

Efficient Implementation and Processing of a Real-time Panorama Video Pipeline

Marius Tennøe¹, Espen Helgedagsrud¹, Mikkel Næss¹, Henrik Kjus Alstad¹, Håkon Kvale Stensland¹,
Vamsidhar Reddy Gaddam¹, Dag Johansen², Carsten Griwodz¹, Pål Halvorsen¹

¹University of Oslo / Simula Research Laboratory

²University of Tromsø

Abstract—High resolution, wide field of view video generated from multiple camera feeds has many use cases. However, processing the different steps of a panorama video pipeline in real-time is challenging due to the high data rates and the stringent requirements of timeliness.

We use panorama video in a sport analysis system where video events must be generated in real-time. In this respect, we present a system for real-time panorama video generation from an array of low-cost CCD HD video cameras. We describe how we have implemented different components and evaluated alternatives. We also present performance results with and without co-processors like graphics processing units (GPUs), and we evaluate each individual component and show how the entire pipeline is able to run in real-time on commodity hardware.

I. INTRODUCTION

Panoramic images are often used in applications that require a wide field of view. Some application scenario examples include surveillance, navigation, scenic views, infotainment kiosks, mobile e-catalogs, educational exhibits and sports analysis. In such systems, video feeds are captured using multiple cameras, and the frames are processed and stitched into a single unbroken frame of the whole surrounding region. This process is, however, very resource hungry. Each individual frame must be processed for barrel distortion, rotated to have the same angle, warped to the same plane, corrected for color differences, stitched to one large panorama image, encoded to save storage space and transfer bandwidth and, finally, written to disk. Additionally, the stitching includes searching for the best possible seam in the overlapping areas of the individual frames to avoid seams through objects of interest in the video.

In this paper, we describe our implementation of a real-time panorama video pipeline for an arena sports application. Here, we use a static array of low-cost CCD HD video cameras, each pointing at a different direction, to capture the wide field of view of the arena. These different views are slightly overlapped in order to facilitate the stitching of these videos to form the panoramic video. Several similar non-real-time stitching systems exist (e.g., [17]), and a simple non-real-time version of this system has earlier been described and demonstrated at the functional level [8], [20]. Our initial prototype is the first sports application to successfully integrate per-athlete sensors, an expert annotation system and a video system, but due to the non-real-time stitching, the panorama video was only available to the coaches some time after a game. The previous version also did not use any form of color correction or dynamic seam detection. Hence, the

static seam did not take into account moving objects (such as players), and the seam was therefore very visible. However, new requirements like real-time performance and better visual quality have resulted in a new and improved pipeline. Using our new real-time pipeline, such systems can be used during the game. In particular, this paper therefore focuses on the details of the whole pipeline from capturing images from the cameras via various corrections steps for panorama generation to encoding and storage of both the panorama video and the individual camera streams on disks. We describe how we have evaluated different implementation alternatives (both algorithms and implementation options), and we benchmark the performance with and without using graphics processing units (GPUs) as an co-processors. We evaluate each individual component, and, we show how the entire pipeline is able to run in real-time on a low-cost 6-core machine with a GPU, i.e., moving the 1 frame per second (fps) system to 30 fps enabling game analysis during the ongoing event.

The remainder of the paper is structured as follows: Next, in section II, we give a brief overview of the basic idea of our system. Then, we briefly look at some related work (section III), before we describe and evaluate our real-time panorama video pipeline in section IV. Section V provides a brief discussion of various aspect of the system before we conclude the paper in section VI.

II. OUR SPORTS ANALYSIS SYSTEMS

Today, a large number of (elite) sports clubs spend a large amount of resources to analyze their game performance, either manually or using one of the many existing analytics tools. For example, in the area of soccer, several systems enable trainers and coaches to analyze the gameplay in order to improve the performance. For instance, Interplay-sports, ProZone, STATS SportVU Tracking Technology and Camargus provide very nice video technology infrastructures. These systems can present player statistics, including speed profiles, accumulated distances, fatigue, fitness graphs and coverage maps using different charts, 3D graphics and animations. Thus, there exist several tools for soccer analysis. However, to the best of our knowledge, there does not exist a system that fully integrates all desired features in real-time, and existing systems still require manual work moving data between different components. In this respect, we have presented Bagadus [8], [20], which integrates a camera array video capture system with a sensor-based sport tracking system for player statistics and a system

for human expert annotations. Our system allows the game analytics to automatically playout a tagged game event or to extract a video of events extracted from the statistical player data. This means that we for example can query for all sprints faster than X or all situations where a player is in the center circle. Using the exact player position provided by sensors, a trainer can also follow individuals or groups of players, where the videos are presented either using a stitched panorama view of the entire field or by (manually or automatically) switching between the different camera views. Our prototype is currently deployed at an elite club stadium. We use a dataset captured at a premier league game to experiment and to perform benchmarks on our system. In previous versions of the system, the panorama video had to be generated offline, and it had static seams [8]. For comparison with the new pipeline presented in section IV, we next present the camera setup and the old pipeline.

