CECS

CENTER FOR EMBEDDED & CYBER-PHYSICAL SYSTEMS

OpenCV.js: Computer Vision Processing for the Web

Sajjad Taheri, Alexander Veidenbaum, Alexandru Nicolau and Mohammad R. Haghighat

Center for Embedded and Cyber-Physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

{sajjadt, alexv, anicolau}@uci.edu, mohammad.r.haghighat@intel.com

CECS Technical Report 17-2

July 24, 2017

OpenCV.js: Computer Vision Processing for the Web

Sajjad Taheri, Alexander Veidenbaum, Alexandru Nicolau
Computer Science Department, Univerity of California, Irvine
sajjadt,alexv,anicolau@uci.edu

Mohammad R. Haghighat
Intel Corp
mohammad.r.haghighat@intel.com

ABSTRACT

The Web is the most ubiquitous computing platform. There are already billions of devices connected to the web that have access to a plethora of visual information. Understanding images is a complex and demanding task which requires sophisticated algorithms and implementations. OpenCV is the defacto library for general computer vision application development, with hundreds of algorithms and efficient implementation in C++. However, there is no comparable computer vision library for the Web offering an equal level of functionality and performance. This is in large part due to the fact that most web applications used to adopt a client-server approach in which the computational part is handled by the server. However, with HTML5 and new client-side technologies browsers are capable of handling more complex tasks. This work brings OpenCV to the Web by making it available natively in JavaScript, taking advantage of its efficiency, completeness, API maturity, and its community's collective knowledge. We developed an automatic approach to compile OpenCV source code into JavaScript in a way that is easier for JavaScript engines to optimize significantly and provide an API that makes it easier for users to adopt the library and develop applications. We were able to translate more than 800 OpenCV functions from different vision categories while achieving near-native performance for most of them.

Keywords

Computer vision; Web; JavaScript; Performance

1. INTRODUCTION AND MOTIVATION

JavaScript has rapidly evolved from a programming language designed to add scripting capabilities to make static web pages more appealing[1] into the most popular and ubiquitous programming language deployed on billions of devices [2, 3]. With emergence of new technologies such as WebVR and WebRTC, the popularity of Internet Of Things(IOT) platforms, and with the abundance of visual data, computer vision processing on the web will have numerous applications.

However, computer vision usually has high computational cost, due to 1) sheer amount of computation especially on images with higher resolution and high frame rates, 2) complex algorithms to process and understand the visual data, 3) real time requirements for interactive applications. JavaScript is a scripting dynamically-typed language, which makes it performance-wise inferior to languages such as C++. With the web based application development, the general paradigm is used to be deploying computationally complex logic on the server. However, with recent client side technologies, such as Just in time compilation, web clients are able to handle more demanding tasks.

There are recent efforts to provide computer vision for web based platforms. For instance [4] and [5] have provided lightweight JavaScript libraries that offer selected vision functions. There is also a JavaScript binding for Node.js, that provides JavaScript programs with access to OpenCV libraries. There are requirements for a computer vision library on the web that are not entirely addressed by the above mentioned libraries that this work seeks to meet:

1. High performance: Computer vision processing requires a large amount of complex computation. This substantial computational cost demands efficient implementation and careful optimization.

2. Comprehensiveness: Complex computer vision applications often incorporate several algorithms from different domains to such as image processing, machine learning, and data analysis. Hence, it is amenable to provide programmers wit a comprehensive list of functionality.

3. Portability: Library must be portable across all diverse web based platforms including browsers and IOT devices.

4. Adoptability: It should be easy for the community to adopt the library. This requires documentation, tutorials and online forums.

