OpenCV.js: Computer Vision Processing for the Open Web Platform

Sajjad Taheri, Alexander Vedienbaum, Alexandru Nicolau
Department of Computer Science, University of California, Irvine
Irvine, California
{sajjadt,alexv,anicolau}@ics.uci.edu

Ningxin Hu
Intel Corporation
Shanghai, PRC
ningxin.hu@intel.com

Mohammad R. Haghighat
Intel Corporation
Santa Clara, California
mohammad.r.haghighat@intel.com

ABSTRACT

The Web is the world's most ubiquitous compute platform and the foundation of digital economy. Ever since its birth in early 1990's, web capabilities have been increasing in both quantity and quality. However, in spite of all such progress, computer vision is not mainstream on the web yet. The reasons are historical and include lack of sufficient performance of JavaScript, lack of camera support in the standard web APIs, and lack of comprehensive computer-vision libraries. These problems are about to get solved, resulting in the potential of an immersive and perceptual web with transformational effects including in online shopping, education, and entertainment among others. This work aims to enable web with computer vision by bringing hundreds of OpenCV functions to the open web platform. OpenCV is the most popular computer-vision library with a comprehensive set of vision functions and a large developer community. OpenCV is implemented in C++ and up until now, it was not available in the web browsers without the help of unpopular native plugins. This work leverage OpenCV efficiency, completeness, API maturity, and its communitys collective knowledge. It is provided in a format that is easy for JavaScript engines to highly optimize and has an API that is easy for the web programmers to adopt and develop applications. In addition, OpenCV parallel implementations that target SIMD units and multiprocessors can be ported to equivalent web primitives, providing better performance for real-time and interactive use cases.

CCS CONCEPTS

• **Information systems** → **World Wide Web**; • **Computing methodologies** → **Computer vision**; • **Theory of computation** → *Parallel algorithms*;

KEYWORDS

Computer vision; Multimedia; Web; JavaScript; Parallel Processing; Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MMSys'18, June 12–15, 2018, Amsterdam, Netherlands
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5192-8/18/06…$15.00
https://doi.org/10.1145/3204949.3208126

ACM Reference Format:
Sajjad Taheri, Alexander Vedienbaum, Alexandru Nicolau, Ningxin Hu, and Mohammad R. Haghighat. 2018. OpenCV.js: Computer Vision Processing for the Open Web Platform. In *Proceedings of 9th ACM Multimedia Systems Conference (MMSys'18).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3204949.3208126

1 INTRODUCTION

The Web is the most ubiquitous compute platform with billions of connected devices. It's popularity in online commerce, entertainment, science and education has been increasing in a tremendous way. There is also an ever growing amount of multimedia content on the web. Despite such rapid progress in qunatity and quality of content, computer vision processing on the web browsers has not been a common practice. One approach taken by developers is 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. 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), and 3) lack of a comprehensive computer vision library to develop apps. We show that this work along with other recent developments on the web front, will address those limitations and empower 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 of third-party plugins. Among them Web Real-Time Communication (WebRTC) allows capturing and peer-to-peer transportation of multimedia content, and video element API can be used to display videos. Recently, immersive web provides web applications with access to Virtual Reality (VR) and Augmented Reality (AR) contents that makes new engaging user experiences on web possible.
(2) **Improved JavaScript performance**: JavaScript is the dominant language of the web. Since it is a scripting language with dynamic typing, it's performance is inferior to native languages such as C++. Multimedia processing often involves sheer amount of computation and complex algorithms which makes it very expensive. With client side technologies, such as Just-In-Time (JIT) compilation, and with the introduction of WebAssembly (WASM), a portable binary format for the

