Machine Vision and Applications (2023) 34:42
https://doi.org/10.1007/s00138-023-01391-5

SPECIAL ISSUE PAPER

Video2Flink: real-time video partitioning in Apache Flink and the cloud

Dimitrios Kastrinakis[1] · Euripides G.M. Petrakis[1]

Received: 14 July 2022 / Revised: 2 January 2023 / Accepted: 5 March 2023 / Published online: 5 April 2023

Abstract
Video2Flink is a distributed highly scalable video processing system for bounded (i.e., stored) or unbounded (i.e., continuous) and real-time video streams with the same efficiency. It shows how complicated video processing tasks can be expressed and executed as pipelined data flows on Apache Flink, an open-source stream processing platform. Video2Flink uses Apache Kafka to facilitate the machine-to-machine (m2m) communication between the video production and the video processing system that runs on Apache Flink. Features that make the combination of Apache Kafka and Apache Flink a desirable solution to the problem of video processing are the ease of customization, portability, scalability, and fault tolerance. The application is deployed on a Flink cluster of worker machines that run on Kubernetes in the Google Cloud Platform. The experimental results support our claims of speed showing excellent speed-up results for all tested video resolutions. The highest (i.e., more than seven times) speed-up was observed with the videos of the highest resolutions and in real time.

Keywords Video processing · Video shot detection · Apache Flink · Apache Kafka · Kubernetes

1 Introduction

High-resolution video data are constantly being produced in a wide range of application fields ranging from surveillance and cinema to social media and streaming platforms of entertainment or news content. This generated the need to process video in real time (i.e., as quickly as it is produced). Extracting meaningful descriptions from videos is the first step toward content understanding as well as indexing and searching by content. The first step toward this process is the detection of consecutive frames showing continuous action in time and space, referred to as *shots*.

Processing in-stream video from online video sources in real time is still a research problem that merits further consideration. Compressed (e.g., MPEG-4) video is typically more complex to process especially when it is not converted to raw format first. On the other hand, processing raw video can provide more accurate content results but requires a great amount of memory space and significant bandwidth. Single frames of raw video can reach sizes of over 6 megabytes (for a typical 1920x1080 resolution colored video). Frames of this size are too big to process without splitting them up into smaller parts. In video production, multiple videos need to be processed simultaneously in real time. This might require the deployment of video processing solutions on large infrastructures such as the computer grid or the cloud. Minimizing the monetary and energy cost of a solution is a challenging problem.

Video2Flink handles video segmentation as a stream processing problem [13] to take advantage of the latest developments in data processing using stream processing and virtualized platforms (e.g., the cloud). A substantial advantage of the approach is that it makes the implementation of video processing independent of the hardware platform so that it can be easily replicated (i.e., ported) to servers or the cloud of different vendors. Video2Flink platform runs on a Kubernetes (K8s) [2,14] cluster in the Google Cloud Platform (GCP). Most cloud providers offer Kubernetes as a service to developers to support efficient application deployment, orchestration, and monitoring.

Apache Flink [6] is the state-of-the-art native real-time analytics engine for bounded and unbounded data streams in the database, data analytics, and machine learning

Dimitrios Kastrinakis and Euripides G.M. Petrakis these authors contributed equally to this work.

✉ Euripides G.M. Petrakis
petrakis@intelligence.tuc.gr

Dimitrios Kastrinakis
dkastrinakis1@tuc.gr

1 School of Electrical and Computer Engineering, Technical University of Crete (TUC), 73100 Chania, Crete, Greece

application domains. Video2Flink shows how complicated video processing tasks can be expressed and executed as pipelined fault-tolerant dataflows on Apache Flink. Overall, Video2Flink is a cost-effective, easy-to-implement system for processing high-resolution video in real time. The solution combines Apache Flink with Apache Kafka [18,25] a state-of-the-art publish-subscribe platform whose purpose is to facilitate machine-to-machine (m2m) communication between the video production and the video processing system that runs on Apache Flink.

The videos are generated at one end (the video producer) and are split into smaller blocks that are easier to transfer in a highly distributed system and process in parallel. The blocks are evenly distributed to multiple Apache Kafka partitions. An equal number of parallel Apache Flink pipelines consume that data simultaneously. They process each block by applying a series of transformations (i.e., Flink operators). The sum of these transformations leads to detecting camera shots (i.e., camera breaks and gradual transitions).

Scaling this system up is realized by simply increasing the number of partitions for the input data in an Apache Kafka topic as well as by configuring the Apache Flink application with higher parallelism. This allows Video2Flink to process multiple videos of different resolutions simultaneously at the frame rate of each video (i.e., 24 frames per second or faster). Additional desirable characteristics of Video2Flink are (a) *cost-effectiveness*: only the required computing nodes are utilized (i.e., in the cloud, the end-user is charged per use of computer resources); (b) *scalability*: Video2Flink can take advantage of auto-scaling features of Kubernetes [14] (i.e., the number of compute nodes is determined automatically); and (c) *portability*: Kubernetes enables deployment of the application on server infrastructures and in the cloud.

Related work is discussed in Sect. 2. Issues related to Video2Flink architecture design and implementation are discussed in Sects. 3 and 4, respectively, followed by an analysis of its performance in Sect. 5. Conclusions and issues for future work are discussed in Sect. 6.

2 Related work and background

Video segmentation is the process of partitioning video into meaningful sequences of frames based on similarity [23,26]. It is a fundamental problem in video content application domains that acquire, produce, or distribute video content such as robotics, surveillance, video conferencing, sports, and the Web. It plays an important role in video production for improving viewing experience and monetizing content [1] (e.g., for finding where to insert a break for advertisements, creating promotional videos using representative shots, and for creating video summaries). The use of summaries and keyframes reduces video data and enables video indexing, browsing, and retrieval in large video repositories of the video provider (e.g., in Video of Demand).

Video segmentation can be also formulated as a classification problem on pixels or objects. This is referred to as video object segmentation [16] (i.e., detecting frame sequences with an object of interest) and paves the way to semantic video segmentation [8], a far more complex problem that aims at the complete understanding of video content by detecting human-understandable concepts in scenes (e.g., an event or a person of interest). This, in turn, lays the foundation for video analytics [4] for mining valuable insights from video and exploiting this information to reach a business objective (e.g., increasing sales, car collision avoidance, etc.). The renaissance of machine learning and deep learning over the past few years has delivered a new generation of semantic video segmentation methodologies that promise good performance [17]. However, object or semantic video segmentation methods, in addition to being intrinsically and inextricably linked to video content, often suffer from reduced accuracy and non-real-time performance.

Histogram techniques for video segmentation are popular for being fast, reliable, and independent of the application domain [3]. A *camera break* indicates the simplest form of the boundary between two different shots. A more complex form of shot change can be a *gradual transition* such as a fade-in or fade-out, a dissolve, or a wipe. Camera movements (e.g., pan and zoom) can be considered a boundary between two shots. The textbook approach for detecting video shots relies on the comparison of intensity level histograms of adjacent frames. This comparison is largely unaffected by slow camera movements and general in-shot object motion since it ignores spatial changes inside a frame.

$$SD_i = \sum_{j=1}^{G} \|H_i(j) - H_{i+1}(j)\|. \tag{1}$$

SD_i is normalized by $M \times N$, the total number of pixels in the frame. If the sum of differences between two adjacent frames is higher than a given threshold T_b, a camera shot is detected. A good threshold T_b will increase the accuracy of the method. Low thresholds might allow many false positives, while an attempt to catch camera shots with a tighter threshold would be prone to missing some true transitions. T_b can be determined experimentally (i.e., by calculating all full-length histogram differences on the video) and by the formula $T_b = \mu + \alpha \cdot \sigma$, where μ and σ are the median and standard deviation, respectively, of the frame-to-frame differences and α is a constant. Experimental evidence [26] indicates that a good value for α is between 5 and 6. Under a Gaussian distribution, the probability that a frame difference that does not belong in a transition will be over this threshold is close to zero. For detecting gradual transitions, the *Twin-*

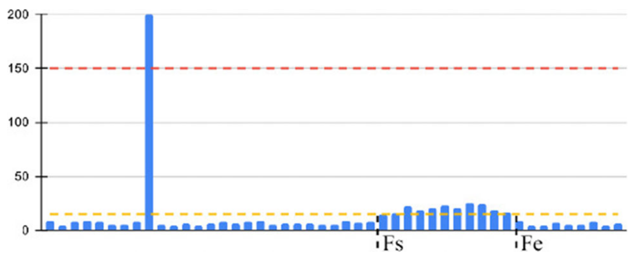


Fig. 1 A sequence of histogram differences with a camera cut, and a gradual transition between frames F_s and F_e

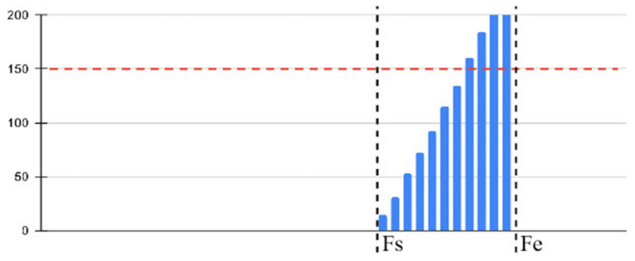


Fig. 2 Accumulated sum of differences surpassing T_a and T_b thresholds (red and yellow dashed lines)

Comparison method applies a second lower threshold T_a too. If a frame-to-frame difference, calculated with Eq. 1 is over T_a, the frame is marked and the next differences are continuously summed up. If the accumulated sum of differences is over T_b, then a gradual transition is found. The accumulation of the differences continues until a single frame-to-frame difference drops below the T_a threshold. Figure 1 illustrates the detection of both types of camera shots. Figure 2 shows the accumulated sum of histogram differences to detect a gradual transition.

