Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads

Sadjad Fouladi, Riad S. Wahby, and Brennan Shacklett, *Stanford University;*
Karthikeyan Vasuki Balasubramaniam, *University of California, San Diego;*
William Zeng, *Stanford University;* Rahul Bhalerao, *University of California, San Diego;*
Anirudh Sivaraman, *Massachusetts Institute of Technology;*
George Porter, *University of California, San Diego;* Keith Winstein, *Stanford University*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Encoding, Fast and Slow:
Low-Latency Video Processing Using Thousands of Tiny Threads

Sadjad Fouladi S, Riad S. Wahby S, Brennan Shacklett S,
Karthikeyan Vasuki Balasubramaniam Ψ, William Zeng S, Rahul Bhalerao Ψ,
Anirudh Sivaraman MIT, George Porter Ψ, Keith Winstein S

Stanford University S, *University of California San Diego* Ψ, *Massachusetts Institute of Technology* MIT

Abstract

We describe ExCamera, a system that can edit, transform, and encode a video, including 4K and VR material, with low latency. The system makes two major contributions.

First, we designed a framework to run general-purpose parallel computations on a commercial "cloud function" service. The system starts up thousands of threads in seconds and manages inter-thread communication.

Second, we implemented a video encoder intended for fine-grained parallelism, using a functional-programming style that allows computation to be split into thousands of tiny tasks without harming compression efficiency. Our design reflects a key insight: the work of video encoding can be divided into fast and slow parts, with the "slow" work done in parallel, and only "fast" work done serially.

1 Introduction

The pace of data analysis and processing has advanced rapidly, enabling new applications over large data sets. Providers use data-parallel frameworks such as MapReduce [8], Hadoop [12], and Spark [32] to analyze a variety of data streams: click logs, user ratings, medical records, sensor histories, error logs, and financial transactions.

Yet video, the largest source of data transiting the Internet [6], has proved one of the most vexing to analyze and manipulate. Users increasingly seek to apply complex computational pipelines to video content. Examples include video editing, scene understanding, object recognition and classification, and compositing. Today, these jobs often take hours, even for a short movie.

There are several reasons that interactive video-processing applications have yet to arrive. First, video jobs take a lot of CPU. In formats like 4K or virtual reality, an hour of video will typically take more than 30 CPU-hours to process. A user who desires results in a few seconds would need to invoke thousands of threads of execution—even assuming the job can be parallelized into thousands of pieces.

Second, existing video encoders do not permit fine-grained parallelism. Video is generally stored in compressed format, but unlike the per-record compression used by data-parallel frameworks [2], video compression relies on temporal correlations among nearby frames. Splitting the video across independent threads prevents exploiting correlations that cross the split, harming compression efficiency. As a result, video-processing systems generally use only coarse-grained parallelism—e.g., one thread per video, or per multi-second chunk of a video—frustrating efforts to process any particular video quickly.

In this paper, we describe ExCamera, a massively parallel, cloud-based video-processing framework that we envision as the backend for interactive video-processing applications. A user might sit in front of a video-editor interface and launch a query: "render this 4K movie, edited as follows, and with scenes containing this actor given a certain color shade." As with any interactive cloud editor (e.g. Google Docs), the goal is to execute the task quickly and make the result immediately accessible online.

ExCamera makes two contributions:

1. A framework that orchestrates general-purpose parallel computations across a "cloud function" service (§ 2). The advent of such services—AWS Lambda, Google Cloud Functions, IBM OpenWhisk, and Azure Functions—permits new models of interactive cloud computing. A cloud function starts in milliseconds and bills usage in fraction-of-a-second increments, unlike a traditional virtual machine, which takes minutes to start and has a minimum billing time of 10 minutes (GCE) or an hour (EC2). Though these services were designed for Web microservices and event handlers, AWS Lambda has additional features that make it more broadly useful: workers can run arbitrary Linux executables and make network connections. ExCamera invokes thousands of C++-implemented Lambda functions in seconds.

2. A video encoder intended for massive fine-grained parallelism (§ 3), built to avoid the traditional compromise of parallel video encoding: the inability to benefit from visual similarities that span parts of the video handled by different threads. ExCamera encodes tiny chunks of the video in independent threads (doing most of the "slow" work in parallel), then stitches those chunks together in a "fast" serial pass, using an encoder written in explicit state-passing style with named intermediate states (§ 4).

Summary of results

We characterized ExCamera's performance and found that it can summon 3,600 cores (representing about 9 teraFLOPS) within 2.5 seconds of job startup. For video encoding, we tuned ExCamera and compared it with existing systems using a 15-minute animated movie in 4K resolution [20], encoding into the VP8 compressed-video format [29]. ExCamera achieved comparable compression to existing systems, at the same quality level relative to the original uncompressed video, and was many times faster. The evaluation is fully described in Section 5.

System	**Bitrate at 20 dB SSIM** (lower is better)	**Encode time** (lower is better)
ExCamera[6,16][1]	27.4 Mbps	2.6 minutes
ExCamera[6,1][2]	43.1 Mbps	0.5 minutes
`vpxenc` multi-threaded	27.2 Mbps	149 minutes
`vpxenc` single-threaded	22.0 Mbps	453 minutes
YouTube H.264[3]	*n/a*	36.5 minutes
YouTube VP9	*n/a*	417 minutes

Results were similar on a 12-minute live-action 4K video [23]:

System	**Bitrate at 16 dB SSIM** (lower is better)	**Encode time** (lower is better)
ExCamera[6,16]	39.6 Mbps	2.2 minutes
ExCamera[6,1]	66.0 Mbps	0.5 minutes
`vpxenc` multi-threaded	36.6 Mbps	131 minutes
`vpxenc` single-threaded	29.0 Mbps	501 minutes

This paper proceeds as follows. In Section 2, we introduce our framework to execute general-purpose parallel computations, with inter-thread communication, on AWS Lambda. We discuss the problem of fine-grained parallel video encoding in Section 3, and describe ExCamera's approach in Section 4. We then evaluate ExCamera's performance (§ 5), discuss limitations (§ 6), and compare with related work (§ 7).

ExCamera is free software. The source code and evaluation data are available at https://ex.camera.

2 Thousands of tiny threads in the cloud

While cloud services like Amazon EC2 and Google Compute Engine allow users to provision a cluster of powerful machines, doing so is costly: VMs take minutes to start and usage is billed with substantial minimums: an hour for EC2, or 10 minutes for GCE. This means that a cluster of VMs is not effective for running occasional short-lived, massively parallel, interactive jobs.

Recently, cloud providers have begun offering *microservice frameworks* that allow systems builders to replace long-lived servers processing many requests with short-lived workers that are dispatched as requests arrive. As an example, a website might generate thumbnail images using a "cloud function," spawning a short-lived worker each time a customer uploads a photograph.

Because workers begin their task quickly upon spawning and usage is billed at a fine grain, these frameworks show promise as an alternative to a cluster of VMs for short-lived interactive jobs. On the other hand, microservice frameworks are typically built to execute asynchronous lightweight tasks. In contrast, the jobs we target use thousands of simultaneous threads that execute heavyweight computations and communicate with one another.

To address this mismatch we built mu, a library for designing and deploying massively parallel computations on AWS Lambda. We chose AWS Lambda for several reasons: (1) workers spawn quickly, (2) billing is in subsecond increments, (3) a user can run many workers simultaneously, and (4) workers can run arbitrary executables. Other services [4, 10, 19] offer the first three, but to our knowledge none offers the fourth. We therefore restrict our discussion to AWS Lambda. (In the future it may be possible to extend mu to other frameworks.)

