Scalable Distributed Visual Computing for Line-Rate Video Streams

Chen Song*
Simon Fraser University
Burnaby, British Columbia, Canada
csa102@sfu.ca

Jiacheng Chen*
Simon Fraser University
Burnaby, British Columbia, Canada
jca348@sfu.ca

Ryan Shea
Simon Fraser University
Burnaby, British Columbia, Canada
rws1@cs.sfu.ca

Andy Sun
Simon Fraser University
Burnaby, British Columbia, Canada
hpsun@sfu.ca

Arrvindh Shriraman
Simon Fraser University
Burnaby, British Columbia, Canada
ashriram@cs.sfu.ca

Jiangchuan Liu
Simon Fraser University
Burnaby, British Columbia, Canada
jcliu@cs.sfu.ca

ABSTRACT

The past decade has witnessed significant breakthroughs in the world of computer vision. Recent deep learning-based computer vision algorithms exhibit strong performance on recognition, detection, and segmentation. While the development of vision algorithms elicits promising applications, it also presents immense computational challenge to the underlying hardware due to its complex nature, especially when attempting to process the data at line-rate.

To this end we develop a highly scalable computer vision processing framework, which leverages advanced technologies such as Spark Streaming and OpenCV to achieve line-rate video data processing. To ensure the greatest flexibility, our framework is agnostic in terms of computer vision model, and can utilize environments with heterogeneous processing devices. To evaluate this framework, we deploy it in a production cloud computing environment, and perform a thorough analysis on the system's performance. We utilize existing real-world live video streams from Simon Fraser University to measure the number of cars entering our university campus. Further, the data collected from our experiments is being used for real-time predictions of traffic conditions on campus.

CCS CONCEPTS

• **Computer systems organization** → **Real-time system architecture**; *Cloud computing*; • **Information systems** → *Multimedia streaming*; *Data stream mining*; • **Computing methodologies** → *Visual inspection*;

KEYWORDS

Scalability, Visual computing, Line-rate processing

*C. Song and J. Chen contributed equally to this work.

MMSys'18, June 12–15, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5192-8/18/06…$15.00
https://doi.org/10.1145/3204949.3204974

ACM Reference Format:
Chen Song, Jiacheng Chen, Ryan Shea, Andy Sun, Arrvindh Shriraman, and Jiangchuan Liu. 2018. Scalable Distributed Visual Computing for Line-Rate Video Streams. In *MMSys'18: 9th ACM Multimedia Systems Conference, June 12–15, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3204949.3204974

1 INTRODUCTION

Recent developments and breakthroughs in computer vision have made possible applications such as auto-driving vehicles, self-checkout stores (e.g. Amazon Go), and high-accuracy surveillance. With state-of-the-art algorithms approaching human-level performance on core vision tasks including recognition, detection, and segmentation, people can now extract reliable and accurate information from images or videos and then conduct post processing according to the specific tasks.

Nowadays, deep learning based methods have exhibited strong performance, enabling computers to analyze such data automatically and reliably. However, processing speed is one of the major obstacles that prevents people from using deep learning based computer vision algorithms on large-scale video data. Computers must produce analysis results no slower than cameras generate new video data. Otherwise, the amount of data waiting to be analyzed will grow continuously, and eventually the waiting time for newly created data will be rendered irrelevant. In other words, a reasonable analysis system has to process video streams at "line-rate". Unfortunately, many effective algorithms and models are so complex that few current hardware systems are capable of executing them for video data at line-rate.

In real world scenarios, the ability to analyze video is especially important. Since most events and actions are expressed by continuous scenes, a single image might not capture enough information. With many successful techniques for image analysis being proposed and perfected, a common way to analyze a video is to do image-based analysis first, and possibly consider the temporal connection later to further investigate the video. This processing manner is exactly suitable for distributed computing, through which the video frames can be distributed to a cluster, and image-level information can be efficiently extracted. The video analysis process can thus be largely accelerated. In addition to accuracy, processing time, or in other words, the response time of the video analyzing system, is

also an important factor for real world applications. Many applications require a fast response for the analysis result to be useful. If one considers the large amount of video streams generated in those scenarios and the high computing complexity of current computer vision models, it can be very challenging to satisfy both accuracy and the speed of large-scale video analysis.

There are several solutions to the efficiency problem. For example, one can purchase and use a very powerful computer to process the video data. However, monolithic computers have disadvantages. First of all, they are expensive, which makes the analysis task prohibitive. Additionally, the maximum computational power a single machine can have is limited. When the computer vision algorithm is highly complicated and/or the number of video streams is very large, it is possible that even the most powerful machine is unable to process them at line-rate. In other words, this approach is not scalable. An alternative solution to the efficiency problem uses simplified computer vision models if videos are required to be processed quickly. This solution is not ideal either because the accuracy of the analysis is sacrificed.

In this paper, a scalable distributed visual computing system capable of executing any given computer vision algorithms on video streams is designed and implemented. The features of the proposed system matches the points we discussed above:

- The system is scalable, which means the computing resources assigned to it can be adjusted easily. This enables the system to handle even large amount of video data and process them at line-rate.
- The system processes video streams in a distributed manner, cuts live video streams on the temporal domain and distributes video slices to a cluster with scalable computing resources. The analysis for many video chunks are conducted simultaneously and temporal relations between video chunks can be considered in post processing.
- The system is designed to support heterogeneous underlying devices, which means the computer vision algorithms can run on CPUs, GPUs, or other underlying hardware according to the configuration of the cluster. Though largely accelerating the execution of most vision models, powerful GPUs are not necessarily required by the system. A cluster consisting of sufficient commodity CPUs can satisfy most computing tasks as well.

