UNIVERSITY OF OSLO
Department of Informatics

Efficient implementation and processing of a real-time panorama video pipeline with emphasis on background subtraction

Master's thesis

Marius Tennøe

Efficient implementation and processing of a real-time panorama video pipeline with emphasis on background subtraction

Marius Tennøe

Abstract

The Bagadus system has been introduced as an automated soccer analysis tool, and consists of an analysis subsystem, tracking subsystem and video subsystem. By automating the integration of these subsystems, Bagadus allows for simplified soccer analysis, with the goal of improving athletes' performance. The system is currently installed at Alfheim stadium in Tromsø, Norway. A part of the video subsystem is the generation of panorama videos from four HD cameras. However, the pipeline for panorama video generation in the first version of the system did not manage to do this online and in real-time.

In this thesis, we present how to build an improved panorama stitcher pipeline that is able to stitch video from four HD cameras into a panorama video online and in real-time. We describe in detail the architecture and modules of this pipeline, and analyze the performance, where we demonstrate real-time, live capture, processing and storage of four individual camera feeds and generation of a panorama video on a single machine. In addition, we focus on how background subtraction can be used to improve the pipeline. As part of this, we discuss how we can utilize player position data to improve the background subtraction process, and also discuss in detail how to optimize the background subtraction process on CPU and GPU.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors Pål Halvorsen, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam and Carsten Griwodz, who have been great support, providing feedback, discussions, guidance and advice for the development of the Bagadus system and this thesis. In addition, Kai-Even Nilsen has been a great help when installing and upgrading the system at Alfheim.

Furthermore, I would like to thank and acknowledge the work done by Espen Oldeide Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad and Simen Sægrov, of whom I have been working closely with to build and improve the Bagadus system. They have all been great for discussions, feedback, advice and implementions, while making the work on this thesis really enjoyable.

Finally, I wish to thank my family, girlfriend and friends for their support.

Oslo, April 24, 2013
Marius Tennøe

Chapter 1

Introduction

1.1 Background

Today, many large sports clubs use a lot of resources for analyzing and improving the performance of their players. This kind of analysis is done either manually or by use of automatic systems. The goal is to improve player performance, strategies and planning in the most effective way. Soccer is a sport where such analysis systems are important, and examples of existing systems are Interplay [4], ProZone [5], STATS SportVU Tracking Technology [6] and Camargus [7]. These systems provide data like player speed, heart rate, fatigue, fitness graphs, etc.

Such systems all contain several subsystems, such as video subsystems and event annotation subsystems, but these all require manual steps to successfully integrate with each other. For instance, in Interplay, video streams are manually analyzed by trained operators that mark events, such as goals, offsides, and penalties. In comparison, SportsVU uses cameras to automatically locate players, which is then used for analysis. However, using video for automatic player localization requires lots of resources and is inaccurate. Another way to locate players at all times would for instance be to use a sensor-based system, like the tracking system by ZXY Sport Tracking [8] (ZXY), where players' location, speed, heart rate, etc. are sampled several times per second during matches or training sessions, by use of antennas and sensor belts on the players.

A common subsystem and tool in such analysis systems is video, which allows coaches to replay important events. The videos can then be shown to the relevant players, which in turn can see the situations themselves to understand what needs to be improved. There are several solutions for integrating video, such as having dedicated camera personel per player tracking him/her during a match. This is expensive, however, both in respect to equipment, processing, and human resources, and is not very accurate. A solution becoming more and more common today, is the use of several cameras to record everything that happens in the field concurrently, meaning all possible events get recorded. This makes it easier to retrieve information from the footage, and allows for creation of stitched panorama videos of the whole field. However, the creation of such panorama videos requires a lot of processing power. Camargus is a good example of such a system, where they use 16 cameras to capture the whole field, and provide a stitched panorama video of matches. Nevertheless, Camargus does not

directly integrate with an annotational system for tagging events.

As stated, existing systems contain many manual steps for integrating the different subsystems used for analysis. To address these shortcomings, we present Bagadus [2, 9,10]. Bagadus is a system that targets to automate all of these steps, and therefore integrates a camera array for video capture, a sensor system for retrieving player statistics, and a system for human expert annotations. System events can both be tagged by an expert, or automatically tagged by analyzing data from the sensor system. Either way, this allows users to playback events automatically. Furthermore, by use of the sensor subsystem, Bagadus knows the positions of the players at all times, which allows for accurate video tracking of specific players. When viewing video footage, users are able to switch between the different cameras, in addition to viewing a stitched panorama video. The generation of this panorama video is supposed to be done in real-time and online, but the current Bagadus implementation [2, 9, 10] does not contain an optimal stitcher pipeline for fulfilling these requirements, and the resulting panorama video contains several visual artifacts.

1.2 Problem Definition

A goal is to increase the performance of the Bagadus panorama stitching pipeline. There exist a lot of work done on panorama stitching, such as [11–15]. Another good example is Camargus [7], which we mentioned above. However, there are issues with these systems that make them unfitting for our needs, such as the use of expensive and specialized equipment, reduced visual quality, closed and/or commercial source, and lacking real-time performance.

In this thesis, we investigate how we can improve the old Bagadus panorama stitcher pipeline, both in performance and visual results. For the visual improvements part, we will emphasize the use of background subtraction. To improve the stitcher performance, we research how the existing architecture can be restructured, and how we can split the task of image stitching into several sub modules running in a pipelined fashion. As part of this, we will also investigate the possibilities of boosting the performance by use of heterogeneous processing architectures for massive parallelism. The goal is to create a pipeline for stitching frames from four cameras online and in real-time, while it at the same time processes and stores the four individual streams. Furthermore, to improve the visual results of the panorama stitcher pipeline, we will research adding new modules and algorithms. This includes investigating the need for implementing new algorithms and architecture changes. The end result should be a subjectively better looking panorama video consisting of fewer visual artifacts. To further improve the visual quality, we will emphasize the use of background subtraction as a tool in the pipeline (while others emphasize other parts), and must therefore investigate background subtraction in detail to determine the usability in this scenario. We must thus look into different background subtraction algorithms, important aspects and parameters, general performance optimizations, performance on different architectures, and the possibilities of utilizing the knowledge about player positions to improve accuracy and performance.

1.3 Limitations

The selection of algorithms for stitching images to a large panorama has already been discussed in [2, 9, 10]. We will therefore not investigate other algorithms, but use the one selected here, i.e., investigate means for improving performance. This also means that we will not go into the details on how the stitching algorithms work, because that is beyond the scope of this thesis. However, this does not limit the possibilities of researching how we can modify the existing implementation to make it both faster and more visually pleasing.

1.4 Research Method

In this thesis, we design, implement and evaluate a prototype for the improved panorama stitcher pipeline of the Bagadus system. The prototype is deployed in a real life scenario at Alfheim stadium in Tromsø, where the actual users are able to interact with it. The research method utilized is based on the *Design* methodology described by the ACM Task Force on the Core of Computer Science [16].

1.5 Main Contributions

The main contribution of this thesis has been to install the new and improved panorama stitcher pipeline as part of the Bagadus system at Alfheim stadium in Tromsø. This includes installing a web interface for scheduling recordings. The new and improved pipeline performs fast enough to fulfill the real-time requirements needed for the system, and stores both non-stitched and stitched footage. All of this is done on a single, inexpensive computer with commodity hardware. In addition to an increase in performance, we have also improved the visual quality of the panorama. A part we especially focus on is how we can use background subtraction to improve the panorama. However, to further improve the visual results, we need to change to a more optimal camera setup.

By improving this pipeline, we have shown how it is possible to design a pipeline for processing large amounts of video to generate a video panorama, all of this in real-time, by use of external processing units, such as GPUs.

In addition, we have been able to submit and publish a poster at the GPU Technology Conference 2013, which described how it is possible to build a pipeline for creating panorama videos in real-time using GPUs [17]. We have also submitted a paper to ACM Multimedia 2013 [18], where the pipeline is presented.

1.6 Outline

In the remainder of this thesis, we continue in Chapter 2 by describing the existing Bagadus system in more detail. This means looking at the goals of the system, the different subsystems and their tasks, limitations, and improvements for the existing implementation. We especially look into how the old, off-line panorama video stitcher

pipeline works. Before looking deeper into the improvements of the old Bagadus stitching pipeline, we will explain Nvidia CUDA, a framework for utilizing the power of GPUs for parallel processing tasks, in Chapter 3. Then, in Chapter 4, we will describe in detail the new and improved Bagadus panorama stitcher pipeline. This includes describing the goals, the general architecture, the different modules and components, and design choices of the new pipeline. In addition, we investigate the performance, scalability, and also the web interface for scheduling new recordings. Following this, in Chapter 5, we start investigating background subtraction, and how this image analysis tool is implemented. This includes comparison of different algorithms, optimization techniques both on CPU and GPU, and how we can modify the background subtractor to utilize the knowledge of player positions, which are provided by the use of sensor data. Furthermore, we investigate different applications for background subtraction in the Bagadus system, such as depth map calculation and dynamic seam stitching. Finally, in Chapter 6, we summarize the findings in this thesis, draw conclusions on our results, and discuss some future works.

Chapter 2

Bagadus

In this chapter we start by discussing the basic idea behind the Bagadus system. We continue by discussing the important subsystems, such as the video capture part of the video subsystem, the analytics system, the tracking subsystem, and the first panorama stitcher prototype. Finally we discuss how all of these systems are integrated in the demo player created for the Bagadus system, and highlight what needs to be improved.

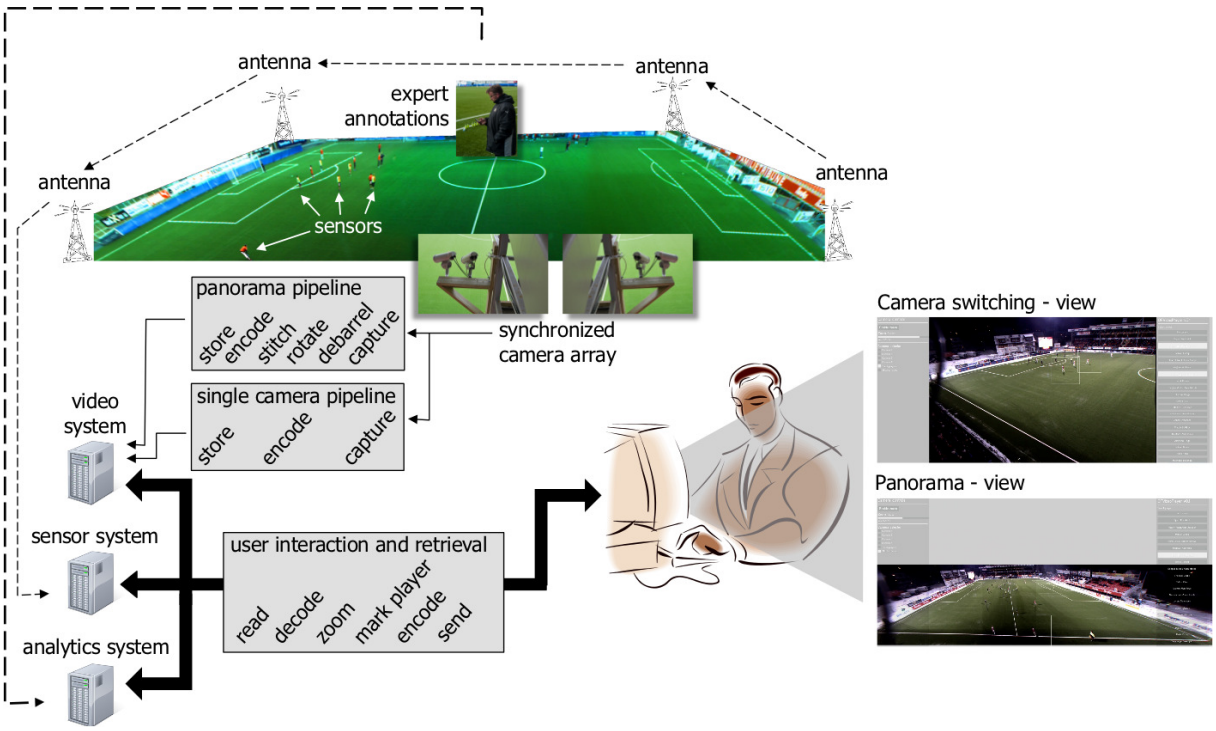


Figure 2.1: Bagadus architecture

2.1 The basic idea

As mentioned in section 1.1 and discussed in [2,9,10] (from now on referred to as the old Bagadus version), current soccer analysis systems, like Camargus and SportVU,

usually consist of several subsystems, such as a video subsystem for recording video footage from matches and training sessions, and annotation subsystem for marking and describing events. The problem with these systems is that they contain manual steps for integrating the subsystems and components to one large system. In addition to being more error prone than automation, manual labor leads to processing times so high that it is not possible to provide output from the system during half-times, which limits the usability of the system.

The basic idea of the Bagadus system is therefore to integrate the subsystems and components needed in such a soccer analysis system, and automate the process of integration between them. To be able to automate all of this, Bagadus contains three main subsystems: the analytical system, which is responsible for tagging and storing events; the tracking subsystem, which is responsible for tracking player positions and storing player data and statistics; and the video subsystem, which records, processes and stores video footage from the whole field. The general Bagadus architecture can be seen in figure 2.1. Here we can see the video subsystem consisting of several video cameras covering the whole field, plus pipelines for storing stitched and non-stitched videos. The tracking subsystem can be seen as antennas around the field that collect player data from sensor belts the players are wearing. The analytical subsystem with annotations can be seen where the coach is using his mobile device to mark events during a session.

One of the goals of the Bagadus system is to provide the coaches with processed footage so fast that they can access it and play it back during the break between periods. This means that the end-to-end delay of the system needs to be as small as possible, so that the coaches can view all the footage from the previous period and provide detailed feedback to the players before the next period.

2.2 Video capture

An important part of the video subsystem is the recording of frames. Without video footage, we would not be able to provide the viewer with video that corresponds to events and tracking data, which makes the system way less useful, and not providing anything new.

2.2.1 Camera setup

One of the goals in the Bagadus system has been to be able to use relatively inexpensive and common hardware, especially excluding any expensive special purpose hardware. This is reflected in the hardware setup, including on the camera side. The cameras used are four Basler acA1300 - 30gc [19] industrial Ethernet-based cameras, with 1/3-inch imageing sensors supporting 30 fps and a max resolution of 1294×964 pixels.

The cameras output videos in the YUV color space, using the YUV 4:2:2 pixel format. In YUV, Y is the luminance and U and V are the chroma/color components. More precisely, U is the difference $Blue - Y$ and V is difference $Red - Y$. Humans are more sensitive to differences in luminance, so the color components can be compressed by subsampling. In YUV 4:2:2, the sample rate of the two chroma components are halved

in the horizontal dimension, reducing the bandwidth requirements by 1/3. In YUV 4:2:0, the sample rate of the chroma components are halved in both the horizontal and vertical directions, leading to a reduction in bandwidth requirements of 50%. Figure 2.2 shows an example of YUV 4:2:0. More information about YUV and chroma subsampling can be found in [1] and [20].

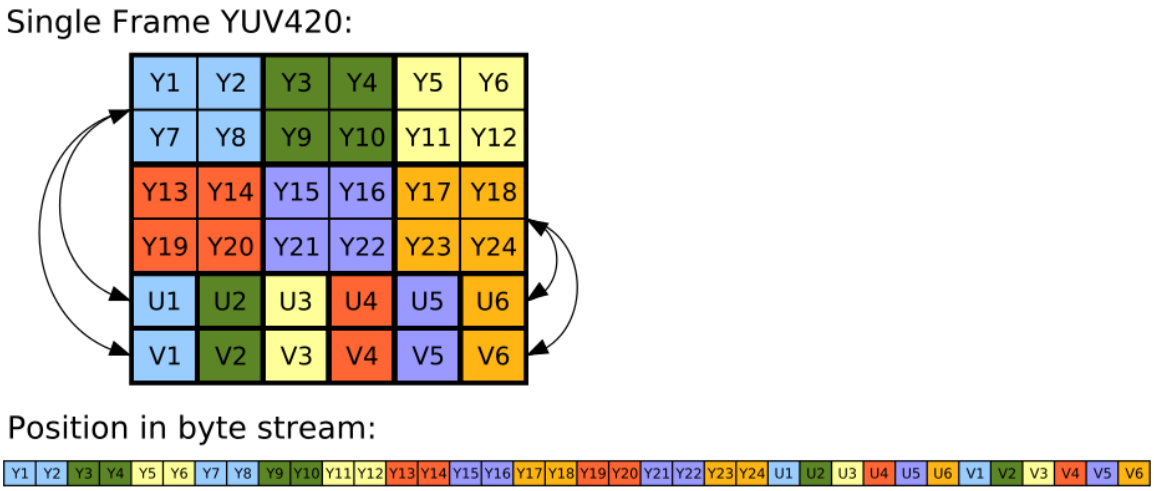


Figure 2.2: Example of YUV 4:2:0 [1]

The cameras are mounted with Kowa 3.5 mm wide angle lenses, and were connected to two computers, i.e. two cameras per computer. Due to the wide angle lenses, which gives the cameras a field-of-view of about 68 degrees, we are able to cover the complete field with these four cameras. The setup can be seen in figure 2.3.

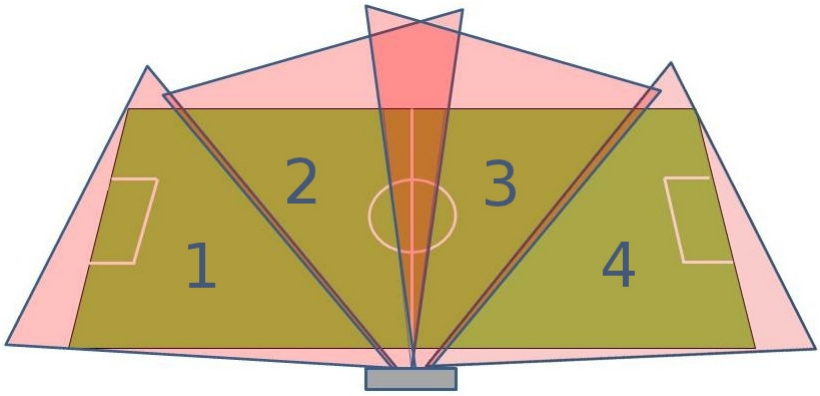


Figure 2.3: Camera setup at Alfheim stadium.

2.2.2 Frame synchronization

An important step when recording frames is the synchronization of frames between the recording cameras. We want to stitch the images, so it is therefore important that all the

corresponding frames are recorded at the same time. In the original implementation, several machines were used to record, and these machines were not connected to the internet. A trigger box created by Simula were used to trigger the camera shutters at the correct frequencies. The trigger box schematics and firmware can be found at [21]. The use of several machines and no NTP connection made synchronization between frames from different cameras difficult. The camera-synchronization was in this case secured by use of TimeCodeServer, a server passing messages between the machines to synchronize the cameras.

2.3 Analytics subsystem

One goal of the Bagadus system is to allow coaches to tag events during matches or training sessions, and then be able to retrieve these events later to review and analyze them. This is done by integrating Bagadus with the Muithu system [22]. Muithu is a lightweight, non invasive and mobile system for notational analysis. During a match or training session, coaches use a mobile phone with Windows Phone 7.5 and a specially designed application for marking events. The application contains several sets of tiles, where the user interacts with the tiles in a drag-and-drop fashion. This is considered fast and intuitive, and can be configured with different input tiles and hierarchies. The root level contains an overview of the players, like in figure 2.4(a). The second level contains a set of tiles for different events the players can be part of, such as scoring a goal. Here the user drags a player onto an event to mark it in the system, which stores the event in a database for later retrieval. Figure 2.4(b) shows an example of this.

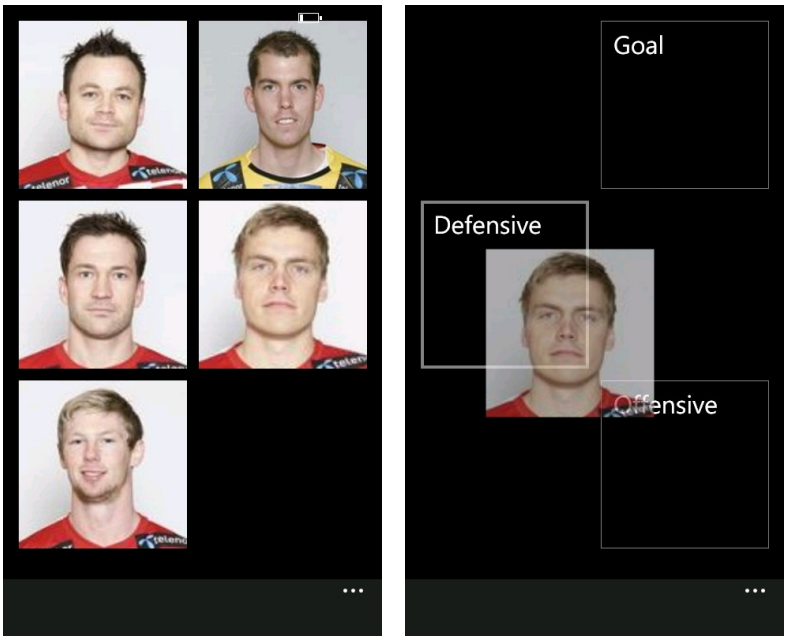
An important aspect of using such an event system, is to synchronize events with the corresponding recorded video frames. The accuracy needed is not as high as for the synchronization between frames and ZXY data samples, but the requirement is still there. This level of synchronization for events and frames can be ensured by connecting to a common NTP server.

2.4 Tracking subsystem

The tracking subsystem is responsible for tracking players by use of a sensor network, and to be able to provide player positions as pixel coordinates in the recorded videos.

2.4.1 ZXY sensor system

ZXY Sports Tracking [8] (ZXY) is a company that delivers sports tracking solutions to sports clubs and others. ZXYs system uses wireless radio technologies over the 2.45 GHz and 5.2 GHz bands, with several antennas installed on stadiums using the system. The players then wear a ZXY sensor chip that registers data such as position, speed, heart rate, etc. All of this data is sent to and stored in a relational database. ZXY reports a sampling rate of up to 40 Hz, with an estimated error margin of ±0.5 meters [23] on the newest sensors. There is a version of the system currently installed



(a) Selecting a player.

(b) Dragging player to event type.

Figure 2.4: Muithu event tagging

at Alfheim Stadium, belonging to Tromsø IL, but this is an older version that only supports a sampling rate of up to 20 Hz, with a maximum error margin of ±1 meter.

2.4.2 Video frame - ZXY data synchronization

One very important aspect when discussing the player tracking, is the need for synchronization between every video frame recorded and the corresponding ZXY data samples. If we have a time difference and/or time drift here, we will quickly see that the tracking fails by lagging behind or being ahead of the video. In the old version of the Bagadus system, this synchronization was done by hand.

It is also important to note that, as mentioned above, the max sampling rate of the ZXY system currently installed at Alfheim is 20 HZ, while the cameras record at a rate of 30 frames per second. This means that we do not have a 1:1 mapping between ZXY data samples and frames. The relationship is 2:3, so this was solved by simply reusing the previous ZXY data sample every third frame. This has proved to be a sufficient solution.

2.4.3 Sensor coordinate to pixel mapping

Before we can properly use the ZXY coordinate data, we need to map the real-world ZXY coordinates into pixel positions. This is done by first finding a transformation matrix, which is a 3x3 matrix that describes how to translate between the ZXY and image planes, and is found by using OpenCV. When this homography has been found, we can use it to warp between these two planes. More details about this process can

be found in the old Bagadus version.

With such a mapping between pixels and sensor coordinates, the tracking subsystem allows for many scenarios. For instance it allows the viewers to digitally zoom onto players and follow these, in addition to selecting cameras automatically if a tracked player moves out of one camera and into another.

2.5 First stitching pipeline prototype

One of the goals of the Bagadus system is to provide a stitched panorama video to the viewers, consisting of a panorama generated with the footage from all four cameras. To be able to deliver this, Bagadus needs a pipeline as part of the video subsystem for stitching the recorded frames into a single panorama video. The reason for building a pipeline, is that this pattern of chained tasks passing data to the next task, fits the stitcher pattern of several distinct, consecutive steps well. The stitching pipeline of the first Bagadus prototype is described in this section.

2.5.1 Important libraries

There exist many free image processing libraries and toolkits that can be utilized, and in the Bagadus system, we utilize several libraries to make implementation easier and faster.

OpenCV

In the old Bagadus version, and in this thesis, OpenCV is used to solve several of the tasks at hand. OpenCV [24] is an open source computer vision library, released under BSD license, supporting Windows, Linux, Mac OS, iOS and Android. It focuses on real-time applications, and is implemented in C and C++. OpenCV contains a lot of modules and functionality for computer vision tasks, such as stitching, warping, image representations and viewing.

NorthLight

Another image processing library used is the NorthLight library. NorthLight is developed by the Verdione project at Simula Research Laboratory [25]. It is a library that aims at being a common interface between the most popular open source image processing libraries. The Verdione project has high performance and real-time requirements, so its requirements align well with Bagadus'. We therefore utilize much of the functionality implemented in this project, for instance by using NorthLight's VideoFrame-objects to represent frames, using it to access Baslers camera SDK [26], using it to encode video with x264 [27], and to convert between image formats with ffmpeg [28].

2.5.2 Pipeline steps

As part of the video subsystem, in addition to viewing single camera videos, we want coaches, players, etc. to be able to view a stitched video panorama that combines all the cameras as a single, large video. To be able to do this, a panorama stitcher pipeline was created in the old Bagadus version. Here, different algorithms and implementations for doing this are discussed, with advantages and disadvantages. In the end, the pipeline seen in figure 2.5, was described. We will now describe this pipeline shortly.

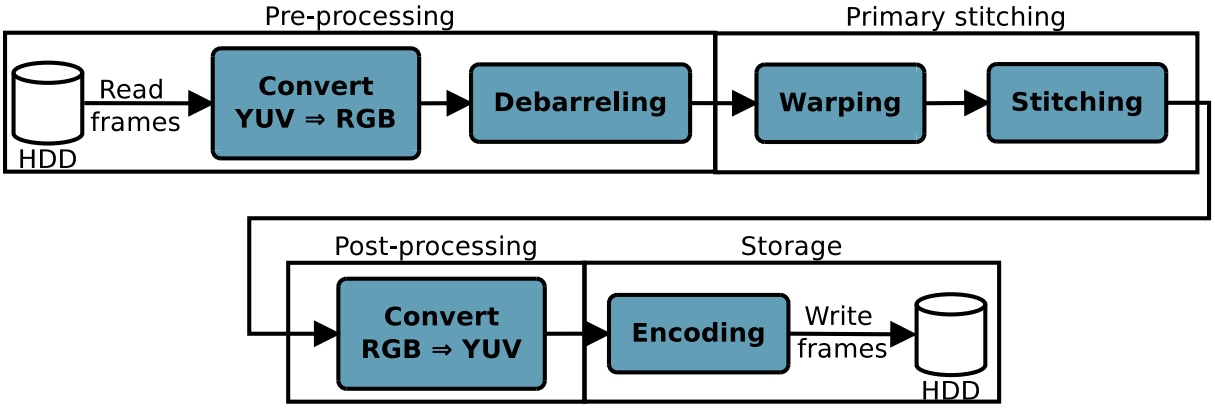


Figure 2.5: The old Bagadus stitching pipeline

Reading and first conversion steps

The frames from the cameras were recorded and stored as raw YUV frames. The first step in this pipeline is therefore to read these files from disk. Operations such as debarelling are color space agnostic, so we could use YUV internally in the whole pipeline, but it was rather decided that the internal pixel representation format should be RGB, due to RGB being somewhat easier to understand and work with. The following step is therefore to convert from YUV to RGB.

Debarreling step

When recording frames with a wide angle lens, the recorded frames suffer from barrel distortion. In an image suffering from barrel distortion, the image magnification decreases when moving away from the optical center of the image, leading to the distortion pattern we see in figure 2.6. Before we can do anything with the images, this distortion must be removed, which is done in this step. There are mathematical formulas for this, and there exist many implementations. In the old Bagadus version, the debarrelling function in OpenCV is used.

To be able to debarrel the images, we need to know a set of barrel distortion coefficients for all the camera lenses, which will be parameters for the debarreling function. This calibration is done by use of a board with a chess pattern [29], using OpenCVs functions for calculating the coefficients. Even though the lenses we use are equal, they are not 100% identical, so we need to calibrate all the cameras to retrieve the debarrel coefficients for each of them. This, however, were never done. This is because,

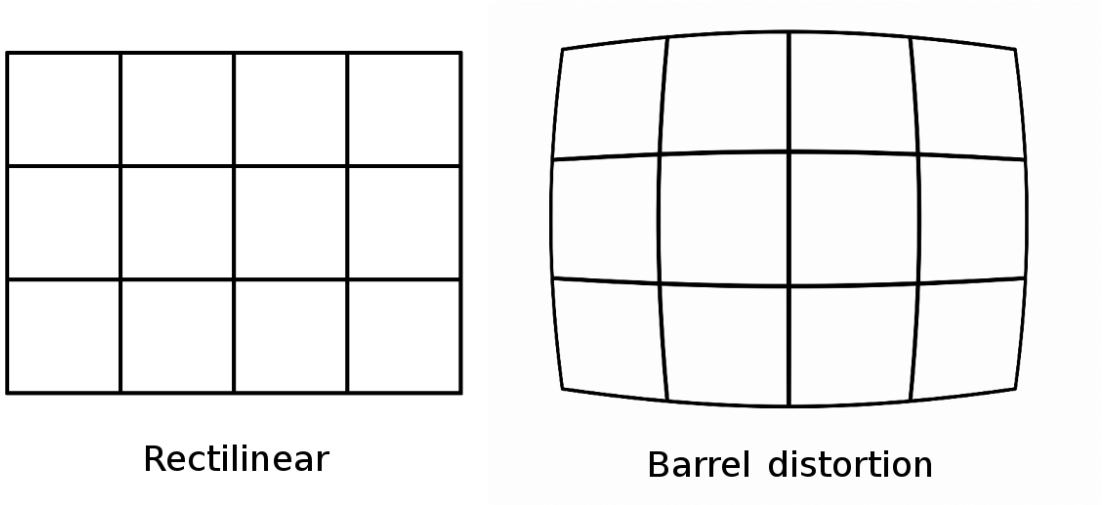


Figure 2.6: Original and rectilinear image vs. barrel distorted version [2]

even though it should be done for every camera, the results were good enough for the purpose of building this pipeline when the coefficients for only one of the cameras were retrieved. The coefficients were therefore reused on all the cameras. This is a step that should be properly followed when installing the system outside the lab. For more details about barrel distortion, see the old Bagadus version.

Warping step

To be able to stitch the four cameras, we need to do a 2D transformation of the camera frames, so that these are aligned, transformed, and ready for being stitched onto a common panorama plane. This 2D transformation based stitching algorithm (named *Homography-based stitching* in the old Bagadus version) was selected due to the good performance compared to OpenCVs auto stitchers. The transformation part of this stitching algorithm is to first select one of the cameras as the primary, or reference, projection/plane. The goal is then to transform the other cameras to fit the same plane as the reference camera. This means that for the rest of the cameras, we need to find the homography, i.e. the transformation relationship, between the current camera plane and the primary camera plane. This can be done during system setup, because the camera positions are static. To calculate such a homography, we need to find common points in the different camera-pairs, such as field corners, goal posts, lines, etc. When sent as parameters to an OpenCV function, these common positions result in a transformation matrix per camera. The transformation matrix is a 3x3 matrix that explains how each pixel should be moved to transform the frame to the target plane, i.e. of the reference camera. The transformation matrix of the reference plane is an identity matrix, in other words one that tells the warper not to move any pixels.

When these homographies have been found, we can use them to warp between the projections. This means that the warper warps all the cameras but the primary camera to fit the projection of the primary one, leading to an easy task for the stitching step itself. An important part of this warping is selecting an interpolation algorithm. Pixel interpolation is necessary every time pixels are remapped, and is caused by pixel

values being remapped to positions that are not precisely mapped to a pixel. This happens because pixel locations are discrete, with limited precision, so when pixel values are moved to pixel positions not precisely mapped, we need to evaluate the new pixel values of the nearby pixels. Interpolation therefore works by using known data, in our case pixel values, to estimate values at unknown points, i.e. at mapped pixel positions. The interpolation algorithm used in the old Bagadus version is, due to performance, nearest neighbor. A more detailed explanation about interpolation can also be found here.

The warper implementation used is based on OpenCV.

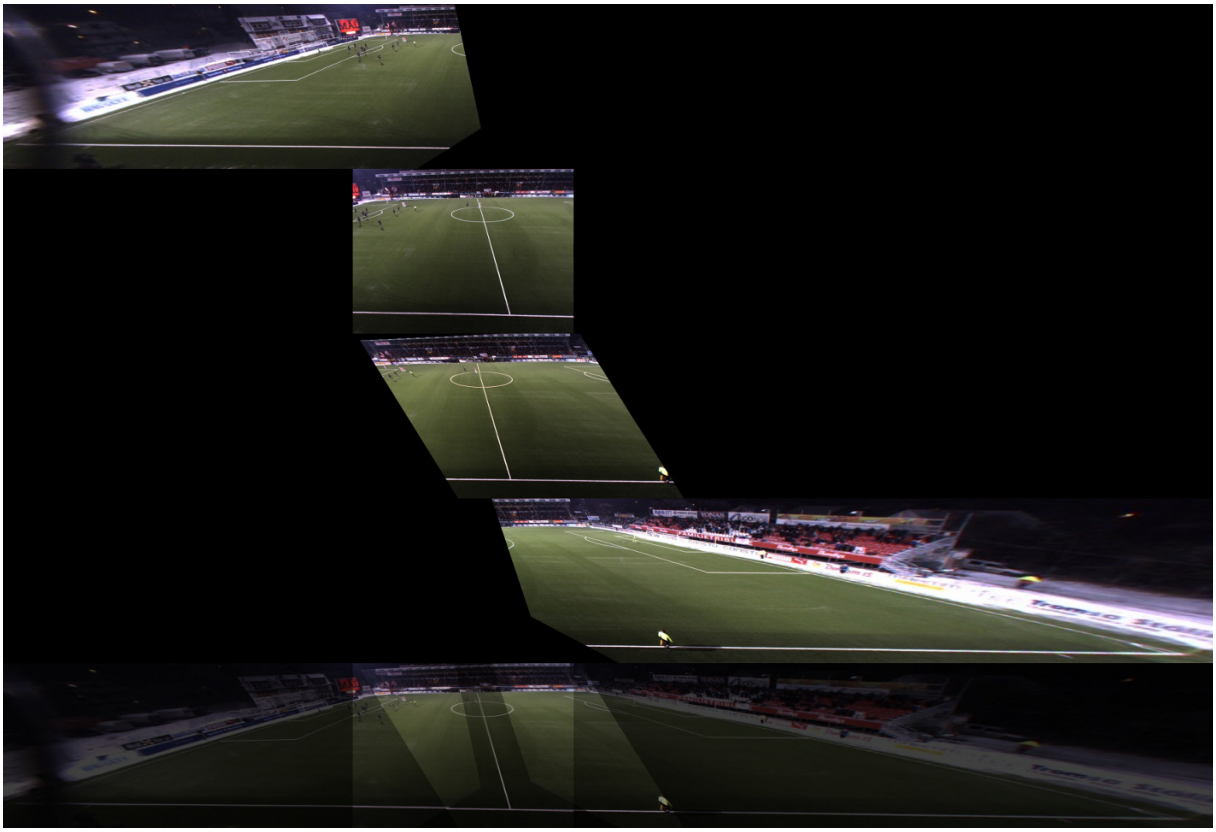


Figure 2.7: The stitching process.

Stitching step

By selecting such a transformation-based stitching algorithm, implementation of the stitching step itself is fairly straight forward. At this point, all frames have been warped to fit the projection of the reference camera, which can be considered as the common panorama plane. At the camera setup in section 2.2.1, it was made sure that there was a decent amount of overlap between the cameras. The next step in the old pipeline is then to calculate the seams in the overlapping areas between cameras 1 and 2, 2 and 3, and 3 and 4. These seams determine what camera each pixel in the panorama frame will be copied from. The seams in the the original pipeline are calculated manually by finding an offset per overlap where we can draw a straight, vertical line through the overlapping area, and these seams are completely static.

When the seams have been determined, the next step is to create the actual panorama. This is done by first creating an empty frame, large enough to contain the whole panorama. The stitcher then loops through all pixels for all the cameras, and copies the pixels between the seams to the right and left for that camera into the correct position in the panorama buffer. When all cameras have been processed, the resulting panorama is cropped to remove empty, black areas in the image. Figure 2.7 shows the process of copying the four warped camera frames into a single, large panorama frame. The highlighted areas are regions where the cameras overlap. The resulting panorama can be seen in figure 2.9.

Advantages of using static seams are that the seam calculation is basically free, and can be calculated before running the pipeline. On the downside however, the static seam is not optimal, and we are able to see clear, visual artifacts in the generated panorama image. This is especially an issue when players are crossing a seam, as we can see in figure 2.8, which results in ghosting effects.

Figure 2.8: Artifact caused by player crossing stitch seam

Second conversion step

RGB is used internally in the pipeline, and the encoder in the storage step requires YUV 4:2:0, so the next step of the pipeline therefore converts the panorama output from RGB to YUV.

Figure 2.9: Example panorama with static seams

Storage step

The last step is to store the resulting panorama frames. An important aspect of the system is the format in which we want to store the data. There are several possibilities, such as dumping raw YUV data to disk, or encoding and storing the frames as H.264. In the old Bagadus version, this is discussed in detail. When selecting a format, there is a trade-off between image quality, storage size requirements and compression ratio, encoding time, and writing time. The image quality and compression ratio should be as high as possible, while the writing time and encoding time should be as small as possible, but this proves to be difficult to achieve. The solution selected is to write the resulting video streams to disk by encoding the frames as lossless H.264 and then write them in blocks of 90 frames per file. This means that each H.264 file is no longer than 3 seconds. H.264 does not support custom metadata, so the timestamp of the first frame in each 90-frame-file is part of the filename. This allows us to search in the video streams both forwards and backwards, and also allows us to synchronize ZXY data, Muithu event data, and video frames on the player side of the Bagadus system.

