Towards Efficient and Scalable Computer Vision Systems

by

Mohamed A. Helala

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Faculty of Graduate Studies (Computer Science)
University of Ontario Institute of Technology

Supervisor(s): Ken Q. Pu and Faisal Z. Qureshi

Contents

Abstract

Towards Efficient and Scalable Computer Vision Systems

Mohamed A. Helala

Doctor of Philosophy

Faculty of Graduate Studies

University of Ontario Institute of Technology

2018

There is a large growth in hardware and software systems capable of producing vast amounts of image and video data. These systems are rich sources of continuous and possibly infinite image and video streams. This motivates researchers to build scalable computer vision systems that utilize data-streaming concepts for large-scale processing of visual data streams. However, several challenges still exist in building large-scale computer vision systems. The main challenge is the lack of formal and scalable mechanisms and frameworks for building and optimizing large-scale visual processing. Moreover, several fundamental computer vision tasks are computationally expensive and inefficient for scaling up for large-scale processing. This thesis presents formal methods and algorithms that aim to overcome these challenges and improve building and optimizing large-scale computer vision systems.

We first describe a formal algebra framework for the mathematical description of computer vision pipelines for processing image and video streams. The algebra defines a set of abstract and concurrent operators with well-defined semantics for building scalable computer vision systems. It naturally describes feedback control and provides a formal and abstract method for data-stream manipulation, adaptive parameter selection, dynamic reconfiguration, incremental optimization, and defining common optimization and cost models.

Second, we present new algorithms for efficiently processing image and video streams in two areas of computer vision: pixel-labelling problems and automatic visual surveillance. For pixel-labelling problems, we develop the sub-volume cost-filtering approach for solving both stereo-vision and optical-flow problems. The approach leverages sparse processing of the cost volume to achieve faster runtimes with comparable accuracy to the state-of-the-art algorithms. For automatic visual surveillance, we develop a new online algorithm for automatic lane and road-boundary detection. The algorithm runs in real time and is adaptive and able to handle several challenging environmental conditions.

Finally, we express the road-boundary detection algorithm using our stream algebra. We use it as a case study for developing common optimization methods for parameter tuning in large-scale streaming pipelines.

Chapter 1

Introduction

Nowadays, our ability to generate images and videos has dramatically increased with the ubiquitous access to camera-equipped devices, such as smartphones and tablets. This created an increasing trend in recording personal memories and visually sharing stories, especially in social network platforms, such as YouTube [1] and Instagram. [2] We can anticipate that this trend will continue to increase in the future.

This huge growth of visual content motivates researchers to build systems that support web-scale computer vision by building scalable algorithms. The target is to process and analyze a vast amount of visual content as long-lasting streams. Such streams include image and video sequences (possibly infinite) from traffic-camera networks, vehicular vision systems, and vision-based Internet-of-things systems [188, 128, 178, 119]. We refer to this family of data streams as vision streams.

Moreover, many classical vision problems, such as tracking, object detection, object counting, edge detection, background modelling, etc., can be cast within the stream-processing framework. Online vision algorithms can be implemented as stream-processing functions. There are also several recent algorithms [145, 123, 175, 119, 72, 176, 105] that applied data-streaming concepts to efficiently solve different computer vision prob-

[1]YouTube: https://www.youtube.com/ (*last accessed on 7 September 2017*).
[2]Instagram: https://www.instagram.com/ (*last accessed on 7 September 2017*).

lems. Examples of online algorithms that deal with streams of images and video include streaming hierarchical video segmentation [175], photo stream alignment [176], human body segmentation [72], storyline inference from web photo streams [105], human activity prediction from video streams [145], and analysis of traffic-video streams [81].

Managing and processing data streams has been an active topic in the field of database and information systems [3, 29, 168, 13, 8]. However, most data-stream research has been focused on streams of textual, numerical, and semi-structured data. Consequently, the resulting stream-processing systems have been built and optimized for text-stream processing by relying on and extending database operators (such as relational algebraic operators and related analytical methods). To efficiently process textual data streams, the database community developed stream-algebra frameworks [52, 38, 43, 55]. Stream algebra extends relational algebras for data-stream processing and provides 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.

Several platforms have been developed by building upon and extending stream algebras. Examples of such platforms include Apache Storm [12], Amazon Kinesis [107], Apache Kafka [10], and Spark Streaming [11]. The success of these platforms comes from their ability to intelligently process vast amounts of streams and adapt to changes in computational resources. These platforms process data streams in an online fashion. Apache Storm [12] is an example of a distributed computation system for processing large-scale data streams. It is fault-tolerant and scalable with a guarantee of data processing. Storm [12] defines a pipeline as a directed graph topology with Spout operators and Bolt operators. A Spout reads data tuples from a given source and submits them to Bolts. A Bolt is a generic data-processing operator that receives multiple input streams and submits multiple output streams. Storm provides the Trident API that implements a set of streaming extensions of the relational algebra operators on top of the Bolt and Spout concepts of Storm. The Amazon Kinesis [107] platform can acquire and process data streams in real time from thousands of different web sources at a data rate of several terabytes per hour. It also can integrate with streaming frameworks, such as Storm, for stream processing. Apache Kafka [10] is a distributed messaging system that can read and write streams. It can be used to write scalable stream-processing pipelines for real-time data processing. Kafka also stores streams and replicates them for fault tolerance. It also provides a domain-specific language that extends relational algebra operators for data-stream processing. Spark Streaming [11] allows scalable and fault-tolerant real-time stream processing. It receives data streams and splits them into batches that are processed in parallel using a set of stream-processing operators, extending the relational algebra.

These large-scale processing platforms provide different implementations of the database stream-algebra frameworks and are optimized for processing textual streams. However, researchers in computer vision have tried to utilize these platforms for processing vision streams. For example, Yu et al. [180] developed a video parsing and evaluation platform using Spark Streaming and Kafka for processing long surveillance videos. In this platform, users write computer vision processing modules using Spark Streaming, and Kafka is used for communicating messages between different modules. Tabernik et al. [158] developed a web service for object detection using hierarchical models. Their method is implemented as a pipeline in Storm using custom-built Spouts and Bolts. Zhang et al. [183] developed a system architecture for online surveillance-video processing based on Kafka and Spark streaming. In this system, users can write video tasks that process blocks of video data.

Despite these efforts to customize text-stream-processing platforms for computer vision, there are still several problems in building large-scale computer vision systems. These problems stem from the nature of image and video streams and the complexity of computer vision algorithms. Vision streams are more complex than text streams. They usually have high dimensional features, complex/multi-modal data, and a wide range of noisy samples. This induces a very large number of intermediate results generated by computer vision tasks in large-scale systems, thus creating large latencies in moving data in a distributed system. Moreover, computer vision tasks deal with different types of videos and images, such as binary images, images with different colour models, and videos with different frame rates, environments, illuminations, etc. To deal with these different conditions, a computer vision task can be implemented using different algorithms with different accuracy and speed profiles depending on the content, type, and speed of incoming data. Thus, a large-scale computer vision system should enable dynamic reconfiguration of the system to switch between different runtime profiles to match changes in incoming streams. In addition, computer vision algorithms usually require a larger number of input parameters than text-stream-processing algorithms. It is not clear how to adaptively tune this large number of parameters in large-scale systems to dynamically adapt to changes in content, type, and speed of incoming vision streams. These problems suggest that a formal framework is needed to solve the challenges of large-scale processing of vision streams.

Building large-scale visual processing systems also requires scalable and efficient computer vision algorithms. Although several vision algorithms can be executed in an online fashion, many computationally expensive algorithms cannot. Examples of these algorithms include scale-invariant feature transform (SIFT) keypoint extraction [118, 116] and dense labelling problems, such as optical flow and stereo vision [143, 170].

1.1 Problem Statement

This thesis addresses two main problems: (1) building formal, scalable, and efficient easy-to-use mechanisms and frameworks for building and optimizing large-scale visual processing and (2) accelerating existing computationally expensive computer vision algorithms while achieving accuracy comparable to the state of the art. The first problem stems from the fact that there is no formal data-stream processing framework for computer vision. We are interested in the stream-processing approach, which is an emerging direction for big visual data analysis. The second problem focuses on developing efficient algorithms for computationally expensive computer vision algorithms. In this work, we address pixel-labelling problems and traffic-video analysis to accelerate their performance for large-scale stream processing.

1.1.1 Stream Algebra

Building formal stream-algebra frameworks has been addressed in the database community [52, 38, 43, 55] for efficiently processing textual data streams. 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. Stream algebras exist in the database literature to provide an effective description of queries on event or relational streams. They provide a framework for query analysis and optimization. These algebras bring several advantages, such as providing a formal and abstract method for data-stream manipulation, resolving blocking operations, scheduling asynchronous and bulk-wise processing tasks, implementing dynamic execution plans, applying incremental evaluation, scaling up data processing, and defining common pipeline optimization and cost models.

Moreover, one of the most important building blocks for pipeline optimization is feedback control, which allows the implementation of tasks, such as parameter tuning and incremental optimization. Stream algebras have the benefit of formally defining feedback-control loops. Broy et al. [38] defined streaming pipelines as data-flow networks and proposed a feedback operator to describe feedback loops. Although this operator was theoretically defined, the same concept has been used as a guideline for applying feedback control in database systems [102, 114, 164, 165, 100, 84]. Kapitanova et al. [102] developed a formal specification language that models queries on textual data streams. They also used feedback-control primitives to provide quality of service (QoS) management mechanisms in data-streaming systems. Yi et al. [164] applied feedback control to satisfy quality requirements for processing continuous queries in a data-stream management system by controlling the application behaviour in situations such as an overloaded system. These examples show that having a stream algebra with a formal set of streaming operators may be useful for efficiently developing common optimization methods.

The key performance metrics for streaming workflows are throughput and latency [26, 25, 24]. The throughput measures the rate at which data tuples enter or exit a system. Equivalently, the period is the inverse of throughput, and it measures the interval between the system entry times of two consecutive tuples. The latency is the interval between the system entry and exit times for a given tuple, so it measures the overall response time of the system in processing the tuple. Data tuples may have different latencies hence the maximum response time defines the system latency. Although a stream algebra can mathematically define a given streaming workflow, the aim is to construct a mathematical expression that provides an efficient execution plan. So, it is naturally desirable to define mathematical expressions that minimize latency and maximize throughput; however, the two criteria are opposite to each other, and one should find a good trade-off.

1.1.2 Building Efficient Online Algorithms

There is currently progress in developing online computer vision pipelines [145, 123, 175, 119, 72, 176, 105] that manipulate vision streams. These applications span several areas, such as analysis of web photo collections, activity recognition, surveillance cameras, and satellite imagery. For example, Schuster et al. [148] proposed a method for real-time detection of unusual regions in surveillance-video streams. The method partitions each image and extracts a local model for each partition. Then, the model is continuously updated toward scene changes by applying several heuristic rules. The method was applied to guide camera operators to areas of attention.

Gunhee et al. [70] proposed another method for multiple foreground segmentation of similar objects within an image stream. This method oversegments each image into a set of segments, which are grouped using an iterative scheme into a k-region foreground model. The algorithm was tested on Flickr photo streams and the ImageNet dataset and showed promising results.

Cao et al. [42] proposed a method for recognizing human activities from video streams in which part of the activities are missing. They cast the problem as a probabilistic framework and used sparse coding to calculate the likelihood of a test video belonging to a certain activity.

Ryoo [145] also presented a recent algorithm that can predict the type of activity early that is happening between two humans in an input video stream. They addressed six human activities: hand shaking, hugging, kicking, pointing, punching, and pushing. The algorithm operates in two main stages, an offline learning stage, and an online prediction stage.

Kim et al. [105] proposed an algorithm to build common storylines from Flickr photo streams and to discover the relations between them. Their method jointly aligns and segments large-scale web photo streams by applying message-passing-based optimization.

Xuand et al. [175] also proposed a recent streaming framework that approximates full video hierarchical segmentation to work in a constant memory space. Their algorithm works in a feedforward streaming fashion by dividing an input video stream into a set of clips. Then, for each clip, a segmentation hierarchy is generated using an automatic semi-supervised method that uses a Markovian assumption to relate the current hierarchy to the previous one.

Despite the previous progress in developing online vision algorithms, there are also challenges resulting from user-defined vision algorithms. For example, the algorithms can have unpredictable or high processing rates based on the input size and content, which causes some algorithms to employ different approximations to support real-time requirements. Many algorithms also have several runtime parameters, which are selected by isolated experimental evaluations, and usually lack runtime tuning when integrated into other algorithms. Another important challenge is the need to design vision algorithms for heterogeneous hardware platforms (mobile to heterogeneous hardware processors), which makes the algorithms hard to debug and extend.

In computer vision, an example set of problems that face the previous challenges are pixel-labelling problems. These problems define several fundamental computer vision operations that include stereo vision, optical flow, and image segmentation. A pixel-labelling problem takes a set of labels $\mathcal{L}$ and a set of pixels P as input. An assignment cost is then calculated for assigning each pixel $p \in P$ to a label $l \in \mathcal{L}$. This defines a 3D cost volume with depth $|\mathcal{L}|$ and each slice $l \in \mathcal{L}$ has the assignment costs of l to pixels in P. The problem is then to find the best assignment that minimizes the total assignment cost.

In the case of stereo vision and optical flow, the set P defines a pair of input images, whereas the set $\mathcal{L}$ defines a range of disparities and displacements, respectively. The disparity is defined as the distance between two corresponding pixels in the left and right images of a stereo pair. For image segmentation, the aim is to partition an image into a set of segments that are known as superpixels. In its simple case, P defines a single-input image, and $\mathcal{L}$ is binary with two labels for defining foreground and background regions. For the task of semantic image segmentation [187], $\mathcal{L}$ represents multiple labels corresponding to different object classes in the scene. Oftentimes, subpixel accuracy is required for optical flow and stereo vision, which makes the label space size very large and requires efficient processing.

Moreover, dealing with large-displacement labels requires expanding the label space, which further increases the size of the cost volume. This makes current accurate algorithms for stereo vision [23, 91, 122] and optical flow [170, 15, 143, 60, 18] slow and inefficient for online processing of vision streams.

However, much research has been done to accelerate performance. For example, Bao et al. [18] developed a fast hardware-accelerated algorithm for optical flow, but at the cost of sacrificing accuracy. This makes their approach unusable in many cases where accurate optical flow is required.

FlowNet [60] is another method that uses deep learning and hardware acceleration to post runtime performance. FlowNet requires a large set of training images with ground truth, which, as indicated by [60], is difficult to obtain in practice. This lack of training data results in the inability of the learned model to generalize on different datasets. We envision that an online algorithm for solving pixel-labelling problems should leverage sparse processing of cost volume to achieve faster runtimes, have comparable accuracy to the state of the art, and be able to trade off accuracy for speed to handle unpredictable or high processing rates.

Automatic visual surveillance is another area of computer vision with similar challenges. It is not surprising that traffic cameras are being installed in increasing numbers on roads and highways in and around big urban centres. This trend for installing and using traffic cameras will continue, and the number of traffic cameras will continue to increase. Such trends cause the data collected by the smart cameras to experience unprecedented growth and require automated techniques for consuming and analyzing this data. Automatic detection of road boundaries is a fundamental task in automatic visual surveillance and can greatly help subsequent traffic-analysis tasks, such as determining vehicle flow, erratic driving, stranded vehicles, etc.

However, developing online algorithms for this task is challenging due to the different environmental and lighting conditions in incoming video streams (e.g., unlit highways captured at night). For example, recently, Kong et al. [109] developed a method for detecting road boundaries using Gabor filters; however, the method is slow and has a degraded accuracy when dealing with noisy images from challenging environmental conditions.

Brust et al. [39] developed a deep learning-based scheme (CN24) for automatically detecting road boundaries. The CN24 scheme computes a confidence map that assigns the likelihood of belonging to the road region to each pixel. Still, CN24 has slow runtime and performs poorly on some traffic scenes with challenging environmental conditions. We think that future algorithms for automatic visual surveillance should be online, adaptive, and able to handle different environmental conditions in incoming traffic-video streams.

1.2 Contributions

This thesis has four main contributions. First, we develop a stream-algebra framework for manipulating vision streams, which can be used to mathematically describe the pipelines of several state-of-the-art techniques in computer vision. To our knowledge, we are the first to propose this algebra for computer vision tasks. The algebra has a common notation and defines a set of concurrent algebraic operators that provide a new abstraction for computer vision operations and can be used to build scalable computer vision pipelines. The developed algebra extends the algebra frameworks in databases and provides operators for controlling data flow rates. It also provides a natural description of feedback control, which is the fundamental task for several advanced optimizations, such as adaptive optimization and parameter tuning.

Second, we develop new algorithms for efficiently processing image and video streams in two areas of computer vision: pixel-labelling problems and automatic visual surveillance. For pixel-labelling problems, we develop the sub-volume cost-filtering approach for solving both stereo-vision and optical-flow problems. The approach leverages sparse processing of cost volume to achieve faster runtimes with comparable accuracy to the state-of-the-art algorithms. For automatic visual surveillance, we develop a new online algorithm for automatic lane and road-boundary detection. The algorithm runs in real time and is adaptive and able to handle several challenging environmental conditions. It outperforms other state-of-the-art approaches for automatic lane and road-boundary detection.

Third, we describe an implementation of the stream algebra and use the developed computer vision algorithms as case studies. Each algorithm is described as a streaming workflow and implemented using our algebraic operators. A throughput versus latency analysis is also performed to show the performance gains that the algebra provides to each developed workflow.

Finally, we use the streaming workflow of our automatic lane and road-boundary detection algorithm as a case study for developing common optimization methods for parameter tuning of large-scale pipelines. We show that the feedback primitives of the developed stream algebra can effectively implement and scale the sequential model-based optimization methods [95, 94, 21, 96] for parameter tuning of the stream-processing functions in large-scale computer vision pipelines.

The rest of the thesis is organized as follows. Chapter 2 gives a brief overview of the existing work on data-stream processing algebras and systems. It also discusses the current approaches for solving pixel-labelling problems and automatic road-boundary detection. A summary of general parameter-tuning algorithms is also given. Chapter 3 presents our stream-algebra formulation for computer vision streaming pipelines and shows how it can efficiently describe several state-of-the-art techniques in computer vision. The chapter also presents the feedback-control primitives of our algebra and its powerful description of state-of-the-art computer vision algorithms implementing adaptive optimization and parameter tuning. Chapter 4 presents our online algorithms for solving pixel-labelling problems and automatically detecting road boundaries from input video streams. We refer to these algorithms as algebra functionals and present several results that show the efficiency of the algorithms in processing image and video streams while maintaining accuracy comparable to the state of the art. Chapter 6 shows a case study that implements our road-boundary detection algorithm using the algebra framework. The chapter also presents our general parameter-tuning algorithm for online computer vision systems and presents several results that demonstrate its effectiveness. Chapter 7 summarizes our most important findings and offers a discussion of the most promising directions for improving our work.

1.3 List of Publications

The work presented in this thesis has appeared in the following conference and journal publications:

1. Fast Estimation of Large Displacement Optical Flow Using Dominant Motion Patterns & Sub-Volume PatchMatch Filtering, Mohamed A. Helala, Faisal Z. Qureshi, 14th Conference on Computer and Robot Vision (CRV), Edmonton, Alberta, Canada, May 2017 (Best Paper Award).

2. A Formal Algebra Implementation for Distributed Image and Video Stream Processing, Mohamed A. Helala, Ken Q. Pu, Faisal Z. Qureshi, Proc. 10th ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC), Paris, France, September 2016.

3. Automatic Parsing of Lane and Road Boundaries in Challenging Traffic Scenes, Mohamed A. Helala, Faisal Z. Qureshi, Ken Q. Pu, SPIE Journal of Electronic Imaging, 2015.

4. Accelerating Cost Volume Filtering Using Salient Subvolumes and Robust Occlusion Handling, Mohamed A. Helala, Faisal Z. Qureshi, 12th Asian Conference on Computer Vision (ACCV), Singapore, Nov. 2014.

5. Towards Efficient Feedback Control in Streaming Computer Vision Pipelines, Mohamed A. Helala, Ken Q. Pu, Faisal Z. Qureshi, 2nd Workshop on User-Centered Computer Vision (UCCV) in conjunction with Asian Conference on Computer Vision (ACCV) 2014, Singapore, Nov. 2014.

6. A Stream Algebra for Computer Vision Pipelines, Mohamed A. Helala, Ken Q. Pu, Faisal Z. Qureshi, 2nd Workshop on Web-scale Vision and Social Media (VSM) in conjunction with CVPR 2014, Columbus, Ohio, June. 2014.

7. Road Boundary Detection in Challenging Scenarios, Mohamed A. Helala, Ken Q. Pu, Faisal Z. Qureshi, Proc. 9th IEEE International Conference on Advanced Video and Signal-Based Surveillance (AVSS), Beijing, China, Sept. 2012.

The following publications are a result of the work carried out during the course of my doctoral studies. We chose not to include these works in this thesis:

1. Constructing Image Mosaics Using Focus Based Depth Analysis, Mohamed A. Helala, Faisal Z. Qureshi, IEEE Winter Conference on Applications of Computer Vision (WACV), Lake Placid, NY, USA, March 2016.

2. Mosaic of Near Ground UAV Videos Under Parallax Effects, Mohamed A. Helala, Luis A. Zarrabeitia, Faisal Z. Qureshi, Proc. 6th ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC), Hong Kong, China, Oct. 2012.

Chapter 2

Background

2.1 Stream Algebra

Communicating sequential processes (CSPs) [88] is one of the early defined formal languages for algebraic description of communication patterns in concurrent systems. It is the parent language that is extended by all database stream-algebra frameworks [52, 38, 43, 55]. In addition, CSP belongs to a general mathematical framework of concurrency known as process algebra. In CSP, systems are described using their component processes with the assumption that they work independently and concurrently. Each independent process either runs sequentially or is composed of other concurrent primitive processes. Processes then interact with each other using the process algebra operators. A simple example of a CSP is a pipe, or pipeline, which defines a sequence of processes. Each process receives inputs only from its predecessor and produces outputs only to its successor.

The process algebra notation of CSP defines two main primitives: events and primitive processes.

Definition 2.1.1 (Primitive Process) *A primitive processes P represents fundamental operations or behaviours.*

Example 2.1.2 *STOP and SKIP are two popular CSP primitive processes. STOP is the process that communicates nothing, and SKIP is the process that signals successful termination.*

Definition 2.1.3 (Events) *In terms of concurrency, events represent communication between processes and are described by a pair c.v, where c is the communication channel and v is the message passed.*

Example 2.1.4 *The equation* $\alpha c(P) = \{v | c.v \in \alpha P\}$ *defines the set of all messages* v *that can be communicated to process* P *on channel c, where* αP *defines all messages that must be created before engaging process* P*.*

Moreover, CSP also defines a set of algebraic operators that mathematically define the permissible methods for constructing concurrent systems using events and primitive processes. The operators include Prefix, InterfaceParallel, Interleaving, Hiding, Deterministic-Choice, and Nondeterministic-Choice.

Definition 2.1.5 (Prefix) $(x \rightarrow P)$*. This operator combines one or more events and a process to form a new process. For example, the new process* $(x \rightarrow P)$ *specifies that the event* x *must first occur before engaging process* P*.*

Definition 2.1.6 (InterfaceParallel) $P||\{a\}||Q$*. This operator specifies that both processes* P *and* Q *run concurrently and can synchronously engage in event* a*.*

Definition 2.1.7 (Interleaving) $P|||Q$*. This operator represents the interleaving form of concurrency, where the two processes* P *and* Q *accept the same input, and their execution is arbitrarily and concurrently interleaved. We alternate the communication over either channel* P *or channel* Q*, whichever is first available.*

Definition 2.1.8 (Hiding) $P/\{c\}$*. Given a process* P *that accepts a set of events, the Hide operator drives an abstraction of P such that the environment observes one or more* *input events c and returns an abstract process representing the activities following c in P. For example, if $P = (c \rightarrow \mathrm{STOP} | d \rightarrow a \rightarrow \mathrm{STOP})$, then $P/\{d\} = (a \rightarrow \mathrm{STOP})$, where $|$ is the Boolean or operator.*

Definition 2.1.9 (Deterministic-Choice) $(a \rightarrow P) \square (b \rightarrow Q)$. *This is a composite operator that allows the environment to choose between two component processes. Given an initial event, the environment resolves the choice by selecting the process that can engage the event.*

Example 2.1.10 *The process $(a \rightarrow P) \square (b \rightarrow Q)$ is a process that can accept either event a or b and communicate to its corresponding component process $(a \rightarrow P)$ or $(b \rightarrow Q)$, respectively. If both component processes can accept the same input, the choice is resolved nondeterministically.*

Definition 2.1.11 (Nondeterministic-Choice) $(a \rightarrow P) \sqcup (a \rightarrow Q)$. *This operator allows the choice between two component processes similar to the Deterministic-Choice; however, the processes must have the same input events, and the choice is made arbitrarily without any control from the environment.*

The CSP operators define a formal syntax for writing legal CSP expressions. They also provide a mathematical framework for describing and analyzing large-scale concurrent systems. This framework influenced the design of several programming languages, such as the Go language. [1] It was also used to describe and verify the concurrency patterns of several industrial systems [20].

Building upon the concepts of CSP, the database community formulated algebra frameworks [52, 38, 43, 55] for processing and manipulating data streams using the concepts of stream processing, such as pipelines. In such cases, a data stream represents a finite or infinite sequence of data items or tuples, and a pipeline is a chain of processes running concurrently with the output of each process used as the input of the

[1]Go language: https://golang.org/ (*last accessed on 7 September 2017*).

next process. For example, Broy et al. [38] used data-flow networks to model concurrent computation of data-streaming pipelines. They studied the algebra of data-flow networks by representing the streaming pipelines as a graph of stream-processing functions. Their algebraic framework is deduced from basic network algebra (BNA) [152], which is a popular framework for representing data-flow networks using the calculus of flownomials. This calculus is used to describe concurrency and behaviour of directed flowgraphs in both deterministic (synchronous) and nondeterministic (asynchronous) cases.

Carlson and Lisper [43] presented an event algebra for detection of composite events in reactive systems. Examples of reactive systems include real-time and embedded systems, where execution is controlled by external events. These systems must detect events and react to them with appropriate responses. The input to such systems in most cases is a complex set of events referred to as complex events. The event algebra allowed the description of such composite events, and a transformation algorithm is used to convert the algebraic expressions into a form that can be processed by systems with limited resources. The algebra includes five operators: (1) disjunction $A \vee B$, denoting the occurrence of either event A or B; (2) conjunction $A + B$, denoting the occurrence of both event A and B; (3) negation $A - B$, denoting the occurrence of A under no occurrence of B; (4) sequence $A; B$, denoting the occurrence of B after A; (5) temporal restriction A_τ, referring to the occurrence of event A for a number of times less than τ. Given a composite event expression, the transformation algorithm converts the expression into an event stream form that meets predefined bounds on computational resources.

Demers et al. [55] presented another algebra for processing arbitrary event streams. The algebra has a data model for representing event streams and a set of operators for processing and transforming streams. They defined a data stream as follows:

Definition 2.1.12 (Data stream) *A data stream is an infinite set of events $\langle \bar{a}, t_0, t_1 \rangle$, where $\bar{a}$ is a relation following the database relational model, t_0 denotes the start time of the event, and t_1 is its end time.*

The algebra operators include unary and binary operators. The unary operators are only applied to the relation part of every event in the incoming stream to produce a corresponding event in the output stream. The algebra also defined aggregate functions used to process a sequence of events, as in SQL. They presented a case study that showed the effectiveness of their algebra in describing and implementing queries over stock-quote streams.

2.1.1 Feedback Control in Data-Stream Systems

Feedback control is a very important area in the field of control systems and has a wide range of applications in several industrial plants. The main goal is to monitor and control a process by inspecting its output and utilizing it as a feedback signal. This signal is compared against the desired output using a controller that measures the error and feeds it to the controlled process to adjust the output to the desired response. This type of control system that uses a feedback signal to self-adjust its output is called a *closed-loop control system*, also referred to as a *feedback-control system* [126].

Feedback-control systems can have one or more feedback loops. Such systems have controlled processes that are univariate or multivariate [126]. Univariate processes have a single controlled input variable (or parameter) and produce a single output variable. Thus, only one feedback loop with a single controller is required to self-adjust the output variable. The resulting control system in this case is referred to as a *single-loop feedback-control system*. On the other hand, multivariate processes are typically found more often in practice, where a process has multiple controlled input variables and produces one or more output variables. Moreover, in more complex scenarios, a multivariate process may have feedback signals coming from the outputs of other processes. To handle such cases, the control system must have multiple feedback loops and is referred to in literature as a *multi-loop feedback-control system*.

Feedback control is one of the main building blocks for optimizing data-streaming pipelines. This enables tasks such as parameter tuning and incremental optimization. The main benefits of database stream algebras are the ability to define common pipeline optimization and cost models. They can also define formal methods for implementing dynamic execution plans and scaling up data processing. Feedback control can also be described by database stream algebras.

Broy et al. [38] provided a theoretical definition of a feedback operator to describe feedback loops. This work was extended to apply feedback control in database systems [102, 114, 164, 165, 100, 84]. Kapitanova et al. [102] developed a formal language for modelling data-stream queries. They also applied feedback control to develop management mechanisms for QoS. Yi et al. [164] applied feedback control to maintain an acceptable QoS for continuous queries in stream-processing systems. Their method controls and adapts the application behaviour in unstable system conditions. Examples of such conditions include overloaded systems, where load shredding should be applied to ignore some inputs to maintain the desired QoS. Yi et al. [165] extended the work by [164] by applying feedback control to guide load shedding, while reducing the data-processing delays in stream-processing systems. Li et al. [114] developed a feedback-control strategy to schedule continuous and one-time queries executed under predefined timing constraints. Their strategy minimizes the number of query deadline violations, while improving the query quality. All these example methods build upon the concepts of database stream algebra to efficiently use feedback control to optimize textual stream processing.

2.2 Computer Vision Functionals

There is a need for scalable and efficient computer vision algorithms to build large-scale computer vision pipelines. However, several computer vision algorithms are computationally expensive. In this section, we review examples of computationally expensive algorithms that include road-boundary detection in traffic surveillance and two application areas of pixel-labelling problems, optical flow and stereo vision. As each application has a large body of literature, we only focus on the methods relevant to our work.

2.2.1 Stereo Vision

Given a stereo image pair with a left image I_1 and a right image I_2, the task of stereo vision is to assign each pixel $p \in I_1$, a disparity or displacement that defines its corresponding pixel $p' \in I_2$. The images are usually rectified by transforming and aligning them such that corresponding pixels appear on the same row. So, the disparity is only estimated along the horizontal axis. Stereo vision is usually formulated as pixel labelling problem where an assignment function $f : P \rightarrow \mathcal{L}$ is defined to map each pixel in the set of image pixels P, a label $l \in \mathcal{L}$ in the label space $\mathcal{L}$ that defines the disparity range. The solution to this labelling problem is usually referred to as a disparity map.

In stereo vision, global energy minimization based on Markov random fields (MRFs) is a popular approach for solving pixel-labelling problems. In this approach, the labelling assignment energy is defined as,

$$E(f) = E_d(f) + E_s(f), \tag{2.1}$$

where E_d is a data cost energy that penalizes wrong label assignments, and E_s is a smoothness cost energy E_s that penalizes the assignment of different labels to neighbouring pixels. For example, Ben et al. [23] proposed a global energy minimization method based on a variational framework. The method defines two coupled energy functionals for both stereo matching and occlusion handling (OH). The data term for the stereo matching energy uses a robust L^1 norm to encode differences in colour and gradient, whereas the smoothness term uses a total variation regularizer that encodes the difference in disparity labels. The OH energy is formulated as a diffusion process and a separate minimization process fills the occluded gaps.

Pal et al. [134] provided a similar idea by experimenting with different optimization methods, such as belief propagation variational message passing [172], and graph cuts [54]. They proposed the sparse variational message-passing method to reduce the optimization time. However, the runtime is still over 100 seconds.

Taniai et al. [159] improved the efficiency of MRF inference for stereo vision using graph cuts by introducing locally shared labels. The idea is to assign each pixel and region a set of randomly initialized labels. Then, a disparity map is estimated by spatial propagation and refinement. The method provides better accuracy than [134], however, at a much higher computational cost.

Cech et al. [45] proposed a global optimization method that processes a small fraction of the disparity space to produce a semi-dense disparity map. Their method starts by finding a set of correspondence seeds in the disparity space. Then seed growing is performed by traversing seeds' neighbouring disparities that provide the minimum matching scores. The method achieved two orders of magnitude faster performance than exhaustively searching the entire disparity space.

The global optimization methods, however, do not scale well with large label spaces usually found in stereo-vision and optical-flow problems. Local stereo matching methods provide fast alternative techniques for solving pixel-labelling problems. Historically speaking, local stereo matching methods appeared before the global optimization methods. These methods were summarized by Brown et al. [36] into three main categories: (1) block matching, (2) feature matching, and (3) gradient-based optimization. The block matching methods estimate the motion at a certain pixel by defining a patch around the pixel and searching for corresponding patches in the other image [182, 28]. The gradient-based optimization methods minimize a function that is typically chosen as the sum of squared differences over a small local region [124]. The feature matching methods find corresponding features between the given images [139, 30].

Hirschmuller et al. [86] proposed a real-time local stereo block matching method.

The method builds upon standard local window correlation that matches local windows around pixels under the assumption that all pixels inside the matching window have the same disparity. This assumption, however, fails at object borders. The method handles this weakness by developing three extensions: (1) an approach for multiple supporting window matching is developed to increase correct matches and reduce errors at object borders, (2) an error filter is used to invalidate uncertain matches, (3) a post-processing step is applied to perform border correction and improve object borders.

The early work of Yoon and Kweon [179] proposed a method for weighted-cost aggregation by adaptively defining a support window for each pixel. For each window, the weights of pixels are calculated by comparing the colour and spatial distance to the centre pixel.

Several studies have been proposed based on the same idea, which includes the method by Hosni et al. [90] (see surveys in [162] and [163]). However, the main limitation of this early work is the large computational cost required to calculate weights and aggregate costs with a complexity dependent on the matching window size.

The recent development of edge-aware filtering (EAF) methods [74, 136] provided a fast and efficient alternative for cost filtering and aggregation. EAF performs image filtering while maintaining the intensity changes and preserving the edges of a given guidance image. Hosni et al. [91] proposed the cost-volume-filtering framework that applies EAF to perform edge-aware smoothing of the assignment costs. The framework uses the input image as the guidance image and efficiently applies EAF using the guided image filter [74] that has a complexity independent of the filter size. Lu et al. [121] extended the work by further speeding up cost filtering and aggregation by restricting the filter to a set of points in a shape-adaptive support window. Despite the efficiency of local cost-volume-filtering methods against MRF global approaches, they linearly scale with the label space size. This makes them unfeasible for handling the large cost volumes usually found in high-resolution images and solutions with subpixel accuracy.

Several recent studies have attempted to improve the computational time of cost-volume filtering. Min et al. [131] reduced the complexity of cost aggregation for stereo vision by introducing a histogram-based disparity pre-filtering scheme. This scheme filters a restricted set of candidate disparities for each pixel. The complexity of weighted filtering is also reduced by sampling the matching window. Although the method provides better performance than [91], it traverses the entire cost volume to build the histogram.

Lu et al. [123] sped up filtering by integrating the *PatchMatch* randomized search with EAF. The method uses superpixel segmentation for a compact representation of image regions. Then, *PatchMatch* random search and propagation is applied at the superpixel level to propagate and refine disparity labels. The speed increase results from the sublinear complexity of *PatchMatch* in the label space size.

The framework of Anandan [7] is one of the early work in developing the coarse-to-fine strategy for stereo matching. The framework starts by obtaining rough motion estimates from a large-scale intensity image level. The estimates are then refined by intensity information at smaller scales and propagated to the neighbouring pixel under the smoothness constraint to produce the output disparity map.

Sizintsev et al. [150] proposed a coarse-to-fine refinement procedure that improves binocular disparity estimates near 3D surface discontinuities. The approach builds upon the standard coarse-to-fine block matching framework by adapting match window support across scales to reduce errors in disparity estimates near boundaries. The approach is also extended to handle regions with half-occlusions and colour uniformity.

Furuta et al. [63] proposed a coarse-to-fine strategy for cost-volume filtering to efficiently handle large label spaces. Their method builds upon the idea that different scales should have correlated labels. Thus, they use the disparity output at lower scales to discard unimportant labels at the original scale.

Occlusion handling is also an important step for generating accurate disparity maps. The idea is to fill the gaps found at mismatched locations. Several techniques have been proposed for OH, for example, Sun et al. [156] developed a global energy minimization method that fills occlusions such that the left and right images are visibly consistent. This is performed by adding a term in the energy formulation that encourages smoothness in filling occlusions or gaps.

Min et al. [132] handled occluded regions similarly by introducing an energy term and applying an iterative optimization scheme for filling gaps. Yang et al. [177] proposed a global energy minimization framework for stereo vision. The framework used an iterative refinement step to fill occluded regions based on colour segmentation and plane fit. Ben-Ari et al. [23] provided a similar energy minimization formulation with an energy term for handling occluded regions, and a solution is obtained using iterative optimization. Hosni et al. [91] handled occluded regions using a post-processing method. The method traverses the disparity map row by row and fills occluded pixels by the smallest disparity of the closest non-occluded pixels. This results in some undesired artifacts, which are minimized by applying weighted-median filtering to obtain the output disparity map.

2.2.2 Optical Flow

For optical flow, we are also estimating displacements of pixels between two input images; however, the displacements are 2D vectors representing motion in both horizontal and vertical axes. A large body of literature for optical-flow estimation exists; thus, we only focus on techniques relevant to our work dealing with large label spaces typically found in high-resolution images and large-displacement optical flow. For a more comprehensive survey, we refer the reader to [62].

The multiscale coarse-to-fine strategy is a popular method for dealing with large-displacement optical-flow estimation [5]. This strategy constructs a multiscale pyramid for the input images. Motion estimation starts at the coarse resolution that has the slowest motion velocity. The labels are then propagated to finer resolution to obtain more accurate motion flow. This strategy, however, cannot deal with small and thin structures, which are usually lost at the coarse resolution.

To deal with this problem, Steinbruecker et al. [153] identified correspondences between input images. The method avoids the linearization formulation of optical-flow introduced by Horn and Schunck [89] that is only valid for small motions. It also avoids the need for coarse-to-fine warping of one image to another. However, finding correspondences is expensive, which makes the method inefficient in handling large motion fields found in large resolution images. Brox and Malik [37] applied keypoint matching within a variational model to handle small and thin structures and handle large motion ranges. Xu et al. [174] improved the method proposed in [37] by incorporating a series of discrete fusion moves.

Weinzaepfel et al. [170] proposed DeepFlow, which relies on dense feature matching to deal with large-displacement optical-flow estimation. DeepFlow uses DeepMatching, which has $O(M^2)$ space and time complexity, where M is the number of pixels [142]. It therefore requires several more orders of magnitude of memory than other state-of-the-art methods.

The PCA-layers method developed by Wulff et al. [173] provided an efficient algorithm that relies on an offline learning stage to estimate a principal component analysis (PCA) model from sparse feature matches. A dense optical flow is estimated from a layered flow model that uses sparse matches and the learned PCA model.

Yang et al. [99] also provided a more accurate algorithm that uses piecewise homography models to estimate optical flow. The algorithm shows improvements regarding non-translational motions with strong projections, however, at a large computational cost of about 500 seconds.

PatchMatch [19] and its variants [19, 155, 110] can efficiently compute approximate nearest neighbour fields (ANNFs), where the target is to find for each $k \times k$ patch in one image, its corresponding patch in another image, and k is the patch radius. ANNFs can be used to set up coarse correspondences between images. The methods of [50, 18, 27] employ

ANNFs to compute initial optical flow. In addition, ANNF-based approaches attempt to minimize dissimilarity between patches without enforcing spatial coherence (i.e., patches in one image may have corresponding patches at arbitrary locations in the other image). Chen et al. [50], who used ANNFs to set up the initial optical flow, addressed this issue through motion segmentation. They set up a global optimization problem whose solution gives state-of-the-art large-displacement optical-flow results. This method, however, has very high computational costs. Besse et al. [27] also proposed a method that relies upon ANNFs to compute the initial optical flow. Belief propagation is used to refine the initial optical flow.

The EAF methods [136, 74] have been shown to provide a fast alternate to the global energy minimization techniques for solving pixel-labelling problems [91, 123, 18]. These methods are often referred to as cost-volume filtering. Hosni et al. [91] successfully implemented a cost-volume-filtering framework that applies the guided image filter [74] for optical flow and stereo-disparity estimation. Their framework has fared well on the Middlebury benchmarks [17].

Cost-volume filtering methods scale linearly with the size of the label space. This renders these methods inefficient for dealing with high-resolution images or computing motion detail, preserving optical flow. SimpleFlow [160] attempts to accelerate the filtering process by providing a sublinear solution, albeit at the cost of reduced accuracy.

Lu et al. [123], on the other hand, reduced computational cost by randomly picking out candidate regions in the cost volume. The regions are picked using a PatchMatch [19] search over the entire cost volume. Computational savings are minimal as the entire cost volume is searched.

More recently, Bao et al. [18] proposed an algorithm that also integrates EAF with PatchMatch. They can achieve this speed increase by applying a hierarchical matching step that downsamples the input images (Step 1). For each pixel in one downsampled image, the search is restricted to similar pixels in the other image during the ANNF setup

(Step 2). Finally, labels are propagated from the downsampled image to the original image (Step 3). Both Steps 1 and 2 adversely affect the accuracy of this method. The approach presented in [123] is computationally expensive, as it sets up ANNFs by random search over the entire cost volume. The method in [18] trades off speed for accuracy and provides a graphics processing unit (GPU) implementation that is about 100 times faster than the method in [123].

Optical-flow estimation algorithms that use ANNFs to set up correspondences suffer from the spatial coherence problem. This stems from the fact that an ANNF search does not constrain the search radius for finding correspondences. To address this issue and to enforce spatial coherence, [50] uses motion segmentation, and [123] and [18] rely on EAF.

2.2.3 Road-Boundary Detection in Traffic-Video Surveillance

The previous methods developed for automatic lane and road detection can be classified into three main categories: (1) activity-driven, (2) feature-driven, and (3) model-driven. This section provides a brief survey of the techniques developed in each category. We refer the reader to [103, 85] for a more detailed survey.

Activity-driven methods: The activity-driven techniques [51, 130, 154] use vehicular motion to build an activity map for the traffic scene and divide the road region into active (road) and inactive (non-road) regions. The work by Stewart et al. [154] developed one of the earliest activity-driven methods. Their method accumulates an activity map that records scene changes resulting from vehicular motion. Then, the traffic scene is divided into either active or inactive areas.

Melo et al. [130] built on the idea by [154] and developed a method that incorporates the Kalman filter to track moving vehicles. Then, they modelled the resulting motion trajectories using second-degree polynomials and applied K-means clustering to calculate lane centres. Recently, Chen et al. [51] extended the work by [130] by developing a trajectory similarity distance to improve clustering.

Feature-driven methods: The feature-driven methods [6, 109, 146] rely on low-level image features, such as colours and textures to detect the lane and road boundaries. Aly [6] developed a method for detecting lane marks in urban roads. His method applies lane analysis using selective regions, which requires camera calibration and uses inverse perspective mapping to construct a top view of the road. Next, the algorithm applies image filtering and thresholding to extract lane features. Finally, the random sample consensus (RANSAC) algorithm is used to ignore feature outliers and fit polynomials to lane boundaries.

Satzoda et al. [146] proposed a similar method to [6] for detecting lanes. However, their work processes selected image bands and applies steerable filters, which are orientation-selective convolution kernels, to extract lane features. They also used a lane geometric model to deal with feature outliers. Kong et al. [109] developed an alternative method that divides the road-detection problem into two tasks: (1) estimation of the road vanishing point and (2) segmentation of the road region based on the estimated vanishing point. The estimation task applies Gabor filters to extract texture orientations and feeds them into a soft voting scheme to estimate the vanishing point. Gabor filters are bandpass filters that analyze an image at different scales and orientations. The segmentation task uses the detected vanishing point as a constraint to identify the dominant road boundary.

Model-driven methods: The model-driven methods [189, 169, 39, 4] perform either road classification or model fitting. Road classification aims to learn a prior model for road regions, which is used later to assign each pixel a likelihood of belonging to a road region. For example, Brust et al. [39] presented an algorithm that uses convolutional neural networks to classify image patches as belonging to either road or non-road regions. The algorithm learns a prior model that incorporates both spatial and appearance information of image patches belonging to road regions. Then, the neural network generates a classification map that assigns the likelihood of belonging to a road region to each pixel.

Model fitting methods [189, 169, 4] match a geometric road model to the traffic scene. Wang et al. [169] proposed a lane detection method. This method applies edge detection on the input image and partitions it into several horizontal segments. Then, the algorithm assumes perspective parallel lane lines and detects a set of control points along the mid-line of the lane. These points are used to define an active contour model, which is a contour that alters its shape and position to obtain minimal energy state. In this case, energy minimization is used to deform the contour to both the left and right to detect the lane boundary.

Zhou et al. [189] proposed another lane geometrical model that has four parameters: starting position, lane original orientation, lane width, and lane curvature. Their algorithm has three stages: (1) offline calibration to estimate the camera parameters, (2) model parameter estimation to locate the lane width and dominant orientation, and (3) model matching to find the best lane model. Recently, the method by [4] extends the previous techniques by using geographical information to estimate several road priors. Then, it develops a road generative model that combines the road priors with other contextual cues extracted from the traffic scene, such as horizon lines, lane marks, and vanishing points. The generative model is used to construct a confidence map that assigns each pixel a likelihood of belonging to a road region.

2.3 Parameter Tuning

Parameter tuning is one the important tasks in building large-scale computer vision pipelines. This is because the pipeline needs to adapt to changes in input vision streams, such as different lighting, environmental, and scene conditions. This section provides a review of the parameter-tuning problem, which is also referred to in literature as the algorithm configuration problem.

Let us assume an algorithm A is parametrized by a set of parameters $\theta \in \Theta$ belonging to a parameter space Θ. We define an input set of problem instances D and a target performance metric given by the function $m(\theta, \pi)$. This metric computes the performance of algorithm A on instance $\pi \in D$. The goal of the algorithm configuration is to find a $\hat{\theta}$ that optimizes the metric m on D. The metric m usually defines the running time or output accuracy of algorithm A. However, several challenges exist in configuring the parameters of many algorithms, which include that (1) algorithms may be expensive to compute [101, 94] and that (2) metric functions usually do not have closed-form representations to calculate gradients.

Several approaches have been proposed for solving the algorithm configuration problem. These approaches can be classified into either model-based or model-free approaches. The difference is whether a model is used to describe the dependency between the parameter settings and the target performance objective.

Model-free approaches [2, 32, 31, 97, 96, 9] are popular due to their simplicity and the ability to use them out of the box. These approaches have been successful in optimizing several algorithms, especially for constraint programming problems. Examples of these methods include the methods by [2] and [32] which addressed the optimization of numerical parameters.

The method by [2] used a heuristic based on a local search procedure, whereas [32] proposed the racing algorithm, which begins with an input set of candidate parameter settings or configurations and iteratively evaluates them on a stream of input instances. This continues until enough statistical evidence of the performance of each setting is gathered. The configurations with poor performance are then eliminated, and the surviving configurations are used again as input. Several other methods have been also proposed for the tuning of categorical parameters, which include the racing algorithm by [31] and the iterated local search algorithm ParamILS by [97, 96]. Moreover, genetic algorithms have been used by [9] for parameter tuning.

Model-based approaches [95, 101, 94, 21, 96] have also been studied in literature. These approaches have the advantage of being more appealing than the model-free approaches. This is due to their ability to interpolate the response surface of the target performance metric over the parameter space using a small set of tested parameter settings, providing intuition regarding the response of unseen parameter settings that cannot be determined using model-free approaches. This also provides the advantage of extrapolating to unseen locations of the parameter space to optimize the target objective function over the parameter space. Several model-based approaches have been proposed for solving the algorithm configuration problem.

A popular approach is sequential model-based optimization (SMBO) [95, 94, 21, 96], which iterates between two main steps: fitting a model and using the model to select parameter settings. Initially, the fitting stage interpolates the performance metric using a set of initial settings with their measured metric values. Then, extrapolation is used to find any unseen optimal setting, which is added to the initial set of parameters after measuring its metric value. The main advantage of this approach is that it treats algorithms as blackbox functions with no closed-form representation.

The SMBO approach, which was first proposed by Bartz-Beielstein et al. [21], uses an efficient global optimization algorithm based on the work by [101] for optimizing blackbox functions using Gaussian process (GP) models. They also developed a sequential parameter optimization (SPO) toolbox that included an automated SMBO algorithm for tuning numerical parameters. The work by [95] extended the work by [21] by proposing the time-bounded SPO method that reduces the runtime overhead required for computing the GP response-surface models. This is done by using approximate GP models and randomly sampling parameters during optimization. The method is also time-bound, thereby forcing a user-defined time budget on every run of the algorithm parameter optimization. Thus, early terminating parameter settings resulting in high computational costs.

Random forests [34] have also been used with several other methods [22, 93] to model response surfaces. Random forests are similar to decision trees but with real values as their leaves. They have been shown to provide good performance for optimizing categorical parameters [22, 93].

Chapter 3

Scalable Computer Vision Systems

This chapter presents a stream algebra for the formal description of computer vision pipelines. The algebra is defined using the mathematical definitions of stream processing [52, 38, 43, 55] and communicating sequential processes (CSP) [88]. These definitions have been found useful in developing several programming languages and concurrency models. For example, CSP was the basis for the Go language concurrency model. Stream processing also influenced the design of several algebra frameworks in databases for processing queries over event and relational streams (see Section 2.1). Our stream algebra has three main components: 1) a common notation for expressing workflows, 2) a set of data-processing and flow-control operators, and 3) a set of formal semantics used to write workflow expressions.

The outline of this chapter is as follows. Section 3.1 presents our algebra and its main components. It also discusses several state-of-the-art computer vision algorithms and shows how our stream algebra can effectively describe them using equations over vision streams. Section 3.2 shows how our algebra can naturally and efficiently describe feedback control. Section 3.3 gives a general discussion of the different optimizations enabled by the algebra for efficiently processing image and video streams. Finally, Section 3.4 presents the algebra implementation in the Go language.

3.1 Stream Algebra

Our stream algebra [76, 77, 78] consists of three main components: a common notation for expressing pipelines, a set of data-processing and flow-control operators, and the formal semantics used to write algebraic expressions. This section gives an overview of the notation and semantics of the algebra and the algebra operators.

3.1.1 Notation

Definition 3.1.1 (Data streams) *A data stream is defined as an infinite sequence of data tuples, which we can write to and read from using the functions:*

$$\begin{aligned} \lambda x &\rightarrow s \\ &\leftarrow s. \end{aligned}$$

For the algebra definition, we refer to the set of all streams as $\mathbf{S}$. A stream $S \in \mathbf{S}$ over a set I is an infinite sequence of elements of the set I. In our case, I represents the set of images, and S can be a sequence of unordered or ordered images. Also, zero-based indices define the location of elements in S, where $S[0], S[1], S[2], ...$ define the first, second, third, etc., elements. To signify that a stream contains tuples of a specific type T, we use the notation $\mathbf{S}\langle T\rangle$.

Definition 3.1.2 (Stream operator) *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.*

The following constructs are used to define the algebra operators:

Definition 3.1.3 (Atomicity) *We define a set of statements executed as an atomic operation using* `{` statements `}`.

Definition 3.1.4 (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, and* $\textbf{\textit{loop}}_j$ *designated as the* j*-th loop.*

Definition 3.1.5 (Shared state) *A state defined as* ***state*** u *indicates a shared state and can be accessed by the next loops.*

Definition 3.1.6 (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*.*

Definition 3.1.7 (Attribute access) *We use* $x.y$ *to access attribute* y *from the composite variable* x*.*

In addition, a streaming pipeline is defined as follows:

Definition 3.1.8 (Streaming pipeline) *A streaming pipeline is a graph* $G = (V, E)$ *with vertices* V*, representing operators, and edges* E*, representing the direction of data communication. We refer to this graph as the workflow graph.*

The operator definition and implementation follow the following notation:

Definition 3.1.9 (Operator declaration) *A stream operator* X *is a mapping function that can have zero or more parameters. The parameters are the user-defined functions and their functional parameters. Given the functional signature of each parameter, we derive a stream operator and declare it using the following format:*

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

The semantics of the derived operator are defined using the declared notational constructs that include atomicity, shared state, concurrency, stream I/O, and attribute access.

Our algebra operators are categorized into three main categories: 1) first-order, 2) higher-order, and 3) rate-control. To define the first-order and higher-order operators, we studied the database operators used to process relational data streams and redefined them to match the notation and semantics of our algebra. These database operators include the Map, Reduce, and Filter operators that are used for data transformations on relational streams [115, 46]. They also include the Scatter, Gather, and Merge operators that are usually used for data processing in data-parallel frameworks [73, 71]. For data rate-control, we studied the Latch operator introduced by [44] to interpolate tuples between the actual tuples of a given input stream. We also introduced two operators, Cut and LeftMult, which we think are important in describing and integrating computer vision algorithms working at different data flow rates.

3.1.2 First-order Operators

We start by formulating simple first-order operators that typically process a single-input stream to produce a predefined number of output streams.

Definition 3.1.10 (Map) *The operator* synchronously *reads from a single-input stream S_{in}, performs a user-defined mapping function on input tuples, and writes the computed value to an output stream. The operator is parametrized by a list of user-defined mapping functions $f : \textsc{List}\langle X \times P \to Y \rangle$ and a vector of user-defined parameters $p_0 : \textsc{List}\langle P \rangle$. Each function $f[i]$ receives the incoming tuple and one or more user-defined functional parameters of type P. These parameters control the behaviour of the user-defined function $f[i]$. The operator has two state variables i and p for holding the index of the current active mapping function and the vector of parameters for all mapping functions. A lookup function is used to extract a command section of each incoming tuple, which can change the current active function index and/or the user-defined functional parameters:*

$$\frac{f : \textsc{List}\langle X \times P \to Y \rangle}{\textsc{map}(f, p_0) : \mathbf{S}\langle X \rangle \to \mathbf{S}\langle Y \rangle}$$

```
**state**      $i = 0;\ p = p_0$
**loop** :     $x \leftarrow S_{\text{in}}$
               $j, y = \text{lookup}(x)$
               **if** $j$ is defined **then** $i = j$
               **if** $y \neq null$ **then** $p[i] = y$
               $f[i](x, p[i]) \rightarrow S_{\text{out}}$
```

Definition 3.1.11 (Reduce) *This operator is similar to Map, but it keeps track of an additional internal shared state* $u : U$ *and is parametrized by a list of mapping functions* $g : \text{List}\langle U \times X \times P \rightarrow U \times Y\rangle$ *and an initial state* u_0*:*

$$\frac{u_0 : U,\quad g : \text{List}\langle U \times X \times P \rightarrow U \times Y\rangle}{\text{reduce}(g, u_0, p_0) : \mathbf{S}\langle X\rangle \rightarrow \mathbf{S}\langle Y\rangle}$$

```
**state**      $u = u_0;\ i = 0;\ p = p_0$
**loop** :     $x \leftarrow S_{\text{in}}$
               $j, y = \text{lookup}(x)$
               **if** $j$ is defined **then** $i = j$
               **if** $y \neq null$ **then** $p[i] = y$
               $u, z = g[i](u, x, p[i])$
               $z \rightarrow S_{\text{out}}$
```

Notice that, if the input list of functions $f : \text{List}\langle X \times P \rightarrow Y\rangle$ to the Map or Reduce operator has an empty set of parameters P, we can simply omit this set and write the list of functions as $f : \text{List}\langle X \rightarrow Y\rangle$. Here, we assume that the input set of parameters to every function is optional. Moreover, a Map (f, p_0) or Reduce (g, u_0, p_0) operator that has an empty initial set of input parameters p_0 can have this set omitted and treated as optional. In such cases, we can simply write Map (f) or Reduce (g, u_0). Moreover, if a Map (f) receives a list of functions f that contains only one function $h = f[0]$, we can simplify expressions by writing Map (h). Similarly, this assumption applies to Reduce.

Definition 3.1.12 (Filter) *This operator is a special case of the Map operator and is parameterized by a predicate $\theta : X \to$ boolean. The operator has two output streams S^1_{out} and S^2_{out}. The incoming readings that meet the predicate are forwarded to S^1_{out} and others that do not are sent to S^2_{out}:*

$$\frac{\theta : X \to \text{Boolean}}{\text{FILTER}(\theta) : \mathbf{S}\langle X\rangle \to \mathbf{S}\langle Y\rangle \times \mathbf{S}\langle Y\rangle}$$

```
loop : $x \leftarrow S_{\text{in}}$
       if $\theta(x)$ then $x \to S^1_{\text{out}}$ else $x \to S^2_{\text{out}}$
```

Definition 3.1.13 (Source) *The 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 $h : U \to U \times Y$:*

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

```
state    $u = u_0$
loop :   $u, y = h(u)$
         $y \to S_{\text{out}}$
```

Definition 3.1.14 (Copy) *This operator synchronously reads and duplicates every input tuple to all outgoing streams. It is parametrized by the number of output streams:*

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

$$\begin{aligned} \textbf{loop} \quad : \quad & x \leftarrow S_{\text{in}} \\ & x \rightarrow S_{\text{out}}[i] \quad \text{for all } i \le n \end{aligned}$$

Definition 3.1.15 (Ground) *The operator destroys the incoming stream:*

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

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

3.1.3 Rate-control Operators

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

Definition 3.1.16 (Latch) *The operator takes a single incoming stream S_{in} and produces two outgoing streams S^1_{out} and S^2_{out}. For each incoming tuple, the operator writes it synchronously to S^2_{out} and asynchronously to S^1_{out}. For asynchronous writing, the operator duplicates tuples of the incoming stream if S^1_{out} has a faster data rate than S_{in} and samples the incoming stream when S^1_{out} is slower than S_{in}. Notice that sampling means that the output stream will lose some incoming tuples:*

$$\text{LATCH} : \mathbf{S} \rightarrow \mathbf{S} \times \mathbf{S}$$

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

Definition 3.1.17 (Cut) *This operator is similar to Latch; however, each incoming tuple is asynchronously written only once to S^1_{out}. When S^1_{out} has a faster data rate than S_{in}, nil is used for the extra writes:*

$$\text{CUT}() : \mathbf{S} \to \mathbf{S} \times \mathbf{S}$$

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

Definition 3.1.18 (LeftMult) *LeftMult has two input streams S^1_{in} and S^2_{in} and one output stream S_{out}. This operator applies a Latch on S^2_{in} to produce the streams S^1 and S^2. It then grounds S^1 and outputs pairs (x_1, x_2), where $x_1 \in S^1_{\text{in}}$ and $x_2 \in S^2$. Thus, the output data rate is dependent on S^1_{in} and is independent of S^2_{in}. This operator is a generalized sampling operator, as S^1_{in} can be thought of as a clock stream that samples S^2_{in}.* **RightMult** *can be similarly defined:*

$$\text{LEFT-MULT} : \mathbf{S}\langle X_1 \rangle \times \mathbf{S}\langle X_2 \rangle \to \mathbf{S}\langle X_1 \times X_2 \rangle$$

$$\begin{array}{lcl} S^1,\ S^2 & = & \text{LATCH}(S^2_{\text{in}}) \ ; \ \text{GROUND}(S^2) \\ \textbf{loop} & : & \begin{bmatrix} \leftarrow S^1_{\text{in}} \\ \leftarrow S^1 \end{bmatrix} \to S_{\text{out}} \end{array}$$

3.1.4 Higher-order Operators

Section 3.1.2 describes the first-order operators, where the number of input and output streams are predefined by the operator definition. The operators also are parametrized by simple functions. Any pipelined composition of first-order operators results in first-order operators as well. This section extends the algebra by presenting higher-order operators. These operators process collections of streams and have functions and first-order operators as their input parameters.

Definition 3.1.19 (Mult) *The operator has k incoming streams, 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:*

$$\textsc{mult}() : \mathbf{S}^k \to \mathbf{S}$$

$$\textbf{loop} \;:\; \begin{bmatrix} \leftarrow S_{\text{in}}[1] \\ ... \\ \leftarrow S_{\text{in}}[k] \end{bmatrix} \to S_{\text{out}}$$

Definition 3.1.20 (Add) *This operator also has k incoming streams and one output stream S_{out}. The operator asynchronously reads values from the incoming stream and performs the best effort to sequentially write them to the outgoing stream:*

$$\textsc{add}() : \mathbf{S}^k \to \mathbf{S}$$

$$\begin{aligned} &\text{for all } i \leq \textbf{len}(S_{\text{in}}) \\ &\qquad \textbf{loop} : \{x \leftarrow S_{\text{in}}[i]; x \to S_{\text{out}}\} \end{aligned}$$

Definition 3.1.21 (Scatter) *The operator synchronously receives an input stream and generates a list of output streams. The operator is parameterized by two functions: $f : X \to \textsc{List}\langle Y \rangle$ and $p : Y \to \mathbb{N}$, where 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:*

$$\frac{f : X \to \textsc{List}\langle Y\rangle \quad , \quad p : Y \to \mathbb{N}}{\textsc{scatter}(f,p) : \mathbf{S}\langle X\rangle \to \textsc{List}\langle \mathbf{S}\langle X\rangle\rangle}$$

```
$\textsc{scatter}(f,p) : S_{\text{in}} \mapsto \mathcal{S}_{\text{out}}$
  let $\mathcal{S}_{\text{out}} = \textsc{Empty-List}\langle \mathbf{S}\langle X\rangle\rangle$
   loop:  $\mathbf{y} = f(\leftarrow S_{\text{in}})$
          $y_i \to \mathcal{S}_{\text{out}}[p(y_i)]$   for all $y_i \in \mathbf{y}$
```

Definition 3.1.22 (Merge) *The operator is the inverse of Scatter in the sense that it merges a collection of incoming streams back into a single outgoing stream. The operator reads from n incoming streams of type X into a buffer of size n (one slot for each incoming stream). A selection function is used to pick the element in the buffer to be written to the outgoing stream. The selection function $f : X \to (Y, \leq)$ has a partial order over Y that is used to determine the smallest element (w.r.t. $\leq$), remove it from the buffer, and write it to the output stream:*

$$\frac{f : X \to (Y, \leq)}{\textsc{merge}(f,\leq) : \textsc{List}\langle \mathbf{S}\langle X\rangle\rangle \to \mathbf{S}\langle X\rangle}$$

```
$\textsc{merge}(f) : \mathcal{S}_{\text{in}} \mapsto S_{\text{out}}$
  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
              $i^* = \text{argmin}_{\leq}\{f(B[i])\}$
              $\{B[i^*] \to S_{\text{out}}; B[i^*] = \textbf{nil}\}$
          end if
```

Definition 3.1.23 (List-Map) *This operator is a higher-order operator that has collections of streams as input and output. The operator is a generalization that allows us* *to apply a composition of first-order pipelines to the list of streams. Given a collection of streams* $\mathcal{S}_{\text{in}}$ *generated by Scatter, one can apply a pipeline of first-order operators on each stream* $\mathcal{S}_{\text{in}}[i]$*:*

$$\frac{h : \mathbf{S}\langle X\rangle \to \mathbf{S}\langle Y\rangle}{\textsc{list-map}(h) : \textsc{List}\langle \mathbf{S}\langle X\rangle\rangle \to \textsc{List}\langle \mathbf{S}\langle Y\rangle\rangle}$$

```
$\textsc{list-map}(h) : \mathcal{S}_{\text{in}} \mapsto \mathcal{S}_{\text{out}}$
    $\textbf{let } \mathcal{S}_{\text{out}} = \textsc{Empty-List}\langle \mathbf{S}\langle Y\rangle\rangle$
    for all $i \le \textbf{len}(\mathcal{S}_{\text{in}})$
     $\textbf{loop}:$  $y \leftarrow h(\mathcal{S}_{\text{in}}[i])$
                $y \to \mathcal{S}_{\text{out}}[i]$  .
```

The higher-order operators allow the algebraic description of large-scale stream pipelines processing collections of streams and having high degrees of concurrency. Figure 3.3 shows the Scatter, List-Map, and Merge parallel processing pattern naturally expressed in our algebra. This pattern is popularly utilized in several concurrent processing environments, such as distributed clusters, graphics processing units (GPUs), and multi-core processing. Later, in Section 3.3, we will discuss the ability of our algebra to enable automatic optimization of pipeline performance by automatically finding the best execution plan. Such a plan may involve automatic replacement of a simple first-order Map operator by a more optimal execution pattern, such as the one shown in Figure 3.3.

3.1.5 Examples

There is a continuous need for computer vision algorithms that can process vision streams in real time. In this section, we present several examples of algorithms that successfully applied data-streaming concepts in processing vision streams [145, 175]. Particularly and without loss of generality, we will show how our stream algebra can effectively describe the vision pipelines of these algorithms using a set of equations over data streams. These algebraic definitions are provided as examples to guide researchers through the process of describing their own vision pipelines. The algorithms are selected to address a diverse range of vision problems, such as activity recognition, analysis of photo streams, video segmentation, online dictionary learning, and active learning. The complexity of the algorithms varies from a simple one that defines one online operator to complex ones that have several concurrent operators.

Activity Recognition

In computer vision, the activity recognition task aims to recognize the behaviour and actions performed by single or multiple moving objects in video streams. Recently, Ryoo [145] developed a method for early recognition of human actions. These actions are hand shaking, hugging, kicking, pointing, punching, and pushing. Initially, the algorithm performs offline learning to build action models. Then, the models are later used by the algorithm to predict actions online.

To learn action models, for each target action, the algorithm receives a set of training videos. Each video is then processed to extract 3D spatio-temporal features. The extracted features from all videos are then clustered, and the centroids of the clusters are used to define k visual words. A bag-of-words model is then defined for each action as a histogram of occurrence counts of the visual words. For each training video i, the algorithm builds an integral histogram of visual words $H^i = (H^i_0, H^i_1, ..., H^i_j, ...)$ by extracting visual words and accumulating the words over time. Here, H^i_j $(k = |H^i_j|)$ is the histogram of visual words extracted from video i and accumulated up to frame j. An action model is then defined as the integral histogram resulting from averaging all the integral histograms of the action training videos. We define the set D as the set of all learned action models.

For online prediction of actions, an input video stream is processed by initially splitting the stream into a sequence of partitions or clips $C = \{C_q | q = 0, 1, ...\}$, where each has a duration $\triangle t$ frames. The bag-of-words model is then used to define a set of visual words for each clip C_q by extracting 3D spatio-temporal features. The visual words are then accumulated to build the integral histogram $H = (H_0, H_1, ..., H_q, ...)$, where H_q is the histogram of visual words accumulated up to clip C_q. The algorithm then matches the integral histogram H with every action model in D and builds the likelihood stream $L = (L_0, L_1, ..., L_q, ...)$, where L_q is a vector capturing the matching score of H_q against every action model in D. The predicted action stream $A = (A_0, A_1, ..., A_q, ...)$ is then computed for each clip C_q using $A_q = \arg\max_{0 \le i \le |L_q|} L_q^i$, where A_q is the index of the action with the maximum matching score in L_q.

To describe the online prediction step [145] using our algebra, the following data types are defined:

$\texttt{Frame} : \texttt{2DImage}; \quad \texttt{Feature} : \mathbb{R}^m; \quad \texttt{Video} : \mathbf{S}\langle\texttt{Frame}\rangle; \quad \texttt{Clip} : \textsc{List}\langle\texttt{Frame}\rangle,$

where a `Frame` is a 2D image, a `Feature` is a vector belonging to the m-dimensional space $\mathbb{R}^m$, a `Video` is a stream of images, and a `Clip` is a list of frames for a given time interval in a video stream of type `Video`. Using these data types, we can start describing the online prediction step [145] by first defining the following function:

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

```
g(u, x) =  {  if duration(u) ≥ △t then
                 $u' = \varnothing; y = u$
              else
                 $u' = u \oplus x$    //append $x$ to clip $u$
                 $y = \varnothing$
              return$(u', y)$  }.
```

This function takes a clip u' and a video frame x as input and outputs the two clips

u' and y. The function continues buffering the incoming video frames into the clip u'. When the duration of frames in u' is greater than or equal to a predefined interval $\triangle t$, the function first copies u' into y, then clears the content in u'. The Reduce operator can be used with the function g to partition the incoming video stream into a set of non-overlapping clips and create the following stream:

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

Here, arrows define the progression of time. The Filter operator can be used to remove the empty clips and produce the clip stream $C : \mathbf{S}\,\langle \texttt{Clip} \rangle$ using the following algebraic expression:

$$C \triangleq \textsc{Filter}(\lambda\, x : \ |x| \neq 0) \circ \textsc{Reduce}(g, \varnothing)(V), \tag{3.1}$$

$$\bullet_{C^0} \rightarrow \bullet_{C^1} \rightarrow \cdots \rightarrow \bullet_{C^q} \rightarrow \cdots$$

The operator $\circ$ forwards the output of the right operand to the input of the left operand. The 3D spatio-temporal features are then extracted from each clip in the stream C using the function $f_1 : \texttt{Clip} \rightarrow \textsc{List}\,\langle \texttt{Feature} \rangle$. The Map operator can be parametrized by f_1 to produce the feature stream $F : \mathbf{S}\,\langle \textsc{List}\,\langle \texttt{Feature} \rangle \rangle$:

$$F \triangleq \textsc{Map}(f_1)(C). \tag{3.2}$$

To transform the extracted features into visual words, [145] defined the function $f_2 : \textsc{List}\,\langle \texttt{Feature} \rangle \rightarrow \textsc{List}\,\langle \texttt{Word} \rangle$. This function leverages the learned bag-of-words model to produce a visual word for each feature vector in the incoming list of features. The f_2 then outputs a list of visual words. The Map operator parametrized with f_2 can be defined using the following algebraic expression to produce the visual-word stream $W : \mathbf{S}\,\langle \textsc{List}\,\langle \texttt{Word} \rangle \rangle$:

$$W \triangleq \textsc{Map}(f_2)(F). \tag{3.3}$$

An integral histogram can be then generated from the stream W using the following function:

$$g_2 : \text{Histogram} \times \textsc{List}\,\langle\texttt{Word}\rangle \rightarrow \text{Histogram} \times \text{Histogram}$$

$$\begin{aligned} g_2(u,x) = \quad \{ \quad & u' = u + x \\ & \text{return}(u', u') \\ \}, & \end{aligned}$$

where the operator + incrementally updates the histogram u using the incoming list of visual words x. The updated histogram u' is then returned as output. The Reduce operator parametrized by the function g_2 can then generate the stream H, which defines the integral histogram:

$$H \triangleq \textsc{Reduce}(g_2, \text{empty-histogram})(W), \tag{3.4}$$

$$\begin{tikzcd} \bullet_{H^0} \arrow[r] & \bullet_{H^1} \arrow[r] & \cdots \arrow[r] & \bullet_{H^q} \arrow[r] & \cdots \end{tikzcd}$$

Using the set D of learned action models, the function $d : \text{Histogram} \times \text{ActionModel} \rightarrow \mathbb{R}$ can be defined to match incoming histograms to learn action models in D and produce a matching score for every histogram in the stream H (see [145] for details). We can then use the function d to define the function $f_3 : \text{Histogram} \rightarrow \textsc{List}\,\langle\mathbb{R}\rangle$, where:

$$f_3(x) = \{\text{return } [d(x, D[i]) \text{ for } i \leq |D|]\}.$$

This function outputs a likelihood vector for each histogram in stream H. The vector contains the matching scores to all action models in D. The Map operator parametrized by the function f_3 can then be defined using the following expression to produce the likelihood stream $K : \mathbf{S}\left\langle \textsc{List}\left\langle \mathbb{R} \right\rangle \right\rangle$:

$$K \triangleq \textsc{Map}(f_3)(H), \tag{3.5}$$

$$\begin{tikzcd} \bullet_{K^0} \arrow[r] & \bullet_{K^1} \arrow[r] & \cdots \arrow[r] & \bullet_{K^q} \arrow[r] & \cdots \end{tikzcd}$$

Notice that the stream K has a likelihood vector for each clip in the stream C. The incoming likelihood vectors in K are then accumulated into one likelihood vector that defines the output predicted likelihood over all incoming clips seen so far. This is performed by [145] using a Bayesian combination function $f_4 : \textsc{List}\left\langle \mathbb{R} \right\rangle \times \textsc{List}\left\langle \mathbb{R} \right\rangle \to \textsc{List}\left\langle \mathbb{R} \right\rangle$. The f_4 function can be used to define the g_3 function as follows: $g_3 : \textsc{List}\left\langle \mathbb{R} \right\rangle \times \textsc{List}\left\langle \mathbb{R} \right\rangle \to \textsc{List}\left\langle \mathbb{R} \right\rangle \times \textsc{List}\left\langle \mathbb{R} \right\rangle$

$$\begin{aligned} g_3(u, x) = \quad \{ \quad & u' = f_4(u, x) \\ & \text{return}(u', u') \\ \}, & \end{aligned}$$

which performs the accumulation of incoming likelihood vectors x into the state vector u. The Reduce operator parametrized by the function g_3 can be used to produce the accumulated likelihood stream $L : \mathbf{S}\left\langle \textsc{List}\left\langle \mathbb{R} \right\rangle \right\rangle$:

$$L \triangleq \textsc{Reduce}(g_3, \mathbf{0})(K), \tag{3.6}$$

$$\begin{tikzcd} \bullet_{L^0} \arrow[r] & \bullet_{L^1} \arrow[r] & \cdots \arrow[r] & \bullet_{L^q} \arrow[r] & \cdots \end{tikzcd}$$

Notice that $\mathbf{0}$ is a vector of zeros. As a last step, [145] outputs the predicted activity that has the maximum likelihood score. This can be performed using the function $f_5 = \lambda x : \arg\max_{0 \le i \le |x|} x_i$. The Map operator can then be parametrized by the function f_5 to produce the output activity stream $A : \mathbf{S}\langle\mathbb{R}^{+}\rangle$:

$$A \triangleq \textsc{Map}(f_5)(L), \tag{3.7}$$

$$\begin{tikzcd} \bullet_{A^0} \arrow[r] & \bullet_{A^1} \arrow[r] & \cdots \arrow[r] & \bullet_{A^q} \arrow[r] & \cdots \end{tikzcd}$$

Hierarchical Video Segmentation

Hierarchical video segmentation aims to divide the space-time video volume into a hierarchy of 3D space-time segments with consistent appearance. The space-time video volume is a 3D volume (x, y, t) with (x, y) defining the 2D image space and t defining time. Thus, at t_0, we have frame 0 and, at t_n, we have frame n. The traditional method [69] for performing hierarchical video segmentation is to process the entire input video at once. This is usually performed by initially building an oversegmentation that creates a large set of 3D segments. Then, neighbouring segments are iteratively merged together to produce the output segmentation hierarchy. This traditional method, however, exhausts memory resources and constrains the size of input videos due to the need to process the entire video at once.