A. Camera setup

To record the high resolution video of the entire soccer field, we have installed a camera array consisting of four Basler industry cameras with a 1/3-inch image sensors supporting 30 fps at a resolution of 1280×960. The cameras are synchronized by an external trigger signal in order to enable a video stitching process that produces a panorama video picture. The cameras are mounted close to the middle line, i.e., under the roof of the stadium covering the spectator area approximately 10 meters from the side line and 10 meters above the ground. With a 3.5 mm wide-angle lens, each camera covers a field-of-view of about 68 degrees, and the full field with sufficient overlap to identify common features necessary for camera calibration and stitching, is achieved using the four cameras. Calibration is done via a classic chessboard pattern [25].

B. The offline, static stitching pipeline

Our first prototype focused on integrating the different subsystems. We therefore did not put large efforts into real-time performance resulting in an unoptimized, offline panorama video pipeline that combined images from multiple, trigger-synchronized cameras as described above. The general steps in this stitching pipeline are: 1) correct the images for lens distortion in the outer parts of the frame due to a wide-angle fish-eye lens; 2) rotate and morph the images into the panorama perspective caused by different positions covering different

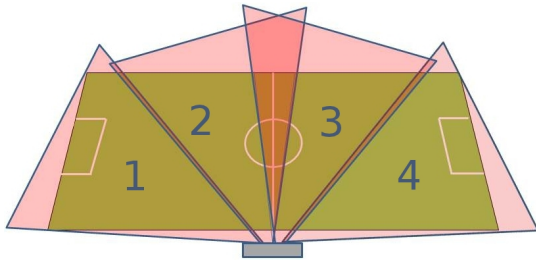


Figure 1. Camera setup at the stadium.

areas of the field; 3) rotate and stitch the video images into a panorama image; and 4) encode and store the stitched video to persistent storage. Several implementations were tested for the stitching operation such as the OpenCV planar projection, cylindrical projection and spherical projection algorithms, but due to the processing performance and quality of the output image, the used solution is a homography based algorithm.

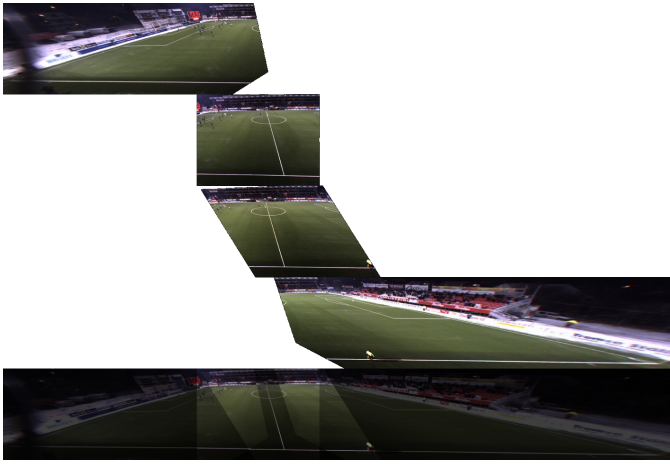
The first step before executing the pipeline, is to find corresponding pixel points in order to compute the homography between the camera planes [9], i.e., the head camera plane and the remaining camera planes. When the homography is calculated, the image can be warped (step 2) in order to fit the plane of the second image. The images must be padded to have the same size, and the seams for the stitching must be found in the overlapping regions (our first pipeline used static seams). Figure 2(a) shows the process of merging four warped images into a single, stitched panorama image, leading to the image found in figure 2(b). We also calculate the homography between the sensor data plane and the camera planes to find the mapping between sensor data coordinates and pixel positions.

As can be seen in the figure, the picture is not perfect, but the main challenge is the high execution time. On an Intel Core i7-2600 @ 3.4 GHz and 8 GB memory machine, the stitching operation consumed 974 ms of CPU time to generate each 7000×960 pixel panorama image [8]. Taking into account that the target display rate is 30 fps, i.e., requiring a new panorama image every 33 ms, there are large performance issues that must be addressed in order to bring the panorama pipeline from a 1 fps system to a 30 fps system. However, the stitching operations can be parallelized and parts of it offloaded to external devices such as GPUs, which, as we will see in section IV, results in a performance good enough for real-time, online processing and generation of a panorama video.

III. RELATED WORK

Real-time panorama image stitching is becoming common. For example, many have proposed systems for panorama image stitching (e.g., [6], [11], [14]–[16]), 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 frame-per-second (fps) video.

One of these existing systems is Camargus [1]. 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 Immersive Cockpit [22] 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



(a) Processing the images for stitching. Each of the 4 cameras, represented by the top four layers, are warped and padded to a horizontal line (the view of the second camera). Then, these 4 views are superimposed on each other. The highlighted areas in the bottom image show where the views overlap (where the seams are found).