2.5.3 Performance

The performance of the first prototype of the stitcher pipeline is affected by the fact that it was meant to be a proof-of-concept for integration between the different sub-systems. The performance is therefore unoptimized. When run on DevBox 1, with the specifications in table C.1, the performance numbers can be seen in table 2.1. It is quite clear that the most resource demanding operations are done in the primary stitching steps, which consists of the warping and the stitching itself, so an optimized stitcher pipeline would have to focus especially on speeding up these operations. It is obvious that this stitching pipeline is not real-time, and it therefore has to be run off-line.

	Mean time (ms)
YUV $\Rightarrow$ RGB	4.9
Debarreling	17.1
Primary stitching	974.4
RGB $\Rightarrow$ YUV	28.0
Storage	84.3
Total	1109.0

Table 2.1: Old pipeline performance

2.6 The Bagadus demo

In the old Bagadus version, to demonstrate the total integration of the video, tracking and analysis subsystems, a Bagadus player was created. The player can be seen in figure 2.10. On the left side, marked in red, we can see a list of all the players. All player names here are retrieved from the ZXY database. By pressing one or more of the player names, the application starts tracking the players. This tracking is done by drawing a square around every tracked player. In addition, when tracking players, we are able to digitally zoom onto the tracked players by enabling zooming, marked in blue. When tracking players, we can also activate automatic camera selection in the purple panel. This makes the application switch between the different camera streams, based on the camera that shows the largest number of players. In the yellow panel, we have the camera selector, which allows the user to switch between the different camera streams manually. If the user presses the button in the cyan panel, the application switches to the stitched panorama video, which of course supports tracking and zooming onto players. Marked in green, in the lower left, we have the list of events. When pressing one of them, the player jumps to the corresponding time in the video, and starts tracking the players being a part of the event. A video demonstration of the Bagadus demo can be found at [30].

Figure 2.10: The Bagadus demo player

2.7 Summary

We have in this chapter looked at how the first Bagadus prototype is structured. The primary goal is to automate the integration between different subsystems needed to create a completely automated soccer analysis system, with possibilities for other users to access this footage later. Bagadus consists of three subsystems: The video subsystem is responsible for recording and storing frames, while also generating a stitched panorama video from the recorded frames. The analytics subsystem, based on Muithu, provides possibilities for annotating events during a match or training session. The tracking system is realized by use of ZXY's sensor system, and provides us with functionality for knowing the exact positions for all players in the videos at all times.

The Bagadus demo application shows us how all of these systems are integrated to provide the experience we want to deliver. The demo is able to play both ordinary and stitched video provided by the video subsystem. The tracking subsystem allows us to track one or more players in the videos, while Muithu provides functionality for event annotation, and then lets us playback such events with the click of a button.

However, there is lot of room for improvements. First, the performance of the stitcher pipeline is far too low to allow for online and real-time panorama video creation. One of the goals is that the coaches should be able to show situations to the players during half-time, so approximately 1 fps like in the first prototype is not fast enough. In addition, the generated panorama contains lots of visual artifacts, such as color differences between the cameras and ghosting effects when players cross a static cut. The performance and the visual results are the issues we are focusing on solving in this thesis.

To speed up the performance of the stitcher pipeline, we want to use graphics processing units (GPUs). This is because GPUs are excellent for executing tasks in parallel, and image processing is generally massively parallelizeable, meaning we would potentially see a large performance increase. We therefore continue in Chapter 3 by looking into Nvidia CUDA, which we will use for the GPU implementations of several components described in Chapter 4.

Chapter 3

Nvidia CUDA

Compute Unified Device Architecture [31] (CUDA) is a platform and programming model for parallel computing, developed by Nvidia, which makes it easy to write code that runs on the massively parallel graphical processing units (GPU). With the help of CUDA, it is possible to speed up parallel applications by an order of magnitude or more. However, this is not trivial for all applications, and depends a lot on the nature of the problem, especially on how parallelizable the problem is.

CUDA is designed to let tasks be parallelized, and then to execute the parallelized version on a GPU by use of thousands of threads. This is data parallelism. In addition, CUDA allows for task parallelism, where it is possible to run several different tasks concurrently on a GPU, even though each individual task is running serialized. To be able to do this, CUDA threads are extremely lightweight, with very little overhead compared to CPU threads [32]. For the programmer, GPU execution is issued by creating and launching a *kernel* for the parallel part of the application. Kernels are then run on the GPU. Furthermore, by moving execution tasks from the CPU to the GPU we offload the CPU, which can result in CPU based tasks executing faster.

3.1 The Fermi architecture

Nvidia has several GPU architectures, and the latest architecture supported by CUDA is the Kepler architecture. However, when we started working on this system, Fermi was the newest architecture available, and is therefore the architecture we will focus on in this thesis. The Fermi architecture was launched in the spring 2010, with the GF100 chipset and Geforce 400-series GPUs. The numbers differ from one GPU architecture to the other, but the general GPU architecture and terms are somewhat similar, so the description of the Fermi architecture also explains a bit of Nvidia's GPUs in general.

The basic processing unit on the Nvidia GPUs is the **Stream Multiprocessor** (SM), marked in red in figure 3.1, and in more details in figure 3.2. On Fermi, a GPU consists of up to 16 SMs, located around a common L2 cache. Each SM contains 32 **Stream Processors** (SP), also called CUDA cores, which are responsible for executing instructions. The SPs within a SM all execute the same instruction at a time. Each SP contains a fully pipelined ALU and FPU. Furthermore, each SM contains 16 **load/store units**, which are used for calculating source and destination addresses for 16 threads per clock. Each SM also contains 4 **Special Function Units** (SFU), which are used for special operations,

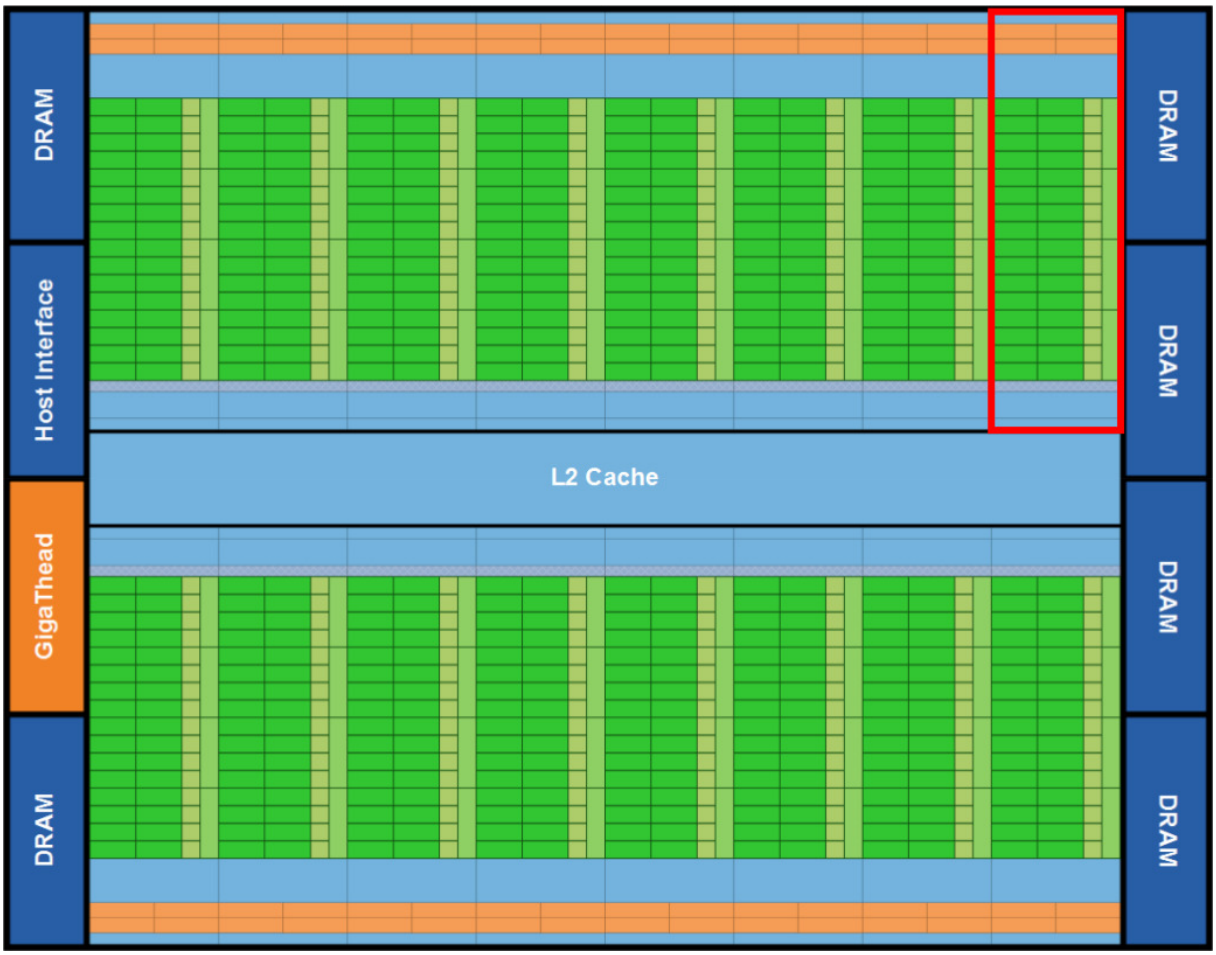


Figure 3.1: The Fermi architecture [3]
An example Stream Multiprocessor (SM) marked in red.

such as trigonometric functions. Each SM schedules groups of 32 parallel threads in what is called a **warp**. Instructions are then issued per warp. To do this, each SM contains two **warp schedulers**, which can schedule, issue and execute two warps in parallel. The Fermi architecture can, by use of very fast context switching, have up to 48 active warps per SM, which equals 1536 threads.

As we can see in [3], Fermi also contains several new features that are improvements from earlier architectures. One of these features is an improved thread scheduler that allows for running different kernels concurrently. This can greatly improve performance of an application consisting of different kernels. Fermi also introduced L1 cache for each SM and a L2 cache, which help increase performance quite a bit. Furthermore, Fermi implements a unified address space that unifies the three types of address space, namely thread private local, block shared, and local. This allows for easier pointer implementation, where one for instance does not need to know what memory space a pointer points to at compile time, and it also enables support for true C++ programs.

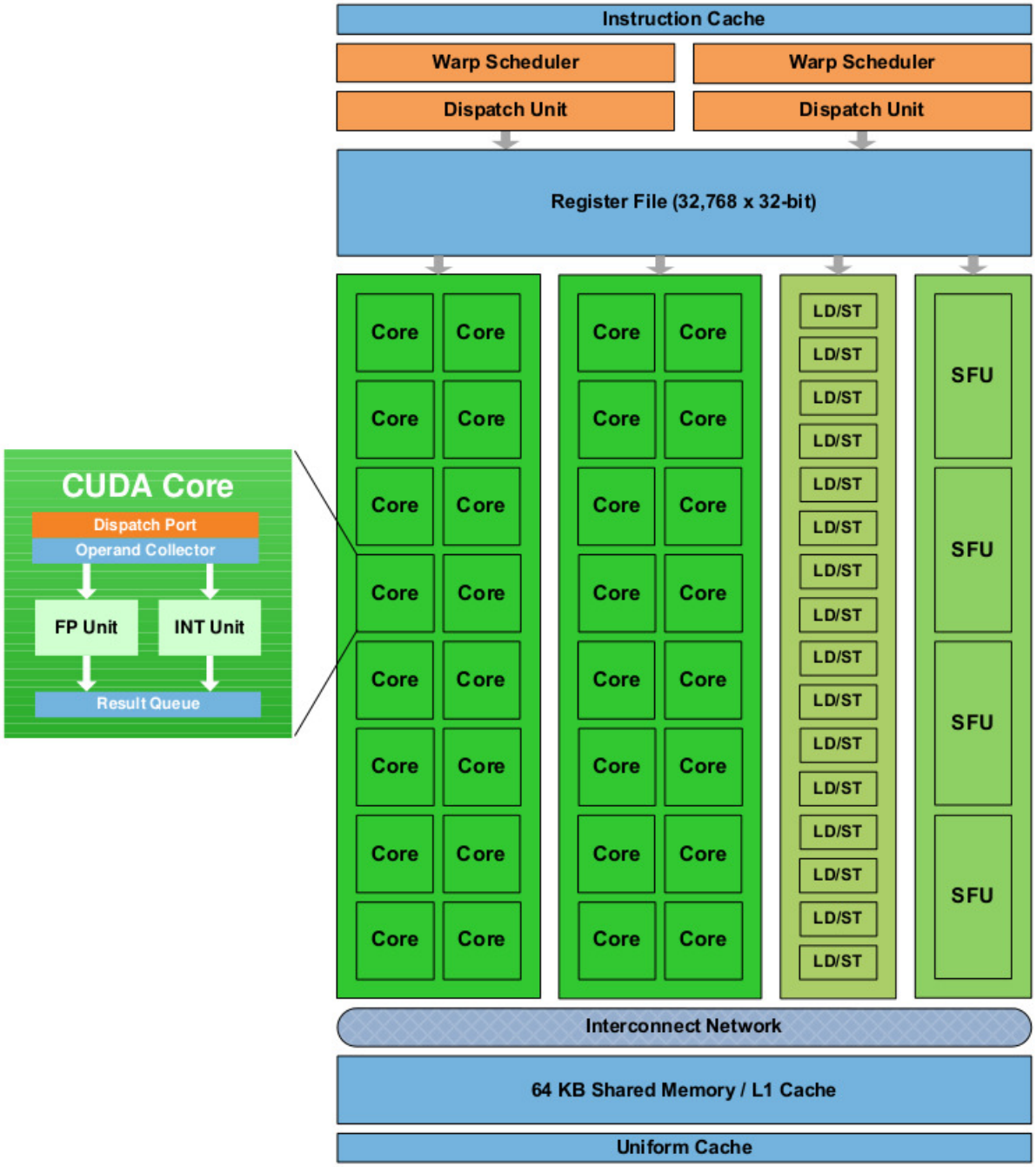


Figure 3.2: A Fermi Stream Multiprocessor [3]

3.2 The CUDA execution model

When writing CUDA applications, the programmer needs to create a kernel. Kernels are executed by a *grid* of thread blocks. A grid is just a collection of completely independent *blocks*. A block however, is a collection of threads that can communicate within the block. Therefore, when launching a kernel, one basically starts execution of concurrent and independent thread blocks. Instructions are then issued per warp.

3.3 Compute capability

The term compute capability is used to describe the capabilities of different Nvidia CUDA enabled GPUs. The existing capabilities are 1.0, 1.1, 1.2, 1.3, 2.0, 2.1, 3.0 and 3.5, where the number describes the capabilities and properties that the GPU has. The first number is the generation, while the second number equals the revision within that generation. All the different compute capabilities have different properties, but generally the higher the number, the better. Of course, compute capabilities are backwards compatible, so for instance GPUs of compute 2.0 can execute CUDA applications written for compute 1.3.

3.4 The memory model

Nvidia GPUs have several different types of memory, spread over three different address spaces, located on-chip and off-chip, designed for different kind of uses and access patterns. The memory types are host memory, global memory, constant memory, texture memory, shared memory, local memory, and registers. We can see a table summary of the different memory types in table 3.1.

Memory type	Location	Cached	Speed	Access scope	Data lifetime
Global	Off-chip	No	100x	All threads	Alloc⇒dealloc
Texture	Off-chip	Yes	1-100x	All threads	Alloc⇒dealloc
Constant	Off-chip	Yes	1-100x	All threads	Alloc⇒dealloc
Shared	On-chip	-	1x	Threads within block	Block
Registers	On-chip	-	1x	Single thread	Thread
Local	Off-chip	No	1-100x	Single thread	Thread

Table 3.1: CUDA memory types

3.4.1 Host memory

The host (CPU) memory is the main memory in the computer, which is controlled by the CPU. The access times to this memory from the device are high, and is limited by both the bandwidth of the PCI Express (PCIe) interface, which for the PCIe x16 3.0 standard is 16 GB/s in each direction [33], and the latency of commands on the PCIe bus, which in [34] was found to be approximately 10 µs. The programmer should therefore avoid transferring unnecessary data back and forth between the host and the device. In addition, this memory is not directly accessible from the GPU threads, other than when using pinned memory.

3.4.2 Global memory

Global memory is the largest memory on the GPU, located off-chip in the device DRAM (see figure 3.1), and is globally accessible by all the threads on the device, in addition to the CPU. This, however, comes at the price of access time. Due to the size of the global

memory, it is slower than other types of GPU memory. However, on newer devices of compute 2.x and higher, global memory can be cached in a L2 cache of limited size. This makes global memory more convenient for the programmer to use, compared to before, because the advantages of other cached memory types, such as texture and constant memory, are not as big as they used to, while global memory also remains easier to use. Nevertheless, it is important to not rely on GPU caches like you would for CPU caches, because there are too many threads per cache. The lifetime of data in global memory is from it is allocated in the host code until it is deallocated here.

3.4.3 Texture memory

Texture memory is another kind of global memory located off-chip, accessible by every thread. In comparison to ordinary global memory, it has several different properties. First of all, it is accessed by the threads in a read only manner, meaning the threads cannot write to the texture memory. Furthermore, texture memory is cached, which can increase performance. However, the texture memory is designed for spatial access patterns, which means that the caching is optimized for this. Texture memory also has other nice properties, such as hardware supported filtering and interpolation as part of the read process. As with global memory, data lifetime is from allocation to deallocation in the host code.

3.4.4 Constant memory

Constant memory is a limited amount of memory, located off-chip, accessible by every thread, meant to store shared constants used by the threads. The constant memory is cached, so the access to it is very fast. On the current versions of CUDA and GPU architectures, the constant memory is of size 64 KB [35]. The data lifetime is also here from allocation to deallocation in host code.

3.4.5 Shared memory

Shared memory is a limited amount of memory shared between the threads of a block. Shared memory allows threads of a thread block to cooperate by sharing a common memory space. However, this memory is only shared among the threads of that block, other blocks have their own shared memory space. Shared memory is located on-chip, and has approximately the same access speed as registers, making it very fast, and preferred for repeated accesses and writes. The data lifetime here, however, is equal to the block lifetime.

Note, however, that the programmer needs to be careful here to avoid memory bank conflicts. Basically, shared memory is split into equally sized memory modules, named banks. This means that memory accesses of n addresses that spread over n banks can be serviced in parallel. However, if more than one of these addresses access the same bank, the accesses need to be serialized, which can affect performance quite a bit. This means that the effective bandwidth of the transfer is reduced by a factor equal to the number of separate memory requests. There is one exception however, which

is when all memory requests are for the same address. In this case the requests can be fulfilled by a single broadcast.

It is interesting to note here that in older architectures, there was only 16 KB shared memory per SM. However, on Fermi (compute 2.0) and onwards, each SM has a total of 64 KB register memory that can be configured to be 16 KB of L1 cache and 48 KB of shared memory, or vice versa.

3.4.6 Registers

The registers are the fastest kind of memory. They are located on-chip, and are accessed per thread. A SM contains a limited amount of registers that are shared between the threads of that SM. In other words, the more threads per SM, the less registers per thread. Register memory is for instance used for storing single, local variables for a thread, and the data lifetime is therefore equal to the thread lifetime.

3.4.7 Local memory

Local memory is private local memory for a single thread. Even though it is private for each thread, it is located off-chip, physically in the device DRAM. It is therefore slower than for instance shared memory. Local memory is used by the compiler instead of registers, when the amount of register space is used up. However, we can prevent this by decreasing the amount of threads per SM, which increases the amount of registers available per thread, like we saw in the previous section. The data lifetime of local memory is equal to the thread lifetime.

3.5 Memory coalescing

When loads and stores to global memory are coalesced, the memory of one warp in compute 2.x and half-warp in compute 1.x can be sent in one single transfer, or possibly 2 in some cases, which can substantially increase memory access performance. This means that we must be very careful when considering the access patterns to global memory in our code. The requirements for coalesced memory transfers differ a bit between the different compute capabilities, but we focus on the Fermi architecture, and therefore only discuss this for compute 2.x. In compute 2.x, the requirements for coalesced memory accesses are: "the concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of cache lines necessary to service all of the threads of the warp" [36, p. 24].

An example of coalesced global memory access in the L1 cache can be seen in figure 3.3. Here we see that the threads access a single cache line, aligned to 128 bytes, leading to coalesced memory access, and only one 128 byte memory transfer, marked in red. In figure 3.4, we see an example of unaligned and sequential addresses, leading to uncoalesced access and two 128 byte L1 transfers, both marked in red.

For information about memory coalescing and different patterns for coalesced memory access in older compute capabilities, see Alexander Ottesens masters thesis [37].

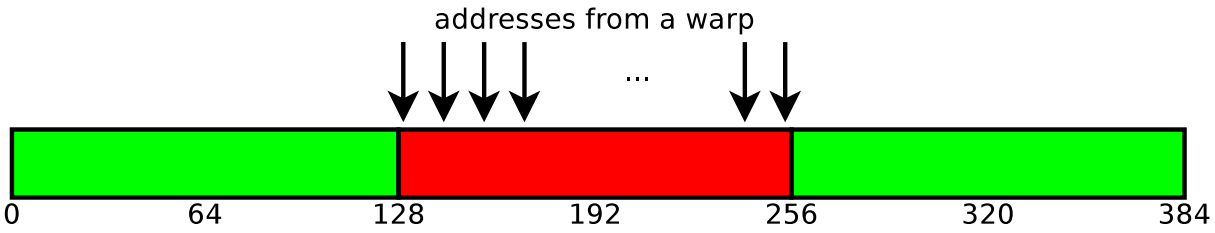


Figure 3.3: Coalesced access

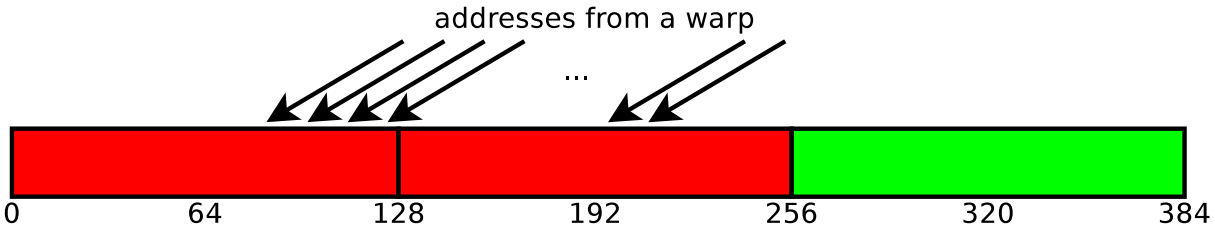


Figure 3.4: Uncoalesced access

3.6 Occupancy

To be able to achieve the theoretical memory bandwidth, we need to have enough active transactions to hide latencies. To be able to measure this, we use *occupancy*. Occupancy is simply a measure of how well the GPU is utilized at a given time. More precisely, occupancy is the ratio between the number of active warps and the maximum number of possible active warps, i.e. $occupancy = activewarps/maximumactivewarps$

One might think that the occupancy should be close to 1 at all times, but this is not always possible. Many developers report that an occupancy of 0.66 is optimal, while the authors of the Zivkovic algorithm on GPU [38] report that, through testing, they found 0.5 to be optimal. There are several reasons for this. First of all, we saw in sections 3.4.6 that a single SM has a limited amount of register space to share for all of its threads. Therefore, if the threads typically are computationally heavy and contain a lot of calculations, they require more registers for this than what stupid and small threads with many memory accesses do. This means that the block size should be smaller when the threads require more registers, leading to a lower occupancy, but can be larger if the register usage is small, leading to a higher occupancy.

Furthermore, as mentioned earlier, shared memory and L1 cache share a pool of 64 KB of memory per SM. This pool can be divided into 16 kB of shared memory and 48 KB of L1 cache, or vice versa. In other words, in threads using lots of L1 cache, we should dedicate the larger part of the memory pool to the L1 cache. If not, occupancy will decrease because there is not enough cache space available. In comparison, if the threads use much shared memory, it is smart to dedicate the most of this memory pool to shared memory. If not, occupancy will decrease because of too little shared memory being available.

In addition, each SM can only have 8 active blocks at a time, so if we select a too small block size, we do not utilize the SMs very well, and the occupancy will therefore drop.

3.7 Summary

In this chapter, we have discussed the CUDA framework used for executing computational tasks in parallel on GPUs. We started by discussing the Fermi architecture and the CUDA execution model. Next, we explained compute capability, and what this means in practice, both for functionality and performance.

To be able to properly understand how to optimize CUDA applications, it is necessary to have a good understanding of the memory architecture, so this was explained next. An important part of this is also the act of structuring the code and data to ensure coalesced memory accesses. Finally, we discussed the occupancy measurement, and how the optimal occupancy level depends on the application.

In the next chapter, we will describe how we have been able to create an improved Bagadus panorama stitcher pipeline, which is able to process frames in real-time. As we will see, CUDA has been a great tool for realizing this.

Chapter 4

The improved Bagadus Panorama Stitcher Pipeline

4.1 Motivation

As we have seen, one of the goals of the Bagadus system is to generate panorama images in real time when recording from the cameras. So far, as we can see in Chapter 2, the creation of these panoramas has been done off-line, and far from real time. We therefore have to find a way to speed things up. There exist a lot of research and implementations for panorama stitching pipelines, so the next step is to look at these, and see if anyone is fitting for our needs, or if we need to build such a pipeline ourselves.

4.2 Related work

Real-time panorama image stitching is becoming more common. For example, many have proposed systems for panorama image stitching (e.g., [11–15]), and modern operating systems for smart phones like Apple iOS and Google Android support generation of panorama pictures in real-time. However, the definition of real-time is not necessarily the same for all applications, and in this case, real-time is similar to "within a second or two". For video, real-time has another meaning, and a panorama picture must be generated in the same speed as the display frame rate, e.g., every 33 ms for a 30 frames-per-second (fps) video in our scenario.

One of these existing systems is Camargus [7]. The people developing this system claim to deliver high definition panorama video in real-time from a setup consisting of 16 cameras (ordered in an array), but since this is a commercial system, we have no insights to the details. Another example is the system Immersive Cockpit [39] which aims to generate a panorama for tele-immersive applications. They generate a stitched video which capture a large field-of-view, but their main goal is not to give output with high visual quality. Although they are able to generate video at a frame rate of about 25 fps for 4 cameras, there are visual limitations to the system, which makes the system not well suited for our scenario.

Moreover, Baudisch et al. [40] present an application for creating panoramic images, but the system is highly dependent on user input. Their definition of real time is

"panorama construction that offers a real-time preview of the panorama while shooting", but they are only able to produce about 4 fps, which is far below our 30 fps requirement. A system similar to ours is presented in [41], which computes stitch-maps on a GPU, but the presented system produces low resolution images, and is limited to only two cameras. The performance is within our real-time requirement, but the timings are based on the assumption that the user accepts a lower quality image than the cameras can produce.

Haynes [42] describes a system by the Content Interface Corporation that creates ultra high resolution videos. The Omnicam system from the Fascinate [43, 44] project also produces high resolution videos. However, both these systems use expensive and specialized hardware, and also require bulky recording equipment and personnel presence at all times. The system described in [42] also makes use of static stitching. A system for creating panoramic videos from already existing video clips is presented in [45], but it does not manage to create panorama videos within our real-time definition. As far as we know, the same issue of real-time is also present in [40, 46–48].

In summary, existing systems (e.g., [39, 45–48]) do not meet our demand of being able to generate the video in real-time, and commercial systems (e.g., [7, 42]) as well as the systems presented in [43, 44] do often not fit into our goal to create a system with limited resource demands. The system presented in [41] is similar to ours, but we require high quality results from processing a minimum of four cameras streams at 30 fps. Thus, due to the lack of a low-cost implementations fulfilling our demands, we have implemented our own panorama video processing pipeline which utilize processing resources on both the CPU and GPU. An overview and an evaluation of our proposed system is presented in the next sections.

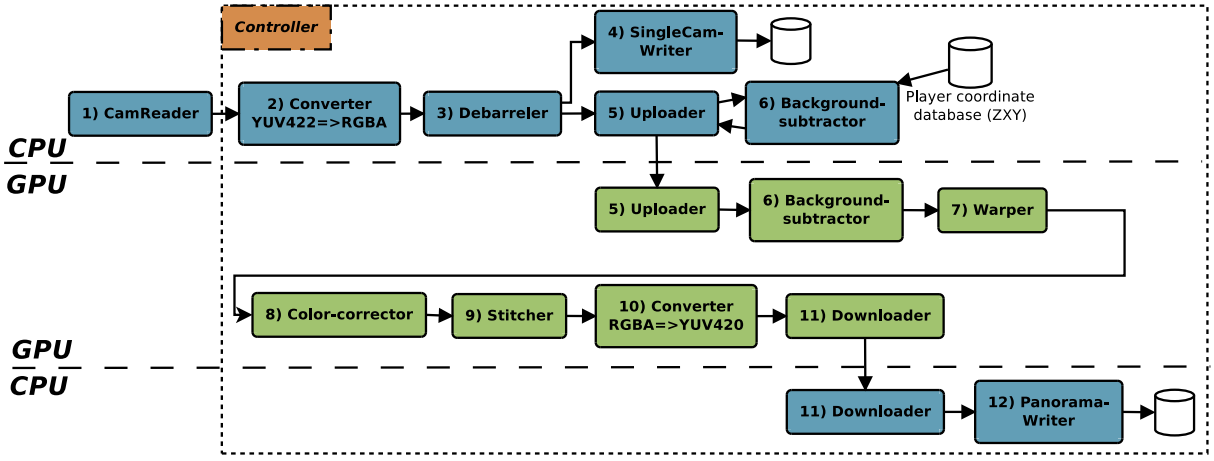


Figure 4.1: The panorama stitcher pipeline

4.3 Improved setup

There are a few changes in the general setup of the Bagadus system in this version. First of all, in the previous version, we had two cameras per computer when recording, and none of these were connected to the internet. We also had one trigger box, shared

by these computers and cameras. This led to some problems, such as difficulties of synchronizing the frame timestamps between the machines. This was solved by use of a separate TimeCodeServer, which used message passing to synchronize the recording clocks on the computers, see section 2.2.2. In addition, there was a rare case where the trigger box would drift slightly due to temperature differences, so that a frame would drop now and then, potentially leading to a small frame drift. In the newest version of the system, we use only one machine for recording from all four cameras, connected to the internet, with one trigger box for camera shutter synchronization. All frame sets are then given a NTP-based time-stamp to mark when they were recorded. This ensures synchronization between the cameras without the need for any extra code. The use of only one trigger box and one computer also solves the slight trigger box-drift problem.

Furthermore, in the old version, the synchronization between the ZXY data and the video frames was done by hand, but we want it to be automatic. We have solved this by configuring the camera recorder and the ZXY database to use the same NTP server. The maximum possible time difference is then so small that we can consider the ZXY data and video frames properly and automatically synchronized.

4.4 Architecture

Due to the performance of modern GPUs for image processing, we decided to utilize this, and therefore wanted as much of the pipeline as possible to be running on a GPU. This decision has affected our architecture quite a lot, and explains why the pipeline contains two parts; one part executing on the CPU and one part on the GPU, as seen in figure 4.1. This decision also explains the motivation behind some of the modules, such as the Downloader found in section 4.5.11 and Uploader found in section 4.5.5. In addition to the choice of using GPUs, we made several other decisions that are reflected in the architecture.

4.4.1 Pipeline startup and initialization

The startup of the pipeline is straight forward. First of all, we need to parse the input parameters. Currently, the pipeline takes the startup time-stamp and recording length as input arguments. If the startup time has not yet occurred, the pipeline waits. When the time has come, the pipeline initializes CUDA, and selects a CUDA device for processing. Currently it selects the CUDA device with the highest number of CUDA cores. When this is done, it initializes the pipeline by creating a new PanoramaPipeline object, and then launches it. The reason for waiting until the startup time before initializing everything, is to be able to schedule several pipeline recordings at the same time. This means that several pipeline processes should be able to wait at the same time, as long as their recording schedules do not overlap. If we did not do it this way, only the first scheduled pipeline would be allowed to run, due to the CUDA resources then being locked by this process. The extra startup time from recording start to actual recording start caused by this is mitigated by launching the pipeline when there are 30 seconds left until scheduled recording start. We found this to be a decent solution, because it is

always better to record some extra frames, than lose some, and 30 seconds proved to be more than enough.

4.4.2 The Controller

The pipeline needs some sort of central controller component to make sure that all modules are synchronized and can cooperate on the same task, errors are caught and handled, buffers and resources are initialized, etc. Without such a component, we would not be able to reach the same level of synchronization. One interesting point here, is how the modules are cooperating and communicating. In short: they do not communicate directly with each other. The only component of the pipeline that all the modules are communicating with, is the controller. The general inter-component communication is done by use of signaling, mutexes, barriers, flags, etc. The controller then uses its state and all of its available information to decide what to do next.

Controller implementation

The general pattern of the controller execution is as follows:

1. Initialize all modules. This is done by creating new module objects, which results in internal module initialization in the constructors. Note that currently, the controller is responsible for initializing all buffers for the Debarreler and Converter itself.

2. For as long as the pipeline is running, loop:

   (a) Wait until the CamReader module has retrieved the next set of frames

   (b) Get the next set of frames from the CamReader

   (c) For all modules, 0 to M, transfer data from the output buffers of module N, to the input buffers of module N + 1. When possible, this is done by swapping buffer pointers, which means that no memory is copied (except the pointer), saving processing time. If the source (output) buffer and target (input) buffer are of different size, this must be done by memory copies (for instance *cudaMemcpy2D()*). Overall though, this is possible with pointer swapping, which drastically increase controller performance.

   (d) Check for and handle potential pipeline frame drops

   (e) Broadcast a signal to all modules (except the reader) to make them process the new data located in the input buffer.

   (f) Wait for all modules to finish processing by use of a barrier

3. Cleanup and provide safe pipeline termination

In other words, we can see that the controller is the component responsible for transferring frames and data through the pipeline, and at the same time control the execution of the modules. Note that when modules are finished processing and program control is returned to the controller component, all module data remain in their

buffers. The output data is then either copied to the next module's input buffers, or the buffers themselves are passed to the next module by use of pointer swapping, as described above.

It is important to remember that when looking at the module processing times, especially compared to the real-time constraint, we need to add the controller processing times to each module, because the modules need to wait for the main controller before they can process a new frame. The controller is therefore a common overhead in all modules (except the CamReader module), and it is therefore important that the controller is as lightweight as possible.

4.4.3 General module design

All the modules follow a common design. First of all, all modules contain a module controller. This thread is the one responsible for communicating with the main controller thread. This is done by use of locks, barriers, signals and counters. Each module controller generally follow these steps:

1. Loop until the pipeline stops
   (a) Wait for a signal from the main controller
   (b) Increase the internal module frame counter
   (c) Execute the module's processing tasks, either by signaling module subthreads to do the processing, by launching CUDA kernels, or by executing the tasks by itself
   (d) Wait for all module threads to finish by use of a barrier

This is a simplification of the general pattern the module controllers follow. Note that in the execution step, there are three different cases of how the processing is executed. In the cases of the CPU-based modules consisting of a single thread, this module controller also executes the processing tasks itself. However, when it is a CPU-based module containing several slave processing threads, the module controller simply signals the slave threads to execute, much in the same way as the main controller signals the module controllers. The last case is for GPU-based modules, where the module controller simply launches a set of CUDA kernels, which then executes on the GPU.

Module buffers

Generally, all modules have two sets of buffers: one or more *input* buffers, and one or more *output* buffers. Exceptions to this are the end-modules, such as the CamReader and the writers, where the reader does not have any input buffers, and the writers do not have any output buffers, as long as we do not count the disk itself as a large buffer. Detailed information about the input and output buffers can be seen in table 4.1. The general pattern here is that all the CPU-based modules, i.e. the blue modules in figure 4.1, have their buffers located in ordinary system RAM on the CPU side, located on the heap. The GPU modules, however, have their buffers located on the GPU, in global device memory. The Uploader and Downloader are exceptions, due to them being

responsible for transferring data between the CPU and the GPU, and therefore need buffers on both sides. The Uploader also needs an extra set of buffers on the GPU, due to it using double buffering when transferring data. This design leads to a relatively common processing pattern of reading input data from the input buffer, processing the data, then writing the processed data to the output buffer. After a module has finished executing an iteration, data remains in the buffers so that the Controller can pass data between the modules. As long as new modules follow this same interface, it is fairly easy to add, re-implement, modify, and remove modules, without having to rewrite any other code than a few lines in the main controller, in addition to the module itself.

Module	Host (CPU)	Device (GPU)
Reader	In: 4 x raw camera stream Out: 4 x YUV frame	-
Converter	In: 4 x YUV frame Out: 4 x RGBA frame	-
Debarreler	In: 4 x RGBA frames Out: 4 x RGBA frames	-
SingleCamWriter	In: 4 x RGBA frame	-
Uploader	In: 4 x RGBA frame	Out: 2 x 4 x RGBA frame Out: 2 x 4 x bytemap
BGS	-	In: 4 x RGBA frame In: 4 x bytemap Out: 4 x RGBA frame (unmodified) Out: 4 x bytemap
Warper	-	In: 4 x RGBA frame In: 4 x bytemap Out: 4 x warped RGBA frame Out: 4 x warped bytemap
Color-correcter	-	In: 4 x warped RGBA frame In: 4 x warped bytemap Out: 4 x color-corrected RGBA frame Out: 4 x color-corrected bytemap
Stitcher	-	In: 4 x color-corrected RGBA frame In: 4 x color-corrected bytemap Out: 1 x stitched RGBA frame
YuvConverter	-	In: 1 x stitched RGBA frame Out: 1 x stitched YUV frame
Downloader	Out: 1 x stitched YUV frame	In: 1 x stitched YUV frame
PanoramaWriter	In: 1 x stitched YUV frame	-

Table 4.1: Pipeline module buffers

4.4.4 The frame delay buffer

There is a short delay of approximately 2 seconds before ZXY coordinate data is accessible from the database during a match [49]. In addition, the time for querying the database for tracking data takes some time (approximately 600-700 ms, as we can see in section 4.7). As we will see in section 4.5.6, the BackgroundSubtractor needs ZXY coordinate data to execute optimally. However, the time from a frame entering the pipeline until it reaches the BackgroundSubtractor is way less than these ~3 seconds. To prevent unwanted drift between frames and the corresponding ZXY data, we need a way to handle this. This is why the pipeline also contains a frame delay buffer, so that

we are sure to have retrieved the corresponding ZXY data from the database before the frames arrive to be processed. We decided to put this buffer between the Debarrel and Uploader modules, because we wanted it to be as close to the BackgroundSubtractor as possible, while we at the same time wanted it to stay on the CPU for an easier implementation. The size of this buffer is 150 * 4 frames. This gives us a buffer containing $\frac{1}{30}\ \frac{seconds}{frame} \times 150\ frames = 5\ seconds$ of frames. The buffer size can easily be modified if necessary.

