Measuring Opencv.js performance with Wasm execution engine in desktop, embedded and mobile browsers

Carlos A. Pérez[1]

CINAPTIC, Applied Research Centre of Information Technologies, Universidad Tecnológica Nacional, Facultad Regional Resistencia. French 404, (3500) Resistencia, Chaco, Argentina.

cperez@rec.utn.edu.ar

Abstract. Current browsers have sophisticated execution environments for Javascript, and fast rendering engines. With the advent of HTML5, they accept digital cameras, and they can process, in real time, video streaming between browsers, allowing instant communications. In addition, the introduction of the low-level virtual machine (LLVM) allows image-processing libraries to be delivered, alongside web pages, as specialized scripts that execute in browser, with significant speed gains when compared to traditional Javascript engines. This make the browser a very suitable platform to deliver web applications with heavy image processing tasks, that execute at native speeds. However, measuring such performance in modern browsers is a demanding challenge. In this paper, a set of recommended practices to use and to benchmark Opencv.js are presented and obtained figures on several testbeds are discussed. Measurements involved a desktop PC, a selection of smartphones with mainstream processors, and a Raspberry Pi single-board computer, which resulted in several findings that confirm the maturity of mobile an embedded browser for image-processing with Javascript at client side, running at native speeds.

Keywords: devices, Javascript, web assembly, OpenCV, performance

1 Introduction

Nowadays, modern browsers are complex software facilities, that not only render web pages, but also feature intricate internals in order to execute scripts, support multithreading, use hardware acceleration, perform sophisticated compilation, accept several media formats and offer compatibility with latest internet standards. The last addition is the ability to run scripts at near-native speed, or even exceed native speed, with *web assembly*, a compact binary representation that is heavily optimized [1].

Opencv.js is the Javascript version of the well-known OpenCV written in C/C++. In order to achieve the fastest possible performance in browsers, is built upon the new LLVM [2] approach, which takes OpenCV source code and emits intermediate LLVM representation called *bitcode*. A second compiler, Emscripten [3], takes bitcode and generates a code that can be executed directly in browsers, in two possible formats: *asm.js* and web assembly *Wasm*. Asm.js is a subset of Javascript, a highly optimizable

code in runtime because it uses ahead-of-time compilation techniques, achieving near-to-native execution speed. In addition, asm.js can be further optimized by means of Bynarien [4] compiler, that emits Wasm, a size-efficient and portable binary format aimed to achieve native execution speeds. Web assembly should not be perceived as a separate technology from Javascript, but as an extension of script execution engine in the browser.

In this work, we present some methods, findings and cautions that must be observed in such measurement tasks, specially in browsers with memory and processor limitations, like the ones used in embedded or mobile systems. This paper is structured as follows: section 2 present techniques to measure events in browsers, section 3 present some Javascript techniques used to deal with **opencv.js** large script, section 4 present obtained results, and in section 5 we present our conclusions.

2 Measurement techniques in browsers

Regarding event measurement, modern browsers have two domains: the pre-render domain, when scripts are still not completely loaded, and scripting domain, when Javascript execution engine has access to all DOM objects and exposed script methods.

2.1 Measuring events in pre-render time.

Elapsed time consumed with downloadable resources can be monitored using `window.performance` object, but this forces to add some Javascript custom code. In pre-render times, a browser built-in facility is preferable to be used. In Chromium-based browsers, this is called Performance Event Recorder. This allow to register any event raised by browser, which is logged into memory, including every frame of video stream. When events of interests are finished, recording is stopped and a JSON document is created by the browser. Since browser can record a very significant number of events per second, page operation is noticeably slower when in debug mode. This tool is often referred as a *profiler* and obtained JSON document is called a *profile*. Since it can reach hundreds of mebibytes, is almost impossible its usage in mobile and embedded devices.

To load big size script files, such as opencv.js, there are two methods, synchronous and asynchronous. Synchronous method is a blocking one, since script suspends execution until the whole script is loaded. However, this can lead to a poor user experience, especially with files that big. To improve user experience, it is recommended to use an asynchronous method, which is non-blocking. As with all **async** operations, a callback must be specified to notify the user the script finished loading, as follows

```
<script async src="../js/opencv.js"
     onload="OpencvOK();" type="text/javascript">
</script>
```

The callback `OpencvOK()` must be present in same page to flag a correct loading. In fast browsers, script location relative to page document is not relevant. However, in our tests, best results were obtained, for Raspberry Pi (Raspbian) browsers, if script markup is located at the bottom of page.

