A Stream Algebra for Computer Vision Pipelines

Mohamed A. Helala, Ken Q. Pu, Faisal Z. Qureshi
Faculty of Science, University of Ontario Institute of Technology, Oshawa, ON, Canada
{*Mohamed.Helala, Ken.Pu, Faisal.Qureshi*}*@uoit.ca*

***Abstract*—Recent interest in developing online computer vision algorithms is spurred in part by a growth of applications capable of generating large volumes of images and videos. These applications are rich sources of images and video streams. Online vision algorithms for managing, processing and analyzing these streams need to rely upon streaming concepts, such as *pipelines*, to ensure timely and incremental processing of data. This paper is a first attempt at defining a formal stream algebra that provides a mathematical description of vision pipelines and describes the distributed manipulation of image and video streams. We also show how our algebra can effectively describe the vision pipelines of two state of the art techniques.**

***Keywords*-Image and Video Streams; Stream Algebra; Computer Vision Pipelines; Stream Operators**

I. Introduction

Advances in computation, storage and networking have increased our ability to generate, store and consume vast amounts of data manyfolds. The recent emphasis on "Big Data" methods is a response to this development. There is a lot of interest in theories, algorithms and techniques for managing, analyzing and exploiting the data that we are now able to generate and collect. Big data research is in its infancy still, and much work is yet to be done. A promising direction of research in big data is *stream processing*. Stream processing refers to the class of techniques that deal with continuous streams of data. Data streams can take many forms and come from a variety of sources [1], [2], [3], [4]. E.g., Twitter[1] feeds can be considered text streams, videos collected by traffic cameras can be seen as video streams, temperature and humidity readings from a collection of physical sensors[2] installed at a winery can be seen as a data stream, etc. In order to efficiently process these streams, stream algebras [5], [6], [7] have been proposed in the database literature as a formal language for building and optimizing data streaming pipelines. This paper proposes a *stream algebra* that formalizes stream processing computer vision systems capable of dealing with image and video streams. We refer to these streams as *Vision Streams*.

Our interest in stream processing computer vision systems stems from three observations. First, text-based stream processing has found wide-spread use in database and information system communities [8], [9], [10], [11], [12]. Stream processing systems for managing and analyzing text-streams have been studied in that community for many years now and various stream algebras have been developed in an effort to formalize and understand text stream processing. Text stream algebras, however, rely upon relational algebra and related data analytic methods and cannot be readily used for designing stream processing systems that deal with vision streams. Second, many classical computer vision algorithms, and especially those that deal with multiple images and videos, can be naturally formulated as a vision stream processing problem. Online vision algorithms, in particular, stand to benefit from vision stream processing [13], [14], [15], [4], [16], [17], [18]. Lastly, the emergence of large scale camera networks and web scale vision has brought the need of big data computer vision algorithms into sharp focus. The recently announced Amazon Kinesis platform, for example, can acquire and process, in real-time, data streams from thousands of different web sources at a data rate of several terabytes per hour [19]. These observations show that developing stream algebras for vision stream processing is an essential step towards realizing big data computer vision systems of the future.

As stated earlier, this paper develops an algebra for constructing stream processing pipelines for dealing with vision streams. The proposed algebra can be used to realize vision systems for a wide range of computer vision applications by treating existing domain specific vision algorithms as *stream operators*. Furthermore, the proposed algebra is defined by formal semantics, which enables us to leverage formal methods for studying, designing and optimizing vision streams.

A. Stream Algebra

To date a number of stream algebras have been developed within the database and information systems communities in an attempt to describe queries over events or relational streams [20], [5], [6], [7]. There are many advantages to expressing stream processing systems using these algebras. For example, these algebras can be used to study these systems formally with a view to resolve blocking operations, schedule asynchronous tasks, implement dynamic execution plans, apply incremental evaluation, define common pipeline optimization and cost models, etc.

Existing stream algebras, however, are designed for textual streams with well-defined schemas. Consequently, existing algebras are not suitable for vision streams, which

[1]Twitter: www.twitter.com (*last accessed on 20 March 2014*).

[2]TandD Corporation: www.tandd.com (*last accessed on 20 March 2014*).

are characterized by complex multi-modal data and high-dimensional features. Furthermore, vision stream processors need careful tuning to achieve acceptable rates. Indeed many existing computer vision algorithms work in batch mode and cannot be used as stream processors.[3] Other challenges exist as well. For example, processing times of a vision algorithm may vary widely due to changes in the input size or content. Vision algorithms typically have several run-time parameters that control their accuracy or speed. These parameters are typically tuned in isolation of other vision algorithms that may present in the system. Moreover, vision algorithms may exist in multiple run-time environments (CPU, GPGPU, FPGA, etc.) with different accuracy and processing speed profiles. Dynamic run-time environment selection of a particular vision algorithm (or vision stream operator) should ideally take into account other vision algorithms in the system.

