A Formal Algebra Implementation for Distributed Image and Video Stream Processing

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

We are interested in building scalable computer vision systems for distributed processing of big visual data. We apply data streaming concepts, namely stream algebra operators, which have been proven effective in the database literature. The operators collectively form an algebra over data streams. The algebra has well defined semantics. It naturally describes online computer vision algorithms and their feedback control and tuning algorithms.

In this work, we present the first implementation of such algebra at large scale. Our implementation provides a high level programming interface for constructing and executing vision workflow graphs while hiding the data transfer and concurrency details. It also allows feedback control and dynamic reconfiguration of vision algorithms. A case study is discussed showing a streaming workflow for online lane and road boundary detection and describing the flexibility and effectiveness of the algebra for building complex distributed applications.

Keywords

stream algebra; workflow graphs; programming frameworks; distributed vision processing

1. INTRODUCTION

The recent advances in our ability to store, compute, share and consume data introduced what is often referred to as the *big data* era. This brought several challenges in many computing areas to develop scalable algorithms and frameworks capable of processing big data. The computer vision area is of no exception due to the continuous growth of applications generating vast volumes of visual data. Examples of such applications include satellite imagery, video surveillance, and online photo and video sharing websites (e.g Flickr[1]). As these applications usually produce a stream of data, stream processing becomes an important research direction for handling the growth of data. This direction was also motivated by the ability to express many computer vision algorithms as streaming online methods. For example, several works have been proposed for streaming hierarchical video segmentation [20], human body segmentation from video stream [10], photo stream alignment [16], and inference of storylines from web photo streams [21, 14]. These methods are formulated as pipelines (or workflows) that process image or video streams, which we refer to as *Vision Streams*. The pipeline concept is borrowed from the database literature where it was found useful in scaling up algorithms, however, the methods lack a formal definition of pipelines. This limits the ability to utilize and integrate these algorithms into larger systems. This problem has been solved in databases using stream algebras [4, 5, 8, 7].

A stream algebra is a formal language for mathematically defining workflow graphs. It defines a set of abstract and concurrent operators that take data streams as their operands to produce output streams. The operators have formal semantics to declare and construct streaming workflows as mathematical expressions. This formal definition provides the advantage of building general methods for optimizing workflow performance, implementing feedback control and enabling dynamic reconfiguration of operators.

The database stream algebras are only applied to structured data such as text, and cannot describe processing of vision streams that have an unstructured content. A recent work by Helala *et al.* [12, 11] proposed a stream algebra for describing computer vision workflows. The algebra provided a set of formal operators for both stream processing and flow control. Although this algebra is only theoretically defined, it was shown usefully in describing several online vision methods [18, 20, 17, 6, 14]. In this paper, we present the first implementation of this algebra. The implementation defines a programming framework for building distributed workflow graphs for computer vision systems using the algebraic operators of [12, 11]. We discuss the architecture of the framework and the programming interface. The framework provides several advantages over current approaches such as feedback control and dynamic reconfiguration of operators at runtime. We present a case study on online lane and road boundary detection from traffic video streams. The results show the running time statistics gathered by our framework at runtime with and without using feedback control and dynamic reconfiguration. Section 2 provides the relevant work and Section 3 provides a summary of the algebra in [12, 11]. Next, Section 4 presents the algebra implementation and Section 5 discusses the case study and results. Finally, Section 6 concludes the paper.

2. RELATED WORK

Stream algebra. There has been a large interest in the database community for developing formal algebras for data stream processing. For example, the work of Demers *et al.* [8] developed a stream algebra as a declarative language for querying event streams. Chkodrov et al. [7] also developed a stream algebra that maps the relational algebra operators into streaming versions. Broy et al. [4] related stream algebras to the calculus of flownomials and basic

[1]Flickr: https://www.flickr.com/ (*last accessed on 1 May 2016*).

ICDSC '16 September 12–15, 2016, Paris, France

DOI: 10.475/123_4

network algebra. They developed a set of algebraic operators for constructing workflow graphs from stream processing functions. Carlson and Lisper [5] presented an algebra for event detection with a set of algebraic operators for manipulating event streams.

Previous systems and frameworks. Several frameworks have been developed for distributed processing of data streams. For example, GStreamer [9] implements multimedia processing pipelines. Storm [19] allows the construction of distributed workflow graphs. Hive [1] constructs distributed vision systems by connecting modules through a plug-in interface. These frameworks, however lacks a formal description of stream processing and the benefits of having a stream algebra.

3. STREAM ALGEBRA

We address workflow graphs expressed using the formal stream algebra of [12]. This algebra consists of three main components: a common notation for expressing workflows, a set of data processing and flow control operators, and the formal semantics used to write workflow expressions. This section gives an overview of the algebra operators and how we can use them to express processing pipelines and fork-join graphs.

Algebra presented in [12] defines a data stream as an infinite sequence of data tuples. Functions $\lambda x \rightarrow s$ and $\leftarrow s$ can be used to *write to* and *read from* a data stream, respectively. The set $\mathbf{S}$ is a the set of all streams and a stream operator is a function $h : \mathbf{S}^m \rightarrow \mathbf{S}^n : S^1_{\text{in}}, \ldots, S^m_{\text{in}} \rightarrow S^1_{\text{out}}, \ldots, S^n_{\text{out}}$ that maps n input streams to m output streams.

3.1 Notation

The following constructs are used by [12] to define operators:

- *Shared State*: a shared state is defined as **state** u. This indicates that the shared state u is accessed by the following loops.
- *Atomicity*: we define a set of statements executing as an atomic operation using { *statements* }.
- *Concurrency*: an infinite loop is defined as **loop** : *body_of_ loop*. The loop applies the body logic iteratively on input stream tuples. The loop runs in its own thread. If an operator defines several concurrent loops, they all share the defined states. $\textbf{loop}_j$ designate the j-th loop.
- *Stream I/O*: the function $x \leftarrow s$ reads a tuple from stream s into x, and the function $e \rightarrow s$ writes a tuple e to stream s.
- *Attribute Access*: we use $x.y$ to access attribute y from composite variable x.

