Exploiting Multi-level Parallelism for Low-latency Activity Recognition in Streaming Video

Ming-yu Chen,† Lily Mummert,* Padmanabhan Pillai,* Alex Hauptmann,† Rahul Sukthankar*†
†Carnegie Mellon University, *Intel Labs Pittsburgh

ABSTRACT

Video understanding is a computationally challenging task that is critical not only for traditionally throughput-oriented applications such as search but also latency-sensitive *interactive* applications such as surveillance, gaming, videoconferencing, and vision-based user interfaces. Enabling these types of video processing applications will require not only new algorithms and techniques, but new runtime systems that optimize latency as well as throughput. In this paper, we present a runtime system called Sprout that achieves low latency by exploiting the parallelism inherent in video understanding applications. We demonstrate the utility of our system on an activity recognition application that employs a robust new descriptor called MoSIFT, which explicitly augments appearance features with motion information. MoSIFT outperforms previous recognition techniques, but like other state-of-the-art techniques, it is computationally expensive — a sequential implementation runs 100 times slower than real time. We describe the implementation of the activity recognition application on Sprout, and show that it can accurately recognize actions at full frame rate (25 fps) and low latency on a challenging airport surveillance video corpus.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; D.2 [**Software**]: Software Engineering

General Terms

Algorithms Design Performance

Keywords

Parallel Computing, Cluster Applications, Multimedia, Sensing, Stream Processing, Computational Perception

Author contact addresses: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; Intel Labs Pittsburgh, 4720 Forbes Avenue, Suite 410, Pittsburgh PA 15213. Author e-mail addresses: mychen@cs.cmu.edu, lily.b.mummert@intel.com, padmanabhan.s.pillai@intel.com, alex@cs.cmu.edu, rahuls@cs.cmu.edu

ACM Multimedia Systems 2010 Scottsdale Arizona

Figure 1: Activity recognition on Gatwick airport video. Our system recognizes actions in full frame rate video with low latencies to enable interactive surveillance applications.

1. INTRODUCTION

Video is becoming ubiquitous in daily life for applications ranging across surveillance, entertainment, communications, and natural user interfaces. The rate at which video is being generated has accelerated demand for machine understanding of rich media to enable better content-based search capabilities of both stored and streaming data. Systems for processing video have been traditionally evaluated according to two metrics: the accuracy with which they can recognize events of interest, and the rate at which data can be processed. However, as interactive video applications become more prominent, a third key metric, latency, is becoming increasingly important. Latency directly impacts the effectiveness of many real-world applications because these tasks require that the results of video understanding be made immediately available to the user (see Figure 1). Examples of applications that are sensitive to latency include monitoring and surveillance scenarios where the operator must be quickly alerted in the event of an emergency, and vision-based user interfaces or immersive environments where even moderate latencies can unacceptably degrade the user's experience. There has been extensive research on frame-rate processing

Figure 2: Interest points detected with SIFT (left) and MoSIFT (right). Green circles denote interest points at different scales while magenta arrows illustrate optical flow. Note that MoSIFT identifies distinctive regions that exhibit significant motion, which corresponds well to human activity while SIFT fires strongly on the cluttered background.

of video, but simply achieving the desired throughput in a system does not necessarily lead to any improvement in terms of latency.

A major barrier to the widespread deployment of video understanding algorithms has been their computational expense. For instance, current methods for recognizing activities in surveillance video typically involve spatio-temporal analysis such as computing optical flow and 3D SIFT descriptors at multiple scales for every frame in a high-resolution stream. Fortunately, the increasing availability of large-scale computer clusters is driving efforts to parallelize video applications so that they can be mapped across a distributed infrastructure. The majority of these efforts, such as MapReduce [13] and Dryad [18], focus on efficient batch analysis of large data sets; while such systems accelerate the offline indexing of video content, they do not support continuous processing. A smaller set of systems provide support for the continuous processing of streaming data [1, 4, 12, 39] but most of these focus on queries using relational operators and data types, or are intended for mining applications in which throughput is optimized over latency. This paper presents a novel approach to the problem. We exploit both coarse- and fine-grained parallelism in the task to achieve low latencies while processing high-resolution video at full frame rate.

In this paper, we present a cluster-based distributed runtime system called Sprout that achieves low latency by exploiting the parallelism inherent in video understanding applications. We demonstrate the utility of our system on a activity recognition application that employs a novel and robust descriptor called MoSIFT, which exploits continuous object motion explicitly calculated from optical flow and integrates it with distinctive appearance features. Although computationally expensive like other state-of-the-art techniques, the proposed approach outperforms existing algorithms on standard action recognition data sets. These results validate our belief that the added computational complexity of sophisticated descriptors is warranted. Although straightforward implementations of our method can process relatively small collections of videos, such as the popular KTH dataset [36], they cannot scale to the large real-world corpora that are the primary focus of our research. The 2008 TRECVID event detection evaluation [38] uses recently released surveillance camera footage from the London Gatwick airport consisting of 100 hours of full-frame video acquired from five cameras. In terms of throughput, the straightforward implementation runs 100 times slower than real time on a single-threaded system, and would need more than a year to process the Gatwick data on a single machine. By contrast, the same approach implemented using Sprout on a cluster of 15 8-core machines can process the video corpus at full frame rate with low latency. It is noteworthy that the Sprout framework enabled the parallelized implementation to be built in just a few days; manually implementing a similar parallel system without Sprout could take experienced developers weeks or months of effort. Although our system is presented and evaluated in the context of a surveillance scenario, it is applicable to many computationally-intensive multimedia processing algorithms that require high throughput and are sensitive to latency.

This paper makes three main contributions. First, we argue that achieving high recognition accuracy in video understanding requires computationally expensive methods. This is supported by experiments using two state-of-the-art features, the recent Laptev et al. [24] descriptor and our own robust descriptor, MoSIFT. Second, we outline a general framework for enabling low-latency processing of full frame rate video that exploits both the coarse- and fine-grained parallelism inherent in typical multimedia understanding algorithms. Specifically, we describe a novel runtime, Sprout, that distributes video processing over a cluster of multi-core machines and present experiments that characterize its throughput and latency benefits. Finally, we present an implementation of a low-latency surveillance system that can perform activity recognition on large quantities of streaming video. We perform a series of detailed experiments to characterize the benefits of coarse- and fine-grained parallelism in terms of both latency and throughput.

The paper is organized as follows. We describe the core of the activity recognition approach in Section 2 (MoSIFT). Section 3 describes the Sprout architecture. Section 4 presents details of how the algorithm was parallelized. Section 5 describes the experiments and results, followed by Section 6 discussing related research efforts, with Section 7 providing a summary of the effort and an outline of future work.

2. ACTIVITY RECOGNITION IN VIDEO

Activity recognition forms the core of most video understanding systems, whether for surveillance, video gaming interfaces, or retrieval applications. In this section, we briefly review current approaches to the problem and describe a feature representation, MoSIFT, that we employ in our system for extracting semantic content. This descriptor matches (or exceeds) state-of-the-art descriptors in terms of recognition accuracy both on established action recognition datasets and on the challenging Gatwick airport surveillance collection. However, like other state-of-the-art methods, MoSIFT requires significant computation and is slow when implemented in a sequential manner. These experiments validate our decision to use MoSIFT in our case study of parallelization for latency and throughput in the remainder of the paper.

2.1 Extracting semantic features from video

Current approaches to action recognition in video are typically structured as follows: (1) identify a set of semantically-interesting regions in the video; (2) characterize the spatio-temporal neighborhood at each interest point as a feature vector, which is often quantized using a codebook; (3) aggregate the set of features extracted in a video snippet to generate a histogram of their occurrence frequencies; (4) treat this histogram as a high-dimensional vector and classify it using a machine learning technique trained on human-annotated video sequences.