The high-level view of the system is to establish a general visual computing platform for applying advanced computer vision models to live video streams and generating valuable information at line-rate. Sec. 2 discusses the related works in terms of large-scale video processing, cloud-based computer vision, and multimedia processing with big-data frameworks. In Sec. 3, we present the architecture of the proposed system and analyzes each component in detail. Sec. 4 shows the result of our experiments with regard to the parameter of the system and its performance under different settings. Finally, in Sec. 5, we discuss potential use cases of the proposed system.

2 RELATED WORK

A common use case of large-scale video analysis is video surveillance. The number of surveillance cameras in the world is growing dramatically. According to IHS (Information Handling Services Markit Ltd.), there were 245 million surveillance cameras installed globally as of 2014 [8], and an estimated 130 million more will be shipped in 2018 [16]. These surveillance cameras produce a tremendous amount of video data every single second. However, a large portion of this data is simply stored on a disk and discarded after a few months, without any useful analysis. To this end researchers have been developing automated video surveillance systems requiring minimal human interaction. The promise of real-time tracking of activities of interest, such as crime, or assisted living, has driven researchers to explore both the possibilities of such a system, as well as its ramifications. In 2007, Girgensohn *et al.* proposed an integrated multi-camera surveillance system capable of tracking a person from camera to camera [6]. In 2013, a forum held by the IEEE discussed the history and future of video surveillance, as well as the challenges faced for fully autonomous video surveillance to become a reality [13]. One of the major challenges described was the difficulties in moving such a system from a laboratory to a real-world environment—Lam explores, and provides a framework for, the interaction costs of such a system [10], while Dadashi *et al.* provides guidelines for system usability to aid operators [3].

Another key challenge raised in the forum is the fact that video surveillance has turned largely into a data problem, and the lack of processing power has resulted in videos sitting on mass storage devices, waiting for analysis after the event has occurred. However, with the success of cloud computing, this hardware constraint is no longer an issue: Zhong *et al.* presented a real-time object recognition system by leveraging the cloud [18], and in 2015, Agrawal *et al.* presented a comprehensive, distributed computer vision system that can process gigapixels of data [1]. At the same time, researchers have also explored the effects of compression on predominant computer vision techniques [5].

Similarly, recent advancements in big data frameworks present a unique opportunity to tackle the video processing and analysis aspect of a surveillance system. In 2010, Pereira *et al.* proposed a method for parallelized encoding by splitting and merging along the temporal domain [12]. By using the split-and-merge method, researchers have explored using Hadoop for facial detection [7], and later for general purpose computer vision [14]. In 2015, Yang *et al.* presented a similar scalable system utilizing Spark instead of Hadoop [17]. According to Yang *et al.*, Spark has a superior efficiency compared to other distributed computation frameworks such as MapReduce, and is therefore preferred in distributed video data processing tasks.

The existing literature suggests a need to further explore systems for scalable line-rate processing of video stream data. To this end, we present a scalable distributed visual computing system capable of executing any given computer vision models on video streams at line-rate.

3 SCALABLE LINE-RATE VISUAL COMPUTING

The proposed scalable distributed visual computing system consists of three major modules. In this section, we first provide a high-level overview about the architecture of the system, clarify the term

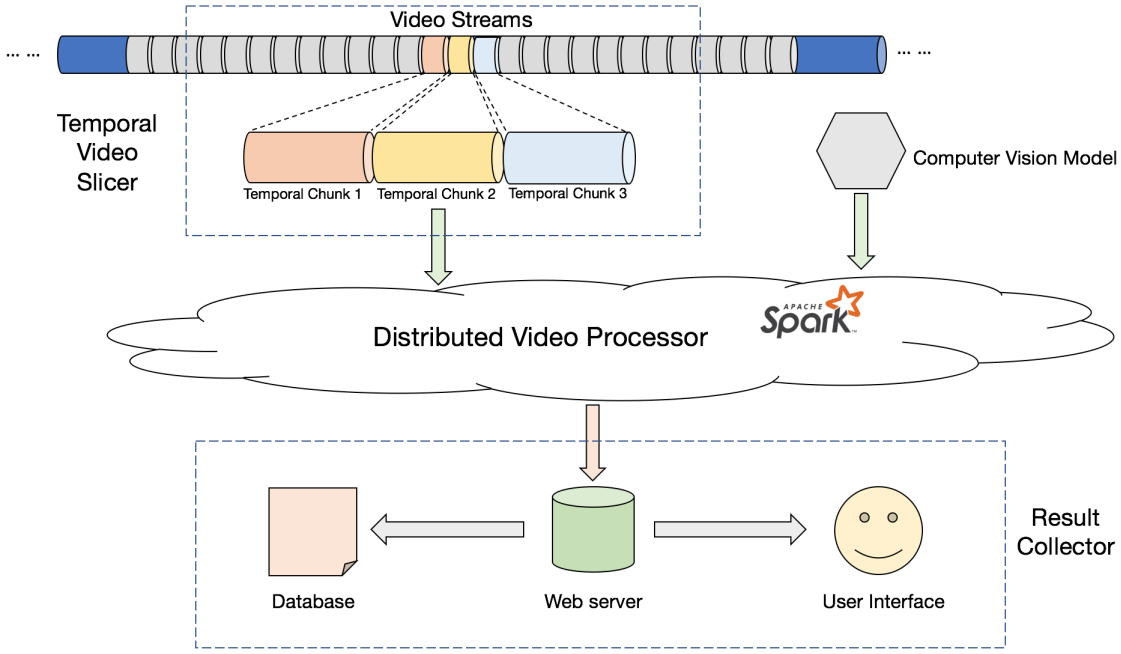

Figure 1: The overall architecture of our scalable distributed video processing system

"line-rate processing", and introduce each individual module of the system in detail.

