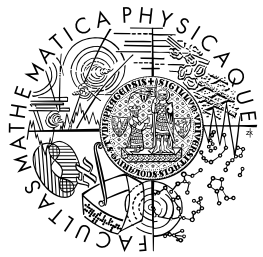

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Vojtěch Antošík

Module for real-time object detection in video stream

Department of Software Engineering

Supervisor of the bachelor thesis: prof. RNDr. Tomáš Skopal, Ph.D.
Study programme: Computer Science
Study branch: Programming and software systems

Prague 2020

This is not a part of the electronic version of the thesis, do not scan!

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express my gratitude to prof. RNDr. Tomáš Skopal, Ph.D., the supervisor of this thesis, for his feedback, his patient guidance and all advice he has given me.

I would also like to thank my family for their continued support during my studies and time spent working on this thesis.

Title: Module for real-time object detection in video stream

Author: Vojtěch Antošík

Department: Department of Software Engineering

Supervisor: prof. RNDr. Tomáš Skopal, Ph.D., Department of Software Engineering

Abstract: Over the last few years surveillance cameras have become ubiquitous. With so many cameras, analyzing the output manually has become very laborious and inefficient. In recent years, however, a lot of development has been focused on automatic video processing using artificial intelligence. There are many deep learning models for object detection offering basic low-level analysis. This thesis builds upon these models and creates an efficient video processing pipeline that serves as a base for further higher-level analyses. We aim to develop sufficiently fast video processing pipeline that will be able to process surveillance camera video streams in real-time while maintaining low CPU utilization. We move as much of the pipeline as possible to the GPU, with the data never leaving the GPU memory before the very end of the pipeline, and therefore leaving most of the CPU computational power for further data analysis. Our testing shows that our implementation achieves performance very close to real-time with 1080p video even on common consumer hardware.

Keywords: video surveillance, real-time video processing, real-time object detection

Contents

Introduction

The accessibility of surveillance cameras has been steadily improving in recent years. With cameras covering more and more public places, it is possible to use them in various smart city applications ranging from security and traffic monitoring to marketing. Due to an increasing amount of utilized cameras, manual analysis and processing has become very inefficient. But with the increasing power of common hardware and rapid development in artificial intelligence, much of the work can be done automatically without human intervention.

In the last few years, artificial intelligence, and especially machine learning, has gone through rapid development and has become very popular in many fields. Video processing in particular mainly benefits from the development of new deep learning models for object detection in images together with the ever-increasing computing power of widely accessible hardware.

The aforementioned deep learning models, however, only offer basic low-level analysis. Such results are usually not very useful and require further analysis and processing. Without such higher-level analysis, the results would still require significant amount of manual labor to obtain useful information which defeats the whole purpose of automated video analysis.

Goals

The goal of this thesis is to develop an efficient real-time video processing pipeline using selected CNN-based object detector and thus provide base for various analytical tools from the Videolytics project. We aim to create a sufficiently flexible pipeline architecture to allow for potential later improvements in any of the stages and simple extensibility.

Apart from the basic pipeline implementation we propose a few modifications that may speed up the pipeline, especially in the inference stage, while not sacrificing too much quality. We then test the trade-off between quality and performance of the whole process.

While the pipeline can serve as a base for many interesting analytical tasks, those are beyond the focus of this thesis. Our main focus is on the pipeline itself, using already existing object detector, and optimizing preprocessing and post-processing phases and data transfers inside the pipeline.

Thesis structure

Considering our pipeline is a part of a bigger project, we decided to begin with a brief overview of the Videolytics project as a whole. We therefore dedicate chapter 1 to description of the entire project as well as description of the separate modules that will be part of this project.

Chapter 2 contains a quick overview of important concepts related to our application. We mostly focus on object detection, which is a crucial part of this pipeline, and also briefly skim through some information regarding manipulation of image data and how they are stored.

Chapter 3 is an analysis of our solution. It mostly consists of discussion of various design decisions and overall architecture of the pipeline. It does not contain all the implementation details and is mostly intended as a high-level overview of the entire solution rather than detailed documentation.

Finally, the chapter 4 contains various experiments and performance evaluation tests of our application and its parts. It tries to identify bottlenecks of the pipeline and potential performance problems our pipeline might introduce in the rest of the project.

The thesis also contains two appendices documenting the entire solution. The first one, appendix A, is primarily made for users of the pipeline. It contains a manual on how the application shall be used, what options does it accept and what do they do. Appendix B, on the other hand, contains all the internal implementation details and is primarily intended for programmers who want to extend or modify this pipeline and for those who want to get more familiar with its internals.

1. Videolytics project

With recent development in deep learning and computer hardware, automatic real-time video analysis has become both possible and accessible and is currently a highly demanded functionality in various fields. Deep learning models are, however, not suitable for every case. Obtaining enough labeled training data can be potentially very complicated or even impossible. Hence analytical modeling might still be a viable approach especially for complex tasks where training data are not available or obtaining them would not be cost-effective.

Our project should serve as a complex analytical tool that will combine the aforementioned approaches using modules with various levels of abstraction. Since the project should be modular — which will lead to better flexibility for future development and easier maintenance of the modules — the architecture will be built around one central database (see fig. 1.1). All the extracted features will be saved in this database and therefore available to other modules for further analysis. We should then be able to gradually progress from basic low-level features to complex statistical information by using a series of modules with different levels of abstraction and feature complexity as illustrated in fig. 1.2. Using the modular approach, we will be able to optimize each module separately and control the aggregations in it and thus use the most efficient approach for each part.

The system is primarily targeted at surveillance cameras and their use in various smart city applications. It should be able to provide basic low-level information as well complex statistics from the aggregated data in a user-friendly way through a simple web interface (available at `videolytics.ms.mff.cuni.cz`). This thesis will focus on the very first module of the analytical pipeline that uses deep learning model for real-time object detection in video stream and thus provides a base for all the analytical modules in the project.

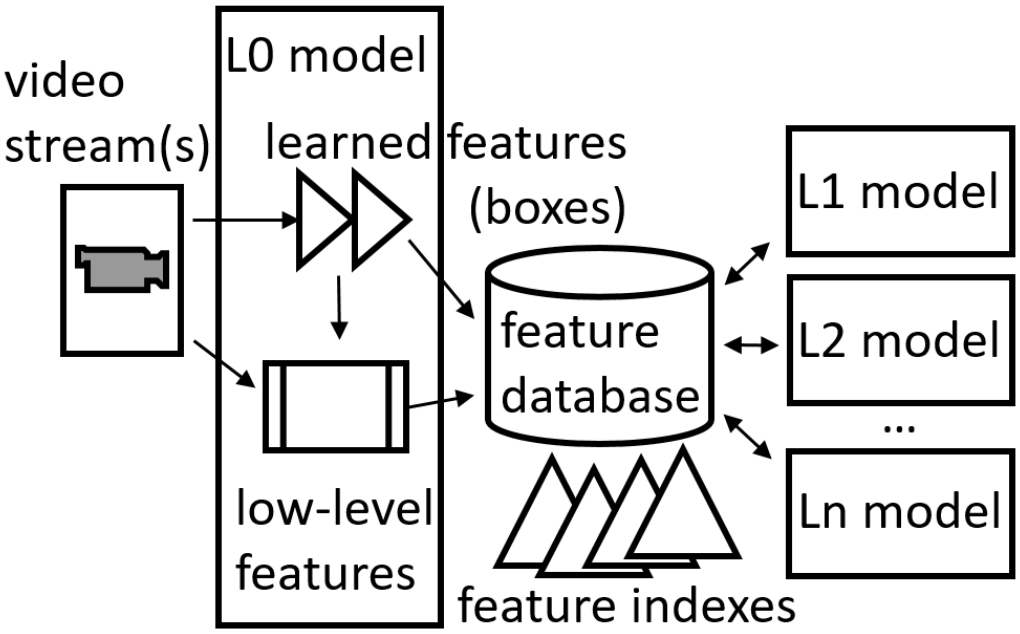

Figure 1.1: Schema of the Videolytics project architecture.

Figure 1.2: We extract basic low-level features from the video stream and further aggregate them into ones with a higher level of abstraction. Basic detections from the left image are being aggregated into trajectories of moving objects in the right image.

2. Background

In chapter 3 we will be discussing design decisions and problems our application has to deal with. The implementation contains various concepts and technologies that the reader might not be familiar with. In order to allow reader to easily understand our implementation, this chapter will offer a basic overview of these concepts and technologies. Those who are already familiar with presented topics may safely skip this chapter.

2.1 Storing image data

A big part of our application is concerned with manipulation of image data. Hence we would like to remind the reader of a few basic concepts, related to representation of image data, that might be relevant for our implementation. We do not intend to offer a detailed description of these concepts but rather basic principles that should be sufficient to make our implementation easily understandable.

2.1.1 Color model

This is a concept most people will likely be familiar with to some extent. A color model is an abstract mathematical model that describes how colors can be represented (usually using tuples of numbers). Such models often use three or four components to describe a given color. The color models relevant for our implementation are called RGB an YUV.

RGB

RGB is an additive color model that uses three primary colors - red, green and blue (hence its name) - that are added together to reproduce a variety of different colors. Since it is an additive model, it is used mainly to represent and display images in electronic systems that emit light (such as monitors, TVs or projectors).

YUV

This color model is usually used to encode videos or images for storage or transport. Contrary to RGB, it does not use mixing of primary colors to represent a given color but instead represents each color as a combination of one luma (Y) component and two chrominance (U and V) components.

Figure 2.1: UV plane with a fixed Y value of 0, 0.5 and 1 respectively. U goes from -0.5 on the left to +0.5 on the right and V from -0.5 at the bottom to +0.5 at the top.

While this might sound like a very arbitrary way to divide colors, such model has some very useful properties.