In the next sections, we briefly describe AWS Lambda (§ 2.1); discuss the mismatch between Lambda and our requirements, and the architecture that mu uses to bridge this gap (§ 2.2); detail mu's software interface and implementation (§ 2.3); and present microbenchmarks (§ 2.4). We present an end-to-end evaluation of mu applied to massively parallel video encoding in Section 5 and discuss mu's limitations in Section 6.

2.1 AWS Lambda overview

AWS Lambda is a microservice framework designed to execute user-supplied *Lambda functions* in response to asynchronous *events*, e.g., message arrivals, file uploads, or API calls made via HTTP requests. Upon receiving an event, AWS Lambda spawns a *worker*, which executes in a Linux container with configurable resources up to two, 2.8 GHz virtual CPUs, 1,536 MiB RAM, and about 500 MB of disk space. AWS Lambda provisions additional containers as necessary in response to demand.

To create a Lambda function, a user generates a package containing code written in a high-level language (currently Python, Java, Javascript, or C#) and *installs* the package using an HTTP API. Installed Lambda functions are *invoked* by AWS Lambda in response to any of a number of events specified at installation.

[1] "ExCamera[6,16]" refers to encoding chunks of six frames independently in parallel, then stitching them together in strings of 16 chunks.

[2] Encodes independent chunks of six frames each, without stitching.

[3] Because YouTube doesn't encode into the VP8 format at this resolution or expose an adjustable quality, we report only the total time between the end of upload and the video's availability in each format.

At the time of writing, AWS Lambda workers using maximum resources cost 250 μ¢ per 100 ms, or $0.09 per hour. This is slightly less than the closest AWS EC2 instance, c3.large, which has the same CPU configuration, about twice as much RAM, and considerably more storage. While Lambda workers are billed in 100 ms increments, however, EC2 instances are billed hourly; thus, workers are much less expensive for massively parallel computations that are short and infrequent.

2.2 Supercomputing as a (μ)service

With mu, we use AWS Lambda in a different way than intended. Instead of invoking workers in response to a single event, we invoke them in bulk, thousands at a time. The mismatch between our use case and Lambda's design caused several challenges:

1. Lambda functions must be installed before being invoked, and the time to install a function is much longer than the time to invoke it.
2. The timing of worker invocations is unpredictable: workers executing warm (recently invoked) functions spawn more quickly than those running cold functions (§ 2.4). In addition, workers may spawn out of order.
3. Amazon imposes a limit on the number of workers a user may execute concurrently.
4. Workers are behind a Network Address Translator (NAT). They can initiate connections, but cannot accept them, and so they must use NAT-traversal techniques to communicate with one another.
5. Workers are limited to five minutes' execution time.

To illustrate the effect of these limitations, consider a strawman in which a user statically partitions a computation among a number of threads, each running a different computation, then creates and uploads a Lambda function corresponding to each thread's work. First, this requires the user to upload many different Lambda functions, which is slow (limitation 1). Second, workers will spawn slowly, because each one is cold (limitation 2). Third, if there are execution dependencies among the workers, the computation may deadlock if workers are spawned in a pathological order and the number of needed workers exceeds the concurrency limit (limitations 2 and 3). Finally, workers cannot communicate and must synchronize indirectly, e.g., using the AWS S3 block store (limitation 4).

To address issues 1–4, we make three high-level decisions. First, mu uses a long-lived coordinator that provides command and control for a fleet of ephemeral workers that contain no thread-specific logic. Instead, the coordinator steps workers through their tasks by issuing RPC requests and processing the responses. For computations in which workers consume outputs from other workers, mu's coordinator uses dependency-aware scheduling: the coordinator first assigns tasks whose outputs are consumed, then assigns tasks that consume those outputs. This helps to avoid deadlock and reduce end-to-end completion time.

Second, all workers in mu use the same generic Lambda function. This Lambda function is capable of executing the work of any thread in the computation. As described above, at run time the coordinator steps it through its assigned task. This means the user only installs one Lambda function and thus that workers spawn quickly because the function remains warm. Third, we use a *rendezvous server* that helps each worker communicate with other workers. The end result is a highly parallel, distributed, low-latency computational substrate.

This design does not completely sidestep the above limitations. Amazon was willing to increase our concurrent worker limit only to 1,200 per AWS region (the default is 100). We still need to partition our largest computations among several regions. In addition, the five-minute worker timeout seems unavoidable. For ExCamera this limitation does not cause serious problems since the system aims for smaller end-to-end latencies, but it prevented us from benchmarking certain alternative systems (§ 5).

2.3 mu software framework

Workers. mu workers are short-lived Lambda function invocations. When a worker is invoked, it immediately establishes a connection to the coordinator, which thereafter controls the worker via a simple RPC interface. As examples, the coordinator can instruct the worker to retrieve from or upload to AWS S3; establish connections to other workers via a rendezvous server; send data to workers over such connections; or run an executable. For long-running tasks like transferring data and running executables, the coordinator can instruct the worker to run its task in the background.

The user can include additional executables in the mu worker Lambda function package (§ 2.1).[4] The worker executes these in response to RPCs from the coordinator.

Coordinator. The coordinator is a long-lived server (e.g., an EC2 VM) that launches jobs and controls their execution. To launch jobs, the coordinator generates events, one per worker, using AWS Lambda API calls via HTTP (§ 2.1). These HTTP requests are a bottleneck when launching thousands of workers, so the coordinator uses many parallel TCP connections to the HTTP server (one per worker) and submits all events in parallel.

The coordinator contains all of the logic associated with

[4] We statically link these executables to make sure they will run in the worker's environment.

a given computation in the form of per-worker finite-state-machine descriptions. For each worker, the coordinator maintains an open TLS connection, the worker's current state, and its state- transition logic. When the coordinator receives a message from a worker, it applies the state-transition logic to that message, producing a new state and sending the next RPC request to the worker.

Rendezvous. Like the coordinator, the rendezvous server is long lived. mu's rendezvous is a simple relay server that stores messages from workers and forwards them to their destination. This means that the rendezvous server's connection to the workers can be a bottleneck, and thus fast network connectivity between workers and the rendezvous is required. Future work is implementing a hole-punching NAT-traversal strategy to address this limitation.

Developing computations with mu. To design a computation, a user specifies each worker's sequence of RPC requests and responses in the form of a finite-state machine (FSM), which the coordinator executes. mu provides a toolbox of reusable FSM components as a Python library. These components hide the details of the coordinator's communication with workers, allowing a user to describe workers' tasks at a higher level of abstraction.

The simplest of mu's state-machine components represents a single exchange of messages: the coordinator waits for a message from the worker, sends an RPC request, and transitions unconditionally to a new state. For long, straight-line exchanges, mu can automatically pipeline, i.e., send several RPCs at once and then await all of the responses. This is useful if the coordinator's link to workers has high latency.

To encode control-flow constructs like if-then-else and looping, the toolbox includes *state combinator* components. These components do not wait for messages or send RPCs. Instead, they implement logic that encodes conditional FSM transitions. As an example, an if-then-else combinator might check whether a previous RPC succeeded, only uploading a result to S3 upon success. In addition to control flow, mu's state combinators implement cross-worker synchronization and encode parallel execution of multiple RPCs.

Implementation details. mu comprises about 2,700 lines of Python and 2,200 lines of C++. Workers mutually authenticate with the coordinator and rendezvous servers using TLS certificates.

