Sprocket: A Serverless Video Processing Framework

Lixiang Ao
University of California, San Diego
liao@cs.ucsd.edu

Liz Izhikevich
University of California, San Diego
eizhikev@ucsd.edu

Geoffrey M. Voelker
University of California, San Diego
voelker@cs.ucsd.edu

George Porter
University of California, San Diego
gmporter@cs.ucsd.edu

ABSTRACT

Sprocket is a highly configurable, stage-based, scalable, serverless video processing framework that exploits intra-video parallelism to achieve low latency. Sprocket enables developers to program a series of operations over video content in a modular, extensible manner. Programmers implement custom operations, ranging from simple video transformations to more complex computer vision tasks, in a simple pipeline specification language to construct custom video processing pipelines. Sprocket then handles the underlying access, encoding and decoding, and processing of video and image content across operations in a highly parallel manner. In this paper we describe the design and implementation of the Sprocket system on the AWS Lambda serverless cloud infrastructure, and evaluate Sprocket under a variety of conditions to show that it delivers its performance goals of high parallelism, low latency, and low cost (10s of seconds to process a 3,600 second video 1000-way parallel for less than $3).

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Computer systems organization** → **Cloud computing**; • **Information systems** → *Multimedia information systems*;

ACM Reference Format:
Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of ACM Symposium on Cloud Computing , Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 12 pages.
https://doi.org/10.1145/3267809.3267815

1 INTRODUCTION

Frameworks such as MapReduce [10], Spark [35], and GraphLab [24] have made it possible to process terabytes of traditional data sources, such as logs, transaction records, and other Web data, as ordinary routine tasks. However, little attention has been given to processing video content, an increasingly dominant source of data both in terms of bytes generated and bytes transmitted on the Internet. Video is increasingly subjected to a rapidly expanding "pipeline" of computation, including color, lighting, and shading adjustments, digital watermarks, format, resolution, and frame-rate modifications, and transcoding for eventual delivery. Moreover, new forms of machine learning and computer vision algorithms enable extracting semantically meaningful information from video frames, enabling algorithms to "peer inside" what was previously an opaque source of data. Nonetheless, extracting knowledge from raw video is a challenge, due in part to the inability to easily process video content, as well as the sheer size of video objects.

In this paper we describe the design and implementation of Sprocket, a serverless video processing. Sprocket enables developers to program a series of pipelined operations over video content in a modular, extensible manner. By composing operations, developers are able to build custom processing pipelines whose elements can be reused and shared. Sprocket handles the underlying access, loading, encoding and decoding, and movement of video and image content across operations in a highly parallel manner. Our goal with Sprocket is to not only support traditional video processing operations and transformations, such as transcoding to lower resolutions or applying filters, but also to enable much more sophisticated video processing applications, such as answering queries of the form, "Show just the scenes in the movie in which Wonder Woman appears".

We specifically target cloud services because they are an ideal platform for our goals of low latency, high parallelism at low cost, and support for new kinds of data processing applications. Tools that operate with low latency become interactive and more useful. With Sprocket, we want developers to be able to create applications that let users interactively transform video or perform queries across video content and stream the results. Independent of the length of video input, we show that Sprocket applications can start streaming results to users in seconds.

To achieve interactivity, though, we need to employ high degrees of parallelism to reduce processing time, while also minimizing the monetary cost of using such extensive parallelism. As a result, we designed and implemented Sprocket to use the new serverless frameworks offered by cloud providers, such as Amazon's Lambda and Microsoft's and Google's Function platforms. The serverless platforms use containers for virtualization and isolation, and applications can allocate and use thousands of them at millisecond timescales. Furthermore, providers offer these containers with a billing model that also charges at sub-second time scales. Running a Sprocket application that can process a video with 1,000-way concurrency using Lambdas on a full-length HD movie costs about $3 per hour of processed video.

Finally, the availability of sophisticated tools on cloud platforms enables new kinds of applications for video content. Cloud providers are increasingly offering support for a wide variety of image processing and computer vision APIs, ranging from OCR to face and object detection and recognition. We show how a Sprocket application consisting of just a few stages can use the Amazon Rekognition service to perform sophisticated queries on video, such as returning just the scenes in a movie featuring a particular actor, and again stream the results in seconds.

In the remainder of this paper we start by providing background on video data properties and challenges. We then describe the design and implementation of Sprocket, in particular the pipeline programming model and support for data streaming and worker scheduling. Finally, we evaluate Sprocket's performance from various perspectives, including where it spends its time executing pipeline operations, how it exploits parallelism and intrinsic properties of the processing video, how it meets streaming deadlines when generating output, the performance characteristics of a complex application, and how it compares to other general processing frameworks.

2 BACKGROUND

Video data is one of the predominant forms of data in the world: 70% of consumer Internet traffic is compressed video content [7]. Video streaming service providers like YouTube and Netflix, and social networks like Facebook and Instagram, receive, edit, encode, and stream multiple TBs of video data every day. Processing video data, however, presents unique challenges due to its unique properties.

As 4K and VR videos become more common, the size of video files increase rapidly: 4K videos typically have a bit-rate of over 30 Mbps, and a 2-hour-long movie can be 30 GB in size. Even applying a simple transformation on these videos can take hours using a single machine. In production environments, the video processing pipeline can become very complicated. Some large-scale applications require hundreds of tasks to be executed. The efficient scheduling and execution of video processing pipelines becomes a challenging task in these systems. Sprocket's use of a serverless framework allows for scheduling and processing to happen seamlessly and efficiently.

Working with the individual frames of a video is also no trivial matter. Since video encoding takes advantage of both spatial and temporal similarity for compression, individual frames become dependent on other frames. The encoder thus inserts "keyframes" to allow for groups of pictures (GOP) to be independent of each other. This structure, however, results in individual frames being non-uniform in size, as shown in Figure 1, and thus can greatly affect the behavior of sending, storing, and computing on the data. Accessing frames within a GOP requires at least partial decoding of that segment of the video, since all but the initial frame in the GOP are encoded as differences from that primary image. As we will show, Sprocket not only handles this decode operation on behalf of application pipelines, but also uses the encoded video to enable its approach to straggler mitigation, described in Section 3.6.

The content of the video itself can also greatly affect the behavior of any video-processing system. For example, a system that is running face detection will inherently take more time on a video that has many faces, as opposed to no faces. Furthermore, some Sprocket pipelines contain data dependencies, in addition to higher-level control dependencies, that impose unique requirements on its scheduling approach, as described in the next section. Sprocket takes the content of the video into account by dynamically dedicating more serverless computing resources when processing certain scenes, as described in Section 3.5.4.

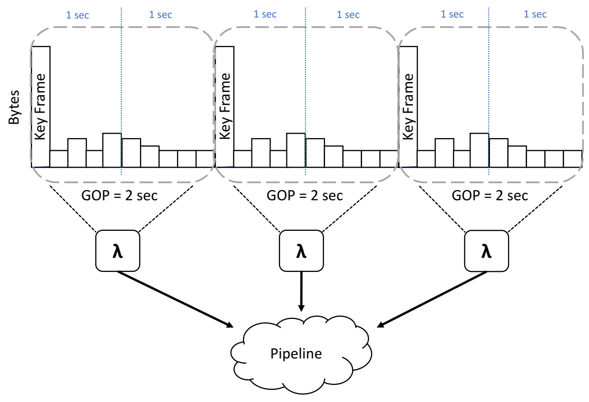


Figure 1: Frame sizes in a GOP.