To solve this problem, Xuand et al. [175] proposed a recent streaming framework that approximates full video hierarchical segmentation and requires constant memory resources. This algorithm starts by dividing the input video into a sequence of non-overlapping clips of duration $\triangle t$. Each clip is represented as a 3D space-time volume and is segmented into a collection of 3D space-time segments. Then, hierarchical clustering is applied on these segments to generate a segmentation hierarchy. The method also uses a Markovian assumption to build the hierarchy of the current clip using the hierarchy generated for the previous clip, thus linking the sequence of segmentation hierarchies generated for the incoming stream of clips.

To describe the method by [175] using our algebra, we reuse the data types and functions defined in Section 3.1.5. Given the video stream $V \in \texttt{Video}$, a non-overlapping clip stream $C : \mathbf{S}\langle\texttt{Clip}\rangle$ can be generated using the Reduce operator parametrized by the function g (see Section 3.1.5). The method by [175] can be defined by the function f_6 : Hierarchy $\times$ `Clip` $\rightarrow$ Hierarchy. This function takes the current clip and hierarchy generated for the previous clip as input. The function then uses the previous hierarchy to construct and output a new segmentation hierarchy for the input clip. Using f_6, the following function can be defined:

$$g_4 : \textsc{List}\langle\mathbb{R}\rangle \times \textsc{List}\langle\mathbb{R}\rangle \rightarrow \textsc{List}\langle\mathbb{R}\rangle \times \textsc{List}\langle\mathbb{R}\rangle$$

$$\begin{aligned} g_4(u,x) = \quad \{ \quad & u' = f_6(u,x) \\ & \text{return}(u',u') \\ \}. & \end{aligned}$$

Notice that this function takes as input a clip x and a state variable u that stores the recent generated hierarchy. The function then uses f_6 to create a new hierarchy and assigns it to output and state variables. The Reduce operator parametrized by the g_4 function then produces the stream of segmentation hierarchies $H : \mathbf{S}\langle\text{Hierarchy}\rangle$:

$$H \triangleq \textsc{Reduce}(g_4, \text{empty-hierarchy})(C). \tag{3.8}$$

Several other computer vision algorithms [119, 105, 120] can be similarly described using our algebra.

3.2 Feedback Control for Streaming Computer Vision Pipelines

Feedback control is an essential task in several online computer vision algorithms [105, 108, 149, 47, 98] that perform parameter tuning or iterative optimization (see Section 2.1.1). It is obvious that our stream algebra can describe feedforward pipelines, which is referred to in literature as *open-loop systems* [126]. In this section, we show that our stream algebra can naturally express feedback control in computer vision pipelines.

3.2.1 Algebraic Description of Feedback Control

Section 2.1.1 provided a brief overview of the basic concepts of feedback-control systems. This section builds upon these concepts and defines a formal description of feedback control consistent with the definition of our stream algebra.

Let us assume a feedforward pipeline exists containing a sequence of Map and Reduce operators $\{X_1, ..., X_k\}$ as shown in Figure 3.1a. Each operator X_j has an input stream I_{j-1} and an output stream I_j. The output stream of each operator X_j is fed as the input stream of the next operator X_{j+1}. The streams I_0 and I_k define the pipeline input and output streams, respectively.

Each streaming operator can be treated as a multivariate process with the user-defined parameters of the mapping functions in either Map or Reduce operators representing the input-controlled variables. Figure 3.1b shows a single-loop feedback-control system for controlling the operators X_2 to X_j in the feedforward pipeline of Figure 3.1a. A single feedback loop is created using a return stream R, two operators E_1 and E_2, and feedback stream F. The return stream is a copy of the output stream obtained using the Copy operator:

$$I'_k, R_1 \triangleq \textsc{Copy}(2)(I_k). \tag{3.9}$$

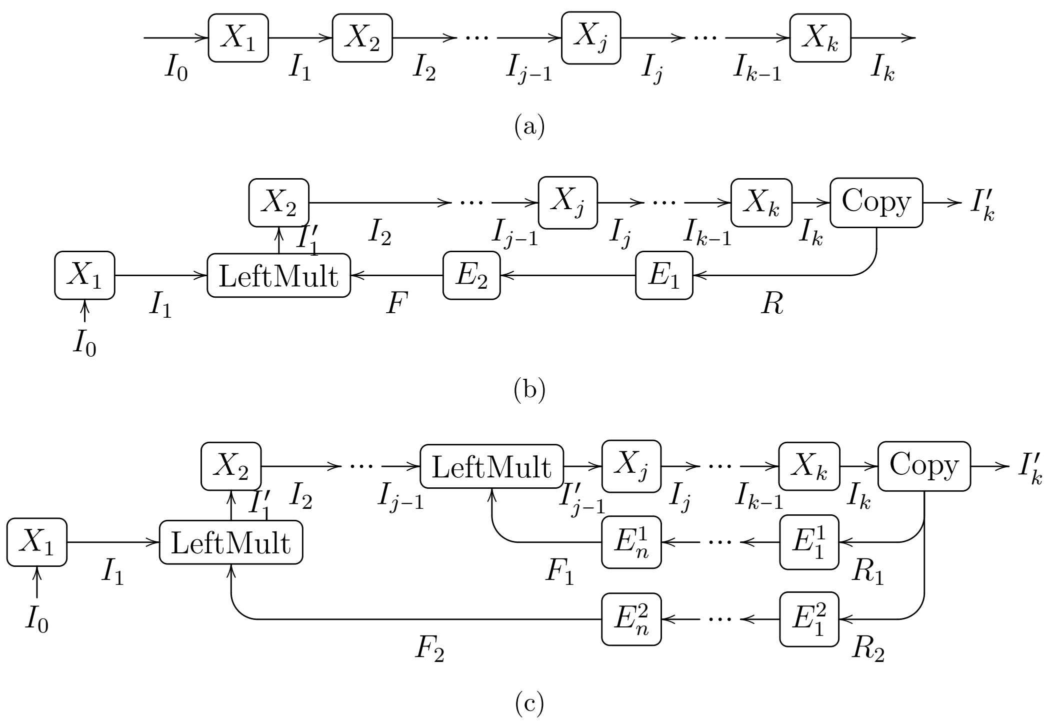


Figure 3.1: Example of a multi-loop feedback-control system: a) feedforward streaming pipeline; b) single-loop feedback-control system for controlling a set of operators in the streaming pipeline shown in (a); c) multi-loop feedback-control system for controlling a set of operators in the streaming pipeline shown in (a). Each operator X_i represents Map or Reduce and has the incoming stream I_j and the outgoing stream I_{j+1}. The two streams I_0 and I_k refer to the pipeline input and output streams, respectively. The multi-loop feedback-control system has two feedback loops. Each loop receives a copy R_t ($t \in \{1,2\}$) of the return stream R and has a sequence of controllers $\{E_t^1, ..., E_n^t\}$ with the output of the last controller defining the feedback stream F_t.

Figure 3.1c extends the single-loop feedback-control system in Figure 3.1b to a multi-loop feedback-control system with two feedback loops that are created and defined similarly. Each feedback loop has a return stream R_t for $t \in \{1,2\}$, a sequence of operators $E_1^t, ..., E_n^t$, and a feedback stream F_t. The return streams are copies of the output stream obtained using the Copy operator:

$$I'_k, R_1, R_2 \triangleq \textsc{Copy}(3)(I_k). \tag{3.10}$$

One can also apply Filter, Cut, or Latch to obtain the return streams. Both the return R_t and feedback F_t streams represent the feedback signals in traditional feedback-control systems. The operators $E_1^t, ..., E_n^t$ in each feedback loop process the return stream to generate a feedback stream F_t. Later, the feedback stream is merged with the input stream of the target-controlled operator using merging operators similar to the LeftMult operator in Figure 3.5c. These merging operators mimic the same functionality of controllers in traditional feedback-control systems and are defined for the two feedback loops of Figure 3.5c as follows:

$$I'_{j-1} \triangleq \textsc{LeftMult}()(I_{j-1}, F_2), \tag{3.11}$$

$$I'_1 \triangleq \textsc{LeftMult}()(I_1, F_1). \tag{3.12}$$

Here, I_{j-1} and I_1 are the input streams to the operators X_1 and X_j, respectively. This algebraic description shows that our stream algebra can naturally express feedback control. For the task of parameter tuning, the operators defined in the feedback loop can evaluate the current output and generate a feedback stream that optimizes the controlled operators. Equations 3.11 and 3.12 show LeftMult operators that merge feedback streams with the input streams to the controlled operators.

Notice that the feedback loops in Figure 3.5c have their data-flow rate synchronized with the flow rate of the feedforward pipeline. If the return streams are generated using the Cut operator instead of the Copy operator, the data-flow rate of the feedback loops will be asynchronous and independent of the feedforward pipeline. In this case, the operators defined in the feedback loops will process samples of the output stream. Furthermore, if an Add operator is used instead of the LeftMult operator in Equations 3.11 and 3.12, every feedback stream F_t will have its elements interleaved with the corresponding input stream of the controlled operator. This interleaving is useful in iterative optimization where the output is fed back as input for reprocessing.

Although Figure 3.5c shows a multi-loop feedback-control system with two feedback loops, more feedback loops can be defined by increasing the number of return streams generated by the Copy operator. For m return streams, we can have m feedback loops to control m operators.

3.2.2 Examples

There is considerable interest in developing online computer vision algorithms that use feedback control to perform parameter tuning or iterative optimization tasks. These tasks allow an online algorithm to adapt itself continuously to different scene contexts or iteratively improve output results over time. In this section, we discuss two state-of-the-art algorithms [47, 105] that process vision streams and apply feedback control to perform parameter tuning and iterative optimization. Without loss of generality, we will discuss how we can effectively express the feedback control of these algorithms using our stream algebra.

Online Adaptation of Tracking Parameters

Object tracking is an important problem in computer vision. It aims to capture the trajectories taken by single or multiple moving objects over time. The main challenges for object tracking algorithms are scene lighting and environmental changes, which can result in inaccurate tracking results. To solve this problem, Chau et al. [47] developed an algorithm that applies online parameter tuning to continuously optimize the performance of a given tracking algorithm to unexpected scene changes. The method operates in two stages: an offline learning stage and online control stage.

To perform online learning, [47] started by taking the following inputs: 1) a collection of training videos belonging to different scene contexts, 2) a set of annotated areas for each video indicating moving objects, 3) a set of annotated trajectories, and 4) a tracking algorithm with a predefined set of control parameters. To track humans, [47] used the appearance-based tracking algorithm by [48], which has six input parameters. The method by [47] then extracts context features from each frame of every input training video. Each feature is a vector of six elements representing the characteristics of human objects, which include their density, occlusion, and appearance. Later, the similarities between context feature vectors are used to partition each video into a sequence of clips. For each clip, the tracking algorithm is applied, and parameter optimization is used to learn its best parameter setting. Then, for each scene context, the features of all its training videos are clustered, and the best parameter settings are selected for each cluster. Finally, a database D is used to save all context clusters and their selected parameters.

In the online control stage, [47] processed an incoming video stream $V = \{V_i | i = 0, 1, 2, ...\}$. The method uses the histogram of oriented gradients (HOG)-based detector by [53] to locate human objects in every frame $V_i \in V$. The output is a stream $B = \{B_i | i = 0, 1, 2, ...\}$, where B_i is the list of detected objects in V_i. Later, a temporal window of interval $\triangle t_1$ is defined to record all detected objects in the sequential frames belonging to the window. Then, for every frame V_i, the method defines the pair (V_i, A_i), where A_i is the list of detected objects in the temporal window ending with frame V_i. This generates the stream $U = \{U_i | i = 0, 1, 2, ...\}$, with $U_i = (V_i, A_i)$. A feedback stream $F = \{F_i | i = 0, 1, 2, ...\}$ is also defined, with each element F_i representing the best parameter setting found so far. The stream F is multiplied by the stream U to produce the tracker input stream $Q = \{Q_i | i = 0, 1, 2, ...\}$, where $Q_i = (V_i, A_i, F_i)$. The tracking algorithm receives the stream Q and performs the following actions: 1) sets the tracking control parameters to $F_i \in Q_i$, 2) assigns trajectories T_i for moving objects in frame V_i, and 3) outputs the stream $J = \{J_i | i = 0, 1, 2, ...\}$, where $J_i = (V_i, A_i, T_i)$. Elements in the stream J are then recorded in a temporal window of interval $\triangle t_2$. For each frame V_i, a clip C_i is defined using the temporal window ending with frame V_i. Every clip stream C_i is then attached to the corresponding element $J_i \in J$ to produce the final output stream $H = \{H_i | i = 0, 1, 2, ...\}$, where $H_i = (V_i, B_i, T_i, C_i)$. The stream H is then copied to stream R, which defines the input to the feedback loop. A parameter-tuning algorithm takes the stream R as input and produces the stream of parameter settings F. The tuning algorithm keeps track of the current best parameter setting and performs the following steps. 1) It uses $(V_i, B_i, T_i) \subset H_i$ to compute two scores: an object interaction score s_1 and a tracking error score s_2. 2) Next, s_1 and s_2 are compared against the predefined thresholds th_1 and th_2, respectively. 3) Then, if $s_1 > th_1$ and $s_2 > th_2$, an error state is defined, the closest context for $C_i \in H_i$ in database D is selected, and its parameter setting is used as the current best parameter setting. 4) Finally, if $s_1 \leq th_1$ and $s_2 \leq th_2$, then the algorithm continues using the previously found best parameter setting. The tuning algorithm writes the best parameter setting to stream F, which is multiplied by stream U to produce the tracker input stream Q. This completes the feedback-control loop defined by [47] for parameter tuning.

To describe the parameter-tuning algorithm of [47] in the proposed stream algebra, we use the feedback-control definitions in Section 3.2 and define the following data types:

$\texttt{Frame} : \texttt{2DImage}; \quad \texttt{Video} : \mathbf{S}\langle\texttt{Frame}\rangle; \quad \texttt{Clip} : \textsc{List}\langle\texttt{Frame}\rangle$

$\text{Histogram} : \textsc{List}\langle\mathbb{R}\rangle; \quad \texttt{Object} : \mathbb{R}^8 \times \text{Histogram}; \quad \texttt{Params} : \mathbb{R}^6$

$\texttt{FrameInfo} : \texttt{Frame} \times \textsc{List}\langle\texttt{Object}\rangle; \quad \texttt{Trajectory} : \textsc{List}\langle\mathbb{R}^2\rangle;$

$\texttt{TrackInput} : \texttt{Frame} \times \textsc{List}\langle\texttt{Object}\rangle \times \texttt{Params}$

$\texttt{TrackInfo} : \texttt{Frame} \times \textsc{List}\langle\texttt{Object}\rangle \times \textsc{List}\langle\texttt{Trajectory}\rangle$

$\texttt{LoopBack} : \texttt{Frame} \times \textsc{List}\langle\texttt{Object}\rangle \times \textsc{List}\langle\texttt{Trajectory}\rangle \times \texttt{Clip},$

where `Video` defines a video stream, `Frame` is a 2D image, `Clip` is a sequence of frames, Histogram is a vector of integer numbers, and `Object` defines a moving object formed by the pair (a, b), where $a : \mathbb{R}^8$ is an object feature vector and b : Histogram. The feature vector defines the following features: 1) 2D shape ratio, 2) 2D area, 3) colour covariance (RGB), and 4) dominant colour (RGB). In addition, `Params` is a 6D vector defining the tracking algorithm parameters [48], and `FrameInfo` is a 2D vector (v, w), where $v : \texttt{Frame}$ and $w : \textsc{List}\langle\texttt{Object}\rangle$. Moreover, `TrackInput` is a 3D vector (v, w, p), where $p : \texttt{Params}$. The `Trajectory` is a path taken by a moving object described as a list of 2D points, and `TrackInfo` is a 3D vector (v, w, e), where $e : \textsc{List}\langle\texttt{Trajectory}\rangle$. Furthermore, `LoopBack` is 4D vector (v, w, e, c), where $c : \texttt{Clip}$. The method by [47] starts by receiving an input video stream $V \in \texttt{Video}$. The Copy operator then duplicates the V stream using the following algebraic expression:

$$V_1, V_2 \triangleq \textsc{Copy}()(V). \tag{3.13}$$

Given the V_1 stream, moving objects are detected in every incoming frame $V_i \in V_1$. This was performed by [47] using the function $f_1 : \texttt{Frame} \rightarrow \textsc{List}\left<\texttt{Object}\right>$ that implements the HOG-based object detection algorithm by [53]. The Map operator parametrized by this function can be used to produce the object stream $B : \mathbf{S}\left<\textsc{List}\left<\texttt{Object}\right>\right>$:

$$B \triangleq \textsc{Map}(f_1)(V_1). \tag{3.14}$$

The following function is also defined to maintain the detected objects within a temporal window of interval $\triangle t_1$:

$$g_1 : \textsc{List}\left<\texttt{Object}\right> \times \textsc{List}\left<\texttt{Object}\right> \rightarrow \textsc{List}\left<\texttt{Object}\right> \times \textsc{List}\left<\texttt{Object}\right>,$$

```
$g_1(u, x) =$ {  for all  $z \in u$
                    if ($\text{now}() - \text{arrivaltime}(z) \geq \triangle t_1$) then
                        $u = u \ominus z$   //remove $z$ from $u$
                 $u = u \oplus x$   //append $x$ to $u$
                 return$(u, u)$  }
```

. The function keeps appending new detected objects x to list u, while deleting older objects. The function then returns the updated list of objects. The Reduce operator parametrized by the function g_1 can be used to construct the object summary stream

$M : S\langle \textsc{List}\langle \texttt{Object}\rangle\rangle$:

$$M \triangleq \textsc{Reduce}(g_1, \text{Empty-List})(B). \tag{3.15}$$

Now, the Mult operator can synchronize the two streams M and V_2. Remember that the V_2 stream is a copy of the input stream generated by Equation 3.13. The Mult operator can then generate the stream $U : S\langle \texttt{FrameInfo}\rangle$:

$$U \triangleq \textsc{Mult}()(V_2, M). \tag{3.16}$$

Given a feedback stream $F : S\langle \texttt{Params}\rangle$ generated by the feedback-control loop by [48], the LeftMult operator is defined to merge the stream F with the stream U. This generates the output stream $Q : S\langle \texttt{TrackInput}\rangle$ using the algebraic equation:

$$Q \triangleq \textsc{LeftMult}()(U, F). \tag{3.17}$$

Let $f_2 : \texttt{TrackInput} \to \texttt{TrackInfo}$ be a function defining the tracking algorithm. The function receives the vector $(v, w, p) : \texttt{TrackInput}$ as input and processes both v and w using the parameter vector p. Then, the function constructs a list of trajectories and appends it to the vector (v, w) to generate the output $y : \texttt{TrackInfo}$. The Map operator parametrized by the function f_2 can be defined to produce the trajectory stream $J : S\langle \texttt{TrackInfo}\rangle$ using the equation:

$$J \triangleq \textsc{Map}(f_2)(Q). \tag{3.18}$$

To describe the feedback-control loop by [47], the Copy operator can be used to copy the output stream J into two streams: the final output stream J' and the return stream R:

$$R, J' \triangleq \textsc{Copy}(2)(J). \tag{3.19}$$

We can now define the following function:

$$g_2 : \texttt{Clip} \times \texttt{TrackInfo} \to \texttt{Clip} \times \texttt{LoopBack},$$

```
$g_2(u,x) =$ { for all $z \in u$
                  if (now() − arrivaltime($z$) ≥ $\triangle t_2$) then
                      $u = u \ominus z$   //remove $z$ from $u$
               $u = u \oplus x.v$   //append frame $x.v$ to $u$
               $y = x \oplus u$   //append clip $u$ to $x$
               return($u, y$) }.
```

This function keeps track of a temporal window u, which is a clip over the video frames of the incoming $x : \texttt{TrackInfo}$ vectors in a time interval $\triangle t_2$. It then appends the clip u to the input x to generate the output $y : \texttt{LoopBack}$. The Reduce operator parametrized with the function g_2 can be then defined to process the stream R and produce the loopback stream $H : S\langle\texttt{LoopBack}\rangle$:

$$H \triangleq \text{Reduce}(g_2, \text{Empty-List})(R). \tag{3.20}$$

To calculate the quality of the tracking algorithm, [47] defined the two error functions: $f_3 : \texttt{FrameInfo} \to \mathbb{R}$ and $f_4 : \texttt{TrackInfo} \to \mathbb{R}$. The f_3 and f_4 functions define the object interaction score and tracking error score, respectively. These functions can be used to define the following parameter-tuning function:

$$g_3 : \texttt{Params} \times \texttt{LoopBack} \to \texttt{Params} \times \texttt{Params},$$

```
$g_3(u, x) =$ {  $s_1 = f_3(x.v, x.w)$
                 $s_2 = f_4(x.v, x.w, x.e)$
                 if $(s_1 > th_1$ and $s_2 > th_2)$ then
                     $p =$ search-db$(x.c, D)$
                     return$(p, p)$
                 return$(u, u)$  }.
```

This function keeps track of the best parameter setting and assigns it to the state variable u. Initially, the g_3 function calculates the error scores for the incoming vector $x : \texttt{LoopBack}$. If the scores are larger than the predefined thresholds, then a new parameter setting is found by searching the database D for the best parameter setting matching the context of the current clip $x.c$. The new parameter setting is then written to the output. If the error scores are below the thresholds, then the incoming parameter setting is passed to the output. The Reduce operator parametrized with the function g_3 can be defined to produce the feedback stream $F : S\langle\texttt{Params}\rangle$:

$$F \triangleq \text{Reduce}(g_3, \text{Initial-Params})(H). \tag{3.21}$$

Notice that the stream F was merged with the stream U in Equation 3.17 using the LeftMult operator to produce the tracker input stream Q. This competes the description of the parameter-tuning method by [47] using the single-loop feedback-control definitions of the proposed stream algebra.

Iterative Optimization for Aligning Photo Streams

Nowadays, several photo hosting websites store vast amounts of personal image and video collections. These visual collections are usually received and stored as photo streams organized in chronological order. Kim et al. [105] developed a method that processes

Flickr photo streams to build common storylines. The method starts by grouping photo streams from different Flickr users, such that each group has a common user activity. The method then processes each group individually, where a group has n photo streams $I = \{I_k | k = 1, 2, 3, n\}$. The method attaches the following attributes to every photo: 1) a visual descriptor defined as a spatial pyramid histogram, 2) the capture time, and 3) a set of the foreground regions, which is initially set to an empty list. A photo stream I_k is partitioned into a sequence of photo blocks $B_k = \{B_{\{k,i\}} | i = 0, 1, 2, \}$, where each block $B_{\{k,i\}}$ contains photos taken during an interval $\triangle t_1$ (e.g., a day). A block also records the earliest and latest capture times of its photos. The method works by iterating between the following tasks: an alignment task and a co-segmentation task.

For the alignment task, the method takes as input a set of n photo streams. A list of n blocks is then formed by reading one block from each stream. This produces the block-list stream $L = \{L_i | i = 0, 1, 2, \}$, where $L_i \in L$ is a list of n blocks. Each incoming list of blocks L_i is then filtered to select a subset of blocks $E_i \subset L_i$, such that all blocks in E_i overlap in time by a period of at least $\triangle t_2$ (e.g., an hour). A vector (E_i, N) is then defined, where N is a variable defining the number of algorithm iterations and is initially set to zero. This produces the overlapped blocks stream $Q = \{Q_i | i = 0, 1, 2, ...\}$, where $Q_i = (E_i, N_i)$. A feedback stream F is then defined with the same data type as the Q stream. The streams Q and F are then added together to produce the stream $P = \{P_j | j = 0, 1, 2, ...\}$, where $P_j = (E_j, N_j)$. Notice that elements in the P stream are formed by interleaving the elements of the streams Q and F. Given a list of blocks $E_j \in P_j$, the technique computed the similarity of every block $b \in E_j$ to all other blocks in E_j. The similarity is computed using two distance functions, $f_4 : \text{Photo} \times \text{Photo} \rightarrow \mathbb{R}$ and $f_5 : \text{Time} \times \text{Time} \rightarrow \mathbb{R}$. Given two photos $x \in b_1$ and $y \in b_2$ that belong to two blocks b_1 and b_2, the f_4 function computes the distance between x and y. First, f_4 tests whether the input images x and y have foreground regions assigned. If so, f_4 compares the histograms of these regions. If foreground regions do not exist, f_4 compares the spatial pyramid histograms of the input photos. The f_5 function computes the difference of the capture times between x and y. The method then defines the energy function $f_6 : \text{Block} \times \text{Block} \to \mathbb{R}$ that computes the similarity between two blocks b_1 and b_2 by summing the nearest-neighbour distances computed using f_4 and f_5 between photos in b_1 and b_2. Every list of blocks $E_j \in P_j$ is then mapped to a graph $G_j : \text{Graph}\langle\text{Block}, \text{Block} \times \text{Block}\rangle$ that has blocks in E_j as vertices. Two blocks have an edge in G_j if they have the smallest distance from each other. The alignment step then produces the graph of blocks stream $Z = \{Z_j | j = 0, 1, 2, ...\}$, where $Z_j = (G_j, N_j)$.

The co-segmentation stage processes the stream Z by mapping every graph $G_j \in Z_j$ into a corresponding graph $Y_j : \text{Graph}\langle\text{Photo}, \text{Photo} \times \text{Photo}\rangle$. The vertices of the graph Y_j are the entire set of photos in all blocks in G_j, and two photos x and y have an edge if they are a corresponding pair that belong to different blocks. In addition, edges are added between every photo x and the k-nearest neighbours in its block. The photos-graph stream $M = \{M_j | j = 0, 1, 2, ...\}$ is then produced, where $M_j = (G_j, N_j, Y_j)$, and N_j is also incremented by 1. Later, the co-segmentation technique by [104] is applied to assign a set of m foreground regions for each photo in $Y_j \in M_j$. Photos with already-defined foreground regions also have their regions refined.

The output of the co-segmentation task is then fed back to the input of the alignment task and the method by [105] keeps iterating between the two tasks. This is performed using a feedback loop that splits the photos-graph stream M into two streams: the final output stream M' and the return stream R. The splitting is based on the iteration number variable $N_j \in M_j$. If $N_j < N_{stop}$, then M_j is forwarded to the return stream; otherwise, M_j is sent to the final output stream M'. Notice that N_{stop} is the maximum number of iterations for each element $M_j \in M$. The algorithm then maps each element of the return stream R into the vector $F_l = (E_l, N_l)$ to produce the feedback stream $F = \{F_l | l = 0, 1, 2, ...\}$. Here, E_l is a list of all blocks defining the vertices in graph $G_l \in R_l$, and N_l is the iteration number. As stated before, the streams F and Q are added together to produce the interleaved stream P used in the alignment step. Notice that the blocks in F have photos with defined foreground regions. These regions are used to obtain more accurate matching between photos in the alignment step. Having better matching also improves the output of the co-segmentation step. This completes the feedback loop that implements iterative optimization. To describe the iterative optimization algorithm using our stream algebra, the following data types are defined:

$$\texttt{Shape} : \textsc{List}\langle\mathbb{R}^2\rangle\,; \quad \texttt{Region} : \texttt{Shape} \times \text{Histogram}$$

$$\texttt{Photo} : \texttt{2DImage} \times \texttt{Time} \times \text{Histogram} \times \textsc{List}\langle\texttt{Region}\rangle \times \mathbb{R}$$

$$\texttt{Block} : \textsc{List}\langle\texttt{Photo}\rangle \times \texttt{Time}^2; \quad \texttt{BlocksStruct} : \textsc{List}\langle\texttt{Block}\rangle \times \mathbb{R};$$

$$\texttt{BlocksGraphStruct} : \textsc{Graph}\langle\texttt{Block}, \texttt{Block} \times \texttt{Block}\rangle \times \mathbb{R}$$

$$\texttt{PhotosGraphStruct} : \texttt{BlocksGraphStruct} \times \textsc{Graph}\langle\texttt{Photo}, \texttt{Photo} \times \texttt{Photo}\rangle$$

A type `Shape` defines a 2D point list. A `Region` is a vector (s, h), where $s : \texttt{Shape}$ is the region boundary and $h :$ Histogram is the feature descriptor of the region. A `Photo` is a vector (a, t, h, r, nn), where $a : \texttt{2DImage}$ is a 2D image, $t : Time$ is the capture time of the photo, $h :$ Histogram is a feature descriptor, $r : \textsc{List}\langle\texttt{Region}\rangle$ is a list of computed foreground regions, and nn is a pointer to the nearest-neighbour photo. A `Block` is a vector (b, t_1, t_2), where $b : \textsc{List}\langle\texttt{Photo}\rangle$. In addition, $t_1 : Time$ and $t_2 : Time$ are the earliest and latest capture times for all photos in b. A `BlocksStruct` is a vector (w, itr), where $w : \textsc{List}\langle\texttt{Block}\rangle$, and $itr : \mathbb{R}$ is a variable recording the iteration number. A `BlocksGraphStruct` is a vector (c_1, itr), where c_1 is a graph with blocks as vertices. A `PhotosGraphStruct` is a vector (q, c_2), where $q : \texttt{BlocksGraphStruct}$, and c_2 is a graph with photos as vertices. To simplify the discussion, we assume three input photo streams $I = \{I_k | k = 1, ..., 3\}$. To start processing the input streams, we define the function g_4:

$$g_4 : \texttt{Block} \times \texttt{Photo} \rightarrow \texttt{Block} \times \texttt{Block}$$

```
$g_4(u,x) =$ {  if duration$(u) \geq \triangle t_1$ then
                    $u' = \varnothing; y = u$
                else
                    $u' = u \oplus x$    //append $x$ to Block $u$
                    $y = \varnothing$
                return$(u', y)$  }.
```

The g_4 function partitions an incoming stream of photos into a set of blocks. The function buffers the incoming photos into a state variable u that defines a block. While buffering, the output y is set to an empty block. The buffering continues until the block u has its duration larger than a predefined interval $\triangle t_1$ (24 hours as defined by [105]). At this time, the output y is set to the block u, then u is reset back to an empty list. The Reduce operator parametrized by the function g_4 and followed by the Filter operator can be used to map every stream $I_k \in I$ into stream $B_k : S\langle\texttt{Block}\rangle$:

$$B_k, E_k \triangleq \textsc{Filter}(\lambda\, x:\ |x| \neq 0) \circ \textsc{Reduce}(g_4, \text{Empty-List})(I_k). \tag{3.22}$$

$$\textsc{Ground}()(E_k). \tag{3.23}$$

Remember that the $\circ$ operator forwards the output of the right operand to the input of the left operand. The Filter operator discards the empty blocks produced by the Reduce operator. These empty blocks are forwarded to the E_k stream and discarded using the Ground operator. The Mult operator can then synchronize the streams B_1, B_2 and B_3 to produce the block-list stream $L : S\langle\textsc{List}\langle\texttt{Block}\rangle\rangle$:

$$L \triangleq \textsc{Mult}()(B_1, B_2, B_3). \tag{3.24}$$

Next, we define the function $f_7 : \textsc{List}\langle\texttt{Block}\rangle \rightarrow \texttt{BlocksStruct}$ that filters every list

$L_i \in L$ to select blocks that overlap in time by a period of at least $\triangle t_2$ (an hour as defined by [105]). The function also sets the iteration number in the returned vector to zero. The Map operator parametrized by the f_7 function can be used to produce the stream $Q : S\langle \texttt{BlocksStruct} \rangle$:

$$Q \triangleq \textsc{Map}(f_7)(L). \tag{3.25}$$

Given the feedback stream $F : S\langle \texttt{BlocksStruct} \rangle$, the Add operator can merge the two streams Q and F together to produce the stream $P : S\langle \textsc{List}\langle \texttt{BlocksStruct} \rangle\rangle$:

$$P \triangleq \textsc{Add}()(Q, F). \tag{3.26}$$

To process the P stream, we define the functions $f_8 : \texttt{BlocksStruct} \rightarrow \texttt{BlocksGraphStruct}$ and $f_9 : \texttt{BlocksGraphStruct} \rightarrow \texttt{PhotosGraphStruct}$, where f_8 converts the list of blocks in each element $P_j \in P$ into a graph G_j : Graph$\langle$Block, Block $\times$ Block$\rangle$. As discussed before, an edge exists between two blocks in G_j if and only if the two blocks have the smallest distance between each other. The function f_9 converts a graph of blocks to a graph of photos and increments the iteration counter variable by 1. It also applies co-segmentation on the graph of photos to assign each photo a set of foreground regions or improve an existing one. We can define two Map operators parametrized by the functions f_8 and f_9 to produce the graph of photos stream $M : S\langle \texttt{PhotosGraphStruct} \rangle$:

$$M \triangleq \textsc{Map}(f_9) \circ \textsc{Map}(f_8)(P). \tag{3.27}$$

To describe the feedback loop in our stream algebra, we apply the Filter operator on the stream M to produce the output stream M' and the return stream R:

$$M', R \triangleq \textsc{Filter}(\lambda\, x :\ x.q.itr \geq N_{stop})(M). \tag{3.28}$$

Notice that the Filter operator applies a predicate on the q : $\texttt{BlocksGraphStruct}$ of every vector $M_j \in M$. The predicate performs the test $q.itr \geq N_{stop}$. If the test is valid, the vector M_j is sent to the stream M', otherwise it is sent to the R stream. A function f_{10} : `PhotosGraphStruct` → `BlocksStruct` can then be defined to convert the data type of elements in R to `BlocksStruct`. This is by copying the iteration number and generating a list of blocks from the vertices of the blocksgraph in every element of the stream R. The Map operator parametrized by the function f_{10} can be used to produce the feedback stream $F : S\langle\texttt{BlocksStruct}\rangle$:

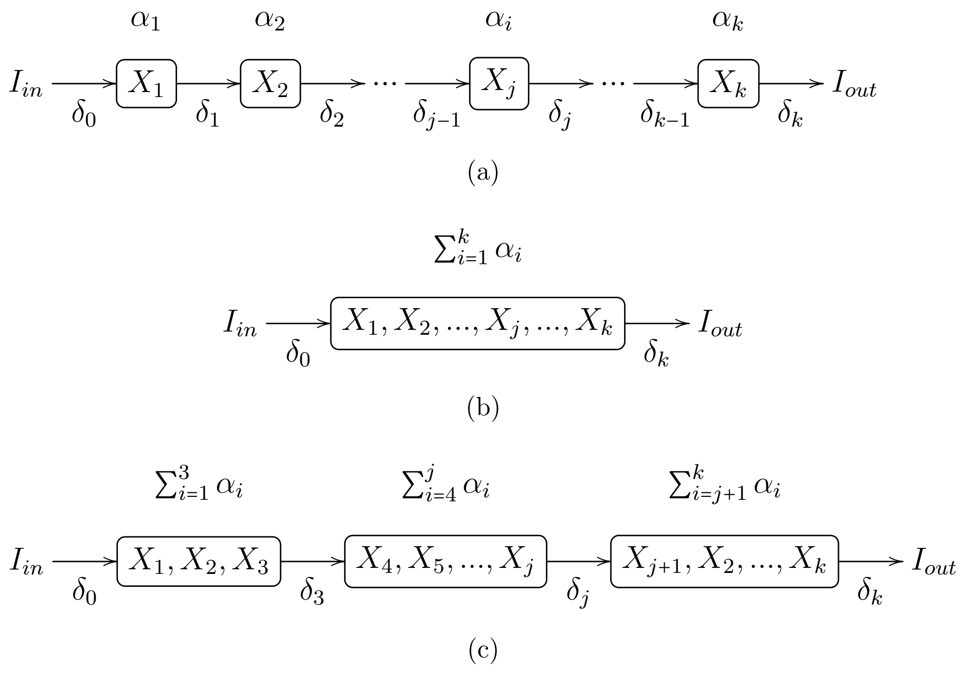
$$F \triangleq \textsc{Map}(f_{10})(M_2). \tag{3.29}$$

Notice that the stream F is merged with the stream Q using the Add operator of Equation 3.26 to produce the stream P. This completes the description of the single-loop feedback loop defined by [105]) for iterative optimization.

3.3 Discussion

The stream algebra provides operators for data processing and rate control. The data-processing operators implement data transformations on the input vision streams. The rate-control operators manipulate the data rates and flow of data. Examples of these operators include Cut, Latch, and LeftMult. Rate-control operators can resolve blocking and slow operations by synchronizing and matching between the different data-flow rates of vision streams, thus supporting real-time streaming. For example, the Latch and Cut operators decouple the data-flow rates of the input and output streams. Latch maintains the last received element and outputs it according to the downstream data rate. The LeftMult operator also matches the data-flow rates of two input streams. In the algebraic description of the online tracking example (see Section 3.2.2), Equation 3.17 shows how LeftMult can latch on the feedback stream $F : S\langle\texttt{Params}\rangle$, multiply it by the stream $U : S\langle\texttt{FrameInfo}\rangle$ to produce the output stream $Q : S\langle\texttt{TrackInput}\rangle$. If streams F and U

have different data-flow rates, LeftMult guarantees that each incoming element in U has the most recent parameter setting attached to the corresponding element in the output stream Q. If we replace the Copy operator of Equation 3.19 with Cut, the feedback loop of the online tracking example will have its data-flow rate decoupled from the feedforward pipeline. In large-scale systems, the rate-control operators allow synchronization between the data-flow rates of different computer vision tasks in vision stream-processing pipelines. This provides the important advantage of handling unbounded data rates of continuous (and possibly infinite) vision streams.



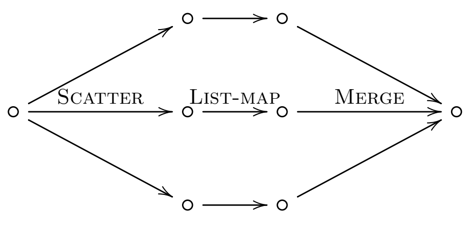
α_1 α_2 α_i α_k
I_{in} X_1 X_2 $\cdots$ X_j $\cdots$ X_k I_{out}
δ_0 δ_1 δ_2 δ_{j-1} δ_j δ_{k-1} δ_k
(a)
$\sum_{i=1}^{k} \alpha_i$
I_{in} $X_1, X_2, ..., X_j, ..., X_k$ I_{out}
δ_0 δ_k
(b)
$\sum_{i=1}^{3} \alpha_i$ $\sum_{i=4}^{j} \alpha_i$ $\sum_{i=j+1}^{k} \alpha_i$
I_{in} X_1, X_2, X_3 $X_4, X_5, ..., X_j$ $X_{j+1}, X_2, ..., X_k$ I_{out}
δ_0 δ_3 δ_j δ_k
(c)

Figure 3.2: An example of a pipelined versus a non-pipelined (sequential) implementation: a) feedforward streaming pipeline defined by the set of data processing operators $X = \{X_1, .., X_k\}$; b) a non-pipelined implementation of (a); (c) the linear pipeline in (a) partitioned into three intervals. For $i \in \{1, .., k\}$, α_i defines the computation time of operator X_i. For $j \in \{1, .., k-1\}$, δ_j defines the communication time for transferring the output of operator X_j to operator X_{j+1}. The communication time for receiving the input and producing the output is defined by δ_0 and δ_k, respectively. Each interval in (c) has its operators' core functions merged together using function composition.

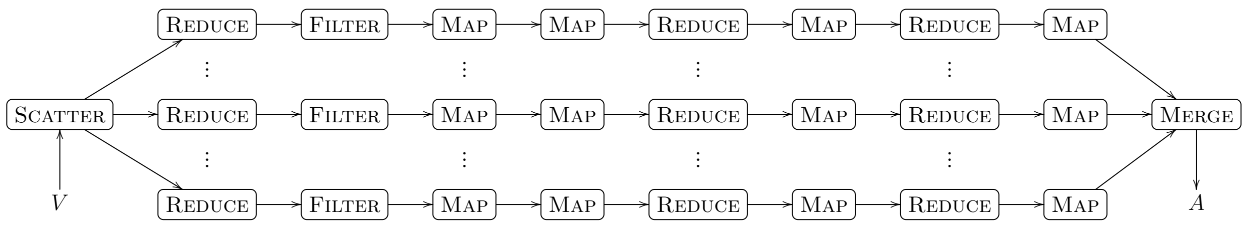
In previous sections, we presented several examples that show the ability of our stream algebra to naturally describe several computer vision algorithms. We expressed the vision pipeline of each example using formal algebraic equations that manipulate vision streams. For each algorithm, we decompose it into a set of stages. Each stage defines a core function wrapped by one of our common data processing operators, that include Map, Reduce and Filter. Then, the flow control operators handle and manipulate the data flow between core functions. So, our algebra controls and hides both concurrency and data flow between core algorithmic functions. The algebra provides an abstract representation that highlights the semantics of the different algorithmic components of computer vision algorithms and simplifies the process of building scalable computer vision systems. For example, we were able to decompose the activity recognition algorithm by [145] (see Section 3.1.5) into a set of core functions, which are described by the Map and Reduce operators of our algebra. Notice that these core functions are usually standard computer vision operations such as extraction of visual words and construction of integral histograms. The algebraic expressions also highlight how the core function is integrated with other functions to construct the required algorithms. Moreover, algebraic expressions show the intermediate transformations on the streams required to integrate core functions. For example, Equation 3.1 shows the buffering of incoming video frames to form clips and the filtering of empty clips. Equation 3.15 shows the use of function g_1 that maintains the detected objects within a temporal window. We can also scale up the pipeline of the activity recognition algorithm to process multiple videos as shown in Figure 3.3b. Here, we apply a parallel processing pattern formed using a Scatter, ListMap, and Merge operators to execute multiple pipelines in parallel. Scatter receives the stream $V : \mathbf{S}\langle\texttt{Frame} \times \mathbb{R}\rangle$, where each element $(\texttt{f}, \texttt{vid}) \in V$ contains a video frame $\texttt{f}$ and a video identifier $\texttt{vid}$. Scatter makes sure that frames belonging to the same video are forwarded to and processed by the same pipeline. Merge then combines the output predictions of the parallel pipelines to produce the output activity stream $A : \mathbf{S}\langle\mathbb{R}^+\rangle$.

Figure 3.2 shows an example of a pipelined versus a non-pipelined implementation of algorithms. Figure 3.2a shows a feedforward pipeline $X = \{X_1, ..., X_k\}$, that can represent one of the feedforward pipelines that describe computer vision algorithms in Section 3.1.5. Each operator X_i wraps a core function that has a computation time α_i, where $i \in \{1, .., k\}$. Moreover, the communication time for transferring the output of operator X_j to the next operator X_{j+1} is defined by δ_j, where $j \in \{1, .., k-1\}$. The communication time for receiving the input and producing the output is defined by δ_0 and δ_k, respectively. The efficiency of a pipelined versus a non-pipelined (sequential) implementation of a given algorithm is described in terms of throughput and latency. For the pipeline X, the throughput is $T_p = 1/P_p$, where $P_p = \max\{\alpha_1, ..\alpha_k, \delta_0, ..., \delta_k\}$ is the period defined as the largest communication or computation time in the pipeline. The throughput of a non-pipelined implementation is $T_n = 1/P_n$, where $P_n = \max\{\sum_{i=1}^{k} \alpha_i, \delta_0, \delta_k\}$. A pipeline has a better throughput as long as $T_p > T_n$. This can be achieved when $\max\{\delta_0, ..., \delta_k\} < \sum_{i=1}^{k} \alpha_i$, in other words, the largest communication time in a pipeline is less than the total computation time of all algorithm stages. Notice that δ_0 and δ_k are usually very small and negligible compared to other computation and communication costs. The latency of a pipelined implementation is $L_p = \delta_0 + \sum_{i=1}^{n}\{\alpha_i + \delta_i\}$, which defines the total time between the entry and exit times for a given tuple. For a non-pipelined implementation, the latency is $L_n = \delta_0 + \delta_k + \sum_{i=1}^{n} \alpha_i$. It is clear that $L_n < L_p$, so the penalty we have for a pipelined implementation is the additional communication time required for transferring data between the pipeline stages or operators.

Several optimizations can also be allowed on computer vision pipelines to tune performance and improve the allocation of computational resources. For example, the Map(f_1) operator followed by another Map (f_2) operator can be replaced by a single Map ($f_2(f_1)$) using function composition. A Reduce operator followed by the Map operator can be replaced by a single Reduce operator that applies the mapping function to its output. These optimizations can minimize the latency of a pipeline by eliminating large communication costs. In this case, we can trade off and optimize throughput versus latency using load balancing methods [25, 26, 24]. These methods partition a linear pipeline of



(a)

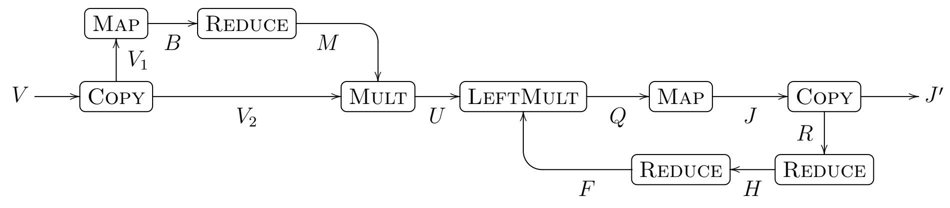


(b)

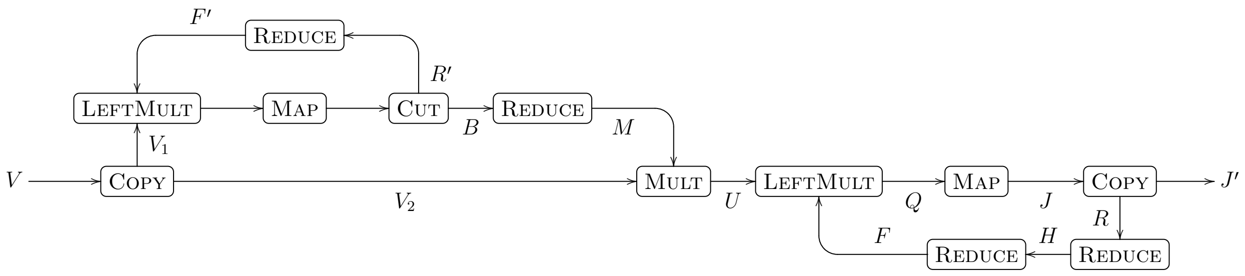
Figure 3.3: Expressing a parallel processing pattern in our stream algebra: (a) a parallel processing pattern using Scatter, ListMap, and Merge operators; (b) applying the parallel processing pattern in (a) to scale up the activity recognition pipeline discussed in Section 3.1.5 and process incoming video frames in parallel.

n operators (see Figure 3.2a) into a set of k groups (or intervals) such that the latency (sum of costs) per group is minimized. Figure 3.2c shows an example of partitioning the linear pipeline in Figure 3.2a into three intervals. Each interval has its operators merged together into one operator. If k represents the number of available cores in a multi-core computing platform, then merging operators into k intervals improve the allocation of computational resources. We aim to study these load balancing problems in the future. The Map operator can be also replaced by the parallel processing pattern in Figure 3.3. The runtime can dynamically choose between these different implementation plans to maximize throughput and reduce latencies required to move data between concurrent stream-processing operators.

Moreover, the Map and Reduce operators receive a list of functions as input. These functions can be different algorithms that perform the same task but with different accuracy and runtime profiles. For example, different algorithms exist for foreground segmentation or stereo vision. This enables dynamic reconfiguration by allowing the



(a)



(b)

Figure 3.4: Extending the single-loop feedback control of the online tracking example [47] to multi-loop feedback control: (a) the derived pipeline with single-loop feedback control for tuning the parameters of the tracking algorithm; (b) the pipeline with a second feedback control loop for tuning the parameters of the HOG-based people detector algorithm [53].

data-processing operators to switch between different functions at runtime. This is very important in large-scale systems processing a large number of incoming streams with different data rates. An incoming stream may have its data-flow rate change at a much faster rate. In this case, the pipeline may decide to switch the current processing functions to faster functions to match the new incoming data-flow rate. This decision can be performed dynamically using the feedback-control mechanisms of our stream algebra. Therefore, our algebra opens a new research direction in enabling dynamic reconfiguration in large-scale computer vision pipelines.

Our algebraic description of feedback control also creates new research problems in optimizing large-scale computer vision pipelines. These problems include resource reallocation, parameter tuning, and performance tuning. For example, our algebra fits naturally in describing the single-loop feedback control for the online tracking example [47] in Section 3.2.2. One can also extend this example to a multi-loop feedback control by tuning the parameters of both the tracking algorithm and the HOG-based people detector algorithm [53] used to locate moving objects. Figure 3.4a shows the pipeline of [47] expressed in our stream algebra with single-loop feedback control to tune the tracking algorithm. Figure 3.4b shows that we can also create a second feedback control loop to tune the people detector algorithm [53]. Notice that the second feedback loop is independent and starts by applying a Cut operator to sample the stream $B : \mathbf{S}\langle\textsc{List}\langle\texttt{Object}\rangle\rangle$, which defines the output of the people detector algorithm. This results in the return stream $R' : \mathbf{S}\langle\textsc{List}\langle\texttt{Object}\rangle\rangle$. A Reduce operator can then take the R' stream as input, locate objects that are strong candidates for people, fine-tune the learned people model used by [53], and output a stream F' of fine-tuned models. Later, a LeftMult operator is applied to attach with every incoming frame in the V_1 stream, a fined-tuned model from the F' stream. The LeftMult operator then produces the input stream of the Map operator that updates and executes the people detector algorithm. This example shows that multi-loop feedback control can be applied to different operators in a large-scale computer vision pipeline. Moreover, multi-loop feedback control can be applied in vision pipelines with multiple output streams. In this case, a different feedback loop can be defined for each output and rate-control operators, such as Cut and Latch, which can be used to decouple the data-flow rates of the feedback loops from the forward pipelines.

The iterative optimization example presented in Section 3.2.2 shows that our formal feedback-control description can be used to express and scale up tasks, such as incremental evaluation and adaptive learning. Pipeline instrumentation is also an important problem that requires further research, such as enabling performance monitoring, real-time debugging, bottleneck identification, and blocking resolution. Such tasks are important in developing and operating large-scale computer vision pipelines processing unbounded datasets. Other open research problems also include stream clustering and online classification of large-scale data.

3.4 Algebra Implementation

We implemented the stream-algebra framework in the Go language, which has good support for building scalable and concurrent systems. For example, the stream reading and writing functions $x \leftarrow s$ and $x \rightarrow s$ defined by our algebra are equivalent to channels in the Go language. The framework also supports OpenCV, [1] which allows programmers to access a larger set of computer vision algorithms when writing Map and Reduce operators. A special set of Map operators was also developed for viewing images and plotting graphs. The algebra implementation defines a programming interface for developers to compose pipelines using algebraic operators that can be chained together.

3.4.1 Algebra Implementation in Go Language

The full source code of our implementation is given in Appendix C. In this section, we are going to discuss the Map operator as an example for implementing the algebra operators in Go language. Then, we will show how it can wrap a function that performs Canny edge detection [41]. Finally, we will see the implementation of a simple pipeline that reads a stream of images and extracts edges from each incoming image. Notice that we will only focus our discussion on the important implementation details.

The key data structure in the algebra implementation is `NProcessesor` (see Section C.1 for full definition). This data structure represents a generic operator that follows Definition 3.1.9. The key fields in this data structure are:

```
type NProcessor struct {
        *ProcessorInfo
        Inputs   []chan T
        Outputs  []chan T
        F        func(inputs ...chan T) []chan T
        G      *OGraph
```

[1]OpenCV: https://opencv.org/ (*last accessed 2 September 2017*).

```
    ....
}
```

The `Inputs` and `Outputs` fields are arrays of Go channels. A Go channel is a pipe connecting concurrent go-routines, and a go-routine is a thread that is executed independently. So, a channel represents an incoming or an outgoing stream. The function `F` is the main operator function. It receives one or more incoming streams and produces one or more outgoing streams. The `G` field is a pointer to the workflow graph that contains the operator. The graph is defined using the `OGraph` data structure, which contains the following key fields (see section C.2):

```
type OGraph struct {
        *graph.Graph
        nodes_map    map[string]graph.Node  // a map for storing graph nodes
        ....
        edges_info   map[string]*EdgeInfo   // a map for storing input and output edges
        ....
}
```

The `OGraph` data structure inherits from a graph data structure to define a workflow graph that represents a streaming pipeline. Nodes represent operators and edges represent connections between operators. The data structure is augmented with additional fields that define the connections between the operators in a streaming pipeline. The `nodes_map` field is a map data structure used for fast lookup of a graph node associated with a certain operator. The `edges_info` field defines the edges of each operator with each edge storing data for input and output channels.

The `NProcessesor` data structure also inherits from the `ProcessorInfo` data structure. `ProcessorInfo` defines the user-defined mapping functions that process incoming data tuples. It is defined as,

```
type Functions []*Function
type ProcessorInfo struct {
        Name   string
        _type  int
        Funcs    Functions
        FuncIdx int //Current active Function
}
```

This data structure has the **`Name`** field that defines a unique name for the operator, the **`Funcs`** field that represents a list of user-defined functions, and the **`FuncIdx`** field that defines the index of the current active user-defined function. The structure also has the **`_type`** field that defines the operator type as either Map, Reduce, Cut, Latch,.., etc. The **`Function`** data structure is defined as follows:

```
type T interface{}        // hold values of any type
type Parameter struct {
        Value float64
        Low   float64 // lower bound
        High  float64 // upper bound
}
type Params map[string]Parameter
type Function struct {
        FuncName   string
        FuncParams Params
        State      T
        Mapper     func(T, Params) T
        Reducer    func(T, T, Params) (T, T)
}
```

This data structure is used by users to define data processing functions. It has the **`FuncName`** field that defines a unique function name. The function can be stateful or stateless. If stateful, the user defines both the **`State`** field and the **`Reducer`** function. In this case, the function is used by a Reduce operator as per Definition 3.1.11. If stateless, the user defines the **`Mapper`** function and leaves the **`State`** and the **`Reducer`** fields undefined. The function is then used by a Map operator as per Definition 3.1.10. The `FuncParams` field defines a map of parameters, where keys and values represent parameter names and their values, respectively. Notice that the `Parameter` data structure defines current value, upper-bound, and lower-bound of the parameter.

Using the previously discussed data structures, a Map operator can be defined as follows:

```go
func (g *OGraph) Map(funcs Functions, attribs ...T) *aGraph {
        proc := g.NewProcessor(nil, []chan T{make(chan T)}, OP_MAP)
        proc.Funcs, proc.FuncIdx = funcs, 0
        ....
        proc.F = func(inputs ...chan T) []chan T {
                ....
                proc.Inputs = inputs
                go func() {
                        ....
                        for {
                                x, ok := <-proc.Inputs[0]
                                if !ok {
                                        break
                                }
                                ....
                                UpdateSettings(proc.ProcessorInfo, x)
                                params := proc.Funcs[proc.FuncIdx].FuncParams
                                y := proc.Funcs[proc.FuncIdx].Mapper(x, params)
                                ....
                                proc.Outputs[0] <- y
                        }
                }()
                return proc.Outputs
        }
        return &aGraph{g, proc}
26 }
```

The Map operator is defined according to Definition 3.1.10. It takes as input, a list of user-defined functions and set of optional attributes. Remember that Map receives a single input stream and produces a single output stream. An instance of the `NProcessesor` data structure is created in line 2 with only one output channel defined. We then assign the list of functions to the `Funcs` field of the `NProcessesor` structure. In line 5, we define the operator main function that receives incoming streams and produces the outgoing streams. Line 7 assigns the array of incoming streams to the `Inputs` field of the `NProcessesor` structure. In the case of Map, this array has only one stream. Line 8 executes a go-routine for the Map operator. This routine runs a loop that reads each incoming element from the input stream (or channel) and tests if the stream is closed or not. The `UpdateSettings` function in line 16 is the lookup function that examines the header of the incoming element. It also updates the index of the current active mapping function and its list of parameters. Line 17 then executes the current active `Mapper` function. The output is then forwarded to the output stream in line 19. Notice that the `aGraph` data structure, used in line 24, is a wrapper for the algebra operators and is used for chaining and connecting operators in an operator graph.

To define a simple pipeline that performs edge detection using a Map operator, the following `Canny` function is defined using OpenCV,

```go
func Canny(x T , z Params) T {
        var (image, gray        *opencv.IplImage
                t1, t2 float64 = 60, 180)

        if thr, ok := z["thr1"]; ok {t1 = thr.Value}
        if thr, ok := z["thr2"]; ok {t2 = thr.Value}
        image = x.(*opencv.IplImage)
        gray = opencv.CreateImage(image.Width(), image.Height(), opencv.IPL_DEPTH_8U, 1)
        opencv.CvtColor(image.Ptr(), gray.Ptr(), opencv.CV_BGR2GRAY)
        opencv.Smooth(gray.Ptr(), gray.Ptr(), opencv.CV_BLUR, 3, 3, 0, 0)
        opencv.Canny(gray.Ptr(), gray.Ptr(), t1, t2, 3)
        edges := FindEdges(gray)
        gray.Release()

        return T[]{image, edges}
}
```

Notice that the `Canny` function receives an input image `x` and a list of parameters `z`. Here the parameters are the two popular thresholds used by the Canny edge detector. The function executes several OpenCV operations to extract the list of edges and returns a vector containing both the input image and the detected edges.

In order to produce the source stream of images, we define the following generator function that uses OpenCV to read a sequence of 1000 images stored on disk,

```
type Spout struct {ci  int}
func (sp *Spout) Read() loopy.T {
        x= opencv.LoadImage(fmt.Sprintf("data/%s.jpg", string(sp.ci))
        sp.ci++
        If  sp.ci > 1000 {return STOP}
        return x
}
```

The `STOP` signal informs a Source operator to terminate. Now, we can construct the following pipeline for edge detection,

```
g := NewOGraph("graph")
spout := &Spout{}
detect_edges := alg.Functions{&alg.Function{
        FuncName:  "canny",
        Mapper:      Canny,
        FuncParams:   alg.Params{"thr1": alg.Parameter{Low: 0, High: 255, Value: 60},
                                  "thr2": alg.Parameter{Low: 0, High: 255, Value: 180}}}}
```

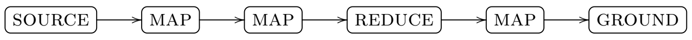
```
 8  view_edges := alg.Functions{&alg.Function{
 9          FuncName:          "view_edges",
10          Mapper:          ViewEdges}}
11
12  g.Source(spout).Map(detect_edges).Map(view_edges).Ground()
13  g.Execute()
```

Notice that the pipeline starts by creating a new workflow graph in line 1. Then, the object `spout` is defined in line 2. This object implements the `Read` generator function. Line 3 defines a list of functions that only contains the `Canny` Mapper function. The parameters of the Canny edge detector is also defined. Similar to the definition of the `Canny` Mapper function, we also define the `ViewEdges` Mapper function that displays the extracted edges. The pipeline is constructed in line 8 and starts by creating a Source operator that receives the `spout` object. A Map operator is then applied on the output stream of the Source operator to extract edges using the `Canny` function. The operator produces a stream containing images and their extracted edges. Another Map operator is then applied to display the extracted edges. Finally, a Ground operator ends the pipeline.

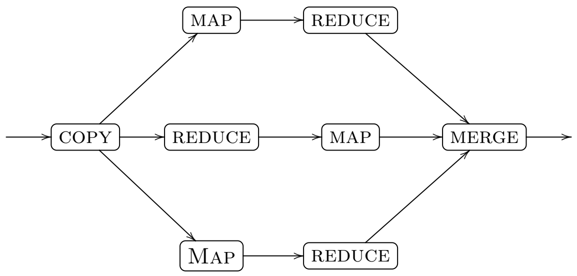
Similar to the definition of the `Canny` Mapper function, we wrap several OpenCV functions into either Mapper or Reducer functions. So, a new OpenCV interface is defined that is compatible with our algebraic operators.

3.4.2 Building Pipelines Using the Algebra Implementation

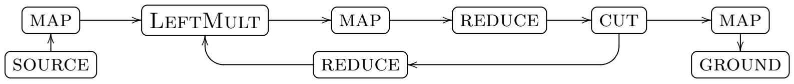
To build a pipeline using our algebra implementation, an operator graph should be defined by chaining the data-processing and flow-control operators. For example, to implement the feedforward pipeline in Figure 3.5a, for the mapping operators, we assume the three list of functions f_1, f_2, and f_3 and the set of parameters p_1, p_2, and p_3. For the Reduce operator, we also assume a list of functions g_1, a state variable u_1, and a set of parameters q_1. For the Source operator, a generator function h and a state variable u_0 are defined.



(a)



(b)



(c)

Figure 3.5: Different examples of workflow graphs: (a) pipeline graph; (b) fork-join graph; and (c) pipeline graph in (a) with a feedback-control loop.

The pipeline can then be written using our algebra Go language implementation as follows:

```
g := NewGraph("fork-join")
g.Source(u_0,h).Map(f_1,p_1).Map(f_2,p_2).Reduce(u_1,g_1,q_1).Map(f_3,p_3).Ground()
g.Execute()
```

Notice that a graph g is initially defined, and the dot operator is used to add the algebra operators and chain them together. Here, the dot operator forwards the output stream of its left operand to the input stream of its right operand. The Source operator can generate an image or video stream that can be later processed by subsequent operators until we reach Ground. The `Execute()` function distributes the pipeline tasks to available computational resources and concurrently executes the operators.

Computer vision pipelines with parallel processing patterns, such as the pipeline in Figure 3.5b, can be also implemented in our algebra using the following Go language code:

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

Again, line 1 defines a new empty operator graph and assigns it a name. The next lines then add operators to the graph using the dot operator. Here, each operator takes an extra argument that defines a unique operator name. Line 2 defines the Copy operator. Line 3 defines the Merge operator, which takes a function f to merge elements of the incoming streams. Lines 4, 5, and 6 define the three parallel branches in Figure 3.5b. To link these branches with the Copy and Merge operators, two special functions are defined, `LinkOut` and `LinkIn`. Both functions take a list of strings, with each string defining a unique operator name. Lines 7 and 8 use the `LinkOut` and `LinkIn` functions to connect the operator graphs together. Finally, line 9 executes the graph.

The `LinkOut` function starts by obtaining the list of unassigned output channels of its first input operator. The function also creates a list of input channels containing the unassigned input channels of all other given operators ordered by their position in function arguments. Output channels are then linked to their corresponding input channels.

The `LinkIn` function operates in a similar manner. The function obtains the list of unassigned input channels of its first input operator. The function creates a list of output channels containing the unassigned output channels of all other given operators ordered by their position in function arguments. Then, input channels are linked to their corresponding output channels.

We can also implement feedback control using our stream-algebra implementation. Figure 3.5c shows a similar pipeline to Figure 3.5a, but with feedback control defined. This pipeline can be implemented as follows:

```
1 g := NewGraph("feedback")
2 g.Source($u_0, h$). Map($f_1, p_1$,"m1")
3 g.LeftMult("lm").Map($f_2, p_2$).Reduce($u_1, g_1, q_1$).Cut("ct")
4 g.Map($f_3, p_3$,"m3").Ground()
5 g.Reduce($u_4, g_4, q_4$, "r4")
6 g.LinkOut("ct", "m3", "r4")
7 g.LinkIn("lm", "m1", "r4")
8 g.Execute()
```

Here, line 1 creates the graph. Line 2 defines the branch that contains the Source operator. Line 3 defines the feedforward branch controlled by the feedback loop. Line 4 defines the output branch that contains Map followed by Ground. Line 5 defines the Reduce operator of the feedback loop branch. Lines 6 and 7 link all branches together using the `LinkOut` and `LinkIn` functions. Finally, the graph is executed on line 8.

The previous examples show the ability of our stream-algebra implementation in defining and executing computer vision pipelines. Moreover, higher-order operators defined in Section 3.1.4 can be used similarly to scale up computer vision pipelines to process a large number of continuous and possibly infinite vision streams.

Chapter 4

Efficient Computer Vision Functionals: Pixel Labelling

Discrete pixel labelling is an important area of computer vision that includes a set of fundamental problems, such as interactive image segmentation, stereo vision, optical flow, multi-view image mosaicing, and object recognition. The approaches for solving these problems are usually cast within energy minimization. They involve an assignment problem $f : \mathcal{L} \to P$ that selects the label of minimal cost $l \in \mathcal{L}$ from a set of labels $\mathcal{L}$ for each image pixel $p \in P$. The costs are represented as a 3D volume $w \times h \times L$ for $L = |\mathcal{L}|$ and an image of width w and height h. This volume is called the cost volume and has each slice recording the assignment costs of a certain label. A solution is found by minimizing the total assignment cost.

Global optimization based on Markov random fields (MRF) is a traditional approach for obtaining solutions; however, it is computationally expensive. Local optimization methods provide a more efficient alternative. These methods rely on local cost filtering and aggregation and are referred to as cost-volume filtering. Local methods, however, still need to traverse and filter the entire cost volume. This renders their performance very slow in processing large cost volumes usually found in optical flow and high-resolution images.

In this chapter, we present our sparse cost-volume filtering approach, which restricts local filtering and aggregation to a selected set of salient sub-volumes, resulting in a significant improvement of performance. We discuss two techniques for selecting the sub-volumes, the feature-based and segmentation-based methods. The feature-based method depends on feature matching and keypoint detection, whereas the segmentation-based method relies on superpixel segmentation and nearest-neighbour fields. We also present an occlusion handling (OH) technique to fill occluded regions resulting from incorrect label assignments. This is performed using a label propagation method inspired by simulated annealing [68]. A complexity analysis of our methods shows a linear complexity $O(n)$ for the segmentation-based method, where $n = w \times h$, and $O(n \times k)$ for the feature-based method, where k is the number of keypoints. We are interested in two important types of pixel-labelling problems, optical flow and stereo vision; however, our method is general and applicable to other labelling problems, such as image segmentation.

The proposed method is described in our stream algebra and implemented as a mult-GPU streaming pipeline. The pipeline can process image and video streams at near real-time rates. We also discuss how our stream algebra can scale up the pipeline to utilize the available GPU resources and process multiple video streams simultaneously.

The contributions of this chapter are: (1) we present the sparse cost-volume filtering approach that achieve state-of-the-art runtime performance numbers for pixel labelling problems, on standard optical flow and stereo vision benchmarks; (2) we develop two methods for efficiently identifying salient sub-volumes within cost volume based on feature matching and superpixel segmentation; (3) we show that the sparse cost-volume filtering approach can be implemented with a computational complexity linear in the image size and independent of the label space size; and (4) we propose a robust gap filling strategy for occlusion handling and refinement of computed label assignments.

4.1 Introducing Pixel-Labelling Problems

In this chapter, we focus on two instances of pixel-labelling problems, optical flow and stereo vision. Generally, the desired solution defines a map that (i) has spatial smoothness, (ii) matches assignment costs, and (iii) respects 3D surface boundaries. Given an image pair, optical-flow estimation is a fundamental problem in computer vision. Optical flow describes two-dimensional (2D) motions of objects (usually individual pixels) between the two images, and it is often the first step in many computer vision techniques, such as motion detection, object segmentation, video encoding and compression, scene analysis, activity recognition, etc. Many of these techniques treat optical flow as a pixel-labelling problem, where each label $l \in L$ is a 2D vector (u, v) describing the pixel displacement between two input images [50, 18, 123, 91]. Despite great advances in optical-flow estimation since the early seminal works by Horn and Schunck [89] and Lucas and Kanade [124], we still need better methods for dealing with large-displacement optical-flow estimation. Most existing methods for optical flow ignore higher-order terms in the optical-flow constraint equation, which leads to poor performance when dealing with large motions [62]. For stereo vision, there has been substantial interest in developing pixel-labelling techniques to estimate disparity maps from stereo image pairs [91]. The label $l \in L$ is defined as a disparity value d, and the goal is to assign a disparity to each pixel.

Both optical flow and stereo vision can be formulated based on MRFs and be solved using global optimizers [54]. The assignment cost is typically written as an energy function with two terms: a data term and a smoothness term. The data term accounts for per-pixel label assignment, whereas the smoothness term considers labels assigned to neighbouring pixels. A solution can then be obtained using traditional energy minimization algorithms, such as graph cut [33] and belief propagation [171, 157, 57]. Though these techniques give reasonably accurate results, they have a large computational cost, which limits their applicability to large label spaces usually found in high-resolution images.

Moreover, dealing with large-displacement motions requires that we expand the label space, increasing the size of the cost volume. This leads to increased memory requirements and longer processing times. It may also result in noisier final optical flow [184]. Local filtering methods [121, 91, 123], on the other hand, provide an efficient alternative by providing locally and spatially smooth label assignments against the globally smooth assignments produced by MRFs.

Local filtering was first applied by [179] and [144] for stereo vision. The method by [179] had a high computational cost and had little value compared to global optimization, whereas the technique of [144] approximated filtering using a fast implementation that provides a considerable speed increase with a loss of accuracy. The benefits of local methods for pixel-labelling problems have been shown by Hosni et al. [91], where edge-aware *guided filters* have been used to achieve high-quality results comparable to the global optimization methods for different pixel-labelling problems, including optical flow and stereo vision. Edge-aware filtering (EAF) was chosen for its linear-time complexity that does not depend on the kernel size.

Lu et al. [121] developed a much faster filtering algorithm based on multi-point regression. The same authors [123] extended the work further by developing the *PatchMatch* filter, which combines EAF with the *PatchMatch* algorithm. Their work provided a much further speed increase with a sublinear complexity in the label space size.

Given the current trend to pack more pixels per image, we also need more efficient methods for solving pixel-labelling problems. We envision that these methods will be able to trade accuracy versus speed, adapting to imaging artifacts, such as large motions, motion blur, occlusions, etc., and leveraging the available hardware in the best way possible. Motivated by this vision, our sparse cost-volume filtering approach allows large-displacement optical-flow and stereo-vision estimation. It relies upon sparse processing, which can be tuned to achieve the desired accuracy versus speed balance.

There are several advantages of the feature-based and segmentation-based methods.

They leverage sparse filtering to achieve state-of-the-art runtime performance numbers on standard benchmarks. They also allow efficient identification of salient sub-volumes within the cost volume. Moreover, an OH scheme is developed for filling gaps resulting from parallax and erroneously labelled regions that can be applied independently as a refinement step for improving initial label assignments of other methods [91, 121]. Finally, the computational complexity of our superpixel-based method is linear in the image size and independent of the size of the label space.

4.2 Cost-Volume Filtering

Given an input image pair (I_1, I_2), our goal is to assign a label $l = (u, v) \in L$ to each pixel $(x, y) \in I_1$. The label represents the displacement of pixel $(x, y) \in I_1$ to $(x + u, y + v) \in I_2$, and L is the label space. It is easy to extend this idea to multiple images $(I_1, I_2, \cdots, I_n)$. In this case, without the loss of generality, the first image I_1 is usually referred to as the reference image I_{r}, and we deal with image pairs of the form (I_{r}, I_k), where $I_k \in \{I_2, \cdots, I_n\}$.