4.4.5 Handling frame drops

Another important part of the pipeline is how it is designed to handle frame drops. In our pipeline, there are two kinds of frame drops; drops on the camera side, and drops in the pipeline itself.

Camera frame drops

The drops on the camera side happen when the camera drivers in the CamReader module (section 4.5.1) fail to return a frame. This can happen for several reasons, such as transient camera errors, trigger box timing errors, or an overloaded CPU. The way to solve this is to simply reuse the previously read frame, like we can see in section 4.5.1. This is a very non-intrusive way of handling errors, and is very hard to notice with few dropped frames.

Pipeline frame drops

The pipeline needs to handle frames that are dropped when it uses more than the real time threshold on the previous iteration. This can for instance happen due to a generally overloaded CPU, OS interrupts, or file access interrupts. The camera reader module puts new frame sets into a 4x1 frame buffer, by overwriting the previous frames. This means that if the pipeline uses too long to process a frame, the frame it was meant to process next is overwritten by the next one. This will in the long run lead to a visible drift in the frames compared to their proper recording time.

We handle this by having the reader module increase its frame counters on each read. Then, when the Controller is to retrieve the next frame set from the reader output buffer, it also checks this counter. If it is the next to be read, everything is processed normally. However, if the counter is higher than expected, we know that we have lost one or more frame sets, and we must therefore properly handle this. To do this, all the modules contain a drop counter buffer. The controller then pushes the current, lost frame number into all the modules' drop counter buffers. On every iteration, each module starts by checking its drop counter buffer to see what the first number in this buffer is. If this number equals the module's current frame counter, it knows that this frame is to be skipped, and therefore pops the front of the buffer and returns at once. If the frame is not to be skipped, it processes the new frame correctly. The writer modules are a bit different, because they at the skip frame-case write the previous frame (which is cached) directly to disk, instead of returning immediately. Using these

drop counter buffers allows us to drop frames safely in a pipeline fashion by reusing the last successfully processed frame.

4.4.6 Pipeline execution pattern

It is important to note the execution pattern of the pipeline, where the modules are executed in a pipelined fashion. This means that it executes using the same pattern as, for instance, a CPU instruction pipeline. In other words, the different operations are executing in parallel, but do not process the same input data at the same time, where the data is moved one step further in the pipeline for each time unit. For us, this means that the modules all execute in parallel, but the modules process different frame sets concurrently. The only exception is the SingleCamWriter and Uploader modules, which, when ignoring the frame delay buffer, process the same frame concurrently.

We find a visualization of this in figure 4.2. Here we see that as soon as the modules receive an input frame set, they execute in parallel with the other modules. However, a module never processes the same frame at the same time as another module, except for the SingleCamWriter and Uploader. Note that the frame delay buffer has been hidden in the figure, because it does not affect the execution pattern, and only makes the figure more complex. This pattern means that, when analyzing the performance, as long as each module operates within the real-time requirement of 33 ms, while processing in parallel with the other modules, the whole pipeline can be considered to operate in real-time. We will look at the performance in section 4.7.

Input frame number (time)	Cam-Reader	Converter	Debarreler	SingleCam-Writer + Uploader	BGS	Warper	Color-Corrector	Stitcher	Yuv-Converter	Downloader	Panorama-Writer
n	n										
n + 1	n + 1	n									
n + 2	n + 2	n + 1	n								
n + 3	n + 3	n + 2	n + 1	n							
n + 4	n + 4	n + 3	n + 2	n + 1	n						
n + 5	n + 5	n + 4	n + 3	n + 2	n + 1	n					
n + 6	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n				
n + 7	n + 7	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n			
n + 8	n + 8	n + 7	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n		
n + 9	n + 9	n + 8	n + 7	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n	
n + 10	n + 10	n + 9	n + 8	n + 7	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n

Figure 4.2: The improved pipeline execution pattern
The frame delay buffer is hidden to minimize figure complexity.

4.4.7 Optimizing x264 storage settings

In section 2.5.2 we saw that the processed videos are encoded and stored as 3 second long H.264 files, with the time-stamp of the first frame in the file name. However, there were some x264 encoder settings that were not explored. A few different x264 presets were tested in [2], but not in detail. Due to the strict performance requirements in this pipeline, we had to be sure that our encoder settings were optimal. When selecting these settings, it is important to be aware of the general trade-offs of encoding speed, compression ratio and visual quality. We optimally want very fast encoding speeds,

large compression ratios, and lossless image quality, but this is not possible in real-life. For instance, when increasing the compression ratio, the encoding speed generally lowers, and the image quality might lower, depending on the settings. Furthermore, if we want lossless image quality, we cannot compress the video as much without either losing visual quality and/or increasing the encoding speed. We generally want to store the videos lossless, with fast encoding, which comes at the cost at not so optimal compression ratios and therefore larger storage requirements.

The settings we found to result in the best encoding performance while retaining the best resulting image quality, was to use the same x264 profile mentioned in [2], but in addition lower the bit-rate requirements. By lowering the bit-rate requirements slightly, the resulting 3 second files became smaller, the encoding time was lowered by a substantial amount, while the visual quality remained close to lossless. In addition, we activated x264 frame slicing to improve encoding times. X264 frame slicing is targeted at low-latency encoding, such as in our pipeline. A detailed description of x264 slicing can be found in [50], but it basically consists of slicing a frame into several sub-frames, and dedicating one thread for encoding each slice. We found the optimal amount of threads dedicated to slicing to be four per frame.

4.5 Pipeline module details

To properly understand how the panorama stitcher pipeline works, we need to take a look at all the modules, how they work, what they do, and how they cooperate.

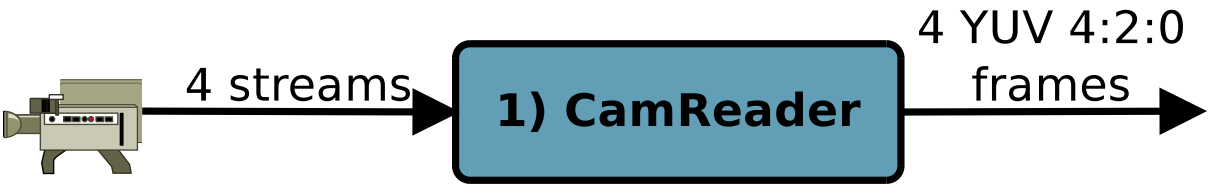


Figure 4.3: The CamReader module

4.5.1 Retrieving frames from the source - The CamReader module

The first module is one of the most important of all the modules: The camera reader. The CamReader module is the one responsible for retrieving frames from the connected cameras so that these can be processed in the pipeline. The frame source, i.e. the cameras, is the component that decides the real time requirements that the pipeline needs to fulfill. Our cameras output frames at a speed of 30 frames per second. This means that the pipeline has a deadline of processing a frame of 1/30th of a second, i.e approximately 33 ms. If each module consume less time than this to process each frame, and each processed frame is stored to disk every 33 ms, the whole pipeline can be considered real-time.

Module implementation

This module is located and executed on the CPU, and runs as one dedicated thread per camera, i.e. four threads in our scenario. As input it takes the direct frame streams from the cameras. Meanwhile, the output is provided as four YUV 4:2:2 frames (formatted as YUYV) of the correct resolution. As we can see from the camera specifications [19], the maximum supported frame size is 1294 x 964 pixels. However, this is slightly limited by the Basler Pylon camera drivers, which results in a maximum retrievable frame size of 1280 x 960 pixels.

During development, we implemented a mock-up PanoramaReader that read YUV files stored on disk, which worked well for a small time period. Eventually, this reader was replaced by the proper CamReader that retrieves actual, live frames from the cameras. This shows us that it is easy to re-implement and replace modules, without having to make large changes to the pipeline. This means that we could replace the current reader with a new reader that for instance would read from cameras from another manufacturer.

When concerning the access to the camera drivers, the NorthLight library contains a wrapper around the drivers, which makes driver- and camera-interaction in the CamReader module rather straightforward and easy.

The CamReader has an execution pattern where each reader thread consists of a while-loop that loops for as long as there is a functioning connection to a camera, or as long as the whole pipeline is running. The general execution for each thread is as follows:

1. For as long as all CamReader threads are retrieving frames and the pipeline is still active
   (a) Try to retrieve a frame, with a timeout of 34 ms
   (b) On frame retrieval timeout, clone the previous frame, but update its time-stamp to now
   (c) On frame retrieval success, store the new frame in the output buffer, with a time-stamp set to now
   (d) Wait for the other reader threads before continuing

The synchronization between the four cameras is secured by the trigger box. The trigger box is set to synchronize the camera shutters at a frequency of 30 Hz, i.e. the same frame-rate as the maximum supported FPS for the cameras.

Figure 4.4: The Converter module

4.5.2 Converting frames to correct format - The Converter module

Before we can start processing the frames in the pipeline properly, we need to convert them to a format that is easier to handle and process than YUV. The Converter module is therefore responsible for converting from YUV 4:2:2 read by the CamReader module, into the format used in the rest of the pipeline. We decided that we want to use the RGBA color format for internal frame representation in the pipeline. First of all, RGB is simpler to work with than YUV and is conceptually easier to understand, in addition to that RGB is the color format needed by for instance the color corrector module. The main reason for using RGBA and not just plain RGB, is the same as mentioned in [38], i.e. that by using RGBA, we promote coalesced memory access on the GPU, which improves memory access times, and therefore kernel execution times on the GPU. The converter therefore converts all the frames from YUV format to RGBA format. The Converter module can be seen in 4.1, numbered as module 2.

Module implementation

The converter is also located on the CPU, takes four YUV4:2:2 frames as input from the cameras, and provides four RGBA frames as output. The performance of the conversion process is so good that we did not need to parallelize the converter into one thread per camera. It therefore consists of a single thread only, sequentially converting a frame for each camera per iteration. The implementation is based on the NorthLight library, which again relies on *ffmpeg* and *swscale* for converting frames. The execution is simplified as follows:

1. For all cameras 0 to N
   (a) Convert the input frame for camera n from YUV 4:2:2 (YUYV) to YUV 4:2:0
   (b) Convert this frame from YUV 4:2:0 to RGBA

The conversion from YUV 4:2:2 $\rightarrow$ RGBA is done in two steps due to limitations in the NorthLight frame converter. This means that we must convert to YUV 4:2:0 first, before converting to RGBA. However, swscale is highly optimized, so, as we will see later, the performance is still good enough. This could also have been done using SSE instructions, but is not done because the runtime gain would have been very small and is not required.

Figure 4.5: The Debarreler module

4.5.3 Removing barrel distortion - The PanoramaDebarreler

As we have seen earlier in Chapter 2, the frames provided by the cameras are barrel distorted due to the wide angle lenses. Before we can do anything more with these images, we need to remove this barrel distortion. Our scenario is of a soccer field, so the visual impact of the barrel distortion is extra visible due to the straight lines on the field, and it is therefore extra important to debarrel the frames. This is the responsibility of the Debarreler module.

Module implementation

Like the previous modules, this module is running on the CPU. As input it takes four barrel distorted RGBA frames, and provides four debarreled RGBA frames as output. Like in the old Bagadus system, the debarreler needs a set of barrel distortion coefficients for each camera, but these are calculated as part of the configuration steps before running the pipeline. Like in the old pipeline, the current debarreling function is provided by OpenCV. This debarreling implementation is so slow that it does not fulfill the real-time constraints when run sequentially for all the cameras. We therefore had to parallelize it by assigning a dedicated debarreler thread per camera stream, and let this thread run the debarreling function for a single frame. This practically cuts the debarreling time by a factor of four. In addition, we selected nearest neighbor as interpolation algorithm, due to performance [2]. Each Debarreler-thread therefore executes the following pseudo code:

1. Run OpenCVs debarreling function, with the correct debarreling coefficients, for the current frame

OpenCVs debarreling function is slow, but as we will see later, fast enough for our pipeline. However, to offload the CPU and have a larger margin for processing spikes, it is desirable to improve this module by replacing OpenCVs debarrel function with a faster implementation.

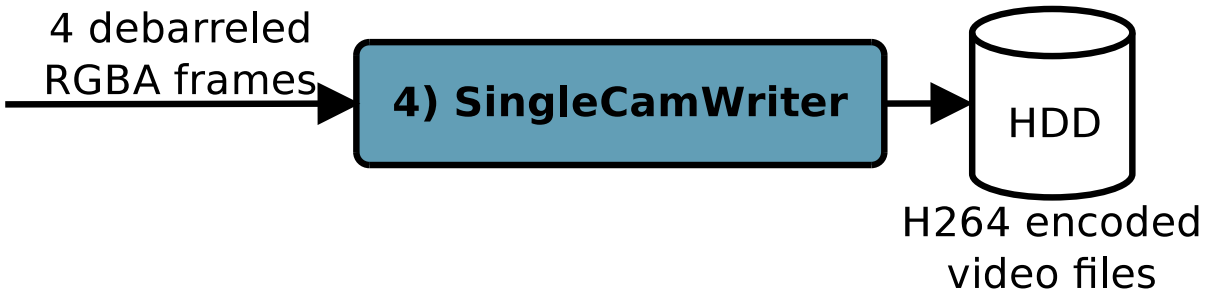


Figure 4.6: The SingleCamWriter module

4.5.4 Writing the original camera frames to disk - The SingleCamWriter module

In addition to creating a stitched panorama video based on the four cameras, we want to store the original recorded frames, and we therefore need the SingleCamWriter mod-

ule. We want to write these frames to disk just after the debarreler, and not before, because barrel distorted videos are not really useful, so we might as well debarrel the frames before storing the videos.

Module implementation

The SingleCamWriter takes four debarreled RGBA frames as input, and writes sequential frames to video files on disk. It runs on the CPU, and consists of a dedicated thread per camera. As seen in section 2.5.2, we encode the frames as 3 second long H.264 files and store them on disk, with the file number and time stamp in the file name. The different cameras are separated on disk by writing to a folder per camera. Each SingleCamWriter-thread therefore executes as follows:

1. Convert from RGBA to YUV 4:2:0, needed in our H.264 encoder.
2. If we have written 3 seconds of frames to the same file, close the current file stream and open a new one, with updated time-stamp and counters.
3. Use the H.264-encoder in NorthLight to encode the frame
4. Write H.264 encoded data

The conversion, encoding and writing parts are logically three different operations, and could therefore be separated into different pipeline modules. However, the conversion and writing to disk parts are negligible compared to the encoding bit, so there is in practice no point in separating these into separate modules. The conversion operation is based on the NorthLight library, which again relies on *ffmpeg* and *swscale*, which is highly optimized, for converting frames. The performance is therefore very good.

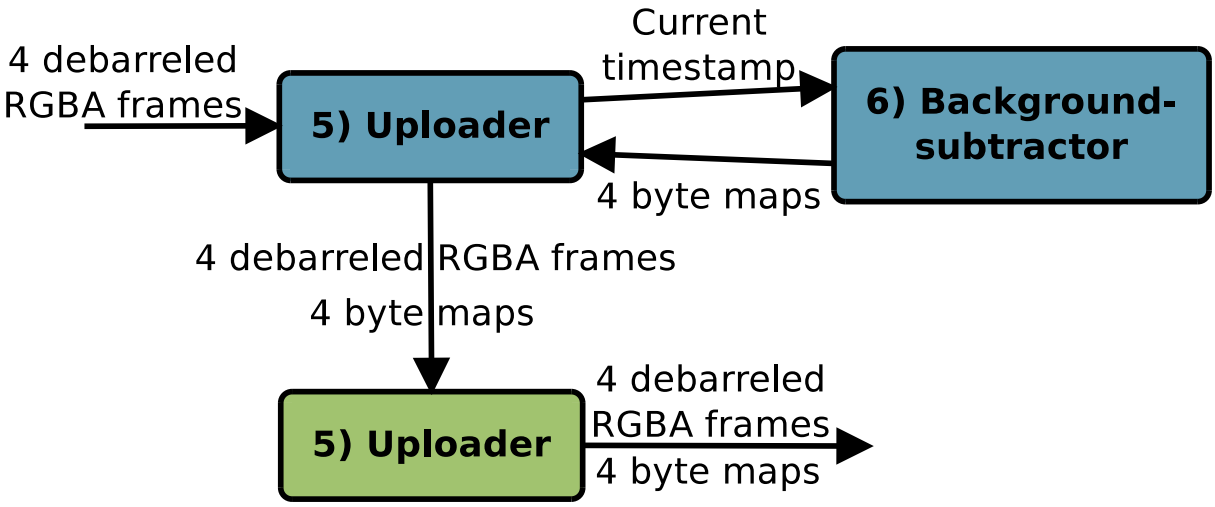


Figure 4.7: The Uploader module

4.5.5 Transferring frames to the GPU - The Uploader module

From here on in the pipeline, most of the modules are executing on the GPU. To be able to do this, we need to transfer the required input to the GPU. This is the task of the Uploader module. However, as we will see below, the BackgroundSubtractor module has a part running on the CPU, calculating some byte maps needed on the GPU. Therefore, the Uploader has currently also the responsibility of executing this part of the BackgroundSubtractor, and then transfer the resulting byte map to the GPU.

Module implementation

The Uploader is running as a single thread on the CPU, and transfers data to the GPU. It takes four debarreled RGBA frames as input from the CPU, and provides four RGBA frames and four byte maps as output on the GPU. When transferring data to the GPU, we have several choices: We can use the synchronous cudaMemcpy() function to transfer the data sequentially, or we can use the asynchronous cudaMemcpyAsync() function to copy the data asynchronously and in parallel with CUDA streams. In addition, when using asynchronous transfers with pinned host memory, we can utilize double buffering. Double buffering and asynchronous transfers have several advantages, where the most important advantage is better interleaving with kernel execution, data transfers and CPU execution. We therefore implemented the Uploader to use this, which results in it needing twice the buffers for transferring to the GPU, as we can see in table 4.1.

In addition to the transfers, the BackgroundSubtractor contains a CPU part, where some byte maps are calculated. This calculation is currently executed by the Uploader, and the resulting byte maps are then transferred to the device in the same way as the corresponding RGBA frames.

The execution flow of the Uploader is therefore:

1. If a BackgroundSubtractor module exists, calculate the player pixel byte maps
2. For all cameras 0 to M
   (a) Begin asynchronous transfer of frame for camera N
   (b) If there was a BGS module, begin asynchronous transfer of byte map for camera N

4.5.6 Executing background subtraction - The BackgroundSubtractor module

We will go much more into detail on background subtraction in Chapter 5, but we will provide a small summary here. First of all, background subtraction is the process of analyzing a video stream and determining what pixels are considered background, and which pixels are considered foreground. This is a useful tool, as it can provide us with pixel precise information about where the players on the field are. This information can later be used for several things, as we will see, such as improving the stitcher module

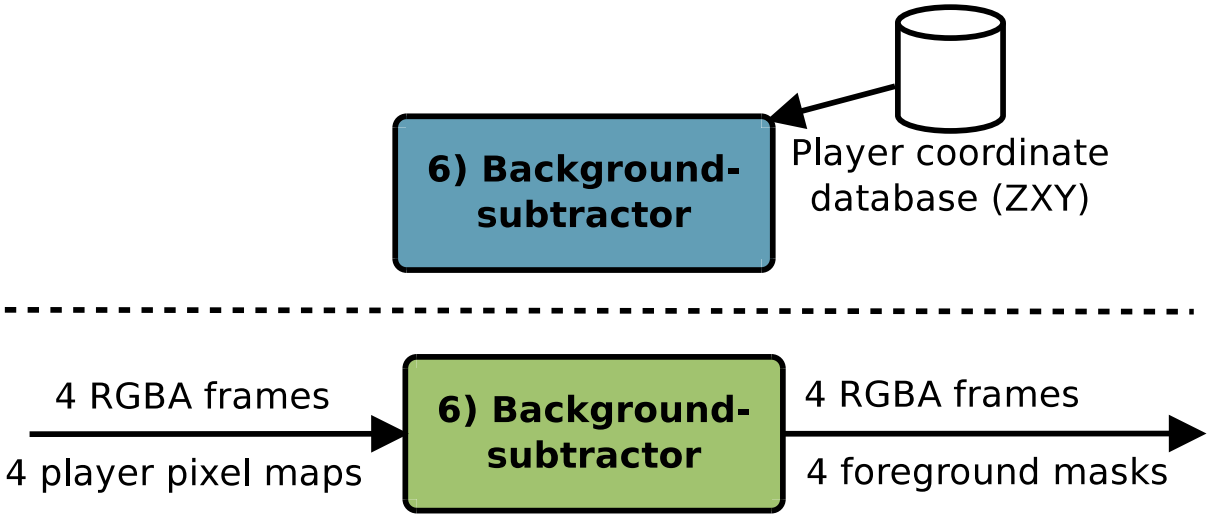


Figure 4.8: The BackgroundSubtractor module

(section 5.8.1), and improving the performance and accuracy of depth map creation (section 5.8.2). Due to this, we want to add a background subtraction component to our pipeline, and this analysis is executed by the BackgroundSubtractor module.

In our scenario, due to us having statically positioned cameras, and knowing the coordinates of the players, we also know the approximate player pixel positions. By utilizing this knowledge, we can improve our background subtractor, both in performance and precision. This is done by translating player positions into pixel positions, and then only process pixels close to these location, while automatically classifying the rest as background. These player pixels, including a safety margin, are set to 1 in a player pixel lookup map, for later use during the BGS analysis.

Module implementation

The BackgroundSubtractor module is running partially on the CPU and partially on the GPU. The part running on the CPU is the one responsible for calculating the player pixel lookup maps. To get the data from the database containing the ZXY data for the players, we have a dedicated thread that retrieves ZXY samples from the database when needed. The actual creation of these byte maps is currently executed by the Uploader module, but could easily be split into a separate module.

The GPU part of the BGS module is the part actually running the background subtraction. It takes four RGBA frames and four corresponding player pixel lookup maps as input, and provides the unmodified RGBA frames and the corresponding foreground masks as output. All of these buffers are located in global CUDA memory.

The implementation described in Chapter 5 has been modified and refactored to properly fit into the pipeline. This means several changes, for instance adding better use of C++ objects, and adding support for analyzing several images (one from each camera) in parallel. In addition, the module has a fall-back mode for situations where there is no ZXY data available. In this fall-back mode, the BGS module simply processes the whole field, instead of using ZXY data to limit the amount of pixels to be processed.

A simplification of the ZXY data retriever thread is:

1. Retrieve ZXY session start
2. Retrieve player information
3. While the pipeline is active:
   (a) If the ZXY data cache size in memory is below a certain threshold, retrieve a chunk of data from the ZXY database.

The execution of the player pixel lookup map creation is:

1. Create empty byte map of size 1280 * 960
2. Retrieve and remove the sample belonging to the current frames' time-stamp from the local ZXY cache.
3. For all the players:
   (a) Translate the ZXY coordinate to pixel data
   (b) In the new byte map, set the translated pixel position, including a margin at all sides, to 1
4. Return the byte map

The execution of the GPU side of the BGS module is:

1. For all cameras 0 to M
   (a) For every pixel, if the pixel is 0 in the player pixel lookup map, mark it as background at once. Else, calculate the pixel status to either background, foreground or shadow.

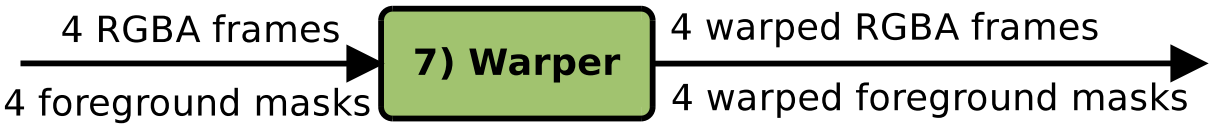


Figure 4.9: The Warper module

4.5.7 Warping the frames to fit the panorama - The Warper module

The Warper module is the module responsible for warping the camera frames to fit the stitched panorama image. By warping, we mean twisting, rotating and skewing the images to fit the common panorama plane. This is necessary because the stitcher will assume that the input images are perfectly warped and aligned to be stitched to a large panorama. The warper also warps the foreground masks provided by the BGS module. This is because the Stitcher module will use the masks at a later point, and expects the masks to fit perfectly to the corresponding warped camera frames.

Module implementation

This module is running only on the GPU, and takes four RGBA camera frames and four foreground masks as input. As output it provides four warped RGBA frames and four warped foreground masks, all of these of a new resolution. Due to the use of the Nvidia Performance Primitives (NPP) library [51], this function is rather simple. The most important part of the module is the warp parameters needed, in other words the interpolation algorithm and the set of transformation matrices. The interpolation algorithm used is the same as in the Debarreler, i.e. nearest neighbor, selected due to the performance. The transformation matrices have been generated on system setup, and are used here. As soon as we have these arguments, the NPP library makes implementation easy for us:

1. For all cameras 0 to M
   (a) Warp the input camera frame for camera *n* by using nppiWarpPerspective_8u_C4R()
   (b) Warp the foreground mask for camera *n* by using nppiWarpPerspective_8u_C4R()

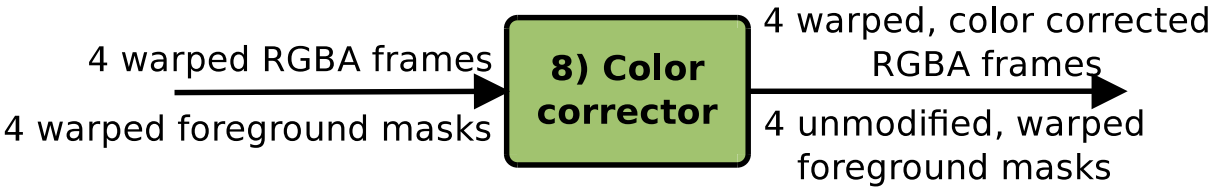


Figure 4.10: The ColorCorrector module

4.5.8 Correcting color differences - The ColorCorrector module

When recording frames from our four cameras, we let the cameras adapt to the lightning individually, due to the different lighting conditions between them. This means that, for us to generate the best looking panorama video, we need to correct the colors of all the frames to remove the inter-camera color disparities. In the stitched output videos of the original pipeline, color disparities between the cameras, and therefore seams, are clearly visible, which we can see in figure 4.12(a). However, with a color-corrector (CC) module, these color differences can be limited by a large margin. In figure 4.12(c), we see the results of using a dynamic stitching seam (which we will explain in section 4.5.9) without any color correction; the seam is clearly visible because of color differences. However, in figure 4.12(d), we use the same seam, only with color correction this time. The results in this example are great; with the color correction it is near impossible to see where the seam is going.

Module implementation

The CC module takes four warped RGBA frames and four warped foreground masks as input. As output it provides four color corrected and warped RGBA frames, in addition to the unmodified, warped foreground masks. The foreground masks are simply

sent through the module to be used by later modules. To begin with, we must find the overlapping regions between the cameras. This can be done during initialization of the system, and is currently done manually. Furthermore, to be able to correct the color differences, each camera has a corresponding set of correction coefficients. The correction coefficient sets are arrays of three RGB values, one for each channel, and describes the color differences between overlapping cameras. The current algorithm is relatively simple, and corrects color differences in the cameras sequentially from the left to the right. Due to its simplicity, it does not remove all color differences, and struggles to provide a good output during difficult lighting conditions. The pseudo-code of the color-corrector is:

1. Select the leftmost camera, camera 1, as the primary camera, i.e. set its correction coefficients to 1. This means that camera 1 will be used as the baseline for color adjustment, and will therefore not have any colors modified.
2. For the remaining cameras, 2 to M:
   (a) Calculate the color correction coefficients for the current camera, n, by comparing a subset of its region overlapping with camera $n-1$ with the same subset from camera $n-1$
   (b) Color-correct the frame for the current camera, n, by applying its color correction coefficients.

More details about color correction can be found in the master's thesis of Mikkel Næss [52].

Figure 4.11: The Stitcher module

4.5.9 Stitching the frames together - The Stitcher module

The next GPU module, the Stitcher, is the module where the panorama stitching actually takes place. It is based on the original Bagadus stitcher, where we use 2D transformations and create seams between the overlapping camera frames, and then copy pixels from the images based on these seams. These frames need to fit the same projection/plane, which is why we need the warper in the previous step. The old stitcher used fixed, straight cuts for seams, which means that fixed, rectangular areas from each frame were copied directly to the output panorama frame. These static cut panoramas are generally very fast to create, but contains lots of graphical errors, such as in figure 4.12(a), where we can see a player being cut by the straight cut, resulting in visual artifacts. We want to create better seams for better visual results, and therefore introduce a dynamic cut stitcher. The goal of the dynamic cut stitcher is to calculate

dynamic cuts for each frame, so that the seam is as invisible as possible, while also avoiding cutting through players to reduce the amount of visual artifacts. An example of how the final seam can look is seen in figure 4.12(b). A dynamic seam without color-correction can be seen in figure 4.12(c), while such a seam with color-correction is found in figure 4.12(d).

Module implementation

The Stitcher is executing on the GPU, and takes four warped foreground masks and four corresponding, warped and color corrected RGBA camera frames as input. As output it provides a single, stitched RGBA panorama frame.

The dynamic cut stitcher starts by calculating the seams between the frames. This is done by first creating a rectangle of adjustable width over the static seam areas. All pixels within this seam area are then treated as graph nodes, where the graph is directed from the bottom to the top and each pixel points to the three adjacent pixels/nodes above. The left-most and right-most pixels only point to the two adjacent pixels available. The weight of these edges are calculated by a custom weight function that compares the absolute color differences between the corresponding pixels in each of the frames we are currently trying to stitch. The weight function also checks the value of the corresponding foreground mask, to see if there are currently any players in that pixel. If so, the weight of that edge is set to a high weight to prevent seams passing through players. This has the effect of making the edges between nodes where the color differs or players are present have much larger weights.

The next step is then to run Dijkstra's algorithm [53] on the resulted pixel graphs to find the minimal cost routes from the bottom of the seam cut areas to the top. The graph is directed upwards, which means that we can only traverse the graph directly upwards or diagonally upwards, and therefore only get one node per horizontal position. When we loop through the graphs in this way, we get the new cut offsets by adding the node's horizontal position to the base offset. In total, this results in the dynamic seams, which we can then use for copying pixels to create the panorama frame. The pseudo-code of the stitcher can be summarized as:

1. Calculate pixel edge weights, as described above
2. Calculate the best dynamic seams by using Dijkstra on the graphs from the step 1, as described above.
3. For all cameras 0 to M
    (a) Copy pixels from the warped camera frame to the corresponding location in the target frame buffer, based on the dynamic seams.

The performance of stitching frames from four cameras together is found in table 4.2. We can see that the performance is very good, and it is interesting to note that the CPU version is currently slightly faster than our GPU version. This is caused by searches and branches often being more efficient on traditional CPUs. However, further optimization of the CUDA code will likely improve this GPU performance. In

addition, when running on the GPU, we can avoid adding unnecessary transfers between the GPU and CPU in the middle of our pipeline, which would have increased the load on the PCIe bus and potentially lowered the whole pipeline performance. Note that the low minimum times on GPU are caused by the frame drop handling described in 4.4.5, while the maximum is caused by the first frame set taking longer to process than the others. More details about this dynamic stitching can be found in the master's thesis of Espen Helgedagsrud [54].

	Min	Max	Mean
CPU (Intel Core i7-2600)	3.5	4.2	3.8
GPU (Nvidia Geforce GTX 680)	0.0	23.9	4.8

Table 4.2: Dynamic stitching (ms).

(a) The original stitch pipeline in [10] and [2]: a fixed cut stitch with a straight vertical seam, i.e., showing a player getting distorted in the seam.

(b) The new stitch pipeline: a dynamic stitch with color correction, i.e., the system search for a seam omitting dynamic objects (players).

(c) Dynamic stitch with **no** color correction. In the left image, one can see the seam search area between the red lines, and the seam in yellow. In the right image, one clearly see the seam, going outside the player, but there are still color differences.

(d) Dynamic stitch **with** color correction. In the left image, one can see the seam search area between the red lines, and the seam in yellow. In the right image, one cannot see the seam, and there are no color differences. (Note that the seam is also slightly different with and without color correction due the change of pixel values when searching for the best seam after color correction).

Figure 4.12: Stitcher comparison - improving the visual quality with dynamic seams and color correction.

Figure 4.13: The YuvConverter module

4.5.10 Converting the frame format back to YUV - The YuvConverter module

Before storing the stitched panorama image, we need to, like in the SingleCamWriter module, convert the image back from RGBA to YUV 4:2:0, which is the required format in the H.264 encoder. The reason for having a dedicated module on the GPU for this, is that the converter is not fast enough on the CPU, even with the relatively fast *ffmpeg/swscale* implementation in NorthLight. Converting between video formats is a massively parallelizable task, so using the GPU for this is natural.

Module implementation

The YuvConverter module is running on the GPU. The input is a single stitched panorama frame in RGBA format, and the output is a stitched panorama frame in YUV 4:2:0 format. In this module we utilize the NPP library to first convert the input from RGBA to YUV 4:4:4. The NPP library does not have a converter function for converting from YUV 4:4:4 to YUV 4:2:0, so this part must be done manually in CUDA. The execution is therefore:

1. Convert the input frame from RGBA to YUV 4:4:4 by use of nppiRGBToYCbCr_8u_AC4P3R()
2. Copy the resulting Y-channel directly to the Y-channel of the target buffer
3. For all samples in the U channel:
   (a) If this sample is supposed to be a sample in the YUV 4:2:0 U channel, copy the value to the target buffer U channel, at the correct position
4. For all samples in the V channel:
   (a) If this sample is supposed to be a sample in the YUV 4:2:0 V channel, copy the value to the target buffer V channel, at the correct position

Note that the conversion from RGBA to YUV 4:2:0 can be done in one step, by writing the whole conversion in CUDA ourselves, using known formulas. However, for the sake of getting this module to work quickly, we implemented it first using the current solution. Like we will see later in section 4.7, the performance of the YuvConverter with this non-optimal solution, proved to be fast enough, so we did not prioritize implementing the more optimal solution of direct RGBA $\Rightarrow$ YUV 4:2:0 conversion. This is therefore future work.

4.5.11 Transferring the panorama frames back to the CPU - The Downloader module

Now that we are done with the stitching itself, we need to transfer the panorama output back to the host, i.e. the CPU memory. This is done by the Downloader module.

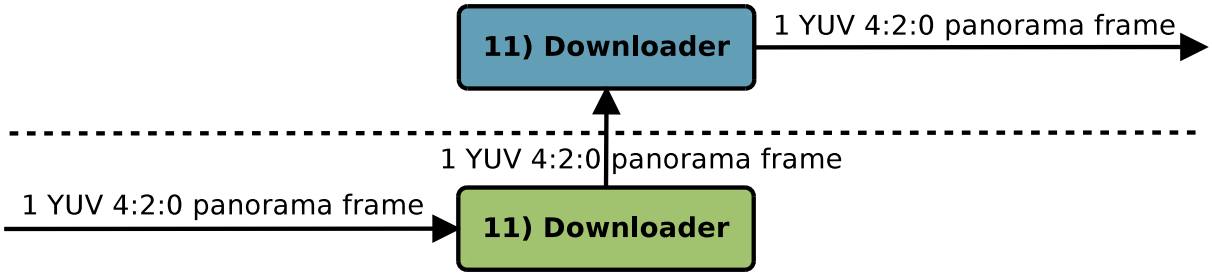


Figure 4.14: The Downloader module

Module implementation

The Downloader is running as a single thread on the CPU, and transfers a stitched panorama frame from the GPU to the CPU. As input it takes a stitched YUV 4:2:0 panorama frame located on the GPU, and the provided output is a stitched YUV 4:2:0 panorama frame on the CPU. The Downloader is much simpler than the Uploader. For instance, it does not need to calculate any player pixel lookup maps, and it only has to transfer a single frame. This is reflected in the execution:

1. Copy the panorama frame from the GPU to the CPU by use of cudaMempcy()

As we can see, the Downloader module is transferring the frame synchronously. We could have used double buffering and asynchronous transfers like in the Uploader, but the Downloader has no other tasks, so using a simple synchronous cudaMemcpy() is more than good enough.

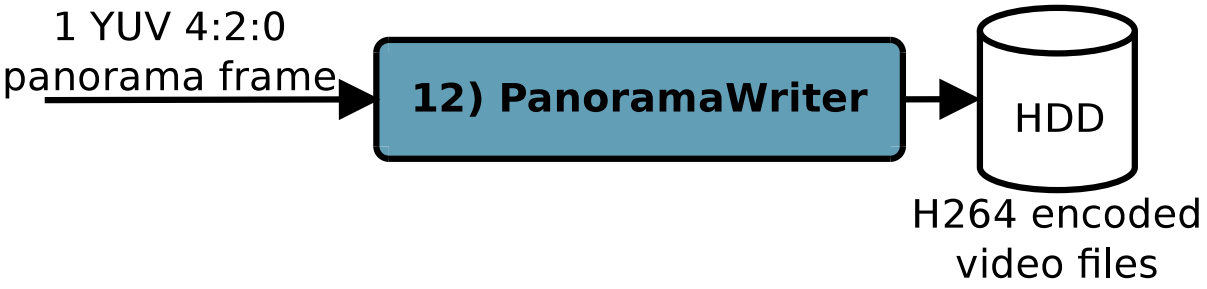


Figure 4.15: The PanoramaWriter module

4.5.12 Storing the panorama images - The PanoramaWriter module

We finally get to the last step in the pipeline, which is to store the stitched panorama frames as video to the disk. This is the responsibility of the PanoramaWriter module. The results of the PanoramaWriter are video files stored on disk, that are possible to playback on any kind of video player that supports H.264-decoding.