Since human vision is more sensitive to luma than chrominance, this model allows us to use a technique called chroma subsampling, which reduces size of the encoded image by only saving chrominance sample for a group of pixels rather than each individual pixel. This allows us to reduce size if the image data while retaining images that are perceived as higher quality ones when compared to straightforward resolution reduction.

The aforementioned property is not exclusive to YUV. But historically, YUV was created for its other advantage. During the transition from black and white to color TV, it was possible to add color using a new carrier for chrominance channels without breaking compatibility with black and white TVs. Those would still only use the luma channel and therefore display the grayscale version of the encoded video (luma with chrominance equal to 0 corresponds to shades of gray as seen in fig. 2.1).

2.1.2 Color space

Color model on its own is more or less arbitrary system since it has no connection to a universally known system of color interpretation. We therefore need a reference set of colors that we can map our model to. This is why we use so called absolute color spaces (e. g. CIEXYZ) that have colors defined colorimetrically without reference to external factors.

When we add a mapping between our model and a reference color space we get a gamut, which is a subset of colors from the reference color space we are able to represent with our model. This itself defines a color space for the given color model.

2.1.3 Pixel format

We already know what information we have to store in order to represent certain colors but there are multiple ways to store this information in memory. Without exactly specifying how such data should be interpreted, we will not be able to reproduce the encoded colors. There are three frequently used types of pixel formats used to store image data — planar, packed and a combination of these called semi-planar. Each of them has certain advantages and disadvantages but in many cases they are interchangeable.

Packed

This format only uses one plane for all the channels. Components of each pixel are "packed" together (stored right next to each other) in memory (see fig. 2.2). This is a significant disadvantage when any of the channels is subsampled because in such case odd rows will be different from even ones and odd and even pixels in a single row might have different format. However, this format may be useful when working with numeric libraries (such as NumPy) since it can be easily stored in a generic n-dimensional array (this is the layout usually used to store such arrays). This type of format (8-bit BGR) is used by the detector in our application.

Planar

Sometimes it might make sense to separate different channels into separate planes. Pixel formats using such approach are called planar. Each channel has its own plane where samples from adjacent pixels are stored right next to each other (see fig. 2.2). One of the biggest advantages of planar formats is that they can be easily used in cases where chroma subsampling has been applied. Mainly so because the chroma information is separated and leaving out some of its samples does not deform the entire plane as in packed formats.

Semi-planar

This type is well suited for images where chroma subsampling has been applied, maybe even more so than the aforementioned planar formats. As an example, we may take an image with 4:2:0 chroma subsampling (shown in fig. 2.2). In planar format, luma and chroma planes would have different width and it would be complicated to store them in a single array. Meanwhile, in semi-planar format, the combined chroma plane would be just as wide as the luma plane and the entire image could be therefore saved easily in a single two-dimensional array.

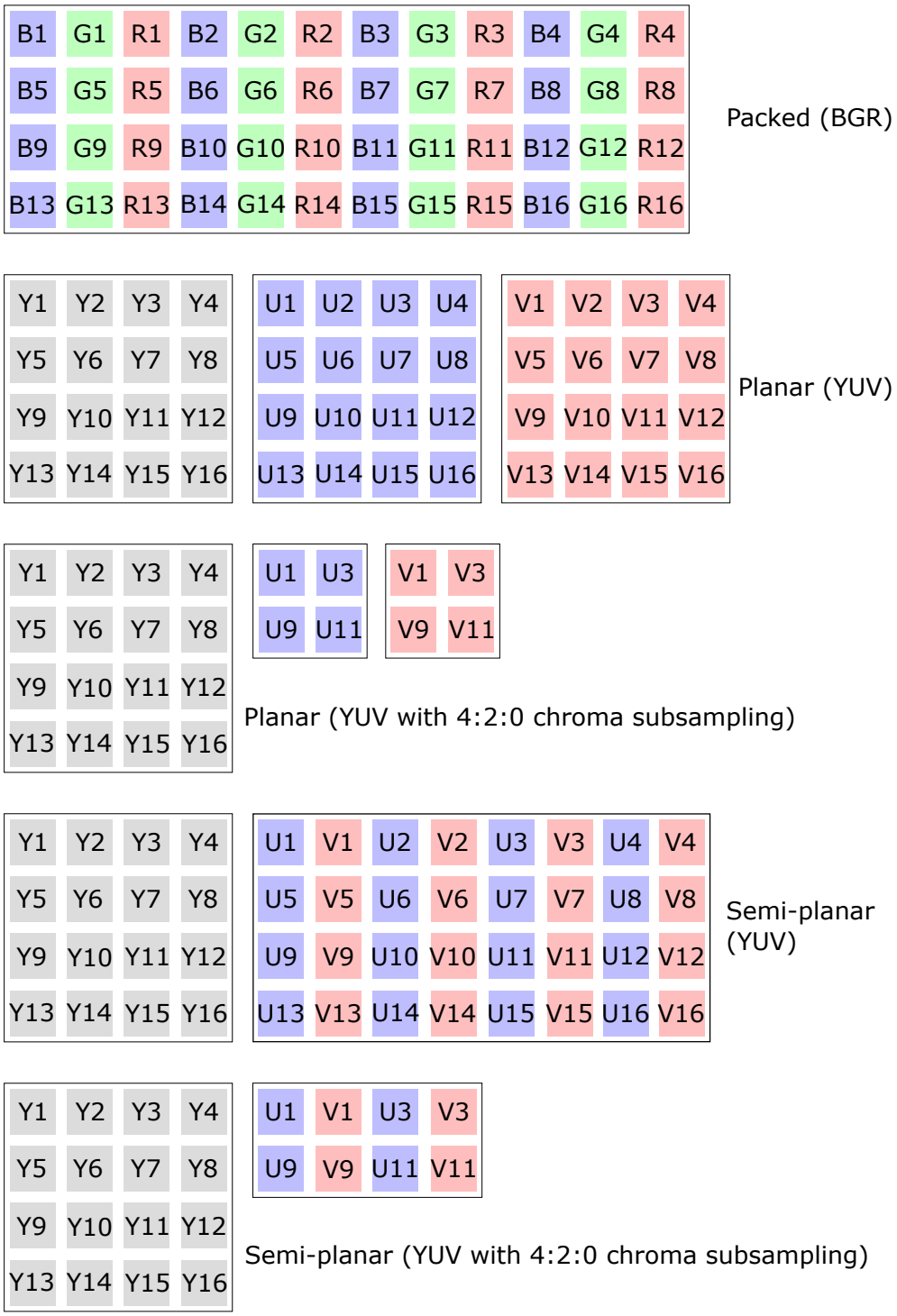

Figure 2.2: Example encoding of 4x4 image using different pixel format types with and without chroma subsampling.

2.2 Object detection

Object detection is the most crucial stage of our pipeline as it has great impact on the results of our application, both in terms of performance and precision. Hence we pay close attention to the available solutions and consider their suitability for our task. Despite the importance of object detection for our solution we only offer an overview of different techniques. We focus on their advantages and disadvantages and do not discuss them in much detail as that is beyond the scope of this thesis.

Our pipeline has rather high requirements with regard to performance and not many detectors are capable of delivering such performance, which restricts our options to only a few quite recent detectors. Even though there is no strict definition of what is considered real-time, we may consider 25 frames per second to be enough for the purpose of surveillance video analysis.

2.2.1 Two-stage detectors

Originally, most detectors used two-stage approach. The first stage for detection of objects inside the image and the second one for classification of these predicted objects.

The simplest approach to the first stage is to take an object classifier and evaluate it at various locations and scales across the image. As an example of such approach we may consider DPM [1], which uses a sliding window technique, evaluating the classifier at evenly spaced locations throughout the image.

Further development introduced a more sophisticated approach to the first stage. Instead of simple classification of numerous parts of the input, R-CNN [2] (and its subsequent improved modifications Fast R-CNN [3] and Faster R-CNN [4]) introduced region proposal methods that significantly improved performance by reducing the number of regions that require classification.

Despite all the improvements, especially in Faster R-CNN, two-stage detectors still tend to fall behind in performance and usually fail to offer real-time performance. That is a significant disadvantage in our case since our performance requirements are rather high.

2.2.2 One-stage detectors

In recent years, a new type of detector architecture has been developed. It combines both object detection and classification into one stage, which usually results in overall faster detection system.

While they have the potential to offer faster and simpler solution, the highest accuracy is currently still achieved by object detectors based on two-stage approach popularized by R-CNN. This might, however, change in the future as the trade-off between inference speed and precision is slowly improving in favor of one-stage approach.

For our needs we focus on two most popular one-stage detector architectures and compare their suitability for use in our pipeline.

You Only Look Once

Redmon et al. [5] introduced a completely different approach to object detection by joining detection and classification into a single stage. This new approach brings significant performance improvements as well as much simpler training since the detector can be trained end-to-end.

Originally, YOLO claimed to achieve 45 frames per second on Titan X GPU. This number was, however, further improved to 56 FPS with the introduction of YOLOv2 [6]. To achieve such results, it did sacrifice some precision but still managed to outperform other real-time systems [7][8] available at the time.

Both versions of YOLO mentioned so far achieve great inference speed but they do sacrifice on quality. This trade-off was slightly balanced with the introduction of yet another version, YOLOv3 [9], which introduced some changes to improve precision. While this newest version does give up a bit of speed, it can still be considered fast enough for real-time applications.

Single Shot MultiBox Detector

Another system offering truly real-time performance was presented by Liu et al. [10] only a few months after YOLO. It managed to outperform YOLO both in terms of precision and inference speed. And while both are based on the same principle, there are some important differences.

Contrary to YOLO, SSD's detection system is not grid-based. This might result in greater ability to detect objects in close proximity when compared to YOLO. This ability can be very useful for surveillance cameras as it may yield better detections of people in crowd for example.

