# 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 popular acceleration methods in JavaScript execution environments including Chrome, Firefox, Node and Deno. We focus evenly on adopting the same codebase to take advantage of every method, benchmarking our solutions and caveats of building libraries compatible with multiple environments. To compare performance, we use a simplified standard Hough transform algorithm. As a reference points of our benchmarks, we use sequential version of the algorithm written in both JavaScript and C++. Our study shows that Chrome is the fastest JS environment in every benchmark and Firefox is the slowest in which we identified optimization problems. Overall WebGL appeared as the fastest and without parallel execution native C++ addon in Node is the most performant. We hope that this analysis will help to find the most efficient way to speed up execution making JavaScript a more robust environment for CPU-intensive computations.

**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.

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 CPU-intensive 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 evolving, 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 CPU-intensive computing potential of 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]). 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], 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 Node's younger brother providing a modern ecosystem with built-in support for ESModules. 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. 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 which provides high compatibility and the possibility of transforming assets. 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. We built one library per acceleration method.

## 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 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 to curves $(x, y)$ using polar coordinates $(\theta, \rho)$ according to (1) [20].

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

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 $S_\theta$ and $S_\rho$ denotes angular and pixel sampling respectively. We benchmark each method for various problem sizes keeping everything constant but $S_\theta$. We implemented version simplified from commonly seen ones. Our implementation defines the anchor point of polar coordinates in the upper left corner of the image instead of its center. It also bases the voting process on a simple threshold instead of analyzing the image space [32].

We believe that the algorithm is sufficient for performing valuable benchmarks since it requires many iterations to fill the accumulator and enforces intensified memory usage for input data and the accumulator. Relying heavily on sin and cos functions, also allows us to test their performance. Because of that, we implemented two variants of each algorithm - *non-LUT* and *LUT*. First one uses standard sin and cos functions and second one caches their results in lookup table. As shown in Figure 1, we also use the image after threshold operation instead of commonly used edge-detection. It requires more pixels to be processed thus increases problem size.

Table 1: 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

Each benchmark lasts $5s$ minimum and $30s$ maximum or 50 runs. We want to rely on the most likely execution scenarios which are influenced by JS engine optimizations, resulting in shorter execution time. To resolve this cold start problem we use coefficient of variance metric ($c_v$). We start the actual benchmark after the window of 5 executions where $c_v \leq 1‰$.

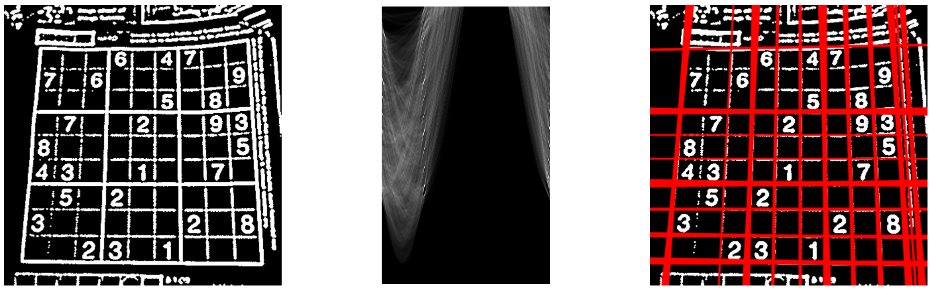
Table 2: Versions of environments.

| 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 |

Benchmarks were performed on a platform equipped with Intel® Core™ i7-12700KF CPU and Nvidia 970 GTX GPU using Ubuntu 20.04.1. The CPU had frequency scaling turned off and due to its hybrid architecture only 4 P-Cores were enabled for a benchmark using `taskset` utility.

## 4 Results and details

In this section, we present execution times for each method depending on angular sampling $S_\theta$ which should result in linear computational complexity. Section 4.1 shows times for sequential execution which is then marked as a grey area for comparison. Every chart contains native C++ times.



(a) Input image (b) Accumulator (c) Detected lines

Fig. 1: Input image (419x421), accumulator and visualized result of sequential SHT algorithm in *non-LUT* variant ($S_\theta = 1, S_\rho = 1$).

