Performance and analysis of acceleration methods in JavaScript environments based on simplified standard Hough transform algorithm

Damian Koper and Marek Woda

Faculty of Information and Communication Technology,
Wroclaw University of Science and Technology
kopernickk@gmail.com, marek.woda@pwr.edu.pl

Abstract. In this paper, we present analysis of acceleration methods of choice in JavaScript execution environments including browser, Node and Deno. We focus evenly on adopting the same codebase to take advantage of every method, benchmarking our solutions and analysis of a toolchain building libraries as compatible as possible with multiple environments. To compare performance, we use a simplified standard Hough transform algorithm with threshold as voting phase. As a reference points of our benchmarks, we use sequential version of the algorithm written in both JavaScript and C++.

Keywords: javascript · acceleration · sht · standard hough transform · node · browser · deno · webgl · webpack · wasm · simd · workers · coldstart

1 Introduction

JavaScript(JS) has become one of the most widely used programming languages in the world [36]. However, it has not been widely adopted to perform scientific computing and data analysis by a community and maintainers for a long time. This includes the development of libraries and features of the language itself. JavaScript is a dynamically typed scripting language, evaluated in the runtime and has only single thread available to the developer in general. Released in 1995, it was executed only in an isolated sandbox environment of a web browser [33] and was used mostly to add dynamic content to websites.

This has changed since more JS environments were released which was allowed by an evolution of JS engines. With the release of Node [7] in 2009 we were able to run server-side code using V8 engine [10] from the Chromium project [1]. At this point, there were at least two ways of building libraries - one for the browser environment (AMD [16]) and one for the CommonJS (CJS) [2] format used by Node. With the release on ECMAScript 2015 [12], better known as the ES6 standard, the modularity of JS code was standardized as ESModules [4]. They are now supported by the vast majority of environments but with an ecosystem that big (1851301 npm packages as of 2022-01-19 [6]) it is still common to build and publish libraries in legacy formats.

The reason why heavy computing in JS is not a popular solution is complex. First of all, slow evolution, the predefined and narrow purpose of the language at first has slowed down the efforts to adapt the language to the task of heavy computation. Secondly, not unified module format and differences between environments in the implementation of corresponding features were not encouraging the community to develop multi-platform libraries. Finally, in the meantime when JS was being developed in its way, other and more popular solutions like MatLab, Python, and R language were already around with their ecosystems ([5], [31], [21]).

In this paper, we check the state of scientific and data analysis multi-platform computing potential in JS. We try to find how the same algorithm performs when implemented using multiple acceleration methods in popular JS environments with as few adjustments as possible for each method. We also want to find if all of these methods can be built into a single output format with the same library interface for maximum compatibility. We hope to find interesting facts and caveats about each analyzed environment and acceleration method.

2 Related work

Analyzing the problem we have to look at it from different perspectives. The first one is the performance of the native JS, specifically the performance of JS engines. We analyze performance of V8 and SpiderMonkey engines focusing on the first one. It was shown that simple optimizations, even changing one line of code, can increase performance due to the JIT optimizations in the runtime ([27], [28], [22], [35]). With known Thread-level speculation, could also be beneficial to the performance [26] but it is not implemented in common JS engines.

The second perspective is the performance of each environment. We analyze performance in web browsers - Chrome and Firefox with V8 and SpiderMonkey [9] as JS engine respectively. We also analyze the same code on server-side environments - Node and Deno [3] with both using V8 as JS engine. Node, written in C++, has proven to be great environment for I/O heavy tasks outperforming other environments ([25], [17]). Deno, written in Rust, is a younger brother of Node providing a modern ecosystem with built-in support for ESModules and TypeScript. It keeps up the performance with Node with insignificant differences [18].

The last aspect is the performance of acceleration methods in each environment. Analyzing CPU-heavy algorithm we want to adjust it and compare its performance against available acceleration methods in each environment. Multithreading is available in each environment in some form of "Worker". Parallel execution increases performance as expected [19] but this performance may differ between environments.

Another way of increasing performance, this time in server-side environments, is to use native plugins which were developed primarily to allow usage of existing codebase written in other languages. Node and Deno use plugins that can be written in C++ [8] and Rust [14] respectively. The same reason for development

stands behind WebAssembly(WASM) [30]. It allows to compile and make any code portable across environments performing better than native JS. Yet the study shows, that depending on the case we can get mixed performance results ([37], [24]).

The last analyzed method is the usage of GPGPU (General-purpose computing on graphics processing units). It involves tricky usage of the WebGL's graphics pipeline not to generate graphics, but using its advantages, executing algorithms that can be massively parallelized [34]. The caveats of this method are slow data transfer between CPU and GPU, and lack of shared memory which is the characteristic of the pipeline itself. There were attempts of trying to standardize API specific to GPGPU (e.g. WebCL) and the most promising today is WebGPU [15].