Despite the updated YOLOv3 claiming to outperform SSD both in terms of inference speed and precision, we will settle with SSD as there happens to be an SSD version by Dobranský [11] that is modified and trained specifically for use in video surveillance. This makes it a better choice than YOLO since creating a detector optimized for video surveillance from scratch would be far beyond the scope of this thesis.

3. Solution analysis

With the required background, we can move to the solution itself. Before getting into the details of the implementation, we have to decide what the solution should look like. This chapter will discuss various design decisions and problems we might encounter during the implementation as well as their solutions. We will not include any implementation details as those will be thoroughly discussed later in appendix B.

3.1 Hardware limitations

Object detection is not a simple task and despite all the performance improvements we have seen in recent years, it still pushes hardware to its limits. Only a few years back even the common consumer hardware has gotten powerful enough to allow us to get even close to what might be considered real-time object detection. But, as seen in fig. 3.1, we have slowly reached a point where single core performance of modern CPUs is not getting much better over time. What currently brings the biggest possible performance improvements are increasing core counts. Not only do we have CPUs with tens of physical cores but even GPUs with hundreds or even thousands of them are now common. To keep up with this trend and to utilize as much computing power offered by the current hardware as possible, we have to adapt our applications accordingly, we have to parallelize. And thanks to the nature of our task, we have various opportunities to do so.

Our application revolves around object detection, which is based on deep learning, and image processing where lots of calculations independent of each other take place. This is exactly where GPUs excel and can significantly speed up the process. But since these cases are rather low-level and specific to particular module, they will be discussed more thoroughly in subsequent sections where we take a closer look on each module separately.

Even thought certain modules may benefit from the use of GPU, the performance of the entire application can still be significantly improved. Apart from the improvements that rely on the independence of separate pixels, we can introduce parallelism relying on the independence of entire frames that allows us to process more of them simultaneously. One approach to such problem that fits our solution well is pipelining. It allows us to separate different processing stages from each other and execute them simultaneously on different data.

These requirements partly limit our choice of programming language.

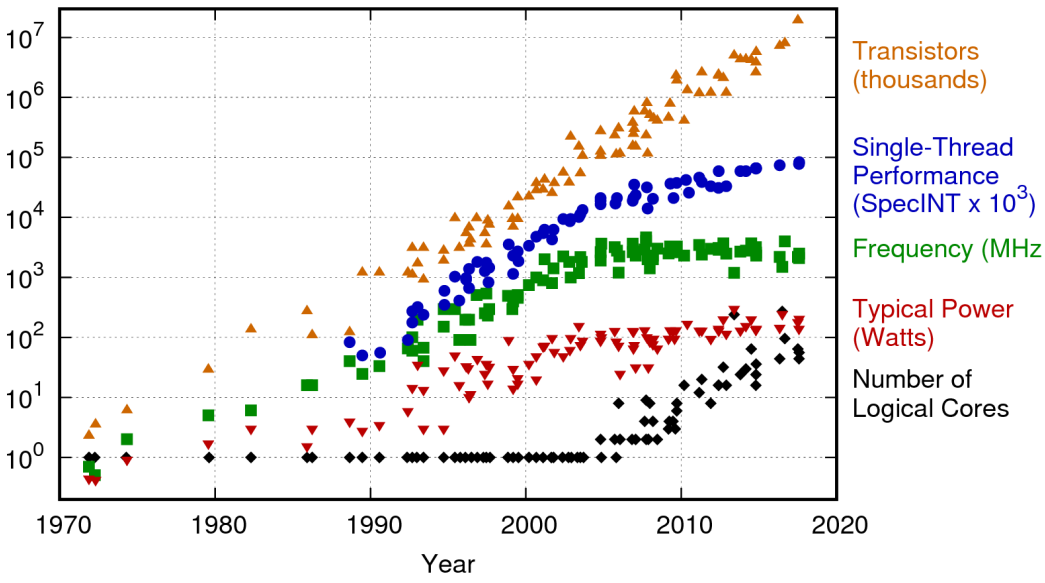

Figure 3.1: Processor data collected from SPECint benchmark with single threaded performance improvements slowing down and core counts rising in recent years. Source: `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`

Originally, we wanted to keep things simple and settle with Python since it is a comfortable language that would allow us to focus on functionality and the SSD we are using is written in it. But Python alone, since it is an interpreted language, proved to be inappropriate due to its poor performance. This can be somewhat mitigated as Python is written in C and it is rather simple to write extension code usable from Python in C or C++ to speed up certain computationally intensive parts of the application. But even then Python would not be a good choice because of its poor parallelization support. We would very likely end up writing almost the entire solution in C or C++ and only ”glue” fairly high-level modules together with Python. To avoid the need for developing any future modules as extensions for Python, which requires certain amount of boiler plate code, we decided to keep the entire solution in C++ and do the exact opposite. Instead of extending Python and using C++ code from it, we embed the interpreter and use it inside our C++ code. This means that only one module (the one embedding the interpreter) is required to cross the border between languages and other modules, even the future ones, will only be written in one language. Furthermore, C++ is the language of choice for various NVIDIA APIs that we might need in the future to accelerate certain parts of the application using GPU, which is yet

another point in favor of this language.

3.2 Pipeline architecture

As decided in the preceding section, we want to approach our task as a pipeline that processes image data in a series of independent stages. There are, however, a few limitations we have to keep in mind. We want our pipeline to be as generic as possible for future development but we want it to be effective enough to not bottleneck our solution or waste an unnecessary amount of system resources just to preserve its maximum genericity. We have therefore considered two different approaches to architecture of the pipeline itself when looking for a compromise between these two.

3.2.1 Non-generic approach

Originally, we tried a solution that was specifically tailored to the functionality required at that moment. Such solution lacked in flexibility and was quite hard to modify. In case of additional required functionality in the future, we would have to rewrite significant amount of the already existing code to again create a fixed architecture that is now able to fit all the required modules. Moreover, such changes might introduce new unnecessary bugs in the existing code and would lead to an increasingly complex solution that is harder to manage with every added functionality.

Another disadvantage of such approach is rather unclear structure of the pipeline. While offering a great freedom when it comes to possibilities of interaction between modules, it does not encourage a maintainable solution that makes different stages truly independent. Modules may be connected with each other in many different and often unexpected ways, which makes them harder to maintain as their modification may significantly influence other modules as well.

In our case, the entire solution was only intended to add required input and output capabilities to the detector it is based on. The SSD module was a central part of the pipeline that constructed and managed all the other modules as illustrated in fig. 3.2. During its own construction, all remaining modules were also constructed and initialized with all the necessary data passed between them. In the future as soon as somebody would want to extend the pipeline by, for example, adding a new preprocessing module, the shortcomings would become apparent. Such change would only be possible by modifying either input or detector stage as one of those would have to either construct and manage a new stage or take care of the preprocessing

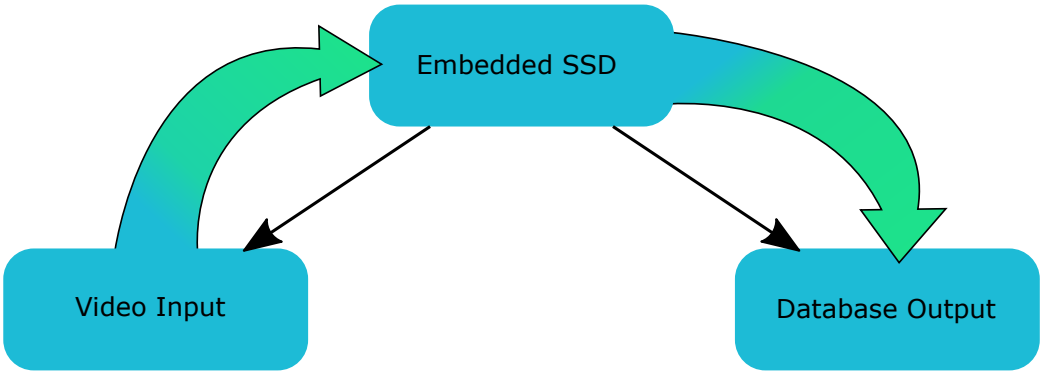

Figure 3.2: Schema of the original architecture with green arrows showing flow of data and black arrows signifying ownership.

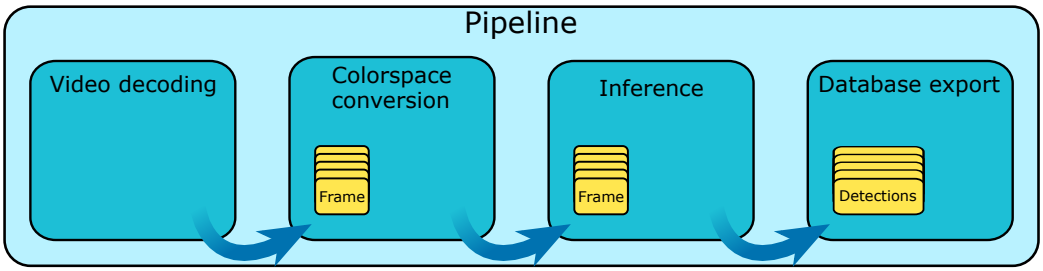

Figure 3.3: Schema of the generic version of the architecture. One object owns and manages all the modules and facilitates exchange of data between them.

itself. This would involve modification of at least one existing module and would only further complicate any future modifications.

3.2.2 Generic approach

Considering all the disadvantages of the initial solution, we decided to move towards a much more generic architecture. Instead of creating a fixed hierarchy of modules that requires significant modifications even for small changes in functionality, we now use a single generic object that owns and manages a series of single-purpose modules and facilitates the exchange of data between them as seen in fig. 3.3.