2.4 mu microbenchmarks

In this section, we run microbenchmarks on AWS Lambda using mu to answer two questions:

1. How does cold vs. warm start affect end-to-end latency for massively parallel jobs using mu?

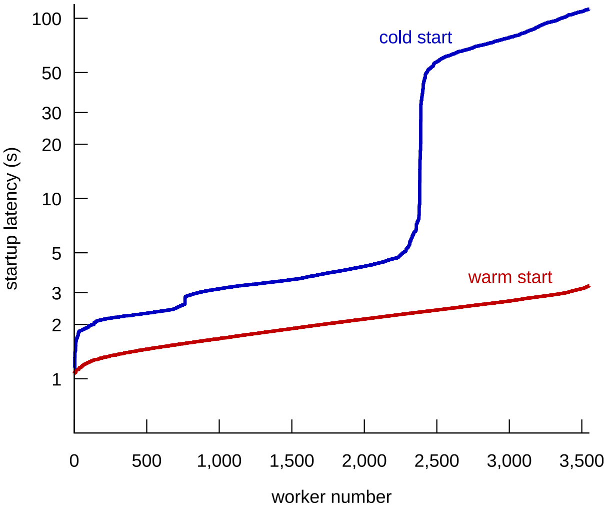


Figure 1: Characterization of cold vs. warm startup time. On a warm start, mu established 3,600 TLS connections to AWS Lambda, spawned 3,600 workers, and received inbound network connections from each worker, in under three seconds (§ 2.4). On a cold start, the delay was as long as two minutes.

2. How much computational power can mu provide, and how quickly, for an embarrassingly parallel job?

In sum, we find that cold starts can introduce minutes of delay, and that a mu job with thousands of threads can run at about nine TFLOPS and starts in just a few seconds when warm. In Section 5, we evaluate mu using end-to-end benchmarks involving worker-to-worker communication and synchronization.

Setup and method. In these microbenchmarks, we invoke 3,600 workers across eight AWS Lambda regions, 450 workers per region. We choose this number because it is similar to the largest computations in our end-to-end evaluation of ExCamera (§ 5). Each worker runs with the maximum allowable resources (§ 2.1). We run a mu job on every worker that executes LINPACK [16], a standard floating-point benchmark that factorizes a random matrix and uses the result to solve a linear equation. Our experiments use double-precision floating point and a $5,000 \times 5,000$ matrix.

Cold and warm start. Figure 1 shows typical worker startup latency for cold and warm Lambda functions. In both cases, there is about a 1-second delay while the coordinator establishes many parallel TLS connections to the AWS Lambda API endpoint. For cold behavior, we install a new mu worker Lambda function and record the timing on its first execution. Only a few workers start before 2 seconds, and about 2,300 have started by 5 seconds; this delay seems to be the result of Lambda's provisioning additional workers. The last 1,300 cold workers are delayed by more than one minute as a result of rate limiting logic,

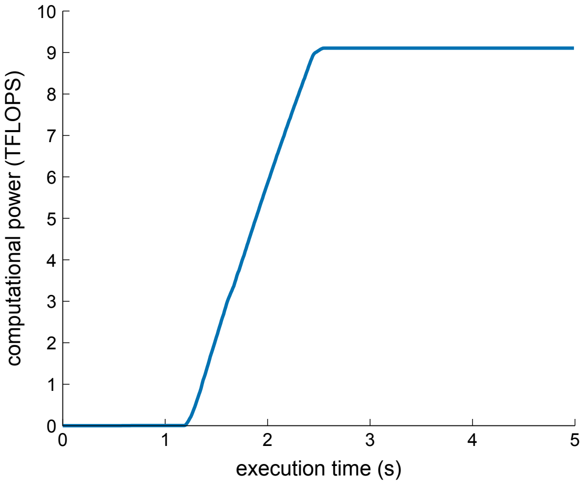


Figure 2: Characterization of floating-point computational power for an embarrassingly parallel job (after PyWren [14]). mu starts 3,600 workers in less than 3 seconds. Each worker executes about 2.5 MFLOPS, for a total of 9 TFLOPS.

which we confirm via the AWS Lambda web interface.

To produce warm behavior, we invoke the same mu computation three times in succession. The result is that AWS Lambda provisions many workers and can start them all very quickly: all 3,600 are started within 3 seconds.

Raw compute power. Figure 2 shows floating-point computational power versus time for a typical (warm) execution of the LINPACK benchmark. All workers start within 3 seconds, and each takes about 3 minutes to run this benchmark. On average, workers execute about 2500 MFLOPS each, for a total of about 9 TFLOPS.

3 Fine-grained parallel video encoding

The previous section described ExCamera's execution engine for running general-purpose parallel jobs with inter-thread communication on a "cloud function" service. We now describe the initial application of this framework: low-latency video encoding to support an interactive application. Parallelizing this workload into tiny tasks was a major focus of our work, because video encoding had not been considered a finely parallelizable operation without compromising compression efficiency.

Compressed video accounts for about 70% of consumer Internet traffic [6]. In digital-video systems, an encoder consumes raw frames and turns them into a compressed bitstream. A corresponding decoder converts the bitstream back to frames that approximate the original input and can be displayed to the viewer.

Video-compression techniques typically exploit the correlations between nearby frames. For example, if a background does not change between two frames, those sections of the image don't need to be repeated in the bitstream. If part of the image is in motion, the encoder can still take advantage of the correlation by using "motion-compensated prediction," which moves around pieces of earlier frames to produce an approximation of the new frame. An encoder spends most of its CPU time searching for such correlations to reduce the size of the output.

These techniques cause compressed frames to depend on one another in a way that makes it impossible to start decoding in midstream. In practice, however, applications often want to start in midstream. For example, a television viewer may change the channel, or a YouTube or Netflix client may want to switch to a higher-quality stream.

To allow this, video encoders insert *Stream Access Points* in the compressed bitstream—one at the beginning, and additional ones in the middle. A Stream Access Point resets the stream and serves as a dependency barrier: the decoder can begin decoding the bitstream at a Stream Access Point, without needing access to earlier portions of the bitstream. The same concept[5] goes by different names in different video formats: a "closed Group of Pictures" (MPEG-2 [17]), an "Instantaneous Display Refresh" (H.264 [11]), or a "key frame" (VP8/VP9 [26, 29]). We use the term "key frame" in this paper.[6]

Chunks of compressed video separated by a key frame are essentially independent. This can be useful in parallel video encoding. Each thread can encode a different range of the video's frames, outputting its own compressed bitstream independently, without making reference to compressed frames in another thread's portion. The resulting bitstreams can simply be concatenated.

But key frames incur a significant cost. As an example, a raw frame of 4K video is about 11 megabytes. After compression in the VP8 format at 15 Mbps, a key frame is typically about one megabyte long. If the video content stays on the same scene without a cut, subsequent frames—those allowed to depend on earlier frames—will be about *10–30 kilobytes*. Spurious insertion of key frames significantly increases the compressed bitrate.

This phenomenon makes low-latency parallel video encoding difficult. To achieve good compression, key frames should only be inserted rarely. For example, systems like YouTube use an interval of four or five seconds. If frames within each chunk are processed serially, then multi-second chunks make it challenging to achieve low-latency pipelines, especially when current encoders process 4K videos at roughly 1/30th (VP8) or 1/100th (VP9) of real time on an x86-64 CPU core. In the next section, we describe how ExCamera addresses this difficulty.

[5] Formally, a Stream Access Point of type 1 or type 2 [18].