It is important to mention the building ecosystem provides high compatibility and the possibility of transforming assets. In terms of heavy computing, it matters how the library is built, because some optimization or transpilation techniques may interfere with implementation specific to a particular acceleration method. We use Webpack [11] as a module bundler with ESModule [13] as the desired output for each library built.

3 Benchmarking

We tested performance of mentioned acceleration methods in different environments. Implementation status and reason if not implemented is shown in table 1. Tested JS environments and their versions are described in table 2.

As an algorithm to benchmark these methods against we chose a simplified standard variant of Hough transform(SHT) [29]. Choosing single algorithm over the whole benchmark suite gives us granular control over implementation, building process and adaptation for each acceleration method. Hough transform, in the standard variant, is used to detect lines in binary images. It maps points to values in an accumulator space, called Hough or parameter space. Unlike the original transform [23], modern version maps points (x, y) using polar coordinates (θ, ρ) according to (1) [20].

Table 1: Analyzed acceleration methods and environments.

	Chrome	Firefox	Node	Deno
Sequential	✓	✓	✓	✓
Native addon	✗ [1]	✗ [1]	✓	✗ [2]
asm.js	✓	✓	✓	✗ [2]
WASM	✓	✓	✓	✗ [2]
WASM+SIMD	✓	✓	✓	✗ [2]
Workers	✓	✓	✓	✓
WebGL	✓	✓	✗ [2]	✗ [1]
WebGPU	✗ [3]	✗ [3]	✗ [1]	✗ [3]

[1] Not available in environment
[2] Requires external package or non-C++ codebase
[3] Unstable or under a flag

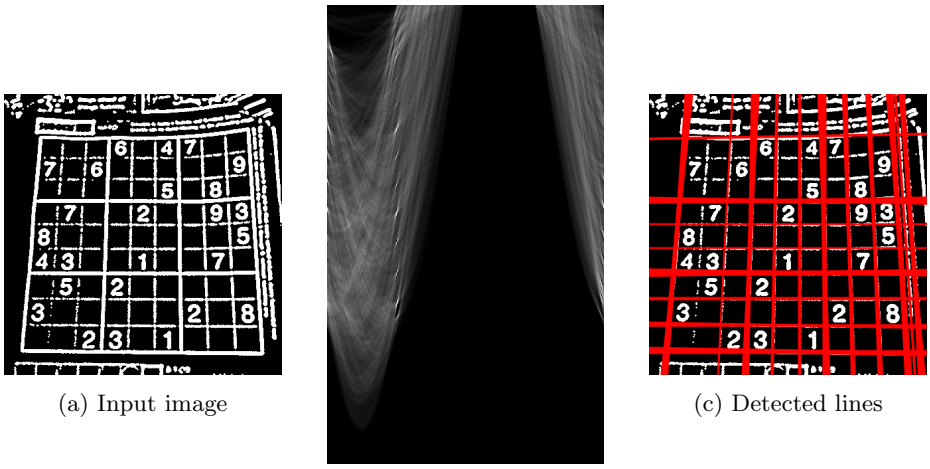
$$f(x, y) = \rho(\theta) = x \cos\theta + y \sin\theta \tag{1}$$

The resolution of the accumulator determines the precision of line detection, the size of the computational problem, and the required memory. The computational complexity of the sequential algorithm equals $O(wh)$ where w and h are dimensions of an input image. It could be also expressed as $O(S_\theta S_\rho)$ with constant input dimensions where S_θ and S_ρ denotes angular and pixel sampling respectively. We benchmark each method for various problem sizes keeping everything constant but S_θ. We implemented simplified version from commonly seen. Our implementation defined the anchor point of polar coordinates in the upper left corner of the image instead of its center and is bases the voting process on a simple threshold instead of analyzing the image space [32].

Table 2: Versions of environments.

Env	Version	Engine version
Chrome	97.0.4692.71	V8 9.7.106.18
Firefox	96.0	SpiderMonkey 96.0
Node	16.13.2	V8 9.4.146.24-node.14
Deno	1.18.0	V8 9.8.177.6

We believe that the algorithm is sufficient for performing valuable benchmarks since it requires many iterations to populate the accumulator and also enforces intensified memory usage for input data and the accumulator itself.



(a) Input image

(b) Accumulator

(c) Detected lines

Fig. 1: Input image, accumulator and visualized result of sequential SHT algorithm in non-LUT variant ($S_\theta = 1, S_\rho = 1$).