Recent work has also specifically focused on video processing applications. ExCamera is a highly parallel video encoder that encodes small chunks of video in Lambda threads in parallel, ultimately stitching them together into a single final video file [12] (indeed, experience with Lambdas developing ExCamera convinced us of their viability for Sprocket). Other video analytic systems include VideoStorm, which focuses on querying characteristics of streamed live video content and the resource, quality, and delay tradeoffs of scheduling large numbers of queries on a cluster [37]. The Streaming Video Engine is Facebook's framework for processing all user-supplied video content at scale, with a particular focus on reducing the latency of ingesting and re-encoding video content so that it can be shared with other users [17].

3 SPROCKET DESIGN

Before describing the design of Sprocket, we first highlight two example scenarios of its use. First, imagine that a user has a video on their laptop that they are editing, and they wish to see a version of the video with a remapped set of colors. They invoke, either via a command-line tool or through an interface in their editing program, a Sprocket-based filter tool with the new color mapping. The video is uploaded to the cloud, and Sprocket applies the filter to the video and they are able to stream the updated version in their browser within a few seconds.

In a second example, imagine that a user is watching a two-hour long video of a school play, and wants to only watch the portions starring their child. They visit a web page, backed by Sprocket, that lets them submit a URL of the video, along with a URL of their child's face, and within a few seconds they are able to stream an abbreviated version of the play in their browser featuring only those scenes starring their child.

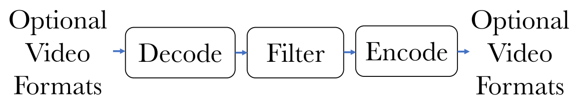


Figure 2: Logical overview of the Video Filter pipeline.

3.1 Overview

Sprocket is a scalable video processing framework designed to transform a single video input according to a user-specified program. Users write these programs in a domain-specific language, described in Section 3.7, which is expressed as a dataflow graph. The dataflow graph consists of vertices and edges, similar to other dataflow systems such as Tez [32]. Edges (which we also refer to as *streams*) convey data between vertices, in particular individual frames of video, groups of frames, or compressed segments of consecutive frames we call *chunks*. Vertices execute individual functions, specified by the user, on data that arrives at their input(s), emitting any resulting frames or chunks to the next part of the DAG via one or more output edges. We refer to a single DAG program as a *pipeline*, and refer to vertices in that DAG as pipeline *stages*.

The modular design of the pipeline and pipeline specification allows developers to build complex video processing systems with little effort. The simplicity of the interactions with the Lambda infrastructure greatly mitigate the management overhead.

In this section, we describe a set of example applications built on the Sprocket framework, highlighting underlying stage implementations used to build those applications. Developers can use these stages to build custom pipelines, and can also implement new stages to extend this set of functionality.

3.2 Application 1: Video filter

Our first application example is a pipeline that applies an image processing filter, drawn from those supported by the `FFmpeg` tool, to an entire video (Figure 2). This pipeline consists of three stages: Decode, Filter, and Encode. An input, which can be in the format of a video link (YouTube, Vimeo, etc.), a cloud-local file hosted on S3, or uploaded by the user, will be sent to multiple Lambda workers in parallel. Each worker will decode a chunk of video into a set of sequential frames. The Lambda worker then invokes `FFmpeg` on each frame. Lastly, each transformed frame will be encoded into an output format, either a single compressed file (e.g., via the Ex.Camera distributed encoder) or to multiple fixed-length chunks suitable for streaming via the MPEG-DASH [9] format.

3.2.1 Stage design. Sprocket will initially evaluate the input video type and handle initiating the parallel nature of the pipeline accordingly. For example, for a video URL input, Sprocket will broadcast the link to each of many parallel download-and-decode workers, along with video metadata and the number of frames assigned to each worker, so that those workers can download and begin processing the input video in parallel.

This Sprocket pipeline consists of three stages:

Decode: This stage decodes a specified chunk of input video into individual frames (in PNG format). Decode receives video metadata as input, along with the timestamp of where in the video to begin decoding, and how many frames to output to the downstream worker.

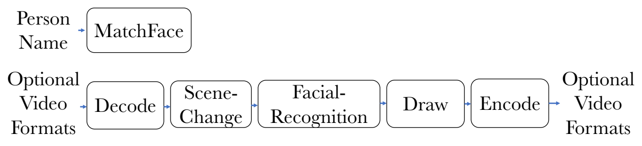


Figure 3: Logical overview of Facial Recognition pipeline stages.

After processing, the decode stage emits the decoded frames to the S3 intermediate storage system.

Filter: The Filter stage applies the `FFmpeg` binary to a chunk of frames. Filter is spawned directly after the Decode stage and receives metadata along with references to the location of the frames stored on intermediate storage. Filter collects the frames from S3 and applies one of its internal filters as specified in the pipeline's pipespec configuration file, described later in Section 3.7.

Encode: This stage is responsible for encoding frames. Encode also uses `FFmpeg`, running in a different Lambda worker, which receives metadata along with references to the location of the frames in S3. Encode collects the frames from S3, encoding them using the specified encoder format, and finally writes them either in MPEG-DASH format or a single compressed output file generated by ExCamera. The final result is stored in S3.

3.3 Application 2: Facial Recognition

A second pipeline we describe implements facial recognition, which demonstrates a more sophisticated set of operations, including calling out to other cloud services to implement the facial recognition support (Figure 3). This pipeline takes an actor's name and a reference video URL as input, and draws a box around the given actor's face in all scenes of the video. This application could support the theatrical production motivating scenario presented earlier.

3.3.1 Stage design. The Facial Recognition pipeline consists of six stages: MatchFace, Decode, SceneChange, FacialRecognition, Draw, and Encode (Figure 3). At the beginning of the pipeline, an actor's name, provided by the user, will be used to locate candidate images from a Web search for later use in facial recognition. In parallel, the provided input video URL is fetched and decoded into fixed-length chunks of frames, currently one second each. Workers in parallel will then run a scene change algorithm on each chunk of frames to bin them into separate scenes. For each set of chunks grouped in scenes, a worker will then run a facial recognition algorithm to determine if the target face is present in that scene. If a face is identified in the scene, the chunk of frames will have a bounding box drawn on all the frames at the appropriate position returned by the vision algorithm API, which is then sent downstream to the Encode stage. If no face is detected, then the group of chunks will be sent directly to Encode.

Along with the stages described in the previous section, the following stages are used in the Facial Recognition pipeline:

MatchFace: The MatchFace stage searches for a target image for the face of a person whose name is specified as a parameter. Sprocket currently uses Amazon's Rekognition API [27], but could also use other service offerings: Microsoft offers a computer vision

API as part of its Cognitive Services cloud offering [25], and Google offers a cloud-hosted vision system for labeling and understanding images through its Google Cloud Vision API [14].

MatchFace invokes one of these third-party image search services (in our case Amazon Rekognition) to find the top-*k* images returned given the provided name. The stage then iterates through the returned images and runs a face detection algorithm, via external API call, to determine whether the chosen target image contains a face. The first image to pass the facial detection algorithm becomes the selected target image. MatchFace then stores the selected image in S3 for eventual use by downstream stages. Unlike other stages, MatchFace itself does not emit any data directly to downstream stages.