Towards achieving these goals we decided to port OpenCV library to JavaScript, so that can be run in different web clients. We call it OpenCV.js. OpenCV[6] is an open source computer vision library that offers a large number of low level kernels and high level applications ranging from image processing, object detection, tracking, and matching. OpenCV provides efficient implementations of algorithms optimized for multiple target architectures such as Desktop and mobile processors and GPUs[7]. OpenCV has a mature API which is well documented with a lot of samples and online tutorials. The source code is also rigorously tested. Although it is developed in C++, there are bindings for other languages such as Python and Java that expand its availability to a broader scope and audience. OpenCV.js is provided as a JavaScript library that works with different

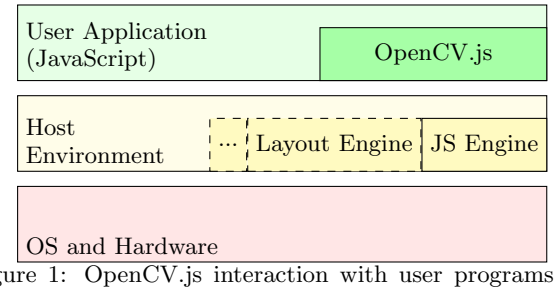

Figure 1: OpenCV.js interaction with user programs and host environments

web environment. As shown in Figure 1, it utilizes the underlying environment to perform computation and media access. Environments rely on JavaScript engines for executing JavaScript logic and utilize rendering and browser engines to display graphics. Two such environments that are considered in this work are: web browsers such as Mozilla Firefox and Node.js[8]. Table 1 provides a comparison between vision libraries that are available to JavaScript programs.

2. METHODOLOGY

This section describes the approach taken to compile OpenCV source code directory into JavaScript equivalent. Modifications to the source code and techniques to adopt it to the web model will be discussed. We have used Emscripten[9] to generate JavaScript code. It is a source to source compiler developed by Mozilla to translates LLVM bitcode to JavaScript. Emscripten targets a subset of JavaScript called ASM.js that allows engines to perform extra level of optimizations.

2.1 ASM.js

ASM.js[10] is a strict subset of JavaScript that is designed to allow JavaScript engines to perform additional optimizations that is not possible with normal JavaScript. Some JavaScript engines such as SpiderMonkey are even able to perform Ahead-Of-Time (AOT) compilation.

ASM.js makes several limiations to the normal JavaScript: 1) data is annotated with explicit type information. This information can be sued to eliminates dynamic type guards. Code listing 1 demonstrate this technique. without this assumption, since addition between different types such as Numbers and Strings leads to different operations, JavaScript engines need to generate guards that check the data type dynamically. 2) ASM.js utilizes JavaScript typed arrays to provide an abstraction for program memory similar to the C/C++ virtual machine. Typed arrays are raw buffers available in JavaScript that engines are highly optimized for working with it. Listing 2 demonstrate how typed arrays can be used to model program memory. 3) Memory is expected to be manually managed by the programmer and no garbage collection is provided.

Listing 1: Type inferrance based optimization

```
function add (x, y) {
  x = x|0;  // x is a 32-bit value
  y = y|0;  // y is also a 32-bit value

  // 32-bit addition
  return x+y;
}
```

Listing 2: Implementation of Strlen function with typed arrays used as program memory

```
// Memory
var Memory = new Uint8Array(256*1024);

function strlen(ptr) {
  ptr = ptr|0;
  var curr = ptr;
  while (Memory[curr]|0 != 0) {
    curr = (curr + 1)|0;
  }
  return (curr-ptr)|0;
}
```

2.2 WebAssembly

While ASM.js provides opportunity to reach near native performance, it has limitations that make it a challenge to use for some targets with low resources such as embedded devices. One major limitation is that the size of the generated JavaScript code tend to be large. This makes parsing JavaScript code to hot spot especially on mobile devices. This motivates WebAssembly[11] to be pushed forward. WebAssembly is a portable size and load-time efficient binary format designed as an alternative target for web compilation.

2.3 Binding Generation and Compilation

Regardless of the target, there is an issue with this approach: as part of the compilation, compiler removes high level language information such as class and function identifiers and assign unique mangled names to them. While this is fine for compiling a C++ programs to executable JavaScript programs, when porting libraries, it will be almost impossible for developers to develop programs through mangled names. To address this issue, we have provided an automatic approach to extract binding information of different OpenCV entities such as functions and classes and expose them to JavaScript properly. This enables the library to have a similar interface with normal OpenCV that many programmers are already familiar with. Table 2 shows equivalent JavaScript data types for common C++ data types . Listing 3 shows how C++ and JavaScript API of OpenCV can be used to implement a filter.