The problem with video processing is the performance especially when video data become big, or acquired in real time (e.g., streaming video). Video processing on a single machine cannot be a real-time process. Several attempts (e.g., Parallel Horus [21]) take advantage of distributed and parallel processing infrastructures (e.g., a grid of machines) and reported significant speed-up compared to the processing of the entire video on a single machine. Video is split into smaller chunks, and video frames (one from each chunk) are processed individually on different machines. This allows a high degree of parallelization with excellent (i.e., many times) speed-up for batched video. The authors report 45:1 speed-up (compared to a sequential implementation) on a grid with 64 servers. The solution cannot work with online data sources (i.e., streaming video). Along the same lines, streaming video engine (SVE) [11] is a parallel video processing framework for Facebook. The videos are split into smaller chunks (i.e., 2 min at the most) that are processed separately on a large cluster of machines. The processing tasks include encoding, segmentation, and video track extraction. SVE parallelizes the storing of videos while processing. Although faults are inevitable due to the incomplete control of processing massive video data, SVE managed to improve system reliability with no impact on latency. SVE achieves a speed-up ranging from two times for short videos (i.e., up to 1MB size) up to nine times for large videos (i.e., 1GB size or more). SVE is designed for processing stored videos rather than video streams in real time as Video2Flink does.

SIAT [24] is a distributed video processing framework that provides multiple video analytics services such as video encoding, RGB to grayscale conversion, keyframe extraction shape extraction, compression, segmentation, and classification. It uses Apache Kafka to distribute video to multiple server machines on the cloud and can work with both streams and batches of data. To achieve high throughput, SIAT defines separate topics and multiple partitions in Apache Kafka for different processing and analysis tasks. The video processing layer is built on top of Apache Spark [22] in conjunction with OpenCV [19]. SIAT achieved up to 3.5 times speed-up. Similar to Video2Flink, SIAT is designed for processing video streams. Video2Flink achieved an even higher speed-up by decomposing the video into smaller chunks (i.e., pixel blocks) and by applying operations at an even lower (i.e., pixel) level than SIAT.

RIDE (2018) [12] is a real-time massive image processing platform on a distributed environment of server machines on the cloud to take advantage of both coarse and fine-grained parallelism (i.e., across multiple machines and multiple CPU or GPU cores). RIDE shares similar ideas with Video2Flink and exploits Apache Kafka for distributing video stream input from multiple sources (e.g., satellites, surveillance cameras) to multiple topics (i.e., one per source type). Multiple workers are assigned to process videos on multiple topic partitions of each video source (i.e., topic). The authors reported up to 5:1 speed-up on six servers (virtual machines) in the cloud. Similar to Video2Flink, video frames are split into blocks for finer parallel processing. To avoid communication overhead, all parts of the same frame are sent to the same worker. As a result, it is possible that some workers remain idle. Compared to Video2Flink, RIDE relies on a custom implementation of both hardware and software. As such, the solution is less principled and hardly replicable. Instead, Video2Flink is replicable and portable to servers or the cloud running Apache Flink (a popular and open-source stream processing platform).

Table 1 summarizes all methods reviewed above and their comparison with Video2Flink. Video2Flink is the only method to achieve high speed-up for real-time (stream) video processing or on non-dedicated hardware infrastructures. Parallel Horus or SVE also reported a significant speed-up on custom software and hardware infrastructures and for stored video only.

Table 1 Video processing systems: comparison based on system features

Properties system	Application	Processing unit	Hardware platform	Software platform	Type of processing	Maximum speed-up
Horus	Object detection	Image frames	Grid	Custom	Batch	45:1 (64)
SVE	Video segmentation	Video segments	Grid	Custom	Batch	9:1 (N/A)
RIDE	Video analytics	Image blocks	Cloud	Kafka, Hadoop	Real time	5:1 (6)
SIAT	Video analytics	Image frames	Cloud	Kafka, Spark, Hadoop	Real time	3.5:1 (N/A)
Video2Flink	Video segmentation	Image blocks	Cloud (K8s)	Kafka, Flink	Real time	7.5:1 (8)

3 Video2Flink architecture

Apache Flink provides a toolbox of operators for implementing transformations on data streams (e.g., filtering, updating state, aggregating). The data flows or *Jobs* (i.e., operations chained together) form directed graphs (*Job Graphs*) that start with one or more sources and end at one or more sinks. The Flink cluster consists of a *Job Manager* and a number of *Task Managers* (workers). The Job Manager controls the operation of the entire cluster: schedules the workers, reacts to finished or failed tasks, load balances the workload among Task Managers, and coordinates checkpoints and recovery from failures. The Task Managers are the machines (servers) that execute the tasks of a workflow. A task represents a chain of one or more operators that can be executed in a single thread or server. A task can be executed in parallel (on separate Task Managers). Each parallel instance of a task is a subtask. The number of subtasks running in parallel is the parallelism of that particular task.

Apache Flink receives data records (or events) from streaming sources such as Apache Kafka? [18,25]. Apache Kafka is an open-source platform that reads data streams in parallel from application sources such as databases, sensors, mobile devices, and cloud services. The workload represents the number of records per second the system receives. Features that make Apache Kafka popular are its customization, portability, scalability, fault tolerance, and security. It applies a custom binary protocol based on TCP and employs a cluster of brokers in order to maintain good performance. The messages are categorized into topics, and for each topic, the messages are published to disk partitions. Each publisher specifies the topic and the partitions to publish and writes the messages to the partitions in a round-robin fashion. Each new message is written (as a byte array) at the end of a partition. As the message load increases, the brokers, the storage as well as the number of subscribers have to increase too to prevent bottlenecks. If the number of brokers is not sufficient to handle the message load, messages may be lost (the producer requests will time-out). Throughput increases with the number of partitions and with the number of consumers that read messages in parallel. The number of consumers reading messages from a topic cannot be greater than the number of partitions.

The system comprises the *Clients*, the *Kafka broker*, and the *Flink cluster*. Each client reads a video and extracts its frames. The input video of Fig. 3 is split into frames, and each frame is split into smaller parts (or blocks). Each block is uniquely identified by a *key*. Each key–block pair forms a message. The messages of a frame are evenly distributed to the Kafka partitions. Each video producer (application) specifies the partitions to publish and writes the messages to these partitions in round-robin. Throughput increases with the number of partitions and with the number of Flink (com-

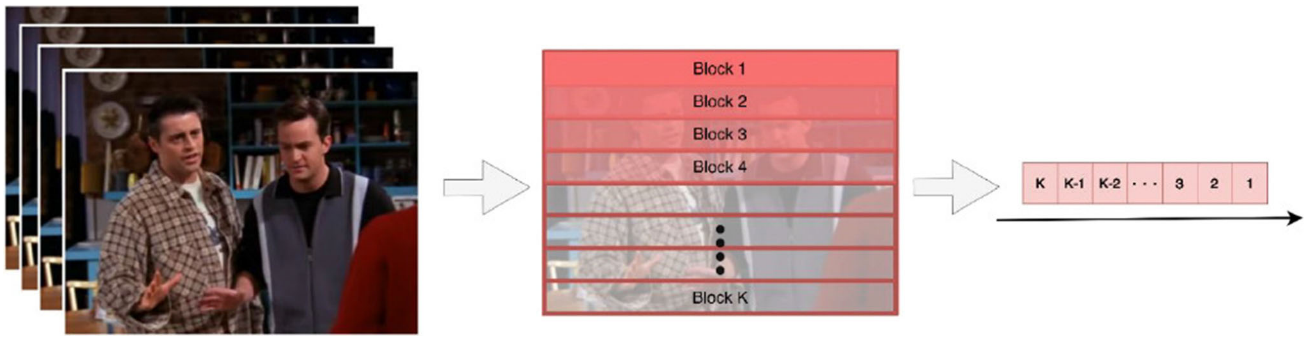


Fig. 3 Video splitting into frames and into a sequence of blocks

pute) nodes processing messages. The number of Flink nodes equals the number N of Kafka partitions.

Figure 4 illustrates three clients (on the left) that send video to N Apache Kafka partitions. In Apache Flink, the messages can be categorized into topics. In Video2Flink, there is one input topic for all video producers. This topic can accept more than one video (of any resolution and frame rate) in parallel. On the right of Fig. 4, N parallel Flink pipelines request data from the Kafka broker. Each pipeline reads from one partition of the input topic. If any shot is detected, it is announced at the output topic of the Kafka producer. The output topic has one partition. The Flink cluster comprises a number of Flink nodes that apply a sequence of five operations (in a pipeline) that read messages from Kafka partitions.

Data can move between worker machines. This adds additional flexibility in defining complex sequences of operators to accomplish even complicated processing tasks. The first operator (A) transforms these pixel data to grayscale if encoded in RGB. If the blocks are in grayscale, the operator is skipped (i.e., forward the grayscale pixel data to the next operator). The next operator (B) generates the histogram of the intensity of each grayscale block it receives. Next, all the histograms generated from all the blocks of a single frame are forwarded to a single operator (C). There, the block histograms of each frame are all summed together to produce the total histogram of that frame. The next operator (D) computes the difference between the histograms of adjacent frames. The last operator (E) receives checks for differences that surpass threshold T_b. In that case, a camera break is detected. This operator also searches for potential gradual transitions. This requires that the differences are kept in the memory; all the differences of adjacent frames exceeding threshold T_s are accumulated and their sum is checked against threshold T_b. In that case, a shot is detected as a result of a gradual change.