SceneChange: The SceneChange stage detects scene changes in a set of decoded frames. It is invoked after the Decode stage, and is sent a reference to the decoded frames stored in S3. SceneChange collects the frames from S3 and, after detecting the scene change offsets (using an algorithm internal to `FFmpeg`), emits an event containing a list of these references to frames stored in S3, paired with a boolean value marking which frames serve as the boundaries of the scene change.

FacialRecognition: The FacialRecognition stage detects if a group of frames contains a target face (e.g., of the provided actor). The FacialRecognition stage is spawned once for every group of frames that make up one scene. We chose this design, rather than running on every frame in the scene, due to rate limits when invoking third-party recognition algorithms. FacialRecognition downloads the frames from S3 and calls the facial recognition algorithm once on every *n* frames in the scene. The facial recognition algorithm returns whether or not the target image was detected in the frame, and a bounding box of the identified face in the original frame. If at least one frame in the scene is found to contain the target face, all frames in the scene are marked as having the target face. The stage then emits an event containing a list of references to the frame in S3 paired with a bounding box of the identified face. If no target face is identified, FacialRecognition emits a list of frame references paired with empty bounding boxes.

Draw: The Draw stage draws a box at an arbitrary position in the frame, in this case provided as a bounding box around a recognized face. Draw is instantiated from the FacialRecognition stage and only continues if a scene was labeled as containing a recognized face. Otherwise, draw automatically skips to its final state and sends references to the frames in S3 directly to the downstream stage. Draw only uses the dimensions from one bounded box to draw the same bounding box on all frames. Therefore, Draw assumes that there is little movement of faces in a single scene. We leave as future work interpolating the position of the box based on sampled points throughout the scene. Draw writes the new frames to S3 and emits an event containing a list of frame references.

An alternative version of a Facial Recognition pipeline can also choose to emit scenes that only contain the recognized face. In this case, the Draw stage would be replaced by a SceneKeep stage that only emits references to frames if a face is recognized. Otherwise, the stage will emit an empty list and those particular frames will never be encoded. SceneKeep would be employed to implement the version of the pipeline that edits out all scenes of a theatrical production that do not include a given actor (e.g., the user's child).

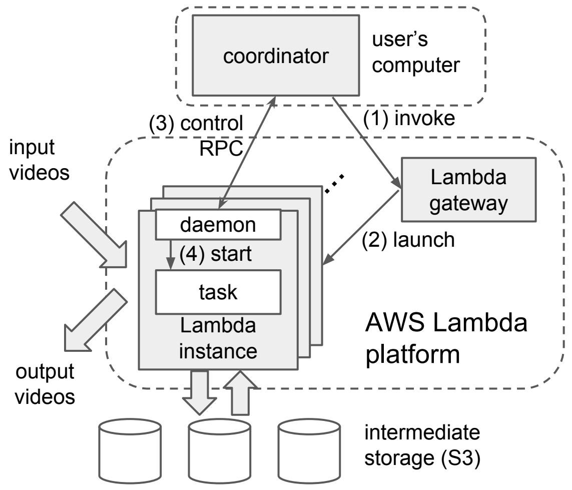


Figure 4: Sprocket Overview

3.3.2 Calling external cloud services. Calling an external API to run a facial recognition algorithm creates different challenges for Sprocket. The first is the additonal latency of calling the external service that is beyond Sprocket's control. The second is that Sprocket may encounter an API call throughput limit that slows down the execution time of stages. This execution latency increases as the parallelism of the pipeline increases, since more concurrent calls create a faster overload of the external API. To address this, Sprocket does two things. First, the facial recognition pipeline includes the scene detection stage, reducing the number of calls to the facial recognition API within one scene. Second, the pipeline has the option to use the streaming scheduler, described in more detail in Section 4.3, which adaptively calculates the minimum amount of Lambdas needed to meet a streaming deadline throughout execution. In this way, Sprocket limits the amount of concurrent API calls needed, thus avoiding API call throughput limits.

3.4 Invoking and managing Lambdas

The Sprocket Coordinator manages the system-wide "control plane," managing the life-cycles of Lambda workers including spawning and tearing them down, and orchestrating the flow of data between workers via the intermediate data storage system. We implemented the Sprocket coordinator in Python, which runs either on a VM in the cloud or on the user's computer. Amazon has since deployed *Step Functions* [30], which is a Lambda workflow management system that serves a similar purpose as our coordinator. Currently AWS Step Functions cannot implement Sprocket pipelines, since support for dynamic levels of parallelism in Step Functions is currently limited. It can scale the number of Lambdas based on an overall workload, but it is not evident how to scale individual stages based on dynamic load or resource optimizations at that stage. Figure 4 shows a detailed overview of the main components of our Python-based coordinator design.

The coordinator maps the execution of one task within the Sprocket dataflow graph to a Lambda worker. An individual Lambda worker is invoked by receiving data on its inputs via the delivery function, which comes either from external input sources or from upstream workers via the intermediate storage system. In the AWS cloud, Lambda instances cannot establish direct network connections between each other, and so we convey all data from upstream workers to downstream workers through the intermediate store, in particular through S3. This approach has the added advantage of enabling the scheduler to decouple the execution lifetimes of upstream and downstream stages, enabling them to exist at disparate times. Although the rather heavy-weight S3 storage interface may seem like a potential performance bottleneck, Jonas et al. [19] have shown that it can scale to support thousands of clients without imposing significant performance penalties.

We indirectly control Lambda workers via a modified version of Mu [12], a framework for managing parallel execution of Lambda instances. The Coordinator sends invocation requests to the AWS API gateway to manage Lambda functions. Once the Lambda instance is started, it spawns a local daemon within the Lambda that establishes a connection back to the Coordinator so that the Coordinator can communicate with and manage them. Each Lambda worker is implemented as a state machine, and the particular state machine employed is sent through this interface. In this way, an individual Lambda does not yet know what role it will play in the overall pipeline until it has connected back to the coordinator to request further instructions. This approach has the potential benefit of reducing variance and stragglers.

The Coordinator and the Lambda workers interact with each other using asynchronous RPC. The Coordinator gives commands to the Lambda workers while the workers reply with status reports. The Coordinator treats the status reports as inputs to its internal DAG dataflow, enabling it to generate new states to send to Lambda workers to further process the overall pipeline. If the status report from a worker indicates an error in executing the most recent command, the coordinator can re-send the command to the worker or potentially spawn a new worker to continue the execution of that task.

We extend Mu in several ways to support Sprocket's design. Instead of batch-style invocation of Lambdas in Mu, we support dynamic creation of tasks during processing, and asynchronous messaging and data transfer among the Lambda workers. Moreover, Sprocket abstracts away from any particular serverless platform and can easily support other platforms such as Google Cloud Functions [13].

3.5 Pipeline scheduling

3.5.1 Managing on-demand Lambdas. Sprocket's dataflow scheduler relies on the stateless nature of AWS Lambda functions. Generally, a cluster or job scheduler will allocate a large set of resources in advance, and then manage that pool of resources over time across a large number of jobs. For example, Mesos [16] manages resources among frameworks, with each having their own scheduler to further manage resources among applications and tasks. Because Sprocket is targeting a serverless cloud environment, the Sprocket scheduler can allocate and deallocate Lambda resources on fine-grained timescales, rather than all at once at system start time. The cloud provider is then responsible for managing that pool of resources.

