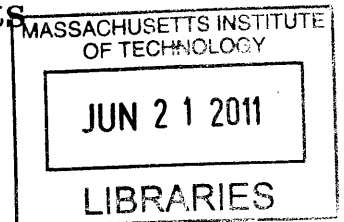


# A Real-Time Computer Vision Library for Heterogeneous Processing Environments

by

Tony J. Liu

S.B., Computer Science and Engineering & S.B., Mathematics  
Massachusetts Institute of Technology (2010)



**ARCHIVES**

Submitted to the  
Department of Electrical Engineering and Computer Science  
in Partial Fulfillment of the Requirements for the Degree of  
Master of Engineering in Electrical Engineering and Computer Science  
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author .....  
Department of Electrical Engineering and Computer Science  
May 18, 2011

Certified by .....  
Dr. Christopher J. Terman  
Senior Lecturer  
Thesis Supervisor

Accepted by .....  
Dr. Christopher J. Terman  
Chairman, Department Committee on Graduate Theses



# A Real-Time Computer Vision Library for Heterogeneous Processing Environments

by

Tony J. Liu

Submitted to the Department of Electrical Engineering and Computer Science  
on May 18, 2011, in partial fulfillment of the  
requirements for the Degree of  
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

With a variety of processing technologies available today, using a combination of different technologies often provides the best performance for a particular task. However, unifying multiple processors with different instruction sets can be a very ad hoc and difficult process.

The Open Component Portability Infrastructure (OpenCPI) provides a platform that simplifies programming heterogeneous processing applications requiring a mix of processing technologies. These include central processing units (CPU), graphics processing units (GPU), field-programmable gate arrays (FPGA), general-purpose processors (GPP), digital signal processors (DSP), and high-speed switch fabrics.

This thesis presents the design and implementation of a computer vision library in the OpenCPI framework, largely based on Open Source Computer Vision (OpenCV), a widely used library of optimized software components for real-time computer vision. The OpenCPI-OpenCV library consists of a collection of resource-constrained C language (RCC) workers, along with applications demonstrating how these workers can be combined to achieve the same functionality as various OpenCV library functions.

Compared with applications relying solely on OpenCV, analogous OpenCPI applications can be constructed from many workers, often resulting in greater parallelization if run on multi-core platforms. Future OpenCPI computer vision applications will be able to utilize these existing RCC workers, and a subset of these workers can potentially be replaced with alternative implementations, e.g. on GPUs or FPGAs.

Thesis Supervisor: Dr. Christopher J. Terman

Title: Senior Lecturer



## Acknowledgments

First and foremost, I would like to thank Professor Chris Terman for his supervision and guidance on this project. Many of his suggestions have helped me focus my efforts on the essential parts of the project and stay on track.

The project would not have been possible without the support from Jim Kulp and Chuck Ketcham at Mercury Federal Systems. From getting me started with OpenCPI development to troubleshooting issues throughout the year, Jim and Chuck have been tremendously helpful.

I would also like to thank all of the folks at Aurora Flight Sciences, especially Michael Price and Jim Paduano. Aside from generously providing me with a place to work, Michael and Jim have given me a ton of perspective on using OpenCV, showing me projects they've been working on and providing suggestions for useful applications.

Finally, I would like to thank my parents for supporting me throughout my academic career.



# Contents

<b>1</b>	<b>Introduction</b>	<b>15</b>
1.1	OpenCPI and OpenCV . . . . .	15
1.2	Thesis Structure . . . . .	16
<b>2</b>	<b>Background</b>	<b>19</b>
2.1	Heterogeneous Processing Environments . . . . .	19
2.1.1	Open Computing Language . . . . .	19
2.1.2	Compute Unified Device Architecture . . . . .	20
2.1.3	DirectCompute . . . . .	20
2.1.4	Other Technologies . . . . .	20
2.2	Computer Vision . . . . .	21
2.3	Project Goals . . . . .	22
2.3.1	Goals for Mercury Federal Systems . . . . .	23
2.3.2	Goals for Aurora Flight Sciences . . . . .	23
<b>3</b>	<b>OpenCPI</b>	<b>25</b>
3.1	Overview of OpenCPI . . . . .	25
3.1.1	Components and Workers . . . . .	26
3.1.2	Authoring Models . . . . .	27
3.1.3	Applications and Containers . . . . .	28
3.2	RCC Workers . . . . .	28
3.2.1	Execution Model . . . . .	29
3.2.2	Worker Interface . . . . .	30

3.3	Example RCC Worker . . . . .	32
3.4	Application Control Interface . . . . .	33
3.4.1	Containers and Applications . . . . .	33
3.4.2	Workers . . . . .	34
3.4.3	Ports . . . . .	34
3.4.4	Buffers . . . . .	35
3.5	Example OpenCPI Application . . . . .	36
<b>4</b>	<b>OpenCV</b>	<b>39</b>
4.1	Overview of OpenCV . . . . .	39
4.2	Module Summary . . . . .	39
4.3	Library Characteristics . . . . .	41
<b>5</b>	<b>OpenCPI-OpenCV Library</b>	<b>43</b>
5.1	Overview of Goals . . . . .	43
5.2	Design Methodology and Challenges . . . . .	44
5.2.1	Division into Workers . . . . .	44
5.2.2	Representing Images . . . . .	45
5.2.3	Properties and Parameters . . . . .	45
5.2.4	Implementation Challenges . . . . .	46
5.3	Project Code . . . . .	46
<b>6</b>	<b>Image Filtering in OpenCPI</b>	<b>47</b>
6.1	Overview of Image Filtering . . . . .	47
6.2	sobel (RCC) . . . . .	47
6.3	scharr (RCC) . . . . .	48
6.4	laplace (RCC) . . . . .	49
6.5	dilate (RCC) . . . . .	49
6.6	erode (RCC) . . . . .	49
6.7	blur (RCC) . . . . .	50
6.8	median (RCC) . . . . .	50

6.9	gaussian_blur (RCC)	50
6.10	Image Filtering Demo	51
<b>7</b>	<b>Feature Detection in OpenCPI</b>	<b>55</b>
7.1	Overview of Feature Detection	55
7.2	canny (RCC)	55
7.3	canny_partial (RCC)	56
7.4	corner_eigen_vals_vecs (RCC)	57
7.5	min_eigen_val (RCC)	57
7.6	good_features_to_track (RCC)	58
7.7	Canny Edge Detection Demo	58
7.8	Corner Tracking Demo	59
<b>8</b>	<b>Motion Tracking in OpenCPI</b>	<b>61</b>
8.1	Overview of Motion Tracking	61
8.2	optical_flow_pyr_lk (RCC)	61
8.3	sobel_32f (RCC)	63
8.4	Optical Flow Demo	63
<b>9</b>	<b>Conclusion</b>	<b>67</b>
9.1	Future Work	67
9.1.1	Additional RCC Workers	67
9.1.2	Alternative Implementations	67
9.2	Summary	68



# List of Figures

3-1	Illustration of component-based systems and applications (from the OpenCPI Technical Summary [6]) . . . . .	28
6-1	Original image of Boston . . . . .	51
6-2	Sobel $y$ -derivative . . . . .	52
6-3	Scharr $y$ -derivative . . . . .	52
6-4	Laplacian . . . . .	52
6-5	Dilate . . . . .	53
6-6	Erode . . . . .	53
6-7	Blur . . . . .	53
6-8	Median . . . . .	54
6-9	Gaussian blur with $\sigma_X = \sigma_Y = 0.8$ . . . . .	54
7-1	Canny schematic diagram with 3 workers . . . . .	59
7-2	Output of Canny edge detection . . . . .	59
7-3	Corner tracking schematic diagram with 3 workers . . . . .	60
7-4	Image with strong corners marked . . . . .	60
8-1	Optical flow schematic diagram for one level with 11 workers (8 shown)	64
8-2	Optical flow result on UAV pilot-view images (from test application)	65



# List of Tables

3.1	RCCWorker Members . . . . .	30
3.2	RCCResult Values . . . . .	32



# Chapter 1

## Introduction

### 1.1 OpenCPI and OpenCV

As technology becomes increasingly powerful, the amount of computation we can do in a real-time setting has also grown significantly. Although much of this growth has been focused around central processing units (CPU) for modern computer systems, it has also fueled the development of specialized processors, such as graphics processing units (GPU), field-programmable gate arrays (FPGA), general-purpose processors (GPP), digital signal processors (DSP), and high-speed switch fabrics.

The needs for high-performance computing have driven the advancement of many of these technologies. In the past, most of these applications could run suitably on computer systems, as huge improvements in technology and frequency scaling continually increased their performance. However, recent years have seen a higher demand for heterogeneity in computer systems. For instance, as CPU speed improvements continue to slow significantly, memory latency is expected to become the primary bottleneck in computer performance. This has created the need for structural changes or custom hardware in modern processors, if performance gains are to continue. [18]

Because of the existing technology already available, it is both impractical and often infeasible to create custom hardware solutions for every high-performance computing application. Heterogeneous computing systems must therefore utilize existing processors and provide an interface to unite different instruction set architectures.