Since this is a pre-render time, the Performance Event Recorder (PER) must be used. In Network pane of PER windows, **opencv.js** is easily noticeable, an example is seen in **Fig. 1**, using Brave browser (Chromium-based) for Windows.

Fig. 1 - Opencv.js loading time

Immediately after loading, opencv.js script is evaluated. Script compiling is also carried at the same time, as seen in **Fig. 2**, evidencing the *ahead-of-time* compilation technique, a distinctive feature of **asp.js** and **Wasm**. However, since opencv.js is delivered in *wasm* mode, that is, already in compiled format, browser compilation times are negligible.

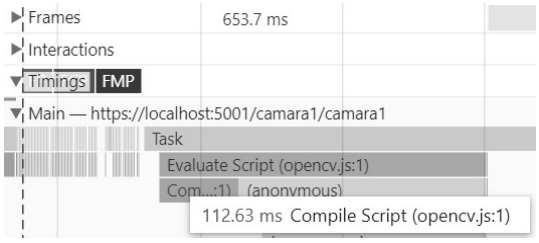

Fig. 2. Script compilation time

2.2 Measuring Javascript events

Obtaining a proper measurement of events during script execution is a tough challenge. To solve this, a High-Resolution Time API [5] was introduced in 2018, authored by W3C. This provides a time origin, and current time in sub-millisecond resolution, such that it is not subject to system clock skew or adjustments. Sub-millisecond resolution is needed in several scenarios, such as calculation of frame rate in script-based animation or cueing an audio segment to a specific point, in order to synchronize audio. A new object with corresponding method was introduced, `performance.now()`, that returns a `float` datatype. This object is tied to a monotonically increasing clock, which is not subject to system clock modifications or clock skew. This ensures proper sign in time subtractions: a newer event minus an older one will always yield a positive value, which is not ensured using `Date.now()`. For instance, to obtain elapsed time of `cv.cvtColor()` method execution, prior and last values of `performance.now()` must be recorded before and after method execution.

3 Recommended practices for opencv.js script

Large scripts are, in general, quick loaded in modern PCs. Loading performance, however, largely depends on network speed when dealing with regular websites and depends on disk transfer rate if script file is locally stored. Due to intrinsic asynchronous nature of current HTTP operations in modern web applications, the latter is quite

noticeably in slow browsers, running in small devices. For instance, evaluating a script can take several seconds in Chromium browsers on Raspberry PI devices, the platform of choice for IoT projects due to its open source nature. On this device, an execution, of another script, can start before the ending of main script evaluation, yielding an unexpected behavior, worsen by absence of visible error messages: the page simply does not work as expected by several seconds.

To ensure that main script is called only when is available in browser internal DOM tree, following practices are recommended.

The script that deals with **opencv.js** functions must be defined as an **async** method (not to be confused with **async** clause in `<script>` tags). For instance, suppose that the page loads **opencv.js** in sync mode, in order to avoid desynchronization problems in slow browsers. This is not enough to prevent null pointers or unexpected behavior when executing main custom script. Suppose that main script is called `index()`, the entry point of Javascript execution in page. The method must be declared as async as follows

```
async function index(){}
```

Any **async** method must **await** any time-consuming operation. For instance, the following code waits for the webcam to be ready:

```
const stream = await navigator.mediaDevices.
                  getUserMedia(constraints);
```

To ensure that main `index()` method is not executed until opencv.js script has been properly pre-processed, **cv** object exposes **onRuntimeInitialized** event, that is handled by an inline callback or anonymous function, which in turns calls the **async** main method `index()`. Namely, the first executable line of the page should be as follows, making the whole script asynchronous, which ensures proper timing in slow devices.

```
cv['onRuntimeInitialized'] = () => { index(); }
```

4 Browser performance with opencv.js

To measure opencv.js operating speed, the following benchmark was arranged, using Javascript and opencv.js. Image was captured using USB webcam. HTML5 video and canvas objects were resized to 640x480 pixels. Once the video stream started, each frame was processed with an RGB-to-grayscale conversion by calling corresponding opencv.js function. Only first 100 captured frames were processed. At each frame, its corresponding elapsed time (partial time) was measured using Performance API. Average, minimum and maximum times were computed, and partial times were accumulated to get total time for whole set. Once the loop is exited, script can build a text file, delimited with blanks, which can be downloaded for analyzing and graphing purposes.