A number of frameworks have recently been proposed within the computer vision community to design vision pipelines [13], [14], [15], [4], [16], [17], [18]. Gstreamer [21] is a widely used streaming library for constructing vision pipelines. Twitter has recently released the Storm [22] framework. None of the existing vision pipeline frameworks, however, define a stream algebra i.e., these frameworks cannot formally define a vision stream or the associated stream operators. Which in turn suggests that these frameworks cannot benefit from formal methods that have proven valuable in understanding, characterizing and optimizing stream processing systems in other domains.

B. Contributions and Outline

The contributions of this work are twofold. First, we present a stream algebra that allows efficient implementation of state-of-the-art computer vision pipelines processing streaming data. Second, we define a set of *concurrent* algebraic operators, which revise the previous operators found in text stream algebras, in such a way to develop an abstraction suitable for mapping computer vision pipelines, and to meet the recent processing patterns used in frameworks such as Gstreamer and Storm. The algebra includes operators for both data processing and flow control, which can be used to build scalable vision pipelines. To the best of our knowledge, ours is the first attempt at developing a stream algebra for computer vision tasks.

The paper is organized as follows: Section II describes the related work. We present the stream algebra in Section III. Then in the following section we describe vision pipelines corresponding to several state-of-the-art computer vision tasks. We discuss the implications of the proposed stream algebra in Section V. Finally, Section VI concludes the paper.

[3]Stream processors must by necessity be online algorithms.

II. Background Literature

This section briefly reviews 1) existing work on stream algebras and 2) frameworks that have been developed within the computer vision and image processing communities to construct processing pipelines for streaming (online) computer vision systems.

A. Stream Algebra in Databases

Database community has been fascinated with formulating algebras for data streams, where data streams are defined as an infinite sequence of tuples. For example, Broy *et al.* [5] defined streaming pipelines as data flow networks. They studied the algebra of these networks and related it to the calculus of flownomials that depends on the basic network algebra. The core of the algebra is a set of algebraic operators that can construct data flow networks as graphs of stream processing functions.

Carlson and Lisper [6] presented an event detection algebra for reactive systems in which the system should respond to external events and produce corresponding actions. The algebra focused on composite events and consisted of five operators: i) disjunction, ii) conjunction, iii) negation, iv) sequence, and v) temporal restriction. The authors also provided an *imperative* algorithm to evaluate the algebraic expressions and produce the corresponding output streams.

Demers *et al.* [7] proposed another algebra that extends event algebras to describe queries on event streams. The algebraic extension includes a set of stream operators with well defined semantics for stock quote streams. The operators include unary operators, binary operators, and aggregators.

B. Stream Processing in Computer Vision

A number of recent vision applications use stream processing concepts for online image and video analysis. These applications span several areas such as the analysis of community photos, activity recognition, video surveillance, and satellite imagery. For example, Schuster *et al.* [23] proposed a method for online detection of unusual regions in surveillance video streams. The method partitions each image and extracts a local model for each partition. These models are continuously updated (according to a set of heuristics) in response to scene changes. The method was applied to guide camera operators to the areas of interest.

Gunhee *et al.* [24] proposed another method for multiple foreground cosegmentation of similar objects within an image stream. This method oversegments each image into a set of segments, which are grouped using an iterative scheme into a k region foreground model. The algorithm is applied to Flickr[4] photostreams and the ImageNet dataset, and the results are promising.

[4]Flickr - http://www.flickr.com (*last accessed on 20 March 2014*).

Cao *et al.* [25] proposed a method for recognizing human activities from video streams in which part of the activities are missing. They cast the problem within a probabilistic framework and use sparse coding to calculate the likelihoods of a test video toward a set of trained activities. Finally, the activity with the maximum likelihood is selected.

III. Proposed Stream Algebra

The stream algebra is defined using the common algebraic definitions of Communicating Sequential Processes (CSPs) and Stream Processing (SP). These definitions have shown useful in several contexts. For example, CSPs influenced the design of the concurrency model of the Go language [26] and UNIX pipes, where SP were used to handle stream queries in Microsoft StreamInsight [27]. The stream algebra has three main parts, notation of a common streaming model shared by the operators, the operators and the formal semantics that define correct pipeline expressions.

A. Notation

We define an algebra for declaratively constructing the vision pipeline.

Definition 1 (Data streams)**.** *A data stream is an infinite sequence of data. Given a stream s, there are two functions for write to, and read from the stream:*

$$\lambda x : x \rightarrow s$$

$$\leftarrow s$$

The set of all possible streams are denoted by $\mathbf{S}$. To signify that a stream is to contain data of a particular type, we will use the generic type notation of $\mathbf{S}\langle T \rangle$ where T is a data type.

Definition 2 (Operators)**.** *A stream operator is a function that maps m input streams to n output streams. (Typically $n = 1$.)*

$$h : \mathbf{S}^m \rightarrow \mathbf{S}^n : S^1_{\text{in}}, \ldots, S^m_{\text{in}} \rightarrow S^1_{\text{out}}, \ldots, S^n_{\text{out}}$$

The definitions of the standard operators are limited to following constructs:

- Shared states:
  state u
  indicates that u is a state for subsequent loops.
- Concurrency:
  loop : *body of loop*
  iterates over the body forever. Note that each loop runs in its own concurrent container (e.g. threads), but all loops of the *same* operator share the states. If there are multiple concurrent loops, we may use $\textbf{loop}_j$ to indicate that it's the j-th concurrent session.
- Atomicity:
  { *statements* }
  executes the statements as an atomic operation[5].
- Stream I/O:
  $x \leftarrow s$ reads from a stream and save the result in x, and $e \rightarrow s$ writes the expression e to stream.