A stream operator is a mapping function that can have zero or more parameters. The parameters can be assigned values or simple functions. In this paper, we also attach to each operator, the attribute `opid` which defines a unique identifier for each operator. Stream operators are the fundamental building blocks for workflow graphs. A workflow graph $G = (V, E)$ is a Directed Acyclic Graph (DAG) with vertices V representing operators and edges E representing the direction of data communication. We are interested in two parallel processing workflow patterns: pipeline graphs, and fork-join graphs. For these graphs, we classify the formal operators defined by [12] into pipelined operators and fork-join operators.

3.2 Pipeline Graphs

The pipeline operators are simple first-order operators that can be used to construct pipeline workflow graphs.

The **Source** operator has no input stream and writes to one output stream. It is parametrized by an initial shared state $u_0 : U$ and a generator function $h : U \rightarrow U \times Y$.

$\text{SOURCE}(u_0, h) : \emptyset \rightarrow \mathbf{S}\langle Y \rangle$

$$\begin{array}{lll} \textbf{state} & & u = u_0 \\ \textbf{loop} & : & u, y = h(u) \\ & & y \rightarrow S_{\text{out}} \end{array}$$

The **Map** operator *synchronously* reads from k input streams, performs a user-defined mapping function $f : X \times P \rightarrow Y$ on input tuples, and writes the computed value to an output stream. The mapping function takes the incoming tuple of type X and a set of function parameters of type P. The operator is parametrized by a list of functions $\mathbf{f} : \text{LIST}\langle X \times P \rightarrow Y \rangle$ and an initial vector of parameters $\mathbf{p_0}$. The initial function is by default at index zero. The input tuple may also contain a commands section that records extra information such as the pairs (`opid`, function-index) and (`opid`, parameters). This information can reconfigure the operator to switch to a new function that uses the supplied parameters. Two functions are defined to access this information, $f_i : \mathbb{Z} \times X \rightarrow \mathbb{Z}$ and $f_p : \mathbb{Z} \times X \rightarrow P$. f_i takes `opid` and incoming tuple as inputs and retrieves a new function index if exists and -1 otherwise. f_p takes the same inputs and returns the function parameters vector if exists and `nil` otherwise.

$\text{MAP}(\mathbf{f}, \mathbf{p_0}) : \mathbf{S}\langle X \rangle \times \ldots \mathbf{S}\langle X_k \rangle \rightarrow \mathbf{S}\langle Y \rangle$

$$\begin{array}{lll} \textbf{state} & & i = 0, \quad \mathbf{p} = \mathbf{p_0} \\ \textbf{loop} & : & x =\leftarrow S_{\text{in}} \\ & & \textbf{if } j = f_i(\texttt{opid}, x), j \neq -1 \textbf{ then } i = j \\ & & \textbf{if } \mathbf{z} = f_p(\texttt{opid}, x), \mathbf{z} \neq \textbf{nil then } \mathbf{p} = \mathbf{z} \\ & & \mathbf{f}[i](x, \mathbf{p}) \rightarrow S_{\text{out}} \end{array}$$

The **Reduce** keeps track of an internal shared state $u : U$ and is parametrized by a list of mapping functions $\mathbf{g} : \text{LIST}\langle U \times X \times P \rightarrow U \times Y \rangle$, an initial vector of parameters $\mathbf{q_0}$, and an initial state u_0.

$\text{REDUCE}(u_0, \mathbf{g}, \mathbf{q_0}) : \mathbf{S}\langle X \rangle \rightarrow \mathbf{S}\langle Y \rangle$

$$\begin{array}{lll} \textbf{state} & & i = 0, \quad u = u_0, \quad \mathbf{q} = \mathbf{q_0} \\ \textbf{loop} & : & x =\leftarrow S_{\text{in}} \\ & & \textbf{if } j = f_i(\texttt{opid}, x), j \neq -1 \textbf{ then } i = j \\ & & \textbf{if } \mathbf{z} = f_p(\texttt{opid}, x), \mathbf{z} \neq \textbf{nil then } \mathbf{q} = \mathbf{z} \\ & & u, y = \mathbf{g}[i](u, x, \mathbf{q}) \\ & & y \rightarrow S_{\text{out}} \end{array}$$

The **Ground** operator destroys the incoming stream.

$\text{GROUND} : \mathbf{S} \rightarrow \emptyset$

$$\textbf{loop} \quad : \quad \leftarrow S_{\text{in}}$$

A pipeline graph is constructed from n first-order operators or stages as shown in Figure 1a. Data tuples enter the pipeline sequentially and are processed concurrently by operators. Each operator receives a data tuple, process it, and submits results to output.

3.3 Fork and Join Graphs

The stream algebra provides several operators for flow control using fork and join operators. The input and output streams can be synchronized or asynchronized. If asynchronized, the input and output streams are decoupled and can have different data rates.

3.3.1 Fork Operators

Copy synchronously reads and duplicates every input tuple to n outgoing streams. It is parameterized by the number of output streams n.

$\text{COPY}(n) : \mathbf{S} \rightarrow \mathbf{S}^n$

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

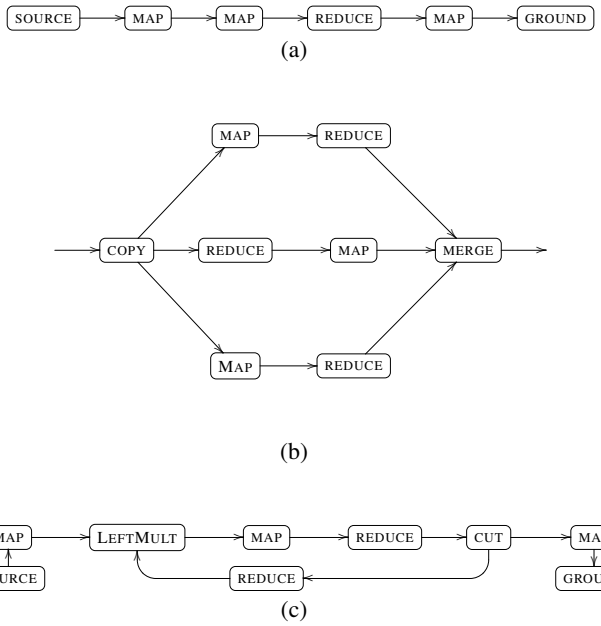


Figure 1: Different examples of workflow graphs. (a) A pipeline graph. (b) A fork-join graph. (c) The pipeline graph in (a) with a feedback control loop.