Key generation Each block is assigned a key. Figure 5 is an example of a block key. It contains the unique identifier of the video (a and b). It also contains the identifier of the block itself (h) and the frame it came from (g). Component (a) is the time when the video started being read by the client (in milliseconds since the Unix epoch); component (b) is a random integer between 0 and 10,000. Together, (a) and (b) compose a unique identifier for each video. Component (c) is the resolution and (d) is the total number of frames. Block size is defined in (e). It indicates how many rows each block contains. If the frame's height is known, the number of blocks per frame can be calculated. Part (f) of the key represents the encoding of the pixel data; 0 is for grayscale and 1 for RGB color. Based on keys, Flink operations know exactly how to distribute the blocks among operations in different pipelines which is essential for accomplishing the required processing.

Video format Video2Flink can process video in grayscale or in a color format using either raw RGB (RGB888) or YUV (YUV420p) encoding. In RGB888, there are 24 bits per pixel (8 bits per color channel). The first Flink operator will transform it to grayscale data (8 bits per pixel) using the formula

$$Y' = 0.299R + 0.587G + 0.114B, \tag{2}$$

where R, G, and B are the original red, green, and blue image components, respectively, and Y' is the gamma-corrected light-intensity (i.e., grayscale) value that takes into account the gamma compression that occurs when creating a color image, and the increased sensitivity toward lower luminance. In the YUV420p encoding, the Y component of each frame is the grayscale frame. The client reads the grayscale frame and sends it to Kafka. Each frame's U and V components are skipped from the file stream. YUV encoding saves bandwidth and processing time on the Flink cluster since no grayscale conversion is necessary.

Block size Splitting a video frame into multiple smaller parts allows to parallelize the processing of a single frame on multiple worker nodes. Smaller blocks of data obviously require less processing power to be processed, and this, in turn, allows the use of multiple weaker worker nodes instead of fewer and larger ones. If the Flink cluster uses more powerful nodes, a larger size for each block can be set. Alternatively, with weaker nodes, a much smaller size can be configured. This allows using the Flink cluster and its resources opti-

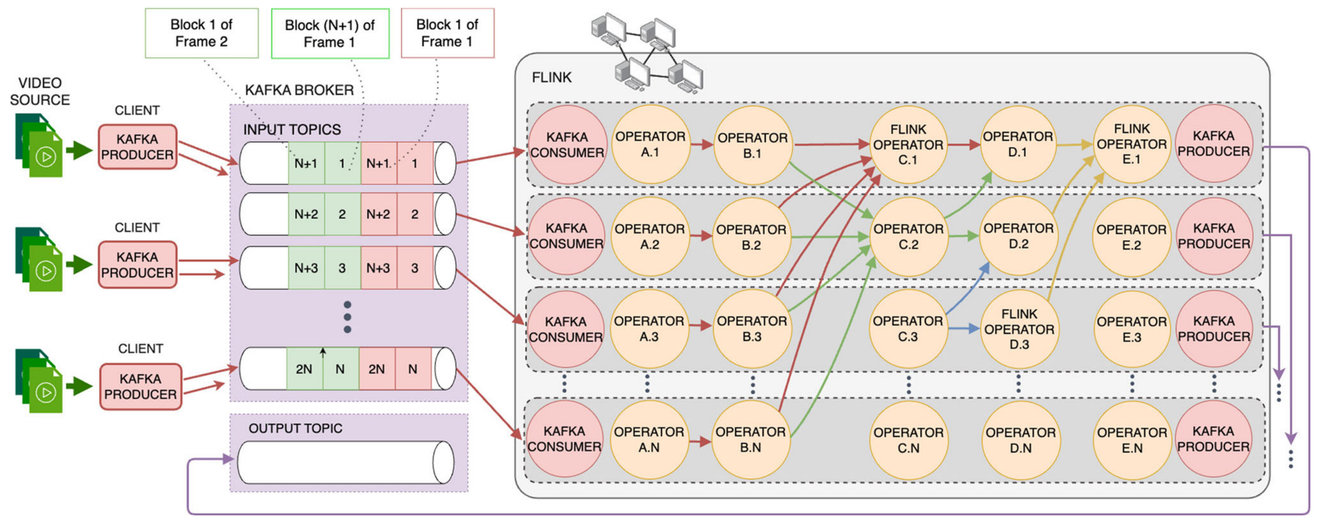


Fig. 4 Video2Flink abstract architecture

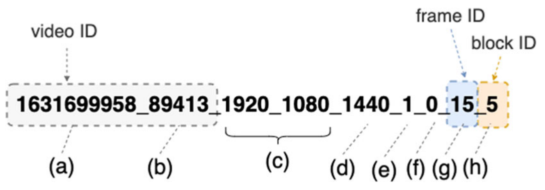


Fig. 5 Example block key

mally. If Video2Flink accepts different videos in the input and each block has the size of a single row, then Video2Flink can run with different block sizes (e.g., for different video resolutions). For a 2560x1440 resolution color frame (24 bits per pixel, 8 bits per color channel), there will be 1,440 blocks, 7.5KB each. For a grayscale frame, the size of a block is exactly one-third of the previous value (i.e., 2.5KB). However, the size of a block need not be equal to the size of a row. The configurability of the block size is an advantage on its own.

Acknowledgment of receipt Apache Kafka is responsible for providing Apache Flink with data. Each block can be sent at most once no matter whether it is received or not by Flink or can be sent at least once provided that an acknowledgment of receipt has been received by the publisher (i.e., if a message is not received it is re-routed to be resent). Apache Kafka supports three levels of acknowledgment. Video2Flink applies to the lowest level (i.e., a producer does not wait for any acknowledgment). The messages are sent at most once, asynchronously, and are immediately considered sent. This improves the latency (and the throughput), but the producers will not be notified if a message is not received. As will be shown in the experiments, even with very high input throughput, all messages are received.

Flink pipeline The pipeline can process both streams and batches of raw video data. Multiple videos can be processed concurrently. They can have different resolutions, encodings, or even block sizes. The distinction between them is achieved through the unique key that the clients generate for each block. Each frame is split into K blocks which are evenly distributed in the N partitions of the input topic.

Video2Flink defines five operators in each pipeline denoted by the letter A, B, C, D, and E. A pipeline is a single set of instances of different operators (e.g., $A.1 \rightarrow B.1 \rightarrow C.1 \rightarrow D.1 \rightarrow E.1$, or $A.1 \rightarrow B.1 \rightarrow C.3 \rightarrow D.2 \rightarrow E.1$ in Fig. 4). A Video2Flink application can run with only one pipeline or multiple. Each operator either processes its input data immediately (e.g., operator A and B) or waits for multiple input data (e.g., operators C, D, E). Some operators make use of a local on-memory *keyed state*. Each operator A is coupled with a Kafka consumer that consumes (i.e., processes) messages from a partition of the input topic. Likewise, each operator E is coupled with a Kafka producer that produces the application's output messages to the output topic. The arrows between the operators indicate the redistribution of data among operations and pipelines during the processing.

There are N instances of each operator working in parallel (i.e., the parallelism is N). The optimal parallelism of the system is estimated experimentally or can be determined automatically by an auto-scaler agent [9].

Keyed state A normal stream of data (e.g., the output of all parallel instances of Operator B) can be transformed into a keyed stream based on a key (e.g., each frame's unique identifier). This is used to group the related results of all the previous operator instances (e.g., all partial histograms of a frame). Each group of related results will be processed together on the same instance of the next operator. For each specific key, a state is created. The state of each key is stored in a single operator instance. All output of the previous operator with the same key will be redirected to that same operator instance. Figure 6 illustrates an example data flow between operator B and operator C. Operator B outputs partial histograms. Operator C receives the partial histograms of each frame and outputs the full histogram. The partial histograms of frame 1 of video 1 (denoted as *Frame1_Video1*) are all sent to the instance $C.1$ of operator C. There, the partial histograms are summed up and sent to the next operator. In order to sum the partial histograms that will arrive at different times, that key's associated state is used to store them locally.

4 Video2Flink operators

Video2Flink defines five operators plus a Source operator (Kafka consumer) and a Sink operator (Kafka producer). Each operator applies a single transformation and has N instances that work in parallel. The number of partitions for the input Kafka topic is always set to the same value.

Source operator It is the host of a Kafka consumer that requests messages from the input topic. Each consumer is automatically assigned a specific partition. No other consumer can use the same partition. The messages from the input topic contain the block pixel data and that block's unique key. The different key components are parsed in order to be used by the operators. The key and raw pixel data are sent to operator A.

Operator A: RGB to grayscale transformation If the input block is in color (24 bits per pixel), it will be converted to grayscale (8 bits per pixel). Otherwise, it is simply forwarded to the next operator unchanged. Figure 7 illustrates this process. The operator identifies if it's in color or in grayscale by the block's key.

