UNIVERSITY OF CALIFORNIA,
IRVINE

# Towards Engineering Computer Vision Systems: From the Web to FPGAs

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Sajjad Taheri

2019

# DEDICATION

To my mother, Nahid.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

I am grateful to my advisors, professor Alex Nicolau and professor Alex Veidenbaum, for their patience, support, and guidance.

I would like to thank Professor Nikil Dutt for his participation in the thesis committee and providing helpful feedback.

I feel fortunate to receive mentorship from Dr. Moh Haghighat, senior principal engineer at Intel. He was always a source of inspiration and motivation for me.

I would like to shout out to my colleagues and partners in crime, Hoda Malekpour, Zhi Chen, Maral Amir, Majid Shoushtari, Bryan Donyanavard, Hamid Nejatollahi, Payman Behnam, Jin Heo, Siavash Rezaei, Laleh Beni, Tiago Muck, Aniket Shivam, Gongjin Sun, and Junjie Shen for their friendship and for sharing the pain.

I would like to thank Ningxin Hu and our Google Summer of Code interns for putting a tremendous effort during the release process of OpenCV.js.

This research was financially supported by UC Irvine graduate division and Intel corporation.

# CURRICULUM VITAE

## Sajjad Taheri

### EDUCATION

**Doctor of Philosophy in Computer Science** **2019**
University of California, Irvine *Irvine, CA*

**Bachelor of Science in Computer Engineering** **2010**
University of Tehran *Tehran, Iran*

### RESEARCH EXPERIENCE

**Graduate Student Researcher** **2014–2019**
University of California, Irvine *Irvine, CA*

### TEACHING EXPERIENCE

**Teaching Assistant** **2015–2019**
University of California, Irvine *Irvine, CA*

### INDUSTRY EXPERIENCE

**Mentor** **June 2017–Sept. 2017**
Google Summer of Code *Irvine, CA*

**Software Engineering Intern** **June 2015–Sept. 2015**
Mozilla *Mountain View, CA*

## REFEREED JOURNAL PUBLICATIONS

**Computer Vision for the Masses** **2018**
Intel Parallel Universe (April 2018 Issue)

## REFEREED CONFERENCE PUBLICATIONS

**AFFIX: Automatic Acceleration Framework for FPGA Implementation of OpenVX Vision Algorithms** **2019**
ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)

**Acceleration framework for fpga implementation of OpenVX graph pipelines** **2018**
IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)

**OpenCV.js: Computer vision processing for the open web platform** **2018**
ACM International Conference on Multimedia Systems (MMSys)

**WebRTCbench: A benchmark for performance assessment of webRTC implementations** **2015**
IEEE Embedded Systems for Real-Time Multimedia (ESTIMedia)

## SOFTWARE

**OpenCV.js** `https://github.com/ucisysarch/opencvjs`
*A computer vision library for the web*

**AFFIX** `https://github.com/sajjadt/affix`
*A framework for FPGA acceleration of computer vision algorithms*

# ABSTRACT OF THE DISSERTATION

Towards Engineering Computer Vision Systems: From the Web to FPGAs

By

Sajjad Taheri

Doctor of Philosophy in Computer Science

University of California, Irvine, 2019

Professor Alexandru Nicolau, Chair

Computer vision is an interdisciplinary field to obtain high-level understanding from digital images. It has many applications that impact our daily lives, such as automation, entertainment, healthcare, etc. However, computer vision is very challenging. This is in part due to the intrinsically difficult nature of the problem and partly due to the complexity and size of visual data that need to be processed. To be able to deploy computer vision in many practical use cases, sophisticated algorithms and efficient implementations are required.

In this thesis, we consider two platforms that are suitable for computer vision processing, yet they were not easily accessible to algorithm designers and developers: the Web and FPGA-based accelerators. Through the development of open-source software components, we highlight challenges associated with vision development on each platform and demonstrate opportunities to mitigate them.

The Web is the world's most ubiquitous computing platform which hosts a plethora of visual content. Due to historical reasons such as insufficient compute performance and lack of API support for acquiring and manipulating images, computer vision is not mainstream on the Web. We show that in light of recent web developments such as vastly improved JavaScript performance and the addition of APIs such as WebRTC, efficient computer vision processing can be realized on web clients. Through novel engineering techniques, we translate a popular open-source computer vision library (OpenCV) from C++ to JavaScript and optimize its performance for the web environment. We demonstrate that hundreds of computer vision functions run in browsers with performance close to their original C++ version. We believe this will result in an immersive and perceptual web with transformational effects, including in online shopping, education, and entertainment, among others.

Field Programmable Gate Arrays (FPGA)s are a promising solution to mitigate the computational cost of vision algorithms through hardware pipelining and parallelism. However, an efficient FPGA implementation of computer vision algorithms requires hardware design expertise and a considerable amount of engineering person-hours. We show that graph-based specifications, such as OpenVX can significantly improve FPGA design productivity. Since such abstraction lacks implementation details, a vision algorithm designer can only focus on the algorithm itself and rely on another party with hardware knowledge to implement the design efficiently on a specific platform. During this process, different implementation configurations that satisfy various design constraints, such as performance and power consumption, can be explored. Furthermore, the graph-based model permits system-level optimizations that are not possible with traditional function-level acceleration. Towards this goal, we develop a framework that optimizes and implements vision algorithms that are described in OpenVX spec on different FPGA architectures. This framework hides low-level hardware optimization and implementation details from computer vision algorithm designers and enables them to quickly develop and verify FPGA implementations of vision algorithms without sacrificing performance.

# Chapter 1

# Introduction

Computer or machine vision is a field of computer science that brings solutions for computers to extract high-level understanding from images. Scientists in this field are trying to improve the quality and quantity of understanding, automate this process, and develop high-performance, and energy-efficient implementations. Computer vision has an ever-increasing number of applications in our daily life. It has wide usage in object detection [83, 59], face recognition [100, 73], autonomous vehicles [32, 45], and health care [33]. Emerging technologies such as Virtual and Augmented Reality (VR and AR) involve numerous computer vision algorithms for tracking, scene understanding, and scene construction [34, 13, 72, 65].

Over the past few decades, several technical advances have contributed significantly to the field of computer vision:

1. Availability of **CMOS sensors** with high resolution to capture high-quality images. For instance, as Figure 1.1 shows, the quality of sensors has increased over several years. High-quality digital cameras capture images details measure the difference between colors with very high accuracy.

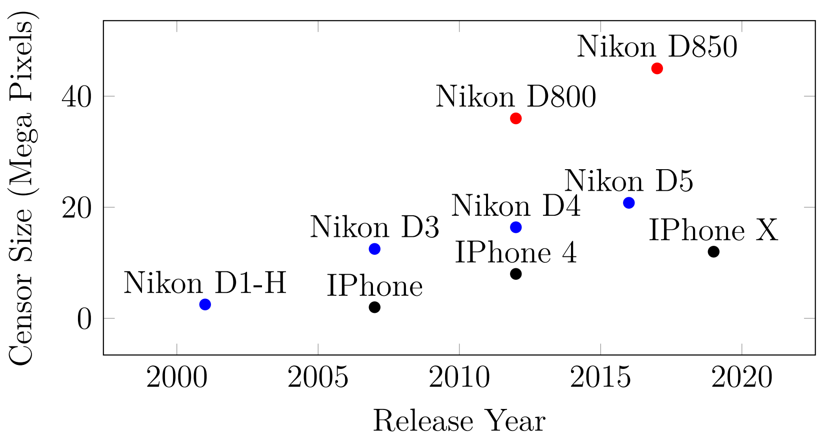


Figure 1.1: Camera sensor size trend

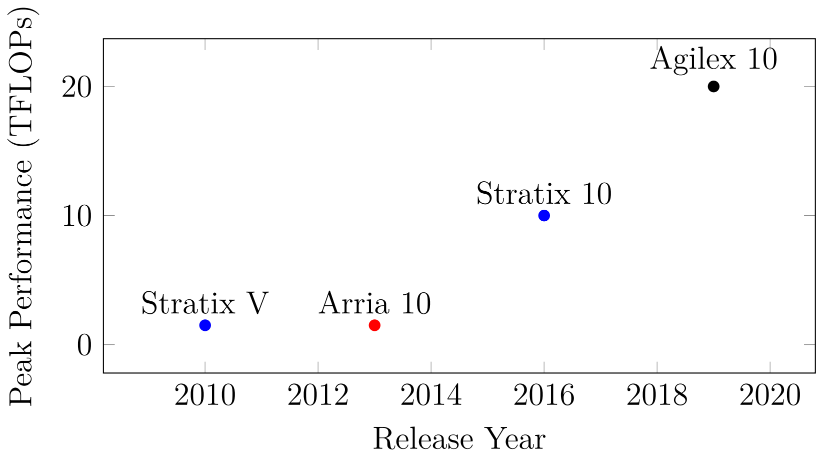


Figure 1.2: Floating-point operation performance of modern FPGAs

2. **Availability of faster hardware with parallel processing capabilities**: Computer vision algorithms exhibit inherent parallelism such that the same set of operations applies to each pixel. Parallel hardware has always been promising solutions for vision algorithms. SIMD processors, GPUs, FPGAs, and specialized ASIC designs can improve performance and power consumption orders of magnitude. Applying state-of-the-art algorithms on large data requires powerful hardware. Figure 1.2 shows the floating-point performance improvement of several high-end FPGAs released by Intel during recent years.

3. **Software stack and open-source libraries** that facilitates building computer vi-

sion systems without reinventing the infrastructure has led to a dramatic increase in software development productivity [16, 49, 101].

4. **Advances in deep learning**: Neural networks can learn the relationships between inputs and outputs that are often nonlinear. After AlexNet breakthrough performance in 2012 [55], Convolutional Neural Networks (CNNs) got huge attention and wide usage. They significantly improve decision processes in different areas such as object detection and classification and are now considered state-of-the-art techniques.

5. **Availability of large image databases collections**: One of the main reasons that contribute to the success of neural networks is the availability of a large amount of labeled training data. Datasets are often categorized toward different various vision tasks such as image classification [28], scene segmentation [23], face detection [47], and activity recognition [87].

## 1.1 Challenges of Portable and Efficient Computer Vision

Computer vision is computationally intensive and involves sophisticated algorithms. The efficient implementation of these algorithms is very challenging and needs expertise not only in computer vision but also software design and hardware optimization. Hence, open-source software such as OpenCV are often used to build real-world systems. OpenCV is designed for computational efficiency and with real-time use cases in mind [16]. OpenCV is very comprehensive, and as shown in Figure 1.3 is implemented as a set of modules. It offers a large number of primitive vision kernels and applications ranging from image processing, object detection, tracking, and deep neural networks (DNNs). Furthermore, it supports a wide variety of operating systems and is optimized for a range of parallel hardware such as

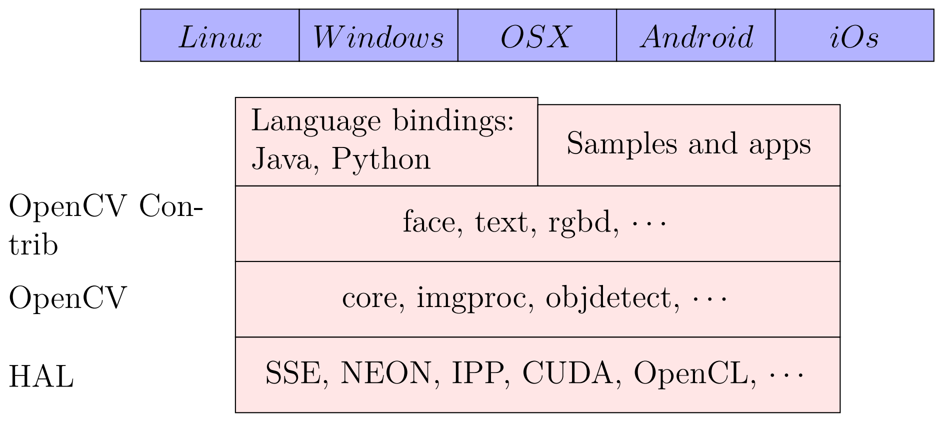

Linux
Windows
OSX
Android
iOs
Language bindings: Java, Python
Samples and apps
OpenCV Contrib
face, text, rgbd, ···
OpenCV
core, imgproc, objdetect, ···
HAL
SSE, NEON, IPP, CUDA, OpenCL, ···

Figure 1.3: OpenCV source code structure and platform support

multicores, vector extensions, and GPUs. It is developed in C++, but bindings for languages such as Python and Java are available.

A critical requirement of real-time computer vision is efficiency. Researchers have developed software, hardware, and software-hardware optimization methods to improve resource consumption, performance, and energy efficiency of computer vision systems. Implementing systems that are portable in regards to performance and power efficiency is not trivial. Since many of the optimizations depend on the underlying hardware, in practice, software implementations such as OpenCV have platform-specific optimizations that are hard-coded and enabled at compile-time. However, manual optimization is very tedious. Utilizing hardware parallelism efficiently without sacrificing portability is the subject of extensive research. Many works have used portable and high-level abstraction of programs and compilers that convert them into optimized binary for the underlying hardware [15, 30, 80].

## 1.2 Thesis Contributions

This dissertation's contributions include two disjoint efforts. We consider two platforms that are suitable for computer vision processing, yet they were not easily accessible to algorithm designers and developers: (1) the Web and (2) FPGA-based accelerators.

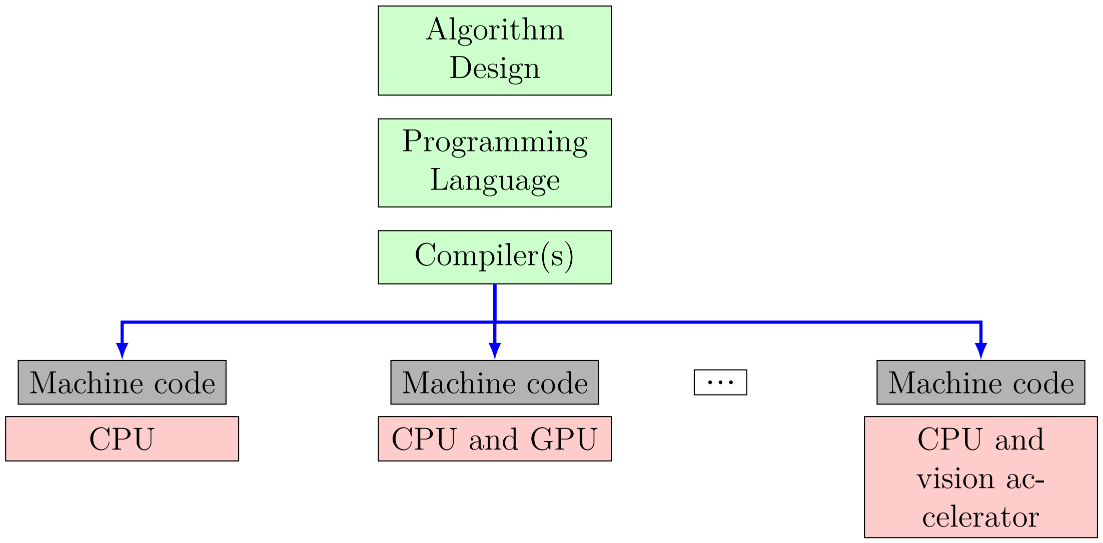


Figure 1.4: Using compilers to produce high-performance machine codes for diverse hardware platform targets

1. The Web is the most ubiquitous computing platform with billions of connected devices. Its popularity in online commerce, entertainment, science, and education has been increasing tremendously. There is also an ever-growing amount of multimedia content on the Web. Despite such rapid progress in quantity and quality of content, computer vision processing on the web browsers has not been a common practice. One approach taken by developers was to offload vision processing tasks to the server. This approach, however, sacrifices user privacy and suffers from always-online requirements and the increase in data transfer bandwidth and latency. With a massive boost in JavaScript execution performance, and HTML5 APIs to access media devices, the Web can unleash the huge potential in education, entertainment, and commerce. However, a web-based computer vision library with quality and quantity of OpenCV to work as infrastructure to build computer vision systems is lacking.

2. Computer vision acceleration improves performance and energy-efficiency dramatically and is in high demand. For example, some applications require detecting hundreds of objects in a few milliseconds. This task is not feasible on modern CPUs even with expert implementation. FPGAs are a promising solution to improve the speed and

mitigate the computational cost of vision algorithms through hardware pipelining and parallelism while offering excellent power efficiency [63, 114, 46, 67]. However, an efficient FPGA implementation of a vision algorithm requires hardware design expertise and a considerable amount of engineering person-hours. Unfortunately, developers may not be familiar with FPGA design, let alone efficient implementation. Even a CPU implementation of vision algorithms using programming languages such as C/C++ is time-consuming and error-prone. Moreover, despite advances in High-Level Synthesis (HLS) tools, they are still not able to generate efficient hardware implementation for vision algorithms developed in general-purpose programming languages. The main reason is that HLS tools do not incorporate specific optimization methods of vision algorithms, which are critical to the efficient implementation of such algorithms.

To address the above-mentioned challenges, this dissertation makes the following list of contributions:

**Web-based platform:**

1. It provides a comprehensive computer vision library for the Web by porting an existing computer vision library that is developed in a native language (i.e., OpenCV) to JavaScript. It provides an expansive set of functions with an optimized implementation that offers a near-native level of performance.
2. It shows how the library performance can be further improved through JavaScript SIMD and thread-based parallelism.

**FPGA-based platform:**

1. It develops a scalable OpenCL library of image processing and computer vision kernels such that they can be configured using various input parameters. This library can be

used in any OpenCL program or as part of a high-level vision framework.

2. It provides a heterogeneous framework called AFFIX to transform OpenVX graph-based algorithms to CPU-FPGA implementation. It employs several algorithm and hardware-specific optimization techniques. This enables vision developers to develop, verify, and test FPGA implementation of a vision algorithm quickly and without the need to know a DSL or FPGAs.

The research presented in this dissertation has been published in peer-reviewed conference proceedings [95, 97, 96, 94].

### 1.2.1 Regarding Software Quality

As part of this dissertation, we developed software libraries and tools for computer vision processing with the focus on maximizing efficiency and portability. We have taken into account the ISO 9126 software quality standards in the development process.[1]

1. **Functionality**: Required functions are correctly implemented.
2. **Usability**: Provided software is easy to use and learn.
3. **Efficiency**: System resources (e.g., memory and CPU) are used efficiently when providing the required functionality.
4. **Portability**: The software can adapt to changes in its environment well.

[1] *https://en.wikipedia.org/wiki/Software_quality#Measurement*

# Chapter 2

# OpenCV.js: Computer Vision Processing for the Open Web platform

## 2.1 Introduction

The Web is the most ubiquitous compute platform with billions of connected devices. Its popularity in online commerce, entertainment, science, and education has been increasing tremendously. There is also an ever-growing amount of visual content on the Web. Despite such rapid progress in quantity and quality of content, computer vision processing on the web browsers has not been a common practice. We believe that the lack of client-side vision processing is due to several limitations:

1. Lack of standard Web APIs to access and transfer multimedia content

2. Inferior JavaScript performance (the standard language of the Web). Computer vision

often complex algorithms that involve a sheer amount of computation, making them very time-consuming.

3. lack of open-source and accessible computer vision libraries to facilitate software development for the Web.

Newly introduced web standards address the limitations mentioned above and can empower the Web with computer vision capabilities:

1. **Addition of camera support and plugin-free multimedia delivery on the Web**: HTML5 introduced several new web APIs to capture, transfer, and present multimedia content in browsers without the need for third-party plugins. Among them, Web Real-Time Communication (WebRTC) allows capturing and peer-to-peer transportation of multimedia content [52, 95], and video element API can be used to display videos. Recently added Immersive Web API (WebXR) can be used to create immersive user experiences through browser hosts by providing access to augmented reality (AR) and virtual reality (VR) capabilities [9].

2. **Improved JavaScript performance**: JavaScript is the dominant programming language of the Web. Since it is an interpreted language with dynamic typing, it's performance is inferior to compiled languages such as C++. With advances in Just-In-Time (JIT) compilation [36, 42, 86, 12], and with the introduction of WebAssembly (WASM) [41], a portable binary format for the web compilation, web clients can reach the near-native level of performance and handle more demanding tasks.

There are already efficient computer vision libraries developed in native languages such as C++. However, they cannot be used in browsers without relying on unpopular browser extensions which pose security and portability issues. There have been few efforts to develop computer vision libraries in JavaScript [113, 62, 43]. However, they often provide a handful of vision functions from certain domains such as object detection, video tracking, or deep learning. Expanding them with new algorithms and optimizing the implementation is a challenging task. On the other hand, Accelerated shape detection API [21] provides functions to detect shapes such as faces and bar codes while the efficient implementation is left to browser vendors. The works mentioned above suffer from the lack of either functionality, performance, or portability. An alternative approach to the client-side computation taken by developers was to offload vision processing tasks to the servers. This approach, however, sacrifices user privacy and suffers from always-online requirements and the increase in data transfer bandwidth and latency.

In this chapter, we describe our effort to provide a computer vision library for the Web that solves the mentioned shortcomings. Our approach is to translate the popular OpenCV library from C++ to a format that is highly optimizable on the Web. We show that this approach works great on the Web for several reasons:

1. OpenCV provides an expansive set of functions with optimized implementations.
2. It performs more efficiently than JavaScript implementations and performance can further improve through parallelism.
3. Developers can have access to a big collection of existing resources such as tutorials and examples.