Interest point detection reduces video data from a large volume of pixels to a sparse but descriptive set of features. Ideally, an interest point detector should densely sample those portions of the video where events occur while avoiding regions of low activity. Therefore, our goal is to develop a method that generates a sufficient but manageable number of interest points that can capture the information necessary to recognize arbitrary observed actions. Popular spatio-temporal interest point detectors [14,23] are spatio-temporal generalizations of established 2D operators developed for image processing, such as the Harris corner detector. Although mathematically elegant, these approaches treat motion in an implicit manner and exhibit limited sensitivity for smooth gestures, which lack sharp space-time extrema [19]. By contrast, the philosophy behind our MoSIFT interest point detector is to treat appearance and motion separately, and to explicitly identify spatially-distinctive regions in a frame that exhibit sufficient motion at a variety of spatial scales, as shown in Figure 2.

The information in the neighborhood of each interest point is expressed using a descriptor that explicitly encodes both an appearance and a motion component. We are not the first to propose representations that do this; several researchers [24, 35] have reported the benefits of augmenting spatio-temporal representations with histograms of optical flow (HoF). However, unlike those approaches, where the appearance and motion information is separately aggregated, MoSIFT constructs a single feature descriptor that concatenates appearance and motion. The former aspect is captured using the popular SIFT descriptor [26] and the latter using a SIFT-like encoding of the local optical flow. In contrast to video cuboids or spatio-temporal volumes, the optical flow representation explicitly captures the magnitude and direction of a motion, rather than implicitly modeling motion through appearance change over time. MoSIFT is a superset of the Laptev et al. detector [24] since MoSIFT not only detects velocity changes but also smooth movements. Additional implementation details of MoSIFT are given in Section 4 and our technical report [11].

We adopt the popular bag-of-features representation for action recognition using MoSIFT interest points, summarized as follows. Interest points are extracted from a set of training video clips. K-Means clustering is then applied over the set of descriptors to construct a codebook. Each video clip is represented by a histogram of occurrence of each codeword (bag of features). This histogram is treated as an input vector for a support vector machine (SVM) [7], with a χ^2 kernel. The χ^2 kernel is defined as:

$$K(x_i, x_j) = exp(-\frac{1}{A}D(x_i, x_j)), \quad (1)$$

where A is a scaling parameter that is determined empirically though cross-validation. $D(x_i, x_j)$ is the χ^2 distance defined as:

$$D(x_i, x_j) = \frac{1}{2}\sum_{k=1}^{m}\frac{(u_k - w_k)^2}{u_k + w_k}, \quad (2)$$

with $x_i = (u_1, ..., u_m)$ and $x_j = (w_1, ..., w_m)$. Prior work has shown that this kernel is well suited for bag-of-words representations [45]. Since the SVM is a binary classifier, to detect multiple actions we adopt the standard one-vs-rest strategy to train separate SVMs for multi-class learning.

2.2 Recognition accuracy: KTH dataset

The KTH human motion dataset [36] has become a standard benchmark for evaluating human action detection and recognition algorithms. The dataset contains six types of human actions (walking, jogging, running, boxing, hand waving and hand clapping), performed by 25 individuals. Each person performs the same action four times under four different scenarios (outdoors, outdoors at a different scale, outdoors with moving camera, and indoors). The dataset consists of 598 low-resolution (160×120) video clips, with each clip containing a single action. Although KTH is much smaller than the datasets that form the focus of our research, it serves as a consistent point of comparison against current techniques.

We follow Niebles et al. [28] in performing leave-one-out cross-validation to evaluate our approach. Leave-one-out cross-validation uses 24 subjects to train action models and then tests on the remaining subject. Performance is reported as the average accuracy over 25 runs. For MoSIFT, we extracted approximately 1.6 million interest points from the whole KTH dataset with the MoSIFT detector, and constructed a 600-word codebook. We trained SVMs using the χ^2 kernel with $A = 0.5$.

Table 1 summarizes our results on the KTH dataset. We observe that MoSIFT demonstrates a significant improvement over current methods, many of which also employ bag-of-features with different descriptors. The lack of motion information in some approaches results in lower performance than the Laptev et al. and MoSIFT techniques, both of which utilize explicit appearance and motion descriptions. For our final comparison we include Ke et al. [19], which uses a boosted cascade that operates solely on optical flow without modeling appearance. Clearly, representing motion alone is not sufficient for activity recognition.

2.3 Recognition accuracy: Gatwick dataset

The 2008 TRECVID surveillance event detection dataset [38] was collected at London Gatwick International Airport. It consists of 50 hours (5 days × 2 hours/day × 5 cameras) of video in the development set and 49 hours in the evaluation set. Each individual video is just over 2 hours long, and contains about 190K frames, recorded at 720 × 576 resolution at 25 frames per second. This dataset contains highly crowded scenes, severely cluttered background, large variance in viewpoints, and very different instances of the same action. Together, these characteristics make activity recognition on this dataset a formidable challenge, both in terms of content analysis and system architecture. To the best

Method	Accuracy
MoSIFT	**95.0%**
Laptev et al. [24]	91.8%
Wong et al. [43]	86.7%
Niebles et al. [28]	83.3%
Dollar et al. [14]	81.5%
Schuldt et al. [36]	71.7%
Ke et al. [19]	62.7%

Table 1: MoSIFT significantly outperforms current methods on the standard KTH dataset.

Action	Random	Laptev et al.	MoSIFT
CellToEar	6.98%	19.42%	22.42%
Embrace	8.03%	29.35%	29.20%
ObjectPut	18.03%	44.24%	46.31%
PeopleMeet	22.32%	44.69%	40.68%
PeopleSplitUp	13.63%	56.91%	57.42%
Pointing	26.11%	41.54%	43.61%
PersonRuns	4.95%	32.56%	36.00%
Average	14.29%	38.39%	**39.38%**

Table 2: MoSIFT significantly improves recognition performance on the 100-hour Gatwick surveillance dataset. The performance is measured by average precision.

of our knowledge, activity recognition on such a large, challenging task with these practical concerns has not been evaluated and reported prior to TRECVID 2008. In that task, 10 events are evaluated: ObjectPut, PeopleMeet, PeopleSplitUp, Pointing, CellToEar, Embrace, PersonRuns, ElevatorNoEntry, TakePicture, and OpposingFlow. Standardized annotations of actions in the development set were provided by NIST.

We evaluate recognition performance in a forced-choice setting (i.e., "which of the 10 events is this?") using the annotations provided by NIST. There were a total of 6,439 events in the development set. The size of the video codebook was increased to 1000 after cross validation on the development set. Since the data were captured by several cameras over 5 different days, we evaluated each camera independently using 5-fold cross-validation and averaged their results. There were not enough annotated examples for OpposingFlow, ElevatorNoEntry and TakePicture to run cross validation; therefore, we do not report performance of these three tasks. We use average precision (AP) as the metric, which is typical for TRECVID high-level feature recognition.

Table 2 shows the performance of the Laptev et al. and MoSIFT algorithms on the Gatwick data set. The experimental results confirm that MoSIFT has more stable recognition performance against the state-of-the art Laptev et al. method.

2.4 Computational requirements

The experimental results on KTH and Gatwick datasets confirm that MoSIFT significantly improves activity recognition. However, the gain in accuracy from our more complicated descriptor comes at the cost of adding significant additional computation. MoSIFT is computationally expensive because it not only scans though different spatial scales but also calculates corresponding optical flows. Thus, even though our proposed method is promising in terms of recognition accuracy, a straightforward implementation of an application based on MoSIFT would exhibit unacceptable performance in terms of throughput and latency. For instance, it would take 416 days just to compute all of the MoSIFT descriptors on the Gatwick dataset using a single machine. Even with a naively parallelized implementation in which frames are processed in a pipelined fashion over a large number of machines, the system would incur a delay of more than 2.5 seconds between the occurrence of an event and an alert. The computational requirements of the Laptev et al. method are similar because like MoSIFT, it also computes optical flow at multiple scales. The expense of these methods motivates the development of a low-latency processing infrastructure for activity recognition applications.