Module implementation

This module is running on the CPU, as a single thread, and only access CPU memory. As input it takes a stitched YUV 4:2:0 panorama frame, and the provided output are

H.264 encoded panorama frames on disk. As with the SingleCamWriter, we want to encode and store frames in 3 second long video files, with a filename consisting of a file number and time-stamp. The execution is relatively equal to the SingleCamWriter, except that we do not need to convert from RGBA to YUV 4:2:0, as this has already been done in the YuvConverter module:

1. If we have written 3 seconds of frames to the same file, close the current file stream and open a new one, with updated time-stamp and counters.
2. Use the H.264-encoder in NorthLight to encode the frame
3. Write H.264 encoded data

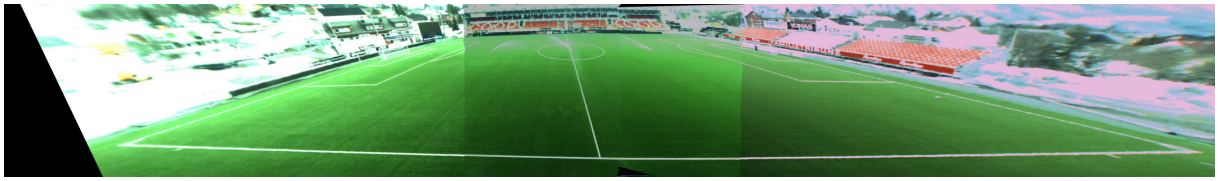
The resulting 3 second files are stored in a folder separate from the single camera video files. Note that the encoding and writing parts are logically different modules, and could therefore be separated. However, as we will see below, the writing to disk part is negligible compared to the encoding bit, so there is in practice no point in separating these tasks.

4.6 Improved panorama pipeline visual results

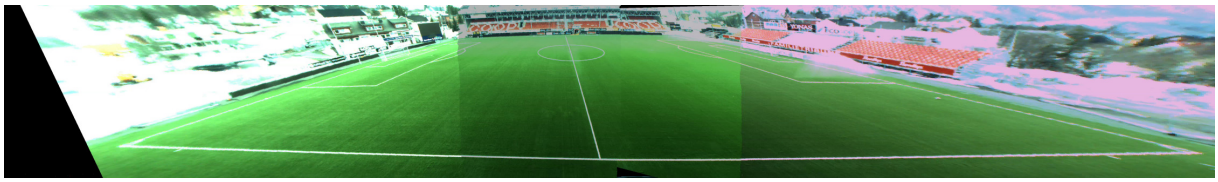
A screenshot of the output of the improved pipeline compared to the old pipeline can be seen in figure 4.16, with examples of different camera settings. We can in figure 4.16(b), figure 4.16(c) and figure 4.16(d) see the dynamic seams between the cameras avoiding players and minimizing the visibility of the cuts. The results are good, where for instance the white lines in the field are cut perfectly, without causing any distortion, and players are avoided. In addition, the warping in the improved panorama is much better than before, where we for instance can see that the bottom line is now connected without warping errors. The only warping errors are seen in the stands area, which is unimportant, and is caused by imperfect system calibration. This can easily be fixed by calibrating the system carefully, or by cropping the result more aggressively. Furthermore, by using color correction, the color differences between the cameras are reduced. However, the color differences, and therefore cuts, are still visible. This is caused by too large color differences, which in turn is caused by too different camera exposure times. By improving and synchronizing the camera exposures, the color correction would work better, leading to even better seams.

Note that the colors and brightness of the improved panorama is currently not perfect. In figure 4.16(b), the standard exposure times are too high, and the white balance is set on startup, meaning that the panorama is too bright with slightly wrong white balance. In figure 4.16(c), we use the same exposure times, but enable automatic white balance. The exposure time problem is an issue currently being investigated, where we can see the current, experimental work on auto exposure enabled in figure 4.16(d). It is also very important to note that the old pipeline output in figure 4.16(a) were recorded in the fall, while the improved pipeline output were recorded in the spring, leading to dramatically different lighting conditions. Nevertheless, even with these imperfect camera settings, figure 4.16 still proves that the technical parts, i.e. the warping, cut calculations and color correction, improve the visual results of the panorama.

(a) Old pipeline output



(b) Improved pipeline output, locked exposure times, locked white balance.



(c) Improved pipeline output, locked exposure times, automatic white balance.

(d) Improved pipeline output, experimental auto exposure enabled, locked white balance

Figure 4.16: Old vs. improved pipeline output
Note that the old pipeline output were recorded in the fall, while the improved pipeline output were recorded in the spring, leading to dramatically different lighting conditions.

4.7 Panorama stitcher pipeline performance

We have seen that the old pipeline was not performing fast enough to process the frames in real-time. The new and improved pipeline is supposed to run in real-time, and it is therefore important to measure the total performance. In figure 4.17, we can see the overall performance of all the modules when running the new pipeline on the computer DevBox 2, with specifications seen in table C.2. CPU modules are marked in blue, and GPU-based modules in green. Note that, as we have seen earlier, the Uploader executes both on CPU and GPU, but we have chosen to mark it in blue here. Even with the controller overhead, we can see that when executing the whole pipeline, all modules perform well below the real-time threshold. However, these module measurements do not prove whether the whole pipeline executes in real-time or not. We therefore have to add new measurements for this.

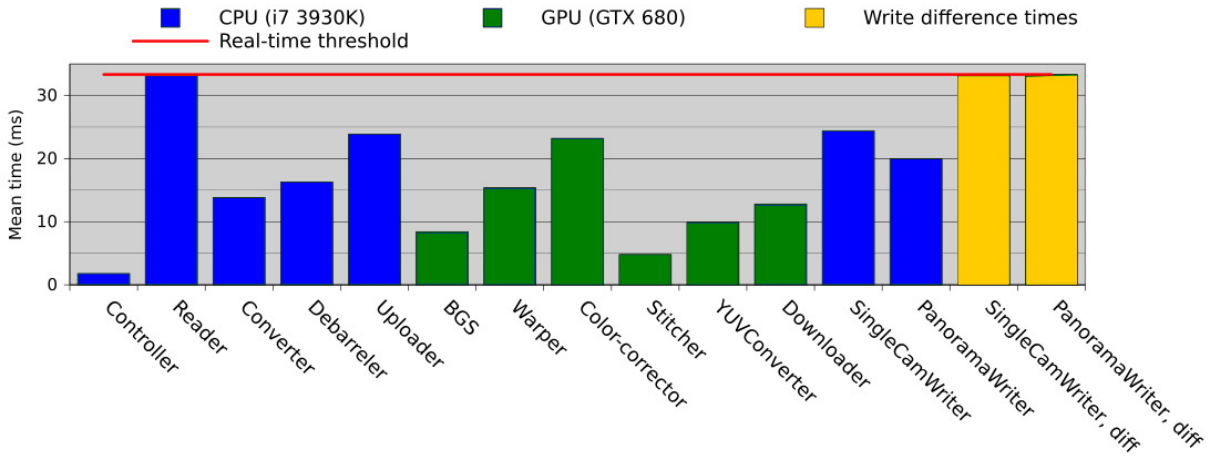


Figure 4.17: Overall pipeline performance

4.7.1 Write difference times

To prove that the pipeline is performing in real-time, we added two new measurements to the benchmark: the difference between the single camera writes and the difference between the panorama writes (marked in yellow in figure 4.17). These numbers measure the difference from when the previous write finished to when the next write finished. These differences are generally controlled by two factors: the camera frame rate and the pipeline performance. The camera frame rate sets the ideal mean times that the pipeline should fulfill. When performing fast enough, the whole pipeline must on each iteration wait for a new frame before continuing, and the mean write differences are then maintained at the real-time threshold. However, on slow iterations, the writer modules are not interrupted, and are allowed to finish. This means that the average write differences increase in these cases. We can from this see that the pipeline increase the mean write difference times when performing bad, but will not decrease these times below the real-time constraint when performing good enough. In other words, bad performance leads to a high pipeline frame drop rate, and affects the mean write differences directly by increasing the mean times. This is a good indication of whether the pipeline is performing in real time or not, because we need to have a new frame written to disk every 33 ms to keep up, which means that the write differences should be equal to the CamReader times. If any modules perform worse than this, the average write differences are quickly affected.

In figure 4.18, we can see the write differences of a 1000 frames run. According to this and figure 4.17, the average write differences are exactly 33 ms, which means that the new pipeline is performing in real time on this configuration.

4.7.2 Old vs. new

A comparison with the old pipeline can be seen in figure 4.19 and table B.2. The interesting modules to compare are the stitcher, warper and RGB to YUV converter. The remaining modules are not available for comparison either because the old pipeline does not contain such a module, or because both the old and new pipeline use the

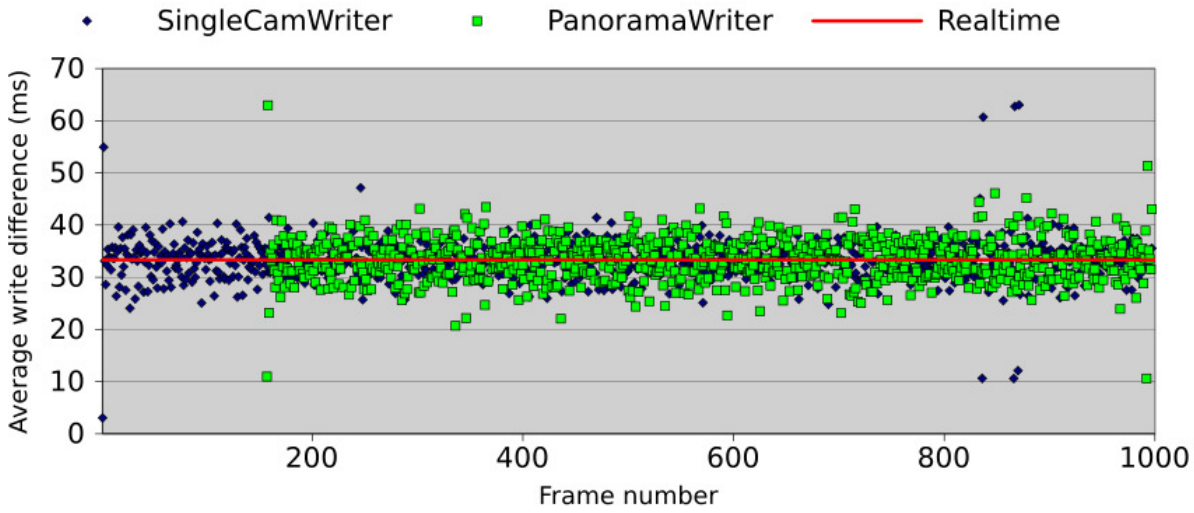


Figure 4.18: Pipeline write differences, 1000 frames run
Note that the delayed start of panorama writes is caused by the frame delay buffer described in section 4.4.4

same CPU implementation, such as the Debarreler. What we see is that all the comparable modules gain massive performance in the new pipeline by moving them to the GPU. The RGB/RGBA to YUV 4:2:0 converter has a performance increase of a factor of 2.7. At the same time, the main stitching operations, i.e. the warper and stitcher, have impressive performance gains of $8.8x$ and $106x$ respectively. Note also that the stitcher in the new pipeline is the improved and more advanced stitcher with a dynamic seam, while still having such good performance. These results can be explained by these modules having a larger degree of potential parallelism, due to the non-optimized, sequential iteration of pixels on huge frames in the old implementations. The Converter already uses the well optimized ffmpeg library, so even though it increases in performance, it is not with such a high factor. Like excepted, the overall performance increases massively compared to the old pipeline.

4.7.3 End-to-end frame delay

As mentioned in section 2.1, the time from an event or frame is recorded to when it is accessible on the system should be as low as possible. This pipeline frame delay, i.e. the end-to-end delay for a frame, is as low as 5.33 seconds in our improved pipeline, as long as the pipeline performs according to the real-time threshold. This number can be explained by frames moving through the pipeline one sequential module at a time. In addition to the ordinary modules, we have the frame delay buffer from section 4.4.4, of 150x4 frames. This means that a frame needs to be moved through 10 sequential modules, plus 150 steps in the buffer. As long as the pipeline performs according to the real-time constraint, frames are moved to the next step every 33rd ms. This gives us

$$(10 + 150) \times 0.033 seconds = 5.33 seconds.$$

This can safely be considered to be short enough for coaches to be able to use the

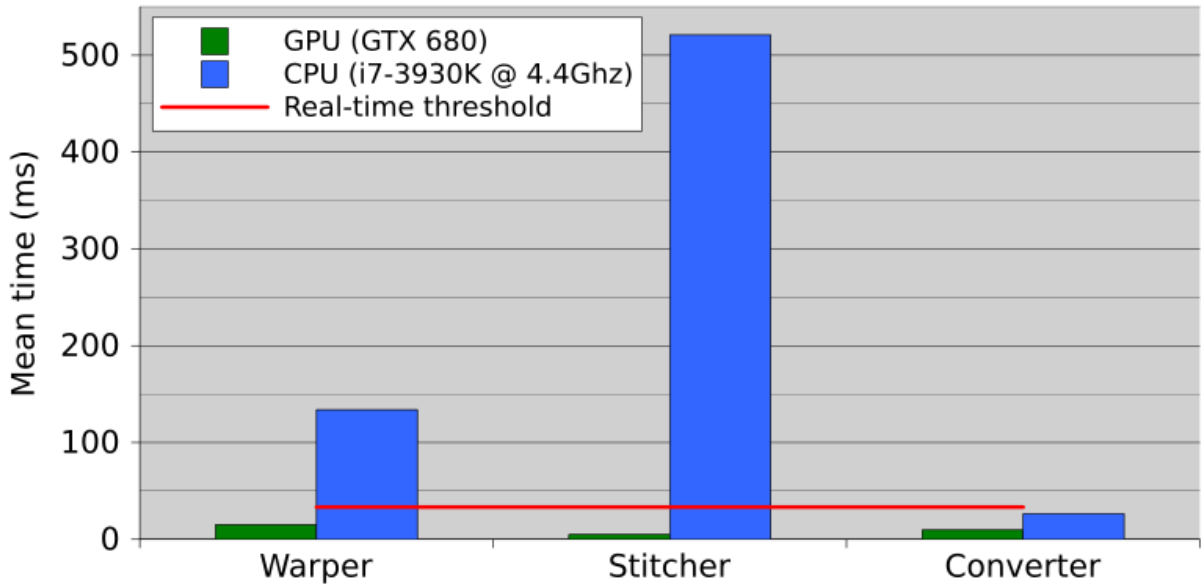


Figure 4.19: Old vs. new pipeline

system during half-time. Note that if the pipeline performs below the real-time threshold, this transforms into:

$(10 + 150) \times max(avg.\ SingleCamWriter\ diffs, avg.\ PanoramaWriter\ diffs)$

i.e. we use the maximum average write differences.

4.8 GPU comparison

When investigating the performance numbers, it is interesting to see what the results are for different architectures and generations of GPUs. We have therefore run performance benchmarks on DevBox 2, with different high end GPUs from different generations, for the sake of comparison. The results can be seen in figure 4.20 and table B.3.

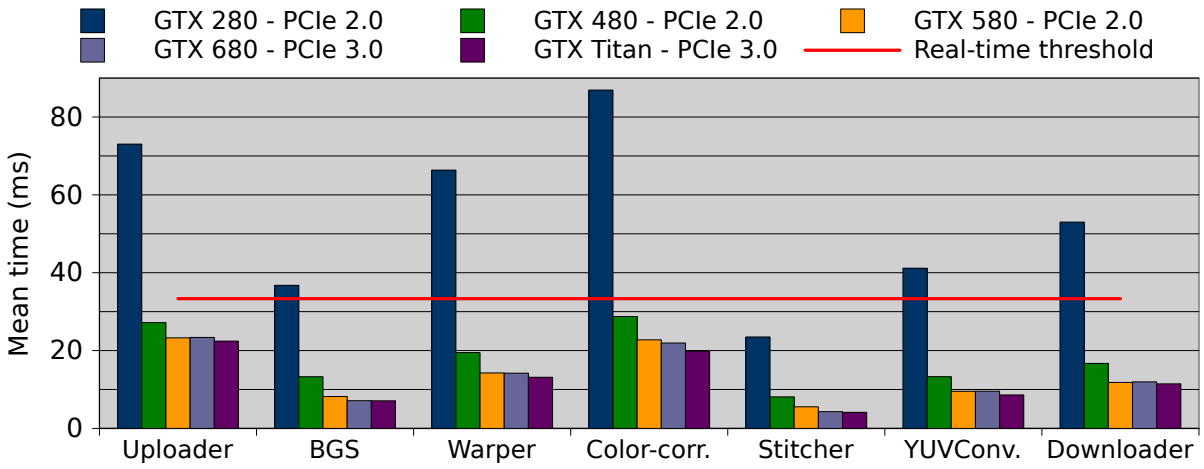


Figure 4.20: GPU performance comparison

There are few surprises in these results. We can see that the more powerful the GPU; the higher the performance. This is primarily caused by the increased frequencies, increased number of CUDA cores, and improved architectures. In addition, the GTX 680 and GTX Titan utilize PCI Express (PCIe) 3.0, compared to the GTX 280, GTX 480 and GTX 580 using PCIe 2.0, and some of the improved performance is also caused by this. However, we would expect a more noticeable performance increase when moving from PCIe 2.0 to 3.0, but this is not the case. This tells us that the PCIe bus, i.e the transfers between host and device, is not the bottleneck in the system. To verify this, the actual bandwidth usage can be calculated precisely. For the Uploader, the bandwidth usage is:

$$\begin{aligned} bandwidthusage &= ((1280 \times 960 \frac{pixels}{frame} \times 4 \frac{byte}{pixel} \times 4) \\ &\quad + (1280 \times 960 \times 4 \frac{byte}{frame})) * 30 \frac{frames}{second} \\ &= 737280000 \frac{byte}{second} \\ &= 737 \frac{MB}{s} \end{aligned} \tag{4.1}$$

For the Downloader, the bandwidth usage is:

$$\begin{aligned} bandwidthusage &= 6742 \times 960 \frac{pixels}{frame} \times 1.5 \frac{byte}{pixel} \times 30 \frac{frames}{second} \\ &= 291254400 \frac{byte}{s} \\ &= 291 \frac{MB}{s} \end{aligned} \tag{4.2}$$

We mentioned in section 3.4.1 that the PCIe 3.0 bandwidth is 16 GB/s in each direction. In other words, we see that the PCIe bus is nowhere near being the bottleneck, having approximately 15.3 GB/s spare bandwidth for the Uploader and 15.7 GB/s spare bandwidth for the Downloader.

We can clearly see that when running on a GTX 480, 580, 680 or Titan, the performance is good enough to fulfill the real-time requirements. However, we see that as soon as we move from a GTX 480 to a GTX 280, the performance decreases greatly, and is nowhere near real-time. This is because the GTX 480 and higher, of compute 2.0 or better, support concurrent kernel execution, as described in section 3.1. The GTX 280, however, uses compute 1.3, which does not support this. This means that only one CUDA kernel can be executed at a time, meaning that all kernel calls must be serialized, which greatly affects performance. Compute 1.3 cards are therefore too slow for the pipeline. Luckily, these are old cards, dating back to 2009 and earlier, and newer ones all support compute 2.0 or higher. In addition, it is interesting to note that the performance increase diminishes when using more powerful GPUs. This indicates that the load on the GPUs are not high enough for maximum GPU utilization.

4.9 CPU core count scalability

Another interesting topic when analyzing the performance of the pipeline, is to see how the performance scales with the number of CPU cores. When developing the pipeline, we had access to a server with the specifications in table C.4. As we can see from the specifications, it has 16 physical CPU cores, and was therefore very useful when analyzing the core scalability. By use of the *taskset* [55] command, we were able to test how the pipeline performed on a varying number of cores. We used a set of 4, 6, 8, 10, 12, 14 and 16 cores in our benchmark.

From the results in figure 4.21, we can see that the pipeline scales very well with the number of CPU cores. By increasing the number of cores from 4 to a maximum of 16, the mean processing times of each module seem to drop in a negative exponential fashion. In other words, it appears to be a practical maximum number of CPU cores, where adding more cores to the pipeline, does not increase performance noticeably. In the current setup on this computer, this number appears to be 12 cores. It is also interesting to note what modules are most affected by the increase in core number. From figure 4.21, we see that the modules gaining the most from the core count increase, is the Debarreler, the Uploader, and the writer-modules. This can easily be explained by the use of CPU threads. The Debarreler utilizes one thread per camera, the Uploader utilizes several threads due to the asynchronous, double buffered transfers, and both the Writer-modules contain an encoding part that massively utilizes threads.

The processing times of the BackgroundSubtractor's ZXY coordinate fetcher thread is not so interesting, due to it executing in a separate thread from the controlled module threads in the pipeline. In addition, the numbers here are not very surprising. No matter the CPU core count, the bottleneck of the thread appears to be the actual fetching of the data from the database. In other words: the BGS ZXY fetcher thread is bottlenecked by the performance of the ZXY database.

Note that GPU modules are irrelevant in these benchmarks, because they are not executing on the CPU. Also note that the performance of the controller is considerably worse on this computer than on DevBox 2 (approximately 4 ms vs 2 ms with all cores activated). This is primarily because of the CPU core frequencies. On the 16 core-server, the CPU has a core frequency of 2.00 GHz, while DevBox 2 runs at a CPU frequency of 4.4 GHz. The Controller is running as a single thread, so it does not scale well with the core count, but it scales very well with the CPU core frequency, as we can see from these results.

Another important aspect of these measurements, is the rate of frame drops in the CamReader module. When using 4 cores, we can from table 4.3 see that the camera driver drops 75 of 1000 frames, in other words 7.5%. However, as soon as we increase the number of cores slightly, the drop rate drops quickly until it stabilizes at 6-8 per 1000 frames at 8 cores and more. The average CamReader processing times are still 33 ms in every configuration due to the trigger box.

The pipeline frame drop rate is even more interesting. We can see that with 4 cores, when processing 1000 frames, the pipeline misses as much as 749 frames, i.e. 74.9%! This leads to a massive frame drift in the resulting video. The drop rate drops quickly when increasing the core count, until it stabilizes at 0-6 frames per 1000 for 10 cores and more. These few cases probably happen due to spikes caused by OS- and IO-interrupts.

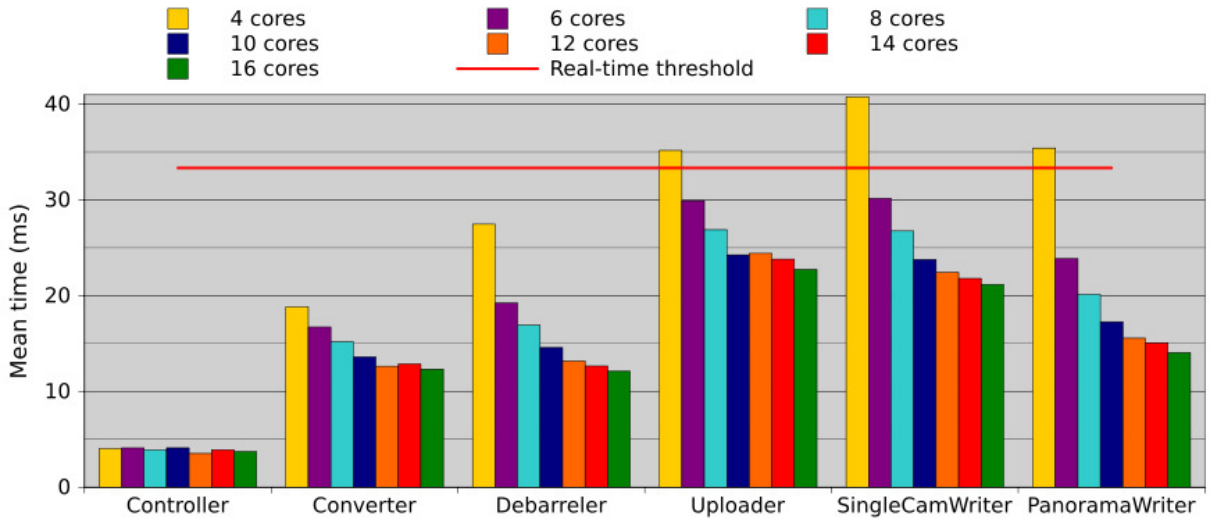


Figure 4.21: CPU core count scalability

	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores	16 cores
Camera frame drops	75	26	7	9	6	8	8
Pipeline frame drops	729	327	67	0	6	3	3

Table 4.3: CPU core count scalability, without frame drop handling, frame drops per 1000 frames processed

4.9.1 Write difference times

It is also important to notice the mean write difference times, as explained in section 4.7.1. From figure 4.22 based on the write difference times, we can see that the pipeline performs too slow with 4, 6 and 8 cores. However, when reaching 10 cores and more, the write differences reach real-time levels.

4.9.2 HyperThreading performance

In table B.5 and the corresponding figures 4.23 and 4.24, we can see the effect of HyperThreading on the performance. When using a lower number of cores, we get a massive performance gain in the modules consisting of many threads, such as the writer modules (due to the H.264 encoders). The other modules generally perform worse or equal to the HT-disabled counterparts. However, note the amount of frame drops, seen in table 4.4. When running 4 cores without HT, the pipeline drops an astounding 1203 frames when processing 1000 frames. This means that more than every second frame read is never processed, and the frame drift becomes extreme. The camera frame drop rate is also rather high, at 223 out of 1000 frame sets. When using 4 cores with HT, both frame drop rates decrease by a large margin. However, when increasing the number of cores, the performance increase caused by HT diminishes. When running 16 cores, the performance is overall decreased with HT enabled, where the only module that really gains anything is the PanoramaWriter. It therefore seems like it is better to disable HT when running more than 8 cores on this architecture.

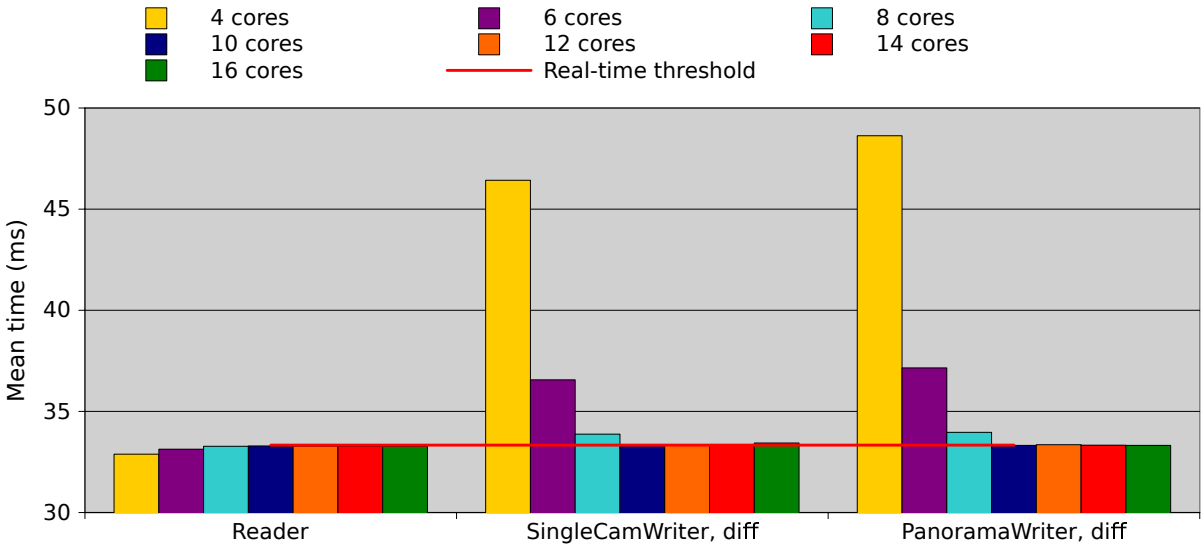


Figure 4.22: CPU core count scalability, write difference times

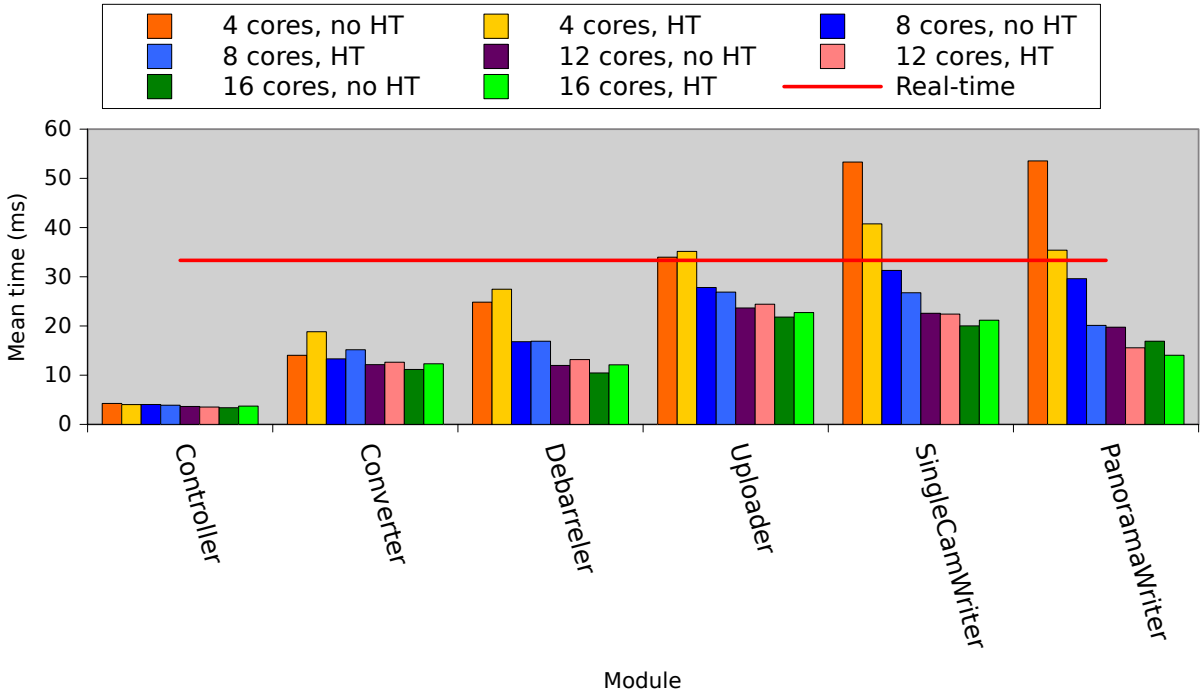


Figure 4.23: HyperThreading scalability

	4 cores, no HT	4 cores, HT	8 cores, no HT	8 cores, HT	16 cores, no HT	16 cores, HT
Camera frame drops	223	75	54	7	5	8
Pipeline frame drops	1203	729	477	67	3	3

Table 4.4: HyperThreading scalability, without drop handling, frame drops per 1000 frames processed

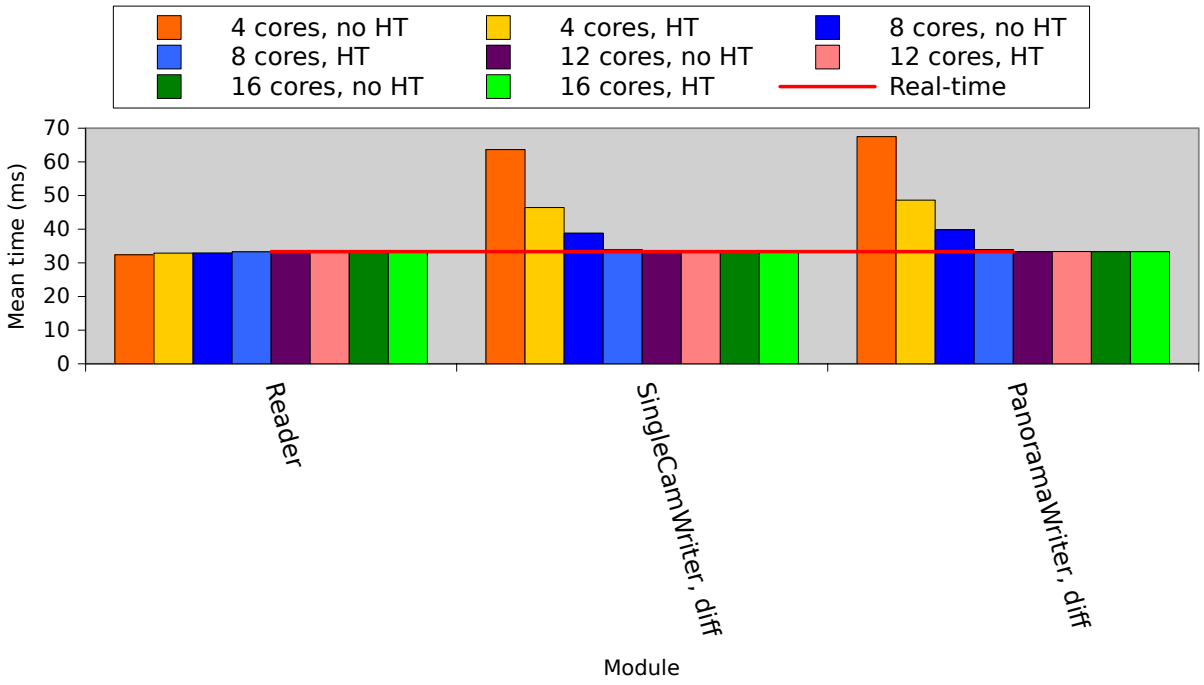


Figure 4.24: HyperThreading scalability, write difference times

By looking at these results, especially the mean write difference times, we see that running 4, 6 or 8 cores on this setup, with or without HyperThreading, is not a feasible solution. We can see that we should at least utilize 10 cores, and from the pattern in figures 4.23 and 4.24, we should optimally disable HyperThreading when using so many cores.

4.10 Frame drop handling performance

So far we have not benchmarked with the pipeline frame drop handling, described in section 4.4.5, activated. This is because this functionality affects the performance of the pipeline modules drastically when the CPU is overloaded. We generally want benchmarks with this function deactivated, because this gives us a better picture of the actual performance and processing times of each module, without direct interference from other components other than the load on the CPU and GPU. However, the pipeline will be installed in the real world with this function activated, so it is very interesting to look at this performance. The effect of enabling frame drop handling when looking at the CPU core count scalability, can be seen in table B.6 and figure 4.25.

The numbers are quite a bit different here than with the drop handling deactivated. First of all, we can see that the average processing times when running on few CPU cores has decreased by a big margin. Especially executing on 4 cores see a huge average performance boost. The mean performance then decreases when increasing the core count, but increases again after we have reached 8 cores. This might seem like an unexpected result, but is very logical. Like we have seen in section 4.4.5, when a module is supposed to skip a frame, it just ignores all processing for that frame. This

means that with an increasing amount of frame drops, the modules will more often return immediately. This of course, leads to the mean processing times decreasing. When running few cores, we can see that the amount of frame drops are huge, with up to 343 frames per 1000 dropped for 4 cores, due to the CPU being overloaded, leading to a situation where the pipeline needs to skip frames frequently. However, compared to the case with frame drop handling disabled, we see that this function reduce the amount of frame drops by approximately 50% in the worst cases, in addition to eliminating frame drifting. By increasing the core count, we offload the CPU, which decreases the frame drop rate. When reaching 10 cores, the frame drop rate stabilizes close to 0, and the performance now increases due to the increasing number of cores. This means that even though the average processing times are relatively low for 4 cores, this comes at the cost of video quality, which is severely degraded due to the need for reusing frames.

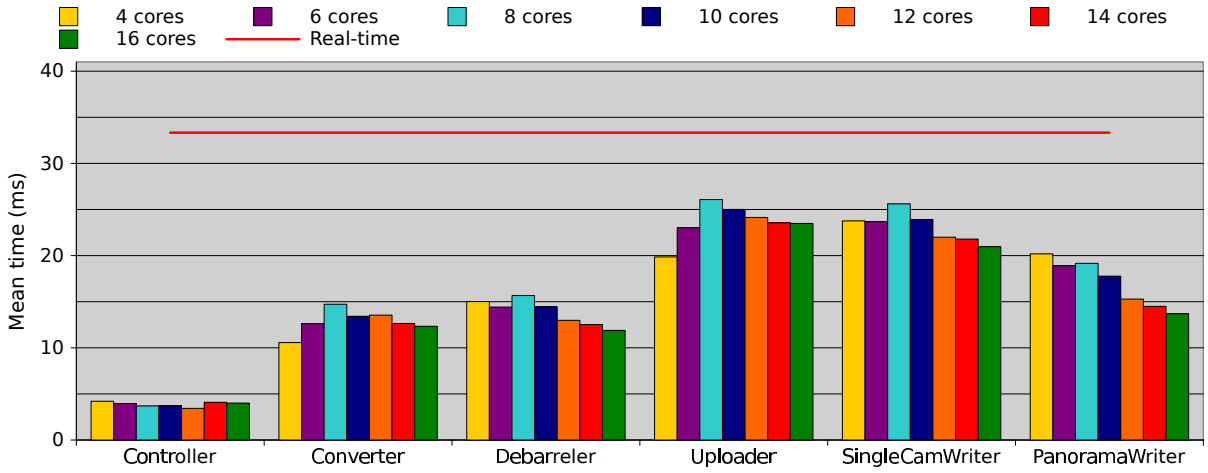


Figure 4.25: Frame drop handling performance

	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores	16 cores
Camera frame drops	41	33	7	4	3	4	4
Pipeline frame drops	343	177	37	6	2	7	3

Table 4.5: CPU core count scalability, with frame drop handling, frame drops per 1000 frames processed

4.10.1 Write difference times

Note in figure 4.26 that the mean differences between when each frame has been written to disk, is, like with frame drop handling disabled, too high when running 4, 6, and 8 cores, even when skipping frames. We saw in section 4.9 that the pipeline frame drop rate affects the writer difference times negatively, while a good performing pipeline can not bring the mean write difference times below the real-time threshold. This is also valid for when we have frame drop handling activated. This is because when dropping

frames, there is first an iteration with high mean processing times, followed by one or more faster iterations. However, these fast iterations, like we have seen, does not decrease the mean time below the real-time threshold, while the slow iteration increases it. However, compared to the benchmarks run with frame drop handling disabled, we can see that with it enabled, the mean write differences are closer to the real-time threshold, due to the drop handling potentially decreasing the amount of slow performing iterations.