## 2.2 Compiling to The Web

Most modern standards-compliant web applications are developed in JavaScript, CSS, and HTML [7]. In this model, HTML and CSS codes describe the HTML components, their appearance, and the layout while the JavaScript code implements the program logic. JavaScript

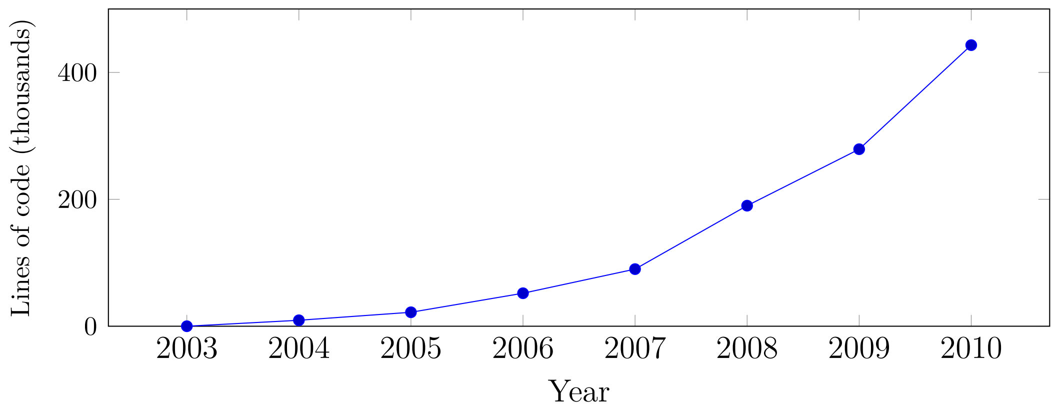


Figure 2.1: Number of JavaScript lines of code in Gmail source code

is an interpreted language and lacks the performance of languages such as C++ where the compiler can translate programs efficiently to the hardware beforehand. As web applications are getting larger and more demanding, web developers are interested in higher performance JavaScript execution. Figure 2.1 for instance shows how Gmail JavaScript code size has increased 50x in the course of 6 years [5].

Since 2004 an era called the browser war started, in which browsers started to compete with each other in JavaScript execution performance. Several approaches based on the idea of Just in time (JIT) compilation were proposed to make JavaScript faster on browsers. For instance, Gal proposed to record traces from program execution and compile the hot traces to efficient machine code [36]. Other approaches such as [42] infer types at run-time and compile individual functions to machine code. However, the compilation is happening at run-time, and since compilation with higher optimization levels is time-consuming, their scope is often limited to an execution trace or a single function. Furthermore, JavaScript is dynamically-typed, and changes in variables type during run-time will require re-compilation of the generated machine code. As an example, consider the example JavaScript function from Figure 2.2. Operator + in the fourth line will have a different implementation based on the operand types. While for numerical operands it would be an arithmetic addition; for string operands, it would be string concatenation. To be able to compile this function to

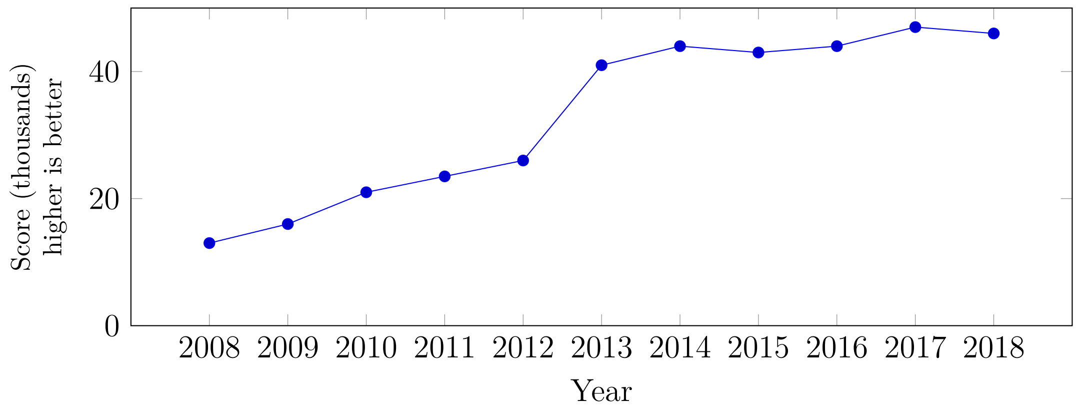


Figure 2.3: Chrome JavaScript engine (V8) benchmark score

machine code, the compiler must be able to infer the types correctly. Even with the correct type inference, the types can change at run time. Hence it might be necessary to monitor the code in case of changes in data types.

Figure 2.3 shows the steady performance improvement of executing the V8 benchmark over time in chrome [18]. However, performance has not improved significantly since 2015. Several approaches, such as plugins [56], ActiveX [76], and PNaCl [29], are proposed to achieve higher (near-native) performance. While they offer high performance, they did have not gained mainstream adoption due to portability and security issues. For instance, Apple forbids Java and Flash plugins on iOS devices, including the iPhone and iPad.

```
function add(a, b) {
 return a + b;
}
```

Figure 2.2: Implementation of *add* Function in JavaScript

### 2.2.1 Portable Low-level Formats for Web Compilation

Mozilla research proposed using a subset of JavaScript called asm.js to improve JavaScript execution performance even beyond JIT capabilities. Asm.js has two properties that allow

JavaScript engines to perform an extra level of optimizations and even, compile the entire program ahead of the execution.

1. It uses type coercion technique to enforce the code to be statically-typed. JavaScript engines can compile statically-typed JavaScript code ahead of the execution. Figure 2.4 displays an example of coercion to specify integer addition.

2. It uses JavaScript arrays to represent program memory such as heap and stack. Most modern browsers have optimized array access. Figure 2.5 shows how various sections of program memory such as stack and heap, can be implemented using typed arrays.

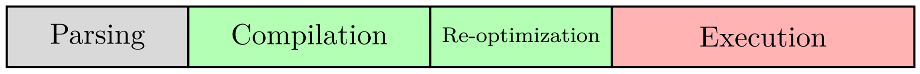
```
function add(x, y) {
 x = x | 0;
 y = y | 0;
 return (x + y) | 0;
}
```

Figure 2.4: Using type coercion to force the JavaScript engine to generate integer addition

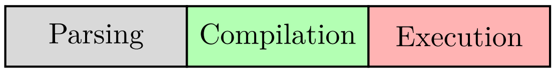
```
var buffer = new ArrayBuffer(32*1024);
var HEAP8 = new Int8Array(buffer);
var HEAP32 = new Int32Array(buffer);

function access_byte(index) {
 return HEAP8[index];
}

function access_word(index) {
 return HEAP32[index];
}
```

Figure 2.5: Using arrays to represent program memory in asm.js

Developing programs manually in asm.js is tedious. In fact, asm.js is intended as a target for translating programs developed in other languages to the Web using a tool called Emscripten. Emscripten is a compiler toolchain developed by Mozilla research to translate LLVM (Low-Level Virtual Machine) bit code to asm.js [112]. Using LLVM will make it possible to convert



(a) JIT



(b) asm.js

Figure 2.6: JIT vs asm.js run-time breakdwon

many languages such as C++ or Java to JavaScript since there a large number of language frontends supported by LLVM. The ability to port programs developed in compiled languages such as C++ to browsers is very compelling. Since not only programs can run faster, a vast amount of existing code becomes available to the Web. Some programmers might even prefer them to develop in JavaScript. For example, Figure 2.7 shows a sample C++ source file that is converted to asm.js equivalent code via Emscripten.

While asm.js performance is impressive, there are several shortcomings. Generated JavaScript programs tend to be very large, and parsing and compiling big JavaScript files becomes the bottleneck, especially on mobile devices with weaker processors. This issue was one of the main motivations for the development of WebAssembly [41].

WebAssembly (WASM) is a binary format that is designed to be a compiler target for the Web. It models an abstract stack machine, where instructions either pushes or pops values to/from the stack. WASM instructions are designed to be as close to native instructions as possible. It is fast to load and runs safely at predictably near-native speed. Although WASM is modeled by a stack machine, browsers will not necessarily follow that execution model and compile it to a format that runs more efficiently on modern processors. WASM can be represented in binary and textual format. The textual format is intended for human maintainability while the binary format is intended for production.

Figure 2.8 shows a simple function in WASM that performs integer addition. It starts with an empty stack, pushes the input parameters into the stack, then the last instruction pops

```cpp
int sum_five(unsigned char* array) {
  int res = 0;
  for (int i = 0; i < 5; ++i) {
    res += array[i];
  }
  return res;
}
```

(a) Sample C++ program

```javascript
var buffer = new ArrayBuffer(32768);
var HEAP8 = new global.Int8Array(buffer);
function sum_five($0) {
 $0 = $0|0;
 var $1 = 0, $10 = 0, $11 = 0, $12 = 0, $13 = 0, $14 = 0,
    $15 = 0, $2 = 0, $3 = 0, $4 = 0, $5 = 0, $6 = 0, $7 =
    0, $8 = 0, $9 = 0, label = 0, sp = 0;
 sp = STACKTOP;
 STACKTOP = STACKTOP + 16|0; if ((STACKTOP|0) >= (
    STACK_MAX|0)) abortStackOverflow(16|0);
 $1 = $0;
 $2 = 0;
 $3 = 0;
 while(1) {
  $4 = $3;
  $5 = ($4|0)<(5);
  if (!($5)) {
   break;
  }
  $6 = $1;
  $7 = $3;
  $8 = (($6) + ($7)|0);
  $9 = HEAP8[$8>>0]|0;
  $10 = $9&255;
  $11 = $2;
  $12 = (($11) + ($10))|0;
  $2 = $12;
  $13 = $3;
  $14 = (($13) + 1)|0;
  $3 = $14;
 }
```

(b) Emscripten-emitted asm.js code

Figure 2.7: Compilation of a sample C++ program into asm.js by Emscripten.

```
(func (param $0 i32) (param $1 i32)
  local.get $1
  local.get $0
  i32.add)
```

Figure 2.8: WASM implementation of an integer addition function in textual format

the operands, calculates their sum, and pushes the result into the stack again. The return value of the function will be the last element left on the stack.

Compared to asm.js, WASM is more compact and is much quicker to parse and compile. WASM will eventually make asm.js obsolete. However, WASM is still under development and not fully supported by older JavaScript engines. We have used Emscripten to compile the OpenCV source code into both asm.js and WASM. Both versions offer the same functionality and can be used interchangeably.

## 2.3 Generating OpenCV.js

This section describes the process of generating OpenCV.js from OpenCV source doe. We have used Emscripten to compile OpenCV to the web. However, for two reasons, several OpenCV components must be excluded:

1. Not all of OpenCV's offerings are compatible with the web. For instance, functions to access media devices such as cameras, and graphical user interfaces, are platform-dependent and cannot be compiled to the web. Those functions, however, can be implemented using HTML5 primitives. For instance, media capture $getUserMedia$ can be used to access media devices, and the Canvas element can display graphics.

2. Some of OpenCV functions are only used in certain application domains that are not common in typical web development. For instance, camera calibration functionality

```cpp
int sum_five(unsigned char* array) {
  int res = 0;
  for (int i = 0; i < 5; ++i) {
    res += array[i];
  }
  return res;
}
```

(a) Sample C++ program

```
(module
  (type $t0 (func))
  (type $t1 (func (result i32)))
  (func $__wasm_call_ctors (type $t0))
  (func $sum_five (export "sum_five") (type $t1) (result i32)
    i32.const 0
    i32.load8_u offset=1024
    i32.const 0
    i32.load8_u offset=1025
    i32.add
    i32.const 0
    i32.load8_u offset=1026
    i32.add
    i32.const 0
    i32.load8_u offset=1027
    i32.add
    i32.const 0
    i32.load8_u offset=1028
    i32.add)
  (table $T0 1 1 anyfunc)
  (memory $memory (export "memory") 2)
  (global $g0 (mut i32) (i32.const 66576))
  (global $__heap_base (export "__heap_base") i32 (i32.const
     66576))
  (global $__data_end (export "__data_end") i32 (i32.const 1029))
  (global $array (export "array") i32 (i32.const 1024))
  (data (i32.const 1024) "\01\02\03\04\05"))
```

(b) Emscripten-emitted WASM code in textual format

Figure 2.9: Compilation of a sample C++ program into WASM by Emscripten.

| **Module** | **Provided Functions** |
|---|---|
| Core | Image manipulation and core arithmetic |
| Image Processing | Numerous functions to process and analyze images |
| Video | Video processing algorithms such as tracking, background segmentation and optical flow |
| Object Detection | Haar and HOG based cascade classifiers |
| DNN | Inference of Caffe, Torch, TensorFlow trained networks |
| GUI | Helper functions to provide graphical user interface, to access web images and videos, and to display content |

Table 2.1: OpenCV.js modules and provided functions

has applications in automation and robotics. To reduce the size of the generated library for general usage, we have identified the least commonly used functions from OpenCV and excluded them from the library. However, since many of the excluded functions can be useful in special use cases, we have provided a mechanism to extend the library with a list of user-selected functions.

.

Table 2.1 categorizes and lists the functions that are included within OpenCV.js.

Computer vision programs developed in C++ can access configuration files on the host machine. Browsers, however, are sandboxed for security purposes and do not give web applications access to regular files on the local machine. OpenCV.js provides a virtual file system that can pack all the necessary files that the program will have to access. The virtual file system can be modified at run-time by adding or modifying the files.

Figure 2.10 shows an overview of OpenCV.js and how it interacts with web applications and standard web APIs. Web applications will use Opencv.js API to access the provided functions as listed in Table 2.1. While the vision functions from OpenCV are compiled either into WASM or asm.js, GUI features, and media capture capabilities are provided by a JavaScript module (util.js). OpenCV.js utilizes standard web APIs such as WebRTC and

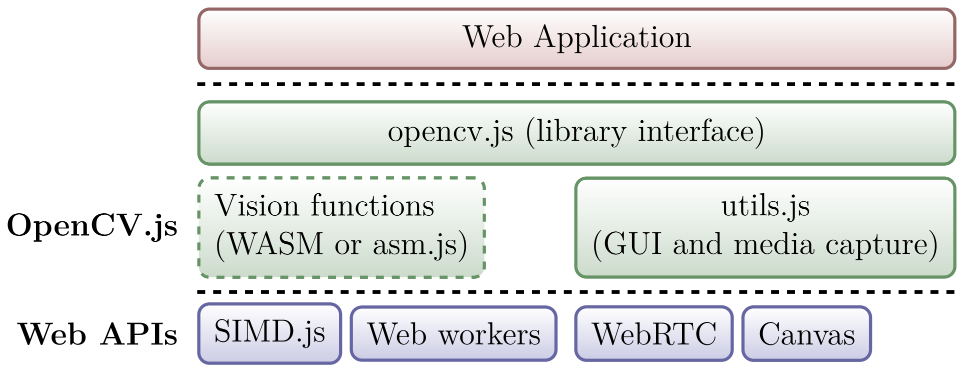

Web Application
opencv.js (library interface)
OpenCV.js
Vision functions (WASM or asm.js)
utils.js (GUI and media capture)
Web APIs
SIMD.js
Web workers
WebRTC
Canvas

Figure 2.10: OpenCV.js components and its interactions with web applications and web APIs

Video/Canvas for media access and GUI capabilities and uses web workers and SIMD.js to implement parallel algorithms.

During the compilation process with Emscripten, C++ high-level language information such as class and function identifiers are replaced with mangled names. It is almost impossible for developers to develop programs through mangled names. We develop an interface for the library that exposes OpenCV entities such as functions and classes to JavaScript similar to normal OpenCV that many programmers are already exposed to it. Since OpenCV is enormous, and growing continuously through new contributions, continually updating the port by hand is impractical. Hence, we propose a semi-automatic approach that takes care of the tedious parts of the translation process while allowing the expert insight that enables high-quality/efficient code production. Figure 5.9 lists the steps involved in the process of converting OpenCV C++ code to JavaScript.

At first, the OpenCV source code is configured to disable components and implementations that are platform-specific or translate poorly to the web. Emscripten provides a framework called Embind that can call arbitrary functions in Emscripten-compiled C++ and pass arguments (which involves type translation) from a JavaScript. To use Embind, we extract information about classes and functions from the OpenCV source code automatically. Figure 2.12 shows the binding information that is generated for a sample OpenCV class. For

```cpp
bool haar_cvt::convert(const String& oldcascade,
                       const String& newcascade)
```

(a) Original function signature

```
function __ZN2cv8haarcvtL7convertERKNS_6StringES3_(
  $oldcascade,
  $newcascade)
```

(b) Generated function signature with the mangled name

Figure 2.11: Compilation of a sample C++ function into JavaScript by Emscripten.

the sake of efficiency, binding information of OpenCV core module, which includes OpenCV main data structure (i.e., ”cv::Mat”), is manually provided.

We maintain a white list of OpenCV classes and functions that are included in the final JavaScript build. This list can be updated by users to include or exclude OpenCV modules and functions. By providing the binding information and functions white list, Emscripten generates a glue code that maps JavaScript symbols to C++ symbols and compiles it along with the rest of the OpenCV library into JavaScript. The output of this process will be a JavaScript file (opencv.js) that serves as the library interface along with WASM or asm.js implementation of OpenCV functions. utils.js which includes GUI, I/O, and utility functions, and is implemented separately, will also be linked with the rest of opencv.js.

## 2.4 Using OpenCV.js

OpenCV.js API is based on OpenCV C++ API and shares many similarities with it. For instance, it exports C++ functions to JavaScript with the same name and signature. It also supports function overloading and default parameters. This similarity makes migration to JavaScript easier for users who are already familiar with OpenCV development in C++. Although OpenCV.js ports C++ classes to JavaScript objects with the same member func-

```cpp
class CV_EXPORTS_W MSER : public Feature2D {
public:
  CV_WRAP static Ptr<MSER> create(
    int _delta=5, int _min_area=60, int _max_area=14400,
    double _max_variation=0.25, double _min_diversity=.2,
    int _max_evolution=200, double _area_threshold=1.01,
    double _min_margin=0.003, int _edge_blur_size=5);

  CV_WRAP virtual void setDelta(int delta) = 0;
  CV_WRAP virtual int getDelta() const = 0;
  // The rest of class declaration
};
```

(a) MSER class declration in OpenCV source code

```cpp
emscripten::class_<cv::MSER, base<Features2D>>("MSER")
  .class_function("create",
     select_overload<Ptr<MSER>(int, int, int, double,double,int,double
        ,douboe,int)>
     (&Wrappers::create_MSER_wrapper)
  .function("setDelta",
     select_overload<int(cv::MSER&)>(&Wrappers::MSERT_set_delta_wrap),
     pure_virtual())
  .function("getDelta",
     select_overload<int(cv::MSER&)>(&Wrappers::MSERT_get_delta_wrap),
     pure_virtual())
  // The rest of class bindings
;
```

(b) Generated binding information for MSER class

Figure 2.12: Example of binding generation

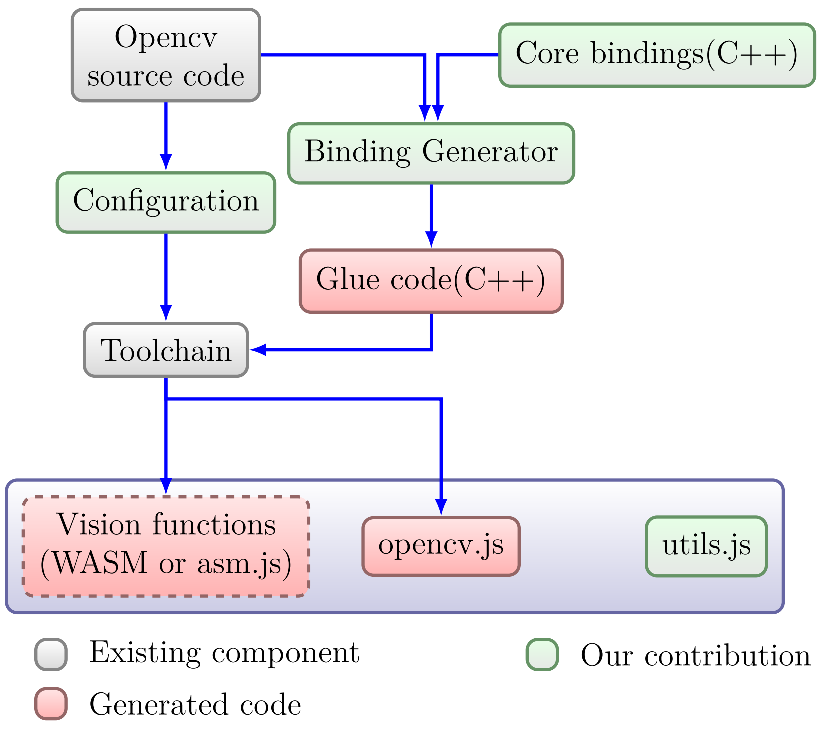


Figure 2.13: The flow of generating OpenCV.js from OpenCV source code

tions and properties, basic data types are different between the two versions. For instance, JavaScript is using double-precision floating-point numbers for all numerical types, whereas, in C++, numerical values have several type options (e.g., short, int, and floating-point). JavaScript engines use the garbage collector(GC) to manage program memory. However, GC activity hurts performance. Hence, OpenCV.js uses static memory management, and programmers are responsible for freeing OpenCV.js objects when they are no longer in use. Since manual memory management for primitive types is tedious, we have used JavaScript equivalents for basic C++ types such as numbers, boolean values, and strings. All $std :: vectors$ are translated into JavaScript arrays except for vectors of $cv :: Mat$. This is particularly helpful since by removing the vector, it will remove all the $cv :: Mat$ elements automatically. Table 2.2 shows equivalent JavaScript data types for basic C++ data types.