3.1 Overview

3.1.1 System Modules. As shown in Fig. 1, the system consists of three modules: (1) temporal video slicer; (2) distributed video processor; and (3) result collector. The system is designed to achieve line-rate video processing by coordinating different machines in a cluster instead of using a single powerful computer.

3.1.2 Line-rate Performance. The proposed system has two inputs and one output. The user provides video streams (either a single stream or a group of multiple streams), and specifies the computer vision model to be applied. The analysis result of a frame is guaranteed to be available T_{delay} after the system receives it (see discussions in Sec. 4.2). The system is said to have a line-rate performance as the speed of output is no slower than the input video streams (both measured in frames per second).

Most video streams, especially the ones created by surveillance cameras as discussed in Sec. 1, generate new video frames consistently. It is assumed that the computer vision algorithm analyzes the video stream on a frame-by-frame basis. It treats each frame as an individual image; analyzes the image, and outputs the result. Theoretically, the minimum possible delay T_{delay} introduced by a line-rate system is equal to the time required to analyze one frame. However, in practice, the extra overhead can be so large that it becomes unreasonable to achieve the theoretical minimum delay. Detailed discussions can be found in Sec 4.

3.1.3 Assumption. The computer vision model should not assume strong temporal dependencies among consecutive frames. This allows the video slicer to make cuts at any point in the temporal domain, without doing harm to the performance (e.g. precision and recall if the model is used for object detection). As most state-of-the-art computer vision models are image-based, which actually first analyze videos in a frame-by-frame manner and then consider temporal connections later, this assumption does not limit the range of applications of the proposed system.

3.2 Temporal Video Slicer

When the video streams arrive at the system, they are cut into small chunks in the temporal domain and buffered to the file system. Although the computer vision model does not make use of the temporal dependency among consecutive frames, extreme care must be taken when deciding the position of cuts.

If temporal compression techniques are used in the input video stream, which is true in most cases, it will be ideal to only make cuts at key frames. Otherwise, an unnecessary decoding and re-encoding overhead will be introduced to the slicer, and this overhead will obviously be an obstacle for line-rate processing.

There exist many open-source solutions to cut a video stream only at key frames, including FFmpeg [2], which is used in our proposed system. FFmpeg allows us to specify the length of each video chunk (in seconds), and makes cuts at the closest key frames to the given timestamps.

3.3 Distributed Video Processor

3.3.1 Major Architecture. As discussed previously, the key to line-rate processing is to generate analysis results at the same speed as the frame rate of the input video stream. It is challenging for a single machine with limited computing resources to consume

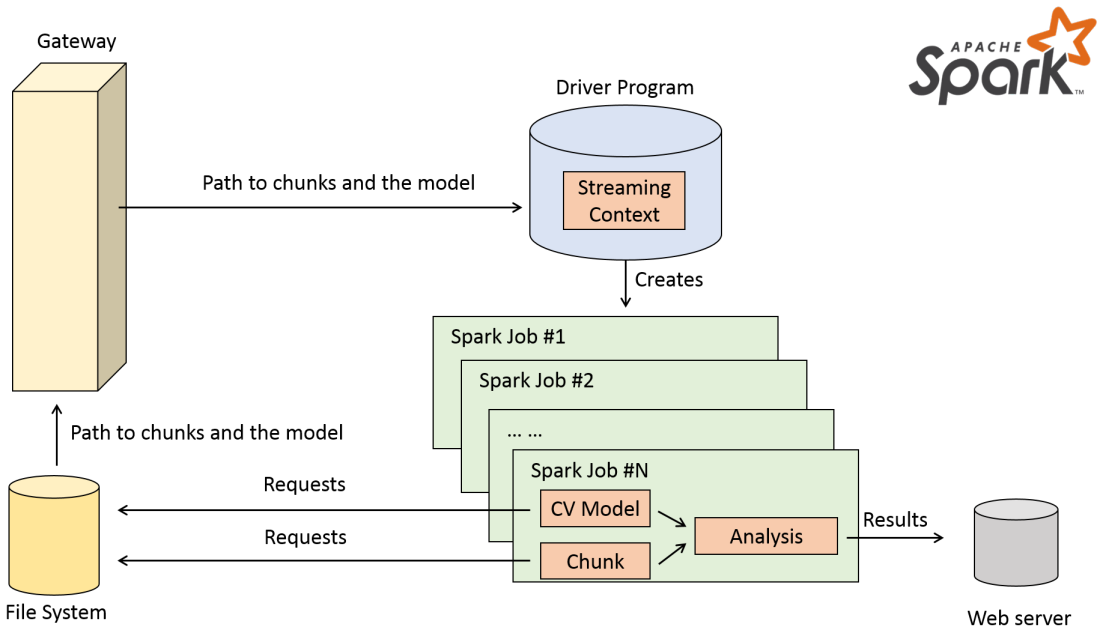

Figure 2: Architecture of the Distributed Video Processor

video data at line-rate. This is especially true when the computer vision algorithm is highly computationally intensive (e.g. complex neural network models). The distributed video processor as shown in Fig. 2 exploits a cluster with adaptive computation resources and processes video chunks in a distributed and parallel manner.

The distributed video processor is built on the streaming module of Apache Spark, which has good properties ideal for a robust distributed system (e.g. fault tolerance). During the execution of the system, the Spark driver program constantly monitors the file system where the outputs of the temporal video slicer will go. If any new video chunks are detected, Spark will encapsulate each chunk into a "task", and further encapsulate all "tasks" into a "job". By default, Spark checks the file system once per second. If the length of each video chunk is greater than one second, Spark will detect at most one newly created chunk every time it checks the file system. As a result, typically, there is only one Spark task per Spark job. When the Spark task is assigned to an executor by the resource manager, the driver program sends the executor the path to the video chunk and the computer vision model in the file system. The executor is then responsible for opening the video chunk, decoding it, and performing analysis on the frames using the vision model. If there are N Spark executors available, a maximum of N video chunks can be processed concurrently.