### 4.1 Sequential

Analyzing benchmark times shown in Figure 2 we can see the advantage of Chrome over Firefox being 1.6× faster in the *non-LUT* variant and 2.08× in the *LUT* one. It it worth to notice the optimization in Firefox for the $S_\theta = 5$ and subsequent sampling values in the *LUT* variant which could be an object of further research. The performance of server-side environments, Node and Deno, since they are very similar environments, has only insignificant differences, yet still being $(1.50, 1.45)\times$ slower that Chrome for both variants.

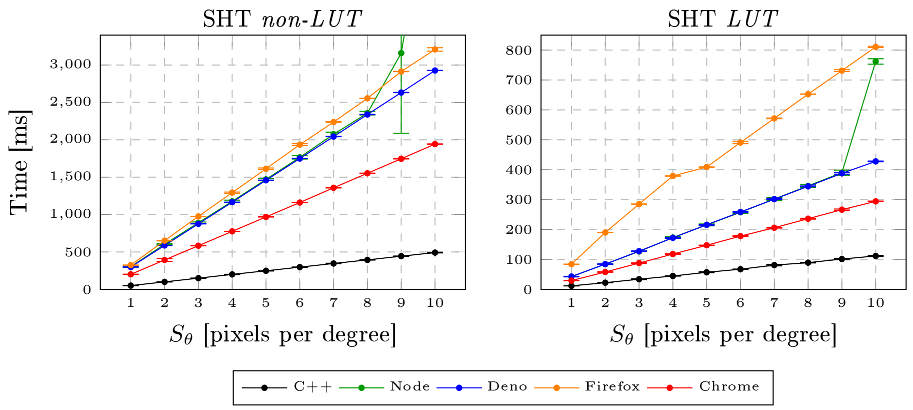
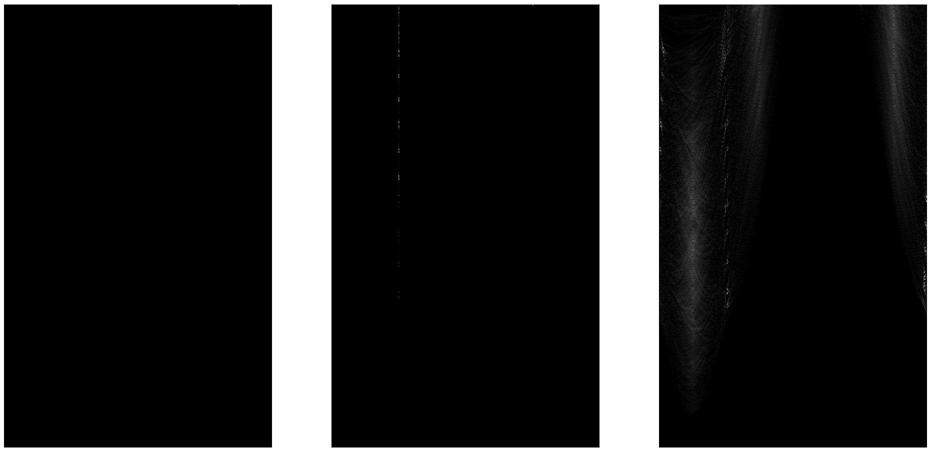


Fig. 2: Sequential SHT execution benchmark results.

We detected one pixel difference in generated accumulator between variants as shown in Figure 3a (upper right corner). We implemented lookup table for *LUT* variants using `Float32Array`. JS internally, without optimizations, represents numbers in double-precision and reduced precision of cached values can have significant impact on detection results.



(a) Sequential $LUT$ (b) WASM (c) WebGL

Fig. 3: Normalized absolute accumulator difference from sequential *non-LUT* variant.

### 4.2 Node C++ addon

C++ addon for Node was built using the same shared library as the C++ version. Thus the difference in performance between native C++ and the addon arise mostly from handling data transfer between C++ – JS boundary since the data needs to be copied and transformed to corresponding C++ structures. Results are shown in Figure 4.