If K is the number of blocks of a frame and N is the parallelism of the operator, then each instance of operator A will process K/N blocks per frame. Each block that arrives in the Kafka consumer from the input topic is sent to the pipeline of operator A. Each instance of operator A converts each input block to grayscale (if needed) ignoring which frame or video each block it comes from. Figure 8 details this process.

Operator B: block histogram computation This operator receives blocks with grayscale pixel data. For each input block, its intensity histogram is computed. As shown in Fig. 8, the output of this operator is redistributed across operators C based on the identifier of the source frame (i.e., in the key of each block) so that each operator C receives data (i.e., block histograms) of the same frame. Whenever a new partial histogram arrives, the operator uses its corresponding state (e.g., the state assigned to key *Frame10_Video3*) to identify the video and the frame it comes from. Figure 9 illustrates this process.

In Fig. 8, four partial histograms of the same frame are routed from operator B (i.e., from all instances of operator B) to operator $C.1$ Similarly, a different set for four partial histograms of another frame (from the same or from a different video) are routed to operator $C.2$. Similar to operator A, each instance of operator B will process K/N blocks of data, per frame for each video and will output the same number of partial histograms. Operator A, operator B, and the source Kafka consumer are chained together and are co-located in the same thread. No data redistribution between these remote operators occurs. This increases the performance and latency of these operators, by reducing the communication overhead to the next operators.

Operator C: full-frame histogram computation The operator receives the histograms of all the blocks of a frame. The operator sums up all the partial histograms and generates the total histogram of that frame. If a frame is split into K blocks, then the operator will wait for K blocks and then will output the histogram. The operation times out after a few seconds and the histogram is computed with the partial histograms received up to this time. Multiple frames can be processed on the same operator concurrently, even from different videos. Each output (i.e., full histogram of a frame) is sent out twice, each time with a different key for adjacent frames in order to compute the difference between any two consecutive frames at the next operator D.

In Fig 12, each instance of operator B outputs K/N partial histograms (only 4 shown). All partial histograms of frame 1 from video 1 (with key $Frame1\_Video1$) are routed to operator $C.1$ (i.e., the first instance of operator C). Similarly, all partial histograms of each frame are routed to an instance of operator C. The same operator instance will output that specific frame's full histogram. The routing of the keyed data to the correct operator instances is handled by Flink.

Operator D: histogram difference computation The operator receives the histogram of two adjacent frames and computes their difference. A key is used for every two sequential frames. For example, for four example frames, $F1$, $F2$, $F3$, $F4$, three keys are created: $F1\_F2$, $F2\_F3$ and $F3\_F4$. For each key, a unique state is created and stored on an operator instance. To distinguish between frames of different videos, the identifier of each video is also added to each key. Figure 11 visualizes this process.

Each full-frame histogram is routed twice by the previous operator C. For example, operator C sends the histogram of frame $F3$ twice, once with key $F2\_F3$ and a second time with key $F3\_F4$. Figure 12 illustrates the distribution of histograms from operator C to operator D. The histogram of frame $F3$ with key $F2\_F3$ will be compared against the histogram of frame $F2$ (with the same key) on operator instance $D.2$ (i.e., the second instance of operator D). That operator instance will output their difference. Likewise, in operator $D.3$, the difference between frame $F3$ and $F4$ will be calculated. Each parallel operator instance will hold data of multiple keys (keyed states). For each key, it waits for the histograms of two adjacent frames to arrive. When the first

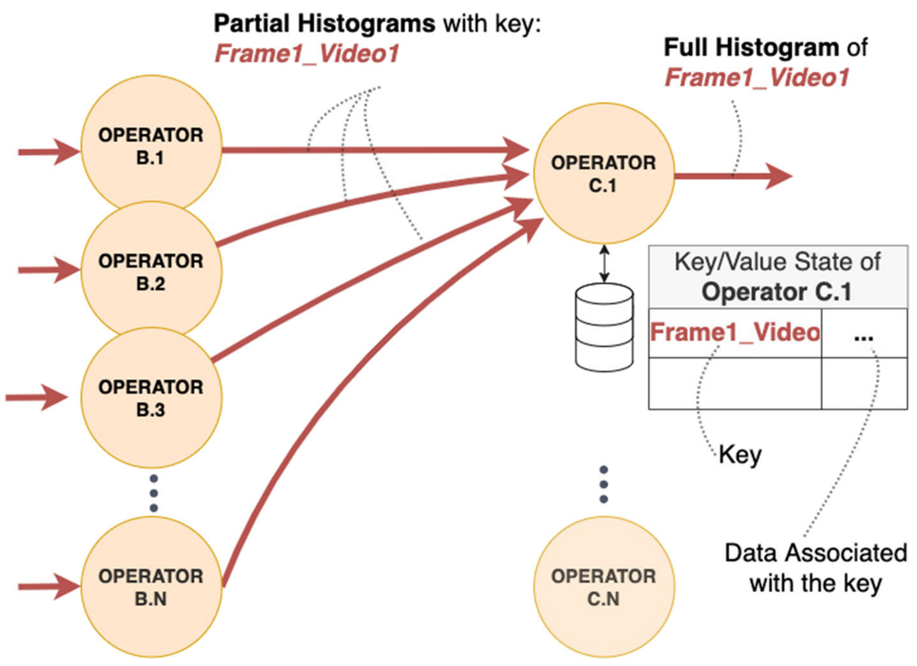
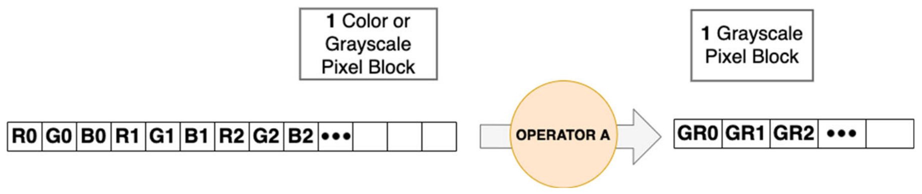


Fig. 6 Example data flow with local key-value state



1 Color or Grayscale Pixel Block
1 Grayscale Pixel Block
R0 G0 B0 R1 G1 B1 R2 G2 B2 •••
OPERATOR A
GR0 GR1 GR2 •••

Fig. 7 Operator *A*

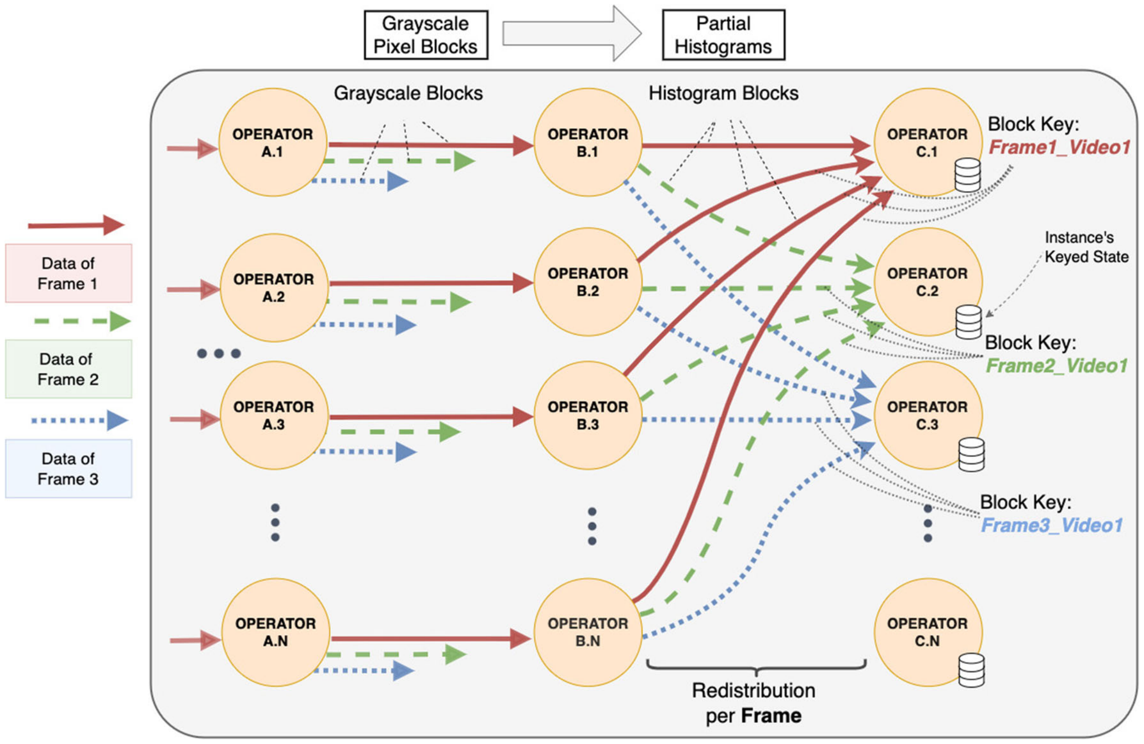


Fig. 8 Graph of operators *A* and *B*

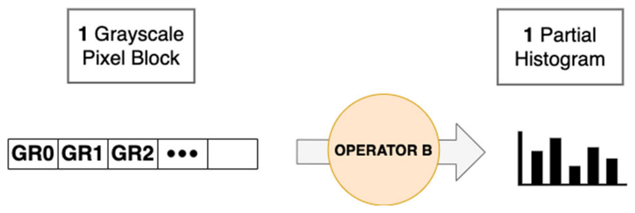


Fig. 9 Operator *B*

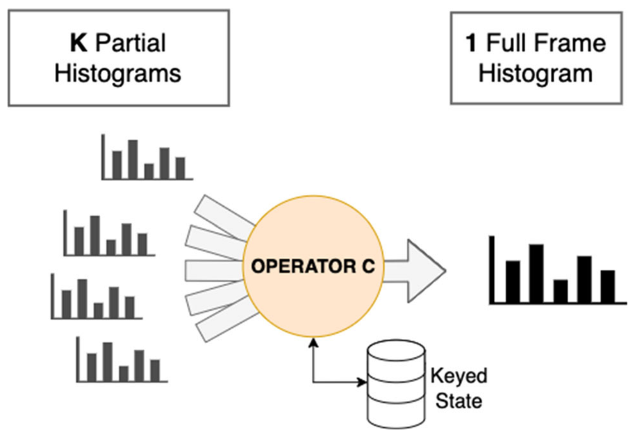


Fig. 10 Operator *C*

histogram arrives, it saves it in the key's local state. When the second histogram arrives, it calculates their difference and outputs the result.

