Cloud-Accelerated Analysis of Subsea High-Definition Camera Data

Aaron Marburg
Applied Physics Laboratory
University of Washington
Seattle, Washington 98105
Email: amarburg@apl.washington.edu

Timothy J. Crone
Lamont-Doherty Ocean Observatory
Columbia University
Palisades, NY 10964
Email: tjcrone@ldeo.columbia.edu

Freidrich Knuth
Marine and Coastal Sciences
Rutgers University
New Brunswick, New Jersey 08905
Email: knuth@marine.rutgers.edu

***Abstract*—The seafloor high-definition camera (CamHD) installed on the Ocean Observatories Initiative (OOI) Cabled Array (CA) provides real-time video of the Mushroom vent at the ASHES hydrothermal field in the Axial Volcano caldera on the Juan de Fuca spreading zone (Figure 1).**

CamHD performs a pre-programmed 13-minute motion sequence every 3 hours. The video captured during this sequence is stored as a 13GB HD video file in the OOI Cyber-Infrastructure (CI) at Rutgers University. As of July 2017 there are approx. 6700 videos in the CI, all of which are publicly accessible through a conventional HTTP interface. Unfortunately, it is impractical for a researcher (and taxing on the CI bandwidth) to download, store, and process the extent of the video archive for analysis.

We describe two elements of our efforts to accelerate CamHD video analysis: a cloud-hosted application which provides a simplified interface for extracting individual frames from CamHD videos in a time- and bandwidth-efficient manner; and a tool for the automatic isolation and identification of video subsets showing a sequence of known camera positions. Automatic identification of these video segments allows rapid and automatic development of e.g., time lapse videos.

I. Introduction

The NSF-funded Ocean Observatories Initiative (OOI) Cabled Array (CA) provides long-term, persistent bandwidth and power, and instantaneous data exfiltration, for over 120 distinct instruments installed off the Oregon Coast. One such instrument is the high-definition camera, CamHD, which is located at the ASHES hydrothermal vent field within the caldera of Axial Seamount. [1] CamHD is situated less than 2 meters from the hydrothermal vent Mushroom (Figure 1), which affords it a sweeping view of the geology and biology of a unique chemotrophic ecosystem. To minimize disturbance to the local ecosystem, CamHD operates on a regular schedule, turning its lights on every three hours to perform an approx. 13-minute-long pre-programmed sequence of pan, tilt and zoom maneuvers.

The resulting video is transmitted as uncompressed 1080i HD video to shore where it is stored at the OOI Cyber Infrastructure (CI) at Rutgers University in both the lossless Apple ProRes video file format, and in the lossy H.264 video format. These two versions of each 13-minute video requires approximately 13 GB and 850 MB apiece, respectively.

Having both short video sequences and long-time-series images of Mushroom allows for powerful analyses of the spatial and temporal evolution of processes including the growth of individual chimneys through mineral deposition, the growth and spread of bacterial mats, and the behavior and population dynamics of macrofauna. [2], [3] These broad-scale analyses can be meaningfully accelerated through computer-vision-based automated analysis tools. [4]

Upon embarking on this early exploratory analyses, we rapidly encountered two confounding issues with data

Fig. 1. The CamHD high-definition camera and Mushroom hydrothermal vent. *(NSF Ocean Observatories Initiative/ROPOS/University of Washington)*

access. First, the sheer volume of data stored in the CI is significant (and ever-growing). Most video manipulation tools are designed for manipulating local files, a service which is not provided by the OOI CI. Instead, videos must be manually downloaded from Rutgers for analysis, a step which is both time consuming and expensive if storing large numbers of the uncompressed video files.

Second, meta-information about the camera's position throughout each video is scant and unreliable. While the camera motion is nominally consistent between videos, small variations in system timing result in significant jitter in camera position as a function of elapsed time within a video. Further, system malfunctions often result in videos which show only a portion of the camera's motion. In either case, there is no *a priori* method, other than manual video review, for correlating a particular field of view on the vent with elapsed time within a video.

The most general solution to the second problem uses the photometric information within the video frames themselves to estimate camera motion, independent of expected video sequencing or the presence/absence of other motion metadata (motor commands, etc.). Such an algorithm could be scaled to analyze the existing archive of CamHD video as well as newly collected videos to provide a consistent parallel metadata stream describing the video contents.