(b) Result of homography based stitching with static seams.

Figure 2. The initial homography-based stitcher

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. [5] 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 (far below our 30 fps requirement). A system similar to ours is presented in [4], which computes stitch-maps on a GPU, but the presented system produces low resolution images (and is limited to 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 [3] describes a system by the Content Interface Corporation that creates ultra high resolution videos. The Omnicam system from the Fascinate [2], [21] project also produces high resolution videos. However, both these systems use expensive and specialized hardware. The system described in [3] also makes use of static stitching. A system for creating panoramic videos from already existing video clips is presented in [7], but it does not manage to create panorama videos within our definition of real-time. As far as we know, the same issue of real-time is also present in [5], [10], [17], [23].

In summary, existing systems (e.g., [7], [10], [17], [22], [23]) do not meet our demand of being able to generate the video in real-time, and commercial systems (e.g., [1], [3]) as well as the systems presented in [2], [21] do often not fit into

our goal to create a system with limited resource demands. The system presented in [4] is similar to our system, 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.

IV. A REAL-TIME PANORAMA STITCHER

In this paper, we do not focus on selecting the best algorithms etc., as this is mostly covered in [8]. The focus here is to describe the panorama pipeline and how the different components in the pipeline are implemented in order to run in real-time. We will also point out various performance trade-offs.

As depicted in figure 3, the new and improved panorama stitcher pipeline is separated into two main parts: one part running on the CPU, and the other running on a GPU using the CUDA framework. The decision of using a GPU as part of the pipeline was due to the parallel nature of the workload, and the decision of using the GPU has affected the architecture to a large degree. Unless otherwise stated (we have tested several CPUs and GPUs), our test machine for the new pipeline is an Intel Core i7-3930K (6-core), based on the Sandy Bridge-E architecture, with 32 GB RAM and an Nvidia GeForce GTX 680 GPU.

A. The Controller module

The single-threaded Controller is responsible for initializing the pipeline, synchronizing the different modules, handling global errors and frame drops, and transferring data between the different modules. After initialization, it will wait for and get the next set of frames from the camera reader (CamReader) module (see below). Next, it will control the flow of data from the output buffers of module N to the input buffers of module $N + 1$. This is done primarily by pointer swapping, and with memory copies as an alternative. It then signals all modules to process the new input and waits for them to finish processing. Next, the controller continues waiting for the next set of frames from the reader. Another important task of the Controller is to check the execution speed. If an earlier step in the pipeline runs too slow, and one or more frames has been lost from the cameras, the controller will tell the modules in the pipeline to skip the delayed or dropped frame, and reuse the previous frame.

B. The CamReader module

The CamReader module is responsible for retrieving frames from the cameras. It consists of one dedicated reader thread per camera. Each of the threads will wait for the next frame, and then write the retrieved frame to a output buffer, overwriting the previous frame. The cameras provide a single frame in YUV 4:2:2 format, and the retrieval rate of frames in the CamReader is what determines the real time threshold for the rest of the pipeline. As described above, the camera shutter synchronization is controlled by an external trigger box, and

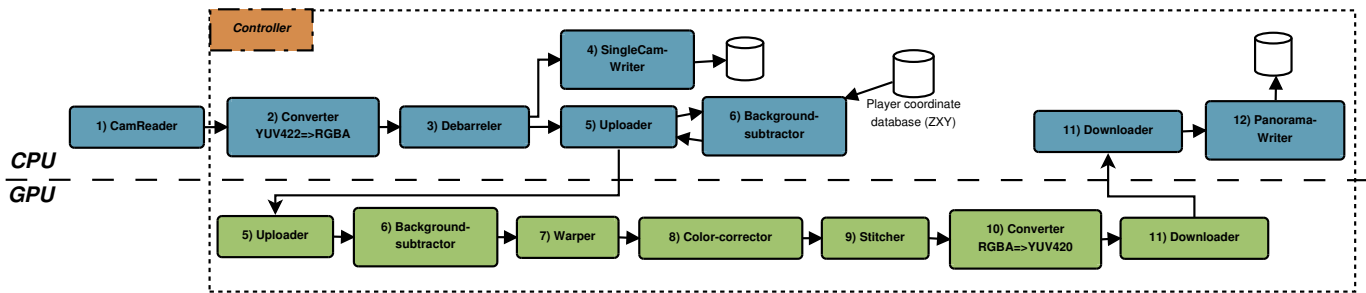


Figure 3. Panorama stitcher pipeline architecture. The orange and blue components run in the CPU and the green components run on the GPU.

in our current configuration, the cameras deliver a frame rate of 30 fps, i.e., the real-time threshold and the CamReader processing time are thus 33 ms.