Every module that we want to include in our pipeline has to fulfil certain requirements. These have been summed up in the form of an abstract class that all the modules have to inherit from. This class represents an interface of a module. Apart from a few mandatory methods that the module has to override, it also contains two queues, one for input and the other for output,

that both hold a certain type of data. The only exceptions to this rule are the first and the last module in the pipeline that have no input and output respectively. The managing pipeline object automatically makes sure that the input and output of neighboring modules match and binds them together so that any subsequent exchange of data can be done through these queues. This approach makes it possible to create a module without any information about the rest of the pipeline other than the required input and output types of the module itself.

This results in an easier way to include new functionality through simple addition of new modules and encourages creation of simple single-purpose modules that only have one standard major way of communication with each other, which subsequently leads to more maintainable modules that can be modified without influencing other modules.

While this approach mitigates some of the biggest issues of the original architecture, it does come with its own drawbacks. Just like the improvements we have mentioned so far, these mostly stem from the strict definition of the data flowing through the pipeline and very limited options when it comes to communication between the modules due to lack of information about other modules in the pipeline. But despite being limiting to some extent, we believe that the generic pipeline architecture solves more severe and important problems than it introduces.

The biggest issue of our new architecture is more complicated and to a certain degree ineffective initialization of modules. As there are fairly limited possibilities of communication between modules, we cannot rely on direct initialization of the modules. We cannot directly pass required information from module to module as we have no information about other modules in the pipeline and therefore have to include the required information in the fixed datatype that we can pass through the queues. This might lead to a memory-wise ineffective datatype that has to include data only required once in the beginning and therefore takes up unnecessary amount of space most of the time. However, this issue can be somewhat solved by dynamic allocation of the initialization information and inclusion of the corresponding pointer instead of the entire structure in the datatype. A single pointer does not take up a lot of space and since the information is only allocated once, the dynamic allocation will not be a significant performance hit.

3.3 Modules

With the overall pipeline architecture out of the way, lets focus on each module separately. For now, only basic functionality is required and we have

therefore only implemented four modules so far. But thanks to the generic architecture of the pipeline, we can easily improve the functionality by adding new modules in the future.

As described in section 3.2.2, all the modules in our pipeline share the same base and are based on roughly the same principle that is enforced by our pipeline design and the restrictions that come with it. Even most future modules will very likely work in a similar manner. They will process data from the input queue and place results in the output queue until they are stopped and their input queue is empty.

3.3.1 Video input

The very first stage of our pipeline involves everything from reading a file or receiving an online stream to the decoding of frames. We have decided not to separate these tasks into two stages as they are very closely related and they should not be the bottleneck of our solution anyway. Due to the sheer amount of different codecs, formats and network protocols and their complexity, this module relies quite heavily on third party libraries (namely those from FFmpeg project). But despite using lot of functionality from these libraries, a lot of boiler plate code is still required and we have therefore decided to only support a handful of them. Thanks to our modular architecture, however, the module can be quite easily modified to support more of them without touching the rest of the pipeline.

One of the reasons this stage would not benefit from splitting is that we already offload part of the workload to the GPU. More specifically to a special dedicated hardware present on all modern NVIDIA GPUs that offers support for decoding of various mainstream video codecs. The resource usage might be negligible when processing a single input stream but with multiple streams at once, it might already pose a problem. This is where the dedicated hardware, called NVDEC, might offer noticeable improvements. As shown in fig. 3.4, its performance should be high enough not to bottleneck our solution as we would get to the limit of other parts much earlier. We also want to leave as much CPU resources as possible for other parts of the Videolytics project and we should therefore use every possibility to move part of the processing away from the CPU.

Currently, we only support decoding with NVDEC and only offer support for a subset of the codecs that NVDEC supports (namely H264 and HEVC). It is possible to add more codecs in the future but it requires mostly writing boiler plate code and we have decided not to do that for now and focus on more interesting parts. Another feature our implementation currently lacks is software decoding that would serve mostly as a fallback solution

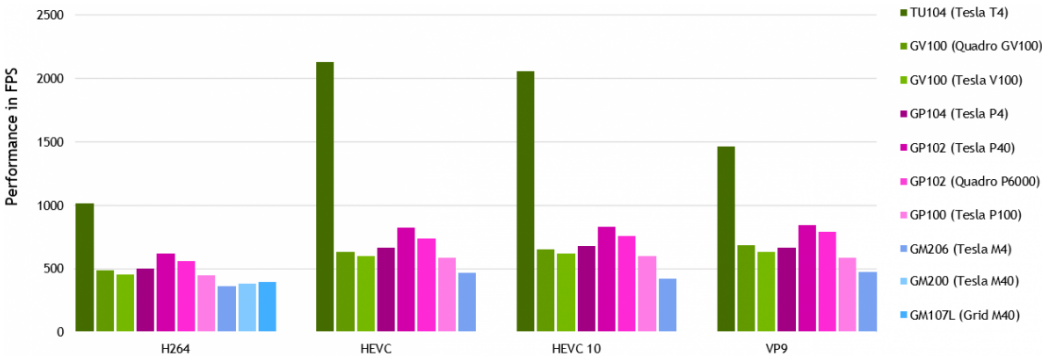

Figure 3.4: Performance of dedicated video decoding hardware on modern NVIDIA GPUs for 1920x1080 video. Source: `developer.nvidia.com/nvidia-video-codec-sdk`

when dedicated hardware either is not available or does not support required codec. That, however, should be something that the FFmpeg libraries should be capable of almost on their own and can therefore be implemented in the future without big modifications.

3.3.2 Color model conversion

Before the inference itself, data in our pipeline have to pass one more module. The video input and the detector, at least in our case, use different color models and are therefore not compatible with each other. Our detector uses RGB model while the decoder outputs frames in YUV. This module therefore serves as a compatibility layer between the input and detector stages. Originally, in the solution that used non-generic approach, the color model conversion was a part of the video input module. But to keep up with the philosophy of simple single-purpose modules that our pipeline design indirectly encourages, we have decided to make it a standalone module. This way it would be possible to change easily modify it, in case we need different conversion, without touching the input module.

As shown in section 2.1, color model conversion is a very simple process and thanks to its nature can be easily done in parallel. It is a simple case of matrix multiplication and can be done for each pixel individually. That is a task well suited for a GPU, which can handle such calculations in parallel. That allows us to once again leave more CPU resources for other tasks and even speed up the conversion at the same time.

Because of the fairly limited functionality of our input stage that currently only supports decoding on the GPU, we have decided to only implement color

model conversions for the pixel formats that might be relevant to us. We therefore only implement conversions from two planar (NV12 and P016) and two semi-planar formats (YUV444 and YUV444-16bit) used by the decoder to a single packed format (BGR24) used by the detector.

3.3.3 Object detection

After the necessary conversion, we can finally pass the data to the detector. Like many other machine learning project, the detector we use is implemented in Python. Since our detector of choice is already implemented, the only thing left for us to solve is passing data between C++ and Python. That is, however, not a big problem since Python is implemented in C and offers a very simple interoperability with C/C++.

Our case, however, is a bit specific. One of the first things that our SSD does when it receives data is moving them to the GPU where the computationally intensive part takes place. But as our data already reside in the GPU memory after the color space conversion, it would be a waste of time and resources to move them back and forth between the CPU and the GPU because such transfers are usually quite time consuming. We therefore need to find a way to keep the data in the GPU memory all the time to avoid the unnecessary transfers. It is sadly not possible to pass a pointer to data in GPU memory directly to the SSD as it is not supported by the machine learning framework our detector uses. With a little modification, however, we are able to pre-allocate an empty storage that is used by the framework to store data in the GPU memory and get its address. This means that we are able to fill this pre-allocated storage with the required data and thus avoid moving them away from the GPU.

3.3.4 Data export

All the processing is now finished but to be able to utilize the obtained information in a useful manner, we have to somehow pass them to other parts of the project for further analysis. As described in chapter 1, the Videolytics project revolves around one central database which is used to share all the information between modules. Since this is not performance-wise a very effective method, we should pay close attention to the performance of the export itself in order to avoid any additional significant performance decrease (especially with regard to latency).

Since we have no way to influence the communication between our machine and the database server, which depends mostly on quality of the connection between them, we have to focus on minimizing the need to send data

back and forth between our machine and the database and efficient processing on both ends.

To achieve the former — minimal need of communication — we shall eliminate any dependencies between data already inserted in the database and those that we are currently trying to insert. This means batching data (in our case frame data and detection data) together into bigger queries that take care of everything at once instead of sending multiple queries and waiting for their results in-between (and hence relying on the connection performance).

Due to the nature of our application, however, it is required to keep the latency at reasonable levels. This means we cannot batch data indefinitely since that would create a significant delay in the availability of data for other subsequent modules of the project (especially the web interface that shall be able to react to user actions with a reasonably low delay).

The reasonable compromise seems to be sending data for each batch of frames, processed by the SSD, separately. To further increase efficiency we can execute the required queries asynchronously and start preparing another batch of frames for export while waiting for the result of previous query (which is perfectly possible since our next query does not depend in any way on the result of the previous one).

4. Experiments and performance evaluation

Performance is a crucial aspect of our solution and we have therefore decided to carry out a series of experiments. They are primarily focused on existing implementation and its current performance but a few additional experiments, concerned with potential future modifications and their improvement potential, are also included.

Testing environment

The included results would have little to no meaning without the knowledge of our testing environment. The exact testing procedures for various tests will be discussed in the respective sections but unless stated otherwise, all the testing was done on the following hardware:

- AMD Ryzen 3600X CPU
- NVIDIA GeForce RTX 2070 Super GPU
- 32GB DDR4 RAM

4.1 Performance of separate modules

The ability to process video in real-time is essential for our application. Hence we have decided to test the performance of various parts of the pipeline separately to assess where the potential bottlenecks are and what causes them. This will allow us to focus on the most performance critical parts where future development may yield the most significant improvements.

To remain as accurate as possible, most tests were carried out using the same generic architecture as our application, which mostly allowed us to keep modules completely identical to the original application.