TODO: Implemented SHT, Non-LUT and LUT variants

TODO: Env x Methods matrix

TODO: library interface and building method

TODO: Comparing benchmarks across environments and methods, js sequential and C++ as reference point

TODO: JIT and cold/warm start, real-world vs synthetic

- TODO: CoV
- TODO: possibility of function extraction

TODO: Testing environment

- TODO: Hybrid CPU architecture and bench details

References

1. Chromium, https://www.chromium.org/
2. CommonJS, https://nodejs.org/docs/latest/api/modules.html
3. Deno, https://deno.land/
4. ESModules, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
5. Matlab, https://www.mathworks.com/products/matlab.html
6. Modulecounts, http://www.modulecounts.com/
7. Node, https://nodejs.dev/
8. Node.js c++ addons, https://nodejs.org/api/addons.html
9. Spidermonkey, https://spidermonkey.dev/
10. V8, https://v8.dev/
11. Webpack, https://webpack.js.org/
12. ECMAScript 6 (2015), https://262.ecma-international.org/6.0/
13. Automated refactoring of legacy javascript code to es6 modules. Journal of Systems and Software **181**, 111049 (2021)
14. How to create a deno plugin in rust (Sep 2021), https://blog.logrocket.com/how-to-create-a-deno-plugin-in-rust
15. (Jan 2022), https://gpuweb.github.io/gpuweb/
16. Calhoun, D.: What is amd, commonjs, and umd? (Apr 2014), https://www.davidbcalhoun.com/2014/what-is-amd-commonjs-and-umd/
17. Chitra, L.P., Satapathy, R.: Performance comparison and evaluation of node. js and traditional web server (iis). In: 2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET). pp. 1–4. IEEE (2017)
18. Choubey, M.: Deno vs node performance comparison: Hello world (Oct 2021), https://medium.com/deno-the-complete-reference/deno-vs-node-performance-comparison-hello-world-774eda0b9c31
19. Djärv Karltorp, J., Skoglund, E.: Performance of multi-threaded web applications using web workers in client-side javascript (2020)
20. Duda, R.O., Hart, P.E.: Use of the hough transformation to detect lines and curves in pictures. Communications of the ACM **15**(1), 11–15 (1972)
21. Gerrard, P., Johnson, R.M.: Mastering scientific computing with R. Packt Publishing Ltd (2015)
22. Gong, L., Pradel, M., Sen, K.: Jitprof: Pinpointing jit-unfriendly javascript code. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering. pp. 357–368 (2015)
23. Hough, P.V.: Method and means for recognizing complex patterns (Dec 18 1962), uS Patent 3,069,654
24. Jangda, A., Powers, B., Berger, E.D., Guha, A.: Not so fast: Analyzing the performance of webassembly vs. native code. In: 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19). pp. 107–120 (2019)
25. Lei, K., Ma, Y., Tan, Z.: Performance comparison and evaluation of web development technologies in php, python, and node. js. In: 2014 IEEE 17th international conference on computational science and engineering. pp. 661–668. IEEE (2014)
26. Martinsen, J.K., Grahn, H., Isberg, A.: Combining thread-level speculation and just-in-time compilation in google's v8 javascript engine. Concurrency and computation: practice and experience **29**(1), e3826 (2017)

27. Meurer, B.: An introduction to speculative optimization in v8 (Nov 2017), https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8
28. Meurer, B.: Javascript performance pitfalls in v8 (Mar 2019), https://ponyfoo.com/articles/javascript-performance-pitfalls-v8
29. Mukhopadhyay, P., Chaudhuri, B.B.: A survey of hough transform. Pattern Recognition **48**(3), 993–1010 (2015)
30. Nießen, T.: WebAssembly in Node. js. Ph.D. thesis, University of New Brunswick. (2020)
31. Oliphant, T.E.: Python for scientific computing. Computing in science & engineering **9**(3), 10–20 (2007)
32. Palmer, P.L., Kittler, J., Petrou, M.: An optimizing line finder using a hough transform algorithm. Computer Vision and Image Understanding **67**(1), 1–23 (1997)
33. Rauschmayer, D.A.: Speaking JavaScript: An In-Depth Guide for Programmers. O'Reilly (2014)
34. Sapuan, F., Saw, M., Cheah, E.: General-purpose computation on gpus in the browser using gpu. js. Computing in Science & Engineering **20**(1), 33–42 (2018)
35. Selakovic, M., Pradel, M.: Performance issues and optimizations in javascript: an empirical study. In: Proceedings of the 38th International Conference on Software Engineering. pp. 61–72 (2016)
36. StackOverflow: 2021 developer surver (2021), https://insights.stackoverflow.com/survey/2021
37. Yan, Y., Tu, T., Zhao, L., Zhou, Y., Wang, W.: Understanding the performance of webassembly applications. In: Proceedings of the 21st ACM Internet Measurement Conference. pp. 533–549 (2021)