Moreover, performing repeated, broad-scale video processing over the extent of the CamHD archive is likely to be an essential activity for future CamHD analyses. However, such large scale video analysis of the archive CamHD archive rapidly runs afoul of the first problem. Rather than relying on the necessary computational resource being co-located with the OOI CI, or maintaining a local mirror of the OOI CI local, we consider a third route where content-aware access of the CI video archive is provided by a service which can be run either locally or publicly on the internet. This service provides a consistent, scalable method for accessing individual frames from CamHD movies on the CI.

We describe the three components of this work separately below. First, Section II describes *lazycache*, a web-based microservice for remote access to video meta-information and retrieval of individual frames from ProRes files stored on the OOI CI. Sections III and III-A then describes the algorithms used to identify and extract static segments (time spans where the camera is stationary) from a given video files, and to run those algorithms *en masse* over the CamHD video archive utilizing both local computers and the commercial cloud. Finally, Section IV describes the process of using photometric comparisons to associated static sections with a predefined set of scene labels.

II. Lazycache for Efficient Frame Access

Our initial development focused methods for time-, storage-, and bandwidth-efficient access to single frames from movies in the CI which did not require downloading full video files. The resulting algorithm relies on the underlying structure of the Quicktime file format used to store the uncompressed CamHD files, and the ability of many web servers (including those used at CI) to provide subsets of files on request. This latter technology is the basis for, for example, "restarting" downloads from websites.

The uncompressed CamHD video files are stored at CI in the Apple Quicktime audio-visual container format, while individual frames of video are stored as data within the Quicktime container encoded in ProRes 422, a "lossless" format.[1] The Quicktime format stores data as a tree of data structures or "atoms," each of which starts with a header containing both atom type and total size. The overall structure of a Quicktime file can be quickly ascertained by enumerating the atom tree within the file and extracting a relatively small number of metadata atoms. [5]

Based on this knowledge of the Quicktime file format a set of software libraries were written in the Go language: *lazyfs* implements the standardized Go interface for a random access, read-only file (`io.ReaderAt`) using HTTP requests to read content from a remote web site.[2] With *lazyfs*, a remote, HTTP-served file can be read like a local file, albeit with a significant speed penalty. On top of this, *lazyquicktime* parses the Quicktime container format,[3] operating identically on conventional files and *lazyfs*. *lazyquicktime* uses a parsimonious read strategy to quickly extract the Atom tree for a given file and identify the metadata required to address individual video frames within the files. Finally, the *go-proress-ffmpeg* library provides a Go wrapper around the *ffmpeg* video encoding/decoding library, taking individual ProRes-encoded frames from a Quicktime file and converting it to a Go-language `Image`.[4] That Image can then be directly manipulated in Go or re-encoded using standard Go libraries to e.g., a PNG of JPEG image file.

[1] ProRes does utilize color space subsampling, so it is lossless in the spatio-temporal dimensions, but lossy in color space.

[2] https://github.com/amarburg/go-lazyfs

[3] https://github.com/amarburg/go-lazyquicktime

[4] https://github.com/amarburg/go-prores-ffmpeg

Together, these libraries provide the necessary tools to remotely access a Quicktime file hosted on a web server, retrieve the index within the video file, and extract individual frames in a bandwidth-efficient manner.

An immediate concern of this approach, however, is that the Go language is less popular than e.g., Python or Matlab for scientific computing. To provide a language-agnostic API, the core Quicktime functionality is wrapped in a networked microservice, known as *lazycache*, which exports the frame extraction functionality via a HTTP API. This allows any language to retrieve images using specially formatted HTTP calls, which can be created using language-specific HTTP-access libraries.[5] *lazycache* also provides machine-readable, JSON-format method for browsing the CI data repository and retrieving CamHD movie metadata (length, number of frames, etc.). For efficiency, *lazycache* caches intermediate results, including extracted video frames. In this way, a single *lazycache* instance can efficiently provide video frames for a large number of hosts. At the same time, as it is a conventional webserver, off-the-shelf load-balancing techniques can be used to distribute client requests across a cluster of lazycache instances.