The primary resource limit that Sprocket faces is a concurrency limit imposed by the provider, which bounds per account the number of Lambda workers that can run concurrently, as well as the rate at which new Lambda workers can be spawned. In this work, we focus on applying Sprocket to a single instance of a single pipeline, and so assume that it can freely use resources up to the account-wide concurrency limit.

When the demand for concurrent workers is within the concurrency limit set by the provider, scheduling is straightforward: any requested task will immediately trigger an invocation of a worker and begin its processing after the worker is launched. We have observed that the vast majority of Lambda workers launch with subsecond latency. In terms of responsiveness, we therefore treat the workers as a homogeneous resource.

When the demand for Lambda workers exceeds the concurrency limit, such as when processing a long video or processing a very deep pipeline, some tasks have to be deferred and scheduled later. Clients can specify which scheduling policy to use, either optimizing for the lowest overall execution time, or optimizing to prioritize earlier frames for streaming results. The "overall time" strategy minimizes the overall job time by increasing the parallelism of each stage and making sure the task is utilizing the most available Lambda workers. The "time to first frame" strategy identifies the tasks that are on the critical path of the eventual output's frame presentation time (the time at which an output frame will be displayed), and assigns these tasks higher scheduling priorities.

3.5.2 Optimizing for streaming. Sprocket's ability to seamlessly allocate and deallocate resources also allows it to change its scheduling behavior in real time, based on the pattern of the current job. Concretely, Sprocket's streaming scheduler continuously keeps track of the amount of time it is taking to process the current frames to determine if it will meet the streaming deadline. If Sprocket is currently ahead of schedule, the streaming scheduler will speculatively put the current pipeline to sleep for the number of seconds nearly equivalent to the difference between the streaming deadline and the current execution time. Putting the executing pipeline to sleep not only optimizes for use of minimum Lambda resources, as the number of new Lambda invocations drops down to zero for the current pipeline, but also lends itself well to achieving load balancing so a simultaneous different pipeline can be run on sprocket. Section 4.3 evaluates Sprocket's streaming performance.

3.5.3 Supporting complex dependencies. Depending on the complexity of the implementation of a vertex in a given dataflow DAG, it is not always straightforward for the scheduler to know when to execute that stage. Indeed, there may exist dependencies across a stage's input that depend on other parts of the DAG. To capture these dependencies, Sprocket defines a *delivery function* for every stage which specifies the dependencies on its inputs. It is the responsibility of the Sprocket framework, not the implementer of the stage, to manage and satisfy these dependencies. Managing the dependencies and scheduling of inputs to continuous query and dataflow systems is a well-studied problem in a number of domains, and Sprocket's delivery functions might benefit from further development (e.g., "moments of symmetry" and "synchronization barriers" as described by Avnur and Hellerstein [5]).

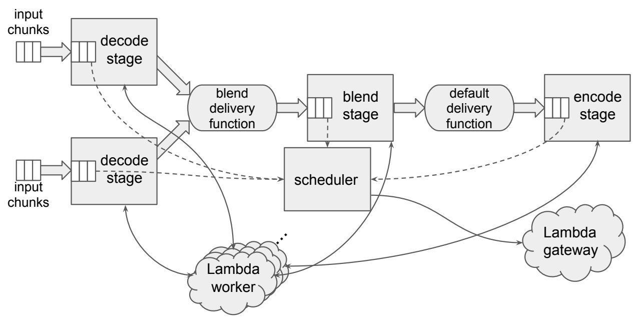


Figure 5: A detailed view of the Sprocket Coordinator for the blend pipeline

In a video pipeline, the most basic dependency is one-to-one: the output from an upstream worker is only processed by a single downstream worker, and the downstream worker only needs the output from a single upstream worker. An example of this dependency is a pipeline that takes a chunk of video as an input, decodes that chunk into a sequence of individual frames, applies a filter to that frame set (e.g., color modification), and then encodes the frames back into a compressed chunk. In this simple example, a downstream worker only needs one input event: once an input is ready, the task can be run immediately.

3.5.4 Customizing delivery functions. An example of a pipeline with complex dependencies is the blend application which superimposes the content of two videos.This blend application might serve as one component of a multi-step "green screen" application which replaces regions of an image of a certain color with a separate live video stream. This application includes two parallel decode stages, each of which converts the input video into individual frames. The next stage is a "blend" node, which needs to overlay each frame of one video with the corresponding frame from the other. An example of this pipeline can be seen in Figure 5.

For the blend application to work, it needs simultaneous access to corresponding frames. We define a *pair delivery function* that is used to capture this dependency. The "pair" delivery function ensures that frame number *i* from one video is delivered to the blend stage simultaneously with frame *i* from the other video. In other words, the delivery function has to wait for both upstream stages to have generated a set of continuous frames bearing the same presentation timestamps ("pts"), which are used to match up the frames from the two input sources. Once such a pairing occurs, the delivery function will deliver the merged input to the downstream worker as an atomic unit to the encode stage. The encode stage then gathers a set of frames before encoding them into a single compressed output video. A delivery function specific to the encoder ensures that it receives the correct number of frames and ensures that they are consecutive and in order.

Delivery functions can also be used to mitigate challenges of complex pipelines. For example, the Facial Recognition pipeline can incur new scheduling challenges based on the properties of the input data. Stragglers might arise, not just based on performance variation within the cloud platform (e.g., a slow Lambda worker), but rather as a data-dependent result of whether or not a given frame contains a face, as has been reported in other contexts [22]. Scenes with recognized faces must eventually go through the Draw stage, thereby unavoidably taking a longer time to complete. Customizing different delivery functions can help address the recognized face straggler problem.

One way to mitigate the recognized face stragglers is to create a delivery function that has the capability to split up the delivery of downstream events based on cut-off markers provided by the upstream stage. With this design, the FacialRecognition stage is able to request the next downstream stage, Draw, to receive only one frame per worker. By dedicating one Lambda worker per frame, the Draw stage completes almost instantly and takes minimal overhead compared to the rest of the pipeline. Other techniques for straggler [3, 36] and skew mitigation [1, 21, 23], drawn from the database and distributed systems literature, could be employed as well.

3.5.5 Partition function comparison . The delivery function serves a similar role as partition functions in Hadoop and Spark, which determine how the system re-distributes data from upstream to downstream elements, ensuring that the partitioning satisfies data dependency requirements, including avoiding skew in the workload distribution. However, the main difference between Sprocket delivery functions and these other partition functions is that, in Hadoop and Spark, they are serialization points in the pipeline: the respective worker will have to wait for all the input data to be ready to apply a partition function to further send the resulting data downstream. The delivery functions in Sprocket, however, deliver whatever work is available, which is particularly useful when processing video. As long as there exists *some* inputs to a stage that satisfy the data dependency requirements, the delivery function will ensure that the downstream worker is invoked with work to do.

3.6 Straggler mitigation

In a pipeline that consists of a large number of workers, stragglers are likely to happen. For example, I/O, CPU scheduling, hardware malfunction and non-deterministic programs can significantly slow down one or more workers. For video applications, it is common for a single worker to have a "wide dependency" of subsequent frames, i.e., many downstream tasks depending on that worker's output, thus stalling the entire pipeline. Such events can directly impact user experiences, such as forcing the user to wait for the streaming of a particular chunk.

