Performance Evaluation of Image Convolution with WebAssembly

Sou Oishi, Kazuya Ishikawa, Haruki Nogami, and Norishige Fukushima

Nagoya Institute of Technology, Gokiso-cho, Showa-ku, Nagoya, Aichi, 466–8555, Japan.

ABSTRACT

Writing image processing in WebAssembly enables the development of highly portable libraries. However, since WebAssembly is a new technology, there are not enough libraries available, and it is only supported by OpenCV. However, OpenCV has not been optimized specifically for WebAssembly. Therefore, in this paper, we develop native code using WebAssembly and compare it with OpenCV's Gaussian filter using the separable Gaussian filter, a classical acceleration method for Gaussian filters. Experimental results showed that the Separable Gaussian filter was faster than OpenCV by performing vector operations in WebAssembly.

Keywords: WebAssembly, image convolution, acceleration

1. INTRODUCTION

Image convolutions, such as Gaussian filtering, are essential in various image processing applications. Image processing is computationally expensive; thus, the filters have been accelerated by native language, i.e., C/C++, and vectorized and parallelized to extract the performance of the native computers. For example, wavelet,[1] finite impulse response filtering,[2–4] infinite impulse response filtering,[5] recursive filtering,[6,7] local Laplacian filtering,[8] bilateral filtering,[9–11] frequency filtering,[12,13] guided image filtering,[14] and K-means clustering.[15] Each implementation is optimized on CPU characteristics and vector units.

In the past, it was difficult to perform such optimizations on the web platform. First of all, drawing graphics on a web page was generally done by replacing them with native JavaScript with image files such as PNG, JPEG, and GIF, or by embedding plug-in data. However, with the development of HTML5 (W3C's HTML5 is now obsolete, and WHATWG's HTML Living Standard is commonly used, but HTML5 is used here to refer to the period after HTML5 was born) and related technologies, the Web has matured as an application platform. One of the drawing elements added in HTML5 is the canvas tag. The canvas tag enables the quick display of graphics such as graphs and image processing results using only standard HTML and JavaScript.

The next point is the lack of performance that differs from native applications using C/C++. This is why asm.js was created as a subset of JavaScript designed for fast execution. The goal of asm.js was to achieve at least half the performance of native JavaScript, and it was supported by browsers such as Google Chrome and Firefox. The asm.js can convert code written in C or C++ into JavaScript with safety and optimization considerations through the Emscripten compiler, and run it in a browser. While asm.js has enabled improved performance, asm.js has the problem of long loading times. The asm.js takes a long time for parsing (the process by which the browser parses the syntax and changes it into an executable form) before executing JavaScript. Especially on mobile devices, loading 40MB of JavaScript code took from 20 to 30 seconds to execute. Therefore, WebAssembly[16,17] was developed to solve this problem. WebAssembly has improved loading times, is a suitable language for HTML5, and is expected to accelerate image processing on the web, also supports vectorized computing, such as single instruction multiple data (SIMD). Note that it is a new technology; thus there are a few implementations: OpenCV (a well-known image processing library), Photon (https://silvia-odwyer.github.io/photon/), and libvips (https://www.libvips.org/). Therefore, there are few evaluations for image processing in WebAssembly, newly evaluated by numerical linear algebra papers.[18]

OpenCV has published OpenCV.js compliant with WebAssembly, but no comparison has been performed to compare whether their implementations are fast enough. Therefore, in this paper, we accelerate Gaussian filtering as a classical method and compare them with OpenCV.js.[19]

Further author information: (Send correspondence to Norishige Fukushima)
Norishige Fukushima: E-mail: fukushima@nitech.ac.jp. This work was supported by JSPS KAKENHI (21H03465).