Figure 2.14 shows a sample JavaScript program. This program uses $MOG2$ method (based on the Gaussian mixture model [118]) provided by OpenCV.js to subtract the background

| **C++ Type** | **JavaScript Type** |
|---|---|
| Numerical types (e.g., int, float) | Number |
| bool | Boolean |
| enum | Constant |
| std::string | String |
| Primitive types (e.g. cv::Point) | Value objects |
| std::vector (of primitive types) | JavaScript Array |
| std::vector (of cv::Mat) | cv.Vector |

Table 2.2: JavaScript equivalent types of basic OpenCV C++ data types

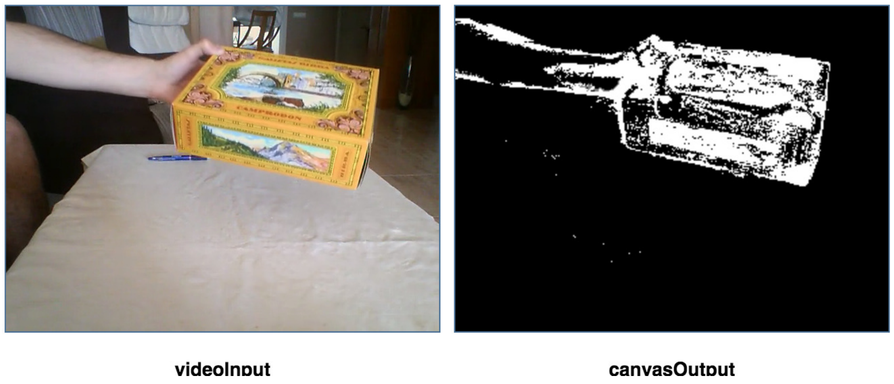
from the input video. This example works on top of a simple HTML page with an HTML5 video element named *videoInput* serving as the input source and a canvas element named *canvasOutput* which renders the program output. In the line 21, *cv.VideoCapture* utility function is invoked to access input video frames from the video element. In lines 22 and 23, two OpenCV matrices are created to hold the input and output frames. In the 24th line, a background subtractor is instantiated. The background subtraction algorithm needs to access the history of frames from the input video to correctly partition the image in the foreground and background. This example assumes that the input video contains 30 frames per second. Hence, a timer is used to invoke *processVideo* function every 1/30 of a second. At every invocation of *processVideo* function, we feed the next frame of the video to the background subtractor and extract the foreground mask (line 10). In line 11, we will display the result on the output canvas and then schedule the function for processing the next video frame (line 14). Figure 2.15 shows two snapshots of running this program inside a browser.

## 2.5 Performance Evaluation

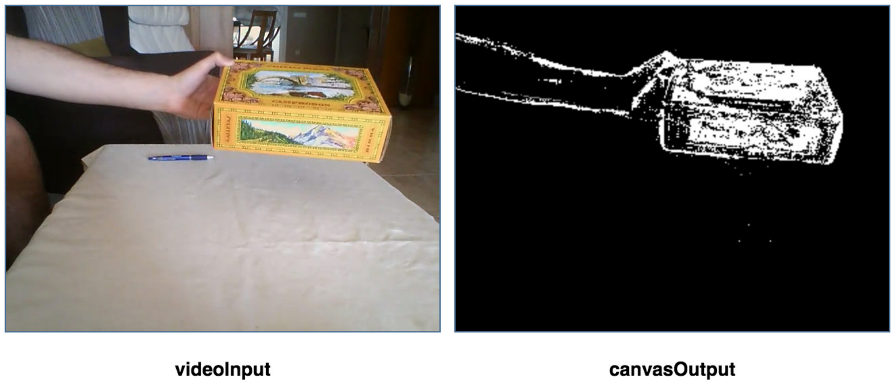
This section presents the performance evaluation of OpenCV.js. Our evaluation workloads include both primitive kernels that perform simple operations such as pixel-wise addition or convolution and more sophisticated vision applications. Our selected vision applications

```
function processVideo() {
  try {
    if (!streaming) { // clean and stop.
      frame.delete(); fgmask.delete(); fgbg.delete();
      return;
    }
    let begin = Date.now();
    // start processing.
    cap.read(frame);
    fgbg.apply(frame, fgmask);
    cv.imshow('canvasOutput', fgmask);
    // schedule the next one.
    let delay = 1000/FPS - (Date.now() - begin);
    setTimeout(processVideo, delay);
  } catch (err) {
    utils.printError(err);
  }
};

var video = document.getElementById('videoInput'),
  cap = new cv.VideoCapture(video),
  frame = new cv.Mat(video.height, video.width, cv.CV_8UC4),
  fgmask = new cv.Mat(video.height, video.width, cv.CV_8UC1),
  fgbg = new cv.BackgroundSubtractorMOG2(500,16,true);
const FPS = 30;
// schedule the first frame
setTimeout(processVideo, 0);
```

Figure 2.14: Example OpenCV.js program to subtract background from the input video



a) First snapshot



b) Second snapshot

Figure 2.15: Two snapshots of the sample OpenCV.js program from Figure 2.14 running in a web browser

include implementation of **Canny**'s algorithm for finding edges [20], finding **face**s using Haar cascades [58], and finding **people** by using histogram of gradients as features [26]. We have used an instance of Firefox 56 running on Intel Corei7-3770 CPU with 8GB of RAM with Ubuntu 16.04 as our set up and ran experiments over sequences of video data (400-600 frames) collected from Xiph.org archive. Figures 2.16 and 2.17 show the performance of simple kernels and vision applications running in the browser compared to their desktop equivalent with parallelism disabled. Experiments are repeated for different pixel types that are supported by the benchmarks. As was shown, in all cases, the performance is close to the native.

While we found WASM and asm.js performance to be close, the WASM build of the library is significantly faster to initialize (more than 20x faster) and is more compact. To reduce the library size, we have used a JavaScript port of the Zlib library to decompress a zipped version of the OpenCV.js at run-time. Figure 2.18 shows the size comparison of different builds of OpenCV.js. Note that the size of the library depends on the list of functions that are distributed with the library, which is subject to frequent changes.

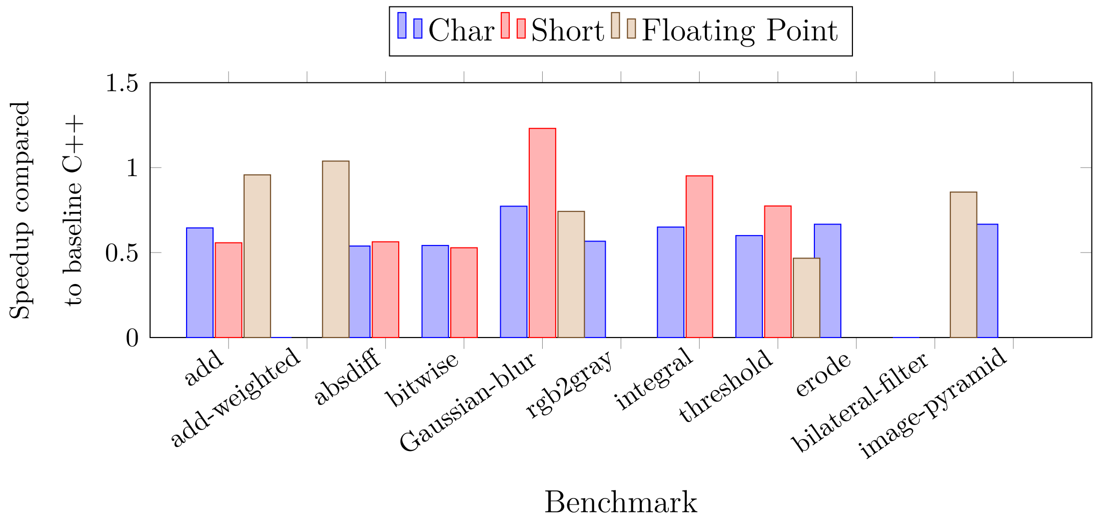


Figure 2.16: Performance comparison of native and WASM versions of primitive kernels

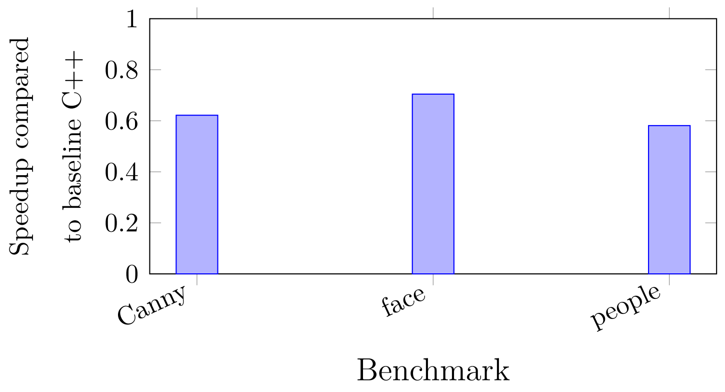


Figure 2.17: Performance comparison of native and WASM versions of vision applications

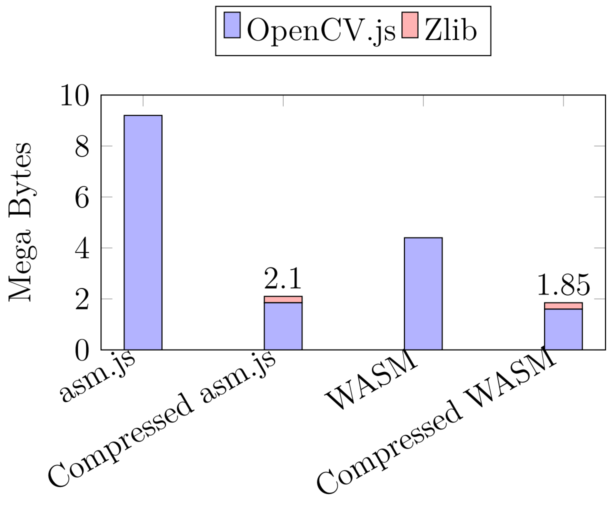


Figure 2.18: Size comparison of asm.js and WASM versions of OpenCV.js*

## 2.6 Availability

OpenCV.js is released as part of the master branch of OpenCV library[1] and can be compiled directly from the latest OpenCV source code releases. This will also ensure that it will be tested to work with all future OpenCV releases and features.

[1]https://www.gihtub.com/opencv/opencv

# Chapter 3

# Parallel Processing in OpenCV.js

Computer vision is computationally demanding. On our evaluation platform, each iteration of Canny, face, and people applications developed with OpenCV.js take on average 7 ms, 345 ms, and 323 ms to process an image with a resolution of 640 by 480 pixels respectively. While in this case, Canny is fast enough to be computed in real-time, face, and people detection examples do not support real-time and interactive use cases. Fortunately, computer vision algorithms are inherently parallel, and with good algorithm design and optimized implementation for parallel hardware, a significant speedup can be achieved. OpenCV already comes with parallel implementations of algorithms for various hardware architectures. In this section, we demonstrate how two parallel processing techniques that target multicore processors (multithreading) and SIMD (Single-Instruction-Multiple-Data) units can be used to significantly improve the performance of OpenCV.js. We have skipped GPU implementations at the moment due to the lack of a standard web API for general-purpose programming on GPUs.

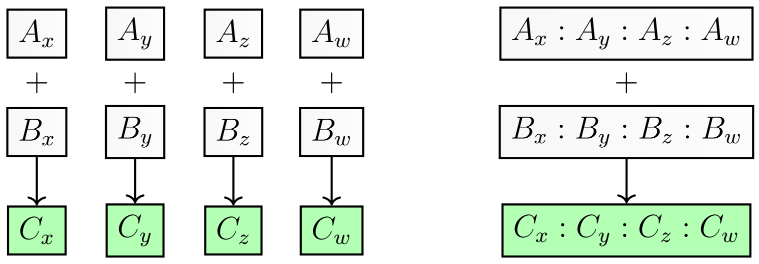


Figure 3.1: Scalar vs SIMD addition of four integers

## 3.1 Parallel Processing using ECMAScript SIMD

ECMAScript SIMD, formerly known as SIMD.js, is a web API that exposes processor vector capabilities to the web [6, 50]. ECMAScript SIMD is compatible with the common subset of Intel SSE2 and ARM NEON instruction sets that runs efficiently on both architectures. Both instruction sets define vector instructions that operate on 128-bit wide vector registers. SIMD instructions can operate all vector data points simultaneously. Figure 3.1 shows how vector registers can be utilized to add four integers using one vector instruction. ECMAScript SIMD defines integer vector types of i8x16, i16x8, i32x4, i64x2 that can hold eight short integers, sixteen bytes, four integers, and two long integers, respectively. It also defines two floating vector types of f32x4 and f64x2 that can hold four floating-point and two double-precision floating-point numbers. Table 3.1 lists ECMAScript SIMD vector instructions that are available for use on each vector type.

SIMD is proven to be very effective in speeding up multimedia, graphics, and scientific workloads [79, 50, 93]. Many OpenCV functions, including core routines, are already implemented using vector intrinsics [79]. We have adopted the work done by [48] to translates OpenCV vectorized implementations using SSE2 intrinsics into JavaScript with SIMD.js instructions. The inclusion of SIMD.js parallel implementations will not affect the library interface. Currently, SIMD.js can only be used in the asm.js context and is supported by Mozilla Firefox and Microsoft Edge browsers. Since SIMD in WASM is planned to have the same spec as SIMD.js, similar performance numbers are expected.

| **Vector Types** | **Operations** |
|---|---|
| i8x16, i16x8, i32x4, i64x2, f32x4, f64x2 | splat, extract_lane_s, replace_lane, add, sub, neg, eq, ne |
| i8x16, i16x8, i32x4, i64x2 | shl, shr_s, shr_u, any_true, all_true, lt_s, lt_u, le_s, le_u, ge_s, ge_u, gt_s, gt_u |
| f32x4, f64x2 | lt, gt, le, ge, abs, min, max, div, sqrt, convert_to_int |
| i8x16, i16x8, i32x4, i64x2, f32x4 | mul |
| i8x16, i16x8 | add_saturate_s, add_saturate_u, sub_saturate_s, sub_saturate_u, narrowing |
| i8x16, i16x8, i32x4 | widening |
| v128 | and, or, xor, not, bitselect, load, store |

Table 3.1: ECMAScript SIMD types and operations

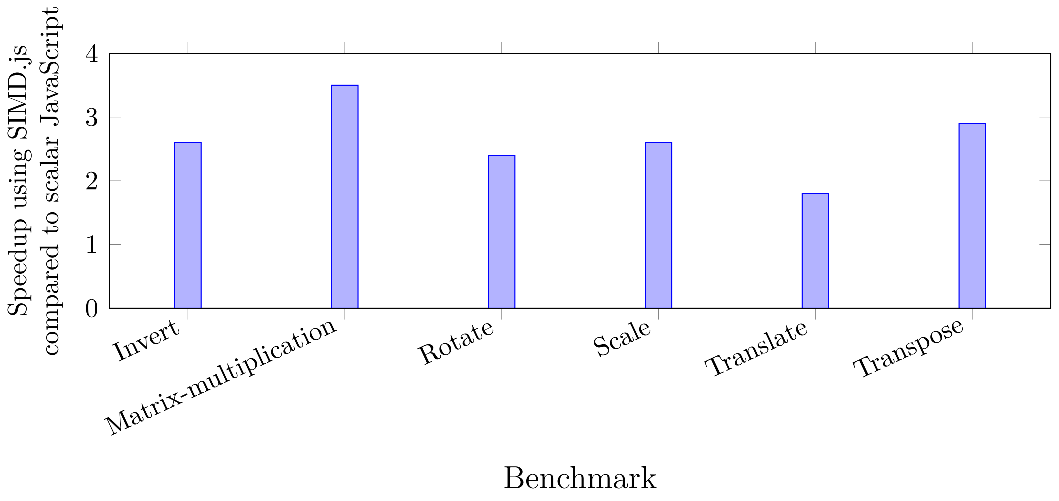


Figure 3.2: Performance comparison of vectorized and scalar JavaScript implementation of matrix operations (asm.js version)

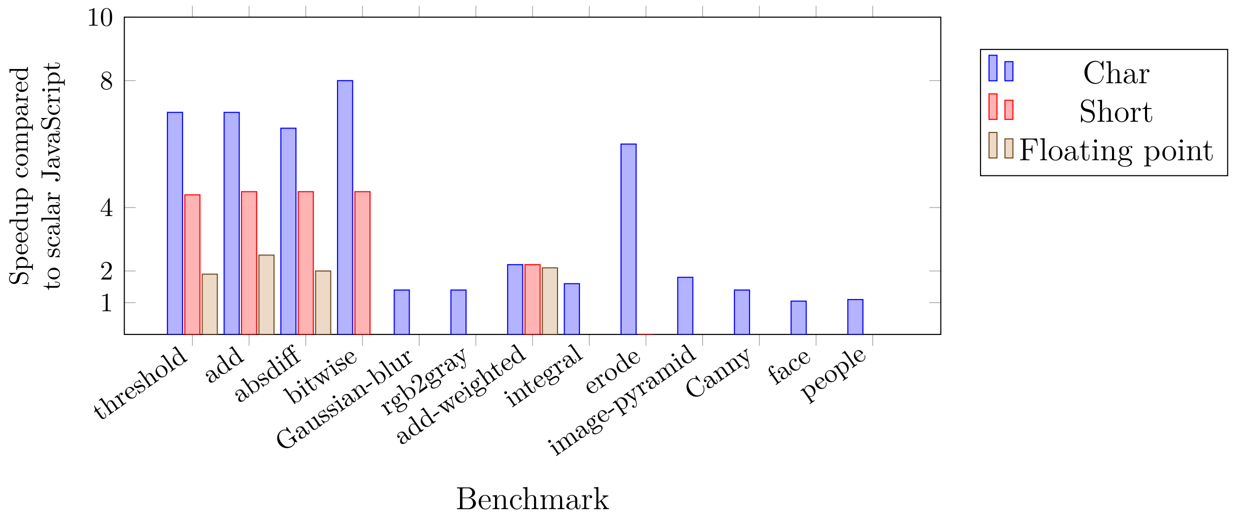


Figure 3.3: Performance improvement with JavaScript vectorization (asm.js version)

Figure 3.3 shows the speedup that is obtained by SIMD.js on selected kernels and applications running on Firefox. Up to 8x speedup is achieved for primitive kernels. As expected, the speedup is higher for smaller data types, since more data points are packed in vector registers. There are fewer vectorization opportunities in complex functions such as Canny, face, and people detection.

## 3.2 Thread-level Parallelism Using Web Workers

JavaScript programs use web workers [8] for parallel processing of compute-intensive tasks. Web workers communicate by passing messages, which incur a high cost for transferring large messages such as images. SharedArrayBuffer [4] is recently proposed as storage that can be shared between multiple web workers. It can be used to implement the shared-memory parallel programming model. OpenCV uses the *parallel_for* framework for parallel implementation of algorithms. In this framework, parallel implementations of algorithms implement the base class *ParallelLoopBody* that process a portion of images. As shown in Figure 3.4, at run-time workloads are partitioned into multiple smaller workloads that a range

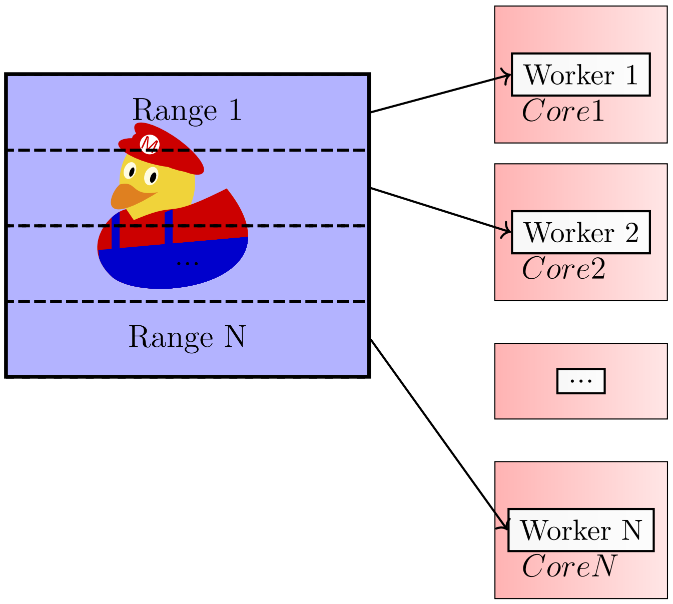


Figure 3.4: multi-threaded image processing with *Parallel_for* implementation

```
public ParallelLoopBody
{
public:
 virtual void operator () (const Range& range) const;
}
```

Figure 3.5: Base class for implementing parallel algorithms in OpenCV

specifies each. These workloads can be distributed among available compute threads for concurrent processing. OpenCV supports a variety of multithreading backends depending on the operating system such as Intel Threading Basic Blocks (TBB) [75], Windows threading, Apple GCD, OpenMP [25], and Posix Threads [17]. With recent Emscripten developments, we were able to translate P-threads API into equivalent JavaScript using web workers with shared array buffers. OpenCV.js build with multithreading support has a pool of web workers and allocate a worker when a new thread is spawned. Besides, it exposes OpenCV API to dynamically adjust the concurrency such as changing the number of concurrent threads such as ”cv.SetNumThreads”.

To observe the performance using multiple web workers, we measured the performance of three application benchmarks that did not gain from SIMD vectorization. We used different

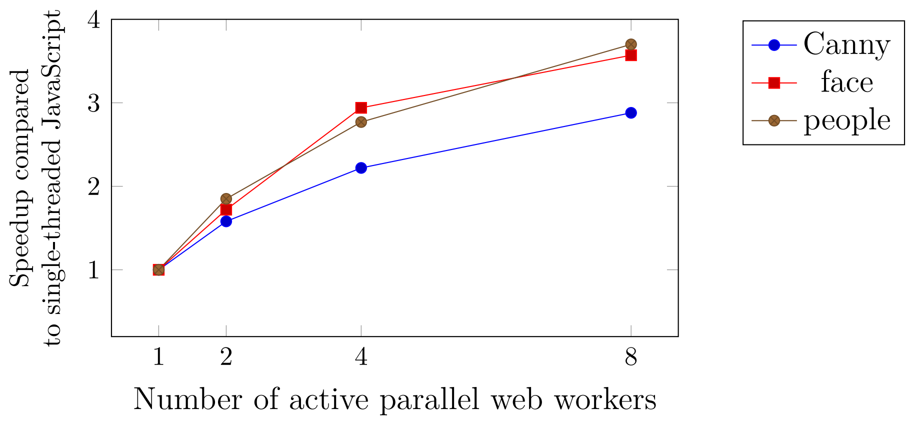


Figure 3.6: Speedup achieved using multiple web workers (asm.js)

numbers of workers up to 8. OpenCV load balancing algorithm divides the workload evenly between threads. Figure 3.6 shows that on a processor with 8 logical cores, between 3 to 4 times performance speedup is obtained. Note that a similar trend is observed on native p-threads implementation of the mentioned functions. It should be noted that parallelism improvement from multiple threads and SIMD are additive.