[6] An "I-frame" is not the same concept: in MPEG standards, I-pictures depend on compression state left behind by previous elements in the bitstream, and do not prevent subsequent frames from depending on the image content of earlier frames.

4 ExCamera's video encoder

ExCamera is designed to achieve low latency with fine-grained parallelism. Our encoder reflects one key insight: that the work of video encoding can be divided into fast and slow parts, with the "slow" work done in parallel across thousands of tiny threads, and only "fast" work done serially. The slower the per-frame processing—e.g., for pipelines that use 4K or 8K frames or sophisticated video compression, or that run a computer-vision analysis on each frame—the more favorable the case for fine-grained parallelism and for our approach.

ExCamera first runs thousands of parallel `vpxenc` processes (Google's optimized encoder), each charged with encoding just ¼ second, or six frames, of video.[7] Each `vpxenc` output bitstream begins with a key frame. Each thread then re-encodes the first frame, using ExCamera's own video encoder, to remove the key frame and take advantage of correlations with the previous thread's portion of the video. Finally, in a serial step, the system "rebases" each frame on top of the previous one, stitching together chains of 16 chunks. The result is a video that only incurs the penalty of a key frame every four seconds, similar to the interval used by YouTube and other systems.

We refer to this algorithm as "ExCamera[6,16]," meaning it stitches together chains of 16 six-frame chunks. In our evaluation, we profile several settings of these parameters, including ones that simply use naive parallel encoding without stitching (e.g., "ExCamera[6,1]"). ExCamera has been implemented with the VP8 format, but we believe the "rebasing" technique is general enough to work with any recent compressed-video format.

Overview of approach. ExCamera encodes videos using a parallel-serial algorithm with three major phases:

1. *(Parallel)* Each thread runs a production video encoder (`vpxenc`) to encode six compressed frames, starting with a large key frame.

2. *(Parallel)* Each thread runs our own video encoder to replace the initial key frame with one that takes advantage of the similarities with earlier frames.

3. *(Serial)* Each thread "rebases" its chunk of the video onto the prior thread's output, so that the chunks can be played in sequence by an unaltered VP8 decoder without requiring a key frame in between.

In this section, we describe the video-processing primitives we built to enable this algorithm (§ 4.1–4.3) and then specify the algorithm in more detail (§ 4.4).

4.1 Video in explicit state-passing style

Traditional video encoders and decoders maintain a substantial amount of opaque internal state. The decoder starts decoding a sequence of compressed frames at the first key frame. This resets the decoder's internal state. From there, the decoder produces a sequence of raw images as output. The decoder's state evolves with the stream, saving internal variables and copies of earlier decoded images so that new frames can reference them to exploit their correlations. There is generally no way to import or export that state to resume decoding in midstream.

Encoders also maintain internal state, also with no interface to import or export it. A traditional encoder begins its output with a key frame that initializes the decoder.

For ExCamera, we needed a way for independent encoding threads to produce a small amount of compressed output each, *without* beginning each piece with a key frame. The process relies on a "slow" but parallel phase—compressing each video frame by searching for correlations with recent frames—and a "fast" serial portion, where frames are "rebased" to be playable by a decoder that just finished decoding the previous thread's output.[8]

To allow ExCamera to reason about the evolution of the decoder as it processes the compressed bitstream, we formulated and implemented a VP8 encoder and decoder in explicit state-passing style. A VP8 decoder's state consists of the *probability model*—tables that track which values are more likely to be found in the video and therefore consume fewer bits of output—and three *reference images*, raw images that contain the decoded output of previous compressed frames:

$$\texttt{state} := (\texttt{prob\_model},\ \texttt{references[3]})$$

Decoding takes a state and a compressed frame, and produces a new state and a raw image for display:

$$\textbf{decode}(\texttt{state},\ \texttt{compressed\_frame}) \rightarrow (\texttt{state}',\ \texttt{image})$$

ExCamera's decode operator is a deterministic pure function; it does not have any implicit state or side effects. The implementation is about 7,100 lines of C++11, plus optimized assembly routines borrowed from Google's `libvpx` where possible. Our implementation passes the VP8 conformance tests.

4.2 What's in a frame?

A compressed frame is a bitstring, representing either a key frame or an "interframe." A key frame resets and

[7] Major motion pictures are generally shown at 24 frames per second.

[8] We use the term by analogy to Git, where commits may be written independently in parallel, then "rebased" to achieve a linear history [5]. Each commit is rewritten so as to be applicable to a source-code repository in a state different from when the commit was originally created.

initializes the decoder's state:

$$\textbf{decode}(\textit{any state}, \texttt{key\_frame}) \rightarrow (\texttt{state}', \texttt{image})$$

An interframe does depend on the decoder's state, because it re-uses portions of the three reference images. An interframe contains a few important parts:

$$\texttt{interframe} := (\texttt{prediction\_modes}, \texttt{motion\_vectors}, \texttt{residue})$$

The goal of an interframe is to be as short as possible, by exploiting correlations between the intended output image and the contents of the three reference slots in the decoder's state. It does that by telling the decoder how to assemble a "prediction" for the output image. Every 4x4-pixel square is tagged with a *prediction mode* describing where to find visually similar material: either one of the three references, or elsewhere in the current frame.

For prediction modes that point to one of the three references, the 4x4-pixel square is also tagged with a *motion vector*: a two-dimensional vector, at quarter-pixel precision, that points to a square of the reference image that will serve as the prediction for this square.

When decoding an interframe, the decoder uses the prediction modes and motion vectors to assemble the *motion-compensated prediction image*. It then adds the *residue*—an overlay image that corrects the prediction—applies a smoothing filter, and the result is the output image. The interframe also tells the decoder how to update the probability model, and which reference slots in the state object should be replaced with the new output image.

The goal is for the prediction to be as accurate as possible so the residue can be coded in as few bits as possible. The encoder spends most of its time searching for the best combination of prediction modes and motion vectors.

4.3 Encoding and rebasing

ExCamera's main insight is that the hard part of encoding an interframe—finding the best prediction modes and motion vectors—can be run in parallel, while the remaining work of calculating the residue is fast enough to be serialized. This enables most of the work to be parallelized into tiny chunks, without requiring a large key frame at the start of each thread's output.

To do this, we had to implement two more VP8 operations, in about 3,400 lines of C++11. The first is a video encoder that can start from an arbitrary decoder state:

$$\textbf{encode-given-state}(\texttt{state}, \texttt{image}, \textit{quality}) \rightarrow \texttt{interframe}$$

This routine takes a state (including the three reference images), and a raw input image, and searches for the best combination of motion vectors and prediction modes so that the resulting interframe approximates the original image to a given fidelity.

Our encoder is not as good as Google's `vpxenc`, which uses optimized vector assembly to search for the best motion vectors. Compared with `vpxenc`, "encode-given-state" is much slower and produces larger output for the same quality (meaning, similarity of the decoder's output image to the input image). However, it can encode given an externally supplied state, which allows the key frame at the start of each chunk to be replaced with an interframe that depends on the previous thread's portion of the video.

The second operation is a "rebase" that transforms interframes so they can be applied to a different state than the one they were originally encoded for:

$$\textbf{rebase}(\texttt{state}, \texttt{image}, \texttt{interframe}) \rightarrow \texttt{interframe}'$$

While encode-given-state creates a compressed frame *de novo*, a rebase is a transformation on compressed frames, taking advantage of calculations already done. Rebasing involves three steps:

1. **Don't redo the slow part.** Rebasing adopts verbatim the prediction modes and motion vectors from the original interframe.