B. First-order operators

We begin the algebra with some classical stream operators. We will illustrate the formal notation we use to define our operators. The operator definitions all follow the following style:

- **Declaration:** an operator, X may be parameterized by zero or more user specified functions. If there are functional parameters, we indicate the functional parameters, we indicate the functional signatures of each parameter, and then show the *derived* operator instance as a stream operator in the following format:

$$\frac{f_1 : \text{signature}_1, \ldots, f_k : \text{signature}_k}{X(f_1, \ldots, f_k) : \mathbf{S}^m \rightarrow \mathbf{S}^n}$$

- **Implementation:** we define the semantics of the derived operator using the syntactic constructs of shared states, concurrency, atomicity and stream I/O.

Map is an operator which *synchronously* reads from k incoming streams to build a k-dimensional vector of readings, and apply a user-defined function to compute the value to be written out to the outgoing stream.

$$\frac{f : X_1 \times X_2 \times \ldots X_k \rightarrow Y}{\text{MAP}(f) : \mathbf{S}\langle X_1 \rangle \times \ldots \mathbf{S}\langle X_k \rangle \rightarrow \mathbf{S}\langle Y \rangle}$$

$$\textbf{loop} \quad : \quad f(\leftarrow S^1_{\text{in}}, \ldots, \leftarrow S^k_{\text{in}}) \rightarrow S_{\text{out}}$$

This operator is parametrized by a simple function f that maps elements from k input streams to one output stream.

Reduce is an operator which maintains internal states. For each reading from the incoming stream, reduce updates the state and generates an output for the outgoing stream. Reduce operators are parametrized by the function that updates the internal state and computes the output value based on the input and the previous state.

$$\frac{u_0 : U \quad , \quad g : U \times X \rightarrow U \times Y}{\text{REDUCE}(g, u_0) : \mathbf{S}\langle X \rangle \rightarrow \mathbf{S}\langle Y \rangle}$$

$$\begin{aligned} \textbf{state} \quad & u = u_0 \\ \textbf{loop} \quad : \quad & u, y = g(u, \leftarrow S_{\text{in}}) \\ & y \rightarrow S_{\text{out}} \end{aligned}$$

Copy makes duplicates of the incoming stream. It is important to observe that **Copy** writes its output synchronously

[5]It is interesting to note that the system level implementation of atomicity can range from the classical semaphores to software transaction memory [28].

on all duplicated outgoing streams. We will see that the duplicated outgoing streams *can* be transformed into asynchronous streams via other operators such as **Cut** and **Latch**. The **Copy** operator has no parameters. $\text{COPY} : \mathbf{S} \to \mathbf{S}^n$

$$\begin{array}{rll} \textbf{loop} & : & x \leftarrow S_{\text{in}} \\ & & x \to S^i_{\text{out}} \quad \text{for all } i \leq n \end{array}$$

Filter only keeps certain readings from the incoming stream. It's parameterized by a predicate.

$$\frac{\theta : X \to \text{boolean}}{\text{FILTER} : \mathbf{S}\langle X\rangle \to \mathbf{S}\langle X\rangle}$$

$$\begin{array}{rll} \textbf{loop} & : & x \leftarrow S_{\text{in}} \\ & & \textbf{if } \theta(x) \textbf{ then } x \to S_{\text{out}} \end{array}$$

C. Rate controlling operators

Latch operator allows the incoming and outgoing streams to be asynchronous (namely transmitting at different rates). It does so by remembering the most recent incoming reading, and aggressively write it to the outgoing stream whenever writes are possible. The streaming rate of the incoming and outgoing streams are *unrelated*. $\text{LATCH} : \mathbf{S} \to \mathbf{S}$.

$$\begin{array}{rll} \textbf{state} & u & \\ \textbf{loop}_1 : & x \leftarrow S_{\text{in}} & \textbf{loop}_2 : \{u \to S_{\text{out}}\} \\ & \{u = x\} & \end{array}$$

Cut operator decouples the outgoing stream from the input stream, just like **Latch**. The difference is that **Cut** does not latch on the last reading if the outgoing stream has a higher streaming rate. Instead, **Cut** guarantees that every incoming reading is written *once* to the outgoing stream. A **nil** value is used for the extra write operations. $\text{CUT}() : \mathbf{S} \to \mathbf{S}$

$$\begin{array}{rll} \textbf{state} & u = \textbf{nil} & \\ \textbf{loop} : & x \leftarrow S_{\text{in}} & \textbf{loop} : \{y = u\ ; u = \textbf{nil}\} \\ & \{u = x\} & y \to S_{\text{out}} \end{array}$$