Although it is possible to convert the majority of OpenCV library to JavaScript, in order to make it portable, some of its functionality can be skipped:

1. There are alternative implementations for some OpenCV functions that are better suited for the web. For instance accessing file system is not trivial in web and functions to access media devices such as cameras, and graphical user interfaces have alternatives. We provide a JavaScript helper module that uses HTML5 primitives to provides users with replacement functions to access to files hosted on the web, media devices through $getUserMedia$ and display graphics using HTML Canvas.

2. OpenCV is very comprehensive. Some of the provided functions are not used in certain applications domains. Including them in the library, will make the library unnecessary bigger. We allow users to select the functions that they wish to port.

Library	Features	Development Language	Portability
Node.js OpenCV Binding	Image processing, object detection and tracking,features framework, machine learning,	C++	No
Tracking.js	Object detection and tracking	JavaScript	Yes
JSFeat	Select functions from Image processing, object detection and feature extracting	JavaScript	Yes
OpenCV.js	Image processing, object detection and tracking,features framework, machine learning,	C++	Yes

Table 1: Comparison of JavaScript Computer Vision Libraries

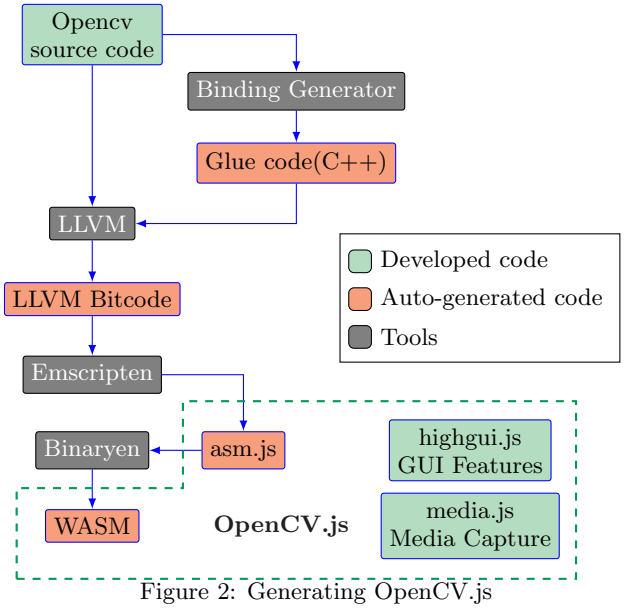

Figure 2: Generating OpenCV.js

Figure 2 lists the steps involved in the process of converting OpenCV C++ code to JavaScript. First OpenCV source code is patched to disable components and implementations that are platform specific, or are not optimized for the web. Next, information about classes and functions that should be exported to JavaScript are extracted. OpenCV source code is already annotated with directive to guide binding generators for other languages such as Python. We have used those directives and utilized Embind(Binding generator for Emscripten) to generate a glue code that maps JavaScript symbols to C++ symbols and compile it along with the OpenCV library to JavaScript. To generate WebAssembly version of the library, we have used BinaryEn toolkit which compiles ASM.js code into WebAssembly. Both ASM.js and WebAssembly targets offer the same functionality and can be used interchangeably if supported by the JavaScript engine. We have developed several helper JavaScript libraries to provide access to media devices and files, and GUI features.

3. EXPERIMENTAL EVALUATION

This section discusses performance evaluation of JavaScript version of selected kernels and applications from OpenCV version 3.1. We used a Desktop computer with Intel Corei7-3770 processor and 8 GBs of RAM, running Ubuntu 16.04 Linux as our setup. Table 3 lists two different environments that are used in our evaluation. For each environment, latest released software version is used. Experiments are performed on long sequences of raw video data (400-600 frames) collected from Xiph.org archive and average execution time is reported.