Our methods follow the general framework of a local correspondence search for computing optical flow [91]. The framework consists of three steps. The first step uses pixel correspondences to set up a cost volume $C(x, y, l)$, which stores the cost of assigning a label $l \in L$ to a pixel (x, y). The second step aggregates costs at each cost slice using EAF. The third step picks the optimal label assignment for each pixel to minimize the overall cost of label assignment. A final step is often used to further refine label assignments, including assigning missing labels. In the case of optical-flow estimation, the desired solution to this label assignment problem is spatially coherent, follows label assignment costs, and preserves edge discontinuities. During filtering, each slice l of the cost volume is processed as follows:

$$C'(x, y, l) = W_{(x,y)} \otimes C(x, y, l), \tag{4.1}$$

where the kernel weights $W_{(x,y)} \in \mathbb{R}^{(2q+1)\times(2q+1)}$ for window $\omega_{(x,y)}$, centred on (x, y), depend on the reference image and are computed for every (x, y). The kernel radius is q. The symbol $\otimes$ denotes the convolution operator. The reference image is often called the *guidance* image. Many schemes select $W_{(x,y)}$ to maintain the intensity changes and preserve the edges of the guidance image [74, 184, 129]. In this work, we use the method proposed by [74] to compute $W_{(x,y)}$. Given $i = (x, y)$ and $j = (x', y')$ such that j are neighbours of i in ω_i, we have the weight $W_i(j) = \frac{1}{(2r+1)^2} \sum_{k:i,j\in\omega_k} \left(1 + \frac{(I_{\mathrm{r}}(i)-\mu_k)(I_{\mathrm{r}}(j)-\mu_k)}{\sigma_k^2+\epsilon}\right)$, where μ_k and σ_k^2 are the mean and variance of I_{r} in ω_k, and ϵ is a regularization parameter. The selection step applies a *winner-takes-all* strategy to pick a label in L that minimizes the assignment cost for each pixel p at location (x, y):

$$l_p = \arg\min_{l} C'(x, y, l). \tag{4.2}$$

Cost-volume filtering is linear in the size of label space, which makes these techniques slow for large L [91, 184]. To build the cost volume $C(x, y, l)$, several strategies can be used depending on the application. For example, in [83], we use the fronto-parallel plane sweep algorithm by [64] to generate a discrete set of disparity planes for finding a stereo-disparity map. This set defines the cost volume. Moreover, [91] assumed a displacement window of $[\Delta w, \Delta h]$ around each pixel for optical flow. Lu et al. [123] considered a continuous range of labels for optical flow and stereo vision and applied the randomized search PatchMatch algorithm to locate the best label for each pixel.

4.2.1 Curse of the Label Search Space

The traditional cost-volume-filtering methods [91, 184] are computationally infeasible for large and continuous label spaces. These label spaces are encountered when dealing with high-resolution images, large motions, and subpixel-accurate optical flow and stereo vision. Although dealing with high-resolution images and large motions is straightforward for producing large label spaces due to searching for more pixels for a solution, it is not clear for subpixel accuracy.

During image acquisition, the continuous scene information is quantized into a discrete set of image pixels; however, we can still fit models that estimate the continuous geometry of the scene and retrieve colouring information at subpixel accuracy [61]. This allows finding highly accurate solutions for optical flow and stereo vision at a fraction of a pixel. Several models have been used for attaining subpixel accuracy. For example, Hosni et al. [91] uses bicubic interpolation to upscale the input images by a certain scale factor. To sense how this scale affects the label space size, we consider solving for optical flow between two input images. For each pixel in the first image, if we assume that we search a 2D displacement range of $[-40, 40] \times [-40, 40]$ pixels in the second image, then a scale factor of 8 produces a label space of $81 \times 81 \times 8 \times 8 = 419904$ labels. Bleyer et al. [125] assumed an infinite (continuous) label space for solving subpixel-accurate stereo vision along horizontal motions. Their algorithm takes two rectified input images with the radial distortion removed. The algorithm then searches for each pixel p in the input image, the 3D plane f_p from which the pixel is projected. The plane is defined as:

$$d_p = a_{f_p} p_x + b_{f_p} p_y + c_{f_p}, \tag{4.3}$$

where d_p defines the disparity (or displacement) of pixel p in the first image to its correspondent location in the second image. A label $l_p = (a_{f_p}, b_{f_p}, c_{f_p})$ is the set of plane coefficients defined over a continuous range of values. This simply generates an infinite label space L. Moreover, the gradient-based methods such as the work of Brox *et al.* [37] also naturally recover subpixel accuracy by assuming a continuous model for motion estimation.

4.2.2 Efficient Traversal of Large and Continuous Label Spaces

To solve pixel-labelling problems for large and continuous label spaces, we need efficient methods to search and traverse these spaces. Bleyer et al. [125] applied the PatchMatch algorithm for the randomized search of the infinite space of 3D planes. Figure 4.1 shows the main steps of the PatchMatch algorithm for two images denoted by A and B. Initially, for each pixel, we define a local window (patch) centred on that pixel. If A is the reference image, we initialize each patch p in A by a random displacement (label) to a correspondent patch in B. A matching cost is calculated for the displacement comparing the two patches. The algorithm iterates between two steps. (1) Labels are propagated from neighbouring pixels of p, and if the propagated label gives better matching cost than the current assigned cost of p, the propagated label replaces the current label of p. (2) A random search is performed around the best displacement in image B to find a better matching cost. A random search starts with the image size and is halved until we reach 1. The PatchMatch algorithm converges after a certain number of iterations, and the solution is referred to as a nearest-neighbour field (NNF). In [125], the displacement comes from 3D plane coefficients. The matching cost is defined using Equation 4.6 along only the x dimension (for horizontal motion), where $d_2 = |\nabla_x I_{\mathrm{r}}(x, y) - \nabla_x I_k(x_k, y_k)|$, and the cost aggregation is done using Equation 4.1. The complexity of the algorithm is $O(mM \log L)$ for a patch of size m, image with M pixels, and a label space size L.

Lu et al. [123] developed a better PatchMatch method with a time and space complexity of $O(M \log L)$ and $O(M)$, respectively. Their method starts by dividing the image into a set of compact superpixels using the simple linear iterative clustering (SLIC) superpixel segmentation algorithm [1]. Rather than defining a patch for each pixel as in the work by [125], the method defines a single patch for each superpixel with cost matching and aggregation done at the superpixel level. As in [125], we randomly search the entire label space.

Chen et al. [50] relied on the concept of dominant motions to locate a set of sub-

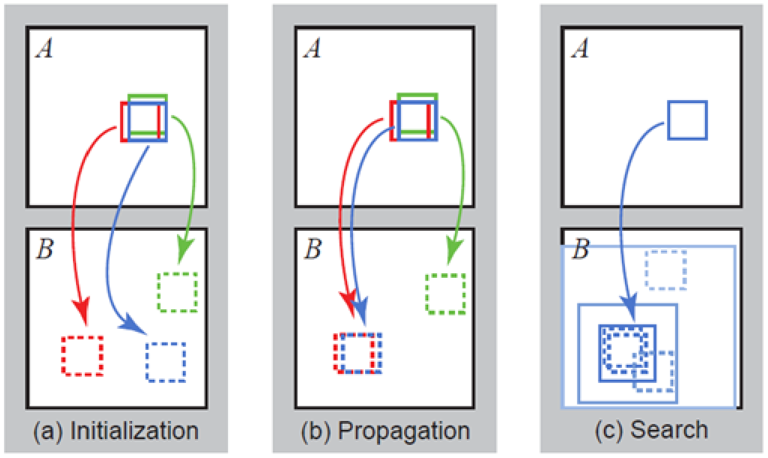


Figure 4.1: Three main stages of the PatchMatch algorithm [19].

volumes inside the cost volume. Then, a multi-label graph cut is used to solve for optical flow, which results in a high computational cost of around 362 seconds to process an image of 640 × 480 pixels. Figure 4.2 (left) shows the sub-volumes constructed by [50], where each sub-volume spans a fraction of the label space and the entire spatial space. Figure 4.2 (middle) shows the sub-volumes randomly searched by [122], where each sub-volume spans a small superpixel in the spatial space and the entire label space. Figure 4.2 (right) shows the sub-volumes constructed by our superpixel-based method spanning a fraction of both the spatial and label spaces.

An obvious direction for building better efficient label space traversal methods is to detect the sub-volumes that contain the best label assignments. In the next sections, we discuss our sub-volume detection strategies.

4.3 Selecting Salient Sub-volumes

We discuss two methods for selecting salient sparse sub-volumes inside the main cost volume. We restrict cost filtering and aggregation to these sub-volumes to obtain reduced runtime and large gain in the overall performance. The first method relies on feature matching, and the second method depends on superpixel segmentation and ANNFs.

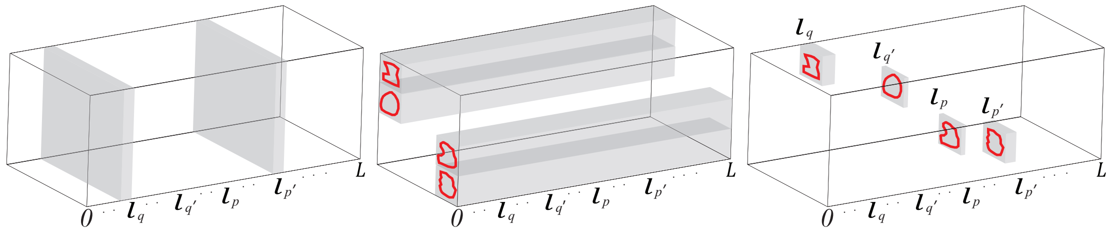
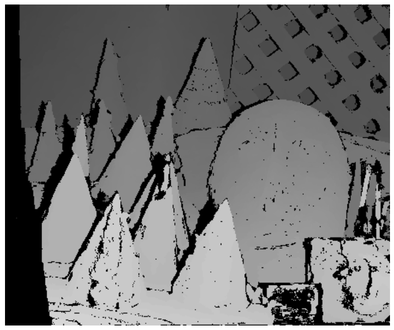


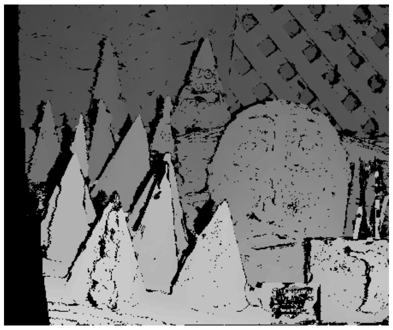
Figure 4.2: Visual comparison of the different methods for locating sub-volumes. Cost volume is $I_{\text{height}} \times I_{\text{width}} \times L$, where I_{height} and I_{width} are the image height and width, respectively, and L is the label space size. (Left) Sub-volumes identified by [50] around dominant motions. (Middle) A sub-volume is defined by [123] for each superpixel (shown in red). (Right) Sub-volumes are defined using the cost-sub-volume filtering (SVF) method [80] for each superpixel, around the dominant motion.

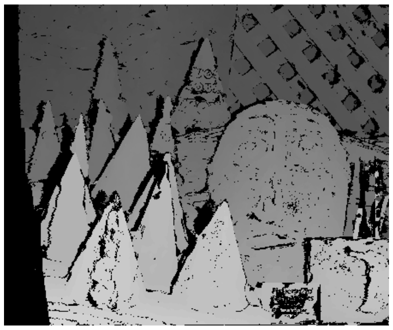
4.3.1 Feature-based Sub-volumes

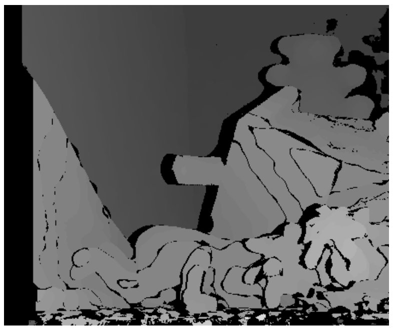
Our feature-based approach is based on two main hypotheses: the visibility hypothesis and the selection hypothesis. In the *visibility hypothesis*, we can divide each cost slice into visible and non-visible regions, where the visible regions have the reference image $I_{\text{ref}}(x, y)$ matching the other image $I_k(x, y)$ well. In the *selection hypothesis*, we can identify the visible regions as salient regions in the cost volume. We refer to the feature-based approach as accelerated cost filtering (ACF) [79].

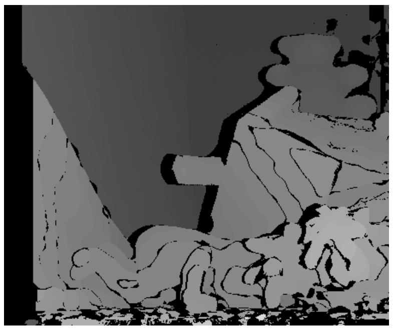
The naive way to find these regions is to traverse the entire cost volume $C(x, y, l)$ looking for the low-cost regions at each slice l. A better way is to extract scale-invariant feature transform (SIFT) keypoints using feature matching between the input images [118, 35]. Given the matched points, their disparity values can be calculated forming a set of seeds (x, y, l') in the cost volume. We can then construct a salient region around each seed (x, y, l') by first centring a local window $b_{l'}(x, y)$ on (x, y, l'). If more than one point exists on the slice l', we define the salient region as the minimum bounding window $b_{l'}$ surrounding all $b_{l'}(x, y)$. Each local window $b_{l'}(x, y)$ has its radius chosen as a fraction $r \times I_{\text{width}}$ of the reference image width. In this work, r is set to either 0.2 or 0.3. Given the discrete representation of the cost volume, the labels of seeds (x, y, l') may not match the location of cost slices well. This problem is solved by considering for each seed (x, y, l'),


(a) CF. % occ. Pix. = 17.6


(b) ACF. % occ. pix. = 19.9


(c) ACF. % occ. pix. = 20


(d) CF. % occ. pix. = 18.46


(e) ACF. % occ. pix. = 18.12

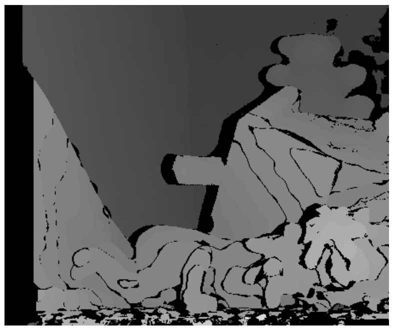

(f) ACF. % of occ. pix. = 18.58

Figure 4.3: Examples of the initial disparity maps generated by our accelerated cost filtering (ACF) stereo-estimation method [79]. The occluded pixels are shown as black regions. Top row (Cones): Disparity maps computed before gap filling: (a) result estimated by cost filtering (CF) [91], (b) ACF ($r = 0.2$), (c) ACF ($r = 0.3$). Bottom row (Teddy): Disparity maps computed before gap filling: (d) result estimated by CF, (e) ACF ($r = 0.2$), (f) ACF ($r = 0.3$). Variable $r \in [0,1]$ controls the radius of the local window used around feature points. Specifically, the local window radius is set to $r \times I_{\text{width}}$. It is worth mentioning that our method achieves less occluded pixels for the Teddy dataset. This shows that our sub-volume filtering scheme can produce more accurate results for some scenes.

a set of neighbouring slices inside the cost volume. Formally, we compute local windows b_l for each cost slice l that has $\|l - l'\| \leq \varepsilon$, and ε is a parameter controlled by the user for the expansion in the cost-volume space. Figure 4.4 shows the extracted keypoints and the detected salient windows at two disparity values in a stereo-vision application.

The detected salient regions in neighbouring cost slices define a sparse set of 3D cuboids inside the cost volume. We call each cuboid a sub-volume. Although the set of detected sub-volumes provides a good summary of the entire cost volume, the feature-based selection method has several limitations. The first problem is requiring the user

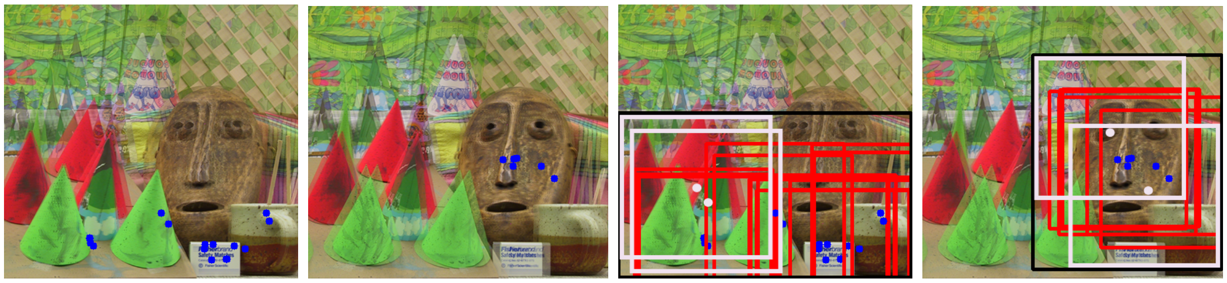


Figure 4.4: Finding salient regions using our feature-based sub-volume selection method. The two left figures show the overlapping between the left and right images at disparity planes 10 and 14, respectively. Each plane is obtained by displacing the right image with respect to the left image by the disparity value. The detected keypoints identify the best matching regions in each plane. The last two images define salient local windows (red squares) around the keypoints. Each plane can have other salient local windows expanded from neighbouring planes. Expanded windows with their keypoints are shown in white in the last two figures. Our algorithm defines the final salient region (black rectangle) for each plane as the bounding rectangle of all defined and expanded local windows. The side length of each local window is defined as $r \times I_{\text{width}}$, $(r = 0.2)$.

to define the r and ε parameters that may differ between different datasets. Another problem is relying on the quality and performance of keypoint detection and matching.

4.3.2 Segmentation-based Sub-volumes

To solve the limitations of feature-based sub-volume selection, we need to develop a new method for dynamically selecting sub-volumes within the cost volume. To do this, we use dominant motions to summarize the pixel displacements between successive frames. A dominant motion is the average motion vector for a group of neighbouring pixels in a pair of images. The idea of dominant motions was used by Chen et al. [50] for optical flow to handle the noise present in an initial optical flow computed using ANNF. However, the dominant motions are extracted using a costly motion-segmentation step. Local perturbations are extracted around each dominant motion to define locally deformed motions. These motions, together with dominant motions, define a salient range within the label space. We extend this idea to cost-volume filtering by observing that dominant motions can divide cost volume slices into visible and non-visible regions. This is akin to using keypoint matching for stereo-disparity estimation, as we did in our first sub-volume selection method [79]. Consequently, it is possible to identify salient sub-volumes in the cost volume using dominant motions.

We start by using the fast edge-preserving algorithm of [18] to compute ANNF using the input image pair. Moreover, [18] achieved a speed increase by downsampling the input images, which degrades the accuracy. Still, the ANNF computed using this method closely matches the ground truth (optical flow) at numerous pixel locations. In addition, [50] performed an empirical study that also supports this observation. It highlights the advantage of using ANNFs for computing optical flow. The ANNF computation does not constrain the search radius between corresponding patches, which allows it to capture large motions between two images. Unlike traditional optical-flow estimation schemes, ANNF computation also preserves small and thin structures. The primary issue with ANNF-based methods for computing optical flow is the existence of noise and missing labels for certain pixel locations. We now discuss how to resolve this issue.

We are interested in a fast algorithm for solving pixel-labelling problems. Therefore, using a costly motion-segmentation step to compute dominant motions to identify the salient sub-volumes within the cost volume does not serve our purpose. Instead we rely upon superpixel segmentation to identify the salient sub-volumes [129, 123, 79]. Our choice stems from the following three observations: (1) The desired solution for optical flow should be spatially smooth and preserve intensity changes of image edges. (2) Compact and spatially uniform superpixels respect image boundaries and include pixels that have a higher chance of sharing similar labels. (3) Processing superpixels reduces the computation complexity and boosts performance. We use the SLIC superpixel segmentation algorithm [1], which scales linearly with the image size and can be efficiently computed in real time.

Specifically, using the SLIC superpixel segmentation algorithm, the input image I is decomposed into K non-overlapping superpixels $\mathcal{S} = \{S_i | i \in [1, K]\}$. The ANNF computed earlier is then used to compute dominant motion μ and motion variance σ^2 for each superpixel S. Specifically, (u, v) for pixels (x, y) belonging to each superpixel S_i are used to estimate μ_i and σ_i^2; $\mu_i = (\widehat{u}, \widehat{v})$ is computed using the mean shift algorithm, and $\sigma_i^2 = (\sum_x (u - \widehat{u})^2, \sum_y (v - \widehat{v})^2)/n$ for $n = |S_i|$. The set $\Omega_\mu = \{\mu_i | i \in [1, K]\}$ defines K dominant motions, and the set $\Omega_{\sigma^2} = \{\sigma_i^2 | i \in [1, K]\}$ defines the motion variances corresponding to K superpixels. Together, these sets are used to define a sparse set of K sub-volumes $\mathcal{V} = \{V_i | i \in [1, K]\}$. The width and height (defined in the image space) of sub-volume V_i is determined by the minimum bounding rectangle B_i of superpixel S_i and its depth (defined in the label space) is set such that it contains all labels l that satisfy $|l - \mu_i| < \beta_s \sigma_i^2$. For the results presented here, β_s (the expansion factor) is set to 1.9. In practice, we bound the expansion by γ_s to avoid situations involving very large σ_i^2 by ensuring that $\beta_s \sigma_i^2 \leq \gamma_s$.

Figure 4.2 shows a comparison between our method of constructing sub-voltumes based on superpixel segmentation against other methods. We refer to this method as the cost-sub-volume filtering (SVF) [80]. Figure 4.2 (left) shows the sub-volumes defined by [50] around the dominant motion patterns. Here, the sub-volumes span the entire image space and very small sections of the label space. Figure 4.2 (middle) shows the searched sub-volumes constructed by [123] for each superpixel, where the image space is partitioned, but the entire label space is randomly searched. Figure 4.2 (right) shows our method of constructing sub-volumes that capture the benefits of both [123] and [50] by partitioning both the label and image spaces.

4.4 Coarse-to-fine Sub-volumes

From a biological point of view, the human brain performs stereoscopic vision at multiple hierarchical scales (from coarse to fine) [67]. This motivates the development of coarse-to-fine approaches for solving pixel-labelling problems, especially optical flow and stereo vision. For example, Zhang et al. [185] extended the cost-volume-filtering approach for cross-scale cost aggregation. This is performed by constructing a set of cost volumes at different scales. Then, Equation 4.2 is extended for the multiscale approach by performing intra-scale cost aggregation while forcing an inter-scale consistency. Given a pyramid of multiscale cost volumes, we can use our feature-based sub-volume selection method to locate salient sub-volumes at each scale. Then, we restrict the cross-scale methods, such as in [185], to perform cost aggregation only with the multiscale sub-volumes. Notice that we only need to build the sub-volumes, rather than construct the entire cost volumes. This saves the time required to both build and process the entire set of cost volumes.

We can also extend our segmentation-based sub-volume selection method to a multi-scale method by identifying a set of sub-volumes for each superpixel at different scales. As discussed before, the sub-volume selection in this case depends on the initial ANNFs computed by the fast edge-preserving PatchMatch method by [18]. In PatchMatch, the patch size affects both the runtime complexity and accuracy. Larger patch sizes are better at enhancing EAF and resolving ambiguities of matching costs, thereby producing higher quality ANNFs. Thus, larger patch sizes are desirable for our method to produce a good set of sub-volumes. However, it is often not possible to specify the optimal patch size without knowing the object scales *a priori*. Therefore, employing multiscale approaches typically solves such issues.

Thus, for multiscale segmentation-based sub-volumes, we construct an n-level image pyramid from the input image starting with the input image and downsampling it by half for successive levels. For all our experiments, we set $n = 3$. Initial labelling maps are computed for each level according to [18]. Next, for each superpixel S_i in the original image, a sub-volume V_i^s is constructed at level s using the labels of pixels belonging to S_i at that level, which constructs a set of sub-volumes $\{V_i^s | s \in [1, n]\}$ for each superpixel S_i. Sub-volumes V_i^s share the same bounding box B_i in the original image. These sub-volumes are subsequently filtered in the original image resolution (i.e., level 1), ensuring that small and thin structures are not lost during filtering. Downsampling the image while keeping the patch size constant has been investigated by [150]. It effectively increases the patch size for computing the initial flow maps (using the method in [18]), improving the quality of sub-volumes. Moreover, it allows sub-volumes defined at coarse scales to capture large displacements with the modest computational expense.

4.5 Patch-Match for Sub-volume Filtering

To generate labelling solutions from the selected sub-volumes, we need to apply cost aggregation. The traditional method is to build and filter the 3D cost sub-volumes spanning the image and label spaces. This is feasible for small and discrete label spaces; however, it becomes unfeasible for large and continuous label spaces usually found in high-resolution images and subpixel-accurate optical flow and stereo vision. A better approach is to randomly search the sub-volumes to find the best label for each pixel. PatchMatch performs the randomized search efficiently by defining a 2D patch for each pixel in one image and randomly sampling the other image to search for the best match. It is independent of the search range and uses the natural homogeneity of image regions to propagate matches to neighbouring areas. Bleyer et al. [125] built upon this algorithm to develop PatchMatch Stereo, which randomly traverses an infinite set of 3D planes for each pixel in the reference image to find the optimal plane at which the pixel matches its projection to the other image. This infinite set of planes defines a cost volume over a continuous label space of 3D planes. The method was shown to outperform the traditional cost-volume-filtering method by [91].

Given our feature-based sub-volumes, we can restrict the PatchMatch algorithm to search inside them for optimal labels. However, a better alternative is to avoid the limitations of feature-based sub-volumes selection and restrict PatchMatch to the segmentation-based sub-volumes. In this case, we can build upon the PatchMatch filter method developed by Lu et al. [123]. This method outperforms [91, 125] and combines PatchMatch with EAF for cost-volume filtering. It uses superpixel segmentation to partition the reference image. Then, it defines a sub-volume for each superpixel that spans the entire cost volume as shown in the middle image of Figure 4.2. The sub-volume has its width and height defined as the superpixel boundary. Each slice is a displacement of the superpixel in the reference image to a correspondent region in the other image. The slice encodes the matching cost between the two regions.

4.5.1 Algorithm

The segmentation-based sub-volumes are filtered using an extended version of a randomized PatchMatch filter [123] that considers both dominant motions of and local deformations within superpixels. Let $G = (\mathcal{S}, \mathcal{S} \times \mathcal{S})$ denote an adjacency graph over the set of superpixels $\mathcal{S}$. In G, the nodes represent superpixels, and the edges encode the neighbourhood relationship. Two superpixels are considered neighbours if they share a boundary. For each superpixel S, the set $\mathcal{N}(S)$ denotes its neighbours. PatchMatch is an iterative technique, and each iteration consists of two steps: label propagation and random search. Graph G supports fast superpixel neighbour queries, allowing the efficient label propagation and random searches used by PatchMatch.

Our algorithm initializes PatchMatch using the ANNF computed earlier and sets up initial costs $\tilde{C}(x, y, l)$ using Equations 4.6 and 4.1. In addition, $\tilde{C}(x, y, l)$ stores the current best (minimal) costs of assigning label l to pixel (x, y). Algorithm 1 summarizes our algorithm. Notice that the aggregate function performs cost aggregation on superpixel S_i using the given label l'. This function sets up a cost slice $C(x, y, l')$ using

	CF [91]	PM [19]	PMF [123]	DM [142, 170]	SVF [80]
Com.	$O(ML)$	$O(mM\log L)$	$O(M\log L)$	$O(M^2)$	$O(M)$
Mem.	$O(M)$	$O(M)$	$O(M)$	$O(M^2)$	$O(M)$

Table 4.1: Computational (Comp.) and memory (Mem.) complexities for the cost filtering (CF) [91], PatchMatch (PM) [19], PatchMatch filter (PMF) [123], Deep Matching (DM) [142, 170], and our cost-sub-volume filtering (SVF) method [80].

Equation 4.6, where pixel $(x,y) \in B_i$, and filters it to compute $C'(x,y,l')$ using Equation 4.1. For each pixel $(x,y) \in B_i$, the function updates its (current best) label to l' if $C'(x,y,l') < \tilde{C}(x,y,l)$.

4.5.2 Complexity Analysis

The complexity of our PatchMatch method is linear $O(M)$ in space and time (see Table 4.1). Let M be the size of the input image I, L be the size of the label space, and $\hat{R} = \sum_{i=1}^{K}|R_i|$ denote the total size of the padded superpixels. For cost aggregation, we use linear-time EAF methods, which have a runtime that is independent of the kernel window size $m = (2r+1)^2$. The complexity of the extended PatchMatch filter algorithm is $O(M + \hat{R}\log 2\gamma_s)$, where $2\gamma_s$ is the search label range of the largest sub-volume. The $O(M)$ term accounts for the initial optical-flow computations using ANNF [18] and the construction of sub-volumes. Specifically, this cost is $\left(\sum_{i=0}^{n-1}\frac{M}{2^i}\right)$, where n equals the number of levels used in coarse-to-fine sub-volume construction. The last term $O(\hat{R}\log 2\gamma_s)$ represents the complexity of our extended PatchMatch filter. As clarified by [123], $O(\hat{R}\log 2\gamma_s) = O(M\log 2\gamma_s)$ because the difference between $\hat{R}$ and M is just a small leading constant. Moreover, γ_s is usually small $\gamma_s \ll L$ and does not depend on L. This makes $O(M\log 2\gamma_s) = O(M)$, as $\log 2\gamma_s$ becomes a leading constant. This analysis shows that the complexity of our algorithm is $O(M)$ and does not depend on L, compared to the original formulation of the PatchMatch filter [123] with approximately $O(M\log L)$ complexity (see Table 4.1).

The memory complexity of our algorithm is $O(2nK + M)$. The first term $O(2nK)$

Algorithm 1 PatchMatch for Sub-volume Filtering

```
Require: $\tilde{C}(x,y,l), \mathcal{S}, \{\mathcal{V}^s\}_{s=1}^n$
Ensure: Updated $\tilde{C}(x,y,l)$
 1: /* Propagating Local Deformations */
 2: for $i = 0$ to $|\mathcal{S}|$ do
 3:     for all $V \in \{V_i^s\}_{s=1}^n$ do
 4:         Pick a random label $l' \in V$
 5:         Aggregate($\tilde{C}, i, l'$)
 6:     end for
 7: end for
 8: /* Propagating motions across neighbours */
 9: for $i = 0$ to $|\mathcal{S}|$ do
10:     for all $S_j \in \mathcal{N}(S_i)$ do
11:         Pick random$(x,y) \in B_j$with best label $l^*$
12:         Aggregate($\tilde{C}, i, l^*$)
13:     end for
14: end for
15: /* Random search */
16: for $i = 0$ to $|\mathcal{S}|$ do
17:     Pick a random $(x,y)$ from $S_i$ with label $l^*$
18:     Select $V \in \{V_i^s\}_{s=1}^n$ such that $l^* \in V$
19:     For all $l' \in V$at exponentially decreasing
         distance from $l^*$ do
20:     Aggregate($\tilde{C}, i, l^*$)
21:     end for
22: end for
```

is used to store the two sets $\{\Omega_\mu^s\}_{s=1}^n$ and $\{\Omega_{\sigma^2}^s\}_{s=1}^n$. The last term $O(M)$ holds the aggregated cost of each pixel. Because $nK \ll M$, $O(2nK + M) = O(M)$.

4.6 Occlusion Handling and Gap Filling

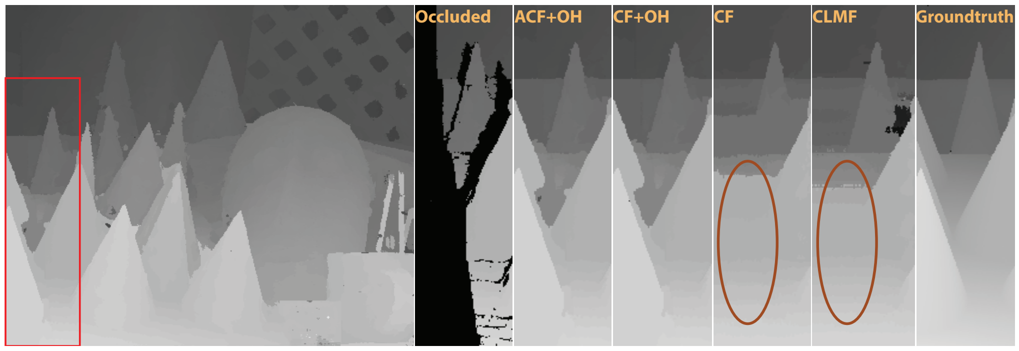
Although the restriction of cost filtering and aggregation to sub-volumes reduce the overall runtime, they add a small noise to the output solution. These noisy regions are detected and result in unlabelled pixels or gaps (see Figure 4.3). The gaps also result from incorrect labels coming from parallax effects and non-overlapped regions in the input images. To detect gaps, we follow the *left-right cross-checking* approach used by [91], where we compute two output labelling solution maps D_1 and D_2, constructed by taking each image of the input pair of images as a reference image. We detect pixel (x,y) as an incorrect label, if it has different values in the solution maps D_1 and D_2. For the next discussion, we will refer to unlabelled and labelled pixels as occluded and non-occluded pixels, respectively.

We develop a gap-filling algorithm that utilizes the previously detected superpixels. It relies upon the following observations. (1) The desired optimal labelling solution should be spatially smooth and maintain the intensity changes of image edges. (2) Compact superpixels align well with image boundaries and have a high likelihood that their neighbouring pixels are assigned similar labels. (3) Using superpixels as the basic units for computations improves performance. Given a set of superpixels $\mathcal{S} = \{S_1, S_2, S_3, \cdots, S_K\}$ resulting from segmenting image I into K compact superpixels, we indicate pixels (x,y) inside the superpixel S by $(x,y) \in S$, and $(x,y) \in [1, I_{\text{width}}] \times [1, I_{\text{height}}]$, for I_{width} and I_{height} representing the width and height of image I.

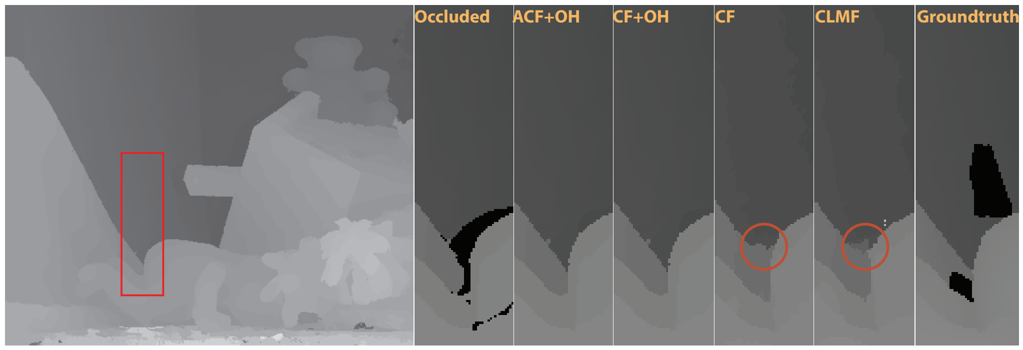
Our gap-filling method starts by calculating for each superpixel $S \in \mathcal{S}$, an occlusion probability $p_{\text{occ}}(S)$, which is defined as:

$$p_{\text{occ}}(S) = \frac{\sum_{(x,y)\in S} Occ(x,y)}{|S|}, \tag{4.4}$$

where $Occ(x,y) = 1$ for occluded pixels and is zero for non-occluded pixels. We also call a superpixel non-occluded if $p_{\text{occ}}(S)$ is zero or all its pixels have consistent labels. $|S|$ indicates the number of inner pixels of $S \in \mathcal{S}$. This divides superpixels into two sets: occluded $\mathcal{S}_{\text{occ}}$ and non-occluded $\mathcal{S}_{\text{nocc}}$, where $\mathcal{S} = \mathcal{S}_{\text{occ}} \cup \mathcal{S}_{\text{nocc}}$. The function $h(S)$ is defined to return the most frequent label $D(x,y)$ for pixels $(x,y) \in S$, when $p_{\text{occ}}(S) < 1$. Given a user-defined threshold τ_{fill}, we set the occluded pixels $(x,y) \in S$ to $h(S)$ if $p_{\text{occ}}(S) < \tau_{\text{fill}}$. After that, the occluded superpixels $\forall S \in \mathcal{S}_{\text{occ}}$ have $p_{\text{occ}}(S) \geq \tau_{\text{fill}}$. In our experiments, we set τ_{fill} to either 0.5 or 0.6.



(a) Comparison on the Cones dataset.



(b) Comparison on the Teddy dataset.

Figure 4.5: Comparison of our accelerated cost filtering (ACF) method before and after using our occlusion handling (OH) technique on the Teddy and Cones Middlebury stereo datasets. We also show a visual comparison between the ACF+OH method against the results of cost filtering (CF), cost filtering and occlusion handling (CF+OH), and cross-based local multipoint filtering (CLMF). The left images show the resulting disparity maps of ACH+OH. In each image, a rectangular region is highlighted, and a close-up of this region is shown in the right columns. The occluded column shows the resulting occluded regions of ACF before post-processing using OH. The ACH+OH column shows the results after post-processing. The CF+OH shows the results after post-processing CF output using OH. The next two columns show the results of CF and CLMF, respectively. The last column shows the ground truth. Red ellipses and circles highlight areas of large errors.

Label Propagation via Simulated Annealing

Given that compact superpixels contain pixels with similar appearance and are most likely to share similar labels, label propagation is performed between superpixels under this local smoothness assumption. We start by defining an adjacency graph $G = (\mathcal{S}, \mathcal{S}\times\mathcal{S})$ that has superpixels as nodes and edges between neighbouring superpixels. Two superpixels are neighbours if they share a common boundary. We denote the set of neighbouring non-occluded superpixels for a given superpixel S by $N_{\text{nocc}}(S)$. We also define the similarity function $sim(S, S') = 1 - \|col(S) - col(S')\|_2 \in [0,1]$ that measures the similarity between any two superpixels. The function $col(S)$ returns the normalized average colour for a superpixel S. Our label propagation algorithm works similarly to the *simulated annealing* (SA) method [68].

SA is a popular iterative metaheuristic that approximates global optimization by finding a global minimum for a given energy function. Given a physical system at state s, each iteration of SA considers moving the system from the state s to a candidate neighbouring state s'. SA makes the transition according to an acceptance probability $p_a(E(s), E(s'), T)$, where T is a parameter called the temperature, and $E(s)$ and $E(s')$ are the energies of the states s and s', respectively. This probability equals 1 if $E(s') < E(s)$ to favour downhill moves, and decreases if the difference $\|E(s) - E(s')\|_2$ is high. Also, at high T, the small differences in energy are ignored to quickly search the space of states for a good minimum. As T is lowered, the algorithm becomes sensitive to small changes in energy to search for the best minimum in the neighbourhood of the found minimum.

Our algorithm starts with an initial state s that contains the two sets $\mathcal{S}_{occ}$ and $\mathcal{S}_{nocc}$ representing the list of occluded and non-occluded superpixels respectively. A new state s' is created by filling a superpixel $S \in \mathcal{S}_{occ}$ and creating two new sets $\mathcal{S}'_{nocc} = \mathcal{S}_{nocc} \cup \{S\}$ and $\mathcal{S}'_{occ} = \mathcal{S}_{occc} - \{S\}$. In our final solution state, we need to make sure that each superpixel $S \in \mathcal{S}_{occ}$ is filled from its best similar neighbour. So, for superpixel $S \in \mathcal{S}_{occ}$, we define our acceptance probability as,

$$p_a(S) = \begin{cases} 1 & \text{if } \max\limits_{S' \in N_{\text{nocc}}(S)} sim(S, S') > T \\ 0 & Otherwise. \end{cases}, \tag{4.5}$$

We define T as a similarity threshold and accept a transition for a certain superpixel $S \in \mathcal{S}_{occ}$ if $sim(S, S') > T$ and S' is the best similar non-occluded neighbouring superpixel. We then fill S using $h(S')$. So the algorithm propagates labels from non-occluded superpixels to their neighbouring occluded superpixels while slowly reducing T by an amount ΔT over time. At high T, we encourage propagating labels between the most similar neighbouring superpixels. As T is lowered, the remaining occluded superpixels will have more filled candidates around them for selecting the best similar neighbour. We never update the non-occluded superpixels. After filling all superpixels (see Figure 4.5), we perform weighted-median filtering as a final refinement step, as in [91]. The pseudocode of our label propagation algorithm is described using the following Algorithm 2.

Algorithm 2 Label Propagation via Simulated Annealing

```
Require: $\mathcal{S}_{occc}, \mathcal{S}_{nocc}, T, \Delta T$
Ensure: $\mathcal{S}_{occ} = \Phi$ and $\mathcal{S} = \mathcal{S}_{nocc}$
 1: $T = 1.0$;
 2: while $\mathcal{S}_{occ} \neq \Phi$ do
 3:     Pick an $S \in \mathcal{S}_{occ}$
 4:     if $N_{\text{nocc}}(S) = \Phi$ then
 5:         continue
 6:     end if
 7:     $S^* = \arg\max\limits_{S} sim(S, S)$, where $S \in N_{\text{nocc}}(S)$
 8:     if $sim(S, S^*) > T$ then
 9:         $\forall (x, y) \in S$, if $Occ(x, y)$ then $D(x, y) = h(S^*)$
10:         $\mathcal{S}_{nocc} = \mathcal{S}_{nocc} \cup \{S\}$
11:         $\mathcal{S}_{occ} = \mathcal{S}_{occ} - \{S\}$
12:     end if
13:     $T = \max(T - \Delta T, 0.0)$
14: end while
```

4.7 Applications

In this section, we apply our sub-volume-filtering framework for the two studied applications of pixel-labelling problems, stereo-vision and optical-flow estimation. In both applications, the input is an image pair (I_1, I_2), and one of the images is identified as the reference image I_r. If we set $I_\mathrm{r} = I_1$, then our task is to assign a label $l = (u, v)$ that corresponds to a displacement in the x and y direction of a pixel $(x, y) \in I_1$ to a pixel $(x + u, y + v) \in I_2$.

4.7.1 Stereo Vision

In the stereo case, the label $l = (u, v)$ defines the displacement in only the x direction with the vertical displacement $v = 0$. The u displacement corresponds to the disparity d with $u = d$.

Cost Computation

A matching cost is computed for a displacement vector $l = (u, v)$ between a pixel $(x, y) \in I_\mathrm{r}$ to a pixel $(x + u, y + v) \in I_k$. We use the truncated absolute difference of the colour and gradient, which was employed by [91, 18, 122] and shown robustness to changes in illumination,

$$C(x, y, l) = (1 - \beta) \min(d_1, \gamma_1) + \beta \min(d_2, \gamma_2), \tag{4.6}$$

where $d_1 = |I_\mathrm{r}(x, y) I_k(x_k, y_k)|$ and $d_2 = |\nabla_x I_\mathrm{r}(x, y) - \nabla_x I_k(x_k, y_k)||$. In addition, β, γ_1, and γ_2 are user-defined parameters, and $0 \leq \beta \leq 1$. Given I_r, we displace I_k one pixel at a time with respect to I_r and calculate a cost slice for each displacement using Equation 4.6. This is done by considering the small disparity range and rectification of images.

Later, we filter costs using Equation 4.1 that employs the guided filter, which has a linear-time complexity and does not depend on the kernel size. Finally, a solution map for I_r is calculated using the winner-takes-all strategy of Equation 4.2.

Occlusion Detection and Filling

We calculate two solution maps (D_1, D_2) for the input image pair (I_1, I_2). The D_2 map is calculated by taking $I_\text{r} = I_2$. We detect the incorrect labels caused by occlusions using the *left-right cross-checking* approach [56, 91, 18, 123] and apply our gap-filling method discussed in Section 4.6 to handle the occluded areas.

Post-processing

Our method assigns the invalid pixels in each occluded superpixel to the most frequent label of its most similar neighbouring superpixel. This creates little artifacts in the output disparity map. To handle these artifacts, we apply a weighted-median filter to the output disparity map and only update the occluded pixels with their weighted labels. For weighting, [91] showed that the weights of the bilateral filter [136] are well suited for stereo matching. These weights are given by:

$$W_{I,j}^{bf} = \frac{1}{K_i} \exp\left(-\frac{|i-j|^2}{\sigma_s^2}\right) \exp\left(-\frac{|I_i - I_j|^2}{\sigma_c^2}\right), \tag{4.7}$$

$$D'(i) = \sum_j W_{i,j}^{bf} D(j), \tag{4.8}$$

where the kernel weights $W_{i,j}^{bf} \in \mathbb{R}^{(2q+1)\times(2q+1)}$ for window ω_i are centred on pixel location $i = (x, y)$. In addition, σ_s^2 and σ_c^2 are smoothing parameters that control the spatial and colour similarity, respectively. Moreover, K_i is a normalization factor, and q is the kernel radius, set to 13 in all our experiments.

4.7.2 Optical Flow

Optical flow is very similar to stereo vision with the flow vectors $l = (u, v)$, which define displacements in both the x and y directions.

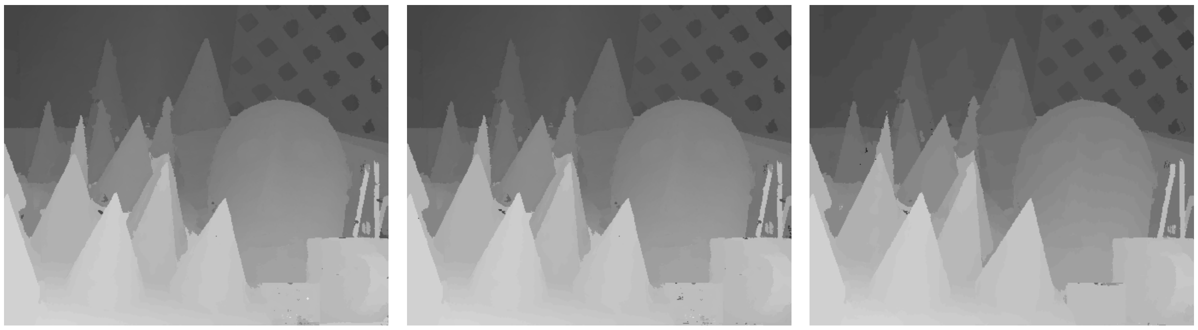

(a) ACF+OH. Err. 7.43% (b) CF+OH 7.41% (c) CLMF+OH. Err. 7.22%

Figure 4.6: The advantage of using our OH method as a post-processing step: (a) using OH improves the all pixel errors of accelerated cost filtering (ACF) method ($r = 0.3$) [79] from 8.49% to 7.43%; (b) using OH improves the all pixel errors of cost filtering (CF) [91] from 8.24% to 7.41%; (c) using OH improves the all pixel errors of CLMF [121] from 7.82% to 7.22%. All results are calculated on the Cones Middlebury stereo dataset with the error threshold = 1.

Cost Computation

The matching costs are computed similar to stereo vision, adding an extra term in Equation 4.6 for matching the gradient in the y direction. Thus, we set $d_2 = |\nabla_x I_{\mathrm{r}}(x,y) - \nabla_x I_k(x_k,y_k)| + |\nabla_y I_{\mathrm{r}}(x,y) - \nabla_y I_k(x_k,y_k)|$. As with stereo, the cost volume is filtered using Equation 4.1, and an initial labelling is obtained using the winner-takes-all strategy of Equation 4.2.

Occlusion Detection and Filling

As with stereo, we apply our gap-filling algorithm to assign labels to the occluded pixels detected using the *left-right cross-checking* approach. However, we apply the fast weighted-median-filtering algorithm presented in [186]. This algorithm approximates the weighted-mean-filtering algorithm described in the previous section by developing a set of fast data structures that have $O(q)$ complexity to find the weighted median. We also apply a second round of forward-backward consistency check to fix any inconsistent label assignments [18].

Subpixel Accuracy

For optical flow, subpixel accuracy is often required to enhance the resulting flow map. We calculate subpixel-accurate optical flow by upsampling the input images. Similar to the scheme presented by [91, 123], our method upsamples the input images by a factor of 8 using bicubic interpolation. This only increases the label space size and causes a small runtime performance decrease since matching costs are computed at the same resolution as the input images.

4.8 Experimental Results

4.8.1 Stereo Vision

Datasets

We perform our experiments on the four standard resolution datasets (Teddy, Cones, Venus, and Taskuba) of the Middlebury stereo benchmark [147] and on the five (Rocks 1, Rocks 2, Moebius, Dolls, and Books) high-resolution Middlebury 2005/2006 datasets [87]. All datasets have complex geometry, textureless regions, and parallax effects resulting from the motion of objects. The datasets have all images rectified with radial distortion removed. The standard datasets have an average of 450×375 pixels, whereas the high-resolution datasets have an average of 1280×1110 pixels. The standard datasets have ground-truth disparity maps with an encoded disparity range of 0.25 to 63.75 pixels. The high-resolution datasets have ground-truth disparity maps with a pixel-accurate disparity range of 200 or 230 starting from 1, and 0 indicates unknown disparity. Notice that testing on both standard and high-resolution datasets provides a more robust performance comparison. While the Middlebury stereo benchmark provides images with a small disparity range, the Middlebury 2005/2006 datasets provide a large disparity range that highlights the speedup gains of ACF.

Results

We perform experiments using our ACF method discussed in Section 4.3.1. When we perform occlusion handling (OH) using our gap-filling method discussed in Section 4.6, we refer to the method as accelerated cost filtering with occlusion handling (ACF+OH). We compare ACF+OH on the Middlebury stereo benchmark standard datasets [147] against cross-based local multipoint filtering (CLMF) [121], variational Mumford-Shah regularization with occlusion handling (VarMSOH) [23], and cost filtering (CF) [91]. Notice that we focus our comparison on local cost-volume filtering methods. The CF method provided a simple and efficient local filtering framework that applied edge-aware filtering on the entire cost volume. It influenced the development of several other methods [129, 184]. CLMF process the entire cost volume by applying local multipoint filtering. VarMSOH is also included in the comparison as a global energy minimization method that applies variational regularization and occlusion handling.

For the high-resolution Middlebury 2005/2006 datasets [87], we compare ACF+OH against CF [91]. For OH, VarMSOH uses a global energy minimization approach, whereas CF and CLMF use the row filling (RF) method by [91], and we refer to them as CF+RF and CLMF+RF.

For algorithmic parameters, ACF has an expansion factor u set to 6 for the standard datasets, and 2 for the high-resolution datasets. We use two values for the window radius $r = 0.2$ and $r = 0.3$. The number of superpixels K is set as in Table 4.6. For OH, ΔT is set to 0.0001 for all datasets, and τ_{fill} is set as indicated in Table 4.6. Later in the discussions, we will illustrate our algorithm sensitivity to the choice of parameter values.

The ACF method is implemented in C++, and all experiments were carried out on a single-core 2.8GHz CPU. On standard size images, SIFT computation takes 0.17 seconds, ACF filtering computation takes 14.39 seconds for $r = 0.3$, and post-processing using OH takes 0.131 seconds. On average, ACF takes around 14.69 seconds on Middlebury standard resolution images. On high-resolution images, ACF takes 0.82 seconds for SIFT

Table 4.2: Quantitative evaluation on Middlebury benchmark datasets [147]. These results are aggregated over Cones, Teddy, Venus, and Tsukuba datasets.

Algorithm	Error threshold = 1		Error threshold = 0.5	
	Rank	% error	Rank	% error
CF+OH	**25**	5.22	30	12.9
CLMF+OH	38	5.14	66	16.9
ACF+OH ($r = 0.3$) [79]	30	5.26	33	13
ACF+OH ($r = 0.2$) [79]	39	5.45	37	13.3
CF+RF [91]	42	5.55	27	12.8
CLMF+RF [121]	37	**5.13**	64	16.7
ACF+RF($r = 0.3$) [79]	64	5.99	45	13.4
ACF+RF($r = 0.2$) [79]	60	5.92	42	13.6
VarMSOH [23]	116	8.17	**21**	**11.8**

computation, 156.82 seconds for cost filtering, and 0.2 seconds for OH. On average, ACF takes around 157.84 seconds on Middlebury high-resolution images.

Table 4.3: Stereo evaluation results on Middlebury benchmark with error threshold equal to 1.0.

Algorithm	Tsukuba			Venus			Teddy			Cones		
	nocc	all	disk	nocc	all	disk	nocc	all	disc	nocc	all	disk
CF+OH	**1.45**	**1.75**	7.37	**0.19**	**0.37**	2.24	5.85	**10**	16.1	2.6	7.41	7.31
CLMF+OH	2.39	2.69	6.53	0.26	**0.37**	2.23	**5.49**	10.7	**14.2**	2.46	**7.22**	7.10
ACF+OH ($r = 0.3$) [79]	1.45	1.75	7.37	0.19	0.37	2.24	5.94	10.1	16.4	2.61	7.43	7.23
ACF+OH ($r = 0.2$) [79]	1.45	1.75	7.37	0.19	0.37	2.24	6.64	10.7	16.3	2.82	7.79	7.74
CF+RF [91]	1.51	1.85	7.61	0.2	0.39	2.42	6.16	11.8	16	2.71	8.24	7.66
CLMF+RF [121]	2.46	2.78	**6.26**	0.27	0.38	**2.15**	5.50	10.6	**14.2**	**2.34**	7.82	**6.80**
ACF+RF ($r = 0.3$) [79]	1.51	1.85	7.61	0.2	0.39	2.42	6.94	11.3	18.5	3.38	8.49	9.3
ACF+RF ($r = 0.2$) [79]	1.51	1.85	7.61	0.2	0.39	2.42	6.96	11.1	17.1	3.66	9.06	9.8
VarMSOH [23]	3.97	5.23	14.9	0.28	0.76	3.78	9.34	14.3	20	4.14	9.91	11.4

Filtering Time

Table 4.5 provides a comparison of the average filtering time and the average percentage of the occluded pixels between ACF and CF, on both the standard and high-resolution Middlebury datasets. Notice that we do not apply the post-processing steps. We can see that ACF and CF have a comparable accuracy; however, ACF has a significantly faster filtering time. The average filtering time on standard datasets is 18.717 and 28.2 seconds for ACF(r=0.3) and CF, respectively; whereas for high-resolution datasets, these numbers are 159.82 and 505 seconds, respectively. Moreover, the average percentage of occluded pixels is comparable on both standard and high-resolution datasets. The table also shows that the runtime of the row-filling and the occlusion handling (OH) post-processing algorithms are 0.11 and 0.131 seconds, respectively, on standard datasets; whereas, for the high-resolution dataset, these numbers are 1.4 and 0.2 seconds, respectively. This indicates that the OH method has significantly faster runtime than the traditional row-filling (RF) method on high-resolution images. This is expected because OH depends on the number of superpixels, which is much smaller than the number of image pixels.

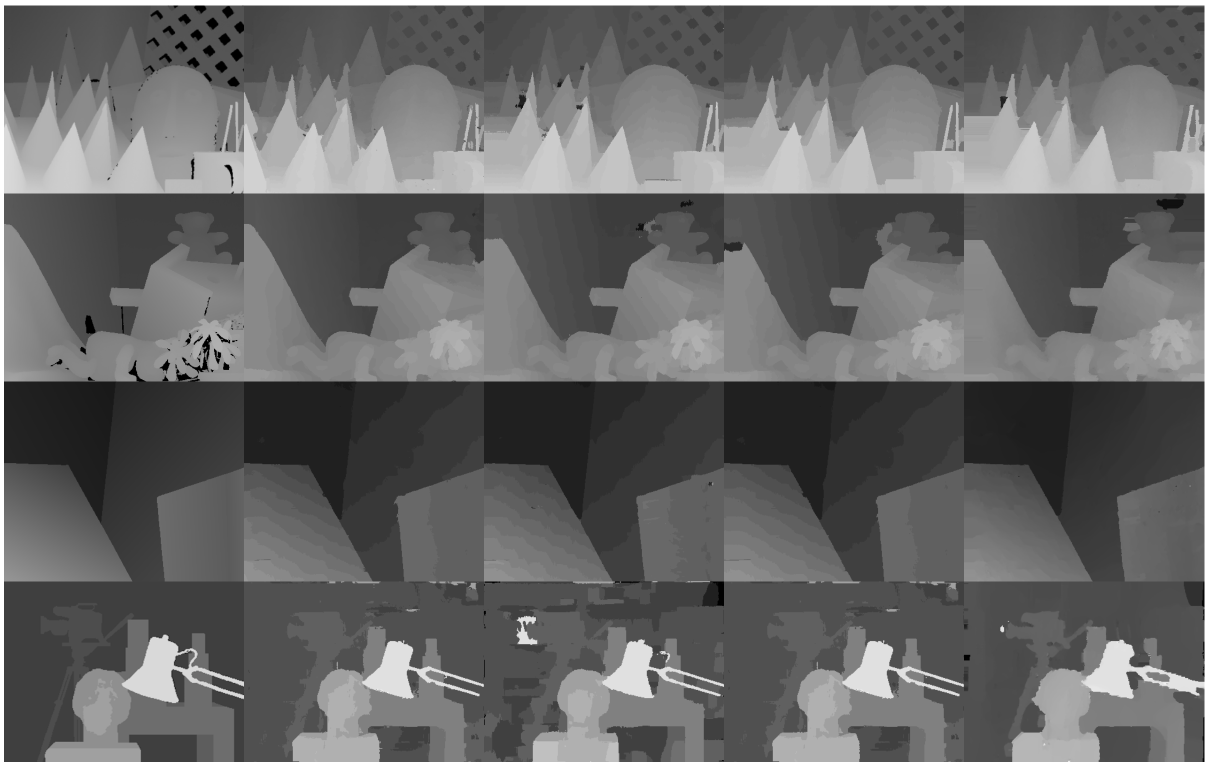

(p) Ground Truth (q) ACF+OH (r) CLMF+RF (s) CF+RF (t) VarMSOH

Figure 4.7: Visually comparing the output disparity maps between accelerated cost filtering with occlusion handling (ACF+OH) [79] and cross-based local multipoint filtering with row filling (CLMF+RF) [121], cost filtering with row filling (CF+RF) [91], and variational Mumford-Shah regularization with occlusion handling (VarMSOH) [23], on the Teddy, Cones, Venus, and Tsukuba datasets of the Middlebury stereo benchmark [147].

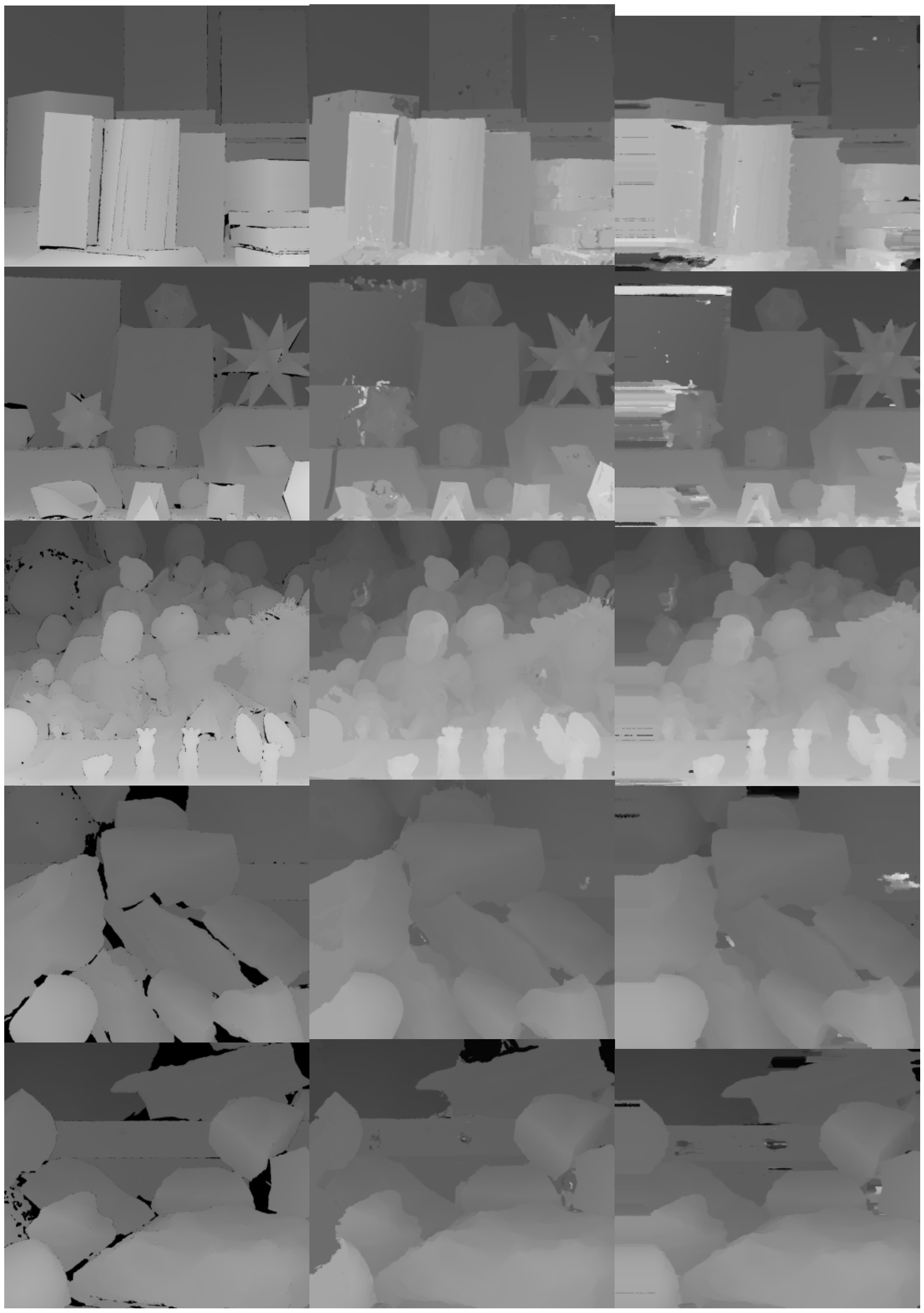


Figure 4.8: Visually comparing the output disparity maps between accelerated cost filtering with occlusion handling (ACF+OH) [79] and cost filtering with row filling (CF+RF) [91], on the Books, Moebius, Dolls, Rocks 1, and Rocks 2 high-resolution Middlebury 2005/2006 datasets [87].

Middlebury Standard Benchmark

Figure 4.7 shows a visual comparison between our accelerated cost filtering with occlusion handling (ACF+OH) method [79] and cross-based local multipoint filtering with row filling (CLMF+RF) [121], cost filtering with row filling (CF+RF) [91], and variational Mumford-Shah regularization with occlusion handling (VarMSOH) [23], on all Middlebury standard resolution datasets. Our OH method consistently fills occluded areas, whereas the traditional RF method causes large distortions.

Table 4.2 shows an accuracy comparison on the Middlebury standard benchmark. The comparison lists the quantitative evaluation results and shows the rank and average percentage of bad pixels (errors) for all compared methods. We list results for two error thresholds: 1, which is the default threshold, and 0.5. The results show that our method has a higher rank over VarMSOH, CF+RF, and CLMF+RF on the error threshold 1. However, CF+RF and VarMSOH have a slightly better performance on the 0.5 threshold. This is because our ACF method does not support subpixel accuracy through slanted planes that give a larger precision by assuming a continuous range of disparities. This requires us to deal with a continuous range of labels, which ACF cannot handle in a feasible time, as it filters sub-volumes one slice at a time. Our segmentation-based subvolume method overcomes this problem, as we will see later in the experiments presented in Section 4.8.2.

Table 4.2 also presents results for our OH method. We notice that ACF+RF, which fills the gaps of ACF using the RF method, has a lower rank of 60 and 64 at the default threshold for $r = 0.2$ and $r = 0.3$, whereas, ACF+OH has a better rank of 30 and 39, respectively. Even using our OH method with the cost filter (CF+OH) improves the rank to 25 over CF+RF [91], which has the rank 42 (Figure 4.6). We can also see a similar accuracy for CLMF+OH and CLMF+RF.

Table 4.3 presents the benchmark results on the four Middlebury standard datasets under the default error threshold. The table lists three values for each dataset representing the following errors: (1) nocc error, which measures the average percentage of incorrect disparities on all non-occluded pixels, (2) all error, which defines the same error as nocc but on all image pixels, and (3) disc error, which provides the same error but on pixels near depth discontinuities. We can see that ACF+OH with $r = 0.3$ and $r = 0.2$ outperformed CF+RF and CLMF+RF on the all error and nearly on every value of the nocc and disc errors. Our method also outperforms VarMSOH on all errors. For OH, our approach boosts the accuracy of ACF over that of the RF method. For example, The Teddy dataset has an all error of 10.1% and 10.7% for ACF+OH ($r = 0.3$) and ACF+OH ($r = 0.2$), compared to 11.3% and 11.1% for ACF+RF, respectively. Moreover, OH boosts accuracy when it is used as a post-processing step for CF and CLMF. For CF, OH reduces every error measure, whereas, for CLMF, OH reduces nocc and all errors.

Table 4.4: Performance comparison between accelerated cost filtering with occlusion handling (ACF+OH) and cost filtering with row filling (CF+RF) on the Middlebury 2005/2006 high-resolution datasets. The reported runtime is for cost-volume filtering.

Dataset	ACF+OH		CF+RF	
	time (sec)	% error	time (sec)	% error
Rocks2	126.21	5.4	471.4	6.95
Books	152.22	16.1	569.22	16.4
Dolls	188.41	14.2	529.84	13.2
Moebius	152.3	10.94	526.38	12.78
Rocks1	180.88	6.2	435.35	6.31
Average	160	10.57	506.44	11.13

Middlebury 2005/2006 Datasets

Figure 4.8 shows a visual comparison between ACF+OH and CF+RF, on the Moebius, Dolls, Rocks1, Rocks 2, and Books high-resolution datasets. ACF+OH provides better results than CF+RF. For example, the first row from top shows results of the Books dataset. CF+RF has very high distortions compared to ACF+OH. The OH method also performs a better job than RF in handling occluded areas. The third row from the top

Table 4.5: Comparison of the average filtering time and the average percentage of occluded pixels between accelerated cost filtering (ACF) [79] and cost filtering (CF) [91] without any post-processing steps on Middlebury standard and high-resolution datasets. Runtimes for row filling (RF) and occlusion handling (OH) post-processing steps are also provided.

Algorithm	Average % occluded pixels		Runtime (seconds)	
	Standard	High Resolution	Standard	High Resolution
ACF(r=0.2)	14.2	-	16.117	-
ACF(r=0.3)	14.39	26.1	18.717	159.82
CF	13.6	26.9	28.2	505
RF	-	-	0.11	1.4
OH	-	-	0.131	0.2

shows the results of the Dolls dataset. It is clear that most of the fine details in the Dolls dataset are preserved in the ACF+OH result, whereas several details are lost in CF+RF result. Similar results are obtained for other datasets.

Table 4.4 shows the results for the Middlebury 2005/2006 high-resolution datasets. The accelerated cost filtering with occlusion handling (ACF+OH) method has an average error of 10.57% over all datasets, whereas cost filtering with row filling (CF+RF) achieves 11.13%. We can also see that our ACF+OH method achieves up to 4 times the speed increase over CF+RF. These results confirm our main contribution that we can restrict filtering within the cost volume to a small set of sub-volumes, while achieving similar (or even better, in some cases) accuracy to filtering the entire cost volume.

Table 4.6: Parameters for occlusion handling (OH) procedure. $\Delta T = 0.0001$.

Dataset	#superpixels	τ_{fill}	Dataset	#superpixels	τ_{fill}	Dataset	#superpixels	τ_{fill}
Cones	1600	0.6	Teddy	2000	0.6	Tsukuba	500	0.5
Venus	1000	0.5	Rocks1	700	0.5	Rocks2	700	0.5
Moebius	1600	0.5	Dolls	1600	0.5	Books	1600	0.5

Sensitivity Analysis

To study how the algorithmic parameters affect the runtime and accuracy of our ACF+OH method, we performed several sensitivity analysis experiments. Our first experiment is

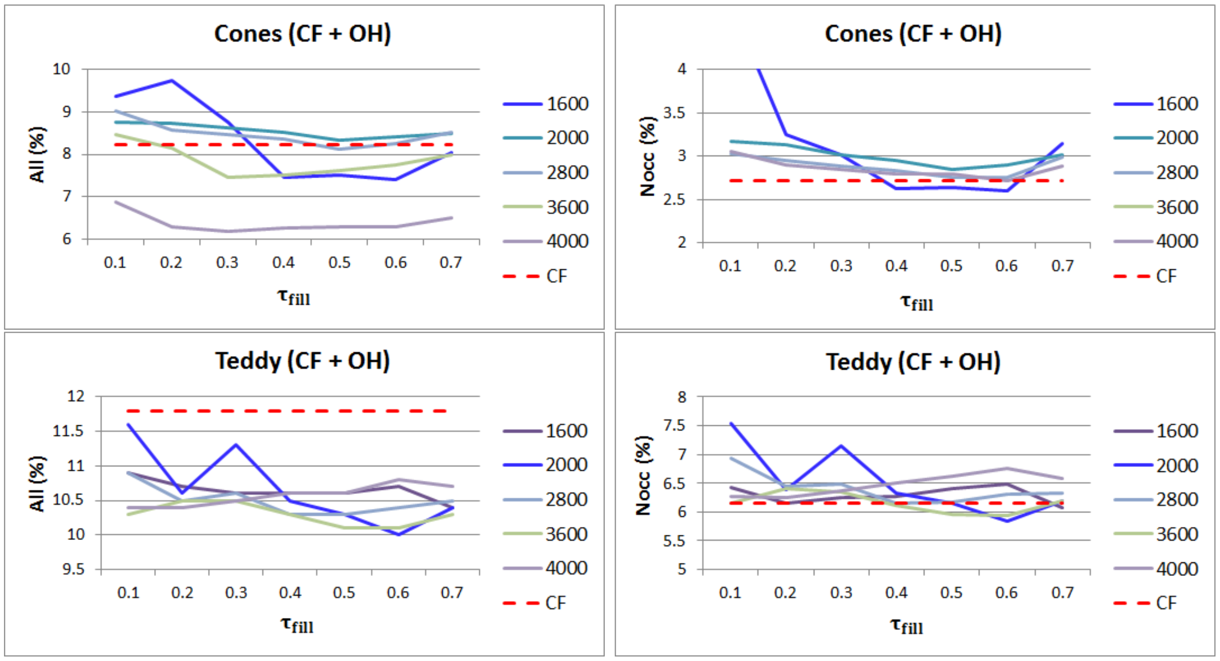


Figure 4.9: Accuracy of cost filtering with row filling (CF+RF) and cost filtering with occlusion handling (CF+OH) against τ_{fill} threshold for the OH method. The dashed red line indicates the accuracy of CF+RF. It is independent of the choice of τ_{fill}. Solid lines indicate the percentage of all (left column) and nocc (right column) errors for different values of the number of superpixels K. The top row shows plots for the Cones dataset, and the bottom row shows plots for the Teddy dataset. *This figure is best viewed in colour.*

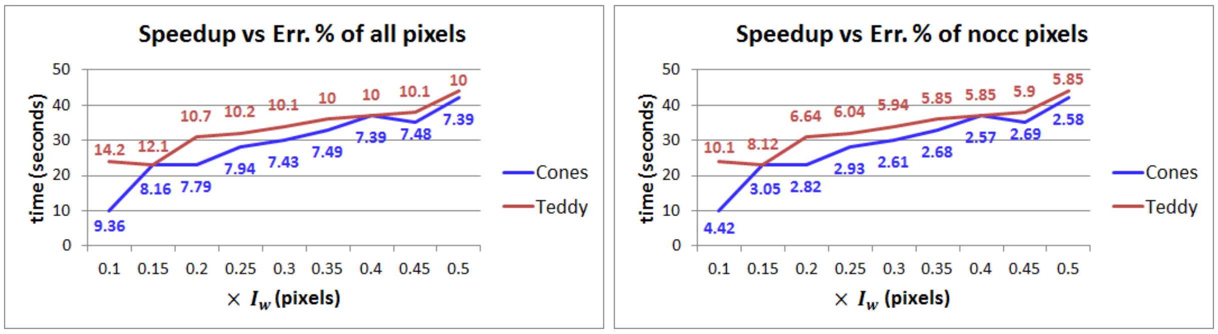


Figure 4.10: Runtime versus accuracy comparison for accelerated cost filtering with occlusion handling (ACF+OH) using different values for the r parameter that controls the size of local window used for defining salient regions. The numbers printed next to the plots represent average percentage errors. *This figure is best viewed in colour.*

shown in Figure 4.10 and presents a time-accuracy trade-off evaluation for ACF+OH while changing the parameter r that defines the local window radius used for constructing salient areas. Note that the local window radius is defined as $r \times I_{\text{width}}$. The x-axis in Figure 4.10 is the local window size defined by r, while the y-axis is the runtime in seconds. Notice that the accuracy increases for large r values, which widens the salient regions. This is expected, as we process more area of the cost volume; however, the accuracy has an unnoticeable variation after $r = 0.3$. This important observation confirms our main contribution that we can restrict filtering to a small set of sub-volumes inside the cost volume while obtaining similar accuracy to filtering the entire volume.

For OH, Figure 4.9 shows plots for the all and nocc errors of the CF+RF and CF+OH methods against a range of values for the τ_{fill} parameter. Each plot also shows several error graphs corresponding to different values of the number of superpixels K. The top row shows the accuracy results for the Cones dataset and the bottom row shows the same results for the Teddy dataset. The plots show that the OH method improves accuracy over a range of values for τ_{fill} and K. This highlights the robustness of our OH method against the selection of parameters.

4.8.2 Optical Flow

Datasets

We evaluate our SVF method on three standard optical-flow benchmarks: (1) MPI Sintel benchmark [40]; (2) Middlebury benchmark [17], and 3) KITTI benchmark [65]. We also evaluate the runtime performance of the proposed method on a 2880×1620 high-resolution image [167] (we assume zero motion and focus only on processing times).

Results

We will show that ACF+OH becomes infeasible for processing very large label spaces, typically found in optical flow. Therefore, we performed optical-flow experiments using our SVF method. When we perform occlusion handling (OH) using our gap-filling method discussed in Section 4.6, we refer to the method as cost-sub-volume filtering with occlusion handling (SVF+OH).