C. The Converter module

The CamReader module outputs frames in YUV 4:2:2 format. However, because of requirements of the stitching pipeline, we must convert the frames to RGBA. This is done by the Converter module using *ffmpeg* and *swscale*. The processing time for these conversions on the CPU, as seen later in figure 6, is well below the real-time requirement, so this operation can run as a single thread.

D. The Debarreler module

Due to the wide angle lenses used with our cameras in order to capture the entire field, the images delivered are suffering from barrel distortion which needs to be corrected. We found the performance of the existing debarreling implementation in the old stitching pipeline to perform fast enough when executed as a dedicated thread per camera. The Debarreler module is therefore still based on OpenCV’s debarreling function, using nearest neighbor interpolation.

E. The SingleCamWriter module

In addition to storing the stitched panorama video, we also want to store the video from the separate cameras. This storage operation is done by the SingleCamWriter, which is running as a dedicated thread per camera. As we can see in [8], storing the videos as raw data proved to be impractical due to the size of uncompressed raw data. The different CamWriter modules (here SingleCamWriter) therefore encode and compress frames into 3 seconds long H.264 files, which proved to be very efficient. Due to the use of H.264, every SingleCamWriter thread starts by converting from RGBA to YUV 4:2:0, which is the required input format by the x264 encoder. The threads then encode the frames and write the results to disk.

F. The Uploader module

The Uploader is responsible for transferring data from the CPU to the GPU, in addition, the Uploader is also responsible for executing the CPU part of the BackgroundSubtractor (BGS) module (see section IV-G). The Uploader consists of a single CPU thread, that first runs the player pixel lookup creation needed by the BGS. Next, it transfers the current

RGBA frames and the corresponding player pixel maps to the GPU. This is done by use of double buffering and asynchronous transfers.

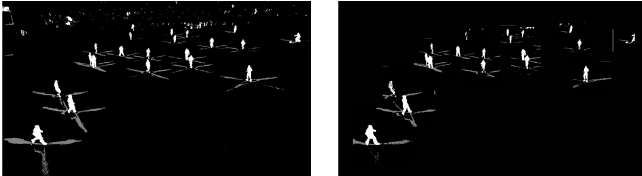
G. The BackgroundSubtractor module

Background subtraction is the process of determining which pixels of a video that belong to the foreground and which pixels that belong to the background. The BackgroundSubtractor module, running on the GPU, generates a foreground mask (for moving objects like players) that is later used in the Stitcher module later to avoid seams in the players. Our BackgroundSubtractor can run like traditional systems searching the entire image for foreground objects. However, we can also exploit information gained by the tight integration with the player sensor system. In this respect, through the sensor system, we know the player coordinates which can be used to improve both performance and precision of the module. By first retrieving player coordinates for a frame, we can then create a player pixel lookup map, where we only set the players pixels, including a safety margin, to 1. The creation of these lookup maps are executed on the CPU as part of the Uploader. The BGS on GPU then uses this lookup map to only process pixels close to a player, which reduces the GPU kernel processing times, from 811.793 microseconds to 327.576 microseconds on average on a GeForce GTX 680. When run in a pipelined fashion, the processing delay caused by the lookup map creation is also eliminated. The sensor system coordinates are retrieved by a dedicated slave thread that continuously polls the sensor system database for new samples.

Even though we enhance the background subtraction with sensor data input, there are several implementation alternatives. When determining which algorithm to implement, we evaluated two alternatives: Zivkovic [26], [27] and Kaew-TraKulPong [13]. Even though the CPU implementation was slower (see table I), Zivkovic provided the best visual results, and was therefore selected for further modification. Furthermore, the Zivkovic algorithm proved to be fast enough when modified with input from the sensor system data. The GPU implementation, based on [19], proved to be even faster, and the final performance numbers for a single camera stream can be seen in table I. A visual comparison of the unmodified Zivkovic implementation and the sensor system-modified version is seen in figure 4 where the sensor coordinate modifica-

| BGS version | Min | Max | Mean |
|--|------|-------|------|
| KaewTraKulPong CPU - unmodified | 40.0 | 66.4 | 48.9 |
| Zivkovic CPU - unmodified | 50.4 | 106.8 | 79.3 |
| Zivkovic CPU - coordinate modification | 10.0 | 51.1 | 23.3 |
| Zivkovic GPU - coordinate modification* | 4.6 | 11.1 | 5.5 |

Table I
EXECUTION TIME (MS) OF ALTERNATIVE ALGORITHMS FOR THE
BACKGROUND SUBTRACTOR MODULE (1 CAMERA STREAM).



(a) Unmodified Zivkovic BGS. (b) Player sensor data modification of Zivkovic BGS.

Figure 4. Background subtraction comparison.

tion reduce the noise as seen in the upper parts of the pictures.