Utilizing HTTP as a transport necessarily induces a performance cost. Relative to a "normal" library (such as might be linked into a Go program or imported into a Python script), *lazycache* incurs three significant costs:

- the time required to encode the images as an image, and subsequently decode on the client,
- the costs of SSL encryption if the HTTPS transport is used. This can be avoided by using unencrypted HTTP.
- the additional time costs for passing data between the client and the server. If the two are co-located on the same machine, the latter is trivial.

Relative to decoding locally-available video files, there is also the significant performance penalty for downloading all data from Rutgers for every operation, although this cost is the same for both the HTTP-based *lazycache* and a more conventional library like *lazyquicktime*. This cost can be balanced against the time- and storage costs of downloading whole files locally before accessing.

(benchmarks here)

A. Deployment

The *lazycache* binary and its dependencies (e.g., ffmpeg, etc.) are packaged as a Docker image, greatly simplifying deployment. For public use, *lazycache* is deployed to the Google App Engine (GAE), a software-as-a-service offering within the Google Cloud Platform product line. A GAE application is described using a service-description language, which in turn references the *lazycache* Docker image description file. GAE uses this file to create a reference Docker image and deploy it across one or more computers in the Google cloud infrastructure, transparently providing the load balancing architecture necessary for those instances to appear as a single endpoint on the internet. Behind the scenes, GAE monitors the health of individual instances and adds additional copies as workload increases. In this way, a GAE-hosted application can respond transparently and automatically to widely varying workloads, and the cost of operation is based solely on the resources (CPU time) consumed. The downside to this architecture is that the system will automatically scale number of instances (and thus running costs) to meet peak loads without intervention or even notification to the app owner.

[5]https://github.com/amarburg/go-lazycache

III. Camera Motion Estimation and its Acceleration

Lazycache provides an efficient method for retrieving single images from a video, an essential first step for archive-wide video analysis. The second task is to automatically identify and label segments of each video based on camera position. This is decomposed into sub-tasks. First, an optical-flow-based method is used to calculate apparent image motion at regular intervals throughout a video files (this section). This is a costly procedure, running at approx. 1/3–1/4 realtime on modern multi-core processor, which would necessitate months of processing per year of archived video. However, this task is trivially parallelizable on a per-video basis and can be accelerated by employing more computers, in this case, large numbers of compute instances purchased in the commercial cloud (Section III-A). Having performed this costly calculation for a given video, statistical techniques can be used to estimate blocks of consistent camera motion (zooming in, panning left, static, etc.), and subsequently use photometric matching to label the static sections based on their field of view (Section IV).

The low contrast, large amount of background motion, and abundance of soft "organic" edges renders feature-based image-registration approaches unreliable, so camera motion is estimated using dense optical flow. Given the highly constrained motion of the camera (only zoom, pan, and tilt), in principal any robust scale-aware correlation-based matching algorithm should provide

good results, however the ready availability of multi-core and GPU-accelerated optical flow algorithms in OpenCV [6] provides a good starting point for development. This optical flow calculation dominates the computational cost of the velocity estimation and is an ideal place for optimization.

The motion estimator algorithm steps through a movie at even intervals (currently every 10 frames). At each step it estimates the local camera velocity by finite differences, estimating the apparent transform between an image a small delta (2 frames) before and after the current frame. These two images are retrieved through a *lazycache* instance, are preprocessed, and heuristics are used to detect and discard known corner cases e.g., if camera lights are off or the frame is out of focus. A dense optical flow algorithm then calculates the apparent motion at each image pixel. This flow image is then downscaled, effectively performing a spatial block-average on the flow estiamte. Finally, the resulting flow field is fit to a similarity transform, which is then stored for that frame. A JSON file containing all estimated similarities for a given movie is stored for later analysis.

A. Scaling of Velocity Estimation

A single CamHD recording contains approximately 25,000 frames (13 minutes at 30 fps). With the the optical flow algorithm processing every tenth frame, a single video requires approximately one hour to process on a high-end Core i7 processor. While this is fast enough to keep up with the rate of video acquisition (one new video every three hours), it insufficient to adequately process the backlog of videos in the CI archive. To address this, the calculation can be scaled out across a cluster of computers, each processing one video at a time. This trivial parallelization offers a linear increase in processing speed with increased computing resources.