2. **Apply motion prediction to new state.** Rebasing applies those prediction modes and motion vectors to the *new* `state` object, producing a new motion-compensated prediction image.

3. **Recalculate residue given original target image.** Rebasing subtracts the motion-compensated prediction image from the *original* input to the encoder: the raw target image that was taken "ex camera," i.e., directly from the camera. This "fast" subtraction produces a new residue, which is encoded into the output $\texttt{interframe}'$.

4.4 The parallel-serial algorithm

We now describe an ExCamera[N,x] encoder pipeline. The algorithm works separately on batches of x threads, each handling N frames, so the entire batch accounts for $N \cdot x$ frames and will contain only one key frame. For example, the ExCamera[6,16] algorithm uses batches of 96 frames each (meaning there will be one key frame every four seconds). Batches are run in parallel. Within each batch, the x threads proceed as follows:

1. *(Parallel)* Each thread downloads an N-image chunk of raw video. At the resolution of a 4K widescreen movie, each image is 11 megabytes.

2. *(Parallel)* Each thread runs Google's `vpxenc` VP8 encoder. The output is N compressed frames: one key frame (typically about one megabyte) followed by $N-1$ interframes (about 10–30 kilobytes apiece).

3. *(Parallel)* Each thread runs ExCamera's **decode** operator N times to calculate the final state, then sends that state to the next thread in the batch.

4. *(Parallel)* The first thread is now finished and uploads its output, starting with a key frame. The other $x-1$ threads run **encode-given-state** to encode the first image as an interframe, given the state received from the previous thread. The key frame from `vpxenc` is thrown away; **encode-given-state** works *de novo* from the original raw image.[9]

5. *(Serial)* The first remaining thread runs **rebase** to rewrite interframes $2..N$ in terms of the state left behind by its new first frame. It sends its final state to the next thread, which runs **rebase** to rewrite all its frames in terms of the given state. Each thread continues in turn. After a thread completes, it uploads its transformed output and quits.

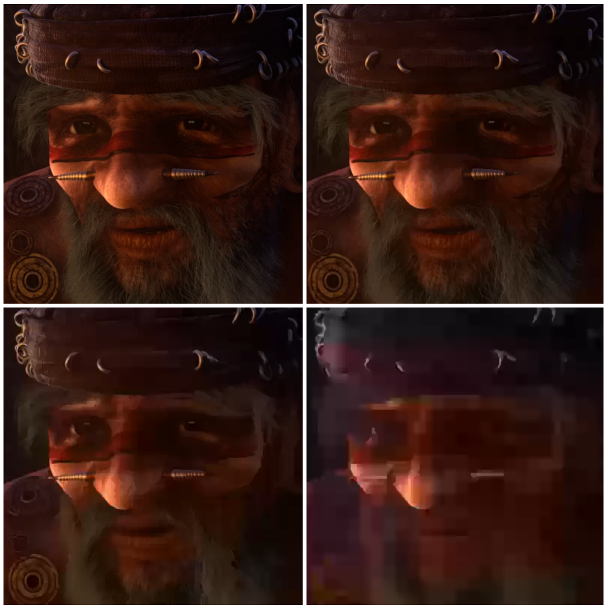
The basic assumption of this process is that rebasing (recalculating residues by subtracting the prediction from the intended image) can be done quickly, while the difficult work of searching for correlations, and therefore motion vectors, can be done in parallel.

5 Evaluation

We evaluated the ExCamera encoder, running on AWS Lambda in the mu framework, on four metrics: (1) job completion time, (2) bitrate of the compressed output, (3) quality of the output, compared with the original, (4) monetary cost of execution.

In summary, ExCamera achieved similar bitrates and quality as a state-of-the-art video encoder—Google's `vpxenc` running with multi-threading on a 128-core machine—while running about 60× faster. Pipelines with fine-grained parallelism and rebasing (e.g. ExCamera[6,16]) achieved similar results to pipelines with coarser-grained parallelism and no rebasing (e.g. ExCamera[24,1]). Our tests were performed on two open-source movies that are available in uncompressed 4K raw format: the 15-minute animated "Sintel" (2010) [20], and the 12-minute live-action "Tears of Steel" (2012) [23].

5.1 Methods and metrics



◰ (original) ◳ SSIM = 14.3 dB
◱ SSIM = 11.4 dB ◲ SSIM = 9.4 dB

Figure 3: Structural similarity (SSIM) [28] correlates with perceived image quality.

To set up the evaluation, we built and executed a mu pipeline (§ 2) to convert the raw frames of Sintel—distributed as individual PNG files—into chunks of raw images in the yuv4mpeg 8-bit 4:2:0 format.[10] Each chunk was uploaded to Amazon S3. To stay under Amazon's limit of 1,200 concurrent workers per AWS region, we spread the workers and the movies across four regions.

Our evaluation uses two other mu pipelines. The first implements the ExCamera[N,x] video-encoding algorithm described in § 4.4. We partitioned the encoding job into four large pieces, each accounting for a quarter of the movie, and ran one piece in each region. Within a region, the algorithm runs x-thread batches independently in parallel. Each batch produces $N \cdot x$ frames of compressed video, the first of which is a key frame.

Within each batch, the mu coordinator assigns tasks with awareness of the serial data-dependency relationships of the encoding algorithm: each thread depends on state objects calculated by the previous thread in the batch. Thus, as workers spawn, the coordinator assigns tasks to batches in a round-robin fashion: all of the first threads across all batches, then all of the second threads, etc. This maximizes the likelihood that threads will make continuous forward progress, avoiding stalls.

The final mu pipeline collects additional performance metrics, defined as follows:

Latency: Total time-to-completion, starting when the coordinator is launched, and ending when all threads have finished uploading compressed output to S3.

Bitrate: Total compressed size divided by duration.

Quality: Fidelity of the compressed frames to the original raw frames. We use the quality metric of *struc-*

[9] We do use the quality, or quantization, settings chosen by `vpxenc` for key frames and interframes to select the quality of the new interframe.

[10] "Tears of Steel" was already available in this format.

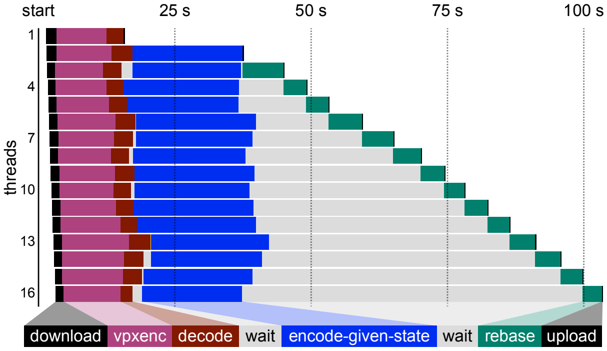


Figure 4: Execution of the ExCamera[6,16] encoder (§ 4.4) for a typical 16-thread batch on a warm start. At two points in the computation, a thread may have to "wait" for state from the thread above. The "slow" work of searching for motion vectors and prediction modes (`vpxenc` and encode-given-state) runs in parallel. The "fast" rebasing step runs serially.

tural similarity (SSIM) [28], calculated by the Xiph `dump_ssim` tool [7]. Figure 3 illustrates how SSIM correlates with perceived image quality.

5.2 Baselines

We benchmarked a range of ExCamera[N, x] pipelines against two alternative VP8 encoding systems:

1. **vpx (single-threaded):** `vpxenc`, running in single-threaded mode on an Amazon EC2 c3.8xlarge instance with a 2.8 GHz Intel Xeon E5-2680 v2 CPU. The source material was on a local SSD RAID array. Encoded output was written to a RAM disk.

