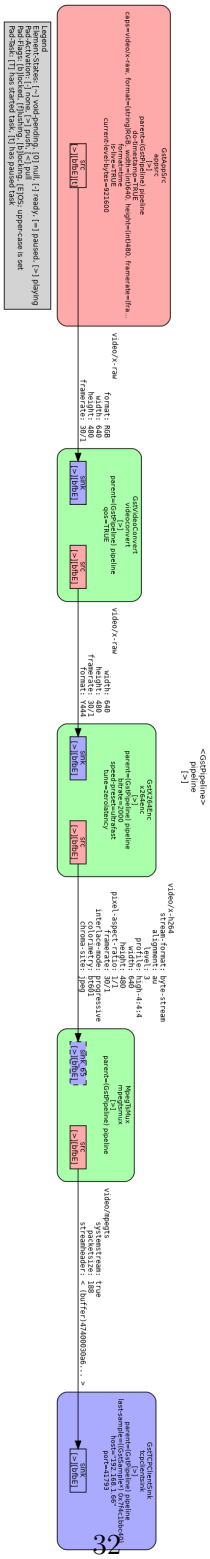


Figure 7.4: Base Gstreamer transmitting pipeline.

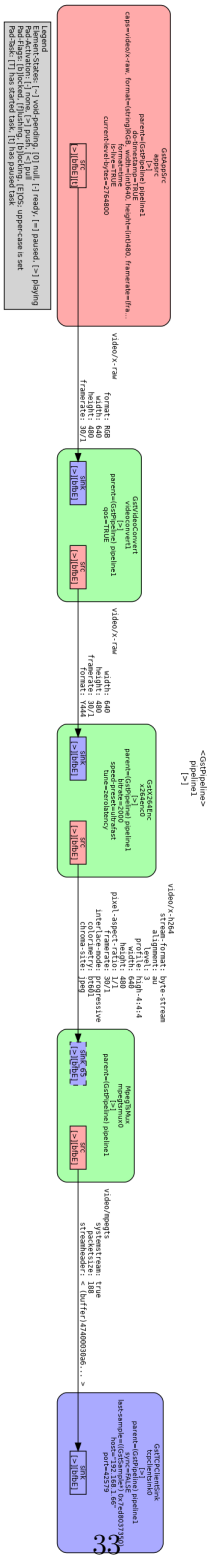


Figure 7.5: CUDA Gstreamer transmitting pipeline.