Left multiply operator reads from two incoming streams S^1_{in} and S^2_{in}, and outputs pairs (x_1, x_2) to the outgoing stream. Unlike **Map**, **left multiple** synchronizes the writes with S^1_{in}, and latches on S^2_{in}. This means that the outgoing rate is determined by the incoming rate of S^1_{in}, and independent of S^2_{in}. One can think of S^2_{in} as a clock stream which triggers the samples of S^2_{in}. Thus, **left multiple** is a generalized sampling operator. $\text{MULT} : \mathbf{S}\langle X_1\rangle \times \mathbf{S}\langle X_2\rangle \to \mathbf{S}\langle X_1 \times X_2\rangle$

$$\textbf{loop} \quad : \quad \begin{bmatrix} \leftarrow S^1_{\text{in}} \\ \leftarrow \text{LATCH}(S^2_{\text{in}}) \end{bmatrix} \to S_{\text{out}}$$

The **right multiply** can be defined similarly.

Add operator merges multiple incoming streams in a greedy fashion. It performs best effort reads on the incoming streams asynchronously, and writes to the outgoing stream. $\text{ADD} : \mathbf{S} \times \mathbf{S} \to \mathbf{S}$.

$$\begin{array}{rll} \textbf{loop} : & x \leftarrow S^1_{\text{in}} & \textbf{loop} : \quad x \leftarrow S^2_{\text{in}} \\ & x \to S_{\text{out}} & \qquad\quad x \to S_{\text{out}} \end{array}$$

D. Higher order operators

In Section III-B, the operators are functions with *streams* as inputs and outputs, and are parameterized by simple functions (as in the cases of MAP and REDUCE). Collectively, they are called first-order operators. Any composition of first-order operators are first order operators as well.

In this section, we extend the operators to include *second-order* operators. These operators have *collections* of streams as inputs and outputs, and are parameterized by functions and first-order operators.

Scatter reads from an incoming stream, but generates a list of outgoing streams. The list of outgoing streams can be arbitrary size. **Scatter** is parameterized by an output generator $f : X \to \text{LIST}\langle Y\rangle$ that computes the list of output values to be emitted from the value read from the incoming stream. **Scatter** is also parameterized by a partition function $p : Y \to \mathbb{N}$ which maps the output values y to the $p(y)$-th stream in the outgoing streams.

$$\frac{f : X \to \text{LIST}\langle Y\rangle \quad , \quad p : Y \to \mathbb{N}}{\text{SCATTER}(f,p) : \mathbf{S}\langle X\rangle \to \text{LIST}\langle \mathbf{S}\langle X\rangle\rangle}$$

$$\begin{array}{rl} \textbf{let } \mathcal{S}_{\text{out}} = \text{EMPTY-LIST}\langle \mathbf{S}\langle X\rangle\rangle & \\ \textbf{loop} & : \mathbf{y} = f(\leftarrow S_{\text{in}}) \\ & \quad y_i \to \mathcal{S}_{\text{out}}[p(y_i)] \quad \text{for all } y_i \in \mathbf{y} \end{array}$$

List map is a higher order operator whose input and output are collections of streams. With **scatter**, one can generate collections of streams, and with first order operators, one can build up stream pipelines using composition. **List map** allows one to apply first-order stream pipelines to collections of streams.

$$\frac{h : \mathbf{S}\langle X\rangle \to \mathbf{S}\langle Y\rangle}{\text{LIST-MAP}(h) : \text{LIST}\langle \mathbf{S}\langle X\rangle\rangle \to \text{LIST}\langle \mathbf{S}\langle Y\rangle\rangle}$$

$$\begin{array}{rl} \text{LIST-MAP} : & \mathcal{S}_{\text{in}} \mapsto \mathcal{S}_{\text{out}} \\ & \textbf{let } \mathcal{S}_{\text{out}} = \text{EMPTY-LIST}\langle \mathbf{S}\langle Y\rangle\rangle \\ & \text{for all } i \leq \textbf{len}(\mathcal{S}_{\text{in}}) \\ & \qquad \textbf{loop} : \quad y \leftarrow h(\mathcal{S}_{\text{in}}[i]) \\ & \qquad\qquad\quad y \to \mathcal{S}_{\text{out}}[i] \end{array}$$

Merge operator is the "inverse" of **scatter** in the sense that it merges a collection of incoming streams back into a single outgoing stream. It reads from all the n incoming streams of type X in the input collection into a buffer of size n (one slot for each incoming stream). A selection function is used to pick the element in the buffer to be written to the outgoing stream. The selection function $f : X \to (Y, \preceq)$ has a partial order over Y which is used to determine that a minimal element (w.r.t. $\preceq$) is to be removed from the buffer and written to the output stream.

$$\frac{f : X \to (Y, \preceq)}{\text{MERGE}(f, \preceq) : \text{LIST}\langle \mathbf{S}\langle X\rangle\rangle \to \mathbf{S}\langle X\rangle}$$

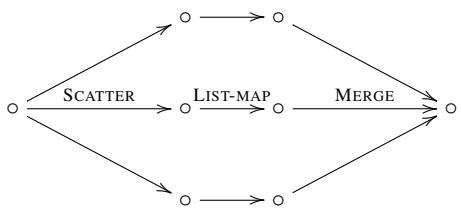

Figure 1. An example of a concurrency pattern expressed in our algebra.