We employ the guided filter proposed in [74, 122] for EAF. The parameter settings for the guided filter are taken from [74, 122], as follows: $\sigma_r = 0.1$, $\beta = 0.9$, $\gamma_1 = 0.039$, $\gamma_2 = 0.016$, and $\epsilon = 0.0001$. The kernel radius q is set to 5. The sub-volume selection parameters that control sparsity are $\beta_s = 1.9$ and $\gamma_s = 2$. We fix the number of superpixels K equal to 1,500, and the number of PatchMatch iterations is set to 7. These parameter values are fixed for all our evaluations.

The SVF method is implemented in C++ using compute unified device architecture (CUDA), and all experiments were carried out using a NVIDIA GeForce GTX 780 GPU. On standard size images, ANNF computation takes around 0.32 seconds, the preprocessing step takes about 0.4 seconds, and the sub-volume PatchMatch filtering takes 0.84 seconds. On average, the SVF takes around 1.56 seconds to process standard resolution images from the Middlebury benchmark. The SVF method also scales well to medium-resolution images, taking on average 1.95 and 2 seconds to compute optical flow from the MPI Sintel and KITTI benchmarks, respectively. A single PatchMatch filtering iteration takes about 0.14 seconds on medium-resolution images.

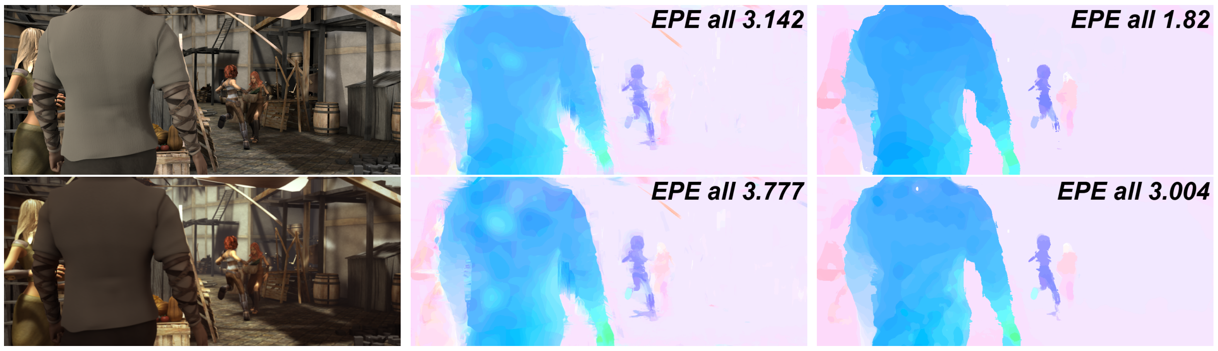


Figure 4.11: MPI Sintel market_1 sequence: (top row) clean pass and (bottom row) final pass. First column shows a pair of images from the sequence. The second column shows initial optical flow computed by [18]. The last column shows optical flow computed using our method. Our method improves the quality of optical flow computed by [18]. End-point error (EPE) *all* values are shown on the images.

Algorithm	Middlebury-Standard(640x480)			Sintel-Medium(1024x463)			Cat-High(2880x1620)		
	CPU	GPU	Multi-GPUs	CPU	GPU	Multi-GPUs	CPU	GPU	Multi-GPUs
SVF [80]	9.38	0.84	0.38	10.6	0.98	0.45	98.4	5.53	2.4
PMF [122]	35	3.4	1.38	44	4.2	1.7	437.5	29	11.23
CF [91]	59526.9	750	251	-	-	-	-	-	-
ACF [79]	35015.8	410.4	137.8	-	-	-	-	-	-
SVF [80]	11	1.56	0.4	14	1.98	0.5	110	9.53	3.2
EPPM [18]	-	0.17	-	-	0.22	-	-	0.32	-
PCA-Layers [173]	10.1	-	-	15.2	-	-	-	-	-
NNF-Local [50]	1073	-	-	-	-	-	-	-	-

Table 4.7: (Top four rows) Comparing filtering times for our cost-sub-volume filtering (SVF) method [80] against that of PatchMatch filter (PMF) [122], cost filtering (CF) [91], and accelerated cost filtering (ACF) [79] schemes. We report these times on a single CPU at 2.8 GHz, a single GPU, and multiple GPUs. Our algorithm and PMF are pipelined on three GPUs; CF and ACF are parallelized on the three GPUs. (Bottom four rows) Comparing the total runtime of SVF against edge-preserving PatchMatch (EPPM) [18], optical flow based on principal component analysis and layered formulation (PCA-Layers) [173], and optical flow with nearest neighbour field (NNF-Local) [50], all runtimes are computed on the same machine except for NNF-Local that we report from the benchmarks. Our algorithm scales linearly with image resolution and does not depend upon the label space size. The PMF depends on the label space size with $O(M \log L)$ complexity.

Filtering Time

Table 4.7 compares the filtering times (using both CPU and GPU) of our SVF method against PatchMatch filter (PMF), CF, and ACF schemes. These runtimes include both cost-volume computation and aggregation. Notice that PMF randomly filters the entire cost volume, while CF performs an exhaustive search. The table also provides a comparison of the total runtime of our method against optical flow based on principal component analysis and layered formulation (PCA-Layers) [173], edge-preserving Patch-Match (EPPM) [18], and optical flow with nearest neighbour field (NNF-Local) [50]. We perform all comparisons on the same machine except for NNF-Local, which we report from the benchmarks due to the current unavailability of the code.

The filtering time comparison is performed on three different images sizes: a) 640×480 (Middlebury dataset), b) 1024×436 (MPI Sintel dataset), and c) 2880×1620 [167]. For PMF, CF, and ACF schemes, the label space size is 410,000 for 640×480 (see

PMF [122]). The label space sizes are 1,600,000 and 102,000 for 1024 × 436 and 2880 × 1620 images, respectively. Subpixel-accuracy computation is turned off for 2880 × 1620 images because of memory constraints. Notice that [122] has $O(M \log L)$ time complexity, whereas our method has $O(M)$ time complexity. Here, M represents the number of pixels (i.e., image resolution), and L is the size of the label space. Both CF and ACF post large runtimes, which renders these methods infeasible for higher-resolution images. Even ACF, which introduced sub-volume filtering, can achieve only a four-fold speed increase over CF.

Multi-GPUs Implementation

The filtering time comparison is also reported on a multi-GPU setup consisting of three GPUs. For this setup, we perform loop unrolling for parallelizing the filtering iterations of our method and PMF. In addition, CF and ACF are parallelized on the three GPUs. Table 4.7 shows a clear performance improvement of our method on this multi-GPU setup. For example, the total runtime of our method reduces to 0.5 seconds on the Sintel dataset, while using an extra GPU for preprocessing.

MPI Sintel Benchmark

The MPI Sintel benchmark contains two rendering passes: (1) a clean pass and (2) final pass (Figure 4.11). The clean pass exhibits large motions and illumination, reflectance, and shading effects, whereas the final pass adds colour correction, defocus, motion blur, and atmospheric effects to the images from the clean pass. The dataset contains 12 sequences for each pass and 564 images.

Figure 4.12 summarizes the MPI Sintel benchmark results of our SVF+OH method against other schemes (with runtimes no greater than 40 seconds per frame). The figure shows average end-point error (EPE) versus runtime on the entire image (*all*), on regions with displacements of more than 40 pixels (*s40+*), and on regions with displacements of less than 10 pixels (*s0-10*). Our method achieves accuracy that is comparable to those obtained by PMF [122] and optical flow with deep matching (DeepFlow) [170], which are two state-of-the-art methods. However, our method posts significantly faster runtimes. For example, our method is nearly 3 times faster than PMF and 10 times faster than DeepFlow. Table 4.7 shows that our method achieves around a five-fold speed increase over PMF for higher-resolution images. The method in DeepFlow has trouble dealing with higher-resolution images due to its large memory requirements (see [142]). See Table 4.1 for a comparison of space/time complexity of our method versus PMF and DeepFlow.

Figure 4.12 also shows that the SVF+OH outperforms several other recent methods, such as PCA-Layers [173] and optical flow with sparse matching (SparseFlow) [161], on both accuracy and speed. For instance, when compared against our method on the same machine using the authors provided code, the PCA-Layers method provides comparable CPU runtime to our method, as seen in Table 4.7. Moreover, PCA-Layers requires an offline training stage. Furthermore, our method outperforms the method by [18] on nearly all error measures (see results on clean passes). For instance, our method improves the EPE values of [18], which we use to compute the initial optical flow (see also Figure 4.11). On the clean pass, for example, the *all* and *s40+* EPE values returned from [18] are 6.494 and 39.152, respectively, whereas the EPE values achieved by our method are 5.450 and 35.933, respectively. On the final pass, these values are 8.377 and 49.083 for [18] and are 7.737 and 46.420 for our method, respectively. As expected, however, [18] posted faster runtimes (see Table 4.7).

The optical flow with convolutional networks and variational refinement (FlowNetS+ft+v) [60] is the only method that imposes a faster runtime over our single GPU implementation on the MPI final pass. Our method, however, outperforms it on accuracy by a large margin on the clean pass. Our method also gives better results on the Middlebury datasets (see Figure 4.12) than those of FlowNetS+ft+v. The runtime of FlowNetS+ft+v is 1.12

seconds compared to 1.95 seconds for our method. The FlowNetS+ft+v method requires a large set of training images with ground truth, which is difficult to obtain in practice, as indicated by [60]. Perhaps its inability to generalize is why FlowNetS+ft+v performs poorly on the Middlebury dataset.

The optical flow with coarse-to-fine PatchMatch (CPM-Flow) method [92] outperforms our method on accuracy, while having a slower runtime. This is because it relies on the edge-preserving interpolation of correspondences for optical flow (EpicFlow) technique, which involves both variational energy minimization and dense interpolation and takes around 3 seconds for post-processing results. Our method is much simpler and can be further accelerated on multi-GPUs (see Table 4.7).

Middlebury Benchmark

Figure 4.13 shows the Middlebury quantitative benchmark results and compares our method against several other techniques. The Middlebury benchmark contains images exhibiting small displacements. Some image pairs, such as Basketball and Backyard do show large displacements. Figure 4.13 (top left) shows the average EPE rank values versus runtime for all image regions. Figure 4.13 (top right) shows the average angle error rank versus runtime for all image regions. Our method outperforms every other method in terms of runtime performance, while achieving comparable (in some cases, only marginally worse) accuracy (see Figure 4.14). Our method outperforms several methods, including EPPM [18], CF [91], EpicFlow [143], dense correspondence fields for optical flow (FlowFields) [15], and DeepFlow [170]. Our method also achieves around 3 times the speed increase over PMF, while demonstrating better accuracy on the angle error rank and a comparable accuracy on the EPE rank. This suggests that our method is able to achieve good performance on both MPI Sintel and Middlebury benchmarks. These evaluations also strengthen the key contribution of our work: a fast method for computing optical flow that can achieve good accuracy on both small- and large-displacement optical-flow estimation problems.

KITTI Benchmark

The KITTI benchmark [65] was obtained by a moving vehicle capturing images of streets with city traffic. Thus, the recorded flow is due to camera motion, which results in scaling and rotation of objects with large smooth areas and few motion boundaries. In addition, the camera creates large distortions along image boundaries. This limits our local SVF method from finding accurate sub-volumes. We share this limitation with other local methods [18, 91, 122]. We think that methods employing global energy minimization [143] will perform better than our local method on KITTI. Figure 4.13 (bottom) shows the average EPE versus runtime of our method against other state-of-the-art methods. It is worth noting that we have a similar accuracy to FlowNetS+ft [60] on KITTI.

Sensitivity Analysis

Figure 4.15 shows a sensitivity analysis of the SVF+OH method on the Middlebury optical flow training datasets. Figure 4.15 (top left) shows a comparison of the average EPE versus the sub-volume expansion upper-bound parameter γ_s and for different values of the expansion factor β_s. It is clear that the error increases when $\gamma_s < 2$ or $\gamma_s > 5$. A low γ_s value results in small sub-volumes that discard several good motion candidates around their mean motion. A large γ_s value, on the other hand, results in large sub-volumes that have large noise. A similar result can be seen when varying the β_s. The lowest average EPE values are for $\beta_s = 1.96$ and the error increase as $\beta_s < 1.96$ or $\beta_s > 1.96$. Figure 4.15 (top left) shows the same comparison as the (top right) figure and results confirm our observations. Figure 4.15 (middle left) shows a comparison of the average filtering time versus the average EPE for different values of the number of superpixels K. Figure 4.15 (middle right) shows a similar comparison for average AE. The filtering times are reported on a single GPU. We can observe that larger values of K provide better accuracy, however at the cost of increased computational costs. Figure 4.15 (bottom row) presents the filtering time versus γ_s for different values of K. We can see that the filtering time increases exponentially with γ_s. The reason is that the computational time of the random search step of PatchMatch grows exponentially. Each PatchMatch iteration of SVF+OH searches for each sub-volume, a sequence of motions at an exponentially decreasing distance from its mean motion. For additional results on the Middlebury training datasets, please see Appendix A.

Figure 4.16 (top row) shows a time-accuracy trade-off evaluation of SVF for different values of K, where K is the number of superpixels. The EPE values are averaged over the entire MPI Sintel testing datasets, and the filtering times are reported on a single GPU. We observe that our method is robust to the choice of K. Furthermore, larger values of K yield better optical-flow estimation results. This, however, comes at the cost of increased processing costs. We found that setting K to 1500 gives a reasonable trade-off between accuracy and time. Similar results have been obtained on Figure 4.15 (middle row) for the Middlebury training datasets. Notice that K, β_s, and γ_s control the sub-volume size; however, we show by experiments that the chosen values of β_s and γ_s allow our method to achieve similar accuracy to PMF that randomly filters the entire cost volume. Increasing γ_s will not carry much benefit. Reducing γ_s, however, reduces accuracy, as we will favour the mean motion of each superpixel.

The middle and bottom rows of Figure 4.16 show the convergence of our method on two datasets, Wall and Ambush 3, on both the clean and final passes. Note that our method typically achieves convergence after seven iterations. The PMF [122], on the other hand, requires 10 iterations for convergence. A closer look at the figures shows that our method outperforms all compared methods on the Wall dataset. This is true for most datasets; however, our method fails on the Ambush 1 and Ambush 3 datasets, as they have large textureless regions. We share this limitation with other local methods [122, 18, 91].

4.9 Limitations

Given that the ACF and SVF methods are local, we share the same limitations of other local methods [122, 18, 91]. ACF and SVF fail on textureless regions with few motion boundaries. Our sub-volume formulation also has the problem of using a translational motion model. This creates problems on more complex motions consisting of rotations and scaling. This limitation is also shared with [122, 18, 91]. One future direction to handle this problem is to extend our formulation to use the homography and similarity transformations proposed by [50] and [99].

4.10 Concurrent Streaming Implementation

Streaming pipeline. Figures 4.12 and 4.13 present runtimes for our method on a single GPU for a fair comparison against other methods. Our method can also be implemented as a pipeline running on multiple GPUs in a concurrent streaming fashion. Table 4.7 compares the filtering times of the SVF against PMF [122] using one CPU, one GPU, and three GPUs. We apply loop unrolling for pipelining on three GPUs (i.e., the iterations needed for convergence are divided among GPUs). Figure 4.16 shows that the proposed method converges after seven iterations, and the iterations are assigned to GPUs as follows: iterations one through two for GPU 1; iterations three through four for GPU 2; and iterations five through seven for GPU 3. The PMF method, on the other hand, converges after 10 iterations, and the iterations are assigned to the three GPUs as follows: iterations one through three for GPU 1; iterations four through six for GPU 2; and iterations seven through ten for GPU 3.

Algebraic description. We describe the pipeline implementation of SVF using our developed stream algebra [76, 77, 78]. We start by defining the following data types:

$\texttt{Frame} : \texttt{2DImage}; \texttt{FramePair} : \texttt{Frame} \times \texttt{Frame};$

$\texttt{FramePyramid} : \textsc{List}\left\langle \texttt{Frame} \right\rangle; \texttt{FramePyramidPair} : \texttt{FramePyramid} \times \texttt{FramePyramid};$

$\texttt{Flow} : \textsc{List}\left\langle \texttt{Frame} \right\rangle; \texttt{FlowPyramid} : \textsc{List}\left\langle \texttt{Flow} \right\rangle$

$\texttt{FlowPyramidPair} : \texttt{FlowPyramid} \times \texttt{FlowPyramid};$

$\texttt{Superpixels} : \texttt{Frame}; \texttt{Subvolume} : \mathbb{R}^4; \texttt{SubvolumesScale} : \textsc{List}\left\langle \texttt{Subvolume} \right\rangle;$

$\texttt{SubvolumesPyramid} : \textsc{List}\left\langle \texttt{SubvolumesScale} \right\rangle;$

$\texttt{Cost} : \texttt{Frame}; \texttt{GPUInfo} : \mathbb{R}^2;$

$\texttt{FlowVector} : \texttt{FramePyramidPair} \times \texttt{FlowPyramidPair}$

$\texttt{SuperpixelsVector} : \texttt{FramePyramidPair} \times \texttt{FlowPyramidPair} \times \texttt{Superpixels}$

$\texttt{SubvolumesVector} : \texttt{FramePyramidPair} \times \texttt{SubvolumesPyramid} \times \texttt{Superpixels} \times \texttt{Cost} \times \texttt{Flow}$

where a `Frame` is a single 2D image, a `FramePair` is the input image pair for optical flow or stereo vision. The `FramePyramid` is a multiscale pyramid of a given frame and `FramePyramidPair` is a pair of frame pyramids. The `Flow` is a pair of frames, one representing displacements in x direction and the other representing displacements in y direction. In addition, `FlowPyramid` is a pyramid that has an optical flow defined at each scale. A `FlowPyramidPair` is a pair of flow pyramids, one for each input image. The `Superpixels` type is a frame that has the value of each pixel defining its superpixel index. A `Subvolume` is a 4D vector $(\mu_x, \mu_y, \sigma_x, \sigma_y)$ defined for each superpixel, specifying the mean and variance of its dominant motion. A `SubvolumesScale` is a list of sub-volumes for all superpixels in a given image at a specific scale. The `SubvolumesPyramid` is a pyramid that has each scale containing the list of sub-volumes for the superpixels at the corresponding scale in a `FramePyramid`. The `Cost` is a frame defining the cost of optical-flow label assignments and `GPUInfo` is a 2D vector (n, id) defining the number of PatchMatch iterations n and the identifier id of the target GPU in a multi-GPU pipeline. The `FlowVector`, `SuperpixelsVector`, and `SubvolumesVector` are vectors composed from the previously defined data types.

Given the datatypes, we assume an incoming stream of frame pairs $I : \mathbf{S}\langle\texttt{FramePair}\rangle$. Each pair $p \in I$ defines an input image pair. We start by defining the function $f_1 : \texttt{FramePair} \rightarrow \texttt{FramePyramidPair}$ that maps each input image into an image pyramid of three scales. This function can be used with the Map operator to define the stream $P : \mathbf{S}\langle\texttt{FramePyramidPair}\rangle$:

$$P \triangleq \textsc{Map}(f_1)(I). \tag{4.9}$$

Given the P stream, we define the function $f_2 : \texttt{FramePyramidPair} \rightarrow \texttt{FlowVector}$ that applies the method of [122] to generate the initial optical flow for each scale in the input image pyramids. The output is a pair of pyramids, one for forward flow and the other for backward flow. The function f_2 can be used with the Map operator to define the output stream $F : \mathbf{S}\langle\texttt{FlowVector}\rangle$:

$$F \triangleq \textsc{Map}(f_2)(P). \tag{4.10}$$

Later, we can apply the function $f_3 : \texttt{FlowVector} \rightarrow \texttt{SuperpixelsVector}$ that generates the superpixel image using the SLIC algorithm [1] for the original scale of the reference image (scale zero in the image pyramid of the first input image). Note that we only need to calculate the superpixels for the reference image where optical flow will be calculated. The function f_3 can be used with the Map operator to define the stream $S : \mathbf{S}\langle\texttt{SuperpixelsVector}\rangle$:

$$S \triangleq \textsc{Map}(f_3)(F). \tag{4.11}$$

Next, we calculate the list of sub-volumes for each scale in the reference image pyramid to produce a sub-volume pyramid using the function $f_4 : \texttt{SuperpixelsVector} \rightarrow \texttt{SubvolumesVector}$. The stream $V : \mathbf{S}\langle\texttt{SubvolumesVector}\rangle$ is then defined using the following Map operator:

$$V \triangleq \textsc{Map}(f_4)(S). \tag{4.12}$$

Now, that we defined the list of sub-volumes for the reference image, we start applying the PatchMatch iterations. We define the function $g : \texttt{SubvolumesVector} \times \texttt{GPUInfo} \rightarrow \texttt{SubvolumesVector} \times \texttt{GPUInfo}$, which executes the PatchMatch iterations and takes as input the vector (s, u). The sub-volume vector $s \in V$ comes from the stream V. In addition, q is a 2D vector $(q.n, q.id)$ defining the number of PatchMatch iterations $q.n$ and the identifier $q.id$ of the target GPU device. Given the three GPU pipeline scenarios discussed in the previous section, we define three status variables u_1, u_2, and u_3 of type `GPUInfo` and assign them the vectors $(2, GPU1)$, $(2, GPU2)$, and $(3, GPU3)$, respectively. Then, we use the following three Reduce operators to create the multi-GPU pipeline and produce the stream $V' : \mathbf{S}\langle\texttt{SubvolumesVector}\rangle$:

$$V' \triangleq \textsc{Reduce}(g, u_1) \circ \textsc{Reduce}(g, u_2) \circ \textsc{Reduce}(g, u_3)(V), \qquad (4.13)$$

where $\circ$ is the composition operator. Note that V' contains the initial output flow computed using our SVF method. The final Map operator can be later defined for post-processing the initially computed flow.

Throughput versus latency analysis. In order to show the benefits of the algebraic description of the SVF pipeline, we study its throughput and latency (See Section 3.3 for the definition of throughput and latency). Throughput is the inverse of the period which is the slowest operator computation or communication time in the pipeline. In our study, we report the period. Figure 4.17a shows the described pipeline for SVF. For standard resolution images, the latency of the Reduce operators to filter each pair of images is 0.88 seconds and the period is 0.38 seconds; whereas for medium-resolution, these numbers are 1.06 0.45 seconds, respectively. Similarly, for higher-resolution images, the latency is 5.9 seconds and the period is 2.4 seconds. These performance numbers indicate that it takes about 0.88 seconds to filter the first pair of images, and then 0.38 seconds for subsequent frames, on standard resolution images. The overhead due to data transfer between multiple GPUs is minimal and is around 0.07 seconds. Notice that the filtering time of our method is 5.53 seconds for single GPU implementation. Using three GPUs also improves the performance of PMF, CF, and ACF; however, given the performance of CF and ACF on low-resolution images, it was infeasible to use CF and ACF for medium and high-resolution images.

We also study the total latency and throughput of the pipeline on medium resolution images. The computational time of the functions executed by the four Map operators in Figure 4.17a are as follows, 0.11, 0.6, 0.35, and 0.1 seconds, in the same left to right order. For the three Reduce operators, their computational times are 0.31, 0.31, and 0.45 seconds. We ignore the communication time between these operators as in our setup they are running on the same machine. So, the total latency of the pipeline is around 2.2 seconds. Notice that the period now is 0.6 seconds as the slowest operation is the calculation of initial optical flow.

Figure 4.17b shows a similar pipeline to Figure 4.17a; however, we use two extra GPUs to execute two replicas in parallel for the Map operator that computes the initial optical flow. Scatter and Merge are used to distribute elements of the P stream in a round-robin fashion to a ListMap operator that executes the two replicas. The ListMap processes two inputs in parallel, so the resulting period of the Scatter, ListMap, and Merge operators is approximately 0.3 seconds or half the period of a single Map operator. On medium-resolution images, the period of our method on a five GPU pipeline is 0.45 seconds (≈ 0.5), which provides better throughput. It is worth mentioning that the period of PMF using five GPUs is 1.75 seconds. So, our study shows that with more GPUs, we can scale up the pipeline to achieve better throughput. This is achieved by using our algebraic operators to construct parallel processing patterns with minimal programming efforts. We think that our stream algebra will be helpful in fast prototyping of large-scale computer vision systems.

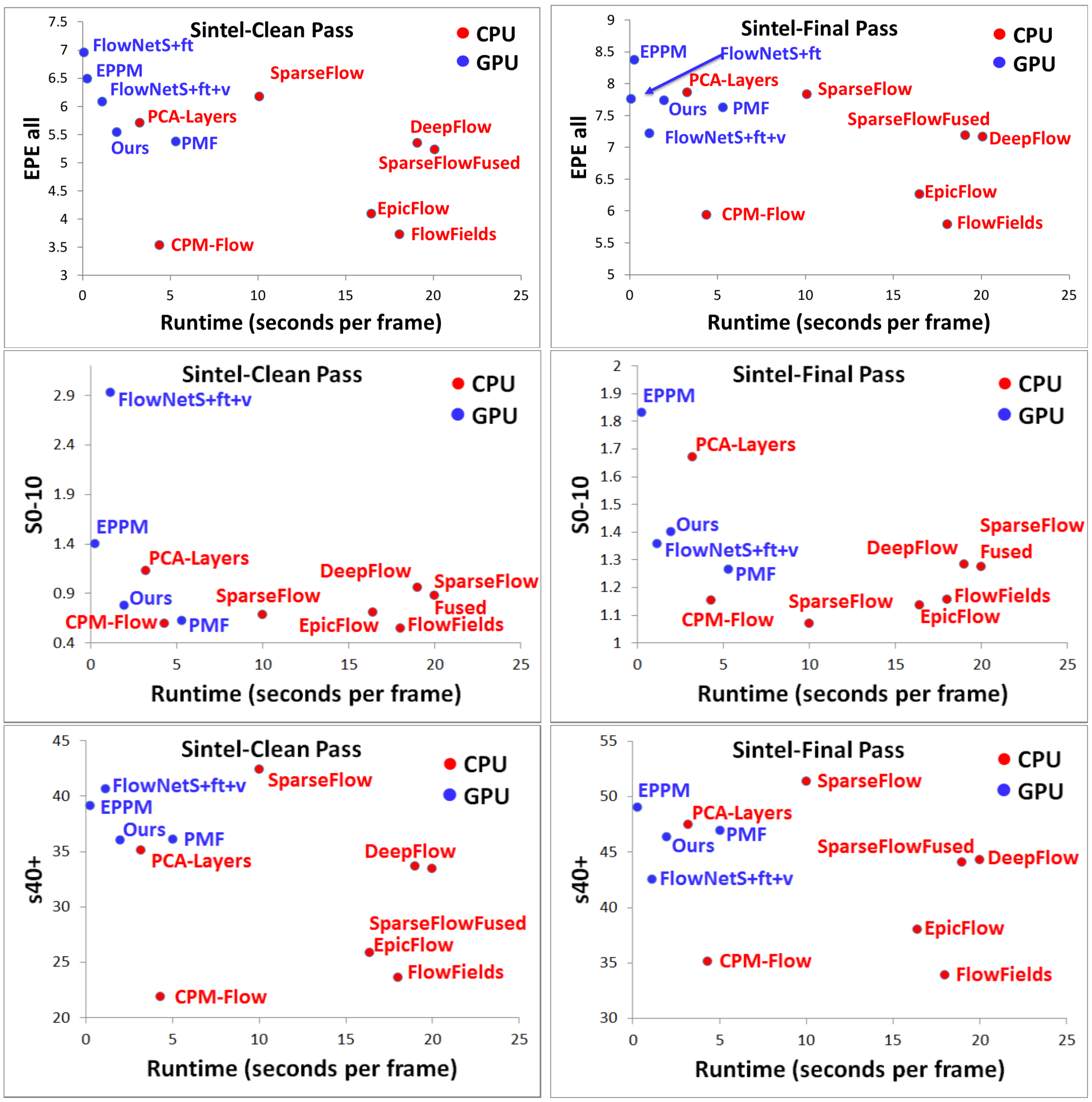


Figure 4.12: Accuracy versus runtime evaluation (summarization) recorded on 24 January 2017 on the MPI Sintel benchmark. We focus on the top-ranked techniques with runtime ≤ 40 seconds. Detailed MPI Sintel results are available online [133] and in Appendix A.

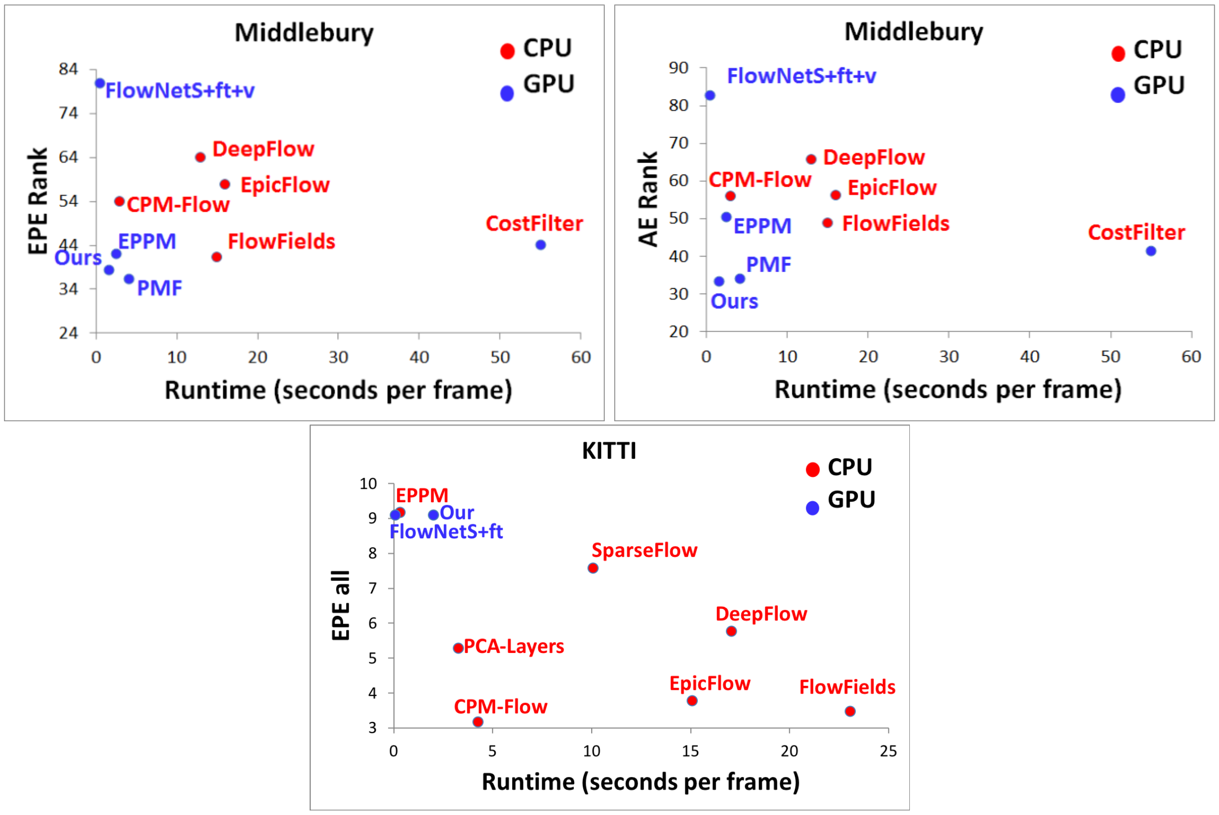
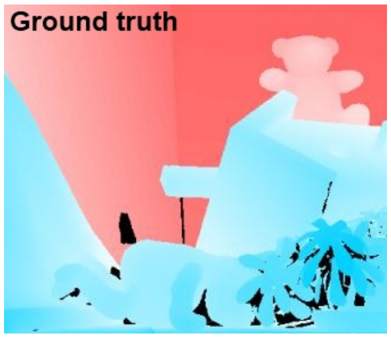
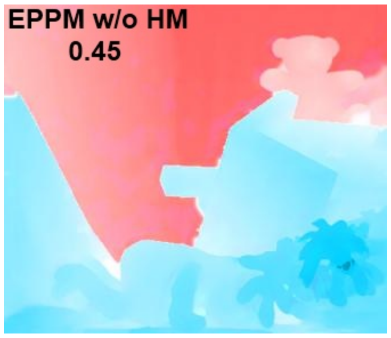


Figure 4.13: Accuracy versus runtime evaluation (summarization) recorded on 24 January 2017 on the Middlebury and KITTI benchmarks. We focus on the top-ranked techniques with runtime ≤ 40 seconds. The EPPM is computed by [18] without hierarchical matching (HM). Detailed results are available online [133] and in Appendix A.

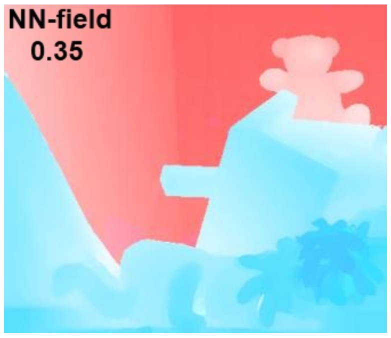
(a) Teddy first frame.



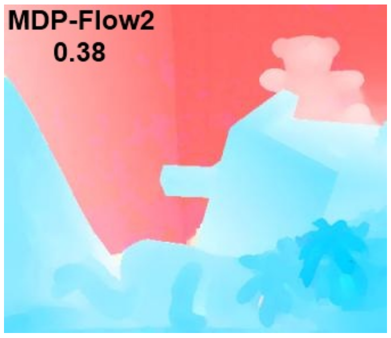
(b) Ground truth.



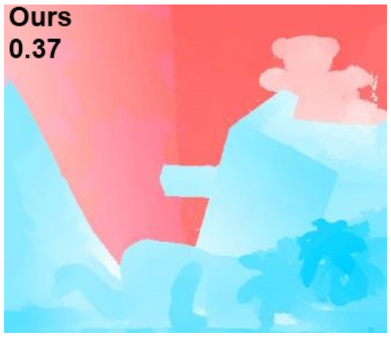
(c) EPPM without HM (0.45).



(d) NNF-Local(0.35).



(e) MDP-Flow 2 (0.38).



(f) SVF+OH (0.37).

Figure 4.14: A visual comparison between our cost-sub-volume filtering with occlusion handling (SVF+OH) method [80] against edge-preserving PatchMatch (EPPM) without hierarchical matching (HM) [18], optical flow with nearest neighbour field (NNF-Local) [50], and motion detail preserving optical flow (MDP-Flow2) [174], on the Teddy Middlebury dataset. Note that EPPM without HM has a runtime of 2.5 seconds. The EPE of each method is displayed in the caption.

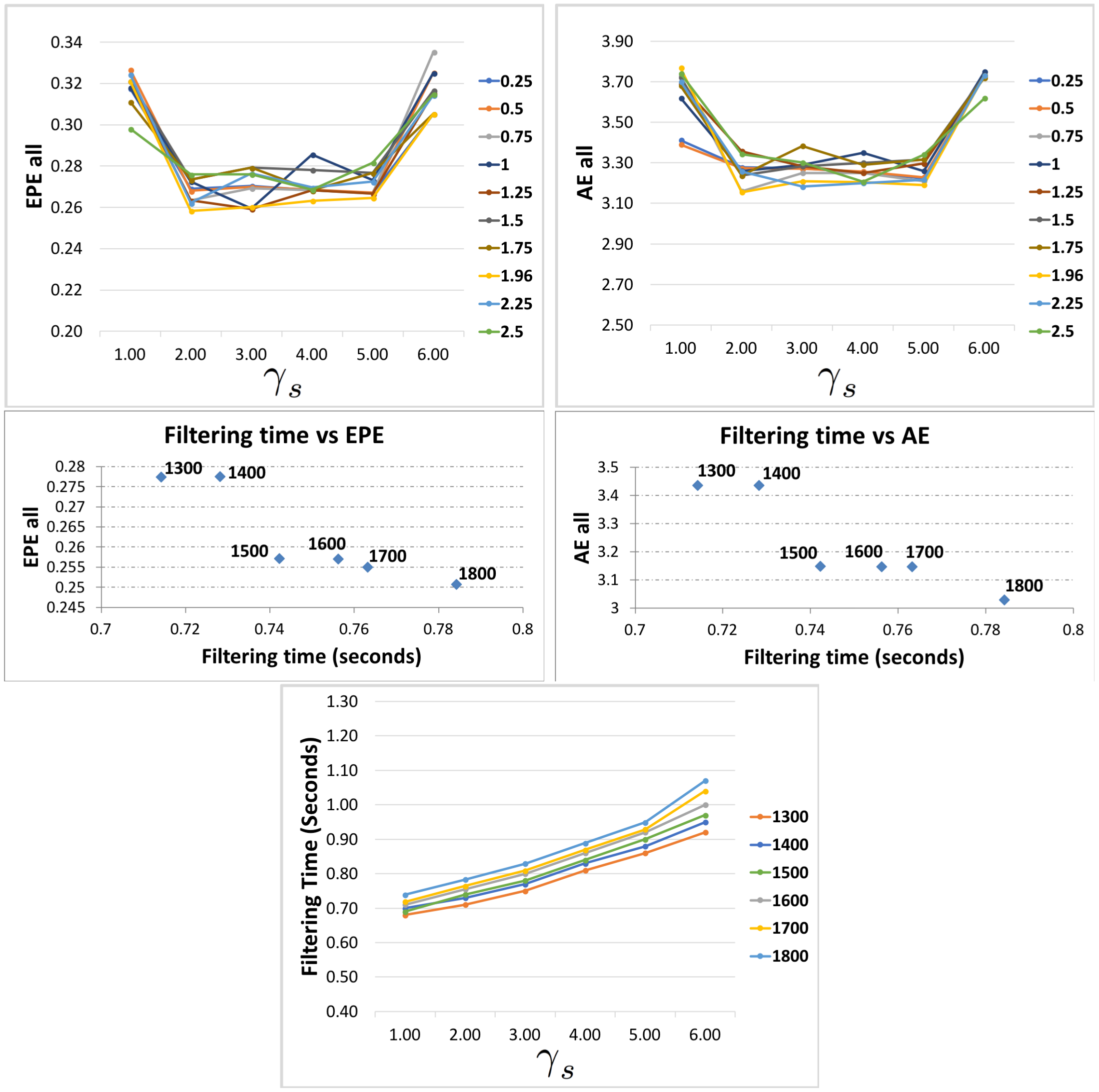


Figure 4.15: A sensitivity analysis of the cost-sub-volume filtering with occlusion handling (SVF+OH) method on the Middlebury optical flow training dataset: (top left) a comparison of the average EPE versus the sub-volume expansion upper-bound parameter γ_s and for different values of the expansion factor β_s; (top right) a similar comparison for the average angle error (AE); (middle row) the filtering time versus accuracy for different values of K (number of superpixels); (bottom row) the filtering time versus γ_s for different values of K.

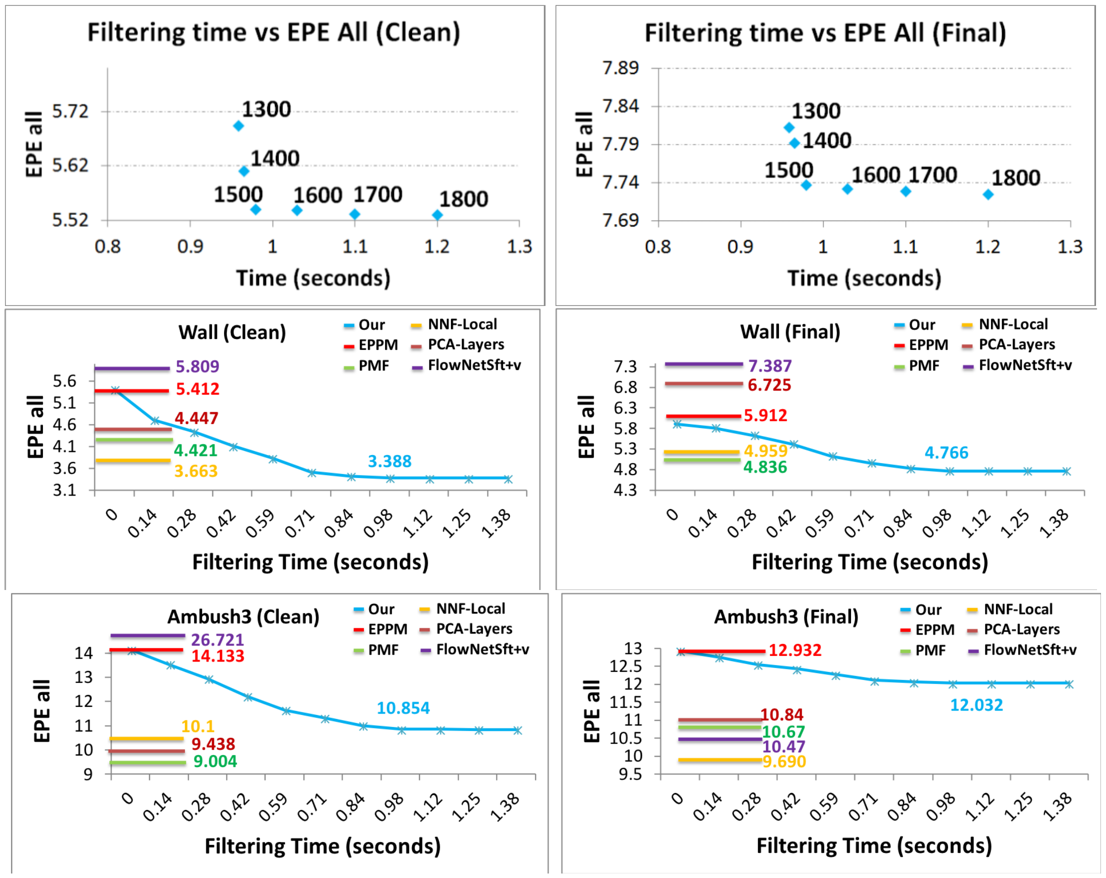


Figure 4.16: A time-accuracy trade-off study: (top row) time versus accuracy for different values of K (number of superpixels) on the clean and final passes of the MPI Sintel testing dataset; (middle and bottom rows) the convergence for 10 iterations (baseline is 7) of our algorithm on the Wall and Ambush 3 datasets compared to other methods.

$$\begin{tikzcd}
I \arrow[r] & \boxed{\textsc{Map}} \arrow[r, "P"'] & \boxed{\textsc{Map}} \arrow[r, "F"'] & \boxed{\textsc{Map}} \arrow[r, "S"'] & \boxed{\textsc{Map}} \arrow[r, "V"'] & \boxed{\textsc{Reduce}} \arrow[r] & \boxed{\textsc{Reduce}} \arrow[r] & \boxed{\textsc{Reduce}} \arrow[r] & V
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
I \arrow[r] & \boxed{\textsc{Map}} \arrow[r, "P"'] & \boxed{\textsc{Scatter}} \arrow[r] & \boxed{\textsc{ListMap}} \arrow[r] & \boxed{\textsc{Merge}} \arrow[r, "F"'] & \boxed{\textsc{Map}} \arrow[r, "S"'] & \boxed{\textsc{Map}} \arrow[r, "V"'] & \boxed{\textsc{Reduce}} \arrow[r] & \boxed{\textsc{Reduce}} \arrow[r] & \boxed{\textsc{Reduce}} \arrow[r] & V
\end{tikzcd}$$

(b)

Figure 4.17: Describing the cost-sub-volume filtering (SVF) method using our stream algebra: (a) the streaming pipeline of SVF executed using three GPUs; (b) the pipeline in (a) executed on five GPUs. Each Reduce operator in (a) executes a predefined number of sub-volume filtering iterations on a dedicated GPU. In (b), three GPUs are used by the Reduce operators, and two GPUs are used to execute two replicas of the second Map operator in (a) that computes the initial optical flow. Scatter and Merge are used to distribute elements of the P stream in a round-robin fashion to a ListMap operator that executes the two replicas, each on its dedicated GPU.

Chapter 5

Efficient Computer Vision Functionals: Traffic Surveillance

Automatic analysis of traffic-video streams is an active research area in computer vision. This area is important due to the large deployment of traffic-surveillance cameras, which can generate terabytes of video per hour. This motivated research to develop vision-based systems that can automatically gather traffic statistics and monitor, detect, and classify significant road events, such as traffic congestion, rule violation, dangerous behaviours, and incidents. One of the crucial tasks in these systems is automatic road and lane detection, which allows a vision system to divide a traffic scene into road and non-road regions. This task has several advantages. For example, it can provide a self-adaption to camera viewpoint changes that result from a human operator or wind. It can also help in reducing the computational time by only processing road regions.

Several techniques have been proposed for road and lane detection. These techniques can be classified into activity-driven [154, 130], feature-driven [151, 6, 109], and model-driven approaches [169, 189]. The activity-driven approaches rely on vehicular motion activity to extract a scene-activity map. This map divides each image into active (road) and inactive (non-road) regions. The feature-driven approaches rely upon the extraction of image features to detect lane and road boundaries. The model-driven approaches define a road geometric model and fit this model to road regions.

Road-boundary detection using traffic cameras is a non-trivial task. This is because many of the traffic cameras typically mounted along highways are not calibrated, and it is tedious and difficult to maintain their calibration. Moreover, these cameras are usually equipped with pan and tilt features to enable human operators to change view and monitor different regions in the traffic scene. This makes the cameras susceptible to strong sway caused by wind. The traffic scene can also be noisy and barely visible under adverse environmental conditions, such as rain, fog, and snow. Figure 5.1 shows several examples of images in traffic-video streams recorded by cameras installed along Ontario Highway 401. These examples present a variety of severe environmental conditions that are usually encountered in traffic-video streams and that make the task of road-boundary detection challenging.

This chapter focuses on the automatic detection of dominant road boundaries in traffic-surveillance imagery. The task is to identify road regions in each image of a traffic-video stream. For many algorithms for traffic analysis, the detection of road regions is a fundamental task used to support other higher-level analyses of the traffic scene by localizing processing on road regions. Examples of such analyses include locating erratic driving behaviour, finding stranded vehicles, monitoring traffic flows, etc. We present a novel online algorithm [81] for automatic detection of dominant road boundaries in traffic cameras. The algorithm starts by extracting and accumulating edge features from each frame in an input video stream (feature-driven method). Hierarchical clustering is then applied to maintain a clustering tree on the accumulated edge features. The clustering tree is automatically updated toward the addition of new edge features. Each cluster contains a subset of accumulated edge features and represents a candidate road boundary. Road boundaries are defined as straight lines, and each cluster has a straight-line model representing the mean of its edge segments (model-driven method). A rank is then assigned to each cluster using χ^2 and student t statistical measures, and the Cartesian product of the top statistically ranked clusters forms a set of candidate pairs for the road boundary. Each pair is later ranked using perspective cues and road vehicular activity (model-driven method). The dominant road boundary is selected as the top-ranked pair.

To incrementally update the clustering tree by the extracted edges of new incoming frames, we experiment with two methods: (1) incremental bottom-up hierarchical clustering and (2) online top-down hierarchical clustering. In the first method, we initialize the clustering tree using the extracted edges of the first incoming frame. Then, for each subsequent incoming frame, we accumulate its extracted edge features. After accumulating sufficient evidence, the algorithm builds a hierarchical bottom-up clustering tree using all previously accumulated edges. We reinitialize the clustering tree after every 25 frames. This number was selected by trial and error. This however requires us to rebuild the entire clustering hierarchy after every 25 frames, which results in a higher runtime performance.

The second method uses the ClusTree [111] algorithm to build an online top-down clustering hierarchy. This online algorithm approximates bottom-up clustering by maintaining a clustering tree over a sliding window of the incoming video stream. When new edges are inserted into the tree, the previous edges are removed. This removes the requirement of the first method to reinitialize the clustering tree and thus significantly improves the runtime performance. The online method also smoothly adapts to sudden changes in road boundaries resulting from changes in the camera view.

From the perspective of stream processing, the incremental bottom-up hierarchical-clustering method acts as a *blocking stream operation* when applied on an incoming video stream. Blocking operations are not desirable for processing data streams, and we will see later in the experimental results that this blocking operation results in a significant performance decrease and limits the scalability of our road-boundary detection algorithm. The online hierarchical-clustering algorithm acts as a *non-blocking stream* *operation*; therefore, it is much faster than the incremental clustering method.

Experimental results are performed on two real-world datasets having traffic-image sequences recorded from several traffic cameras installed along Highway 401. The first dataset contains 14 low-resolution image sequences recorded from different camera locations under a variety of environmental and lighting conditions. Each sequence has 50 frames, 25 for daytime and another 25 for nighttime. The second dataset is a long image sequence of 1,627 frames recorded from a single camera location. This dataset represents a long stream of images and has the camera view changes by a human operator to focus on different regions in the traffic scene.

A comparison is performed between our online and incremental clustering methods and other state-of-the art approaches that include the Gabor filter-based method [109] and the deep learning method by [39]. The results show that our online road-boundary detection method outperforms both [109] and [39] in accuracy and runtime. We also show that using the online clustering method in our road-boundary detection algorithm is significantly faster than (roughly 800 times faster) using the incremental clustering method.

The road-boundary detection algorithm provides four main contributions. First, the algorithm is online and can accurately find dominant road boundaries under severe environmental and lighting conditions. Second, our method can accurately detect road boundaries from low-resolution (320×240) video streams at 20 frames per second. Thus, our method can operate well in bandwidth-limited environments typically found in large camera networks. Third, our method can adaptively detect the road boundary under changing camera-viewing directions. Finally, a statistical measure is developed for ranking clusters, eliminating the need to use prior knowledge or apply heuristics. This statistical measure may be generally useful for cluster ranking in other similar applications.

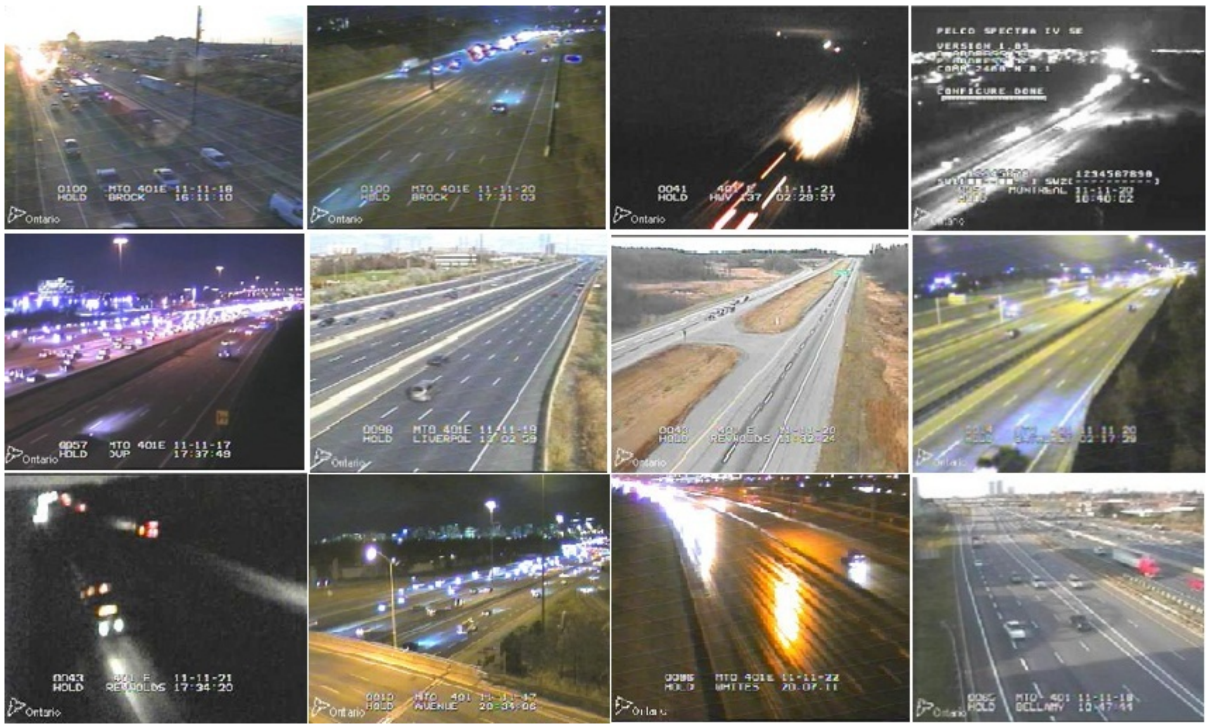


Figure 5.1: Challenging environmental conditions encountered by traffic-surveillance cameras.

5.1 Road-boundary Detection

Our method contains five steps: (1) superpixel segmentation, (2) contour approximation, (3) hierarchical bottom-up clustering, (4) confidence assignment, and (5) pairwise ranking. The following sections discuss each step.

5.1.1 Superpixel Segmentation

Edges are one of the fundamental features for identifying dominant road and lane boundaries in traffic scenes. The main problem of these features is the existence of large noise from changes in environmental and lighting conditions affecting the performance of edge-detection algorithms. To handle such noise, several existing algorithms rely on prior knowledge of the road structure. In this work, we take a different approach. Road regions span large areas in the traffic scene. Although the boundaries of these regions

Algorithm 3 Overview of our algorithm for finding dominant road and lane regions.

```
Require: Image sequence
Ensure: Dominant road boundary
 1: Divide each image into homogeneous regions through superpixel segmentation.
 2: Approximate each superpixel contour with polygons to obtain edges.
 3: Perform bottom-up hierarchical clustering on these edges.
 4: Use statistical measures ($\chi^2$ and student t test) to identify the top-ranked clusters,
    where each cluster represents a road boundary in the image.
 5: Construct top-ranked cluster pairs through perspective filtering and road-activity
    analysis.
 6: The top-ranked cluster pair is returned as the dominant road boundary.
```

may be noisy and occluded, some sections of these boundaries can be detected correctly. Thus, the task is to find these correct segments and use them to detect the entire road boundary.

We start by applying superpixel segmentation to extract a set of abstract regions. Superpixel segmentation [141] divides an image into a set of regions called superpixels, so that each pixel in the image belongs to only one superpixel. Pixels belonging to a superpixel are neighbours sharing similar appearance attributes (e.g., colour and texture). Several superpixel segmentation methods [135, 117, 49, 181] are slow and do not meet the fast processing requirements of traffic-video analysis. TurboPixels [113] and SLIC [1] are fast alternative methods for superpixel segmentation. TurboPixels [113], however, requires an initial manual choice of seed points, and SLIC requires manually setting the superpixel compactness parameter.

In this work, we use the fast superpixel segmentation method by [82]. This method can generate a set of 100 superpixels in 0.339 seconds compared to 5 seconds for TurboPixels. The method is simple and can be parallelized for real-time processing. Figure 5.2 shows an example implementation of the method by [82] on a daylight traffic image. The method starts by performing a morphological open operation followed by a close operation. The open operation smooths noisy areas and contours in images. For example, the operation eliminates tiny bumps and breaks narrow strips. The close operation eliminates small

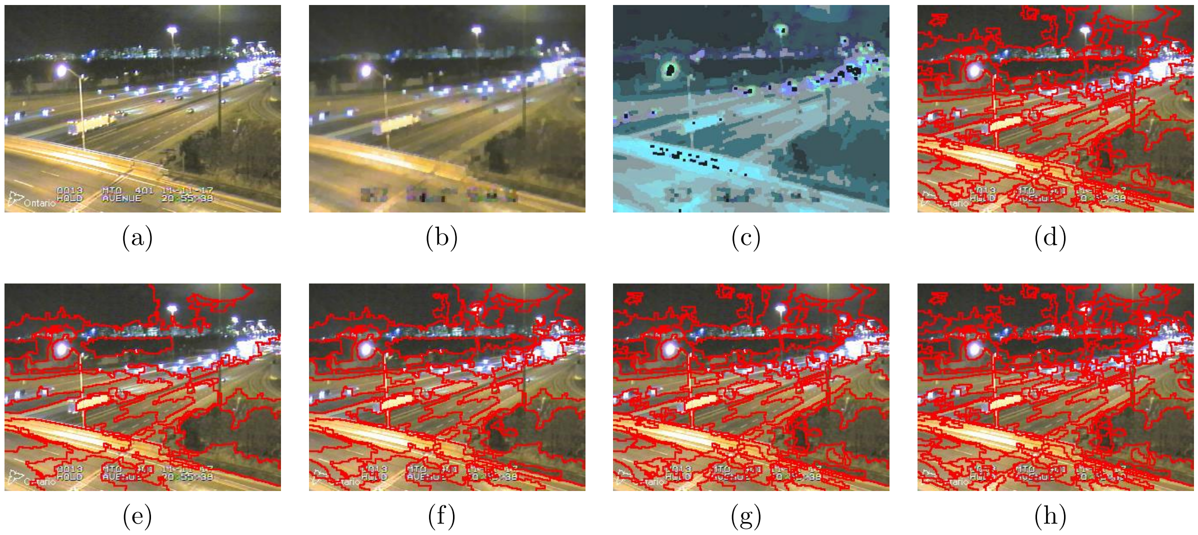


Figure 5.2: Hierarchy of superpixels generated using the method in [82]: (a) input image; (b) result of applying morphological operations; (c) image after hue-saturation-value (HSV) colour quantization; (d) over-segmentation (e) no. of superpixels = 25, (f) no. of superpixels = 50, (g) no. of superpixels = 100, and (h) no. of superpixels = 150.

holes and gaps and helps link noisy contours. The resulting image is then represented in hue-saturation-value (HSV) colour space, and an HSV histogram of 16 × 8 × 8 bins is created. A colour quantization step is then performed by assigning each pixel to the colour bin closest to it. Neighbouring pixels are then merged together based on colour similarity to form an initial set of regions (see Figure 5.2d). Adjacent regions are further merged together to create a hierarchy of superpixels.

Figures 5.2e to 5.2h present superpixel segmentation examples when setting the number of superpixels to values from 25 to 150. These figures show an important observation. Although the boundaries of these superpixels change due to environmental and lighting conditions, large segments of these boundaries still capture stable edges showing the road structure. Our method utilizes this observation to gather stable edge information representing the geometry of the traffic scene.

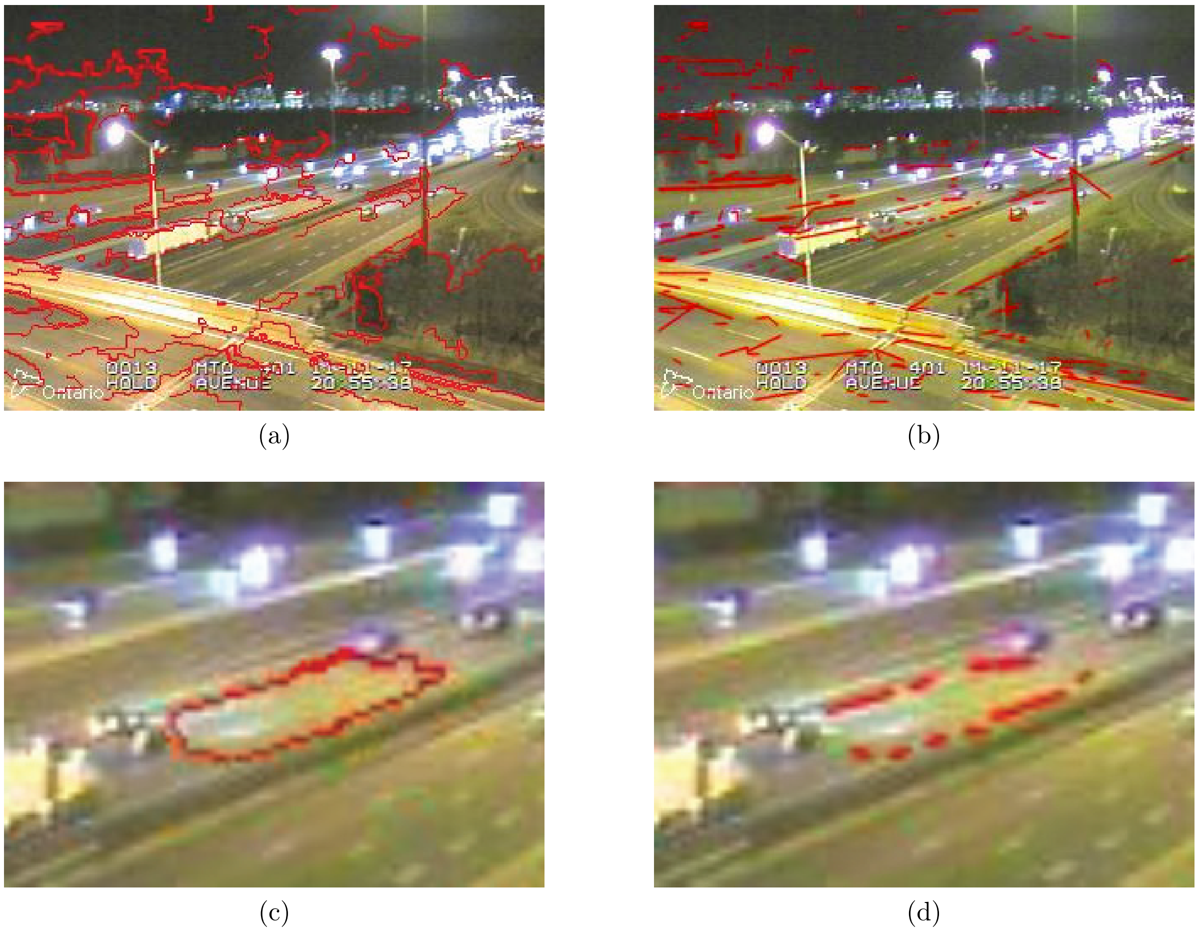


Figure 5.3: Generating an approximate polygon using adaptive sampling: (a) generated polygons for superpixels in Figure 5.2e; (b) edges with a length greater than 10 pixels; (c) selected superpixel from (a); (d) edges with length greater than 10 pixels for the superpixel in (c).

5.1.2 Contour Approximation

To extract edges, our algorithm uses adaptive sampling [59] to obtain a 2D polygon approximation for the contour of each superpixel. Given the superpixel contour described as a 2D parametric curve $\gamma : [0,1] \rightarrow r^2$, adaptive sampling approximates the contour by selecting a set of n points with timing $t_1 < t_2 < \cdots, < t_n$, and $t_0 = 0$ and $t_n = 1$ corresponding to the vertices $v_1 = \gamma(t_1), \cdots, v_n = \gamma(t_n)$ such that n is as small as possible.

Our method starts by randomly selecting two points on each superpixel contour $v_l = \gamma(l)$ and $v_e = \gamma(e)$, where $l = 0$ is the starting point and $e = 1$ is the ending point.

Algorithm 4 Approximate-Contour.

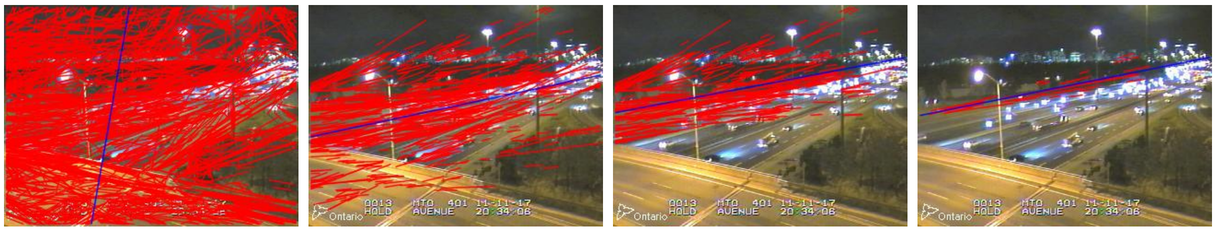
```
Require: $\gamma, l, e$
Ensure: Straight edges.
1: Set $v_l = \gamma(l)$, $v_e = \gamma(e)$, $m = \frac{1}{2}(l+e)$, and $v_m = \gamma(m)$.
2: If the curve tangents at $v_l, v_m$ and $v_e$ are almost parallel, then
     Output the straight segment $\overline{v_l v_e}$.
3: Otherwise,
     Approximate-Contour $(\gamma, l, m)$.
     Approximate-Contour$(\gamma, m, e)$.
```

Then, an approximate polygon is generated using Algorithm 4. The algorithm works by choosing a point $v_m = \gamma(m)$ with $l < m < e$. Then, a flatness test is performed to measure the collinearity of the points v_l, v_m and v_e. This by checking if the two tangents $\overrightarrow{v_l v_m}$ and $\overrightarrow{v_m v_e}$ are nearly parallel. If $v_l v_m v_e$ is a flat segment, it will be listed in the approximate polygon; otherwise, the test is recursively applied on the two intervals $[l, m]$ and $[m, e]$.

Over-sampling is a known problem for adaptive sampling, where a straight segment can be oversegmented into smaller segments. Our algorithm handles such problems by removing intermediate vertices within an almost flat segment. Figure 5.3 presents an example of applying adaptive sampling for polygon approximation of superpixel contours. Figure 5.3b shows approximated polygon segments with lengths greater than 10 pixels. Figures 5.3c and 5.3d present the approximate polygon for a selected superpixel, which shows the effectiveness of Algorithm 4 in approximating the polygons of superpixel contours.

5.1.3 Online Hierarchical Clustering

Our method generates the approximate polygons from each frame and accumulates their line segments over time. Figure 5.4 shows a subset of a clustering hierarchy tree with four nested clusters of line segments accumulated from a sequence of frames. Accumulated line segments will be more concentrated around stable line edges in the scene and less concentrated around temporary edges from a transient change of lighting or environmen-



(a) (b) (c) (d)

Figure 5.4: Subset of a generated clustering hierarchy tree, showing the accumulated line segments in four nested clusters: (a) root cluster, (b) and (c) two nested clusters with large variance, and (d) a good candidate cluster for a road boundary, which has small variance and a large number of accumulated segments.

tal conditions. We use this property to find a set of clusters with small variance and dense colinear segments that can be good candidates for road boundaries.

We start by applying our incremental clustering method [75], where we use agglomerative bottom-up hierarchical clustering with average linkage to build a clustering tree of accumulated line segments. A line segment is defined as a 2D vector (ρ, θ) in polar coordinates. The set $\mathcal{S} = \{\mathbf{s}_i | i = 1 \cdots n\}$ is defined as the set of all line segments generated by a given sequence of frames. Our algorithm normalizes the set $\mathcal{S}$ to zero mean and unit variance. Agglomerative clustering starts by assigning a cluster c_i to each line segment in $\mathbf{s}_i \in \mathcal{S}$. Then, every two closest pairs of clusters are merged together. This merging operation is repeated until one cluster is left representing the root of the generated clustering hierarchy.

As stated earlier, if we incrementally updated the generated clustering tree for every incoming frame, the size of the generated clustering tree will grow over time. This behaviour results in very slow runtime performance as the tree becomes large. This also assumes that the camera does not change its viewing direction and stays fixed. Such an assumption does not hold in reality, and a very noisy clustering tree can be generated. It also becomes infeasible to maintain the clustering tree for long and infinite video streams. Thus, our incremental bottom-up clustering method handles such limitations by dividing the video stream into a sequence of clips, each spanning 25 frames. A clustering tree is then generated for line segments accumulated within every clip. Therefore, our incremental clustering method rebuilds the clustering tree every 25 frames. This results in a performance decrease as the entire hierarchy should be rebuilt from scratch. Moreover, each clip loses the clustering information generated from previous clips.

Our online clustering method provides a better solution that handles the limitations of the incremental clustering method. The online method utilizes the ClusTree [111] algorithm for generating an online top-down hierarchical clustering. The algorithm keeps adding new incoming evidence and removing old information. This solves the limitations of the incremental clustering method by having an order of magnitude speed increase and accurately adapting to changes in the camera view. The ClusTree algorithm builds an R-tree based multidimensional index, where each tree node represents a cluster and has an associated feature vector $CF = (n, LS, SS)$, a timestamp variable t, and pointers to children nodes. The timestamp variable records the last update time of the feature vector of the node, where n is the number of line segments inserted to the node cluster. In addition, $LS = \sum_{i=1}^{n} \mathbf{s}_i$ defines the linear sum of all inserted segments, and $SS = \sum_{i=1}^{n} \mathbf{s}_i^2$ defines the squared sum. For a given cluster, CF defines its sufficient statistics vector. This vector contains the components required to incrementally update the cluster mean and variance. The mean is calculated as $\mu = LS/n$, and the variance is $\sigma^2 = (SS - 2\mu \times LS + \mu^2)/n$ To forget old elements, the CF vector is weighted by an exponential decay function $\omega(\Delta t) = \beta^{\lambda \Delta t}$ controlled by the decay rate λ. Thus, after every cluster update, the components of its feature vector are weighted as follows:

$$n^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i), \tag{5.1}$$

$$LS^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i)\mathbf{s}_i, \tag{5.2}$$

$$LS^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i)\mathbf{s}_i^2, \tag{5.3}$$

where t represents the current time, and for each line segment s_i, ts_i is the arrival time of s_i. It was proved by [111] that this weighting of cluster features maintains both the additive and temporal multiplicity properties. For example, if a cluster did not receive any objects within an interval $[t, t + \Delta t]$, its cluster feature is $CF^{(t+\Delta t)} = \omega(\Delta t) CF^{(t)}$. In addition, a cluster with k children has the cluster feature $CF^{(t+\Delta t)} = \omega(\Delta t) \sum_{i=1}^{k} CF_i^{(t)}$, where $CF_i^{(t)}$ is the cluster feature of the i-th child. We refer the reader to [111] for the full proofs of these properties.

The online top-down clustering works by attaching the current timestamp ts to each new incoming 2D line segment $\mathbf{s}$ described in polar coordinates. When the clustering tree is empty, a new root node is created with a cluster feature vector that is initialized by the incoming 2D line segment object. The timestamp of the new root node is set to ts. The next incoming segment will first be inserted into the root node of the clustering tree. Then, the object descends into the child node that has the closest Euclidean distance between its mean and the object. If we reached a leaf node, a new child cluster will be created for the new object, and we stop descending into the tree. While descending into the tree, the new object is inserted at each nested cluster it passes by. Three actions are performed when inserting a new line segment object $\mathbf{s}$ into a tree node. (1) The feature vector of the node cluster is updated by adding the new object, $n = n + 1$, $LS = LS + \mathbf{s}$, and $SS = SS + \mathbf{s}^2$. (2) The feature vector is multiplied by the decay function $\omega(t - ts)$. (3) The node timestamp is updated to the object timestamp $t = ts$. While descending into the tree, the algorithm also checks for outdated child clusters that have $n^{(t)} < \beta^{\lambda \Delta t_c}$. These clusters are deleted, and Δt_c controls the deletion rate.

Notice that if we keep inserting objects into the clustering tree, it can grow forever and have a very great height. Thus, we usually predefine the maximum height of the clustering tree. When the maximum height is reached at a given leaf node, we only insert new objects to the cluster vector of the node and stop creating new child clusters. Figure 5.4 shows line segments accumulated in four nested clusters of a generated clustering hierarchy. The innermost cluster clearly represents a road edge.

Given an incoming video frame, our online clustering method inserts all approximated line segments into the top-down clustering tree. Then, all clusters visited by the line segments of the frame are recorded and used as the candidate set of road edge clusters for this frame.

5.1.4 Confidence Assignment

The algorithm proceeds by statistically ranking the set of candidate clusters using the cluster variance and number of samples. The target is to penalize clusters having a small number of samples or exhibiting high variance. To perform ranking, we start by modelling a road boundary as a line described by the parametric form:

$$\widehat{m}u + \overrightarrow{v}, u \in R, \tag{5.4}$$

where $\overrightarrow{v}$ and $\widehat{m}$ are an offset vector from the origin and a directional unit vector, respectively. A segment belonging to a road edge is then defined as:

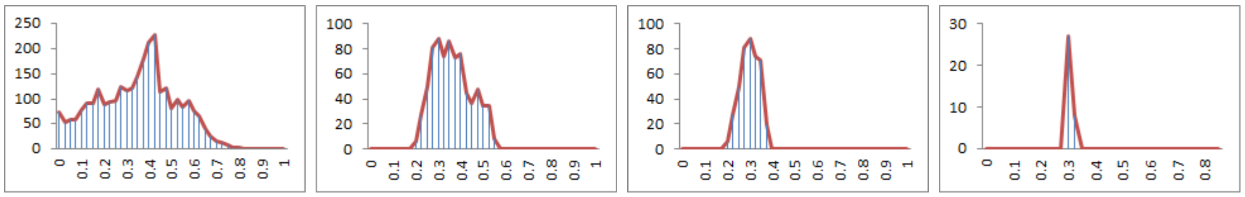
$$\overrightarrow{m_s}u + \overrightarrow{v_s}, u \in [a, b]. \tag{5.5}$$

Using Equations 5.4 and 5.5, we can define the generative model:

$$\overrightarrow{m_s} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \cdot \widehat{m}, \;\; \phi \in \mathcal{N}(\mu_1, \sigma_1^2), \tag{5.6}$$

$$\overrightarrow{v_s} = \rho \overrightarrow{v}, \;\; \rho \in \mathcal{N}(\mu_2, \sigma_2^2). \tag{5.7}$$

Here, ϕ and ρ are Gaussian random variables with noise controlled by the parameters $\theta = (\mu_k, \sigma_k^2)_{k=1,2}$. Our model can then be represented by the probability $P(\pi|\theta)$, where $\pi = (\overrightarrow{m}, \overrightarrow{v})$ defines the hidden variables of a true road-boundary line.


(a) ρ histograms.