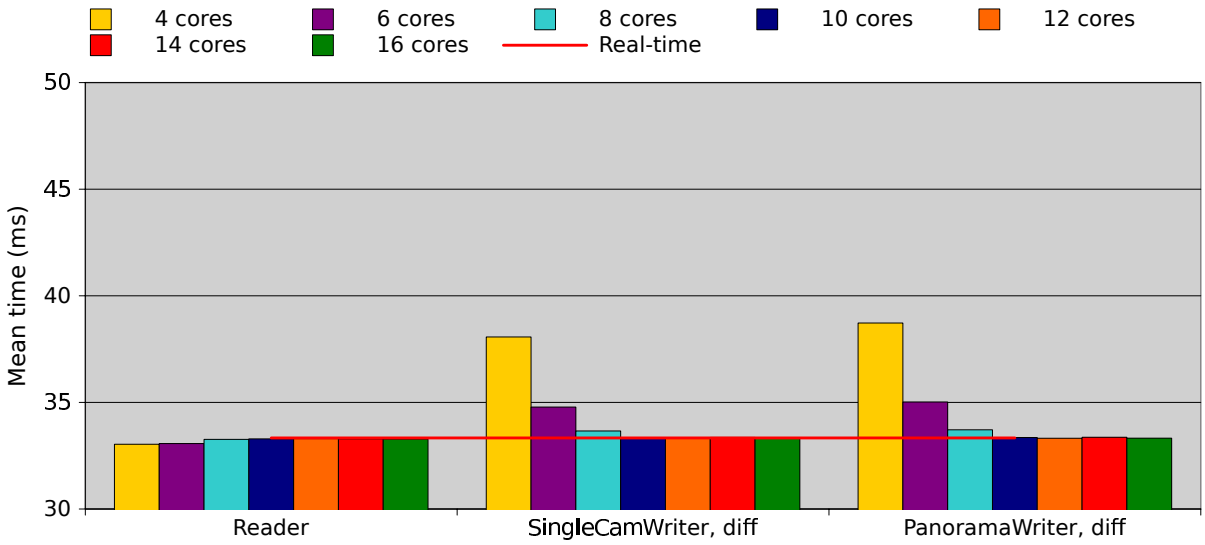


Figure 4.26: Frame drop handling, write difference times

4.11 CPU core speed comparison

In addition to analyzing the scalability of the pipeline with different number of CPU cores, we also want to test the performance at different CPU core frequencies. As we can see from table C.2, the machine installed at Alfheim, i.e. DevBox 2, contains an i7-3930K CPU, which runs by default at a core frequency of 3.2 GHz. By overclocking, we benchmarked the pipeline running on a CPU at 3.2 GHz, 3.5 Ghz, 4 GHz and 4.4 GHz. The module performance can be seen in figure 4.27. Here we see a linear decrease in processing times when increasing the frequency. Interestingly enough though, the controller does not gain much. When looking at the write difference times in figure 4.28, all configurations are performing according to the real-time constraint. However, due to the performance increase when overclocking, we decided to run the CPU on this machine at 4.4 GHz to both minimize the impact of processing spikes and generally provide larger margins.

4.12 The Bagadus web interface

The users of the recording pipeline are not supposed to have a very high computer understanding. In other words, the act of starting the panorama stitcher pipeline by

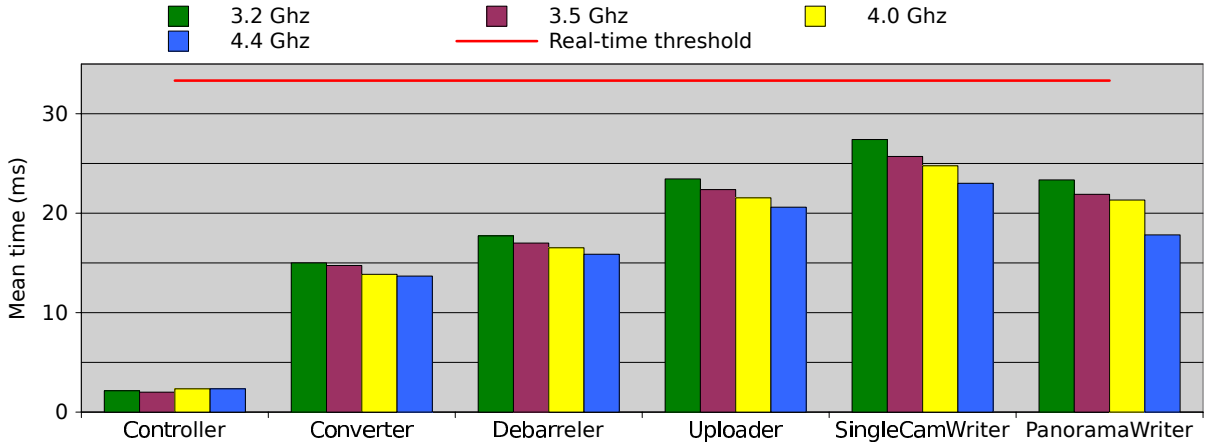


Figure 4.27: CPU frequency comparison

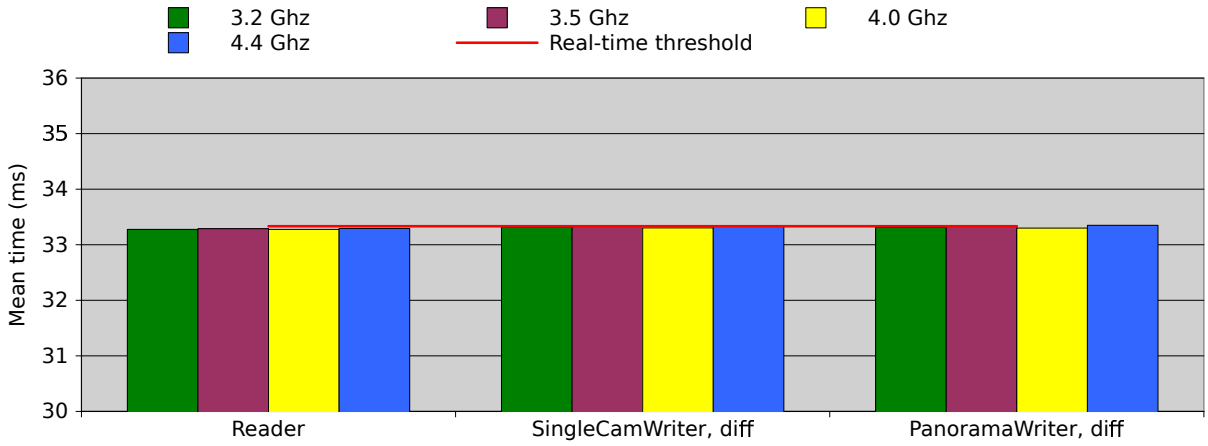


Figure 4.28: CPU frequency comparison, write difference times

use of a command line interface is not a realistic and user friendly alternative. This means that we need a better interface for administrating recording sessions. This is why we created a Bagadus web interface, for having a user friendly front end to the panorama pipeline.

The requirements for such a web interface can be summarized as:

- Users should be able to schedule a new recording session ahead of time. Parameters for such a recording should be:
  - The start of the recording, with minute precision
  - The length of the recording, specified in minutes
- The user should be provided with a list of scheduled recording sessions. The columns of this list should be:
  - The process ID (PID)
  - The time-stamp of when the recording was scheduled by a user

  - The time-stamp of when the recording is scheduled to start
  - The scheduled duration
- The user should be able to cancel and stop scheduled and active sessions
- The web interface should for the sake of client portability utilize standard web technologies
- The design should be logical and fairly modern

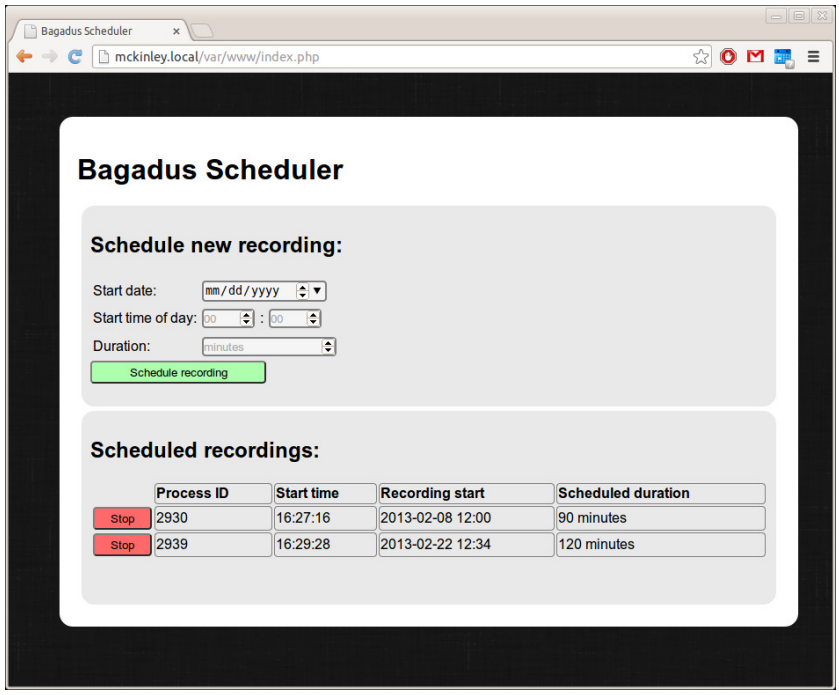
These requirements lead us to the web interface seen in figure 4.29. As we can see from the screenshots, users are able to specify the recording start and duration of a new recording session. When pressing the "Schedule" button, a PHP script launches a new PanoramaPipeline session, with the specified parameters. In addition, the user is provided with a list of active and scheduled sessions, with the columns specified in the requirements. An example can be seen in figure 4.29(a). This list is retrieved by use of the *ps* [56] and *grep* [57] commands, and then parsed in PHP. On every row in this list, the user is also provided with a "Stop"-button. When pressing this button, PHP sends a *sigterm* signal by use of the kill command [58]. This signal is captured by the corresponding pipeline process, which then terminates. An example of the result after stopping a scheduled recording can be seen in figure 4.29(b).

The implementation of this web interface use only common web technologies. We use Apache [59] as web server, running on the same machine as the panorama pipeline, and PHP for server side scripting. We utilize ordinary bash commands to retrieve and execute commands on the server. The client side only consists of HTML 5 pages and CSS to view the pages.

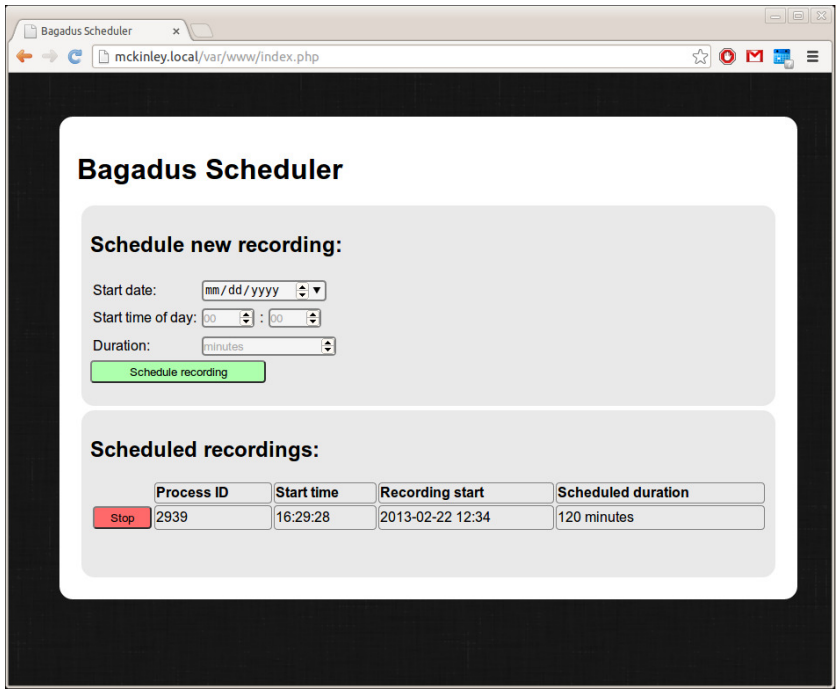
The design requirement must be viewed subjectively, but in the group working on the pipeline, we all agreed that the design was currently good enough for its purpose, and provided an easy to use interface for administrating recordings.

4.13 Issues and improvements

There are several issues and improvements in the new pipeline. First of all, the BGS's CPU part is executing as part of the Uploader. This means that the Uploader does tasks for two modules, which can lead to performance issues. A better solution would be to add a new BGS-based CPU module executing before the Uploader, and then passing its output to the Uploader. Another improvement is to add better configurability. Currently, the pipeline contains some hard coded values and parameters, but in the future, this should really be provided in XML files, or something equal. In addition, we currently only support four cameras, and want to support more in the future. We also want to use higher resolution cameras in the future, such as 2K and 4K cameras. This, however, leads to a large increase in the amount of data to process. Next, as mentioned in section 4.5.3, we still utilize the debarrel function provided by OpenCV. This implementation, however, is relatively slow, so a goal is therefore to re-implement the Debarreler with SSE3 to make it faster. It is currently fast enough, but if we could speed it up, we would be able to offload the CPU somewhat, which will lead to better processing times, and would be extra helpful when moving to larger data loads.



(a) Example of several recordings scheduled



(b) Result of stopping a scheduled recording

Figure 4.29: The Bagadus Scheduler web interface

This would also result in larger margins for processing spikes, potentially leading to fewer frame drops. Furthermore, the pipeline currently contains a 50:50 amount of GPU-based and CPU-based modules. However, many of the tasks currently done on the CPU can be moved to the GPU for better performance, such as the H.264 encoding

and YUV 4:2:2 to RGBA converter.

In addition, as part of making the pipeline more scalable to larger processing loads, we want to explore the possibilities of using multiple GPUs. One way of doing this is to split the modules between different GPUs. Then the data could be transferred between the GPUs via GPUDirect. Another way would be to splice all the frames into n number of slices, and then process these on n different GPUs. To reduce the need for inter-GPU-communication, we would need to cut the frames horizontally. There would be some issues here, though, such as for the dynamic stitching, where we would have to communicate between the GPUs to ensure that the dynamic cut ends and starts at the correct pixels when going from one slice to the next. A quick and naive solution for this, would be to run the modified Dijkstra-algorithm internally within every frame slice, and not globally for that frame. The start and end pixel of the algorithm within each slice would be selected on system setup. This has weaknesses, however. For instance, if a player is located in one of these pixels, the seam would have to cut through him/her.

It would also be interesting to modify the pipeline using expansion cards delivered by Dolphin Interconnect Solutions [60], which allows for cheap and easy distribution of the processing load to several computers.

Furthermore, we would like to investigate the effects of changing the internal pixel representation in the pipeline from RGBA to RGB. We currently use RGBA to promote coalesced memory accesses on the GPU, but this comes at the cost of 33% more data to be transferred over the PCI Express (PCIe) bus. As we saw in section 4.8, the PCIe bus is currently not the bottleneck. However, for better utilization of the PCIe bus, it is interesting to investigate the effects of changing to a more compact representation. Related to this, it would be interesting to investigate the possibilities of using YUV 4:2:0 as the internal pixel representation. YUV 4:2:0 is more compact than RGBA, but research must be done to discover how much code that need to be rewritten, and what the impact on CUDA kernel performance would be. This must either way be investigated when increasing the camera resolution to 2K, because of changes in pixel representation in the cameras.

Furthermore, in section 4.5.10 we shortly discussed that converting from RGBA to YUV 4:2:0 in two steps is inefficient, and an improvement of the pipeline is therefore to reimplement this to do this conversion in one step. Lastly, the camera setup currently in use is not optimal, and results in lots of artifacts, such as parallax errors (see [2] for a description) and inaccuracies when debarreling and warping images. If we could improve the camera setup, parallax errors would be reduced, and we could possibly cover the field with not as wide angle lenses, which would result in less barrel distortion, which again would lead to better debarreling, warping and stitching results.

4.14 Summary

In this chapter, we have seen how we were able to create a pipeline for generating stitched panorama videos in real-time for the Bagadus system. As part of this, we started by discussing how we have improved the old setup, and explained the general architecture of our pipeline, including initialization, in depth analysis of the controller,

general module design, frame drop handling, and more. We continued by going into details about every module in the pipeline, from the CamReader responsible for capturing frames, to the Writer modules, responsible for writing videos as H.264 encoded files to disk. The next step was to analyze the performance of the pipeline, where we discussed the general performance, the scalability in respect to both CPU speed and CPU core count, and compared the performance on different generations of GPUs. To make the pipeline more usable for end-users, we explained how we designed and developed a web interface for managing recording of sessions. This all makes the pipeline completely automatic, with small and inexpensive hardware, and recording can easily be started by a single user using the web interface. We rounded up the chapter by discussing current issues and future improvements.

In the next chapter, we will go much more into detail about background subtraction, which we use as a tool in the improved pipeline.

Chapter 5

Background Subtraction

5.1 What is Background Subtraction?

Background substraction (BGS) is a tool in image analysis and processing, used for extraction of the background from a series of sequential image frames so that we know what pixels can be considered as part of the foreground, and what is the background. BGS applications take video frames as input, and then provides foreground masks that tell whether a pixel is foreground or background as output. BGS is for instance very useful for surveillance applications where there might be installed CCTV cameras to monitor an area. By use of BGS, the system is able to know what is part of the background, such as trees, houses, and pavement, while it also knows what is part of the foreground, such as cars, moving persons, etc. This can for example be used for systems that alert security if there are observed objects of a certain size, such as a human, which can reduce the need for manual surveillance of CCTV footage.

5.2 Related work

There have been done a lot of work on background subtraction, and the past years, many new BGS methods have been developed, all with their own characteristics, advantages and disadvantages. There exist several types of BGS models, where the most basic type is the frame difference model, which calculates the difference in pixels between the current and the previous frame to classify pixels as foreground or background. This is a relatively weak model, however, and is not considered very robust. Another type of model is the mean filter model, where the background is the mean of the n previous frames. An example of such a model is [61]. Next, we have the median filter-based model, where we instead of using the mean of the previous frames, use the median of the previous frames. An example is [62]. These models are fairly simple, and much work has been done on researching better ones. Examples of more advanced BGS models are Gaussian Mixture-based models, like [63] [64] and [65], non-parametric models such as [66] and [67], kernel density estimator models [68], models using Eigenbackgrounds [69], and codebook-based models, such as [70] [71].

As we can see, there exist a huge variety of BGS models of different classifications, and comparing all of them to find their strengths and weaknesses is a big task. Such

a comparison was done by Brutzer, et. al. in [72], where they compare several of the modern background subtraction models, and discuss how they perform in different scenarios under varying lighting and noise conditions. We will use this paper as a basis for our evaluation and selection of BGS model for use in this thesis.

5.3 Background Subtraction Challenges

Brutzer, et. al. refer in [72] to several challenges that makes it harder for a BGS model to return the correct results. These challenges are important to be aware of for us, so that we can pick the most fitting model for our scenario. Therefore, let's take a look at these challenges, and their importance.

- **Gradual illumination changes:** We want the BGS model to be able to handle gradual changes in illumination, such as how the lighting changes during the day outside. This challenge is important in our case, due to the stadium being outside with difficult lighting conditions in Tromsø. Our BGS model must therefore handle this well.

- **Sudden illumination changes:** There might be cases where the BGS model needs to handle sudden changes in illumination. This means that the model needs to properly handle sudden lighting changes, such as when turning of a light in a room. This challenge is not that important for our scenario, because the sun is the major factor in our case, and the stadium lights will be on during a whole match, which means that large changes in lighting will in the most cases happen gradually. We need to be aware of this of course, but there are more important challenges.

- **Dynamic background:** Another problem is the possibility of having a dynamic, i.e. changing and moving background. This could be the case with trees moving in the wind, escalators, etc, which should not be considered as foreground, even when they are moving. This challenge, however, is not very important for us. The football field is very static, and the grass on the field is so short that it does not sway in the wind. There is some movement on the stands, but they are not a part of the field, and can luckily be solved quite easily, as we will see later, by simply ignoring everything but the field itself when doing the background subtraction. However, a challenge rather equal to this is changing weather conditions (see below).

- **Camouflage:** There might be situations where players wear shirts with approximately the same color as the field itself, or possibly the lines in the field. This will give us problems when trying to subtract a player from the background. We therefore need to be aware of this problem when selecting a BGS model.

- **Shadows:** Shadows are an important challenge. When subtracting the background to retrieve the players, we do not want to mark the player's shadows as foreground. It is therefore optimal to select a model that ignores shadows, or possibly marks shadows as a separate value in the foreground mask.

- **Bootstrapping:** Bootstrapping is in BGS the action of getting the system to properly understand what the "empty" background looks like, i.e. how the background looks without any foreground subjects. In a busy scenario, for instance when monitoring a highway, this is difficult, because we cannot simply initialize the system with a photo without any cars. In our scenario however, this is very easy, because we can easily record frames of the empty field, due to it not being busy all the time.

- **Video noise:** Video noise, such as compression artifacts, sensor differences, etc, might provide challenges. However, according to [72], this is not really a problem, but rather the opposite. Brutzer, et. al. noticed that video noise actually made some BGS models work better than they originally did. This is generally not a problem in our scenario.

In addition to these, we would like to add another challenge, which might be an issue in our case:

- **Weather conditions:** In our case, weather conditions might give us many challenges. Weather conditions such as snow, rain, thunderstorms, etc., will result in different lighting conditions, color changes, changing lighting conditions, noise, etc., and needs to be taken care of. Some of this might be solved by providing several initialization frames for the different conditions, such as with snow on the field, but this does not cover everything, for instance moving snow.

5.4 Selecting a BGS model

The Zivkovic-model from [65], by Zoran Zivkovic and F. van der Heijden, is mentioned in [72] as one of the most promising BGS models, and when studying the challenges mentioned above, it seems very suitable for our scenario. It is a Mixture of Gaussians (MOG) model, also called Gaussian Mixture Model (GMM), and is implemented as part of OpenCV, named BackgroundSubtractorMOG2.

As we can see explained in [73], in MOG-based models (many models, such as Zivkovic, are based on the paper by Stauffer and Grimson [63]), each pixel is modeled as a mixture of adaptive Gaussians. On each iteration of the model, i.e. for each frame, these Gaussians are considered belonging to the background as long as they are the ones with the least variance, while also having the most supporting evidence. Pixels not matching any of these background Gaussians are classified as foreground pixels. MOG-based models also contain a history of these Gaussians, and when a pixel is not considered part of the background, the least probable Gaussian, i.e. the one with the highest variance/least supporting evidence, is replaced with a new one.

There is another MOG based BGS model implemented in OpenCV. This is the model by KaewTraKulPong and Bowden [74], named BackgroundSubtractorMOG in OpenCV. The KaewTraKulPong-model is not mentioned in [72], but because it is also a MOG-based model, together with the fact that it is implemented in OpenCV, which means that it is easy to test, we want to compare this one to the Zivkovic model to to see which one is best in our scenario.

There are several other BGS models of different classifications already implemented in OpenCV, but the documentation in most of these are non-existent. In addition, because Zivkovic is a GMM and performed so well in [72], we want to compare it to another GMM, in case MOG based models prove to perform well in our scenario, plus that the parameters are very equal, making direct comparison easier. We therefore only compare these two models in this thesis.

(a) Input frame, camera 2

(b) Initial Zivkovic model results

(c) Initial KaewTraKulPong model results

Figure 5.1: Initial BGS model comparison

In figure 5.1(b), we can see the initial results of the Zivkovic-based implementation. Black pixels are background, while the white pixels are the foreground. We can see that the results are quite accurate. First of all, approximately all player pixels are marked as foreground. Secondly, we see that it supports shadows, where the shadows are marked in gray, which in addition seems to be correct. However, there is some noise in the image, especially in the stands.

The first results of the KaewTraKulPong implementation can be seen in figure 5.1(c).

The results are not very impressive, especially when compared to the Zivkovic results. There is not much noise in the frame, but this comes at the cost of only a small part of the player pixels being classified as foreground, which basically makes the default parameters useless for the KaewTraKulPong implementation.

5.4.1 Parameter selection

As we can see, the initial results with the default parameters need to be improved to properly be able to compare between the models and select one. The next step is therefore to tweak the parameters of the algorithms to get better results.

Zivkovic model

To look at the parameters selected, we have to briefly describe the steps of the Zivkovic GMM algorithm. The first step of such a GMM algorithm, is to classify each new sample $\overrightarrow{x}^{(t)}$ as foreground or background. The sample is considered background if

$$p(\overrightarrow{x}^{(t)}|X_T, BG) > C_{thr} \tag{5.1}$$

where X_T is the history of previous samples, BG is the background, and C_{thr} is the background threshold. This is approximated to be

$$\hat{p}(\overrightarrow{x}^{(t)}|X_T, BG) \sim \sum_{m=1}^{B} \hat{\pi}_m N(\overrightarrow{x}; \hat{\overrightarrow{\mu}}_m, \sigma_m^2 I) \tag{5.2}$$

where $\hat{\overrightarrow{\mu}}_m$ are estimates of the mean values, σ_m^2 are estimates of the variances, $\hat{\pi}_m$ are the weights and I is the identity matrix. The formula for B is

$$B = \arg\min_b \left(\sum_{m=1}^{b} \hat{\pi}_m > (1 - C_f) \right) \tag{5.3}$$

where C_f is a measure of the maximum portion of the data that can belong to the foreground without affecting the background model.

The next step, is then to update the density model of both foreground and background. This is done by

$$\hat{p}(\overrightarrow{x}^{(t)}|X_T, FG + BG) \sim \sum_{m=1}^{M} \hat{\pi}_m N(\overrightarrow{x}; \hat{\overrightarrow{\mu}}_m, \hat{\sigma^2}_m I) \tag{5.4}$$

where M is the number of components. The weight, $\hat{\pi}_m$, is the amount of the data belonging to the mth component. It is updated using the formula

$$\hat{\pi}_m \leftarrow \hat{\pi}_m + \alpha(o_m^{(t)} - \hat{\pi}_m) - \alpha c_T \tag{5.5}$$

where the alpha is $\alpha = 1/T$. T is here the reaction time, in frames, we want on changes in the samples. $o_m^{(t)}$ denotes the ownership of the sample, i.e. what component it belongs to.

To update $\hat{\vec{\mu}}_m$, the following equation is provided:

$$\hat{\vec{\mu}}_m \leftarrow \hat{\vec{\mu}}_m + o_m^{(t)}(\alpha / \hat{\pi}_m)\,\vec{\delta}_m \tag{5.6}$$

where $\vec{\delta}_m = \vec{x}^{(t)} - \hat{\vec{\pi}}_m$

The equation for updating $\hat{\sigma^2}_m$ is:

$$\hat{\sigma^2}_m \leftarrow \hat{\sigma^2}_m + o_m^{(t)}(\alpha / \hat{\pi}_m)(\vec{\delta}^T{}_m \vec{\delta}_m - \hat{\sigma^2}_m) \tag{5.7}$$

The last step is to update the background model, i.e. $p(\vec{x}^{(t)}|X_T, BG)$. This is done by using equation 5.3 to select the components of the GMM that belong to the background

One of the most important parameters in these equations, is the alpha, which we saw in equation 5.5. As mentioned in [65], the alpha detemines the update speed when pixels change. The alpha should be as small as possible to ensure stability in the model, resulting in less noise in the result, but it should also be high enough to react fast enough on sudden changes. As stated, Zivkovic provides the formula for alpha calculation as $alpha = 1/T$. T is here the reaction time we want on sudden changes, measured in frames. If the alpha is too high, players in the field standing still would quickly fade into the background until they start moving again. This is clearly unwanted, as it is not uncommon for players to stand still. However, if the alpha is too low, sudden changes in light, such as lightning strikes, will result in lots of unnecessary noise. However, as we mentioned in section 5.3, in our case, there are few sudden changes, so we want the alpha to be fairly low. Empirically, we found a T of 500 frames, and therefore alpha of 0.002, to be sufficient.

Next, we need to select the threshold for what is considered background. This is the C_{thr} parameter in equation 5.1. If the background model calculation for a pixel gives us a value higher than C_{thr}, the pixel is considered background. I.e. the higher the threshold, the more pixels are considered foreground, and vice versa. We arrived empirically at 0.1 being a good threshold.

Another important parameter is the threshold on the squared Mahalanobis distance used to decide if a pixel is well described by the background model. The squared distance from the mth component is given by $D_m^2(\vec{x}^{(t)}) = \vec{\delta}^T{}_m \vec{\delta}_m / \hat{\sigma^2}_m$. Here we stick with the typical value 4σ, i.e. 4×4, which works well. We also need to set the threshold on the squared Mahalanobis distance used to decide sample ownership when they are close to existing components. If there are no existing components nearby, a new one is created. A small threshold results in generation of more, small components, while a larger threshold leads to fewer but larger, possibly too large, components. The implementation of the model suggests using 3σ, i.e. 3×3. By tweaking we also found this to be sufficient.

Next, we need to set the initial standard deviation for newly generated components. Empirically, we found that a variance of 30, and therefore standard deviation of approximately 5.5 worked best. The next parameter, C_T from equation 5.5, concerns complexity reduction. It is related to the number of samples needed to accept that a component exists. Here we have found that the value of 0.05 which is used in the existing implementation, works well. When selecting the maximum number of gaussians,

we stick to the default number from the paper and implementations, i.e. 5 gaussians, which proves to be sufficient. We also want to select the maximum number of modes in the model. This parameter is not the most important one, and we just use the default value from the implementation, i.e. 4, which proves to be good enough for us.

The last parameters concerns shadow detection. We want to use shadow detection, so first of all, we need to enable this in the model by setting *bShadowDetection* to 1 on initialization. Next, we want to modify the shadow threshold, named *tau* in the implementation. *Tau* is a threshold of how much darker the shadow can be before it is considered to not be a shadow any more, based on [75]. For example, taken from the implementation, 0.5 means that the shadow can be up to 2 times darker. With a too high threshold, all of the pixels not considered background will be classified as foreground, even though some of them actually are shadows. Selecting a too low threshold will classify actual foreground pixels as shadows. We therefore had to test and tweak this value, and found that the optimal value for us was 0.2.

A list of the final parameters can be seen in table 5.1. The results of tweaking the parameters can be seen in figure 5.2(a). The improvement is not that big, but that is because the Zivkovic model performed rather well with default parameters.

Model	Zivkovic
α	0.002 (T = 500)
C_{thr}	0.1
Initial variance	30
Max gaussians	5
C_T	0.05
Mahalanobis thr. (well described)	4x4
Mahalanobis thr. (ownership)	3x3
Max modes/components	4
bShadowDetection	1
Shadow threshold	0.2

Table 5.1: Tweaked Zivkovic model parameters

KaewTraKulPong model

As mentioned, the KaewTraKulPong model [74] is also a GMM, so it therefore uses many equal equations and has many of the same parameters as the Zivkovic model. The main difference between the models are how they are updated on each new sample. However, we will not go into details about the equations in the model here, because the general parameters are equal.

To begin with, the window size parameter is the size of the history, i.e. the maximum number of frames that the model needs to remember when calculating new foreground masks. This is related to the *alpha* value in the Zivkovic model, where the T in the *alpha* formula is equal to the window size in the KaewTraKulPong model. In other words, the larger the window size, the more stable the model stays, while the lower the window size, the faster it responds to sudden changes. For our scenario, we found a window size/history of 10, which equals an *alpha* of 0.1, to be good. This

differs quite a bit from the T value we found for the Zivkovic model, but increasing the window size resulted in less accurate foreground masks.

As for the Zivkovic model, a maximum number of gaussians of 5, which is the default, proved to be sufficient. The background threshold selected, differs a bit from the background threshold from the Zivkovic model, i.e. C_{thr}. In the KaewTraKulPong model, we found 0.3 to be the optimal value, which resulted in a low amount of noise, at the same time as marking enough of the players as foreground. Lowering the threshold resulted in a substantial increase of noise. The initial variance was found to optimally be 5, while the standard deviation threshold used for deciding whether a sample is part of the BG model or not, was best at 3.5. These are also somewhat different from what we found to be optimal for the Zivkovic model. The initial weight was found to be 0.05. Finally, the KaewTraKulPong implementation has a parameter named *minArea*. This parameter sets the minimum allowed bounding box of connected pixels classified as foreground. However, it proved to not have any effect on the results, and was therefore set to 1.

The parameters are summarized in table 5.2. As we can see, even though both the Zivkovic and KaewTraKulPong models are GMMs, the KaewTraKulPong implementation in OpenCV does not allow for parameter selection as detailed as for the Zivkovic implementation. This limits the strength of the model somewhat.

Model	KaewTraKulPong
α	0.1 (T = 10)
C_{thr}	0.3
Initial variance	5
Max gaussians	5
Initial weight	0.05
Std. dev. threshold	3.5
minArea	1

Table 5.2: Tweaked KaewTraKulPong model parameters

The tweaked parameters gave us a much more accurate result, like we can see in figure 5.2(b). All the players are now very visible, including the shadows. We can also see that there is some noise from the white lines, and the stands and sides of the field. Note, however, that optimally, the initial background image should be a clean frame without any foreground objects, but the results seen here are the results of an initial background image containing foreground objects, i.e. players.

5.4.2 Background subtraction model comparison

The comparison of the two BGS algorithms is shown in figure 5.2. The KaewTraKulPong-algorithm is shown in figure 5.2(b), while the Zivkovic algorithm is shown in figure 5.2(a). As we can see, both algorithms are fairly accurate, with some noise in both cases.

The most significant difference, is the shadows. In the Zivkovic algorithm, we can see that the shadows are marked as shadows by use of the gray color, while in the

KaewTraKulPong-algorithm, shadows are always marked as foreground. The original KaewTraKulPong-model contains shadow detection, but the version implemented in OpenCV does not support this. This is a huge advantage in the Zivkovic-based solution, because we can then choose to ignore or accept the shadow as foreground, depending on later stages in the Bagadus pipeline. However, we can also see that the Zivkovic algorithm has some more noise and errors in the shadowy areas, especially in the cases where the shadow of a player crosses the line of the field. Here we can see that the KaewTraKulPong-algorithm has less noise. Zivkovic here notices parts of the shadow, but does not mark it as a shadow, which it should have. This is, however, not that important, because we are more interested in the foreground than the shadows.

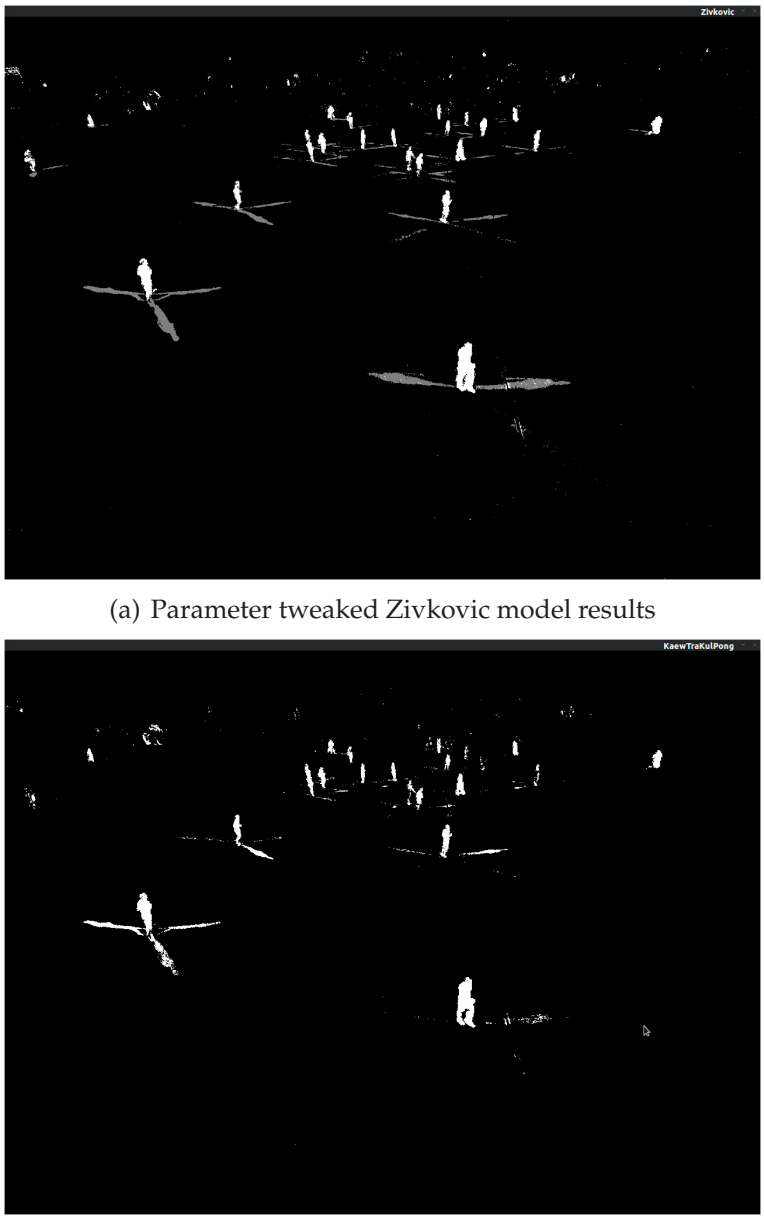
Overall, the Zivkovic model results in a bit more noise, but it is not by a huge margin, and is primarily in the stands area. Furthermore, it is interesting to note that the general accuracy of the Zivkovic algorithm is higher than that of the KaewTraKulPong-algorithm. We can especially see this when looking at the players. We can see that KaewTraKulPong marks less of the players as foreground, while the Zivkovic one marks approximately 100% of each player as foreground. Furthermore, due to the small window size of the KaewTraKulPong-solution, reaction times are so short that the model can potentially mark players as background when they are standing still for short amounts of time. However, when we increased the window size, the amount of noise increased substantially, so this is not optimal. This greatly pushes the selection of algorithm in favor of Zivkovic.