Unfortunately, this can sometimes be a very ad hoc and difficult process. The Open Component Portability Infrastructure (OpenCPI) provides a platform that simplifies programming heterogeneous processing applications requiring a mix of processing technologies, including CPUs, GPUs, and FPGAs.

On the other hand, computer vision is a prime example of a field which can take advantage of these high-performance computing systems. The primary goal of this thesis is to present a computer vision library in the OpenCPI framework, largely based on Open Source Computer Vision (OpenCV), a widely used library of optimized software components for real-time computer vision. The OpenCPI-OpenCV library consists of a collection of resource-constrained C language (RCC) workers, along with applications demonstrating how these workers can be combined to achieve the same functionality as many OpenCV library functions.

OpenCPI applications can be constructed from many workers, often resulting in greater parallelization compared with similar OpenCV applications if run on multi-core platforms. Future OpenCPI computer vision applications can build upon these existing RCC workers, and a subset of these workers can potentially be replaced with alternative implementations, e.g. on GPUs or FPGAs.

This work was done with the generous support from Mercury Federal Systems, the creator of OpenCPI. Moreover, real-time computer vision is an essential component of unmanned aircrafts and developing high-performance computer vision applications could potentially be very useful for future projects at Aurora Flight Sciences.

## 1.2 Thesis Structure

The structure of the thesis is as follows.

Chapter 2 provides the background necessary for putting OpenCPI and OpenCV in perspective. We begin with an overview of existing heterogeneous processing environments, followed by a brief description of problems in computer vision that would benefit from acceleration on a mix of processing technologies. We conclude the chapter by outlining the goals of the project in more detail.

Chapter 3 begins with an overview of OpenCPI, namely the goals and potential applications of the platform. We delve into the technical details of OpenCPI and discuss the interfaces that are used for building OpenCPI applications. To best illustrate these concepts, we walk through the process of writing a simple OpenCPI application.

Chapter 4 gives a summary of OpenCV. We provide a bit of motivation and talk about the history of the computer vision library. We then discuss a few of the essential modules in OpenCV, taking note of certain features and characteristics we will need to consider in porting parts of OpenCV to OpenCPI.

Chapter 5 starts by discussing the goals of a computer vision library in OpenCPI based on OpenCV. We talk about the general approach taken to design this OpenCPI-OpenCV library, as well as some of the challenges faced along the way.

Chapters 6, 7, and 8 provide documentation for the OpenCPI workers that can be put together to perform many common tasks in computer vision. These cover image filtering, feature detection, and motion tracking, respectively. In the three chapters, we also put the OpenCPI workers together into a few applications. In particular, we go over the implementation of an optical flow application using many of these workers and compare this to the standalone OpenCV implementation.

Chapter 9 summarizes the work above and discusses possible avenues for future work with OpenCPI and OpenCV.



# Chapter 2

## Background

### 2.1 Heterogeneous Processing Environments

Although OpenCPI offers a novel approach to working in heterogeneous processing environments, there are quite a few existing technologies that serve a similar purpose. We outline the features and limitations of a selection below. It is notable that there seems to be a lack of support for FPGA environments, one shortcoming addressed by OpenCPI.

#### 2.1.1 Open Computing Language

Originally developed by Apple, Inc., Open Computing Language (OpenCL) [16] provides a programming environment for systems involving a mix of multi-core CPUs, GPUs, Cell-type architectures, and other parallel processors such as DSPs. OpenCL provides parallel computing using both task-based and data-based parallelism. The main feature of OpenCL is giving the ability of any application to access the GPU for non-graphical computing. Moreover, OpenCL is analogous to the industry standards: Open Graphics Library (OpenGL) and Open Audio Library (OpenAL). OpenCL is currently managed by the non-profit technology consortium Khronos Group.

Future versions of OpenCPI will use OpenCL for GPU support.

### 2.1.2 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) [13] is a parallel computing architecture developed by NVIDIA. Developers can use the C programming language, compiled through a PathScale Open64 C compiler, to run code on the GPU. Third party wrappers are also available for a variety of other languages, including Python, Fortran, Java, and MATLAB. However, unlike OpenCL, CUDA-enabled GPUs are only available from NVIDIA.

### 2.1.3 DirectCompute

Microsoft DirectCompute [10] is an application programming interface (API) that supports general-purpose computing on Microsoft Windows Vista and Windows 7. Although it was released with the DirectX 11 API, it is compatible with both DirectX 10 and DirectX 11 GPUs. DirectCompute provides functionality on both NVIDIA's CUDA architecture for GPUs and AMD's platforms, which support DirectX 11.

### 2.1.4 Other Technologies

The three technologies listed above are the main competitors for providing heterogeneous processing environments, but there are a few more worth mentioning.

- BrookGPU [15] is a compiler and runtime implementation of the Brook stream programming language for use in modern graphics hardware, e.g. for non-graphical, general purpose computations. It was developed by the Stanford University Graphics group.
- Lib Sh [4] is a metaprogramming language for GPUs, although it is no longer maintained. RapidMind, Inc. was formed to commercialize the research behind Sh.
- AMD's Stream Computing Software Development Kit (SDK) [1] includes Brook+, an AMD hardware optimized version of the Brook language. Formerly called Close to Metal and initially developed by ATI Technologies, Inc., the AMD

Stream Computing SDK now includes AMD's Core Math Library (ACML) and AMD's Performance Library (APL). The newest version of the ATI Stream SDK also has support for OpenCL.

## 2.2 Computer Vision

As a technological discipline which greatly relies on the computing power of modern processors, computer vision can be thought of as the transformation of images or video to either a decision or a new representation. These transformations are usually helpful for solving particular problems. For example, one might want to sharpen a picture (image filtering), determine whether or not a photograph contains a vehicle (feature detection), or follow the path of an aircraft in a video (motion tracking). A few other applications are surveillance, biomedical analysis, and unmanned flying vehicles. Military uses of computer vision are also abundant, including the detection of enemy vehicles and missile guidance. [2]

Computer vision is closely related to (and shares many techniques with) a number of fields, such as artificial intelligence, image processing, machine learning, optics, signal processing, and mathematics. As highly visual creatures, we are easily deceived into underestimating the difficulty of many tasks in computer vision. The process of turning pixel data, represented as grids of numbers, into useful information, can be challenging to formulate as algorithms. Other issues can include noise and camera calibration. With a constantly expanding realm of real-world applications, computer vision is rapidly growing field with a great need for high-performance computing systems.

A brief overview of the many subfields of computer vision is given below.

- Image filtering involves modifying an image based on a function of a local neighborhood of each pixel. Examples include calculating gradients and smoothing images. These are often building blocks for more complex tasks in computer vision.

- Image restoration focuses removing noise. Sources of this noise include motion blur and noise inherent in sensors. Many techniques from image filtering apply here.
- Feature detection can be more generally thought of a problem in image recognition. These tasks can include seeing if a given object is part of an image, classifying the objects in an image, or scanning the image for a particular condition. Examples include identifying fingerprints and analyzing abnormal cells or tissues in medical images.
- Image segmentation is one way to make computer vision tasks more tractable, by selecting regions of an image that are relevant for further processing. This is closely related to feature extraction.
- Motion tracking involves many tasks related to estimating the velocity at points in a sequence of frames. For instance, the tracking problem involves following the movements of a set of interest objects; optical flow seeks to determine how a set of points are moving relative to the image plane. These problems are highly applicable to robot navigation, for instance.

Most of the existing algorithms for these tasks involve heavy matrix computations and complex data structures. As such, there is a lot of room for potential acceleration on heterogeneous processing environments.

## 2.3 Project Goals

The primary goal of this project is to port a subset of the OpenCV software components to the OpenCPI framework. In completing the work described above, I hope to align the goals of my project with the goals of both Mercury Federal Systems and Aurora Flight Sciences.

### **2.3.1 Goals for Mercury Federal Systems**

As a fairly new open-source technology, OpenCPI would benefit tremendously from a larger user base. An important objective of this project is to have example implementations of OpenCV modules integrated through OpenCPI. The widespread use of OpenCV could lead to increased awareness and use of OpenCPI, especially if there is an enormous performance benefit for certain components. Because OpenCPI is targeted at embedded systems, a computer vision library for OpenCPI could expand the realm of processors used for computer vision, by simplifying the construction of computer vision systems.

Another aspect of this project involves creating sample computer vision applications using OpenCPI. These will demonstrate how various OpenCPI workers can be put together to accomplish various tasks in computer vision, including edge detection and motion tracking. These programs will also complement the documentation for the OpenCPI workers and will encourage collaboration between the OpenCV and OpenCPI communities.

Furthermore, highlighting the defense applications of this project is another goal.

### **2.3.2 Goals for Aurora Flight Sciences**

The successful implementation of various OpenCV modules on multiple types of hardware could prove very useful for Aurora Flight Sciences. Aside from the potential performance boost, using OpenCPI could allow certain tasks to be completed on technologies other than a CPU. For instance, these could include technologies requiring less power and of smaller size, as performance may not be the only consideration. Ideally, this will increase the realm of feasible projects for Aurora Flight Sciences.