It is common for the video chunk to have a relatively high frame rate. In such circumstances, it is unnecessary to analyze every single frame for applications like detection because the detection results of adjacent frames are very similar [9]. To save computational resources and accelerate the process, it is possible to take a sample in every M consecutive frames, and use the analysis result on the sample for all the M frames. However, this kind of decisions should depend on the actual application—if down-sampling is inappropriate, then adding more computing resources to the system will often be the safe solution.

3.3.2 Heterogeneous Processing. Many state-of-the-art computer vision models employ GPUs or even dedicated hardware to accelerate image processing. These neural-network-based models contain large amount of matrix manipulation that can be done in parallel, which is a weakness of CPUs but well handled by GPUs. The distributed video processor is agnostic to the underlying hardware that actually execute the jobs. Once the job itself checks the availability of computing devices, it can assign its computation to the specified ones.

3.4 Result Collector

After the video chunks are processed by each worker node, the analysis results need to be collected and reassembled so the results can be analyzed by the user. In this paper, the result collector is implemented as a web server. After finishing the analysis, each worker node sends the results to the web server through an HTTP POST request. The result collector then stores it to a relational database to be later fetched when presenting to the user. As the computational capacity of each worker node is different, the analysis time for each video chunk is dynamic. As a result, the analysis results may not arrive at the collector in the order the chunks are created. Depending on the application, the result for a specific chunk may only be useful if we already have the results for all previous chunks. In such cases, after the result is received by the collector, it waits until all the previous chunks have analysis results available before delivering to the user.

3.4.1 Chunk Reintegration and Presentation. Since the video stream is split into discrete chunks that are randomly distributed to worker nodes with varying processing capacities, there is no guarantee of in-order completion of the chunks. This lack of temporal cohesion creates a problem when presenting a processed video stream back to the user, as each chunk depends on the preceding chunk.

The solution that we currently employ is to simply buffer the output data by the processing time of the slowest worker. This delay can also be adjusted based on the length of the discrete video slices, the number of available workers, and level of fault-tolerance desired in order to maintain line-rate processing. Given sufficient uniformly powerful hardware, it may be possible to do real-time processing.

The major advantage of this approach is that once the initial buffer period is over, the output chunks can be treated as a reintegrated live stream of processed data rather than independent video slices. This allows us to build a presentation layer utilizing features contained in most modern browsers. The processed live stream is buffered in a video tag, and a transparent canvas is overlaid on top to draw the results of the computer vision analysis.

4 EXPERIMENTS AND ANALYSIS

In this section, we present the experiments for analyzing and measuring the performance of the proposed system.

The accuracy of the system in terms of the precision and recall is largely determined by the quality of the stream and the computer vision model for processing the stream, which are the inputs to the proposed system (see discussions in Sec 3) defined by the user. In this paper, instead of focusing on the accuracy, we analyze the system by:

- The requirement on the amount of computing resources
- The balance between the delay and the overhead

4.1 Experiments Setup

The experiments discussed in this paper are conducted in a production Spark environment managed by Cloudera Manager at our university. The cluster is dedicated for the experiments.

The cluster consists of 11 nodes. There is a gateway node and a history server node. The other 9 nodes are the workers who actually process video stream using the given computer vision model. The nodes in the cluster are interconnected by a 10 Gbps Ethernet connection.

Both the gateway and history server node have 16 vCPUs (Intel(R) Xeon(R) CPU E7-4830 v4 @ 2.00GHz) and 61.4 GB of memory each.

Seven of the worker nodes have 32 vCPUs (Intel(R) Xeon(R) CPU E7-4830 v4 @ 2.00GHz) and 144.1 GB of memory each. The other two worker nodes have 40 vCPUs (Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz) and 112.4 GB of memory each.

The Spark streaming application is submitted using YARN [15] in the client mode, which means the driver program runs on the gateway instead of the worker nodes. We allocate 18 executors when we submit the application. When the system is running with a full workload, a maximum of 18 video chunks can be processed simultaneously. In such circumstances, each worker node is responsible for $18 \div 9 = 2$ chunks.

We use Single Shot Multibox Detector [11] as the computer vision model in our experiments, as it's a popular state-of-the-art object detection model with a good trade-off on accuracy and speed. The models are pre-trained on the PASCAL VOC dataset [4] and are able to identify the 20 classes defined by the dataset.

For our initial test data, we used an hour long video of vehicles entering our university campus on a typical work day. The main objects in the video are vehicles (cars and buses). The video has a resolution of 910x512, a frame rate of 30 FPS, and is encoded in H.264. We create a server to stream this video in an endless loop.

4.2 The Concept of Delay

As discussed previously, it is challenging performing complex computer vision analysis in real-time. We instead focus on developing a system that can perform such analysis at line-rate with some delay. Intuitively, the concept of delay is defined as the amount of time spent on the analysis algorithm. That is, the amount of time elapsed from when a frame is produced until the analysis result of the same frame is available:

$$T_{analysis} = t_{available} - t_{production} \tag{1}$$

If $t_{available}$ is equal to $t_{production}$, which means the analysis result is available "immediately" after a frame is produced, $T_{analysis}$ will be zero. In such circumstances, the system is said to have real-time performance.

An important note to make is that in a real-world system, there can be additional computational overhead to process the analysis result after its availability. Such post-processing can be as simple as drawing bounding boxes around the detected objects, or as complex as sending the result to another system for further analysis. For simplicity, the overhead for post-processing is not included in (1).