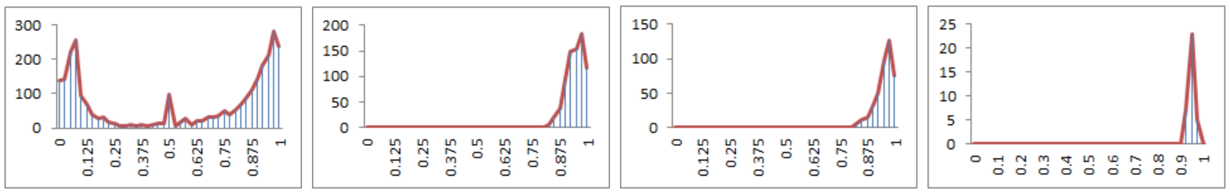

(b) ϕ histograms.

Figure 5.5: Histograms of ρ and ϕ for each cluster in Figure 5.4: (a) ρ histograms and (b) ϕ histograms. The first column is for the root cluster in Figure 5.4 and large noise is expected and seen in the histograms. The second column shows the histograms for the nested cluster in Figure 5.4b. The ϕ histogram is normally distributed, and the ρ histogram is the least noisy with a few clear peaks. The third column shows the histograms for the cluster in Figure 5.4c. The last column presents the histograms for the cluster in Figure 5.4d, which fits a road-boundary edge with very low noise well.

Figure 5.5 presents the ρ and ϕ histograms for the line segments belonging to the clusters in Figure 5.4. The first column presents the ρ and ϕ histograms for the root cluster in Figure 5.4. As seen, the histograms are very noisy with many peaks. Each of these peaks indicates a group of line segments with similar ρ and/or ϕ values. Such groups can be captured later by sibling clusters. The second column shows the histograms of the second nested cluster shown in Figure 5.4b. The ϕ histogram is clearly normally distributed, while the ρ histogram still has several peaks. The last column presents the histograms for the cluster in Figure 5.4d, which are normally distributed and fit a road-boundary edge well with very low Gaussian noise.

To confirm our assumption of normality, we perform the ShapiroWilk test of normality for each histogram in Figure 5.5 [58]. ShapiroWilk tests the null hypothesis that a given sample population $(x_1, ..., x_n)$ is drawn from a true normally distributed population. The test statistics is given by,

$$W = \frac{\left(\sum_{i=1}^{n} a_i x_{(i)}\right)^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}, \tag{5.8}$$

W-stat	0.767	0.938	0.885	0.862
p-value	6.19×10^{-10}	0.396	0.068	0.269
α	0.05	0.05	0.05	0.05
normal	no	yes	yes	yes

(a) ShapiroWilk test of normality for the ρ histograms in Figure 5.5a

W-stat	0.951	0.972	0.959	0.835
p-value	1.23×10^{-2}	0.62	0.681	0.182
α	0.05	0.05	0.05	0.05
normal	no	yes	yes	yes

(b) ShapiroWilk test of normality for the ϕ histograms in Figure 5.5b

Table 5.1: ShapiroWilk test of normality for the ρ and ϕ histograms in Figure 5.5: (a) ShapiroWilk test for the ρ histograms in Figure 5.5a; (b) ShapiroWilk test for the ϕ histograms in Figure 5.5b. Every column shows the test result of the corresponding histogram in the same left to right order. We test the null hypothesis that the sample data is normally distributed. W-stats is the calculated test statistics. p-value is the probability that the null hypothesis is true. α defines the significance level. The normal row defines the test output, where the null hypothesis is rejected if p-value is less than α, and is accepted otherwise.

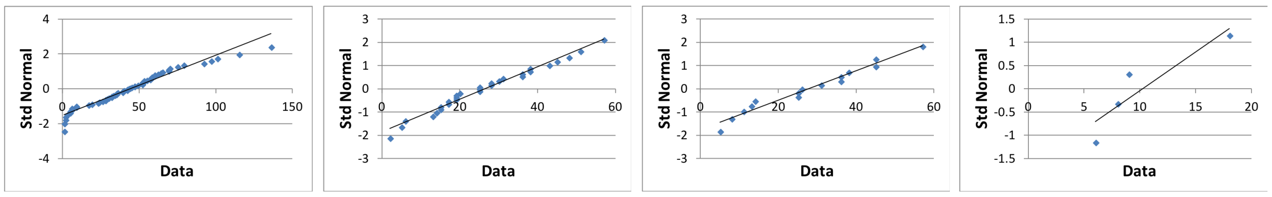
where $x_{(i)}$ is the ith order statistics or the ith smallest value in the given sample population. $\bar{x}$ is the sample mean and the coefficients $a = (a_1, ..., a_n)$ is given by,

$$a = \frac{m^T V^{-1}}{(m^T V^{-1} V^{-1} m)^{1/2}}, \tag{5.9}$$

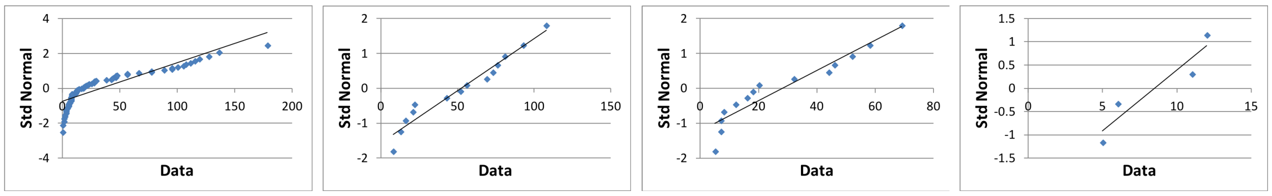
where m and V are the mean and covariance of the order statistics respectively (see [58]). Given the value of the test statistics and following the standard normal distribution, we calculate the p-value which is the probability that the null hypothesis is true. If the p-value is less than a predefined significance level α, the null hypothesis is rejected, otherwise, it is accepted. Here we set α =0.05.

Table 5.1 shows the ShapiroWilk test results. Table 5.1a shows the results for each ρ histogram in Figure 5.5a, whereas Table 5.1b shows the results for each ϕ histogram in Figure 5.5b. The columns in each table show the test outputs in the same left to right order as the corresponding histograms in Figure 5.5. We can see that the test results shown in the bottom row of each table confirm our normality assumption for all nested clusters in Figure 5.4. As expected, the root cluster failed the test.

Figure 5.6 shows the quantile-quantile (Q-Q) plots [58] for visually testing the nor-



(a) Quantile-quantile (Q-Q) plots for comparing ρ histograms in Figure 5.5a against normal distribution.



(b) Quantile-quantile (Q-Q) plots for comparing ϕ histograms in Figure 5.5b against normal distribution.

Figure 5.6: Quantile-quantile (Q-Q) plots for visually testing normality by comparing the probability distribution of each histogram in Figure 5.5 against the normal distribution: (a) The Q-Q plots for each ρ histogram in Figure 5.5a, in the same left to right order; (b) The Q-Q plots for each ϕ histogram in Figure 5.5b, in the same left to right order. If the compared distributions are similar, the points should approximately lie on a line. A reference line is shown in each graph to measure how far the compared distributions deviate from each other.

mality of each histogram in Figure 5.5. Each plot compares the probability distribution defined by each histogram against the normal distribution. If the distributions are similar, the plot points should approximately lie on a line. We show a reference line in each plot to measure how the compared distributions deviate from each other. We can see that all plots have points that approximately follow the reference line except the first plots from the left that correspond to the root cluster.

Figure 5.5 emphasizes the fact that our generative model has expected Gaussian noise. We can also see that hierarchical clustering can generate good cluster summaries for road-boundary edges. Using the generative model, we assume that every cluster c is a subset of a normally distributed population that has its true mean π^* as a road-boundary line. Thus, cluster c has a set of observed random samples drawn from the population, where the cluster mean $\widetilde{\pi}$ is an unbiased estimate of π^*. Our target is to test whether $\widetilde{\pi}$ is a good estimate of π^*. This is done by measuring the quality of a cluster in representing a true boundary. Such quality can be written as:

$$quality\,(c) = P\left(\|\pi^* - \widetilde{\pi}\| \leq \epsilon\right) \tag{5.10}$$

$$= \prod_{\widetilde{x}\in\widetilde{\pi}} \int_{\widetilde{x}-\epsilon}^{\widetilde{x}+\epsilon} \mathcal{N}\left(u|\mu_{\widetilde{x}}, \sigma^2_{\widetilde{x}}\right)du, \tag{5.11}$$

where $quality\,(c)$ has a high values if $\widetilde{\pi}$ fits π^* well and has lower values as $\widetilde{\pi}$ diverges from π^*. Moreover, $\widetilde{\pi}$ is also described as a vector of independent normally distributed random variables with each $\widetilde{x} \in \widetilde{\pi}$ having parameters $(\mu_{\widetilde{x}}, \sigma^2_{\widetilde{x}})$ and $\mu_{\widetilde{x}} \in \pi^*$. For simplicity, we refer to $(\mu_{\widetilde{x}}, \sigma^2_{\widetilde{x}})$ as (μ, σ^2). Given that μ and σ^2 are random variables, $quality\,(c)$ is a random variable with the following expectation:

$$\begin{aligned} E\left(P\left(\|\mu - \widetilde{x}\| \leq \epsilon\right)\right) &= E\left(\int_{\widetilde{x}-\epsilon}^{\widetilde{x}+\epsilon} \mathcal{N}\left(u|\mu, \sigma^2\right)du\right) \\ &= \int_{-\infty}^{\infty} d\mu \int_{0}^{+\infty} d\sigma^2 \int_{\widetilde{x}-\epsilon}^{\widetilde{x}+\epsilon} du \mathcal{N}\left(u|\mu, \sigma^2\right) P\left(\mu, \sigma^2|c\right). \end{aligned} \tag{5.12}$$

Applying conditional probability, we obtain:

$$P\left(\mu, \sigma^2|c\right) = P\left(\sigma^2|\mu, c\right) P\left(\mu|c\right) = P\left(\sigma^2|c\right) P\left(\mu|c\right). \tag{5.13}$$

Now, μ and σ^2 are independent; thus, the integral of Equation 5.12 is:

$$\int_{-\infty}^{\infty} d\mu \int_{0}^{+\infty} d\sigma^2 \int_{\widetilde{x}-\epsilon}^{\widetilde{x}+\epsilon} du \mathcal{N}\left(u|\mu, \sigma^2\right) P\left(\sigma^2|c\right) P\left(\mu|c\right), \tag{5.14}$$

and we can define the inner integral:

$$h(\sigma^2, \widetilde{x} - \mu) = \int_{\widetilde{x}-\epsilon}^{\widetilde{x}+\epsilon} du \mathcal{N}\left(u|\mu, \sigma^2\right) \tag{5.15}$$

$$= \frac{1}{\sqrt{4}}\left(erf\left(\frac{\widetilde{x} - \mu + \epsilon}{\sqrt{2}\sigma}\right) - erf\left(\frac{\widetilde{x} - \mu - \epsilon}{\sqrt{2}\sigma}\right)\right), \tag{5.16}$$

where erf is the error function. We can define $g(\sigma^2) = P(\sigma^2|c)$, $f(\mu) = P(\mu|c)$, and have the inner product:

$$(g.h)(\widetilde{x} - \mu) = \int_0^{+\infty} d\sigma^2 g(\sigma^2) h(\sigma^2, \widetilde{x} - \mu). \tag{5.17}$$

Equation 5.12 then becomes:

$$E\left(P\left(\|\mu - \widetilde{x}\| \leq \epsilon\right)\right) = \int_{-\infty}^{\infty} d\mu \, f(\mu) \, (g.h)(\widetilde{x} - \mu) \tag{5.18}$$

$$= f *_{\mu} (g.h), \tag{5.19}$$

and $*_{\mu}$ is the convolution operator on μ. Equation 5.18 reduces the expected cluster quality to an expression that is easy to evaluate. Notice that Equation 5.19 gives a high expectation value for a given cluster, when the estimated mean $\widetilde{x}$ fits the true mean μ well.

Given the unknown true mean μ and variance σ^2 of a given cluster, we use Bayesian inference to find the posterior probability distribution of σ^2 as follows:

$$P(\sigma^2|c) \propto P(\sigma^2) P(c|\sigma^2) \tag{5.20}$$

$$\propto \frac{1}{\sigma^{n+2}} e^{-\frac{\Sigma_{i=1}^{n}(x_i - \mu)^2}{2\sigma^2}}. \tag{5.21}$$

Here, Jeffrey's prior is used by setting $p(\sigma^2) = \frac{1}{\sigma^2}$. Given the unknown mean μ, we have:

$$P(\sigma^2, \mu|c) \propto \frac{1}{\sigma^{n+2}} e^{-\frac{\Sigma_{i=1}^{n}(x_i - \mu)^2}{2\sigma^2}} \tag{5.22}$$

$$\propto \frac{1}{\sigma^{n+2}} e^{-\frac{\Sigma_{i=1}^{n}(x_i - \widetilde{x})^2}{2\sigma^2}} e^{-\frac{\Sigma_{i=1}^{n}(\mu - \widetilde{x})^2}{2\sigma^2}}. \tag{5.23}$$

The marginal probability distribution for σ^2 is defined as:

$$P(\sigma^2|c) \propto \frac{1}{\sigma^{n+2}} e^{-\frac{\sum_{i=1}^{n}(x_i-\widetilde{x})^2}{2\sigma^2}} \int_{-\infty}^{\infty} e^{-\frac{\sum_{i=1}^{n}(\mu-\widetilde{x})^2}{2\sigma^2}} d\mu \tag{5.24}$$

$$= \frac{1}{\sigma^{n+2}} e^{-\frac{\sum_{i=1}^{n}(x_i-\widetilde{x})^2}{2\sigma^2}} \sqrt{\frac{2\pi\sigma^2}{n}} \tag{5.25}$$

$$\propto (\sigma^2)^{-\frac{(v+2)}{2}} e^{-\frac{vs^2}{2\sigma^2}}, \tag{5.26}$$

where $v = n - 1$ is degree of freedom, and $s^2 = \frac{\sum_{i=1}^{n}(x_i-\widetilde{x})^2}{v}$ is the estimated variance. This defines a scaled inverse chi-squared distribution [66] $\chi^2_{S.inv}(\sigma^2, v, \tau^2)$, which is the distribution of squares of v independent normal random variables that have zero mean and $\tau^2 = s^2$ as the scaling parameter.

$$\chi^2_{S.inv}(\sigma^2, v, \tau^2) \propto (\sigma^2)^{-(1+\frac{v}{2})} e^{-\frac{v\tau^2}{2\sigma^2}} \tag{5.27}$$

The marginal probability distribution for μ can be also defined using Bayesian inference as follows:

$$P(\mu|c) = \int P(\mu, \sigma^2|c) d\sigma^2 \tag{5.28}$$

$$= \int P(\mu|\sigma^2, c) P(\sigma^2|c) d\sigma^2, \tag{5.29}$$

where:

$$P(\mu, \sigma^2|c) \propto P(\mu|\sigma^2, c) P(\sigma^2|c), \tag{5.30}$$

which also implies:

$$\frac{1}{\sigma^{n+2}} e^{-\frac{\sum_{i=1}^{n}(x_i-\widetilde{x})^2}{2\sigma^2}} e^{-\frac{\sum_{i1}^{n}(\mu-\widetilde{x})^2}{2\sigma^2}}$$

$$\propto P(\mu|\sigma^2, c)(\sigma^2)^{-\frac{(v+2)}{2}} e^{-\frac{vs^2}{2\sigma^2}}, \tag{5.31}$$

and this gives:

$$P(\mu|\sigma^2, c) = \mathcal{N}\left(\widetilde{x}, \frac{\sigma^2}{n}\right) \tag{5.32}$$

$$\propto \frac{1}{\sqrt{\sigma^2}} e^{-\frac{n(\mu-\widetilde{x})^2}{2\sigma^2}}. \tag{5.33}$$

Equation 5.29 is then defined as:

$$P(\mu|c) \propto \int_0^\infty \frac{1}{\sqrt{\sigma^2}} e^{-\frac{n(\mu-\widetilde{x})^2}{2\sigma^2}} (\sigma^2)^{-\frac{(v+2)}{2}} e^{-\frac{vs^2}{2\sigma^2}} d\sigma^2 \tag{5.34}$$

$$\propto \int_0^\infty \sigma^{-v-3} e^{-\frac{1}{2\sigma^2}(n(\mu-\widetilde{x})^2+vs^2)} d\sigma^2 \tag{5.35}$$

$$\propto \left(1 + \frac{n(\mu-\widetilde{x})^2}{vs^2}\right)^{-(\frac{v+1}{2})}. \tag{5.36}$$

This is the standard student t distribution [66] defined as,

$$stud(t) \propto \left(1 + \frac{t^2}{2}\right)^{-(\frac{v+1}{2})}, \tag{5.37}$$

where v is again the degree of freedom and $t = \frac{\mu-\widetilde{x}}{s/\sqrt{n}}$. The functions $f(\mu)$ and $g(\sigma^2)$ define the marginal distributions of σ^2 and μ. Given that $g(\sigma^2)$ is a scaled inverse chi-squared distribution and $f(\mu)$ is a student t distribution, confidence intervals can be defined for σ^2 and μ. This allows us to calculate Equation 5.19 numerically. In all experiments, the confidence intervals are set empirically as follows: (1) $\mu = \sigma^2 = 0.95$ and (2) $\epsilon = 0.05$.

Equation 5.19 assigns low confidence to both high-variance clusters and low-variance clusters with a small number of samples. The equation also assigns high confidence to low-variance clusters with many samples. The high-confidence clusters are good candidates for road boundaries, which represent the dominant image edges accumulated from a sequence of frames. Given the cluster confidence, a threshold T is defined to select the top-ranked clusters. In this work, T selects the top 20% ranked clusters. Figure 5.7 shows

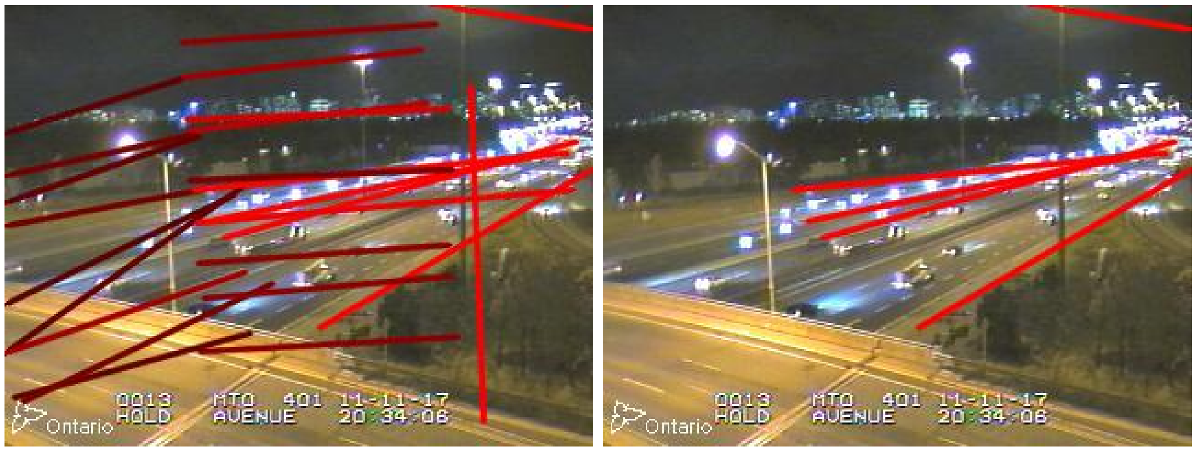


Figure 5.7: Estimated quality for different clusters accumulated from 25 frames: (a) mean of clusters with around 20 line segments; (b) mean lines of clusters with around 60 segments. Increased colour intensity indicates high cluster confidence.

the estimated quality for a set of clusters accumulated over time. Figure 5.7a draws the mean line of each cluster in the top 20% of the highly ranked clusters, where ranking is performed using cluster confidence. Increased colour intensity indicates high cluster confidence. Figure 5.7b shows a set of selected clusters from Figure 5.7a accumulating a large number of line segments. Notice that our quality measure assigns high confidence to the stable edges. Here, the mean line of each cluster is calculated by projecting its line segments on the cluster mean and calculating the extent of all projections. This provides the ability to detect long road edges, even if gaps or missing sections exist due to occlusions or lighting changes.

5.1.5 Pairwise Ranking

Given the set of high-confidence clusters representing the dominant edges accumulated from a stream of traffic-video frames, we detect the road and lane boundaries by applying further filtering operations. To do that, our algorithm applies a heuristic that groups clusters into pairs that are ranked using the camera perspective cues and image vehicular activity. The top-k cluster pairs are then chosen as dominant lane and road boundaries.

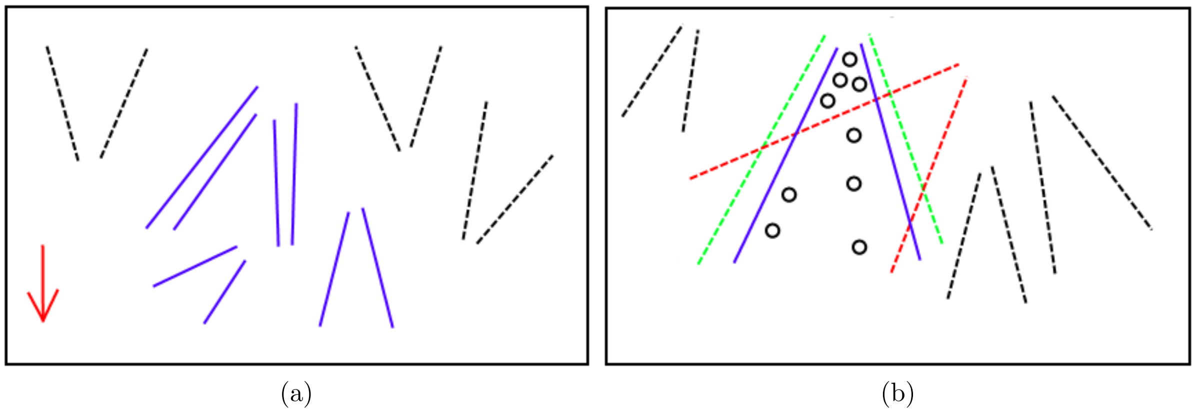


Figure 5.8: Applying perspective filtering and activity ranking on pairs of clusters: (a) Perspective filtering is shown with the red arrow pointing downward; (b) Activity ranking with vehicular traffic is shown as circles. Candidate cluster pairs with solid lines are those preserved after perspective filtering and activity ranking.

Figure 5.8 shows the steps of our pairwise ranking method. It starts by finding the vanishing point for each cluster pair. Then, we use the camera perspective view and the fact that the vanishing point should be located upward in the image to filter out all cluster pairs with the vanishing point pointing downward (see Figure 5.8a). Next, we use a confidence score r_{Conf} and an activity score r_{Activity} to rank the survived pairs, where the rank of cluster pair (i,j) is defined as:

$$r(i,j) = r_{\text{Conf}}^{(i,j)} \times r_{\text{Activity}}^{(i,j)}, \tag{5.38}$$

where:

$$r_{\text{Conf}}^{(i,j)} = \text{Conf}(c_i) \times n_i \times \text{Conf}(c_j) \times n_j, \tag{5.39}$$

$$r_{\text{Activity}}^{(i,j)} = \frac{\#\text{objects within } (i,j) \text{ region}}{\text{area spanned by } (i,j) \text{ region}}, \tag{5.40}$$

where $\text{Conf}(c_i)$ is the cluster c_i quality estimated by Equation 5.10. The area of a cluster pair (i,j) is defined as the image area within the mean lines of clusters c_i and c_j. The quality rank $r_{\text{Conf}}^{(i,j)}$ promotes cluster pairs with low variance and numerous accumulated line segments. The activity rank gives preference to cluster pairs containing high vehicular activity with the assumption that dominant lane and road boundaries enclose vehicular traffic. Notice that vehicular activity is found by detecting the foreground moving objects using background subtraction [112]. Then, our method counts the number of objects lying within the area enclosed by each cluster pair.

5.2 Experimental Results

Experimental results are performed on two datasets of traffic-image sequences: Dataset 1 and Dataset 2. These datasets are captured from traffic-video cameras at several locations along Ontario's Highway 401. Dataset 1 contains 14 video sequences captured from different camera locations at a resolution of 320 × 240. Half of the frames in each sequence (25 frames) are recorded during the daytime and the other half during the nighttime. Dataset 2 contains a very long video sequence of 1,627 frames captured from one camera location at a resolution of 704 × 480. The dataset has both daytime and nighttime frames and exhibits changes in the camera viewpoint. Image sequences in both Dataset 1 and Dataset 2 have a wide range of severe environmental and lighting conditions, such as headlight glare, unlit roads that are only seen by the light of vehicles, shadows, camera shake, changes in traffic density, etc. Moreover, all datasets have frames recorded every 15 minutes to cover the different environmental changes during the day. For ground truth, we manually label the road areas in each image frame in the datasets.

The parameters of our algorithm are set as follows. (1) The number of superpixels is 100. (2) For online clustering, $\beta = 2$, $\lambda = 0.2$, and $\Delta t_c = 3$ (seconds). (3) The maximum clustering tree height is 9.

Our method is compared against the Gabor filter-based method (Gabor method) by [109] and the classification-based scheme (CN24) by [39]. The Gabor method relies on Gabor filters to extract a set of image features that are used to determine the dominant road boundary. The CN24 method uses an offline learning stage to train a deep convolutional neural network for classifying image pixels as either road or non-road. It works by generating a confidence map assigning each pixel the likelihood of being a road. This map is normalized, and pixels with likelihood values greater than 0.5 are considered road. Our algorithm is written in C++ and Go language, and all experiments are performed on a 2.9 GHz quad-core AMD Athlon processor. The algorithm was also implemented as a concurrent streaming pipeline. We will discuss this pipeline later in Section 5.4. The Gabor and CN24 are executed using the authors' code.

To better measure the performance of our algorithm, we define four versions of our algorithm: (1) V1, which does ranking of cluster pairs using only the quality estimate relying on the χ^2 and student t tests; (2) V2, which uses the ranking of V1 plus filtering using the perspective cues; (3) V3, which uses the ranking and filtering of V2 and performs activity ranking; and (4) V4, which is similar to V3 but uses our online hierarchical-clustering method. The V1, V2, and V3 methods use the incremental bottom-up hierarchical-clustering approach.

5.2.1 Dataset 1

Figure 5.9 shows a visual comparison between the results of V1, V2, V3 and V4 methods. We can notice the poor results of V1 and V2 methods. V1 only performs statistical ranking of cluster pairs, which gives preference to cluster pairs that have a large number of samples indicating that they persisted over a sequence of frames. These pairs, however, may be produced by noise or man-made structures in non-road areas. Due to the road perspective, most of the man-made structures exist in the upper part of images, with the lower part mostly covered by road areas. We can see that V1 did not detect any road region in Brock Road location. The detected regions in DVP and Reynolds covers the road area, however, the detected boundaries are attracted to non-road structures and do not align with the actual road boundaries. V1 also has the detected road boundary in Avenue location pointing downward against the actual road perspective. The V2 method extends V1 by filtering using perspective cues. V2, however, did not detect road regions in Brock Road, DVP, and Reynolds, because the top-ranked boundaries are again attracted to non-road structures. Both V1 and V2 performed well on the Liverpool location, however, V2 outperformed V1 on Avenue location. These results match well the outcome of the precision and recall comparisons in Table 5.2, where V1 provides higher precision and lower recall than V2. The reason is that V2 has more cluster pairs than V1 with some pairs covering the road area, while aligned with edges in non-road structures.

Figure 5.10 shows the experimental results of V4 against the results of Gabor [109] and CN24 [39] methods on eight traffic-image sequences generated by different camera locations during the daytime. The first column presents the ground truth generated by manually labelling the road regions. The second and third columns show the results from [109] and [39], respectively. The last three columns show the top three ranked dominant boundary lines generated by V4. Note that the traffic images in the Avenue camera location show a partially occluded highway. Despite such challenges, V4 can detect the correct dominant road boundary.

Figure 5.11 also shows the results of V4 against the results of Gabor [109] and CN24 [39] on eight traffic-image sequences recorded at different camera locations during the nighttime. The first column shows the manually labelled ground truth. The second and third columns show the results from [109] and [39], respectively. The last three columns show the top three ranked dominant road boundaries computed by V4. Note that the road regions are barely visible in some nighttime image sequences. For example, the fifth and eighth rows show dark roads that are only visible by vehicle headlights. Despite these severe environmental challenges, V4 provides a better estimate of the dominant road boundary than the Gabor and CN24 methods.

We also perform comparisons of runtime, precision, and recall [14] between the V1, V2, V3, V4, Gabor, and CN2 methods. Precision is calculated as a/r, where a is the area of the intersection between the ground truth and the detected road boundary, and

r is the total area of the detected road boundary. Recall is measured as a/g, where g is the area of the ground truth.

Figure 5.12 shows comparisons between our V4 method and other techniques. The number shown on the horizontal axis defines the size of the sliding window used for accumulating evidence. Clearly, V4 outperforms the Gabor, CN24, V1, and V2 methods on both precision and recall measures, whereas V4 gives similar performance as V3. For runtime comparison, V4 is shown to be significantly faster than Gabor, CN24, and V3. However, V1 and V2 are not included in the runtime comparison for their clear poor precision and recall accuracy. Notice that V3 and V4 have the same perspective filtering and pairwise ranking steps. The importance of these steps is clear from comparing the performance of the V1 and V2 methods against the V3 method.

Table 5.2 lists the average μ and standard deviation σ of the precision and recall measures. For average precision, the V3 and V4 methods achieve 77% and 80%, respectively, whereas Gabor and CN24 achieve 60% and 40%, respectively. For average recall, V3 and V4 achieve 77% and 75%, whereas Gabor and CN24 achieve 52% and 41%, respectively. The V1 and V2 methods have low precision and recall values. This proves that road-activity cues are important in identifying road regions. Additionally, note that V3 achieves the best accuracy. This is due to the agglomerative bottom-up hierarchical step used in V3, which builds a true clustering hierarchy. The V4 method builds an online top-down hierarchical clustering, which approximates the bottom-up hierarchy. However, V4 is significantly faster than V3, as V4 is 300 times faster than CN24 and 800 times faster than Gabor and V3.

5.2.2 Dataset 2

Dataset 2 contains a single long traffic-video stream recorded by one camera over a period of three days. The camera moves and changes its viewing direction between eight different traffic scenes. Camera movement is done manually by human operators or automatically

Methods	Precision	Recall	Average Runtime (seconds)
V4 [81]	$77\%_{\pm 24\%}$	$75\%_{\pm 25\%}$	0.05
V1 (Conf)	$24\%_{\pm 36\%}$	$45\%_{\pm 39\%}$	0.05
V2 (Conf+Persp)	$16\%_{\pm 25\%}$	$48\%_{\pm 37\%}$	0.05
V3 [75]	$80\%_{\pm 25\%}$	$77\%_{\pm 37\%}$	40
Gabor [109]	$60\%_{\pm 31\%}$	$52\%_{\pm 31\%}$	40.2
CN24 [39]	$40\%_{\pm 21\%}$	$41\%_{\pm 34\%}$	18.8

Table 5.2: Dataset 1. A summary of the mean and standard deviation statistics for the precision and recall comparisons shown in Figure 5.12. Moreover, we show the average runtime (in seconds) of each compared method.

by a predefined schedule. For Gabor and CN24 methods, the camera view changes have no effect on estimating road regions, as these methods process a single frame at a time. Our V3 and V4 methods have to deal with the challenge of changes in camera view as they collect evidence over a sequence of frames. Furthermore, V3 handles this challenge in an ad-hoc fashion by rebuilding the clustering hierarchy from scratch after every 25 frames. However, V4 applies online clustering and can forget old information.

Figure 5.13 shows a visual comparison between V4, CN24, Gabor, and V3 methods on example frames from Dataset 2. Note that the frames show different viewing directions of the monitored traffic scene. The first column shows the ground truth. The remaining columns from left to right show the road boundaries estimated by Gabor, CN24, V3, and V4, respectively. Each row indicates a certain camera-viewing direction. Gabor gives the lowest performance. However, CN24 performs well on daytime frames and poorly on nighttime frames. Moreover, V4 provides the best performance and outperforms V3 when encountering changes in camera-viewing direction. This important observation is also verified by the performance comparison shown in Table 5.3. For average precision, the V4, V3, CN24, and Gabor methods achieve 90%, 47%, 64%, and 78%, respectively. For average recall, the V4, V3, CN24, and Gabor methods achieve 71%, 76%, 49%, and 43%, respectively. These performance numbers confirm that our method can better deal with camera movement. Moreover, the V4 method achieves significantly faster runtimes than the other compared methods. Furthermore, V4 has a 0.13-second average runtime

Methods	Precision	Recall	Average Runtime (seconds)
V4 [81]	$90\%_{\pm 11\%}$	$71\%_{\pm 20\%}$	0.13
V3 [75]	$47\%_{\pm 33\%}$	$76\%_{\pm 22\%}$	200
Gabor [109]	$78\%_{\pm 14\%}$	$43\%_{\pm 12\%}$	54.9
CN24 [39]	$64\%_{\pm 26\%}$	$49\%_{\pm 43\%}$	81.5

Table 5.3: Dataset 2. Mean and standard deviation statistics for the precision and recall comparisons on Dataset 2. The last columns show the average runtime in seconds for each method.

per frame for Dataset 2. This gives a speed increase of about 1,500 times over V3, 600 times over CN24, and 400 times over Gabor.

5.2.3 Sensitivity Analysis

Figure 5.14 shows a sensitivity analysis of the performance of the V4 algorithm against the user-defined parameters. For each experiment, we set the parameters to their default values except for the studied parameter and the number of superpixels N. The top row shows precision and recall versus the decay rate λ used by online hierarchical clustering. This rate controls how fast the algorithm forgets stale edges. The larger the rate, the faster the algorithm to forget state information. We can notice that the recall increases with increasing the number of superpixels, whereas the precision decreases as λ increase. We expect the decrease in precision because the algorithm has a faster forgetting rate and loses sufficient evidence for road boundaries. The middle row of Figure 5.14 presents precision and recall versus different values of the parameter defining the minimum length of segments used for polygon approximation. We again notice that the recall increases while increasing the number of superpixels, whereas precision decreases for all values of the number of superpixels when the minimum segment length is less than 10 pixels or more than 20 pixels. The reason is that a small segment length results in clusters with high noise because of selecting a large number of small segments, whereas a large segment length results in clusters with low evidence because of discarding a large number of segments. The bottom row of Figure 5.14 shows precision and recall versus the ranking threshold used to select the high confidence clusters. We obtain similar results for the recall. It increases while increasing the number of superpixels. For precision, we can see that it increases while increasing the number of superpixels as we lower the value of the ranking threshold. For example, when the ranking threshold is 10%, we can achieve up to 91% precision at N =150. As the ranking threshold increases, the precision decreases for most N values. The reason is that adding more clusters to the high confidence set increases the noise encountered when estimating the dominant pair of clusters representing the road boundary.

5.3 Discussion

The traffic-video datasets used for experimental evaluation show several challenging environmental and lighting conditions encountered in real-world traffic scenes. For example, challenging lighting conditions can be seen in the video sequences recorded from the Brock Road, Yonge, and Liverpool cameras. The side lane in the Bayview location poses a challenge in correctly estimating the road boundary. The Highway 127 location is an example of unlit roads recorded during the night, where the road boundary can be barely recognized using vehicle headlights. The effect of headlight reflections is clear in the road regions of Bathurst, Bayview, and Whites locations under wet environments. Avenue location also is a challenging scenario, where the main monitored highway is occluded by another road. The results confirm that the V4 and V3 methods can detect road boundaries and handle such challenging conditions.

The precision and recall comparison shown in Table 5.2 confirms that road-activity features are important in accurately estimating road boundaries. One can see the poor performance of the V1 and V2 methods that neglect activity cues. The V1 method can only find stable edges; however, these edges may belong to different scene structures other than road boundaries. The V2 method performs perspective filtering, yet cannot identify road regions. The V3 method extends the V2 method by applying activity ranking and provides a good estimate of road boundaries. The V4 method provides comparable accuracy to the V3 method on Dataset 1, at a significantly faster runtime performance. Both the V3 and V4 methods outperform the CN24 and Gabor methods.

The V4 method achieves its faster runtime performance by applying online (top-down) hierarchical clustering, which approximates the bottom-up hierarchical-clustering method used by the V3 method. Online clustering also allows the V4 method to adapt well to changes in the traffic scene. The V3 method cannot forget old information, and a new clustering hierarchy should be generated for every 25 frames. A limitation of the V3 method is that it cannot handle changes in camera-viewing directions. Another limitation of the V3 method is the poor accuracy on segments of traffic-video streams that have unclear road structures or few activities. This is because the V3 method independently processes sequences of 25 frames each, so the initial frames of each sequence do not have enough historical evidence. The V3 method also has high computational time, which rapidly increases when processing high-resolution images due to the accumulation of a large number of edges. The V4 method overcomes the limitations of the V3 method using the fast online hierarchical-clustering algorithm, which smoothly forgets old evidence over time. One can see that the V4 method provides similar precision and recall accuracy to the V3 method on Dataset 1 and outperforms the V3 method on Dataset 2.

It is also clear that the Gabor method has poor performance on Dataset 1. This is because the dataset contains low-resolution images with large noise created by the severe environmental conditions. This causes a noisy output for Gabor filtering, which was used by [109] to estimate the road vanishing point and dominant boundary. Thus, the vanishing point estimate is wrong in most cases, which leads to the poor performance in [109]. Moreover, CN24 also has poor performance on some sequences of Dataset 1. CN24 was trained on a road dataset that contains two sets of sequences. The first set contains clear and visible daylight road regions. The second set contains sequences with severe lighting conditions that have strong shadows and dark roads simulating night scenes. Our aim from the comparison against CN24 is to see how robust a deep learning model, that is trained on a given road dataset, is to deal with environmental conditions in another similar dataset. Although fine-tuning the CN24 model on our dataset may improve its performance, the aim of this work is to develop a method that can deal with unseen challenging conditions that appear in continuous and possibly infinite traffic video streams. Figure 5.10 shows that CN24 performed well on some daylight sequences such as BrockRoad and Liverpool, however it performs poorly on other daylight sequences such as Whites, Yonge, and DVP. CN24 also has an inaccurate prediction of large regions in Reynolds and Hwy-137. Figure 5.11 shows that CN24 failed in most night sequences; however, it was able to detect part of the road regions on some sequences such as Reynolds, Hwy-137, and Montreal. Notice that it is hard to train CN24 on these sequences because major parts of the road regions are dark. In addition, the strong light reflections found in locations such as Whites and Yonge roads are also hard to be detected by CN24. The reason is that these regions exactly match lights coming from road lighting poles and building in non-road areas. Another important problem is that CN24 is trained on individual images while ignoring road activity. This work showed that road activity is an important clue in detecting road regions. So, a possible future direction is to extend the CN24 method by incorporating motion information, such as optical flow maps.

5.4 Concurrent Streaming Implementation

In this section, we express our online road-boundary detection algorithm [81] in the proposed stream algebra. 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 clustered 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 of T are then selected. This generates a ranked cluster 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 generates 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 pairwise 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 line-pair 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 the recent list of detected centroids with every line-pair list $Q_i \in Q$ to produce the stream $U = \{U_i | i = 0, 1, 2, ...\}$. Next, activity ranking is applied on U to construct the ranked pairwise 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 pairwise list $J_i \in J$.

5.4.1 Description Using Algebra

Now, we describe this vision pipeline using the algebra (Figure 5.15a). The data types defined by the algorithm are:

$$\texttt{Frame} : \texttt{2DImage}; \quad \texttt{Video} : \mathbf{S}\langle\texttt{Frame}\rangle; \quad \texttt{Point} : \mathbb{R}^2\ \texttt{Edge} : \mathbb{R}^6; \quad \texttt{Cluster} : \mathbb{R}^4 \times \texttt{Edge};$$

$\texttt{RCluster} : \texttt{Cluster} \times \mathbb{R}$

$\texttt{Pair} : \texttt{Edge} \times \texttt{Edge}; \quad \texttt{RPair} : \texttt{Pair} \times \mathbb{R}$

$\text{Hierarchy} : \textsc{Tree}\,\langle\texttt{Cluster}\rangle$

$\texttt{Params} : \textsc{List}\,\langle\texttt{Parameter}\rangle,$

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:

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

where n is the number of edges in the cluster, t is the last update time of the cluster, 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 $c : \texttt{Cluster}$ and α is the statistical rank of c. A `Pair` is a pair of edge segments. The `RPair` is defined as (p, β), where $p : \texttt{Pair}$, and β is the activity rank of p. A Hierarchy is a tree of clusters. Finally, `Params` is a list of parameters.

We start by copying the incoming video stream $V \in \texttt{Video}$ into two streams V' and V_1 using a Copy operator:

$$V', V_1 \triangleq \textsc{Copy}(2)(V). \tag{5.41}$$

We then apply the Cut operator on V' to obtain the streams V_2 and V_3:

$$V_2, V_3 \triangleq \textsc{Cut}()(V'). \tag{5.42}$$

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 perform superpixel segmentation and contour approximation on every frame in V_2 using the function $f_1 : \textsc{List}\,\langle\texttt{Frame}\rangle \times \texttt{Params} \to \textsc{List}\,\langle\texttt{Edge}\rangle$. This function takes a list of parameters $p_1 : P$ that define the number of superpixels N. In addition, f_1 and p_1 are used with the Map operator to define the stream $E : S\,\langle\textsc{List}\,\langle\texttt{Edge}\rangle\rangle$:

$$E \triangleq \textsc{Map}(d_1) \circ \textsc{Map}(f_1, p_1)(V_2). \tag{5.43}$$

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. We use the Map operator as a viewer to display the output edges using the function $d_1 : X \to X$ (see Figure 5.15b). The function d_1 forwards its input to output while displaying the image content of incoming elements. We then define the function as follows:

$g_1 : \text{Hierarchy} \times \textsc{List}\,\langle\texttt{Edge}\rangle \times \texttt{Params} \quad \to \text{Hierarchy} \times \textsc{List}\,\langle\texttt{Cluster}\rangle$

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

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 define the parameter vector q_1, which contains, for example, the tree height, number of children per node, etc. The g_1 function is used with the Reduce operator to generate the stream $H : S\,\langle\textsc{List}\,\langle\texttt{Cluster}\rangle\rangle$:

$$H \triangleq \textsc{Reduce}(\text{Empty-Tree}, g_1, q_1)(E). \tag{5.44}$$

Given the H stream, we apply a ranking function $f_2 : \textsc{List}\,\langle\texttt{Cluster}\rangle\ \times\texttt{Params} \to$

$\textsc{List}\langle\texttt{RCluster}\rangle$ to statistically rank clusters based on the variance and number of samples. This function together with an empty list of parameters p_2 can be used with the Map operator to generate the ranked cluster stream $F : S\langle\textsc{List}\langle\texttt{RCluster}\rangle\rangle$:

$$F \triangleq \textsc{Map}(f_2, p_2)(H). \tag{5.45}$$

We then apply a threshold function $f_3 : \textsc{List}\langle\texttt{RCluster}\rangle \times \texttt{Params} \to \textsc{List}\langle\texttt{RCluster}\rangle$ on every element in F to choose clusters with ranks larger than a threshold T. A parameter list p_3 is defined for the threshold T. The Map operator parametrized by f_3 and p_3 can then generate the stream $C : S\langle\textsc{List}\ \langle\texttt{RCluster}\rangle\rangle$:

$$C \triangleq \textsc{Map}(f_3, p_3)(F). \tag{5.46}$$

The stream C is converted to a line stream by applying a function $f_4 : \textsc{List}\langle\texttt{RCluster}\rangle \times \texttt{Params} \to \textsc{List}\langle\texttt{Edge}\rangle$. This function maps every cluster $C_{i,j} \in C_i$ into its mean line. The function together with an empty list of parameters p_4 can be used with the Map operator to construct the line stream $L : S\langle\textsc{List}\langle\texttt{Edge}\rangle\rangle$:

$$L \triangleq \textsc{Map}(f_4, p_4)(C). \tag{5.47}$$

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

$$P \triangleq \textsc{Map}(f_5, p_5)(L). \tag{5.48}$$

After that, we define a filtering function $f_6 : \textsc{List}\langle\texttt{Pair}\rangle \times \texttt{Params} \to \textsc{List}\langle\texttt{Pair}\rangle$ 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 f_6 with an empty list of parameters p_6 for use with the Map operator to construct the filtered pairwise stream $Q : S\langle \textsc{List}\langle \texttt{Pair}\rangle\rangle$:

$$Q \triangleq \textsc{Map}(f_6, p_6)(P). \tag{5.49}$$

Now, we need to perform activity ranking on every pair $Q_i \in Q$. To define scene activity, we 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 [106] on every frame in V_3 to obtain a set of foreground regions. We then output the centroids of these regions. This is performed using the function $f_7 : \texttt{Frame} \times \texttt{Params} \to \textsc{List}\langle\texttt{Point}\rangle$. This function together with an empty list p_3 can be used with the Map operator to construct the centroid stream $O : S\langle\textsc{List}\langle\texttt{Point}\rangle\rangle$:

$$O \triangleq \textsc{Map}(f_7, p_7)(V_3). \tag{5.50}$$

We record the extracted centroids over a temporal window with interval $\triangle t$. Thus, we define the function
$g_2 : \textsc{List}\langle\texttt{Point}\rangle \times \textsc{List}\langle\texttt{Point}\rangle \times \texttt{Params} \to \textsc{List}\langle\texttt{Point}\rangle \times \textsc{List}\langle\texttt{Point}\rangle$:

```
$g_2(u, x, p) =$ { for all $z \in u$
                       if (now() − arrivaltime($z$) $\geq p. \triangle t$) then
                           $u = u \ominus z$   //remove $z$ from $u$
                   $u = u \oplus x.v$   //append points $x.v$ to $u$
                   return$(u, u)$ }.
```

This function is added to a list g_2 with parameter vector q_2 that contains only the $\triangle t$ parameter. The function is used with the Reduce operator to generate the activity stream

$A : S\langle \text{LIST}\langle \texttt{Point}\rangle\rangle$:

$$A \triangleq \text{REDUCE}(\text{Empty-List}, g_2, q_2)(O). \tag{5.51}$$

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

$$U \triangleq \text{LEFTMULT}()(Q, A). \tag{5.52}$$

We then apply a ranking function $f_8 : \text{LIST}\langle \texttt{Pair}\rangle \times \text{LIST}\langle \texttt{Point}\rangle \times \texttt{Params} \rightarrow \text{LIST}\langle \texttt{RPair}\rangle$ that ranks every line pair using its attached centroids and generates a list of ranked pairs. The function, along with an empty list of parameters p_8, can be used with the Map operator to build the ranked pairwise stream $J : S\langle \text{LIST}\langle \texttt{RPair}\rangle\rangle$:

$$J \triangleq \text{MAP}(f_8, p_8)(U). \tag{5.53}$$

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. The f_9 function, along with an empty list of parameters p_9, can be used with the Map operator to construct the dominant road-boundary stream $B : S\langle \texttt{RPair}\rangle$:

$$B \triangleq \text{MAP}(f_9, p_9)(J). \tag{5.54}$$

Remember that the V_1 stream is a copy of the input stream generated by Equation 5.41. The Map operator is used as a viewer to display the V_1 stream using the function $d_2 : X \rightarrow X$ (see Figure 5.15b). We synchronize the B stream with the V_1 video stream using the LeftMult operator to construct the output stream $Y : \texttt{Frame} \times \texttt{RPair}$:

$$Y \triangleq \text{LEFTMULT}()(V_1, B). \tag{5.55}$$

Afterwards, we apply the expression:

$$\textsc{Ground}() \circ \textsc{Map}(d_3)(Y), \tag{5.56}$$

first, to view the estimated dominant road boundary on every frame using the drawing function $d_3 : X \to X$, then to release the stream resources using the Ground operator.

Throughput versus latency analysis. We study the throughput and latency of the streaming pipeline of our road boundary detection algorithm. Notice again that throughput is the inverse of the period which is the slowest operator computation or communication time in the pipeline (see Section 3.3). Figure 5.15a shows the algebraic description of this pipeline. We ignore the communication cost between operators because the pipeline is tested on a single machine with data communicated between operators using pointers. The boundary detection branch starting with stream V_2 and ending with stream B has a latency of 0.049 seconds and a period of 0.012 seconds, on the low-resolution images of Dataset 1; whereas for the standard resolution images of Dataset 2, these numbers are 0.125 and 0.03 seconds, respectively. The activity detection branch starting with stream V_3 and ending with stream A has a latency of 0.013 seconds and a period of 0.01 seconds on Dataset 1, whereas for Dataset 2, these numbers are 0.032 and 0.025 seconds, respectively. The bottom branch starting with stream V_1 and ending with output stream Y has a latency of 0.002 seconds and a period of 0.001 seconds, on Dataset 1, whereas for Dataset 2, these numbers are 0.004 and 0.002 seconds, respectively. Notice that the activity detection branch is decoupled from the other branches. The boundary detection and the bottom branches are synchronized together by the Copy operator. So, the overall period and latency are given by the slowest branch, which is the boundary detection branch in this case. The overall latency and period are 0.049 and 0.012 seconds on Dataset 1, whereas for Dataset 2, these numbers are 0.125 and 0.03 seconds, respectively. If Copy is replaced by Cut with V' stream as the asynchronous output, then the boundary detection branch is decoupled from the bottom branch. In this case, the overall latency and period are given by the bottom branch. The overall latency and period become 0.002 and 0.001 seconds on Dataset 1, whereas for Dataset 2, these numbers are 0.004 and 0.002 seconds, respectively. This study shows that our algebra can manipulate the data flow rates by either decoupling or synchronizing slower and faster sections of streaming pipelines.

5.4.2 Implementation

After describing our case study in the stream algebra, we can easily implement it using our stream-algebra implementation (see Section 3.4). It is a simple one-to-one mapping. The following Go language code implements the road-boundary detection pipeline of Figure 5.15a,

```
1   g := NewGraph("boundary-detection")
2   g.Source($u_0,h$).Copy(2, "cp")
3   g.Cut("ct")
4   g.Map($f_1,p_1$,"m1").Map($d_1$).Reduce($u_1,g_1,q_1$).Map($f_2,p_2$).Map($f_3,p_3$)
5   .Map($f_4,p_4$).Map($f_5,p_5$).Map($f_6,p_6$, "m1e")
6   g.Map($f_7,p_7$, "m2s").Reduce($u_2,g_2,q_2$, "r2e")
7   g.LeftMult("lm3").Map($f_8,p_8$).Map($f_9,p_9$, "m3e")
8   g.Map($d_2$, "m4")
9   g.LeftMult("lm5").Map($d_3$).Ground()
10  g.LinkOut("cp","ct","m4")
11  g.LinkOut("ct","m1s","m2s")
12  g.LinkIn("lm3","m1e","r2e")
13  g.LinkIn("lm5","m4","m3e")
14  g.Execute()
```

We start by creating a new operator graph g on line 1. Then, line 2 defines a pipeline branch that generates the source of the video stream V using the Source operator. The Copy operator is then added to copy the V stream into V' and V_1 streams (see Equation 5.41). This operator has the unique name cp We define unique operator names for the start and end operators of every pipeline branch in Figure 5.15a. These names will be used later to link all branches together. Line 3 adds the Cut operator defined in Equation 5.41 to the graph g. Line 4 defines the top branch of Figure 5.15a, which extracts edges and performs online clustering to generate the set of candidate road-boundary clusters. Line 5 defines the foreground segmentation branch that contains the Map and Reduce operators of Equations 5.50 and 5.51, respectively. Line 6 defines a pipeline branch that starts with the LeftMult operator of Equation 5.52. The output of LeftMult is then processed using the Map operators of Equations 5.53 and 5.54, respectively. Line 7 then defines the Map operator that views the V_1 stream in Figure 5.15a. The output branch is defined in line 8 and starts with the LeftMult operator of Equation 5.55. This operator is followed by the Map and Ground operators to implement Equation 5.56. Lines 9 to 12 links all defined branches together to construct the streaming pipeline of the road-boundary detection algorithm. Finally, the `Execute()` function is called in line 13 to run the pipeline.

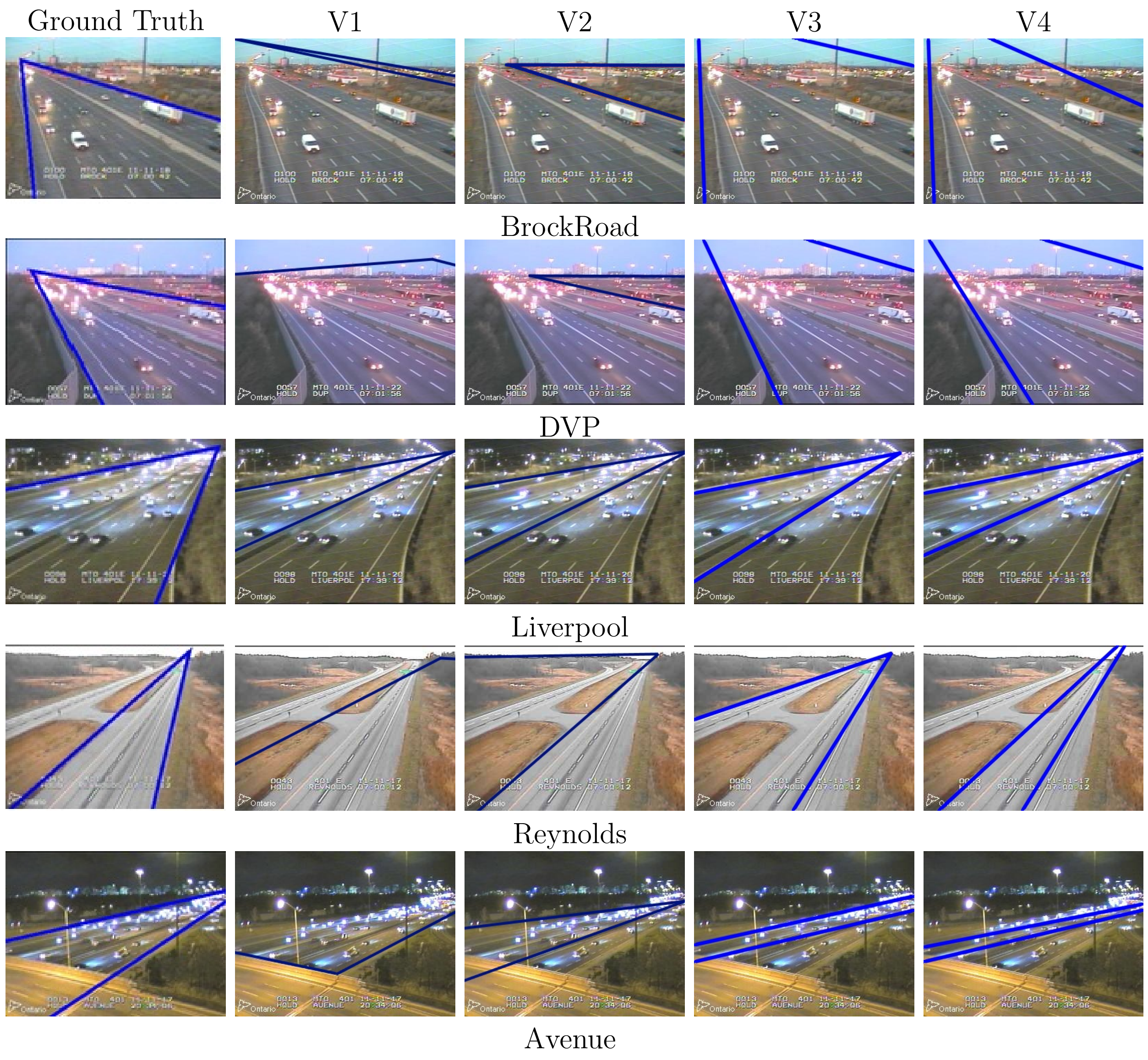


Figure 5.9: A visual comparison between V1, V2, V3 [75], and V4 [81] methods. The V1 method (Conf.) shows the results of our method using only the cluster confidence ranking. The V2 method (Conf + Persp) shows the results of using perspective filtering with cluster confidence ranking. The V3 method [75] uses the incremental bottom-up clustering approach. Note that V4 has comparable accuracy to the V3 method while boosting faster runtime performance.

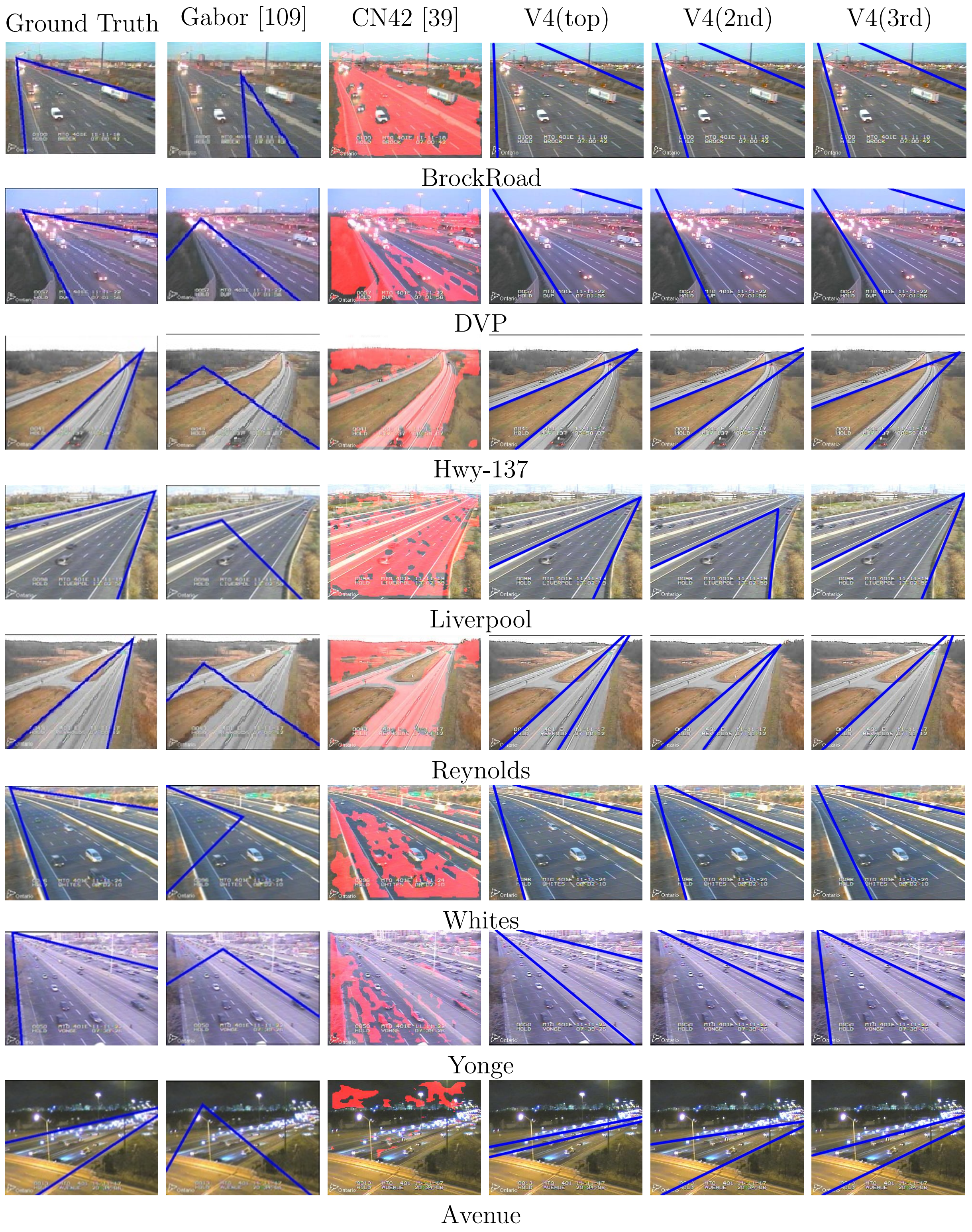


Figure 5.10: Results of applying the V4 method in eight camera locations from Dataset 1 during the daytime. The results are compared against the Gabor-based method [109] and the CN24 method [39]. Each camera location has a different daytime lighting condition, with one camera location having an occluded road. The first column shows the ground truth. The second, third, and fourth columns show the results of [109], [39], and our method, respectively. Moreover, CN24 classifies the road regions as either road or non-road, and the road regions are highlighted in red. The last two columns show the second and third ranked cluster pairs or road boundary.

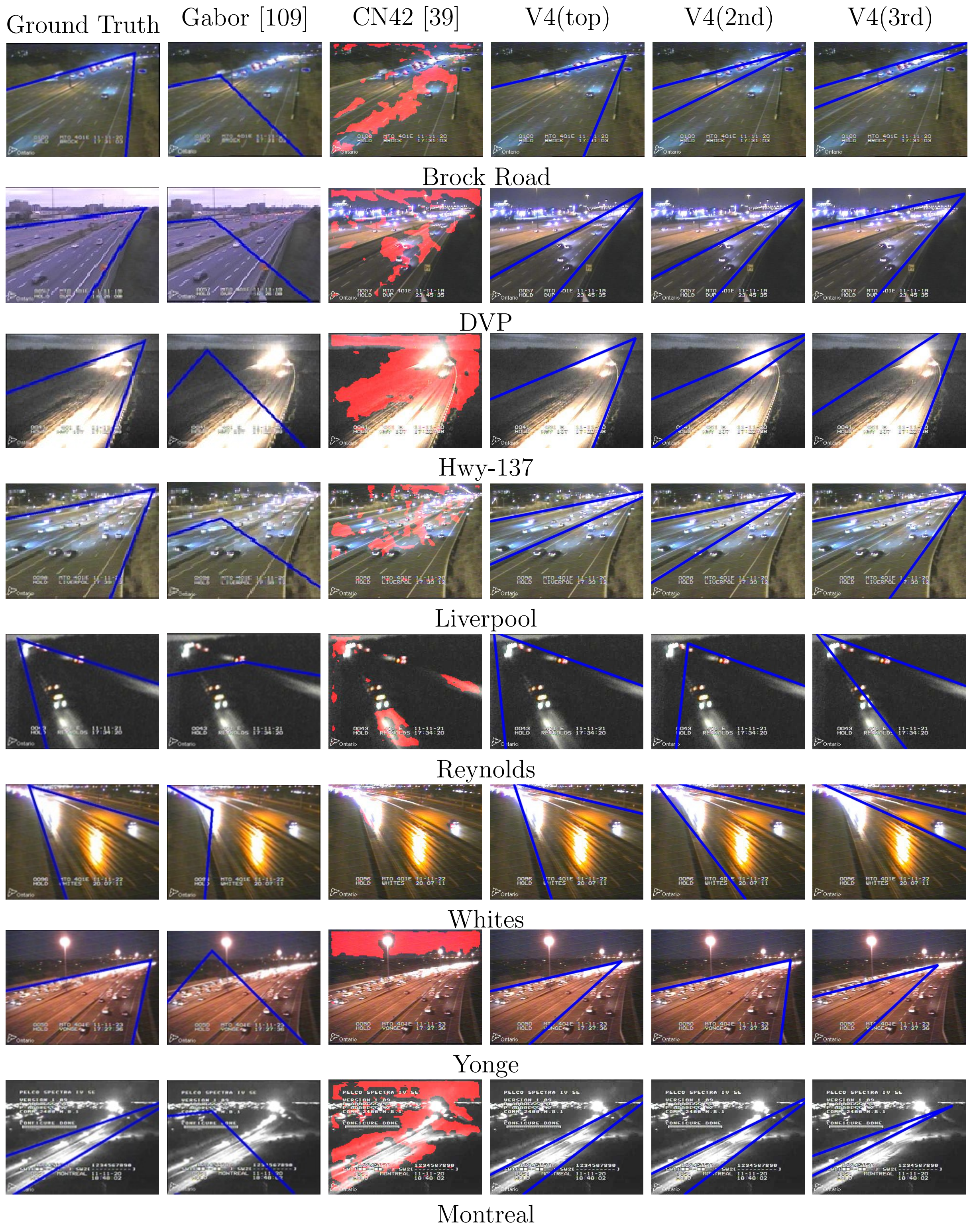
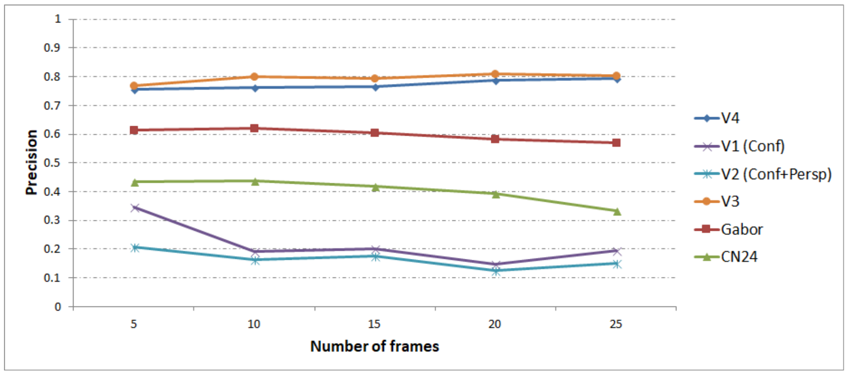
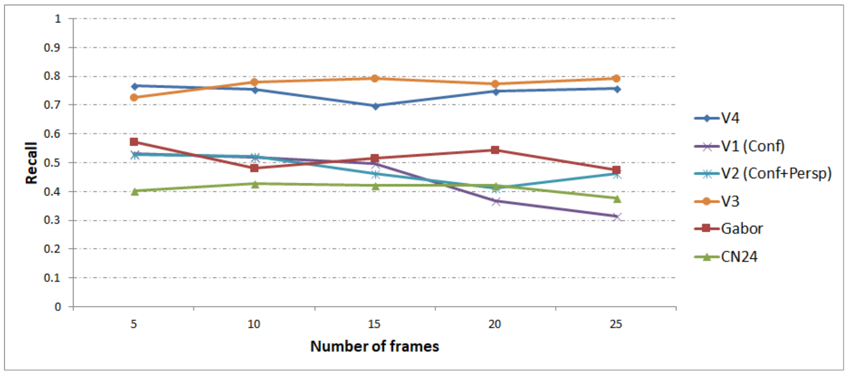


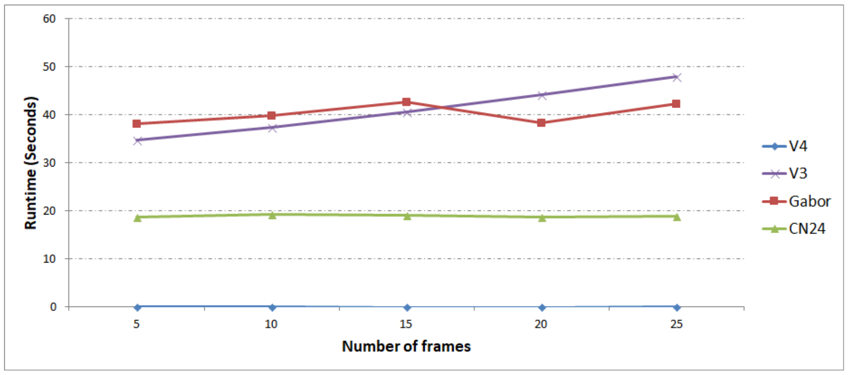
Figure 5.11: Results of applying the V4 method in eight camera locations from Dataset 1 during the nighttime. The results are compared against the Gabor-based method [109] and the CN24 method [39]. Each camera location has a different nighttime lighting condition, with one camera location having an occluded road. The first column shows the ground truth. The second, third, and fourth columns show the results of [109], [39], and our method, respectively. Moreover, CN24 classifies road regions as either road or non-road, and the road regions are highlighted in red. The last two columns show the second and third ranked cluster pairs or road boundary.



(a) Precision



(b) Recall



(c) Runtime

Figure 5.12: Dataset 1. A comparison of the precision, recall, and runtime of the V4 method [81] against the Gabor filter-based method [109] and the classification method (CN24) by [39]. The V1 method (Conf.) shows the results of our method using only the cluster confidence ranking. The V2 method (Conf + Persp) shows the results of using perspective filtering with cluster confidence ranking. The V3 method [75] uses the incremental bottom-up incremental clustering approach. Note that V4 has a comparable accuracy to the V3 method, while boosting faster runtime performance.

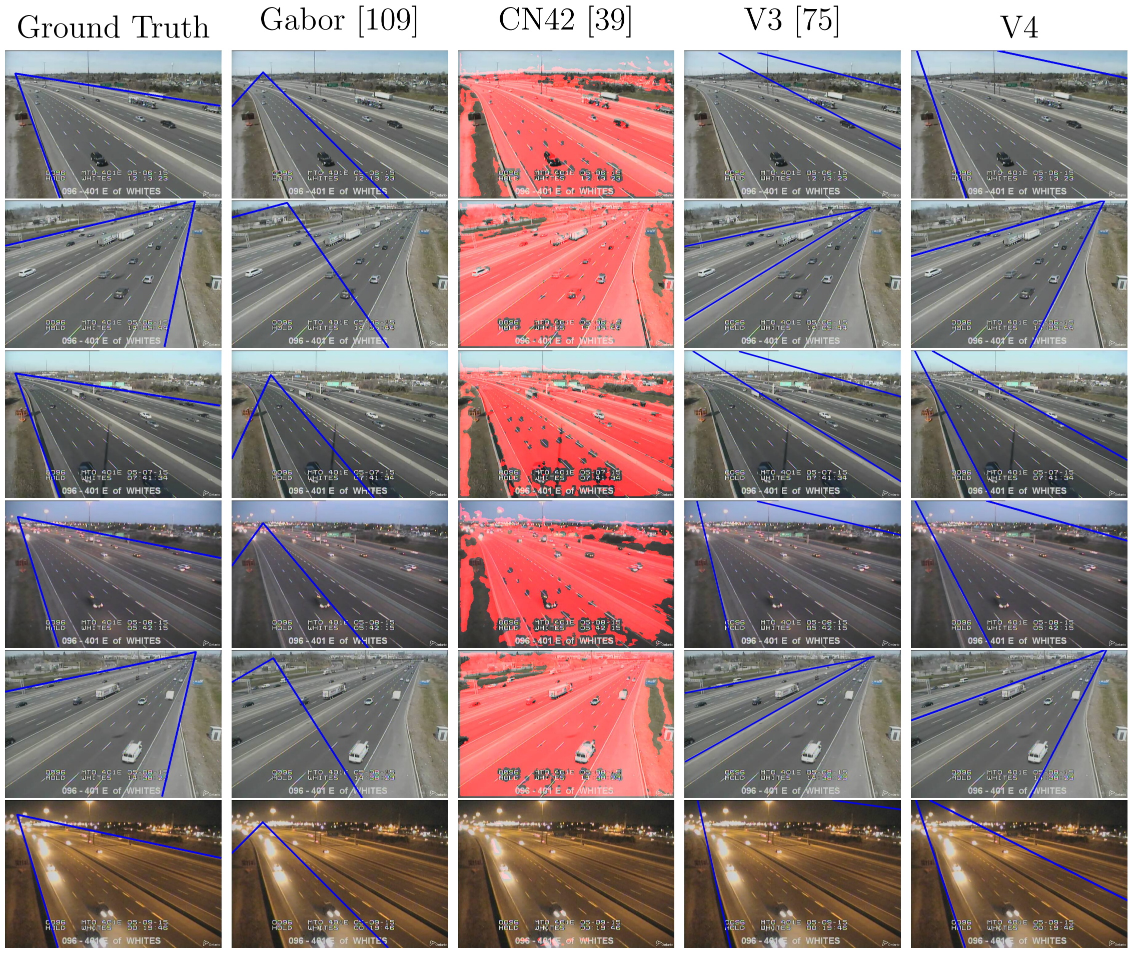


Figure 5.13: Dataset 2. Results of the V4 method [81] in six different camera-viewing directions of the long video-stream dataset, compared to the Gabor-based method [109], the deep learning CN24 method [39], and the V3 algorithm [75]. The first column is the ground truth, and the next three columns show the results from [109], [39], and [75], respectively. Notice that the classified road regions from [39] are highlighted in red. The last column is our top-ranked pair.

