Distributed Video Stitching with PySpark

KOJI KUSUMI, University of California, Los Angeles, USA

This project is extends panorama image stitching to panorama video stitching. Video-stitching has a variety of interesting applications, including but not limited to: virtual reality, augmented reality, and robotics. In this work, I detail my system to create panorama videos. It utilizes Ffmpeg, OpenCV, and PySpark and to process video frames and stitch them together in an efficient manner. This system is composed of a master node that is responsible for segmenting videos and coordinating tasks. Individual worker nodes stitch groups of video segments. Stitching (the most computationally expensive part of the process) can be done in parallel. These individual panorama segments are collected at the master and concatenated to form a final video.

ACM Reference Format:
Koji Kusumi. 2021. Distributed Video Stitching with PySpark. 1, 1 (December 2021), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Image stitching has rich history of development and there are many mature algorithms to solve it. Many existing applications utilize image stitching. For example, people can use their smartphones to take multiple images and stitch them into a panorama or photosphere. This enables users to create images with a much greater field-of-view than what their camera natively captures.

However, applications than utilize video stitching are less prevalent. Image stitching is a very computationally expensive tasks. As such, it is difficult to extend to videos that are composed of hundreds or thousands of individual frames. Attempting to stitch videos using a single machine is prohibitively expensive. This limitation can be overcome by parallelizing work across multiple machines. In this project, I present my approach to doing just that. I will discuss how videos are decomposed, distributed, and successfully processed in parallel. I will also discuss design tradeoffs, design limitations, and areas for potential future work.

2 BACKGROUND

Since there is a rich body of work researching image stitching, I made use of a preexisting algorithm. In this way, I aws able to focus my efforts on system design. The algorithm I used is based on the image stitching algorithm developed by Brown et al. In this algorithm, four computational steps are performed to stitch a panorama. First, SIFT(scale-invariant feature transform) is used to extract image feature points and image descriptors. Once SIFT features have been extracted, features are matched across different images. Once the features are matched, a homography matrix is constructed that describes the projection from one image to the other. Finally, the homography matrix is applied to the images to create a final panorama.

There is also extensive work on distributed video processing. Many projects use Kafka to send individual frames as messages that are processed independently [1][3][5]. While promising, these techniques have a high

Author's address: Koji Kusumi, University of California, Los Angeles, 405 Hilgard Avenue, Los Angeles, California, USA, kojiboji@gmail.com.

XXXX-XXXX/2021/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

network overhead. Individual frames do not make use of information preserved between frames and lead to a huge expansion in data size.

Stride is a distributed video transcoding platform built on Spark[4]. Stride splits videos into segments and distributes them using a distributed file system. Each Spark task does not contain the video data itself. Instead, each task contains the path to the relevant video. Using the path, the workers download the relevant video segment before processing. Morph is another distributed video transcoding platform[2]. Like Stride, it first segments videos. Then, these segmented videos are passed to workers which process them in parallel.

3 VIDEO STITCHING DESIGN

A naive computational flow would consist consists of three steps. First videos would be decomposed into their subsequent frames. Next images in the set of frames would be processed using a regular image stitching algorithms, producing a set of panoramas. Finally, the set of panoramas would be collected to create a panorama video. Unfortunately, this computational flow is inefficient. First, this setup suffers from high network overhead. By reading individual frames, it greatly expands the size of the videos. A set of uncompressed frames can be thousands of times larger than the source video. This huge amount of data can also cause memory issues. In this naive flow, all of the panorama frames must be collected at the master (and stored in memory) before they can be written to a video. Since the uncompressed frames are so large, this limits this setup to very short videos.