This scaling is achieved through a combination of three technologies. First, the optical flow algorithm, along with all necessary dependencies, is also packaged within a Docker runtime container. While introducing an additional layer of complexity, Docker provides highly repeatable, transferrable runtime environment, simplifying software deployment. The interface to the runtime is written as a Python module.

Second, the RQ work queue package is used to coordinate work across computers in the cluster.[6] RQ is a Python-native queueing library which uses the Redis in-memory datastore for coordination. The RQ client uses Python's native introspection capability to serialize Python function calls for execution on worker nodes. Because of this, the RQ worker can be written in a generic manner (it does not need to be customized for the tasks it may be asked to perform), so long as the desired Python libraries are available in the local environment.

Finally, Docker Swarms are used for cluster orchestration. The swarm concept is a recent addition to the Docker ecosystem, and allows multiple computers to be confederated into a centrally managed "swarm" of resources (each running Docker). From a single workstation, containers can be deployed to all members of the cluster.

For the processing detailed here, two independent computer clusters are used, both referencing a single, shared RQ work queue. One consists of a set of three Core i7 desktop computers operating on a shared network. Onto this cluster, multiple copies of both the lazycache server software and the optical flow processor are deployed. On startup, the workers query the Redis server and execute any stored jobs. The load balancing features built into Docker Swarms allows each optical flow processor to distribute its queries to any of the local lazycache instances within the cluster.

The second cluster is constructed from Google Compute Engine (GCE) virtualized PC instances. A single instance acts as a swarm manager (this instance also provides the shared Redis server for both clusters). The worker nodes are defined as a Google Instance Group: a collection of identical compute instances cloned from a reference image. The Google cluster is constructed from eight instances of the `n8-highcpu-1` compute node, a virtual computer with eight virtual cores (vCPUs) and 7.2 GB of memory. These are configured as preemptible instances, an option whereby Google can proactively shut down instances as needed to meet surge demand from other customers. In exchange, preemptible instances are significantly less expensive than equivalent non-preemptible compute instances, at a cost of $0.06 per 8-core instance per hour ($0.48/hour for the entire cluster) versus $0.24/hour for the non-preemptible instances. The Google compute cluster is otherwise configured identically to the desktop cluster.

Both clusters are tasked by inserting tasks into the shared RQ queue. Any idle assets in either cluster will retrieve pending jobs from the queue and start work. In this way the commercial cluster (which has fixed costs when operating) can be started, stopped, and scaled without explicit reconfiguration of the work queue.

Relative benchmarking information for the four com-

[6] http://python-rq.org

TABLE I
COMPARATIVE BENCHMARKS FOR OPTICAL FLOW PROCESSING.

CPU	Freq. (GHz) (GHz)	Threads	Seconds per movie	Net sec. per frame	Actual sec. per frame
Intel Core i7-3770K	3.5	8	5140.33	0.21462	16.613
Intel Core i7-5820K	3.3	12	3370.24	0.14181	16.294
Intel Core i7-6700K	4.0	8	3086.79	0.12857	9.976
Xeon "Sandy Bridge" vCPU	2.6	8	6220.31	0.26655	20.469

puter types (3 generations of Core i7 desktops and the Google compute instances) are given in Table I. Within the table, *CPU* gives the CPU model, with Google virtual CPUs denoted by the relative generation of Xeon processor and clock speed; *Freq.* gives the base CPU clock speed, while *Threads* gives the number of simultaneous threads for the CPU (2 per core for CPUs with Hyper-Threading, and 1 per vCPU on cloud instances). *Seconds per movie* gives the mean wall clock time to process a single movie; *Net seconds per frame* gives the total throughput of the optical flow processor (seconds per movie / frames per movie), while *Actual seconds per frame* gives the wall clock time spent processing each frame. Due to parallelism within the software, each worker processes multiple frames at once.