# Chapter 3

## OpenCPI

### 3.1 Overview of OpenCPI

Original developed by Mercury Federal Systems, the Open Component Portability Infrastructure (OpenCPI) is an open-source software framework for building high-performance applications running on systems containing a mix of processing technologies, e.g. CPUs, GPUs, FPGAs, GPPs, DSPs, and high-speed switch fabrics. The goal of OpenCPI is to improve code portability, interoperability, and performance in FPGA and DSP-based environments by providing well-defined waveform component application programming interfaces (API) with a set of infrastructure blocks that act as a hardware abstraction layer (HAL). OpenCPI is also appropriate for the incorporation of GPU (in progress) and multi-core technologies.

The OpenCPI framework is built on the U.S. Government's Software Communications Architecture (SCA) standard. Moreover, OpenCPI extends component-based architectures into FPGAs and DSPs to decrease development costs and time to market with code portability, reuse, and ease of integration. Using an appropriate mix of industry and government specifications, all interfaces are openly published and non-proprietary.

One of the benefits of OpenCPI is that it allows users to outsource the technology transition management job to others. Using the OpenCPI interfaces, developers can protect their application development investment by cost-effectively moving their

applications to new generations of systems using the latest technologies. OpenCPI is essentially a kit of necessary pieces to create an application platform for component-based applications based on the SCA model extended to a heterogeneous mix of computing, interconnect, and I/O resources.

To overcome the challenges of code portability in FPGA environments in particular, OpenCPI provides a pre-validated set of building blocks to interface the FPGA waveform applications with high-performance switch fabrics, onboard memory, system command and control, and wideband I/O. OpenCPI's non-proprietary interfaces act as an abstraction layer to increase the portability of FPGA applications. A verification suite is also included to facilitate debugging and reduce development time. [8]

More generally, OpenCPI provides an environment that uses the concepts of component-based development (CBD) and component-based architectures (CBA). These ideas were developed as early as the 1990's as an extension of object-oriented concepts, although OpenCPI specifically targets embedded systems. We outline the essential parts of OpenCPI below, in a bottom-up fashion. A more comprehensive overview can be found in the OpenCPI Technical Summary and related documents. [6, 5]

### 3.1.1 Components and Workers

An OpenCPI *component* is a building block for various applications. Components include a functional and interface contract, configuration properties, and ports for interacting with other components. The configuration properties are runtime scalar parameters. The ports allow the components to access various input and output buffers. Implementations of these components are referred to as *workers* and may take a variety of forms, such as compiled binaries for different processors and operating systems.

### 3.1.2 Authoring Models

Because OpenCPI workers may be written in different programming languages and targeted for a variety of processors, there is usually no common API that can be used. As a result, these component implementations are written in a variety of OpenCPI *authoring models*, i.e. languages and APIs that provide a common ground for OpenCPI workers. At the same time, these authoring models aim to achieve efficiency comparable to their native languages and tool environments.

The primary specifications for an authoring model include how a worker is built (compiled, synthesized, linked, etc) for execution in an application and an XML document containing details of the worker schema (data passing interfaces, control interfaces, etc).

The OpenCPI authoring models that are currently available are summarized in the list below. These have been taken from the official OpenCPI Technical Summary. [6]

- RCC (Resource Constrained C-language) for using the C language on software systems ranging down to micro-controllers, DSPs, dedicated cores of a multi-core, etc.
- HDL (Hardware Description Language) for components executing on FPGAs.
- XM (X-Midas) for wrapping X-Midas primitives to function as OpenCPI components.
- OCL (OpenCL-based GPU targeted) for components executing on and written for graphics processors (GPGPUs). This work is in progress.
- SCA components using C++, Portable Operating System Interface for UNIX (POSIX), and Common Object Request Broker Architecture (CORBA) written to be compliant with the Department of Defense's software-defined radio (SDR) standard.

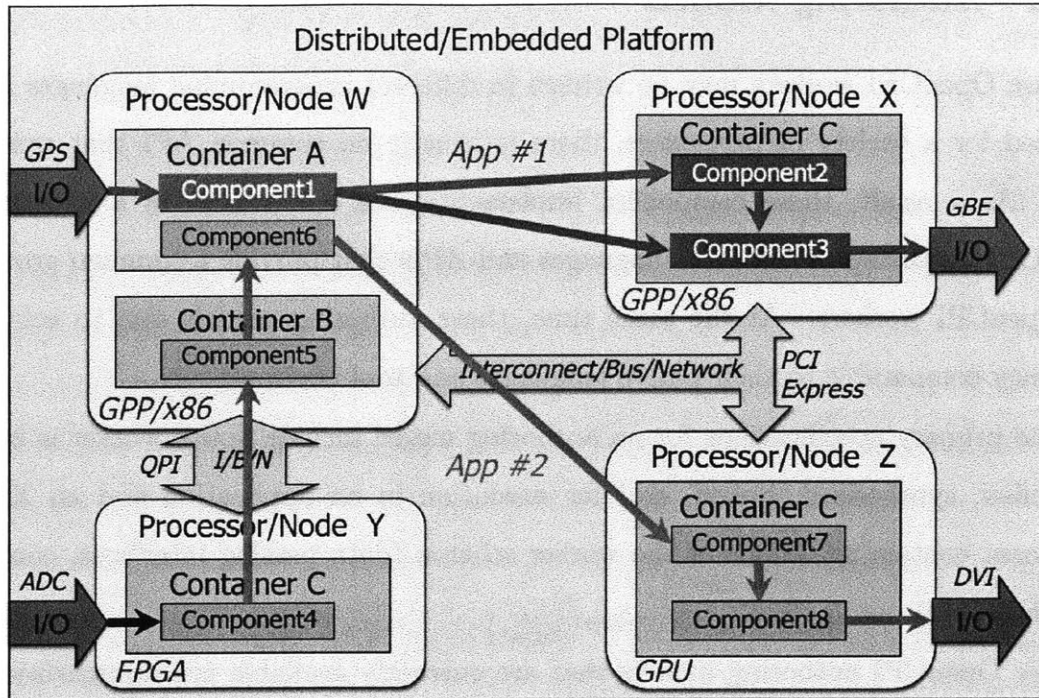


Figure 3-1: Illustration of component-based systems and applications (from the OpenCPI Technical Summary [6])

### 3.1.3 Applications and Containers

An OpenCPI *application* is a composition of components, which as a whole, perform some useful function. This process usually involves connecting the ports of the components and providing an initial configuration of properties to each component. These applications run inside an OpenCPI *container*, an execution environment such as a process on a CPU or a platform set of gates on an FPGA. Although applications can be standalone, note that we can have a master or control OpenCPI application that combines multiple applications running in many containers. The interface for launching and controlling OpenCPI applications is written in C++.

## 3.2 RCC Workers

We focus our attention on the Resource Constrained C-language (RCC) workers. OpenCPI workers written under this model are written in the C language and are

primarily targeted for resource-constrained embedded systems. These are also written to be compliant with the ISO C90 standard, though the integer types from `stdint.h` of C99 are used. Any GPP with a C compiler can serve as an environment to run RCC workers in theory; multi-core CPUs, DSPs, and microcontrollers are the natural targets.

Next, we will review the RCC model details necessary for building our library of computer vision workers. Further details can be found in the OpenCPI RCC Reference document. [7]

### 3.2.1 Execution Model

Each RCC worker resides in a container, which supplies the execution thread(s). Therefore, an RCC worker in the base profile does not need to create any threads; workers also never block. The container will call a `run` method of the worker if certain *run conditions* are true. These conditions mainly consist of a combination of input or output ports being ready and can be defined for each worker. An input port is ready if it is available to read data from, while an output port is ready if it is available to write data to.

Before the `run` method returns, the worker can specify which ports advance, if any. For input ports, this means releasing the buffer; for output ports, this means sending the buffer. The next buffer is implicitly requested.

Because the decision of which buffers to keep or advance is left to the worker, this model supports sliding window algorithms, which we will use frequently for image filtering workers. Moreover, we can support the concept of “zero copy” by attaching output ports of certain workers to input ports of others. This means that when transferring data from one port to the another, we can avoid copying the data if it resides in shared memory (and both workers reside in containers that have access to it).

The rationale for this execution model is a combination of simplicity, efficiency, and minimizing the amount of code in the worker.

### 3.2.2 Worker Interface

Each RCC worker has a required `run` method along with additional optional methods. We focus on the `run` method here. This method requests that the worker perform its normal computation. The method signature is given below.

```
RCCResult run(RCCWorker *self,  
              RCCBoolean timedout,  
              RCCBoolean *newRunCondition);
```

Here, the worker can use its private memory and the state of its ports to decide what to do. The members of `RCCWorker` allow the `run` method to access the worker properties, its ports, and a set of container functions. This is summarized in a partial table of the members of `RCCWorker`; the full table is available in the OpenCPI RCC Reference document. [7]

Table 3.1: RCCWorker Members

Member Name	Member Data Type	Written by
properties	void * const	container
ports	RCCPort []	varies by member
container	const RCCContainer	container

The properties can be set in the container. The ports allow the workers to access the buffers and the number of bytes to read or write, while the buffers themselves each contain a pointer to the data and the maximum message length in bytes.