The problem of this intuitive definition (1) is that, after the system is up and running, $T_{analysis}$ varies from frame to frame. Even if the complexity of the computer vision model is the same for every input frame, which is true for most models, the actual execution time may still be affected by:

- the capacity of the executor responsible for processing the chunk (the cluster has heterogeneous workers)
- the availability of computational resources (there can be other applications running in the cluster when deployed in a production environment)
- the network congestion level (sending and receiving a video stream itself can easily cause huge network congestion)

Given these uncertainties, it is difficult to accurately derive the formula for delay mathematically. However, it is possible to estimate the overall performance of the model when system is kept running for a long time:

$$T_{delay} = \lim_{f \to \infty} \max_{j=0}^{f} T_{analysis}(j) \tag{2}$$

where f is the index of the most recent processed frame in the entire video stream, and $T_{analysis}(j)$ is the analysis time for j^{th} frame.

Because our system distributes the computation in chunks of frames, we need to further generalize (2) to accommodate that:

$$T_{elapsed}(i) = T_{chunk}(i) + \sum_{j=0}^{F} T_{analysis}(j) \quad (3)$$

$T_{elapsed}(i)$ is the amount of time elapsed from the first frame in chunk i is produced to the analysis result of the first frame is available. $T_{chunk}(i)$ is the length of video chunk i (in terms of seconds), and F is the number of frames in chunk i.

Note that the first frame is not processed immediately after its production. It is sent to the worker node after the entire chunk i is received ($T_{chunk}(i)$ after receiving the first frame).

The generalized version of (2) is given by:

$$T_{delay} = \lim_{c\to\infty} \max_{i=0}^{c} T_{elapsed}(i) \quad (4)$$

where c is the index of the most recent processed chunk in the entire video stream.

4.3 The Requirement on Computing Resources

With the discussion on the definition of delay (4), we are able to decide whether a system is able to process inputs at a line-rate through experiments.

After the system starts, we compute $\max_{i=0}^{c} T_{elapsed}(i)$ after the completion of analysis on every chunk. As c increases, if it converges to a value below a certain threshold given by the user (the tolerance on the amount of delay), the resources in the system is considered to be sufficient. The blue line in Fig. 3 shows an example of such situations. Because $\max_{i=0}^{c} T_{elapsed}(i)$ converges to 218 seconds, the system is able to perform computer vision analysis on the inputs at line-rate with a delay of 218 seconds. In this example, we take a sample for analysis every 3 frames.

As c increases, if $\max_{i=0}^{c} T_{elapsed}(i)$ fails to converge, or converges above the tolerance given by the user, the resources in the system is said to be insufficient. The orange line in Fig. 3 shows an example of such situations. In this example, we analyze every frame of input video.

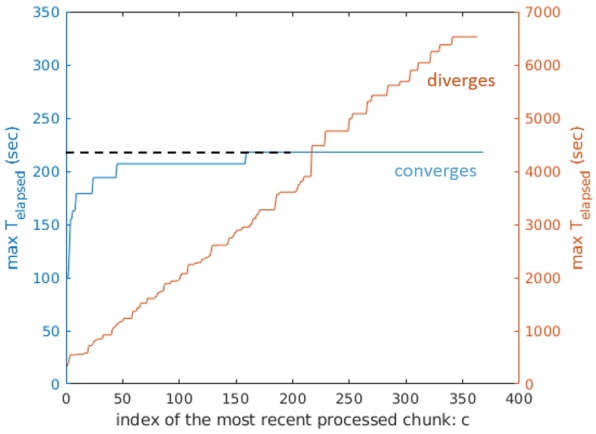

Figure 3: Sufficient resources for line-rate processing

As shown by the blue line in Fig. 3, even if the resources are sufficient for line-rate processing, $\max_{i=0}^{c} T_{elapsed}(i)$ may experience a dramatic increase when c is small. This is because there is only a very limited number of video chunks being processed in the cluster at the beginning of the experiment, and an executor can use all the resources on the worker node (it is the only active executor). When the cluster is given more video chunks, other executors on the same worker node will begin processing as well. Potential resource sharing on the worker nodes can often lead to performance interference, resulting in an increase in $T_{elapsed}$.

4.4 The Balance Between Delay and Overhead

According to equation (3) and (4) defined in Sec 4.2, the amount of delay introduced by the system T_{delay}, is positively correlated to the length of each video chunk T_{chunk}. In simple words, the delay is small when the video chunk is short, and is large when the video chunk is long. To reduce the delay, there is a strong motivation for us to create very small video chunks. In the extreme case, the system has a minimum delay when each chunk has only one video frame.

However, the potentially unavoidable overhead brought by the parallel architecture, including the time spent on allocating resources such as Spark's Resilient Distributed Datasets (RDDs), and broadcasting the computer vision model, discourages us from making a video chunk too small. When the model is used locally on the slowest worker node (with 32 CPUs), which is the bottleneck of the cluster, we find that it takes 18.4sec to analyze one second of the input video. When the same model is used in our proposed system, this value increases largely because of this overhead.

As shown in Fig. 4, the horizontal axis is the length of each video chunk T_{chunk}, and the vertical axes are the computational cost per second input (computed by $\frac{T_{delay}}{T_{chunk}}$), and the amount of delay introduced by the system T_{delay}. In this figure, the resources available in the cluster are guaranteed to be more than enough for all the test cases. We notice that when T_{chunk} is small, the computational cost to process an one-second video is high, indicating a huge overhead. When T_{chunk} increases, the computational cost per second of video converges gradually. When T_{chunk} = 20sec, the computational cost per second video is 19.9sec. The overhead here is $(19.9 - 18.4) \div 18.4 = 8.2\%$. When T_{chunk} = 1sec, the computational cost per second video is 32.0sec, representing an overhead of $(32.0 - 18.4) \div 18.4 = 74.0\%$.