We are also interested in comparing the performance of the algorithms. During this whole chapter, we will be using DevBox 3, with the specifications seen in table C.3, for performance comparison. The results of a benchmark consisting of 9000 frames, i.e. 5 minutes of play can be seen in table 5.3.

Model	KaewTraKulPong	Zivkovic
Min	39.998	50.444
Max	66.423	106.808
Mean	48.947	79.345
Standard deviation	2.793	4.946
Variance	0.008	0.0245

Table 5.3: BGS model performance (ms)

We can see that the Zivkovic solution is substantially slower than the KaewTraKulPong solution, at an average of 61.7%, with a higher variance and standard deviation. This is quite a bit, but neither of the algorithms are currently real-time in the initial implementation (both are using longer than 33 ms per frame). However, if we could get both algorithms to perform in real time, the Zivkovic algorithm would be the preferred one due to the better BGS accuracy, including shadow detection. We therefore select the Zivkovic solution as our starting point. We will look into performance optimization, and see how we can make Zivkovic run in real-time by use of ZXY tracking data, while reducing the amount of noise and maintaining accuracy.



(a) Parameter tweaked Zivkovic model results

(b) Parameter tweaked KaewTraKulPong model results

Figure 5.2: Parameter tweaked BGS model comparison

5.5 Optimization of the BGS process by use of ZXY player data

We have looked at how we can implement a background subtractor quickly, by using one of several existing algorithms. However, like we saw in the previous section, the performance is far from real-time, and the visual results contain a substantial amount of noise in certain areas. We therefore have found a way to exploit the knowledge we have about player positions in our scenario.

5.5.1 The idea

The basic idea of this BGS optimization is to exploit the fact that we know all the players' positions at all times, due to the ZXY sensor belts. In [2], a mapping and translation between the coordinate system of the ZXY sensor data and the camera and panorama planes were discussed. By using this translation, we can convert the real world ZXY coordinates into pixel coordinates in the panorama plane. When using these coordinates, we will for every frame know the pixel location of each player. We know that humans have a limited maximum size, so by knowing their positions, we can simply ignore processing pixels that do not belong to a player, including a safety margin at each side of a player. This means that we can reduce the amount of pixels to be processed by a substantial amount. In addition, it allows us to reduce the amount of noise, such as in the stands area, which we have seen in earlier sections contains lots of noise. Another advantage of using this modification, is on sudden light changes, for instance on lightning strikes. In many BGS models, this sudden change in lighting conditions will be interpreted as foreground, which means that the whole frame suddenly is marked as foreground. With the ZXY modification, only the areas around the players will be affected, while the rest of the frames correctly remain classified as background.

For easier reference, we name our optimization the ZXY Background Subtractor (ZXY BGS). The general code structure can be explained by the following pseudo code:

Loop through all Y pixels ($y = 0 \Rightarrow y = frameheight$)

> **Loop** through all X pixels for the current y ($x = 0 \Rightarrow x = framewidth$)
>
> **If** pixel [x, y] is close to a ZXY player position, analyse the pixel using Zivkovic,
>
> **else** mark the pixel as background

5.5.2 First, naive ZXY BGS implementation

As stated earlier, we have selected the Zivkovic algorithm for basis of our modification. The first version of our ZXY BGS is a relatively naive solution. Here we loop, like in the Zivkovic algoritm, through all pixels. Then, on each pixel, we loop for all players through the corresponding coordinate samples. If the pixel is within one of the player frames with a static safety margin of 100 pixels, we process it, else we mark it as background immediately and continue with the next pixel, without any more processing.

The result is both good and bad. The good part is that it works fairly well, like we can see in figure 5.3, where we see that we now only analyze the area around each player, which reduces the amount of noise by a fair amount. However, the implementation is *much* slower than the unmodified Zivkovic implementation. The naive first implementation gives us the performance results seen in table 5.4.

The performance is approximately ten times slower than that of the KaewTraKulPong algorithm. By use of Intel VTune [76], we find that the cause of this is the new *pixelIsPlayer()* function. This function basically loops through all corresponding player coordinates that have been retrieved from the database and cached, and then checks whether a specified pixel is equal to one of these pixels, +/- the safety margin. This function therefore loops 20 times (once per player with a sensor) *per pixel*, giving huge

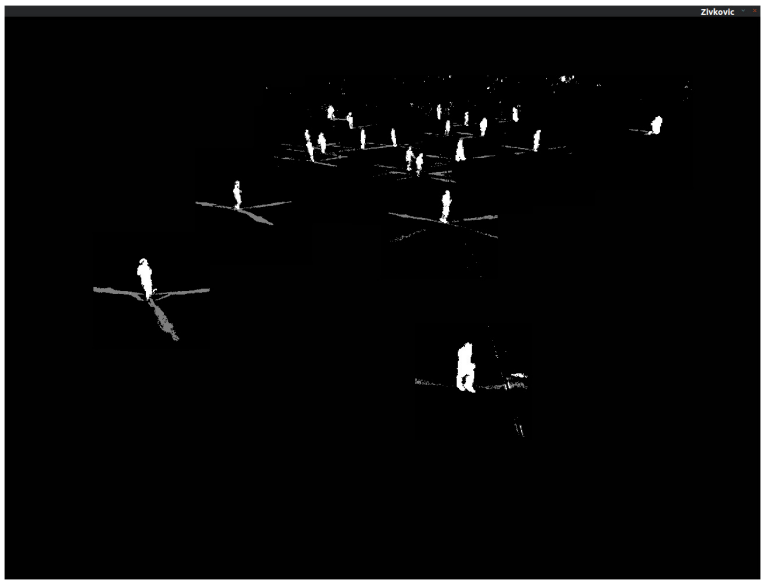


Figure 5.3: Visual results of first, naive ZXY BGS

Min	368.965
Max	462.543
Mean	417.946
Standard deviation	18.688

Table 5.4: Performance of first, naive ZXY BGS with static margins (ms)

amounts of unnecessary processing, of a O(n) complexity. This is clearly not an optimal way to do it, so we search for a better solution.

5.5.3 Optimization of ZXY BGS by use of bitmaps

The problem with the previous attempt, was that we iterated a heavy loop 20 times *per pixel* per frame. We therefore had to find a way to improve this. The solution found is to use a lookup map with the same resolution as the frames, to mark what pixels are to be processed or not. In this case, we are able to calculate this lookup map only once per frame, and then do a lookup for each pixel to see if it is to be processed. On this calculation, we set the player pixels in the lookup map, including the safety margins, to 1. The rest of the pixels remain 0. Then, when executing the BGS process itself, we simply do a lookup on every pixel, and process it if the value is 1, or mark it as background at once and ignore its processing if the value is 0. This lookup has a complexity of O(1), which is a huge improvement.

In this lookup version, the player pixel lookup map is a bitmap, and the safety margin is statically 100 pixels. The classification accuracy is not affected by this optimization, but the performance is affected a lot. The new performance is seen in table 5.5

As we can see, we have improved it to be approximately equal to the unmodified Zivkovic solution. There are however some artifacts that need to be fixed, and we are still not executing in real-time. The bottleneck at this point is the playerPixels-test, i.e.

Min	44.740
Max	94.023
Mean	65.343
Standard deviation	8.089

Table 5.5: Performance of ZXY BGS with bitmap, static margins (ms)

the player pixels lookup, even with the low O(1) lookup complexity. This is because the lookup is done so often that it affects performance.

5.5.4 Optimization of ZXY BGS by use of dynamic player frame sizes

To further improve the algorithm, we can implement a dynamic player frame size for use in the algorithm. This means that each player frame is larger when closer to the camera, and smaller when further away. This has several consequences. The first is that there is much less data to process. This is because we now, depending on the *depth multiplier* used to calculate the margin size, have smaller frames in the background, which results in substantially fewer pixels being processed in the BGS algorithm. Furthermore, we get substantially less noise in the image, because we filter out more of the noisy parts. We can see the resulting foreground mask in figure 5.4. The performance improved noticeably, and can be seen in table 5.6.

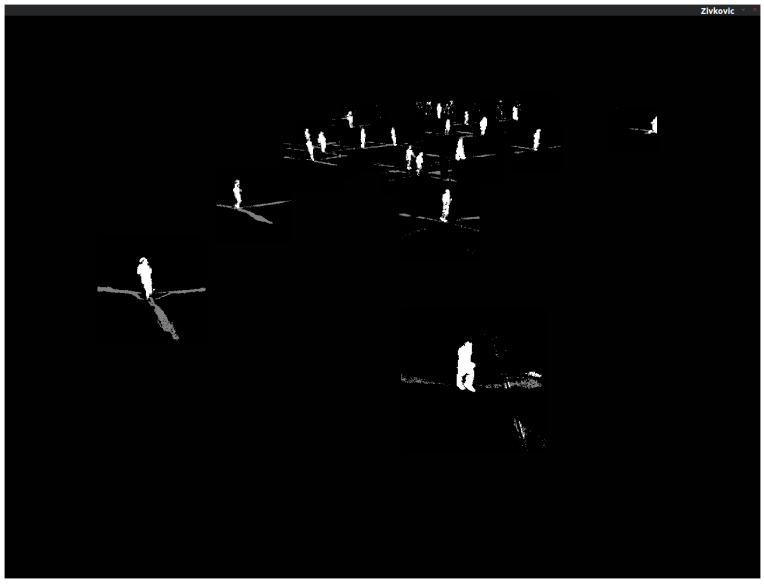


Figure 5.4: Visual results of ZXY BGS with dynamic player frame size

5.5.5 Optimization of ZXY BGS by use of a hashmap for lookup

Because most of the time was spent doing the lookup of the active player pixels, we need to find a faster way to do lookups. The next step is therefore to change structure for storing the player pixel lookup map. A hashmap was therefore the next lookup

Min	33.775
Max	88.559
Mean	49.071
Standard deviation	8.357

Table 5.6: Performance of ZXY BGS with bitmap, dynamic margins (ms)

structure to be tried, due to hashmaps also having an O(1) lookup time. This however, did not result in higher speeds, but rather much worse performance than the bitmap version. The performance can be seen in table 5.7.

Min	926.429
Max	1186.116
Mean	1007.862
Standard deviation	602.772

Table 5.7: Performance of ZXY BGS with bytemap, dynamic margins (ms)

No surprise, Vtune shows us that the bottleneck is the lookup. We see clearly that the lookup of a hashmap is way too slow for this use. This can be easily explained. The reasons for this is that hashmaps are built by use of lots of pointers, and on lookup, we need to follow several pointers to find the wanted lookup value. The O(1) lookup time is only based on the data structural/algorithmic time of retrieving values. Hashmaps are very good for use in cases with a dynamic number of data, but in our case, all the frames are of the same size, and it is therefore much better to use a static data structure for the lookups. We can clearly see this in practice in the benchmarks, where this hashmap solution is way slower than even the first, naive implementation.

5.5.6 Optimization of ZXY BGS by use of an integer map

Another way to improve the performance, would be to change from a bitmap to an integer map. This should be somewhat faster, because integers are the standard word length in the architecture (64 bits on the x86_64 architecture), and the CPU should therefore be better optimized for this word length. In addition, bitmaps result in lots of bit-shifting operations, which add a lot of overhead. The downside of an integer map would be a much higher memory consumption for the lookup map, due to us needing to use a whole integer for storing the same as one bit. I.e. we need 64 times the memory to store the same amount of data. This means that the integer map alone would consume $width * height * sizeof(integer)$ bytes. This means that, for frames of size 1280x960, and an integer size of 64 bit, the memory consumption will be 9.4 megabytes.

Even though this is 64 times the size of the bitmap, the size is still not daunting, so we can accept this amount of memory consumption, as long as the speedup is good enough. Fortunately, as we can see from table 5.8, the speedup proved to be so high that the ZXY BGS now runs in real-time, i.e. with average processing times of less than 33 ms.

Min	17.816
Max	56.112
Mean	28.888
Standard deviation	6.576

Table 5.8: Performance of ZXY BGS with intmap, dynamic margins (ms)

The maximum time per frame is as we can see higher than 33 ms, which it optimally should not have been. There might be several reasons for this. There might be reasons such as other tasks on the PC needing CPU cycles, certain combinations as part of the BGS algorithm, etc. In addition, the standard deviation and variance is higher than the previous, unmodified BGS versions. This can be explained quite easily. This is because, when doing our analysis, the amount of processing to be done varies greatly depending on where the players are. The further away the players are from the camera, the smaller the frames become, and the fewer pixels we need to analyze. In addition, if all the players are outside the whole video frame, there will virtually be no processing, only a quick if-test per pixel. The worst case considering processing time, would be when all the players are closest to the side line near the camera. Then the frames would be of the maximum size, resulting in the most processing. This would, however, still not cover much of the video frame in total, still resulting in much less processing than in the unmodified implementation.

As we can see from the Vtune analysis, the largest bottleneck is still the lookup of player pixels, but the speed has now dramatically increased (the lookup is no longer the only hot-spot factor). However, if we could speedup the lookup even more, the algorithm would be even faster, making the maximum running time smaller, which prevents lag spikes in the processing, and lowers the chances of missing the real time deadline.

5.5.7 Optimization of ZXY BGS by cropping frames

We currently limit the number of pixels to be processed by use of the ZXY tracking data. This means that we can do a quick lookup to find out if we are to process a specific pixel or not, effectively limiting the processing by a substantial amount, which we have already seen. However, this requires a lookup for each pixel, and we have seen that the lookup is the bottleneck in the implementation so far. Therefore, if we find a way to limit the amount of data to be looked up, we can improve the performance even more. Because we have static cameras, and a static soccer field, we know the boundaries of where players are allowed to move. Therefore, we can simply crop the image on the outer borders, to remove these pixels from the BGS analysis, ie. these pixels are totally ignored.

Lookup-based cropping

When executing the algorithm, we know which camera we are currently processing frames for, and we can therefore specify maximum and minimum values for the X and Y pixels. Due to the cameras covering different angles of the field, we need to calculate

the pixels that should be allowed or not, and create a cropping lookup map of what pixels are supposed to be analyzed, much in the same way as the lookup map for the player pixel frames. The creation and calculation of this cropping lookup map is done on creation of the BGS object. Then, when iterating through all the pixels of each frame, we do a lookup to see if the respective pixel is valid for analysis or not, meaning that it is simply ignored if it is marked as invalid. This cropping supports diagonal cuts, and in this benchmark, we have run the algorithm for camera 1, i.e. for the one to the left. The reason for this is that this camera has an angle that results in diagonal soccer field lines, which provides lots of variation with respect to the cropping.

We can see how this cropping works by looking at figure 5.5, where we in figure 5.5(a) see the input frame, in figure 5.5(b) see the corresponding foreground mask without any cropping enabled, and in figure 5.5(c) see corresponding foreground mask with lookup based cropping enabled. In this figure, the gradient areas are the cropped ones ignored by the BGS. There is really no noise reduction here, because the ZXY data always reduces noise more than the cropping, however, the amount of pixels needed to be analyzed is reduced.

The performance of the un-cropped version for camera 1 is seen in table 5.9. We can see that the algorithm is a bit faster on camera 1 than on camera 2, which can be explained by there being fewer frames with many players on the camera, which results in fewer pixels to process. The performance of the lookup-based cropping can also be found in table 5.9. The performance is rather disappointing, but can be explained. The reason for the worse performance is because we now are doing a lookup in a second lookup map. This means that, for most of the pixels, we need to do two lookups, and not only one.

Crop version	No crop	Lookup	Calculation
Min	14.377	20.001	14.374
Max	54.367	63.990	52.629
Mean	27.984	33.441	27.857
Standard deviation	6.826	6.921	6.810

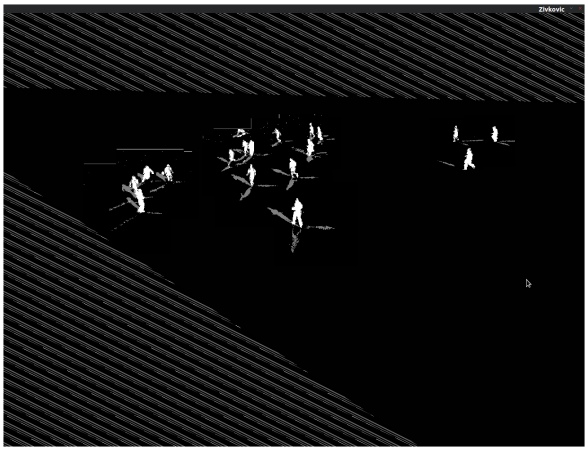
Table 5.9: Performance of ZXY BGS, cropping comparison, camera 1 (ms)

Calculation-based cropping

We can also try to avoid using a lookup for the cropping, and rather for each pixels calculate whether it should be analyzed or not. This can be done by doing this evaluation within the second for-loop, i.e. when iterating over the X-pixels. We then have a few choices: We can decide to calculate borders for both the X-direction and the Y-direction, or we can select one of them. The choice leading to the highest reduction in pixels processed would be cropping in respect to both directions, so we try this. The visual illustration of how the cropping looks is equal to the one for lookup-based cropping, see figure 5.5(c) for a reference. The performance can be seen in table 5.9. As we can see, this performance is equal to the no-crop version. Here we have basically replaced one lookup per pixel with some few extra calculations. The largest hot-spot

(a) Input frame, camera 1

(b) ZXY BGS without lookup-based cropping, camera 1



(c) Visual illustration of ZXY BGS with lookup-based cropping, camera 1

Figure 5.5: Initial BGS model comparison

is still the lookup of player pixels, but we now also have a fair hot-spot in the calculation and testing of the cropping. When compared to the lookup-based cropping, it is easy to believe that the calculation-based cropping would be slower, as it actually access as many or more memory locations than the lookup-based one, in addition to doing calculations in both the for-loops. However, the lookup-based one access different memory locations on each pixel (the lookup map), while the calculation-based one generally only access the same ones on each pixel (these are common values, such as a multiplier specifying the angle of the diagonal cropping lines). Due to the access of the same addresses, it is plausible to believe that these are cached, so the CPU does not need to access memory, which results in higher performance.

Straight, horizontal cropping

What we can see from this, is that these cropping implementations are not great performance optimizations, as they either maintain the same performance, or actually result in longer processing times. However, this is only the case for cropping that results in lots of processing and many memory accesses, such as cropping with diagonal lines, as described above. This is caused by all of the calculation and look ups happening in the innermost for-loop seen in the pseudo code in section 5.5.1.

There is one type of cropping we can attempt that is extremely straight forward, limited, and naive. This cropping exploits the fact about the cameras placed on the long side of the rectangular field. This means that the most of the cropping will be done in the horizontal dimension, i.e. left to right. Let us therefore simply crop by setting static, straight lines in the horizontal dimension, and ignore pixels outside these straight lines. By doing it this naive way, we will not need any calculations for each pixel; we simply ignore all the pixels in each row outside the specific margins at the top and bottom of the frames. This will not result in an optimal cropping of the images, but will still limit the amount of pixels to be iterated substantially. What we do is to set a margin width at the top and a margin width at the bottom of each camera, 0 pixels wide if necessary, and then ignore whole pixel rows by use of these margins. The pseudo code then looks like this:

Loop through all Y pixels accepted by the margins ($y = MIN_Y \Rightarrow y = MAX_Y$)

> **Loop** through all X pixels for the current y ($x = 0 \Rightarrow x = framewidth$)
>
> **If** pixel [x, y] is close to a ZXY player position, analyse the pixel using Zivkovic,
>
> **else** mark the pixel as background

As we can see, the crop-check is only done in the outer for-loop, i.e. only for the Y-pixels. This lowers the amount of lookups and calculations needed to be done by a substantial amount compared to the previous cropping solutions.

Note that there are no diagonal lines in this cropping implementation, so we therefore switch back to camera 2 for a more interesting performance comparison. Figure 5.6 illustrates how such a straight cut reduces the amount of pixels to be processed, where the area marked with a gradient is never processed. The performance of this implementation for camera 2 with 75 pixels margin at the bottom and 100 pixels margin at the top, compared to the un-cropped version, can be seen in table 5.10. We can see that we save approximately 2.2 ms, which is not very much, but it is basically a free optimization with almost no need for change in the implementation.

5.5.8 Optimization of ZXY BGS by use of a byte map

We have now seen that diagonal cropping in all dimensions does not really improve the performance, while straight, horizontal cropping gives a small performance boost. Note however, that the player pixels lookup map is still the primary hot-spot. We therefore want to look at another way to improve these lookup times. In the previous optimization for this map, we changed it from a bitmap to an integer map, due to

Crop version	No crop	Straight, horizontal crop
Min	17.816	14.346
Max	56.112	56.062
Mean	28.888	26.683
Standard deviation	6.576	6.984

Table 5.10: Performance of ZXY BGS, no crop vs straight horizontal crop, camera 2 (ms)

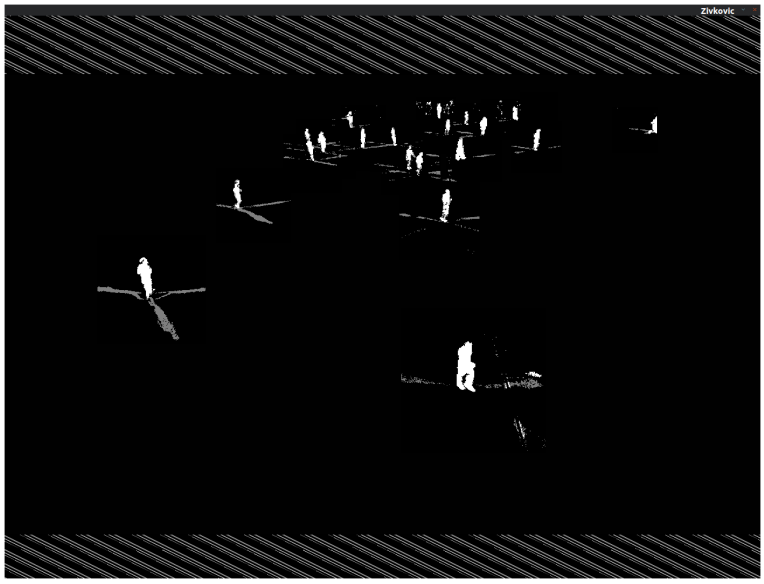


Figure 5.6: Visual illustration of ZXY BGS with straight, horizontal cropping, camera 2

integers being of the native word length of the CPU architecture. There was one thing we did not try at this point, however, and that was to test a byte map. This means that we try to use an ordinary char-map instead of int-map, and see how that performs. The performance of the byte map solution can be seen compared to the integer map solution, both with straight cropping, in table 5.11.

Crop version	Int-map	Byte-map
Min	14.346	9.661
Max	56.062	50.431
Mean	26.683	21.711
Standard deviation	6.984	7.213

Table 5.11: Performance of ZXY BGS, integer map vs. byte map, with cropping, camera 2 (ms)

As we can see, this is much faster than the integer map implementation. There might be several reasons for this, but the most prominent one of these is caused by caching. Because the integer map is so large, the cache is not used as efficiently as for a byte map, which is 1/4th the size of the integer map. This means more cache hits, and therefore less memory access, which speeds up the process.

We therefore have a tradeoff between the three different maps, i.e bitmap, bytemap

and intmap. In the case of the bitmap, we can store it very compactly, and use the cache efficiently. However, the bitmap needs lots of bitwise operations, which together makes the performance rather slow, even with the more efficient caching. In the case of the integer map, we have more efficient processing and calculations, due to the native word length. However, due to the much larger size, the cache is not used as efficiently, and we see more cache misses, and therefore memory accesses. This is a performance hit, but it is still faster than the bitmap solution. We then have the byte-map solution, which is in between. The word length is not as optimal as for an int-map, but due to a much more compact map, we use the cache much more efficiently.

Also, if we take a look at the Vtune analysis, we can see that we no longer have a single hot-spot. The lookup hot-spot is still the largest, but no longer dominating. In the analysis of this implementation, with a run of 9000 frames, the amount of time spent on the playerPixels-lookup, i.e. the use of the byte-map, is 41.066 seconds in total. The next hot-spot is the calculations in the X-pixel for loop, which is 32.868 seconds in total. The third hot-spot is setting the mask of the current pixel to true, i.e. background, if the pixel was not a player pixel. This task takes 10.723 seconds. In other words, the three largest hot-spots has a ratio of approximately 4/3/1. We have in other words really improved the lookup time, and therefore the performance of the implementation.

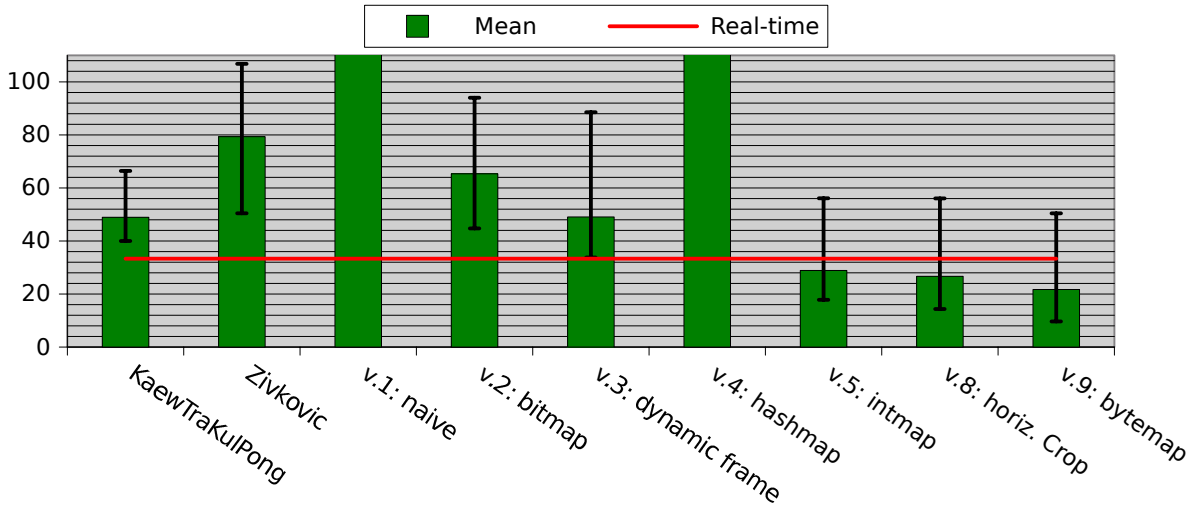


Figure 5.7: Performance of CPU based ZXY BGS implementations, camera 2 (ms)
Note that the lookup- and calculation-based cropping versions are excluded, because they were tested on camera 1.
The bar of the naive ($mean = 417.946$) and hashmap-based ($mean = 1007.862$) implementations are cut due to size.

5.5.9 ZXY BGS CPU performance summary

The performance of the different CPU based ZXY BGS implementations is found in figure 5.7. Note that the lookup- and calculation-based cropping versions are excluded,

because they were tested on camera 1, while the rest were tested on camera 2. Nevertheless, from this figure and the previous version tables, it is clear that the last version, i.e. the byte map-based with horizontal cropping, is the fastest one, while performing in real-time.

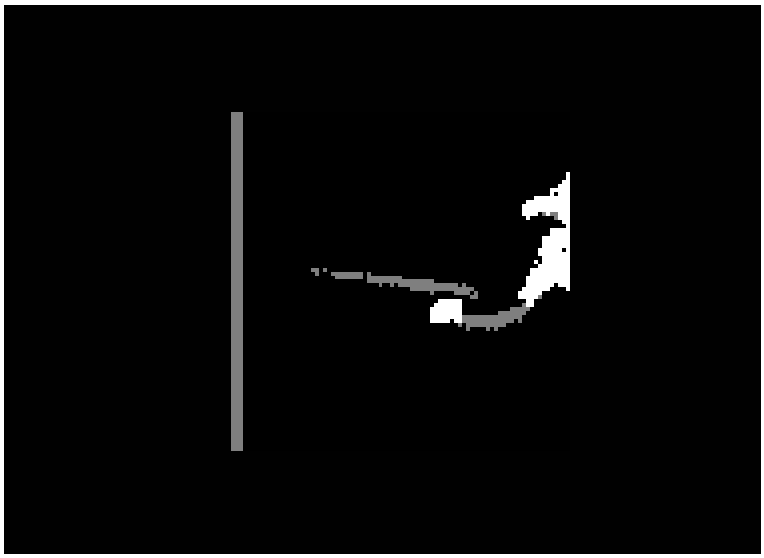


Figure 5.8: Example of ZXY BGS inaccuracy

5.6 ZXY inaccuracy

There are several cases where the frame around the players are not accurate enough, so that the players drift outside of them, and are therefore not completely considered as foreground. An example can be seen in figure 5.8. There are many reasons for this, and several ways to solve these. The main five reasons for this inaccuracy are:

5.6.1 Debarelling parameters

As described in earlier sections, the debarelling function needs some debarelling coefficients as parameter. We calibrated the cameras we had in the lab at that time to retrieve these, and then used these coefficients as parameter for removing the barrel distortion from all the cameras. This, however, gives us some small problems, which is due to the fact that no lenses are completely equal; there are always some small differences. In addition, the cheaper the lens, the larger the differences. This means that the parameters we are using in the debarelling function are not 100% correct for the cameras we use at Alfheim, which leads to small errors in the debarelling result, especially in the corners of the frames. This leads to inaccurate tracking, especially when the players are at the frame corners.

The way to solve this is to calibrate and retrieve the parameters for each camera stationed at Alfheim, and then use these for the individual camera debarellings. This would correct these small errors, and therefore result in more accurate mapping between ZXY coordinates and pixels, which means that the tracking would become more

accurate. Because this is not critical for the sake of this thesis, calibration and correction of these parameters have been ignored.

5.6.2 ZXY sensor inaccuracy

The current version of the ZXY player tracking system installed at Alfheim has an accuracy of ±1 meter. This can result in quite visible inaccuracies when finding the correct coordinate and pixel of a player. 1 meter is relatively high amount of inaccuracy for our scenario, and we can clearly see the result of this several places. There are for instance cases where a player is standing still, but the tracking frame is drifting back and forth, with the player inside or a bit outside. This can even happen when players are standing still in the middle of the frame, i.e. the cause of this large drift is not due to the debarelling inaccuracies.

As we saw in section 2.4.1, according to ZXY, the newest version of the tracking system has an accuracy of ±0.5 meters. Therefore, the solution for this problem would simply be to upgrade the system to the newest version. This, however, is not feasible to do within the time frame of this thesis, and therefore has to be ignored for now.

5.6.3 Time drift

A third possible cause of inaccuracy in the tracking, is time drift. There can be time drift several places in the system, and these are important to be aware of. For instance, it is important that the ZXY tracking system and the frame capture system are synchronized with respect to time, as described in section 2.4.2. If they are not, we can have a situation where the timestamps of the corresponding frames and tracking samples are not correct. This can lead to severe cases of time drift, where whole matches can have out-of-sync tracking and frames. This can make the tracking within that time slot, possibly whole match, totally worthless.

There are ways to minimize the chances of this happening. First of all, by configuring the ZXY system and the storage/panorama pipeline to use the same, local NTP server (as described in section 4.3), we minimize the amount of possible time drift between the systems to a minimum. Also, this makes sure that the systems are kept in sync for as long as they are connected to the same NTP server, and is the preferred solution. Another way to combat this problem, is, if the problem has already occurred, to try to manually shift either the frame timestamps or the ZXY time stamps, so that they are again aligned and in sync. This, however, is not a good solution, and can only be viewed as a backup solution, if bad time drift occurs.

5.6.4 Dropping frames

Another source of tracking inaccuracy is if frames are dropped on capture or in the panorama stitcher pipeline, as described in section 4.4.5. As mentioned here, it is possible that frames are dropped when we are capturing video, for instance due to signaling delays, timing delays in the trigger boxes, and more, and the pipeline might drop frames when performing too slowly. This is solved by reusing the previous frame, both

in the CamRecorder and in the pipeline itself. However, this means that there is a possibility that we process a frame several times in a row. However, the ZXY data is not tied to the capture of frames, so in this case, the sampling of player positions would still continue, even with the same frame several times in a row. Therefore, when trying to track players, we can have cases where it looks like the players are standing still while the ZXY positions are still updating. This lasts until a new frame is recorded where everything goes back to normal. In practice, this is not really a problem, because the rate of frame drops is very low as long as the stitcher pipeline performs fast enough, and the synchronization between frames and ZXY data is still synchronized, so drifting does not occur.

5.6.5 Sampling interval

The highest sampling interval of the ZXY system installed at Alfheim is 20 Hz. In comparison, the frame-rate is of frequency 30 Hz. As we can see, we have a mismatch here. This mismatch is solved by reusing the previous sample each time we reach a third frame, i.e. when ($frameCounter\%sampleCounter == 2$). In theory, this results in a small $ZXY \rightarrow frame$ mismatch for that frame, but this is such a small inaccuracy that we can safely ignore it.

5.7 GPU implementation

We already have the implementation running in real time on one thread, for one camera. Nevertheless, we need it to be much more scalable. Why is this? Why is this not good enough, as we only have four cameras, and already have four cores on the CPU? First of all, in our case, we have enough CPU for processing four cameras in parallel by use of four cores on the CPU. However, the BGS is only a smaller part of a large processing pipeline. This pipeline needs lots of CPU power, and we can therefore not rely on having one core per camera dedicated to BGS processing. In addition, we want the system and implementation of this module to be as scalable as possible. Primarily, we want to be able to use more than four cameras in the future.

Both of these arguments tell us that we must find a better way to process this. This is where the GPU optimization comes in. Image processing is in general a very parallel task, and in the Zivkovic BGS algorithm, all pixels are viewed and processed independently. Therefore, if we can get the BGS algorithm to run on the GPU, we will possibly get a huge performance boost and also a more scaleable solution that should be much more optimal for our scenario.

5.7.1 The GPU hardware

The cards we are using when testing and benchmarking the background subtraction implementation on GPU, are the very low end Nvidia Quadro NVS 295 (G98), the former high end Nvidia Geforce GTX 280 (GT200), and former high end Nvidia Geforce GTX 480 (GF100), running on DevBox 3. The GPU specifications can be found in table C.6. All of these cards are several generations old. The newest one, the GTX 480, is

1 generation old, but it has most of the important CUDA functionality, and is therefore representative of newer high end cards, which are the types of cards we want to use in the Bagadus system.

5.7.2 The existing Zivkovic GPU implementation

As we can see from [38], there already exists a GPU implementation of the Zivkovic algorithm, based on CUDA, which needs both CUDA and OpenCV to run.

First of all, we need to decide which version of the Zivkovic GPU implementation we want to base our modification on. In the article, they explain several levels of optimizations, and implement a CUDA kernel for each of these. In the source code provided, they only provide two of these kernels, plus one extra. The first kernel they provide is kernel 3, which is the one with a Structure of Array pattern, memory coalescing on the Gaussian parameters and pinned memory. The second kernel is kernel 4, which is the one with all the previous optimizations plus asynchronous memory transfers. The last implementation, kernel 5, is a kernel with all the previous optimizations, in addition to templates for easier programming. Kernel 5 is not purely a performance optimization, and is not mentioned in the article, so we do not base our implementation on this one. The one we want to base our implementation on, is kernel 4. The reason is the same as mentioned in [38], i.e. that the kernel uses interleaving of memory transfer, GPU execution and CPU execution. Because our background subtraction is to be a small part of a much larger pipeline, this interleaved execution makes it much more attractive, as the BGS-module in this case can more flexibly process at the same time as other modules.

The performance of this implementation running on our hardware can be seen in table 5.12. Here we can see that the NVS 295 is so weak that it is not able to run the BGS in real-time. However, we did get a small speedup compared to the unmodified, CPU-based version, with a factor of $\frac{79.345ms}{53.605ms} = 1.48$. We can see that the kernel execution time is the dominant part of the total execution time, while the transfer to and from the GPU is a marginal part, which tells us that the NVS 295 is computationally limited. This means that the NVS 295 is too weak for us to use, and it is therefore not interesting to test more on this card, which means that we ignore it from now on.

In comparison, both the GTX 280 and GTX 480 are more than fast enough to perform in real-time, with approximately 5.3 ms total processing time, which tells us that these cards are limited by the transfer between GPU and CPU. This is a speedup of almost 15 times. In addition, we see that the kernel executes 20% faster on the GTX 480 compared to the GTX 280, which is really no surprise, considering that the GTX 480 has many more and faster cores.

Nvidia has published a best practices guide [36] (from now on referred to as the BPG) for optimizing CUDA applications, which focuses on the most important steps in finding bottlenecks, and finding ways to remove these from an application. In the end, what they produce, is a prioritized list of steps to follow when profiling and optimizing CUDA applications. A high priority step in the BPG, is to use the effective bandwidth of the application, i.e. the throughput, for measuring performance and optimization gains. In table 5.13, we see that the GTX 480 has a 0.9 GB/s higher throughput than the GTX 280.

GPU	NVS 295	GTX 280	GTX 480
Total	53.605	5.326	5.365
Kernel	52.508	1.076	0.812

Table 5.12: Performance of unmodified BGS on GPU, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	6.20 GB/s	5.82 MB/s	130.47 GB/s
GTX 280	5.30 GB/s	1.88 MB/s	63.78 GB/s

Table 5.13: Throughput of unmodified implementation

Another important aspect of the BPG, is to maintain a high enough occupancy to hide latency from register dependencies. We discussed occupancy in section 3.6, and in the unmodified Zivkovic GPU implementation, an occupancy of 0.5 was found to be optimal. Connected to this in the BPG, it is stated that to facilitate coalescing and provide optimal computing efficiency, block sizes should be a multiple of the warp size, often between 128 and 256 threads. This is solved in the Zivkovic GPU implementation by limiting the amount of threads per block by making each thread process several pixels. In the following implementations, we maintain this solution.