As noted above, each worker in a cluster runs two instances of the optical flow algorithm increasing the odds that the CPU is fully loaded as each worker transitions between multi- and single-threaded phases. As such, the performance benchmarks do not give the minimum processing time for each processor. The CPU performance times are roughly inline with the synthetic CPU benchmarks for the relative processors, however it is trivial (and inexpensive) to procure large numbers of cloud compute instances. At \$0.06 per instance per hour, each movie require approximately \$0.11 to process, and the total throughput of the cloud compute resources is limited only by the resource limits imposed by Google.

IV. STATIC SECTION ANALYSIS

The velocity estimates produced by the optical-flow technique are used to estimate and label regions of consistent camera motion within each video file. In contrast to the velocity estimation, this algorithm is time efficient and can be run serially on a single computer without explicit parallelization to a cluster.

The algorithm relies on classification of contiguous time regions within each video. After smoothing the estimated X- (pan), Y- (tilt) and scale (zoom) transforms, a hysteresis-based classifier is used to find time segments where the camera appears to be at rest. These "static" segments are labelled by their starting and ending frame numbers. The periods between each static section is then examined and labelled as "zoom in", "zoom out", "pan left", etc. based on its average motion.

Having isolated the frame spans within each movie where the camera is static, the final step is to label or tag each segment corresponding to a known camera position or field of view. As the camera trajectory is pre-programmed, the number and ordering of these camera positions are known *a priori*. In total, 23 distinct static camera positions are present in the current CamHD sampling pattern, at nine distinct camera positions and three zoom levels (the camera is not zoomed to every level at every position).

In practice, minor variations in camera motion, as well as ambiguities in isolating static sections lead to a high degree of variation in the estimated motion sequence. For example, small variations in timing between camera motions can result in some static sections being completely elided. When the camera is at maximum zoom, movement patterns in the resident vent fauna or Schlieren from local venting can be mis-classified as camera motion.

For robustness, classification starts with a photometric classifier. For a small set of "ground truth" videos (approx. 2 per month, $< 1\%$ of all videos), the static regions are isolated automatically, then labelled by hand.

New videos are then labelled through comparison to these hand-labelled videos, with matching currently performed in a brute force manner. After partitioning the new movie into motion segments, a set of $N = 3$ exemplar frames are drawn from each static section. A DFT-based correlation algorithm is used to estimate the correlation minima between each image and M sample frames drawn from each region in the ground truth set. Each correlation results in an estimated translation and an RMS pixel difference at that shift. Results which indicate a shift of greater than 10% of image dimension in either direction are immediately discarded as non-matches. For the remaining $M \times N$ correlations, the