3. A SYSTEM FOR LOW-LATENCY MULTIMEDIA PROCESSING

Sprout is a distributed stream processing system designed to enable the creation of interactive multimedia applications. Interaction requires low end-to-end latency, typically well under 1 second [8,9,27]. Sprout achieves low latency by exploiting the coarse-grained parallelism inherent in such applications, executing parallel tasks on clusters of commodity multi-core servers. Its programming model facilitates the expression of application parallelism while hiding much of the complexity of parallel and distributed programming. In this section, we present an overview of Sprout, the motivation for which is described elsewhere [32].

3.1 Application model

Sprout applications are structured as data flow graphs. The vertices of the graph are coarse-grained processing steps called *stages*, and the edges are *connectors* which represent data dependencies between stages. The data flow model is particularly well suited for multimedia processing tasks because it mirrors the high-level structure of these applications, which typically apply a series of processing steps to a stream of video or audio data.

Concurrency in the data flow model is explicit — stages may execute in parallel, constrained only by their data dependencies and the availability of processors. Task, data, and pipeline parallelism may all be used, but not all of these forms of parallelism decrease latency. Figure 3 illustrates this idea for an image processing task that performs independent processing on frames of a video stream. The sequential application is slow in terms of frame latency and throughput. Inter-frame parallelization pipelines frame processing over multiple instances of the application, improving throughput but not latency. Intra-frame parallelization divides the processing of each frame over multiple processors (e.g., by splitting the frame into tiles, a form of data parallelism), improving both throughput and latency. In practice, these techniques are complementary and may be used in concert.

As we are primarily concerned with data sources that generate data at some given rate, such as video cameras, the data flow in our system is driven by the sources and follows a push model, where data is generated and sent to downstream processing stages as quickly as possible. If a downstream stage is busy, the data is placed in a queue. In contrast, a pull model is driven by the data sinks, and pulls data sources only as quickly as they can be processed by any bottleneck in the system. The pull model is less likely to encounter queueing delays, while the push model makes it easier to pipeline execution across stages and interface to constant rate sources. Our system incorporates mechanisms to mitigate queueing delays and minimize latency for the push model.

Stages within an application employ a shared-nothing model: they share no state, and interact only through connectors. This restriction keeps the programming complexity of individual stages comparable to that of sequential programming, and allows concurrency to be managed by the underlying runtime system.

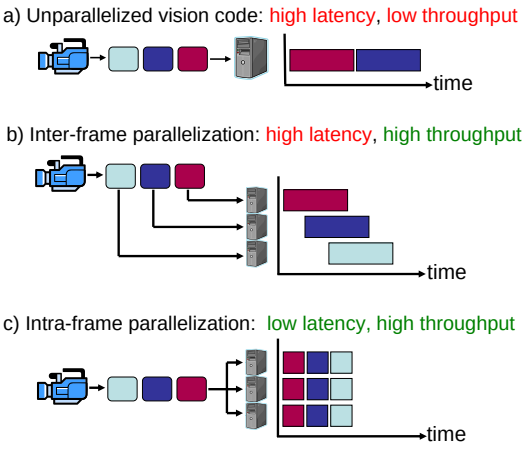


Figure 3: Approaches to parallel execution and effects on latency and throughput.

3.2 Sprout runtime system

Figure 4 illustrates the Sprout architecture. An application runs on a set of processing nodes, each of which hosts one or more *stage server* processes. A stage server runs one or more of the application's stages. Stages are activated and deactivated within stage servers dynamically, providing a mechanism for adjusting stage placement at run time. Stage servers need not be identical; that is, different stage servers may be specialized to host a subset of application stages. Multiple stage servers may run on a processing node to provide distinct functionality or process isolation for stages that require it.

For each application, a management process called the *configuration server* is responsible for the initial and ongoing configuration of the application, including stage and connection startup and shutdown, stage placement, and adjustment of application-specific parameters.

3.2.1 Stages

Stages are coarse-grained, application-specific processing steps. The stages running within a given stage server are executed as separate threads to reduce context switch time. The *stage API* provides the means by which a stage interacts with the run-time environment. The main element of this API is an `exec()` method, which is the only function a stage implementation must define. A stage-specific *firing rule* determines the circumstances under which the runtime system executes the stage. In keeping with the data push model, the default rule executes a stage when all of its inputs are ready, but the stage API allows the firing rule to be customized as needed. Additional API calls provide for stage initialization and shutdown.

A stage may have an arbitrary number of inputs and outputs. Inputs and outputs are strongly typed, and defined in a way that allows access by name. When a stage is executed, input data is accessed using a `get()` method on an input. Output data is sent using a `put()` method on an output. These operations transfer ownership of data objects. An input object is owned by the stage until it is `put()` or deallocated. Ownership of an output object is transferred away from the stage on `put()`, and the object must not be accessed or modified after that point.

In addition, the Sprout APIs permit a stage to export runtime parameters, or *tunables*, to control its operation. Tunables can be either discrete or continuous, and may be adjusted by the user or the system dynamically. Example code for the definition of a stage and its inputs, outputs, tunables, and main execution method is shown in Figure 5.

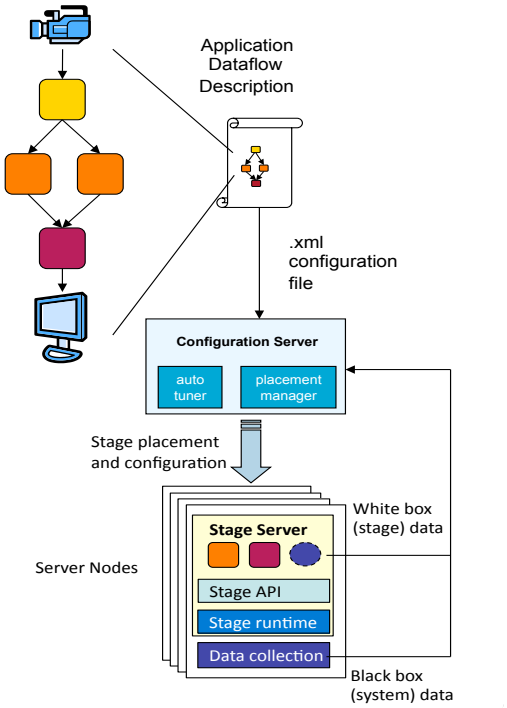


Figure 4: Sprout architecture

Data is delivered to a Sprout application by specialized stages called data *sources* and consumed by data *sinks*. A source is a stage with no inputs and a specialized firing rule that indicates when data is available (e.g., periodic execution for a constant frame rate video source). A sink is simply a stage with no outputs in the data flow graph, although it may generate outputs to external components, such as a file or a display. Implementations for common sources and sinks such as cameras, files, and displays are provided through a system component library. Sources and sinks can also be used as adapters to other applications.

Fork and join structures in the application data flow graph are implemented using *splitter* and *joiner* stages. A splitter divides or copies data. The number of outputs can be fixed (for task parallelism) or variable (for data or pipeline parallelism). Variable outputs provide a means for runtime adjustment of parallelism by the system. Similarly, a joiner merges data, and can have fixed or variable inputs. Generic splitters and joiners of various kinds (e.g., round robin, copy, vector) are provided through a system component library.

While stages are typically implemented as sequential blocks of code, they may also employ parallelism directly. A typical use is multithreaded or vectorized code intended to exploit fine-grained, intra-machine parallelism on multi-core hardware. Examples of fine-grained parallelism include use of APIs such as OpenMP [31] and specialized libraries such as Intel Integrated Performance Primitives [17].