The container functions that are available to the worker (as a dispatch table) can be very useful. These give us more fine-grained control over the buffers and are necessary for implementing sliding window algorithms, for instance. We summarize the essential container functions next, as described in the RCC Reference. [7]

- `RCCBoolean request(RCCPort *port, uint32_t max);`

Request a new buffer be made available as the current buffer on a port. An optional (non-zero) length may be supplied. If the port already has a current

buffer, the request is considered satisfied. The return value indicates whether a new current buffer is available. Non-blocking.

- `void release(RCCBuffer *buffer);`

Release a buffer for reuse. If the buffer is a current buffer for a port, it will no longer be the current buffer. Buffer ownership passes back to the container. Non- blocking. Must be done in the order obtained, per port.

- `RCCBoolean advance(RCCPort *port, uint32_t max);`

Release the current buffer and request that a new buffer be made available as the current buffer on a port. An optional (non-zero) length may be supplied. This is a convenience and efficiency combination of release-current-buffer request. The return value indicates whether a new current buffer is available. Non-blocking.

- `void send(RCCPort *port, RCCBuffer *buffer,  
          RCCOrdinal op, uint32_t length);`

Send a buffer on an output port. If the buffer is a current buffer for a port, this buffer will no longer be the port's current buffer. The `op` argument is an operation or exception ordinal. Buffer ownership passes back to the container. Non-blocking.

- `void take(RCCPort *port, RCCBuffer *releaseBuffer,  
          RCCBuffer *takenBuffer);`

Take the current buffer from a port, placing it at `*takenBuffer`. If `releaseBuffer != NULL`, first release that buffer. Non-blocking. Ownership is retained by the worker. The current buffer taken is no longer the current buffer. Used when the worker needs access to more than one buffer at a time from an input port.

The `timedout` and `*newRunCondition` parameters of the `run` method are not necessary for our purposes. These allow the worker to run if a certain amount of time has passed and change the run condition after executing, respectively.

Finally, we give an overview of the most common values for the return type `RCCResult` in the following table.

Table 3.2: `RCCResult` Values

<code>RCCResult</code> Value	Value Description
<code>RCC_OK</code>	worker operation succeeded without error
<code>RCC_DONE</code>	worker needs no more execution; a normal completion
<code>RCC_ADVANCE</code>	worker is requesting that all ports be advanced

### 3.3 Example RCC Worker

In this section, we give an example of a simple copy RCC worker. This worker simply has one input and one output port. These are defined in an XML specifications file, as shown below.

```
<ComponentSpec Name="copy">
  <DataInterfaceSpec Name="in"/>
  <DataInterfaceSpec Name="out" Producer="true"/>
</ComponentSpec>
```

This allows the skeleton code to be generated for the copy RCC worker. On calling `run`, the worker copies the data from the input buffer to the output buffer and advances both ports.

```
static RCCResult run(RCCWorker *self,
                    RCCBoolean timedOut,
                    RCCBoolean *newRunCondition) {
  ( void ) timedOut;
  ( void ) newRunCondition;
  RCCPort
    *in = &self->ports[COPY_IN],
    *out = &self->ports[COPY_OUT];
```

```

memcpy(out->current.data, in->current.data, in->input.length);
out->output.u.operation = in->input.u.operation;
out->output.length = in->input.length;
return RCC_ADVANCE;
}

```

The worker can be compiled into an object file, which an OpenCPI application can link and execute.

## 3.4 Application Control Interface

The OpenCPI Application Control Interface (ACI) consists of a C++ interface for controlling containers and applications, residing in the namespace OCPI. A typical OpenCPI application will find containers suitable for executing our given workers, then create an application within the container. Runtime instances of relevant workers are then created and their ports are connected; this can be within the container, between containers, or to the control application itself (via external ports). We give an overview below. Complete documentation can be found in the OpenCPI Application Control Interface reference. [5]

### 3.4.1 Containers and Applications

This `OCPI::Container` class represents execution environments for workers.

- `Application *Container::createApplication();`

This method returns a newly created application in the container and must be deleted after it is no longer needed.

- `Worker &Container::Application::createWorker(
 const char *artifact,
 PValue *artifactProperties,
 const *implementation,`

```
const *instance = NULL);
```

This method returns a reference to a worker. For RCC workers, the only arguments needed are the artifact and implementation, typically pointing to a shared object library.

### 3.4.2 Workers

The `OCPI::Worker` class represent worker instances. They are destroyed only when the application they reside in is destroyed.

- `Port &getPort(const char *name);`

This method gets a reference to one of the ports of the worker, with a given name.

- `void &setProperty(const char *name, const char *value);`

This method sets a worker's property by name, giving the value in string form. The available property types are `bool`, `int8_t`, `double`, `float`, `int16_t`, `int32_t`, `uint8_t`, `uint32_t`, `uint16_t`, `int64_t`, `uint64_t`, `const char *`. In OpenCPI, these are named `Bool`, `Char`, `Double`, `Float`, `Short`, `Long`, `UChar`, `ULong`, `UShort`, `LongLong`, `ULongLong`, `String`, respectively, and appear in related methods along with the XML specifications.

### 3.4.3 Ports

The `OCPI::Port` class represent the ports of the workers.

- `void Port::connect(PValue *myProperties,  
Port &otherPort,  
PValue *otherProperties);`

This method connects the ports of two workers, which could be in the same or different containers. Note that the two ports connected must have different roles and cannot both be input or output ports.

- `ExternalPort& Port::connectExternal(PValue *myProperties,  
const char *externalName,  
PValue *externalProperties);`

This method allows the control application to have ports that are connected to the ports of certain workers. The role (producer or consumer) of this port is the opposite of the worker port.

The `OCPI::ExternalPort` class allows communication between the workers and the control application and they are owned by the workers.

- `ExternalBuffer *ExternalPort::getBuffer(uint8_t &data,  
uint32_t &length,  
uint8_t &opCode,  
bool &endOfData);`
- `ExternalBuffer *ExternalPort::getBuffer(uint8_t &data,  
uint32_t &length);`

These overloaded methods are used to retrieve the next available buffer on an external port. The two versions correspond to input and output buffers, respectively. The input version also returns the metadata via references.

### 3.4.4 Buffers

The `OCPI::ExternalBuffer` class represents buffers owned by external ports, i.e. usually ones in the control applicatoin.

- `void ExternalBuffer::release();`

This method is used to discard an input buffer after it has been consumed by the control application.

- `void ExternalBuffer::put(uint8_t opCode,  
uint32_t length,  
bool endOfData);`

This method sends an output buffer and associated metadata after it has been filled by the control application.

### 3.5 Example OpenCPI Application

To put everything together, we describe the process of writing a simple “hello world” OpenCPI application. Before creating the control application, the workers are compiled and the RCC workers are linked into a shared library, `workers.so`. The OpenCPI control application creates a copy worker, an external port to send data to the worker, an external port to received data from the worker, then sends a message. The copy worker then fills its output buffer with the message and the control application will receive this from its external port.

The following C++ code is reproduced from the OpenCPI Application Control Interface document. [5]

```
namespace OCPI {  
    Container &c = ContainerManager::find('RCC')  
    Application *app = c.createApplication();  
    Worker &w = app->createworker('workers.so', NULL, 'copy');  
    Port  
        &win = w.getPort('in'),  
        &wout = w.getPort('out');  
    ExternalPort  
        &myIn = win.connectExternal('aci_out'),  
        &myOut = wout.connectExternal('aci_in');  
    w.start();  
    uint8_t opcode, *idata, *odata;  
    uint32_t ilength, olength;  
    bool end;  
    ExternalBuffer *myOutput = win.getBuffer(odata, olength);  
    assert(myOutput && olength >= strlen('hello') + 1);
```

```

strcpy(odata, 'hello');
myoutput->put(0, strlen('hello') + 1, false);
ExternalBuffer *myInput =
    wout.getBuffer(opcode, idata, ilength, end);
assert(myInput && opcode == 0 &&
        ilength == strlen('hello') + 1 && !end &&
        strcmp(idata, 'hello'));
delete app;
}

```



# Chapter 4

## OpenCV

### 4.1 Overview of OpenCV

The Open Computer Vision (OpenCV) library includes a range of real-time computer vision algorithms, for applications ranging from interactive art to mine inspection. [2] The library includes more than 500 optimized algorithms, and has garnered more than 2 million downloads and over 40 thousand people in the user group. Moreover, OpenCV is released under a BSD license; it is free for both academic and commercial use. The library is available on Linux, Windows, and MacOS, with implementations in C, C++, along with a Python wrapper.

Examples of particularly useful modules for include implementations of the Canny algorithm [3] to find edges and the Lucas-Kanade [9] algorithm to compute optical flow. These algorithms depend heavily on the matrix operations that make up the core of OpenCV. We give a brief overview of the modules in OpenCV, followed by a discussion of structure of the code. Specifically, we will focus on OpenCV 2.1.