Latch receives a single input stream and has two output streams. The operator reads from the incoming stream, synchronously writes to S^1_{out}, and asynchronously writes to S^2_{out}. If input is slower than the asynchronous output, data tuples will be duplicated, and if faster, the asynchronous output will sample the input and looses some tuples.

LATCH : $\mathbf{S} \rightarrow \mathbf{S}^2$ defined as:

$$\begin{array}{ll} \textbf{loop}: & x \leftarrow S_{\text{in}} \\ & \{u = x \\ & x \rightarrow S^1_{\text{out}}\} \end{array} \qquad \textbf{loop} : \{u \rightarrow S^2_{\text{out}}\}$$

Cut follows the same logic of **Latch**, however the operator makes sure that the incoming tuples are written only once to the asynchronous output stream. **nil** is used for the extra writes when the output stream has a faster data rate than the input stream.

CUT() : $\mathbf{S} \rightarrow \mathbf{S}^2$

$$\begin{array}{lll} \textbf{state} & u = \textbf{nil} & \\ \textbf{loop}: & x \leftarrow S_{\text{in}} & \textbf{loop}: \quad \{y = u\,; u = \textbf{nil}\} \\ & \{u = x\,; x \rightarrow S^2_{\text{out}}\} & \qquad\quad y \rightarrow S^1_{\text{out}} \end{array}$$

Scatter synchronously receives an input stream and generates a list of output streams. The operator is parameterized by two functions, $f : X \rightarrow \text{LIST}\,\langle Y\rangle$ and $p : Y \rightarrow \mathbb{N}$. f is a generator function that computes output values and p is a partition function that maps each output value y to the $p(y)$-th output stream.

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

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

3.3.2 Join Operators

Mult has k incoming streams S^k_{in}, and one output stream S_{out}. The operator reads one value at a time from each incoming stream, forms a vector $(x_1, ..., x_k)$, and synchronously writes this vector to the outgoing stream. MULT() : $\mathbf{S}^k \rightarrow \mathbf{S}$

$$\textbf{loop} \quad : \quad \begin{bmatrix} \leftarrow S^1_{\text{in}} \\ \dots \\ \leftarrow S^k_{\text{in}} \end{bmatrix} \rightarrow S_{\text{out}}$$

LeftMult restricts **Mult** to have only two incoming streams S^1_{in} and S^2_{in} and one output stream. It then applies a latch on the second stream S^2_{in}. So, the data rate of the output stream is dependent on S^1_{in} and independent of S^2_{in}.

LEFTMULT : $\mathbf{S}^2 \rightarrow \mathbf{S}$

$$S^1,\ S^2 \quad = \quad \text{LATCH}(S^2_{\text{in}})\ ;\ \text{GROUND}(S^2)$$

$$\textbf{loop} \quad : \quad \begin{bmatrix} \leftarrow S^1_{\text{in}} \\ \leftarrow S^1 \end{bmatrix} \rightarrow S_{\text{out}}$$

Merge operator fuses a set of input streams into one output stream. Given a set of n incoming streams of type X, a buffer of size n is created to have one element per stream. Each stream is then assigned a processing loop to read into the corresponding buffer element only if it is empty. A selection function is defined by user to pick the next output element from the buffer and free its location for next incoming elements. Several selection functions can be defined. Here, we present the function $f : X \rightarrow (Y, \preceq)$ that selects the minimum element in the buffer for output.

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

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**

Figure 1b shows an example of a fork-join graph. The first operator is the fork operator that copies the input stream into three output streams. Each output stream is connected to a processing branch that contains a set of operators. Finally all branches are joined together using the Merge operator.

4. ALGEBRA IMPLEMENTATION

Our algebra implementation provides a simple framework for defining distributed computer vision applications. It is implemented in Go language, which is chosen for its scalability and concurrency features. For example, A stream in the algebra is equivalent to a channel in Go with the read $x \leftarrow s$ and write $x \rightarrow s$ functions defined.

4.1 Framework

The framework provides the previous set of abstract operators and allows an easy way for composing them and building general workflow graphs. The operators themself are reusable components which can be plugged into different graphs. The architecture of our framework contains four layers:

- **Programming Interface.** This interface allows programmers to build operators given a set of user defined functions and parameters. A user can define a workflow graph by chaining the operators together using composition.

- **Operators Graph.** This is a DAG with nodes representing operators and edges representing data transfer channels. The graph is built in the background while a user is composing operators together.

- **Execution Interface.** The execution interface is responsible for executing the graph and performing various performance optimizations. This interface also monitors and tracks the latency and throughput of a workflow graph.

- **Data Transport.** This layer provides an automatic handling of data transfer and communication using Go language channels. Network transfers can be done using network channels[2], which allows a distributed implementation on a cluster of machines. We follow the *one-port* communication model developed by [3, 2]. This model assumes that a single send or receive communication can be performed at anytime on a given stream. However, communications can be performed in parallel on different streams. We enforce this model on the execution platform. So, there exists only a single communication at any time on a certain link, but parallel communications can be performed using different links. Notice that in the distributed mode, channels handles concurrency control and consistency by performing blocking read and write to streams.