3.2.2 Connectors

Connectors define the data dependencies between stages. Connector endpoints map to stage inputs and outputs. Variable numbers of inputs and outputs may be mapped to connectors using policies (e.g., any, all, round robin). For example, a round robin input pol-

```
class ImageScaler :  public Stage {
public:
   Input<IplImage> In;        // Input is an image
   Output<IplImage> Out;      // Output is an image
   ContinuousTunable scale;   // runtime-tunable parameter
   // this macro declares accessors to get elements by name at runtime
   DECLARE_NAMES( 3, In, Out, scale );
   int exec();                // main function
   // no special init(), fini() methods needed; use default firing rule
};

int ImageScaler::exec() {
   IplImage *img = In.get();     // get input image
   float s = scale.getValue();   // get scaling parameter
   int h = img->height*s;
   int w = img->width*s;
   // create image buffer of the right size for the output
   IplImage *out = cvCreateImage(cvSize(w, h),
         img->depth, img->nChannels);
   cvResize(img, out);        // copy and resize function
   Out.put(out);              // send output
   cvReleaseImage(&img);      // delete input image
   return (0);
}
```

Figure 5: A simple stage written in C++ with Sprout APIs. This stage uses OpenCV data structures and library calls to rescale an input image. The scaling factor is tunable at runtime.

```
<Application name="facedetect">
   <Module name="Scale" args="">
      <Inputs>In</Inputs>
      <Outputs>Out</Outputs>
      # define a continuous tuning knob with range 1–10
      <Tunable name="scale" type="continuous"
               best="1">1,10</Tunable>
      <Stage class="ImageScaler">
   </Module>
   # other module specifications here

   <Connector source="FrameSource:Out"
              dest="Scale:In">
   <Connector source="Scale:Out"
              dest="Detect:In">
   # other connector specifications here

   <Servers exec="facedetect"> # name of executable
      # hosts available to run this server
      <Hosts>nodeA,nodeB,nodeC</Hosts>
   </Servers>
</Application>
```

Figure 6: Configuration for a face detection application that uses the ImageScaler stage.

icy allows multiple connectors to provide data to a single input in round robin order. The default policy is a one-to-one mapping between input or output and connector.

The underlying implementation of a connector depends on the location of the stage endpoints. If the connected stages are running in the same process, the connector is implemented as an in-memory queue. Otherwise, a TCP connection is used. Sprout determines the connector type, and handles serialization and data transport through connectors transparently. Hooks are provided for custom marshaling code that may be needed for user-defined classes (e.g., for deep copying, or special memory allocation). All of the remote connections for a particular server are managed by an additional thread that employs nonblocking network operations.

As Sprout uses a push model to move data through application stages, unbounded growth of input queues to slow stages is a concern. To address this problem, connectors employ a queue backpressure mechanism to reduce data inflow. When an output queue of a stage exceeds a small fixed length, its firing rule will prevent additional executions of the stage until the downstream stage has sufficiently emptied the queue. For remote connections, the sum of the lengths of the local output queue and the remote input queue is used. This requires signalling of queue lengths from the downstream server, and is transparently handled by the connector implementation. This mechanism limits the number of data items queued and the total queueing delays in the application.

3.2.3 Configuration

Sprout uses a human-readable configuration file to describes an application's data flow graph. As shown in Fig. 6, the application configuration has three types of specifications: modules, connectors, and servers. A module is specified as either a single stage or as a subgraph, which recursively consists of other modules and connectors. A module specification includes inputs, outputs, and tunable parameters, as well as an indication of the number of parallel instances (if any) that should be launched. The latter allows concise representation of data-parallel stages and subgraphs. Connectors link outputs of modules to inputs of other modules. Server specifications list stage server executables, along with a set of available processing nodes that can host the servers. Optionally, the server specification can include a stage layout, which indicates a complete or partial placement of individual stages on processing nodes. Additional convenience features allow users to split the configuration among multiple files and to define macros. This, along with the ability to define subgraphs as modules, facilitates reuse in the configuration system. As our system is intended to automate replication of stages and degrees of parallelism, the configuration is actually a template of the structure of the application, with guidance for extracting parallelism.

Multiple distinct server executables may be used to form a single application. Each server binary may be specialized to provide a subset of the stages used by an application (e.g., a server with a camera source stage). Additionally, multiple binary support facilitates the use of stages incorporating proprietary code, which can be distributed as binary-only stage servers. Finally, our system permits multiple instances of the same server binary to run on the same node. This feature is useful if process isolation is needed for the stages. A practical example is a stage that uses an external library that is not thread safe; multiple instances of the stage can execute on a single processing node in separate servers.

An application is launched by running a configuration server with a configuration file as input. The configuration server generates a complete initial placement of stages to stage servers (extending any manual layout specified in the configuration), invokes stage servers on the processing nodes if they are not already running, and then activates the appropriate stages in each stage server. The configuration server then directs the stage servers to create input and output connections for each stage, and connect each stage to its downstream neighbors. Once the connections are completed, the stages execute according to their firing rules.

3.2.4 Monitoring

Monitoring of stage and processing node metrics allows runtime adaptations, such as adjusting the level of parallelism, migrating stages, or tuning application-specific parameters. Application-specific or *white-box* observations of stage performance are

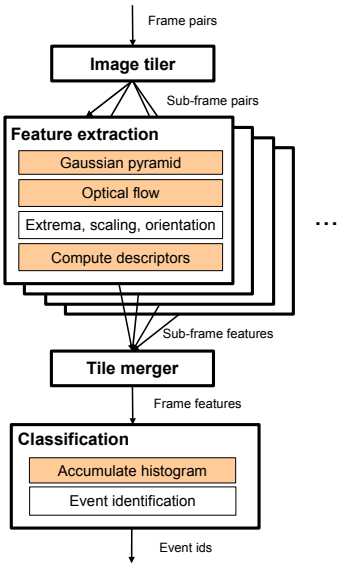


Figure 7: Sprout application graph for the MoSIFT-based activity recognition. Fine-grained parallelism is used within stages for processing steps shown in shaded boxes.

obtained from the stages themselves. Each stage maintains a fixed length circular log of per-execution records. Measurement overhead is controlled via a dynamically set sampling rate. Each log record contains a timestamp, elapsed time, CPU time, and amount of input and output data for each connection. Stage data consists of these log records and a snapshot of connection queue lengths. In addition, application-independent or *black-box* observations may be obtained at the processing node level, such as the utilizations of CPU, network, and disk.

4. PARALLEL MOSIFT

We implemented a parallel activity recognition application using MoSIFT features on Sprout. Figure 7 shows the decomposition of the application into Sprout stages. The implementation uses both coarse-grained parallelism at the stage level, and fine-grained parallelism within stages using OpenMP. This section describes our implementation and the methods used to parallelize its execution.

4.1 Frame pairs and tiling

Since MoSIFT computes optical flow, processing is based on frame pairs. A video data source decomposes the video into a series of frame pairs, which are input to the main processing stages. Since the MoSIFT interest points are local to regions of an image pair, we exploit intra-frame parallelization using an image tiler stage. The tiler divides each frame into a configurable number of uniformly sized overlapping sub-regions. The tiles are sent to a set of feature extraction stages to be processed in parallel. Overlap of the tiles ensures that interest points near the tile boundaries are correctly identified. The tiler also generates meta-data that includes positions and sizes of the tiles, for merging the results of feature extraction.

This tiling approach is an example of coarse-grained parallelization, since it did not need any changes to the inner workings of the feature extraction stage. The Sprout runtime and APIs make it easy to reconfigure applications to make use of such parallelization. As another example of coarse-grained parallelization, we also run parallel instances of the entire graph of stages in Figure 7, using a round-robin data splitter to distribute frame pairs to the parallel instances. This latter technique improves throughput only, while the tiling approach improves both throughput and latency.

4.2 Feature extraction

Like other SIFT-style keypoint detectors, MoSIFT finds interest points at multiple spatial scales. Two major computations are employed: SIFT interest point detection on the first frame to identify candidate features; and optical flow computation between the two frames, at a scale appropriate to the candidate feature, to eliminate those candidates that are not in motion.