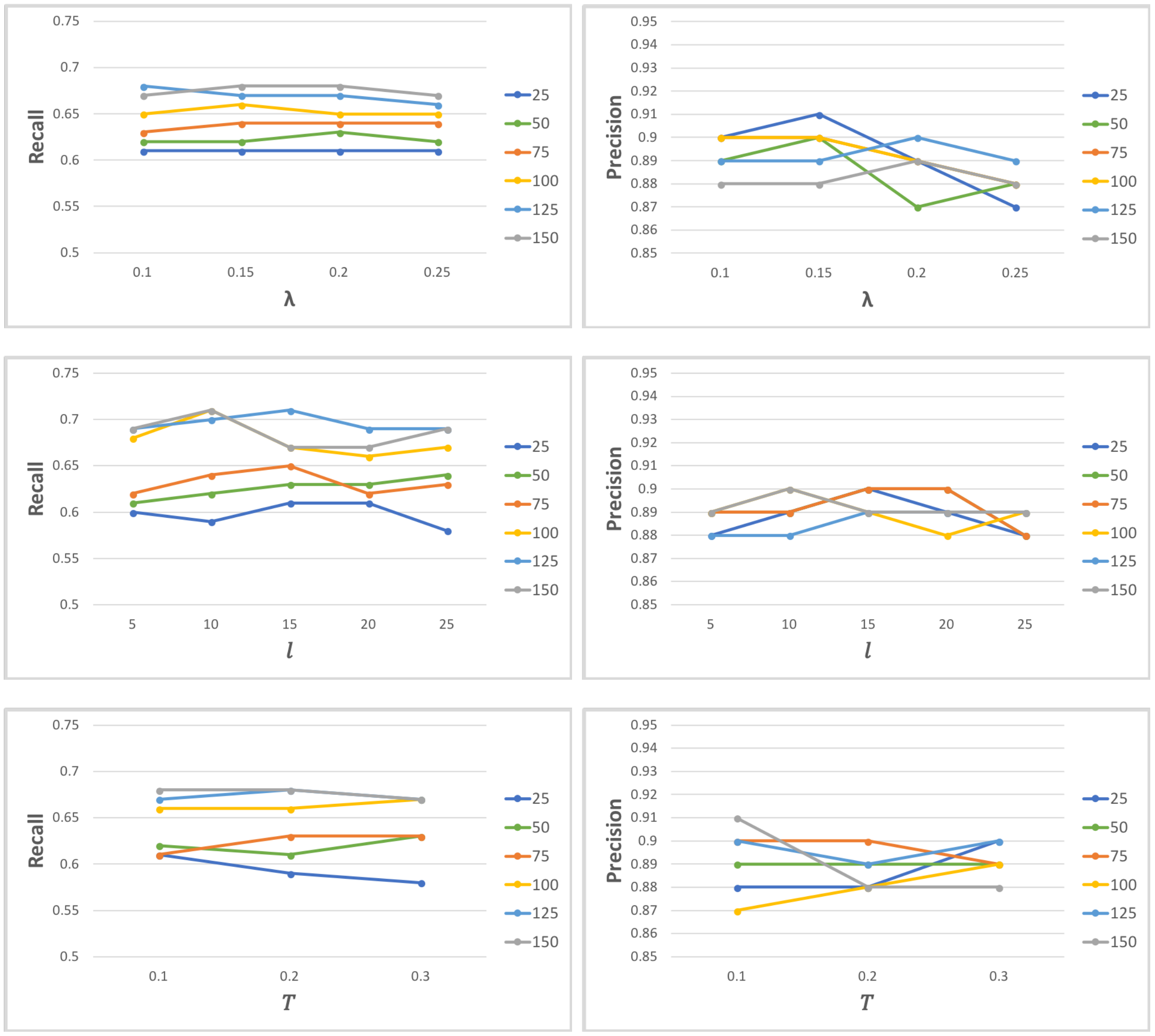
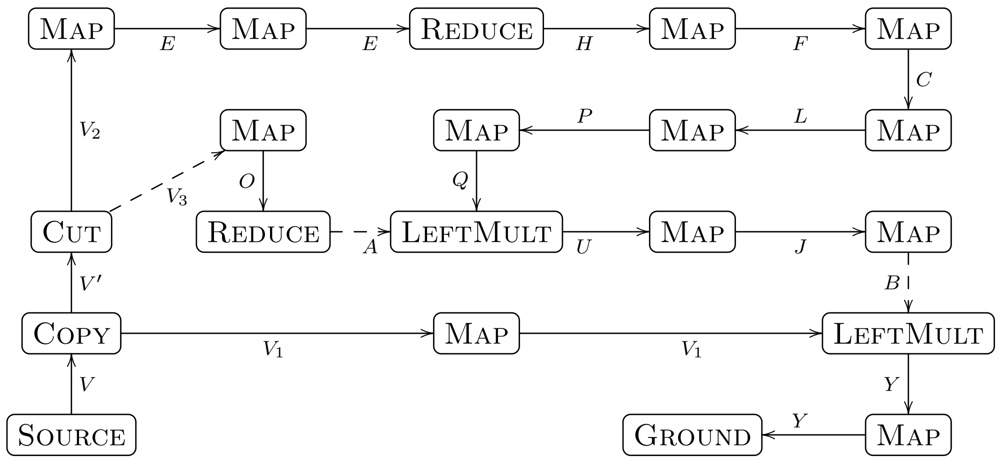
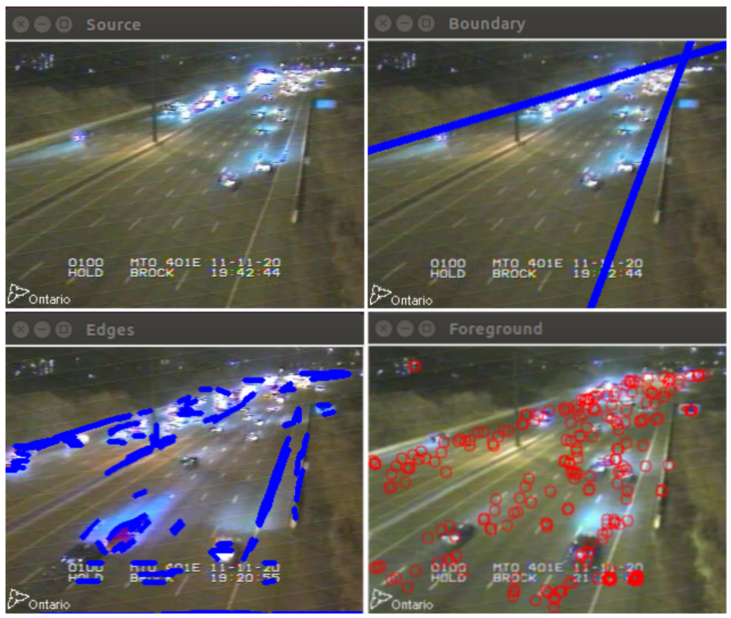


Figure 5.14: A sensitivity analysis of our algorithm on Dataset 2. The analysis study precision and recall against different values of the user-defined parameters and for different values of the number of superpixels N. The top row shows precision and recall versus the decay rate λ used by online hierarchical clustering. The middle row presents the accuracy versus the minimum length of segments used for polygon approximation. The bottom row shows the accuracy versus the ranking threshold used to select the high confidence clusters after confidence assignment.



(a)



(b)

Figure 5.15: Online road-boundary detection algorithm [81] described in the stream algebra: (a) the workflow graph with arrows showing the flow direction of streams; (b) displaying windows showing the V_1 stream, detected edges in stream E, foreground objects in stream A, and detected road boundary in stream Y. In (a), letters on arrows represent stream names and dashed lines indicate decoupled streams. The input stream video is V, and Y is the output video stream that shows the estimated dominant road boundary.

Chapter 6

Performance Tuning of Large-Scale Computer Vision Systems

Parameter tuning is an important problem in computer vision. Most algorithms rely on parameters to control runtime performance or output accuracy. To find the optimal parameter settings, we need to perform parameter tuning. This can be performed either manually, semi-automatically, [108] or automatically [149, 98, 47]. For a vision pipeline that contains several chained stream operators, parameter tuning becomes more challenging. This is because each operator can implement a vision algorithm that has a different set of parameters. Thus, the parameter space becomes large. In addition, these parameters need to be continuously updated toward changes in the input vision stream.

Several parameter-tuning algorithms have been proposed in computer vision for solving specific problems. For example, Kisilev et al. [108] proposed a semi-automatic algorithm that tunes parameters by estimating a preference function that captures user preferences from pairs of the input and output of algorithms. Their parameter-tuning algorithm was tested on simulated data and image-denoising applications. Sherrah [149] proposed another algorithm for continuous real-time parameter tuning of a people-tracking surveillance system. This algorithm works in two phases, an offline learning phase and an online tuning phase. In the offline learning phase, the algorithm learns the best set of parameters. During the online phase, the algorithm incrementally adapts the parameters in a continuous fashion to react to changes in input data. Parameter tuning for long-term tracking was explored by Supancic et al. [98], who proposed an online tracking algorithm that uses self-paced learning to continuously adapt the parameters of an appearance model to the tracked object. Chau et al. [47] also studied parameter tuning for tracking algorithms. They proposed an online technique to tune the parameters of a tracking algorithm to different scene contexts. This technique follows the same scheme of [149] by using an offline learning stage to learn the tracker parameters in different contexts. Then, an online stage continuously examines the tracking quality. If the quality is not good enough, the algorithm detects the current context and tunes the tracking parameters using the previously learned values.

The previous parameter-tuning examples are specific to certain algorithms. A computer vision pipeline has several stages, each implementing a different computer vision algorithm. Take for example, the traffic-analysis pipeline presented in Section 5.4 for road-boundary detection. The pipeline uses several algorithms that include edge detection, foreground segmentation, online clustering, statistical ranking, and activity ranking. Each algorithm has its own parameters, which we set to the selected default parameters. A traffic-video stream can have distinct environmental and scene contexts, such as daytime, nighttime, rain, and wind. Using one parameter setting cannot work well in all contexts. Hence, the vision pipeline is required to have several parameter settings or configurations and dynamically switch between them.

The developed stream algebra presented in Chapter 3 provides an abstraction suitable for describing and implementing efficient and scalable online computer vision pipelines. In Chapters 4 and 5, we presented computer vision algorithms that provide state-of-the-art speedup versus accuracy tradeoff in solving two fundamental computer vision problems: (1) pixel-labelling problems for stereo vision and optical flow and (2) automatic road-boundary detection in traffic-video analysis. For every algorithm, a concurrent streaming pipeline is described and implemented using the stream algebra.

As discussed in Section 3.3, a great advantage of stream algebras in databases is the ability to apply formal and abstract methods for implementing dynamic execution plans, applying incremental evaluation, scaling up data processing, and defining common pipeline optimization and cost models. Sections 3.2.2 and 3.2.2 described examples of representing iterative optimization and parameter-tuning algorithms using our algebra. However, the tuning and optimization methods in these examples are not general and were developed specifically for the algorithms in [47, 105]. In this chapter, we will present a formal method for adaptive parameter tuning of large-scale computer vision pipelines. Specifically, this chapter shows that a general optimizer of numerical parameters, such as the method by [95], can be used with the feedback-control mechanisms of our stream algebra to provide common online parameter optimization for computer vision pipelines. Without loss of generality, the streaming pipeline developed for our automatic road-boundary detection algorithm will be used as a case study.

6.1 Problem Statement

Given a linear computer vision pipeline with a set of operators $X = \{X_i\}_{i=1}^{n}$, we assume that each data-processing operator X_i executes a user-defined function with a set of input parameters P_i. The parameter settings of the pipeline are defined as $\theta = \bigcup_{i\in[1,n]} P_i$, where $\theta \in \Theta$ and Θ is the parameter configuration space for pipeline X. The processing functions are defined arbitrarily with no closed-form representation to allow the computation of gradients to optimize parameters. The functions may also be expensive to compute. Given these assumptions, we follow the general definition of the algorithm configuration problem [97, 96] to define the pipeline configuration problem as follows:

Definition 6.1.1 (Pipeline configuration problem) *Let pipeline X have a distribu-*

tion $\mathcal{I}$ of input instances and a target performance metric $c(\theta, \pi)$ with $\theta \in \Theta$ and on instances $\pi \subset \mathcal{I}$. Let $f(\theta) = E_{\pi \subset \mathcal{I}}[c(\theta, \pi)]$ define the expected performance of pipeline X using parameter settings θ on instances π drawn from $\mathcal{I}$. The pipeline configuration problem aims to find the optimal parameter configuration $\hat{\theta} = \text{argmax}_{\theta \in \Theta}\{f(\theta)\}$.

Sequential model-based optimization (SMBO) is a popular approach for solving general algorithm configuration problems by optimizing expensive blackbox functions [95, 94, 21, 96]. This approach optimizes the target performance function $f : \Theta \rightarrow R$ by sequentially evaluating samples of the parameter space Θ, while minimizing the number of samples required to reach the optimal parameter setting. This is performed by first calculating $y = f(\theta)$ values at a set of input data instances to obtain the initial set $\mathcal{D} = \{(\theta_i, y_i)_{i=1}^{n}\}$, which is referred to as the initial design.

The set $\mathcal{D}$ is used to learn a model that defines a probability distribution f over a continuous range of $\theta \in \Theta$. The SMBO then iterates in three steps: (1) update the posterior expectation of f using the learned model for new observed values (θ, y), (2) build an acquisition or utility function $g(\theta)$ that measures how desirable a certain θ is to maximize f, and (3) select the parameter setting $\hat{\theta} = \text{argmax}_{\theta \in \Theta} g(\theta)$ that maximizes the acquisition function and defines $(\theta, f(\theta))$ as a new observed value.

In the SMBO approach, the model follows a Gaussian stochastic process, formed as a Gaussian process (GP) model. So, f is estimated using a GP model that defines a Gaussian distribution over functions. The model is first learned using the initial design, then used to calculate the expected value of the performance function $\bar{f}(\theta_*) = \mu_*$ at any arbitrary θ_*. Following SMBO, we use the learned GP model to select the next sample θ_o in the parameter space such that it provides improvement over the optimal setting $(\hat{f}, \hat{\theta}) = \text{argmin}_{(f,\theta) \in \mathcal{D}}\{f\}$ seen so far. We can perform this selection by maximizing an acquisition function, usually selected as the popular expected improvement (EI) function [140]. We refer the reader to Appendix B for a detailed derivation of the GP model.

6.2 Feedback Control Using Time-Bounded Sequential Parameter Optimization

In this section, we extend the time-bounded sequential parameter optimization (SPO) algorithm in [95] to tune the parameter settings of different data-processing operators $\{X_1, ..., X_k\}$ in a general streaming pipeline $X = \{X_i\}_{i=1}^{n}$, where $\{X_1, , X_k\} \subseteq X$ and $k \leq n$ (see Figure 6.1a). The data-processing operators include the Map, Reduce, and Filter operators presented in Section 3.1. Each data-processing operator can receive the corresponding parameters of its data-processing function and apply them on incoming inputs. When X is a computer vision pipeline, the input stream I_{in} is a sequence of images or video frames and the output stream I_{out} represents the computed results. Given an input $\alpha \in I_{in}$ and its corresponding computed output result $\pi \in I_{out}$, a performance metric $f(\theta, \pi)$ is defined. This metric describes the quality of a result $\pi \in I_{out}$ computed by the data-processing operators with input parameter setting θ.

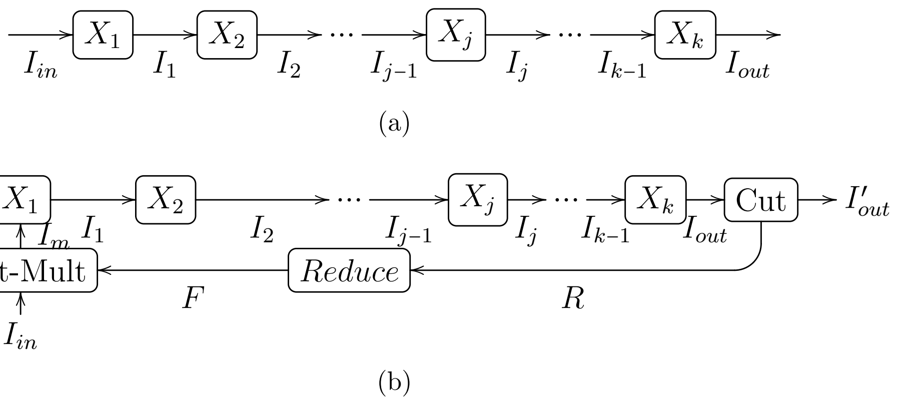


Figure 6.1: Example of a feedback-control system: a) feedforward streaming pipeline; and b) single-loop feedback-control system for controlling a set of operators in the streaming pipeline shown in (a).

A feedback loop is defined using the Cut operator that samples the pipeline output stream I_{out}. Each output sample is processed using the Reduce operator, which applies a time-bounded SPO to produce a feedback stream F that includes the candidate input parameter settings for the controlled operators (see Figure 6.1b). Using the LeftMult

Algorithm 5 Time-bounded SPO

```
Require: $u = (T, W, Y, N, \text{toggle}, \theta_o, t_{max}, n_{max}, f), x = (\pi, \theta, \Delta t)$
Ensure: $u' = (T', W', Y', N', \text{toggle}, \theta'_o, t_{max}, n_{max}, f)$ and $y = \theta_n$
 1: if $x \neq null$ then
 2:     $i_* = \text{GetIndex}(x.\theta, u.W)$;
 3:     $u.Y[i_*] = u.Y[i_*] + u.f(x.\pi, x.\theta)$;
 4:     $u.N[i_*] = u.N[i_*] + 1$;
 5:     $u.T[i_*] = u.T[i_*] + x.\Delta t$;
 6: end if
 7: $u' = u$;
 8: $S = \{i | u'.T[i] \leq t_{max}$ and $N[i] \leq n_{max}\}$;
 9: if $S \neq \Phi$ then
10:     $i_r = \text{RandomSample}(S)$;
11:     $\theta_n = W[i_r]$;
12: else
13:     $u' = \text{RemoveWorst}(u')$;
14:     if $u'.\text{toggle}$ then
15:         $\mathcal{M} = \text{FitModel}(u)$;
16:         $\theta_n = \text{SelectConfiguration}(\mathcal{M})$;
17:     else
18:         $\theta_n = \text{RandomSample}(\Theta)$;
19:     end if
20:     $u' = \text{Append}(u', \theta_n)$;
21:     $u'.\text{toggle} = !u'.\text{toggle}$;
22: end if
23: $y = \theta_n$;
24: $S' = \{i | u'.T[i] \leq t_{max}$ and $N[i] = n_{max}\}$;
25: if $S' \neq \Phi$ then
26:     $i_o = \text{argmax}_{i \in S'}(u'.Y'[i] / u'.N[i])$;
27:     $u'.\theta'_o = u'.W'[i_o]$;
28: end if
```

operator, each parameter setting in the feedback stream $\theta \in F$ is attached to a corresponding input instance $\alpha \in I_{in}$ to produce the merged stream I_m of pairs (α, θ). Each input instance α in the pair $(\theta, \alpha) \in I_m$ is processed by the data-processing operators using the attached parameter setting θ to produce the output $(\theta, \pi) \in I_{out}$.

The time-bounded SPO method is defined using Algorithm 5, which extends [95] for parameter tuning of computer vision pipelines. The algorithm takes as input two vectors u and x and produces two outputs u' and y. The vector u keeps track of the algorithm state through sequential runs and is updated and assigned to u'. The input x is a sample from the pipeline output stream, and the output y is the output parameter setting that will be applied to the next input instance of the pipeline input stream. The state vector u has several components. The $u.W$, $u.Y$, and $u.N$ components define the parameter matrix W and the computed performance vector y in Equation B.2, where $u.W = [\theta_1, ..., \theta_n]$ and $\mathbf{y}_i = Y[i]/N[i]$ for $1 < i < n$. The components $u.t_{max}$ and $u.n_{max}$ define the maximum computational time and the maximum number of samples allowed for evaluating any parameter setting. In addition, t_{max} allows us to ignore early parameter settings resulting in high computational time, whereas n_{max} ensures that parameter settings in $u.W$ are fairly evaluated on a similar number of samples. The component $u.T$ accumulates the computational time taken for evaluating parameter settings on input instances. The $u.\theta_o$ component refers to the optimal parameter setting found so far and is continuously updated after each call to Algorithm 5. The $u.f$ component defines the metric function. The u.toggle continuously switches between two actions: (1) fit a GP model using the updated state u' and find θ that maximizes Equation B.12 and (2) randomly sample a new parameter setting from the parameter space Θ. Notice that the performance of the GP model depends on the initial design. In order to get a good initial design, usually, a costly initialization is required using Latin hypercube sampling [127], which is a method for generating random samples of parameter values. By interleaving between random sampling and Bayesian optimization, we can eliminate the need for a costly initial design [95].

Initially, Algorithm 5 receives a state vector u initialized with a set of n randomly sampled parameter settings from the parameter space Θ. The algorithm starts by testing whether there is an input sample x. If so, then the algorithm finds the index i_* of the parameter setting $x.\theta$ (used to evaluate the output sample $x.\pi$) from the state parameter matrix $u.W$. Next, the algorithm calculates the performance of the parameter setting $x.\theta$ using the metric function f. The measured performance value is accumulated on

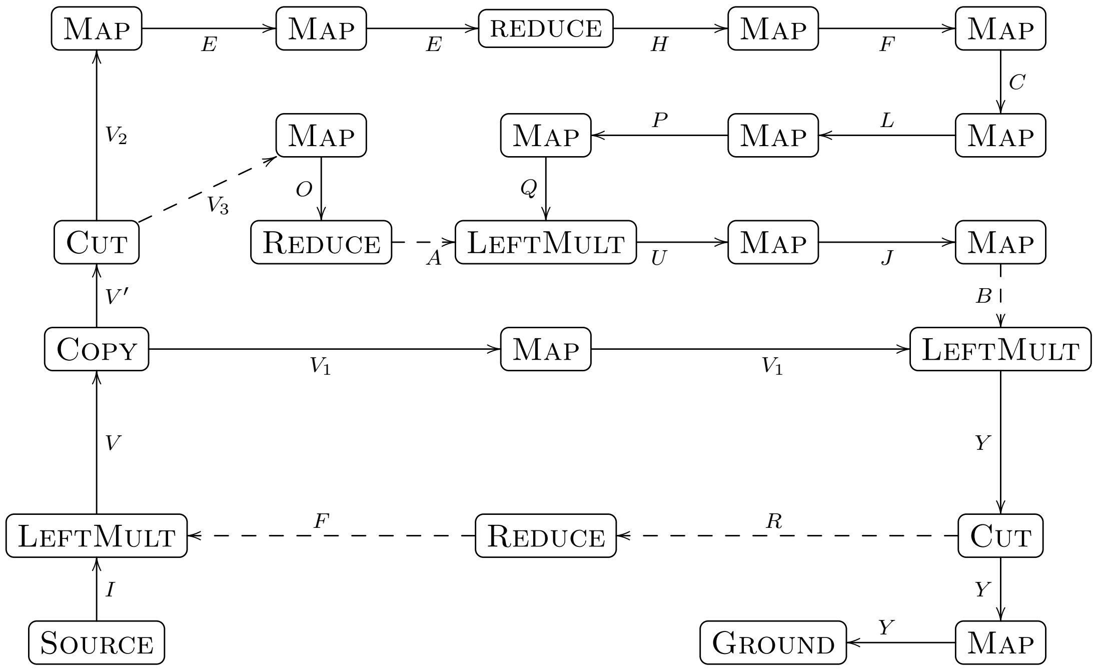
$u.Y[i_*]$, and $u.N[i_*]$ is incremented. In addition, the computational time $u.\Delta t$ taken by the pipeline in computing the output $x.\pi$ is accumulated to $u.T[i_*]$. After updating the state u, it is assigned to the output state u'. In Step 8, the algorithm locates the set S containing all indexes i of parameter settings with accumulated computational times $u'.T[i] \leq t_{max}$ and number of evaluations $u'.N[i] \leq n_{max}$. If the set S is not empty, the algorithm randomly picks a parameter setting from $u.W$ and assigns it to output y for the next evaluation. If S is empty, then all parameter settings in $u.W$ met one or both maximum budgets n_{max} and t_{max}. In this case, the algorithm executes the RemoveWorst function that deletes the worst performing parameter setting from state u. Then, a new parameter setting is either randomly sampled from space Θ or found from a fitted GP model by maximizing Equation B.12. The u'.toggle variable controls the decision. The new setting is then added to the state u' using the append function and the u'.toggle variable is inverted. Finally, Steps 23 to 26 set the output parameter setting that will be evaluated next and finds the optimal parameter setting among the settings that met the maximum number of evaluated samples $u.n_{max}$ with time budgets $\leq t_{max}$.

6.3 Experimental Results

In this section, we present an experimental evaluation of the time-bounded SPO algorithm, Algorithm 5. For simplicity and without loss of generality, we use the streaming pipeline of the road-boundary detection algorithm as our case study (see Section 5.4).

6.3.1 Case Study

Figure 6.2 shows the streaming pipeline implementation of the road-boundary detection algorithm after applying feedback control. A feedback branch is formed by applying the Cut operator to sample the output stream Y and create the return stream R. This can



(a)

Figure 6.2: Online road-boundary detection algorithm [81] described in the stream algebra. Arrows show the flow direction of streams and letters on arrows represent stream names. Dashed lines indicate decoupled streams. The input video stream is I, and Y is the output video stream that shows the estimated dominant road boundary. A feedback loop samples the output stream Y into the return stream R using the Cut operator. This stream is processed using the Reduce operator that executes the time-bounded SPO algorithm and outputs a stream of parameter F. The LeftMult operator then latches on the F stream and attaches a parameter setting to every incoming input instance.

be described using the following equation:

$$R, Y \triangleq \textsc{Cut}()(Y). \tag{6.1}$$

The stream R is then used as input to the Reduce operator that executes the function $g_{spo} : U \times X \to U \times Y$. This function applies Algorithm 5 for sequential parameter optimization. The Reduce operator is initialized with a state vector $u_0 \in U$ that has a set of initial parameter settings sampled from the parameter space Θ (also called the initial design). The operator produces the feedback stream F that contains candidate parameter settings, using the equation:

$$F \triangleq \textsc{Reduce}(u_0, g)(R). \tag{6.2}$$

The LeftMult operator then latches on the R stream and attaches a parameter setting proposal to every incoming instance in the input stream I:

$$V \triangleq \textsc{Left-Mult}()(F, I). \tag{6.3}$$

For the road-boundary detection algorithm, the following numerical parameter vector is identified: $\theta = (N, L, \lambda, T, \epsilon)$, where N is the number of superpixels, L is the line segment length used in the polygon approximation step, λ is the decay rate of online clustering, T is the ranking threshold used to select the top-ranked clusters after confidence assignment, and finally ϵ is the parameter used in Equation 5.15 for calculating cluster confidence. Each parameter has a range of values defined by a lower and upper limit. The limits define the parameter space Θ and are specified as follows: $N \in [25, 150]$; $L \in [5, 30]$; $\lambda \in [0.1, 0.6]$; $T \in [0.1, 0.6]$; and $\epsilon \in [5 \times 10^{-3}, 5 \times 10^{-2}]$.

6.3.2 Experimental Evaluation

In Section 5.2, we showed the results of the online road-boundary detection algorithm on two datasets. Dataset 1 contains 50 short low-resolution video sequences collected from 14 different camera locations at a resolution of 320 × 240. Dataset 2 is a single long video sequence of 1,627 frames recorded at 704 × 480 resolution and has the camera changing its viewing directions to focus on different regions in the traffic scene. All results were computed using the following parameter vector $\theta = (100, 10, 0.2, 0.2, 0.005)$. This vector was found by creating a grid that divides the range of each parameter into five intervals (using the limits defined in the previous section) and randomly selecting a set

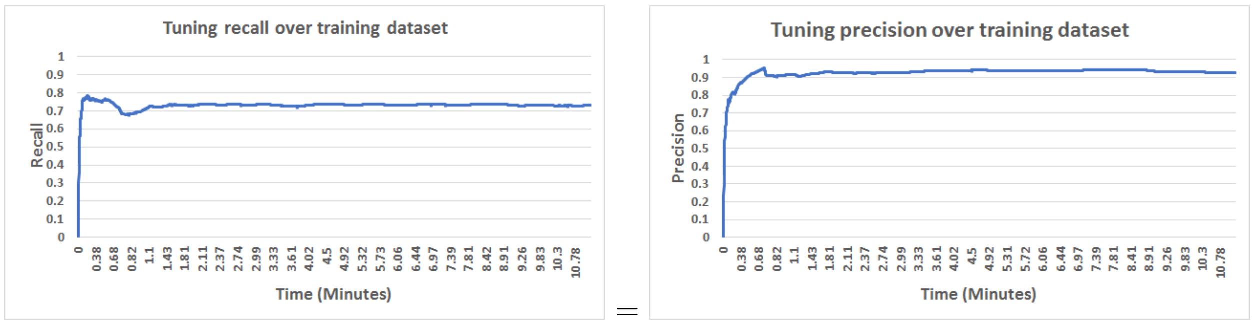


Figure 6.3: Using the time-bounded sequential parameter optimization (SPO) algorithm to select parameter settings that maximize precision and recall measures on the training dataset. At each time step, we record the precision and recall of the optimal parameter setting found so far.

of 50 parameter settings from the grid. A manual search is applied to select the setting vector that results in the best accuracy.

To evaluate the effectiveness of the time-bounded SPO algorithm in tuning the parameters of the case study, we compare the quality of the best parameter setting found using Algorithm 5 to the one selected by manual search. The comparison is performed on the long video stream of Dataset 2, where the first 814 frames of the dataset are used for training the GP model, and the remaining 813 frames are used for testing. The quality measure for a parameter setting θ is defined as follows:

$$Q(\theta) = \sum_{i=1}^{n} \frac{\delta_{\theta i}(P_{\theta i} + R_{\theta i})}{2n}, \qquad (6.4)$$

where $n = 814$ is the number of training frames, $\delta_{\theta i} \in (0, 1)$ is a Boolean determining whether or not frame i was used to evaluate the setting θ. In addition, $P_{\theta i}$ and $R_{\theta i}$ are the precision and recall computed for frame i using the parameter setting θ.

Figure 6.3 shows the performance curves of the time-bounded SPO algorithm in selecting the best parameter settings over time and on the training dataset. Figure 6.3(left) shows the measured recall value of every best parameter setting found over time. Figure 6.3(right) shows a similar plot for precision values. The SPO algorithm is executed

Methods	Precision	Recall	Average Runtime (seconds)
V4 + SPO (Testing)	$92\%_{\pm 20\%}$	$72\%_{\pm 19\%}$	0.13
V4 + SPO (Training)	$93\%_{\pm 18\%}$	$73\%_{\pm 16\%}$	0.13

Table 6.1: Comparing the results of the V4 [81] method with the sequential parameter optimization algorithm on both the training and testing datasets. The figure shows the mean and standard deviation statistics for both precision and recall. The last column shows the average runtime in seconds.

Methods	Precision	Recall	Average Runtime (seconds)
V4 + SPO	$92.5\%_{\pm 19\%}$	$72.5\%_{\pm 17.5\%}$	0.13
V4 [81]	$90\%_{\pm 11\%}$	$71\%_{\pm 20\%}$	0.13
V3 [75]	$47\%_{\pm 33\%}$	$76\%_{\pm 22\%}$	200
Gabor [109]	$78\%_{\pm 14\%}$	$43\%_{\pm 12\%}$	54.9
CN24 [39]	$64\%_{\pm 26\%}$	$49\%_{\pm 43\%}$	81.5

Table 6.2: Comparing the results of the V4 [81] method before and after applying the sequential parameter optimization algorithm on Dataset 2 and using the mean and standard deviation statistics for both precision and recall. The V3 [75], Gabor [109], and CN24 [39] methods are also included in the comparison. The last columns show average runtime in seconds for each method.

for 11 minutes. It took around 2 minutes to converge to an optimal parameter setting with precision of 93% and a recall of 73%. The tuning algorithm then selected parameter settings with nearly similar performance for the remaining time.

Table 6.1 shows results of V4+SPO method on both the training and testing datasets. Table 6.2 compares the results of the V4 [81], V3 [75], Gabor [109], and CN24 [39] methods to that of the V4+SPO method. All results are reported on the entire images of Dataset 2. These results show that both the precision and recall of the V4 method are improved using the time-bounded SPO algorithm. The V4 + SPO pipelined method achieves an average precision of 92% and an average recall of 72%, whereas the V4 pipelined method achieves 90% and 71%, respectively. The runtime is the same for V4 + SPO as in the V4 method. This is because the feedback loop defined by the pipeline of V4 + SPO uses the Cut operator for sampling the output stream, which decouples the output stream from the feedback stream.

6.3.3 Discussion

Experimental results showed the ability of our stream algebra to naturally describe feedback control and to implement a general parameter optimization algorithm for performance tuning of streaming computer vision pipelines. The case study showed that the algebraic feedback-control primitives can be combined with the time-bounded SPO algorithm to tune numerical parameters of pipelined computer vision functions.

The case study focused on tuning the streaming pipeline (see Figure 6.2) for the road-boundary detection algorithm on a single-input stream. However, we can scale up the algorithm to process several streams by creating multiple instances of the streaming pipeline, one for each stream. In this case, the visual content of each stream will guide the parameter optimization of the dedicated processing pipeline of the stream. Thus, learning a different parameter setting for each stream, which is adapted to the lighting, structure, and environmental conditions presented in the stream. However, this requires obtaining a training dataset for each stream. These datasets can be obtained using a human operator that labels the road boundaries in one or more sequences of stream images.

Moreover, a possible future direction is to apply unsupervised learning. This can be performed by developing a quality measure that doesn't depend on a training dataset. For example, the method of [137] extends our approach for road boundary detection to handle non-linear road geometry. This method develops a quality function for road-boundary based on the idea that features of road regions are different than other image regions. Hence, the best road boundary should have its covered road area most distant from other image areas. Given a road boundary defined by a cluster pair (c, c'), the method defines the dissimilarity between the road and non-road regions using *Bhattacharyya distance*,

$$B(c, c') = \sqrt{1 - \sum_{s=1}^{k} H^{in}(s) H^{out}(s)}. \tag{6.5}$$

Here, H^{in} and H^{out} are the normalized colour histogram for area bounded by (c, c') and the outside area respectively. k is the number of bins. Notice that the *Bhattacharyya distance* measures the dissimilarity of two distributions or samples. Generally, it approximates the amount of overlap between two samples or populations. More the overlap, lower the distance.

Equation 6.5 may be used to define a heuristic for the quality measure suitable for unsupervised learning of parameter settings. For example, the best parameter setting should generate road boundaries that maximize equation 6.5. Hence, the quality measure for a certain parameter setting may be defined as,

$$Q(\theta) = \sum_{i=1}^{n} \frac{(B_{\theta i})}{n}. \tag{6.6}$$

n is the number of frames used for evaluating the parameter setting θ. $Q(\theta)$ then calculates for each parameter setting θ, the running average of the dissimilarity between the road and non-road regions, on frames i used for evaluating θ. As a future work, we will be studying quality measures similar to the one defined by equation 6.6 that we think may allow applying unsupervised learning for performance tuning of computer vision systems.

The time-bounded SPO algorithm, Algorithm 5, is applied in the case study as an offline learning approach. In this case, the algorithm finds the optimal parameter setting using the given training and testing datasets. Then, the optimal parameters are used as the default setting for future stream processing. However, online learning is also possible using partially labelled data, where a human operator can manually label some segments of the input stream. These segments can be then used to further optimize and adapt pipeline parameters against previously unseen environmental changes in the traffic scene.

Although the case study applied Algorithm 5 [95] for parameter tuning, several other model-based [101, 94, 21, 96] and model-free [2, 32, 31, 97, 96, 9] parameter-tuning algorithms can also be applied. The racing algorithm [32], for example, can be used if

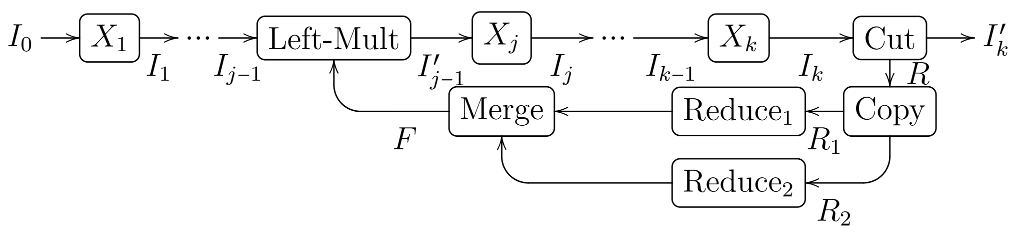


Figure 6.4: Example of applying the idea of boosting to merge the output of two different parameter-tuning algorithms and select the best performing parameter setting.

a list of candidate settings can be predetermined. The best setting can be selected by iteratively evaluating each candidate setting on a stream of input instances. The method of [93] can be used to optimize categorical parameters using decision tree models. Such methods can be applied by replacing the processing function g in Equation 6.2 that parametrizes the Reduce operator of the feedback loop.

Moreover, the feedback-control loop presented in the case study can be extended to multi-loop feedback control (see Figure 3.1) for scaling up parameter optimization. In this case, each feedback loop can have the Reduce operator, applying similar or different parameter-tuning methods. By treating these methods as weak learners for estimating the performance response surface of the quality metric (see Equation 6.4), we can apply the idea of boosting from machine learning [190]. Boosting combines a set of weak learners to produce a single strong learner. Assuming m weak learners, the Merge operator (see Figure 6.4) can find the best parameter setting $\theta\star$ from the output estimates $\pi = \{\theta_1, ..., \theta_m\}$ of weak learners by applying the select function $\theta\star = \text{argmax}_{\theta\in\pi}\{Q(\theta)\}$. Here, each learner produces a pair $(\theta, Q(\theta))$.

The case study, experimental results, and discussion show the effectiveness of using our stream algebra for general parameter tuning of the computer vision pipeline. We think that the ability of our stream algebra to flexibly apply and scale up parameter tuning will open several future opportunities in building and optimizing large-scale streaming computer vision systems.

Chapter 7

Conclusion

7.1 Summary of Contributions

In this work, we addressed existing challenges in building large-scale computer vision systems processing image and video streams. These challenges include the lack of formal and scalable frameworks for building and optimizing streaming computer vision pipelines. Additionally, many existing computer vision algorithms are computationally expensive and cannot efficiently scale up for processing large-scale data. As a step toward in overcoming these challenges, we presented formal methods for building scalable computer vision systems.

First, we described a stream-algebra framework for mathematically expressing computer vision pipelines (online vision systems that process image and video streams). The algebra defines an abstract set of concurrent operators with formal semantics that manipulate image and video streams. The algebra provides operators for both data processing and rate control. The data-processing operators perform data transformations on input image and video streams, whereas the rate-control operators allow decoupling and synchronization between the data-flow rates of different computer vision pipelines, thus supporting seamless integration between different computer vision tasks to build large-scale systems. The algebra also can naturally express feedback-control loops. We presented an algebraic description that can express both single-loop and multi-loop feedback control in computer vision pipelines, thus enabling optimization tasks, such as parameter tuning and iterative optimization. We showed the effectiveness of the algebra in describing several state-of-the-art techniques in computer vision.

We also developed new computer vision algorithms for efficiently processing image and video streams in the areas of pixel-labelling problems and automatic visual surveillance. For pixel-labelling problems, we developed the sparse cost-volume filtering approach for solving the problems of stereo vision and large-displacement optical-flow estimation. The approach leverages sparse processing of cost volume, which can be tuned to trade-off speed versus accuracy by controlling sparsity. We presented two methods for computing the sparse sub-volumes: a feature-based method the applies keypoint matching and a segmentation-based method that relies on superpixel segmentation and nearest-neighbour fields. While the feature-based method is unable to scale up for large cost-volume processing, the segmentation-based method is linear in the image space and can be scaled up to process large cost volumes. Both methods have been shown to outperform several state-of-the-art methods in stereo vision and optical flow. The segmentation-based method was also expressed in our stream algebra and implemented as a multi-GPU streaming pipeline that can be scaled up to process multiple video streams. For automatic visual surveillance, we developed an online method for automatic lane and road-boundary detection in traffic videos recorded by uncalibrated cameras typically mounted along highways. The method can process traffic-video streams in real time. It also can detect roads under severe environmental conditions and when the camera is re-positioned or has its viewing direction changed. The algorithm is expressed in our algebra and implemented as a concurrent streaming pipeline.

Finally, we showed that the feedback-control definitions of our stream algebra can implement a general parameter optimization algorithm for parameter tuning of streaming computer vision pipelines. We used the streaming pipeline of the automatic road-boundary detection as a case study. Experimental results showed the effectiveness of combining the feedback-control primitives of our algebra with a general sequential parameter optimization algorithm as a common optimization method for parameter tuning in large-scale streaming pipelines.

7.2 Future Directions

Our work made several material contributions but also suggests several directions for future research in building efficient and scalable computer vision algorithms and systems.

Efficient computer vision algorithms. Although we developed efficient algorithms for stereo vision, optical flow, and automatic road-boundary detection, there is still considerable room for improving the accuracy and speed of these algorithms for processing image and video streams. Our sparse cost-volume filtering approach can trade off accuracy versus speed or vice versa by controlling sparsity. In the future, we aim to study this feature, which allows tuning the algorithm performance based on the speed and complexity of incoming video streams. We will also apply the algorithm on other pixel-labelling problems, which include multi-label segmentation, co-segmentation, and scene labelling. We think that the accuracy and speed of sub-volume cost filtering can be improved further by integration with deep learning that can estimate costs in pixel-labelling problems. Indeed, convolutional neural networks (CNN) have been recently trained for the task of patch matching in optical flow and stereo vision [16, 166]. They provide promising results on benchmarks. Currently, we are building on this work to further boost the accuracy and performance of our sparse cost-volume filtering approach. We are also planning to study the various applications of our road-boundary detection algorithm in traffic video analysis, which includes detection of excessive speeding, careless driving, and accidents.

Web-scale computer vision. Our stream-algebra framework provides the means to build large-scale computer vision systems capable of processing large volumes of image and video streams. We think that the algebra is a step forward in building web-scale computer vision systems. In the future, we aim to study and build large-scale computer vision systems that process, align, and understand video and photo streams that are publicly available via photo sharing services and websites. This direction is aligned with the current race of developing scalable computer vision algorithms capable of real-time processing of vast amounts of image and video streams. The winning algorithms in this race will not only be accurate but also capable of efficiently dealing with the scale, variation and growing nature of image and video streams. Today, for example, everyone can simply hold up his phone and capture live streams of actions and personal moments. Smart home and traffic surveillance cameras are rapidly increasing in numbers. We can also see the fast development of swarms of flying machines or micro-quadcopters that can be equipped with cameras. This motivates the need to develop efficient frameworks that can easily and seamlessly integrate the different services, algorithms, and components required to build large scale computer vision systems.

Execution plan optimization and performance tuning. Our stream algebra provides the ability to define different execution plans for the computer vision pipeline. For example, we can merge a sequence of Map and Reduce operators in a pipeline using the function composition. A runtime optimizer can then select and switch between different plans to provide load balancing, maximize performance, and reduce latencies required to move data between concurrent stream-processing operators. As a preliminary study, we applied simple load balancing techniques on the linear branches of the traffic surveillance pipeline [138]. For each branch, the target is to partition the branch into a set of intervals whose execution times are well balanced. The execution time of each interval is the sum of processing times of operators in the interval. The number of intervals is an input parameter and usually set to the number of available processor cores allocated to execute the branch. The Map and Reduce operators in each interval are merged together and executed on a single core. As a future direction, we aim to build a runtime optimizer capable of performing throughput versus latency optimization of computer vision pipelines. We are envisioning that with a set of benchmarks of a given computer vision system, the optimizer can suggest the best execution plans for different target environments and help in estimating the required computational resources at different data scales. At runtime, the optimizer will continuously monitor a computer vision system to guarantee optimal execution plan. It will also dynamically identify and resolve bottlenecks by applying parallel execution patterns. Thus resulting in big savings in energy and computational resources that could be wasted by poor assignment of tasks.

Dynamic reconfiguration. Our algebra opens a new research direction in enabling dynamic reconfiguration in large-scale computer vision pipelines. This is enabled by the ability of every data-processing operator to receive a list of functions as input. The runtime can later decide to dynamically switch between these functions to match changes in the content and speed of incoming streams. The decision can be performed using the feedback-control mechanisms of the stream algebra. One interesting future direction is to build an optimizer that dynamically reconfigures the data-processing operators to maintain a predefined level of quality of service. This is by treating functions as system parameters that can be tuned using parameter tuning algorithms. Take for example a Map operator that performs optical flow, one can simply use the top-ranked algorithm on benchmarks as the mapping function. A closer look at the results of the different algorithms on benchmarks can show that some algorithms have better accuracy and speed than others on specific datasets. Thus allowing the Map operator to receive a list of algorithms enables a computer vision system to dynamically configure the operator to optimize accuracy and speed against changes in input data. We believe that a key factor in building successful large-scale computer vision systems is to give them capabilities of self-configuration and self-optimization.

Feedback control. The ability of algebra to naturally express feedback control allows efficient description and implementation of several optimization tasks for tuning the performance of large-scale computer vision pipelines. We aim to study computer vision pipelines with multi-loop feedback control that perform feedback control tasks such as parameter tuning, incremental learning, and iterative optimization. These are important tasks in computer vision. Incremental learning, for example, has been recently used to improve the performance of object detection and tracking. This is performed by using top-ranked detections as new inputs to continuously adapt learned models and incrementally learn model parameters. Iterative optimization is also a fundamental operation in several computer vision tasks such as image segmentation, stereo vision, and human pose estimation. The feedback primitives of the stream algebra not only allow efficient implementation of these tasks but also can scale them up for big visual data processing.

Pipeline instrumentation. The stream algebra also opens a new direction in instrumentation of computer vision pipelines. This allows us to define methods to debug and monitor pipeline accuracy and performance. The current popular debugging mechanisms include log files and dashboards, however there is nothing better than inserting a probe at any vision steam in a large scale system and getting a visual view. The Cut operator, for example, can be inserted dynamically to sample any stream in the pipeline. The sampled stream can be later used as input to complex visualization and monitoring methods. Because the sampled stream has a decoupled data-flow rate, the pipeline data-flow rate will not be affected by the complexity of monitoring methods. We are also planning to implement algorithms that detect failures such as outputs with poor accuracy or inputs causing unpredictable performance and provide the ability to trace operators and visualize streams responsible for these failures.

Stream-algebra distributed implementation. Our stream algebra has been implemented in the Go language. The current implementation can scale up processing on multicore CPU and GPU systems. The algebra also inspired the development of several tools and frameworks such as ReactiveX [1], which is an API for asynchronous programming with more than 150 streaming operators. All these operators can be implemented using the 14 operators of our stream algebra. Moreover, pipelines implemented by ReactiveX are single threaded by default and require developers to plan multi-threading themselves. Currently, we are working to implement our stream algebra in Apache Kafka [10] to enable building scalable computer vision pipelines on large computing clusters. We aim to have a programming platform where a developer or researcher can express his computer vision system using our stream algebra and later compile the system to different target computing environments that include single-core systems, multi-core systems or large computing clusters.

[1]ReactiveX: http://reactivex.io/ (*last accessed on 7 March 2018*).

Appendix A

Supplementary Materials For the Sparse Cost-volume Filtering Approach

The central claim of our work is that the SVF and ACF methods achieve accuracy that is comparable to other schemes for computing optical flow and stereo vision while providing faster runtimes than other techniques. This appendix provides additional results and comparisons of SVF against existing published optical flow estimation techniques on Middlebury, MPI Sintel, and KITTI optical flow benchmarks. It also shows results of ACF on Middlebury standard stereo benchmark and 2005/2006 high-resolution stereo datasets. The results support our assertion.

Middlebury Benchmark

Tables A.1 and A.2 list results of SVF+OH on the Middlebury benchmark. Table A.1 presents ranking based on the average End-Point Error (EPE); whereas, Table A.2 presents ranking based on the average Angle Error (AE).

We perform several additional experiments for testing the performance on the Middlebury optical flow training datasets. Figure A.3 demonstrates the convergence of SVF+OH for 10 PatchMatch iterations (baseline set to 7). Figure A.3a shows average End-Point Error (EPE) versus filtering time, and Figure A.3b shows average Angle Error (AE) versus filtering time. Figure A.4 shows a comparison of filtering time between our SVF method and PatchMatch Filter [122].

Figure A.5 presents the optical flow estimation results of SVF+OH on Middlebury testing datasets using the proposed method.

Figure A.1 and A.2 present the stereo estimation results of ACF+OH on Middlebury standard stereo benchmark and 2005/2006 high resolution stereo datasets, respectively.

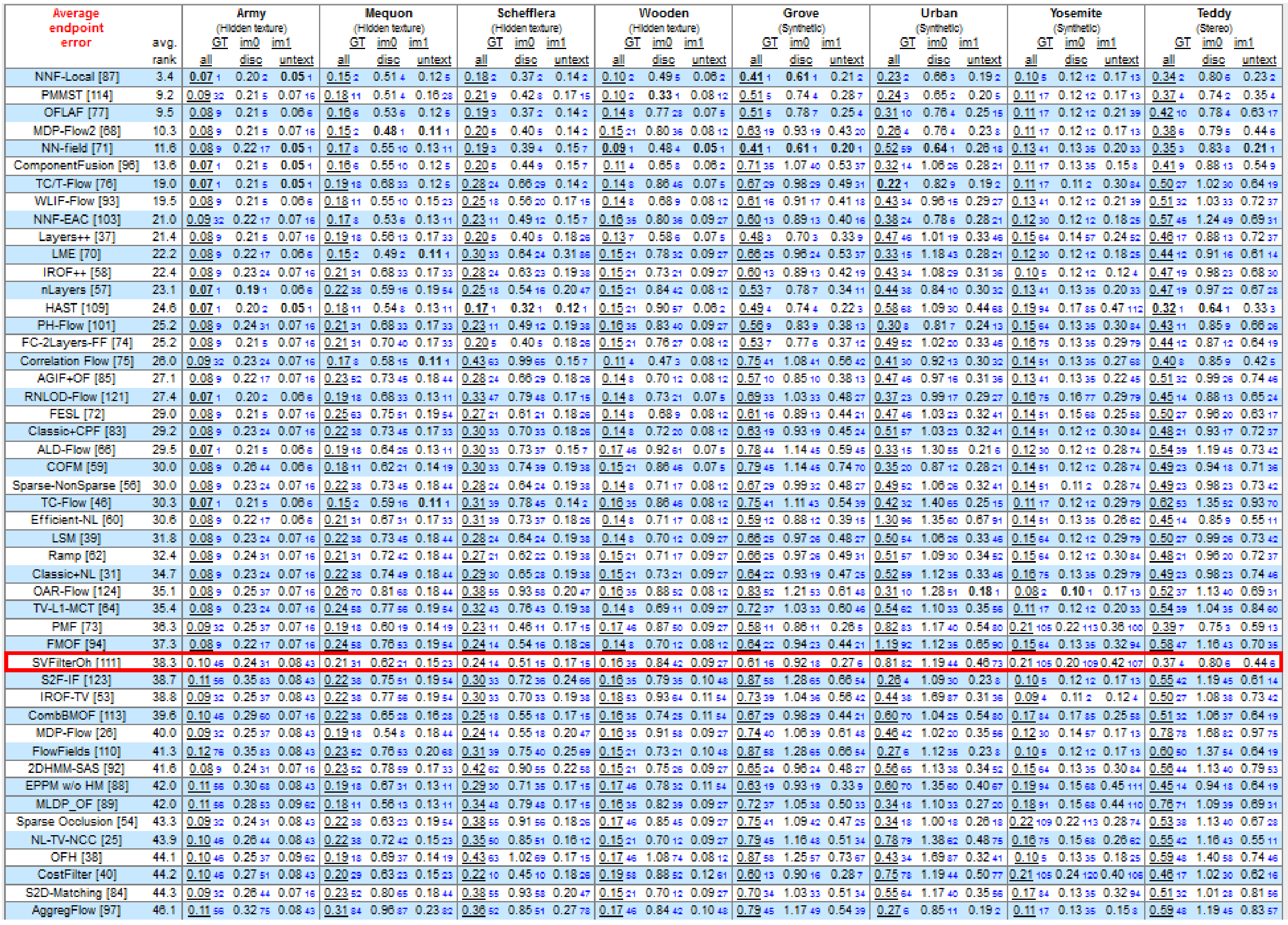
MPI Sintel Benchmark

Tables A.3 and A.4 list results of SVF+OH on MPI Sintel clean and final datasets. We are ranked 21 on the clean pass and 22 on the final pass, out of current standard published work. Notice that the proposed method only takes a fraction of the time of the state-of-the-art methods.

Figures A.6 and A.7 show the computed optical flow on MPI Sintel clean and final datasets, respectively.

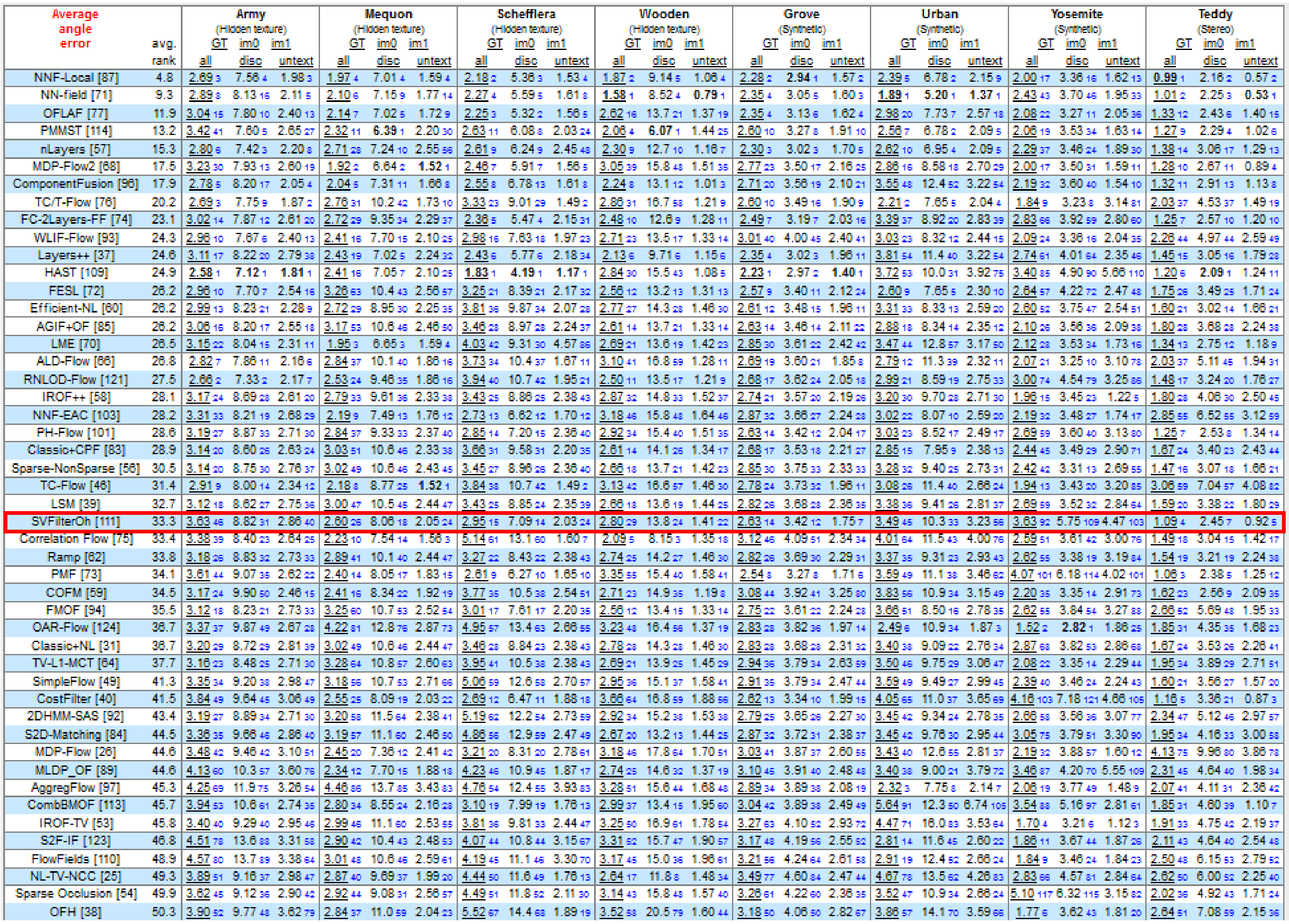
KITTI Benchmark

Tables A.5 lists the results of SVF+OH on the KITTI benchmark. The table is ranked by the percentage of erroneous pixels in non-occluded areas (Out-Noc). Our method achieves an average EPE of 9.1 on all image regions (Avg. All).



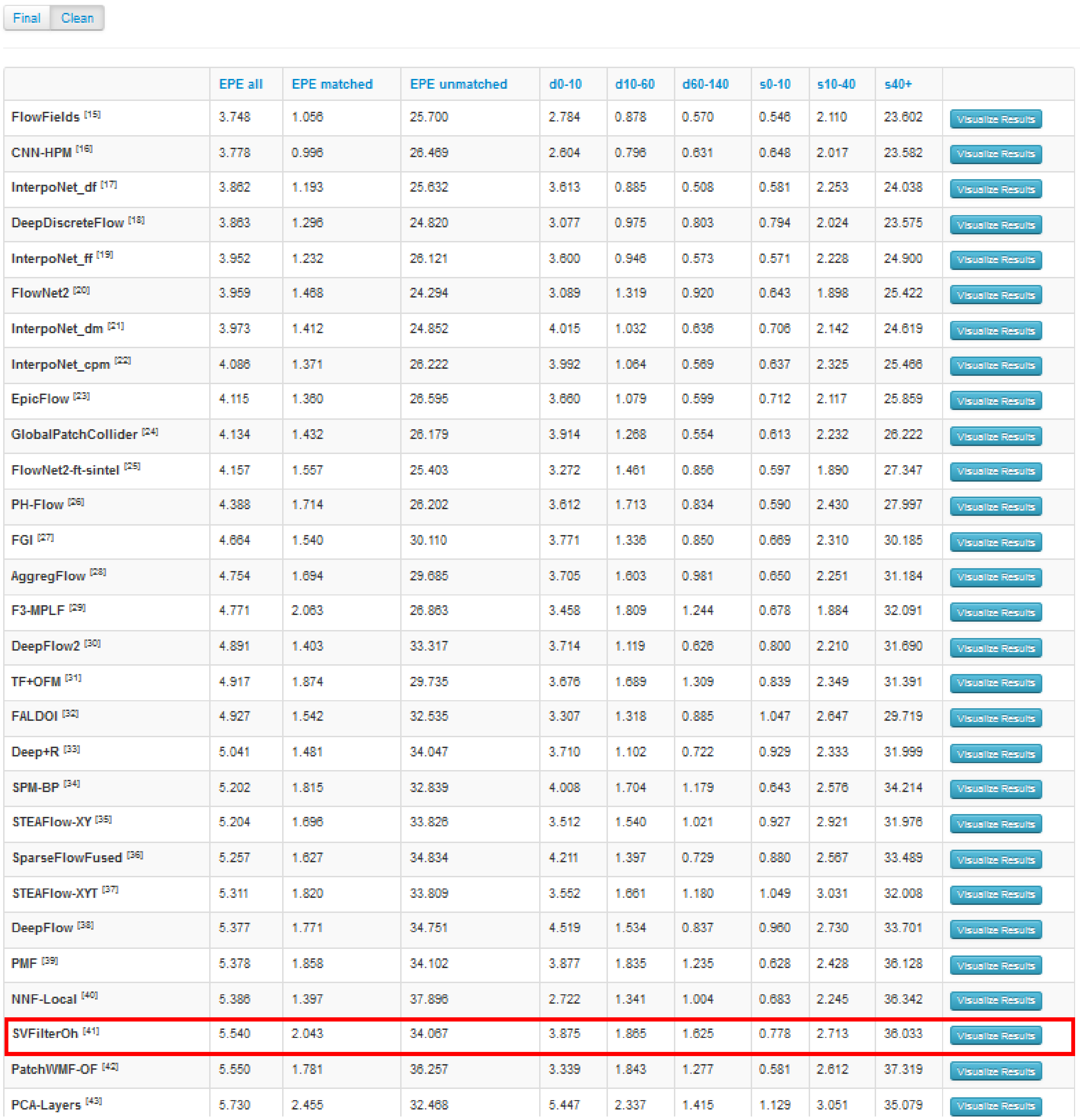
Average endpoint error	avg.	Army (Hidden texture)			Mequon (Hidden texture)			Schefflera (Hidden texture)			Wooden (Hidden texture)			Grove (Synthetic)			Urban (Synthetic)			Yosemite (Synthetic)			Teddy (Stereo)		
		GT	im0	im1	GT	im0	im1	GT	im0	im1	GT	im0	im1	GT	im0	im1	GT	im0	im1	GT	im0	im1	GT	im0	im1
	rank	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext
NNF-Local [87]	3.4	0.07 1	0.20 2	0.05 1	0.15 2	0.51 4	0.12 5	0.18 2	0.37 2	0.14 2	0.10 2	0.49 5	0.06 2	0.41 1	0.61 1	0.21 2	0.23 2	0.66 3	0.19 2	0.10 5	0.12 12	0.17 13	0.34 2	0.80 6	0.23 2
PMMST [114]	9.2	0.09 32	0.21 5	0.07 16	0.18 11	0.51 4	0.16 28	0.21 9	0.42 8	0.17 15	0.10 2	0.33 1	0.08 12	0.51 5	0.74 4	0.28 7	0.24 3	0.65 2	0.20 5	0.11 17	0.12 12	0.17 13	0.37 4	0.74 2	0.35 4
OFLAF [77]	9.5	0.08 9	0.21 5	0.06 6	0.16 6	0.53 6	0.12 5	0.19 3	0.37 2	0.14 2	0.14 8	0.77 28	0.07 5	0.51 5	0.78 7	0.25 4	0.31 10	0.76 4	0.25 15	0.11 17	0.12 12	0.21 39	0.42 10	0.78 4	0.63 17
MDP-Flow2 [68]	10.3	0.08 9	0.21 5	0.07 16	0.15 2	0.48 1	0.11 1	0.20 5	0.40 5	0.14 2	0.15 21	0.80 36	0.08 12	0.63 19	0.93 19	0.43 20	0.26 4	0.76 4	0.23 8	0.11 17	0.12 12	0.17 13	0.38 6	0.79 5	0.44 6
NN-field [71]	11.6	0.08 9	0.22 17	0.05 1	0.17 8	0.55 10	0.13 11	0.19 3	0.39 4	0.15 7	0.09 1	0.48 4	0.05 1	0.41 1	0.61 1	0.20 1	0.52 59	0.64 1	0.26 18	0.13 41	0.13 35	0.20 33	0.35 3	0.83 8	0.21 1
ComponentFusion [96]	13.6	0.07 1	0.21 5	0.05 1	0.16 6	0.55 10	0.12 5	0.20 5	0.44 9	0.15 7	0.11 4	0.65 8	0.06 2	0.71 35	1.07 40	0.53 37	0.32 14	1.06 26	0.28 21	0.11 17	0.13 35	0.15 8	0.41 9	0.88 13	0.54 9
TC/T-Flow [76]	19.0	0.07 1	0.21 5	0.05 1	0.19 18	0.68 33	0.12 5	0.28 24	0.66 29	0.14 2	0.14 8	0.86 46	0.07 5	0.67 29	0.98 29	0.49 31	0.22 1	0.82 9	0.19 2	0.11 17	0.11 2	0.30 84	0.50 27	1.02 30	0.64 19
WLIF-Flow [93]	19.5	0.08 9	0.21 5	0.06 6	0.18 11	0.55 10	0.15 23	0.25 18	0.56 20	0.17 15	0.14 8	0.68 9	0.08 12	0.61 16	0.91 17	0.41 18	0.43 34	0.96 15	0.29 27	0.13 41	0.12 12	0.21 39	0.51 32	1.03 33	0.72 37
NNF-EAC [103]	21.0	0.09 32	0.22 17	0.07 16	0.17 8	0.53 6	0.13 11	0.23 11	0.49 12	0.15 7	0.16 35	0.80 36	0.09 27	0.60 13	0.89 13	0.40 16	0.38 24	0.78 6	0.28 21	0.12 30	0.12 12	0.18 25	0.57 45	1.24 49	0.69 31
Layers++ [37]	21.4	0.08 9	0.21 5	0.07 16	0.19 18	0.56 13	0.17 33	0.20 5	0.40 5	0.18 26	0.13 7	0.58 6	0.07 5	0.48 3	0.70 3	0.33 9	0.47 46	1.01 19	0.33 46	0.15 64	0.14 57	0.24 52	0.46 17	0.88 13	0.72 37
LME [70]	22.2	0.08 9	0.22 17	0.06 6	0.15 2	0.49 2	0.11 1	0.30 33	0.64 24	0.31 86	0.15 21	0.78 32	0.09 27	0.66 25	0.96 24	0.53 37	0.33 15	1.18 43	0.28 21	0.12 30	0.12 12	0.18 25	0.44 12	0.91 16	0.61 14
IROF++ [58]	22.4	0.08 9	0.23 24	0.07 16	0.21 31	0.68 33	0.17 33	0.28 24	0.63 23	0.19 38	0.15 21	0.73 21	0.09 27	0.60 13	0.89 13	0.42 19	0.43 34	1.08 29	0.31 36	0.10 5	0.12 12	0.12 4	0.47 19	0.98 23	0.68 30
nLayers [57]	23.1	0.07 1	0.19 1	0.06 6	0.22 38	0.59 16	0.19 54	0.25 18	0.54 16	0.20 47	0.15 21	0.84 42	0.08 12	0.53 7	0.78 7	0.34 11	0.44 38	0.84 10	0.30 32	0.13 41	0.13 35	0.20 33	0.47 19	0.97 22	0.67 28
HAST [109]	24.6	0.07 1	0.20 2	0.05 1	0.18 11	0.54 8	0.13 11	0.17 1	0.32 1	0.12 1	0.15 21	0.90 57	0.06 2	0.49 4	0.74 4	0.22 3	0.58 68	1.09 30	0.44 68	0.19 94	0.17 85	0.47 112	0.32 1	0.64 1	0.33 3
PH-Flow [101]	25.2	0.08 9	0.24 31	0.07 16	0.21 31	0.68 33	0.17 33	0.23 11	0.49 12	0.19 38	0.16 35	0.83 40	0.09 27	0.56 9	0.83 9	0.38 13	0.30 8	0.81 7	0.24 13	0.15 64	0.13 35	0.30 84	0.43 11	0.85 9	0.66 26
FC-2Layers-FF [74]	25.2	0.08 9	0.21 5	0.07 16	0.21 31	0.70 40	0.17 33	0.20 5	0.40 5	0.18 26	0.15 21	0.76 27	0.08 12	0.53 7	0.77 6	0.37 12	0.49 52	1.02 20	0.33 46	0.16 75	0.13 35	0.29 79	0.44 12	0.87 12	0.64 19
Correlation Flow [75]	26.0	0.09 32	0.23 24	0.07 16	0.17 8	0.58 15	0.11 1	0.43 63	0.99 65	0.15 7	0.11 4	0.47 3	0.08 12	0.75 41	1.08 41	0.56 42	0.41 30	0.92 13	0.30 32	0.14 51	0.13 35	0.27 68	0.40 8	0.85 9	0.42 5
AGIF+OF [85]	27.1	0.08 9	0.22 17	0.07 16	0.23 52	0.73 45	0.18 44	0.28 24	0.66 29	0.18 26	0.14 8	0.70 12	0.08 12	0.57 10	0.85 10	0.38 13	0.47 46	0.97 16	0.31 36	0.13 41	0.13 35	0.22 45	0.51 32	0.99 26	0.74 46
RNLOD-Flow [121]	27.4	0.07 1	0.20 2	0.06 6	0.19 18	0.68 33	0.13 11	0.33 47	0.79 48	0.17 15	0.14 8	0.73 21	0.07 5	0.69 33	1.03 33	0.48 27	0.37 23	0.99 17	0.29 27	0.16 75	0.16 77	0.29 79	0.45 14	0.88 13	0.65 24
FESL [72]	29.0	0.08 9	0.21 5	0.07 16	0.25 63	0.75 51	0.19 54	0.27 21	0.61 21	0.18 26	0.14 8	0.68 9	0.08 12	0.61 16	0.89 13	0.44 21	0.47 46	1.03 23	0.32 41	0.14 51	0.15 68	0.25 58	0.50 27	0.96 20	0.63 17
Classic+CPF [83]	29.2	0.08 9	0.23 24	0.07 16	0.22 38	0.73 45	0.17 33	0.30 33	0.70 33	0.18 26	0.14 8	0.72 20	0.08 12	0.63 19	0.93 19	0.45 24	0.51 57	1.03 23	0.32 41	0.14 51	0.12 12	0.30 84	0.48 21	0.93 17	0.72 37
ALD-Flow [66]	29.5	0.07 1	0.21 5	0.06 6	0.19 18	0.64 26	0.13 11	0.30 33	0.73 37	0.15 7	0.17 46	0.92 61	0.07 5	0.78 44	1.14 45	0.59 45	0.33 15	1.30 55	0.21 6	0.12 30	0.12 12	0.28 74	0.54 39	1.19 45	0.73 42
COFM [59]	30.0	0.08 9	0.26 44	0.06 6	0.18 11	0.62 21	0.14 19	0.30 33	0.74 39	0.19 38	0.15 21	0.86 46	0.07 5	0.79 45	1.14 45	0.74 70	0.35 20	0.87 12	0.28 21	0.14 51	0.12 12	0.28 74	0.49 23	0.94 18	0.71 36
Sparse-NonSparse [56]	30.0	0.08 9	0.23 24	0.07 16	0.22 38	0.73 45	0.18 44	0.28 24	0.64 24	0.19 38	0.14 8	0.71 17	0.08 12	0.67 29	0.99 32	0.48 27	0.49 52	1.06 26	0.32 41	0.14 51	0.11 2	0.28 74	0.49 23	0.98 23	0.73 42
TC-Flow [46]	30.3	0.07 1	0.21 5	0.06 6	0.15 2	0.59 16	0.11 1	0.31 39	0.78 45	0.14 2	0.16 35	0.86 46	0.08 12	0.75 41	1.11 43	0.54 39	0.42 32	1.40 65	0.25 15	0.11 17	0.12 12	0.29 79	0.62 53	1.35 52	0.93 70
Efficient-NL [60]	30.6	0.08 9	0.22 17	0.06 6	0.21 31	0.67 31	0.17 33	0.31 39	0.73 37	0.18 26	0.14 8	0.71 17	0.08 12	0.59 12	0.88 12	0.39 15	1.30 96	1.35 60	0.67 91	0.14 51	0.13 35	0.26 62	0.45 14	0.85 9	0.55 11
LSM [39]	31.8	0.08 9	0.23 24	0.07 16	0.22 38	0.73 45	0.18 44	0.28 24	0.64 24	0.19 38	0.14 8	0.70 12	0.09 27	0.66 25	0.97 26	0.48 27	0.50 54	1.06 26	0.33 46	0.15 64	0.12 12	0.29 79	0.50 27	0.99 26	0.73 42
Ramp [62]	32.4	0.08 9	0.24 31	0.07 16	0.21 31	0.72 42	0.18 44	0.27 21	0.62 22	0.19 38	0.15 21	0.71 17	0.09 27	0.66 25	0.97 26	0.49 31	0.51 57	1.09 30	0.34 52	0.15 64	0.12 12	0.30 84	0.48 21	0.96 20	0.72 37
Classic+NL [31]	34.7	0.08 9	0.23 24	0.07 16	0.22 38	0.74 49	0.18 44	0.29 30	0.65 28	0.19 38	0.15 21	0.73 21	0.09 27	0.64 22	0.93 19	0.47 25	0.52 59	1.12 35	0.33 46	0.16 75	0.13 35	0.29 79	0.49 23	0.98 23	0.74 46
OAR-Flow [124]	35.1	0.08 9	0.25 37	0.07 16	0.26 70	0.81 68	0.18 44	0.38 55	0.93 58	0.20 47	0.16 35	0.88 52	0.08 12	0.83 52	1.21 53	0.61 48	0.31 10	1.28 51	0.18 1	0.08 2	0.10 1	0.17 13	0.52 37	1.13 40	0.69 31
TV-L1-MCT [64]	35.4	0.08 9	0.23 24	0.07 16	0.24 58	0.77 56	0.19 54	0.32 43	0.76 43	0.19 38	0.14 8	0.69 11	0.09 27	0.72 37	1.03 33	0.60 46	0.54 62	1.10 33	0.35 56	0.11 17	0.12 12	0.20 33	0.54 39	1.04 35	0.84 60
PMF [73]	36.3	0.09 32	0.25 37	0.07 16	0.19 18	0.60 19	0.14 19	0.23 11	0.46 11	0.17 15	0.17 46	0.87 50	0.09 27	0.58 11	0.86 11	0.26 5	0.82 83	1.17 40	0.54 80	0.21 105	0.22 113	0.36 100	0.39 7	0.75 3	0.59 13
FMOF [94]	37.3	0.08 9	0.22 17	0.07 16	0.24 58	0.76 53	0.19 54	0.24 14	0.54 16	0.18 26	0.14 8	0.70 12	0.08 12	0.64 22	0.94 23	0.44 21	1.19 92	1.12 35	0.65 90	0.15 64	0.13 35	0.32 94	0.58 47	1.16 43	0.70 35
SVFilterOh [111]	38.3	0.10 46	0.24 31	0.08 43	0.21 31	0.62 21	0.15 23	0.24 14	0.51 15	0.17 15	0.16 35	0.84 42	0.09 27	0.61 16	0.92 18	0.27 6	0.81 82	1.19 44	0.46 73	0.21 105	0.20 109	0.42 107	0.37 4	0.80 6	0.44 6
S2F-IF [123]	38.7	0.11 56	0.35 83	0.08 43	0.22 38	0.75 51	0.19 54	0.30 33	0.72 36	0.24 66	0.16 35	0.79 35	0.10 48	0.87 58	1.28 65	0.66 54	0.26 4	1.09 30	0.23 8	0.10 5	0.12 12	0.17 13	0.55 42	1.19 45	0.61 14
IROF-TV [53]	38.8	0.09 32	0.25 37	0.08 43	0.22 38	0.77 56	0.19 54	0.30 33	0.70 33	0.19 38	0.18 53	0.93 64	0.11 54	0.73 39	1.04 36	0.56 42	0.44 38	1.69 87	0.31 36	0.09 4	0.11 2	0.12 4	0.50 27	1.08 38	0.73 42
CombBMOF [113]	39.6	0.10 46	0.29 60	0.07 16	0.22 38	0.65 28	0.16 28	0.25 18	0.55 18	0.17 15	0.16 35	0.74 25	0.11 54	0.67 29	0.98 29	0.44 21	0.60 70	1.04 25	0.54 80	0.17 84	0.17 85	0.25 58	0.51 32	1.06 37	0.64 19
MDP-Flow [26]	40.0	0.09 32	0.25 37	0.08 43	0.19 18	0.54 8	0.18 44	0.24 14	0.55 18	0.20 47	0.16 35	0.91 58	0.09 27	0.74 40	1.06 39	0.61 48	0.46 42	1.02 20	0.35 56	0.12 30	0.14 57	0.17 13	0.78 78	1.68 82	0.97 75
FlowFields [110]	41.3	0.12 76	0.35 83	0.08 43	0.23 52	0.76 53	0.20 68	0.31 39	0.75 40	0.25 69	0.15 21	0.73 21	0.10 48	0.87 58	1.28 65	0.66 54	0.27 6	1.12 35	0.23 8	0.10 5	0.12 12	0.17 13	0.60 50	1.37 54	0.64 19
2DHMM-SAS [92]	41.6	0.08 9	0.24 31	0.07 16	0.23 52	0.78 59	0.17 33	0.42 62	0.90 55	0.22 58	0.15 21	0.75 26	0.09 27	0.65 24	0.96 24	0.48 27	0.56 65	1.13 38	0.34 52	0.15 64	0.13 35	0.30 84	0.56 44	1.13 40	0.79 53
EPPM w/o HM [88]	42.0	0.11 56	0.30 68	0.08 43	0.19 18	0.67 31	0.13 11	0.29 30	0.71 35	0.17 15	0.17 46	0.78 32	0.11 54	0.63 19	0.93 19	0.33 9	0.60 70	1.35 60	0.40 67	0.19 94	0.15 68	0.45 111	0.45 14	0.94 18	0.64 19
MLDP_OF [89]	42.0	0.11 56	0.28 53	0.09 62	0.18 11	0.56 13	0.13 11	0.34 48	0.79 48	0.17 15	0.16 35	0.82 39	0.09 27	0.72 37	1.05 38	0.50 33	0.34 18	1.10 33	0.27 20	0.18 91	0.15 68	0.44 110	0.76 71	1.09 39	0.69 31
Sparse Occlusion [54]	43.3	0.09 32	0.24 31	0.08 43	0.22 38	0.63 23	0.19 54	0.38 55	0.91 56	0.18 26	0.17 46	0.85 45	0.09 27	0.75 41	1.09 42	0.47 25	0.34 18	1.00 18	0.26 18	0.22 109	0.22 113	0.28 74	0.53 38	1.13 40	0.67 28
NL-TV-NCC [25]	43.9	0.10 46	0.26 44	0.08 43	0.22 38	0.72 42	0.15 23	0.35 50	0.85 51	0.16 12	0.15 21	0.70 12	0.09 27	0.79 45	1.16 48	0.51 34	0.78 79	1.38 62	0.48 75	0.16 75	0.15 68	0.26 62	0.55 42	1.16 43	0.55 11
OFH [38]	44.1	0.10 46	0.25 37	0.09 62	0.19 18	0.69 37	0.14 19	0.43 63	1.02 69	0.17 15	0.17 46	1.08 74	0.08 12	0.87 58	1.25 57	0.73 67	0.43 34	1.69 87	0.32 41	0.10 5	0.13 35	0.18 25	0.59 48	1.40 58	0.74 46
CostFilter [40]	44.2	0.10 46	0.27 51	0.08 43	0.20 29	0.63 23	0.15 23	0.22 10	0.45 10	0.18 26	0.19 58	0.88 52	0.12 61	0.60 13	0.90 16	0.28 7	0.75 78	1.19 44	0.50 77	0.21 105	0.24 120	0.40 106	0.46 17	1.02 30	0.62 16
S2D-Matching [84]	44.3	0.09 32	0.26 44	0.07 16	0.23 52	0.80 65	0.18 44	0.38 55	0.93 58	0.20 47	0.15 21	0.70 12	0.09 27	0.70 34	1.03 33	0.51 34	0.55 64	1.17 40	0.35 56	0.17 84	0.13 35	0.32 94	0.51 32	1.01 28	0.81 56
AggregFlow [97]	46.1	0.11 56	0.32 75	0.08 43	0.31 84	0.96 87	0.23 82	0.36 52	0.85 51	0.27 78	0.17 46	0.84 42	0.10 48	0.79 45	1.17 49	0.54 39	0.27 6	0.85 11	0.19 2	0.11 17	0.13 35	0.15 8	0.59 48	1.19 45	0.83 57

Table A.1: Middlebury benchmark evaluation results ranked by the average End-Point Error (EPE) (recorded on January 24th, 2017). Our method is highlighted by a red box. The table lists top-ranked methods.