Compilation with optimization of trigonometric functions in *non-LUT* variant allowed to gain more performance (2.21×) than compilation of the $LUT$ variant (1.38×) relative to their sequential variants. This case allows us to draw a conclusion that if an algorithm has trigonometric functions and output of which cannot be cached beforehand, the usage of the C++ addon in Node is beneficial.

### 4.3 WebAssembly and asm.js

Benchmark results for asm.js and WASM were shown on figures 5 and 6 respectively. In our case asm.js - a highly optimizable subset of JS instructions, operating only on numeric types and using heap memory - is actually slower in all environments than sequential execution. We suspect that it is caused by the building process. Webpack adds its own module resolution mechanisms that prevent part of the bundle with asm.js code from being recognized and compiled

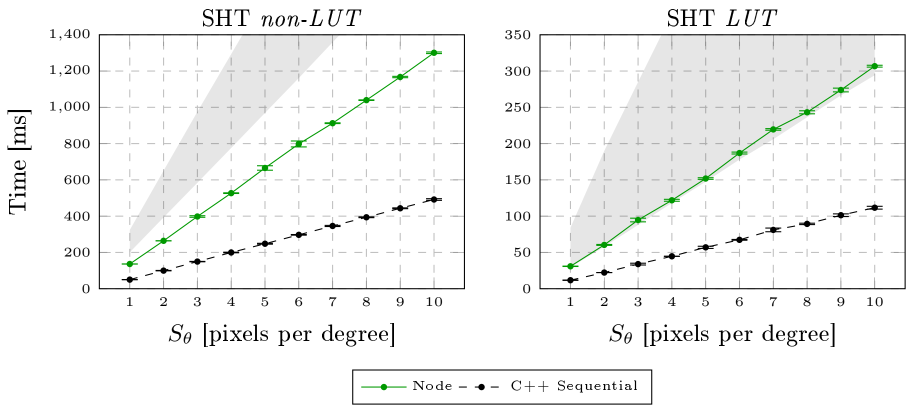


Fig. 4: Node C++ addon SHT execution benchmark results.

ahead-of-time. Performance flame chart from Chrome DevTools tools shows a lack of `Compile Code` blocks, unlike any other isolated asm.js sample.

WASM on the other hand improves performance for *non-LUT* variant and has no effect on *LUT* variant besides preventing optimization mentioned in section 4.1 in Firefox. Again, it is beneficial to use this method if the output if trigonometric functions cannot be cached.

In our C++ implementation we use single precision floating point variables. This results in accumulator differences shown in Figure 3b since WASM distinguishes between `f32` and `f64` types.

## 4.4 WebAssembly SIMD

SIMD instructions in WASM are available from Chrome 91 and Firefox 89 for all users. The usage of SIMD instructions can be done implicitly by letting the compiler (commonly LLVM) perform the auto-vectorization process or explicitly by using vector instructions in code. We tested both solutions resulting in no difference from sequential benchmarks for the first one. Because of that, we present only an explicit usage attempt. In benchmarks shown in Figure 7 we can see that the performance difference between Chrome and Firefox decreased compared to sequential execution and Chrome is only 1.16× faster than Firefox. Moreover, Firefox overtaken Node in performance, which was not as prone to SIMD optimization as other environments.

## 4.5 Workers

All worker benchmarks used concurrency $n = 4$. Results are shown in Figure 8. Because of simplified implementation described in section 3, precisely the center of polar coordinate system in image space, the 3rd worker is redundant. The 3rd vertical quarter of the accumulator will always be empty (Fig. 1b). This was not optimized in our implementation.