As can be seen in Fig. 4, if the system splits the video into 20-second chunks, we will only need $\frac{19.9}{32.0} = 62.2\%$ of resources compared to the situation where 1-second chunks are generated. However, by doing so, a greater delay (398sec versus 32sec) will also be introduced. In a real-world system, although it is hard to do so, one has to carefully balance the relation between the delay and computational overhead.

4.5 Exploration of Scalability

Scalability is a key feature of the proposed system. A system is said to be scalable if its computing capability increases when more hardware resources are provided. Since our visual computing system is built upon Apache Spark, its computing resources can be

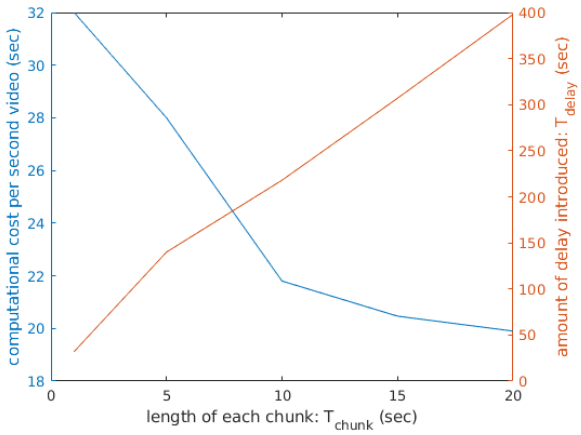

Figure 4: Computational overhead and chunk size

easily adapted by changing the settings through the cluster manager. In our implementation, the cluster manger is Apache Hadoop Yarn, but it can also be Spark's own cluster manager or Apache Mesos. We are interested in investigating how the scaling strategy influences the performance of the system.

There are two different methods to increase the amount of hardware resources in a distributed computing system. Imagine that all the worker nodes in the system are lined up in a horizontal line. We can either increase the hardware capacity of each individual node (known as "vertical scaling"), or increase the number of nodes without changing the hardware specification of each node (known as "horizontal scaling"). This section presents a comparison between these two scaling methods.

The experiment described below only uses a subset of computing resources discussed in Sec. 4.1. With a total of 80 vCPUs, we distribute them in three ways:

- 2 worker nodes, each with 40 vCPUs
- 4 worker nodes, each with 20 vCPUs
- 8 worker nodes, each with 10 vCPUs

In all cases, at a particular time, each worker node is responsible for two video chunks. In the first scenario, there are $2 \times 2 = 4$ video chunks being processed in parallel. In the second and third scenario, there are $2 \times 4 = 8$ and $2 \times 8 = 16$ chunks being processed in parallel, respectively. We keep the system up and running for 30 minutes, and the table below shows the computational costs to process one 10-second video chunk in different setups.

Table 1: Computational costs to process one video chunk

	2×40 vCPUs	4×20 vCPUs	8×10 vCPUs
$t_{average}$ (sec)	5.88	7.63	14.34
$t_{normalized}$ (sec)	1.47	0.95	0.89

[a] $t_{average}$ is the average time to process a video chunk.

[b] $t_{normalized}$ is $t_{average}$ divided by the number of parallel chunks.

As shown in Table 1, $t_{average}$ is an indicator of the delay introduced by the system. When the amount of resources on each individual worker node increases, the processing of each video chunk consumes more hardware resources and is therefore faster. Therefore, if delay is the major concern, vertical scaling will be better than horizontal scaling.

As also shown in Table 1, $t_{normalized}$ is an indicator of the amount of hardware resources used to process a chunk. Recall that the total amount of hardware resources (80 vCPUs) is fixed among the three setups. When N chunks are being processed in parallel, each chunk can be thought of as having used all the 80 vCPUs for $t_{normalized} = \frac{t_{average}}{N}$ time on average. When the number of workers increases, each chunk uses less hardware resources to process. Therefore, to keep a line-rate performance, if the total amount of hardware resources is limited, it will be ideal to scale the system horizontally so that the limited resources are better utilized.

In practice, there can be a limit to vertical scaling. If it is given more resources than required, a computer vision algorithm may not be able to fully utilize them. For example, if the algorithm only allocates 20 threads during its execution, it will be meaningless to equip it with more than 20 vCPUs. In such scenarios, horizontal scaling will be the only method for increasing the hardware capacity of the system.

4.6 Enhancement using Heterogeneous Processing

The heterogeneous processing of the proposed system has been discussed in Sec. 3.3. The experiments in previous sections focus on exploring (1) the settings and major concepts of the system (e.g. chunk size, computing resources, delay); (2) performance under different system settings; and (3) scalability. A CPU-based cluster as presented in Sec. 4.1 is sufficient for these purposes. However, we have also begun initial experiments on a GPU-enabled cluster to investigate the benefits brought by heterogeneous processing.

We perform the same visual computing task, applying the SSD detector on video streams to get analysis result, but this time on a single Geforce GTX 1080 GPU. The experiment shows that it takes 24 seconds on average to process a 20-second video chunk with 30 FPS. This result suggests that GPUs will greatly increase the amount of video data the cluster can process at line-rate due to its highly optimized parallelism and dedicated software-side accelerations (e.g. NVIDIA's CUDA, CuDNN). With the property of heterogeneous processing, the system will be able to work more efficiently and handle more complicated tasks when GPUs are included. Besides, this experiment result justifies the importance of the scalability of our system - one advanced GPU on a single machine fails to handle a line-rate processing task, but if the GPU is deployed in the proposed system, the performance gap can be filled by tens of commodity CPUs and the whole system can easily reach line-rate.