H. The Warper module

The Warper module is 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. Like we have seen from the old pipeline, this is necessary because the stitcher assumes that its input images are perfectly warped and aligned to be stitched to a large panorama. Executing on the GPU, the Warper also warps the foreground masks provided by the BGS module. This is because the Stitcher module at a later point will use the masks and therefore expects the masks to fit perfectly to the corresponding warped camera frames. Here, we use the Nvidia Performance Primitives library (NPP) for an optimized implementation.

I. The Color-corrector module

When recording frames from several different cameras pointing in different direction, it is nearly impossible to calibrate the cameras to output the exact same colors due to the different lighting conditions. This means that, to generate the best panorama videos, we need to correct the colors of all the frames to remove color disparities. In our panorama pipeline, this is done by the Color-corrector module running on the GPU.

We choose to do the color correction after warping the images. The reason for this is that locating the overlapping regions is easier with aligned images, and the overlap is also needed when stitching the images together. This algorithm is executed on the GPU, enabling fast color correction within our pipeline. The implementation is based on the algorithm presented in [24], but have some minor modifications. We calculate the color differences between the images for every single set of frames delivered from the cameras. Currently,

| | Min | Max | Mean |
|---------------------------------|--------|--------|--------|
| CPU (Intel Core i7-2600) | 1995.8 | 2008.3 | 2001.2 |
| GPU (Nvidia Geforce GTX 680) | 0.0 | 41.0 | 23.2 |

Table II
COLOR CORRECTION TIMINGS (MS).

we color-correct each image in a sequence, meaning that each image is corrected according to the overlapping frame to the left. The algorithm implemented is easy to parallelize and does not make use of pixel to pixel mapping which makes it well suited for our scenario. Table II shows a comparison between running the algorithm on the CPU and on a GPU.

J. The Stitcher module

Like in the old pipeline, we use a homography based stitcher where we simply create seams between the overlapping camera frames, and then copy pixels from the images based on these seams. These frames need to follow the same homography, which is why they have to be warped. Our old pipeline used static cuts for seams, which means that a fixed rectangular area from each frame is copied directly to the output frame. Static cut panoramas are faster, but can introduce graphical errors in the seam area, especially when there are movement in the scene (illustrated in figure 5(a)).

To make a better seam with a better visual result, we therefore introduce a dynamic cut stitcher instead of the static cut. The dynamic cut stitcher creates seams by first creating a rectangle of adjustable width over the static seam area. Then, it treats all pixels within the seam area as graph nodes. The graph is then directed from the bottom to the top in such a way that each pixel points to the three adjacent pixels above (left and right-most pixels only point to the two available). Each of these edge's weight are calculated by using a custom function that compares the absolute color difference between the corresponding pixel in each of the two frames we are trying to stitch. The weight function also checks the foreground masks from the BGS module to see if any player is in the pixel, and if so it adds a large weight to the node. In effect, both these steps will make edges between nodes where the colors differs and players are present have much larger weights. We then run the Dijkstra graph algorithm on the graph to create a minimal cost route. Since our path is directed upwards, we can only move up or diagonally from each node, and we will only get one node per horizontal position.

| | 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 III
DYNAMIC STITCHING (MS).

An illustration of how the final seam looks can be seen in figure 5(c), where the seams without and with color correction are shown in figures 5(b). Timings for the dynamic stitching module can be seen in table III. The CPU version is currently slightly faster than our GPU version (as searches and branches

| | Min | Max | Mean |
|------------------------------|------|------|------|
| CPU (Intel Core i7-2600) | 33.2 | 61.1 | 42.1 |
| GPU (Nvidia Geforce GTX 680) | 0.2 | 36.5 | 13.4 |

Table IV
RGBA TO YUV 4:2:0 CONVERSION (MS).

often are more efficient on traditional CPUs), but further optimization of the CUDA code will likely improve this GPU performance. Note that the min and max numbers for the GPU are skewed by frames dropping (no processing), and the initial run being slower.

K. The YuvConverter module

Before storing the stitched panorama frames, we need to convert back from RGBA to YUV 4:2:0 for the H.264 encoder, just as in the SingleCamWriter module. However, due to the size of the output panorama, this conversion is not fast enough on the CPU, even with the highly optimize *swscale*. This module is therefore implemented on the GPU. In table IV, we can see the performance of the CPU based implementation versus the optimized GPU based version.

Nvidia NPP contains several conversion primitives, but not a direct conversion from RGBA to YUV 4:2:0. The GPU based version is therefore first using NPP to convert from RGBA to YUV 4:4:4, and then a self written CUDA code to convert from YUV 4:4:4 to YUV 4:2:0.

L. The Downloader module

Before we can write the stitched panorama frames to disk, we need to transfer it back to the CPU, which is done by the Downloader module. It runs as a single CPU thread that copies a frame synchronously to the CPU. We could have implemented the Downloader as an asynchronous transfer with double buffering like the Uploader, but since the performance as seen in figure 6 is very good, this is left as future work.