Operator E: shot detection The operator receives all histogram differences between all adjacent frames and outputs shot changes (camera cuts and gradual fades) to the output topic. A key is also used on this operator, to distinguish between different videos being processed. The histogram differences do not come in order. The histogram differences are grouped based on the identifier of the video they come from (e.g., $Video1$) by assigning them the same key. All histogram differences with the same key are routed to the same operator instance. The histogram differences from $Video1$ (black arrows) are all sent to instance $E.1$. Likewise, all histogram differences from $Video2$ are sent to instance $E.2$ (purple arrows). In each instance, the histogram differences of the same video share a local state. Figure 13 provides a detailed view of the operator.

Each instance of operator E checks for differences that surpass threshold T_b. In that case, a camera break is detected. It also searches for potential gradual transitions. This requires that the differences are kept in the memory; all the differences of adjacent frames exceeding threshold T_s are accumulated and their sum is checked against threshold T_b. In that case, a shot is detected as a result of a gradual change. Any shot detection messages from that operator instances are sent to the Kafka output topic.

The histogram differences are generated without taking into account the order of the frames. A priority queue on each instance of E operator holds all the histogram differences of each video that has arrived. The top element of the priority queue is always the earliest available histogram difference. Whenever the next histogram difference is required (by the shot detection algorithm), the operator first checks the top element of the queue. If found, it continues with the shot detection algorithm. Otherwise, it waits. In the meanwhile, the arriving elements are inserted into the queue. Figure 14 illustrates the priority queue on operator $E.1$.

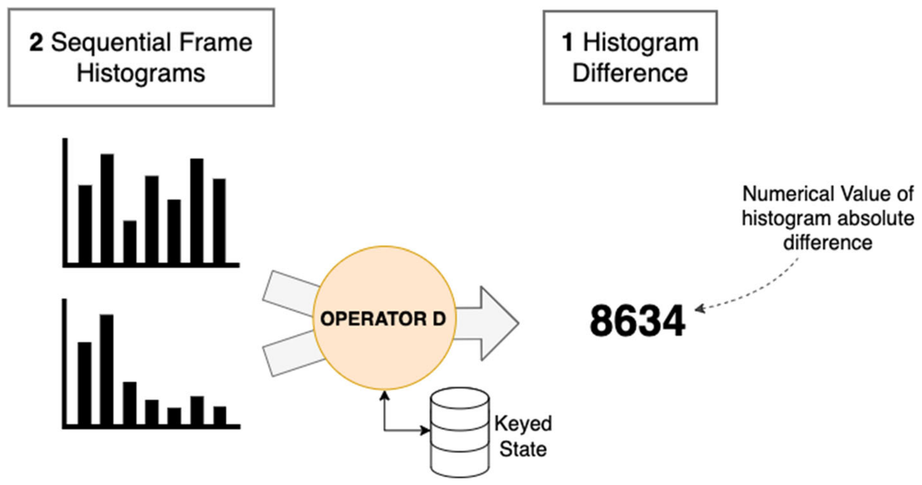
Sink operator The results of the above sequence of operations are aggregated and reported at the output for each video separately. Figure 15 illustrates the mapping or Flink operator instances to N Flink nodes (i.e., parallelism = N). Each node is the host of 5 operator instances. Operator instances chained together are mapped to the same thread. For the sake of simplicity, each node in Fig. 15 has four threads.

5 Experiments

All methods referred to in Sect. 2 (related work) use dedicated or proprietary platforms (e.g., a grid) that cannot be replicated in the Google Cloud Platform (GCP) of our experiments nor they are available to us. Our main concern is to demonstrate the efficacy of Video2Flink based on comparison results reported in the literature (Table 1). The purpose of the following set of experiments is twofold: (a) to confirm that the Video2Flink shot detection method works as intended (i.e., detects all shots correctly without missing any) and (b) to prove that Video2Flink achieves a significant speed-up compared to its sequential counterpart (i.e., running on a single worker machine).

Video2Flink was deployed on two virtual machines (VM) in the Google Cloud Platform. Each VM runs Ubuntu 16.04. Apache Kafka runs in a VM with 8 virtual CPUs (vCPUs) of 3.1GHz base frequency (3.8 turbo frequency) and 32GB RAM. For storage, a high-performance HDD was used with a theoretical maximum of above 700MB/s for sequential reads/writes. The VM was used to run a number of clients (i.e., video producers) while hosting Apache Kafka with multiple topic partitions. Both processes require significant I/O performance (i.e., the data is all in one place and is read in order) and multiple cores for the parallel workload and multiple I/O threads. The eight vCPUs and especially the high-performance HDD are crucial (i.e., a bottleneck in this part of the system would significantly constrain the performance of Video2Flink).

Fig. 11 Operator D


2 Sequential Frame Histograms
1 Histogram Difference
Numerical Value of histogram absolute difference
OPERATOR D
8634
Keyed State

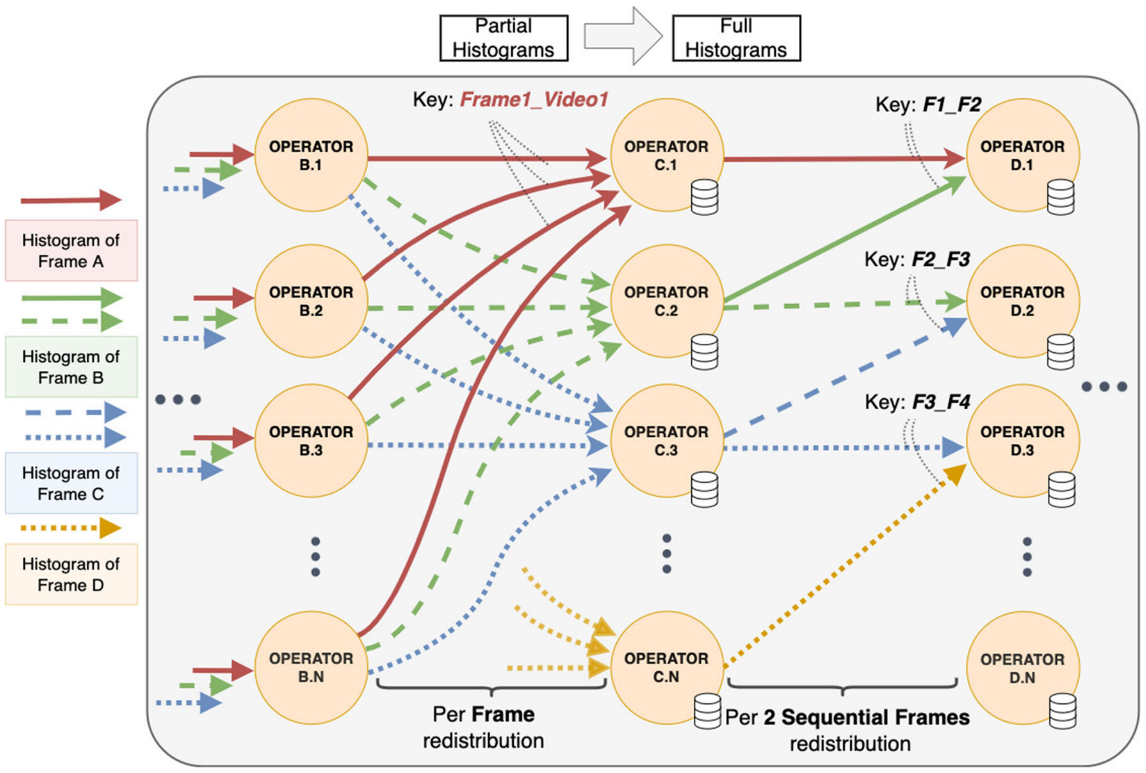


Fig. 12 Distribution of frame histograms from operator C to operator D

Apache Flink runs on Kubernetes [1] on the second VM. The Kubernetes cluster has a pool with eight nodes. Each node has two vCPUs (2.0 GHz base frequency and 2.8 GHz turbo frequency) and 8GB of RAM. For storage, the minimum available option of 30GB was used, since Apache Flink required no local storage. Each node runs on a container-optimized OS with containers provided by Google. To initiate a Flink Job, the job is submitted to the Flink cluster along with the desired number of parallel Flink pipelines (to run on Flink nodes).