The first step to construct a computer vision workflow using our framework is to build an operator graph by linking data processing and flow control operators. Consider for example, the pipeline in figure 1a. Given three lists of user defined functions $\mathbf{f_1}$, $\mathbf{f_2}$, and $\mathbf{f_3}$ for the mapping operators with parameters $\mathbf{p_1}$, $\mathbf{p_2}$, and $\mathbf{p_3}$, a list $\mathbf{g_1}$ with parameters $\mathbf{q_1}$ and state u_1 for the Reduce operator, and a generator function h with initial state u_0 for the Source operator. We can write this pipeline using the following Go language code,

```
g := NewGraph("pipeline")
g.Source(u0,h).Map(f1,p1).Map(f2,p2)
    .Reduce(u1,g1,q1).Map(f3,p3).Ground()
g.Execute()
```

The code defines the graph g, then we use the dot operator to compose the graph using the algebra operators. The dot operator is helpful when the number of output and input streams of two consecutive operators match. It simply forwards the output streams of the current operator to the input streams of the next operator. The pipeline starts with the Source operator which submits image tuples to subsequent processing operators until we reach Ground. We also can attach extra command information to the data tuple. A Map operator may represent a computer vision task such as foreground segmentation or optical flow. Each Map or Reduce operator receives a list of functions that defines different algorithms performing the same task. The ability to pass different functions allows us to do dynamic reconfiguration by switching between functions at runtime. Every function may also receive extra parameters which we can attach to the command section of the incoming message. So, we not only can change the function but we also can modify its behavior using new parameters. Several graphs can also be defined and linked together using fork and join operators. This is helpful when running different visual processing graphs reading from the same data source. In this case, a Copy operator may be used to copy the source stream to the processing graphs. Multiple data sources may be also used in the same graph. The `Execute()` function runs the pipeline in a distributed concurrent fashion. When network channels are used, the operators can run on different machines inside a computing cluster. This allows scaling an application from running on single multi-core computer to multiple machines.

The fork-join graphs cannot be fully composed using the dot operator as we did with pipelines. For example, the number of output streams of a fork operator may not match the number of input streams of subsequent operators that may belong to different parallel branches. To handle this problem, our framework provides two features: 1) we can assign a unique name for each operator to later reference it; 2) the framework provides the `LinkIn` and `LinkOut` functions for linking operators. Both functions receive two parameters: a target operator name and a variable-size list of names for other operators. `LinkIn` connects the output channels of a set of operators with the input channels of a target join operator. It creates an input list by sequentially finding the free input channels of the target operator. An output list is then constructed from the given operators in the variable-size list by selecting the first free output channel from each operator. The output list has the same order and length of the given variable-size list. Finally, the corresponding channels from the input and output are linked together. `LinkOut` behaves similarly but with fork operators. The following code provide an example of using `LinkIn` and `LinkOut` with the fork-join graph in figure 1b,

```
g := NewGraph("fork-join")
g.Map(f1,p1"m1").Reduce(u1,g1,q1,"r1")
g.Reduce(u2,g2,q2,"r2").Map(f2,p2,"m2")
g.Map(f3,p3,"m3").Reduce(u3,g3,q3,"r3")
g.Copy(3, "cp")
g.Merge(f, "mrg")
g.LinkOut("cp", "m1", "r2", "m3")
g.LinkIn("mrg", "r1", "m2", "r3")
```

The code again starts with creating a graph object, then we create the three parallel branches in figure 1b. Each branch is composed using the dot operator and a unique name is given to the first and last algebra operators. Next, we create the fork and join operators and assign them unique names. Finally, the `LinkOut` and `LinkIn` functions are used to connect branches to the fork and join operators.

4.2 Feedback Control and Dynamic Reconfiguration

The stream algebra of [12] has the important advantage of naturally describing feedback control as shown by [11]. Figure 1c presents an example of adding a feedback loop to control the intermediate Map and Reduce operators in the pipeline of Figure 1a. The feedback loop is asynchronous and contains Cut, Reduce, and LeftMult operators. The Cut operator forwards the incoming stream to the next operator in the pipeline, and asynchronously sample this stream to create the feedback stream. The feedback stream is then given as input to a Reduce operator that outputs a stream of commands. These commands are fedback to the asynchronous input of the LeftMult operator, which creates an output stream of vectors containing data and commands. This stream is forwarded to the controlled operators and if commands exist, they are parsed to change behavior as required. The changes may switch the Map and/or Reduce operators to use a different function or supply new parameter values. This allows us to perform dynamic reconfiguration at runtime and react to data changes. We can write this pipeline using the following Go language code,

[2]libchan: https://github.com/docker/libchan (*last accessed on 2 May 2016*).

```
g := NewGraph("feedback")
g.Source(u0, h).Map(f1, p1, "m1"))
g.LeftMult("lm").Map(f2, p2, "m2")
    .Reduce(u2, g2, q2).Cut("ct")
g.Map(f3, p3, "m3")).Ground()
g.Reduce(u4, g4, q4, "r4"))
g.LinkOut("ct", "m3", "r4")
g.LinkIn("lm", "m1", "r4")
g.Execute()
```

The code creates the same pipeline as in the previous section but with the addition of the feedback loop operators. Notice that we create four branches and connect them using the Cut and LeftMult operators.

4.3 Functions, Views and Statistics

Our framework provides OpenCV support through the Go-OpenCV bindings[3]. This allows access to a wide range of stat-of-the-art computer vision algorithms when writing the mapping and reducing functions. These functions can attach the algorithm output to the outgoing message for latter use by subsequent operators. We also created special mapping operators for viewing images and plotting data. A Viewer operator is built as a GUI window to display images. The mapping function in this case retrieves image and data from the incoming message, performs the necessary drawing operations, and returns a new image for display. The incoming messages are then forwarded to output. Dynamic plots can be also created by the Plot operator which utilizes the Gnuplot library[4] to create 2D and 3D plotting. Figure 2b shows four different views for our case study workflow that performs online road-boundary detection. Figure 3 shows dynamic plots for the average runtime of each operator in the top branch of our case study workflow in figure 2a.

Our framework accumulates running time statistics for each operator in a workflow graph. The statistics include the running average and standard deviation for the computation and communication times spent by every operator. This is performed by recording for every message, the entering and exiting times, and attaching them to the message. When this message reaches a Ground operator, the running times are retrieved for every operator and its statistics are updated. This allows distributed calculation of statistics in large workflows containing several branches and Ground operators. Notice that our framework requires no synchronization between Grounds as it assigns each operator, the Ground synchronized with that operator. This assignment is performed before executing the workflow graph. So, when an operator stores running time statistics into a message, it also records the name of the target Ground that will process these statistics. Each Ground then retrieves the statistics record associated with its unique name.