C++ Type	JavaScript Type
Arithmetic types(e.g. int, float)	Number
bool	Boolean
enumeration	Constant
Basic Structures(e.g. cv::Point)	Value Objects
std::vector	Array
std::string	String
C++ objects	JavaScript Objects

Table 2: Exported JavaScript types for common C++ types

Host Environment	JavaScript Engine	CPU	OS
Firefox 55	SpiderMonkey 55	Intel Corei7	Ubuntu 16.04
Node.js 8.1	V8 5.8		

Table 3: Evaluation Platforms

3.1 Selected Vision Functions

We consider two types of benchmarks in our evaluation. The first category includes primitive kernels that perform simple operations such as pixel-wise addition or convolution. They are repeated for different common pixel types (e.g. unsigned chars, short and floats) for each operation. The second category includes sophisticated vision functions that involve a collection of primitive kernels. List of all the benchmarks with a description of their operations is shown in table4. Our elected vision functions include implementation of Canny's algorithm for finding edges[12], ORB algorithm for finding rotation invariant features within a picture[13], finding faces and eyes by using Haar cascades[14], and finding people by analyzing histogram of gradients[15]. Figure 4 depicts response of the mentioned applications to a sample input frame.

3.2 Optimization Trade-offs

In compiling OpenCV to JavaScript, there are several optimization trade-offs that affect the performance the generated code. Among them, ability to enlarge the program memory and behavior of floating point arithmetic have the biggest effect on performance.

3.2.1 Enabling Memory growth

Allowing program memory that are used by ASM.js to en-

```
void erode() {
    Mat image, dst;
    image = imread("image.jpg");

    // Create a structuring element
    int size = 6;
    Mat element = getStructuringElement(
                  MORPH_RECT,
                  Size(2*size+1, 2*size+1),
                  Point(size, size));

    // Apply erosion or dilation on the image
    erode(image, dst, element);

    namedWindow("Input");
    imshow("Input", image);

    namedWindow("Result");
    imshow("Result", dst);
}
```

```
function erode() {
    var image = cv.imread("image.jpg");

    // Create a structuring element
    var size = 6;
    var element = cv.getStructuringElement(
                  cv.MORPH_RECT,
                  [2*size+1, 2*size+1],
                  [size, size]);

    erode(image, dst, element);

    // displaying on canvas with id="Input"
    imshow("Input", image);
    // displaying on canvas with id="Result"
    imshow("Result", dst);
    image.delete();
    dst.delete();
}
```

Figure 3: Erosion implementation using OpenCV C++(left) and JavaScript(right) APIs