# Chapter 4

# A Library for FPGA Implementation of Vision Algorithms

The ever-growing usage of computer vision in our daily lives [2] requires real-time and power-efficient implementations. FPGAs are very effective in accelerating various computer vision algorithms [63, 114, 46, 67]. Unfortunately, developers may not be familiar with FPGA design, let alone efficient implementation. Even a CPU implementation of vision algorithms using programming languages such as C/C++ is time-consuming and error-prone. Moreover, despite advances in High-Level Synthesis (HLS) tools, they are still not able to generate efficient hardware implementation for vision algorithms developed in general-purpose programming languages (e.g., C++) [19, 107, 84]. The main reason is that HLS tools do not incorporate specific optimization methods of vision algorithms, which are critical to the efficient implementation of such algorithms. On the other hand, popular vision libraries, such as OpenCV [16], only provide efficient implementations for CPUs and GPUs. FPGA designers often consider the application-specific design and implementation on FPGAs. For example, several computer vision applications including stereo vision [51, 104, 53], SIFT [103, 74], CNNs and DNNs [35, 38, 115, 116] have been designed and implemented for FPGA acceleration.

This chapter presents a scalable library developed in OpenCL that provides the basic blocks for building computer vision systems on FPGAs. This library supports a wide variety of vision kernels and can be configured with different structural and functional parameters. It can be used as part of an OpenCL program independently or as part of a high-level framework. To make the library portable, We have tried to use vendor-specific functions as little as possible.

## 4.1 Introduction to OpenVX

OpenVX is an open standard for cross-platform acceleration of computer vision algorithms [81]. OpenVX defines a set of primitive and widely used vision kernels that can be connected to describe computer vision pipelines as Directed Acyclic Graphs (DAG). In such DAGs, nodes represent data processing kernels while the edges represent data dependencies between them. This model maps nicely to computer vision development since many vision pipelines can be modeled as DAGs. Figure 4.1 shows how OpenVX specification can be used in computer vision algorithm development process. While it can be used directly to model the application, it can also be used as an implementation backend for high-level computer vision frameworks.

Figure 4.2 shows different stages in lifecycle of an OpenVX algorithm graph. Initially, the vision developer instantiates the OpenVX elements to describe the algorithm graph. After the graph definition is finalized, the graph is verified for soundness. For instance, the graph must be connected, should not include any loops, and parameters passed to the graph elements must have values within the valid range. Upon successful verification, graphs can be executed, possibly for repeated times. No graph verification is needed for subsequence

OpenVX is designed to achieve the following list of objectives:

- **System-level optimization**: OpenVX graph-based model provides greater optimization opportunities above the traditional function level that applies to the whole system [81]. An example of such optimizations is kernel aggregation, in which a specific set of nodes in a graph are replaced with a single node that offers the same functionality. Another example is tiling, in which, images are broken down to smaller sub-images called tiles that can be processed separately. Processing tiles lead to a smaller memory footprint and cache-friendly implementations. Tiles can also potentially be processed in parallel.

- **Portability**: OpenVX abstracts away hardware-specific details from the algorithm specification. Hence graphs can target different platforms.

- **Improved productivity**: Decoupling the specification and the implementation makes developing vision algorithms easier. Computer vision developers can focus on algorithm design and rely on platform vendors to provide the optimized implementation. In addition, a single specification can be translated into different implementations that meet different performance and power budget constraints.

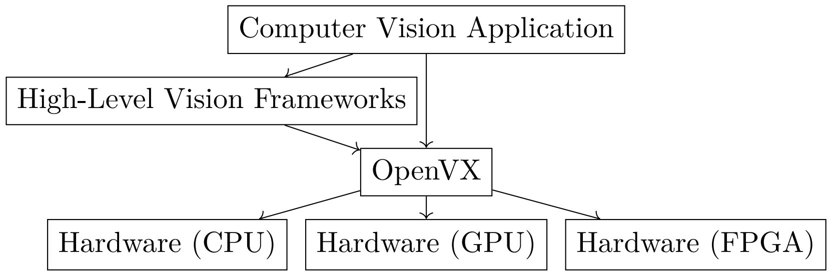


Figure 4.1: Typical OpenVX use cases

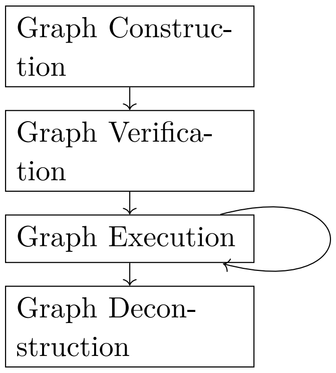


Figure 4.2: OpenVX graph lifecycle

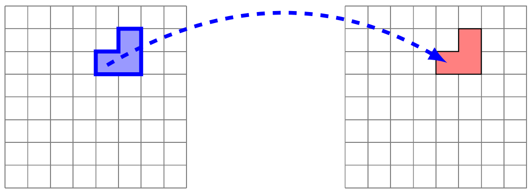
executions unless there are changes in the graph definition. Finally, after the execution is finished, the graph will be deconstructed to free up resources.

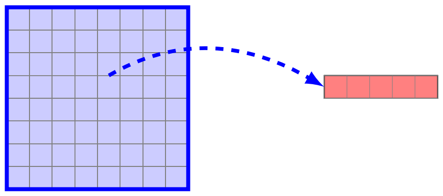
Various hardware vendors have provided their implementation of OpenVX run-time for optimizing and executing the graphs such as Nvidia VisionWorks [3] and AMD AmdoVX [1]. Tagliavini et al. proposed a run-time solution for optimizing OpenVX graphs on embedded many-core accelerators [91, 92]. OpenVX has also been investigated for building computer vision systems with real-time constraints [46, 108, 31, 109].

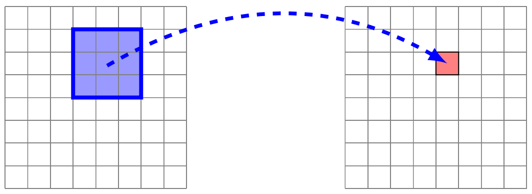
OpenVX defines several vision processing constructs to describe algorithm graphs:

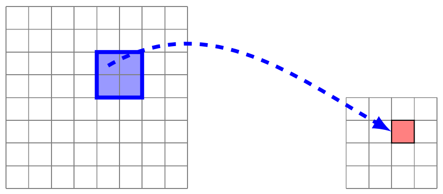
1. **Vision function nodes**: OpenVX provides a set of widely used primitive vision functions that act as basic blocks of building more complex algorithms. We have thoroughly analyzed all the vision functions provided[1] and classified them based on the pattern that they access to input image pixels. Table 4.1 lists various categories of vision functions.

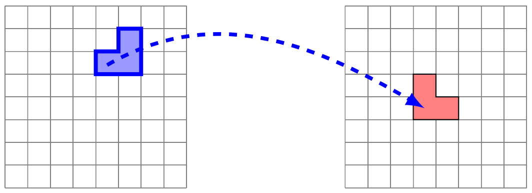
[1]Based on OpenVX version 1.2

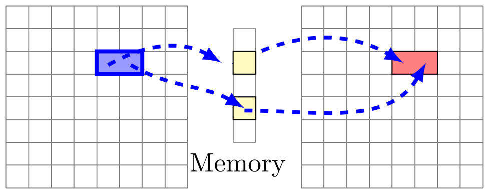

(a) Pixel-wise


(b) Statistical


(c) Fixed-rate stencil


(d) Multirate stencil (downsample)


(e) Geometric



(f) Table lookup

Figure 4.3: Vision function data access patterns

| **Category** | **Formal Definition** | **Vision Functions** |
|---|---|---|
| Pixel-wise | out(x, y) = f(in(x, y)) | absolute difference, accumulate, accumulate squared, accumulate weighted, addition/subtraction, bitwise operations, channel combine, channel extract, color convert, convert bit depth, magnitude, phase, pixel-wise multiplication, threshold, min, max |
| Fixed-rate Stencil | $out(x, y) = \sum_{i=-k}^{i=k} \sum_{j=-k}^{j=k} f(in(x+i, y+j))$ | Box filter, Sobel filter, non-maxima suppression, custom convolution, erode, dilate, Gaussian blur, nonlinear filter, integral image, Median filter |
| Multi-rate Stencil | $out(x, y) = \sum_{i=-k}^{i=k} \sum_{j=-k}^{j=k} f(in(Nx+i, Ny+j))$ | down-sample, scale image |
| Statistical | $out = \sum_{i=0}^{i=Width} \sum_{j=0}^{j=Height} f(in(i, j))$ | histogram, mean, standard deviation, Min,max location |
| Geometric | $out(x, y) = in(h(x, y), h'(x, y))$ | remap, warp affine, warp perspective |
| Table lookup | out(x, y) = table[in(x,y)] | table lookup |
| Non-primitive | N/A | equalize histogram, fast corners, Harris corners, Gaussian image pyramid, optical flow pyramids, Canny edges, LBP, HOG, HoughLinsP |

Table 4.1: Categorization of supported OpenVX vision functions

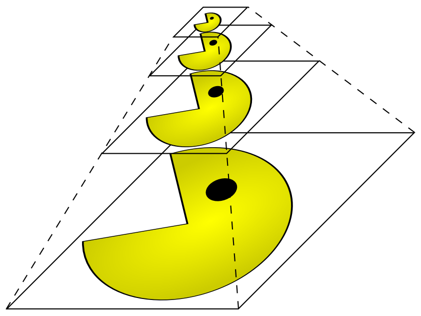


Figure 4.4: An image pyramid with four levels

2. **Data object nodes**: OpenVX defines several objects to represent data used by algorithms, including images, scalars, arrays, and image pyramids. Image pyramids (Figure 4.4) represent an image at multiple scales. For robustness and scale-invariance, computer vision systems often process multiple scales of images. Data objects have properties associated with them, such as their type and size that must be either set during the DAG construction time by the designer or be inferred from the rest of the graph.

3. **Select nodes**: Select nodes implement predication in vision algorithms. They have two input images and produce one output image, all with the same pixel type. A Boolean input value is used to select one of the inputs and set it as the output.

4. **Delay object nodes**: Delay objects are nodes that provide access to arbitrary data objects (e.g., image, array, scalar) from the Nth previous DAG invocation where $N$ is a parameter provided by the algorithm designer. They are in particular used in video processing algorithms that require information from multiple frames. Examples of such algorithms include optical flow, motion detection, and background extraction.

### 4.1.1 Examples

OpenVX can be used to implement a wide variety of image processing and computer vision algorithms. In this section, we describe several illustrative examples.

**Lane Detection (LD) Algorithm** One of the fundamental vision algorithms in autonomous driving cars is Lane Detection (LD). Many LD algorithms are proposed over time [110, 45]. A simple LD algorithm is described in Figure 4.5 while the corresponding OpenVX graph is shown in Figure 4.6. Figure 4.7 shows the output of different stages of the algorithm on a sample input frame. Color input frame captured by a camera in front of a vehicle (Figure 4.7(a)) will be converted to a gray-scale image (Figure 4.7(b)). Then, using a perspective transform, a certain area of interest is considered for further processing (Figure 4.7(c)). The image will be enhanced using a filter and threshold operation to highlight the lanes better (Figure 4.7(d,e)). Finally, lane segments will be detected using probabilistic *Hough* transform (Figure 4.7(f)). The Hough transform algorithm detects geometric shapes such as lines and circles in an image [64].

```
Inputs: Image in_img, Matrix transoform_mat, Matrix filter,
Integer thresh_val
Output: Array line_segments
begin
y_img ← convert_to_grayscale(in_img)
b_img ← warp_perspective(y_img, transoform_mat)
f_img ← filter(b_img, filter)
t_img ← threshold(f_img, thresh_val)
line_segments ← houghlinsp(t_img)
return line_segments
end
```

Figure 4.5: Lane detection algorithm example

,

**Automatic Contrast** adjusts the contrast of the input color (RGB) image based on the histogram of pixel intensity values. Figure 4.8 shows the OpenVX graph of automatic contrast graph.

**Census Transform** calculates visual descriptors based on census transform for an input grayscale image [111]. Figure 4.9 shows the OpenVX graph of automatic contrast graph. We have extended OpenVX with census transform vision function to implement this algorithm.

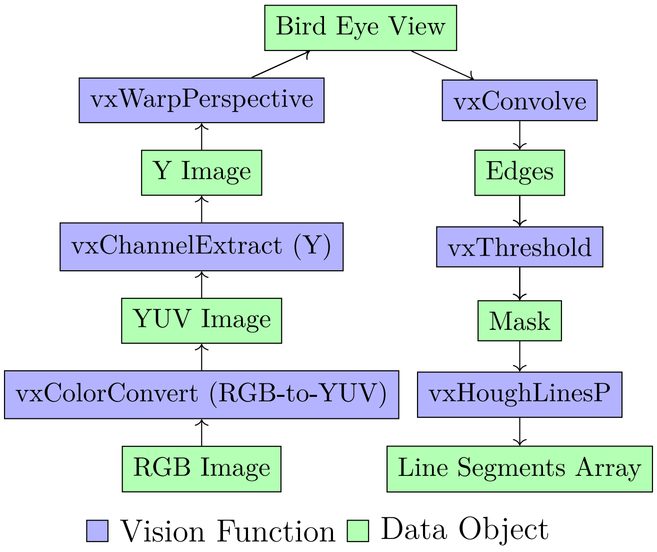
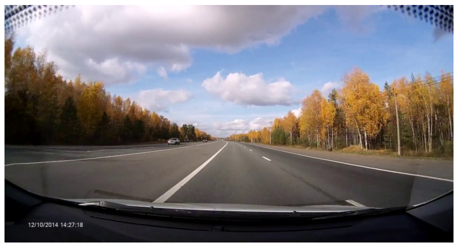
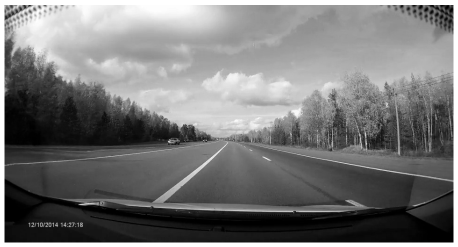
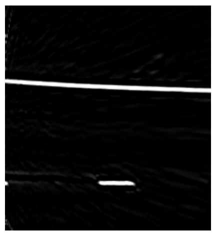


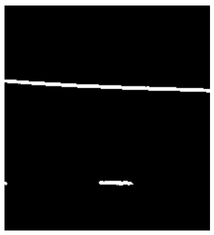
Figure 4.6: Lane detection algorithm graph


(a) Input image


(b) Grayscale image

(c) Bird-eye view


(d) Filtered image


(e) Mask

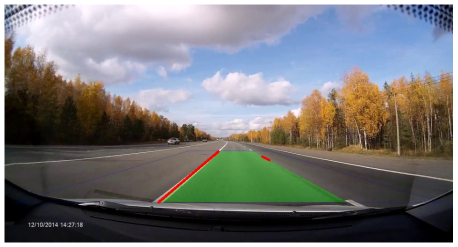

(f) Input image with detected lanes drawn in red

Figure 4.7: Demonstration of lane detection algorithm, (video obtained from intel.com)

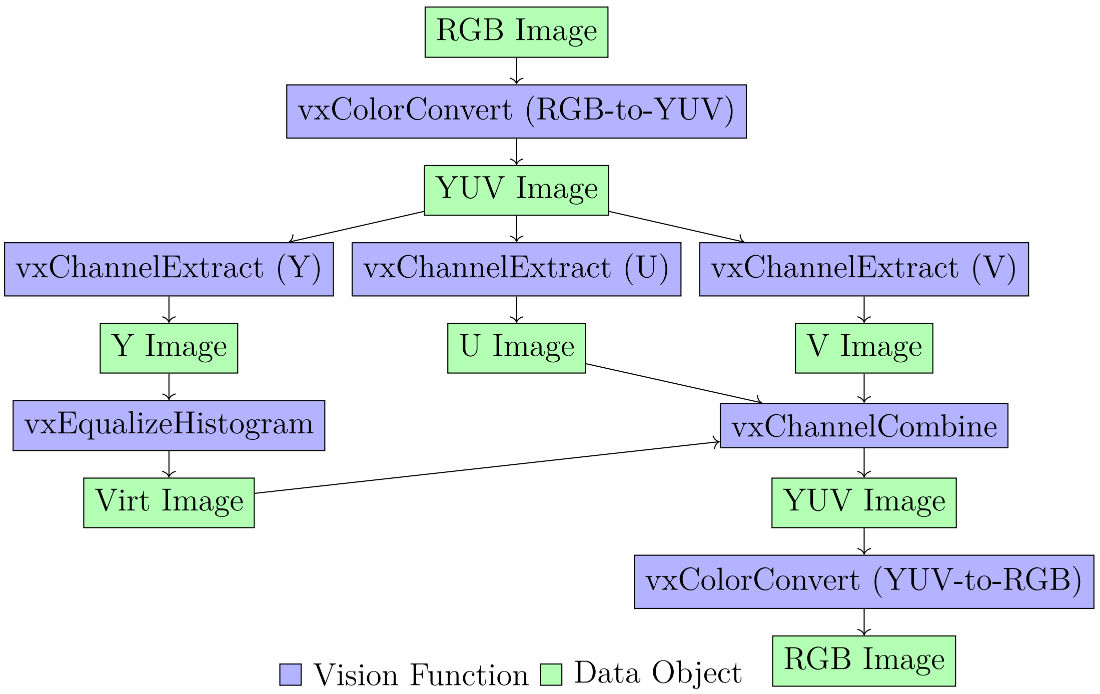


Figure 4.8: Automatic contrast algorithm graph

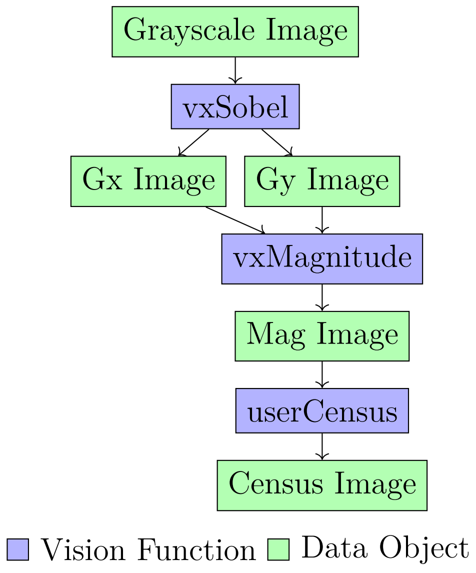


Figure 4.9: Census transform algorithm graph

## 4.2 A Computer Vision Library for FPGAs

This section presents and discusses a library for implementing various OpenVX vision components on FPGAs. This library can be used either directly by developers or as a target for code generation by high-level vision frameworks.

### 4.2.1 FPGA Technology and High-level Syntheis

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured by designers to implement different hardware designs after the manufacturing process. The majority of modern FPGAs use static on-chip RAM to implement programmable digital logic and interconnections. Although programmable logic can implement complex hardware, it is common for modern FPGAs to have hardened processor cores, digital signal processing (DSP) units, dedicated SRAM based memories, and high-performance transceivers. FPGAs offer excellent power and performance efficiency and are used successfully in various application domains such as scientific computing [14, 27, 117], computer vision [105, 60], and artificial intelligence [22].

Hardware Description Languages (HDLs) such as Verilog or VHDL are specialized programming languages that are used to describe concurrent hardware. FPGA Vendors usually provide the toolchain for converting HDL representation to FPGA hardware implementation (referred to as synthesis). HDLs describe systems at a low level, and with exponential growth in design size, it is becoming complicated and time consuming to use them in production. This has led to the popularity of design processes that interpret designs at a higher level of abstraction. Numerous tools have been proposed that use higher-level languages such as C [40, 102, 19], C++ [69], or embedded languages [68] to describe the hardware. They provide a higher level of abstraction to the programmer and postpone hardware compilation until the very end of the design process. Such tools usually convert the input design in a higher-level programming language to HDLs first and then use vendor-provided tools to synthesize the design.

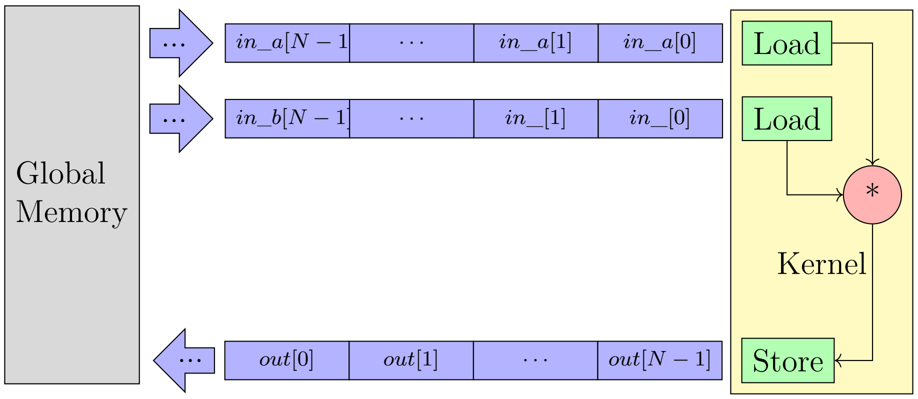
Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms [90]. Supported platforms include CPUs, hardware accelerators such as graphics processing units (GPUs), digital signal processors (DSPs), and FPGAs. OpenCL uses C99 to describe the program on the devices and provide a C++ API to manage the platforms. OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. OpenCL has received strong support from two major FPGA vendors as well. For instance, Intel offers OpenCL SDK, and Xilinx offers the SDAccel toolkit. A study found that using OpenCL to develop image processing compared to HDL design methods improves design productivity up to six times while achieving similar performance and resource usage. However, OpenCL designs incurred extra area overhead of (59% to 70% more logic) [44].