4.1.1 Queue throughput

Since the tests utilize the generic architecture from our solution, we first need to make sure that the architecture itself wouldn't be limiting in any way. Our first test is therefore focused on the queues used for communication between modules. Even though we do not expect them to be a bottleneck of our solution, we want to make sure that especially the locking required to

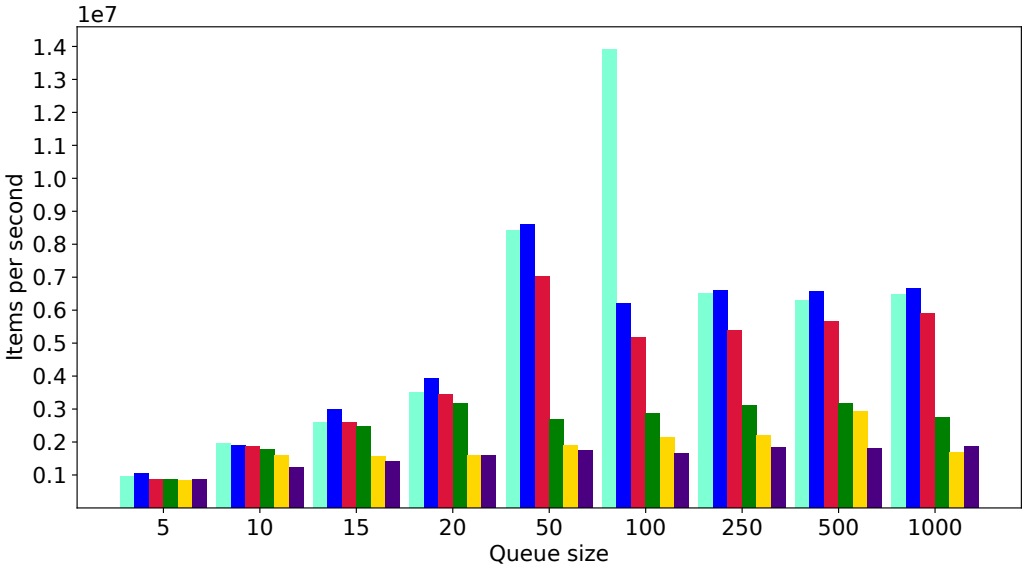

Figure 4.1: Throughput of queues in the pipeline. Data measured for structures of size 4, 16, 64, 256, 1024 and 4096 bytes respectively.

make them thread safe does not have severe performance implications. We therefore measure their throughput with different capacities and different sizes of the datatype used by them. To avoid possible inconsistencies caused by varying CPU utilization during testing each test is run ten times and an average is taken.

As illustrated in fig. 4.1, our expectations are correct. Even for queues with capacity as low as 5 the amount of items passed per second almost reaches a million, which is more than enough for our application. Even though increasing queue capacity increases throughput (especially for smaller objects), large queues barely have any benefits for our application (since we will never be even close to utilizing such throughput) and might even introduce problems such as high memory usage or increased latency. We will therefore settle with smaller queues that should be better suited for our task.

4.1.2 Stream demultiplexing and video decoding

According to our expectations, the first stage of our pipeline should be limited by the video decoding rather than stream demultiplexing. We therefore expect this stage to achieve performance figures roughly similar to those of the dedicated GPU decoding hardware shown in fig. 3.4.

In order to push this stage to its limit, we use an offline video source that

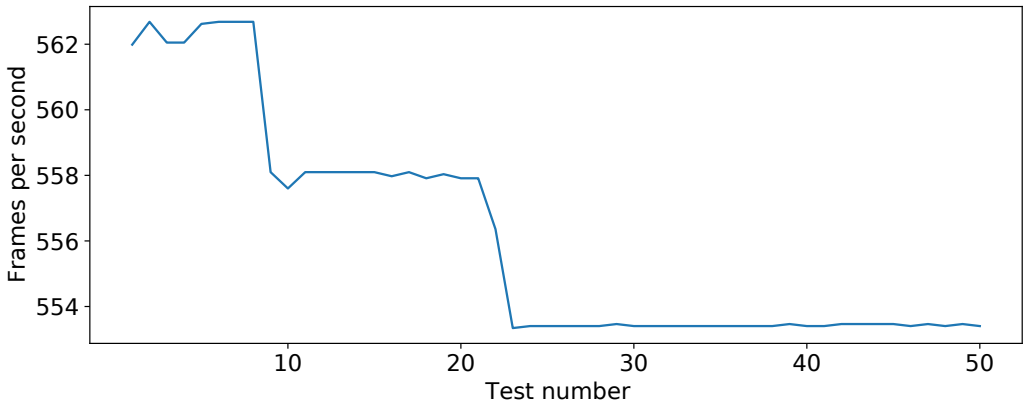

Figure 4.2: Performance of the video input module. Initial decline is caused by thermal constraints of the decoding hardware.

will not be influenced by network quality and can be processed faster than in real time (as opposed to an online stream that our application is mainly intended for). Our test uses an MP4 container with H264 encoded 1080p video stored locally on a solid state drive and we should therefore only be bottlenecked by the performance of the input stage itself.

We use a pipeline with two stages - the video input module and a module that counts and deallocates frames. During our testing, the queue between both modules never contained more than one frame (out of five possible) at a time and we can therefore safely conclude that the counting module is at least as fast as the input one and hence did not slow down the entire testing pipeline and produced accurate results.

We ran 50 identical tests to see how the decoding hardware handles sustained load. As illustrated in fig. 4.2, the results meet our expectations as they are very similar to the NVDEC performance. After the initial decline, which is presumably caused by thermal constraints of the decoding hardware, the performance leveled off slightly above 550 FPS. Such performance is sufficient even for multiple concurrent streams and will not bottleneck our application.

4.1.3 Color model conversion

The color model conversion is a rather simple task that can be easily parallelized and since we execute it on the GPU, we expect the performance to be quite high. We have decided to measure performance for each pixel format we currently support in our application. Each test is executed fifty times to make sure there is no significant performance decline over time.

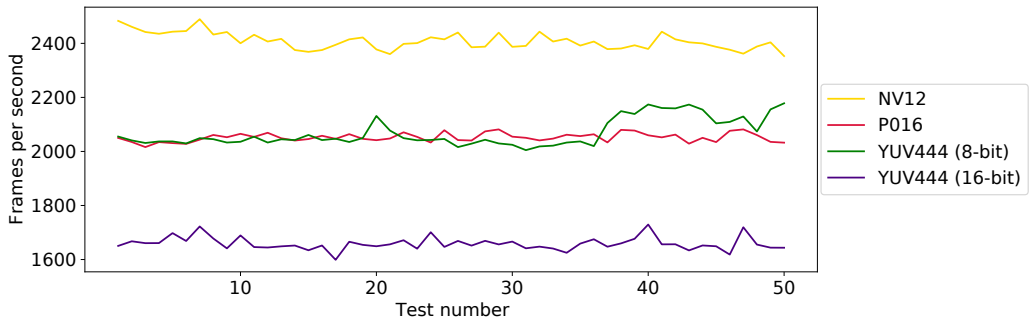

Figure 4.3: Color model conversion performance for various pixel formats over fifty consecutive tests.

According to fig. 4.3, there seems to be no performance decline over time and as expected, the only performance difference between different pixel formats seems to be caused by the total amount of data we have to process (16-bit vs 8-bit color and 4:2:0 vs 4:4:4 chroma subsampling). Even for the most demanding format (16-bit color with no chroma subsampling) the performance is high enough to handle even multiple streams.

4.1.4 Object detection

Another GPU accelerated module to be tested is our single shot detector. It is by far the most computationally intensive module in our pipeline. It is very likely to be the bottleneck of our application and a part that will require more focus in the future.

During the testing, we use a pipeline that consists of three stages - the first one allocates frames on the GPU, the second one runs the inference and the third one counts frames, deallocates them and measures time. For this test, we do not require our frames to be initialized with any meaningful data as the sole purpose of this test is to measure inference speed and not precision. Both testing modules by far exceed the performance of the inference one and thus will not bottleneck the testing pipeline and will measure accurate results.

Each test used a different batch size for the inference module and measured an average inference speed over 1000 frames. As seen in fig. 4.4, larger batches offer quite noticeable speed improvements. We are, however, limited by the memory consumption since bigger batches consume a significant amount of GPU memory (the biggest batch we were able to fit in our GPU memory had 16 frames).

Another thing to keep in mind is that this test only used the GPU for

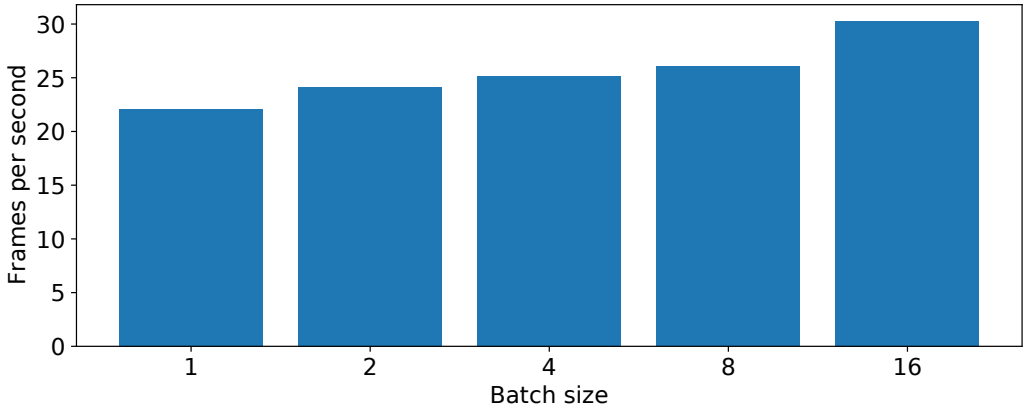

Figure 4.4: SSD performance for different batch sizes. Bigger batches offer noticeably higher average inference speed.

the inference module. In reality, the GPU will also be used for other stages of the pipeline such as color conversions and video decoding. Even tough the other modules are much less computationally intensive and require less memory, they might still affect the overall performance.