$\text{MERGE}(f) : \mathcal{S}_{\text{in}} \mapsto S_{\text{out}}$
State : B where $|B| = |\mathcal{S}_{\text{in}}|$.
for each $S_{\text{in}} = \mathcal{S}_{\text{in}}[i]$:
loop : $\{$**if** $B[i] ==$ **nil then** $B[i] \leftarrow S_{\text{in}}\}$
end for
loop : **if nil** $\notin B$ **then**
$i^* = \text{argmin}_{\preceq}\{f(B[i])\}$
$\{B[i^*] \rightarrow S_{\text{out}}; B[i^*] =$ **nil**$\}$
end if

The higher order functions allow our algebra to generate and work with collections of streams. This is particularly important when working with stream runtime environments with high degrees of concurrency. By utilizing patterns such as the one in Figure 1, the algebra can express utilization of concurrent processing environments such as distributed server clusters, multicore processors and GPGPU processing quite naturally.

We will demonstrate in Section V that the higher order operators can be instrumental in automatic optimization of first-order stream processing. Namely, user can express the processing using simple MAP, which can be automatically compiled into an equivalent, but more optimal execution plan involving SCATTER, LIST-MAP and MERGE.

IV. COMPUTER VISION ALGORITHMS

There is a continuous need for computer vision algorithms that can process large scale vision streams in realtime or provide a series of approximate results, that are improved over time. In this section, we present two state-of-the-art algorithms, that successfully applied data streaming concepts for processing vision streams. Particularly and without loss of generality, we will show how our stream algebra can effectively describe the vision pipelines of these algorithms using a set of equations over data streams.

A. Activity Recognition

The goal of activity recognition is to identify the type of actions that are occurring between one or more objects in a sequence of images. Ryoo [13] presented a recent algorithm for early prediction of human activities. They addressed six human activities: hand shaking, hugging, kicking, pointing, punching and pushing. The algorithm operates in two main stages, an offline learning stage and an online prediction stage.

In the offline learning stage, the algorithm receives as input, a group of human activities and a set of training videos for each human activity. Then, 3-D spatio-temporal features are extracted from each video. A Bag-of-Words (BoW) model is then constructed by clustering the features of all videos into k visual words. This model is used to build an integral histogram for each training video. The integral histogram is a sequence of visual word histograms $H_i = (H_i^0, H_i^1, ..., H_i^j, ...)$ where H_i^j is the accumulated histogram of visual words in video i up to frame j, and $k = |H_i^j|$. The algorithm then defines one integral histogram for each human activity by averaging the integral histograms of all its input videos. The output integral histogram works as an activity model and we will refer to all output models by the set D.

In the online prediction stage, the algorithm receives an input video stream. This stream is divided into a sequence of clips $C = \{C_q | q = 0, 1, ...\}$ of duration $\triangle t$. For each clip C_q, the algorithm extracts the 3-D spatio-temporal features and converts them into visual words using the learned BoW model. Then, an integral histogram $H = (H^0, H^1, ..., H^q, ...)$ is constructed where H^q is the accumulated histogram of visual words up to the qth clip. As time progress, the algorithm uses dynamic programming to compare the integral histogram H towards the activity models in D and generates a likelihood stream $L = (L^0, L^1, ..., L^q, ...)$, where L^q is the accumulated likelihood vector up to the qth clip. Note that each activity has an element in L^q. The algorithm then outputs an activity stream $A = (A^0, A^1, ..., A^q, ...)$, where $A^q = \arg\max_{0 \leq i \leq |L^q|} L_i^q$ is the activity with the maximum likelihood value in L^q. Now we will describe the online prediction stage using our algebra. The data types defined by the algorithm are,

`Frame` : `2DImage`
`Feature` : $\mathbb{R}^l$
`Video` : $\mathbf{S}\langle$`Frame`$\rangle$
`Clip` : LIST $\langle$`Frame`$\rangle$

where a `Frame` is a single 2D image, a `Feature` is a vector in a high dimensional space $\mathbb{R}^l$, a `Video` is a stream of frames, and a `Clip` is a list of frames that represent a certain time interval in a `Video`. Given the data types, we start by dividing an incoming video stream $V \in$ `Video` into a stream of non-overlapping clips using the function,

$g : \texttt{Clip} \times \texttt{Frame} \rightarrow \texttt{Clip} \times \texttt{Clip}$

$$g(u, x) = \{ \quad \text{if duration}(u) \geq \triangle t \text{ then}$$
$$u' = \emptyset; y = u$$
$$\text{else}$$
$$u' = u \oplus x \quad \text{//append } x \text{ to clip } u$$
$$y = \emptyset$$
$$\text{return}(u', y)$$
$$\}$$