5. CASE STUDY

In this section, we express the online road-boundary detection algorithm of [13] in the proposed stream algebra. We refer the reader to [12] for the description of other state-of-the-art vision pipelines. The algorithm receives an input video stream $V = \{V_i | i = 0, 1, 2...\}$. Then, it applies edge detection on each frame $V_i \in V$ by extracting N superpixels from each V_i and applying polygon approximation. The resulted edges are incrementally added to a hierarchical clustering tree by applying an online algorithm that maintains clustering over a temporal window of interval Δt. The algorithm generates a sequence of updated clustering trees $H = \{H_i | i = 0, 1, 2, ...\}$. For each tree H_i; the algorithm statistically ranks the clusters based on the number of edges and variance. Clusters with ranks larger than a threshold T are then selected. This generates a ranked clusters stream $C = \{C_i | i = 0, 1, 2, ...\}$, where each $C_i \in C$ is a list of top ranked clusters from $H_i \in H$, at time i. For every list C_i, each cluster $C_{i,j} \in C_i$ is mapped to its mean line, and generate the line stream $L = \{L_i | i = 0, 1, 2...\}$. The method then performs a Cartesian product of each set $L_i \in L$ by itself and eliminates pairs with similar elements. This generates a pair-wise stream $P = \{P_i | i = 0, 1, 2, ...\}$. After that, the approach applies perspective filtering on every $P_i \in P$ to remove line pairs that do not have their vanishing points heading upward in the image. This generates the filtered stream of line pairs Q.

After generating the pair-wise stream Q, the algorithm ranks every pair in Q_i based on the road activity. This is performed by taking the input stream V and applying background subtraction to detect moving objects. The centroids of these objects are recorded over a temporal window of the same interval Δt used by online hierarchical clustering. Then, the method attaches with every pair-wise list $Q_i \in Q$ the recent list of detected centroids and generate a stream $U = \{U_i | i = 0, 1, 2, ...\}$. Next, activity ranking is applied on U to construct the ranked pair-wise stream J. Finally, the algorithm outputs the dominant road-boundary stream $B = \{B_i | i = 0, 1, 2, ...\}$, where $B_i = \arg\max_{x \in J_i} rank(x)$ represents the top-ranked pair from every pair-wise list $J_i \in J$.

5.1 Description Using Algebra

Now we describe this vision pipeline using the algebra (Figure 2a). The data types defined by the algorithm are,

$$\texttt{Frame} : \texttt{2DImage}; \quad \texttt{Video} : \mathbf{S}\,\langle \texttt{Frame} \rangle; \quad \texttt{Point} : \mathbb{R}^2$$
$$\texttt{Edge} : \mathbb{R}^6; \quad \texttt{Cluster} : \mathbb{R}^4 \times \texttt{Edge};$$
$$\texttt{RCluster} : \texttt{Cluster} \times \mathbb{R}$$
$$\texttt{Pair} : \texttt{Edge} \times \texttt{Edge}; \quad \texttt{RPair} : \texttt{Pair} \times \mathbb{R}$$
$$\texttt{Hierarchy} : \textsc{Tree}\,\langle \texttt{Cluster} \rangle$$
$$\texttt{Params} : \textsc{List}\,\langle \texttt{Parameter} \rangle$$

where a `Frame` is a single 2D image, a `Video` is a stream of frames, a `Point` is a 2D vector, and an `Edge` is a straight line segment $(x_1, x_2, y_1, y_2, \rho, \phi)$, where (ρ, ϕ) represents the edge in polar coordinates. A `Cluster` is represented in sufficient statistics $(\widehat{\phi}, \widehat{\rho}, n, t, s_{max})$, where,

$$\widehat{\phi} = (\textstyle\sum_{i=0}^{n} \phi_i, \sum_{i=0}^{n} \phi_i^2) \quad \widehat{\rho} = (\textstyle\sum_{i=0}^{n} \rho_i, \sum_{i=0}^{n} \rho_i^2),$$

n is the number of edges in the cluster, t is the cluster's last update time, and s_{max} is the line segment that encloses the projection of all cluster edges on its mean line. We define `RCluster` as (c, α), where c : `Cluster` and α is c's statistical rank. A `Pair` is a pair of edge segments. `RPair` is defined as (p, β), where p : `Pair` and β is p's activity rank. A `Hierarchy` is a tree of clusters. Finally, `Params` is a list of parameters.

We start by copying the incoming video stream $V \in$ `Video` into two streams V' and $V1$ using a COPY operator,

$$V', V1 \triangleq \textsc{Copy}(2)(V) \tag{1}$$

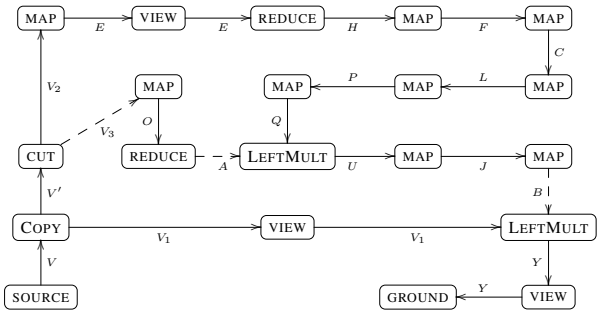
We then apply a CUT operator on V' to obtain the streams $V2$ and $V3$,

$$V2, V3 \triangleq \textsc{Cut}()(V') \tag{2}$$

Note that the three streams V', V_1, and V_2 are all copies of the original stream V with the same flow rate, however, V_3 is a sampled version of V' with a decoupled flow rate. We now process V_2, and return later to discuss the use of streams V_1 and V_3. We apply edge detection on every frame in V_2 using the functions list $\mathbf{f_1}$: $\textsc{List}\,\langle \texttt{Frame} \rightarrow \textsc{List}\,\langle \texttt{Edge} \rangle \rangle$. These functions implement

[3]Go-OpenCV: https://github.com/lazywei/go-opencv (*last accessed on 2 May 2016*).

[4]Gnuplot: http://www.gnuplot.info/ (*last accessed on 2 May 2016*).



(a)

(b)

Figure 2: The online road-boundary detection algorithm of [13] described in the stream algebra. (a) The workflow graph with arrows showing the flow direction of streams. Letters on arrows represent stream names. Dashed lines indicate decoupled streams. V is the input video stream, and Y is the output video stream that shows the estimated dominant road boundary. (b) Displaying windows generated from four View operators showing the V_1 stream, detected edges in stream E, foreground objects in stream A, and detected road boundary in stream Y.