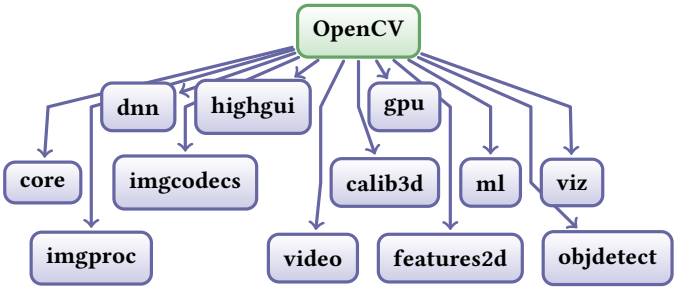


Figure 1: OpenCV is implemented as a collection of modules

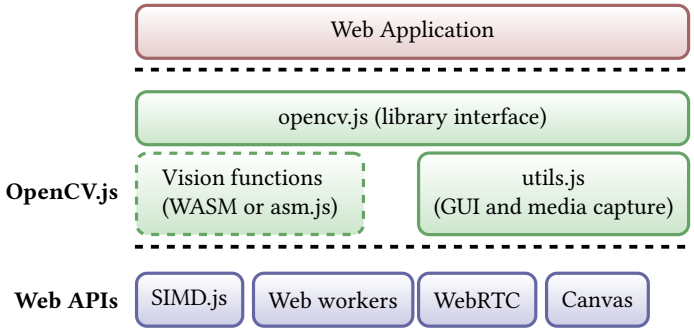


Figure 2: OpenCV.js components and its interactions with web applications and web APIs

web compilation [1], web clients are able to reach near native level of JavaScript performance and handle more demanding tasks.

(3) **A comprehensive computer vision library**: Although there are several computer vision libraries developed in native languages such C++, 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 [2–4]. However, they often provide a handful of vision functions from certain domains such as object detection, video tracking or deep neural networks. Expanding them with new algorithms and optimizing the implementation is a challenging task. On the other hand, Accelerated shape detection API [5] provides functions to detect shapes such as faces and bar codes while the efficient implementation is left to browser vendors. The above mentioned works suffer from lack of either functionality, performance or portability. As an alternative approach, we take advantage of an existing computer vision library that is developed in a native language (i.e. OpenCV) and make it to work on the web. We show that this approach works great on the web for several reasons: 1) expansive set of functions with optimized implementation is provided, 2) it performs more efficiently than normal JavaScript implementations and performance will even improve through parallelism, and 3) developers can access a big collection of existing resources such as tutorial and examples.

2 OPENCV.JS: OPENCV IN JAVASCRIPT

In the native world, OpenCV [6] is the de-facto library for computer vision development. OpenCV is very comprehensive, and as shown by Fig. 1, 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, machine learning, and deep neural networks (DNNs). OpenCV also provides efficient implementation that targets both scalar and parallel hardware such as multiple processor cores and GPUs.

We translate OpenCV from C++ into JavaScript and refer to it as OpenCV.js. Table 1 categorizes and lists the functions that are included within OpenCV.js. Several OpenCV modules excluded for two reasons:

(1) Not all of OpenCV's offerings is compatible with the web. For instance high level GUI and I/O module("highgui") which provides functions to access media devices such as cameras, and graphical user interfaces, is platform dependent and cannot be compiled to the web. Those functions however, can be implemented using HTML5 primitives. For instance, *getUserMedia* can be used to access media devices, and 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 module ("calib3d") has applications in automation and robotics. In order to reduce the size of the generated library for the general usage, we have identified the least commonly used functions from OpenCV and excluded them from the library. However, since there are still many functions that might be useful for special use cases, we have provided a way to build the library with a list of user-selected functions.

.

Fig. 2 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 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 Video/Canvas to provide media access and GUI capabilities, and uses web workers and SIMD.js to implement parallel algorithms.

3 TRANSLATING OPENCV TO JAVASCRIPT AND WEBASSEMBLY

In this section, we describe our approach and the tools that are used to generate OpenCV.js from OpenCV source code. We have used Emscripten [7], a LLVM-based source-to-source compiler developed by Mozilla that converts C++ code into JavaScript. Originally Emscripten targets a typed subset of JavaScript called *asm.js* that, due to its simplicity, allows JavaScript engines to perform extra level of optimizations. In fact, it is even possible to compile *asm.js*

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 1: OpenCV.js modules and provided functions

functions before execution. While performance is impressive, parsing and compiling big JavaScript files could become bottleneck, especially for mobile devices with weaker processors. This was one of the main motivations for development of WebAssembly (WASM) [1]. WASM is a portable size and load-time efficient binary format designed as a target for web compilation. Compared to asm.js, it is more compact and is much faster to parse and compile. WASM will eventually make asm.js obsolete. However, WASM is still under development and is not fully supported on older JavaScript engines. We have used Emscripten to compile OpenCV source code into both asm.js and WASM. Both version offer the same functionality and can be used interchangeably.