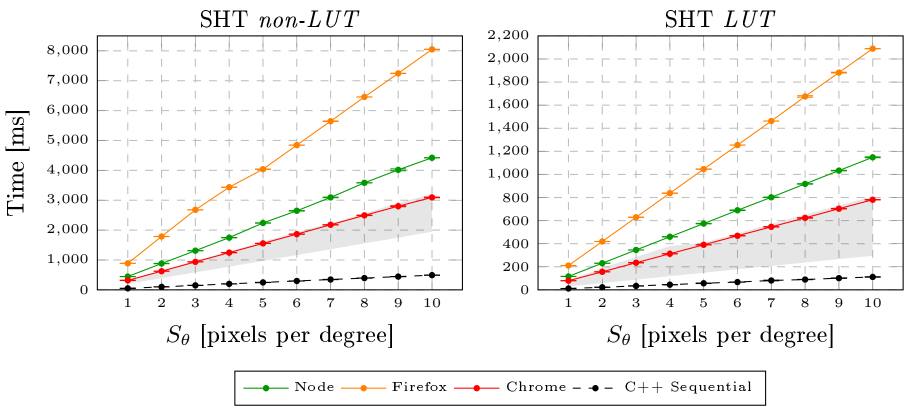


Fig. 5: asm.js SHT execution benchmark results.

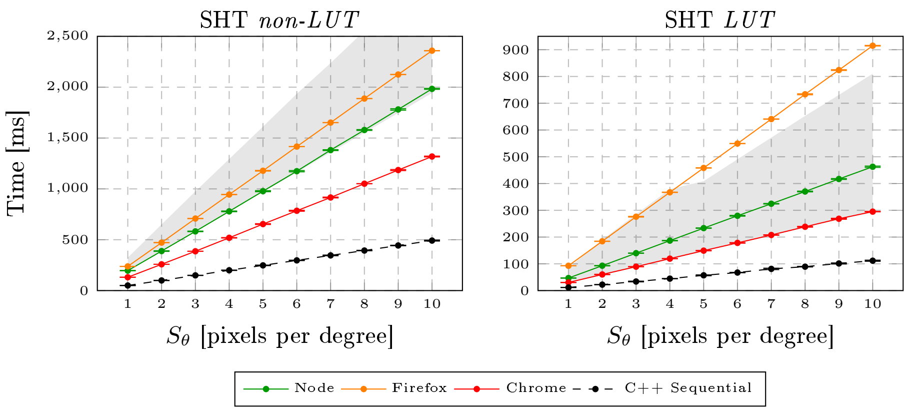


Fig. 6: WASM SHT execution benchmark results.

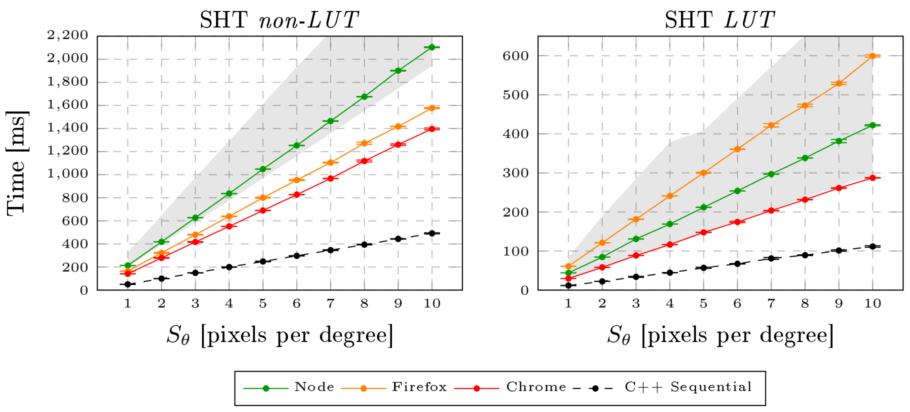


Fig. 7: WASM SIMD (explicit) SHT execution benchmark results.

The table 3 shows the speedup and its efficiency for environments and variants. The big difference in speedup efficiency between variants again shows us how demanding trigonometric functions are. Only the accumulator filling process was parallelized thus the speedup difference between environments is expected since the worker calculations take less time due to the lookup tables. Our implementation can be improved to achieve better performance because the voting process is not parallelized. Even though, the current state of implementation still allows us to compare this method across environments.