M. The PanoramaWriter module

The last module, executing on the CPU, is the Writer that writes the panorama frames to disk. The PanoramaWriter uses the same settings as the SingleCamWriter (three second H.2264 video files).

N. Pipeline performance

In order to evaluate the performance of our pipeline, we used an off-the-shelf PC with an Intel Core i7-3930K processor and an nVidia GeForce GTX 680 GPU. We have benchmarked each individual component and the pipeline as a whole capturing, processing and storing 1000 frames from the cameras.

In the initial pipeline [8], the main bottleneck was the panorama creation (warping and stitching). This operation alone used *974 ms per frame*. As shown by the breakdown into individual components' performance in figure 6, the new pipeline has been greatly improved. Note that all individual components run in real-time running concurrently on the same set of hardware. Adding all these, however, gives times far larger than 33 ms. The reason why the pipeline is still running

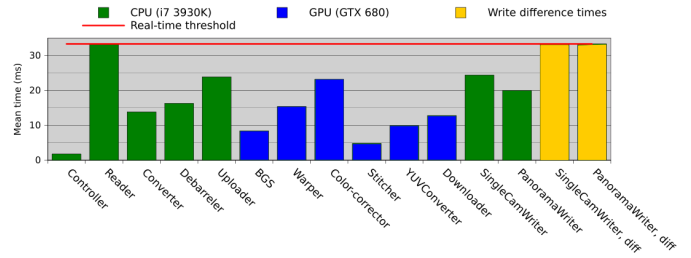
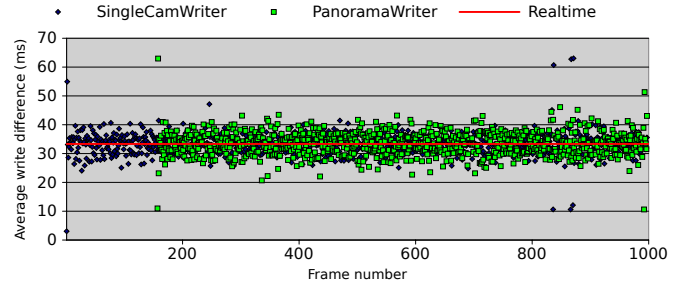
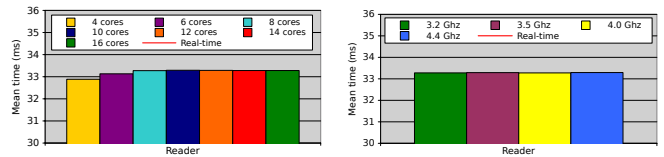


Figure 6. Improved stitching pipeline performance, module overview (GeForce GTX 680 and Core i7-3930K)



(a) Inter-departure time for all processed frames.



(b) Core count scalability.

(c) Core frequency scalability.

Figure 7. Inter-departure time of frames when running the entire pipeline. In a real-time scenario, the output rate should follow the input rate (given here by the trigger box) at 30 fps (33ms).

in real-time is because several frames are processed in parallel. Note here that all CUDA kernels are executing at the same time on a single GPU, so the performance of all GPU modules are affected by the performance of the other GPU modules. On earlier GPUs like the GTX 280, this was not allowed, but concurrent CUDA kernel execution was introduced in the Fermi architecture [18] (GTX 480 and above). Thus, since the Controller module schedules the other modules according to the input rate of 30 fps, the amount of resources are sufficient for real-time execution.

For the pipeline to be real-time, the output rate should follow the input rate, i.e., deliver all output frames (both 4 single camera and 1 panorama) at 30 fps. Thus, to give an idea of how often a frame is written to file, figure 7 shows individual and average frame inter-departure rates. The figures show the time difference between consecutive writes for the generated panorama as well as for the individual camera streams. Operating system calls, interrupts and disk accesses will most likely cause small spikes in the write times (as seen in the scatter plot in figure 7(a)), but as long as the average times are equal to the real-time threshold, the pipeline can be considered real-time. As we can see in figures 6, 7(b) and 7(c), the average frame inter-arrival time (Reader) is equal to the average frame inter-departure time (both SingleCamWriter and



(a) The original stitch pipeline in [8]: a fixed cut stitch with a straight vertical seam, i.e., showing a player getting distorted in the seam.

(b) 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.

(c) 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 5. Stitcher comparison - improving the visual quality with dynamic seams and color correction.

PanoramaWriter). This is also the case testing other CPU frequencies and number of available cores. Thus, the pipeline runs in real-time.