In order to take in account specific characteristics for each platform, a desktop PC, a Raspberry Pi and three smartphones were used. PC was equipped with an Intel i7 processor 7700HQ, running Windows 10 Pro, with five mainstream browsers: Brave, Chrome and Microsoft Edge Developer as Chromium-derived browsers, and Mozilla Firefox and Microsoft Edge as other browses. Three different smartphone models were

selected to reflect differences among processor families: A Nokia 6.1 with Qualcomm Snapdragon 636 (2018 midrange processor), a LG V20 with Qualcomm Snapdragon 821 (2016 high-end processor), and a Samsung Galaxy with Qualcomm 845 (2018 high-end processor) were used as clients. Table 1 shows devices, operating systems and browsers used for benchmarks. For web server, a Raspberry Pi model 3B+ was used, which hosted web application that generates the pages that performed measurements.

Based on time per each frame, a chart with average convergence in time was plotted, which exhibited a sort of stabilization curve, asymptotical to a final, steady value. This allowed to define two phases: a transient phase, when page is loaded and stream begins to be processed, and a steady-state phase, when average elapsed times falls into the band of 5%. This standard establishes that, when an output measure, variable in time, falls within a band defined by an upper bound of 105% and a lower bound of 95%, with respect to the final value, and stays within those limits in a permanent way, it is said that the system has entered in steady state.

Fig. 3 shows PC evolution of average elapsed time per frame. In this graph, findings are: Chrome browser was slowest (2.6 ms) but exhibited the fastest convergency: after only 2 frames Chrome entered steady state. Firefox measurements are always rounded to 1 ms, due to lack of support for Performance API, which difficult a proper comparison. Chromium-base browsers performed similarly in final figures (around 1.4 ms) and exhibited long transient states (more than 90 frames). MS Edge browser was the all-around top performer, with 0.714 ms per frame, and a transient time of about 40 frames. Chrome and Edge exhibited a very stable performance frame after frame, while Chromium-based and Firefox browsers were more sensitive at external events in the host system, showing more deviation in a frame-by-frame basis.

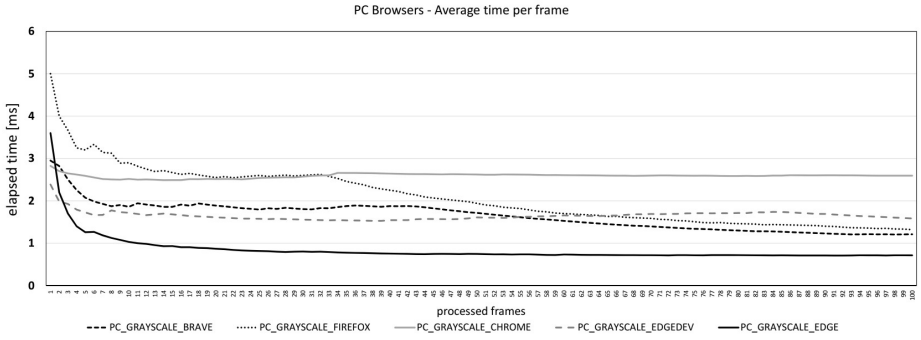

Fig. 4 PC opencv.js average time per frame evolution

Fig. 5 shows **opencv.js** performance with smartphone browsers. Web pages were the same than in PC benchmarks (Raspberry Pi web application). In smartphones, browsers were stock Chrome, Brave and Mozilla Firefox. Some findings are as follows. Mobile browsers do not exhibit a consistent behavior, fastest systems do no guarantee fastest execution, e.g., LG V20 running Firefox, started 3 times slower than the same browser running in Nokia 6.1, which is a midrange phone. This could be due to overhead caused by custom UI layer in V20, that is absent in "pure" Android devices like the Nokia. In addition, operating system optimization seems to be a key component, since Nokia featured an Android v.9.0 (One), whereas the much powerful V20 ran an

older v.8.1. With Firefox, even the RPI system exhibited acceptable transient times (31 frames), whereas LG V20 transient time was almost 100 frames. The fastest system was Samsung S9+, where all browsers yielded almost same figures. On Nokia 6.1, all browsers exhibited excellent transient times, less than 20 frames. The fast-overall browser for all mobile systems, despite processor speed, was Brave Browser.