Table 3: Speedup metrics for worker acceleration method ($S_\theta = 1, p = 4$).

| Env. | Speedup | Efficiency |
|---|---|---|
| Chrome | 2.99 | 0.75 |
| Firefox | 2.82 | 0.70 |
| Node | 3.25 | 0.81 |
| Deno | 2.70 | 0.67 |
| Chrome *LUT* | 1.85 | 0.46 |
| Firefox *LUT* | 1.89 | 0.47 |
| Node *LUT* | 1.76 | 0.44 |
| Deno *LUT* | 1.51 | 0.38 |

We share the accumulator array between workers using `SharedArrayBuffer` and it is important to mention that our implementation does not use `Atomics` since every worker operates on a different part of the array. According to our benchmarks, `Atomics` tends to slow down performance and were not necessary in this case.

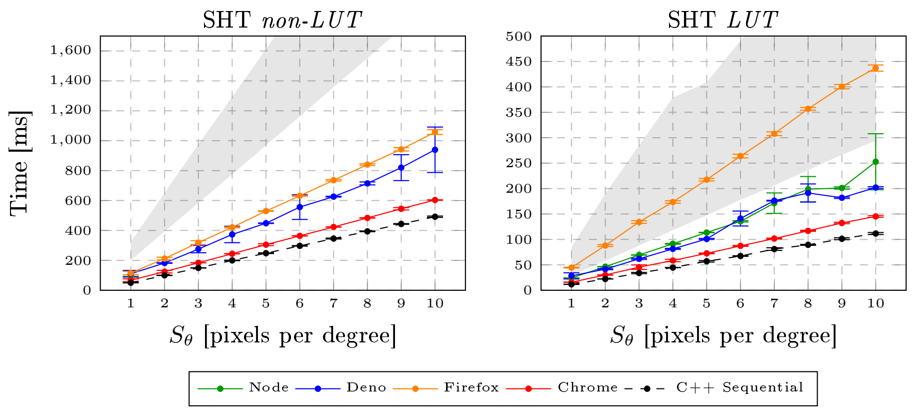


Fig. 8: Workers SHT execution benchmark results with concurrency $n = 4$.

## 4.6 WebGL

Our last acceleration method uses a GPGPU to fill the accumulator array. With help of the WebGL and the gpu.js library, we implemented kernel functions calculating every pixel separately. It is the only possible solution since the WebGL pipeline does not provide shared memory. This results in a bigger accumulator difference shown in Figure 3c. First of all, the pipeline provides only single-precision operations. Secondly, for every accumulator value - pair ($\theta$, $\rho$), we had to sum image pixels laying on a possible line. This operation is prone to rounding errors. Additionally, the minification of output bundle provided by Webpack was interfering with the way the gpu.js library transpiles code to a GLSL language. We had to construct the function from string to prevent minification of the kernel function – `new Function('return function (testImage){...}')()`.

This method has the biggest result variance which comes directly from communication between CPU and GPU. It has also the biggest cold start times since the kernel has to be compiled by an environment on the first run. There is no big difference between both variants because in the *non-LUT* variant each thread on the GPU has to calculate *sin* and *cos* functions once which is not a significant overhead.

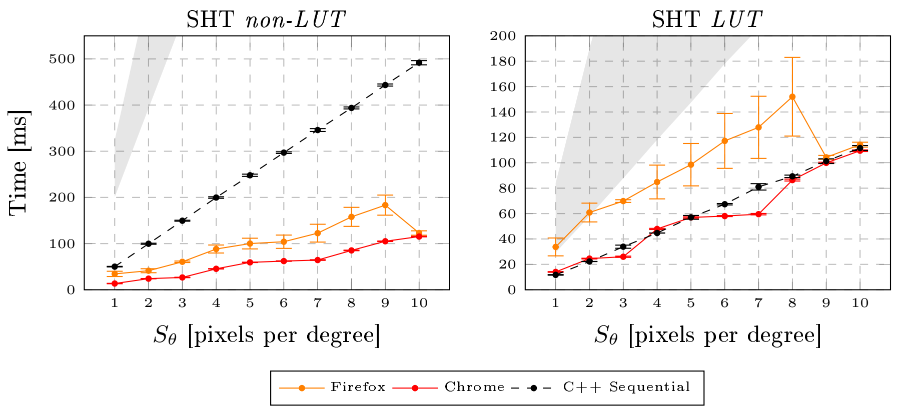


Fig. 9: WebGL SHT execution benchmark results.