As said above and seen in figure 7(a), there is a small latency in the panorama pipeline compared to writing the single cameras immediately. The total panorama pipeline latency, i.e. the end to end frame delay from read from the camera until written to disk, is equal to 33 ms per sequential module (as long as the modules perform fast enough) plus a 5 second input buffer (the input buffer is because the sensor system has at least 3 second latency before the data is ready for use). The 33 ms are caused by the camera frame rate of 30 fps, meaning that even though a module may finish before the threshold, the Controller will make it wait until the next set of frames arrive before it is signaled to re-execute. This means that the pipeline latency is 5.33 seconds per frame on average.

V. DISCUSSION

Our soccer analysis application integrates a sensor system, soccer analytics annotations and video processing of a video camera array. There already exist several components that can be used, and we have investigated several alternatives in our research. Our first prototype aimed at full integration at the system level, rather than being optimized for performance. In this paper, however, our challenge has been of an order of magnitude harder by making the system run in real-time on low-cost, off-the-shelf hardware.

The new real-time capability also enables future enhancements with respect to functionality. For example, several systems have already shown the ability to serve available panorama video to the masses [10], [17], and by also generating the panorama video live, the audience can mark and follow particular players and events. Furthermore, ongoing work also include machine learning of sensor and video data to extract player and team statistics for evaluation of physical and tactical performance. We can also use this information to make video playlists [12] automatically giving a video summary of extracted events.

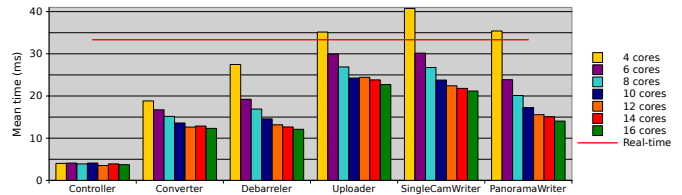


Figure 8. Core count scalability

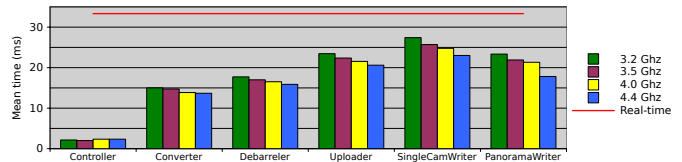


Figure 9. CPU frequency scalability

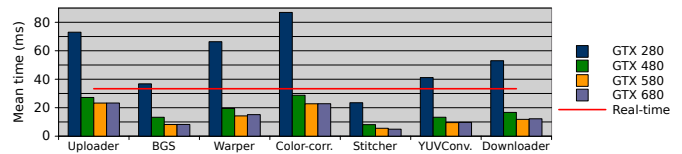


Figure 10. GPU comparison

Due to limited availability of resources, we have not been able to test our system with more cameras or higher resolution cameras. However, to still get an impression of the scalability capabilities of our pipeline, we have performed several benchmarks changing the number of available cores, the processor clock frequency and GPUs with different architecture and compute resources. Figure 8¹ shows the results changing the number of available cores that can process the many concurrent threads in the CPU-part of pipeline (figure 7(b) shows that the pipeline is still in real-time). As we can observe from the figure, every component runs in real-time using more than 4 cores, and the pipeline as a whole using 8 or

¹Note that this experiment was run on a machine with more available cores (16), each at a lower clock frequency (2.0 GHz) compared to the machine installed at the stadium which was used for all other tests.

more cores. Furthermore, the CPU pipeline contains a large, but configurable number of threads (86 in the current setup), and due to the many threads of the embarrassingly parallel workload, the pipeline seems to scale well with the number of available cores. Similar conclusions can be drawn from figure 9 where the processing time is reduced with a higher processor clock frequency, i.e., the pipeline runs in real-time already at 3.2 GHz, and there is almost a linear scaling with CPU frequency (figure 7(c) shows that the pipeline is still in real-time). Especially the H.264 encoder scales very good when scaling the CPU frequency. With respect to the GPU-part of the pipeline, figure 10 plots the processing times using different GPUs. The high-end GPUs GTX 480 and above (Compute 2.x and higher) all achieve real-time performance on the current setup. The GTX 280 is only compute 1.3 which does not support the concurrent CUDA kernel execution in the Fermi architecture [18], and the performance is therefore slower than real-time. As expected, more powerful GPUs reduce the processing time. For now, one GPU fulfills our real-time requirement, we did therefore not experiment with multiple GPUs, but the GPU processing power can easily be increased by adding multiple cards. Thus, based on these results, we believe that our pipeline easily can be scaled up to both higher numbers of cameras and higher resolution cameras.

VI. CONCLUSIONS

In this paper, we have presented a prototype of a real-time panorama video processing system. The panorama prototype is used as a sub-component in a real sport analysis system where the target is automatic processing and retrieval of events at a sports arena. We have described the pipeline in detail, where we use both the CPU and a GPU for offloading. Furthermore, we have provided experimental results which prove the real-time properties of the pipeline on a low-cost 6-core machine with a commodity GPU, both for each component and the combination of the different components forming the entire pipeline.