The correctness of the shot detection method is verified using two videos from a TV show, referred to as *Video A* and *Video B*. The duration of each video is 60 s with 24 frames per second. Four copies are created from each video with four different resolutions (i.e., 960x540, 1280x720, 1920x1080, 2560x1440). The two threshold values T_b and T_S of the twin comparison shot detection algorithm was computed in advance (i.e., offline). To determine the actual number of shots, the shot detection algorithm was run as a monolithic

[1] https://nightlies.apache.org/flink/flink-docs-release-1.13/docs/deployment/resource-providers/native_kubernetes/

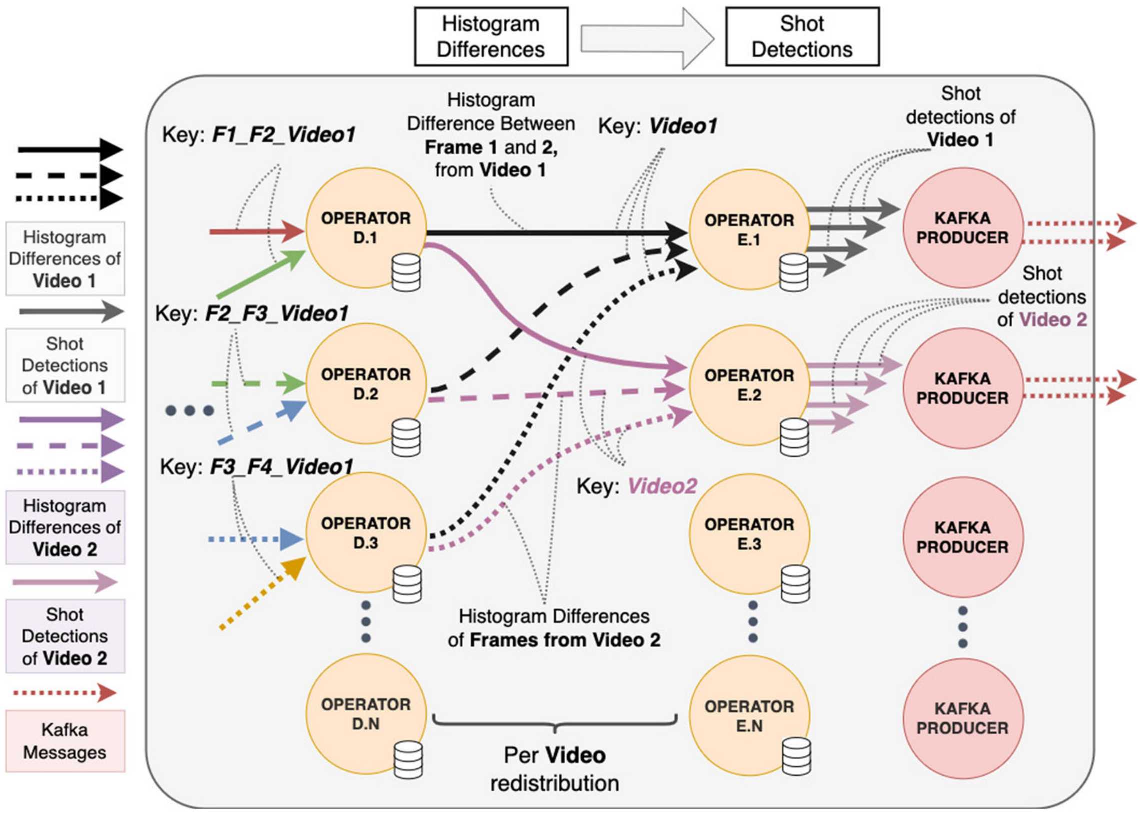


Fig. 13 Histogram differences with the same key are routed from D to the same instance of E operator

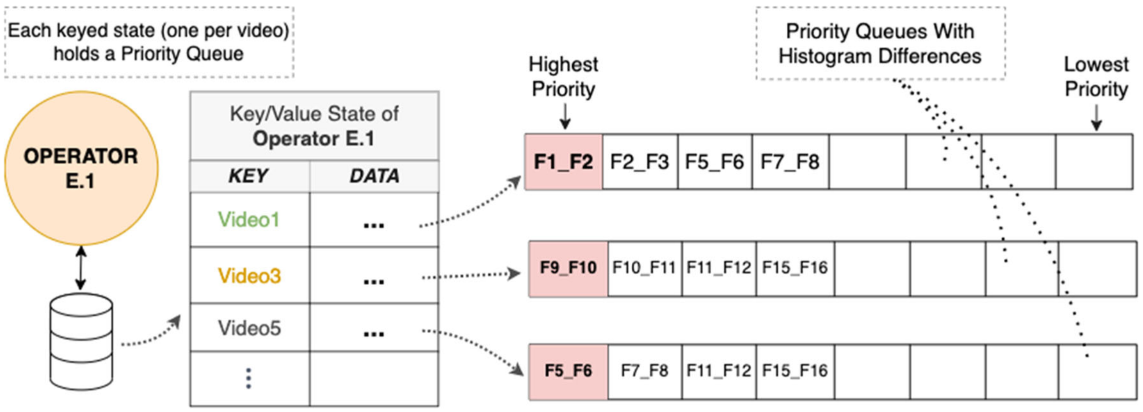


Fig. 14 A priority queue holding the histogram differences that have arrived on operator $E1$

Java program on each video. The parallelization was set to four pipelines (i.e., four parallel instances of each operator) which is also the number of Apache Kafka partitions. The detected shots were displayed per type of shot (i.e., camera cuts or gradual fades). Video2Flink gave the same results for all the different resolutions and for both videos (i.e., 14 shots for *Video A* and 7 for *Video B*). The results were also visually inspected for correctness. This verified that no false shots were detected without missing any.

5.1 Speed-up

The goal of this experiment is to study the latency and the speed-up of Video2Flink as a function of the input throughput. Latency is the delay (in seconds) between the time a video is routed to Apache Kafka until shot detection is complete. Throughput is the amount of video data (in MB/second) that is transferred and processed. Latency and throughput are correlated (i.e., latency declines with throughput). The input throughput can derive from any number of clients with videos

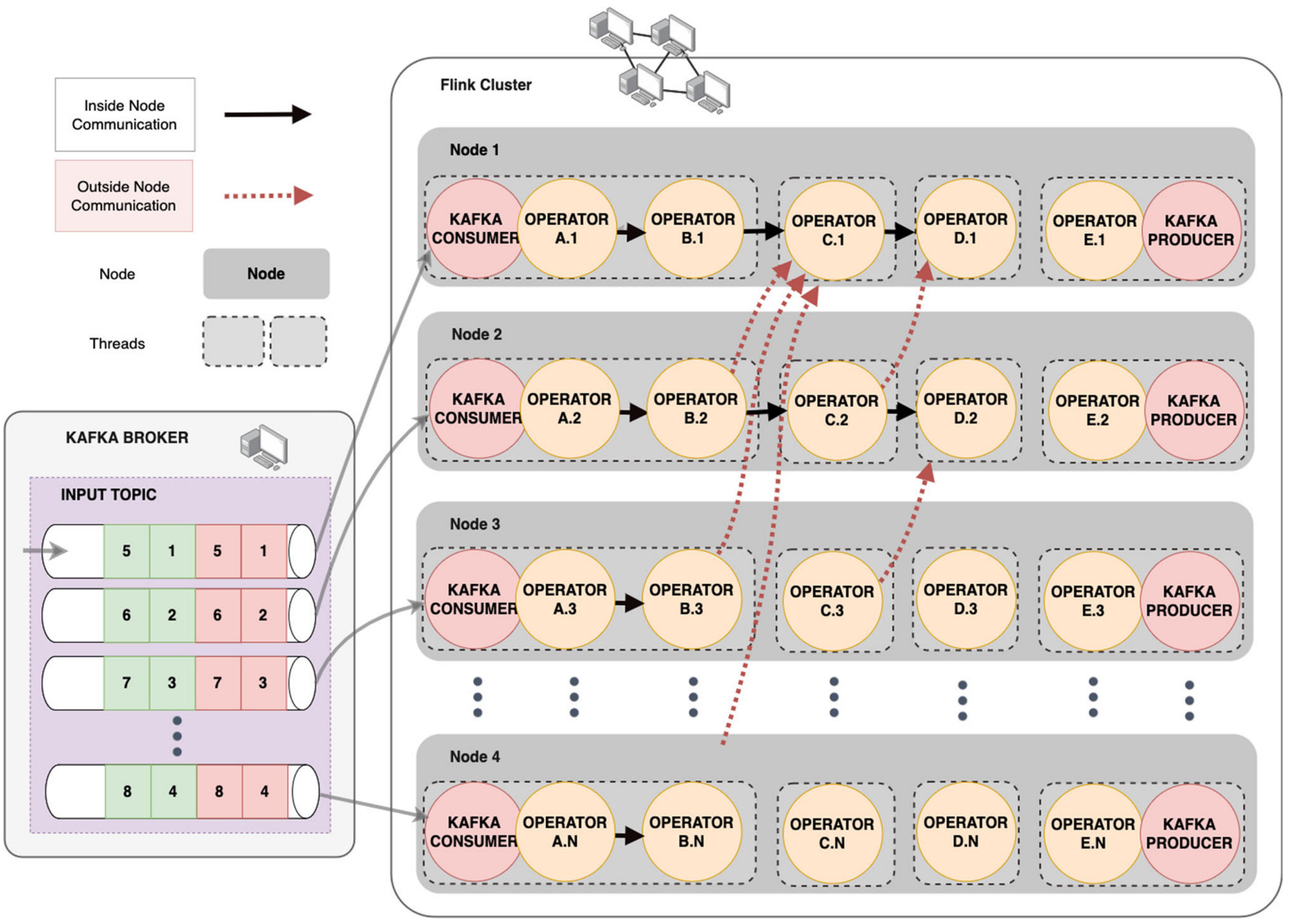


Fig. 15 Mapping of operators to Flink nodes and threads

of the same or even different resolutions. An input throughput of 35.6MB/s can come from two clients sending a 540p video (17.8MB/s bitrate) each. The size of the block was fixed to 200KB.