This function keeps appending the incoming frames to a clip u' and setting the output y to an empty clip. When the clip size is greater or equals to a time interval $\triangle t$, the function copies the clip to the output y and resets u' back to an empty list. We can now use the function g with a REDUCE operator to obtain the stream,

$$\bullet_{\emptyset} \rightarrow \ldots \rightarrow \bullet_{C^0} \rightarrow \bullet_{\emptyset} \rightarrow \ldots \rightarrow \bullet_{C^1} \rightarrow \cdots$$

where the arrow indicates time direction. This stream can be filtered using a FILTER operator to remove the empty elements and obtain the clip stream $C : \mathbf{S}\langle\texttt{Clip}\rangle$,

$$C \triangleq \textsc{Filter}(\lambda\, x :\ |x| \neq 0) \circ \textsc{Reduce}(g, \emptyset)(V)$$
$$\bullet_{C^0} \rightarrow \bullet_{C^1} \rightarrow \cdots \rightarrow \bullet_{C^q} \rightarrow \cdots$$

where the $\circ$ operator takes the output stream from the right operand and feeds it as an input stream to the left operand. The algorithm then converts each clip in C to a list of features using the function $f_1 : \texttt{Clip} \rightarrow \textsc{List}\langle\texttt{Feature}\rangle$, which can be used with a MAP operator to obtain the features stream $F : \mathbf{S}\langle\textsc{List}\langle\texttt{Feature}\rangle\rangle$,

$$F \triangleq \textsc{Map}(f_1)(C)$$

Another algorithm function $f_2 : \textsc{List}\langle\texttt{Feature}\rangle \rightarrow \textsc{List}\langle\texttt{Word}\rangle$ is defined by [13] and uses the learned BoW model to transform every feature from the input list to a corresponding word in the output list. This function can be used with a MAP operator to define the word stream $W : \mathbf{S}\langle\textsc{List}\langle\texttt{Word}\rangle\rangle$,

$$W \triangleq \textsc{Map}(f_2)(F)$$

The word stream is then used to construct an integral histogram by using the function,

$$g_2 : \text{Histogram} \times \textsc{List}\langle\texttt{Word}\rangle \rightarrow \text{Histogram} \times \text{Histogram}$$
$$g_2(u, x) = \{ \quad u' = u + x$$
$$\text{return}(u', u')$$
$$\}$$

where $+$ accumulates the input list of words on the histogram u to construct a new histogram u', which is copied to the output. So, the function g_2 can be used together with a REDUCE operator to define a histogram stream H which is itself an integral histogram,

$$H \triangleq \textsc{Reduce}(g_2, \text{empty-histogram})(W)$$
$$\bullet_{H^0} \rightarrow \bullet_{H^1} \rightarrow \cdots \rightarrow \bullet_{H^q} \rightarrow \cdots$$

The algorithm then calculates a likelihood score for each histogram H^q in the stream H. This score is calculated using a function $d : \text{Histogram} \times \text{ActivityModel} \rightarrow \mathbb{R}$ that calculates the distance between H^q to an activity model from the set $D \in \textsc{List}\langle\text{ActivityModel}\rangle$ of learned activity models (see [13] for details). The d function is used to define a new function,

$$f_3 : \text{Histogram} \rightarrow \textsc{List}\langle\mathbb{R}\rangle$$
$$f_3(x) = \{\text{return } [d(x, D[i]) \text{ for } i \leq |D|]\}$$

that returns a likelihood vector for each histogram H^q in the stream H. Note that this vector has one entry for each activity. So, the function f_3 can be used with a MAP operator to define a likelihood stream $K : \mathbf{S}\langle\textsc{List}\langle\mathbb{R}\rangle\rangle$,

$$K \triangleq \textsc{Map}(f_3)(H)$$
$$\bullet_{K^0} \rightarrow \bullet_{K^1} \rightarrow \cdots \rightarrow \bullet_{K^q} \rightarrow \cdots$$

The algorithm of [13] then defines a function for Bayesian combination of likelihoods $f_4 : \textsc{List}\langle\mathbb{R}\rangle \times \textsc{List}\langle\mathbb{R}\rangle \rightarrow \textsc{List}\langle\mathbb{R}\rangle$, which can be used to define the function,

$$g_3 : \textsc{List}\langle\mathbb{R}\rangle \times \textsc{List}\langle\mathbb{R}\rangle \rightarrow \textsc{List}\langle\mathbb{R}\rangle \times \textsc{List}\langle\mathbb{R}\rangle$$
$$g_3(u, x) = \{ \quad u' = f_4(u, x)$$
$$\text{return}(u', u')$$
$$\}$$

which keeps returning an accumulated likelihood vector and can be used with a reduce operator to obtain the accumulated likelihood stream $L : \mathbf{S}\langle\textsc{List}\langle\mathbb{R}\rangle\rangle$,

$$L \triangleq \textsc{Reduce}(g_3, \mathbf{0})(W)$$
$$\bullet_{L^0} \rightarrow \bullet_{L^1} \rightarrow \cdots \rightarrow \bullet_{L^q} \rightarrow \cdots$$

and $\mathbf{0}$ is the zero vector. Finally a function $f_5 = \lambda x : \arg\max_{i \leq |x|} x_i$ is applied to each vector in L to return the index of the activity of maximum likelihood. We can use f_5 as a parameter to a final MAP operator and obtain the output activity stream $A : \mathbf{S}\langle\mathbb{R}^+\rangle$,

$$A \triangleq \textsc{Map}(f_5)(L)$$
$$\bullet_{A^0} \rightarrow \bullet_{A^1} \rightarrow \cdots \rightarrow \bullet_{A^q} \rightarrow \cdots$$