Speculative execution [10, 36] is widely used to tackle stragglers. When the framework detects which workers are slower, it duplicates the worker's tasks to other nodes. However, by the time a straggler is detected and the speculative task is launched, it may already be too late to mitigate the straggler. To deal with this problem, proactively cloning tasks [2] can be used. However, this approach requires sending extra copies of input data to the workers and may potentially cause resource contention.

In Sprocket, we combine the two approaches, while avoiding the disadvantages of both, by exploiting the characteristics of input video chunks. Concretely, with straggler mitigation, we choose to use input video chunks that decode into a group of pictures (GOP) that is twice as long than the usual chosen length (e.g., a two-second chunk instead of the usual one-second chunk). Sprocket then sends both seconds of the chunk to the worker dedicated to processing the first second of the GOP, and the worker dedicated to processing the second second of the GOP. We call both workers a "pair". A worker first processes its assigned part of the GOP. After it finishes, it checks if the other worker in the pair is finished. If not, it continues to work on the other worker's part of the GOP, which in effect becomes speculative execution. Because the other worker's data is already there, speculative execution can be conducted efficiently.

Although workers in a pair have identical data, we are not doubling the total amount of data transferred. As seen in Figure 1, the second half of a GOP is smaller in size due to the absence of the key frame. We calculate that, in total, the extra data sent is less than a quarter of the original data.

3.7 Programming Sprocket applications

Sprocket programmers construct pipelines using a domain-specific pipeline specification language. In a pipeline specification ("pipespec"), a user first specifies the set of stages making up the pipeline, and then they specify the directed edges that connect those stages. Each edge must connect an upstream endpoint to a downstream endpoint, except for input/output endpoints that represent external sources and sinks of video to the pipeline.

The pipespec allows for individual stages to be parameterized with stage-specific parameters (e.g., target bit-rate parameter of an encoding stage), system-wide parameters (e.g., which Lambda function should be used to implement a stage), and additional data dependency information (Section 3.5.3). Similarly, edges can be parameterized to provide context and directives, e.g., how to convey data through the intermediate storage system to downstream stages or optimizations for interacting with stable storage such as S3.

Below is an example pipespec for a sample "blend" application, described in Section 3.5.4 which superimposes the content of two videos.

```
{ "nodes:
  [ {
      "name": "decode_0",
      "stage": "decode"
    }, {
      "name": "decode_1",
      "stage": "decode"
    }, {
      "name": "blend",
      "stage": "blend",
      "delivery_function":
        "pair_delivery_func"
    }, {
      "name": "encode",
      "stage": "encode"
    }
  ],
  "streams":
  [ {
      "src": "input_0:chunks",
      "dst": "decode_0:chunks"
    }, {
      "src": "input_1:chunks",
      "dst": "decode_1:chunks"
    }, {
      "src": "decode_0:frames",
      "dst": "blend:frames_0"
    }, {
      "src": "decode_1:frames",
      "dst": "blend:frames_1"
    }, {
      "src": "blend:frames",
      "dst": "encode:frames"
    }, {
      "src": "encode:chunks",
      "dst": "output_0:chunks"
    }
  ]
}
```

Listing 1: The Blend pipespec example. Note that input_0 and output_0 are bound to runtime parameters.

Edges include a source and a destination which specify upstream and downstream endpoints, respectively. An endpoint is defined by a stage name and an edge identifier within the stage, separated by a colon. In this way a stage can specify multiple streams that connect to it, providing broadcast/multicast semantics. The prefix of the edge identifier also implicitly indicates the data type of the stream: only matched data types can be sent through it, and the source and destination types have to match.

4 EVALUATION

In this section we evaluate Sprocket's behavior from various perspectives, including where it spends its time executing pipeline operations, how it exploits parallelism, how it meets streaming deadlines when generating output, the performance characteristics of a complex application, and how it compares to other general processing frameworks. We execute Sprocket using the Lambda service in AWS's North Virginia region (`us-east-1`). The Lambda instances use 3GB of memory, and the Coordinator runs on an AWS EC2 `c5.xlarge` instance. Table 1 shows the properties of the test videos we use.

In general, the results we report in this section use "warm" Lambda instances, which do not include the delays for creating a new container, initializing the runtime, etc. The benefits of warm Lambdas are well known in the community (e.g., [33]) because they reduce Lambda creation delays considerably: in our experience, the time to invoke 1,000 cold Lambdas is on the order of 6 seconds, while the time to invoke 1,000 warm Lambdas is an order of magnitude less at 600 ms. We report results with warm Lambdas because we expect them to be a reasonably common case; our experience is that warm Lambdas are cached for at least 15 mins, and often much longer (if a VM has at least one active instance, [33] reports 27 mins, and 1–3 hours).

4.1 Time breakdown

First we show where Sprocket workers spend their time when executing pipeline operations. Figure 6 shows the execution times for 734 workers performing a simple operation on the TEARS OF STEEL video (Table 1). Each worker operates on a one-second chunk of video, reading and decoding it in one stage, converting it from color to grayscale using `FFmpeg` in a second stage, and then encoding and writing it back to S3 as the final stage. Each bar in the graph shows the total execution time for the worker, and the bar breaks down the time into processing (encode, grayscale, decode), accessing video chunks from storage (each stage reads and writes to S3), and any waiting time.

For this pipeline, overall execution time is split roughly evenly as follows: across all of the workers, on average 34.7% of time is spent encoding and decoding, 30.4% is spent grayscaling, and 27.6% of

Name	Length	FPS	Content	Resolution
EARTH [11]	10 hours	25	Nature	1080p
SYNTHETIC EARTH	Variable	25	Uniform 1 second chunks from Earth	1080p
SINTEL [28]	14 min 48 sec	24	Animation/Action	1080p
TEARS OF STEEL [31]	12 min 14 sec	24	Action/Science Fiction	1080p
NATURE [26]	60 sec	25	Trailer/Contains no faces	720p
AVENGERS TRAILER [4]	35 sec	24	Trailer/Action/Contains faces	720p
INTERVIEW [8]	45 sec	24	Interview/Contains faces	720p

Table 1: Details of input videos used in the experiments.

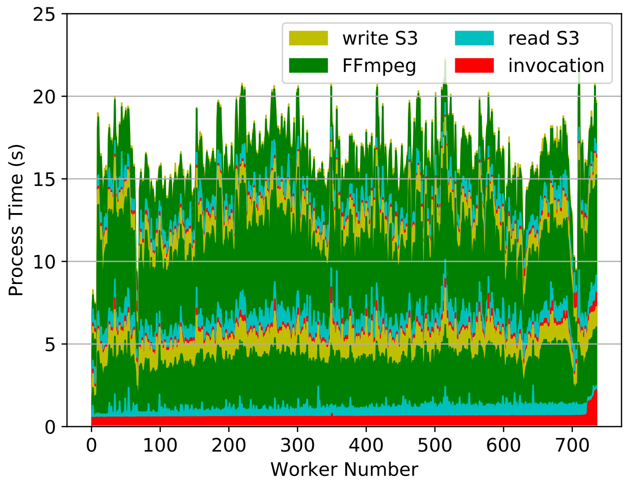


Figure 6: Execution time breakdown running the grayscale pipeline on the TEARS OF STEEL video.

time is spent reading and writing to S3, while the remaining 7.3% is spent waiting for the creation and initialization of the workers. There is much variation among workers, but the variation is primarily due to the nature of the video input: chunks of video that contain less data (are more compressible) take less time to process and read/write to S3.