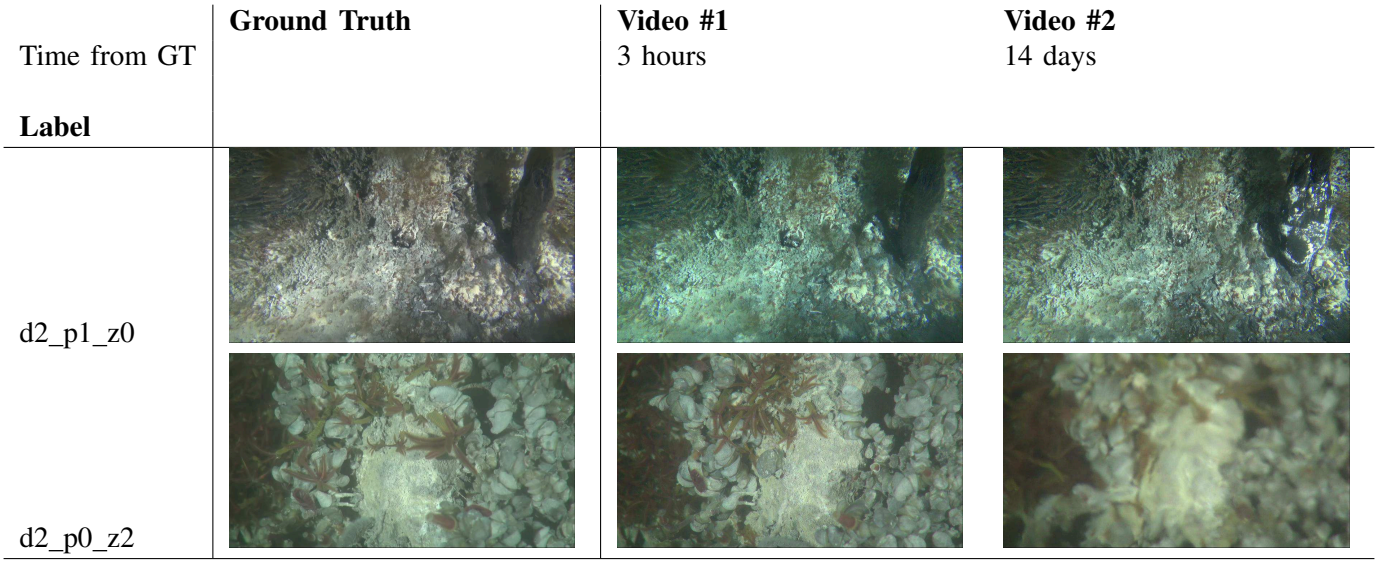

Fig. 2. Sample static regions identified by comparison to ground truth imagery. Video #1 is the next video in sequence, captured 3 hours later. Video #2 is from two weeks later. Scene *d2_p1_z0* in both videos, and *d2_p0_z2* for video #1 were labelled by photometric matching to the ground truth image. For the *d2_p0_z2* segment from video #2, the local currents have shifted such that hot vent fluids are now intruding between the camera and the vent. In this case, the photometric matching fails, however the label is correctly inferred by comparing the labels of consecutive static segments to the reference sequence.

highest and lowest RMS errors are discard and the mean is used as an aggregate matching score between the particular static section in the movie being examined and the associated region.

These scores are then sorted, and the match is considered good if the ratio between the highest and second-highest RMS is less than a preset ratio. If this heuristic is not achieved, the segment remains unlabelled.

After attempting photometric matching on all static segments in a movie, two post-processing steps are used to resolve any remaining unidentified frames. First, each unlabelled segment is compared photometrically to its nearest neighbors which *have* been successfully labelled (both before and after). If this correlation matches with a very low RMS, the unlabelled segment assumes the label of the neighbor. Finally, *a priori* region sequence is considered. If an unlabelled region is preceded and followed by labelled regions, the sequence is compared to a reference sequence of regions.

A sample of the resulting processing is shown in Figure 2.

V. Conclusion

All software and resulting metadata from this project are available publicly through repositories on the Github software sharing site, and are described in the project blog.[7]. Automatically-generated metadata (frame velocities and labelled regions) are stored in JSON files in a Github repository, including documentation on the file format for development of new analyses.[8]. The authors welcome inquiries about further CamHD processing and use of the video meta-information.

References

[1] J. R. Delaney, D. S. Kelley, A. Marburg, M. Stoermer, H. Hadaway, K. Juniper, and F. Knuth, "Axial seamount - wired and restless: A cabled submarine network enables real-time, tracking of a mid-ocean ridge eruption and live video of an active hydrothermal system juan de fuca ridge, ne pacific," in *OCEANS 2016 MTS/IEEE Monterey*, Sept 2016, pp. 1–8.

[2] F. Knuth, M. Vardaro, L. Belabassi, M. Smith, L. Garzio, M. Crowley, J. Kerfoot, and O. Kawka, "The ocean observatories initiative: Unprecedented access to real-time data streaming from the cabled array through ooi cyberinfrastructure." in *Ocean Sciences 2016*, 2016.

[3] F. Knuth, L. Belabassi, L. Garzio, M. Smith, M. Vardaro, and A. Marburg, "Automated qa/qc and time series analysis on ooi high-definition video data," in *OCEANS 2016 MTS/IEEE Monterey*, Sept 2016, pp. 1–4.

[4] T. Crone, A. Marburg, and F. Knuth, "Using the ooi cabled array hd camera to explore geophysical and oceanographic problems at axial seamount," in *presented at the 2016 Fall Meeting, AGU, San Francisco, CA*, 2016.

[5] *QuickTime File Format Specification*. Apple Computer, Inc., 2016.

[6] Itseez, "Open source computer vision library," https://github.com/itseez/opencv, 2015.

[7] https://camhd-analysis.github.io/public-www/

[8] https://github.com/CamHD-Analysis/CamHD_motion_metadata