Figure 4.10a shows an example OpenCL kernel that multiplies two arrays element-wise and stores the results in the destination array. In this example, arrays are annotated with __*global* identifier to specify that they are allocated on FPGA off-chip memory. There is no implicit memory hierarchy such as caches on the FPGAs, and the designer must explicitly decide on the memory allocation. Usually, both off-chip and fast on-chip memories are available. However, since on-chip memory resources on FPGA are limited, larger data types such as images will be allocated on external DRAM. Figure 4.10b shows the generated FPGA hardware after synthesizing the sample kernel using the Intel OpenCL SDK. It consists of load and store modules and a floating-point multiplication unit. In this example, global memory is mapped to the FPGA external SDRAM. While external SDRAM has high latency, it is possible to stream consecutive data with no additional latency penalty in a pipelined fashion.

Loop pipelining is an effective technique to improve the throughput of loops synthesized on

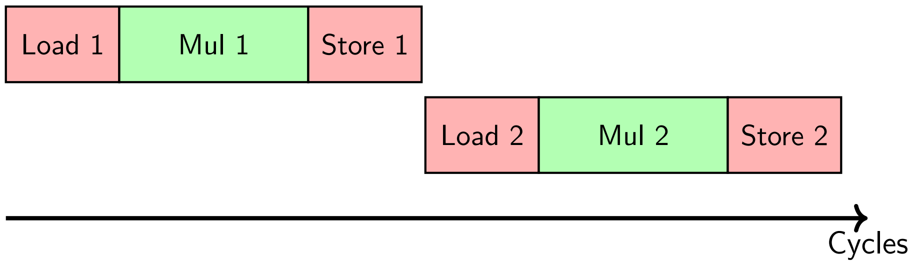
```
float mul(float a, float b) {
  return a * b;
}
__kernel void kernel_mul(
    __global const float* in_a,
    __global const float* in_b,
    __global float* out,
    const unsigned int N) {
  for(uint i=0; i < N; i++) {
    out[i] = mul(in_a[i], in_b[i]);
  }
}
```

(a) An OpenCL sample program

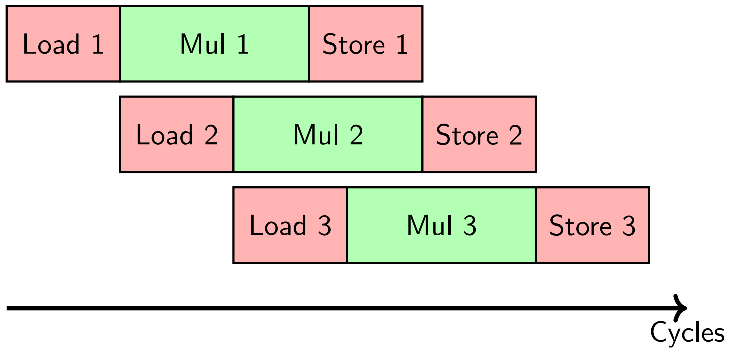


(b) Streaming contigous data between external SDRAM and FPGA

Figure 4.10: Synthesis of a sample OpenCL kernel



(a) Normal loop execution



(b) Pipelined loop execution

Figure 4.11: Effect of pipelinig in FPGA synthesis of loops

FPGAs by overlapping multiple invocations of the loop body. Figure 4.11 demonstrates the effect of loop pipelining on overall FPGA throughput. To fully benefit from this technique, however, all the operations of the loop body must be fully pipelined [24]. Computer vision workloads often involve processing large arrays of data (i.e., images). Hence, loop pipelining will be critical for having high throughput.

### 4.2.2 Library Design and Implementaion

This section discusses the OpenCL library that provides the basic blocks for building computer vision systems on FPGAs. We believe that with the comprehensive support and popularity of OpenCL in FPGA development, it is an excellent implementation choice. To make the library portable, We have tried to use vendor-specific functions as little as possible. However, during development, we used Intel FPGAs and their OpenCL SDK for evaluation of the library [11, 10].

We have designed the library with the following considerations in mind:

1. **Dataflow support**: FPGA designs that support stream processing or dataflow improve the computation throughput and save on on-chip memory usage.
2. **Configurability**: The ability to configure the vision components with different structural parameters such as parallelism and arithmetic precision is essential for exploring the design space and find various trade-offs in terms of performance, energy consumption, and resource usage.
3. **Genericity**: Several vision functions have similar data access patterns or apply to the same data type with varying bit-width. Hence, having multiple base templates that are highly optimized leads to better productivity and improves the maintainability of the library.
4. **Scalability**: Developed FPGA vision kernels can process pixels in parallel to improve the throughput. It is desired that the kernel resource usage grows linearly with the higher parallelism. We also desire the maximum FPGA working frequency not to drop significantly with higher parallelism since it affects the overall system performance.

The proposed library consists of the following elements: (1) compute kernels, (2) communication pipes, and (3) memory access kernels.

### Compute Kernels

Compute kernels implement various OpenVX vision functions. The list of supported vision functions provided by OpenVX is categorized in table 4.1. These categories comprise primitive functions such as *Pixel-wise, Fixed-rate Stencil, Multi-rate Stencil, Statistical, Geometric, and Table-lookup* and non-primitive functions that can be represented by a set of primitive functions. Figure 4.3 demonstrates the behavior of each category. Such classification is useful for two reasons: (1) It allows us only to implement a generic form for each category. Generic implementations are highly optimized and can be customized at compile time to represent specific vision functions. This categorization also facilitates extending the list with additional vision functions that are not part of the OpenVX specification. (2) Data access patterns can guide graph optimization, graph pipelining, and partitioning.

We have implemented several template classes depending on data access patterns of kernels. They can be customized with different vision functions, but many parts of the design can be reused. Vision function nodes operate on data streams specified by FPGA pipes. They can be connected to form deep pipelines on the FPGA. Parameters that can configure the behavior of kernels at compile-time include input/output data types (e.g., uchar, short, float), SIMD width to set the parallelism, numerical precision, and internal buffer size. Each kernel template also accepts an OpenCL function as the vision operator. Such functions can be developed by users to extend the OpenVX list of vision functions. Vision functions can be configured to operate on different values of SIMD width ranging from 1 to 32 (limited by hardware constraints such as communication bandwidth). By increasing SIMD width, kernels work on super-pixels, which are a collection of neighboring pixels.

Figures 4.13, 4.14, 4.15, and 4.16 demonstrate general hardware realization of different categories of vision function nodes. Pixel-wise kernels apply the vision function on streams of incoming pixels and produce one output pixel at each cycle. Stencil kernels use the line buffer implementation to process a stream of incoming data [39, 88]. To maximize data reuse, they use an internal buffer to store a window from the input image. This window includes multiple rows of the image that is needed to calculate the output pixel. For example, Figure 4.12 highlights the image pixels that required to calculate a 3x3 convolution from the current

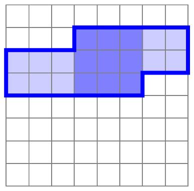


Figure 4.12: Using the line buffer for streaming image processing

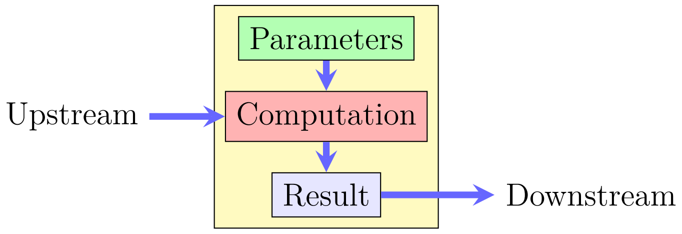


Figure 4.13: General implementation of pixel-wise kernels

window. Geometric kernels access pixels using an order that is different from the input order. Such order is usually calculated from a geometric transformation. Geometric kernels have no control over the order that upstream kernels produce data. Hence, to implement those kernels, incoming data needs to be stored in the memory first. Next, data is streamed from memory following the order specified by the geometric transformation. Statistical kernels need to process the whole input pixels to calculate a single scalar value. Hence, they need to process all the incoming pixels and update a scalar value. Their result is available once all input pixels are processed.

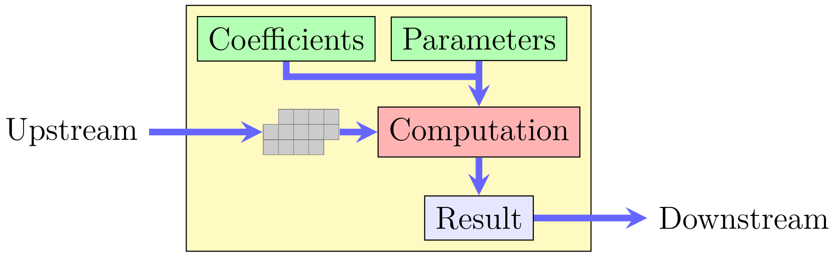


Figure 4.14: General implementation of stencil kernels

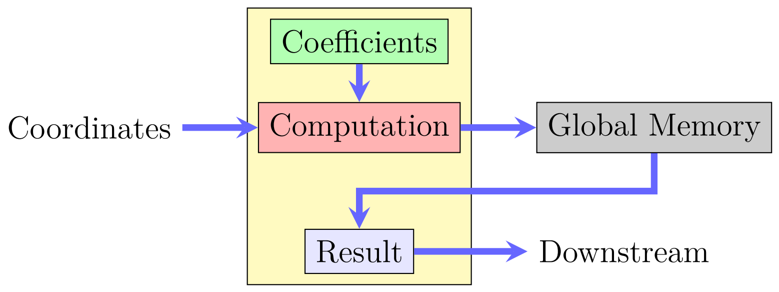


Figure 4.15: General implementation of gemotric transformation kernels

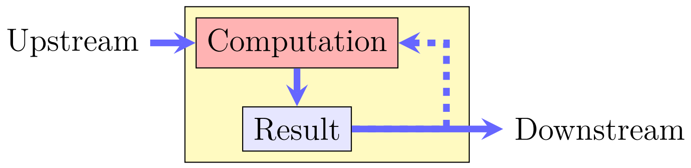


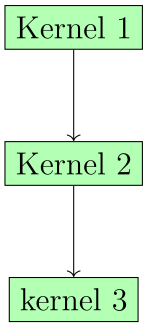
Figure 4.16: General implementation of statistical kernels

### FPGA Pipes

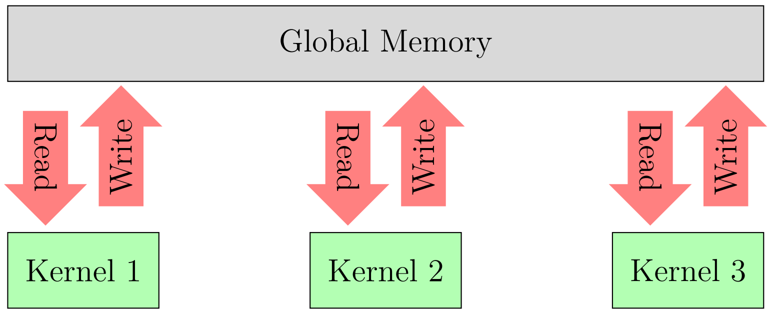
Compute kernels on FPGA communicate through pipes. Pipes are on-chip FIFOs that can be customized by their width (i.e., data size) and depth (i.e., number of elements). Pipes are also used for storing scalar values and small arrays for future references of the vision algorithm. As shown in Figure 4.17b, pipes are a more efficient alternative for inter kernel communication through FPGA global memory. However, they can only be realized between two endpoint kernels (i.e., a producer and a consumer). A special kernel is provided that replicates the input data to connect one kernel to multiple kernels.

### Host Pipes

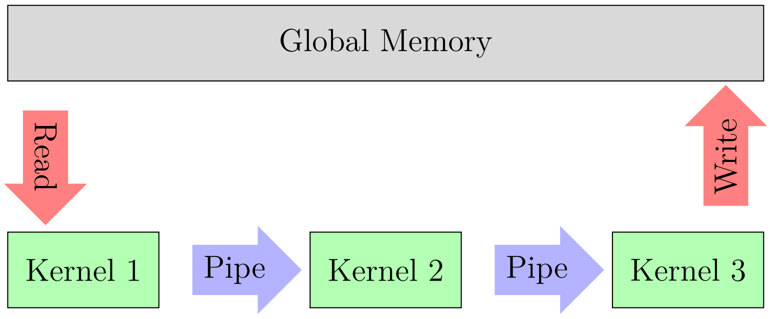
Host pipes establish low latency channels between the host and the FPGA. They bypass the FPGA global memory (i.e., DRAM) that is used in typical data transfer between the host and the FPGA [85]. Figure 4.18 demonstrates a comparison of two host-FPGA transfer mechanisms. Using host pipes allows simultaneous data transfer, and computation, hence better performance and energy efficiency can be obtained.



(a) Data dependancy graph

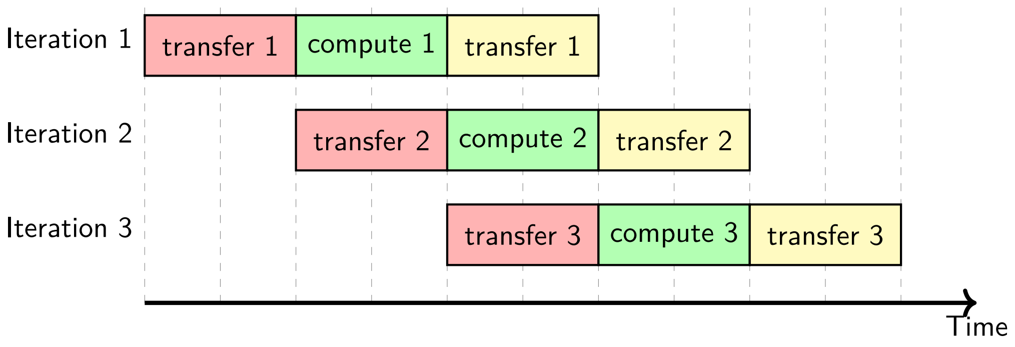


(b) Kernels communicate using global memory

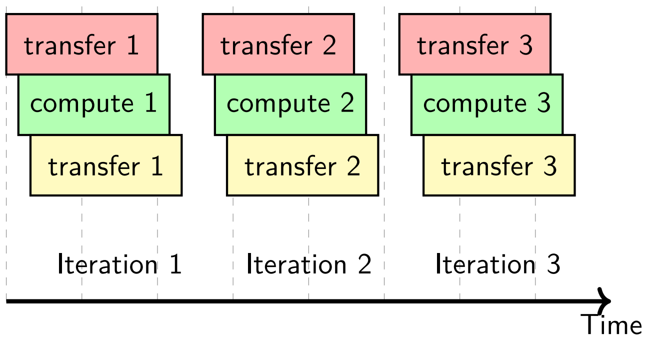


(c) Kernels communicate via on-chip pipes

Figure 4.17: Comparison of kernel communication mechanism. Using pipes reduces the number of accesses to global memory (external DRAM).



(a) Memory-based host-FPGA communication



(b) Pipe-based host-FPGA communication

Figure 4.18: Comparison of host FPGA transfer mechansims

**Global Memory (DRAM) Access**

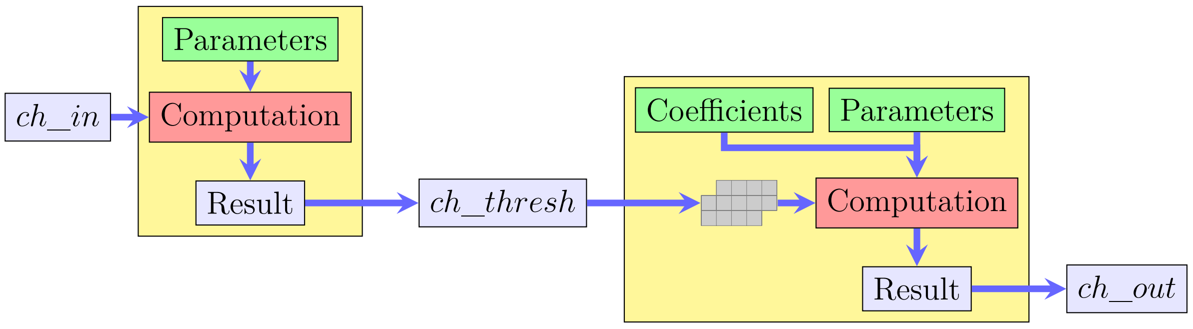
Even with dataflow processing, vision algorithms might require a large amount of data storage that needs to be stored in the FPGA external memory. For example, vision algorithms require to maintain a large state/model. Some algorithms are split into multiple FPGA partitions where partitions need to communicate via a large amount of intermediate data. We have defined two special kernels that can load to and store data from FPGA memory. The global memory address will be obtained at run-time and will be passed to these kernels as a kernel argument by the host program.

### 4.2.3 Programming Interface

We have defined a Domain Specific Language (DSL) embedded in OpenCL to describe computer vision pipelines on the FPGA. In this language, C-style macros are used to instantiate and specialize the generic vision functions and pipes, as well as establish the connection between them. This library can be incorporated in regular OpenCL programs. Figure 4.19 shows an example usage of the library to describe a simple pipeline.

```
#define SIMD_SZ 8
PIPE(p_in, uchar, SIMD_SZ, 0)
PIPE(p_thresh, uchar, SIMD_SZ, 0)
PIPE(p_out, uchar, SIMD_SZ, 0)
PIXEL_WISE(SIDM_SZ, threshold, p_in, p_thresh)
STENCIL(SIDM_SZ, filter, p_thresh, p_out)
```

(a) Example program



(b) Corresponding hardware

Figure 4.19: Describing a simple FPGA pipeline with the proposed library API

# Chapter 5

# A Framework for FPGA Acceleration of Computer Vision Algorithms

Implementing software that is portable in regards to performance and power efficiency is very challenging. Since many of the optimizations are specific to the underlying hardware, most software implementations are optimized manually for each particular hardware platform. The manual optimization of programs for each hardware platform is a time-consuming and complicated process. Several works have proposed the idea of decoupling algorithm specification and implementation. This way, these two tasks can be done separately and even by different specialists. They often define a custom Domain-Specific Language (DSL) for algorithm specification that imposes several constraints from the application domain. Such specifications can be later translated into the hardware implementation using an optimizing compiler.

Halide is a DSL embedded in C++ to describe image processing pipelines, designed with a focus on computational photography [80, 77]. Halide defines a systematic model based on stencil pipelines to explore the trade-off between locality, exploitation of parallelism, and redundant re-computation. Halide, however, does not support vision kernels with complex data access patterns. It also requires the scheduling policy and tile size to be explicitly specified by the user. Rigel [78] extends Halide language by supporting more advanced data access patterns, including multi-rate kernels and image pyramids. However, it still does not support statistical and geometric kernels. In addition, it is primarily designed to describe image processing algorithms on FPGAs, and there is no CPU support. PolyMage [66, 82] converts an image processing algorithm developed using a custom DSL into a parallel implementation. It makes use of a polyhedral compiler to optimize image processing applications using tiling, and fusion of image processing stages and memory allocation. Data-dependent operations, such as statistical and table lookup operations, and also computations that have a considerable input data reuse (e.g., matrix multiplication) are out of the PolyMage's scope of polyhedral overlapped tiling analysis.

All these frameworks may require manual optimizations to exploit locality and parallelism, which require much time, effort, and expertise [37]. Besides, they lack hardware-software co-optimization that is necessary for the efficiency of heterogeneous systems. In this chapter, we present our framework for FPGA accelerating of vision algorithms that are described using a high-level and graph-based specification based on OpenVX. Such high-level graphs abstract away many implementation details such as memory allocation and hardware parallelism. We show how such a representation can be translated into heterogeneous implementation consists of CPU and FPGA. Towards this end, we provide an automatic acceleration framework for FPGA implementation of OpenVX graph pipelines. To achieve this goal, we integrate computer vision domain knowledge, hardware/software optimization techniques, and high-level synthesis methodologies. We have developed primitive computer vision processing elements using efficient and customizable OpenCL components that can be configured and connected to form a larger pipeline on the FPGA. Our framework applies several high- and low- level optimization methods to the vision graph and employs the developed OpenCL library to generate an efficient implementation.

Few prior works [70, 71, 96] develop solutions for FPGA implementation of OpenVX graphs. However, these solutions support only a limited set of OpenVX kernels and are not able to provide efficient implementations for more complex kernels. They also suffer from a lack of automation, configurability, and hardware/software co-optimizations. In essence, none of the previous works were able to provide a general and automatic framework to convert high-level algorithms to CPU-FPGA heterogeneous acceleration platforms.

In summary, we make the following contributions:

- We suggest high-level graph-based and low-level FPGA-specific optimizations for vision algorithms that are defined by OpenVX spec.
- We develop a framework for the automatic translation of algorithm graphs into optimized FPGA-CPU implementation. This framework enables vision developers to develop, verify, and test vision algorithm quickly without the knowledge on a DSL or any software/hardware description languages.

In this chapter, we suggest and describe a process for translating high-level OpenVX graphs to optimized implementations for heterogeneous platforms consists of CPUs and FPGA accelerators. In this process, the input graph is partitioned into CPU and FPGA partitions. Also, various optimizing transformations are applied to the input graph. We have categorized these transformations into high-level hardware-agnostic and low-level hardware-specific transformations and implemented them as part of a framework called AFFIX.