During compilation process with Emscripten, C++ high level language information such as class and function identifiers are replaced with mangled names. It will be almost impossible for developers to develop programs through mangled names. To enables the library to have a similar interface with normal OpenCV that many programmers are already familiar with, we provide binding information of different OpenCV entities such as functions and classes and expose them to JavaScript properly. Since OpenCV is large, and growing continuously through new contributions, continuously 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. Fig. 3 lists the steps involved in the process of converting OpenCV C++ code to JavaScript. At first, OpenCV source code is configured 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 will be extracted from OpenCV source code. For 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 should be included in the final JavaScript build. This list can be updated to include or exclude OpenCV modules and/or functions. By using the binding information and function white list, we generate a glue code that maps JavaScript symbols to C++ symbols and compile it with Emscripten along with the rest of OpenCV library into JavaScript. The output of this process will be a JavaScript file (opencv.js) that serves as library interface along with WASM or asm.js implementation of OpenCV functions. utils.js which includes GUI, I/O and utility functions, and is implementd seaparatley, will also be linked with the rest of opencv.js.

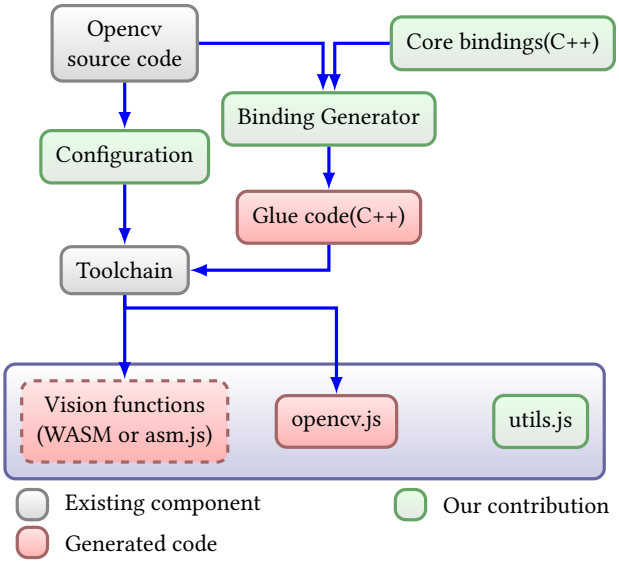


Figure 3: Generating OpenCV.js

4 USING OPENCV.JS

OpenCV.js API is based on OpenCV C++ API and shares many similarities with it. For instance, C++ functions are exported to JavaScript with the same name and signature. Function overloading and default parameters are also supported in JavaScript version. This makes it alot easier to migrate to JavaScript for users who are already familiar with OpenCV development in C++. Although OpenCV C++ classes are ported to JavaScript objects with the same member functions and properties, basic data types are different between the two versions. For instance, JavaScript is using Number (double precision floating point) for all numerical types. JavaScript engines use garbage collector(GC) to manage program memory. However, GC activity has negative impact on the 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. Table 2 shows equivalent JavaScript data types for basic C++ data types.

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)	Array
std::vector (of cv::Mat)	cv.Vector

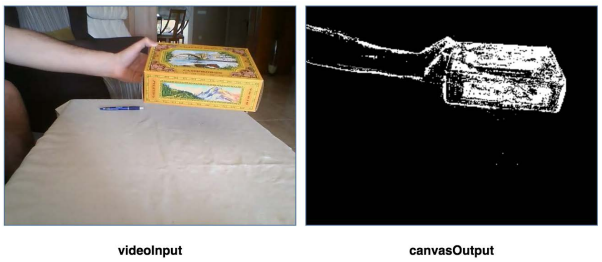
Table 2: Exported JavaScript types for OpenCV basic C++ types

Listing 1 shows a sample JavaScript program which uses *MOG2* method (Gaussian mixture model based) provided by OpenCV.js to subtract the background from the input video. This example works on top of a simple HTML page with a HTML5 video element named *videoInput* serving as the input source and a canvas element named *canvasOutput* which renders the program output. In line 2, *cv.VideoCapture* utility function is used to access input video frames from the video element. In lines 3 and 5, two OpenCV matrices are created to hold the input and output frames. In line 7 a background subtractor instance is created. MOG algoritm will be applid to every frame on the input video. 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 20). In line 21 we will display the result on the output canvas and then schedule the function to be called for the next frame (line 24). Fig. 4 shows two snapshots of this program running inside a browser.