The entire system is under constant development, and new functionality is added all the time. Ongoing work therefore also include scaling the panorama system up to a higher number of cameras and to higher resolution cameras. So far, the pipeline scales nicely with the CPU frequencies, number of cores and GPU resources. We plan to use PCI Express-based interconnect technology from Dolphin Interconnect Solutions for low latency and fast data transfers between machines. Experimental results in this respect is though ongoing work and out of scope in this paper.

REFERENCES

- [1] Camargus - Premium Stadium Video Technology Infrastructure. <http://www.camargus.com/>.
- [2] Live ultra-high resolution panoramic video. <http://www.fascinate-project.eu/index.php/tech-section/hi-res-video/>. [Online; accessed 04-march-2012].
- [3] 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].
- [4] M. Adam, C. Jung, S. Roth, and G. Brunnett. Real-time stereo-image stitching using gpu-based belief propagation. 2009.
- [5] P. Baudisch, D. Tan, D. Steedly, E. Rudolph, M. Uyttendaele, C. Pal, and R. Szeliski. An exploration of user interface designs for real-time panoramic photography. *Australasian Journal of Information Systems*, 13(2):151, 2006.
- [6] M. Brown and D. G. Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.
- [7] D.-Y. Chen, M.-C. Ho, and M. Ouhyoung. Videovr: A real-time system for automatically constructing panoramic images from video clips. In *Proc. of CAPTECH*, pages 140–143, 1998.
- [8] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaus, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen. Bagadus: An integrated system for arena sports analytics - a soccer case study. In *Proc. of MMSys*, pages 48–59, 2013.
- [9] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.
- [10] K. Huguenin, A.-M. Kermarrec, K. Kloudas, and F. Taiani. Content and geographical locality in user-generated content sharing systems. In *Proc. of NOSSDAV*, 2012.
- [11] J. Jia and C.-K. Tang. Image stitching using structure deformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(4):617–631, 2008.
- [12] D. Johansen, H. Johansen, T. Aarflot, J. Hurley, Å. Kvalnes, C. Gurrin, S. Sav, B. Olstad, E. Aaberg, T. Endestad, H. Riiser, C. Griwodz, and P. Halvorsen. DAVVI: A prototype for the next generation multimedia entertainment platform. In *Proc. of ACM MM*, pages 989–990, Oct. 2009.
- [13] P. Kaewtrakulpong and R. Bowden. An improved adaptive background mixture model for realtime tracking with shadow detection, 2001.
- [14] A. Levin, A. Zomet, S. Peleg, and Y. Weiss. Seamless image stitching in the gradient domain. *Computer Vision-ECCV 2004*, pages 377–389, 2004.
- [15] Y. Li and L. Ma. A fast and robust image stitching algorithm. In *Proc. of WCICA*, volume 2, pages 9604–9608, 2006.
- [16] A. Mills and G. Dudek. Image stitching with dynamic elements. *Image and Vision Computing*, 27(10):1593–1602, 2009.
- [17] O. A. Niamut, R. Kaiser, G. Kienast, A. Kochale, J. Spille, O. Schreer, J. R. Hidalgo, J.-F. Macq, and B. Shirley. Towards a format-agnostic approach for production, delivery and rendering of immersive media. In *Proc. of MMSys*, pages 249–260, 2013.
- [18] 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].
- [19] V. Pham, P. Vo, V. T. Hung, and L. H. Bac. Gpu implementation of extended gaussian mixture model for background subtraction. 2010.
- [20] S. Sægrov, A. Eichhorn, J. Emerslund, H. K. Stensland, C. Griwodz, D. Johansen, and P. Halvorsen. Bagadus: An integrated system for soccer analysis (demo). In *Proc. of ICDS*, Oct. 2012.
- [21] 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.
- [22] W.-K. Tang, T.-T. Wong, and P.-A. Heng. A system for real-time panorama generation and display in tele-immersive applications. *IEEE Transactions on Multimedia*, 7(2):280–292, 2005.
- [23] C. Weissig, O. Schreer, P. Eisert, and P. Kauff. The ultimate immersive experience: Panoramic 3d video acquisition. In K. Schoeffmann, B. Merialdo, A. Hauptmann, C.-W. Ngo, Y. Andreopoulos, and C. Breiteneder, editors, *Advances in Multimedia Modeling*, volume 7131 of *Lecture Notes in Computer Science*, pages 671–681. Springer Berlin Heidelberg, 2012.
- [24] Y. Xiong and K. Pulli. Color correction for mobile panorama imaging. In *Proc. of ICIMCS*, pages 219–226, 2009.
- [25] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 666–673, 1999.
- [26] Z. Zivkovic. Improved adaptive gaussian mixture model for background subtraction. In *Proc. of ICPR*, pages 28–31 (vol. 2), aug. 2004.
- [27] Z. Zivkovic and F. van der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recognition Letters*, 27(7):773–780, 2006.