Candidate interest points are determined using SIFT [26] on the first frame of the pair. SIFT interest points are scale invariant and all scales of a frame image must be considered. A Gaussian is employed as a scale-space kernel to produce a scale space of the first frame. The whole scale space is divided into a sequence of octaves and each octave is further subdivided into a sequence of intervals, where each interval is a scaled frame. The number of octaves and intervals is determined by the frame size. The first interval in the first octave is the original frame. Images from different octaves and intervals form a Gaussian pyramid which covers multiple scales.

Since MoSIFT uses optical flow across all levels of the scale space representation, it requires a Gaussian pyramid for each image in the frame pair. These are computed in parallel in two separate threads. The optical flow is then computed between corresponding frames in the Gaussian pyramid. We parallelize this set of computations using OpenMP to assign loop invocations to a set of threads. As image size and computation time varies over the octaves, we do not parallelize by octave. Rather, we parallelize by interval, assigning computation for a particular interval index across all octaves to a single thread. This ensures a balanced load among the threads for the optical flow computations.

Difference of Gaussian (DoG) images, which approximate the output of a bandpass Laplacian of Gaussian operator, are needed to find SIFT interest points. A DoG pyramid is computed by subtracting adjacent intervals of the Gaussian pyramid for the first frame in the pair. As with the optical flow, we parallelize this computation by intervals to equally partition work among the threads.

As in SIFT, once the pyramid of DoG images has been generated, the local extrema (minima/maxima) of the DoG images across adjacent scales are used as the candidate interest points. The algorithm scans through each octave and interval in the DoG pyramid and extracts all of the possible interest points at each scale. Unlike SIFT, candidate points are then checked against the optical flow pyramid. Candidate points are selected as MoSIFT interest points only if they contain sufficient motion in the optical flow pyramid at the appropriate scale.

The final step in the feature extraction stage is descriptor computation. Since interest points are independent, descriptors are computed in parallel over the interest points, limited only by the available cores on the processing node. The MoSIFT descriptor explicitly encodes both appearance and motion. The appearance component is the 128-dimensional SIFT descriptor for the given patch, briefly summarized as follows. The magnitude and direction for the intensity gradient is calculated for every pixel in a region around the interest point in the Gaussian-blurred image. An orientation histogram with 8 bins is formed, with each bin covering 45 degrees. Each sample in the neighboring window is added to a histogram bin and weighted by its gradient magnitude and its distance from the interest point. Pixels in the neighboring region are normalized into 256 (16×16) elements. Elements are grouped as 16 (4×4) grids around the interest point. Each grid contains its own orien-

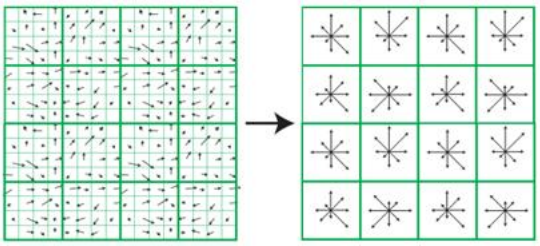


Figure 8: MoSIFT aggregates appearance and motion information using a SIFT-like scheme. Figure adapted from [26].

tation histogram to describe sub-region orientation. This leads to a SIFT feature vector with 128 dimensions (4×4×8 = 128). Each vector is normalized to enhance its invariance to changes in illumination. Figure 8 illustrates the SIFT descriptor grid aggregation. The same idea of grid aggregation is applied to motion. The optical flow describing local motion at each pixel is a 2D vector with the same structure as the gradient describing local appearance. This enables us to encode motion with the same scheme as that used by SIFT for appearance. A key benefit of this aggregation approach is that our descriptor becomes tolerant to small deformations and partial occlusion (just as standard SIFT was designed to be tolerant to these effects). The two aggregated 128-dimensional histograms (appearance and optical flow) are concatenated to form the MoSIFT descriptor, which is a vector of 256 dimensions.

4.3 Tile merger and classification

After feature descriptors are constructed, each feature extraction stage sends the descriptors to a tile merger stage, which collects the feature descriptors and adjusts their positions in the whole frame. In the classification stage, features are mapped to codewords in a previously-generated camera-specific codebook. A histogram is generated for the current frame pair, and accumulated into histograms representing different time windows. The histogram is constructed in parallel over the features, up to the number of available cores. Finally, an SVM is used on normalized histograms to identify specific activities.

5. EVALUATION

We evaluate the runtime performance of MoSIFT-based activity recognition on Sprout in two ways. First, we examine the effect of scene content on execution time. Second, we examine the extent to which coarse- and fine-grained parallelism can be used to improve latency and throughput. Our experiments are performed on a cluster of 15 compute servers connected via a 1 Gbps Ethernet switch. Each cluster node has eight 2.83 GHz Intel Xeon processor cores and 8 GB RAM, and runs Ubuntu Linux 7.04. All of our experiments use data from the TRECVID London Gatwick airport video corpus described in Section 2.3. Because MoSIFT feature extraction consumes the vast majority of the application's run time, our experiments measure execution time for all application stages in Figure 7 and all network latencies, except the classification stage, which adds approximately 20 ms to the costliest frames.

5.1 Effect of scene content

The computational cost of the activity recognition application described in Section 4 consists broadly of fixed and variable terms. The fixed cost involves per-pixel processing of a frame, such as constructing Gaussian pyramids and computing optical flow. The variable cost depends on the number of features identified in the frame, such as in the histogram computation step.

	Min	Mean	Max	Total
Camera 1	1	184	880	34,551,904
Camera 2	0	96	1,646	17,950,868
Camera 3	0	161	931	30,186,555
Camera 4	0	6	254	1,108,186
Camera 5	1	187	1,108	34,971,610

Table 3: Number of interest points in Gatwick video.

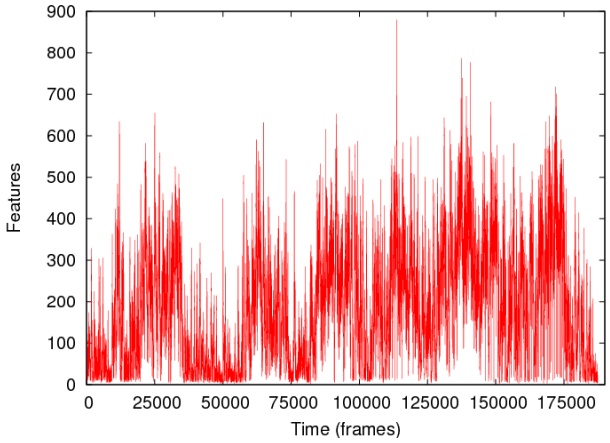


Figure 9: Variation in number of interest points detected over time for Camera 1.

Table 3 shows the number of MoSIFT features extracted from the first day's segment for each of the five cameras. The maximum number of features extracted from camera 2 (1,646) is due to a horizontal sync artifact in two frames of the video that generate apparent motion. Without these two frames, the maximum number of features extracted from the camera 2 video is 393. The number of features extracted from each frame pair can vary significantly over time, as shown in Figure 9 for camera 1, due to groups of people moving through the scene. Figure 10 shows the effect of scene content on feature extraction latency of full-frame (untiled) processing on a single core for all five two-hour video segments.

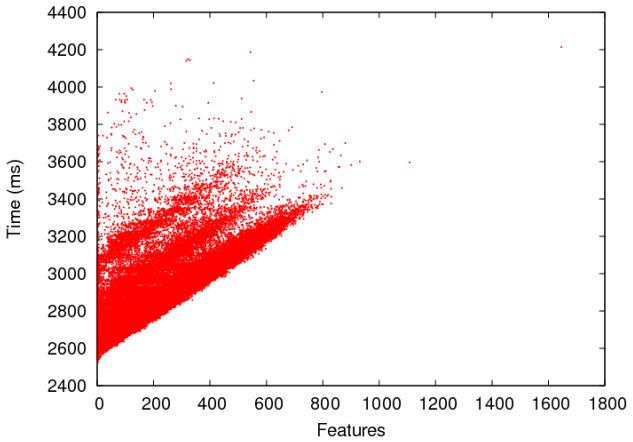


Figure 10: Latency as a function of number of features for sequential implementation.