Listing 1: Background subtraction example

```
1  var video = document.getElementById('videoInput'),
2      cap = new cv.VideoCapture(video),
3      frame = new cv.Mat(video.height, video.width,
4              cv.CV_8UC4),
5      fgmask = new cv.Mat(video.height, video.width,
6               cv.CV_8UC1),
7      fgbg = new cv.BackgroundSubtractorMOG2(500,16,true);
8  const FPS = 30;
9  function processVideo() {
10     try {
11         if (!streaming) { // clean and stop.
12            frame.delete(); fgmask.delete(); fgbg.delete();
13            return;
14         }
15         let begin = Date.now();
16         // start processing.
17         cap.read(frame);
18         fgbg.apply(frame, fgmask);
19         cv.imshow('canvasOutput', fgmask);
20         // schedule the next one.
21         let delay = 1000/FPS - (Date.now() - begin);
22         setTimeout(processVideo, delay);
23     } catch (err) {
24         utils.printError(err);
25     }
26 };
27 // schedule the first frame.
28 setTimeout(processVideo, 0);
```

(a) Processing a frame



(b) Processing another frame

Figure 4: Demonstration of background subtraction example on a video

5 PERFORMANCE EVALUATION

This section presents performance evaluation of OpenCV.js. Our evaluation includes both primitive kernels that perform simple operations such as pixel-wise addition or convolution, and more sophisticated vision applications. Our selected vision applications include implementation of **Canny**'s algorithm for finding edges [8], finding **face**s using Haar cascades [9], and finding **people** by using histogram of gradients as features [10]. 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. Figs. 5 and 6 show the performance of simple kernels and vision applications compared to their native equivalent that use OpenCV scalar build (not using parallelism). Experiments are repeated for different pixel types that is supported by the benchmarks. As it shown, in all cases, the performance is close to the native which is impressive for JavaScript. While we found WASM and asm.js performance to be close, WASM version of library is significantly faster to initialize and is more compact (5.3 MB vs 10.4 MB).

6 PARALLEL PROCESSING IN OPENCV.JS

Computer vision is computationally demanding, since a lot of computations need to be performed on massive number of pixels. For instance, each iteration of Canny, face, and people benchmarks take on average 7 ms, 345 ms and 323 ms to process an image with resolution of "640x480" respectively. While Canny is fast enough to be computed in real-time, face and people detection examples are very expensive and cannot be used in real time and interactive use cases. Fortunately, computer vision algorithms are inherently

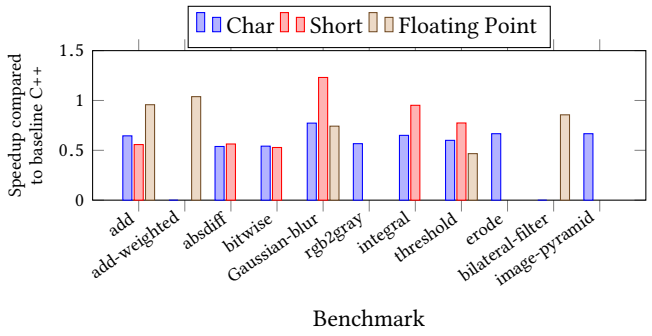


Figure 5: Performance comparison of native and WebAssembly versions of primitive kernels

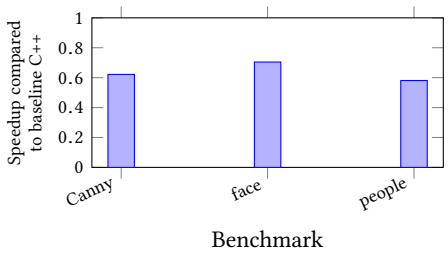


Figure 6: Performance comparison of native and WebAssembly versions of vision applications