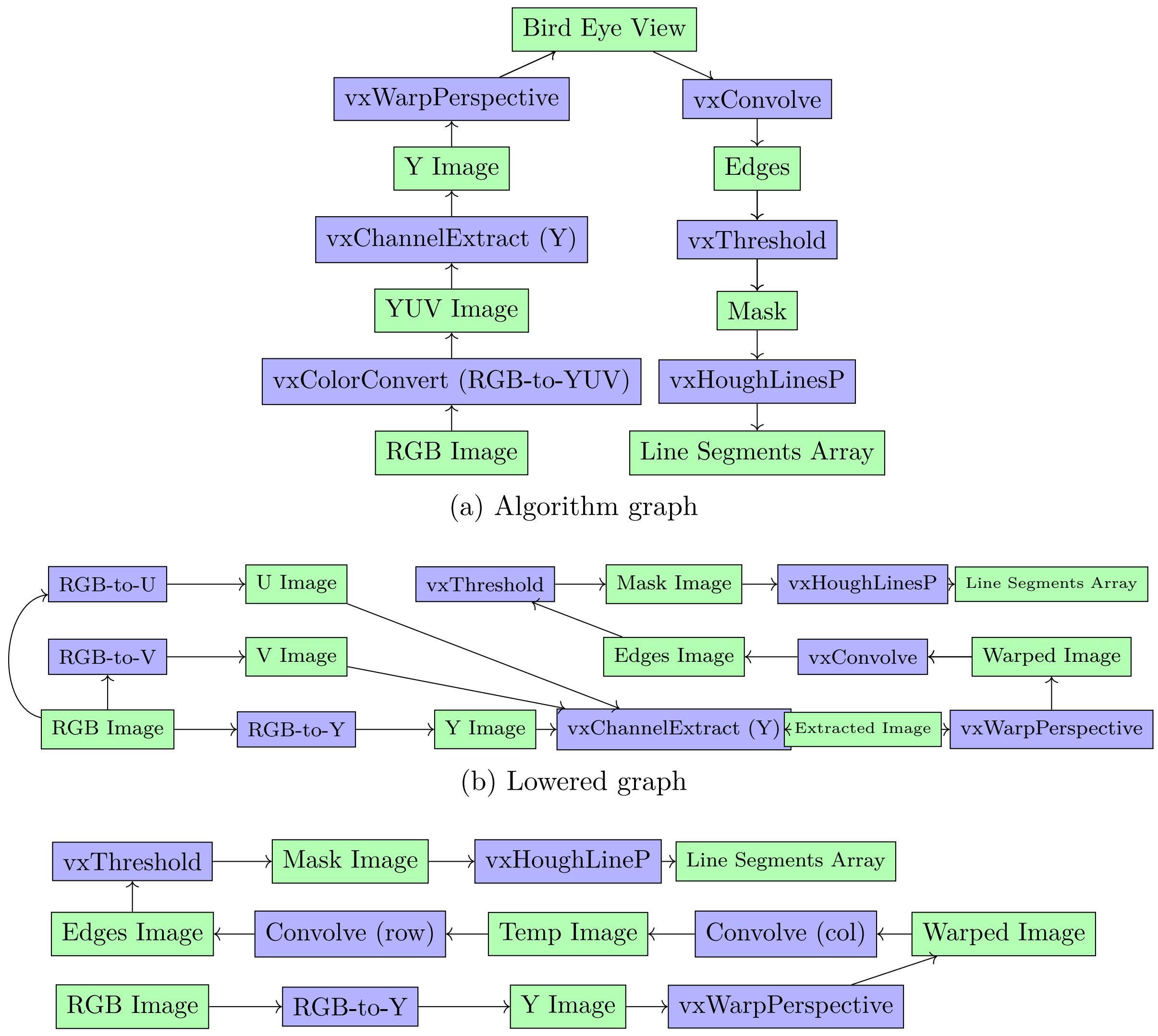
## 5.1 High-Level Optimizing Transformations

These transformations prune and simplify the input graph to achieve a more efficient implementation. These techniques are hardware-agnostic and do not consider information about the specific target FPGAs. To enable these transformations, we have annotated the vision function nodes with information such as data access patterns and their input/output data types. Currently, AFFIX includes the following list of high-level transformations:

- Lowering: This step transforms the input graph by replacing the algorithm's non-primitive kernels (if any) with primitive kernels. Moreover, it replaces primitive kernels that have multiple outputs with multiple single-output nodes. Graph lowering makes the analysis for applying optimizations simpler for AFFIX. Figure 5.1b shows the transformed LD algorithm after applying the lowering step. For instance, in the case of LD algorithm, the non-primitive *vxColorConvertNode* kernel is replaced with three primitive kernels *RGB-to-Y*, *RGB-to-U*, and *RGB-to-V*.

- Dead Node Elimination: This step eliminates graph nodes that are not connected to an output node. Figure 5.1c demonstrates applying this step to the lowered LD algorithm. Since *ChannelExtract* node, only extracts Y Image, there will be no path from *RGB-to-U* and RGB-to-V nodes to the output. Therefore, AFFIX recognizes them as dead nodes and eliminates them from the graph.

- Separable and Symmetric Filter Implementation: Several vision functions involve convolution. If the coefficient matrix is separable, convolution can be implemented using two simpler convolutions [57]. For instance, kernels such as *Gaussian Blur* have N*N matrix convolutions that can be replaced with two convolutions with 1*N and N*1 coefficient matrices. Table 5.1 shows the logic resource savings percentage by using separable filter implementation of the Gaussian filter compared to the non-separable baseline version. By increasing the filter size, the amount of saving increases as well. The reason is that separable filter involves $2 \times n$ Multiply-And-Accumulate (MAC) operations, while the baseline version with non-separable implementation demands $n^2$ MAC operations.

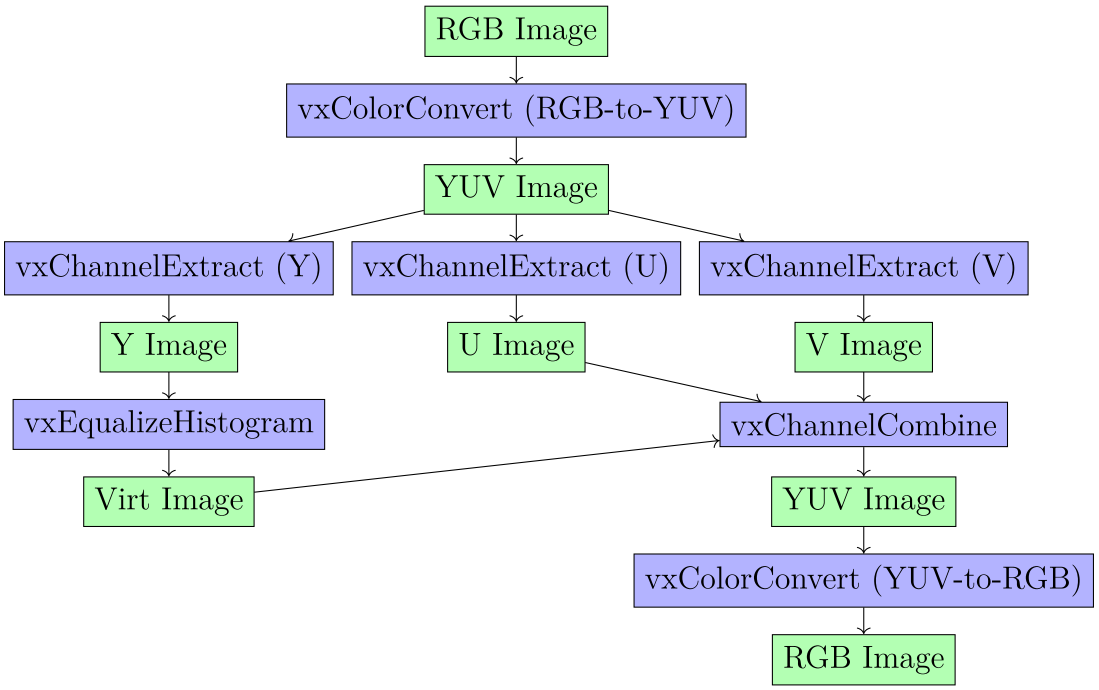


(a) Algorithm graph

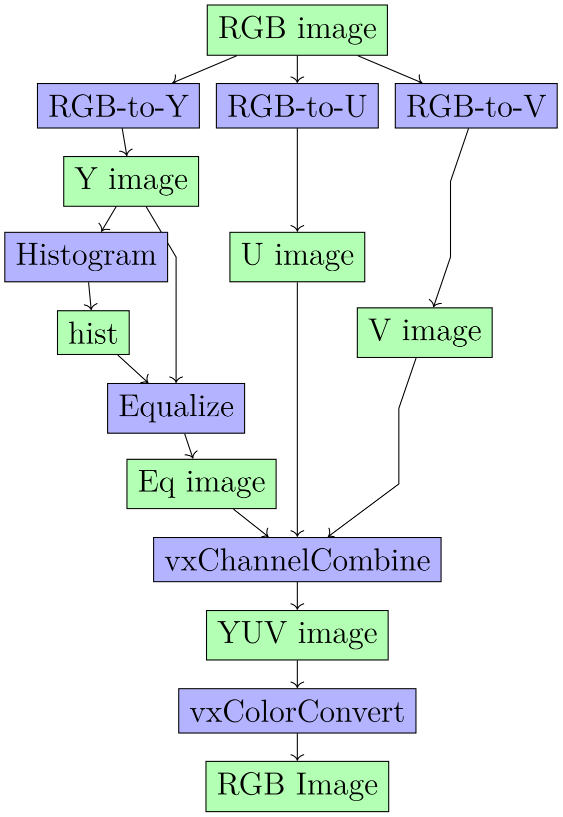
(b) Lowered graph

(c) Optimized graph

Figure 5.1: High level optimization of lane detection algorithm



(a) Algorithm graph



(b) Lowered graph

Figure 5.2: High level optimization of Automatic contrast algorithm

Figures 5.1 and 5.2 demonstrate the effect of high-level transformations on two sample algorithm graphs.

## 5.2 Pipelining using Graph Partitioning

Pipelining the algorithm graph on FPGA is the main technique to increase throughput. However, realizing all graph nodes on a single FPGA pipeline is not always feasible.

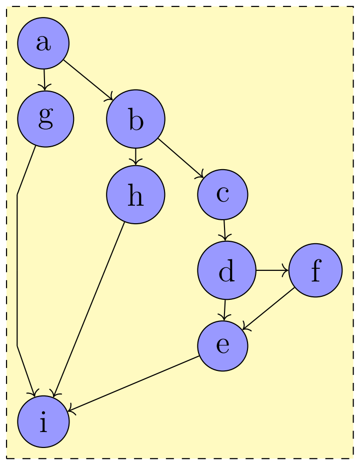
- Not all vision kernels can be implemented efficiently on FPGA. We have identified such kernels in the current version of the framework. If the input DAG contains one (or more) of these kernels, AFFIX will map them to the CPU for execution. For instance, AFFIX already knows *HoughLinesP* kernel in LD algorithm should be run on CPU (Figure 5.4). Although it might be possible to implement some of these kernels on FPGA using specific heuristics, doing that makes AFFIX very complex.
- Not all kernels that are mapped to the FPGA can be part of the same pipeline. While, graphs that only consist of *pixel-wise*, *stencil*, and *table lookup* kernels can be implemented in a single pipeline, *statistical* and *geometric* kernels have data access dependencies that limit the put constraints on pipelining opportunities. In this case, The pipeline partitioning divides the primitive DAG obtained after high-level optimization into sub-partitions. A different pipeline implements each partition.

  1. Statistical nodes cannot be mapped to the same pipeline with their successors. The reason is that these kernels have to process the whole input before generating a valid output(e.g., mean() function). Hence, all of their downstream kernels must be implemented in different pipelines. In such cases, AFFIX will split the graph into multiple partitions. Each partition is implemented as a separate pipeline.
  2. Geometric nodes cannot be in the same pipeline as their predecessors. The reason

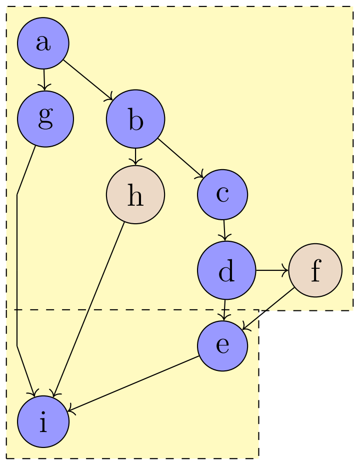
is that inputs of geometric kernels cannot be streamed directly from the upstream kernel. This is because geometric kernels must access data in an order that is specified by a transformation matrix which is most likely different than the input stream order. Thus, the input stream must be entirely saved in memory first. Geometric nodes then can load the saved data, and hence they must be the entry points of a new partition.

Figure 5.3 demonstrates partitioning of sample DAGs with various kernel types. DAG in the case (a) only consists of pixel-wise, stencil, and table lookup kernels. Thus, the whole of the DAG can be in the pipeline. Case (b) uses two statistical nodes (nodes h and f). Hence, their successor nods (nodes e and i) must be implemented as a separate pipeline. Case (c) uses two geometric transformation nodes (nodes h and f). Therefore, they cannot be in the same partition as their predecessors. Case (d) uses two CPU nodes. CPU nodes cannot be part of any FPGA pipeline. Furthermore, their no predecessors and successors can not map to the same FPGA partitions.

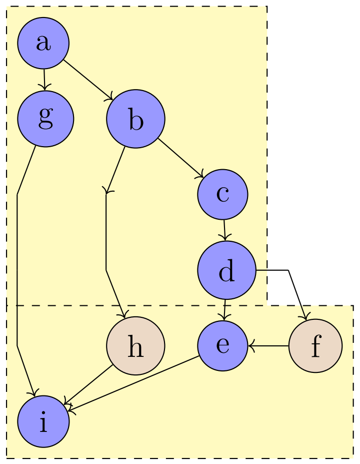
Data communication between two FPGA partitions is done through using FPGA internal and external memory while FPGA and CPU partitions communicate through host pipes. Given the mentioned partitioning constraints for a graph, there may be several valid partitioning schemes. Since we have found the communication to be the major performance bottleneck, we formulated the partitioning problem as a Mixed Integer Linear Program (MILP) and aimed to find a partitioning that minimizes data communication across partitions as described by Eq. 5.1. In this formulation, $V$ is the set of DAG nodes and *E(u,v)* specifies data transfer size between two nodes $u$ and $v$. *conflict[u][v]* is a table of Boolean values that specifies whether each pair of vertices in $V$ cannot be mapped to the same partition (True case). The objective is to minimize equation 5.1 subjects to mapping and partitioning constraints. $P_v$ in (2) is a variable that represents the partition number (a positive integer) that node $v$ is mapped to. $Y_{uv}$ is a Boolean variable which its true value indicates if two nodes u and v are



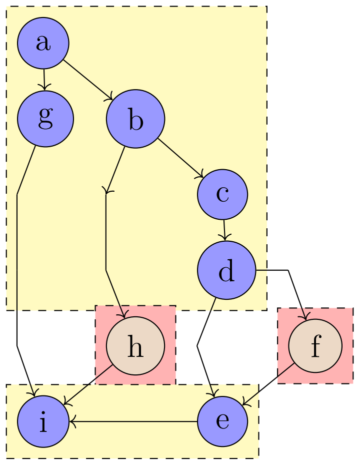
a) Graph with only pixel-wise stencil, and lookup nodes



b) Graph with statistical nodes (h and f)



c) Graph with geometric nodes (h and f)



d) Graph with CPU nodes (h and f)

🟨 FPGA Partition 🟥 CPU Partition

Figure 5.3: Partitioning of a sample algorithm graph with different vision node types composition

mapped to different partitions. Constraint (4) sets the value of $Y_{uv}$ to be 1 if $P_u! = P_v$ and constraint (5) enforces mapping of incompatible nodes to different partitions. To formulate constraint (4) in canonical MILP form, we used the well-known Big-M method [106]. In this method, M is an integer that must be larger than $|P_u - P_v|$. We picked M to be equal to the number of nodes in the graph.

$$Minimize \sum_{v \in V} \sum_{u \in V} E(u,v) \times y_{uv} \tag{5.1}$$

subject to:

$$\forall v \in V : |V| > P_v > 0 \tag{5.2}$$

$$\forall u, v \in V : Y_{u,v} \in \{0, 1\} \tag{5.3}$$

$$\forall v, u \in V : Y_{uv} = 1 \; if \; P_u! = P_v \; else \; 0 \tag{5.4}$$

$$\forall v, u \in V : Y_{uv} == conflict[u][v] \tag{5.5}$$

Figure 5.4 shows the partitioning of the graph of the LD algorithm after high-level optimizations. The algorithm graph is partitioned into three partitions: *HoughLinesP* function is implemented in a CPU partition. The rest of the graph is mapped to the FPGA. However,

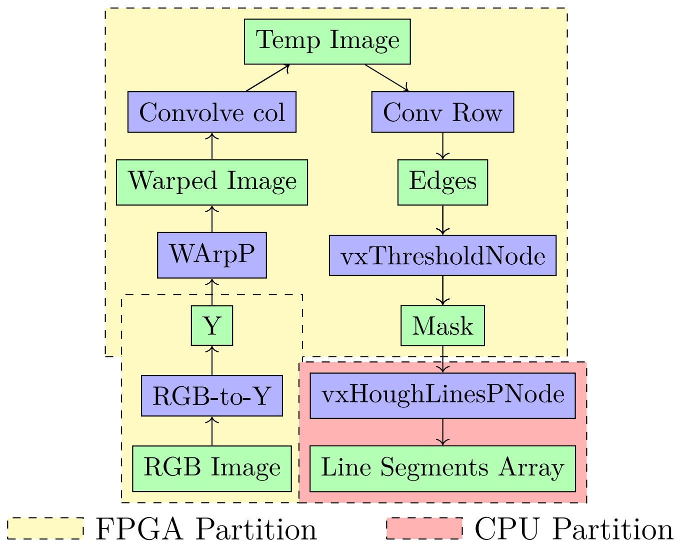


Figure 5.4: Partitioning of the lane detection algorithm graph

since *WarpPerspecitve* is a geometric function, the graph is segmented into one or more partitions. As another example, Figure 5.5 demonstrates partitioning of Automatic contrast algorithm. Since this algorithm uses histogram information (has one statistical node), its graph is partitioned into two pipelines on the FPGA.

### 5.2.1 Data Streaming Patterns

AFFIX instantiates vision functions and pipes to form pipelines on the FPGA. Since the data streams through the pipeline, all the vision functions on the pipeline must be able to process the input in the same order that input data is arriving. Pixel-wise, memory lookup, and statistical nodes can process images with any arbitrary streaming order (i.e., cases a Figure 5.6 ). However, stencil vision function nodes use the sliding window implementation and can only process data streamed in raster order.

The sliding window implementation requires storing multiple rows of the input image around the current pixel. When processing large input data sets, storing multiple rows of the input image per every kernel leads to excessive usage of on-chip memory resources. One method

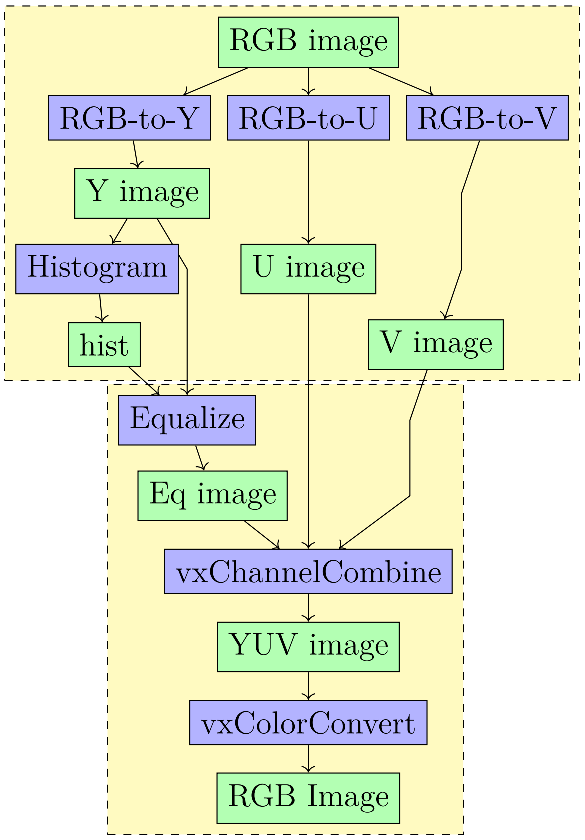


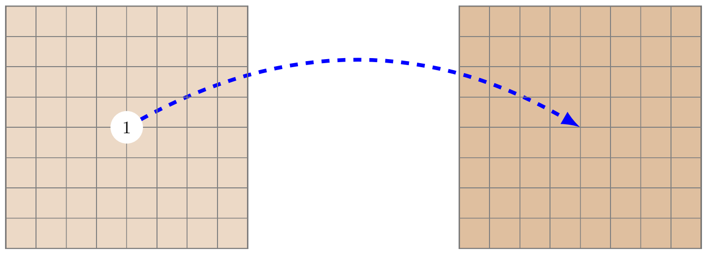
Figure 5.5: Partitioning of automatic contrast algorithm graph

to save on on-chip memory resource usage is to partition the input data into sub-images with smaller width size (i.e., cases b Figure 5.6). This way, the sliding window size can be shrunk accordingly. However, since pixels at sub-image boundaries are missing, sub-image processing must overlap with each other to cover the missing pixels, which leads to re-computation. Hence a balance between area-usage and re-computation must be considered. While both case b and c can work on stencil kernel and they have equal memory requirement, case b requires less re-computation since boundary area is smaller. In algorithms that include up-sampling and down-sampling nodes, the window size of kernels that operates on down/upsampled images must be adjusted accordingly. For example, if the input data has a resolution of W*H, after going through down-sampling with the scale of ”1/2”, it will be W/2*H/2.