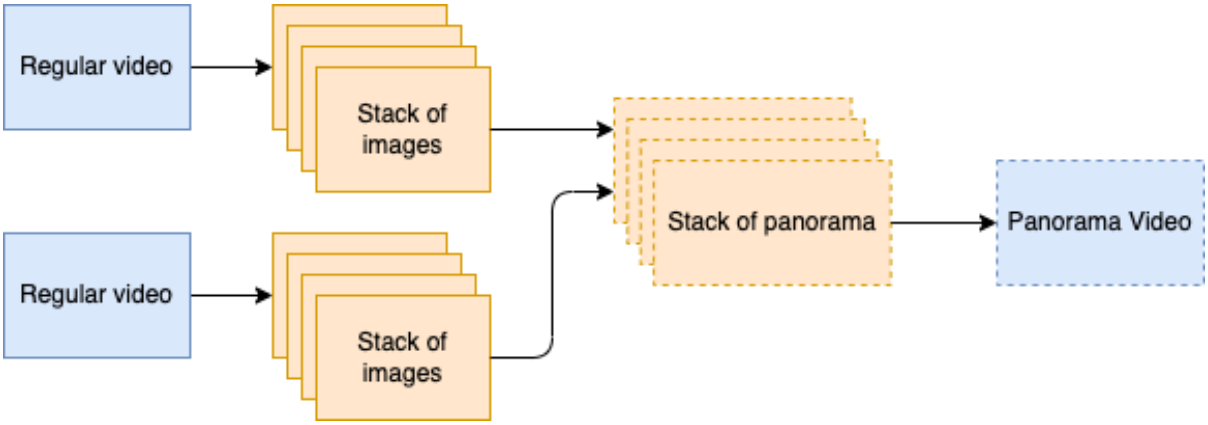


Fig. 1. A diagram showing a naive computational flow

The flow that this project uses is slightly more complex. First the master cuts input videos into reasonably sized segments. These videos are uploaded to S3 so that any worker can access them. Then Spark distributes tasks to different workers. Each task contains the S3 key name for its assigned segments. Workers access their videos from S3 and stitch frames one at a time. Since only one frame is being stitched at a time, workers require very little memory. When a task is complete, the final panorama segments are uploaded to S3. Finally, the master downloads all the video segments and concatenates them into a single video.

3.1 Libraries and System Setup

The libraries and frameworks that I used are Ffmpeg, OpenCV, PySpark, and boto3. FFMPEG is a open-sourced software project that contains a suite of tools for handling video and audio. It is used to segment input videos and concatenate panorama segments into a final video. OpenCV is a library containing implementations of a

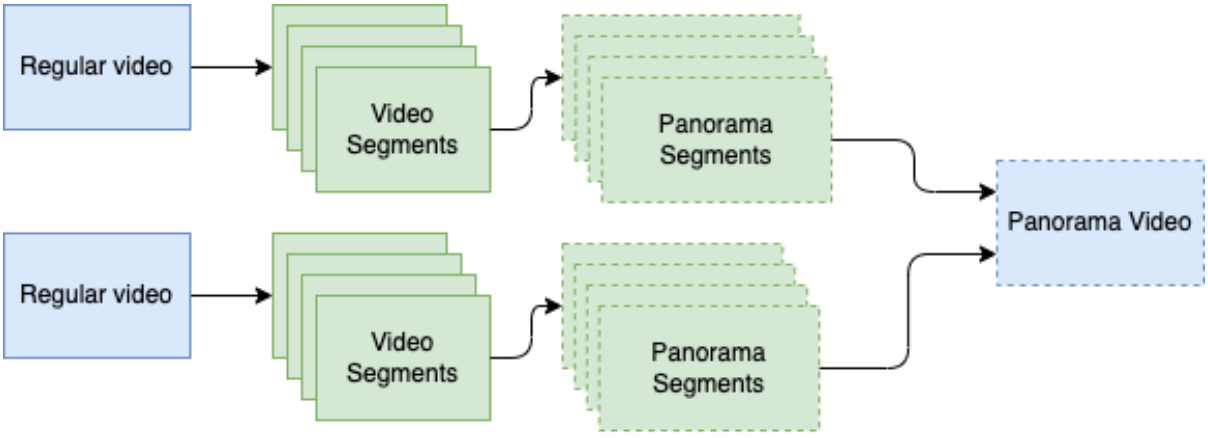


Fig. 2. A diagram showing an improved computational flow. Videos are split it segments to reduce network usage. Videos are split on key frames to avoid re-encodes

variety of computer vision algorithms. OpenCV was originally written in C++ for fast performance. However, it has Python bindings I used. OpenCV is used to stitch frames together and make panoramas.