   This represents the best possible compression and quality that can be achieved with a state-of-the-art encoder, but takes about 7.5 hours to encode Sintel.

2. **vpx (multi-threaded):** `vpxenc` in multi-threaded mode and configured to use all cores, running on the same kind of instance as above. This represents a common encoding configuration, with reasonable tradeoffs among compression, quality, and speed. Encoding Sintel takes about 2.5 hours. We also ran `vpxenc` on a 128-core x1.32xlarge machine, with the same results. `vpxenc`'s style of multi-threading parallelizes compression *within* each frame, which limits the granularity of parallelism.

5.3 Results

Microbenchmark. We first present a breakdown of the time spent by an ExCamera[6,16] encoder. This serves as a microbenchmark of the mu system and of the algorithm. Figure 4 shows the results of a typical batch. mu's dependency-aware scheduler assigns the first worker that AWS Lambda spawns to a task whose output will be consumed by later-spawning workers. The figure shows that the total amount of "slow" work dwarfs the time spent on rebasing, but the serial nature of the rebasing step makes it account for most of the end-to-end completion time.

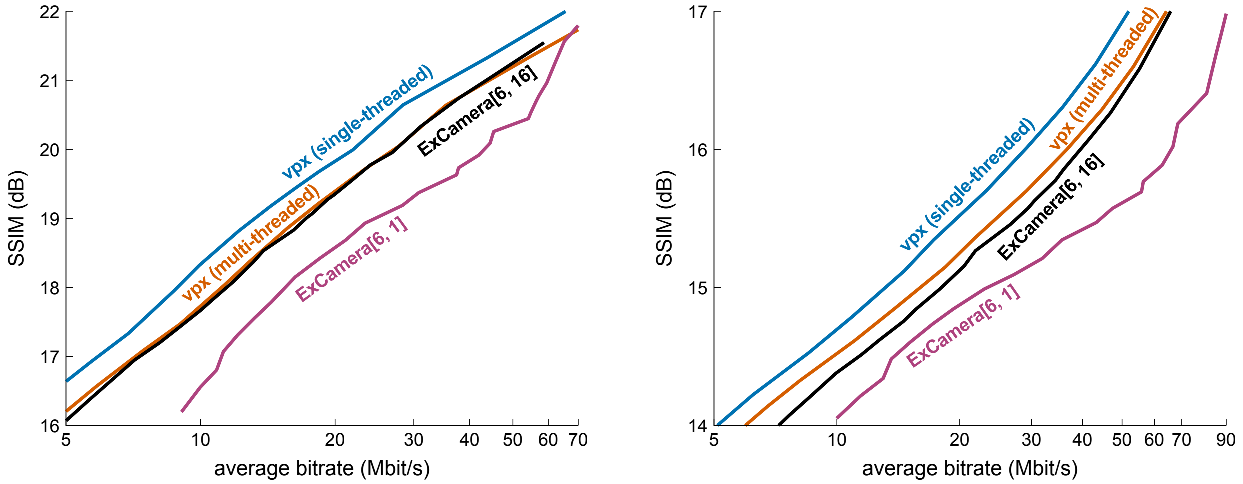
The analysis confirms that our own encode routine (encode-given-state) is considerably slower than Google's `vpxenc`, especially as the latter is encoding six frames and our routine only one. This suggests ExCamera has considerable room for optimization remaining.

Encoder performance. We ran ExCamera and the baselines with a variety of encoding parameters, resulting in a range of quality-to-bitrate tradeoffs for each approach. Figure 5 shows the results. As expected, vpx (single-threaded) gives the best quality at a given bitrate, and naive parallelism (ExCamera[6,1]) produces the worst. On the animated movie, ExCamera[6,16] performs as well as multi-threaded `vpxenc`, giving within 2% of the same quality-to-bitrate tradeoff with a much higher degree of parallelism. On the live-action movie, ExCamera is within 9%.

We also measured each approach's speed. We chose a particular quality level: for Sintel, SSIM of 20 dB, representing high quality, and for Tears of Steel, SSIM of 16 dB. We ran ExCamera with a range of settings that produce this quality, as well as the baselines, to compare the tradeoffs of encoding speed and compressed bitrate. We took the median of at least three runs of each scheme and linearly interpolated time and bitrate between runs at adjacent quality settings when we could not achieve exactly the target quality.

Figure 6 shows the results. ExCamera sweeps out a range of tradeoffs between 60× to 300× faster than multi-threaded `vpxenc`, with compression that ranges between 10% better and 80% worse. In some cases, pipelines with coarser-grained parallelism and no rebasing (e.g. ExCamera[24,1]) outperformed pipelines with finer-grained parallelism and rebasing (e.g. ExCamera[6,16]). This suggests inefficiency in our current implementation of **encode-given-state** and **rebase** that can be improved upon, but at present, the value of fine-grained parallelism and rebasing may depend on whether the application pipeline includes other costly per-frame processing, such as a filter or classifier, in addition to compressing the output. The costlier the per-frame computation, the more worthwhile it will be to use fine-grained threads.

YouTube measurement. To compare against a commercial parallel encoding system, we uploaded the official H.264 version of Sintel, which is 5.1 GiB, to YouTube. YouTube appears to insert key frames every 128 frames, and we understand that YouTube parallelizes at least some encoding jobs with the same granularity. The upload took 77 seconds over a gigabit Ethernet connection from



(a) Sintel (animated)

(b) Tears of Steel (live-action)

Figure 5: Quality vs. bitrate for four VP8 video encoders on the two 4K movies. ExCamera[6,16] achieves within 2% (Sintel) or 9% (Tears of Steel) of the performance of a state-of-the-art encoder (multi-threaded `vpxenc`) with a much higher degree of parallelism.

Stanford. We ran `youtube-dl --list-formats` every five seconds to monitor the availability of the processed versions. Counting from the end of the upload, it took YouTube 36.5 minutes until a compressed H.264 version was available for playback. It took 417 minutes until a compressed VP9 version was available for playback.

Because YouTube does not encode 4K VP8 content, and does not have adjustable quality, these figures cannot be directly compared with those in Figure 6. However, they suggest that even in systems that have no shortage of raw CPU resources, the coarse granularity of available parallelism may be limiting the end-to-end latency of user-visible encoding jobs.

Cost. At AWS Lambda's current pricing, it costs about $5.40 to encode the 15-minute Sintel movie using the ExCamera[6,16] encoder. The encoder runs 3,552 threads, each processing ¼ second of the movie. The last thread completes after 2.6 minutes, but because workers quit as soon as their chunk has been rebased and uploaded, the average worker takes only 60.4 seconds. In a long chain of rebasing, later threads spend much of their time waiting on predecessors (Figure 4). A more-sophisticated launching strategy could save money, without compromising completion time, by delaying launching these threads.

6 Limitations and future work

At present, ExCamera has a number of limitations in its evaluation, implementation, and approach. We discuss these in turn.