4.2 Burst parallelism

A key benefit of using serverless cloud infrastructure is the opportunity for inexpensive burst parallelism. For short periods of time, users can inexpensively launch computationally intensive jobs that exploit large-scale parallelism. We show how Sprocket takes advantage of this opportunity with an experiment that processes jobs in constant time for inputs that do not exceed the parallelism limits of the underlying infrastructure. Figure 7 shows the completion times of Sprocket executing the grayscale pipeline as a function of video length for four videos. In this experiment, we scale the number of workers with the length of input video. Each worker processes one second of video, and the x-axis shows the length of input video and hence number of workers used (e.g., 600 workers process the first 600 seconds of video). We show curves up to 1,000 workers (the limit of parallelism that we reliably and consistently extract from AWS), unless the video length is less than 1000 seconds. We measure both the job completion time (total) and the time 99% tasks finish (99th percentile). Each point corresponds to the average of five runs, and the error bars show the minimum and maximum values.

For the SYNTHETIC video, in which every chunk has the same content, Sprocket achieves constant execution time until 800 workers, where times increase slightly due to slight resource contention on S3 accesses. For the EARTH video, the time increase from 200 to 400 workers is caused by data size increases in the video content, since the first few minutes of the EARTH video are highly compressible. In SINTEL, the increased processing time near the end of the video is caused by more data variance near the final stage of the movie. TEARS OF STEEL has more data variance among chunks than other videos throughout the movie, so the distance between total time and 99th percentile is larger.

4.3 Streaming

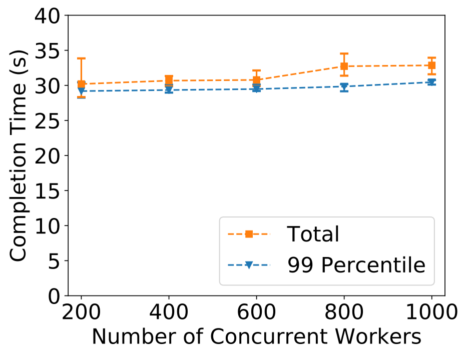
In previous experiments we used Sprocket as a batch system focusing on completion time for the entire video. But we designed Sprocket to operate equally well as a streaming system. In this mode, once Sprocket finishes processing the first result frame, users can start to stream the video. Recall from Section 3 that workers encode video chunks into standalone DASH segments. As a result, even if a video is hours in length it does not need to be serialized into a single final video file before viewing. Sprocket is also able to handle streaming input, thus being able to process input video that is most recent. Hence, streaming is seamless as long as Sprocket processes and delivers subsequent video chunks in time.

To demonstrate this behavior, we execute the grayscale pipeline on the first hour of the EARTH video. Figure 8 shows the completion time of each worker and its chunk of video for the variable amount of workers determined by the streaming scheduler (Section 3.5.2) for 1,000 workers and for 20 workers. The dashed line corresponds to the deadline that Sprocket needs to meet for seamless streaming. The line starts at the completion time of the first chunk since users have to wait for Sprocket to process it. But once started, Sprocket can easily meet the deadline for the remainder of the video.

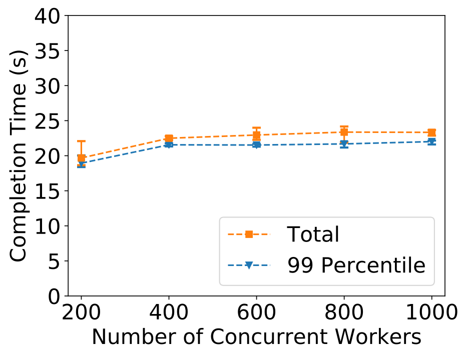
The inset graph zooms in to the first 100 seconds of video, clearly showing stairstep behavior for both the 20-worker and streaming scheduler configuration. Each step corresponds to one wave of n workers executing in parallel, and zooming into the graph shows the steady and stable performance over time as waves of workers process the video.

4.4 A complex pipeline

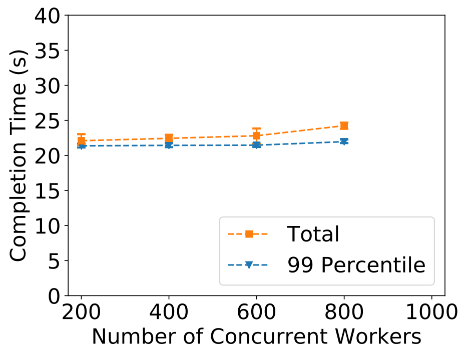
Sprocket's behavior is highly dependent on the properties of the input video. Straightforward filters or transformations of video chunks



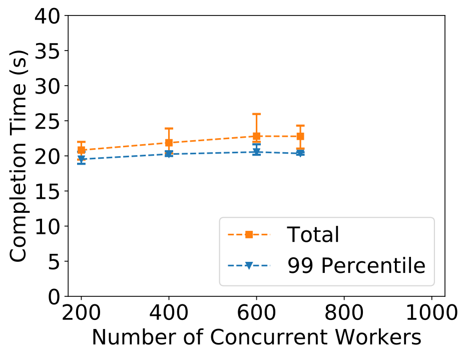
(a) Synthetic



(b) Earth



(c) Sintel



(d) Tears of Steel

Figure 7: Matching Lambda parallelism to video length.

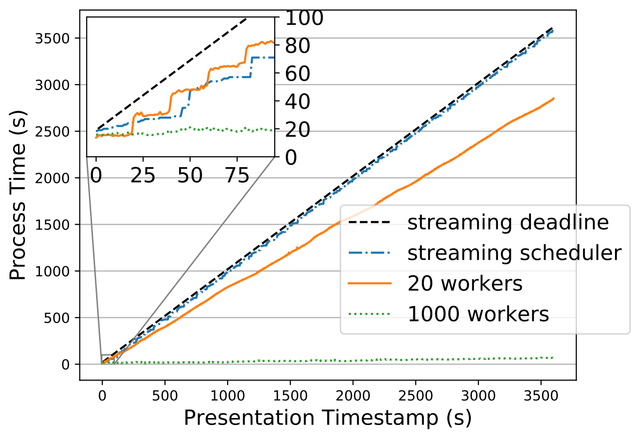


Figure 8: Streaming the EARTH video through the grayscale pipeline.