For each input throughput, Video2Flink was run with parallelization 8 and its latency is compared against that of a monolithic Java program (i.e., Video2Flink with parallelization 1). The number of Apache Kafka partitions is equal to the respective parallelization values of each run (i.e., 8 and 1). Figure 16 summarizes the results of this experiment for YUV video. For all but the highest input throughput, the latency is less than 60 s (i.e., the duration of the videos). This result supports the claim that Video2Flink can process in real time (e.g., streaming video).

Video2Flink is at least three times faster (for low throughput) reaching a maximum speed-up of 7.19 at 178 MB/second input throughput. After that throughput value, speed-up started to decline. The decline can be attributed to Apache Kafka's performance. At the highest input throughput (over 200MB/second), Apache Kafka struggles to send data fast enough to Flink. This is in line with the results reported by Lazidis and Petrakis [15]. Increasing the parallelization at this point would not improve speed-up. The performance bottleneck of Apache Kafka is due to the I/O with the hard disk. The highest throughput for a system requires a high degree of customization of both, system and hardware which is not possible to do a public cloud (i.e., Video2Flink runs on GCP).

This claim is supported by the results of Fig. 17 showing the latency of Apache Kafka and Apache Flink as well as the Flink to Kafka lag. The Flink to Kafka lag indicates how much slower (in seconds) was Flink compared to the time it takes Kafka to send the video. For example, for low input throughput, Apache Kafka took 10 s less (on average) to send the video than it took Apache Flink to process it. For higher throughput, the processing time of Flink was at least equal to the time that Apache Kafka required to send the video. At that point, the performance of Apache Flink is constrained by the performance of Apache Kafka.

The low speed-up observed for low throughput is attributed to the fact that a small part of each video (i.e., equal to block size) cannot be parallelized. This has a negative impact

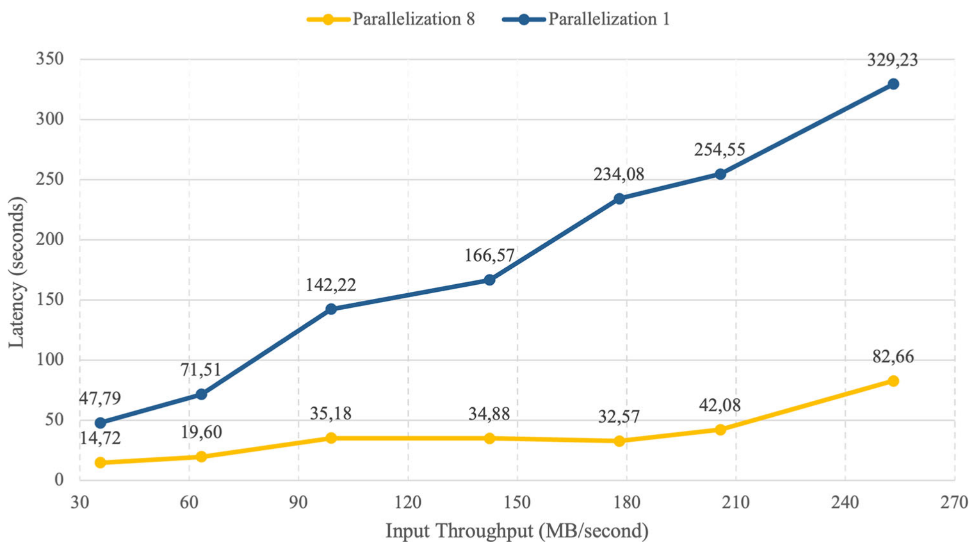


Fig. 16 Video2Flink latency (in seconds) for parallelization 8 on YUV video

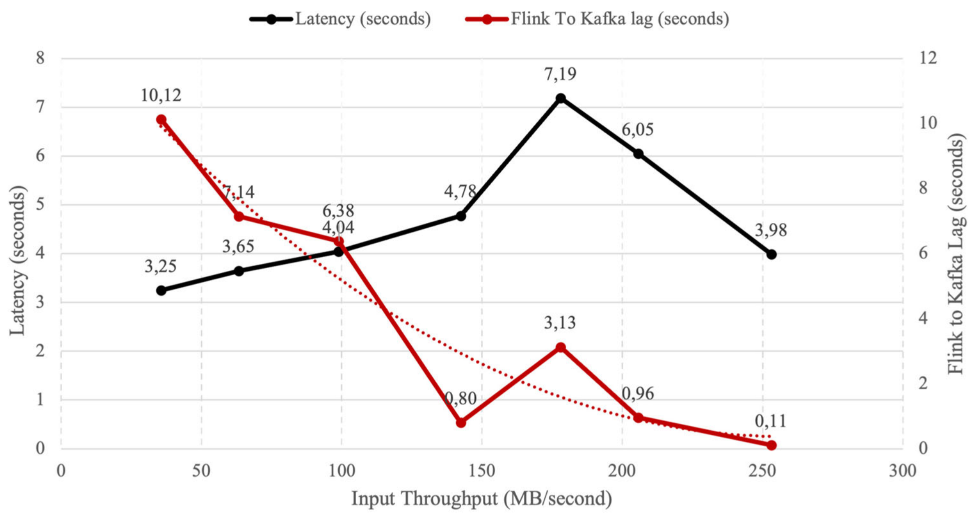


Fig. 17 Video2Flink speed-up and Flink to Kafka lag for parallelization 8 and YUV video

on performance especially for low input throughput [10] as worker machines may run under-utilized. On the other hand, smaller block sizes also adversely affect performance (i.e., the communication within Flink nodes increases). The optimal value of block size depends on the infrastructure (GCP in this work) and can only be determined by experimentation.

Figure 18 shows the latency for RGB video. The results show up to 15% slower response times compared to the results for the YUV video. The RGB videos are slightly harder to transfer and process since the full data has to be sent to Flink. For the YUV videos, only two-thirds of the video data (the Y component) has to be sent. RGB video has to be transformed into grayscale, whereas the Y component data of YUV videos is already in that form.

Figure 19 reports the latency of Apache Kafka and Flink as well as the Flink to Kafka lag for RGB video. Regarding the decline of speed-up for the highest resolution video, it is attributed again to the same root cause as in the previous experiment for YUV videos.

6 Conclusions and future work

Video2Flink2F is a distributed video processing method for Apache Flink. It takes full advantage of Apache Flink's capabilities to process small video chunks (i.e., at the size of a frame row or smaller) in parallel. The experimental results support our claim that Video2Flink can process video in real time (e.g., streaming video). Video2Flink achieved a very high speed-up (i.e., 7.19 for YUV video) for input throughput 200 MB/second on 8 processors, which is close to the theoretical limit (i.e., the speed-up cannot be higher than the number of processors working in parallel). This high speed-up is attributed to the optimal mapping of the shot detection algorithm to low level (i.e., at the finer pixel level) Apache Flink operations while minimizing their inter-dependencies. Video2Flink is a reasonably complex system with a plethora of intertwined configurations. Optimizing the performance of the processing platform required deep analysis of its capabilities (i.e., features, strengths, and weaknesses) and of its

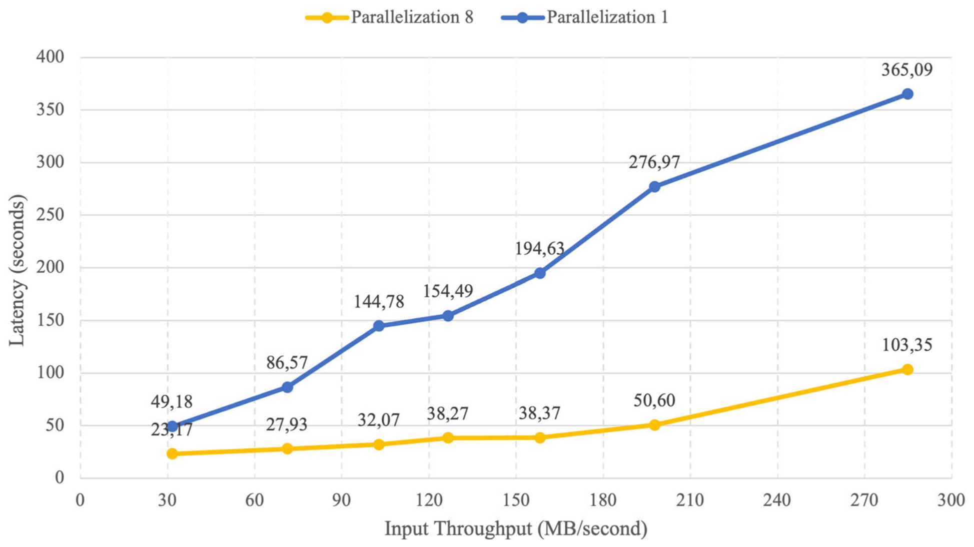


Fig. 18 VVideo2Flink2F response time (in seconds) for parallelization 8 on RGB video