5.7.3 ZXY optimization of Zivkovic on the GPU

The ZXY optimization of the the Zivkovic GPU algorithm is fairly straight forward, in the same way as the one for the CPU. There are, however, several things to consider. First of all, we need to be aware of the limitations and characteristics of the CUDA architecture, where many of these have been explained in Chapter 3. Furthermore, as stated in the BPG, it is important to analyze the old application to find ways to parallelize sequential code. This has already been done for the Zivkovic algorithm itself in [38]. However, we need to do this for the ZXY part too. To begin with, we found that the generation of the player pixel lookup map was not very parallelizable, and therefore decided to keep it sequential on the CPU. However, when accessing the lookup map, we only access each pixel once, and never modify it, meaning that the access to this lookup map could easily be parallelized.

The ZXY optimized solution is therefore to first, on each frame, create the player pixel lookup byte map on the CPU, in the same way as for the CPU implementation. Then we copy this lookup map to the GPU memory, together with the corresponding frame. The ZXY BGS kernel is then launched, where each GPU thread loops through a limited amount of pixels, and for each pixel the thread does a lookup on that pixel to see if it is supposed to be processed or not. If not, it marks that pixel as background, and continues on the next pixel. This should in theory improve processing times. The pseudo code can be seen here:

Calculate the player pixel lookup map

Transfer the lookup map to GPU

Launch the ZXY BGS CUDA kernel, with m threads.

All threads: Loop through *n* number of unique pixels

If playerPixels[currentPixel] is set, process the pixel

else mark the pixel as background

When testing different implementations to find the best optimizations, we use the Nvidia Visual Profiler [77] to profile the application to determine the bottlenecks and hotspots that need to be improved.

Global memory implementation

The first version of the ZXY optimization is implemented by storing the playerPixel lookup map in global GPU memory. Global memory is located off-chip, and is not cached, but it is large, and more than large enough to store the playerPixels lookup map. When implementing it, we had to be very careful to ensure that the accesses to the lookup map in global memory was coalesced whenever possible. Coalesced memory access was described in section 3.5, and is mentioned as an important step in the BPG. We ensure this by making sure the lookup map is aligned properly, and that threads access the map in a coalesced pattern. Furthermore, we do not introduce any *__syncthreads()* operations, so we avoid using these inside divergent code, which we can see from the BPG is highly discouraged. This is because *__syncthreads()* works like a barrier for CUDA threads, meaning that if it is used within divergent code, it can lead to errors and deadlocks.

GPU	GTX 280	GTX 480
Total	5.479	5.526
Kernel	0.499	0.328

Table 5.14: Performance of ZXY BGS on GPU, global memory, mean times (ms)

The performance of the global memory-based implementation can be seen in table 5.14. As we can see, the kernel performance on the GTX 480 and GTX 280 are much better on the ZXY modified version than the original BGS implementation, with factors higher than 2 times. However, when we look at the total processing time per frame, we see that the GTX 480 and GTX 280 have approximately the same performance. These numbers all make sense, and can be explained by the bottlenecks on the cards. As mentioned in section 5.7.2, for the NVS 295, the bottleneck is the processing itself due to fewer processing cores, which is why it is not used. However, on the faster GPUs, we are limited by the memory transfer between the GPU and CPU. From this we can see that the optimization using the ZXY player data does not provide a substantial decrease in total processing time, but actually a slight increase. The reason for this is that we now need to transfer more data between the host and device than before, because we also need to transfer the playerPixels lookup map.

At the same time, the kernel processing times have decreased substantially. The main reason for this, is that there are so much data processed in parallel, so when we exclude the processing of pixels by use of the playerPixels lookup map, we reduce the amount of data to be processed by a large amount. However, as stated as a high

priority step in the BPG, we should work on avoiding different execution paths within the same warp. This is because threads within the same warp share program counters, meaning these diverging threads need to be serialized, and this can obviously have a severe impact on performance. In other words, within the warps with divergent execution paths, we expect that this optimization actually increase execution times. In our scenario, the kernel code is basically partitioned into two large divergent execution paths ("process current pixel" and "do not process current pixel"), that are run serialized within a warp. However, this does not seem to be much of a problem. This is because one of these paths, the "do not process current pixel" path, is negligible, so it does not take any extra time to run these instructions after (or before, depending on the scheduler) the first path has finished.

Furthermore, if *all* threads within a warp can exclude their pixels, and therefore follow the same execution path, it will drastically lower the processing time for that warp. Overall, a substantial amount of the field will not contain players, and will therefore not be processed. We will therefore have many warps where all threads are to ignore all pixels, which basically ignores all processing in that warp. In turn, this makes the warp-threads exit quickly, allowing other threads in other warps to execute. We can see this from the kernel execution benchmarks, where we see that the overall kernel execution times has decreased drastically on both GPUs.

The throughput of the GTX 280 and GTX 480 can be seen in table 5.15. Here we can see that the throughput has decreased slightly, which most likely is caused by the addition of the player pixel lookup map to the memory transfers.

GPU	Average	Minimum	Maximum
GTX 480	5.96 GB/s	7.95 MB/s	131.78 GB/s
GTX 280	5.28 GB/s	1.95 MB/s	64.14 GB/s

Table 5.15: Throughput of global memory implementation

	Time (ms)
Min	0.646
Max	5.682
Mean	1.479

Table 5.16: Player pixel lookup map creation performance

Next, it is also interesting to benchmark the performance of the playerPixels lookup map creation. The timing results from a benchmark run on the Intel Core i7 960 of DevBox 3 can be seen in table 5.16. In other words, we see that a large part of the total execution time is caused by the creation of the lookup map needed for the ZXY optimization. For instance, with the GTX 480, 1.5 ms of the total 5.5 ms (27%) of processing time was due to the lookup map generation.

However, even with this lookup map generation cost and the equal total processing times compared to the unmodified version, the ZXY optimization is desired. This is because when run in a pipeline fashion, i.e. in parallel with other modules, such as in Chapter 4, the cost of creating this map can be hidden, which basically makes it a

free operation. Furthermore, when run as part our panorama pipeline, the transfers between $CPU \rightarrow GPU$ and $GPU \rightarrow CPU$ is done only once for all modules per frame, which means that the cost caused by this transfer will be limited, even with the extra lookup map transfer. The substantial improvement in kernel processing times is therefore a huge advantage. In addition, we also get superior visual results with less noise, as described and shown in section 5.5. Overall, compared to the CPU based version, the performance has increased by a factor of approximately 3.9.

Memory optimizations

As mentioned the BPG, memory optimizations are the most important area for performance. Therefore, when we see the performance of the global memory implementation, and how we get a very low memory bandwidth utilization, it is natural that we focus on the memory performance. In other words, we want to improve the access times of the player pixels lookup map to optimize the throughput. The unmodified parts of the implementation has already been optimized by use of coalescing, the SoA pattern and asynchronous, double buffered transfers, so we focus on optimizing the access to the player pixels map to begin with. We have already looked at the different kinds of Nvidia GPU memory in section 3.4, so this is a good place to begin.

Constant memory implementation

What about putting the lookup map in constant memory? We have seen that the constant memory is located off-chip, but it is cached, so the access should be reasonably fast. There is however one catch: the limited size of the constant memory. As mentioned before, the constant memory in Nvidia GPUs is limited to 64 KB. Meanwhile, in our scenario of a resolution of 1280×960 pixels, using a char representation, the player pixel lookup map size is $1280 * 960 * 1 = 1.17MB$. In other words, the amount of constant memory is not nearly large enough to hold the lookup map (not to speak of four when integrated in the pipeline with four cameras). It is therefore obvious that we cannot use constant memory for storing the lookup map for speeding up the access to it.

Shared memory implementation

As mentioned in the BPG, using shared memory instead of global memory is a good way to increase an application's performance. Furthermore, shared memory is useful to avoid redundant transfers from global memory. We therefore want to try using shared memory to store the player pixels lookup map. This means that we first retrieve the lookup map, and temporarily store it in global memory on the device. Then, on each kernel launch, we transfer the corresponding parts of the lookup map to the shared memory for that block, which is then accessed and used. The processing times of this solution can be seen in table 5.17, and the throughput in table 5.18.

We can see that the total processing times are a bit worse than the global memory implementation for the GTX 480 and GTX 280, the throughput is pretty much equal, while the kernel execution times are considerably slower. The reasons for this are quite obvious: To be able to copy the lookup map into shared memory, we first need to copy

GPU	GTX 280	GTX 480
Total	5.568	5.606
Kernel	0.621	0.431

Table 5.17: Performance of ZXY BGS on GPU, shared memory, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	5.91 GB/s	8.22 MB/s	130.83 GB/s
GTX 280	5.27 GB/s	1.94 MB/s	63.89 GB/s

Table 5.18: Throughput of shared memory implementation

the data to global memory. Then, on kernel launch, we need to copy the map into shared memory for that block. This means that we need to copy data to and access the global memory anyway, and in addition get two extra accesses to shared memory. Shared memory is fast, but these extra accesses are strictly unnecessary. Furthermore, because we only access the lookup map once per pixel, and this access is coalesced, using shared memory for caching is not very suited here.

Pinned memory implementation with zero copying

The next step in the optimizations is to test pinned memory. Pinned memory is a mechanism for disabling paging for a block of memory on the host. This allows the GPU to access the memory directly, without the need for the virtual memory overhead. This drastically increases the effective memory bandwidth, and can therefore speedup the implementation.

When using pinned memory, one can also use zero copying. It is mentioned in the BPG as a low priority optimization, targeted at integrated GPUs without dedicated memory. However, it is easy to implement, so we want to see how it affects performance. Zero copying is a feature that allows a device thread to access host memory directly and use it as if it was ordinary device memory, as long as the memory is non-pageable (i.e. pinned). This is always a performance boost on integrated devices with no dedicated device memory, because it saves many unnecessary memory copies when device and host memory basically is the same. However, it only gives better performance on devices with dedicated device memory in a limited amount of cases, because each instance leads to a PCIe transfer. Transactions should therefore be coalesced. In most cases it should be faster to transfer the memory to the device first, and then access it directly on the device itself.

Pinned memory with zero copying is not supported on all GPUs, and does in our case only work on the GTX 480. The performance can be seen in tables 5.19 and 5.20. We can see that the average total processing time per frame has increased, and the average throughput has decreased quite a bit from the global memory implementation, while the kernel execution time also has increased. This can be explained by what we have just mentioned, that zero copying is only in rare cases a performance boost on devices with dedicated device memory, because it is targeted at GPUs without such memory.

GPU	GTX 280	GTX 480
Total	N/A	5.628
Kernel	N/A	0.401

Table 5.19: Performance of ZXY BGS on GPU, pinned memory with zero copying, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	5.59 GB/s	5.96 MB/s	131.54 GB/s
GTX 280	N/A	N/A	N/A

Table 5.20: Throughput of pinned memory implementation with zero copying

Pinned memory and asynchronous memory transfers

Even though zero copying did not work well, we can still try to exploit pinned memory. Using pinned memory also allows us to transfer memory to the device asynchronously. This is done by using CUDA streams. A CUDA stream is just a sequence of instructions to be run in order on the device, and CUDA allows to run several streams in parallel. Together, this allows for asynchronous transfer.

The way this is implemented is by running several streams, at least one copy stream and one execution stream, in parallel. The execution stream is responsible for executing the BGS algorithm itself, while the copy stream is responsible for copying memory to the device asynchronously and in parallel with the execution stream. This means that one needs double buffering for the data to be transferred, i.e. in our case two buffers for storing the player pixels lookup map. The copy stream transfers the next data unit to one buffer, while the execution stream accesses the current data unit from another, equally sized buffer. Then, when the copying of the next data unit is done and the execution on this unit can begin, the copying stream can start transferring a new data unit to the old buffer, overwriting the old and no longer usable data. This technique basically interleaves the CPU execution, GPU execution and GPU memory transfers.

This technique is already implemented for the transfer of the input and output frames, so we want to replicate the same behavior for the player pixels lookup map, which is the data unit in our case. Since we already have some asynchronous transfer, we can exploit this and reuse the existing streams. This means that we only need to add double buffering for the player pixels map, and replace the ordinary cudaMemcpys to cudaMemcpyAsyncs, i.e. the asynchronous equivalent of cudaMemcpy. Therefore, the execution stream and copy stream remains the same, i.e. *execStream* and *copyStream*. On the GPU side we use global memory to store the lookup maps, with the same kernel performance as described in the global memory section.

The performance of this implementation with two streams, i.e. one copy stream and one execution stream, is seen in tables 5.21 and 5.22. From this, we can see that the performance for both GTX cards are similar to the global memory implementation. The throughput of the GTX 480 has slightly increased while the average processing time per frame has decreased ever so slightly. In comparison, the GTX 280 actually has slightly worse performance, where the processing time per frame has increased by a small amount, the average throughput barely has decreased, and the kernel performance is

equal. These differences can be explained by random fluctuations, so all in all, the performance can be considered equal, but with the advantage of the execution and data transfers being interleaved.

GPU	GTX 280	GTX 480
Total	5.540	5.501
Kernel	0.499	0.325

Table 5.21: Performance of ZXY BGS on GPU, pinned memory with asynchronous transfer and 1 copy stream, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	6.01 GB/s	5.96 MB/s	131.3 GB/s
GTX 280	5.21 GB/s	1.74 MB/s	63.75 GB/s

Table 5.22: Throughput of pinned memory implementation with asynchronous transfer and 1 copy stream

Another interesting thing to test with the asynchronous transfer, is to use more than one copy stream. In this case, we add one more copy stream that is dedicated to transferring the player pixels lookup map, while the original copy stream is dedicated to copying frames back and forth between GPU and host. The new copy stream is named *copyStream2*. The performance can be seen in tables 5.23 and 5.24. As we can see from these results, performance is equal to the ones for only one copy stream. Due to this, we rather prefer the previous solution with only one copy stream, as this is conceptually cleaner, and with theoretically less overhead.

GPU	GTX 280	GTX 480
Total	5.546	5.495
Kernel	0.500	0.326

Table 5.23: Performance of ZXY BGS on GPU, pinned memory with asynchronous transfer and 2 copy streams, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	6.01 GB/s	5.96 MB/s	131.54 GB/s
GTX 280	5.20 GB/s	1.74 MB/s	63.72 GB/s

Table 5.24: Throughput of pinned memory implementation with asynchronous transfer and 2 copy streams

Shared and pinned memory implementation

By using the shared memory implementation above, we first need to transfer the player pixels lookup map from the host to global memory on the device, and then transfer it

to shared memory later on, which is unnecessary overhead. What we can try to do instead, is to use pinned memory to directly transfer the lookup map into the shared memory in a coalesced way. As before, use of pinned memory and zero copying does not work on the GTX 280, only the GTX 480. The performance results can be seen in tables 5.25 and 5.26.

We can see that the throughput takes quite a performance hit compared to the global memory implementation, and ends up at an average of 5.64 GB/s, compared to the 5.96 GB/s of the global memory version. We can also see that the average total processing time per frame has increased marginally from the global memory implementation, 5.53 ms to 5.58 ms, while the kernel processing time has increased substantially from 328 µs to 533 µs. This can be explained by banking conflicts in the shared memory. The throughput hit can be explained by, as before, zero copying not being optimal on a dedicated GPU like the GTX 480, except for a very few cases.

GPU	GTX 280	GTX 480
Total	N/A	5.579
Kernel	N/A	0.533

Table 5.25: Performance of ZXY BGS on GPU, shared and pinned memory, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	5.64 GB/s	5.96 MB/s	131.42 GB/s
GTX 280	N/A	N/A	N/A

Table 5.26: Throughput of shared and pinned memory implementation

Texture memory implementation

The last type of memory we want to test, is texture memory. We have already seen that texture memory is located off-chip, but is cached, so this caching can result in lower access times and better performance for us. The caching is optimized for spatial locality, which means that threads of a warp reading data close to each other will benefit from the cache. In our case, threads of the same warp will read texture addresses from the player pixels lookup map close to each other, so this should work well. The performance of this implementation is seen in tables 5.27 and 5.28

We can see that the overall performance remains the same as the global memory implementation. The total processing times can be considered equal, while the average throughput remains the same. The kernel execution times are also equal. This can be explained by texture memory actually being stored in global memory, with some caching. The effect of the cache is apparently not that effective, which we can see due to the approximately same performance as the global memory implementation.

Here it is very interesting to use texture memory for also storing and accessing the input frames, and not only the lookup map. This can hopefully improve performance for the whole implementation, not only the ZXY optimization part, due to texture

GPU	GTX 280	GTX 480
Total	5.507	5.498
Kernel	0.489	0.333

Table 5.27: Performance of ZXY BGS on GPU, texture memory, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	5.96 GB/s	7.95 MB/s	131.18 GB/s
GTX 280	5.28 GB/s	1.94 MB/s	63.58 GB/s

Table 5.28: Throughput of texture memory implementation

caching. When running benchmarks on this improved texture memory implementation, we get the results from table 5.29 and 5.30.

We can see from the numbers that this implementation is, compared to the first texture implementation, approximately the same on the GTX 480 and somewhat slower on the GTX 280. This is most probably because the original global memory implementation of the input images are made sure to be coalesced. This results in very efficient access to the input images, making the global memory use not that bad. In addition, the access patterns to the texture memory is not optimal considering the spatial locality caching. For instance for the GTX 480, the texture cache hit rate is on average 69%. This lowers the access times somewhat, leading to approximately the same access times as global memory, but with some extra overhead.

GPU	GTX 280	GTX 480
Total	5.480	5.534
Kernel	0.517	0.334

Table 5.29: Performance of ZXY BGS on GPU, improved texture memory implementation, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	5.96 GB/s	7.95 MB/s	131.3 GB/s
GTX 280	5.27 GB/s	2.02 MB/s	63.89 GB/s

Table 5.30: Throughput of improved texture memory implementation

Lookup map word size

The last few optimizations are concerning the word size used in the lookup map. The access patterns to the input and output frames are already coalesced, as seen in [38]. Due to the way we have implemented the player pixels lookup map, and the pattern the kernel process pixels, the memory accesses to the player pixels lookup map are also coalesced. However, it would still be interesting to test other data types to use for representing the lookup map, to see if this can further improve performance while retaining memory coalescence. In these tests we use the global memory implementation because of both performance and implementation simplicity.

Integer representation: The first change of representation is to use integers as the basic type in the lookup map, instead of the original chars. The results of the benchmarks are seen in tables 5.31 and 5.32. We can see that the average throughput has drastically decreased, compared to the original char-lookup-map version. In addition, the total processing times have increased substantially, which can be explained by the need for transferring more data from the CPU to the GPU, due to integers resulting in a larger lookup map. The kernel performance has also decreased, as expected, due to the increase in word length limiting the amount of coalesced memory accesses. It is also interesting to see that the kernel performance hit on the GTX 280 is much larger than on the GTX 480. This can be explained by the compute 2.x, and higher, architectures having looser requirements for memory coalescence, while also having some limited L2 caching of global memory.

Clearly, using integers for representing the player pixels lookup map does not increase the performance; rather the opposite.

GPU	GTX 280	GTX 480
Total	6.661	6.602
Kernel	0.649	0.342

Table 5.31: Performance of ZXY BGS on GPU, global memory with integer representation of player pixels lookup map, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	5.74 GB/s	7.95 MB/s	131.78 GB/s
GTX 280	5.06 GB/s	2.02 MB/s	63.69 GB/s

Table 5.32: Throughput of global memory implementation with integer representation of player pixels lookup map

Short representation: The last change of representation is to use shorts as the basic type in the lookup map. The results can be found in tables 5.33 and 5.34. Here we see a relatively clear drop in throughput for both GTX cards. In addition, the average total processing times have increased, which is primarily caused by the increased size of the lookup map, leading to slower transfer to the device. Furthermore, the kernel processing times have increased, but not as much as for the integer based version. The reason for the kernel performance decrease, is the same as for the integer version, i.e. short representation leading to worse memory coalescence. Like for the integer version, the performance decrease is larger for the GTX 280, for the same reasons, i.e. no caching and stricter requirements for memory coalescence.

In other words, we see that changing from chars to shorts in the lookup map decreases the performance.

5.7.4 ZXY BGS GPU performance summary

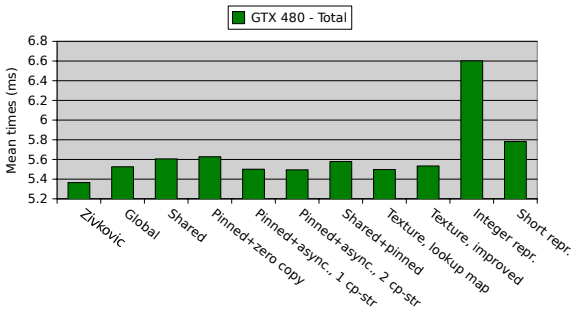
The performance of the different GPU based ZXY BGS implementations is found in figure 5.9. Here we can see mean processing times of the GTX 480 and GTX 280, for both

GPU	GTX 280	GTX 480
Total	5.815	5.782
Kernel	0.522	0.338

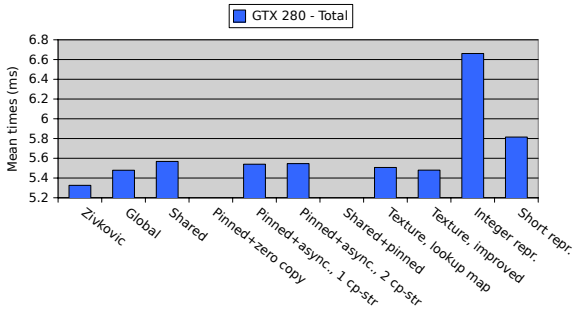
Table 5.33: Performance of ZXY BGS on GPU, global memory with short representation of player pixels lookup map, mean times (ms)

GPU	Average	Minimum	Maximum
GTX 480	5.81 GB/s	7.96 MB/s	130.95 GB/s
GTX 280	4.93 GB/s	2.00 MB/s	63.89 GB/s

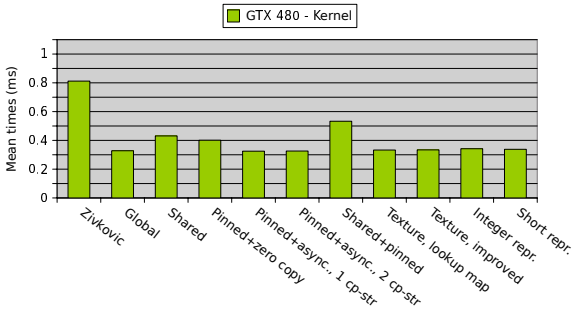
Table 5.34: Throughput of global memory implementation with short representation of player pixels lookup map



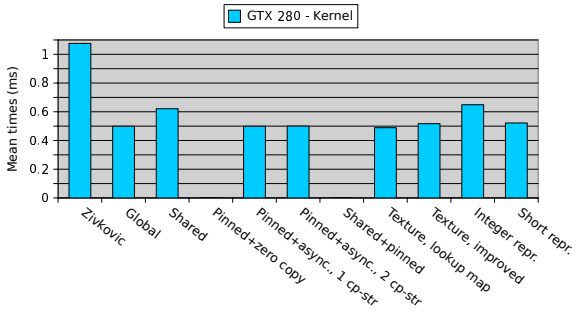
(a) Mean total performance of ZXY BGS implementations, GTX 480



(b) Mean total performance of ZXY BGS implementations, GTX 280



(c) Mean kernel performance of ZXY BGS implementations, GTX 480



(d) Mean kernel performance of ZXY BGS implementations, GTX 280

Figure 5.9: Mean performance of ZXY BGS GPU implementations

kernels and total processing. We can see that for the kernels, versions using global and texture memory, i.e. the *'global memory'* implementation, both *'pinned + asynchronous transfer'* implementations and both *'texture memory'* implementations, perform the best. When looking at the total processing times, the unmodified version is the fastest. However, of the modified versions, the same versions as for the kernel benchmarks perform the best.

5.7.5 Remaining "CUDA C Best Practices Guide" optimizations

We have mentioned that the BPG contains several rated steps for optimizing CUDA applications. We have so far been through most of them, but some have not been tested. First of all, as a high priority step, we have not tried to minimize the data transfer between host and device by executing sequential code on the GPU. The PCIe bus is often the bottleneck, so if we limit the amount of data to be transferred, even if the task itself is executing sequentially on the GPU, we can potentially increase the performance. The task in question is the player pixel lookup map generation. A way to implement this, is to retrieve the next set of player position samples from the database on the CPU, and then transfer these to the GPU. The amount of data to transfer would in this case be less than for transferring the whole lookup map. The generation of the lookup map would then be executed on the GPU. As long as the generation of this map on GPU is approximately equal to the generation on the CPU, we would see a performance increase.

There exist many libraries for fast math computing in CUDA. These are already optimized for their use, but often at the cost of precision. However, if the speed of the calculation is more important than the precision of the result, to use these libraries is an easy way to improve performance. Such libraries are of no use for our implementation, and this step is therefore ignored.

There also exist specialized math functions for some calculations, in addition to the more general ones. These often have better performance than the general implementations, due to the fact that these can make use of limitations in the input, size, problem area, etc, to optimize the performance. Therefore, if a specialized math function exists; use it. However, as for the fast math libraries, there are no cases where this can be used.

Another step is to use signed integers rather than unsigned integers as loop counters. The semantics of unsigned integers are well defined in C, which gives the compiler little freedom to optimize the use of an unsigned integer. However, signed integers are not well defined, which means the compiler can optimize a bit more aggressively. Therefore, when using signed instead of unsigned integers as loop counters, the compiler can optimize more freely, leading to a bit better performance, for very little programming work. However, we have not spent any time on this, because we did not add any loops to the unmodified Zivkovic GPU implementation, while the unmodified implementation already use signed integers for its loop counters.

Division and modulo operations are expensive, but shift operations are very fast. Therefore, if we want to divide an integer by a multiple of 2, or use modulo by a multiple of 2, we can rather use shift operations to save processing cycles. However, we did not implement any division or modulo operations, which made this step pointless for us.

Furthermore, it is important to avoid automatic conversions between floats and doubles, which cost processing cycles. This can happen when for instance dividing a float by 2.0. 2.0 is then treated as a double, which then needs to be converted to a float. This can be prevented by simply using the f suffix, which forces it to be a float, such as 2.0f. There are not doubles and floats in the ZXY modification, and this step is already implemented in the Zivkovic GPU code.

We have already discussed that branching and diverging code can decrease the

overall performance. However, the compiler contains mechanisms for branch predication to prevent warps from diverging. The programmer can help with this, by for instance unrolling loops using the "pragma" keyword. We discussed the branch divergence earlier, where we saw that this was not a problem. This step is therefore omitted.

5.7.6 The optimal implementation for a standalone Background Subtractor

When looking at the performance of the implementations, we see that the general bottleneck is the transfer between the CPU and the GPU. As [38] states, asynchronous transfers allows the execution of CPU execution, GPU transfers and GPU execution to be interleaved, so we want a solution that exploits this to reduce the cost of CPU⇔GPU transfers. The optimal solution for a standalone ZXY Background Subtractor is therefore the implementation using pinned memory for asynchronous memory transfers with global memory for storing the lookup map, described in section 5.7.3. In addition to providing the fastest kernel and the fastest data transfers, this version allows for asynchronous data transfer between host and device by use of pinned memory and double buffering.

However, we need to be aware of the nature of double buffered, asynchronous transfers. This is because that, when we retrieve the next foreground mask from the BGS, this is the mask for the input frame two frames earlier. This is caused by the new input frame not being processed directly, but merely being copied to the GPU on the first step, and processed on the second, then transferred back on the third. Then, on the second next call for the BGS (with new input frames), we get the corresponding foreground. One solution would be for the user of the BGS to always cache the two previous input frames, but this adds some unnecessary complexity for the user.

Another solution would be to let the ZXY BGS kernel copy the input frame that corresponds to the new foreground mask to a separate GPU buffer, and then transfer it back to the CPU together with the corresponding foreground mask. Of course, adding more memory transfers affect performance. The processing times of this solution can be seen in table 5.35. We can see that the total average processing times per frame have increased to approximately 8.7 ms per frame on the GTX 480 and GTX 280, both still safely within the real time constraints.

GPU	GTX 280	GTX 480
Total	8.746	8.687
Kernel	0.499	0.325

Table 5.35: Performance of standalone ZXY BGS, with caching of corresponding input frame on GPU, mean times (ms)

An even better solution, however, would be to cache the two previous input frames on CPU in the C++ object that wraps around the CUDA kernel. This way we do not need to transfer anything extra back to the CPU from the GPU, which saves several milliseconds of processing time per frame, and uses less of the limited PCIe bandwidth. This hides the complexity from the user, in addition to not increasing the total processing times of the global memory/asynchronous transfer-implementation described

in section 5.7.3 by a large margin. The performance of this solution can be found in table 5.36. We see that the kernel performance is approximately the same, while the total processing times have increased by 1.7 ms compared to the non-caching solution. This extra delay is caused by two extra memcpys; one for retrieving the corresponding frame, and one for storing the new input frame. However, the performance is 1.5 ms better than the GPU-cache based version. Compared to the fastest CPU version, this is a speedup factor of 3. This is therefore the variant we use in our standalone ZXY BGS application, named ZxyBackgroundSubtractorGPU.

GPU	GTX 280	GTX 480
Total	7.200	7.204
Kernel	0.525	0.325

Table 5.36: Performance of standalone ZXY BGS, with caching of corresponding input frame on CPU, mean times (ms)

An example showing the accuracy of the application can be seen in figure 5.10, where we have replaced the white foreground pixels with the actual video pixels. The image is cropped to remove the outer areas that, thanks to the ZXY modification, contain no noise. In the resulting image, there is almost no noise, where most of the existing noise is caused by the bench area being included by the player pixels lookup. In addition, the accuracy of the players are excellent, where only one player is partially hidden due to ZXY inaccuracies, as explained in section 5.6. The other players, however, are completely classified as foreground.

Another thing worth mentioning, is the fact that we implemented the ZxyBackgroundSubtractorGPU to contain a fall-back mode for situations where it does not have ZXY player data. In this situation, the BGS algorithm degrades into the unmodified Zivkovic-implementation, which means that the whole field is processed.

5.7.7 The optimal implementation for the Bagadus stitching pipeline

We have now looked at what the optimal solution for a standalone ZXY BGS is. However, we need to modify the BGS somewhat to make it work as a module in the Bagadus panorama pipeline. First of all, as part of the pipeline, the BGS module itself no longer needs to transfer data between the CPU and GPU, because this is done in separate Uploader and Downloader modules. We can therefore strip the BGS of these tasks. Therefore, the kernel we use is the global memory-based one, described in section 5.7.3. Nevertheless, it needs to be slightly modified to support several cameras concurrently, but this should not affect performance in any way, other than the increased processing load. In addition, the calculation of the player pixel lookup maps has to be done on the CPU, so the BGS module ends up with a GPU part and a CPU-part, as described in section 4.5.3.

Concerning the input and output buffer pattern described in chapter 4, we need to do few modifications. The BGS already has the necessary input buffers, and the foreground mask as output buffer. We therefore only need to add a video frame output buffer, because we need to pass the input frames corresponding to the foreground masks further into the pipeline. The buffers can be seen in table 5.37.

Figure 5.10: ZxyBackgroundSubtractorGPU accuracy

Input	Size (Byte)	Output	Size (Byte)
Player pixel lookup map	*pixelcount*	Foreground mask	*pixelcount*
Input frame	$4 \times$ *pixelcount*	Corresponding input frame	$4 \times$ *pixelcount*

Table 5.37: Background subtractor module input and output

Note, however, that even though we do not transfer from the CPU to the GPU directly in the BGS module itself, the advantages of the asynchronous transfer we have discussed earlier are so great, that we choose to use this technique in the Uploader module, which is responsible for transferring frames and player pixel lookup maps to the GPU.

5.8 Background Subtractor applications

Now that we have looked in detail into the process of background subtraction, and how we can optimize it both in accuracy and performance, it is interesting to take a look at application usage for BGS in our scenario.

5.8.1 Panoramic stitching with dynamic seam calculation

In the old Bagadus version, the visual artifacts created by the static seam stitcher are quite visible, especially when a player passes through a seam, which we saw an example of in figure 2.8. This is why the dynamic seam based stitching algorithm in section 4.5.9 was introduced.

We have already discussed how this algorithm works. In short, it starts by calculating a new stitching seam for each overlapping area between two neighbor cameras.

This is done for every new frame set. Each seam is calculated by going from the bottom of a frame to the top, through this overlapping area, where the seam will follow a least weighted path. The search area through the overlap has a limited width, and the pixels within it are treated as graph nodes. The weight of the edges are calculated by using a custom weight function that compares the absolute color differences between the corresponding pixels in each of the currently overlapping frames. Dijkstra's algorithm is then used to find the cheapest path through this overlapping area. To improve performance and visual results, the search algorithm is not allowed to move to pixels directly to the left, right or downwards. By using the absolute color value differences as the edge weights, the search algorithm calculates a seam that goes through the path in the frames where the difference between the two frames are the smallest. This means that we find the path leading to the least visible seam. For instance, when the search algorithm reaches sudden color changes, like for a player pixel, the algorithm finds a path around the player, without cutting him or her.

However, the algorithm described so far contains a critical flaw; what if some of the players wear suits of green color? Then there would be possibilities where the cheapest path found is through a player, due to them also being partially green. This would then lead to visual artifacts in the seam, similar to what we see in figure 2.8, only that the seam now cuts differently. An example of this happening is seen in figure 5.11(a), where the white player is cut due to the white lines. A visualization of the problem can be seen in figure 5.11(b).

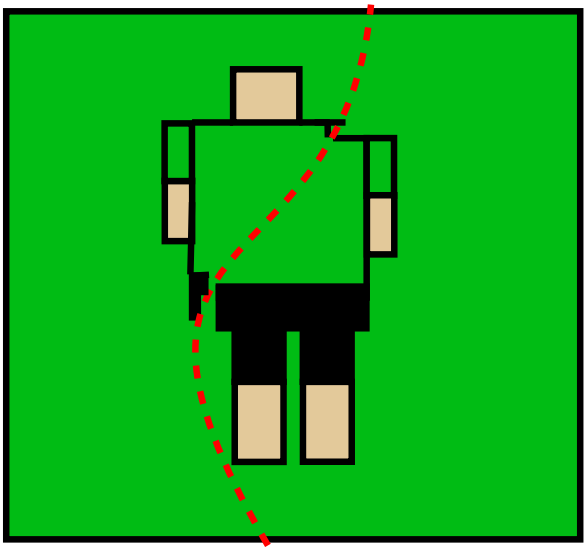
To prevent this, the algorithm is modified to support the use of background subtraction. As described in earlier sections, the background subtractor provides foreground masks for each camera as output. In these masks, background, shadows and foreground are marked with different values. The dynamic seam algorithm can then use these foreground masks, by increasing the weight of all edges that lead to a pixel marked as foreground in the masks. The new weight is set to a very high value, which ensures that the seam never will cross a player pixel, other than in extreme cases, such as the rare case when player pixels fill the overlapping search area from side to side. Compared to the case in figure 5.11(b), where the green player was cut by the seam, using the foreground mask provided by the background subtractor, the seam now avoids the player, and therefore eliminates visual artifacts. See figure 5.11(c) for a visualization.

As usual, when the seam, which is returned as horizontal pixel offsets for each vertical pixel for each overlap, has been calculated, the stitcher copies pixels from the correct frames into the correct positions in the new panorama frame. More details about this dynamic seam stitching algorithm can be found in [54], by fellow master student Espen Helgedagsrud.

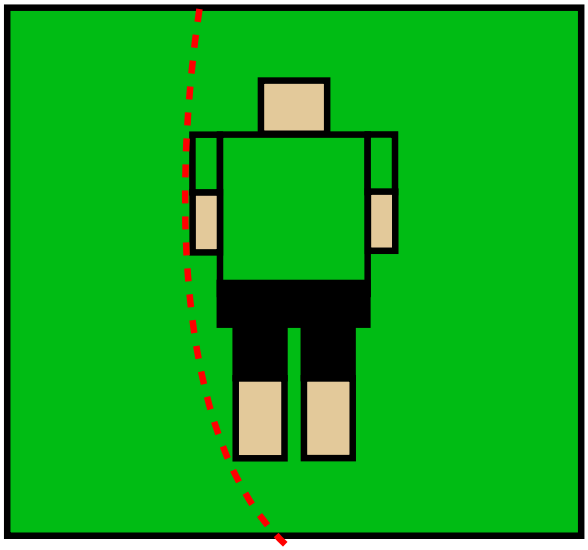
5.8.2 Depth map creation

In parallel with the work on the improved pipeline, Henrik Kjus Alstad, a fellow master student at Simula, has been doing research on real-time depth-map estimation in his master thesis [78], using soccer as a case study. In short, a depth map is an array or image channel, consisting of a value for each pixel, denoting the distance between the view point and the surface in the corresponding pixel. The brighter the value, the

(a) Example of dynamic seam cutting player. The picture is a bit too bright due to bad auto exposure.