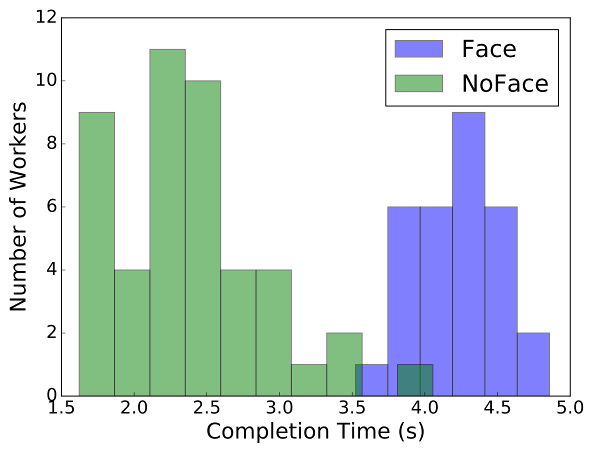


Figure 9: FacialRecognition stage behavior with and without faces.

perform the same work on each frame. The behavior of a more complex pipeline that recognizes and draws a box around a given actor's face, however, greatly depends on whether the input video contains a face, and whether that face is the one being queried. To demonstrate, we run the FacialRecognition pipeline on the NATURE and INTERVIEW videos (we use these short videos due to rate limits of Amazon's Rekognition API, as discussed in Section 3.3.2).

Figure 9 illustrates the bimodal execution times of the pipeline's Lambda workers in the FacialRecognition stage, depending upon the presence of a face in a given frame. Lambda workers processing the NATURE video, which has no faces, take between 1.5 and 4 seconds to complete the FacialRecognition stage. This execution time consists of the time it takes to invoke Amazon Rekognition to detect any faces present. Since no input frames contain faces, the stage passes along the frames immediately. For the INTERVIEW video, all the frames have faces, so the Lambda workers will experience increased execution times. When Rekognition does detect a face in a frame, it makes a second API call to compare the detected face with the target face. This second call adds another 1.5 to 3.5 seconds of execution

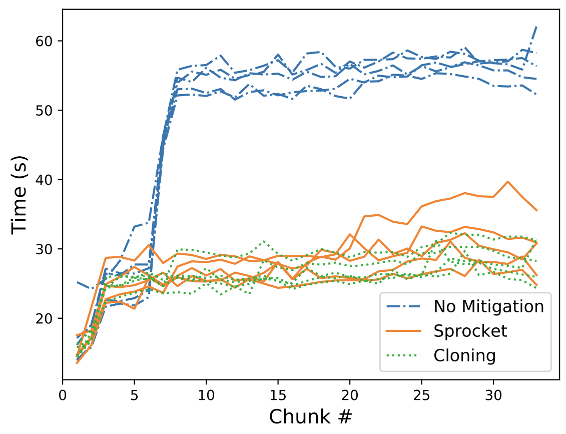


Figure 10: Comparison of chunk delivery time with various straggler mitigation strategies.

time, resulting in the bimodal execution times. For this experiment, the total execution time for the 60-second NATURE video with no faces was 32 seconds, and the execution time for the 45-second INTERVIEW video with faces was 45 seconds.

4.5 Straggler mitigation

We evaluate the straggler mitigation strategy in the decode stage of the FacialRecognition pipeline. In the FacialRecognition pipeline, there is a serialization point between two stages, causing the delay of a single chunk to delay all later chunks.

We measure the pipeline performance without straggler mitigation, with a simple cloning strategy, and with Sprocket's straggler mitigation strategy (Section 3.6) for comparison. The cloning strategy duplicates each task, and when a task finishes the decoding process, it sends a message to terminate the other task. Since it is difficult to create reproducable straggler situations when using AWS organically, we instead emulate stragglers by forcing a particular task to sleep for 30 seconds before decoding. As shown in Figure 10, if there is no straggler mitigation, a straggler in this pipeline can delay the delivery of many chunks. When using cloning or Sprocket's straggler mitigation, though, the delay is nearly reduced to the expected time to process a chunk.

Both task cloning and Sprocket's straggler mitigation removed the effects of the straggler. But they have different costs: task cloning performs much more redundant work than Sprocket's approach. Table 2 presents the total Lambda time in the decode stage under the different strategies, averaged across five runs.

As explained in Section 3.6, after a worker finishes processing its own part of the GOP, it continues to work on the other worker's part of the GOP if the pair's work is not finished. In the case that there is no straggler, this strategy only creates little extra Lambda running time, because the finish time for two workers in the same GOP is often very close. Once speculative execution finishes, the worker sends a notification to the straggler worker to terminate it immediately. In terms of cost, the slight difference in extra Lambda running time causes a negligible difference in extra costs. Although the cloning strategy duplicates each task, it does not double the total Lambda running time when there is no straggler — it costs only 41% more Lambda running time compared to no mitigation or Sprocket's approach.

	No Straggler (sec)	Straggler (sec)
No Mitigation	181.05±6.17	218.58±15.44
Cloning	259.54±8.23	252.17±3.58
Sprocket	184.25±5.75	191.77±6.02

Table 2: Total Lambda running time and standard deviation in the Decode stage under different straggler mitigation strategies. The higher Lambda run times for cloning results in higher costs compared with Sprocket's strategy.

The reason is twofold. First, once the faster task finishes the decoding task, it sends a stop message to the slower duplicated task, and the slower task can immediately stop what it has been doing and skip writing to S3. As a result, writing output data to S3 is never duplicated. Second, there is an inherit advantage of cloning — the faster decode time of the two tasks decides the average decode time. The cloning strategy also has a faster average task time, but it requires more concurrent Lambda workers, and thus more total Lambda running time. Sprocket's straggler mitigation is almost as effective as cloning the tasks, yet cloning costs 41% more Lambda running time than Sprocket when there is no straggler, and around 31% when there is one.

4.6 Alternatives

The rise in data processing requirements has led to the development of a number of parallel data processing frameworks. Tools such as MapReduce [10], Hadoop [15], and Spark [29] have become essential components for many organizations processing large volumes of semi-structured, primarily textual data. A number of distributed-computing frameworks, such as Dryad [18], Apache Tez [32], Apache Kafka [20], and Hyracks [6] further implement pipeline-oriented computation and support arbitrary DAG-structured computation, with data passed along edges in a computation graph. These environments are particularly beneficial when they can amortize their resource footprint over a large number of user requests, enabling increased resource efficiency and job throughput via optimized resource allocation, assignment, and scheduling decisions (e.g., [16, 34]).

Sprocket, however, focuses instead on individual users who want to submit a single job to the cloud, and thus cannot amortize the resource footprint of any acquired resources over anything else. We would like to enable these users to program their video processing application independently, without requiring major providers to implement their desired functionality. As a result, we focus on serverless cloud environments which are well-suited to this deployment scenario. The ability to have near-instant, extremely bursty parallelism on demand is a compelling alternative to requiring a traditional, dedicated cluster-based approach. The on-demand nature of cloud serverless infrastructure allows any user to run jobs consisting

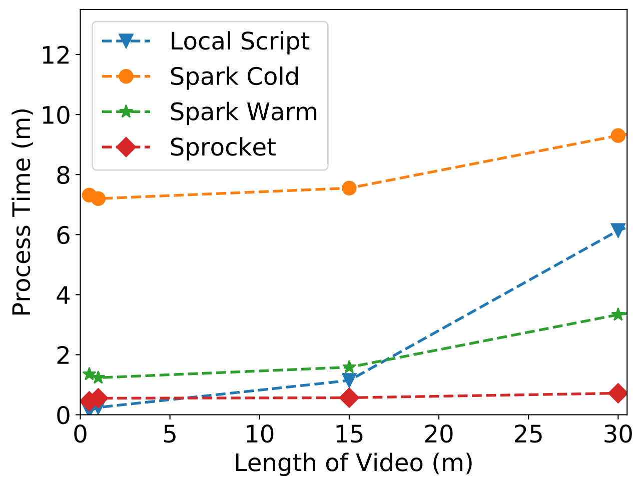


Figure 11: Comparing different video pipeline implementations.

of thousands of parallel executions for short periods of time at low cost, even for a single job.