B. Other vision problems

Due to space limitation, we briefly describe the algebra of another example algorithm that implements hierarchical video segmentation [15] in a streaming fashion. This algorithm starts by dividing the input video into a sequence of non-overlapping clips of duration $\triangle t$. Each clip is represented as a 3D space-time volume and is segmented into a collection of 3D space-time segments. Then, hierarchical clustering is applied on these segments to generate a segmentation hierarchy. The method also uses a Markovian assumption to build the hierarchy of the current clip using the hierarchy generated for the previous clip. In order to map this technique into our algebra, we will reuse the definitions of the previous section. Where again, we have an input video $V \in \texttt{Video}$ that is divided into a stream of non-overlapping clips $C : \mathbf{S}\langle\texttt{Clip}\rangle$. This can be done using the same $\textsc{Reduce}(g, \emptyset)(V)$ operator used in the algebra of the previous algorithm. The core of [15] is a function f_6 : Hierarchy $\times$ `Clip` $\rightarrow$ Hierarchy that maps each clip from the stream C, together with the segmentation hierarchy computed for the previous clip, to a new segmentation hierarchy. This function can be used to define a function g_4 : Hierarchy $\times$ `Clip` $\rightarrow$ Hierarchy $\times$ Hierarchy, that has a similar body to the function g_3 with only replacing f_4 with f_6. The function g_4 can be used with a REDUCE operator to define the output stream of segmentation hierarchies $H : \mathbf{S}\langle\text{Hierarchy}\rangle$ where,

$$H \triangleq \textsc{Reduce}(g_4, \text{empty-hierarchy})(C).$$

Other algorithms [4], [18], [14], that span different computer vision problems, can be similarly defined using our algebra.

V. Discussions

The examples shown in the previous section, suggest how our stream algebra may fit naturally in expressing a wide range of computer vision algorithms that manipulate different vision streams. In each discussed algorithm, we define the vision pipeline using a set of formal equations that use a common set of abstract operators. Thereby providing an abstract description that better illustrates the semantics of computer vision operations, and helps in scaling up the techniques for big data analysis. For example, the algebraic description of the activity prediction example highlights the most salient algorithmic functions. These functions are f_1 for feature extraction, f_2 for transforming features using the learned BoW model, f_3 for likelihood estimation using a statistical model, and f_4 for accumulating likelihoods using a Bayesian framework. The remaining operations of [13] can be efficiently and precisely expressed in our algebra.

In addition, The algebra allows several optimizations on the vision pipelines, that can maximize performance and enhance resource allocation. For example, a REDUCE operator followed by a MAP operator can both be replaced by one equivalent REDUCE operator that apply the map function to its output. Another optimization can replace the MAP operator with the concurrency pattern shown in figure 1. Thereby, maximizing the throughput of a heavyweight map operation. One can see that the two previous optimizations are conflicting and it is up to the runtime to select the optimal execution plan that maximize the overall pipeline performance. The algebra also provides a group of operators that can control the data flow rates such as CUT, LATCH, COPY, ADD and LEFT MULTIPLE. These operators allow flexible integration of different computer vision algorithms, that work at different data rates, without interfering one another. They also can support realtime streaming by efficiently implementing blocking resolution and rate matching. This is important for synchronizing between different processing tasks in streaming pipelines and can help in dealing with unbounded data rates of large (and possibly infinite) scale data.

Our stream algebra opens new research directions in computer vision. For example, feedback control loops can be expressed using the COPY and ADD operators which allow adaptive parameters selection, performance tuning and resource reallocation. Pipeline instrumentation is another problem, which can enable real-time debugging, performance monitoring and bottleneck identification. The algebra also opens new directions in stream clustering and on the fly class discovery. For example, the SCATTER operator can be seen as a clustering operator that clusters the input data stream into different output stream clusters.

Limitations: The selection of stream operators presented here is the result of a careful study of stream algebras found in the database literature. While we cannot prove it, we expect that the proposed set of stream operators is able to describe a variety of computer vision applications. Much work is still needed to show that the proposed set of operators is minimal and complete with respect to a given set of vision applications. It is also not immediately obvious how the proposed stream algebra can dynamically tune application specific vision algorithms embedded within the stream operators to guarantee quality of service.