### 4.2 Module Summary

The OpenCV 2.1 library for C/C++ is divided into a few areas:

- **core:** The Core Functionality

- `imgproc`: Image Processing
- `features2d`: Feature Detection and Descriptor Extraction
- `flann`: Clustering and Search in Multi-Dimensional Spaces
- `objdetect`: Object Detection
- `video`: Video Analysis
- `highgui`: High-level GUI and Media I/O
- `calib3d`: Camera Calibration, Pose Estimation and Stereo
- `ml`: Machine Learning

To give an idea of the algorithms available in OpenCV, we list a core selection below, broken into a few categories. These were gathered from combining the C/C++ references. [11]

- **Image Filtering:** Blur, Dilate, Erode, Laplace, MorphologyEx (transformations using erosion and dilation as building blocks), PyrDown and PyrUp (down-sampling and upsampling step of Gaussian pyramid decomposition), Smooth, Sobel, Scharr, and applying arbitrary kernels
- **Geometric Image Transformations:** LogPolar, Remap, Resize, Rotate, WarpAffine, WarpPerspective
- **Miscellaneous Image Transformations:** DistTransform (distance to closest zero pixel), FloodFill, Inpaint, Integral, Threshold, PyrMeanShiftFiltering, PyrSegmentation, WaterShed, GrabCut (last four are image segmentation algorithms)
- **Feature Detection:** Canny, CornerHarris, ExtractSURF, FindCornerSubPix (refines corners), GetStarKeypoints (the StarDetector algorithm), GoodFeaturesToTrack (determines strong corners), HoughCircles and HoughLines (finds circles and lines using a Hough transform), PreCornerDetect, SampleLine

- **Object Detection:** Haar Feature-Based Cascade Classifier, HaarDetectObjects
- **Motion Analysis and Object Tracking:** CalcGlobalOrientation, CalcMotionGradient, CalcOpticalFlow (many methods: block matching, Horn-Schunck, Lucas-Kanade with and without pyramids), CamShift, KalmanCorrect, KalmanPredict, MeanShift, SegmentMotion, SnakeImage

## 4.3 Library Characteristics

In this section, we comment on general characteristics of the OpenCV code. These observations will be essential in the design of the OpenCPI computer vision library.

- **OpenCV Data Structures:** For simpler code in many of the algorithms described earlier, OpenCV uses its own data structures. Most of these reside in the OpenCV core module, and include classes or structures for points, vectors, images, matrices, and related operators (e.g. matrix inverse, singular value decomposition, etc).
- **Use of C++ Templates:** OpenCV supports a variety of image types, i.e. different numbers color channels and different pixel depths. Many of the algorithms require that an image be converted into a specific type (e.g. grayscale with 8-bit pixel depths), while others run specific, optimized code depending on the image type. This is achieved by relying heavily on template classes.
- **Use of C++ STL:** Using built-in data structures and algorithms available in the C++ Standard Template Library also simplifies the OpenCV code tremendously. For instance, vectors are commonly used, as is sorting.
- **Use of SSE Instructions:** Because OpenCV is heavily optimized, there can be a huge performance benefit to using Streaming SIMD Extensions (SSE) instruction set. For instance, it is often possible to pack multiple pixels into

a single 128-bit register. This leads to loop unrolling as an optimization (e.g. going down an image 2 or 4 lines at a time).

- **Ad hoc Implementations:** As OpenCV is open source, this also means that there are many contributors to the library. One result is that most OpenCV library functions are treated as a black box, without much in terms of shared standards. Of course, the same data structures are used for representing images and many lower-level algorithms are commonly reused as building blocks for more complex routines.

Many of these characteristics will present challenges in constructing an OpenCV-based computer vision library in OpenCPI.

# Chapter 5

## OpenCPI-OpenCV Library

### 5.1 Overview of Goals

This OpenCPI-OpenCV library will consist of a collection of RCC workers, along with a couple OpenCPI applications combining them in interesting ways. The primary goal of a computer vision library in OpenCPI is to provide much of the same functionality as OpenCV, with the potential for acceleration using alternative hardware. The structure of the library will therefore be a bit different. First, we outline the goals we want this OpenCPI-OpenCV library to achieve.

- **Consistency:** For the OpenCV library functions we do implement, the goal is to stick to the OpenCV interfaces as closely as possible.
- **Simple Conversion:** The process of converting an existing OpenCV application to an OpenCPI application should be as straightforward as possible. This goes hand in hand with the previous goal, as keeping the same (or very similar) interfaces between the two libraries will facilitate the conversion process, especially when the programming model is very different.
- **Minimize Changes:** We want to minimize the amount of code we need to change from OpenCV. This will make it easier to adapt the OpenCPI computer vision library to new releases of OpenCV.

- **Good Granularity of Workers:** For our OpenCPI workers in the library, we want them to be fine-grained enough so that future implementations can be swapped in and out with ease. On the other hand, having each worker perform a greater amount of work reduces the complexity of OpenCPI vision applications. We need a compromise between the two.
- **Immediate Execution:** The OpenCPI workers and applications should follow an execution model that does as much as possible with the given data. For instance, a worker couple take in an image line by line, or a video frame by frame. The more we focus on achieving this, the more changes we will presumably have to make to the existing OpenCV code.

Keeping these goals in mind, we wish to strike a balance in the design.

## 5.2 Design Methodology and Challenges

To achieve the goals stated above, we first need to identify an interesting subset of functions in OpenCV to port to OpenCPI. I chose three broad areas to focus on. The first is image filtering, which includes tasks such as calculating gradients and blurring images. Second, the area of feature detection includes corner and edge detection. Third, motion tracking encompasses algorithms such as those for calculating optical flow.

The next couple tasks are to decide on a methodology for determining the granularity of a worker, decide how to represent images, find a substitute for function parameters, and discuss a strategy for implementing the RCC workers.

### 5.2.1 Division into Workers

Although many OpenCV functions are fairly simple and would naturally fit the execution model of a worker, most high-level functions would be too complex to put into a single worker. Many of these complex algorithms rely on other OpenCV functions. However, creating a worker for each OpenCV function is excessive; the ideal balance

lies somewhere in between, and is best decided on a case-by-case basis. This is further discussed in the documentation for the RCC workers.

### 5.2.2 Representing Images

Images will be represented as either grayscale or RGB with 8-bit pixel depths. The dimensions and number of color channels of an image will be put in a worker's properties section, along with additional metadata (optional). An OpenCPI vision application will need to pass this information to the workers when setting them up. The pixel data will be contiguous in memory, though the OpenCPI workers may only need a couple lines at a time. Video streams will be represented simply as a sequence of images under these specifications.

On the other hand, loading data into OpenCPI applications will rely on OpenCV functions. Because the images or videos used can come in many different digital formats (e.g. JPEG, PNG, GIF, AVI, etc), we delegate this task to the OpenCV functions where the conversion process is already implemented. These tasks may be replaced by stream-based OpenCPI workers in the future.

### 5.2.3 Properties and Parameters

In addition to the image information, we can take advantage of a worker's properties to essentially substitute for function parameters. However, this only provides a partial solution. There are many OpenCV functions that take pointers to temporary buffers, output buffers, etc, which are filled after the function call. The output can naturally be put in a worker's output buffer, but this is not always so simple. Most OpenCV image processing functions will produce a processed image of some sort, but some can produce a list of feature locations, for instance. Therefore, we will need to make slight changes to the OpenCV interface to port some functions to OpenCPI.

Nevertheless, we use one input and one output port for most workers, unless we specify otherwise.

### 5.2.4 Implementation Challenges

The implementation details are quite important here. Unfortunately, there seems to be no easy way to automate this process. The most reasonable solution was a substantial rewrite of the OpenCV code in order to port the relevant functions to OpenCPI. The primary reasons for this are given below.

- **RCC and C++ STL:** Because the OpenCPI RCC workers are written in pure C, we cannot rely on C++ STL as OpenCV does. This means essentially reimplementing some of the STL functionality.
- **Substitutions for OpenCV Core:** Similar to the issue with STL, the RCC workers in OpenCPI will not have access to the OpenCV header files. Thus, we cannot take advantage of OpenCV's internal data structures and must either re-implement these or use our own.
- **Avoiding C++ Templates:** The template classes in OpenCV are primarily used for converting between different pixel depths and color schemes. Fixing the image types (grayscale and RGB with 8-bit pixel depths) allows us not to worry about these in OpenCPI.

Though these realizations severely reduce the scope of the project, we focus on porting a few commonly used algorithms in OpenCV. The documentation of the RCC workers is covered in the next few chapters.

## 5.3 Project Code

The code for the RCC workers and demo applications is available on the web (under the GNU General Public License, version 2) at:

<https://github.com/tonyliu-mit/opencpi-opencv>

All work was done on a platform using 64-bit CentOS 5.3.