PySpark is a package with Python bindings for Spark. Spark is one of the most widely used distributed computing frameworks. I used the PySpark framework to parallelize our stitching operations across multiple threads and machines. I chose Spark for its ease-of-use. It has extensive documentation and there are a plethora of example projects. In addition, since development was in Python, it is easy to prototype designs and made quick, iterative changes.

Finally, boto3 is used to create an S3 client. This client is used to access S3 in our python code.

3.2 Data Input

The first step in the data pipeline is cutting the input videos to create clips. Video are cut into even sized clips (10 seconds was found to be a reasonable time). To avoid expensive re-encodes, the input videos are cut on key frames. Unfortunately, key frames do not line up perfectly with the 10 second barriers. Thus, videos are cut slightly before and slightly after the clip barriers. Thus, each 10 seconds clip consists of three video segments: a short segment containing the very beginning, a large segment in the middle, and another short segment at the end. The short segments that contain the 10 seconds barriers overlap with adjacent clips. The input videos are segmented using Ffmpeg. Once the segments are generated, they are uploaded to S3, and their names are recorded.

3.3 Data Processing

The next step of the data pipeline is the actual image stitching. The driver creates tasks for each 10 second clip. When a task is created, the driver assigns all overlapping video segments to it. After the tasks are generated, they are converted into an RDD and distributed to the workers. A key deviation from a standard Spark programs is that rows in this RDD are not the actual data we care about. Instead, they contain information on where to find data. In this way, we are not utilizing Sparks in memory data capabilities. Instead, Spark is used more like a task managing framework. After a worker recieves a task, it reads one frame from each view and stitches them together to create a panorama. This panorama written to a temporary, local file. When one video segment is

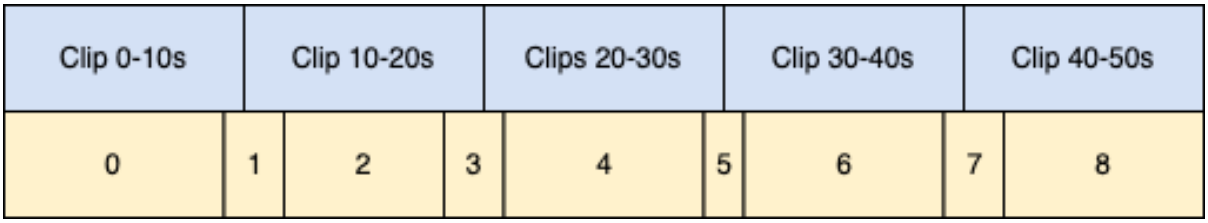


Fig. 3. A showing the overlap between clips and segments. The task assigned the clip from 0 to 10 seconds would utilize segments 0 and 1. The task assigned the clip from 10-20s would utilize segments 1, 2, and 3

completely read, the next one is opened. After the entire 10 second clip has been stitched, the temporary local file is uploaded to S3 and deleted. Its S3 key is returned to the driver.

3.4 Data Output

After the all the tasks are collected, the driver machine downloads all of the panorama segments from S3. Then, it uses Ffmpeg to concatenate them together. Since the panorama segments share the same dimensions, framerate and encoder, concatenating them can be done without any re-encodes.

4 EVALUATION

First I will explain how I collected the data used for evaluation. Then, I discus how I evaluated single-machine performance. Finally, I will cover how I deployed to a cluster to measure mulit-machine performance.