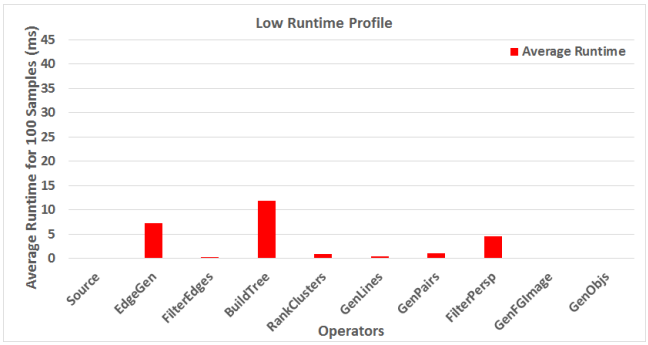
Sobel and Canny edge detectors. A list of parameters $\mathbf{p_1}$ are defined with two elements for the threshold parameters of the two detectors. $\mathbf{f_1}$ and $\mathbf{p_1}$ are used with a Map operator to define the stream $E : S\langle \text{LIST}\langle \texttt{Edge}\rangle\rangle$,

$$E \triangleq \text{MAP}(\mathbf{f_1}, \mathbf{p_1})(V_2) \circ \text{VIEW}(d_1) \tag{3}$$

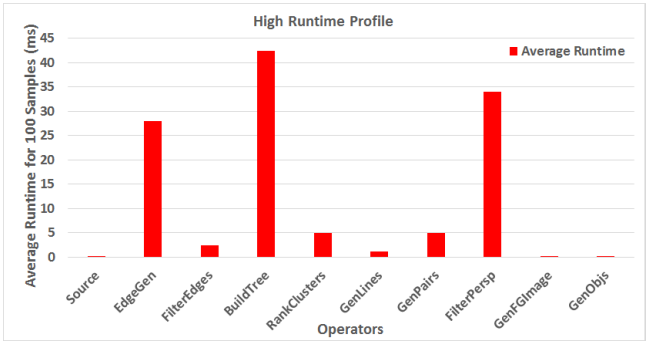
The $\circ$ operator is a composition operator that takes the output stream from the right operand and feeds it as an input stream to the left operand. Notice that we use the View operator to display the output edges using the function $d_1 : X \rightarrow \texttt{Image}$ (see Figure 2b). We then define the function,

$$g_1 : \texttt{Hierarchy} \times \text{LIST}\langle \texttt{Edge}\rangle \times \texttt{Params} \rightarrow \texttt{Hierarchy} \times \text{LIST}\langle \texttt{Cluster}\rangle$$

$$g_1(u, x, p) = \{\ u.\text{add}(x); \quad \text{//add edges } x \text{ to } u$$
$$y = u.\text{last-touched}(); \quad \text{//get last added clusters.}$$
$$\text{return}(u, y)\ \}$$



(a)



(b)

Figure 3: Dynamic plots for a set of operators in our case study workflow graph. (a) A low runtime profile with feedback control to adjust the threshold parameters of edge detection and output an average number of 200 edges. (b) A high runtime profile without feedback control.

which keeps updating a given clustering tree u by adding new edges. Then, a list of all clusters touched by the added edges are returned as the output y. We add this function to the list $\mathbf{g_1}$ and define the parameters vector $\mathbf{q_1}$ which contains for example, the tree height, number of children per node, etc. This function is used with a Reduce operator to generate the stream $H : S\langle \text{LIST}\langle \texttt{Cluster}\rangle\rangle$,

$$H \triangleq \text{REDUCE}(\text{Empty-Tree}, \mathbf{g_1}, \mathbf{q_1})(E). \tag{4}$$

Given the H stream, we apply a ranking function $f_2 : \text{LIST}\langle \texttt{Cluster}\rangle \rightarrow \text{LIST}\langle \texttt{RCluster}\rangle$ to statistically rank clusters based on variance and number of samples. This function is added to list $\mathbf{f_2}$ with parameters $\mathbf{p_2}$ for use with a Map operator to generate the ranked clusters stream $F : S\langle \text{LIST}\langle \texttt{RCluster}\rangle\rangle$,

$$F \triangleq \text{MAP}(\mathbf{f_2}, \mathbf{p_2})(H). \tag{5}$$

We then apply a threshold function $f_3 : \text{LIST}\langle \texttt{RCluster}\rangle \rightarrow \text{LIST}\langle \texttt{RCluster}\rangle$ on every element in F to choose clusters with ranks larger than a threshold T. This function is added to list $\mathbf{f_3}$ with empty parameters $\mathbf{p_3}$ to generate the stream $C : S\langle \text{LIST}\langle \texttt{RCluster}\rangle\rangle$,

$$C \triangleq \text{MAP}(\mathbf{f_3}, \mathbf{p_3})(F). \tag{6}$$

The stream C is converted to a line stream by applying a function $f_4 : \text{LIST}\langle \texttt{RCluster}\rangle \rightarrow \text{LIST}\langle \texttt{Edge}\rangle$. This function maps every cluster $C_{i,j} \in C_i$ into its mean line. The function is added to list $\mathbf{f_4}$ with empty parameters $\mathbf{p_4}$ for use with a Map operator to construct the line stream $L : S\langle \text{LIST}\langle \texttt{Edge}\rangle\rangle$,

$$L \triangleq \textsc{Map}(\mathbf{f_4}, \mathbf{p_4})(C). \quad (7)$$

Now, we apply a Cartesian product function f_5 : LIST ⟨Edge⟩ → LIST ⟨Pair⟩ on every list $L_i \in L$ by itself and remove pairs with similar elements. The function is added to list $\mathbf{f_5}$ with empty $\mathbf{p_5}$ for use with a Map operator to construct the pair-wise stream $P : S\,\langle \textsc{List}\,\langle \texttt{Pair}\rangle\rangle$,