Though performance of GPUs seems to be dominant on running most state-of-the-art computer vision models, this does not mean CPUs are irrelevant for running computer vision models. A good number of highly optimized libraries are trying to close the gap between CPUs and GPUs on the tasks with heavy matrix manipulations, and we also make some exploration on this in our experiments. Libraries such as Intel MKL can largely improve CPU

performance. Libraries including OpenCV and Caffe already have MKL-equipped version, and are now harnessing more computing power from CPUs taking advantage of these dedicated software optimization. [1]

5 USE CASES

In this section, we first discuss how the users can deploy the proposed system in the cloud using services such as Amazon EC2, and then we show the potential use cases for the proposed system. Recall that the major characteristics of the systems are: (1) high scalability, which enables it to handle multiple video streams and complex computer vision algorithms simultaneously; (2) line-rate processing, which makes it applicable to situations requiring high response.

As presented in Sec. 3, the distributed visual computing system is built upon Apache Spark, FFmpeg, among other tools and libraries. Users can either deploy the system on their own cluster or take advantage of cloud service such as Amazon EC2 for deployment. If the system is deployed on EC2, the user will be able to adapt the computing resources based on the workload of actual tasks, and the scalability of the system will make it easy for users to get ideal performance economically.

5.1 Traffic Flow Monitoring at Line-Rate

A simple and common use case of the proposed system is sensor-free traffic flow monitoring. By using (most likely existing) cameras, the system can track cars entering and leaving an intersection, and estimates the situation of traffic flow (i.e. the congestion level). Since current computer vision algorithms can already classify different type of cars or recognize the license plate, it's also possible to record all these details in a database for further analysis.

The main campus of our university is located on a mountain with only one major entrance, which is also a four-way intersection. The campus security has deployed a high-resolution camera at this intersection. However, the camera is only used to record traffic videos. No computer vision analysis had been applied on these videos before our system. In our experiment, the proposed system is used to receive video stream from this camera, and count the number of cars and buses entering the campus.

To implement the vehicle counting functionality, the proposed system needs to be equipped with a properly designed computer vision algorithm. In this case, SSD detector [11] again is chosen as the general-purpose object detector. Pre-trained on PASCAL VOC, the detector is able to give a bounding box for objects of 20 common classes with high accuracy. For every single frame, SSD detector is applied for getting the bounding box of all vehicles(i.e. buses and cars) in the scene. Because the camera is fixed, the vehicles moving toward one direction will always go through certain regions in the captured video. Therefore, counting the number of vehicles moving toward certain direction is equivalent to counting the non-repeated occurrences (one vehicle is only counted once) of vehicles in certain small region in the video.

[1] intel-caffe:https://github.com/intel/caffe
OpenCV with dedicated CPU accelerations: https://github.com/opencv/opencv/wiki/DNN-Efficiency

Fig. 5 is an illustration of the counting method. The number of vehicles travelling along the direction of blue arrow is the same as the number of vehicles crossing the red segment. The 2D coordinates of detection in adjacent frames are compared, and the comparison result is used to make sure that there is no repetitive counting. Note that there exists temporal dependency when the detection results from adjacent frames are compared, which means that there will be counting errors at the boundary of video slices. However, the error will be tolerable if the length of each video slice is reasonably large. This is because the probability of a vehicle crossing the counting region at exactly the last moment of a video slice is rather small when the length of a video slice is large. We choose a time slice length that will not result in repetitive counts given the traffic flow in the location of the security cameras we use. This variable could be adjusted based on the deployment scenario.

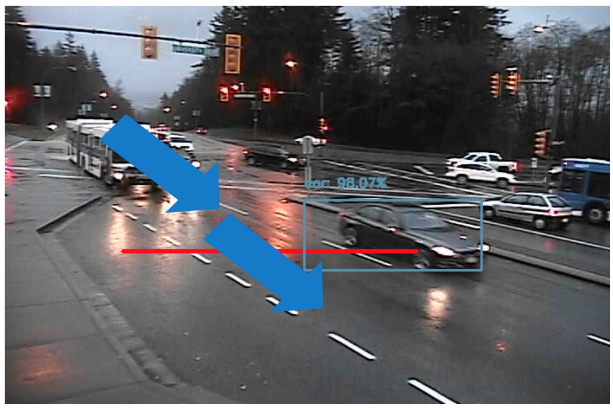

Figure 5: The illustration of our vehicle counting method. The blue arrow is the direction of cars entering campus, the red segment is the counting region. The system counts a vehicle when its detected bounding box crosses the counting region.

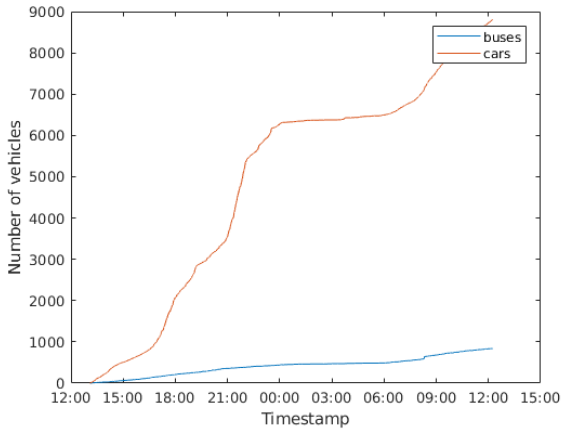

Figure 6: Cumulative number of vehicles entering the campus from Nov. 29 to 30, 2017

Fig. 6 shows the statistics from November 29 (Wednesday) to November 30 (Thursday), 2017. This figure reveals an interesting phenomenon: At most times, the number of buses and the number of cars increases simultaneously (the first-order derivatives of the two lines are positively correlated). However, at around 21:00, we notice a very large number of cars entering the campus, while the number of buses is much smaller. This suggests that many people are travelling to the campus late at night (possibly students live on the main campus, but take courses elsewhere). Potentially, the bus operating company should consider increasing the frequency of its services during this period.