# Chapter 6

## Image Filtering in OpenCPI

### 6.1 Overview of Image Filtering

Here are brief descriptions and documentation for the basic image filtering RCC workers. For all of these workers, I chose a fixed kernel size of  $3 \times 3$ , which seems to be the most commonly used size. There are a couple other reasons for this. The RCC workers have an XML spec file which includes the minimum number of buffers it must have. This is easy to modify and it's best to not make this too much larger than necessary. Moreover, this allows us to (slightly) optimize the RCC workers by unrolling loops.

Each of these workers follows the same execution pattern. It keeps a buffer history of 3 lines (including the current line). After receiving 2 lines, it will produce a row of zeros, and thereafter produce another line for every line it receives.

We list each worker with its properties and types. This is followed by a brief description of what it does and any additional notes. These workers can take images up to 1MB. Moreover, every worker in this section has one input port and one output port.

### 6.2 sobel (RCC)

- **height** (ULong): image height in pixels (and equivalently, bytes)

- **width** (ULong): image width in pixels
- **xderiv** (Bool): calculate  $x$ -derivative if nonzero, otherwise calculate  $y$ -derivative

Either calculates the  $x$ -derivative, using the Sobel kernel:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

or the  $y$ -derivative using the Sobel kernel:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

### 6.3 scharr (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels
- **xderiv** (Bool): calculate  $x$ -derivative if nonzero, otherwise calculate  $y$ -derivative

Either calculates the  $x$ -derivative, using the Scharr kernel:

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

or the  $y$ -derivative using the Scharr kernel:

$$\begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix}$$

## 6.4 laplace (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels

Convolves the image with the following kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

## 6.5 dilate (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels

The OpenCV function has an optional **iterations** argument, but this is rarely greater than 1 and it is just as easy to loop in the OpenCPI application. The result has

$$dst(x, y) = \max_{(x', y')} src(x + x', y + y'),$$

where the maximum is taken over  $(x', y') \in \{-1, 0, 1\}^2$ .

## 6.6 erode (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels

Similar to the above, but the result has

$$dst(x, y) = \min_{(x', y')} src(x + x', y + y'),$$

where the minimum is taken over  $(x', y') \in \{-1, 0, 1\}^2$ .

## 6.7 blur (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels
- **normalize** (Bool): normalizes the result if nonzero

Convolves the image with the following kernel:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

and scales (divides by 9) if necessary.

## 6.8 median (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels

Applies a median filter, with the result having

$$dst(x, y) = \text{median } src(x + x', y + y'),$$

taken over  $(x', y') \in \{-1, 0, 1\}^2$ .

## 6.9 gaussian\_blur (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels
- **sigmaX** (Double): standard deviation for  $x$  direction

- **sigmaY** (Double): standard deviation for  $y$  direction

If either sigmaX or sigmaY are less than 0, they are instead set to the default value of 0.8. Convolves the source image with a Gaussian kernel. This is the result of applying two separable linear filters of the form

$$G_i = \alpha \cdot \exp\left(-\frac{(i-1)^2}{2\sigma^2}\right),$$

for  $i = 0, 1, 2$  and  $\alpha$  chosen such that  $\sum G_i = 1$ .

## 6.10 Image Filtering Demo

This demo application simply takes an image and worker name, then applies the appropriate filter to the image. Note that for a couple of the workers, the properties must be modified and the application must be rebuilt. The resulting image is both displayed and saved as a file. The original image I used to test the workers is shown below.



Figure 6-1: Original image of Boston

Here are the output images from each of the workers.



Figure 6-2: Sobel  $y$ -derivative



Figure 6-3: Scharr  $y$ -derivative

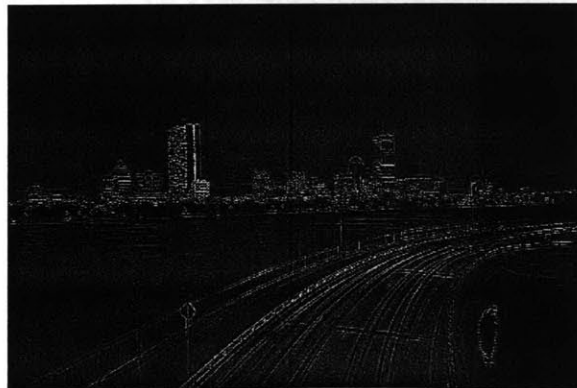


Figure 6-4: Laplacian



Figure 6-5: Dilate



Figure 6-6: Erode



Figure 6-7: Blur



Figure 6-8: Median



Figure 6-9: Gaussian blur with  $\sigma_X = \sigma_Y = 0.8$

# Chapter 7

## Feature Detection in OpenCPI

### 7.1 Overview of Feature Detection

The feature detection workers in this section can be combined to perform two primary tasks. The first is edge detection using Canny's algorithm. This can be done in two ways, using either the `canny` worker by itself, or a combination of two `sobel` workers to calculate gradients combined with a `canny_partial` worker.

The second task is to determine strong corners on an image. Although this could be implemented as a single worker, I chose to split it up into three separate steps. Even though they are sequential, this division allows us the two workers `corner_eigen_vals_vecs` and `min_eigen_val` to be reused for other purposes. Moreover, the features found by `good_features_to_track` can be used as input for more complex routines.

As in the previous chapter, we list each worker with its properties and types. This is followed by a brief description of what it does and any additional notes. These workers can take images up to 1MB. Unless otherwise specified, a worker has one input port and one output port.

### 7.2 `canny` (RCC)

- `height` (ULong): image height in pixels (and equivalently, bytes)

- **width** (ULong): image width in pixels
- **low\_thresh** (Double): value for edge linking
- **high\_thresh** (Double): value for finding initial segments

This worker takes in an image and produces an output image using the Canny algorithm. After calculating the  $x$  and  $y$  gradients, we perform a non-maxima suppression step. This is followed by tracking edges, i.e. applying a hysteresis threshold to the pixels. That is, everything above the higher threshold is accepted as an edge pixel, and everything below the lower threshold is rejected. For the pixels in-between the two thresholds, it is part of an edge if it is adjacent to a pixel that is already marked.

### 7.3 canny\_partial (RCC)

- **height** (ULong): image height in pixels (and equivalently, bytes)
- **width** (ULong): image width in pixels
- **low\_thresh** (Double): value for edge linking
- **high\_thresh** (Double): value for finding initial segments

This worker has two input ports: one for the  $x$ -derivative and one for the  $y$ -derivative. These can be fed into the worker any number of lines at a time (as long as the number of lines is the same between the  $x$  and  $y$ -derivatives). Given an input buffer of size  $N$ , the worker processes  $\lfloor N/W \rfloor$  lines, where  $W$  is the width of the image. The non-maxima suppression step can be started, although we have to wait until the entire image is available before tracking edges.

Given these gradients, the worker produces an output image using the Canny algorithm, just like the `canny` worker.

## 7.4 corner\_eigen\_vals\_vecs (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels

As a part of many corner detection algorithms, this worker is analogous to the OpenCV function that calculates eigenvalues and eigenvectors of image blocks. It takes in an image and produces an matrix of the same size, but with 3 channels and floating point numbers. More specifically, for each pixel  $p$ , let  $S(p)$  be the  $3 \times 3$  neighborhood surrounding the pixel. The 3 values

$$\left( \sum_{S(p)} (dI/dx)^2, \sum_{S(p)} (dI/dx \cdot dI/dy), \sum_{S(p)} (dI/dy)^2 \right),$$

are stored as 32-bit floating point numbers. These are arranged in row-major order in the output buffer.

## 7.5 min\_eigen\_val (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels

This worker takes the output of the worker `corner_eigen_vals_vecs` and uses the gradients to find the smaller of two eigenvalues of the matrix

$$M = \begin{bmatrix} \sum_{S(p)} (dI/dx)^2 & \sum_{S(p)} (dI/dx \cdot dI/dy) \\ \sum_{S(p)} (dI/dx \cdot dI/dy) & \sum_{S(p)} (dI/dy)^2 \end{bmatrix}.$$

The result is a buffer with  $\text{height} \times \text{width}$  32-bit floating point numbers, representing the minimal eigenvalue of the derivative covariation matrix for every pixel.

## 7.6 `good_features_to_track` (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels
- **max\_corners** (ULong): maximum number of corners to find
- **quality\_level** (Double): multiplier for the minimum eigenvalue; specifies the minimal accepted quality of image corners
- **min\_distance** (Double): limit specifying the minimum possible distance between the returned corners; Euclidean distance is used

The worker finds the strong corners with big eigenvalues in the image. The worker takes in the minimal eigenvalue for every source image pixel. Then it performs non-maxima suppression (only the local maxima in neighborhood are retained). The next step rejects the corners with the minimal eigenvalue less than `quality_level` times the maximum eigenvalue over the entire image. Finally, the worker ensures that the distance between any two corners is not smaller than `min_distance`. The weaker corners (with a smaller min eigenvalue) that are too close to the stronger corners are rejected.

This differs from the OpenCV function in a few ways. First, we use the minimum eigenvalue operator always (omitting the parameters for the Harris operator). There is no need to provide temporary buffers, as the worker can create and free those. We fix the block size to be 3, as this is the default. Finally, the number of corners can be gathered from the size of the output buffer.