6.1 Limitations of the evaluation

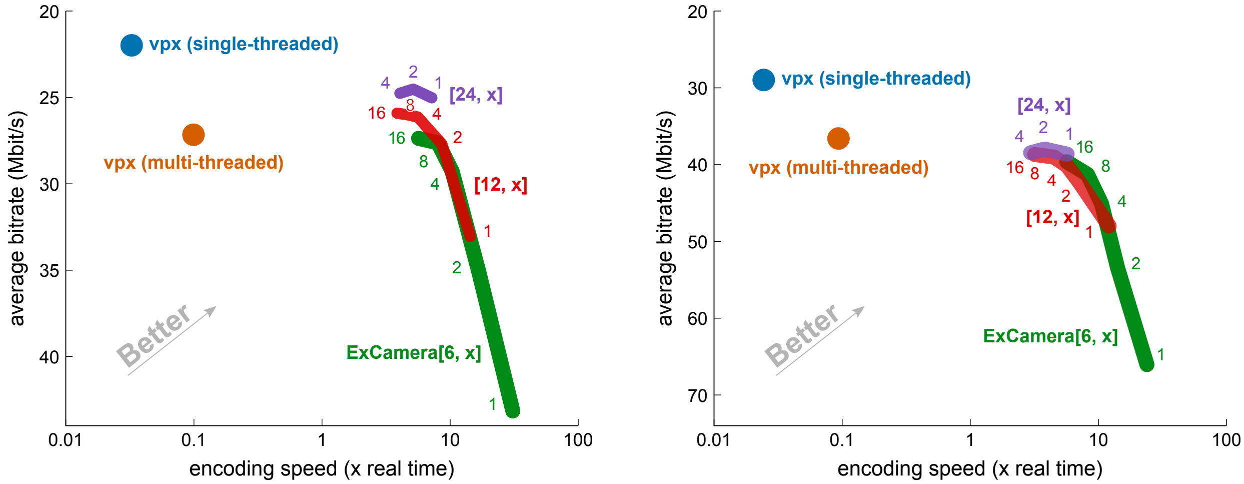
Only evaluated on two videos. We have only characterized ExCamera's performance on two creative-commons videos (one animated, one live-action). While these are widely used benchmarks for video encoders, this may have more to do with their availability in uncompressed formats than suitability as benchmarks. We will need to verify experimentally that ExCamera's results generalize.

If everybody used Lambda as we do, would it still be as good? To the best of our knowledge, ExCamera is among the first systems to use AWS Lambda as a supercomputer-by-the-second. ExCamera slams the system with thousands of TLS connections and threads starting at once, a workload we expect not to be characteristic of other customers. We don't know if Lambda would continue to provide low latencies, and maintain its current pricing, if ExCamera-like workloads become popular.

6.2 Limitations of the implementation

encode-given-state is slow and has poor compression efficiency. The microbenchmark of Figure 4 and other measurements suggest there is considerable room for optimization in our encode-given-state routine (§ 4.4). This is future work. We explored building further on `vpxenc` so as to avoid using our own *de novo* encoder at all, but did not achieve an improvement over the status quo.

Pipeline specification is complex. In addition to parallel video compression, ExCamera supports, in principle, a range of pipeline topologies: per-image transformations followed by encoding, operations that compose multiple input frames, edits that rearrange frames, and computer-



(a) Sintel (SSIM of 20 dB)

(b) Tears of Steel (SSIM of 16 dB)

Figure 6: Bitrate vs. encoding speed, at constant quality relative to the original movie. ExCamera sweeps out a range of tradeoffs.

vision analyses on each frame. We are working to design a pipeline-description language to specify jobs at a higher level of abstraction.

Worker failure kills entire job. Because ExCamera's jobs run for only a few minutes, and the mu framework only assigns tasks to Lambda workers that successfully start up and make a TLS connection back to the coordinator, mu does not yet support recovery from failure of a Lambda worker in the *middle* of a task. Producing Figure 6 required 640 jobs, using 520,000 workers in total, each run for about a minute on average. Three jobs failed.

ExCamera encodes VP8. What about newer formats? It took considerable effort to implement a VP8 decoder, encoder, and rebase routine in explicit state-passing style. ExCamera only works with this format, which was designed in 2008 and has been largely superseded by the newer VP9. Although Google released a VP9 version of `vpxenc` in 2013, the VP9 specification was not released until April 2016. To the best of our knowledge, there has not yet been an independent implementation of VP9.[11] We believe the rebasing technique is general enough to work with VP9 and other current formats, but hesitate to predict the efficacy without an empirical evaluation. We hope to persuade implementers of the benefit of ExCamera's explicit-state-passing style and to provide a similar interface themselves.

6.3 Limitations of the approach

Many video jobs don't require fine-grained parallelism. ExCamera is focused on video pipelines where the processing of a single frame consumes a large amount of CPU resources—e.g., encoding of large (4K or VR) frames, expensive computer-vision analyses or filtering, etc. Only in these cases is it worthwhile to parallelize processing at granularities finer than the practical key-frame interval of a few seconds. For encoding "easier" content (e.g., standard-definition resolutions) without extensive per-frame computation, it is already possible to run a naively parallel approach on multi-second chunks with acceptable end-to-end latency. Figure 6 shows that if the application is simply encoding video into the VP8 format, with no expensive per-frame processing that would benefit from fine-grained parallelism, pipelines with coarser granularity and no rebasing (e.g. ExCamera[24,1]) can sometimes perform as well as pipelines with finer granularity and rebasing (e.g., ExCamera[6,16]).

Assumes video is already in the cloud. We assume that the source material for a video pipeline is already in the cloud and accessible to the workers—in our cases, loaded into S3, either in raw format (as we evaluated) or compressed into small chunks. Many video pipelines must deal with video uploaded over unreliable or challenged network paths, or with compressed source video that is only available with infrequent Stream Access Points.

7 Related work

Data-processing frameworks. Batch-processing frameworks such as MapReduce [8], Hadoop [12], and Spark [32] are suited to tasks with coarse-grained parallelism, such as mining error logs for abnormal patterns. In these tasks, each thread processes an independent subset of the data. It is challenging to express video encoding

[11] Existing software implementations have been written by Google employees or by former employees who worked on `vpxenc`.

within these frameworks, because video compression depends on exploiting the correlations among frames.

A number of additional distributed-computing frameworks have been proposed that implement pipeline-oriented computation. Apache Tez [24] and Dryad [13] support arbitrary DAG-structured computation, with data passed along edges in a computation graph. Such systems could be used to support video-processing pipelines.

Cloud computing. Data-processing frameworks today rely on computational substrates such as Amazon EC2 and Microsoft Azure, which provide heavyweight virtualization through virtual machines that run an entire operating system. In contrast, ExCamera relies on AWS Lambda's cloud functions, which provides lightweight virtualization. For the same monetary cost, it can access many more parallel resources, and can start and stop them faster. To our knowledge, ExCamera is the first system to use such cloud functions for compute-heavy tasks, such as video encoding; current uses [25] are focused on server-side scripts for Web microservices and asynchronous event handlers.

After the submission of this paper, we sent a preprint to a colleague who then developed PyWren [14, 15], a framework that executes thousands of Python threads on AWS Lambda. Our Figure 2 was inspired by a similar measurement in [14]. ExCamera's mu framework differs from PyWren in its focus on heavyweight computation with C++-implemented Linux threads and inter-thread communication.

Parallel video encoding. Parallel video encoding has a substantial literature. Broadly speaking, existing approaches use one of two techniques: intra-frame parallelism, where multiple threads operate on disjoint areas of the same frame [3, 27, 31], or frame parallelism, where each thread handles a different range of frames [1, 22, 30].

Intra-frame parallelism is widely used to enable real-time encoding and decoding with small numbers of threads. This kind of parallelism does not scale with the length of a video, and in practice cannot be increased beyond tens of threads without severely compromising compression efficiency. This limits the speedup available.