4.1 Data Acquisition

The videos used for testing were obtained in real-world scenarios. One video captured video of commuter traffic and the other video was of me juggling in my living room. Two views of each scene were captured; one on a Pixel3 and one on a Pixel3a. All videos had a resolution of 1920x1080 and a were captured at 30 frames per second. The phones were held in a home-made jig that kept their relative orientation constant. The phones were arranged so that they roughly overlapped in one half of the frame. The cameras were started manually. To ensure the best synchronization, the videos were inspected frame-by-frame. In the end, one frame was cut from the left view of the traffic video. For each scene, a 5 minute and 20 second video was taken. To test scalability, these base videos were cut into shorter segments: 80 seconds, 160 seconds, and 320 seconds.

Fig. 4. The rig devised to capture overlapping views from two phone cameras. The camera views overlapped in roughly 1/2 of the frame. The lenses of the cameras were oriented to be as close as possible

Fig. 5. Frames grabbed from the the test videos. Images from the traffic video are on top. Images from the juggling video are below

4.2 Stitching Quality

The stitching algorithm works well for individual frames. However, in some instances, there is noticeable jitter. The computed homograph matrix is not always consistent across frames, so the edges of the panorama video seem to shake. This issues is exacerbated in the traffic video. In the traffic video, the two views are more out of sync, and the high speed movement of cars means that the location of features can differ greatly between views. This problem could probably be avoided if the videos were pragmatically started.

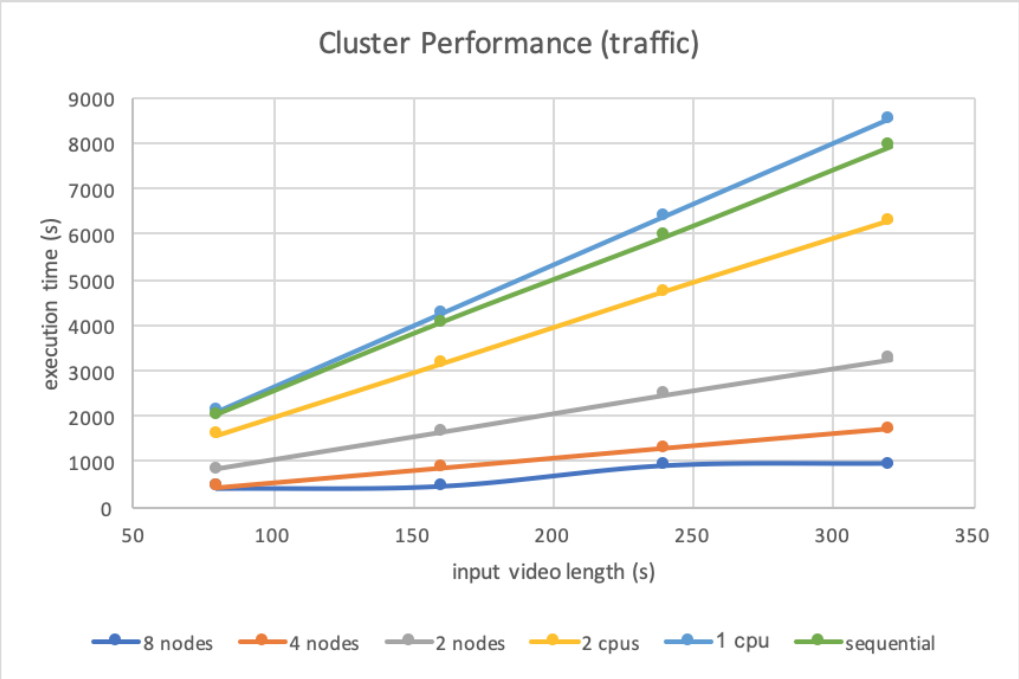
4.3 Single Machine Evaluation

In the first evaluation, I ran the script on a m5ad.large instance and compared it to a single-threaded implementation. The instance has 2 cores, and I tested utilizing both 1 core and 2 cores. The single-threaded implementation does not segment the videos or interact with S3. Instead is opens the base videos and creates a video frame-by-frame. I ran 3 tests for each video length and took the median time.