Cloud providers do provide elastic offerings of more established parallel data processing systems such as Hadoop and Spark, but they are not a good match for Sprocket's goals. In terms of responsiveness, allocating and provisioning clusters potentially takes minutes before the new cluster can begin accepting jobs. Processing video with even simple transformations often does not involve any reduction in the data (input and output sizes are similar); efficiencies afforded by reductions in MapReduce-style computations do not apply to a wide range of video processing tasks.

To make this argument more concrete, we perform a simple experiment to illustrate the performance of a simple video processing application on Amazon's EMR Spark, an EC2 instance, and Sprocket. We use the EARTH video as input, segmented into two-second video chunks, and performed a simple grayscale operation using the `FFmpeg` tool in all frameworks. The Spark implementation used an 18-node cluster, with each node processing a partitioned set of video chunks into resulting mp4 output files. The EC2 implementation executed a batch script running 64 `FFmpeg` processes in parallel on an m4.16xlarge instance, which has 64 virtual cores and 256 GiB of memory. Sprocket used a filter pipeline (Section 3.2) executing a variable number of Amazon Lambdas, one per chunk, using up to 1,000 instances. Intermediate data was stored locally and the final output written to S3.

Figure 11 shows the execution time of the application on each platform as a function of video length. For Spark we show two lines, one including the time to provision and bootstrap the resources on AWS, and the other with just the application time after the cluster is ready. The Sprocket curve represents using either "cold" or "warm" Lambdas; the difference between the two is so small at these time scales that having separate curves would just overlap each other. Our goal is not to present this experiment as a "bakeoff" among the most optimized versions possible, but to illustrate the benefits of using serverless infrastructure for a single job. In particular, the startup time of provisioning cluster resources is significant in existing commercial offerings, which Sprocket avoids using the on-demand nature of Lambdas. (There are monetary advantages as well: the computation costs for a 30-minute video using the Local Script, Spark Warm, and Sprocket were \$2.38, \$1.42, \$0.63, respectively.)

5 CONCLUSION

Sprocket is a parallel data processing framework for video content that uses serverless cloud infrastructure to achieve low latency and high parallelism. Sprocket applications can range from traditional video processing tasks, such as filters and other transformations, to more advanced operations such as facial recognition. Applications consist of a familiar DAG in which vertices execute modular user-defined functions on video frames, and frame data flows along the edges connecting the outputs and inputs of vertices in the graph. Sprocket targets serverless cloud infrastructure such as Amazon Lambda as the execution environment. As a result, Sprocket does not require dedicated infrastructure, can take advantage of the massive parallelism supported by underlying container-based virtualization, and can launch applications on-demand with minimal startup delay.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedback and suggestions. We are also grateful to Cindy Moore for managing software and systems used in this project. Finally, we are particularly grateful to Elizabeth Farkas, Elizabeth Guerrero, Adrian Mendoza, and Zheng Tang. Enrolled in UCSD's Early Scholars Research Program, they were adventurous undergraduates willing to use a research prototype to implement their own video pipeline applications, a podcast outline generator and celebrity face detector. This work was supported in part by NSF grants CNS-1629973, CNS-1763260, CNS-1553490, and CNS-1564185, and by generous research, operational and/or in-kind support via the UCSD Center for Networked Systems (CNS).

REFERENCES

[1] ANANTHANARAYANAN, G., AGARWAL, S., KANDULA, S., GREENBERG, A., STOICA, I., HARLAN, D., AND HARRIS, E. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proceedings of the Sixth European Conference on Computer Systems (EuroSys)* (Salzburg, Austria, April 2011), ACM, pp. 287–300.
[2] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Lombard, IL, April 2013), USENIX Association, pp. 185–198.
[3] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (Vancouver, BC, Canada, October 2010), USENIX Association, pp. 265–278.
[4] Avengers Trailer. https://www.youtube.com/watch?v=eMobkagZu64.
[5] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (Dallas, TX, May 2000), pp. 261–272.
[6] BORKAR, V., CAREY, M., GROVER, R., ONOSE, N., AND VERNICA, R. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)* (Hanover, Germany, April 2011), pp. 1151–1162.
[7] Cisco Visual Networking Index: Forecast and Methodology, 2016–2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html.
[8] Colbert Interview. https://www.youtube.com/watch?v=Y6XXMGUb5kU.
[9] MPEG Dash Industry Forum. http://dashif.org/.

[10] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)* (San Francisco, CA, December 2004), USENIX Association, pp. 137–149.
[11] Earth. https://www.youtube.com/watch?v=wnhvanMdx4s.
[12] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, Mar. 2017).
[13] Google Cloud Functions. https://cloud.google.com/functions/.
[14] Google Cloud Vision API. https://cloud.google.com/vision/.
[15] Apache Hadoop. http://hadoop.apache.org/.
[16] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)* (Boston, MA, March 2011), USENIX Association, pp. 295–308.
[17] Huang, Q., Ang, P., Nykiel, T., Tverdokhlib, I., Yajurvedi, A., IV, P. D., Yan, X., Bykov, M., Liang, C., Talwar, M., Mathur, A., Kulkarni, S., Burke, M., and Lloyd, W. SVE: Distributed Video Processing at Facebook Scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (Shanghai, China, October 2017), ACM.
[18] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys)* (Lisbon, Portugal, 2007), ACM, pp. 59–72.
[19] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (Santa Clara, CA, September 2017), ACM, pp. 445–451.
[20] Apache Kafka. https://kafka.apache.org/.
[21] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)* (Indianapolis, Indiana, June 2010), ACM, pp. 75–86.
[22] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. A Study of Skew in MapReduce Applications. The 5th Open Cirrus Summit, 2011.
[23] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. SkewTune: Mitigating Skew in Mapreduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (Scottsdale, Arizona, May 2012), ACM, pp. 25–36.
[24] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment Vol. 5*, No. 8 (2012), 716–727.
[25] Microsoft Computer Vision and Cognitive Services API. https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/.
[26] Nature. https://www.youtube.com/watch?v=eMobkagZu64.
[27] AWS Rekognition. https://aws.amazon.com/rekognition/.
[28] Sintel. https://www.youtube.com/watch?v=qR5vOXbZsI4.
[29] Apache Spark. http://spark.apache.org/.
[30] AWS Step Functions. https://aws.amazon.com/step-functions/.
[31] Tears of Steel. https://www.youtube.com/watch?v=OHOpb2fS-cM.
[32] Apache Tez. https://tez.apache.org.
[33] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)* (Boston, MA, July 2018), USENIX Association, pp. 133–145.
[34] Apache Yarn. https://hortonworks.com/apache/yarn/.
[35] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI)* (San Jose, CA, April 2012), USENIX Association, pp. 2–2.
[36] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, December 2008), USENIX Association, pp. 29–42.
[37] Zhang, H., Ananthanarayanan, G., Bodik, P., Philipose, M., Bahl, P., and Freedman, M. J. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, March 2017), USENIX Association, pp. 377–392.