Frame parallelism can scale with the length of a videos, but also sacrifices coding efficiency at higher degrees of parallelism (§ 3). Some prior systems [9] employ frame parallelism by searching for natural locations to place key frames, then encoding the ranges between pairs of key frames independently. These systems operate at coarse granularity: thousands of frames per worker. The goal of ExCamera's encoder is to exploit fine-grained parallelism without sacrificing coding efficiency.

Reusing prior encoding decisions. ExCamera's frame-rebasing technique involves transforming an interframe into a new interframe, preserving the motion vectors and prediction modes from an earlier encoding, but recalculating the residue so that the frame can be applied to a new decoder state.

The idea of preserving motion vectors and prediction modes to save CPU is not new: similar techniques have a long history in the context of real-time transcoding (e.g., [21]), where one input stream of compressed video is turned into several output streams at varying bitrates. In these systems, the original motion vectors and prediction modes can be reused across each of the output streams. However, to the best we have been able to determine, ExCamera is the first system to use this idea to enable low-latency parallel video encoding.

8 Conclusion

ExCamera is a cloud-based video-processing framework that we envision as the backend for interactive video applications. It can edit, transform, and encode a video, including 4K and VR material, with low latency.

The system makes two major contributions: a framework to run general-purpose parallel computations on a commercial "cloud function" service with low latency, and a video encoder built with this framework that achieves fine-grained parallelism without harming compression efficiency.

ExCamera suggests that an explicit state-passing style, which exposes the internal state of a video encoder and decoder, is a useful interface that can enable substantial gains for video-processing workloads—applications that will only grow in importance. We encourage the developers of video codecs to implement such abstractions.

9 Acknowledgments

We thank the NSDI reviewers and our shepherd, Dave Maltz, for their helpful comments and suggestions. We are grateful to Timothy Terriberry for feedback throughout this project, and to Philip Levis and Fraser Brown for comments on versions of this paper. We benefited from helpful conversations with Marc Brooker, Tim Wagner, and Peter Vosshall of Amazon, and with James Bankoski, Josh Bailey, Danner Stodolsky, Roberto Peon, and the video-encoding team at Google. This work was supported in part by NSF grants CNS-1528197 and CNS-1564185, a Google Research Award, and an Amazon Cloud Credits award, with additional funding from Dropbox, VMware, and Facebook.

References

[1] AARON, A., AND RONCA, D. High quality video encoding at scale. In *Netflix Tech Blog* (Decem-

ber 9, 2015). http://techblog.netflix.com/2015/12/high-quality-video-encoding-at-scale.html.

[2] Agarwal, R., Khandelwal, A., and Stoica, I. Succinct: Enabling queries on compressed data. In *NSDI* (May 2015).

[3] Akramullah, S., Ahmad, I., and Liou, M. A data-parallel approach for real-time MPEG-2 video encoding. *Journal of Parallel and Distributed Computing 30*, 2 (1995), 129–146.

[4] Azure Functions—Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/.

[5] Chacon, S., and Straub, B. *Pro Git: Git Branching - Rebasing*, 2nd ed. Apress, Berkeley, CA, USA, 2014. https://git-scm.com/book/en/v2/Git-Branching-Rebasing.

[6] Cisco VNI forecast and methodology, 2015–2020. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html.

[7] Xiph Daala repository. https://github.com/xiph/daala.

[8] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[9] ELDER. http://web.archive.org/web/20080105113156/http://www.funknmary.de/bergdichter/projekte/index.php?page=ELDER.

[10] Cloud Functions—Serverless Microservices. https://cloud.google.com/functions/.

[11] *Advanced video coding for generic audiovisual services*, May 2003. Rec. ITU-T H.264 and ISO/IEC 14496-10 (http://www.itu.int/rec/T-REC-H.264-201602-I/en).

[12] Apache Hadoop. http://hadoop.apache.org.

[13] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys* (Mar. 2007).

[14] Jonas, E. Microservices and teraflops. *http://ericjonas.com/pywren.html* (Oct. 2016).

[15] Jonas, E., Venkataraman, S., Stoica, I., and Recht, B. Occupy the Cloud: Distributed computing for the 99%. *https://arxiv.org/abs/1702.04024* (Feb. 2017).

[16] LINPACK_BENCH: The LINPACK benchmark. https://people.sc.fsu.edu/~jburkardt/cpp_src/linpack_bench/linpack_bench.html.

[17] *Generic coding of moving pictures and associated audio information [MPEG-2] – Part 2: Video*, May 1996. Rec. ITU-T H.262 and ISO/IEC 13818-2 (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=61152).

[18] *Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats*, April 2012. ISO/IEC 23009-1 (http://standards.iso.org/ittf/PubliclyAvailableStandards).

[19] IBM Bluemix OpenWhisk. http://www.ibm.com/cloud-computing/bluemix/openwhisk/.

[20] Roosendaal, T. Sintel. In *SIGGRAPH* (Aug. 2011).

[21] Sambe, Y., Watanabe, S., Yu, D., Nakamura, T., and Wakamiya, N. High-speed distributed video transcoding for multiple rates and formats. *IEICE - Trans. Inf. Syst. E88-D*, 8 (Aug. 2005), 1923–1931.

[22] Shen, K., Rowe, L. A., and Delp III, E. J. Parallel implementation of an MPEG-1 encoder: faster than real time. In *SPIE* (July 1995).

[23] Tears of Steel: Mango open movie project. https://mango.blender.org.

[24] Apache Tez. https://tez.apache.org/.

[25] Examples of how to use AWS Lambda. http://docs.aws.amazon.com/lambda/latest/dg/use-cases.html.

[26] *VP9 Bitstream & Decoding Process Specification Version 0.6*, March 2016. http://www.webmproject.org/vp9/.

[27] VP8 Encode Parameter Guide. http://www.webmproject.org/docs/encoder-parameters/.

[28] Wang, Z., Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612.

[29] Wilkins, P., Xu, Y., Quillio, L., Bankoski, J., Salonen, J., and Koleszar, J. VP8 Data Format and Decoding Guide. RFC 6386, Nov. 2011.

[30] x264 Settings. http://www.chaneru.com/Roku/HLS/X264_Settings.htm#threads.

[31] YU, Y., AND ANASTASSIOU, D. Software implementation of MPEG-II video encoding using socket programming in LAN. In *SPIE* (Feb. 1994).

[32] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (Apr. 2012).

Appendix: `vpxenc` command lines

We used `vpxenc` 1.6.0 (Sintel) and 1.6.1 (Tears of Steel) with the below command-line arguments in our evaluations (§ 5). `$QUALITY` is a parameter that indicates the target quality of the compressed video; it ranges from 0 (best) to 63 (worst). When running vpx multi-threaded, `$NPROC` is set to 31 (one fewer than the number of cores, as recommended in the documentation) and `$TOK_PARTS` is set to 3. When running single-threaded, they are set to 1 and 0, respectively.

```
vpxenc --codec=vp8 \
       --good \
       --cpu-used=0 \
       --end-usage=cq \
       --min-q=0 \
       --max-q=63 \
       --buf-initial-sz=10000 \
       --buf-optimal-sz=20000 \
       --buf-sz=40000 \
       --undershoot-pct=100 \
       --passes=2 \
       --auto-alt-ref=1 \
       --tune=ssim \
       --target-bitrate=4294967295 \
       --cq-level=$QUALITY \
       --threads=$NTHREADS \
       --token-parts=$TOK_PARTS \
       -o output_file \
       input.y4m
```