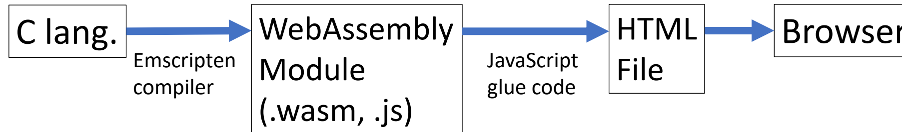

Figure 1. Experimental Flow

2. WEBASSEMBLY

Portability is essential for web applications; thus, it is challenging to use native-tuned code. WebAssembly allows native-tuned code, such as C/C++, to be compiled through Emscripten compiler (https://emscripten.org/), which is a compiler from C/C++ to WebAssembly. for various web applications. In addition, JavaScript's V8 engine supports SIMD, which allows vectorization of processing with CPU power. WebAssembly also supports 128-bit SSE which is one of the vectorizations. However, WebAssembly is not fully executable in the same way as native environments because SIMD instructions are different for each CPU micro-architecture, such as x86 and ARM. For example, some SIMD instructions are not supported, and in some cases, writing in SIMD is slower due to emulation. Another project is currently underway to support SIMD instructions, such as fused multiply-add (FMA), but it is still in the implementation stage. Furthermore, multi-threading, which is indispensable for accelerating performance, is still in the implementation stage, as is SIMD. The enabled SIMD in WebAssembly is implemented as standard in browsers such as Google Chrome and Firefox.

In addition, loading time is an important aspect of WebAssembly development, and another important aspect is that it enables functionality that is not possible with JavaScript processing systems such as a dynamic link. Also, the heap size limit can be relaxed. With JavaScript, each tab of the browser can hold only 4 GB, which is small, so it is possible to use it more widely. The memory management of JavaScript and WebAssembly is completely separate, just as JavaScript manages memory separately for each tab in a browser. Sharing memory between the two, JavaScript and WebAssembly, requires optional settings, and by default, a dedicated area is created in favor of performance and safety. Furthermore, for example, when there is a long array of data if the accessed index is not inside the overall array size, it will cause a security problem. With 64-bit arrays, the hardware points out whether the index is inside the array or not, but with 32-bit arrays, it must be checked. WebAssembly converts the code to one that is easy to check.

Finally, WebAssembly has the following features.

- WebAssembly code can run at native speeds across different platforms, making it fast and efficient.
- WebAssembly is a low-level assembly language, but it has a human-readable text format and is debuggable.
- As with other web technologies, it is secure because it forces the browser to confirm the same-origin and authorization policy.
- It is designed to work with other web technologies and maintain backward compatibility, so it does not break the web.

The processing flow of this paper for image processing on the web is shown in Fig. 1. The code written in C-language is compiled with the Emscripten compiler and converted into WebAssembly code. The code is loaded into HTML as a JavaScript glue and displayed in the browser.

3. IMPLEMENTATION

3.1 Data Sharing

First, the implementation method is described. In JavaScript, the image is retrieved from the HTML canvas tag. The data in the canvas tag is a 4-channel image including RGB and alpha channels, and the data type is

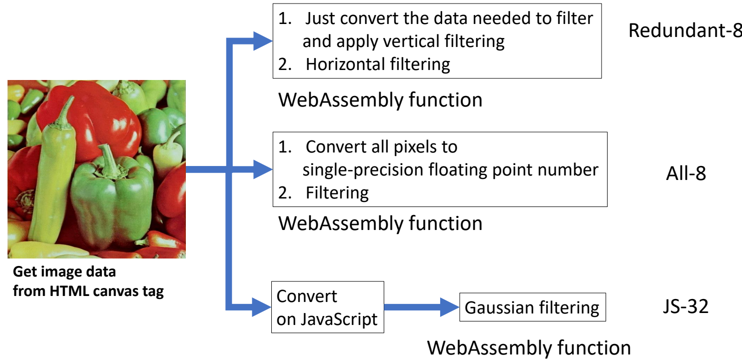

Figure 2. Processing flow of data sharing.

8-bit unsigned integer, which is *Uint8ClampedArray*.[*]. In the process of calculation, 8-bit unsigned integers are cast to single-precision floating-point numbers for vector computing.