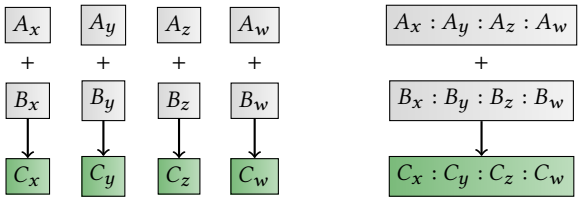


Figure 7: Scalar vs SIMD addition of four integers

parallel, and with good algorithm design and optimized implementation, big speedup can be achieved on parallel hardware. OpenCV already comes with parallel implementation of algorithms for different architectures. We take advantage of two methods that target multicore processors and SIMD (Single-Instruction-Multiple-Data) units to make the JavaScript version faster. We have skipped GPU implementations at the moment due to lack of an standard web API for general purpose programming on GPUs.

6.1 SIMD.js

SIMD.js [11, 12] is a new web API which exposes processor vector capabilities to the web. SIMD.js is based on a common subset of Intel SSE2 and ARM NEON instructions sets that runs efficiently on both architectures. They define vector instructions that operate on 128-bit wide vector registers. A vector register for instance can be used to hold four integers, four single precision floating points, or sixteen bytes. Fig. 7 shows how vector registers can be utilized to add four integers using one vector instruction.

SIMD is proven to be very effective to speedup multimedia, graphics and scientific applications [12–14]. In fact, many OpenCV functions including core routines, are already implemented using vector intrinsics [13]. We have adopted the work done by [15] to translates OpenCV vectorized implementations using SSE2 intrinsics into JavaScript with SIMD.js instructions. Inclusion of SIMD.js implementation will not affect the library interface. Fig. 8 shows the speedup that is obtained by SIMD.js on selected kernels and applications running on Firefox. Up to 8x speedup is obtained for primitive kernels. As expected, speedup is higher for smaller data types. There is less vectorization opportunities in complex functions such as Canny, face and people detection. 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 WebAssembly is planned to have the same spec as SIMD.js, similar performance numbers are expected.

6.2 Multithreading using web workers

JavaScript programs use web workers [16] for parallel processing of compute-intensive tasks. Web workers communicates by passing message which incurs significant cost for large messages such as images. SharedArrayBuffer [17] is recently proposed as a storage that can be shared between multiple web workers. It can be used to implement shared memory parallel programming model. OpenCV uses its' "parallel_for_" framework to implement parallel version of vision functions that can target different multithreading models including POSIX threads (pthreads). With recent Emscripten developments, we were able to translate pthreads API into equivalent JavaScript using web workers with shared array buffers. OpenCV.js with multithreading support, will have a pool of web workers and allocate a worker when a new thread is being spawn. In addition, it exposes OpenCV API to dynamically adjust the concurrency such as changing 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 numbers of workers up to 8. OpenCV load balancing algorithm divides the workload evenly between threads. As shown in Fig. 9, on a processor with 8 logical cores, between 3 to 4 times performance speedup is obtained. Note that similar trend is observed on native pthreads implementation of the mentioned functions.

7 AVAILABILITY

OpenCV.js is already landed to the official OpenCV repository (https://github.com/opencv/opencv/). This will ensure functionality and compatibility of OpenCV.js in future OpenCV releases. Since browser support for SIMD.js and shared web workers is not available yet, parallel processing features is not enabled yet. We have developed expansive online resources to help developers and researchers learn more about OpenCV.js and computer vision in general that can be accessed at https://docs.opencv.org. OpenCV.js can also be used in Node.js based environments. It is published