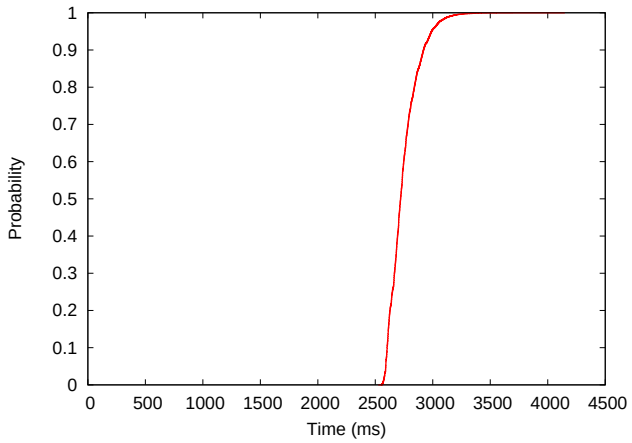


Figure 11: Cumulative distribution of frame latency for sequential implementation.

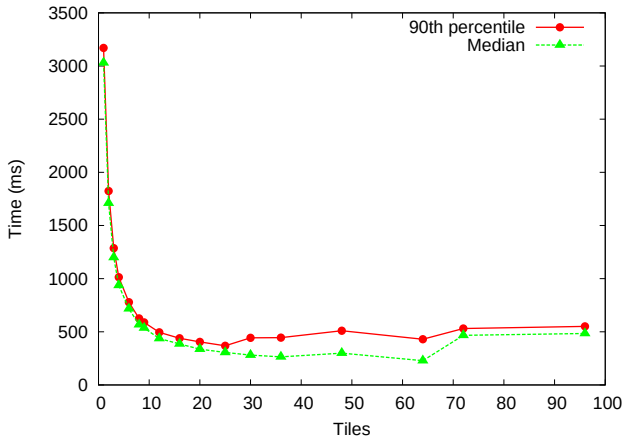


Figure 12: Frame latency vs. number of tiles using coarse-grained parallelism.

The figure shows a clear linear trend with the number of features extracted. We achieve processing at full frame rate by replicating feature extraction stages over a sufficiently large number of processors. However, this approach does not address latency, which remains unacceptably high. As Figure 11 shows, all latency measurements exceed 2.5 seconds.

5.2 Coarse-grained parallelism

As shown in Figure 7, features can be extracted from tiles within a frame in parallel. Tiling is a simple way to introduce coarse-grained parallelism without aggressively refactoring a pre-existing algorithm. To explore worst case performance, we selected a two-minute segment from the set of videos used in Section 5.1 that contained the largest total number of features (camera 1, starting at frame 171475, with 1.25M features). Figure 12 shows that tiling improves feature extraction performance by a factor of eight. While median latency falls to nearly 200 ms for the 64-tile configuration, the 90th percentile increases once the number of tiles exceeds 25. This increase appears to be due to network congestion caused by the feature extraction stages sending output to the tile merger.

The features extracted from tiles can differ from those obtained from a full image. For example, features near tile edges or large features spanning multiple tiles can be missed. We address this issue in two ways. First, tiles can be created with a specified overlap to alleviate edge effects. Second, a tiled pyramid of scaled images can be created to capture larger features. For a camera trained on a distant scene, such as in surveillance, scaled tiles may not be necessary. In practice we find that while the features extracted from tiles can differ from those extracted from a full image, the differences do not affect activity recognition in a significant way. For example, we compared the identification results for 36-way tiling with 10 pixel overlap and no scaling to full image processing for the footage collected on day 1 from camera 2, and found that recognition accuracy decreased by only 0.5%.

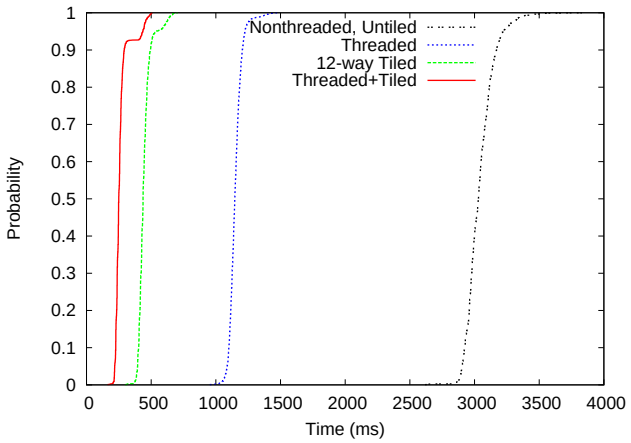
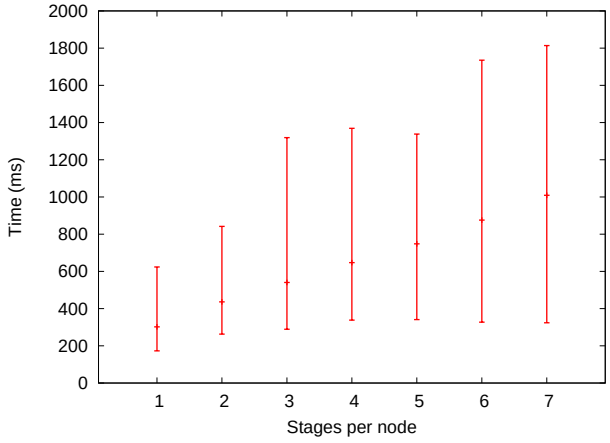


Figure 13: Cumulative distribution of latency using coarse- and fine-grained parallelism. Threading decreases latency by a factor of 2.6, tiling by a factor of 7, and the combination by a factor of 12.

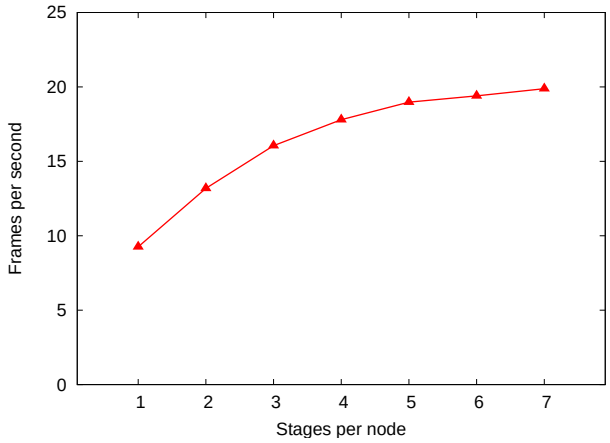
5.3 Fine-grained parallelism

In addition to the coarse-grained parallelism at the level of frame tiles, Sprout allows complementary fine-grained approaches to parallelism. As described in Section 4, using OpenMP, we have created a MoSIFT implementation that utilizes multiple threads in various processing steps. We note that threading requires greater human programming effort than identification and implementation of coarse-grained parallel stages in Sprout, due to concurrency and synchronization issues in the former. In our threaded implementation, some of the processing steps make use of all 8 cores in our machines, while other steps are sequential, or can utilize only 2 or 6 cores at a time. Thus on average we see only 2.5–3x speedup when compared to a single core, nonthreaded version. Figure 13 shows the distribution of execution times for frame pairs from the most feature-intensive 2-minute segment of the data set for a threaded untiled case, 12-way tiled but non-threaded case, and a combination of tiling and threading. The combination of tiling and threading reduces latency dramatically by a factor of 12.