There are two candidates in terms of implementation of the data sharing part (Fig. 2): 8-bit or 32-bit sharing. This part copies or converts image data from a canvas to a shared memory by JavaScript operation, which can be a native operation supported by each browser. When 8-bit data is passed, it can be divided into two types depending on when it is cast to 32-bits float data. Therefore, we consider the following three types:

- *Redundant-8*: After passing 8-bit data to the WebAssembly function, we only convert the necessary data for convolution to single-precision floating-point data, redundantly.
- *All-8*: After passing 8-bit data to the WebAssembly function, we first convert all image data to a single-precision floating-point number image.
- *JS-32*: Before passing data, we convert input 8-bit data to 32-bit float before calling the WebAssembly function on the JavaScript side, and then pass the 32-bit data.

WebAssembly supports 128-bit vector operations, such as SSE; thus, single-precision floating-point numbers can be vectorized in four-pixel unit operations. It is suitable for 32-bit RGBA format since one SIMD operation can handle a pixel operation for each color. For comparison, we also performed an implementation without SIMD. Note that we pass the 4 channels of data from the JavaScript side, but the computation was performed excluding the alpha channel. The cast performed on a 128-bit SSE is shown in Fig. 3. First, eight 8-bit signals are converted to eight 16-bit signals and loaded. Next, the upper and lower 4 units of 16-bit data are each converted to 32-bit integers, and finally, the integers are cast to single-precision floating- point numbers.

3.2 Separable Convolution

Let's introduce image convolution, which is optimized in this paper. We use separable convolution, which is one of the typically accelerated convolutions. The method divides a 2-dimensional convolution into two 1-dimensional kernels. As a result, the computational cost is reduced from $O(M \times N \times m \times n)$ to $O(M \times N \times (m+n))$ in the case of an $M \times N$ image. Let Gaussian filtering (GF) be an example. GF is defined as follows:

$$\boldsymbol{g_p} = \frac{\sum_{\boldsymbol{q} \in N_{\boldsymbol{p}}} \exp(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{2\sigma^2}) \boldsymbol{f_q}}{\sum_{\boldsymbol{q} \in N_{\boldsymbol{p}}} \exp(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{2\sigma^2})}, \tag{1}$$

[*]There are two mode: alphamultiply and premultiplied. The former's RGB values are the as-is value, and the latter's RGB values are pre-normalized by alpha value for fast rendering.

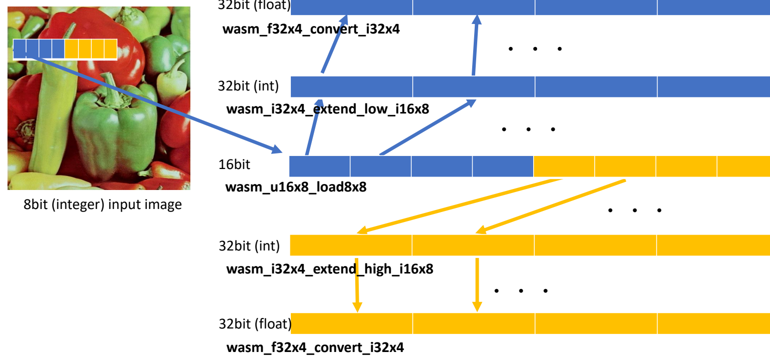

Figure 3. SIMD type conversion from 8-bit integer to 32-bit float.

where $\boldsymbol{f}$ is an input image, $\boldsymbol{g}$ is an output image of GF, $\boldsymbol{p}$ is a pixel position, and $N_{\boldsymbol{p}}$ is the set of the neighboring pixels of $\boldsymbol{p}$. This can be divided as follows:

$$\boldsymbol{g}^V_{\boldsymbol{p}} = \frac{\sum_{\boldsymbol{q}\in N^V_{\boldsymbol{p}}} \exp(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|^2_2)}{2\sigma^2})\boldsymbol{f}_{\boldsymbol{q}}}{\sum_{\boldsymbol{q}\in N^V_{\boldsymbol{p}}} \exp(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|^2_2)}{2\sigma^2})} \tag{2}$$