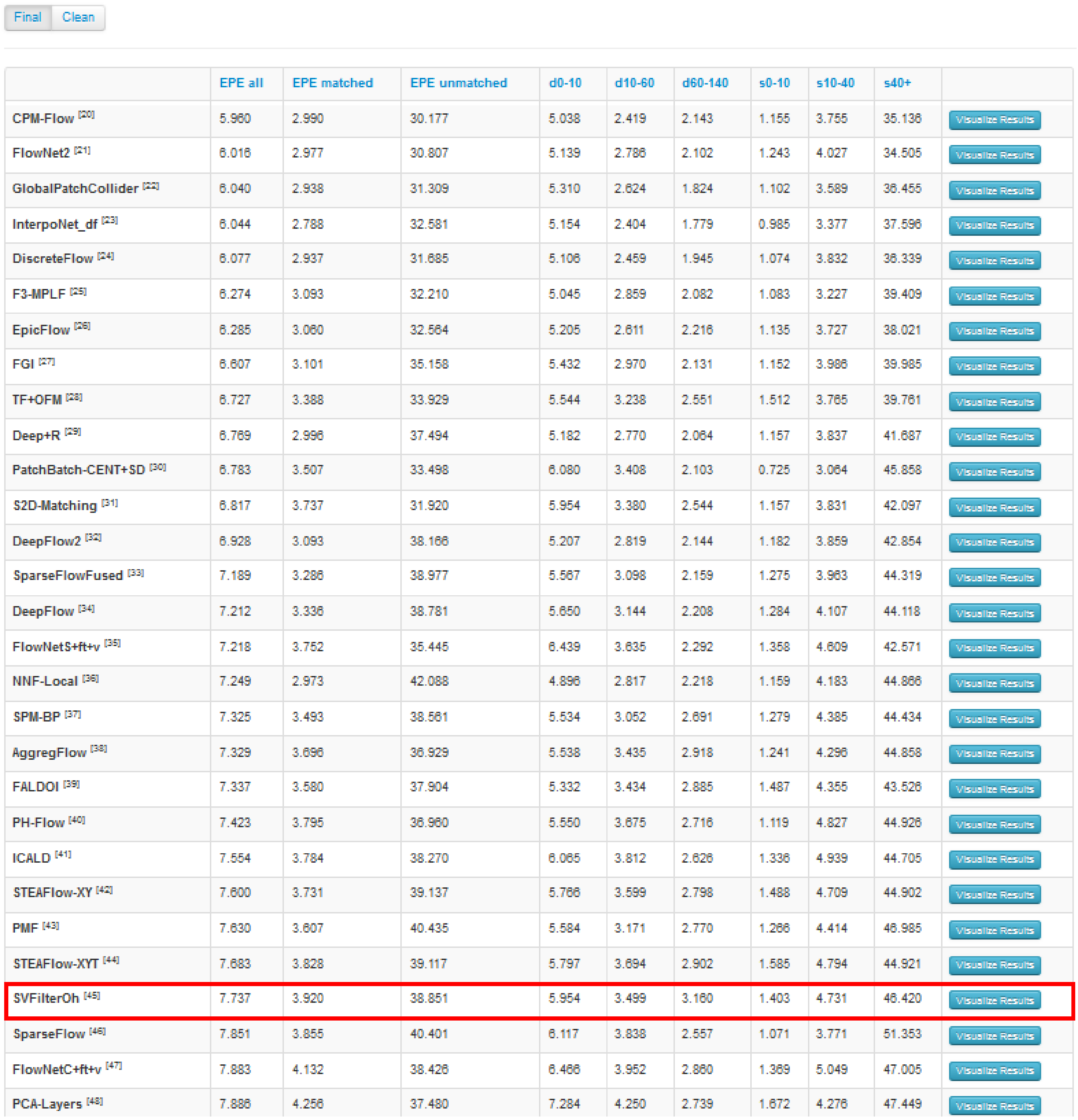
Average angle error	avg. rank	Army (Hidden texture) GT im0 im1			Mequon (Hidden texture) GT im0 im1			Schefflera (Hidden texture) GT im0 im1			Wooden (Hidden texture) GT im0 im1			Grove (Synthetic) GT im0 im1			Urban (Synthetic) GT im0 im1			Yosemite (Synthetic) GT im0 im1			Teddy (Stereo) GT im0 im1		
	rank	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext	all	disc	untext
NNF-Local [87]	4.8	2.69 3	7.56 4	1.98 3	1.97 4	7.01 4	1.59 4	2.18 2	5.36 3	1.53 4	1.87 2	9.14 5	1.06 4	2.28 2	**2.94** 1	1.57 2	2.39 5	6.78 2	2.15 9	2.00 17	3.36 16	1.62 13	**0.99** 1	2.16 2	0.57 2
NN-field [71]	9.3	2.89 8	8.13 16	2.11 5	2.10 6	7.15 9	1.77 14	2.27 4	5.59 5	1.61 8	**1.58** 1	8.52 4	**0.79** 1	2.35 4	3.05 5	1.60 3	**1.89** 1	**5.20** 1	**1.37** 1	2.43 43	3.70 46	1.95 33	1.01 2	2.25 3	**0.53** 1
OFLAF [77]	11.9	3.04 15	7.80 10	2.40 13	2.14 7	7.02 5	1.72 9	2.25 3	5.32 2	1.56 5	2.62 16	13.7 21	1.37 19	2.35 4	3.13 6	1.62 4	2.98 20	7.73 7	2.57 18	2.08 22	3.27 11	2.05 36	1.33 12	2.43 6	1.40 15
PMMST [114]	13.2	3.42 41	7.60 5	2.65 27	2.32 11	**6.39** 1	2.20 30	2.63 11	6.08 8	2.03 24	2.06 4	**6.07** 1	1.44 25	2.60 10	3.27 8	1.91 10	2.56 7	6.78 2	2.09 5	2.06 19	3.53 34	1.63 14	1.27 9	2.29 4	1.02 6
nLayers [57]	15.3	2.80 6	7.42 3	2.20 8	2.71 28	7.24 10	2.55 56	2.61 9	6.24 9	2.45 48	2.30 9	12.7 10	1.16 7	2.30 3	3.02 3	1.70 5	2.62 10	6.95 4	2.09 5	2.29 37	3.46 24	1.89 30	1.38 14	3.06 17	1.29 13
MDP-Flow2 [68]	17.5	3.23 30	7.93 13	2.60 19	1.92 2	6.64 2	**1.52** 1	2.46 7	5.91 7	1.56 5	3.05 39	15.8 48	1.51 35	2.77 23	3.50 17	2.16 25	2.86 16	8.58 18	2.70 29	2.00 17	3.50 31	1.59 11	1.28 10	2.67 11	0.89 4
ComponentFusion [96]	17.9	2.78 5	8.20 17	2.05 4	2.04 5	7.31 11	1.66 8	2.55 8	6.78 13	1.61 8	2.24 8	13.1 12	1.01 3	2.71 20	3.56 19	2.10 21	3.55 48	12.4 52	3.22 54	2.19 32	3.60 40	1.54 10	1.32 11	2.91 13	1.13 8
TC/T-Flow [76]	20.2	2.69 3	7.75 9	1.87 2	2.76 31	10.2 42	1.73 10	3.33 23	9.01 29	1.49 2	2.86 31	16.7 58	1.21 9	2.60 10	3.49 16	1.90 9	2.21 2	7.65 5	2.04 4	1.84 9	3.23 8	3.14 81	2.03 37	4.53 37	1.49 19
FC-2Layers-FF [74]	23.1	3.02 14	7.87 12	2.61 20	2.72 29	9.35 34	2.29 37	2.36 5	5.47 4	2.15 31	2.48 10	12.6 9	1.28 11	2.49 7	3.19 7	2.03 16	3.39 37	8.92 20	2.83 39	2.83 66	3.92 59	2.80 60	1.25 7	2.57 10	1.20 10
WLIF-Flow [93]	24.3	2.96 10	7.67 6	2.40 13	2.41 16	7.70 15	2.10 25	2.98 16	7.63 18	1.97 23	2.71 23	13.5 17	1.33 14	3.01 40	4.00 45	2.40 41	3.03 23	8.32 12	2.44 15	2.09 24	3.36 16	2.04 35	2.26 44	4.97 44	2.59 49
Layers++ [37]	24.6	3.11 17	8.22 20	2.79 38	2.43 19	7.02 5	2.24 32	2.43 6	5.77 6	2.18 34	2.13 6	9.71 6	1.15 6	2.35 4	3.02 3	1.96 11	3.81 54	11.4 40	3.22 54	2.74 61	4.01 64	2.35 46	1.45 15	3.05 16	1.79 28
HAST [109]	24.9	**2.58** 1	**7.12** 1	**1.81** 1	2.41 16	7.05 7	2.10 25	**1.83** 1	**4.19** 1	**1.17** 1	2.84 30	15.5 43	1.08 5	**2.23** 1	2.97 2	**1.40** 1	3.72 53	10.0 31	3.92 75	3.40 85	4.90 90	5.66 110	1.20 6	**2.09** 1	1.24 11
FESL [72]	26.2	2.96 10	7.70 7	2.54 16	3.26 63	10.4 43	2.56 57	3.25 21	8.39 21	2.17 32	2.56 12	13.2 13	1.31 13	2.57 9	3.40 11	2.12 24	2.60 9	7.65 5	2.30 10	2.64 57	4.22 72	2.47 48	1.75 26	3.49 25	1.71 24
Efficient-NL [60]	26.2	2.99 13	8.23 21	2.28 9	2.72 29	8.95 30	2.25 35	3.81 36	9.87 34	2.07 28	2.77 27	14.3 28	1.46 30	2.61 12	3.48 15	1.96 11	3.31 33	8.33 13	2.59 20	2.60 52	3.75 47	2.54 51	1.60 21	3.02 14	1.66 21
AGIF+OF [85]	26.2	3.06 16	8.20 17	2.55 18	3.17 53	10.6 46	2.46 50	3.46 28	8.97 28	2.24 37	2.61 14	13.7 21	1.33 14	2.63 14	3.46 14	2.11 22	2.88 18	8.34 14	2.35 12	2.10 26	3.56 36	2.09 38	1.80 28	3.68 28	2.24 38
LME [70]	26.5	3.15 22	8.04 15	2.31 11	1.95 3	6.65 3	1.59 4	4.03 42	9.31 30	4.57 86	2.69 21	13.6 19	1.42 23	2.85 30	3.61 22	2.42 42	3.47 44	12.8 57	3.17 50	2.12 28	3.53 34	1.73 16	1.34 13	2.75 12	1.18 9
ALD-Flow [66]	26.8	2.82 7	7.86 11	2.16 6	2.84 37	10.1 40	1.86 16	3.73 34	10.4 37	1.67 11	3.10 41	16.8 59	1.28 11	2.69 19	3.60 21	1.85 8	2.79 12	11.3 39	2.32 11	2.07 21	3.25 10	3.10 78	2.03 37	5.11 45	1.94 31
RNLOD-Flow [121]	27.5	2.66 2	7.33 2	2.17 7	2.53 24	9.46 35	1.86 16	3.94 40	10.7 42	1.95 21	2.50 11	13.5 17	1.21 9	2.68 17	3.62 24	2.05 18	2.99 21	8.59 19	2.75 33	3.00 74	4.54 79	3.25 86	1.48 17	3.24 20	1.76 27
IROF++ [58]	28.1	3.17 24	8.69 28	2.61 20	2.79 33	9.61 36	2.33 38	3.43 25	8.86 25	2.38 43	2.87 32	14.8 33	1.52 37	2.74 21	3.57 20	2.19 26	3.20 30	9.70 28	2.71 30	1.96 15	3.45 23	1.22 5	1.80 28	4.06 30	2.50 45
NNF-EAC [103]	28.2	3.31 33	8.21 19	2.68 29	2.19 9	7.49 13	1.76 12	2.73 13	6.62 12	1.70 12	3.18 46	15.8 48	1.64 46	2.87 32	3.66 27	2.24 28	3.02 22	8.07 10	2.59 20	2.19 32	3.48 27	1.74 17	2.85 55	6.52 55	3.12 59
PH-Flow [101]	28.6	3.19 27	8.87 33	2.71 30	2.84 37	9.33 33	2.37 40	2.85 14	7.20 15	2.36 40	2.92 34	15.4 40	1.51 35	2.63 14	3.42 12	2.04 17	3.03 23	8.52 17	2.49 17	2.69 59	3.60 40	3.13 80	1.25 7	2.53 8	1.34 14
Classic+CPF [83]	28.9	3.14 20	8.60 26	2.63 24	3.03 51	10.6 46	2.33 38	3.66 31	9.58 31	2.20 35	2.61 14	14.1 26	1.34 17	2.68 17	3.53 18	2.21 27	2.85 15	7.95 9	2.38 13	2.44 45	3.49 29	2.90 71	1.67 24	3.40 23	2.43 44
Sparse-NonSparse [56]	30.5	3.14 20	8.75 30	2.76 37	3.02 49	10.6 46	2.43 45	3.45 27	8.96 26	2.36 40	2.66 18	13.7 21	1.42 23	2.85 30	3.75 33	2.33 33	3.28 32	9.40 25	2.73 31	2.42 42	3.31 13	2.69 55	1.47 16	3.07 18	1.66 21
TC-Flow [46]	31.4	2.91 9	8.00 14	2.34 12	2.18 8	8.77 25	**1.52** 1	3.84 38	10.7 42	1.49 2	3.13 42	16.6 57	1.46 30	2.78 24	3.73 32	1.96 11	3.08 26	11.4 40	2.66 24	1.94 13	3.43 20	3.20 85	3.06 59	7.04 57	4.08 82
LSM [39]	32.7	3.12 18	8.62 27	2.75 36	3.00 47	10.5 45	2.44 47	3.43 25	8.85 24	2.35 39	2.66 18	13.6 19	1.44 25	2.82 26	3.68 28	2.36 35	3.38 36	9.41 26	2.81 37	2.69 59	3.52 32	2.84 64	1.59 20	3.38 22	1.80 29
SVFilterOh [111]	33.3	3.63 46	8.82 31	2.86 40	2.60 26	8.06 18	2.05 24	2.95 15	7.09 14	2.03 24	2.80 29	13.8 24	1.41 22	2.63 14	3.42 12	1.75 7	3.49 45	10.3 33	3.23 56	3.63 92	5.75 109	4.47 103	1.09 4	2.45 7	0.92 5
Correlation Flow [75]	33.4	3.38 39	8.40 23	2.64 25	2.23 10	7.54 14	1.56 3	5.14 61	13.1 60	1.60 7	2.09 5	8.15 3	1.35 18	3.12 46	4.09 51	2.34 34	4.01 64	11.5 43	4.00 76	2.59 51	3.61 42	3.00 76	1.49 18	3.04 15	1.42 17
Ramp [62]	33.8	3.18 26	8.83 32	2.73 33	2.89 41	10.1 40	2.44 47	3.27 22	8.43 22	2.38 43	2.74 25	14.2 27	1.46 30	2.82 26	3.69 30	2.29 31	3.37 35	9.31 23	2.93 43	2.62 55	3.38 19	3.19 84	1.54 19	3.21 19	2.24 38
PMF [73]	34.1	3.61 44	9.07 35	2.62 22	2.40 14	8.05 17	1.83 15	2.61 9	6.27 10	1.65 10	3.35 55	15.4 40	1.58 41	2.54 8	3.27 8	1.71 6	3.59 49	11.1 38	3.46 62	4.07 101	6.18 114	4.02 101	1.06 3	2.38 5	1.25 12
COFM [59]	34.5	3.17 24	9.90 50	2.46 15	2.41 16	8.34 22	1.92 19	3.77 35	10.5 38	2.54 51	2.71 23	14.9 35	1.19 8	3.08 44	3.92 41	3.25 80	3.83 56	10.9 34	3.15 49	2.20 35	3.35 14	2.91 73	1.62 23	2.56 9	2.09 35
FMOF [94]	35.5	3.12 18	8.23 21	2.73 33	3.25 60	10.7 53	2.52 54	3.01 17	7.61 17	2.20 35	2.56 12	13.4 15	1.33 14	2.75 22	3.61 22	2.24 28	3.66 51	8.50 16	2.78 35	2.62 55	3.84 54	3.27 88	2.66 52	5.69 48	1.95 33
OAR-Flow [124]	36.7	3.37 37	9.87 49	2.67 28	4.22 81	12.8 76	2.87 73	4.95 57	13.4 63	2.66 55	3.23 48	16.4 56	1.37 19	2.83 28	3.82 36	1.97 14	2.49 6	10.9 34	1.87 3	1.52 2	**2.82** 1	1.86 25	1.85 31	4.35 35	1.68 23
Classic+NL [31]	36.7	3.20 29	8.72 29	2.81 39	3.02 49	10.6 46	2.44 47	3.46 28	8.84 23	2.38 43	2.78 28	14.3 28	1.46 30	2.83 28	3.68 28	2.31 32	3.40 38	9.09 22	2.76 34	2.87 68	3.82 53	2.86 68	1.67 24	3.53 26	2.26 41
TV-L1-MCT [64]	37.7	3.16 23	8.48 25	2.71 30	3.28 64	10.8 57	2.60 63	3.95 41	10.5 38	2.38 43	2.69 21	13.9 25	1.45 29	2.94 36	3.79 34	2.63 59	3.50 46	9.75 29	3.06 47	2.08 22	3.35 14	2.29 44	1.95 34	3.89 29	2.71 51
SimpleFlow [49]	41.3	3.35 34	9.20 38	2.98 47	3.18 56	10.7 53	2.71 66	5.06 59	12.6 58	2.70 57	2.95 36	15.1 37	1.58 41	2.91 35	3.79 34	2.47 44	3.59 49	9.49 27	2.99 45	2.39 40	3.46 24	2.24 43	1.60 21	3.56 27	1.57 20
CostFilter [40]	41.5	3.84 49	9.64 45	3.06 49	2.55 25	8.09 19	2.03 22	2.69 12	6.47 11	1.88 18	3.66 64	16.8 59	1.88 56	2.62 13	3.34 10	1.99 15	4.05 65	11.0 37	3.65 69	4.16 103	7.18 121	4.66 105	1.16 5	3.36 21	0.87 3
2DHMM-SAS [92]	43.4	3.19 27	8.89 34	2.71 30	3.20 58	11.5 64	2.38 41	5.19 62	12.2 54	2.73 59	2.92 34	15.2 38	1.53 38	2.79 25	3.65 26	2.27 30	3.45 42	9.34 24	2.78 35	2.66 58	3.56 36	3.07 77	2.34 47	5.12 46	2.97 57
S2D-Matching [84]	44.5	3.36 35	9.66 46	2.86 40	3.19 57	11.1 60	2.46 50	4.86 56	12.9 59	2.47 49	2.67 20	13.2 13	1.44 25	2.87 32	3.72 31	2.38 37	3.45 42	9.76 30	2.95 44	3.05 75	3.79 51	3.30 90	1.95 34	4.16 33	3.00 58
MDP-Flow [26]	44.6	3.48 42	9.46 42	3.10 51	2.45 20	7.36 12	2.41 42	3.21 20	8.31 20	2.78 61	3.18 46	17.8 64	1.70 51	3.03 41	3.87 37	2.60 55	3.43 40	12.6 55	2.81 37	2.19 32	3.88 57	1.60 12	4.13 75	9.96 80	3.86 78
MLDP_OF [89]	44.6	4.13 60	10.3 57	3.60 76	2.34 12	7.70 15	1.88 18	4.23 46	10.9 45	1.87 17	2.74 25	14.6 32	1.37 19	3.10 45	3.91 40	2.48 48	3.40 38	9.00 21	3.79 72	3.46 87	4.20 70	5.55 109	2.31 45	4.64 40	1.98 34
AggregFlow [97]	45.3	4.25 69	11.9 75	3.26 54	4.46 86	13.7 85	3.43 83	4.76 54	12.4 55	3.93 83	3.28 51	15.6 44	1.68 48	2.89 34	3.89 38	2.08 19	2.32 3	7.75 8	2.14 7	2.06 19	3.77 49	1.48 9	2.07 41	4.11 31	2.36 42
CombBMOF [113]	45.7	3.94 53	10.6 61	2.74 35	2.80 34	8.55 24	2.16 28	3.10 19	7.99 19	1.76 13	2.99 37	13.4 15	1.95 60	3.04 42	3.89 38	2.49 49	5.64 91	12.3 50	6.74 105	3.54 88	5.16 97	2.81 61	1.85 31	4.60 39	1.10 7
IROF-TV [53]	45.8	3.40 40	9.29 40	2.95 46	2.99 46	11.1 60	2.53 55	3.81 36	9.81 33	2.44 47	3.25 50	16.9 61	1.78 54	3.27 63	4.10 52	2.93 72	4.47 71	16.0 83	3.53 64	1.70 4	3.21 6	1.12 3	1.91 33	4.75 42	2.19 37
S2F-IF [123]	46.8	4.51 78	13.6 88	3.31 58	2.90 42	10.4 43	2.48 53	4.07 44	10.8 44	3.15 67	3.31 52	15.7 47	1.90 57	3.17 48	4.19 56	2.55 52	2.81 14	11.6 45	2.60 22	1.86 11	3.67 44	1.87 26	2.11 43	4.64 40	2.54 48
FlowFields [110]	48.9	4.57 80	13.7 89	3.38 64	3.01 48	10.6 46	2.59 61	4.19 45	11.1 46	3.30 70	3.17 45	15.0 36	1.96 61	3.21 56	4.24 64	2.61 58	2.91 19	12.4 52	2.66 24	1.84 9	3.46 24	1.84 23	2.50 48	6.15 53	2.79 52
NL-TV-NCC [25]	49.3	3.89 51	9.16 37	2.98 47	2.87 40	9.69 37	1.99 20	4.44 50	11.6 49	1.76 13	2.64 17	11.8 8	1.48 34	3.49 77	4.60 84	2.47 44	4.67 78	13.5 62	4.26 83	2.83 66	4.57 81	2.84 64	2.62 50	6.00 52	2.25 40
Sparse Occlusion [54]	49.9	3.62 45	9.12 36	2.90 42	2.92 44	9.08 31	2.56 57	4.49 51	11.8 52	2.11 30	3.14 43	15.8 48	1.57 40	3.26 61	4.22 60	2.36 35	3.52 47	10.9 34	2.66 24	5.10 117	6.32 115	3.15 82	2.02 36	4.92 43	1.71 24
OFH [38]	50.3	3.90 52	9.77 48	3.62 79	2.84 37	11.0 59	2.04 23	5.52 67	14.4 68	1.89 19	3.52 58	20.5 79	1.60 44	3.18 50	4.06 50	2.82 67	3.86 57	14.1 70	3.59 66	1.77 6	3.62 43	1.81 20	2.64 51	7.08 59	2.15 36

Table A.2: Middlebury benchmark evaluation results ranked by the average Angle Error (EPE) (recorded on January 24th, 2017). Our method is highlighted by a red box.


Final Clean

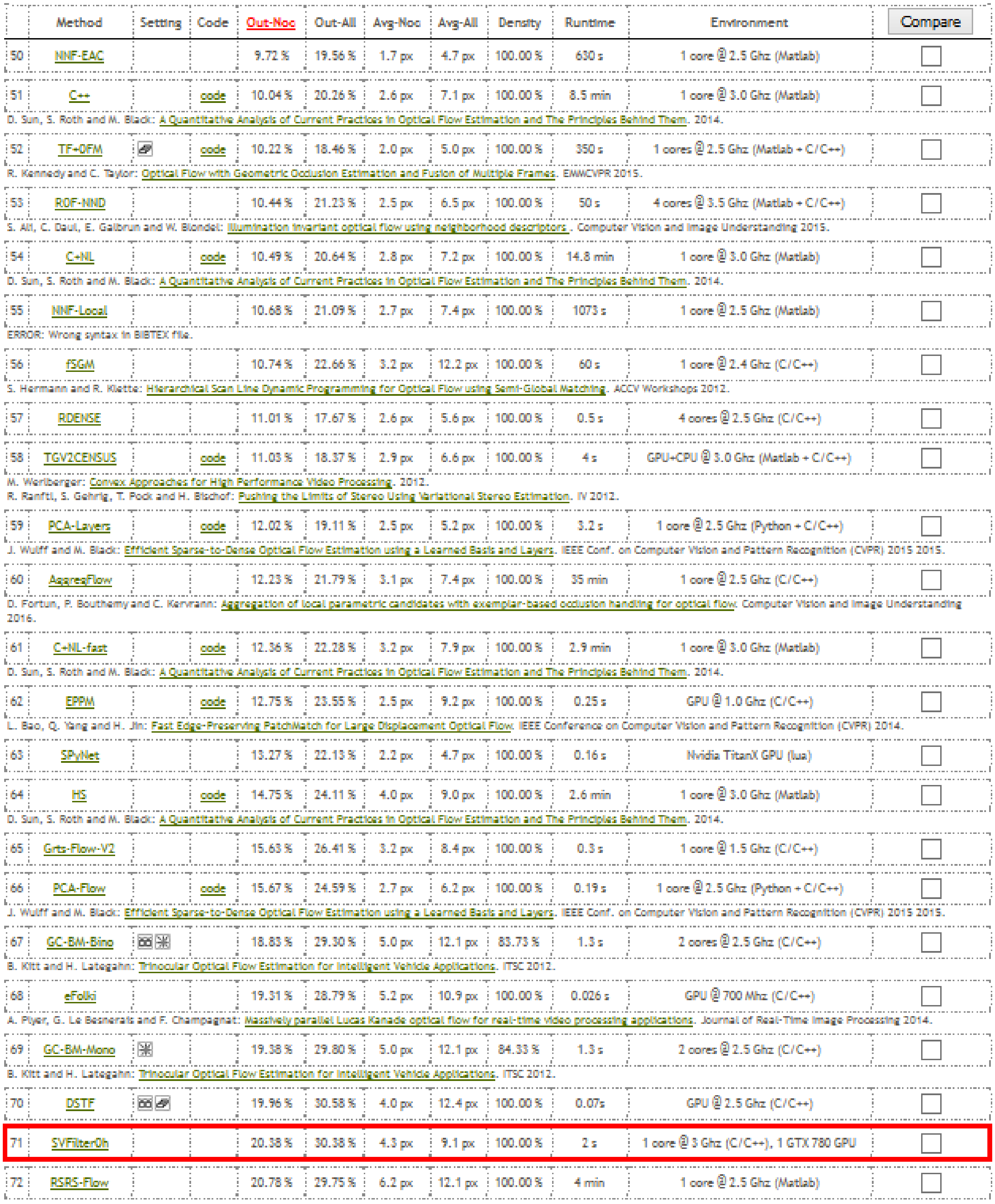
	EPE all	EPE matched	EPE unmatched	d0-10	d10-60	d60-140	s0-10	s10-40	s40+	
FlowFields [15]	3.748	1.056	25.700	2.784	0.878	0.570	0.546	2.110	23.602	Visualize Results
CNN-HPM [16]	3.778	0.996	26.469	2.604	0.796	0.631	0.648	2.017	23.582	Visualize Results
InterpoNet_df [17]	3.862	1.193	25.632	3.613	0.885	0.508	0.581	2.253	24.038	Visualize Results
DeepDiscreteFlow [18]	3.863	1.296	24.820	3.077	0.975	0.803	0.794	2.024	23.575	Visualize Results
InterpoNet_ff [19]	3.952	1.232	26.121	3.600	0.946	0.573	0.571	2.228	24.900	Visualize Results
FlowNet2 [20]	3.959	1.468	24.294	3.089	1.319	0.920	0.643	1.898	25.422	Visualize Results
InterpoNet_dm [21]	3.973	1.412	24.852	4.015	1.032	0.636	0.706	2.142	24.619	Visualize Results
InterpoNet_cpm [22]	4.086	1.371	26.222	3.992	1.064	0.569	0.637	2.325	25.466	Visualize Results
EpicFlow [23]	4.115	1.360	26.595	3.660	1.079	0.599	0.712	2.117	25.859	Visualize Results
GlobalPatchCollider [24]	4.134	1.432	26.179	3.914	1.268	0.554	0.613	2.232	26.222	Visualize Results
FlowNet2-ft-sintel [25]	4.157	1.557	25.403	3.272	1.461	0.856	0.597	1.890	27.347	Visualize Results
PH-Flow [26]	4.388	1.714	26.202	3.612	1.713	0.834	0.590	2.430	27.997	Visualize Results
FGI [27]	4.664	1.540	30.110	3.771	1.336	0.850	0.669	2.310	30.185	Visualize Results
AggregFlow [28]	4.754	1.694	29.685	3.705	1.603	0.981	0.650	2.251	31.184	Visualize Results
F3-MPLF [29]	4.771	2.063	26.863	3.458	1.809	1.244	0.678	1.884	32.091	Visualize Results
DeepFlow2 [30]	4.891	1.403	33.317	3.714	1.119	0.626	0.800	2.210	31.690	Visualize Results
TF+OFM [31]	4.917	1.874	29.735	3.676	1.689	1.309	0.839	2.349	31.391	Visualize Results
FALDOI [32]	4.927	1.542	32.535	3.307	1.318	0.885	1.047	2.647	29.719	Visualize Results
Deep+R [33]	5.041	1.481	34.047	3.710	1.102	0.722	0.929	2.333	31.999	Visualize Results
SPM-BP [34]	5.202	1.815	32.839	4.008	1.704	1.179	0.643	2.576	34.214	Visualize Results
STEAFlow-XY [35]	5.204	1.696	33.826	3.512	1.540	1.021	0.927	2.921	31.976	Visualize Results
SparseFlowFused [36]	5.257	1.627	34.834	4.211	1.397	0.729	0.880	2.567	33.489	Visualize Results
STEAFlow-XYT [37]	5.311	1.820	33.809	3.552	1.661	1.180	1.049	3.031	32.008	Visualize Results
DeepFlow [38]	5.377	1.771	34.751	4.519	1.534	0.837	0.960	2.730	33.701	Visualize Results
PMF [39]	5.378	1.858	34.102	3.877	1.835	1.235	0.628	2.428	36.128	Visualize Results
NNF-Local [40]	5.386	1.397	37.896	2.722	1.341	1.004	0.683	2.245	36.342	Visualize Results
SVFilterOh [41]	5.540	2.043	34.067	3.875	1.865	1.625	0.778	2.713	36.033	Visualize Results
PatchWMF-OF [42]	5.550	1.781	36.257	3.339	1.843	1.277	0.581	2.612	37.319	Visualize Results
PCA-Layers [43]	5.730	2.455	32.468	5.447	2.337	1.415	1.129	3.051	35.079	Visualize Results

Table A.3: MPI Sintel benchmark evaluation results ranked by the average End-Point Error (EPE all) on the clean pass (recorded on January 24th, 2017). Our SVF+OH method is highlighted by a red box. The table lists the top-20 ranked methods.


Final Clean

	EPE all	EPE matched	EPE unmatched	d0-10	d10-60	d60-140	s0-10	s10-40	s40+	
CPM-Flow [20]	5.960	2.990	30.177	5.038	2.419	2.143	1.155	3.755	35.136	Visualize Results
FlowNet2 [21]	6.016	2.977	30.807	5.139	2.786	2.102	1.243	4.027	34.505	Visualize Results
GlobalPatchCollider [22]	6.040	2.938	31.309	5.310	2.624	1.824	1.102	3.589	36.455	Visualize Results
InterpoNet_df [23]	6.044	2.788	32.581	5.154	2.404	1.779	0.985	3.377	37.596	Visualize Results
DiscreteFlow [24]	6.077	2.937	31.685	5.106	2.459	1.945	1.074	3.832	36.339	Visualize Results
F3-MPLF [25]	6.274	3.093	32.210	5.045	2.859	2.082	1.083	3.227	39.409	Visualize Results
EpicFlow [26]	6.285	3.060	32.564	5.205	2.611	2.216	1.135	3.727	38.021	Visualize Results
FGI [27]	6.607	3.101	35.158	5.432	2.970	2.131	1.152	3.986	39.985	Visualize Results
TF+OFM [28]	6.727	3.388	33.929	5.544	3.238	2.551	1.512	3.765	39.761	Visualize Results
Deep+R [29]	6.769	2.996	37.494	5.182	2.770	2.064	1.157	3.837	41.687	Visualize Results
PatchBatch-CENT+SD [30]	6.783	3.507	33.498	6.080	3.408	2.103	0.725	3.064	45.858	Visualize Results
S2D-Matching [31]	6.817	3.737	31.920	5.954	3.380	2.544	1.157	3.831	42.097	Visualize Results
DeepFlow2 [32]	6.928	3.093	38.166	5.207	2.819	2.144	1.182	3.859	42.854	Visualize Results
SparseFlowFused [33]	7.189	3.286	38.977	5.567	3.098	2.159	1.275	3.963	44.319	Visualize Results
DeepFlow [34]	7.212	3.336	38.781	5.650	3.144	2.208	1.284	4.107	44.118	Visualize Results
FlowNetS+ft+v [35]	7.218	3.752	35.445	6.439	3.635	2.292	1.358	4.609	42.571	Visualize Results
NNF-Local [36]	7.249	2.973	42.088	4.896	2.817	2.218	1.159	4.183	44.866	Visualize Results
SPM-BP [37]	7.325	3.493	38.561	5.534	3.052	2.691	1.279	4.385	44.434	Visualize Results
AggregFlow [38]	7.329	3.696	36.929	5.538	3.435	2.918	1.241	4.296	44.858	Visualize Results
FALDOI [39]	7.337	3.580	37.904	5.332	3.434	2.885	1.487	4.355	43.526	Visualize Results
PH-Flow [40]	7.423	3.795	36.960	5.550	3.675	2.716	1.119	4.827	44.926	Visualize Results
ICALD [41]	7.554	3.784	38.270	6.065	3.812	2.626	1.336	4.939	44.705	Visualize Results
STEAFlow-XY [42]	7.600	3.731	39.137	5.766	3.599	2.798	1.488	4.709	44.902	Visualize Results
PMF [43]	7.630	3.607	40.435	5.584	3.171	2.770	1.266	4.414	46.985	Visualize Results
STEAFlow-XYT [44]	7.683	3.828	39.117	5.797	3.694	2.902	1.585	4.794	44.921	Visualize Results
SVFilterOh [45]	7.737	3.920	38.851	5.954	3.499	3.160	1.403	4.731	46.420	Visualize Results
SparseFlow [46]	7.851	3.855	40.401	6.117	3.838	2.557	1.071	3.771	51.353	Visualize Results
FlowNetC+ft+v [47]	7.883	4.132	38.426	6.466	3.952	2.860	1.369	5.049	47.005	Visualize Results
PCA-Layers [48]	7.886	4.256	37.480	7.284	4.250	2.739	1.672	4.276	47.449	Visualize Results

Table A.4: MPI Sintel benchmark evaluation results ranked by the average End-Point Error (EPE all) on the final pass (recorded on January 24th, 2017). Our SVF+OH method is highlighted by a red box. The table lists top-20 ranked methods.



	Method	Setting	Code	Out-Noc	Out-All	Avg-Noc	Avg-All	Density	Runtime	Environment	Compare
50	NNF-EAC			9.72 %	19.56 %	1.7 px	4.7 px	100.00 %	630 s	1 core @ 2.5 Ghz (Matlab)	☐
51	C++		code	10.04 %	20.26 %	2.6 px	7.1 px	100.00 %	8.5 min	1 core @ 3.0 Ghz (Matlab)	☐
D. Sun, S. Roth and M. Black: A Quantitative Analysis of Current Practices in Optical Flow Estimation and The Principles Behind Them. 2014.											
52	TF+OFM	▨	code	10.22 %	18.46 %	2.0 px	5.0 px	100.00 %	350 s	1 cores @ 2.5 Ghz (Matlab + C/C++)	☐
R. Kennedy and C. Taylor: Optical Flow with Geometric Occlusion Estimation and Fusion of Multiple Frames. EMMCVPR 2015.											
53	ROF-NND			10.44 %	21.23 %	2.5 px	6.5 px	100.00 %	50 s	4 cores @ 3.5 Ghz (Matlab + C/C++)	☐
S. Ali, C. Daul, E. Galbrun and W. Blondel: Illumination invariant optical flow using neighborhood descriptors . Computer Vision and Image Understanding 2015.											
54	C+NL		code	10.49 %	20.64 %	2.8 px	7.2 px	100.00 %	14.8 min	1 core @ 3.0 Ghz (Matlab)	☐
D. Sun, S. Roth and M. Black: A Quantitative Analysis of Current Practices in Optical Flow Estimation and The Principles Behind Them. 2014.											
55	NNF-Local			10.68 %	21.09 %	2.7 px	7.4 px	100.00 %	1073 s	1 core @ 2.5 Ghz (Matlab)	☐
ERROR: Wrong syntax in BIBTEX file.											
56	fSGM			10.74 %	22.66 %	3.2 px	12.2 px	100.00 %	60 s	1 core @ 2.4 Ghz (C/C++)	☐
S. Hermann and R. Klette: Hierarchical Scan Line Dynamic Programming for Optical Flow using Semi-Global Matching. ACCV Workshops 2012.											
57	RDENSE			11.01 %	17.67 %	2.6 px	5.6 px	100.00 %	0.5 s	4 cores @ 2.5 Ghz (C/C++)	☐
58	TGV2CENSUS		code	11.03 %	18.37 %	2.9 px	6.6 px	100.00 %	4 s	GPU+CPU @ 3.0 Ghz (Matlab + C/C++)	☐
M. Werlberger: Convex Approaches for High Performance Video Processing. 2012. R. Ranftl, S. Gehrig, T. Pock and H. Bischof: Pushing the Limits of Stereo Using Variational Stereo Estimation. IV 2012.											
59	PCA-Layers		code	12.02 %	19.11 %	2.5 px	5.2 px	100.00 %	3.2 s	1 core @ 2.5 Ghz (Python + C/C++)	☐
J. Wulff and M. Black: Efficient Sparse-to-Dense Optical Flow Estimation using a Learned Basis and Layers. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) 2015 2015.											
60	AggregFlow			12.23 %	21.79 %	3.1 px	7.4 px	100.00 %	35 min	1 core @ 2.5 Ghz (C/C++)	☐
D. Fortun, P. Bouthemy and C. Kervrann: Aggregation of local parametric candidates with exemplar-based occlusion handling for optical flow. Computer Vision and Image Understanding 2016.											
61	C+NL-fast		code	12.36 %	22.28 %	3.2 px	7.9 px	100.00 %	2.9 min	1 core @ 3.0 Ghz (Matlab)	☐
D. Sun, S. Roth and M. Black: A Quantitative Analysis of Current Practices in Optical Flow Estimation and The Principles Behind Them. 2014.											
62	EPPM		code	12.75 %	23.55 %	2.5 px	9.2 px	100.00 %	0.25 s	GPU @ 1.0 Ghz (C/C++)	☐
L. Bao, Q. Yang and H. Jin: Fast Edge-Preserving PatchMatch for Large Displacement Optical Flow. IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2014.											
63	SPyNet			13.27 %	22.13 %	2.2 px	4.7 px	100.00 %	0.16 s	Nvidia TitanX GPU (lua)	☐
64	HS		code	14.75 %	24.11 %	4.0 px	9.0 px	100.00 %	2.6 min	1 core @ 3.0 Ghz (Matlab)	☐
D. Sun, S. Roth and M. Black: A Quantitative Analysis of Current Practices in Optical Flow Estimation and The Principles Behind Them. 2014.											
65	Grts-Flow-V2			15.63 %	26.41 %	3.2 px	8.4 px	100.00 %	0.3 s	1 core @ 1.5 Ghz (C/C++)	☐
66	PCA-Flow		code	15.67 %	24.59 %	2.7 px	6.2 px	100.00 %	0.19 s	1 core @ 2.5 Ghz (Python + C/C++)	☐
J. Wulff and M. Black: Efficient Sparse-to-Dense Optical Flow Estimation using a Learned Basis and Layers. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) 2015 2015.											
67	GC-BM-Bino	▨ ▨		18.83 %	29.30 %	5.0 px	12.1 px	83.73 %	1.3 s	2 cores @ 2.5 Ghz (C/C++)	☐
B. Kitt and H. Lategahn: Trinocular Optical Flow Estimation for Intelligent Vehicle Applications. ITSC 2012.											
68	eFolki			19.31 %	28.79 %	5.2 px	10.9 px	100.00 %	0.026 s	GPU @ 700 Mhz (C/C++)	☐
A. Plyer, G. Le Besnerais and F. Champagnat: Massively parallel Lucas Kanade optical flow for real-time video processing applications. Journal of Real-Time Image Processing 2014.											
69	GC-BM-Mono	▨		19.38 %	29.80 %	5.0 px	12.1 px	84.33 %	1.3 s	2 cores @ 2.5 Ghz (C/C++)	☐
B. Kitt and H. Lategahn: Trinocular Optical Flow Estimation for Intelligent Vehicle Applications. ITSC 2012.											
70	DSTF	▨ ▨		19.96 %	30.58 %	4.0 px	12.4 px	100.00 %	0.07s	GPU @ 2.5 Ghz (C/C++)	☐
71	SVFilterOh			20.38 %	30.38 %	4.3 px	9.1 px	100.00 %	2 s	1 core @ 3 Ghz (C/C++), 1 GTX 780 GPU	☐
72	RSRS-Flow			20.78 %	29.75 %	6.2 px	12.1 px	100.00 %	4 min	1 core @ 2.5 Ghz (Matlab)	☐

Table A.5: KITTI benchmark evaluation results ranked by percentage of erroneous pixels in non-occluded areas (Out-Noc) (recorded on January 24th, 2017). Our SVF+OH method is highlighted by a red box.

MPI Sintel	time (seconds)	Middlebury	time (seconds)
FlowNetS+ft+v	1.05‡		
FlowNetC+ft+v	1.12‡	PMF	4.12‡
		DeepFlow	13
PCA-Layers	3.2	FlowFields	15
PMF	5.3‡	EpicFlow	16
SParseFlow	10	CostFilter	55
EpicFlow	16.4	ALD-Flow	61
FlowFields	18	IROF++	187
DeepFlow	19	MDP-Flow2	342
SparseFlowFused	20	NN-field	362
Deep+R	142.8	LME	476
DiscreteFlow	180	TF+OFM	600
NNF-Local	673+†	NNF-Local	673+†
TF+OFM	600+†	PH-Flow	800+†
PH-Flow	800+†	S2D-Matching	1200
PatchWMF-OF	520	AggregFlow	1642
AggregFlow	1642+†	TC-Flow	2500
S2D-Matching	1920	nLayers	36150

†Methods reports runtime on standard images (640×480). This runtime will increase when processing MPI Sintel images (1024×436). ‡Runtime is reported on a single GPU. Note that the overall runtime of our method drops to 0.5 seconds (MPI Sintel) when using 4 GPUs.

Table A.6: The runtime of our SVF+OH method compared to the top-ranked methods on the MPI Sintel and Middlebury benchmarks. Our method is highlighted by red. MPI Sintel benchmark does not report runtimes, so runtimes for other methods are collected from the corresponding papers and other benchmarks. Note also that not all runtimes are for GPUs. FlowNetS+ft+v, FlowNetC+ft+v, DeepFlow and EpicFlow methods have very poor accuracy on Middlebury benchmark and do not fall into the top-ranked methods (See Table A.1 and A.2). Our method is the fastest among the top-ranked methods in the Middlebury benchmark.

Additional Results

Table A.6 shows the runtime of our SVF+OH method compared to the top-ranked methods on the MPI Sintel and Middlebury benchmarks. Notice that, FlowNetS+ft+v, FlowNetC+ft+v, DeepFlow and EpicFlow methods have very poor accuracy on Middlebury benchmark and do not fall into the top-ranked methods (See Table A.1 and A.2).

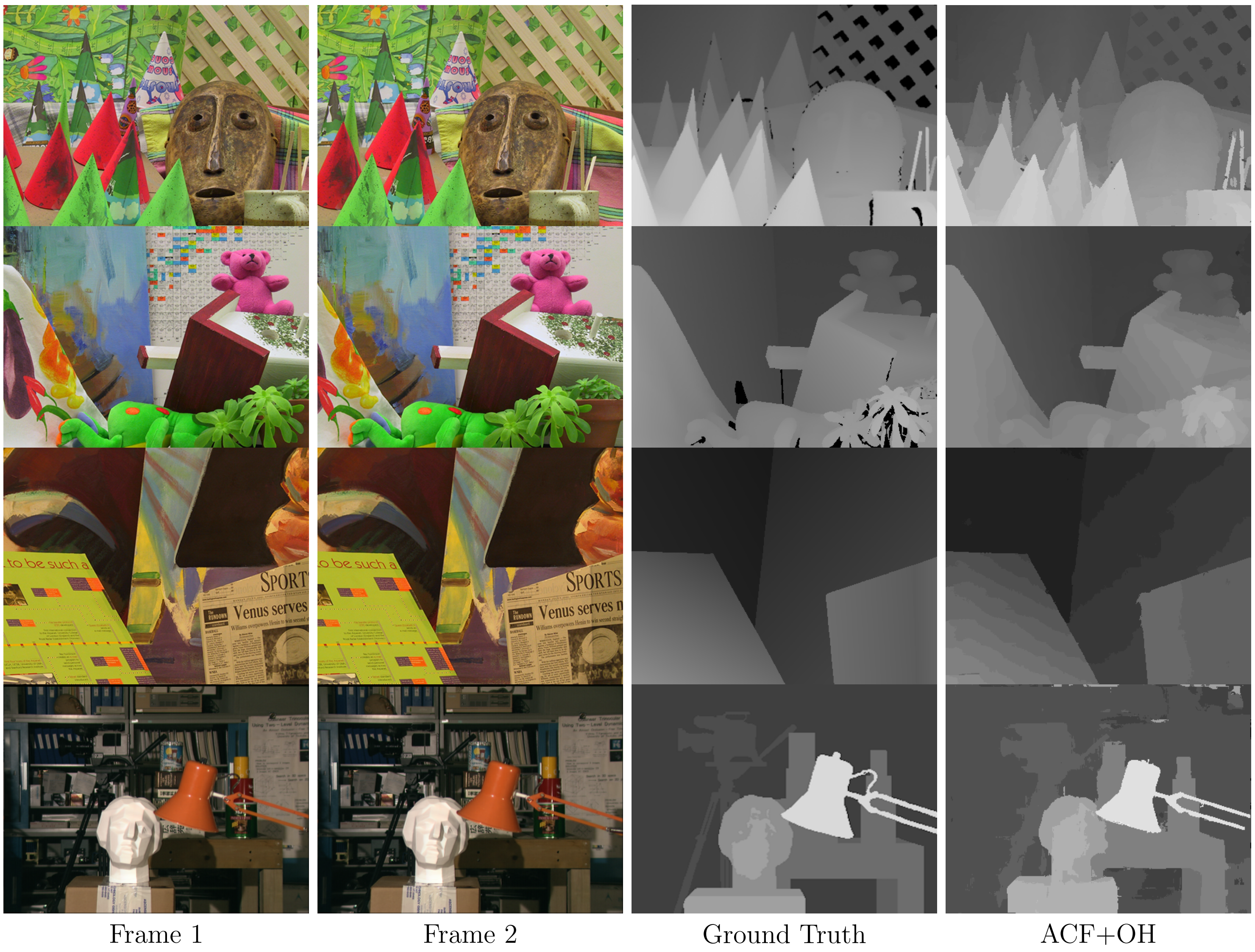


Figure A.1: Results of our ACF+OH method on the four datasets of Middlebury standard stereo benchmark.

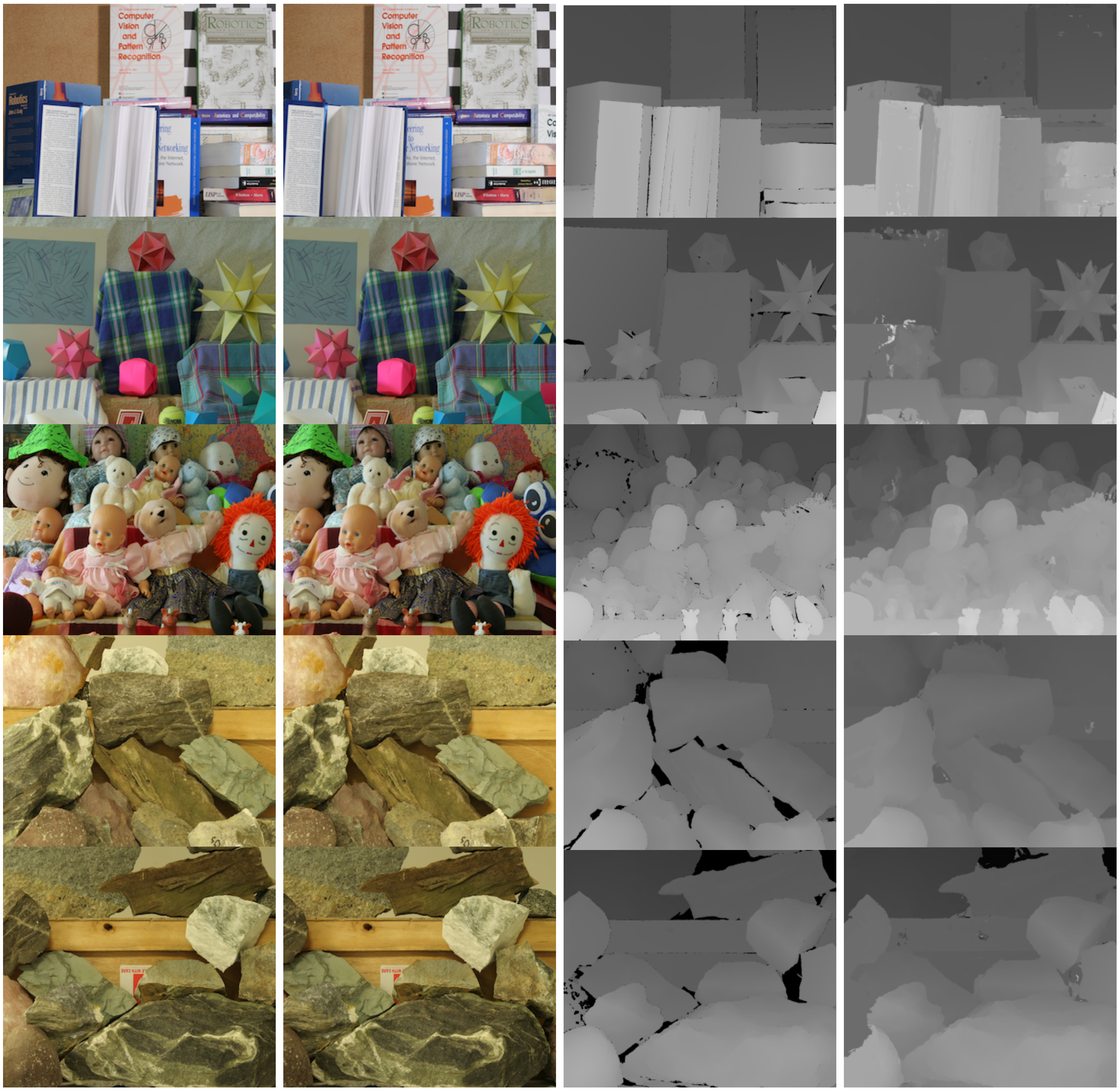
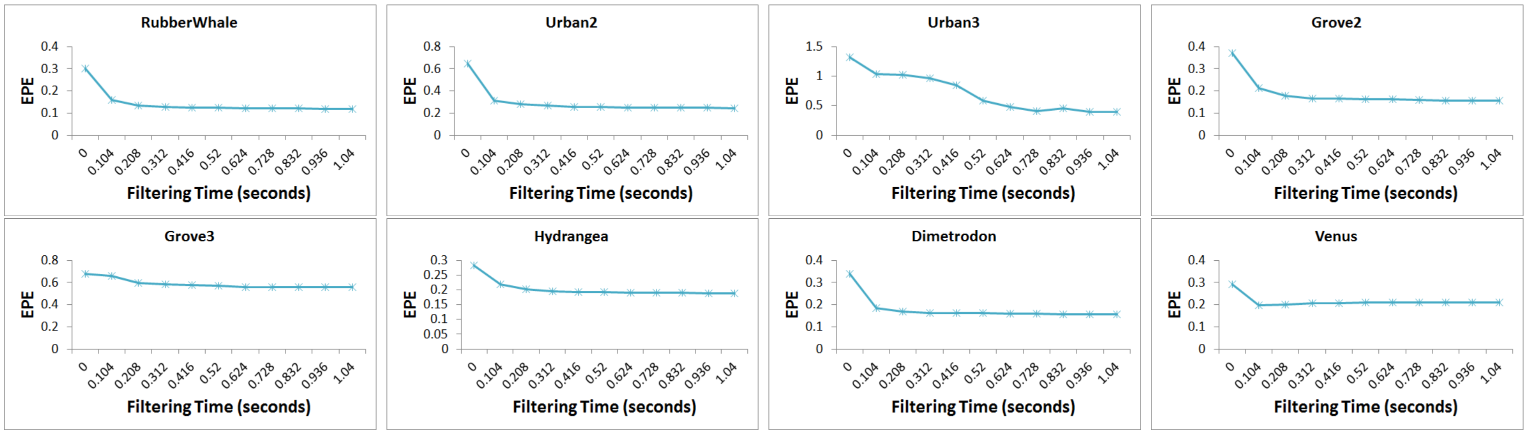
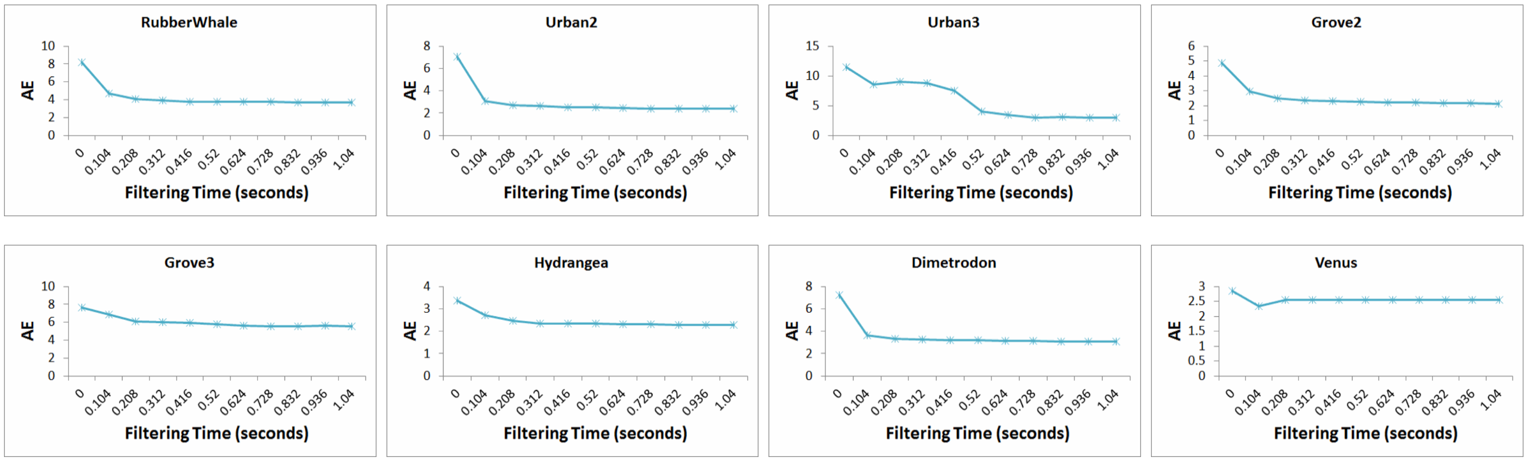


Figure A.2: Results of our ACF+OH method on the Books, Moebius,Dolls, Rocks 1, and Rocks 2 high-resolution Middlebury 2005/2006 stereo datasets.



(a) Average End-Point Error (EPE) versus filtering time for the Middlebury training datasets



(b) Average Angle Error (AE) versus filtering time for the Middlebury training datasets

Figure A.3: The convergence for 10 PatchMatch iterations (baseline is 7) of our SVF+OH method on the Middlebury training datasets.

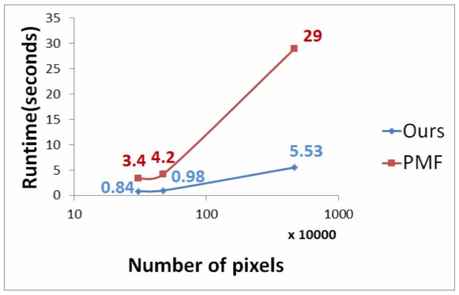


Figure A.4: A comparison of filtering time between our SVF method and PMF. SVF grows linearly and PMF exponentially.

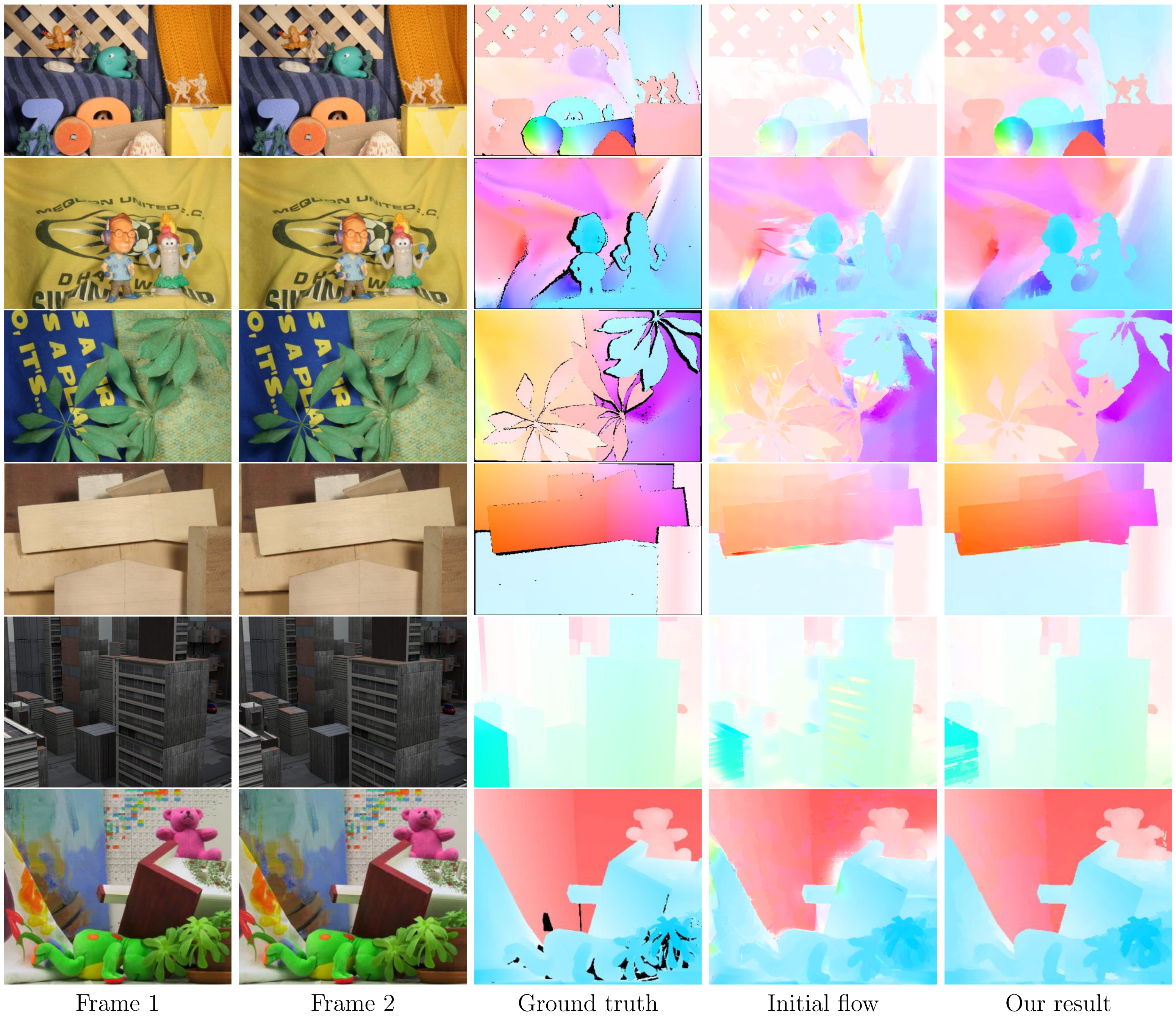


Figure A.5: Example results on the Middlebury optical flow testing benchmark for our SVF+OH method. Note that the initial flow is computed in about 0.2 seconds using the method in [123].

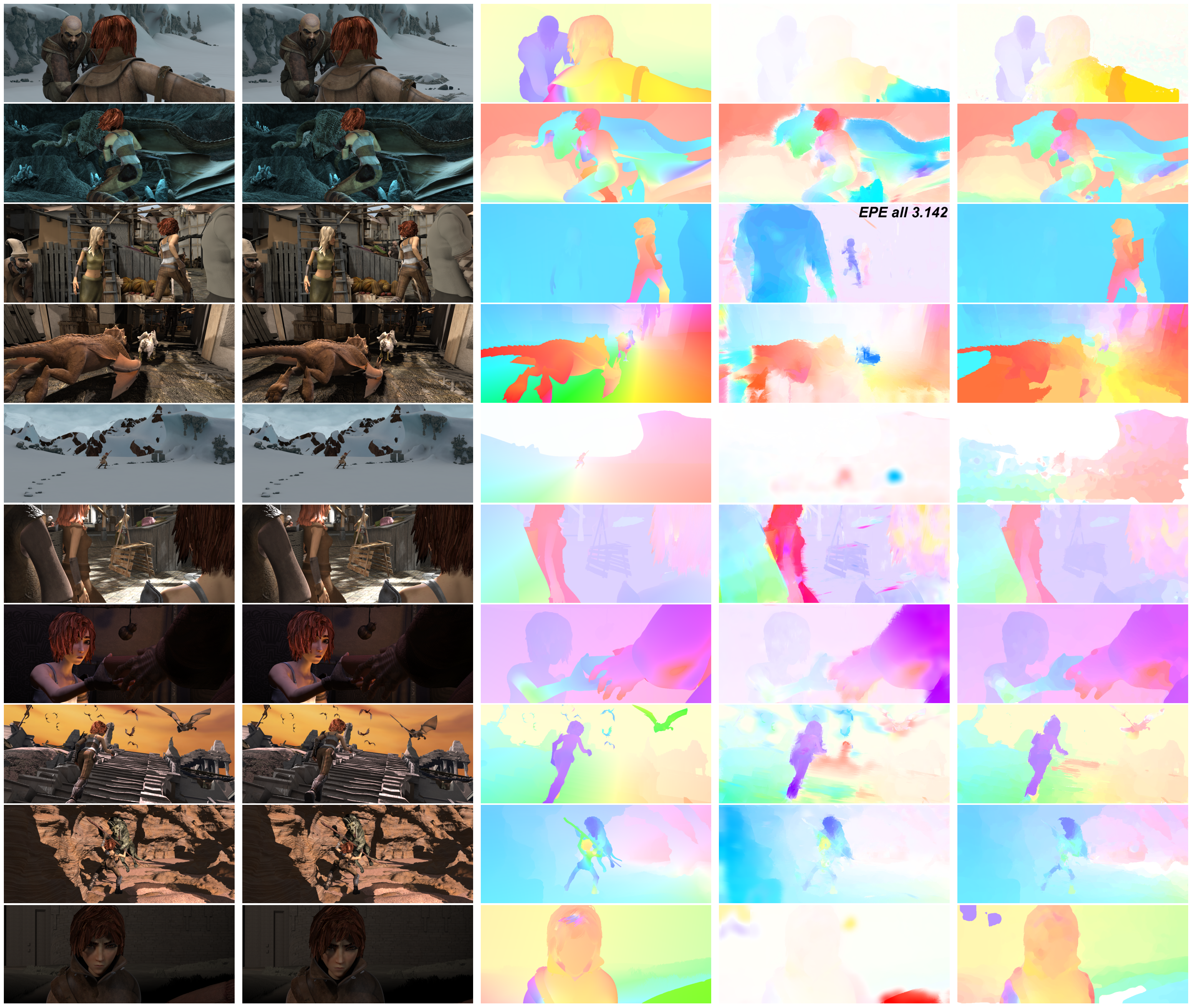


Figure A.6: Example results on the clean pass of the MPI Sintel benchmark for our SVF+OH method. Note that the initial flow is computed in about 0.25 seconds using the method in [123].

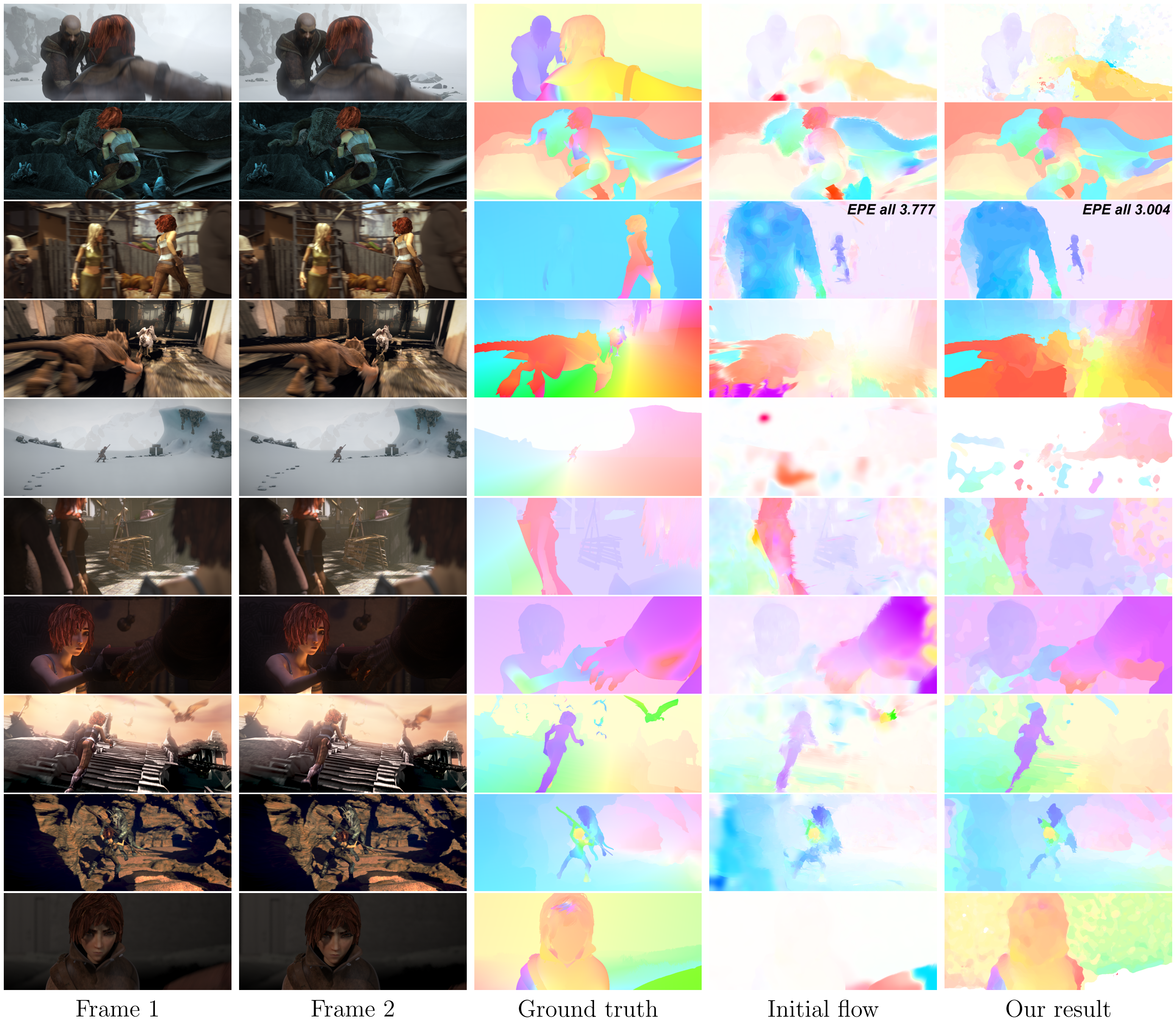


Figure A.7: Example results on the final pass of the MPI Sintel benchmark for our SVF+OH method. Note that the initial flow is computed in about 0.25 seconds using the method in [123].