$$P \triangleq \textsc{Map}(\mathbf{f_5}, \mathbf{p_5})(L). \quad (8)$$

After that, we define a filtering function f_6 : LIST ⟨Pair⟩ → LIST ⟨Pair⟩ that applies perspective filtering on every pair $P_{i,j} \in P_i$. This function returns a list that only contains pairs with vanishing points heading upward in the image. We add this function to list $\mathbf{f_6}$ with empty $\mathbf{p_6}$ for use with a Map operator to construct the filtered pair-wise stream $Q : S\,\langle \textsc{List}\,\langle \texttt{Pair}\rangle\rangle$,

$$Q \triangleq \textsc{Map}(\mathbf{f_6}, \mathbf{p_6})(P). \quad (9)$$

Now, we need to perform activity ranking on every pair $Q_i \in Q$. In order to define scene activity, we go back and use the V_3 stream. Remember that, the V_3 stream is a sampled version of the video stream V', which is itself a copy of the input video stream V. We apply background subtraction [15] on every frame in V_3 to get a set of foreground regions. We then output the centroids of these regions. This is performed using the function f_7 : Frame → LIST ⟨Point⟩ added to list $\mathbf{f_7}$ with empty $\mathbf{p_3}$, which together with a Map operator construct the centroids stream $O : S\,\langle \textsc{List}\,\langle \texttt{Point}\rangle\rangle$,

$$O \triangleq \textsc{Map}(\mathbf{f_7}, \mathbf{p_7})(V_3). \quad (10)$$

We record the extracted centroids over a temporal window with interval $\triangle t$. So, we define a function
g_2 : LIST ⟨Point⟩×LIST ⟨Point⟩×Params → LIST ⟨Point⟩× LIST ⟨Point⟩

$$\begin{aligned} g_2(u,x,p) = \quad \{ \quad & \text{for all } z \in u \\ & \quad \text{if } (\text{now}() - \text{arrival-time}(z) \geq p.\triangle t) \text{ then} \\ & \quad\quad u = u \ominus z \quad \text{//remove } z \text{ from } u \\ & u = u \oplus x.v \quad \text{//append points } x.v \text{ to } u \\ & \text{return}(u, u) \quad \} \end{aligned}$$

This function is added to a list $\mathbf{g_2}$ with parameters vector $\mathbf{q_2}$ that contains only the $\triangle t$ parameter. The function is used with a Reduce operator to generate the activity stream $A : S\,\langle \textsc{List}\,\langle \texttt{ Point}\rangle\rangle$,

$$A \triangleq \textsc{Reduce}(\text{Empty-List}, \mathbf{g_2}, \mathbf{q_2})(O). \quad (11)$$

Now that we have the activity stream A, it is synchronized with the filtered pairwise stream Q using a LeftMult operator. This operator latches on A, and generates a stream U : $S\,\langle \textsc{List}\,\langle \texttt{Pair}\rangle \times \textsc{List}\,\langle \texttt{Point}\rangle\rangle$,

$$U \triangleq \textsc{LeftMult}()(Q, A). \quad (12)$$

We then apply a ranking function f_8 : LIST ⟨Pair⟩ × LIST ⟨Point⟩ → LIST ⟨RPair⟩ that ranks every line pair using its attached centroids, and generates a list of ranked pairs. The function is added to list $\mathbf{f_8}$ with empty parameters $\mathbf{p_8}$ for use with a Map operator to build the the ranked pair-wise stream $J : S\,\langle \textsc{List}\,\langle \texttt{RPair}\rangle\rangle$,

$$J \triangleq \textsc{Map}(\mathbf{f_8}, \mathbf{p_8})(U). \quad (13)$$

The algorithm then applies the function $f_9 = \lambda x : \arg\max_{y \in x} r_{\text{Activity}}(y)$ on every element of stream J. This function returns the line pair with the maximum activity rank. We add this function to list $\mathbf{f_9}$ with empty $\mathbf{p_9}$ for use with a Map operator to construct the dominant road-boundary stream $B : S\,\langle \texttt{RPair}\rangle$,

$$B \triangleq \textsc{Map}(\mathbf{f_9}, \mathbf{p_9})(J). \quad (14)$$

We synchronize the B stream with the V_1 video stream coming from the View operator that displays the source stream (see Figure 2b). Remember that V_1 is a copy of the input video stream. This synchronization is performed using a LeftMult operator to construct the output stream Y : Frame × RPair,

$$Y \triangleq \textsc{LeftMult}()(V_1, B). \quad (15)$$

Afterwords, we apply the expression
$\textsc{Ground}() \circ \textsc{View}(\acute{d}_3)(Y)$, first to view the estimated dominant road-boundary on every frame using the drawing function $\acute{d}_3$: $X \rightarrow$ Image, then to release the stream resources.

5.2 Implementation

After describing our case study in the stream algebra, we easily implement it using our programming framework. It is a simple one-to-one mapping. The following Go language code implements the studied workflow graph of Figure 2a,

```
g := NewGraph("mygraph")
g.Source(h, p0).Copy(2, "cp")
g.Cut("ct")
g.Map(f1, p1, "m1s").View(d1).Reduce(u1, g1, q1)
    .Map(f2, p2).Map(f3, p3).Map(f4, p4)
    .Map(f5, p5).Map(f6, p6, "m1e")
g.Map(f7, p7, "m2s").Reduce(u2, g2, q2, "r2e")
g.LeftMult("lm3").Map(f8, p8)
    .Map(f9, p9, "m3e")
g.View(d2, "v4")
g.LeftMult("lm5").View(d3).Ground()
g.LinkOut("cp","ct","v4")
g.LinkOut("ct","m1s","m2s")
g.LinkIn("lm3","m1e","r2e")
g.LinkIn("lm5","v4","m3e")
```

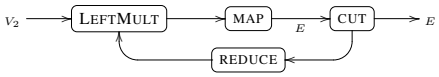
Where u_1 and u_2 define the Empty-Tree and Empty-List initial states respectively. As before, we start by creating a workflow graph, then we add the first branch beginning with the Source operator. Next, we define the Cut operator with the name "ct". Later, the edge detection and clustering branch is defined. Notice that we give unique names to branch start and end operators to reference them later in the code. We also define the foreground segmentation branch that contains a Map followed by Reduce. The code then follows by defining the activity ranking branch starting with the LeftMult "lm3" and ending with the Map "m3e". The last branch then starts by LeftMult "lm5", views the output, and grounds the stream. Later, we use the LinkOut and LinkIn functions to connect branches using the defined fork and join operators.