4.1.5 Export to database

As mentioned in section 3.3.4, performance of this section might be to some extent influenced by the quality of connection between the machine our pipeline is running on and the database. In our case this influence is reduced by the fact that we are running both the pipeline and the database on the same machine (some communication overhead is still present).

Since we want to isolate this module for the testing and keep the test results reproducible, we have decided to create the testing data manually with strictly specified properties. We test the performance with different combinations of parameters — namely different batch sizes, and number of detections in a single frame — to see how it depends on various aspects of the export. We have also decided to include performance results of the export module when crop exporting is turned on (although we expect this feature to be mostly used for further development and debugging and not for production use — for both performance and security reasons).

As seen in fig. 4.5, the performance varies greatly for different parameters. While for the low detection counts batch size makes a big difference, with the increasing complexity of the exported data (and especially if crops are involved) its importance declines quite rapidly. This is likely caused by the

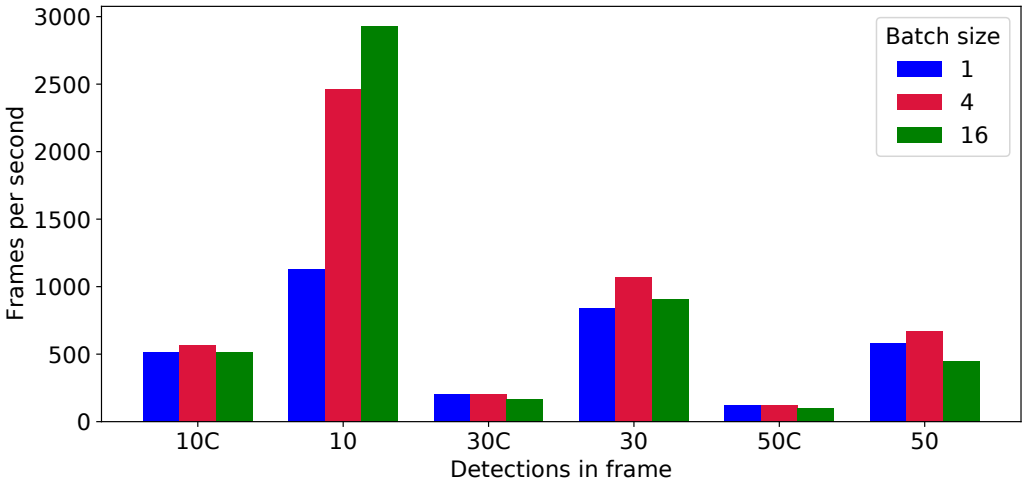

Figure 4.5: Database export performance for different combinations of batch size and number of detections per frame. Columns suffixed with `C` have crop export turned on.

fact that with more complex data majority of time is spent in the data preparation part and the communication overhead (presumably since we are running everything on the same machine) becomes negligible. With crop exporting turned off this preparation mostly means converting all the results from Python objects to data types suitable for communication with database. When the crop export is turned on the performance difference becomes very profound. Mainly so because our GPU to CPU memory transfers are not very well optimized, which is definitely something future development should be focused on.

Despite all these deficiencies, we are still able to achieve satisfactory export performance with the lowest measured export speed reaching roughly 99 FPS, which is currently enough for our needs.

4.2 Overall pipeline performance

While the performance of separate modules offers a good hint about where the bottlenecks should be expected, we would still like to test the pipeline as a whole and make sure our assumptions about its bottlenecks are correct.

To eliminate any network influence and ensure availability of data for processing, we tested the pipeline on an locally stored 1080p video. As seen in fig. 4.6, each frame spends a majority of time in the pipeline in the SSD module. This comes as no surprise since even during the isolated testing the

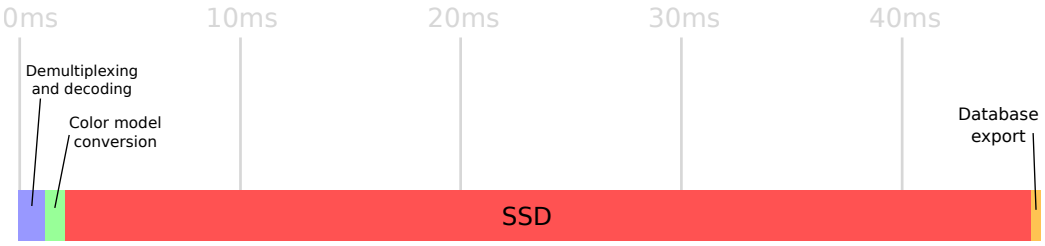

Figure 4.6: Average time spent in different stages by each frame.

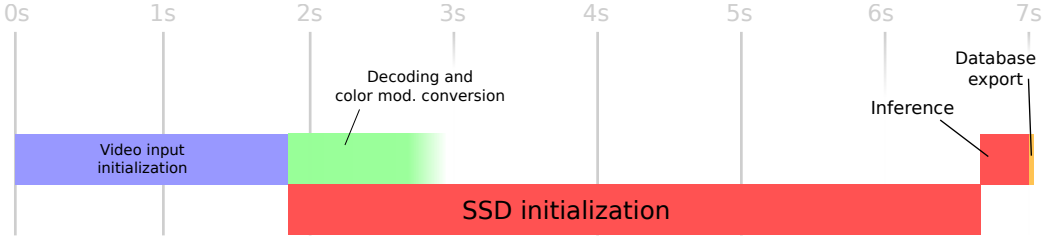

Figure 4.7: Initialization timeline from application start until successful export of the first batch.

SSD was by far the slowest part of the pipeline. This clearly confirms that the inference stage is in fact the bottleneck of our entire pipeline.

However, measured results are still not satisfactory because when tested separately, the inference module is able to keep the GPU utilization around 98% most of the time while with the complete pipeline GPU utilization hovers between 60 and 85%. This means there should be a potential room for improvement since the SSD on its own is able to fully saturate the GPU, which is currently not happening.

Upon further examination of the profiling information of CUDA code, we were unable to identify a single problematic part that would be causing our rather low GPU utilization. The suboptimal performance seems to be mostly caused by unoptimized manipulation with memory in the entire application and dependencies between data in different stages. Due to these dependencies, we have to wait for synchronization before being able to process the given data further.

Another thing to keep in mind apart from the average performance is the start-up latency. While it might not be very important for the processing itself, it greatly affects the user experience because the initial delay can be quite significant. We have measured the initialization in our simulated environment instead of offline video because connecting to online stream and extracting all the required information from it is not an insignificant part of the initialization time. As shown in fig. 4.7, despite our efforts to minimize

the latency the initialization still introduces quite a significant delay.

Conclusion

Our thesis was focused on efficient real-time object detection in video stream. We have briefly reviewed possible approaches to object detection using deep neural networks and their suitability for the task. We have then discussed potential obstacles we might encounter when implementing the module and how they can be resolved and to assess the viability of proposed solutions, we have implemented the module.

- In chapter 2, we have reviewed different approaches to object detection and discussed their advantages and disadvantages with regard to our solution.
- We proposed a generic and easily extensible solution of the detection pipeline and analysed its components required for basic functionality in chapter 3.
- By implementing the proposed solution and testing its functionality in a simulated environment together with other parts of the Videolytics project, we have demonstrated the viability of the solution and the entire project.
- Through thorough performance evaluation in chapter 4, we have tried to identify bottlenecks and limitations of the implemented module and propose parts on which development shall be focused in the future.

Future work

During the implementation, we have encountered several opportunities that might be viable for further improvement or research:

- Because of current rapid development in deep learning and its applications the state-of-the-art object detection models are being improved very quickly. Since the object detection is currently the most demanding part of the solution and its bottleneck, these improved models might yield significant improvements in the overall performance and usability of the solution.
- It might be possible to exploit the static nature of surveillance cameras to preprocess images in such a way that inference will be able to achieve similar precision with less data, which would alleviate the bottleneck created by the inference module while not sacrificing significantly on quality.

- There are countless possibilities for improvement in the video input stage. In its current state, the application only supports a small subset of existing codecs and formats. For example fall-back to CPU decoding might be quite important since only a handful of codecs is supported by GPUs.

- It is desirable to process multiple streams at once. The practicality of the simplest solution, where each stream has its own process, is very limited by the memory consumption of the deep learning model. It might be therefore necessary to introduce support for multiple streams in a single process that would share the model (only possible if the videos have identical resolution).

Bibliography

[1] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[2] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[3] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[4] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[5] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[6] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[7] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[8] Mohammad Amin Sadeghi and David Forsyth. 30hz object detection with dpm v5. In *Computer Vision – ECCV 2014*, pages 65–79. Springer International Publishing, 2014.

[9] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[10] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[11] Marek Dobranský. Object detection for video surveillance using the SSD approach. Master's thesis, Charles University, Prague, 2019.

A. User documentation

Our object detection pipeline was originally developed to be part of the Videolytics project but can be easily utilized elsewhere since it is a standalone application. This appendix shall therefore serve as a guide for anyone who wants to use this application as it is while not bothering himself with the internal details of it.

A.1 Installation

There are currently two ways to install our application. You can either use Docker, which is recommended, or install natively. For now, the application only works on Linux and due to its performance requirements, we discourage using virtualization to run it on another operating system.

A.1.1 Docker

We strongly recommend using this option as it takes care of (almost) all the application's dependencies and properly sets up the environment. Together with our code, we provide a dockerfile that can be used to automatically create a docker image with our application and all the required dependencies. We were not able to ship Video Codec SDK files in the attachment and it cannot be downloaded automatically since the download page requires login. After downloading the SDK make sure you place the entire downloaded folder in the top level directory of our project (it should be called *Video_Codec_SDK*).