(b) Dynamic stitching seam, camouflaged player, no BGS usage

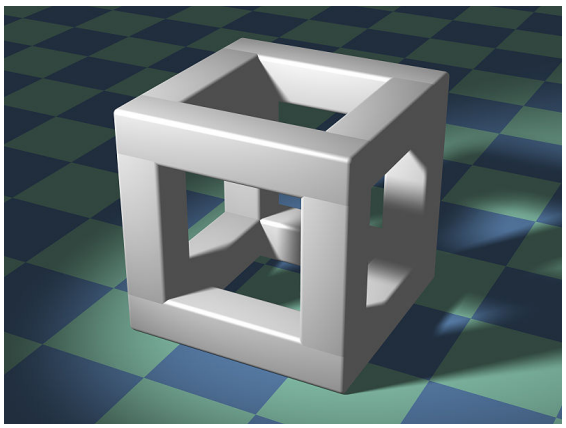


(c) Dynamic stitching seam, camouflaged player, BGS usage

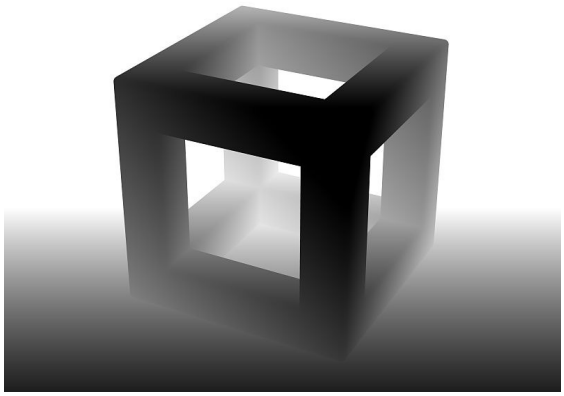
Figure 5.11: Dynamic seam calculation comparison - with and without BGS usage.

farther away the object in that pixel is, and vice versa. The dimensions of a depth map are therefore the same as for the corresponding input image. An example of an image with its corresponding depth map, can be seen in figure 5.12. We can here see that the darker areas are closer to the view point than the lighter areas.

The method to calculate depth maps selected, is to use two cameras viewing the field from slightly different positions. This is called stereo matching, and is more accurate than using only one camera. For more details about this, see [78]. The pipeline described can be seen in figure 5.13. The first, off-line step, is to calibrate the cameras, which needs to be done each time the two cameras are moved. Then, the first on-line step, is to rectify the images. This means that we transform the images from the two cameras to project onto a common plane (more details can be found in Henrik's thesis).



(a) The original input image [79]



(b) The corresponding depth map, visualized as a gray scale image [80]

Figure 5.12: A depth map

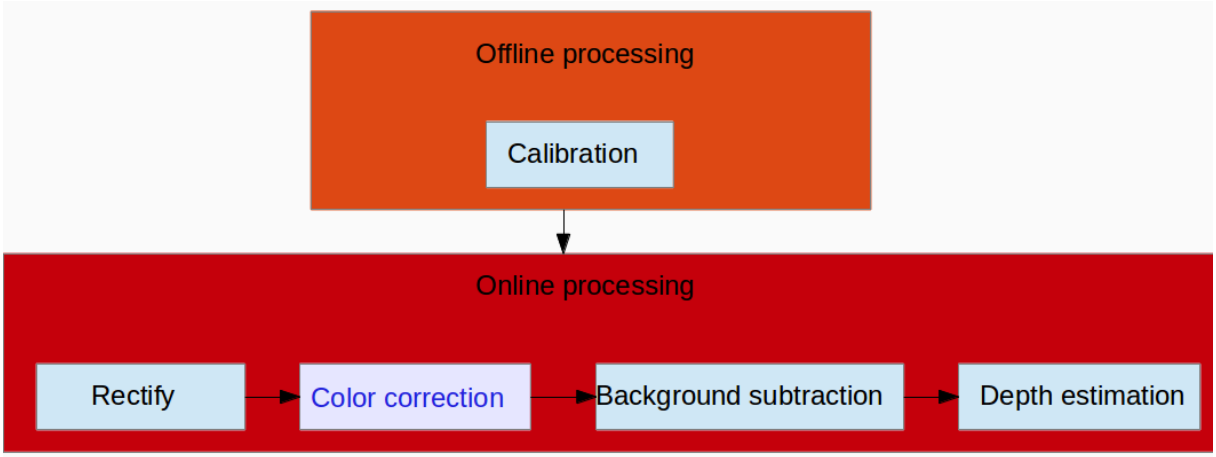


Figure 5.13: The depth map calculation pipeline

The next step is to do color correction. This is currently not done, but a color corrector has already been implemented for the Bagadus panorama stitcher pipeline (see [52] and section 4.5.8), and this implementation can be modified to work for the depth map calculation pipeline.

The next step is where the background subtractor comes into play. The process of calculating depth maps are computationally very expensive, and to generate them in real-time at resolutions of 1280x960 pixels is a huge challenge. The goal by using BGS is to decrease the amount of pixels to be processed during the depth estimation step, to decrease the total processing load. The ZXY BGS is therefore used to calculate the depth for only the areas around the soccer players, and then use a simpler depth model for the remaining image. This leads to both better performance and more accurate visual results with less noise. The last step is where the depth estimation is executed and depth maps calculated.

A visual example of the effects of using BGS during depth map calculation can be found in figure 5.14. In figures 5.14(a) and 5.14(b) we find the input images. In

figure 5.14(c), we find the disparity map resulting from the calculations. The disparity is inversely proportional to the depth, meaning that the closer the object in the pixel is to the cameras, the higher (and therefore brighter) the values are. The result of applying background subtraction on the disparity map is seen in figure 5.14(e). We see that BGS greatly decreases the amount of noise.

5.8.3 Visual features during delivery to user

Another area where background subtraction can prove to be useful, is during delivery of video content to the user. For instance, by using the background subtractor, players can be emphasized on playback by removing all pixels but the player pixels, and put the players on a monotone background, such as we did in figure 5.10. In addition, this can allow for more accurate player tracking, where we can use the foreground masks to track player pixels and limit inaccuracies in the ZXY sensors, and then use the ZXY data to lookup what player each foreground pixel belong to.

Furthermore, background subtraction can make the process of visual ball tracking easier. This is because we can use the foreground masks to filter out irrelevant pixels from the input frames, potentially leading to better tracking accuracy. Note, however, that in this case, as long as the ball does not contain any tracking sensors, we can not use the ZXY modification. This is caused by the ball in many cases being outside the player pixel boxes, meaning that it would often be marked as background.

5.9 Future works

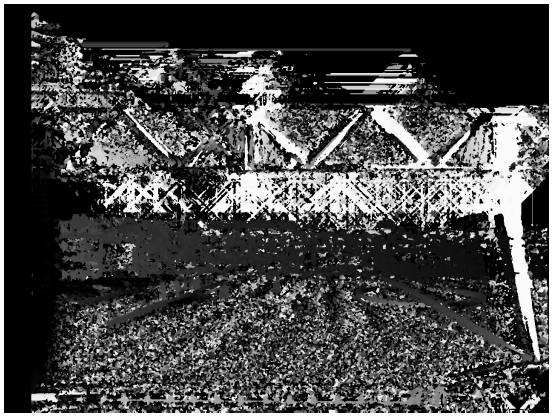
There are some future works for the ZXY Background Subtractor. First of all, we have not been able to record footage for and test the current algorithm well enough under some difficult weather conditions, such as snow, rain and lightning. This should therefore be done. Furthermore, we have only implemented and compared two background subtraction models, Zivkovic [65] and KaewTraKulPong [74], both Mixture of Gaussians type of models. An interesting step would therefore be to test different kinds of models, such as those mentioned in section 5.2, to see how they perform in the Bagadus scenario, and especially how well they perform with the ZXY optimization. In addition, as we saw in section 5.7.5, we have not investigated the effects of moving the player pixel lookup map generation to the GPU, which can be done by first transferring the coordinate data for all the players to the GPU, and then generate the lookup maps here.

5.10 Summary

In this chapter, we have gone into detail about the process of background subtraction. We have seen that background subtraction is the process of extracting the background from a video, and to create a mask of pixels belonging to the foreground. There are several challenges different BGS models need to solve, and we discussed which challenges are most critical in our scenario. For us, BGS is used as a tool to find the player pixels in the videos, which can then be used in the panorama pipeline described in Chapter 4.

(a) Example input image of the left camera

(b) Example input image of the right camera



(c) Resulting disparity map, *without* BGS

(d) The corresponding foreground mask for the left camera

(e) Resulting disparity map, *with* BGS

Figure 5.14: The effect of using background subtraction during depth map estimation

As part of this, we evaluated two promising background subtraction models, tweaked parameters to improve classification accuracy, and finally selected the Zivkovic model to be used.

After this, we discussed how we could optimize this BGS model, both in classification accuracy and performance, by exploiting the knowledge about player positions. This started by optimizing the ZXY BGS algorithm on CPU, where we went into details about how to make it perform in real-time, which we managed. The next step was to move the ZXY BGS to GPU to speed it up even more, which was done by modifying an existing GPU implementation of the Zivkovic model. We here looked into details about improving both total performance and the kernel performance even more, and described the best choices for making a standalone ZXY BGS application, plus how to modify the existing implementation to fit in a module in the improved panorama stitcher pipeline.

We have also looked at some different types of applications that can benefit from background subtraction. We first looked at the dynamic stitching seam algorithm that is used in the improved panorama pipeline, described in Chapter 4, and how we can eliminate potential visual artifacts by utilizing the foreground masks provided by the BGS module. We also saw how BGS can be used to improve the process of depth map creation. The benefits here are both in visual results and performance. Finally, we shortly discussed some possible scenarios on the playback side where BGS can be utilized. In the future, more research should be done on other kinds of applications that could benefit from the ZXY BGS, especially on the playback side of the Bagadus system.

We have in summary in this chapter, looked at what BGS is and how to optimize it with our tracking knowledge, both on CPU and GPU. In the next chapter we will summarize this thesis, discuss the main contributions, and look at future work.

Chapter 6

Conclusion

In this chapter, we summarize our work and present our primary contributions. In the end, we look at future improvements of the work done.

6.1 Summary

In this thesis, we have improved the old panorama stitcher pipeline of the Bagadus system, which have been described in Chapter 2, to run in real-time on a single commodity computer. In addition to increasing the performance, we have also been able to improve the visual results of the resulting panorama video.

Before we could start discussing the improved pipeline, we had to look at how to use CUDA for utilizing the power of GPUs for parallel execution tasks. This included looking at the Fermi architecture, the execution model, and especially the memory model. This was all described in Chapter 3.

We then discussed the improved pipeline. To be able to improve the performance of the old pipeline, we had to completely restructure the architecture of the old CPU based pipeline, reuse very few of the older CPU modules, and add many new ones. Examples of new modules are a dedicated warper module, dedicated stitching module, modules for transferring data between the CPU and the GPU, a color-corrector module, and a background subtractor module. This improved pipeline was described in detail in Chapter 4. To get such a large speedup compared to the old pipeline, we had to utilize the power of GPUs, by use of CUDA. Several modules, such as the BackgroundSubtractor, ColorCorrector, Warper and Stitcher were therefore built to execute on the GPU for faster execution. Furthermore, we discussed the performance of the new pipeline, and how it scaled based on different hardware configurations. We also looked at the web interface for scheduling and managing session recordings.

The background subtractor was added as a tool to improve the visual results of the generated panorama video. In this thesis, we have emphasized the details of background subtraction in Chapter 5, and how we were able to improve both the speed and accuracy of our BGS algorithm by using player tracking data. The tracking data was used, because it allows us to limit the amount of pixels to be processed in each frame. Here, we also went into details about how to optimize this algorithm both on the CPU and on the GPU. On the CPU, we looked at techniques such as player pixel lookup maps, dynamic and static cropping. On the GPU, we looked at memory related

optimizations, such as memory coalescence, different memory types and different representations. We have also seen how we can utilize this background subtractor to optimize the visual results of the dynamic seam stitcher used in the pipeline, to increase the performance and visual results of depth map creation, and to use it for different scenarios on the playback side of the system.

6.2 Main Contributions

We have in this thesis shown how a pipeline can be built to create stitched panorama videos in real-time from four HD-ready cameras, based on the Bagadus soccer scenario. This pipeline has been installed at Alfheim stadium, and a goal is to use it this season. We have shown how we can utilize GPUs to increase the performance, while the resulting panorama is of better visual quality. By doing this, we managed to get the pipeline to run on a single, inexpensive commodity computer, without any expensive and specialized hardware. This speedup allows coaches to access the generated panorama videos and single camera videos 5.3 seconds after a frame has been recorded, which means that coaches are able to use the system during half-time. As part of this, we have also installed a web interface at Alfheim for allowing coaches to schedule new recordings.

We have also analyzed and discussed the process of background subtraction in detail, and especially focused on how it is possible to optimize the BGS process, both in accuracy and performance. Here we investigated how we could utilize the known player position data to reduce the processing load, which in turn improved accuracy and performance. We have also found several examples of applications for this BGS, and there are more potential applications for using BGS as a tool in the future.

As part of optimizing both the BGS process and the stitching pipeline, we learned about techniques for optimizing applications, both on the CPU and GPU. For instance, on the CPU, we learned about increasing performance by reducing hot-spots and bottlenecks, such as by doing lookups instead of calculations, and by modifying the application to utilize the CPU caches better. On the GPU, we have learned about writing and optimizing applications in CUDA. A big part of this has been to learn how to increase GPU memory performance, because this is one of the most important aspects of CUDA application performance. A valuable lesson here, has been to see how CUDA application optimizations need to be targeted at specific architectures and GPUs for the best performance.

Furthermore, we have been able to submit and publish a poster at the GPU Technology Conference 2013 [17], where we presented the improved pipeline. We have also submitted a paper to ACM Multimedia 2013 [18], where the pipeline is described.

6.3 Future work

There is some work left as future work. We mentioned some of them for the panorama stitcher pipeline in section 4.13. A future goal here is to improve the performance of the pipeline even more. This includes moving more modules to the GPU, executing the CPU part of the BGS module as a separate module, optimizing existing modules,

and investigating the effects of changing the internal pixel representation from RGBA to RGB or YUV. A larger step is to research the possibilities of utilizing more than a single GPU for better scalability. Other future work is to test the pipeline with more than four cameras, and to test cameras with higher resolutions, such as 2K- and 4K-cameras. Furthermore, to support this, the pipeline needs better configurability and support for configuration-files, which also makes installation of the pipeline easier.

On the background subtraction side of things, future work includes testing the existing algorithm under more harsh conditions to stress test the ZXY BGS model even more. It would also be interesting to modify other existing BGS models of other classifications with this ZXY modification to find the ultimate BGS implementation for use in the Bagadus system. Furthermore, we have seen a couple of applications for the usability of BGS, and future works would also include research on other applications in the Bagadus system that could utilize BGS.

The most critical step at this point seen from a system perspective, however, is to deploy a proper video player that can be used by coaches to access the recorded sessions, including use of annotated events and tracking data, such as demonstrated in [2].

Appendix A

Accessing the source code

The source code for the Bagadus system, including what is described in this thesis, can be found at `https://bitbucket.org/mpg_code/bagadus`. To retrieve the code, run *git clone git@bitbucket.org:mpg_code/bagadus.git*.

Appendix B

Extra Tables

Computer	DevBox 2
Controller	1.791
Reader	33.285
Converter	13.855
Debarreler	16.302
Uploader	23.892
Uploader, BGS part*	13.202
BGS	8.423
Warper	15.391
Color-corrector	23.220
Stitcher	4.817
YUVConverter	9.938
Downloader	12.814
SingleCamWriter	24.424
PanoramaWriter	19.998
SingleCamWriter, diff	33.339
PanoramaWriter, diff	33.346
BGS, ZXY query†	657.597
Camera frame drops/1000	4
Pipeline frame drops/1000	0

Table B.1: Overall pipeline performance
Mean times (ms)
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

Pipeline Version	New (GPU)	Old (CPU)
Warper	15.141	133.882
Stitcher	4.912	521.042
Converter	9.676	26.520

Table B.2: Old vs new pipeline
Mean times (ms). DevBox 2.

GPU	GTX 280	GTX 480	GTX 580	GTX 680	GTX Titan
Uploader	73.036	27.188	23.269	23.375	22.426
BGS	36.761	13.284	8.193	7.123	7.096
Warper	66.356	19.487	14.251	14.191	13.139
ColorCorrector	86.924	28.753	22.761	21.941	19.860
Stitcher	23.493	8.107	5.552	4.307	4.126
YUVConverter	41.158	13.299	9.544	9.566	8.603
Downloader	53.007	16.698	11.813	11.958	11.452

Table B.3: GPU comparison, mean processing times (ms)

Module	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores	16 cores
Controller	4.023	4.103	3.898	4.107	3.526	3.906	3.717
Reader	32.885	33.132	33.275	33.292	33.287	33.280	33.281
Converter	18.832	16.725	15.170	13.601	12.635	12.874	12.319
Debarreler	27.469	19.226	16.903	14.573	13.171	12.659	12.106
Uploader	35.157	29.914	26.883	24.253	24.422	23.814	22.725
Uploader, BGS part*	18.482	15.865	14.325	13.171	12.474	12.505	11.834
SingleCamWriter	40.752	30.160	26.754	23.776	22.416	21.800	21.173
PanoramaWriter	35.405	23.865	20.119	17.272	15.567	15.084	14.050
SingleCamWriter, diff	46.427	36.563	33.875	33.317	33.355	33.331	33.438
PanoramaWriter, diff	48.629	37.152	33.965	33.320	33.354	33.330	33.320
BGS, ZXY query†	685.404	671.347	660.456	675.240	692.639	639.769	688.503
Camera frame drops/1000	75	26	7	9	6	8	8
Pipeline frame drops/1000	729	327	67	0	6	3	3

Table B.4: CPU core count scalability, without frame drop handling, mean times (ms)
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

	4 cores, no HT	4 cores, HT	8 cores, no HT	8 cores, HT	16 cores, no HT	16 cores, HT
Controller	4.257	4.023	4.044	3.898	3.391	3.717
Reader	32.392	32.885	32.947	33.275	33.278	33.281
Converter	14.041	18.832	13.319	15.170	11.164	12.319
Debarreler	24.840	27.469	16.808	16.903	10.453	12.106
Uploader	33.980	35.157	27.818	26.883	21.809	22.725
Uploader, BGS part*	16.417	18.482	13.405	14.325	11.143	11.834
SingleCamWriter	53.313	40.752	31.290	26.754	20.023	21.173
PanoramaWriter	53.544	35.405	29.613	20.119	16.903	14.050
SingleCamWriter, diff	63.642	46.427	38.845	33.875	33.323	33.438
PanoramaWriter, diff	67.494	48.629	39.831	33.965	33.319	33.320
BGS, ZXY query†	680.114	685.404	708.971	660.456	643.523	688.503
Camera frame drops/1000	223	75	54	7	5	8
Pipeline frame drops/1000	1203	729	477	67	3	3

Table B.5: HyperThreading scalability, without drop handling, mean times (ms)
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

Module	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores	16 cores
Controller	4.204	3.955	3.694	3.706	3.436	4.094	4.006
Reader	33.037	33.070	33.266	33.290	33.301	33.286	33.277
Converter	10.566	12.614	14.726	13.419	13.544	12.640	12.335
Debarreler	15.015	14.421	15.666	14.458	12.981	12.514	11.891
Uploader	19.857	23.015	26.076	25.008	24.137	23.554	23.487
Uploader, BGS part*	14.859	14.447	14.314	12.913	12.614	12.411	11.644
SingleCamWriter	23.763	23.689	25.607	23.910	21.995	21.792	20.969
PanoramaWriter	20.187	18.908	19.163	17.771	15.286	14.497	13.695
SingleCamWriter, diff	38.070	34.782	33.661	33.352	33.327	33.358	33.324
PanoramaWriter, diff	38.724	35.019	33.715	33.353	33.319	33.366	33.323
BGS, ZXY query†	656.593	679.531	669.598	699.519	641.223	636.265	668.108
Camera frame drops/1000	41	33	7	4	3	4	4
Pipeline frame drops/1000	343	177	37	6	2	7	3

Table B.6: CPU core count scalability, with frame drop handling, mean times (ms)
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

Module	No optimizations	O2	O3
Controller	4.006	4.045	3.821
Reader	33.277	33.308	33.302
Converter	12.335	12.162	12.576
Debarreler	11.891	12.162	12.100
Uploader	23.487	17.336	17.377
Uploader, BGS part†	11.644	5.644	5.399
SingleCamWriter	20.969	21.659	21.555
PanoramaWriter	13.695	14.695	14.797
SingleCamWriter, diff	33.324	33.327	33.321
PanoramaWriter, diff	33.323	33.323	33.317
BGS, ZXY query*	668.108	694.356	632.797
Camera frame drops/1000	4	2	3
Pipeline frame drops/1000	3	0	0

Table B.7: Compiler optimization comparison, mean times (ms)

* *Not a separate module, but is a part of the total Uploader time usage*

† *Not a module affecting the real-time constraint of the pipeline. Is executing separately*

Appendix C

Hardware Specifications

Computer name	DevBox 1
CPU	Intel Core i7-2600 @ 3.4 GHz
GPU	Nvidia Geforce GTX 460
Memory	8 GB DDR3 @ 1600 MHz
Pipeline output storage	Local NAS

Table C.1: DevBox 1 specifications

Computer name	DevBox 2
CPU	Intel Core i7-3930K @ 4.4 GHz
GPU	Nvidia Geforce GTX 680
Memory	32 GB DDR3 @ 1866 MHz
Pipeline output storage	Samsung SSD 840 Series, 500 GB

Table C.2: DevBox 2 specifications

Computer name	DevBox 3
CPU	Intel Core i7-960 @ 3.20GHz
GPU	Nvidia Geforce GTX 480
Memory	6 GB DDR3 @ 1066 MHz
Pipeline output storage	N/A

Table C.3: DevBox 3 specifications

Computer name	Server
CPU	2x Intel Xeon E5-2650 @ 2.0 GHz
GPU	Nvidia Geforce GTX 580
Memory	64 GB DDR3 @ 1600 MHz
Pipeline output storage	Samsung SSD 840 Series, 500 GB

Table C.4: Server specifications

GPU	Quadro NVS 295	Geforce GTX 280	Geforce GTX 480
Code name	G98	GT200	GF100
CUDA cores	8	240	480
Graphics clock	540 MHz	602 MHz	700 MHz
Compute capability	1.1	1.3	2.0
Total memory size	256 MB GDDR3	1024 MB GDDR3	1536 MB GDDR5
Memory clock	695 MHz	1107 MHz	1848 MHz
Memory interface	64-bit	512-bit	384-bit
Memory bandwidth	11.2 GB/s	141.7 GB/s	177.4 GB/s

Table C.5: GPU specifications, part 1

GPU	Geforce GTX 580	Geforce GTX 680	Geforce GTX Titan
Code name	GF110	GK104	GK110
CUDA cores	512	1536	2688
Graphics clock	772 MHz	1006 MHz	837 MHz
Compute capability	2.0	3.0	3.5
Total memory size	1536 MB GDDR5	2048 MB GDDR5	6144 MB GDDR5
Memory clock	4008 MHz	6000 MHz	6008 MHz
Memory interface	384-bit	256-bit	384-bit
Memory bandwidth	192.4 GB/s	192.2 GB/s	288.4 GB/s

Table C.6: GPU specifications, part 2

Appendix D

Improved Panorama Pipeline - Compiler Optimizations

We analysed and investigated the performance of the improved pipeline on several levels in Chapter 4. In addition to these tests, it is interesting to see how the GCC compiler can help to optimize the code. By use of several levels of optimization flags sent to the compiler, we can hopefully get the pipeline to perform better. We compare the performance of no compiler optimizations, using the O2 flag, and using the O3 flag. The mean processing times per module can be seen in table B.7 and figure D.1.

The results are quite interesting. We can see that of all the modules, only the Uploader gains noticeable performance from running any level of compiler optimization. This module however, gains a massive performance boost. These results can be explained rather easily. The controller itself does not gain much, due to it being relatively simple. The CamReader waits for new frames to be retrievable from the cameras, so it should therefore always use 33 ms in average. The Converter, Debarreler, SingleCamWriter and PanoramaWriter all use external libraries, such as ffmpeg, OpenCV and x264, for the heaviest processing. These libraries are installed on the machines, and has therefore already been compiled, so they do not gain much. The performance with and without such optimizations are therefore approximately equal for these modules. The GPU-based modules are not shown here, because they are executed on the GPU, and cannot be optimized by GCC. Note however, that the CUDA NVCC compiler always run compiler optimizations.

As stated, the only module that really gains anything from this, is the Uploader, and more specifically the part generating the player pixel lookup maps. This is because this code is written with almost no external library usage, and barely utilizes the internal NorthLight library (which is compiled together with our pipeline). In addition, it contains lots of nested loops and logic for handling ZXY data. This means that there are lots of potential for GCC to optimize the code. We can see that this halves the processing times of the lookup map creation, and in total cuts the processing times of the Uploader module by 25%.

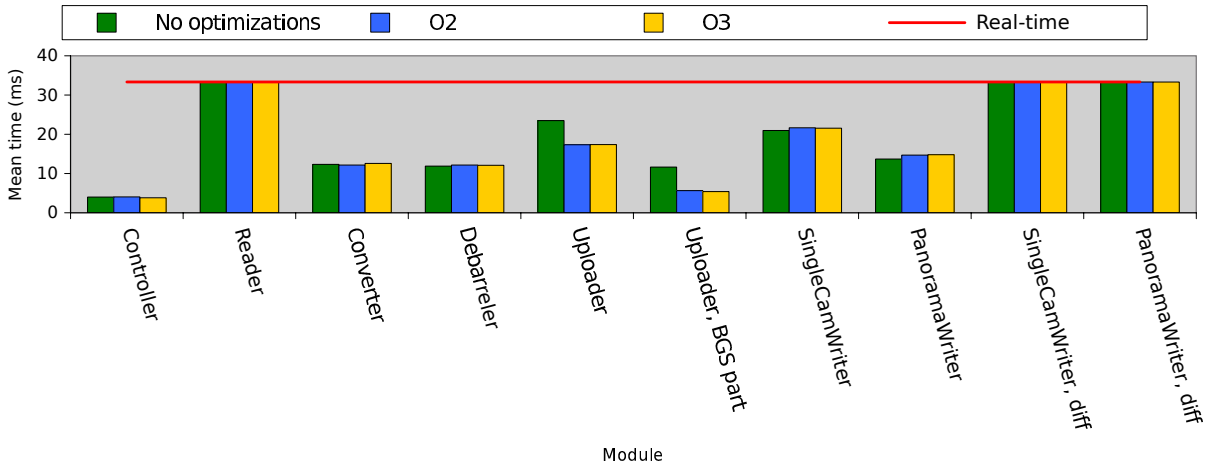


Figure D.1: Compiler optimization comparison

Bibliography

[1] Yuv. http://en.wikipedia.org/wiki/YUV. Accessed April 19, 2013.

[2] Simen Sægrov. Bagadus: next generation sport analysis and multimedia platform using camera array and sensor networks. Master's thesis, University of Oslo, 2012.

[3] nVIDIA. Nvidia's next generation cuda compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010. [Online; accessed 08-march-2013].

[4] Interplay sports. http://www.interplay-sports.com/.

[5] Prozone. http://www.prozonesports.com/.

[6] Stats technology. http://www.sportvu.com/football.asp.

[7] Camargus - premium stadium video technology inrastructure. http://www.camargus.com/.

[8] Zxy sport tracking. http://www.zxy.no/.

[9] Simen Sægrov, Alexander Eichhorn, Jørgen Emerslund, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Bagadus: An integrated system for soccer analysis (demo). In *Proceedings of the International Conference on Distributed Smart Cameras (ICDSC)*, October 2012.

[10] Pål Halvorsen, Simen Sægrov, Asgeir Mortensen, David K. C. Kristensen, Alexander Eichhorn, Magnus Stenhaug, Stian Dahl, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, and Dag Johansen. Bagadus: An integrated system for arena sports analytics - a soccer case study. In *Proceedings of the ACM Multimedia Systems conference (MMSys)*, February 2013.

[11] Matthew Brown and David G Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[12] Anat Levin, Assaf Zomet, Shmuel Peleg, and Yair Weiss. Seamless image stitching in the gradient domain. *Computer Vision-ECCV 2004*, pages 377–389, 2004.

[13] Jiaya Jia and Chi-Keung Tang. Image stitching using structure deformation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(4):617–631, 2008.

[14] Alec Mills and Gregory Dudek. Image stitching with dynamic elements. *Image and Vision Computing*, 27(10):1593–1602, 2009.

[15] Yao Li and Lizhuang Ma. A fast and robust image stitching algorithm. In *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, volume 2, pages 9604–9608. IEEE, 2006.

[16] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.

[17] Marius Tennøe, Espen O Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Håkon Kvale Stensland, Pål Halvorsen, and Carsten Griwodz. Realtime panorama video processing using nvidia gpus. GPU Technology Conference, March 2013.

[18] Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Efficient implementation and processing of a real-time panorama video pipeline. Submitted for publication, ACM Multimedia, 2013.

[19] Basler industrial cameras - ace series - aca1300-30gc. `http://www.baslerweb.com/products/ace.html?model=167&language=en`.

[20] Chroma subsampling. `http://en.wikipedia.org/wiki/Chroma_subsampling`. Accessed April 19, 2013.

[21] Martin Stensgård. Verdione camera trigger box (mini version). git@bitbucket.org:mastensg/verdione-mini-box.git, 2013.

[22] Dag Johansen, Magnus Stenhaug, Roger Bruun Asp Hansen, Agnar Christensen, and Per-Mathias Høgmo. Muithu: Smaller footprint, potentially larger imprint. In *Proceedings of the IEEE International Conference on Digital Information Management (ICDIM)*, pages 205–214, August 2012.

[23] Stian Dahl (ZXY). Personal communication.

[24] Itseez. Opencv | opencv. `http://opencv.org/`, 2013.

[25] Verdione. `http://www.verdione.org/`. Is at the time of delivery down. Google cache: `http://webcache.googleusercontent.com/search?q=cache:http://verdione.org/`.

[26] Software download - pylon / firmware | basler. `http://www.baslerweb.com/Support_SW_Downloads-18498.html?type=20&series=0&model=0`.

[27] Videolan - x264, the best h.264/avc encoder. `http://www.videolan.org/developers/x264.html`.

[28] Ffmpeg. `http://ffmpeg.org/`.

[29] Camera calibration with opencv. http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html, 2013.

[30] Bagadus - an integrated system for soccer analysis - youtube. http://www.youtube.com/watch?v=1zsgvjQkL1E, 2013.

[31] Parallel programming and computing platform | cuda | nvidia. http://www.nvidia.com/object/cuda_home_new.html.

[32] Nvidia. Cuda programming model overview. 2008.

[33] PCI-SIG. Pci express® 3.0 frequently asked questions pci-sig. http://www.pcisig.com/specifications/pciexpress/resources/PCIe_3_0_External_FAQ_Nereus.pdf.

[34] Rune Johan Hovland. Latency and bandwidth impact on gpu-systems. dec 2008.

[35] Nvidia. *CUDA C Programming Guide*, oct 2012.

[36] Nvidia. *CUDA C Best Practices Guide*, oct 2012.

[37] Alexander Ottesen. Efficient parallelisation techniques for applications running on gpus using the cuda framework. Master's thesis, University of Oslo, 2009.

[38] Vu Pham, Phong Vo, Vu Thanh Hung, and Le Hoai Bac. Gpu implementation of extended gaussian mixture model for background subtraction. 2010.

[39] Wai-Kwan Tang, Tien-Tsin Wong, and P-A Heng. A system for real-time panorama generation and display in tele-immersive applications. *Multimedia, IEEE Transactions on*, 7(2):280–292, 2005.

[40] Patrick Baudisch, Desney Tan, Drew Steedly, Eric Rudolph, Matt Uyttendaele, Chris Pal, and Richard Szeliski. An exploration of user interface designs for real-time panoramic photography. *Australasian Journal of Information Systems*, 13(2):151, 2006.

[41] Michael Adam, Christoph Jung, Stefan Roth, and Guido Brunnett. Real-time stereo-image stitching using gpu-based belief propagation. 2009.

[42] Software stitches 5k videos into huge panoramic video walls, in real time. http://www.sixteen-nine.net/2012/10/22/software-stitches-5k-videos-huge-panoramic-video-walls-real-time/, 2012. [Online; accessed 05-march-2012].

[43] Live ultra-high resolution panoramic video. http://www.fascinate-project.eu/index.php/tech-section/hi-res-video/. [Online; accessed 04-march-2012].

[44] O. Schreer, I. Feldmann, C. Weissig, P. Kauff, and R. Schafer. Ultrahigh-resolution panoramic imaging for format-agnostic video production. *Proceedings of the IEEE*, 101(1):99–114, Jan.

[45] Ding-Yun Chen, MurphyChien-Chang Ho, and Ming Ouhyoung. Videovr: A real-time system for automatically constructing panoramic images from video clips. In Nadia Magnenat-Thalmann and Daniel Thalmann, editors, *Modelling and Motion Capture Techniques for Virtual Environments*, volume 1537 of *Lecture Notes in Computer Science*, pages 140–143. Springer Berlin Heidelberg, 1998.

[46] Kevin Huguenin, Anne-Marie Kermarrec, Konstantinos Kloudas, and Francois Taiani. Content and geographical locality in user-generated content sharing systems. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 2012.

[47] Omar A. Niamut, Rene Kaiser, Gert Kienast, Axel Kochale, Jens Spille, Oliver Schreer, Javier Ruiz Hidalgo, Jean-Francois Macq, and Ben Shirley. Towards a format-agnostic approach for production, delivery and rendering of immersive media. In *Proceedings of Multimedia Systems (MMSys)*, pages 249–260, February 2013.

[48] Christian Weissig, Oliver Schreer, Peter Eisert, and Peter Kauff. The ultimate immersive experience: Panoramic 3d video acquisition. In Klaus Schoeffmann, Bernard Merialdo, AlexanderG. Hauptmann, Chong-Wah Ngo, Yiannis Andreopoulos, and Christian Breiteneder, editors, *Advances in Multimedia Modeling*, volume 7131 of *Lecture Notes in Computer Science*, pages 671–681. Springer Berlin Heidelberg, 2012.

[49] Stian Dahl (ZXY). Personal communication.

[50] Jason Garrett-Glaser. git.videolan.org git - x264.git blob - doc threads.txt. git://git.videolan.org/x264.git, 2010.

[51] Nvidia performance primitives | nvidia developer zone. `https://developer.nvidia.com/npp`.

[52] Mikkel Næss. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on color correction. Master's thesis, University of Oslo, 2013.

[53] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[54] Espen Oldeide Helgedagsrud. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching. Master's thesis, University of Oslo, 2013.

[55] Unknown. taskset(1) - linux man page. `http://linux.die.net/man/1/taskset`. Accessed: February 2013.

[56] Unknown. ps(1) - linux man page. `http://linux.die.net/man/1/ps`. Accessed: January 2013.

[57] Unknown. grep(1) - linux man page. `http://linux.die.net/man/1/grep`. Accessed: February 2013.

[58] Unknown. kill(1) - linux man page. `http://linux.die.net/man/1/kill`. Accessed: January 2013.

[59] The Apache Software Foundation. Welcome! - the apache http server project. `http://httpd.apache.org/`, 2012. Accessed: February 2013.

[60] Dolphin interconnect solutions - pci express interconnect. `http://www.dolphinics.com/`.

[61] L. Maddalena and A. Petrosino. A self-organizing approach to background subtraction for visual surveillance applications. *Image Processing, IEEE Transactions on*, 17(7):1168–1177, 2008.

[62] N.J.B. McFarlane and C.P. Schofield. Segmentation and tracking of piglets in images. *Machine Vision and Applications*, 8(3):187–193, 1995.

[63] Chris Stauffer and W. E L Grimson. Adaptive background mixture models for real-time tracking. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 2, pages –252 Vol. 2, 1999.

[64] Stephen J. McKenna, Sumer Jabri, Zoran Duric, Azriel Rosenfeld, and Harry Wechsler. Tracking groups of people. *Computer Vision and Image Understanding*, 80(1):42 – 56, 2000.

[65] Zoran Zivkovic and Ferdinand van der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recognition Letters*, 27(7):773 – 780, 2006.

[66] Liyuan Li, Weimin Huang, Irene Y. H. Gu, and Qi Tian. Foreground object detection from videos containing complex background. In *Proceedings of the eleventh ACM international conference on Multimedia*, MULTIMEDIA '03, pages 2–10, New York, NY, USA, 2003. ACM.

[67] O. Barnich and M. Van Droogenbroeck. Vibe: A powerful random technique to estimate the background in video sequences. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 945–948, 2009.

[68] Ahmed Elgammal, David Harwood, and Larry Davis. Non-parametric model for background subtraction. In *FRAME-RATE WORKSHOP, IEEE*, pages 751–767, 2000.

[69] N.M. Oliver, B. Rosario, and A.P. Pentland. A bayesian computer vision system for modeling human interactions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):831–843, 2000.

[70] Kyungnam Kim, T.H. Chalidabhongse, D. Harwood, and L. Davis. Background modeling and subtraction by codebook construction. In *Image Processing, 2004. ICIP '04. 2004 International Conference on*, volume 5, pages 3061–3064 Vol. 5, 2004.

[71] Kyungnam Kim, Thanarat H. Chalidabhongse, David Harwood, and Larry Davis. Real-time foreground-background segmentation using codebook model. *Real-Time Imaging*, 11(3):172–185, June 2005.

[72] S. Brutzer, B. Hoferlin, and G. Heidemann. Evaluation of background subtraction techniques for video surveillance. pages 1937–1944, june 2011.

[73] Birgi Tamersoy. Background subtraction. Lecture, University of Texas, sep 2009.

[74] P. Kaewtrakulpong and R. Bowden. An improved adaptive background mixture model for realtime tracking with shadow detection, 2001.

[75] A. Prati, I. Mikic, M.M. Trivedi, and R. Cucchiara. Detecting moving shadows: algorithms and evaluation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(7):918–923, 2003.

[76] Intel. Intel® vtune™ amplifier xe 2013. `http://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[77] Nvidia visual profiler | nvidia developer zone. `https://developer.nvidia.com/nvidia-visual-profiler`.

[78] Henrik Kjus Alstad. Towards real-time depth estimation in large spaces — a soccer case study. Master's thesis, University of Oslo, 2013.

[79] Cubic structure. `http://en.wikipedia.org/wiki/File:Cubic_Structure.jpg`. Accessed: 11/04/2013.

[80] Cubic frame stucture and floor depth map. `http://en.wikipedia.org/wiki/File:Cubic_Frame_Stucture_and_Floor_Depth_Map.jpg`. Accessed: 11/04/2013.