## 5 Conclusions

We performed various benchmarks of the same algorithm in Chrome, Firefox, Node and Deno environments (Tab. 2). In each one we tested available and popular acceleration methods including native addon, WASM alone, WASM with SIMD instructions, multi-threading with workers and GPGPU using WebGL graphics pipeline. We did not test every method on every environment because of being unavailable, unstable, under a flag or basing on non-C++ codebase (Tab. 1). Summarized results for the same problem size are shown in table 4. In every benchmark Chrome appears as the fastest environment with Firefox being 2.24×, Node 1.49× and Deno 1.59× slower in general. As expected, without involving parallel execution, the Node C++ native addon brings the best results across all environments. Server-side environments preformed similarly with slight predominance of Node over Deno.

According to our results, the performance of *LUT* variant was always better than its *non-LUT* counterpart. Trigonometric functions are demanding but our study shows that using native addon or compiling code to WASM can prevent significant performance loss, especially in the Firefox environment. We can see that Firefox is not able to optimize code as well as other environments, where using vector instructions explicitly actually lowers performance, increasing it in Firefox. When using lookup tables results may be different. Looking at Chrome performance of all WASM methods with *LUT* variant we can see that performance is roughly the same with the sequential. Data exchanged between JS and WASM must be transformed and copied, which takes time, so it is not safe to assume performance benefit with intensive memory input and output when adopting WASM. We also identified problem with asm.js. which wasn't compiled ahead-of-time. We suspect that the bundling system and minification process prevented environments from recognizing asm.js specific code.

To sum up, all environments serve similar cases but differ in terms of the performance of various acceleration methods. It is important to analyze which method best suits our needs depending on requirements. With all this, it is important to remember that no acceleration method can increase the performance of an algorithm like improvement of the computational complexity of the algorithm itself.

Table 4: Comparison of implemented methods in analyzed environments.

| | | **Execution time[ms]** | | | |
|---|---|---|---|---|---|
| **LUT** | **Method** | **Chrome** | **Firefox** | **Node** | **Deno** |
| | JS Sequential | 200 (1.00) | 324 (1.62) | 302 (1.51) | 298 (1.49) |
| | C++ addon | - (-) | - (-) | 136 (-) | - (-) |
| | asm.js | 317 (1.00) | 887 (2.80) | 444 (1.40) | - (-) |
| | WASM | 131 (1.00) | 238 (1.82) | 196 (1.50) | - (-) |
| | WASM SIMD (impl.) | 130 (1.00) | 239 (1.84) | 197 (1.52) | - (-) |
| | WASM SIMD (expl.) | 141 (1.00) | 164 (1.16) | 214 (1.52) | - (-) |
| | Workers | 67 (1.00) | 115 (1.72) | 92 (1.37) | 110 (1.64) |
| | WebGL | **13** (1.00) | 34 (2.62) | - (-) | - (-) |
| ✓ | JS Sequential | 29 (1.00) | 84 (2.90) | 43 (1.48) | 43 (1.48) |
| ✓ | C++ addon | - (-) | - (-) | 31 (-) | - (-) |
| ✓ | asm.js | 79 (1.00) | 210 (2.66) | 116 (1.47) | - (-) |
| ✓ | WASM | 30 (1.00) | 93 (3.10) | 47 (1.57) | - (-) |
| ✓ | WASM SIMD (impl.) | 30 (1.00) | 93 (3.10) | 47 (1.57) | - (-) |
| ✓ | WASM SIMD (expl.) | 30 (1.00) | 61 (2.03) | 44 (1.47) | - (-) |
| ✓ | Workers | 16 (1.00) | 45 (2.81) | 24 (1.50) | 28 (1.75) |
| ✓ | WebGL | **14** (1.00) | 34 (2.43) | - (-) | - (-) |
| | **Geometric mean (non-LUT)** | (1.00) | (1.87) | (1.47) | (1.56) |
| | **Geometric mean (LUT)** | (1.00) | (2.69) | (1.51) | (1.61) |
| | **Geometric mean (all)** | (1.00) | (2.24) | (1.49) | (1.59) |

## 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)