To create the image simply execute:

```
docker build -f LivED/docker/lived -t lived .           (CUDA 10.2)
docker build -f LivED/docker/lived_cu101 -t lived .     (CUDA 10.1)
docker build -f LivED/docker/lived_cu91 -t lived .      (CUDA  9.2)
```

This should leave you with a docker image called *lived* that is ready for use. Be aware that the above mentioned command must be run from the top-level directory of the project (i. e. the one containing *LivED* and *SSD* folders) and that the CUDA version should match that supported by your GPU driver (you can check the supported version using **nvidia-smi**).

In case you would also like to build performance tests, add the appropriate build argument:

```
docker build -f LivED/docker/lived
             -t lived
             --build-arg COMPILE_TESTS=1
             .
```

A.1.2 Native

You might want to install the application natively instead of using Docker. In that case you have to get all the dependencies yourself and properly set up the environment. Please note that we have only tested this on Ubuntu 18.04 and the required packages might not be available on your distribution. In that case you would have to build the dependency from source. Also be aware that at the build time, the header files are usually required and they might come in a separate package. With each dependency we provide the version we have tested our application with but newer version should generally work just as well.

Dependencies

To build the solution, you will need:

- `g++` [7.4.0] (or another C++ compiler)
- `CMake` [3.10.2]
- `Python 3` [3.6] (Header files required.)
- `libavcodec, libavformat, libswscale` [3.4] (Part of the FFmpeg project. Header files required.)
- `libpq` [10.12] (Shipped with PostgreSQL. Header files required.)
- `NVIDIA Video Codec SDK` (Has to be downloaded from the official site: `https://developer.nvidia.com/nvidia-video-codec-sdk`)

To run the application, you will need:

- `Python 3` [3.6], `pip` [9.0.1]
- `libavformat` [3.4]
- `libpq` [10.12]
- `libsm6` [1.2.2]

Building

After getting all the required dependencies, you are ready to build the solution. To make this step easier, we use CMake. We recommend an out-of-tree build in a separate directory to keep everything organized. First, prepare the build directory:

```
cd LivED
mkdir build
cd build
```

Now build the solution using:

```
cmake ..
make
```

Optionally, you can specify what type of build you want (*Release* or *Debug*) and whether you want to also compile performance tests. We also recommend using more threads to execute make (using `-j` option, ideally the number of logical cores of your processor). The resulting commands can look like this:

```
cmake -DCMAKE_BUILD_TYPE=Release -DCOMPILE_TESTS=1 ..
make -j 8
```

A.2 Running the pipeline

Our application has only a very simple command line interface. The user can specify input URL (for local files prefixed with `file:`) and a few options that affect processing and output. If any of the options is specified more than once only the last occurrence is taken into account. Option and its corresponding parameter (if any is required) have to be divided by whitespace character. To start the pipeline, simply execute:

```
LivED [options] <input URL>
```

For docker images:

```
docker run --rm --gpus all lived ./LivED [options] <input URL>
```

Proper termination of the pipeline can be achieved by sending `SIGINT` to the runnning process.

Options

- `-w <file>, --weights <file>`
  - Specify path to saved SSD weights.
  - If no path is specified, pre-trained weights are downloaded but since they are for a different model they might only be useful for example for performance testing.
- `-n <network>, --network <network>`
  - Choose network used for SSD.
  - Accepted values:
    - `resnet`
    - `vgg`
    - `xception_[A, B, C, D, E, F, G, H, J]`
    - `nasnet`
    - `ssdtc`
  - Defaults to `resnet`.
- `-s <size>, --net-size <size>`
  - Specify size of the network.
  - Only applicable to `resnet`. (Has no effect for other networks)
  - Defaults to `34`.
- `-b <size>, --batch-size <size>`
  - Specify number of frames processed in one batch.
  - Defaults to `1`.
- `-c <name>, --camera-name <name>`
  - Specify name used when storing camera information in database.
  - Defaults to `N/A` or file name in case of local files.
- `-f <limit>, --frame-limit <limit>`
  - Set maximum capacity of queues between modules.
  - Defaults to `5`.
- `-d, --drop-frames`

  - If a queue is full, drop the frame instead of waiting for free space.

- `-e, --export-crops`
  - Store the cropped frame for every detection in the database.

B. Implementation internals

This appendix describes the internal implementation of the entire solution and is therefore intended mostly for programmers wanting to extend our solution or make their own based on a similar architecture. All the source code discussed in this part is also included as an attachment of this thesis.

We expect the reader to be familiar with C++ as it is the main language used in our implementation and also to have a basic knowledge of Python which is used in one of the modules to integrate an existing SSD written in it.

B.1 Pipeline

To provide a sufficiently flexible architecture that will be able to adapt to the future development of the pipeline, we created a small generic pipeline library. Apart from the **Pipeline** itself, it also contains an abstract **Module** class which describes an interface every module has to implement in order to fit in the pipeline. To make development a little more comfortable we have also included a **TSQueue** class that should facilitate exchange of data between modules.

B.1.1 Communication between modules

Arguably the most important problem our pipeline solves is passing data between modules. To be able to easily extend or modify the pipeline in the future, each module included in it should have some sort of "standard input" and "standard output". If we force all the modules to adhere to this rule and to specify what type of input data is expected and what type of output data will be produced, we can easily check that the supplied modules will be compatible with each other and establish a communication channel between them. This allows the programmer to create an extension module without modifying or even understanding existing ones solely based on the input and output of neighboring modules. It also alleviates the need to write a lot of boiler plate code to facilitate communication in longer pipelines.

Our **TSQueue** class serves exactly this purpose. Every **Module** has one such input and one output queue (apart from modules at the beginning and at the end of the pipeline). The class itself is nothing more than a simple wrapper around **std::queue** that adds thread safety so that the queue can be safely shared by two concurrently running modules. It also allows the

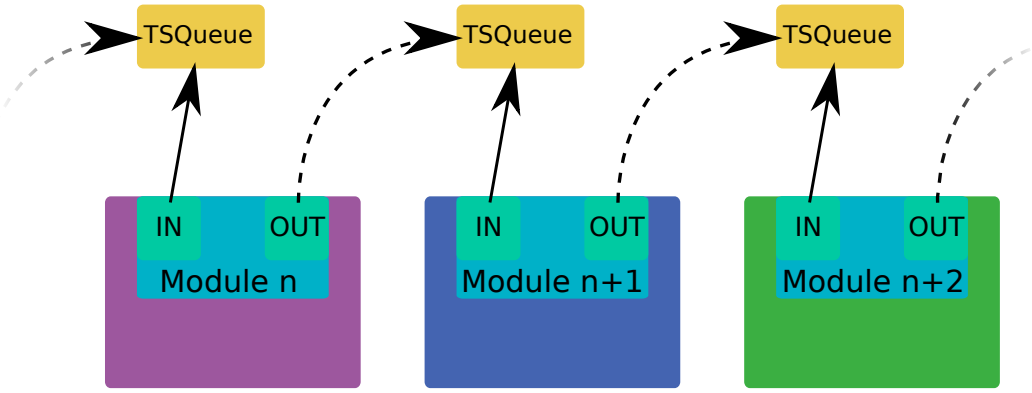

Figure B.1: Standardized module interface allows for easy binding of outputs to existing inputs.

programmer to set maximum capacity of the queue and can be used in two different modes — blocking and non-blocking.

B.1.2 Module interface

To be able to include modules in our pipeline and facilitate communication between them, we need to define a common interface that will be supported by all modules. This interface is represented by the abstract **Module** class.

As illustrated in fig. B.1, every module inherits from this class and therefore contains two shared pointers to **TSQueue**. During construction every module automatically allocates its own input queue. These queues are then bound to the outputs of respective preceding modules during construction of the **Pipeline**.

Apart from having input and output, every module has to override pure virtual methods `start` and `stop` to offer a standardized way to start and stop processing that can be used by the pipeline.

B.1.3 Building the pipeline

The most straightforward way to implement a pipeline, where every stage is derived from certain class, is using some kind of collection (e. g. **std::vector**) of pointers to the base class. In our case, such approach would be problematic since our stages or not derived from a single class but rather from one instance of a class template. This means that we have no single common base class and hence no suitable type for our pointers. One possible solution might be to create another class — lets call it **ModuleBase** — that would serve as a

common ancestor to all instances of **Module** template. We would then be able to use **ModuleBase** for our stage pointers.

While this approach does solve the above mentioned problem, it also creates a new one. **ModuleBase** cannot possibly include all the required members — for example input and output queues of different types — and we would therefore have to dynamically cast these pointers at runtime to one of the derived types (**Module** with appropriate input and output type). It is however impossible to guess what type we shall cast it to which makes the casting impossible and the entire approach unusable.

We have therefore decided to rely on templates instead and make the pipeline generic. It would therefore be able to contain stages of any type (not necessarily derived from **Module**) that contain all the required members. We have, however, decided to restrict this only to classes derived from **Module** to keep the semantics as originally intended — **Pipeline** should be filled with **Module**s and not arbitrary types that coincidentally happen to have identically named members.

Another advantage of using templates is the fact that module compatibility — matching input and output types and proper inheritance from **Module** — is checked at compile time. While we lose the possibility of having dynamic pipeline that can be change during runtime, we think that the reliability is more important in our case and compile time checks mean no unwelcome surprises during runtime.

B.1.4 Running the pipeline

When the **Pipeline** has been built, it is ready to start running. As soon as the `run` method is called, all the modules are started by the pipeline using their `start` methods. The starting is done in reverse order — from the last module to the first — to make sure that earlier modules will only start producing data once all the subsequent modules were started. This guarantees that any time consuming initialization done in the `start` method will not result in dropped frames.