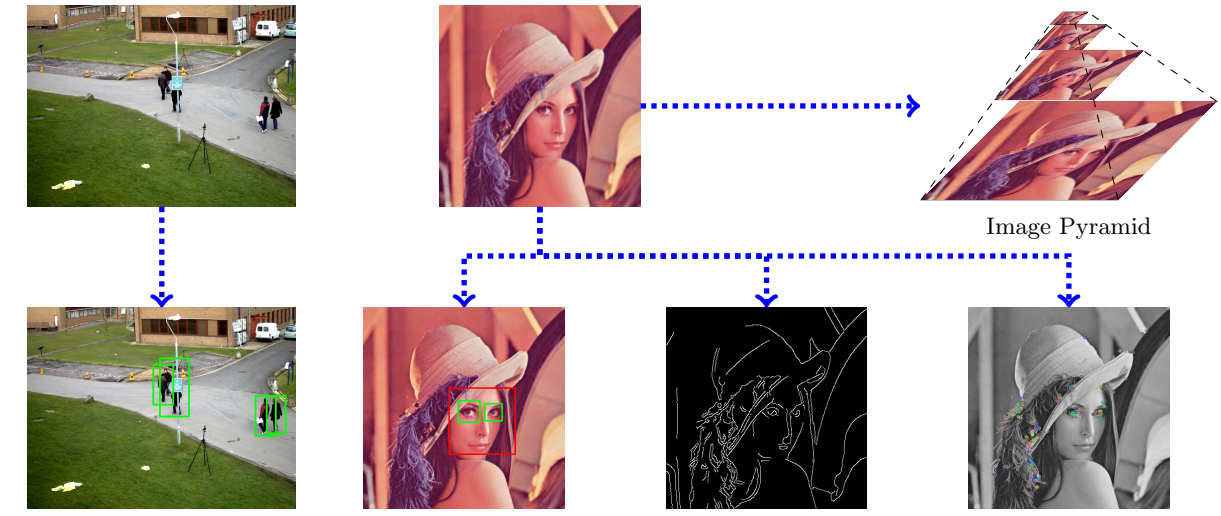

Figure 4: Demonstration of selected computer vision applications

Name	Module	Data type	Description
add	Core	char - Short - float	Pixel-wise addition with saturation
absdiff		char - short	Pixel wise absolute difference
bitwise		char - short	Pixel wise bit-wise operations
addweightd		float	Pixel-wise weighted addition
integral	Image Processing	char - short	Image integral
threshold		char - short - float	Simple threshold operation
gblur		char	Gaussian Blur with 3x3, 5x5 and 7x7 kernels
bilat		float	Bilateral Filter
erode		char	Erosion operation (morphology)
rgb2gray		RGB	Converting color (RGB) images to grayscale
canny		grayscale	Canny edge detection algorithm
Pyramids		grayscale	Creating image pyramid with 4 layers
ORB	Feature2d	grayscale	ORB feature extraction algorithm
Face detection	Object Detection	grayscale	Face detection using Haar cascades
People detection		grayscale	People detection using Histogram of Oriented Gradients(HOG)

Table 4: Selected Vision Functions

large is very helpful, especially in cases that the peak amount of memory needed during run-time is not known beforehand. However, allowing memory size to grow disables several optimization by JIT compilers and degrades performance. Enlarging memory also requires copying the entire underlying array which is also expensive. We have found this feature to significantly affects performance on some JavaScript engines and disabled it.

Figures 5 and 6 show performance of different integer operations on both Node.js and Firefox when memory size is fixed. As shown, performance of integer operations is very competitive to the native implementation. They tend to be better in WebAssembly implementation due to better compiler optimization. Performance of floating point operations will be discussed in the following section.

3.2.2 Floating point Arithmetic Performance

JavaScript language uses double precision floating point to represents every numerical value including integer and single precision floating point numbers. While double precision floating point offers higher precision, using it to implementing single precision operations might lead to erroneous results in some cases. Emscripten supports generating both precise and imprecise floating point arithmetic. We compile the library in both modes and report their performance. As it can be seen in Figure 8, performance is close to the native as most JavaScript engines including V8 and SpiderMonkey optimize floating point computations internally. However there is one exception. Since some of the WebAssembly floating point operations are not optimized by V8 used by Node.js, there are major slowdowns. However, we can report that the latest version of V8 engine (6.1) has fixed those issues and performance is comparable to ASM.js version.

3.3 Overheads

Compiling native code into JavaScript often generates large files for which downloading, parsing and compiling become a challenge. We have used Zlib compression to reduce the size of the library. For evaluation of size and initialization time of the library, we use a port of OpenCV with 800 functions from modules such as core, image processing, object detection, features framework, machine learning, photo. We consider four different versions of the library: 1)ASM.js, 2)WebAssembly, 3)compressed ASM.js and 4)compressed WebAssembly. A JavaScript port of Zlib(compiled with Emscripten) is used to decompress the g-zipped library at run time. Zlib library overhead is included in our report. Figures 10 reports the total size of the library. WebAssembly target is 2x more compact than the ASM.js target. Compressing the library further reduces the size by 3-4 times. Library can become even more compact by removing the unnecessary modules and components. Figure 11 reports the time spent on initializing the library on Firefox.

4. CONCLUSIONS

Web is the universal computing platform with abundance of visual data. Computer vision processing requires sophisticated algorithms and implementations which are computationally demanding. Due to web limitations, a comprehensive framework for computer vision is not available. With new client-side technologies, web is capable of taking advantage of compiled languages. This work brings years of OpenCV development to the web with near-native performance. It also complements it by replacing its platform dependant components with portable alternatives that are implemented using native web and HTML5 technologies.

5. AVAILABILITY

Source code including build instructions, tests and examples is published on GitHub [1]. Work is still ongoing to improve the documentation and provide interactive tutorials.