A common drawback of threading is that the number of cores used varies over time as execution enters and leaves parallel sections. As a result, placing multiple instances of such code on a single machine becomes problematic as these may contend for processing resources. Figure 14 shows the effect of placing multiple instances of 12-way tiled, threaded stages on each machine. Although throughput increases as the number of processing instances increases, the latency increases and becomes more variable due to nondeterministic resource contention, defeating the latency advantages of the threaded implementation.



(a) Latency



(b) Throughput

Figure 14: Effect of placing multiple threaded stages on each eight-core host.

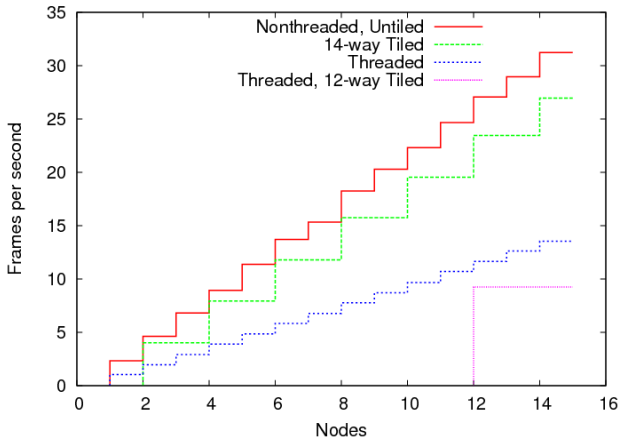


Figure 15: Scaling up throughput with number of nodes using pipelined coarse- and fine-grained parallel configurations.

5.4 Achieving high throughput

Although our focus has been primarily to reduce the high processing latency of MoSIFT, we also need to enable full-frame rate processing. We achieve high throughput by replicating the tiling and feature extraction stages over multiple machines and processors. Our goal is to scale throughput with effective use of available hardware resources. For the untiled, nonthreaded configuration, we place 7 replicas per machine (reserving one core for I/O). The second configuration uses a 14-way tiled setting, allowing one replica for every two machines. For the threaded configurations, we limit placement to a single threaded stage per machine based on the findings in the previous sections. These do not utilize the cores very effectively, and achieve relatively poor throughput compared to the nonthreaded configurations. Figure 15 shows that only the nonthreaded, untiled and 14-way tiled configurations achieve full frame rate on our 15-node cluster. Both of these configurations scale well with additional machines, but as shown in Figure 13, the tiled configuration has significantly better latency with the same set of resources.

5.5 Lessons learned

In summary, while we find that Sprout successfully allows multiple mechanisms to be used for parallelization (as demonstrated in Figure 15), great care needs to be taken to select an appropriate number of processing nodes, determine the optimal allocation of stages to nodes (Figure 14), adjust the degree of data parallelism (e.g., the tiling granularity; Figure 12), and alleviate any network contention issues (Figures 12 and 13) that may arise. This forms the basis of our future work to more robustly handle dynamic adjustment of these system parameters and mitigate overheads of cluster parallelization.

6. RELATED WORK

We briefly survey related work in computer vision on automatically understanding video, and in distributed systems on efficiently processing large quantities of streaming data.

6.1 Activity recognition in video

There has been much research in activity recognition and this has been applied in a number of domains, including visual surveillance, human computer interaction, and video retrieval. Aggarwal et al. [2] give an overview of the various tasks involved in human motion analysis. Hu et al. [16] review work on visual surveillance in dynamic scenes and analyze possible research directions. Generally, activity recognition consists of two main steps: feature extraction and pattern classification. Feature extraction can be further divided into two categories, one based on holistic features (e.g., [5, 6, 19]), and the other based on local descriptors [14, 20, 23, 24, 28, 35, 36]. The pattern classification approaches can be grouped into two categories: (1) those based on stochastic models such as HMM [44] and pLSA [28], and (2) those based on statistical models such as ANN [40], NNC [14], SVM [36], LPBoost [29] or AdaBoost [15, 19].

In terms of holistic features, Bobick et al. [6] create temporal templates, including motion-energy images and motion-history images to recognize human movement. Ke et al. [19] employ a volumetric representation incorporating the horizontal and vertical components of optical flow. Blank et al. [5] regard human actions as three dimensional shapes induced by silhouettes in the space time volume. Rodriguez et al. [34] use a frequency domain technique, called the Maximum Average Correlation Height (MACH) filter, to recognize single-cycle human actions. Holistic features

can achieve impressive results but can be sensitive to pose, occlusion, deformation and cluttered backgrounds. This has motivated research on part-based models that employ local features.

Methods based on feature descriptors around local interest points are currently popular in object recognition. These part-based approaches assume that a collection of distinctive parts can effectively describe the whole object. Recently, these methods have been extended to the spatio-temporal domain and applied to activity recognition in video. As discussed in Section 2, such methods typically employ an interest point detector in conjunction with a local feature descriptor. Popular interest point detectors for video are often spatio-temporal extensions of well-known 2D interest point operators, such as the Harris corner (e.g., [23]). Alternately, the detector can focus exclusively on patterns in the temporal domain (e.g., [14]), or extrema in local space-time (e.g., [30]). An alternative to using a feature detector is to uniformly sample the spatio-temporal volume (at multiple scales), and to compute local descriptors over this dense grid. Our approach for MoSIFT has been to employ a standard 2D interest point detector on a single frame and to retain only those interest points that exhibit sufficient motion.

Research on descriptors for local spatio-temporal regions has considered both appearance and motion, either in implicit or explicit forms. At one extreme, Shechtman et al. [37] extend the notion of 2D image correlation to 3D space-time volumes. Ke et al. [20] oversegment the spatio-temporal volume into supervoxels and assemble these to match parts of target actions. Klaser et al. [21] construct a local descriptor based on histograms of oriented 3D spatio-temporal gradients, while Willems et al. [41] build a representation that is scale-invariant in both space and time. All of these methods are amenable to parallelization using Sprout. Our case study, MoSIFT, is closest to Laptev et al. [24], where appearance and motion information is aggregated into bags-of-features and then recognized using a learned discriminative classifier. However, as discussed in Section 2, our representation unites appearance and motion in a single descriptor, which improves recognition accuracy.

6.2 Stream processing systems

FlowVR [3] and Stampede [33] both provide support for distributed execution of interactive multimedia applications on compute clusters. An application is structured as a data flow of processing modules and explicit data dependencies. Modules execute asynchronously on separate threads, and the underlying system transports data between modules transparently. FlowVR focuses on integration of disparate modules that execute at different rates or may themselves encompass parallel code, and a hierarchical component model that facilitates composition of large applications [25]. Unlike in Sprout, latency and parallelization are controlled by hand tuning of module code, execution rates, and placement on compute nodes. Stampede emphasizes space-time memory (STM), a distributed data structure for holding time-indexed data, as a key abstraction around which applications are constructed. While modules are placed on compute nodes to minimize latency, the placement algorithm assumes that the number of modules and data-parallel variations is small enough to precompute optimal configurations [22]. Sprout assumes a shared-nothing model based on explicit data channels between modules and makes no assumptions about the number of modules or configurations.

Systems such as Aurora [12], Borealis [1], and TelegraphCQ [10] provide support for continuous queries over data streams. These systems are used for applications such as financial data analysis, traffic monitoring, and intrusion detection. Data sources supply tuples (at potentially high data rates) which are routed through an acyclic network of windowed relational operators. Operators and data are distributed over compute nodes to achieve a quality of service goal, typically a function of performance (e.g., latency), accuracy, and reliability. Quality of service is managed by dynamically migrating operators, partitioning data, shedding load, and reordering operators or data. Although these systems process streaming data, perform runtime adaptation, and consider real-time constraints, they are limited to relational operators and data types.

System S [4] provides support for user-defined operators, stream discovery, dynamic application composition, and operator sharing between applications. It has been used to process multimedia streams, and assumes a resource-constrained data center environment in which utilization is high and jobs may be rejected. Compute resources are allocated to applications to maximize an importance function, typically a weighted throughput of output streams [42], unlike Sprout which is primarily concerned with low latency.