5.3 Discussions

The case study presented here shows that it is relatively straightforward to describe complex, scalable workflow graphs using our programming framework. One interesting feature is the ability to insert the Viewer operator at different locations to display intermediate processing results. Although here we chain the Viewer operator into the pipeline which may induce an overhead in the overall latency. We can insert a Cut-Viewer-Ground branch at any graph edge to sample the data stream without affecting performance. This is used in cases where the display function has large overhead.

One can also use feedback control in our case study to monitor

the output of one or more streaming operators and refine their future behavior. Figure 3 shows the benefits of using an asynchronous feedback control loop. The figure shows the average runtime for the source operator and all other operators in the top branch of Figure 2a with and without using the following feedback loop that controls the number of output edges from the edge detection algorithm,



This feedback loop asynchronously samples the stream of edges E and controls the edge detection Map operator using a Reduce operator. Reduce monitors the number of output edges n and stores a user desired number of edges k (set to 200 in experiments), the most recent number of edges n_r and the current value of the edge detection threshold parameter T. When Reduce receives the current output list of edges, it sets $n_r = n$ and compares n_r with k. If $n_r = k$, we leave T unchanged. If $n_r > k$, we increase the threshold parameter $T = T + \Delta T$ by a small step ΔT to reduce future number of edges. If $n_r < k$, we decrease the parameter $T = T - \Delta T$ to increase future number of edges. The adjusted parameter T are then fedback by attaching it to the next input to the Map operator using the LeftMult operator. The Map operator then updates the edge detection parameter. As all operators after edge detection depends on the number of generated edges, having k output edges on average, keeps a low runtime profile in figure 3a. If we did not use feedback control, we may encounter a runtime profile similar to the one in Figure 3b, in case of images containing large set of edges.

This simple example of using feedback control shows the large range of problems that can be described using our framework. Examples of such problems include adaptive learning, performance monitoring, parameter tuning, real-time debugging, and bottleneck identification and prevention. Multi-view streams of a given scene can also be joined using a Mult operator and forwarded to a Reduce operator for keeping track of temporal relations. In addition one can use parallel processing patterns such as Scatter-ListMap-Merge for scaling up an expensive computational task. Here, ListMap executes a list of Map operators in parallel. All these use cases suggests the importance of our programming framework in implementing scalable real world applications.

6. CONCLUSIONS

We present a programming framework for building scalable computer vision systems and distributed processing of image and video streams. The framework is the first implementation of the stream algebra proposed by [12, 11]. It is implemented in Go programming language and exploits its concurrency and scalability features. The framework is demonstrated on an online road boundary detection system, where it has proved its effectiveness in describing and implementing complex computer vision streaming workflows. We also support feedback control and dynamic reconfiguration of streaming operators, and show how these features can tune operators to maintain a desired runtime profile. In the future, we want to evaluate our programming framework on larger and more complex computer vision systems and distributed environments.

7. REFERENCES

[1] A. Afrah, G. Miller, D. Parks, M. Finke, and S. Fels. Hive: A distributed system for vision processing. In *IEEE ICDSC*, pages 1–9, September 2008.

[2] P. B. Bhat, C. Raghavendra, and V. K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63(3):251 – 263, 2003.

[3] P. B. Bhat, C. S. Raghavendra, and V. K. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 15–24, May 1999.

[4] M. Broy and G. Stefanescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258(1-2):99 – 129, 2001.

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

[6] D. Chau, J. Badie, F. Bremond, and M. Thonnat. Online tracking parameter adaptation based on evaluation. In *IEEE International Conference on AVSS*, pages 189–194, Aug 2013.

[7] G. Chkodrov, P. Ringseth, T. Tarnavski, A. Shen, R. Barga, and J. Goldstein. Implementation of stream algebra over class instances, Google patents, jan 2013.

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

[9] GStreamer. http://gstreamer.freedesktop.org. Accessed: 2016-05-2.

[10] N. Harbi and Y. Gotoh. Spatio-temporal human body segmentation from video stream. In *Computer Analysis of Images and Patterns*, volume 8047, pages 78–85. Springer, 2013.

[11] M. A. Helala, K. Q. Pu, and F. Z. Qureshi. *2nd Workshop on User-Centered Computer Vision (UCCV) in conjunction with ACCV 2014*, chapter Towards Efficient Feedback Control in Streaming Computer Vision Pipelines, pages 314–329. November 2014.

[12] M. A. Helala, K. Q. Pu, and F. Z. Qureshi. A stream algebra for computer vision pipelines. In *IEEE CVPR Workshops*, June 2014.

[13] M. A. Helala, F. Z. Qureshi, and K. Q. Pu. Automatic parsing of lane and road boundaries in challenging traffic scenes. *Journal of Electronic Imaging*, 24(5):053020, 2015.

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

[15] K. Kim, T. Chalidabhongse, D. Harwood, and L. Davis. Background modeling and subtraction by codebook construction. In *ICIP*, volume 5, pages 3061–3064, Oct 2004.

[16] C. Loy, T. Hospedales, T. Xiang, and S. Gong. Stream-based joint exploration-exploitation active learning. In *CVPR*, pages 1560–1567, 2012.

[17] C. Lu, J. Shi, and J. Jia. Online robust dictionary learning. In *IEEE CVPR*, pages 415–422, 2013.

[18] M. S. Ryoo. Human activity prediction: Early recognition of ongoing activities from streaming videos. In *ICCV*, pages 1036–1043, Barcelona, Spain, 2011.

[19] Twitter's Storm. http://storm.incubator.apache.org. Accessed: 2016-05-2.

[20] C. Xuand, C. Xiong, and J. Corso. Streaming hierarchical video segmentation. In *ECCV*, volume VI, pages 626–639, 2012.

[21] J. Yang, J. Luo, J. Yu, and T. Huang. Photo stream alignment and summarization for collaborative photo collection and sharing. *IEEE Trans. on Multimedia*, 14(6):1642–1651, 2012.