6. ACKNOWLEDGMENTS

This work is supported by the Intel Corporation. The authors would like to thank OpenCV and Emscripten developer community for their helpful comments and anonymous reviewers for their helpful suggestions.

7. REFERENCES

[1] Charles Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012.

[2] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, volume 15, 2009.

[3] Elizabeth Latronico, Edward A Lee, Marten Lohstroh, Chris Shaver, Armin Wasicek, and Matthew Weber. A vision of swarmlets. *IEEE Internet Computing*, 19(2):20–28, 2015.

[4] Foat Akhmadeev. *Computer Vision for the Web*. Packt Publishing Ltd, 2015.

[5] Eduardo Lundgren, Thiago Rocha, Zeno Rocha, Pablo Carvalho, and Maira Bello. tracking. js: A modern approach for computer vision on the web. *Online]. Dosegljivo: https://trackingjs. com/[Dostopano 30. 5. 2016]*, 2015.

[6] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[7] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Real-time computer vision with opencv. *Communications of the ACM*, 55(6):61–69, 2012.

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

[9] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

[10] asm.js. http://asmjs.org, 2017.

[11] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF 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.

[1] https://www.github.com/ucisysarch/opencvjs

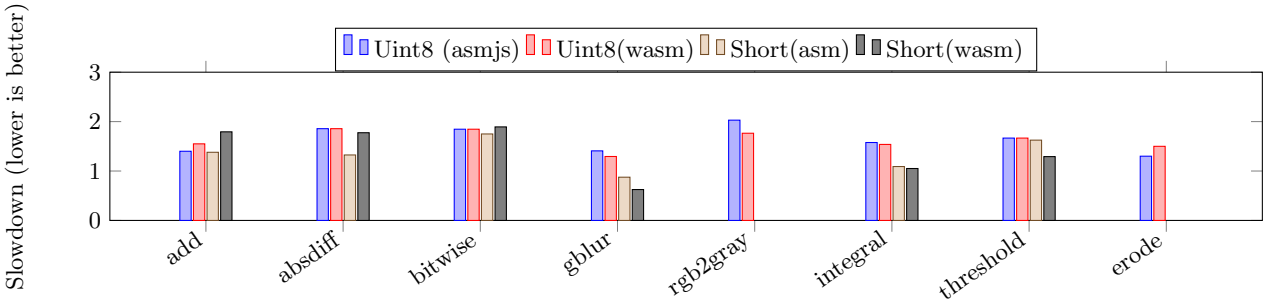

Figure 5: Performance comparison of integer arithmetic (Firefox)

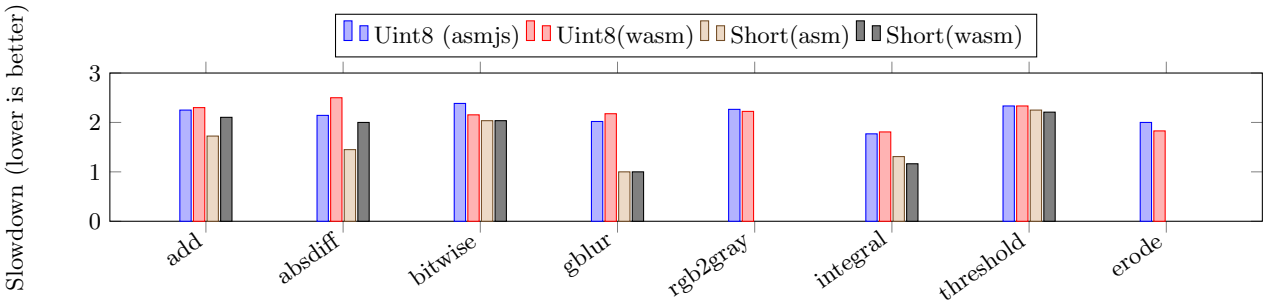

Figure 6: Performance comparison of integer arithmetic (Node.js)