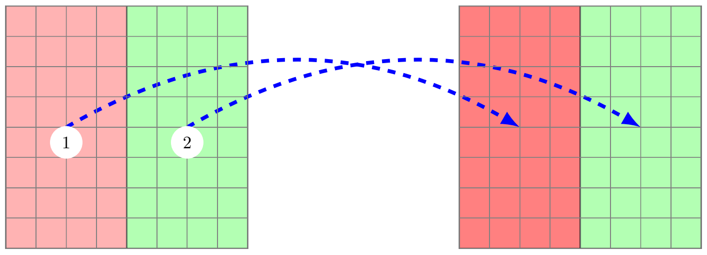
## 5.3 FPGA-Specific Optimizing Transformations

AFFIX splits the graph that is obtained after applying high-level optimizations into CPU and FPGA partitions. It uses OpenCL to implement the FPGA partitions. It applies several low-level optimizations on OpenCL implementation of the vision components through HLS pragmas. OpenCL compilers usually support pragma directives to guide optimization and hardware generation such as loop unrolling, kernel optimization, interface generation, and memory optimization. The primary goals of low-level optimization techniques are to enhance overall performance(FMAX), achieve ”ideal pipelines” (i.e., pipelines with Initiation Interval (II) of 1”) for loops with a large number of iterations, and save FPGA resource usages without significant performance degradation. These optimization methods include deciding on loop unrolling factors, SIMD-width, local memory configuration (i.e., port numbers ), pipe depth and width, and sliding window size.

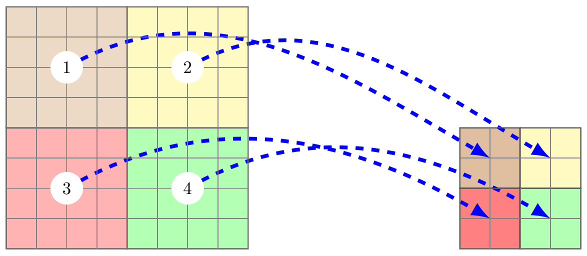
- FPGA Pipes Depth Optimization: To prevent deadlock, and save on FPGA resources,



a) Whole image



b) Columns



c) tiles (Downsampling)

Figure 5.6: Different policies to partition and stream the images. Cases b and c assume sliding window implementation for images with 4 columns. To stream all the cases raster ordering is used.

| **Filter Size** | **5x5** | **7x7** | **9x9** | **11x11** | **13x13** |
|---|---|---|---|---|---|
| Logic Saving (%) | 5 | 16 | 26 | 36 | 44 |

Table 5.1: Logic saving of separable implementation of 2D filters on Arria10 compared to non-separable baseline implementation

pipe depth must be optimized. AFFIX calculates optimal pipe depth by matching the latency of input ports of all nodes in the DAG. Latency is a function of two factors. It depends on the kernel compute function. The depth of the pipelining for each kernel depends on the function units. It also depends on the kernel type. Pixel-wise functions have additional latency of zero, while stencil functions need to fill up the sliding window first and have variable latency that depends on the window size. AFFIX adjusts the optimal pipe depth by examining the RTL implementation of the design generated by the OpenCL compiler. After extracting it, it re-compiles the design with the proper pipes' depth set.

- Sliding Window Size Optimization: Stencil kernels use sliding window implementation to support data streaming. Sliding windows must be large enough to store "K" rows of the input image (for a (2k by 2K) convolution). However, a large window size incurs a performance/area overhead. The input image will be partitioned according to the calculated window size.

## 5.4 Automatic Flow

We propose the flow depicted by Figure 5.9 for converting high-level algorithms into hardware-software implementation.

As depicted in Figure 5.9, it consists of three phases to verify, analyze, and generate FPGA and CPU implementations. Users must describe the vision algorithm in a DAG format using

```
#define SIMD_SZ 8
#define WIN_SZ 320

// Partition 1
PIPE(p_in, uint, SIMD_SZ, 0)
PIPE(p_y, uchar, SIMD_SZ, 0)
SRC(p_in)
RGBTOY(SIMD_SZ, p_in, p_y)
SAVE(p_y)

// Partition 2
PIPE(p_warped, uchar, SIMD_SZ, 0)
PIPE(p_conv_row, uchar, SIMD_SZ, 0)
PIPE(p_con_col, uchar, SIMD_SZ, 0)
PIPE(p_thresh, uchar, SIMD_SZ, 0)
float[9] conv_col = {...};
float[3] conv_row = {...};
WARP_LOAD(p_warped, SIMD_SZ)
CONV_ROW(p_warped, p_conv_row, 9, conv_row, ...)
CONV_COL(p_conv1, p_con_col, 3, conv_col, ...)
THRESH(p_conv_col, SIMD_SZ, thresh_val, p_thresh)
SINK(p_thresh)
```

Figure 5.7: Simplified OpenCL code for FPGA implementation of the lane detection algorithm example

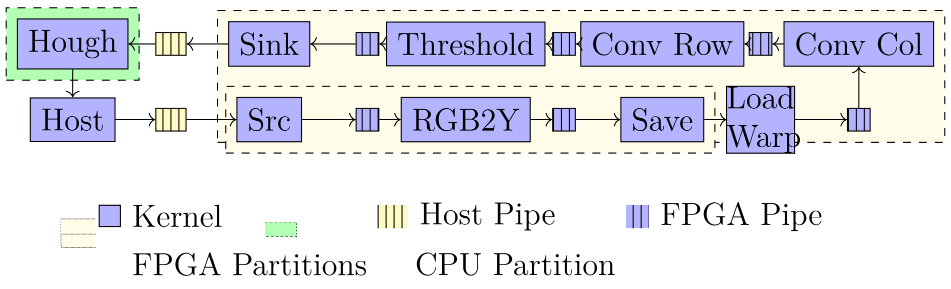


Figure 5.8: Visualization of the generated system for lane detection algorithm graph

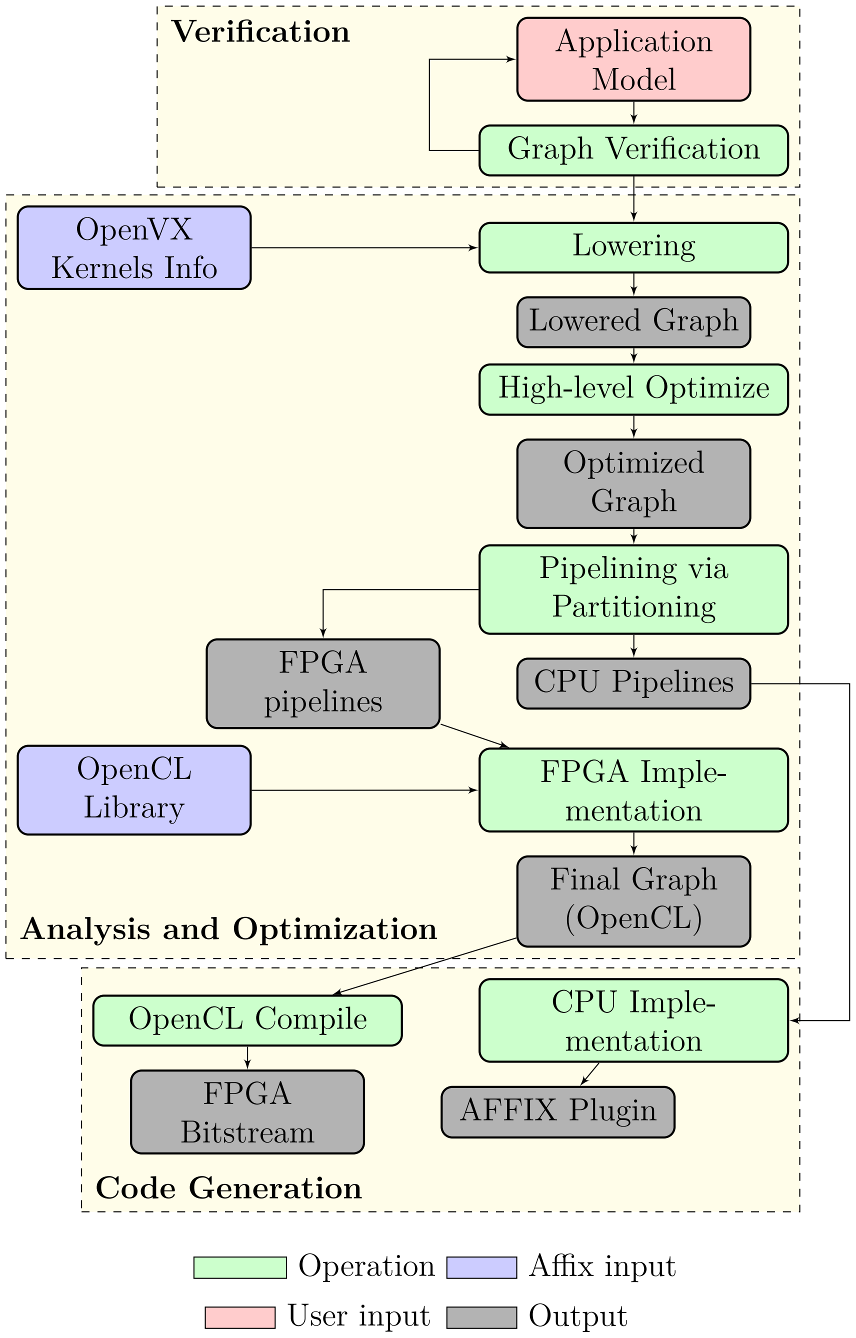


Figure 5.9: AFFIX framework flow

the AFFIX provided API. Additionally, users can specify the SIMD-size as a parallelism parameter, which is helpful for design space exploration. It starts with verification of the user input algorithm followed by lowering the input DAG. The output of this step is a graph that only consists of primitive vision functions that are easier to analyze and optimize. Then, AFFIX applies high-level optimizations on the primitive graph resulting in a more efficient algorithm representation. Next, AFFIX splits the graph into CPU and FPGA partitions based on node availability and data dependency constraints. AFFIX aims to find the largest FPGA partitions that can be implemented as a single pipeline on the FPGA. Finally, FPGA-specific (low-level) optimization techniques will be applied across FPGA partitions to obtain the final optimized OpenCL code. In cases which there are multiple partitions, partitions are sorted based on the topological ordering and executed consecutively.[1] The third phase generates both FPGA hardware and CPU software components of the system. FPGA components are implemented by specializing in the constructs provided via an OpenCL library. Most of the kernel parameters are either inferred from the provided DAG (e.g., input dimension, vision operator, matrix coefficients) or determined automatically using high and low-level optimization techniques (e.g., buffer size). The only parameter that has to be set by a user is the SIMD width to adjust the required performance. OpenCL compiler is used to translate the generated OpenCL program into the FPGA bitstream. In addition to a computational pipeline, the final hardware incorporates a standard PCI interface to the host, OpenCL controller logic, and an optional FPGA DRAM interface.

## 5.5 Usage and Software Architecture

We have developed a host program that exploits OpenCL run-time API to allocate data on FPGA DRAM, pass kernel arguments, and coordinate execution of OpenCL kernels on FPGA. Software components are compiled separately as plugins (e.g., plugin.so) and are

[1]More optimized scheduling and execution are left for future work.

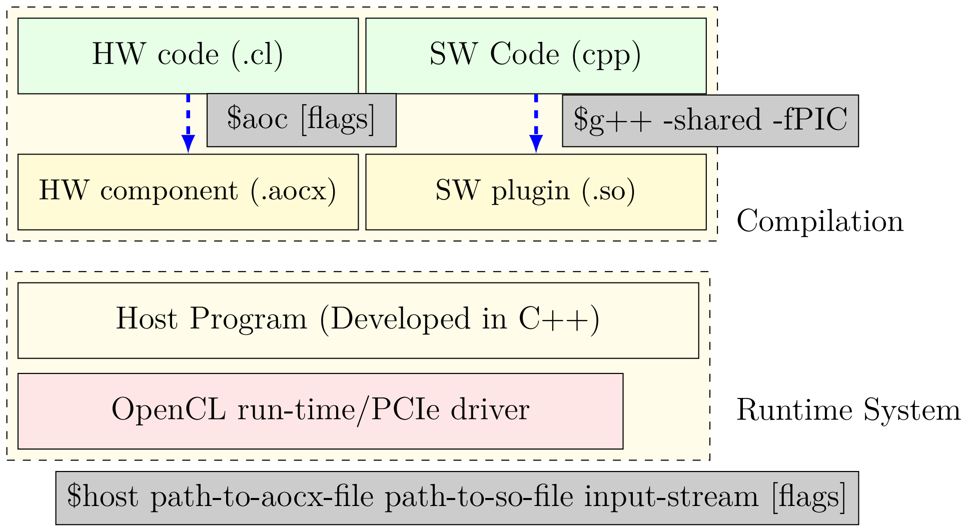

HW code (.cl)
SW Code (cpp)
$aoc [flags]
$g++ -shared -fPIC
HW component (.aocx)
SW plugin (.so)
Compilation
Host Program (Developed in C++)
OpenCL run-time/PCIe driver
Runtime System
$host path-to-aocx-file path-to-so-file input-stream [flags]

Figure 5.10: AFFIX software architecture and components

loaded at run-time by the host program. They implement an interface that allows the host program to invoke their vision functions. AFFIX makes use of a fast OpenCL emulator provided by Intel OpenCL SDK that allows us to emulate the whole algorithm on CPU for verification purposes. AFFIX also supports compiling OpenCL-based vision kernels with profiling enabled that measures and reports performance of kernel execution on the FPGA.

### 5.5.1 User-Defined Kernels

AFFIX supports user-defined kernels. However, users must provide CPU and FPGA implementation for the new vision functions. Since we have already integrated AFFIX with an open-source computer vision library (i.e., OpenCV), access to a wide variety of functions for developing user-defined kernels is available.

## 5.6 Evaluation

We have developed several vision algorithms using the AFFIX framework and measured their performance numbers, including total execution time, average power consumption, and resource usage of the FPGA components. The characterization of algorithms is demonstrated in Table 5.2. To develop some of the algorithms such as Census and ColorCopy, we have to extend the OpenVX vision functions list by implementing new functions.

| **Benchmark** | **Domain** | **No Function Nodes** | **No Extension vision Functions** | **No CPU nodes** | **No Geo Nodes** | **No Stats Nodes** | **No Graph Partitions** |
|---|---|---|---|---|---|---|---|
| Automatic Contrast | Image Processing | 6 | 0 | 0 | 0 | 1 | 2 |
| Lane Detection | Image Processing | 6 | 0 | 0 | 1 | 0 | 3 |
| Color Copy | Color Printing | 42 | 4 | 1 | 0 | 0 | 3 |
| Census Transform | Visual Descriptors | 4 | 1 | 0 | 0 | 0 | 1 |
| SIFT keypoints | Visual Descriptors | 116 | 2 | 0 | 0 | 0 | 1 |

Table 5.2: Workload characterization

**Automatic contrast** adjusts the contrast of the input color (RGB) image based on the histogram of pixel intensity values.

**Lane detection** algorithm finds lane segments in the input color image. It uses perspective transformation to focus on ”bird-eye-view” of the grayscale image and searches for line segments using probabilistic Hough transformation.

**Color copy** prepares and enhances the input color image for color printing. It involves color conversion between different color spaces such as LAB and CMYK, several image processing operations, and half-toning algorithm based on error diffusion. Color conversions are done using large 3-D lookup tables. Tetrahedral interpolation of color spaces involves floating-point calculations that can be implemented using FPGA DSPs [54]. Error diffusion algorithm [89] has a non-supported data access pattern and is executed on the CPU. The rest of the graph is implemented in two partitions (pipeline stages) on the FPGA.

**Census transform** calculates visual descriptors based on census transform for an input grayscale image [111].

**SIFT Keypoints** searches for image's representative key points in multiple scales of the Gaussian image pyramid (We set pyramid levels to five) [61]. The candidate keypoints then can be associated with descriptor vectors, which are useful for pattern recognition but are excluded in our implementation. This algorithm is implemented in a single pipeline on the FPGA. The output of this algorithm will be a list of detected keypoints (coordinates, detection scale).

The CPU used in experiments was an Intel Core i7-4770 processor with 8 logical cores. Arria10 GX FPGA reference board with 1150K logic units, 1500 DSPs, and 2GB DDR4 attached external memory with Gen3x8 PCIe interface is used as the accelerator. PCIe communication is full-duplex (i.e., simultaneous read/write). By streaming the data between the host and the FPGA, communication, and computation on the FPGA overlap.

We have optimized the input algorithms with our framework and synthesized the output OpenCL code using Intel OpenCL SDK 19.1 with different values for parallelism (i.e., $SIMD\_width$). FPGA board has a PCIe interface of 32 bytes which limits the maximum parallelism level. Hence maximum SIMD_SIZE is 8 for algorithms that operate on color images (RGBX) and 32 for those working with grayscale images.

Table 5.3 reports maximum throughput of the optimized CPU version (using eight cores and vectorized with AVX intrinsics) and heterogeneous implementations of the algorithms produced via the framework with varying degree of parallelism. Reported numbers include the time for both communication and computation. As it is shown, throughput scales proportionally with parallelism level until communication bandwidth between CPU/FPGA reaches maximum PCIe limit. It can be seen that AFFIX implementations can achieve better speed-up compared to CPU implemenation for larger graphs (e.g., SIFT and ColorCopy).

| Benchmark | CPU (AVX+8 Cores) | FPGA SIMD=1 | FPGA SIMD=2 | FPGA SIMD=4 | FPGA SIMD=8 | FPGA SIMD=16 | FPGA SIMD=32 |
|---|---|---|---|---|---|---|---|
| Census transform | 691 | 307 | 592 | 1148 | 2073 | 2764 | 2764 |
| Automatic contrast | 1579 | 394 | 737 | 1508 | 2552 | N/A | N/A |
| Lane detection | 721 | 582 | 1228 | 2369 | 2764 | N/A | N/A |
| Color copy | 399 | 737 | 1037 | 1442 | 1746 | N/A | N/A |
| SIFT keypoints | 37 | 230 | 307 | 592 | 829 | 829 | N/A |

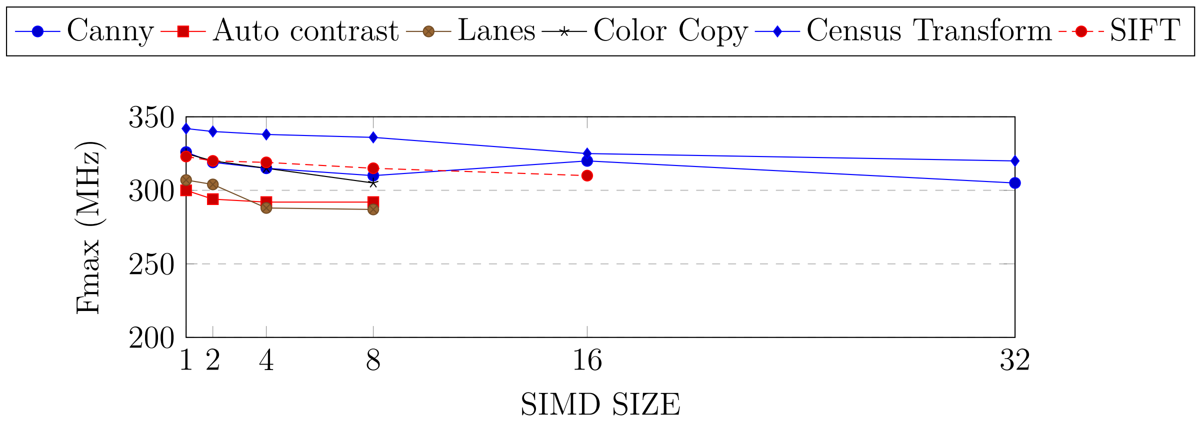
Table 5.3: Throughput comparison of CPU and Arria10 accelerated algorithms with different SIMD width

| Benchmark | CPU (AVX+8 Cores) | FPGA SIMD=1 | FPGA SIMD=2 | FPGA SIMD=4 | FPGA SIMD=8 | FPGA SIMD=16 | FPGA SIMD=32 |
|---|---|---|---|---|---|---|---|
| Census transform | 16.1 | 6.9 | 13.3 | 26.4 | 46 | 60.8 | 59.7 |
| Automatic contrast* | 30.9 | 8.6 | 15.9 | 32.4 | 54.2 | N/A | N/A |
| Lane detection* | 25.1 | 13 | 27.2 | 52.2 | 58.6 | N/A | N/A |
| Color copy | 7.7 | 15.7 | 22 | 30 | 35.4 | N/A | N/A |
| SIFT keypoints | 1.3 | 3.1 | 6.2 | 11.8 | 15.8 | 14.3 | N/A |

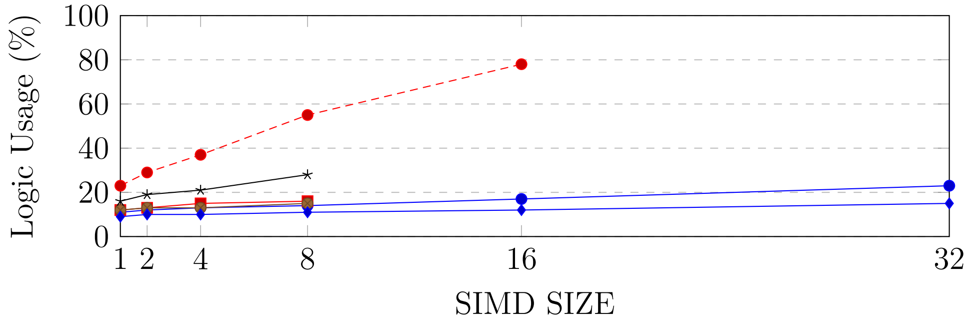
Table 5.4: Throughput per Energy (MegaBytes per Joule) comparison of CPU and Arria10 accelerated algorithms with different SIMD width

Figure 5.11 demonstrates FPGA resource utilization and maximum working frequency of the benchmarks by varying SIMD width. Reported resource usage includes overhead for OpenCL control and board interface components such as PCIe and external memory(about 9% Logic and 10% Memory). The results indicate that by increasing parallelism, the maximum frequency does not drop significantly. Furthermore, resource usage is increased linearly when the amount of parallelism is increased.