Our system can also be useful for campus staffs to collect long-term traffic statistics, according to which the university may want to adapt its course and exam schedule, and parking lot allocation to maintain stable traffic with low congestion.

6 CONCLUSION AND FUTURE WORKS

In this paper, we propose a scalable distributed visual computing system for line-rate processing on video streams. The key feature of the proposed system is scalability and line-rate processing for large-scale video data. By distributing computing tasks to a cluster with heterogeneous underlying hardware, the system is able to utilize hardware resources and produce analysis results using given computer vision model at line-rate. On the basis of our implementation and experiments, we discuss major concepts of the system and further present the mechanism to estimate the requirement on computing resources when the system is deployed for real-life applications. We notice and discuss that there is a balance between the delay and the overhead introduced by our system. And we also explore the scalability of the proposed system by comparing vertical scaling with horizontal scaling. Finally, as an example use case, we present the results of deploying our system on a cloud environment to monitor traffic flow at the entrance of our university.

We note that there are room for potential enhancements. One possibility of future work is to introduce the idea of adaptive chunk slicing. Since the system consists of heterogeneous underlying hardware, it will be ideal if the computation is distributed according to the capacity of each component of the system. This requires the temporal video slicer module of our system to be aware of the computing capacity of each node in the cluster, and then cut video adaptively, assign the longer chunks to worker nodes with high computing capacity, while leave shorter chunks to slower ones. The adaptive chunk slicing will further exploit the computing resource of the system and is a promising future direction of our system.

ACKNOWLEDGMENTS

This work is supported by the Natural Sciences and Engineering Research Natural Sciences and Engineering Research Council (NSERC) of Canada Discovery Grant.

REFERENCES

[1] Harsh Agrawal, Clint Solomon Mathialagan, Yash Goyal, Neelima Chavali, Prakriti Banik, Akrit Mohapatra, Ahmed Osman, and Dhruv Batra. 2015. Cloudcv: Large-scale distributed computer vision as a cloud service. In Mobile cloud visual media computing. Springer, 265–290.

[2] Fabrice Bellard. 2017. Ffmpeg multimedia system. FFmpeg.[Last accessed: November 2017]. https://www. ffmpeg. org/about. html (2017).

[3] Nastaran Dadashi, A Stedmon, and T Pridmore. 2009. Automatic components of integrated CCTV surveillance systems: Functionality, accuracy and confidence. In Advanced Video and Signal Based Surveillance, 2009. AVSS'09. Sixth IEEE International Conference on. IEEE, 376–381.

[4] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2015. The Pascal Visual Object Classes Challenge: A Retrospective. International Journal of Computer Vision(IJCV) (2015).

[5] Wu-chi Feng, Ryan Feng, Paul Wyatt, and Feng Liu. 2016. Understanding the Impact of Compression on Feature Detection and Matching in Computer Vision. In Multimedia (ISM), 2016 IEEE International Symposium on. IEEE, 457–462.

[6] Andreas Girgensohn, Don Kimber, Jim Vaughan, Tao Yang, Frank Shipman, Thea Turner, Eleanor Rieffel, Lynn Wilcox, Francine Chen, and Tony Dunnigan. 2007. DOTS: support for effective video surveillance. In Proceedings of the 15th ACM international conference on Multimedia. ACM, 423–432.

[7] Arto Heikkinen, Jouni Sarvanko, Mika Rautiainen, and Mika Ylianttila. 2013. Distributed multimedia content analysis with MapReduce. In Personal Indoor and Mobile Radio Communications (PIMRC), 2013 IEEE 24th International Symposium on. IEEE, 3497–3501.

[8] N Jenkins. 2015. 245 million video surveillance cameras installed globally in 2014. IHS Technology (2015).

[9] Pavel Korshunov and Wei Tsang Ooi. 2005. Critical video quality for distributed automated video surveillance. In Proceedings of the 13th annual ACM international conference on Multimedia. ACM, 151–160.

[10] Heidi Lam. 2008. A framework of interaction costs in information visualization. IEEE transactions on visualization and computer graphics 14, 6 (2008).

[11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. In European Conference on Computer Vision(ECCV).

[12] Rafael Pereira, Marcello Azambuja, Karin Breitman, and Markus Endler. 2010. An architecture for distributed high performance video processing in the cloud. In Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on. IEEE, 482–489.

[13] Fatih Porikli, Francois Bremond, Shiloh L Dockstader, James Ferryman, Anthony Hoogs, Brian C Lovell, Sharath Pankanti, Bernhard Rinner, Peter Tu, and Péter L Venetianer. 2013. Video surveillance: past, present, and now the future [DSP Forum]. IEEE Signal Processing Magazine 30, 3 (2013), 190–198.

[14] Hanlin Tan and Lidong Chen. 2014. An approach for fast and parallel video processing on Apache Hadoop clusters. In Multimedia and Expo (ICME), 2014 IEEE International Conference on. IEEE, 1–6.

[15] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 5.

[16] J Woodhouse. 2017. Market for storage used for video surveillance worth $1.7bn in 2017. IHS Technology (2017).

[17] Shuai Yang and Bin Wu. 2015. Large scale video data analysis based on spark. In Cloud Computing and Big Data (CCBD), 2015 International Conference on. IEEE, 209–212.

[18] Yu Zhong, Pierre J Garrigues, and Jeffrey P Bigham. 2013. Real time object scanning using a mobile phone and cloud-based visual search engine. In Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility. ACM, 20.