MapReduce [13] and Dryad [18] are systems that allow large data sets to be processed in parallel on a compute cluster. MapReduce applications consist of user-specified *map* and *reduce* phases, in which key-value pairs are processed into intermediate key-value pairs, and then values with the same intermediate key are merged. Dryad admits a more general application structure; a job consists of an acyclic data flow graph of sequential processing modules. Both systems operate from stored data rather than streams, and are employed in off-line rather than interactive applications. Like Sprout, MapReduce and Dryad provide simple programming abstractions and handle many of the messy details of distributed computation.

7. CONCLUSION

Efficient and automatic processing of streaming video content at low latencies is critical for a large class of applications in surveillance, gaming, intelligent environments and vision-based user interfaces. This paper makes three significant contributions to the field. First, we propose a novel representation for video content that significantly improves the accuracy of activity recognition in video. However, our proposed method, MoSIFT, is computationally expensive, so naive implementations on a single processor are impractical for the large-scale real-world video collections that form the primary focus of our work. Thus, the second contribution of this paper is a novel general framework, Sprout, for leveraging clusters of multi-core processors to significantly improve latency and throughput. Finally, we present an implementation of a surveillance system built using Sprout and demonstrate, using a series of detailed experiments, that taking advantage of coarse- and fine-grained parallelism inherent in multimedia algorithms enables us to achieve significant benefits in terms of both latency and throughput. While these experiments emphasize the surveillance aspects of our work, our system can easily enable other algorithms to process streaming video for a wide variety of multimedia applications. For instance, we have employed the same architecture in conjunction with a completely different vision algorithm to create a gestural interface for an interactive multi-player game.

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proc. Innovative Data Systems Research*, 2005.

[2] J. K. Aggarwal and Q. Cai. Human motion analysis: a review. In *Proc. Workshop Nonrigid and Articulated Motion Workshop*, 1997.

[3] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. FlowVR: a middleware for large scale virtual reality applications. In *Proc. Euro-Par*, 2004.

[4] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *Proc. Workshop on Data Mining Standards, Services, and Platforms*, 2006.

[5] M. Blank, L. Gorelick, E. Shechtman, M. Irani, and R. Basri. Actions as space-time shapes. In *ICCV*, 2005.

[6] A. F. Bobick and J. W. Davis. The recognition of human movement using temporal templates. *IEEE Trans. PAMI*, 23(3), 2001.

[7] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proc. Computational Learning Theory*, 1992.

[8] J. Brady. A theory of productivity in the creative process. *IEEE Computer Graphics and Applications*, 6(5), May 1986.

[9] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *Proc. SIGCHI*, 1991.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. Innovative Data Systems Research*, 2003.

[11] M.-Y. Chen and A. Hauptmann. MoSIFT: Recognizing huamn actions in surveillance videos. Technical Report CMU-CS-09-161, Carnegie Mellon University, 2009.

[12] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. Innovative Data Systems Research*, 2003.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1), 2008.

[14] P. Dollar, V. Rabaud, G. Cottrell, and S. Belongie. Behavior recognition via sparse spatio-temporal features. In *IEEE Workshop on PETS*, 2005.

[15] A. Fathi and G. Mori. Action recognition by learning mid-level motion features. In *CVPR*, 2008.

[16] W. Hu, T. Tan, L. Wang, and S. Maybank. A survey on visual surveillance of object motion and behaviors. *IEEE Trans. Systems, Man and Cybernetics*, 34(3), 2004.

[17] Intel Corporation. *Intel Integrated Performance Primitives for Intel Architecture*, August 2008.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems*, 2007.

[19] Y. Ke, R. Sukthankar, and M. Hebert. Efficient visual event detection using volumetric features. In *ICCV*, 2005.

[20] Y. Ke, R. Sukthankar, and M. Hebert. Event detection in crowded videos. In *ICCV*, 2007.

[21] A. Kläser, M. Marszałek, and C. Schmid. A spatio-temporal descriptor based on 3D-gradients. In *BMVC*, 2008.

[22] K. Knobe, J. M. Rehg, A. Chauhan, R. S. Nikhil, and U. Ramachandran. Scheduling constrained dynamic applications on clusters. In *Proc. Supercomputing*, 1999.

[23] I. Laptev and T. Lindeberg. Space-time interest points. In *ICCV*, 2003.

[24] I. Laptev, M. Marszalek, C. Schmid, and B. Rozenfeld. Learning realistic human actions from movies. In *CVPR*, 2008.

[25] J.-D. Lesage and B. Raffin. A hierarchical component model for large parallel interactive applications. *The Journal of Supercomputing*, July 2008.

[26] D. Lowe. Distinctive image features form scale-invariant keypoints. *IJCV*, 60(2), 2004.

[27] R. B. Miller. Response time in man-computer conversational transactions. In *Proc. AFIPS*, 1968.

[28] J. C. Niebles, H. Wang, and L. Fei-Fei. Unsupervised learning of human action categories using spatial-temporal words. In *BMVC*, 2006.

[29] S. Nowozin, G. Bakir, and K. Tsuda. Discriminative subsequence mining for action classification. In *ICCV*, 2007.

[30] A. Oikonomopoulos, I. Patras, and M. Pantic. Spatiotemporal salient points for visual recognition of human actions. *IEEE Trans. Systems, Man, and Cybernetics*, 36(3), 2005.

[31] OpenMP Architecture Review Board. *OpenMP Application Program Interface – Version 3.0*, May 2008.

[32] P. Pillai, L. Mummert, S. Schlosser, R. Sukthankar, and C. Helfrich. SLIPStream: scalable low-latency interactive perception on streaming data. In *Proc. NOSSDAV*, 2009.

[33] U. Ramachandran, R. Nikhil, J. M. Rehg, Y. Angelov, A. Paul, S. Adhikari, K. Mackenzie, N. Harel, and K. Knobe. Stampede: a cluster programming middleware for interactive stream-oriented applications. *IEEE Trans. Parallel and Distributed Systems*, 14(11), 2003.

[34] M. Rodriguez, J. Ahmed, and M. Shah. ActionMACH: a spatio-temporal maximum average correlation height filter for action recognition. In *CVPR*, 2008.

[35] K. Schindler and L. Van Gool. Action snippets: How many frames does human action recognition require? In *CVPR*, 2008.

[36] C. Schuldt, I. Laptev, and B. Caputo. Recognizing human actions: A local SVM approach. In *ICPR*, 2004.

[37] E. Shechtman and M. Irani. Space-time behavior-based correlation or how to tell if two underlying motion fields are similar without computing them? *IEEE Trans. PAMI*, 29(11), 2007.

[38] TRECVID 2008. `http://www-nlpir.nist.gov/projects/tv2008/tv2008.html`.

[39] D. S. Turaga, B. Foo, O. Verscheure, and R. Yan. Configuring topologies of distributed semantic concept classifiers for continuous multimedia stream processing. In *ACM Multimedia*, 2008.

[40] L. Wang, X. Geng, C. Leckie, and R. Kotagiri. Moving shape dynamics: A signal processing perspective. In *CVPR*, 2008.

[41] G. Willems, T. Tuytelaars, and L. Van Gool. An efficient dense and scale-invariant spatio-temporal interest point detector. In *ECCV*, 2008.

[42] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Proc. ACM/IFIP/USENIX Middleware*, 2008.

[43] S.-F. Wong and R. Cipolla. Extracting spatiotemporal interest points using global information. In *ICCV*, 2007.

[44] J. Yamato, J. Ohya, and K. Ishii. Recognizing human action in time-sequential images using hidden markov model. In *CVPR*, 1992.

[45] J. Zhang, M. Marszalek, S. Lazebnik, and C. Schmid. Local features and kernels for classification of texture and object categories: A comprehensive study. *IJCV*, 73(2), 2007.