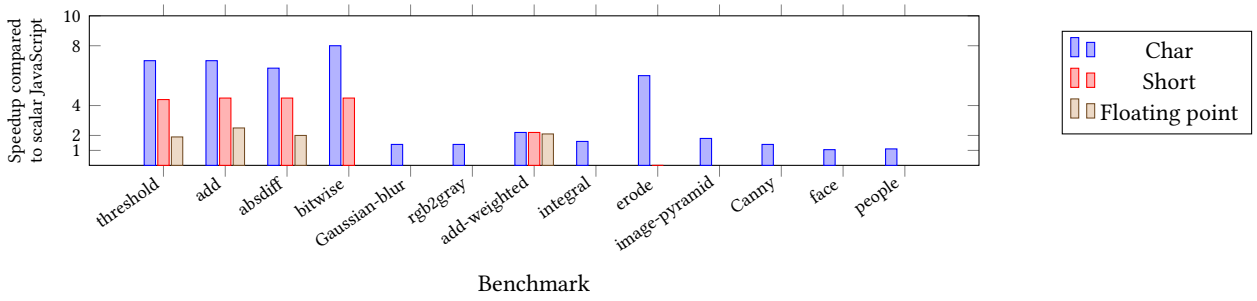


Figure 8: Performance improvement using SIMD.js (asm.js)

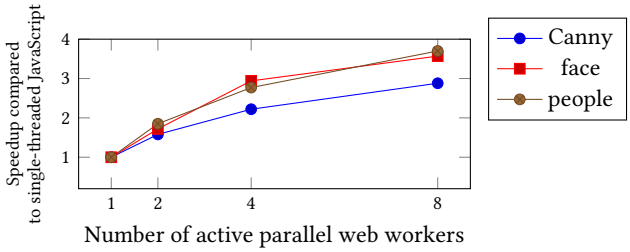


Figure 9: Speedup achieved using multiple Web workers (asm.js)

on the Node Package Manager (NPM) at: https://www.npmjs.com/package/opencv.js.

8 CONCLUSION

This work brings years of OpenCV development in computer vision processing to the web with high efficiency. It provides a collection of carefully selected computer vision functions ranging from image processing, object detection, video analysis, features extraction, and deep neural networks. Experimental results demonstrate high capability of the developed framework. Thanks to JavaScript portability, for the first time, a large collection of vision functions can be used not only on web browsers but also on Node.js based embedded systems and cross-platform Desktop development (e.g., Electron). We believe it will be a great asset for many emerging web applications and experiences including virtual and augmented reality. In addition, we have provided a large collection of computer vision tutorials using OpenCV.js that we hope will be helpful for education and research purposes.

9 ACKNOWLEDGMENTS

This work is supported by the the Intel Corporation. We are grateful to Congxiang Pan, Gang Song, and Wenyao Gan for their contributions through Google Summer of Code (GSoC) program. We also would like to thank OpenCV community for their support and helpful feedback.

REFERENCES

[1] 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.
[2] Eugene Zatepyakin. Jsfeat-javascript computer vision library: https://github.com/inspirit/jsfeat. Accessed: 2018-15-4.
[3] Eduardo Lundgren, Thiago Rocha, Zeno Rocha, Pablo Carvalho, and Maira Bello. tracking.js: A modern approach for computer vision on the web: https://trackingjs.com, 2016. Accessed: 2018-15-4.
[4] Masatoshi Hidaka, Yuichiro Kikura, Yoshitaka Ushiku, and Tatsuya 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.
[5] Miguel Casas-Sanchez. Accelerated shape detection in images: https://wicg.github.io/shape-detection-api/, 2018. Accessed: 2018-15-4.
[6] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.
[7] 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.
[8] John Canny. A computational approach to edge detection. In *Readings in Computer Vision*, pages 184–203. Elsevier, 1987.
[9] 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.
[10] 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.
[11] Simd.js specification: http://tc39.github.io/ecmascript_simd/. Accessed: 2018-15-4.
[12] Ivan Jibaja, Peter Jensen, Ningxin Hu, Mohammad R Haghighat, John McCutchan, Dan Gohman, Stephen M Blackburn, and Kathryn 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.
[13] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Real-time computer vision with opencv. *Communications of the ACM*, 55(6):61–69, 2012.
[14] Sajjad 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.
[15] Peter Jensen, Ivan Jibaja, Ningxin Hu, Dan Gohman, and John Mc-Cutchan. Simd in javascript via c++ and emscripten. In *Workshop on Programming Models for SIMD/Vector Processing*, 2015.
[16] Web workers: https://www.w3.org/TR/workers/, 2015. Accessed: 2018-21-2.
[17] Ecmascript 2018 language specification: https://tc39.github.io/ecma262/, 2017. Accessed: 2018-15-4.