Appendix B

Gaussian-Process Regression

In the SMBO approach of chapter 6, the model follows a Gaussian stochastic process, formed as a Gaussian process (GP) model. We start by having the input training points $\mathcal{D} = \{(\theta_i, y_i), i = 1,, n\}$, where $y_i = f(\theta_i)$ is the value of the performance metric function for an algorithm or pipeline at the parameter setting θ_i. We assume that the y_i values are noisy samples and define y as a normally distributed random variable with noise ϵ:

$$y = f(\theta) + \epsilon, \qquad \epsilon = \mathcal{N}(0, \sigma_y^2). \tag{B.1}$$

We assume a prior distribution on f that is defined using a GP model, where a GP is a Gaussian distribution over functions. The GP in this case is described by a mean function $m(\theta)$ and a covariance function $k(\theta, \theta')$. Given the set $\mathcal{D}$, we assume that the performance response surface $y = [y_1,, y_n]$ is described by a GP given by the multivariate distribution:

$$y \sim \mathcal{N}(m(X), K + \sigma_y^2 I), \tag{B.2}$$

where $X = [\theta_1, ..., \theta_n]$ and $m(X) = [m(\theta_1),, m(\theta_n)]$ and K is an $n \times n$ matrix with entries $[K]_{i,j} = k(\theta_i, \theta_j)$, and I is the identity matrix. As in [95], we assume a zero mean GP, where $m(X) = 0$. The similarity kernel $k : \Theta \times \Theta \to R^+$ is chosen to measure the similarity between any two different parameter settings $\{\theta, \theta'\} \subset \Theta$. A popular choice of the kernel is the squared exponential covariance function [140], which is defined as:

$$k(\theta_i, \theta_j) = \sigma_f^2 \exp\left(\sum_{k=1}^{d}\left(\frac{(\theta_{ik} - \theta_{jk})^2}{2l_k^2}\right)\right), \tag{B.3}$$

where σ_f and $l_1, ..., l_d$ are kernel parameters defined for parameter settings with d dimensions. The marginal likelihood $p(y|X)$ is given by:

$$p(y|X) = \int p(y|\mathrm{f}, X) p(\mathrm{f}|X) \; d\mathrm{f}, \tag{B.4}$$

$$p(\mathrm{f}|X) = \mathcal{N}(0, K), \tag{B.5}$$

$$p(y|\mathrm{f}, X) = p(y|\mathrm{f}) = \prod_{i=1}^{n} \mathcal{N}(y_i|f_i, \sigma_y^2). \tag{B.6}$$

This results in $p(y|X) = \mathcal{N}(y|0, K_y)$, where $K_y = K + \sigma_y^2 I$. The kernel parameters $[\sigma_y, \sigma_f, l_1, ..., l_d]$ can be learned in closed form by maximizing the log marginal likelihood $\log p(y|X)$ (see [140] for derivations).

Given an arbitrary unseen parameter setting θ_* and using the learned GP model, we can form the joint distribution:

$$\begin{bmatrix} y \\ y_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K_y & K_* \\ K_*^T & K_{**} \end{bmatrix}\right), \tag{B.7}$$

where $K_{**} = k(\theta_*, \theta_*)$ and $K_* = [k(\theta_*, \theta_1), ..., k(\theta_*, \theta_n)]$. Using conditional normal distributions, a predictive distribution $p(y_*|\theta_*, X, y)$ can be calculated as:

$$p(y_*|\theta_*, X, y) = \mathcal{N}(y_*|\mu_*, \Sigma_*), \tag{B.8}$$

$$\mu_* = K_*^T K_y^{-1} y, \tag{B.9}$$

$$\Sigma_* = K_{**} - K_*^T K_y^{-1} K_*. \tag{B.10}$$

Thus, we can use the GP model to calculate the expected value of the performance function $\bar{f}(\theta_*) = \mu_*$ at any arbitrary θ_*. Following SMBO, we use the learned GP model to select the next sample θ_o in the parameter space such that it provides improvement over the optimal setting $(\hat{f}, \hat{\theta}) = \text{argmin}_{(f,\theta)\in\mathcal{D}}\{f\}$ seen so far. We can perform this selection by maximizing an acquisition function. This function describes the desirability or utility of each parameter setting $\theta \in \Theta$ in maximizing the target performance of the metric function $f(\theta)$. Although several acquisition functions have been proposed in the literature, the expected improvement (EI) function is the most popular [140]. This function is defined as:

$$\text{EI}(\theta) = \text{E}\left[max(0, f(\theta) - f(\hat{\theta}))\right], \tag{B.11}$$

where θ is the current best parameter settings seen so far. Thus, the parameter setting that maximizes this function has the expectation of improving over the best setting $\hat{\theta}$ found so far. The important property of EI is that it has a closed-form expression that can be computed under the GP model using integration by parts:

$$\text{EI}(\theta) = \begin{cases} (\mu(\theta) - f(\hat{\theta})\Phi(Z) + \sigma(\theta)\phi(Z) & \text{if } \sigma(\theta) > 0 \\ 0 & \text{if } \sigma(\theta) = 0 \end{cases} \tag{B.12}$$

$$Z = \frac{\mu(\theta) - f(\hat{\theta})}{\sigma(\theta)}, \tag{B.13}$$

where $\phi(Z)$ and $\Phi(Z)$ are the probability density function and the cumulative distribution function of the standard normal distribution, respectively. Equation B.12 has a high value at locations where the expected performance surface $\mu(\theta) = \bar{f}(\theta)$ is larger than the current best value $f(\hat{\theta})$. Furthermore, EI is also high at locations that we did not explore yet, which have large $\sigma(\theta)$ values indicating high uncertainty. Hence, EI allows a trade-off between exploration versus exploitation. We then compute $\theta_o = \text{argmax}_{\theta\in\Theta}\{EI(\theta)\}$. The derivatives of Equation B.12 can be computed analytically, and θ_o can be found using standard gradient solvers (see [140] for derivations).

Appendix C

Go Code for the Algebra Implementation

The Algebra is implemented as a Go language package. The package name is `alg` and it contains four main files: 1) `alg.go` contains the algebra operators, 2) `exgraph.go` contains types and functions for building, scheduling and executing workflow graphs, 3) `consts.go` contains various used constants, and 4) `messages.go` includes the types that define the header and body of messages communicated by the operators.

C.1 `alg.go`

```go
package alg
import (
	"sync"
	"time"
	"uuid" // https://github.com/satori/go.uuid
)

//#################################################################
//                    Basic Types
```

```go
//#################################################################

type T interface{}
type Spout interface {
        Read() T
}
type Cloneable interface {
        Clone() T
}
type Disposable interface {
        Dispose()
}

//#################################################################
//                   Functions
//#################################################################

type Parameter struct {
        Value float64
        Low   float64 // lower bound
        High  float64 // upper bound
}

type Function struct {
        FuncName   string
        FuncParams Params
        State      T
        Mapper     func(T, Params) T
        Reducer    func(T, T, Params) (T, T)
}

type Functions []*Function
type Params map[string]Parameter
```

```

//##################################################################
//                  Processor
//##################################################################

type ProcessorInfo struct {
        Name   string
        _type  int
        Funcs    Functions
        FuncIdx int //Current active Function
}

type NProcessor struct {
        *ProcessorInfo
        WRStatus int
        ERStatus int
        InStack  *ProcessorStack
        OutStack *ProcessorStack
        Inputs   []chan T
        Outputs  []chan T
        F        func(inputs ...chan T) []chan T
        G      *OGraph
        C        *sync.Cond
        Composite bool
}

type Element struct {
        value *ProcessorInfo
        next  *Element
}

type ProcessorStack struct {
        top   *Element
        size  int
        mutex *sync.Mutex
}

func (g *OGraph)  NewProcessor(inchans []chan T, outchans []chan T, _type int) *NProcessor {
        return &NProcessor{ProcessorInfo: &ProcessorInfo{FuncIdx: -1, _type: _type,
                Name: uuid.NewV4().String()},
                Inputs: inchans, Outputs: outchans,
                InStack:  &ProcessorStack{size: 0, mutex: &sync.Mutex{}},
                OutStack: &ProcessorStack{size: 0, mutex: &sync.Mutex{}},
                C:        sync.NewCond(&sync.Mutex{}),
                ERStatus: ST_RUN,
                G : g,
                WRStatus: ST_RESUME,
                Composite: false}
}

func (p *NProcessor) Wait() bool {
        if p.WRStatus == ST_REQWAIT {
                p.C.L.Lock()
                p.WRStatus = ST_WAIT
                p.C.Wait()
                p.WRStatus = ST_RESUME
                p.C.L.Unlock()
                if p.ERStatus == ST_EXIT {
                        return false
                }
        }
        return true
}

func (p *NProcessor) Resume(newERState int) {
        if p.WRStatus == ST_WAIT {
```

```go
		p.C.L.Lock()
		p.ERStatus = newERState
		p.C.Broadcast()
		p.C.L.Unlock()
	}
}

func (p *NProcessor) WaitMessage(x T, chans ...chan T) (bool, bool) {
	if x == nil {
		return false, true
	}
	switch t := x.(type) {
	case *cM:
		if t.end == ""  t.end != p.Name {
			for _, c := range chans {
				c <- x
			}
		}
		p.ERStatus = t.ERStatus
		p.WRStatus = t.WRStatus
		return true, p.Wait()
	default:
	}
	return false, true
}

func (p *NProcessor) ParseAttrib(attribs []T) *NProcessor{
	var pproc *NProcessor = nil
	st := 0
	if len(attribs) > 0{
		switch t := attribs[0].(type){
		case string:
			p.Name = t
```

```
				st = 1
			}
		}
		for i:=st;i<len(attribs); i+=2 {
			switch attribs[i].(int) {
			case OP_ATTRIB_NAME:
				p.Name = attribs[i+1].(string)
			case OP_ATTRIB_FUNC_IDX:
				p.FuncIdx = attribs[i+1].(int)
			case OP_ATTRIB_ER_STATUS:
				p.ERStatus = attribs[i+1].(int)
			case OP_ATTRIB_WR_STATUS:
				p.WRStatus = attribs[i+1].(int)
			case oP_ATTRIB_PREV_PROC:
				pproc = attribs[i+1].(*NProcessor)
			case oP_ATTRIB_COMPOSITE:
				p.Composite = attribs[i+1].(bool)
			}
		}
		return pproc
}

func (p *ProcessorInfo) AddTimeInfo(t int, x T) {

		ct := time.Now()
		switch y := x.(type) {
		case []T:
			p.AddTimeInfo1(t, ct, y)
		case T:
			p.AddTimeInfo1(t, ct, y)
		}
}

```

```
func (proc *ProcessorInfo) AddTimeInfo1(t int, ct time.Time, Z ...T) {
        name := proc.Name
        // record time
        for _, x := range Z {
                if x == nil {
                        continue
                }
                switch c := x.(type) {
                case *M:
                        var (
                                tinfo TimeInfo
                                ok    bool
                        )
                        if c == nil {
                                continue
                        }
                        if tinfo, ok = c.TmInfo[name]; !ok {
                                tinfo = TimeInfo{}
                        }
                        if t == PROC_ENTER_TIME  t == PROC_BOTH_TIME {
                                tinfo.InTime = ct
                        }
                        if t == PROC_LEAVE_TIME  t == PROC_BOTH_TIME {
                                tinfo.OutTime = ct
                        }
                        c.TmInfo[name] = tinfo
                }
        }
}

// Return the stack's length
func (s *ProcessorStack) Len() int {
        return s.size
```

```
}

// Push a new element onto the stack
func (s *ProcessorStack) Push(v *ProcessorInfo) {
        s.top = &Element{value: v, next: s.top}
        s.size++
}

// Remove the top element from the stack and return it's value
// If the stack is empty, return nil
func (s *ProcessorStack) Pop() (value *ProcessorInfo) {
        if s.size > 0 {
                value, s.top = s.top.value, s.top.next
                s.size--
                return
        }
        return nil
}

func (s *ProcessorStack) ExecStack(x T) T {
        if x == nil {
                return nil
        }
        y := x
        for e := s.top; e != nil; e = e.next {
                pi := e.value
                pi.AddTimeInfo(PROC_ENTER_TIME, y)
                if e.value._type == OP_REDUCE {
                        s.mutex.Lock()
            state := e.value.FuncIdx].State
            params := e.value.Funcs[e.value.FuncIdx].FuncParams
                        e.value.Funcs[e.value.FuncIdx].State, y =
                                e.value.Funcs[e.value.FuncIdx].Reducer(e.value.Funcs[state, y,
```

```go
                                        params)
                        s.mutex.Unlock()
                } else {
                        y = e.value.Funcs[e.value.FuncIdx].Mapper(y,
                                e.value.Funcs[e.value.FuncIdx].FuncParams)
                }
                pi.AddTimeInfo(PROC_ENTER_TIME, y)
        }
        return y
}

//###################################################################
//                  Functions for manipulating settings
//###################################################################

func ReadNewSettings(name string, x T) (int, Params) {
        switch t := x.(type) {
        case *M:
                if t != nil {
                        if f, ok := t.FuncInfo[name]; ok {
                                return f.FuncIdx, f.FuncParams
                        }
                }
        }
        return -1, Params{}
}

func UpdateSettings(proc *ProcessorInfo, x T) {
        name := proc.Name

        f_idx, params := ReadNewSettings(name, x)
        if f_idx != -1 {
                proc.FuncIdx = f_idx
```

```go
				proc.Funcs[proc.FuncIdx].FuncParams = params
		}

}

//##############################################################
//##############################################################
// First order operators
//##############################################################
//##############################################################
// 1. Data processing
//##############################################################

// Source
// It joins `group` and returns a Source processor
// which generates a data stream using the given
// spout.
func (g *OGraph) Source(s Spout, attribs ...T) *aGraph {
	proc := g.NewProcessor(nil, []chan T{make(chan T)}, OP_SOURCE)
	g.Register(proc, proc.ParseAttrib(attribs))
	proc.F = func(inputs ...chan T) []chan T {
		g.group.Add(1)
		proc.Inputs = inputs
		go func() {
			defer g.group.Done()
			defer close(proc.Outputs[0])
			var x T
			for {
				t := time.Now()
				x = s.Read()
				if x == nil {
					break
				}
```

```
                                proc.AddTimeInfo1(PROC_ENTER_TIME, t, x)
                                proc.AddTimeInfo(PROC_LEAVE_TIME, x)
                                proc.Outputs[0] <- proc.OutStack.ExecStack(x)
                                if !proc.Wait() {
                                        break
                                }
                        }
                }()
                return proc.Outputs
        }
        return &aGraph{g, proc}
}

// Grounding sink
// It joins `group` and returns a sink processor
// which discards *all* of the inputs from its upstream.
// A sink processor is one that accepts an incoming stream, but
// has no output stream.
func (g *OGraph) Ground(attribs ...T) *aGraph {
        proc := g.NewProcessor(nil, []chan T{}, OP_GROUND)
        g.Register(proc, proc.ParseAttrib(attribs))
        proc.F = func(inputs ...chan T) []chan T {
                g.group.Add(1)
                proc.Inputs = inputs
                go func() {
                        defer g.group.Done()
                        for x := range proc.Inputs[0] {
                                // need to implement a disposing function.
                                comm, state := proc.WaitMessage(x, proc.Outputs...)
                                if !state {
                                        break
                                }
                                if x != nil && !comm {
                                        x = proc.InStack.ExecStack(x)
                                        proc.AddTimeInfo(PROC_ENTER_TIME, x)
                                        (x.(Disposable)).Dispose()
                                        ut := time.Now()
                                        proc.AddTimeInfo1(PROC_LEAVE_TIME, ut, x)
                                }
                        }
                }()
                return proc.Outputs
        }
        return &aGraph{g, proc}
}

// Map processor:
// It joins `group` and uses a function `f`. The returned
// processor applies the function `f` to each input reading `x`
// from the upstream, and writes `f(x)` to the downstream.
func (g *OGraph) Map(funcs Functions, attribs ...T) *aGraph {
        proc := g.NewProcessor(nil, []chan T{make(chan T)}, OP_MAP)
        proc.Funcs, proc.FuncIdx = funcs, 0
        g.Register(proc, proc.ParseAttrib(attribs))
        proc.F = func(inputs ...chan T) []chan T {
                g.group.Add(1)
                proc.Inputs = inputs
                go func() {
                        defer g.group.Done()
                        defer close(proc.Outputs[0])
                        for {
                                x, ok := <-proc.Inputs[0]
                                if !ok {
                                        break
                                }
                                comm, state := proc.WaitMessage(x, proc.Outputs...)
```

```go
                if !state {
                    break
                }
                if !comm {
                    x = proc.InStack.ExecStack(x)
                    proc.AddTimeInfo(PROC_ENTER_TIME, x)
                    UpdateSettings(proc.ProcessorInfo, x)
                    params := proc.Funcs[proc.FuncIdx].FuncParams
                    y := proc.Funcs[proc.FuncIdx].Mapper(x,
                      params)
                    y1 := proc.OutStack.ExecStack(y)
                    proc.AddTimeInfo(PROC_LEAVE_TIME, y1)
                    proc.Outputs[0] <- y1
                }
            }
        }()
        return proc.Outputs
    }
    return &aGraph{g, proc}
}

// Reduce processor:
// It maintains an internal state u which is initialized
// by `u0`. For each reading `x` from the incoming stream,
// reduce updates the state `u` using the `g` function and
// generates an output `y` for the outgoing stream.
func (g *OGraph) Reduce(u0 T, funcs Functions, attribs ...T) *aGraph {
    proc := g.NewProcessor(nil, []chan T{make(chan T)}, OP_REDUCE)
    proc.Funcs, proc.FuncIdx = funcs, 0
    g.Register(proc, proc.ParseAttrib(attribs))
    proc.F = func(inputs ...chan T) []chan T {
        g.group.Add(1)
        u := u0
				proc.Funcs[proc.FuncIdx].State = u0
				proc.Inputs = inputs
				go func() {
					defer g.group.Done()
					defer close(proc.Outputs[0])
					defer (u.(Disposable)).Dispose()
					var y T
					for {
						x, ok := <-proc.Inputs[0]
						if !ok {
							break
						}
						comm, state := proc.WaitMessage(x, proc.Outputs...)
						if !state {
							break
						}
						if !comm {
							if x != nil {
								x = proc.InStack.ExecStack(x)
								proc.AddTimeInfo(PROC_ENTER_TIME, x)
								UpdateSettings(proc.ProcessorInfo, x)
								params := proc.Funcs[proc.FuncIdx].FuncParams
								u, y = proc.Funcs[proc.FuncIdx].Reducer(u, x,
								  params)
								proc.Funcs[proc.FuncIdx].State = u
								y1 := proc.OutStack.ExecStack(y)
								proc.AddTimeInfo(PROC_LEAVE_TIME, y1)
								proc.Outputs[0] <- y1
							} else {
								proc.Outputs[0] <- x
							}
						}
					}
```

```
		}()
		return proc.Outputs
	}
	return &aGraph{g, proc}
}

// Copy processor:
// It makes duplicates o the incoming stream. It is
// important to observe that Copy writes its output
// synchronously on all duplicated outgoing streams.
func (g *OGraph) Copy(n int, attribs ...T) *aGraph {
	outs := make([]chan T, n)
	for i := 0; i < n; i++ {
		outs[i] = make(chan T)
	}
	proc := g.NewProcessor(nil, outs, OP_COPYN)
	g.Register(proc, proc.ParseAttrib(attribs))
	proc.F = func(inputs ...chan T) []chan T {
		proc.Inputs = inputs
		g.group.Add(1)
		go func() {
			defer g.group.Done()
			defer func() {
				for i := 0; i < n; i++ {
					close(proc.Outputs[i])
				}
			}()
			for x := range proc.Inputs[0] {
				comm, state := proc.WaitMessage(x, proc.Outputs...)
				if !state {
					break
				}
				if !comm {
```

```go
					x = proc.InStack.ExecStack(x)
					proc.AddTimeInfo(PROC_ENTER_TIME, x)
					if x != nil {
						for i := 1; i < n; i++ {
							y := (x.(Cloneable)).Clone()
							proc.AddTimeInfo(PROC_LEAVE_TIME, y)
							proc.Outputs[i] <- y
						}
						proc.AddTimeInfo(PROC_LEAVE_TIME, x)
						proc.Outputs[0] <- x
					} else {
						for i := 0; i < n; i++ {
							proc.Outputs[i] <- x
						}
					}
				}

			}
		}()
		return proc.Outputs
	}
	return &aGraph{g, proc}
}

// Filter processor:
// It forwards certain readings from the incoming
// stream that meets the predicate `p` to the first
// output channel `c1` and send the remaining readings
// to `c2`.
func (g *OGraph) Filter(funcs Functions, attribs ...T) *aGraph {
	proc := g.NewProcessor(nil, []chan T{make(chan T), make(chan T)}, OP_FILTER)
	proc.Funcs, proc.FuncIdx = funcs, 0
	g.Register(proc, proc.ParseAttrib(attribs))
	proc.F = func(inputs ...chan T) []chan T {
		proc.Inputs = inputs
		g.group.Add(1)
		go func() {
			defer g.group.Done()
			defer close(proc.Outputs[0])
			defer close(proc.Outputs[1])
			for x := range proc.Inputs[0] {
				comm, state := proc.WaitMessage(x, proc.Outputs...)
				if !state {
					break
				}
				if !comm {
					x = proc.InStack.ExecStack(x)
					proc.AddTimeInfo(PROC_ENTER_TIME, x)
					UpdateSettings(proc.ProcessorInfo, x)
					params := proc.Funcs[proc.FuncIdx].FuncParams
					dec := proc.Funcs[proc.FuncIdx].Mapper(x, params).(bool)
					proc.AddTimeInfo(PROC_LEAVE_TIME, x)
					if dec {
						proc.Outputs[0] <- x
					} else {
						proc.Outputs[1] <- x
					}
				}
			}
		}()
		return proc.Outputs
	}
	return &aGraph{g, proc}
}

// //################################################################
// // 2. Rate Control
// //###########################################################

// Latch processor:
// It allows the incoming and outgoing channels to be
// asynchronous (namely transmitting at different rates).
// it returns two channels, the original input channel `c1` and
// the output channel `c2`.
func (g *OGraph) Latch(attribs ...T) *aGraph {
	proc := g.NewProcessor(nil, []chan T{make(chan T), make(chan T)}, OP_LATCH)
	g.Register(proc, proc.ParseAttrib(attribs))
	proc.F = func(inputs ...chan T) []chan T {
		var (
			u       T
			proceed bool = true
			clone   bool = true
		)
		proc.Inputs = inputs
		g.group.Add(1)
		go func() {
			defer g.group.Done()
			defer close(proc.Outputs[1])

			for x := range inputs[0] {
				comm, state := proc.WaitMessage(x, proc.Outputs...)
				if !state {
					break
				}
				if !comm {
					x = proc.InStack.ExecStack(x)
					proc.AddTimeInfo(PROC_ENTER_TIME, x)
					if x != nil && clone {
						if u != nil {
							(u.(Disposable)).Dispose()
						}
						u = (x.(Cloneable)).Clone()
					}
					proc.AddTimeInfo(PROC_LEAVE_TIME, x)
					proc.Outputs[1] <- x
				}
			}
			proceed = false
		}()
		g.group.Add(1)
		go func() {
			defer g.group.Done()
			defer close(proc.Outputs[0])
			var y T
			for proceed {
				y = u
				clone = false
				proc.AddTimeInfo(PROC_ENTER_TIME, y)
				if y != nil {
					y = (u.(Cloneable)).Clone()
				}
				clone = true
				proc.AddTimeInfo(PROC_LEAVE_TIME, y)
				proc.Outputs[0] <- y
			}
		}()
		return proc.Outputs
	}
	return &aGraph{g, proc}
}

// Cut processor:
```

```go
// It allows the incoming and outgoing channles to be
// asynchronous (namely transmitting at different rates).
// it returns two channel, the original input channel `c1` and
// the output channel `c2`. The operator guarantees that every
// incoming reading is written once to the outgoing channel.
// A nil value is used for the extra write operations.
func (g *OGraph) Cut(attribs ...T) *aGraph {
	proc := g.NewProcessor(nil, []chan T{make(chan T), make(chan T)}, OP_CUT)
	g.Register(proc, proc.ParseAttrib(attribs))
	proc.F = func(inputs ...chan T) []chan T {
		g.group.Add(1)
		var (
			u       T
			proceed bool = true
			clone   bool = true
		)
		proc.Inputs = inputs
		go func() {
			defer g.group.Done()
			defer close(proc.Outputs[1])
			for x := range proc.Inputs[0] {
				comm, state := proc.WaitMessage(x, proc.Outputs...)
				if !state {
					break
				}
				if !comm {
					x = proc.InStack.ExecStack(x)
					proc.AddTimeInfo(PROC_ENTER_TIME, x)
					if x != nil && u == nil && clone {
						u = (x.(Cloneable)).Clone()
					}
					proc.AddTimeInfo(PROC_LEAVE_TIME, x)
					proc.Outputs[1] <- x
```

```go
                                }
                        }
                        proceed = false
                }()
                g.group.Add(1)
                go func() {
                        defer g.group.Done()
                        defer close(proc.Outputs[0])
                        for proceed {
                                y := u
                                proc.AddTimeInfo(PROC_ENTER_TIME, y)
                                clone = false
                                if y != nil {
                                        y = u
                                        u = nil
                                }
                                clone = true
                                proc.AddTimeInfo(PROC_LEAVE_TIME, y)
                                proc.Outputs[0] <- y
                        }
                }()
                return proc.Outputs
        }
        return &aGraph{g, proc}
}

// Multiply processor:
// It reads from two incoming channels inputs[0] and
// inputs[1] and outputs pairs (x1; x2) to the outgoing
// channel `c1`. Unlike Map, multiply synchronizes the
// writes with inputs[0] and latches with inputs[2]. It
// also forwards inputs[2] to `c2`.
func (g *OGraph) LeftMultiply(attribs ...T) *aGraph {
	proc := g.NewProcessor(nil, []chan T{make(chan T), make(chan T)}, OP_LEFT_MULTIPLY)
	g.Register(proc, proc.ParseAttrib(attribs))
	proc.F = func(inputs ...chan T) []chan T {
		var clatch chan T
		// outs := Latch1(group).F(inputs[1])
		proc.Inputs = inputs
		latch1 := g.Latch(oP_ATTRIB_COMPOSITE, true).proc
		close(latch1.Outputs[1])
		latch1.Outputs[1] = proc.Outputs[1]
		outs := latch1.F(inputs[1])
		clatch, proc.Outputs[1] = outs[0], outs[1]
		g.group.Add(1)
		go func() {
			defer g.group.Done()
			defer close(proc.Outputs[0])
			for x := range inputs[0] {
				proc.AddTimeInfo(PROC_ENTER_TIME, x)
				if y, ok := <-clatch; ok {
					proc.AddTimeInfo(PROC_ENTER_TIME, y)
					yy := []T{x, y}
					proc.AddTimeInfo(PROC_LEAVE_TIME, x)
					proc.AddTimeInfo(PROC_LEAVE_TIME, y)
					proc.Outputs[0] <- yy
				}
			}
		}()
		return proc.Outputs
	}
	return &aGraph{g, proc}
}

// Multiply processor:
// It reads from multiple incoming channels and writes to one
// outgoing channel. 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.
func (g *OGraph) Multiply(attribs ...T) *aGraph {
        proc := g.NewProcessor(nil, []chan T{make(chan T)}, OP_MULTIPLY)
        g.Register(proc, proc.ParseAttrib(attribs))
        proc.F = func(inputs ...chan T) []chan T {
                k := 0
                ok := false
                proc.Inputs = inputs
                g.group.Add(1)
                go func() {
                        defer g.group.Done()
                        defer close(proc.Outputs[0])
                        for {
                                k = 0
                                y := make([]T, len(proc.Inputs))
                                for _, x := range proc.Inputs {
                                        y[k], ok = <-x
                                        if ok {
                                                proc.AddTimeInfo(PROC_ENTER_TIME, y[k])
                                                k++
                                        }
                                }
                                if k == 0 {
                                        break
                                }
                                proc.AddTimeInfo1(PROC_LEAVE_TIME, time.Now(), y...)
                                proc.Outputs[0] <-  proc.OutStack.ExecStack(y[0:k])
                        }
                }()
                return proc.Outputs
```

```
	}
	return &aGraph{g, proc}
}

// Add processor:
// It merges multiple incoming channels in a greedy fashion.
// It performs best effort reads on the incoming collection
// of channels asynchronously, and writes to one outgoing
// channel.
func (g *OGraph) Add(attribs ...T) *aGraph {
	proc := g.NewProcessor(nil, []chan T{make(chan T)}, OP_ADD)
	g.Register(proc, proc.ParseAttrib(attribs))
	proc.F = func(inputs ...chan T) []chan T {
		k := len(inputs)
		proc.Inputs = inputs
		for i, cin := range inputs {
			g.group.Add(1)
			go func(i int, cin chan T) {
				defer g.group.Done()
				for x := range cin {
					proc.AddTimeInfo(PROC_ENTER_TIME, x)
					proc.AddTimeInfo(PROC_LEAVE_TIME, x)
					proc.Outputs[0] <- proc.OutStack.ExecStack(x)
				}
				k--
				if k == 0 {
					close(proc.Outputs[0])
				}
			}(i, cin)
		}
		return proc.Outputs
	}
	return &aGraph{g, proc}
```

```
}

// //################################################################
// //################################################################
// // Higher order operators
// //################################################################

// Scatter processor:
// It reads from an incoming channel, but generates a list of
// outgoing channels. The list of outgoing channels can be
// arbitrary size controlled by the fan out parameter `fout`.
// It is parameterized by the generator function `f` and a
// partition function `p`. `f` computes for each incoming value,
// a vector of emitted values to the output channels. `p` maps
// each emitted value to one output channel, and it has the
// signature `p(emitted_element, vector_index, fout)`.
func (g *OGraph) Scatter(f func(T) []T, p func(T, int, int) int, fout int, attribs ...T)
  *aGraph  {
        outs := make([]chan T, fout)
        for i := 0; i < fout; i++ {
                 outs[i] = make(chan T)
        }
    proc := g.NewProcessor(nil, outs, OP_SCATTER)
        g.Register(proc, proc.ParseAttrib(attribs))
        proc.F = func(inputs ...chan T) []chan T {
                proc.Inputs = inputs
                g.group.Add(1)
                go func() {
                        defer g.group.Done()
                        defer func() {
                                for i := 0; i < n; i++ {
                                        close(proc.Outputs[i])
                                }
```

```
			}()
		for x := range proc.Inputs[0] {
			comm, state := proc.WaitMessage(x, proc.Outputs...)
				if !state {
					break
				}
				if !comm {
					x = proc.InStack.ExecStack(x)
					proc.AddTimeInfo(PROC_ENTER_TIME, x)
					if x != nil {
						v := f(x)
						for i, y := range v {
							proc.Outputs[p(y, i, fout)] <- y
						}
					}
				}
			}
		}()
		return proc.Outputs
	}
	return &aGraph{g, proc}
}

// Merge processor:
// It merges a collection of incoming channels back into a
// single outgoing channel. A function `f` continuously receives
// a buffer of the same size as the number of input channels.
// Every element in this buffer contains either an input element
// or nil. `f` must perform a merge or selection operation on the
// buffer and returns the resulted element. 'f' also returns the
// element index in case of a selection operation or -1 in case
// of a merge operation. Note that input channels should be from
// decoupled sources. In case of Scatter, only the merge
// operation is supported.
func (g *OGraph)  Merge(f func([]T) (int, T), attribs ...T)  *aGraph {
	proc := g.NewProcessor(nil, []chan T{make(chan T)}, OP_MERGE)
	g.Register(proc, proc.ParseAttrib(attribs))
	proc.F = func(inputs ...chan T) [] chan T {
		proc.Inputs = inputs
		g.group.Add(1)
		k := len(inputs)
		buf, ok := make([]T, k), make([]bool, k)
		for i := 0; i < k; i++ {
			ok[i] = true
		}
		go func() {
			defer close(proc.Outputs[0])
			defer g.group.Done()
			for k > 0 {
				for i := 0; i < len(proc.Inputs); i++ {
					if buf[i] == nil && ok[i] {
						buf[i], ok[i] = <-proc.Inputs[i]
						if !ok[i] {
							k--
						}
					}
				}
				i, y := f(buf)
				if y != nil {
					proc.Outputs[0] <- y
				}
				if i < 0 {
					buf = make([]T, len(inputs))
				} else {
					buf[i] = nil
				}
```

```
                        }
                }()
                return proc.Outputs
        }
        return &aGraph{g, proc}
}
```

C.2 exgraph.go

```
package alg
import (
        "fmt"
        "graph" // https://github.com/twmb/algoimpl/tree/master/go/graph
        "math"
        "runtime"
        "sync"
        "time"
)

//#################################################################
//                  Workflow Graph
//#################################################################

type ChanInfo struct {
        In_idxs  []int //indecies of input channels
        Out_idxs []int //indecies of output channels
}
type EdgeInfo struct {
        chans      map[string]*ChanInfo // channels used for a graph edge
        n_inchans  int        // number of input channels
        n_outchans int  // number of output channels
}
```

```

type OGraph struct {
        *graph.Graph
        nodes_map    map[string]graph.Node  // a map for storing graph nodes
        split_nodes  map[string]graph.Node  // a map for storing split nodes
        gnd_nodes    map[string]graph.Node  // a map for storing ground nodes
        edges_info   map[string]*EdgeInfo   // a map for storing input and output edges (channels)
        inChan_mask  map[string][]string
        outChan_mask map[string][]string
        Alpha        float64
        DecayInt     float64                // Decay interval
        SchInt       float64                // Scheduling interval
        Active       bool                // Not used now
        NumCpu       int                        // number of cpus for scheduling
        monProc      *NProcessor // Monitor processor
        TL, TP       float64                // Thresholds for Period and Latency
        group                  *sync.WaitGroup
        // updates    map[string]*ProcInfoList
        // views      []ParamsViewer
}

type aGraph struct{
        *OGraph
        proc *NProcessor
}


func NewOGraph() *OGraph {
        return &OGraph{Graph: graph.New(graph.Directed),
                nodes_map:   make(map[string]graph.Node),
                split_nodes: make(map[string]graph.Node),
                gnd_nodes:   make(map[string]graph.Node),
                edges_info:  make(map[string]*EdgeInfo),
		inChan_mask: make(map[string][]string),
		outChan_mask: make(map[string][]string),
		Alpha:      0.2, DecayInt: 5000, SchInt: 10000,
		Active: true, NumCpu: runtime.NumCPU(), monProc: nil,
		TL: 100, TP: 60, group: &sync.WaitGroup{}}
}

func (g *OGraph) AddProc(proc *NProcessor) {
	n := g.MakeNode()
	g.nodes_map[proc.Name] = n
	g.edges_info[proc.Name] = &EdgeInfo{map[string]*ChanInfo{}, 0, 0, nil}
	g.outChan_mask[proc.Name] = make([]string, 0)
	g.inChan_mask[proc.Name] = make([]string, 0)
	proc.G = g
	*g.nodes_map[proc.Name].Value = proc
	if proc._type != OP_MAP && proc._type != OP_REDUCE && proc._type != OP_GROUND {
		g.split_nodes[proc.Name] = n
	}
	if proc._type == OP_GROUND {
		g.gnd_nodes[proc.Name] = n
	}
}

func (g *OGraph) Get(name string) *NProcessor {
	if n, ok := g.nodes_map[name]; ok{
		return (*n.Value).(*NProcessor)
	}else{
		panic(fmt.Sprintf("Couldn't find name %s in Get method", name))
	}
}

func (g *OGraph) Dispose() {}
func (g *OGraph) Connect(name1 string, name2 string, out_idxs, in_idxs []int) {
        g.MakeEdge(g.nodes_map[name1], g.nodes_map[name2])
        for i:=0;i<len(out_idxs); i++{
                if out_idxs[i] < len(g.outChan_mask[name1]){
                        panic(fmt.Sprintf("Output channel %d of operator %s is already occupied
                          by %s, failed to reset it to %s",
                                out_idxs[i], name1, g.outChan_mask[name1][out_idxs[i]], name2))
                }else{
                        g.outChan_mask[name1] = append(g.outChan_mask[name1], name2)
                }
        }
        for i:=0;i<len(in_idxs); i++{
                if in_idxs[i] < len(g.inChan_mask[name2]){
                        panic(fmt.Sprintf("Input channel %d of operator %s is already occupied
                          by %s, failed to reset it to %s",
                                in_idxs[i], name2, g.inChan_mask[name2][in_idxs[i]], name1))
                }else{
                        g.inChan_mask[name2] = append(g.inChan_mask[name2], name1)
                }
        }
        if cinfo, ok := g.edges_info[name2].chans[name1]; ok {
                cinfo.In_idxs = append(cinfo.In_idxs, in_idxs...)
                cinfo.Out_idxs = append(cinfo.Out_idxs, out_idxs...)
        }else{
                g.edges_info[name2].chans[name1] = &ChanInfo{in_idxs, out_idxs}
        }

        g.edges_info[name2].n_inchans = g.edges_info[name2].n_inchans + len(in_idxs)
        g.edges_info[name1].n_outchans = g.edges_info[name1].n_outchans + len(out_idxs)
}

func (g *OGraph) LinkOut(fork string,  ops ...string) {
        // for free output channgels from the fork operator
```

```
	k := len(ops)
	if _, ok := g.nodes_map[fork]; !ok {panic(fmt.Sprintf("Couldn't find fork operator %s
	  in LinkOut", fork))}
	for i := 0; i<k; i++{
		if _, ok := g.nodes_map[ops[i]]; !ok{
			panic(fmt.Sprintf("Couldn't find listed operator %s in LinkOut", ops[i]))
		}
	}
	f_offset := len(g.outChan_mask[fork])
	ops_offsets := make([]int, k)
	for i := 0; i<k; i++{
		ops_offsets[i] = len(g.inChan_mask[ops[i]])
	 }

	// perform the linking operation
	for i := 0; i<k; i++{
		g.Connect(fork, ops[i], []int{f_offset+i}, []int{ops_offsets[i]})
	}
}

func (g *OGraph) LinkIn(join string, ops ...string) {
	// for free output channgels from the fork operator
	k := len(ops)
	if  _, ok := g.nodes_map[join]; !ok {panic(fmt.Sprintf("Couldn't find join operator %s
	  in LinkIn", join))}
	for i := 0; i<k; i++{
		if _, ok := g.nodes_map[ops[i]]; !ok{
			panic(fmt.Sprintf("Couldn't find listed operator %s in LinkIn", ops[i]))
		}
	}
	j_offset := len(g.inChan_mask[join])
	ops_offsets := make([]int, k)
	for i := 0; i<k; i++{
```

```
                ops_offsets[i] = len(g.outChan_mask[ops[i]])
         }

        // perform the linking operation
        for i := 0; i<k; i++{
                g.Connect(ops[i], join, []int{ops_offsets[i]}, []int{j_offset+i})
        }
}

func (g *OGraph) Register(proc, pproc *NProcessor) {
        if proc.Composite {return}
        g.AddProc(proc)
        if pproc != nil{
                idxs := make([]int, len(pproc.Outputs))
                for i,_ := range idxs{idxs[i] = i}
                g.Connect(pproc.Name, proc.Name, idxs, idxs)
        }
}

func (g *OGraph) Wait() {
        g.group.Wait()
}

func (g *aGraph) Source(s Spout, attribs ...T) *aGraph{
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Source(s, attribs...)
}

func (g *aGraph) Ground(attribs ...T) *aGraph{
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Ground(attribs...)
}

```

```
func (g *aGraph) Map(funcs Functions, attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Map(funcs, attribs...)
}

func (g *aGraph) Reduce(u0 T, funcs Functions, attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Reduce(u0, funcs, attribs...)
}

func (g *aGraph) Copy(n int, attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Copy(n, attribs...)
}

func (g *aGraph) Filter(funcs Functions, attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Filter(funcs, attribs...)
}

func (g *aGraph) Latch(attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Latch(attribs...)
}

func (g *aGraph) Cut(attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Cut(attribs...)
}

func (g *aGraph) LeftMultiply(attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.LeftMultiply(attribs...)
```

```
}

func (g *aGraph) Multiply(attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Multiply(attribs...)
}

func (g *aGraph) Add(attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Add(attribs...)
}

func (g *aGraph) Scatter(f func(T) []T, p func(T, int, int) int, fout int,attribs ...T)
  *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Scatter(f, p, fout, attribs...)
}

func (g *aGraph) Merge(f func([]T) (int, T), attribs ...T) *aGraph {
        attribs = append(attribs, oP_ATTRIB_PREV_PROC, g.proc)
        return g.OGraph.Merge(f, attribs...)
}

func (g *OGraph) Execute() {
        for name2, e_info := range g.edges_info {
                chans := make([]chan T, e_info.n_inchans)
                in_proc := (*g.nodes_map[name2].Value).(*NProcessor)
                // now connect chans
                for name1, chan_info := range e_info.chans {
                        out_proc := (*g.nodes_map[name1].Value).(*NProcessor)
                        for i, idx := range chan_info.In_idxs {
                                chans[idx] = out_proc.Outputs[chan_info.Out_idxs[i]]
                        }
                }
                in_proc.F(chans...)
        }
}
```

C.3 messages.go

```
package alg
import(
        "time"
)

type FuncInfo struct {
        FuncIdx    int
        FuncParams Params
}

type TimeInfo struct {
        InTime  time.Time
        OutTime time.Time
}

type MHeader struct {
        FuncInfo  map[string]FuncInfo
        TmInfo  map[string]TimeInfo
        Attribs map[string]T
}

type M struct {
        *MHeader
        Value T
}
```

```

type cM struct {
	start, end string
	WRStatus   int
	ERStatus   int
	value      T
}

func (m *M) Clone() T {
	// copy time info and share OpInfo
	if m == nil {
		return nil
	}
	tinfo := map[string]TimeInfo{}
	for k, v := range m.TmInfo {
		tinfo[k] = v
	}
	if m.Value == nil {
		return &M{&MHeader{FuncInfo: m.FuncInfo,
			TmInfo: tinfo, Attribs: map[string]T{}}, nil}
	} else {
		return &M{&MHeader{FuncInfo: m.FuncInfo,
			TmInfo: tinfo, Attribs: map[string]T{}},
			(m.Value.(Cloneable)).Clone()}
	}

}

func (m *M) Dispose() {
	if m != nil && m.Value != nil {
		(m.Value.(Disposable)).Dispose()
	}
}
```

```

func (m *M) Exists() bool {
	if m == nil {
		return false
	}
	if m.Value == nil {
		return false
	}
	return true
}

func (m *MHeader) AddTimeInfo(tinfo map[string]TimeInfo) {
	if tinfo == nil {
		return
	}
	for k, v := range tinfo {
		if _, ok := m.TmInfo[k]; !ok {
			m.TmInfo[k] = v
		}
	}
}

func NewMessage(v T) *M {
	return &M{&MHeader{FuncInfo: map[string]FuncInfo{},
		TmInfo: map[string]TimeInfo{}, Attribs: map[string]T{}}, v}
}

func MessageV(x T) T {
	if x == nil {
		return nil
	}
	switch t := x.(type) {
	case *M:
		if t == nil {return nil}
		return t.Value
	default:
		return x
	}
}

func MessageH(x T) *MHeader {
	if x == nil {
		return nil
	}
	switch t := x.(type) {
	case *M:
		if t == nil {return nil}
		return t.MHeader
	default:
		return nil
	}
}

func Message(x T) *M {
	if x == nil {
		return nil
	}
	switch t := x.(type) {
	case *M:
		if t == nil {return nil}
		return t
	default:
		return nil
	}
}
```

C.4 consts.go

```go
package alg
const (
	OP_SOURCE int = iota
	OP_GROUND
	OP_MAP
	OP_REDUCE
	OP_FILTER
	OP_COPY
	OP_COPYN
	OP_LATCH
	OP_CUT
	OP_LEFT_MULTIPLY
	OP_MULTIPLY
	OP_ADD
	OP_SCATTER
	OP_MERGE
	OP_MISC
)

const (
	OP_ATTRIB_NAME int = iota
	OP_ATTRIB_FUNC_IDX
	OP_ATTRIB_WR_STATUS
	OP_ATTRIB_ER_STATUS
	oP_ATTRIB_PREV_PROC    // internal use only
	oP_ATTRIB_COMPOSITE
)

const (
	ST_RUN int = iota
	ST_EXIT
)

const (
        ST_REQWAIT int = iota
        ST_WAIT
        ST_RESUME
)

const (
        PROC_ENTER_TIME int = iota
        PROC_LEAVE_TIME
        PROC_BOTH_TIME
)
```

Bibliography

[1] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Susstrunk. SLIC superpixels compared to state-of-the-art superpixel methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(11):2274–2282, Nov 2012.

[2] B. Adenso-Diaz and M. Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operational Research*, 54(1):99–114, Jan 2006.

[3] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *International Conference on Very Large Data Bases*, volume 29, pages 81–92, Sep 2003.

[4] J.M. Alvarez, A.M. Lopez, T. Gevers, and F. Lumbreras. Combining priors, appearance, and context for road detection. *IEEE Transactions on Intelligent Transportation Systems*, 15(3):1168–1178, Jun 2014.

[5] L. Alvarez, J. Weickert, and J. Snchez. Reliable estimation of dense optical flow fields with large displacements. *International Journal of Computer Vision*, 39(1):41–56, Aug 2000.

[6] M. Aly. Real time detection of lane markers in urban streets. In *IEEE Intelligent Vehicles Symposium*, pages 7–12, Jun 2008.

[7] P. Anandan. A computational framework and an algorithm for the measurement of visual motion. *International Journal of Computer Vision*, 2(3):283–310, Jan 1989.

[8] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *ACM Special Interest Group on Management of Data*, pages 577–588, Jun 2013.

[9] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *International Conference on Principles and Practice of Constraint Programming*, pages 142–157, 2009.

[10] Apache Kafka. `http://kafka.apache.org`. Accessed: 2016-07-17.

[11] Apache Spark. `http://Spark.apache.org`. Accessed: 2016-07-17.

[12] Apache Storm. `http://storm.apache.org/`. Accessed: 2016-07-17.

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

[14] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* New York: ACM Press, Addison-Wesley, 1999.

[15] C. Bailer, B. Taetz, and D. Stricker. Flow fields: Dense correspondence fields for highly accurate large displacement optical flow estimation. In *International Conference on Computer Vision*, pages 4015–4023, Dec 2015.

[16] C Bailer, K Varanasi, and D Stricker. Cnn-based patch matching for optical flow with thresholded hinge embedding loss. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2710–2719, Jul 2017.

[17] S. Baker, D. Scharstein, J.P. Lewis, S. Roth, M.J. Black, and R. Szeliski. A database and evaluation methodology for optical flow. *International Journal of Computer Vision*, 92(1):1–31, Mar 2011.

[18] L. Bao, Q. Yang, and H. Jin. Fast edge-preserving patchmatch for large displacement optical flow. *IEEE Transactions on Image Processing*, 23(12):4996–5006, Dec 2014.

[19] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics*, 28(3):24:1–24:11, Jul 2009.

[20] G. Barrett. Model checking in practice: the t9000 virtual channel processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, Feb 1995.

[21] T. Bartz-Beielstein, C. W. G. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *IEEE Congress on Evolutionary Computation*, volume 1, pages 773–780, Sept 2005.

[22] T. Bartz-Beielstein and S. Markon. Tuning search algorithms for real-world applications: A regression tree based approach. In *IEEE Congress on Evolutionary Computation*, pages 1111–1118, Jun 2004.

[23] R. Ben-Ari and N. Sochen. Stereo matching with mumford-shah regularization and occlusion handling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(11):2071–2084, Nov 2010.

[24] A. Benoit, Ü. Çatalyürek, Y. Robert, and E. Saule. A survey of pipelined workflow scheduling: Models and algorithms. *ACM Computing Surveys*, 45(4):50:1–50:36, August 2013.

[25] Anne Benoit, Harald Kosch, Veronika Rehn-Sonigo, and Yves Robert. Multi-criteria scheduling of pipeline workflows (and application to the jpeg encoder). *International Journal of High Performance Computing Applications*, 23(2):171–187, May 2009.

[26] Anne Benoit and Yves Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica*, 57(4):689–724, Aug 2008.

[27] F. Besse, C. Rother, A. Fitzgibbon, and J. Kautz. Pmbp: Patchmatch belief propagation for correspondence field estimation. *International Journal of Computer Vision*, 110(1):2–13, Oct 2014.

[28] D. N. Bhat and S. K. Nayar. Ordinal measures for image correspondence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(4):415–423, Apr 1998.

[29] A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl. Moa: Massive online analysis, a framework for stream classification and clustering. In *Journal of Machine Learning Research*, volume 11, pages 44–50, Sep 2010.

[30] F. Bignone, O. Henricsson, P. Fua, and M. Stricker. Automatic extraction of generic house roofs from high resolution aerial imagery. In *European Conference on Computer Vision*, pages 83–96, Apr 1996.

[31] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, Jul 2002.

[32] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. *F-Race and Iterated F-Race: An Overview*, pages 311–336. Springer Berlin Heidelberg, 2010.

[33] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, Nov 2001.

[34] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[35] M. Brown, G. Hua, and S. Winder. Discriminative learning of local image descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):43–57, Jan 2011.

[36] M. Z. Brown, D. Burschka, and G. D. Hager. Advances in computational stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(8):993–1008, Aug 2003.

[37] T. Brox and J. Malik. Large displacement optical flow: Descriptor matching in variational motion estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(3):500–513, Mar 2011.

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

[39] C. Brust, S. Sickert, M. Simon, E. Rodner, and J. Denzler. Convolutional patch networks with spatial prior for road detection and urban scene understanding. In *International Conference on Computer Vision Theory and Applications*, pages 11–14, March 2015.

[40] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In *European Conference on Computer Vision*, pages 611–625, Oct 2012.

[41] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, Nov 1986.

[42] Y. Cao, D. Barrett, A. Barbu, S. Narayanaswamy, H. Yu, A. Michaux, Y. Lin, S. Dickinson, J.M. Siskind, and S. Wang. Recognize human activities from partially observed videos. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2658–2665, Jun 2013.

[43] J. Carlson and B. Lisper. An event detection algebra for reactive systems. In *ACM International Conference on Embedded Software*, pages 147–154, Sep 2004.

[44] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *International Conference on Very Large Data Bases*, pages 215–226, Aug 2002.

[45] J. Cech and R. Sara. Efficient sampling of disparity space for fast and accurate matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2007.

[46] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, Bradshaw R, and N. Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–375, Jun 2010.

[47] D.P. Chau, J. Badie, F. Bremond, and M. Thonnat. Online tracking parameter adaptation based on evaluation. In *IEEE Conference on Advanced Video and Signal-Based Surveillance*, pages 189–194, Aug 2013.

[48] D.P. Chau, F. Bremond, and M. Thonnat. A multi-feature tracking algorithm enabling adaptation to context variations. In *International Conference on Imaging for Crime Detection and Prevention*, pages 1–6, Nov 2011.

[49] C. Chefd'Hotel and A. Sebbane. Random walk and front propagation on watershed adjacency graphs for multilabel image segmentation. In *International Conference on Computer Vision*, pages 1–7, Oct 2007.

[50] Z. Chen, H. Jin, Z. Lin, S. Cohen, and Y. Wu. Large displacement optical flow from nearest neighbor fields. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2443–2450, Jun 2013.

[51] Z. Chen, Y. Yan, and T. Ellis. Lane detection by trajectory clustering in urban environments. In *International IEEE Conference on Intelligent Transportation Systems*, pages 3076–3081, Oct 2014.

[52] G.B. Chkodrov, P.F. Ringseth, T.T. Tarnavski, A. Shen, R.S. Barga, and J. Goldstein. Implementation of stream algebra over class instances, Google patents, Jan 2013.

[53] E. Corvee and F. Bremond. Body parts detection for people tracking using trees of histogram of oriented gradient descriptors. In *IEEE Conference on Advanced Video and Signal-Based Surveillance*, pages 469–475, Aug 2010.

[54] A. Delong, A. Osokin, H.N. Isack, and Y. Boykov. Fast approximate energy minimization with label costs. *International Journal of Computer Vision*, 96(1):1–27, Jan 2012.

[55] 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.

[56] G. Egnal and R. P. Wildes. Detecting binocular half-occlusions: empirical comparisons of five approaches. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(8):1127–1133, Aug 2002.

[57] P.F. Felzenszwalb and D.P. Huttenlocher. Efficient belief propagation for early vision. *International Journal of Computer Vision*, 70(1):41–54, Oct 2006.

[58] A. Field. *Discovering Statistics Using IBM SPSS*. SAGE Publications, Thousand Oaks, California, United States, 5th edition, 2017.

[59] L. Figueiredo. Adaptive sampling of parametric curves. In *Graphics Gems V*, pages 173–178. Academic Press, 1995.

[60] Philipp Fischer, Alexey Dosovitskiy, Eddy Ilg, Caner Hazirbas Philip Hausser, and Vladimir Golkov. Flownet: Learning optical flow with convolutional networks. In *International Conference on Computer Vision*, pages 2758–2766, Dec 2015.

[61] Robert B. Fisher. Subpixel estimation. In *Computer Vision, A Reference Guide*, pages 775–777. 2014.

[62] D. Fortun, P. Bouthemy, and C. Kervrann. Optical flow modeling and computation: A survey. *Computer Vision and Image Understanding*, 134:1–21, May 2015.

[63] R. Furuta, S. Ikehata, T. Yamasaki, and K. Aizawa. Coarse-to-fine strategy for efficient cost-volume filtering. In *IEEE International Conference on Image Processing*, pages 3793–3797, Oct 2014.

[64] D. Gallup, J.-M. Frahm, P. Mordohai, Y. Qingxiong, and M. Pollefeys. Real-time plane-sweeping stereo with multiple sweeping directions. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, Jun 2007.

[65] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, Jun 2012.

[66] A. Gelman, J. Carlin, H. Stern, D. Dunson, A. Vehtari, and D. Rubin. In *Bayesian Data Analysis*. Florida : CRC Press, 2004.

[67] E. Gheorghiu and C. J. Erkelens. Spatial-scale interaction in human stereoscopic vision in response to sustained and transient stimuli. *Vision Research*, 44(6):563–575, Mar 2004.

[68] V. Granville, M. Krivanek, and J.P. Rasson. Simulated annealing: a proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):652–656, Jun 1994.

[69] M. Grundmann, V. Kwatra, M. Han, and I. Essa. Efficient hierarchical graph-based video segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2141–2148, Jun 2010.

[70] K. Gunhee and E.P. Xing. On multiple foreground cosegmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 837–844, Jun 2012.

[71] H. Han, H. Jung, H. Eom, and H. Yeom. Scatter-gather-merge: An efficient star-join query processing algorithm for data-parallel frameworks. *Cluster Computing*, 14(2):183–197, Jun 2011.

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

[73] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *ACM SIGMOD International Conference on Management of Data*, pages 511–524, Jun 2008.

[74] K. He, J. Sun, and X. Tang. Guided image filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409, Jun 2013.

[75] M. A. Helala, K. Q. Pu, and F. Z. Qureshi. Road boundary detection in challenging scenarios. In *IEEE Conference on Advanced Video and Signal-Based Surveillance*, pages 428–433, Sep 2012.

[76] M. A. Helala, K. Q. Pu, and F. Z. Qureshi. A stream algebra for computer vision pipelines. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 800–807, Jun 2014.

[77] M. A. Helala, K. Q. Pu, and F. Z. Qureshi. Towards efficient feedback control in streaming computer vision pipelines. In *Asian Conference on Computer Vision Workshops*, pages 314–329, Nov 2014.

[78] M. A. Helala, K. Q. Pu, and F. Z. Qureshi. A formal algebra implementation for distributed image and video stream processing. In *International Conference on Distributed Smart Camera*, pages 84–91, Sep 2016.

[79] M. A. Helala and F. Z. Qureshi. Accelerating cost volume filtering using salient subvolumes and robust occlusion handling. In *Asian Conference on Computer Vision*, pages 316–331, Jun 2014.

[80] M. A. Helala and F. Z. Qureshi. Fast estimation of large displacement optical flow using dominant motion patterns & sub-volume patchmatch filtering. In *Conference on Computer and Robot Vision*, pages 64–71, May 2017.

[81] 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):53–20, Oct 2015.

[82] M. A. Helala, M. Selim, and H. Zayed. A content based image retrieval approach based on principal regions detection. *International Journal of Computer Science Issues*, 9(1):204–213, Jul 2012.

[83] M. A. Helala, L. A. Zarrabeitia, and F. Z. Qureshi. Mosaic of near ground uav videos under parallax effects. In *International Conference on Distributed Smart Cameras*, pages 1–6, Oct 2012.

[84] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[85] A.B. Hillel, R. Lerner, D. Levi, and G. Raz. Recent progress in road and lane detection: a survey. *Machine Vision and Applications*, 25(3):727–745, Apr 2014.

[86] H. Hirschmuller, P. R. Innocent, and J. Garibaldi. Real-time correlation-based stereo vision with reduced border errors. *International Journal of Computer Vision*, 47(1):229–246, Apr 2002.

[87] H. Hirschmuller and D. Scharstein. Evaluation of cost functions for stereo matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, Jun 2007.

[88] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[89] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1):185–203, Aug 1981.

[90] A. Hosni, M. Bleyer, M. Gelautz, and C. Rhemann. Local stereo matching using geodesic support weights. In *IEEE International Conference on Image Processing*, pages 2093–2096, Nov 2009.

[91] A. Hosni, C. Rhemann, M. Bleyer, C. Rother, and M. Gelautz. Fast cost-volume filtering for visual correspondence and beyond. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(2):504–511, Feb 2013.

[92] Y. Hu, R. Song, and Y. Li. Efficient coarse-to-fine patchmatch for large displacement optical flow. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 5704–5712, Jun 2016.

[93] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523, Jan 2011.

[94] F. Hutter, H. Hoos, and K. Leyton-Brown. An evaluation of sequential model-based optimization for expensive blackbox functions. In *Annual Conference Companion on Genetic and Evolutionary Computation*, pages 1209–1216, 2013.

[95] F. Hutter, H. Hoos, K. Leyton-Brown, and K. Murphy. Time-bounded sequential parameter optimization. In *International Conference on Learning and Intelligent Optimization*, pages 281–298, Jan 2010.

[96] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, Sep 2009.

[97] F. Hutter, H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *National Conference on Artificial Intelligence*, volume 2, pages 1152–1157, Jul 2007.

[98] J. Supancic III and D. Ramanan. Self-paced learning for long-term tracking. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2379–2386, Columbus, Ohio 2013.

[99] H. Li J. Yang. Dense, accurate optical flow estimation with piecewise parametric model. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1019–1027, Jun 2015.

[100] P. Janert. *Feedback Control for Computer Systems*. O'Reilly Media, Inc., 2013.

[101] D. Jones, M. Schonlau, and W. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, Dec 1998.

[102] K. Kapitanova, S. H. Son, W. Kang, and W. T. Kim. Modeling and analyzing real-time data streams. In *IEEE International Symposium on*

Object/Component/Service-Oriented Real-Time Distributed Computing, pages 91–98, March 2011.

[103] V. Kastrinaki, M. Zervakis, and K. Kalaitzakis. A survey of video processing techniques for traffic applications. *Image and Vision Computing*, 21(4):359–381, Apr 2003.

[104] G. Kim and E.P. Xing. On multiple foreground cosegmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 837–844, Jun 2012.

[105] G. Kim and E.P. Xing. Jointly aligning and segmenting multiple web photo streams for the inference of collective photo storylines. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 620–627, Jun 2013.

[106] K. Kim, T.H. Chalidabhongse, D. Harwood, and L. Davis. Background modeling and subtraction by codebook construction. In *IEEE International Conference on Image Processing*, volume 5, pages 3061–3064, Oct 2004.

[107] Amazon Kinesis. `aws.amazon.com/kinesis/`. Accessed: 2014-02-27.

[108] P. Kisilev and D. Freedman. Parameter tuning by pairwise preferences. In *British Machine Vision Conference*, pages 4.1–4.11, Sep 2010.

[109] H. Kong, J. Audibert, and J. Ponce. General road detection from a single image. *IEEE Transactions on Image Processing*, 19(8):2211–2220, Aug 2010.

[110] S. Korman and S. Avidan. Coherency sensitive hashing. In *International Conference on Computer Vision*, pages 1607–1614, Nov 2011.

[111] P. Kranen, I. Assent, C. Baldauf, and T. Seidl. Self-adaptive anytime stream clustering. In *IEEE International Conference on Data Mining*, pages 249–258, Dec 2009.

[112] K. Kyungnam, T. Chalidabhongse, D. Harwood, and L. Davis. Background modeling and subtraction by codebook construction. In *IEEE International Conference on Image Processing*, volume 5, pages 3061–3064, Oct 2004.

[113] A. Levinshtein, A. Stere, K. Kutulakos, D. Fleet, S. Dickinsonl, and K. Siddiqi. Turbopixels: Fast superpixels using geometric flows. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(12):2290–2297, Dec 2009.

[114] X. Li, Z. Jia, and R. Zhang. Feedback control real-time scheduling over data streams. *Journal of Computational Information Systems*, 6:1051–1059, 04 2010.

[115] E. Liarou, R. Goncalves, and S. Idreos. Exploiting the power of relational databases for efficient stream processing. In *International Conference on Extending Database Technology: Advances in Database Technology*, pages 323–334, Mar 2009.

[116] C. Liu, J. Yuen, A. Torralba, J. Sivic, and W.T. Freeman. Sift flow: Dense correspondence across different scenes. In *European Conference on Computer Vision*, pages 28–42, Oct 2008.

[117] M. Liu, O. Tuzel, S. Ramalingam, and R. Chellappa. Entropy rate superpixel segmentation. In *Proc. of IEEE CVPR*, pages 2097–2104, Jun 2011.

[118] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[119] C.C. Loy, T.M. Hospedales, T. Xiang, and S. Gong. Stream-based joint exploration-exploitation active learning. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1560–1567, Jun 2012.

[120] C. Lu, J. Shi, and J. Jia. Online robust dictionary learning. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 415–422, Jun 2013.

[121] J. Lu, K. Shi, D. Min, L. Lin, and M.N. Do. Cross-based local multipoint filtering. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 430–437, Jun 2012.

[122] J. Lu, H. Yang, D. Min, and M. N. Do. Patch match filter: Efficient edge-aware filtering meets randomized search for fast correspondence field estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP(99):1854–1861, Jul 2016.

[123] J. Lu, H. Yang, D. Min, and M.N. Do. Patch match filter: Efficient edge-aware filtering meets randomized search for fast correspondence field estimation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1854–1861, Jun 2013.

[124] D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *International Joint Conferences on Artificial Intelligence*, volume 2, pages 674–679, Aug 1981.

[125] C. Rhemann M. Bleyer and C. Rother. Patchmatch stereo - stereo matching with slanted support windows. In *British Machine Vision Conference*, pages 14.1–14.11, Sep 2011.

[126] T.E. Marlin. *Process Control: Designing Processes and Control Systems for Dynamic Performance*. Chemical Engineering Series. McGraw-Hill, 1995.

[127] M. D. McKay, R. J. Beckman, and W. J. Conover. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, Apr 1979.

[128] A.H. Meghdadi and P. Irani. Interactive exploration of surveillance video through action shot summarization and trajectory visualization. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2119–2128, Dec 2013.

[129] X. Mei, X. Sun, W. Dong, H. Wang, and X. Zhang. Segment-tree based cost aggregation for stereo matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 313–320, June 2013.

[130] J. Melo, A. Naftel, A. Bernardino, and J. Santos-Victor. Detection and classification of highway lanes using vehicle motion trajectories. *IEEE Transactions on Intelligent Transportation Systems*, 7(2):188–200, Jun 2006.

[131] D. Min, J. Lu, and M.N. Do. A revisit to cost aggregation in stereo matching: How far can we reduce its computational redundancy? In *International Conference on Computer Vision*, pages 1567–1574, Nov 2011.

[132] D. Min and K. Sohn. Cost aggregation and occlusion handling with WLS in stereo matching. *IEEE Transactions on Image Processing*, 17(8):1431–1442, Aug 2008.

[133] MPI-Sintel Optical Flow Benchmark. `http://sintel.is.tue.mpg.de/results`. Accessed: 2017-01-24.

[134] J. Pal, J. J. Weinman, L. C. Tran, and D. Scharstein. On learning conditional random fields for stereo. *International Journal of Computer Vision*, 99(3):319–337, Sep 2012.

[135] S. Paris and F. Durand. A topological approach to hierarchical segmentation using mean shift. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, Jun 2007.

[136] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand. Bilateral filtering: Theory and applications. *Foundations and Trends in Computer Graphics and Vision*, 4(1):1–73, 2009.

[137] Nurjahan Parvin. Robust curved road boundary identification using hierarchical clustering. Master's thesis, University of Ontario Institute of Technology, Oshawa, Canada, 2013.

[138] A. Pinar and C. Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, Aug 2004.

[139] S. Randriamasy and A. Gagalowicz. Region based stereo matching oriented image processing. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 736–737, Jun 1991.

[140] C Rasmussen and C. Williams. *Gaussian Processes for Machine Learning.* MIT Press, Jan 2006.

[141] X. Ren and J. Malik. Learning a classification model for segmentation. In *Proc. of IEEE ICCV*, pages 10–17, Oct 2003.

[142] J. Revaud, P. Weinzaepfel, Z. Harchaoui, and C. Schmid. Deep Convolutional Matching. Technical report, INRIA, 2015.

[143] J. Revaud, P. Weinzaepfel, Z. Harchaoui, and C. Schmid. EpicFlow: Edge-Preserving Interpolation of Correspondences for Optical Flow. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1164–1172, Jun 2015.

[144] C. Richardt, D. Orr, I. Davies, A. Criminisi, and N.A. Dodgson. Real-time spatiotemporal stereo matching using the dual-cross-bilateral grid. In *European Conference on Computer Vision*, volume 6313, pages 510–523. 2010.

[145] M. S. Ryoo. Human activity prediction: Early recognition of ongoing activities from streaming videos. In *International Conference on Computer Vision*, pages 1036–1043, Nov 2011.

[146] R.K. Satzoda and M.M. Trivedi. Selective salient feature based lane analysis. In *International IEEE Conference on Intelligent Transportation Systems*, pages 1906–1911, Oct 2013.

[147] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1-3):7–42, Apr 2002.

[148] R. Schuster, R. Mörzinger, W. Haas, H. Grabner, and L. Van Gool. Real-time detection of unusual regions in image streams. In *International Conference on Multimedia*, pages 1307–1310, Oct 2010.

[149] J. Sherrah. Learning to adapt: A method for automatic tuning of algorithm parameters. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 414–425, Dec 2010.

[150] M. Sizintsev and R. P. Wildes. Coarse-to-fine stereo vision with accurate 3d boundaries. *Image and Vision Computing*, 28(3):352–366, Mar 2010.

[151] A. Soumelidis, G. Kovacs, J. Bokor, P. Gaspar, L. Palkovics, and L. Gianone. Automatic detection of the lane departure of vehicles. In *IFAC Symposium on Transportation Systems*, pages 1045–50, Jun 1997.

[152] G. Stefănescu. *Network Algebra*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[153] F. Steinbrücker, T. Pock, and D. Cremers. Large displacement optical flow computation without warping. In *International Conference on Computer Vision*, pages 1609–1614, Sep 2009.

[154] B. Stewart, I. Reading, M. Thomson, T. Binnie, K. Dickinson, and C. Wan. Adaptive lane finding in road traffic image analysis. In *IEEE International Conference*

on Road Traffic Monitoring and Control, pages 133–136, Napier Univ. Edinburgh, Apr. 1994.

[155] J. Sun. Computing nearest-neighbor fields via propagation-assisted kd-trees. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 111–118, Jun 2012.

[156] J. Sun, Y. Li, S.B. Kang, and Heung-Yeung Shum. Symmetric stereo matching for occlusion handling. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 399–406, Jun 2005.

[157] J. Sun, N.N. Zheng, and H.Y. Shum. Stereo matching using belief propagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(7):787–800, Jul 2003.

[158] D. Tabernik, L. Čehovin, M. Kristan, M. Boben, and A. Leonardis. A web-service for object detection using hierarchical models. In *International Conference on Computer Vision Systems*, pages 93–102, Jul 2013.

[159] T. Taniai, Y. Matsushita, and T. Naemura. Graph cut based continuous stereo matching using locally shared labels. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1613–1620, Jun 2014.

[160] M. Tao, J. Bai, P. Kohli, and S. Paris. Simpleflow: A non-iterative, sublinear optical flow algorithm. *Computer Graphics Forum*, 31(2pt1):345–353, May 2012.

[161] R. Timofte and L.V. Gool. Sparse flow: Sparse matching for small to large displacement optical flow. In *EEE Winter Conference on Applications of Computer Vision*, pages 1100–1106, Jan 2015.

[162] B. Tippetts, D. Lee, K. Lillywhite, and J. Archibald. Review of stereo vision algorithms and their suitability for resource-limited systems. *Journal of Real-Time Image Processing*, 11(1):5–25, Jan 2013.

[163] F. Tombari, S. Mattoccia, L. Di Stefano, and E. Addimanda. Classification and evaluation of cost aggregation methods for stereo correspondence. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, Jun 2008.

[164] Y. Tu, M. Hefeeda, Y. Xia, S. Prabhakar, and S. Liu. Control-based quality adaptation in data stream management systems. In *International Conference on Database and Expert Systems Applications*, pages 746–755, Aug 2005.

[165] Y. Tu, S. Liu, S. Prabhakar, B. Yao, and W. Schroeder. Using control theory for load shedding in data stream management. In *IEEE International Conference on Data Engineering*, pages 1491–1492, Apr 2007.

[166] S. Tulyakov, A. Ivanov, and F. Fleuret. Weakly supervised learning of deep metrics for stereo reconstruction. In *International Conference on Computer Vision*, pages 1348–1357, Oct 2017.

[167] ultrahdwallpapers. Cat Staring Wallpaper. `http://www.ultrahdwallpapers.net/animals/cat_staring-wallpaper-3840x2160`. [Online; accessed 11-July-2015].

[168] D. Wang, E. A. Rundensteiner, and T. Richard I. Ellison. Active complex event processing over event streams. *Proceedings of the Very Large Data Base Endowment*, 4(10):634–645, Jul 2011.

[169] Y. Wang, E. Khwang Teoh, and D. Shen. Lane detection and tracking using b-snake. *Image and Vision Computing*, 22(4):269–280, Apr 2004.

[170] P. Weinzaepfel, J. Revaud, Z. Harchaoui, and C. Schmid. Deepflow: Large displacement optical flow with deep matching. In *International Conference on Computer Vision*, pages 1385–1392, Dec 2013.

[171] Y. Weiss and W.T. Freeman. On the optimality of solutions of the max-product belief propagation algorithm in arbitrary graphs. *IEEE Transactions on Information Theory*, 47(2):723–735, Feb 2001.

[172] J. Winn and C. M. Bishop. Variational message passing. *Journal of Machine Learning Research*, 6:661–694, Dec 2005.

[173] J. Wulff and M.J. Black. Efficient sparse-to-dense optical flow estimation using a learned basis and layers. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 120–130, Jun 2015.

[174] L. Xu, J. Jia, and Y. Matsushita. Motion detail preserving optical flow estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(9):1744–1757, Sep 2012.

[175] C. Xuand, C. Xiong, and J.J. Corso. Streaming hierarchical video segmentation. In *European Conference on Computer Vision*, volume VI, pages 626–639, 2012.

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

[177] Q. Yang, L. Wang, R. Yang, H. Stewenius, and D. Nister. Stereo matching with color-weighted correlation, hierarchical belief propagation, and occlusion handling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(3):492–504, March 2009.

[178] S. Yenikaya, G. Yenikaya, and E. Düven. Keeping the vehicle on the road: A survey on on-road lane detection systems. *ACM Computing Surveys*, 46(1):2:1–2:43, Jul 2013.

[179] K. Yoon and I. S. Kweon. Adaptive support-weight approach for correspondence search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):650–656, Apr 2006.

[180] K. Yu, Y. Zhou, D. Li, Z. Zhang, and K. Huang. A large-scale distributed video parsing and evaluation platform. In *Chinese Conference on Intelligent Visual Surveillance*, pages 37–43, Oct 2016.

[181] S. Yu and J. Shi. Multiclass spectral clustering. In *International Conference on Computer Vision*, volume 1, pages 313–319, Nice, France, Oct 2003.

[182] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual correspondence. In *European Conference on Computer Vision*, pages 151–158, Berlin, Heidelberg, May 1994.

[183] H. Zhang, J. Yan, and Y. Kou. Efficient online surveillance video processing based on spark framework. In *International Conference on Big Data Computing and Communications*, pages 309–318, Jul 2016.

[184] K. Zhang, Y. Fang, D. Min, L. Sun, S. Yang, S. Yan, and Q. Tian. Cross-scale cost aggregation for stereo matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1590–1597, Jun 2014.

[185] K. Zhang, Y. Fang, D. Min, L. Sun, S. Yang, S. Yan, and Q. Tian. Cross-scale cost aggregation for stereo matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1590–1597, Jun 2014.

[186] Qi Zhang, Li Xu, and Jiaya Jia. 100+ times faster Weighted Median Filter (WMF). In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2830–2837, June 2014.

[187] W. Zhang, S. Zeng, Dequan Wang, and X. Xue. Weakly supervised semantic segmentation for social images. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2718–2726, June 2015.

[188] Bin Zhao, Li Fei-Fei, and E.P. Xing. Online detection of unusual events in videos via dynamic sparse coding. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 3313–3320, Jun 2011.

[189] S. Zhou, J. Xi, J. Gong, G. Xiong, and H. Chen. A novel lane detection based on geometrical model and gabor filter. In *IEEE Intelligent Vehicles Symposium*, pages 59–64, Jun 2010.

[190] Z. Zhou. *Ensemble Methods: Foundations and Algorithms*. Chapman & Hall/CRC, 1st edition, 2012.