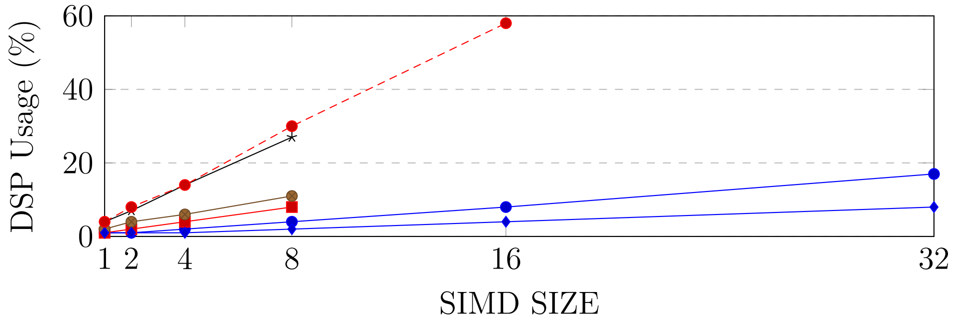
Table 5.5 shows the power consumption of the CPU and FPGA implementations with different SIMD size. We have used LIKWID [99] tool to measure the energy/power of both the host processor and host memory. FPGA power usage is extracted using post-fit analysis. Dynamic power accuracy is further improved via RTL simulation of the designs. Power numbers are reported by table 5.5. Baseline FPGA configuration (PCIe loopback, and OpenCL run-time) consumes 18 Watts adds to the reported numbers. We measured that the host processor consumes 25W for PCIe I/O and reading input frames from file to memory.



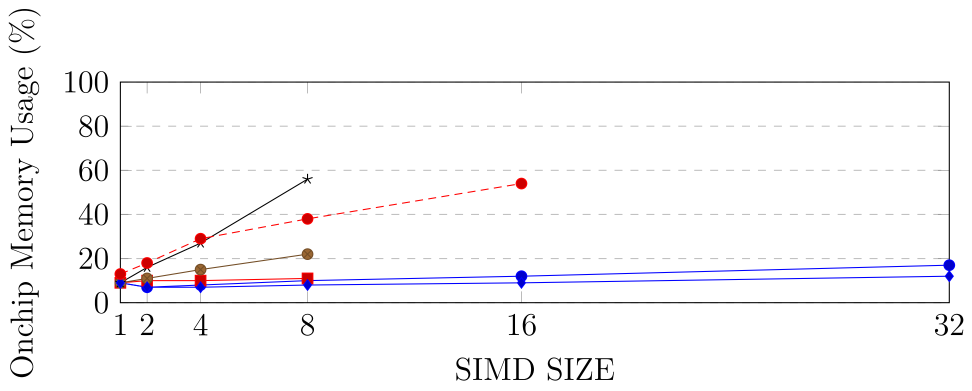
(a) Maximum frequency



(b) Logic usage



(c) DSP usage



(d) RAM usage

Figure 5.11: Arria 10 resource utilization and maximum working frequency of algorithms with different SIMD width

| **Benchmark** | **CPU (AVX+8 Cores)** | **FPGA SIMD=1** | **FPGA SIMD=2** | **FPGA SIMD=4** | **FPGA SIMD=8** | **FPGA SIMD=16** | **FPGA SIMD=32** |
|---|---|---|---|---|---|---|---|
| Census Transform | 42 | 18.3 | 18.5 | 18.8 | 19 | 19.4 | 20.2 |
| Automatic Contrast* | 50 | 20.1 | 20.3 | 20.4 | 21 | N/A | N/A |
| Lane Detection* | 28 | 18.8 | 19 | 19.3 | 21.1 | N/A | N/A |
| Color Copy | 51 | 20.6 | 20.9 | 21.9 | 23.1 | N/A | N/A |
| SIFT keypoints | 35 | 22 | 23.6 | 24.5 | 27.2 | 33.1 | N/A |

Table 5.5: Design power consumption (Watts) of CPU and Arria10 accelerated algorithms with different SIMD width

## 5.7 Availability

This chapter proposed a general-purpose, configurable, and easy-to-use framework called AFFIX to accelerate OpenVX computer vision algorithms on heterogeneous CPU-FPGA platforms. AFFIX source code is available on GitHub[2] under the BSD license. We believe it will be a helpful asset for vision researchers and developers to quickly develop, test, verify, and accelerate their proposed algorithms.

[2]https://github.com/sajjadt/affix

# Chapter 6

# Conclusions and Future Directions

Computer vision is complex and compute-intensive. Building systems with high performance and energy efficiency requires expertise not only on computer vision but also in software design and hardware optimization. Computer vision systems designers often rely on open-source software stacks such as OpenCV or Caffee frameworks to build systems. Such software stacks are developed by experts and contain numerous software and hardware optimizations. This dissertation focus is on two compute platforms for building computer vision systems that were not easily accessible to algorithm designers and developers: the Web and FPGA-based accelerators. Existing solutions such as OpenCV either do not map to them or maps very poorly.

The Web is the world's most ubiquitous computing platform which hosts a plethora of visual content. Due to historical reasons such as insufficient compute performance and lack of API support for acquiring and manipulating images, computer vision is not mainstream on the Web. This dissertation shows that in light of recent web developments such as vastly improved JavaScript performance and the addition of APIs such as WebRTC, efficient computer vision processing can be realized on web clients. It provides a comprehensive computer vision library for the Web by porting the OpenCV library to JavaScript and referring to it as OpenCV.js. OpenCV.js provides an expansive set of functions with an optimized implementation that offers a near-native level of performance. Performance can further improve through JavaScript-based SIMD and thread-based parallelism. We believe this will result in an immersive and perceptual web with transformational effects, including in online shopping, education, and entertainment, among others. With improved performance and excellent portability and security, JavaScript has been subject to widespread usage in several other domains such as server-side programming [98], desktop, and embedded systems. This makes the provided library applicable to a wider range of applications.

Field Programmable Gate Arrays (FPGA)s are a promising solution to mitigate the computational cost of vision algorithms through hardware pipelining and parallelism. However, an efficient FPGA implementation of computer vision algorithms requires hardware design expertise and a considerable amount of engineering person-hours. This dissertation explored the advantages of a domain-specific representation of computer vision and image processing based on OpenVX spec in FPGA design. Since such representation lacks implementation details, different implementation configurations that satisfy various design constraints, such as performance and power budget, can be explored. Towards this goal, this dissertation suggests a framework called AFFIX for automatic optimizing and implementation of vision algorithms on different FPGA architectures. AFFIX hides low-level hardware optimization and implementation details from computer vision algorithm designers and enables them to quickly develop and evaluate various FPGA implementations of vision algorithms without sacrificing performance.

The research presented in this dissertation can be extended in several aspects.

- We show promising performance improvements from parallel execution of OpenCV functions based on ”asm.js” version of the library. However, asm.js is being phased out in favor of WASM. OpenCV.js is currently under active development to support SIMD and multithreaded processing in the context of WebAssembly. We have skipped GPU based parallelism in the current version of OpenCV.js. However, we think the upcoming WebGPU standard can be used to take advantage of GPUs for OpenCV computing tasks.

- AFFIX acceleration framework can be improved in several ways:
  - Deep Neural Network (DNN) Integration: DNNs have significantly improved image classification and recognition over classical methods. The proposed framework can be extended with DNN inference functionality to implement algorithm graphs that are a mix of DNN layers and vision nodes.
  - More sophisticated scheduling policies: In the current version of the framework, CPU functions are executed sequentially. By processing images in tiles, not only more cache-friendly implementation is achieved, it is possible to run multiple functions that even might have data dependencies in parallel.

# Bibliography

[1] Amd openvx open-source on github. `https://github.com/GPUOpen-ProfessionalCompute-Libraries/amdovx-core`. Accessed: 2018-09-01.

[2] Computer vision technology permeates our daily lives. `https://www.iotforall.com/computer-vision-applications-in-daily-life/`. Accessed: 2018-09-01.

[3] Cuda accelerated computer vision library. `https://developer.nvidia.com/embedded/visionworks`. Accessed: 2018-09-01.

[4] Ecmascript 2018 language specification: `https://tc39.github.io/ecma262/`, 2017. Accessed: 2018-15-4.

[5] Gmail, past, present, and future. `https://www.usenix.org/conference/webapps-10/gmail-past-present-and-future`. Accessed:2019-06-16.

[6] Simd.js specification: `http://tc39.github.io/ecmascript_simd/`. Accessed: 2018-15-4.

[7] Web design and applications. `https://www.w3.org/standards/webdesign/`. Accessed: 2019-01-09.

[8] Web workers: `https://www.w3.org/TR/workers/`, 2015. Accessed: 2018-21-2.

[9] Webxr. `https://immersive-web.github.io/webxr/`. Accessed: 2018-09-01.

[10] *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide*. Intel, 2019.

[11] *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*. Intel, 2019.

[12] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving javascript performance by deconstructing the type system. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 496–507, New York, NY, USA, 2014. ACM.

[13] E. Barsom, M. Graafland, and M. Schijven. Systematic review on the effectiveness of augmented reality applications in medical training. *Surgical endoscopy*, 30(10):4174–4183, 2016.

[14] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, et al. Maxwell-a 64 fpga supercomputer. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 287–294. IEEE, 2007.

[15] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *International Conference on Supercomputing*, volume 27. Vienna, Austria, 1997.

[16] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[17] D. R. Butenhof. *Programming with POSIX threads.* Addison-Wesley Professional, 1997.

[18] M. Bynens. Celebrating 10 years of v8: `https://v8.dev/blog/10-years`, 2018. Accessed: 2019-15-4.

[19] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[20] J. Canny. A computational approach to edge detection. In *Readings in Computer Vision*, pages 184–203. Elsevier, 1987.

[21] M. Casas-Sanchez. Accelerated shape detection in images: `https://wicg.github.io/shape-detection-api/`, 2018. Accessed: 2018-15-4.

[22] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[23] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3213–3223, June 2016.

[24] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From opencl to high-performance hardware on fpgas. In *22nd international conference on field programmable logic and applications (FPL)*, pages 531–534. IEEE, 2012.

[25] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[26] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[27] C. De Schryver. *FPGA Based Accelerators for Financial Applications.* Springer, 2015.

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[29] A. Donovan, R. Muth, B. Chen, and D. Sehr. Pnacl: Portable native client executables. *Google White Paper*, 2010.

[30] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[31] G. A. Elliott, K. Yang, and J. H. Anderson. Supporting real-time computer vision workloads using openvx on multicore+ gpu platforms. In *2015 IEEE Real-Time Systems Symposium*, pages 273–284. IEEE, 2015.

[32] A. Ess, T. Mueller, H. Grabner, and L. J. Van Gool. Segmentation-based urban traffic scene understanding. In *BMVC*, volume 1, page 2. Citeseer, 2009.

[33] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean. A guide to deep learning in healthcare. *Nature medicine*, 25(1):24, 2019.

[34] H. Fang, S. Ong, and A. Nee. A novel augmented reality-based interface for robot path planning. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 8(1):33–42, 2014.

[35] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *2009 International Conference on Field Programmable Logic and Applications*, pages 32–37. IEEE, 2009.

[36] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[37] F. Grull, M. Kirchgessner, R. Kaufmann, M. Hausmann, and U. Kebschull. Accelerating image analysis for localization microscopy with fpgas. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 1–5. IEEE, 2011.

[38] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159. IEEE, 2017.

[39] Z. Guo, B. Buyukkurt, and W. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *ACM SIGPLAN Notices*, 39(7):249–256, 2004.

[40] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *16th International Conference on VLSI Design, 2003. Proceedings.*, pages 461–466. IEEE, 2003.

[41] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200. ACM, 2017.

[42] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM.

[43] M. Hidaka, Y. Kikura, Y. Ushiku, and T. Harada. Webdnn: Fastest dnn execution framework on web browser. In *Proceedings of the 2017 ACM on Multimedia Conference*, MM '17, pages 1213–1216, New York, NY, USA, 2017. ACM.

[44] K. Hill, S. Craciun, A. George, and H. Lam. Comparative analysis of opencl vs. hdl with image-processing kernels on stratix-v fpga. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 189–193. IEEE, 2015.

[45] 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, 2014.

[46] D. Honegger, H. Oleynikova, and M. Pollefeys. Real-time and low latency embedded computer vision hardware based on a combination of fpga and mobile cpu. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 4930–4935. IEEE, 2014.

[47] V. Jain and E. Learned-Miller. Fddb: A benchmark for face detection in unconstrained settings. 2010.

[48] P. Jensen, I. Jibaja, N. Hu, D. Gohman, and J. Mc-Cutchan. Simd in javascript via c++ and emscripten. In *Workshop on Programming Models for SIMD/Vector Processing*, 2015.

[49] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[50] I. Jibaja, P. Jensen, N. Hu, M. R. Haghighat, J. McCutchan, D. Gohman, S. M. Blackburn, and K. S. McKinley. Vector parallelism in javascript: Language and compiler support for simd. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 407–418. IEEE, 2015.

[51] S. Jin, J. Cho, X. Dai Pham, K. M. Lee, S.-K. Park, M. Kim, and J. W. Jeon. Fpga design and implementation of a real-time stereo vision system. *IEEE transactions on circuits and systems for video technology*, 20(1):15–26, 2009.

[52] A. B. Johnston and D. C. Burnett. *WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web.* Digital Codex LLC, 2012.

[53] R. Kamasaka, Y. Shibata, and K. Oguri. An fpga-oriented graph cut algorithm for accelerating stereo vision. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2018.

[54] J. M. Kasson, W. Plouffe, and S. I. Nin. Tetrahedral interpolation technique for color space conversion. In *Device-Independent Color Imaging and Imaging Systems Integration*, volume 1909, pages 127–139. International Society for Optics and Photonics, 1993.

[55] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[56] A. Labour, M. Papakipos, S. Okasaka, and J. Timanus. Safe browser plugins using native code modules, Jan. 8 2013. US Patent 8,352,967.

[57] C. Lanczos. *Linear differential operators*, volume 18. SIAM, 1997.

[58] R. Lienhart and J. Maydt. An extended set of haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, volume 1, pages I–I. IEEE, 2002.

[59] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[60] R. Liu, J. Yang, Y. Chen, and W. Zhao. eslam: An energy-efficient accelerator for real-time orb-slam on fpga platform. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 193:1–193:6, New York, NY, USA, 2019. ACM.

[61] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[62] E. Lundgren, T. Rocha, Z. Rocha, P. Carvalho, and M. Bello. tracking.js: A modern approach for computer vision on the web: `https://trackingjs.com`, 2016. Accessed: 2018-15-4.

[63] W. J. MacLean. An evaluation of the suitability of fpgas for embedded vision systems. In *null*, page 131. IEEE, 2005.

[64] J. Matas, C. Galambos, and J. Kittler. Robust detection of lines using the progressive probabilistic hough transform. *Computer Vision and Image Understanding*, 78(1):119–137, 2000.

[65] E. Matsas and G.-C. Vosniakos. Design of a virtual reality training system for human–robot collaboration in manufacturing tasks. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 11(2):139–153, 2017.

[66] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 429–443. ACM, 2015.

[67] E. Nikahd, P. Behnam, and R. Sameni. High-speed hardware implementation of fixed and runtime variable window length 1-d median filters. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(5):478–482, 2016.

[68] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

[69] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan. Xilinx vivado high level synthesis: Case studies. 2014.

[70] H. Omidian and G. G. Lemieux. Exploring automated space/time tradeoffs for openvx compute graphs. In *Field Programmable Technology (ICFPT), 2017 International Conference on*, pages 152–159. IEEE, 2017.

[71] H. Omidian and G. G. Lemieux. Janus: A compilation system for balancing parallelism and performance in openvx. In *Journal of Physics: Conference Series*, volume 1004, page 012011. IOP Publishing, 2018.

[72] S. Park, G. Tarasuk-levin, C. Min, U. Chingunde, and A. Chandwani. Augmented reality aided navigation, June 6 2017. US Patent 9,672,648.

[73] O. M. Parkhi, A. Vedaldi, A. Zisserman, et al. Deep face recognition. In *bmvc*, volume 1, page 6, 2015.

[74] J. Peng, Y. Liu, C. Lyu, Y. Li, W. Zhou, and K. Fan. Fpga-based parallel hardware architecture for sift algorithm. In *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 277–282. IEEE, 2016.

[75] C. Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[76] G. Pour. Understanding software component technologies: Javabeans and activex. In *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)*, pages 398–398, June 1999.

[77] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Trans. Archit. Code Optim.*, 14(3):26:1–26:25, Aug. 2017.

[78] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Trans. Archit. Code Optim.*, 14(3):26:1–26:25, Aug. 2017.

[79] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Real-time computer vision with opencv. *Communications of the ACM*, 55(6):61–69, 2012.

[80] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[81] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill. Addressing system-level optimization with openvx graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 644–649, 2014.

[82] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Resource conscious reuse-driven tiling for gpus. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 99–111, New York, NY, USA, 2016. ACM.

[83] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[84] S. Rezaei, C.-A. Hernandez-Calderon, S. Mirzamohammadi, E. Bozorgzadeh, A. Veidenbaum, A. Nicolau, and M. J. Prather. Data-rate-aware fpga-based acceleration framework for streaming applications. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2016.

[85] S. Rezaei, K. Kim, and E. Bozorgzadeh. Scalable multi-queue data transfer scheme for fpga-based multi-accelerators. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 374–380. IEEE, 2018.

[86] H. N. Santos, P. Alves, I. Costa, and F. M. Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[87] C. Schuldt, I. Laptev, and B. Caputo. Recognizing human actions: a local svm approach. In *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, volume 3, pages 32–36. IEEE, 2004.

[88] K. Seto, H. Nejatollahi, J. An, S. Kang, and N. Dutt. Small memory footprint neural network accelerators. In *20th International Symposium on Quality Electronic Design (ISQED)*, pages 253–258, March 2019.

[89] R. Steinberg and L. Floyd. An adaptive algorithm for spatial greyscale. *Proc. Soc. Inf. Disp.*, 17:75–77, 1976.

[90] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[91] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. Adrenaline: An openvx environment to optimize embedded vision applications on many-core accelerators. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 289–296. IEEE, 2015.

[92] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. Optimizing memory bandwidth exploitation for openvx applications on embedded many-core accelerators. *Journal of Real-Time Image Processing*, 15(1):73–92, 2018.

[93] S. Taheri. Bringing the power of simd.js to gl-matrix: `https://hacks.mozilla.org/2015/12/bringing-the-power-of-simd-js-to-gl-matrix/`, 2015. Accessed: 2018-15-4.

[94] S. Taheri, P. Behnam, E. Bozorgzadeh, A. Veidenbaum, and A. Nicolau. Affix: Automatic acceleration framework for fpga implementation of openvx vision algorithms. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 252–261, New York, NY, USA, 2019. ACM.

[95] S. Taheri, L. A. Beni, A. V. Veidenbaum, A. Nicolau, R. Cammarota, J. Qiu, Q. Lu, and M. R. Haghighat. Webrtcbench: a benchmark for performance assessment of webrtc implementations. In *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–7. IEEE, 2015.

[96] S. Taheri, J. Heo, P. Behnam, J. Chen, A. Veidenbaum, and A. Nicolau. Acceleration framework for fpga implementation of openvx graph pipelines. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–227. IEEE, 2018.

[97] S. Taheri, A. Vedienbaum, A. Nicolau, N. Hu, and M. R. Haghighat. Opencv.js: Computer vision processing for the open web platform. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, pages 478–483, New York, NY, USA, 2018. ACM.

[98] S. Tilkov and S. Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[99] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, 2010.

[100] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. In *Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 586–591. IEEE, 1991.

[101] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu. scikit-image: image processing in python. *PeerJ*, 2:e453, 2014.

[102] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *2010 18th IEEE annual international symposium on field-programmable custom computing machines*, pages 127–134. IEEE, 2010.

[103] J. Vourvoulakis, J. Kalomiros, and J. Lygouras. Fpga-based architecture of a real-time sift matcher and ransac algorithm for robotic vision applications. *Multimedia Tools and Applications*, 77(8):9393–9415, 2018.

[104] W. Wang, J. Yan, N. Xu, Y. Wang, and F.-H. Hsu. Real-time high-quality stereo vision system in fpga. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(10):1696–1708, 2015.

[105] J. Weberruss, L. Kleeman, D. Boland, and T. Drummond. Fpga acceleration of multi-level orb feature extraction for computer vision. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2017.

[106] W. L. Winston, M. Venkataramanan, and J. B. Goldberg. *Introduction to mathematical programming*, volume 1. Thomson/Brooks/Cole Duxbury; Pacific Grove, CA, 2003.

[107] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-level synthesis of dynamic data structures: A case study using vivado hls. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 362–365. IEEE, 2013.

[108] K. Yang, G. A. Elliott, and J. H. Anderson. Analysis for supporting real-time computer vision workloads using openvx on multicore+ gpu platforms. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 77–86. ACM, 2015.

[109] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang. Making openvx really" real time". In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 80–93. IEEE, 2018.

[110] Y. U. Yim and S.-Y. Oh. Three-feature based automatic lane detection algorithm (tfalda) for autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 4(4):219–225, 2003.

[111] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual correspondence. In *Proceedings of the Third European Conference on Computer Vision (Vol. II)*, ECCV '94, pages 151–158, Berlin, Heidelberg, 1994. Springer-Verlag.

[112] A. Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM.

[113] E. Zatepyakin. Jsfeat-javascript computer vision library: `https://github.com/inspirit/jsfeat`. Accessed: 2018-15-4.

[114] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[115] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen. Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas. In *Proceedings of the International Conference on Computer-Aided Design*, page 56. ACM, 2018.

[116] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24. ACM, 2017.

[117] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu. Hitgraph: High-throughput graph processing framework on fpga. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2019.

[118] Z. Zivkovic and F. Van Der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern recognition letters*, 27(7):773–780, 2006.