When using 2 cores Spark outperforms the single-threaded version. Conversely, when restricted to only 1 core, the Spark version performs worse. A point of interest is that when Spark changes from 1 to 2 cores. I observed is only a 1.3x speedup instead of the theoretical 2x speedup. This could be due to the extra overhead associated with Spark communication.

4.4 Multi-Workers Evaluation within Cluster

Next I conducted experiments on a EC2 cluster. The nodes in the cluster had 2 virtual CPUs, 6GB RAM, and 128GB disk storage. I ran three different setups: 2 nodes, 4 nodes, and 8 nodes. In each setup, every node had one Spark worker. One node was selected in each cluster to be a Spark master. This node was also where the main script was run. I found that Spark has a higher throughput than the single-machine implementation. As the



Setup	FPS	Speedup
sequential	0.0404	1
8 nodes	0.3399	8.42
4 nodes	0.1858	4.60
2 nodes	0.0982	2.43
local[2]	0.0509	1.26
local[1]	0.0376	0.93

Fig. 6. Execution time versus video Length for video of traffic. Table details data for the test on 320 second video.

number of workers increase, I did not get perfect scaling. This is likely due to the fact that there is still some sequential code in the script. The segmenting and concatenating of videos is only done on the driver and is not parallelized. Again, I ran 3 tests for each video length and took the median time.

The program scales well across number of nodes. However, it scales poorly across number of cpus. This indicates that the limiting resources is not compute power.

4.5 Future Work

A few assumptions are made about the input videos. Currently, the script expects videos to have the same dimensions and frame rates. Enabling usage of videos with different dimensions would be rather straightforward. However, enabling stitching of videos with different frame rates would be more complex. Open questions are how to stitch pairs out of sync frames, whether to use the lower or higher input frame rate, and whether or not to interpolate missing frames. Adding these two features are promising areas for future work.

Another area of future work would be finding the resource bottleneck of the program.

Setup	FPS	Speedup
sequential	0.0477	1
8 nodes	0.3911	8.20
4 nodes	0.2173	4.55
2 nodes	0.1161	2.43
local[2]	0.0603	1.26
local[1]	0.04503	0.97

Fig. 7. Execution time versus video Length for video of me juggling. Table details data for the test on 320 second video.

5 CONCLUSION

In this work, I detailed an extension from state-of-the-art image stitching to video stitching. The program decomposes given videos to segments, distributes them by reference using S3, stitches them together using OpenCV and Spark, and finally concatenates segments to create a final stitched video. The image stitching algorithm was tested and demonstrated good speedup in a cluster. There are some minor improvements that could be explored in the future.

REFERENCES

[1] Ahsan Nabi Dar. *Video Analytics in Scala with Akka Actors, FFmpeg, GraphicsMagick, OpenCV and OpenIMAJ*. 2020. URL: https://darnahsan.medium.com/video-analytics-in-scala-with-akka-actors-ffmpeg-graphicsmagick-opencv-and-openimaj-b12735336b27.

[2] Guanyu Gao and Yonggang Wen. "Morph: A Fast and Scalable Cloud Transcoding System". In: *Proceedings of the 24th ACM International Conference on Multimedia*. MM '16. Amsterdam, The Netherlands: Association for Computing Machinery, 2016, pp. 1160–1163. ISBN: 9781450336031. DOI: 10.1145/2964284.2973792. URL: https://doi.org/10.1145/2964284.2973792.

[3] Kevin Horan. *Distributed Video Streaming with Python and Kafka*. 2018. URL: https://medium.com/@kevin.michael.horan/distributed-video-streaming-with-python-and-kafka-551de69fe1dd.

[4] Sajad Sameti, Mea Wang, and Diwakar Krishnamurthy. "Stride: Distributed Video Transcoding in Spark". In: *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. 2018, pp. 1–8. DOI: 10.1109/PCCC.2018.8711214.

[5] Kai Yu et al. "A Large-Scale Distributed Video Parsing and Evaluation Platform". In: vol. 664. Dec. 2016, pp. 37–43. ISBN: 978-981-10-3475-6. DOI: 10.1007/978-981-10-3476-3_5.