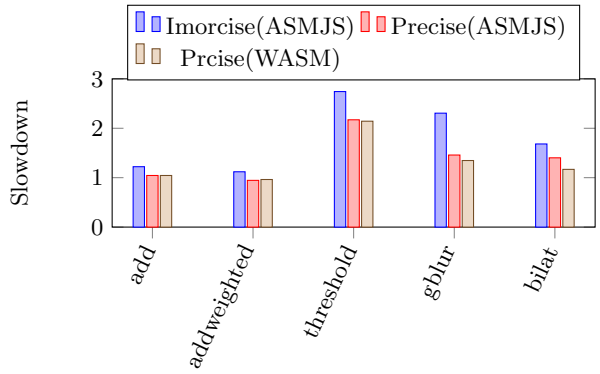

Figure 7: Performance comparison of floating point arithmetic (Firefox)

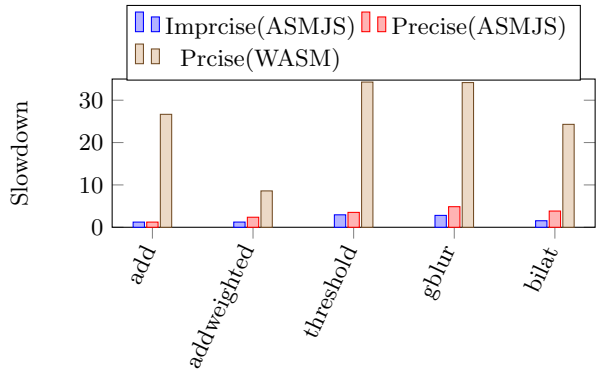

Figure 8: Performance comparison of floating point arithmetic (Node.js)

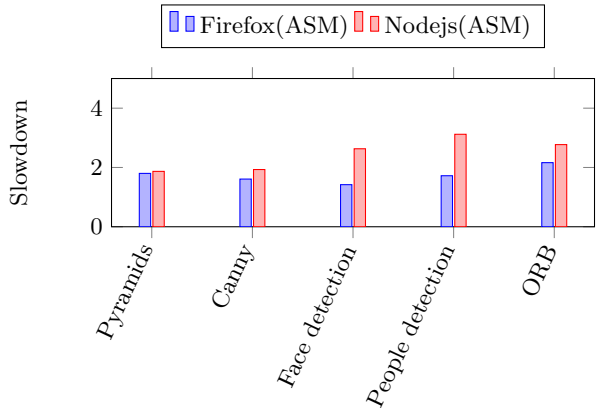

Figure 9: Performance comparison of select vision applications

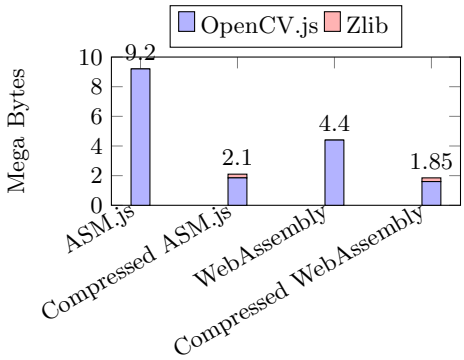

Figure 10: Library Size

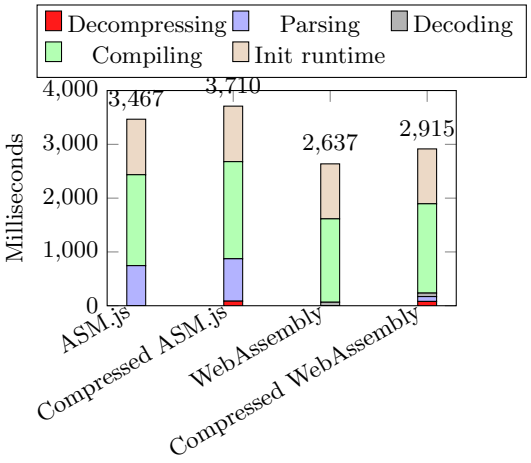

Figure 11: Average Start-up Time on Firefox

[12] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[13] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE international conference on*, pages 2564–2571. IEEE, 2011.

[14] Rainer Lienhart and Jochen 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.

[15] Navneet Dalal and Bill 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.