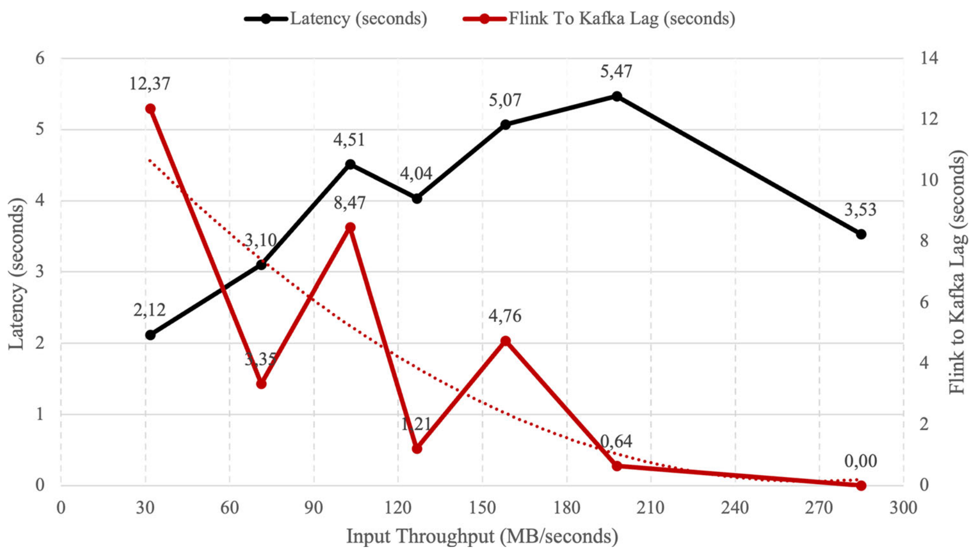


Fig. 19 Video2Flink speed-up and Flink to Kafka lag for parallelization 8 and RGB video

interoperability with Apache Kafka which is challenged with the task of driving a large amount of video data to Apache Flink.

Future extensions of the system include incorporating an auto-scaling agent [9] to support the automatic adaptation of parallelism to varying video workloads in the input, a method for automatically adapting the twin comparison method thresholds to the input video type, and an extension to support the processing of native MPEG-2- and MPEG-4-encoded video. Processing MPEG-encoded videos with minimal transcoding [5] would ultimately solve multiple of this system's bottlenecks, mainly its Kafka transferring times between the video providers and the Flink processing pipeline. Video segmentation research is currently limited by the lack of benchmark datasets [7,20]. To the extent these benchmarks become available soon, testing Video2Flink on more datasets and comparing its performance with other state-of-the-art methods is also left as future work.

Acknowledgements We are grateful to Google for the Google Cloud Platform Education Grants program. The work has received funding from the European Union's Horizon 2020—Research and Innovation Framework Programme H2020-SU-SEC-2019, under Grant Agreement No. 883272—BorderUAS.

Funding Open access funding provided by HEAL-Link Greece.

References

1. Amazon: Detecting video segments in stored video, Amazon Rekognition Developer Guide. (2022) https://docs.aws.amazon.com/rekognition/latest/dg/segments.html
2. Baier J.: Getting Started with Kubernetes. Packt Publishing, Birmingham - Mumbai (2017) https://www.oreilly.com/library/view/getting-started-with/9780135237823/, 2nd Edition
3. Cotsaces, C., Nikolaidis, N., Pitas, I.: Video shot detection and condensed representation. A review. IEEE Signal Process. Mag. **23**(2), 28–37 (2006)
4. Couto, J., Lazama, F.: A guide to video analytics: applications and opportunities (2022) https://tryolabs.com/guides/video-analytics-guide, tryo-Labs
5. Doulaverakis, C., Vagionitis, V., Zervakis, M., et al.: Adaptive Methods for Motion Characterization and Segmentation of MPEG Compressed Frame Sequence. In: Intern. Conference on Image Analysis and Recognition (ICIAR 2004), Porto, Portugal, pp 310–317 (2004) https://link.springer.com/book/10.1007/b100437
6. Flink: Apache Flink - Stateful Computations over Data Streams. (2022) https://flink.apache.org/, the Apache Software Foundation
7. Galasso, F., Nagaraja, N., Cardenas, T., et al.: A unified video segmentation benchmark: Annotation, metrics and analysis. In: IEEE International Conference on Computer Vision (ICCV 2013), pp. 3527–3534. Los Alamitos, CA, USA (2013) https://doi.ieeecomputersociety.org/10.1109/ICCV.2013.438
8. Garcia-Garcia, A., Orts-Escolano, S., Oprea, S., et al.: A survey on deep learning techniques for image and video semantic segmentation. Appl. Soft Comput. **70**, 41–65 (2018). https://doi.org/10.1016/j.asoc.2018.05.018
9. Giannakopoulos, P., Petrakis, E.G.: Smilax: Statistical Machine Learning Autoscaler Agent for Apache FLINK. In: Advanced Information Networking and Applications (AINA 2021), pp. 433–444. Canada, Toronto (2021) https://link.springer.com/chapter/10.1007/978-3-030-75075-6_35
10. Gustafson, J.L.: Amdahl's Law. In: Encyclopedia of Parallel Computing. Springer Link (2011) https://doi.org/10.1007/978-0-387-09766-4_77
11. Huang, Q., Petchean, A., Knowles, P., et al.: SVE: Distributed Video Processing at Facebook Scale. In: ACM Symposium on Operating Systems Principles (SOSP 2017), pp. 87–103. Shanghai, China (2017) https://doi.org/10.1145/3132747.3132775
12. Kim, Y.K., Kim, Y., Jeong, C.S.: RIDE: real-time massive image processing platform on distributed environment. EURASIP J. Image Video Process. (2018). https://doi.org/10.1186/s13640-018-0279-5
13. Kleppmann, M.: Making Sense of Stream Processing. O' Reilly, Sevastopol, CA (2016) https://www.oreilly.com/library/view/making-sense-of/9781492042563/, 2nd Edition
14. Kubernetes: Kubernetes - Production-Grade Container Orchestration (2022) https://kubernetes.io
15. Lazidis, A., Tsakos, K., Petrakis, E.G.M.: Publish-subscribe approaches for the IoT and the cloud: functional and performance evaluation of open-source systems. Internet of Things **19**(100), 538 (2022) https://doi.org/10.1016/j.iot.2022.100538
16. Lendave, V.: A guide to video object segmentation for beginners (2021) https://analyticsindiamag.com/a-guide-to-video-object-segmentation-for-beginners/
17. Minaee, S., Boykov, Y., Porikli, F., et al.: Image Segmentation Using Deep Learning: A Survey. IEEE Trans on Pattern Analysis and Machine Intelligence (IEEE PAMI) **44**(7), 3523–3542 (2022) https://www.computer.org/csdl/journal/tp/2022/07/09356353/1rigXK0s5Ak
18. Narkhede, N., Shapira, G., Palino, T.: Kafka The definitive guide. O'Reilly media, real time data and stream processing at scale (2017) https://spark.apache.org/
19. OpenCV: OpenCV AI Courses (2022) https://opencv.org/, the OpenCV team
20. Perazzi, F., Pont-Tuset, J., McWilliams, B., et al.: A Benchmark Dataset and Evaluation Methodology for Video Object Segmentation. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016), Las Vegas, USA, pp 724–732 (2016) https://ieeexplore.ieee.org/document/7780454
21. Seinstra, F.J., Geusebroek, J., Koelma, D., et al.: High-performance distributed video content analysis with parallel-horus. IEEE MultiMedia **14**(4), 64–75 (2007)
22. Spark: Apche Spark - Unified engine for large-scale data analytics (2022) https://spark.apache.org/, the Apache Software Foundation
23. Tekalp, M.: Video and image processing in multimedia systems. O' Reilly (2015) https://www.oreilly.com/library/view/digital-video-processing/9780133991116/
24. Uddin, M.A., Alam, A., Tu, N.A., et al.: SIAT: a distributed video analytics framework for intelligent video surveillance. Symmetry **11**(7), 1–22 (2019)
25. Vinka, E., Johansson, L.: Apache Kafka. Cloudkarafka (2019) https://www.cloudkarafka.com
26. Zhang, H.J., Kankanhalli, A., Smoliar, W.S.: Automatic partitioning of full-motion video. Multimedia Syst. **1**, 10–28 (1993). https://doi.org/10.1007/BF01210504

Euripides G.M. Petrakis received a B.Sc. in Physics from the National University of Athens, Greece, in 1986 and a PhD from the University of Crete, Greece, in 1993. Since 2013 he has been a Professor of Computer Science at the School of Electrical and Computer Engineering of the Technical University of Crete (TUC). His research activities cover various fields, starting from Computer Vision and Information Systems (in his early career years), Semantic Web, and later, Software Engineering, IoT, and Cloud Computing. He has authored more than 160 papers in Tier-1 international journals and conferences. He is an Associate Editor of Elsevier's Internet of Things journal. He is a Senior Member of the IEEE. He has been awarded the TUC 'excellence award' for 2022.

Dimitrios Kastrinakis recently graduated from the Technical University of Crete, Chania, Crete, with a degree in Electrical and Computer Engineering, in 2022. His research focuses on Software Engineering, Cloud Computing, Video Processing, and Machine Learning. He is currently a software engineer at Epignosis, an innovative e-learning company.