In the end, this worker produces a buffer with at most `max_corners` pairs of 32-bit floating point numbers  $(x, y)$ . These are the locations of the corners on the image.

## 7.7 Canny Edge Detection Demo

This demo application strings together two `sobel` workers and the `canny_partial` worker as shown in the schematic diagram below. These all reside in the same con-

tainer.

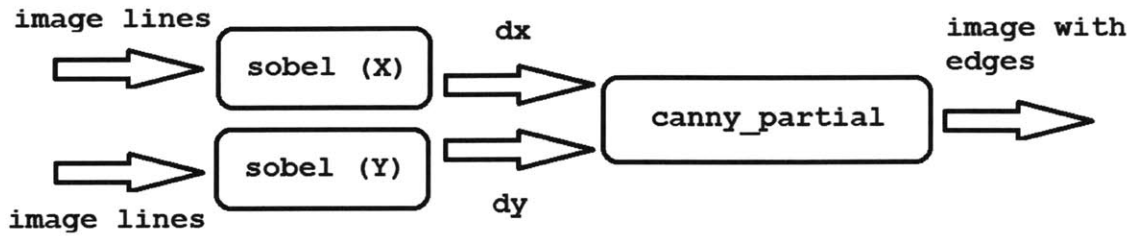


Figure 7-1: Canny schematic diagram with 3 workers

The image is fed to both `sobel` workers line by line. At every step, calling `dispatch` on the container causes the `sobel` workers to produce the gradients and the `canny_partial` worker to start the non-maxima suppression step. Once it has received the entire image, it tracks edges and produces an output image with the edges marked. The default thresholds of 10 (low) and 100 (high) are used. The result is shown below, using the same sample input image as earlier.



Figure 7-2: Output of Canny edge detection

## 7.8 Corner Tracking Demo

Next, we present a demo application that finds strong corners on an image. This application uses `good_features_to_track` and the two other workers `corner_eigen_vals_vecs`

and `min_eigen_val` to produce the minimum eigenvalues. The connections are shown in the schematic diagram.

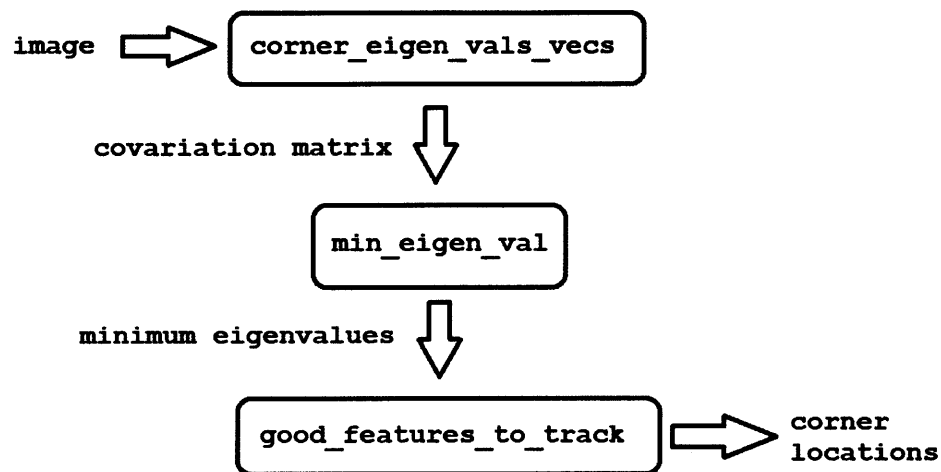


Figure 7-3: Corner tracking schematic diagram with 3 workers

We use the values of 50 for `max_corners`, 0.03 for `quality_level`, and 5 for `min_distance`. The resulting image (with features marked as red circles) is shown below.



Figure 7-4: Image with strong corners marked

# Chapter 8

## Motion Tracking in OpenCPI

### 8.1 Overview of Motion Tracking

The primary motion tracking worker in this section can be used to perform optical flow, computed using the Lucas-Kanade algorithm with pyramids. In order to minimize the changes to the OpenCV code, we introduce a new worker, `sobel_32f`, for producing gradients that are 32-bit floating point numbers. As this is considerably more complex than our previous workers, we use the optical flow application to demonstrate the features of OpenCPI.

As in the previous chapters, we list each worker with its properties and types. This is followed by a brief description of what it does and any additional notes. These workers can take images up to 1MB.

### 8.2 `optical_flow_pyr_lk` (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels
- **win\_height** (ULong): height of search window at this pyramid level
- **win\_width** (Ulong): width of search window at this pyramid level

- **level (ULong)**: level of pyramid, zero-based (single level is 0)
- **term\_max\_count (ULong)**: termination iterations for the iterative search algorithm
- **term\_epsilon (Double)**: termination epsilon (if search window moves less than epsilon) for the iterative search algorithm
- **deriv\_lambda (Double)**: relative weight of the spatial image derivatives impact to the optical flow algorithm (0 for only image intensity, 1 for only derivatives, and anything inbetween corresponds to a weighted average)

This worker has 10 input ports and 3 output ports and encapsulates one level of the Lucas-Kanade optical flow algorithm using pyramids. It essentially takes in two images and feature locations in the first image, and produces new locations for the features in the second image.

More specifically, the 10 input ports are divided as follows.

- The two images  $A$  and  $B$ : as 8-bit single-channel images.
- First-order gradients for both images: these four  $(dx_A, dy_A, dx_B, dy_B)$  are instead stored as 32-bit floating point single-channel images.
- Second-order gradients for the first image: these three  $(dxdy_A, d^2x_A, d^2y_A)$  are also stored as 32-bit floats.
- Feature locations on the first image: these are stored as pairs  $(x, y)$  of 32-bit floats.

The 3 output ports as follows.

- Feature locations: stored as pairs  $(x, y)$  of 32-bit floating point numbers.
- Statuses: boolean values indicating whether or not the corresponding features have been found.

- **Errors:** differences between patches around the original and moved points, stored as 32-bit floating point numbers.

Compared to the OpenCV implementation, we keep all of the parameters, except an optional flags parameter. This allows us to use self-chosen initial estimations of the features, but the default is just initializing these to the previous estimations (and works well enough in practice).

### 8.3 `sobel_32f` (RCC)

- **height** (ULong): image height in pixels
- **width** (ULong): image width in pixels
- **xderiv** (Bool): calculate  $x$ -derivative if nonzero, otherwise calculate  $y$ -derivative

This works identically to the `sobel` worker, although there are two output ports: one is the usual 8-bit pixel depth gradient and the other output produced is a 32-bit floating point single-channel image. This is because the output of this worker might need to feed into another `sobel_32f` worker, which accepts 8-bit pixel depth images. The worker can also take multiple lines at a time; after receiving  $N$  bytes in the buffer, it will process  $\lfloor N/W \rfloor$  lines, where  $W$  is the width of the image.

Although this worker computes gradients using the Sobel operator, we put it in the motion tracking section because of the floating point output. This can be useful for image filtering, but we rarely need the full gradients (instead of rounding to  $[0, 255]$ ) offered by floats in those scenarios. For motion tracking and optical flow in particular, this is usually necessary for more accurate results.

### 8.4 Optical Flow Demo

Using the optical flow worker, we put together the necessary connections to demonstrate one level of Lucas-Kanade optical flow using pyramids.

The parameters are set as 10 for `win_height` and `win_width`, 0 for `level`, 30 for `term_max_count`, 0.01 for `term_epsilon`, and 0.5 for `deriv_lambda`. We need a total of 11 workers, the 3 for feature detection (as described in the previous chapter), along with 8 others as shown below.