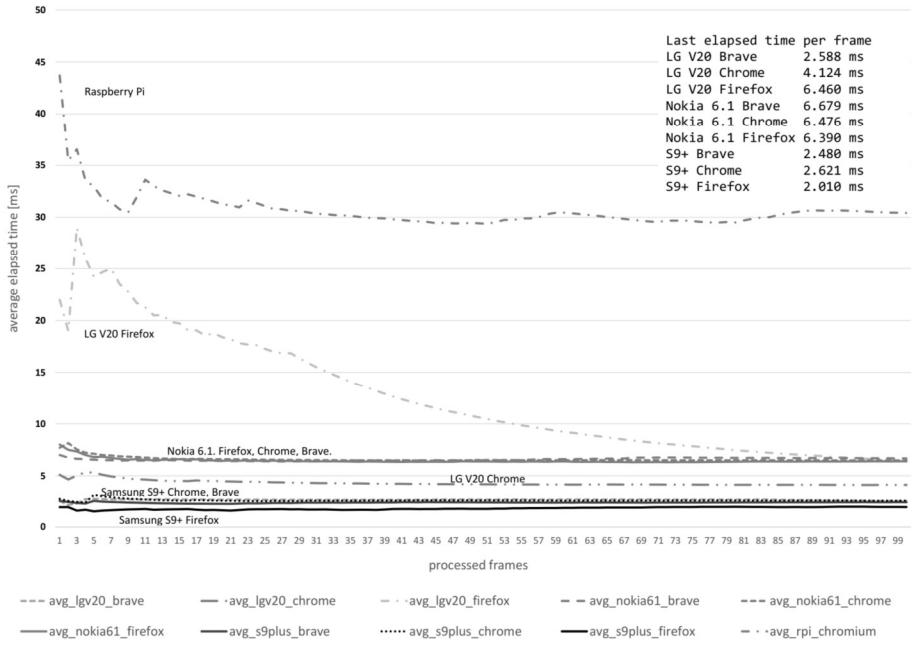

Fig. 5 smartphone and embedded opencv.js average performance evolution

5 Conclusions

In this paper, good practices to program Javascript with opencv.js computer vision library were presented, in order to get a common code for desktop, mobile an embedded browser. Benchmark techniques and cautions to measure opencv.js basic operations were also presented, with a discussion of obtained results. Modern browsers have come to a mature age, yielding fast operation, solid and repeatable performance, low transient times and dependable operation. Raspberry PI board, despite its limited computing resources, can manage significant processing load with opencv.js, that allows even to process real-time video tasks. Brave browser, a chromium-based version, exhibited best figures for overall performance in a variety of mobile and desktop platforms, and it can fulfill every image processing task with opencv.js. In conclusion, web pages with opencv.js arise as the programming platform of the future, given the ease of programming, wide availability and compatibility with internet standards.

References

[1] A. Haas, A. Rossberg, D. L. Schuff and B. L. Titzer, "Bringing the Web up to Speed with WebAssembly," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, Barcelona, Spain, 2018.

[2] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization. CGO 2004.*, San Jose, CA. USA., 2004.

[3] Emscripten Contributors, "Emscripten compiler," [Online]. Available: https://emscripten.org/docs/tools_reference/emcc.html. [Accessed 30 04 2019].

[4] Github, "Webassembly/Binaryen," Open source software, [Online]. Available: https://github.com/WebAssembly/binaryen. [Accessed 30 04 2019].

[5] W3C, "High Resolution Time Level 2," 01 03 2018. [Online]. Available: https://www.w3.org/TR/hr-time-2/. [Accessed 30 04 2019].

[6] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," in *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*, Portland, OR. USA., 2011.