The stopping is done in a similar manner except for the direction. Using the its `stop` method, every module is terminated from the first one to the last one. This will prevent earlier modules from filling up the queue when the next module has already terminated. Contrary to `run`, which is non-blocking, `stop` will usually have to wait before worker thread of each module is joined. Due to that, since we use signals to notify the pipeline to terminate, the stopping should be done in a new thread because signal handler might be executed by any thread (including worker thread from one of the modules) of the process and could result in a deadlock.

B.2 Modules

So far we have only implemented four modules required for basic functionality. All of them are based on the same principle and their workflow looks quite similar. They take an input, process it and send the result further. Because of this, the main structure of all the modules is fairly similar.

To be able to work concurrently, our modules all contain a single worker thread and a single volatile variable used to signal termination. Once the thread is started by the `start` method, it processes data in a loop until it is notified through the variable to terminate and has processed everything in its input queue.

Even though it is not exactly necessary, our existing modules mostly use opaque pointer to hide members used only by the internal implementation. Mainly so because some modules use CUDA and hiding everything related to it behind an opaque pointer avoids inclusion of CUDA headers into everything that uses these modules.

B.2.1 Video input

Our **VideoInput** class takes care of stream demultiplexing and video decoding. As both tasks are rather laborious, we rely quite heavily on third-party software — namely libraries from FFmpeg for demultiplexing and NVDEC for video decoding. Since these libraries are only required in this module we try to hide everything regarding them behind and opaque pointer to keep everything that uses this module clean.

VideoInput is a special case of module that has no input (it is derived from **Module** with input type **void**). Therefore, instead of processing data from the input queue, it has to take its input data from somewhere else. In this case either from a local file or from an online stream.

Initialization

Before we start the processing, we have to initialize all the required libraries. We do most of the initialization in the constructor to avoid further delay when the pipeline is being started. We initialize CUDA, prepare CUDA context and initialize libavformat. We however do not connect to the stream yet. It would not be a problem in case of offline videos but for online videos it might lead to unnecessary buffering of the input before starting the pipeline, which is something we would like to avoid.

Once the pipeline is started, we can connect to the stream and get basic information about it. As soon as we get that information, we send an empty

frame containing this information further into the pipeline to start initialization of other modules that might depend on this information (in our case especially the SSD which requires width and height of the processed video). After passing the information to other modules, we initialize the rest of our our input module and we are ready to start the processing.

Processing

The main processing loop of the input module first reads a frame from the input source defined be user (either a local file or an online stream), discards everything in the input container except for the video stream and applies bitstream filters if necessary (based on the container format and codec). This can be all be done using the FFmpeg libraries, which saves us from writing a great amount of code to support various container formats and codecs.

After obtaining the encoded frames from the input stream, we are ready to start decoding them. The decoding is in our case done using the dedicated hardware on NVIDIA GPUs which is exposed through the NVDEC API. However, using the decoding hardware directly can be quite laborious and since the majority of programs uses it in a very similar way, the API offers a wrapper class that partially automates this process.

The **CUvideoparser** class is a push parser of the encoded video data. It allows the programmer to control the important parts of the workflow using callbacks and automates the rest as illustrated in fig. B.2. When using this class, we only have to define three callbacks and the rest was already done for us.

The first required function is `parserSequenceCallback`. It is called everytime the parser encounters a new "sequence" of frames (i. e. video with different properties). It therefore has to initialize a **CUvideodecoder** for the current video and prepare it for decoding. So far, we have only implemented the first initialization since our pipeline only supports a single source anyway. Any reinitialization attempt will therefore throw an exception.

Once the parser has received enough encoded data for a single frame `parserDecodeCallback` is called. Its purpose is to start the asynchronous decoding of the gathered frame data on the dedicated GPU hardware.

Lastly, once a decoded frame is ready, `parserDisplayCallback` is called. This function, in our case, copies the decoded frame from the decoder's output surface to a new dynamically allocated memory and enqueues the corresponding **Frame** object to the output queue.

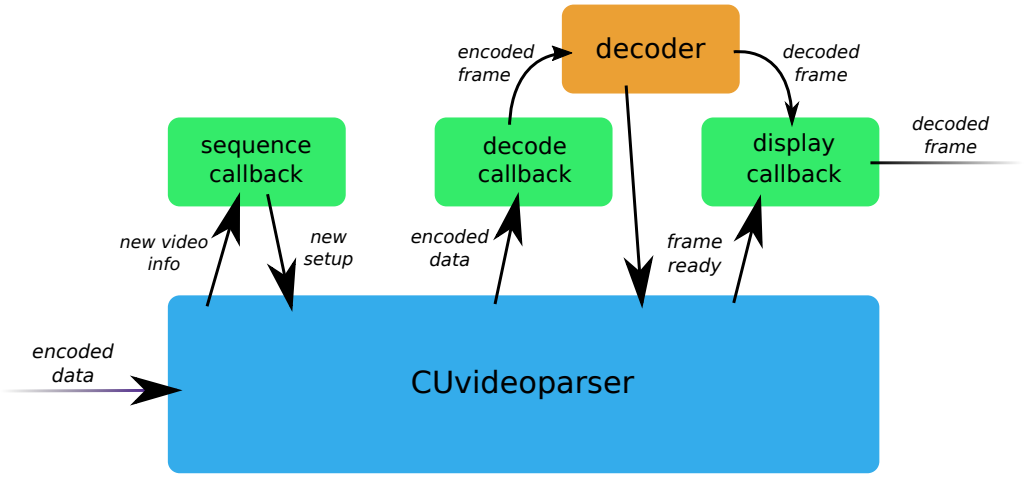

Figure B.2: CUvideoparser coordinates video input processing and lets the programmer implement only the most important parts through callbacks.

Termination

Since this module is the first one inside the pipeline and controls the flow of data into the pipeline, it should be able to somehow notify the pipeline that the end of the input was reached and the pipeline should terminate. Our pipeline already supports notifications from the outside world via signals and we have therefore decided to use the same way of communication for this module. Upon reaching the end of input, we raise a `SIGINT` and hence notify our pipeline to start the process of termination.

B.2.2 Color model conversion

ColorSpaceConvertor is the simplest module in our pipeline. Its sole purpose is to convert frames from the YUV color model and various pixel formats produced by the decoder to the packed BGR format used by our SSD. Some of the corresponding reverse conversions might be useful in the future if we, for example, decide to encode crops stored in the database for faster export. But since the current version does not require them and we do not expect them to be necessary in the near future, we have decided not to implement them.

Current implementation relies on two templated conversion functions — one for planar (`yuv444ToRgb`) and one for semi-planar (`semi420ToRgb`) formats — that use a common pixel conversion function. The pixel conversion function (`pixelYuvToRgb`) performs a simple matrix multiplication which calculates corresponding RGB values from the YUV ones and clamps the

result if it happens to be out of range.

Because performance is critical for our application, we execute all the conversions in parallel on the GPU. For the subsampled semi-planar formats we use an execution configuration where each 32×2 block of threads converts a 64×4 block of pixels and for the planar formats without subsampling a configuration where each 32×2 block of threads converts a block of pixels of the same size. This way we should be able to achieve slightly more efficient memory reads and writes (thanks to memory coalescing) and hence keep the performance high enough.

B.2.3 Object detection

The most important part of our pipeline is undoubtedly the object detection. The single shot detector we use — descibed in detail in thesis by Dobranský [11] — is implemented in Python. The functionality of this module therefore mostly revolves around the Python interpreter and passing data between C++ and Python.

Since the required functionality is quite limited, we have decided to avoid another dependency and instead of using **Boost.Python** implement the interpreter embedding ourselves.

Initialization

Before using the module, we have to initialize the interpreter and prepare the required Python functions. We try to do as much initialization as possible in the constructor but since certain parts require information about the stream, we have to resort to lazy initialization for those. The most significant action requiring video metadata is the initialization of the SSD itself, which can take up to 3-4 seconds (hence the empty frame sent by the video input immediately after connecting to the stream mentioned in appendix B.2.1).

Processing

As the frames are coming in we append them to a batch and as soon as the batch reaches the specified size we start the inference. The `inference` function returns a tuple containing all the detection data. The result is then sent further into the pipeline to be exported to the database.

The only remaining problem is how to efficiently move frame data to the SSD. The SSD is implemented in PyTorch and therefore expects a PyTorch **Tensor** as an input. As far we know, however, there is no way to create a **Tensor** from already existing GPU data (in a way that the **Tensor** object

would ”adopt” the existing allocated memory as its storage). The most efficient solution we have come up with is to allocate an empty GPU **Tensor** and copy frame data into it. Furthermore, if we want to retain the capability to export crops in the export stage, it would still be necessary to copy the frame data somewhere else for later use anyway and this solution therefore seems to be ideal.

B.2.4 Data export

As discussed in section 3.3.4, the main focus of our implementation is to minimize the network communication, eliminate dependencies of subsequent queries on previous ones and minimize the influence of network overhead on the performance.

Initialization

The only initialization required for this module is connecting to the database. This can be done already in the constructor to avoid unnecessary delay after starting the stage.

When the the first frame of a certain camera is received, we insert a record for that camera into the database. Since camera insertion only happens once per video stream (and in the current implementation therefore once per application runtime), we have decided to keep our solution simple and execute this query synchronously as it should not have a noticeable effect on the overall performance.

Processing

To minimize the influence of network overhead we implement the communication with the database in an asynchronous manner. This way, we can already be preparing next batch of data while the query for current batch is being executed. In and ideal case, we would be able to send data to the database as soon as possible. This is, however, not possible since `libpq` only allows one asynchronous query to be executed at a time on a single connection and requires its result to be received before executing another one. But despite this limitation, asynchronous query execution still brings performance improvements. In our implementation this asynchronous query is implemented by the **Query** class. As soon as we start execution of one such query, we create a new **Query** object and start preparing data for it while the first query is still running.