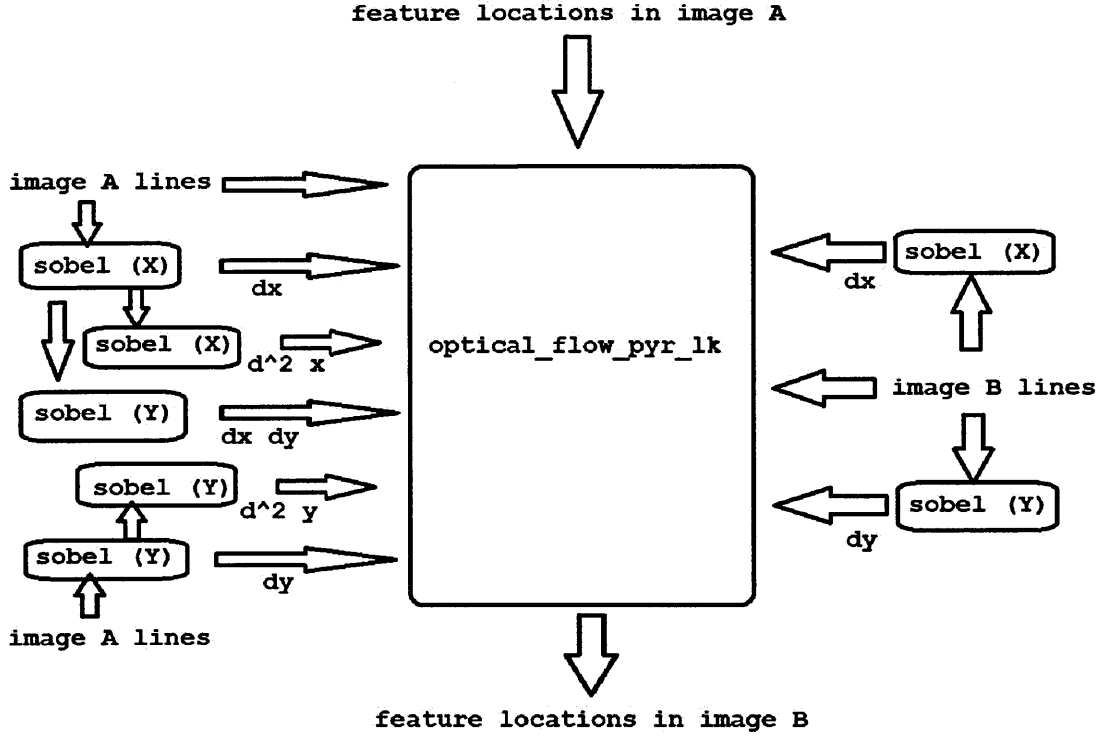


Figure 8-1: Optical flow schematic diagram for one level with 11 workers (8 shown)

We can give the application a video stream or simply two images. After getting the updated feature locations, we check the statuses and errors to see if the flow is valid. If so, we can draw an arrow between the two point locations (from old to new) in the original image.

Although the workers for this demo have been written and tested, the OpenCPI control application is still in progress. We discuss a few issues that made the process of constructing this demo application rather difficult.

First, creating our `sobel_32f` worker to have two output ports led to an unexpected result. In a few instances of these workers, the 8-bit output was unnecessary (e.g. for the second worker in a sequence for calculating second-order gradients).

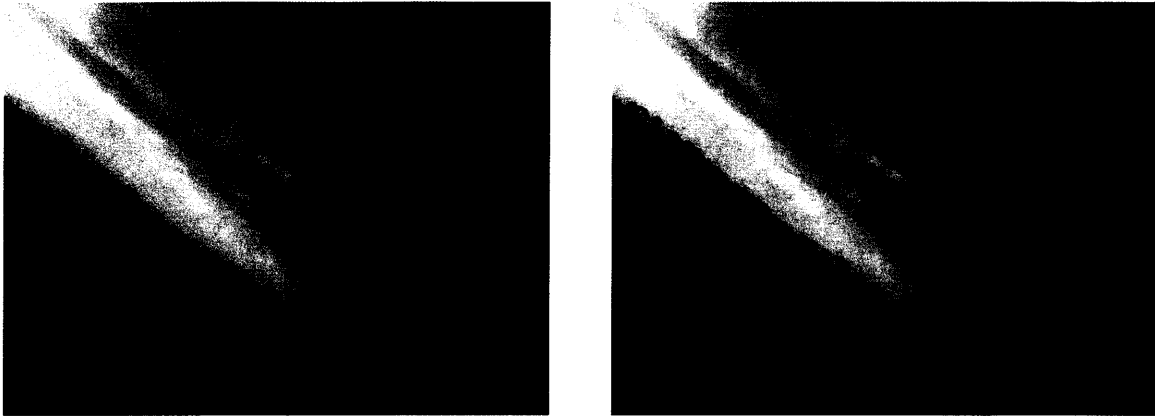


Figure 8-2: Optical flow result on UAV pilot-view images (from test application)

However, without connecting these ports to anything, the OpenCPI control application could not run properly. This was easily fixed by connecting these ports to dummy external ports.

On the other hand, it was attempted to connect one output port to multiple input ports. As this failed, more instances of the `sobel_32f` workers and the feature detection workers had to be created. For instance, the feature locations in the first image had to go to both the optical flow worker and the control application, in order for the resulting flow to be drawn in the image. This required two sets of the feature detection workers.

After these were addressed, there seemed to be an issue with the workers not dispatching or running properly (which unfortunately remains unresolved). Although the `optical_flow_pyr_lk` worker is run only after all its input ports are ready, the addition of this worker to the application prevented the other workers (e.g. Sobel operators and feature detection) from running. The most likely cause is memory-related, as decreasing the buffer sizes and number of workers in the application had an effect on the outcome. It is hoped that this work will be continued and completed.



# Chapter 9

## Conclusion

### 9.1 Future Work

Continuation of work on this project can take one of two directions.

#### 9.1.1 Additional RCC Workers

Writing more RCC workers based on OpenCV functions will expand the OpenCPI-OpenCV library. Modules of particular interest include other feature detection algorithms, such as the Harris corner detector, finding circles and lines using the Hough transform, and calculating feature maps. Other implementations of optical flow, including Gunnar Farneback's algorithm or dense varieties, would be useful as well. We could also create workers for OpenCV functions in the geometric and miscellaneous image transformation categories.

Although it is unlikely to offer a performance boost, having an expanded set of workers will increase the realm of interesting OpenCPI computer vision applications.

#### 9.1.2 Alternative Implementations

On the other hand, replacing existing workers with alternative implementations on an FPGA or GPU, is another interesting direction to take. As writing these workers becomes less ad hoc in OpenCPI, replacing highly-parallelizable RCC workers with

alternative implementations could potentially lead to a huge performance boost. For instance, all of the image filtering workers could probably be implemented to run much faster on an FPGA or GPU. OpenCV has already taken steps in this direction, as the latest version 2.2 [12] has experimental GPU support using CUDA. Implementations of optical flow on FPGAs [17] has also been explored.

## 9.2 Summary

The goal of this project was to port a subset of modules from OpenCV to OpenCPI. The porting process turned out more difficult than expected, as much of it essentially amounted to rewriting STL and template-ridden C++ code into C. Nevertheless, the collection of RCC workers and OpenCPI computer vision applications implemented for this project will serve as a solid base for future work using OpenCPI and OpenCV. Moreover, it can serve as a set of rough guidelines for the division of OpenCV functions into many composite workers.

As the FPGA and GPU portions of OpenCPI become more mature, this set of workers will be a foundation for experimenting with alternative implementations. This will hopefully facilitate hardware acceleration for interesting computer vision applications using OpenCPI.

# Bibliography

- [1] AMD. *ATI Stream Technology: GPU and CPU Technology for Accelerated Computing*. <http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx>.
- [2] G. Bradski. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.
- [3] J. Canny. *A Computational Approach To Edge Detection*. IEEE Trans. Pattern Analysis and Machine Intelligence, 8(6):679-698, 1986.
- [4] Intel Corporation. *Sh: a high-level metaprogramming language for modern GPUs*. <http://www.libsh.org/>.
- [5] J. Kulp. *OpenCPI Application Control Interface*. Mercury Federal Systems, Inc., 2010. <http://www.opencpi.org/documentation.php>.
- [6] J. Kulp. *OpenCPI Technical Summary*. Mercury Federal Systems, Inc., 2010. <http://www.opencpi.org/documentation.php>.
- [7] J. Kulp and J. Miller. *OpenCPI RCC Reference*. Mercury Federal Systems, Inc., 2010. <http://www.opencpi.org/documentation.php>.
- [8] J. Kulp and S. Siegel. *OpenCPI HDL Reference*. Mercury Federal Systems, Inc., 2010. <http://www.opencpi.org/documentation.php>.
- [9] B.D. Lucas and T. Kanade. *An iterative image registration technique with an application to stereo vision*. Proceedings of Imaging understanding workshop, 1981: 121-130.
- [10] Microsoft. *DirectCompute Lecture Series Resources*. <http://archive.msdn.microsoft.com/DirectComputeLecture>.
- [11] OpenCV. *OpenCV 2.1 C/C++ References*. <http://www.opencv.willowgarage.com/documentation/cpp/index.html> and <http://www.opencv.willowgarage.com/documentation/c/index.html>.
- [12] OpenCV. *OpenCVWiki: OpenCV\_GPU*. [http://www.opencv.willowgarage.com/wiki/OpenCV\\_GPU](http://www.opencv.willowgarage.com/wiki/OpenCV_GPU).

- [13] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.
- [14] J. Shi and C. Tomasi. *Good Features to Track*. 9th IEEE Conference on Computer Vision and Pattern Recognition. Springer, 1994.
- [15] Stanford University Graphics Lab. *BrookGPU*.  
<http://graphics.stanford.edu/projects/brookgpu/>.
- [16] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki. *The OpenCL Programming Book*. Fixstars Corporation, 2010.
- [17] Z. Wei, D. Lee, and B.E. Nelson. *FPGA-based Real-time Optical Flow Algorithm Design and Implementation*. Journal of Multimedia, Vol. 2, No. 5, 2007.
- [18] W.A. Wulf and S.A. McKee. *Hitting the memory wall: implications of the obvious*. ACM SIGARCH Computer Architecture News, Vol. 23, Issue 1, 1995.