VI. Conclusion

This paper develops a first of its kind stream algebra for formally defining computer vision pipelines—online vision systems that deal with incoming image and video data (vision streams). The algebra treats vision streams as operands and defines a set of concurrent operators, which can succinctly describe computer vision pipelines. We have demonstrated the expressiveness of our algebra by describing two state-of-the-art computer vision application. The proposed algebra brings forth formal methods to design, analyze and optimize vision pipelines. Such algebras are urgently needed as we move towards "big data" vision systems. Our work opens up many exciting avenues for research, including stream

clustering, feedback control, instrumentation, performance tuning and global optimization and dynamic reconfiguration of vision pipelines.

REFERENCES

[1] B. Zhao, L. Fei-Fei, and E. Xing, "Online detection of unusual events in videos via dynamic sparse coding," in *CVPR*, Colorado Springs, June 2011, pp. 3313–3320.

[2] A. Meghdadi and P. Irani, "Interactive exploration of surveillance video through action shot summarization and trajectory visualization," *IEEE Trans. on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2119–2128, Dec 2013.

[3] S. Yenikaya, G. Yenikaya, and E. Düven, "Keeping the vehicle on the road: A survey on on-road lane detection systems," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 2:1–2:43, Jul. 2013.

[4] C. Loy, T. Hospedales, T. Xiang, and S. Gong, "Stream-based joint exploration-exploitation active learning," in *CVPR*, 2012, pp. 1560–1567.

[5] M. Broy and G. Stefanescu, "The algebra of stream processing functions," *Theoretical Computer Science*, vol. 258, no. 1-2, pp. 99 – 129, 2001.

[6] J. Carlson and B. Lisper, "An event detection algebra for reactive systems," in *Proceedings of the 4th ACM International Conference on Embedded Software*, 2004, pp. 147–154.

[7] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "A general algebra and implementation for monitoring event streams," Cornell University, Technical Report, 2005.

[8] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *VLDB*, vol. 29, 2003, pp. 81–92.

[9] A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl, "Moa: Massive online analysis, a framework for stream classification and clustering," in *JMLR*, 2010, pp. 44–50.

[10] D. Wang, E. A. Rundensteiner, and T. R. I. Ellison, "Active complex event processing over event streams," *VLDB*, vol. 4, no. 10, pp. 634–645, Jul. 2011.

[11] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom, "Characterizing memory requirements for queries over continuous data streams," Stanford InfoLab, Technical Report 2002-29, May 2002.

[12] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: Fault-tolerant and scalable joining of continuous data streams," in *SIGMOD*, New York, NY, USA, 2013, pp. 577–588.

[13] M. S. Ryoo, "Human activity prediction: Early recognition of ongoing activities from streaming videos," in *ICCV*, Barcelona, Spain, 2011, pp. 1036–1043.

[14] C. Lu, J. Shi, and J. Jia, "Online robust dictionary learning," in *IEEE CVPR*, 2013, pp. 415–422.

[15] C. Xuand, C. Xiong, and J. Corso, "Streaming hierarchical video segmentation," in *ECCV*, vol. VI, 2012, pp. 626–639.

[16] N. Harbi and Y. Gotoh, "Spatio-temporal human body segmentation from video stream," in *Computer Analysis of Images and Patterns*. Springer, 2013, vol. 8047, pp. 78–85.

[17] J. Yang, J. Luo, J. Yu, and T. Huang, "Photo stream alignment and summarization for collaborative photo collection and sharing," *IEEE Trans. on Multimedia*, vol. 14, no. 6, pp. 1642–1651, 2012.

[18] G. Kim and E. Xing, "Jointly aligning and segmenting multiple web photo streams for the inference of collective photo storylines," in *CVPR*, 2013, pp. 620–627.

[19] A. Kinesis, aws.amazon.com/kinesis/, accessed: 2014-02-27.

[20] G. Chkodrov, P. Ringseth, T. Tarnavski, A. Shen, R. Barga, and J. Goldstein, "Implementation of stream algebra over class instances, Google patents," Patent US20 130 014 094 A1, jan, 2013.

[21] GStreamer, http://gstreamer.freedesktop.org, accessed: 2014-01-26.

[22] Twitter's Storm, http://storm.incubator.apache.org, accessed: 2014-01-26.

[23] R. Schuster, R. Mörzinger, W. Haas, H. Grabner, and L. V. Gool, "Real-time detection of unusual regions in image streams," in *Proceedings of the International Conference on Multimedia*, 2010, pp. 1307–1310.

[24] K. Gunhee and E. Xing, "On multiple foreground cosegmentation," in *CVPR*, Providence, USA, June 2012, pp. 837–844.

[25] Y. Cao, D. Barrett, A. Barbu, S. Narayanaswamy, H. Yu, A. Michaux, Y. Lin, S. Dickinson, J. Siskind, and S. Wang, "Recognize human activities from partially observed videos," in *CVPR*, Oregon, USA, June 2013, pp. 2658–2665.

[26] A. Vajda, *Programming Many-Core Chips*. New York Dordrecht Heidelberg London: Springer, 2011.

[27] M. H. Ali, B. Chandramoul, B. S. Raman, and E. Katibah, "Spatio-temporal stream processing in microsoft streaminsight." *IEEE Data Eng. Bull.*, vol. 33, no. 2, pp. 69–74, 2010.

[28] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.