$$\boldsymbol{g}_{\boldsymbol{p}} = \frac{\sum_{\boldsymbol{q}\in N^H_{\boldsymbol{p}}} \exp(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|^2_2)}{2\sigma^2})\boldsymbol{g}^V_{\boldsymbol{q}}}{\sum_{\boldsymbol{q}\in N^H_{\boldsymbol{p}}} \exp(\frac{-\|\boldsymbol{p}-\boldsymbol{q}\|^2_2)}{2\sigma^2})} \tag{3}$$

where $N^V_{\boldsymbol{p}}$ ($N^H_{\boldsymbol{p}}$) is the set of only vertical (horizontal) components containing $\boldsymbol{p}$ (respectively). Of course, there is no difference in output whether this implementation is done vertically or horizontally. In this implementation, the order of the expressions was used.

In this paper's implementation, a vertical filter is applied followed by a horizontal filter. In addition, for efficient data access, we use an interleaved implementation, i.e., tiling by each scanline. The vertical filtering result is stored in one line buffer and the horizontal filtering is performed for the line buffer. This implementation has enough cache efficiency with full tiling implementation, which requires complex implementation.[20] In the following, the filtering method is shown in Fig. 4. Note that the border area is replicated to the border with the row or column at the edge of the original image.

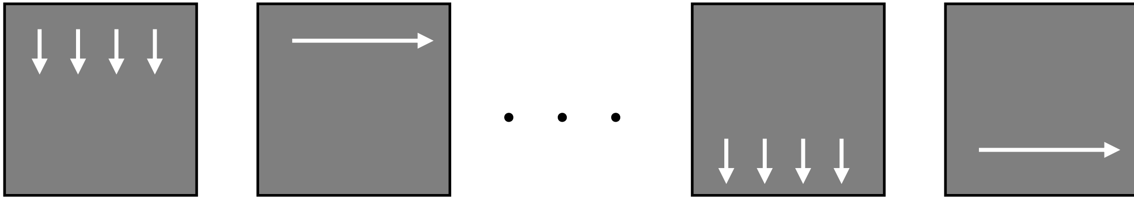

Figure 4. Filtering pass.

4. EXPERIMENTAL RESULTS

We compared the processing time of our WebAssembly implementation of some separable filterings with pre-built OpenCV.js (version: 4.6.0). In addition, since image processing in WebAssembly requires that data be allocated in shared memory between JavaScript and WebAssembly for processing, this paper experimented with three different ways of sharing data.

Our implementation was vectorized by 128-bit SIMD supported by WebAssembly. Note that WebAssembly does not support 256-bit/512-bit SIMD, such as AVX/AVX2/AVX-512. All implementations are not parallelized

(a) Test image (b) Gaussian filtering result

Figure 5. Test image and the result of Gaussian filtering (smoothing parameters: $\sigma = 4$, kernel size is 25×25).

Table 1. The processing time for different implementations of GF [ms]

	OpenCV.js	OpenCV.js built with SIMD	Redundant-8 with SIMD	All-8 with SIMD	JS-32 with SIMD	JS-32 w/o SIMD
Time [ms]	57.36	9.21	27.179	8.524	**8.034**	15.49

because WebAssembly does not support multi-threading, officially. The test CPU was Intel Core i9-9900K CPU @ 3.60GHz. The test image size was 512×512 (Fig. 5(a)). The experimental environment is Windows 10 and Google Chrome (version: 107.0.5304.107, 64-bit). Implementation was done using C-language and compiled into WebAssembly using Emscripten compiler. The image processing parameters were set to $\sigma = 4$, the radius to $r = 3\sigma$, and the kernel size was set to 25×25. The result of Gaussian filtering on Fig. 5(a) is shown in Fig. 5(b).

Table 1 shows the processing time for each type of implementation. The processing time is the average of the results of 1000 trials. First, the implementation of the official OpenCV build was the slowest compared to the others. The official OpenCV version took more than twice as long to compute as that A *non-SIMD* implementation of the JS-32 method. Next, OpenCV.js built with the SIMD option is compared with the implementation. The implementation method resulted in faster results than Redundant-8 and slower results than the other implementations.

Finally, we compare the three shared memory and casting methods. First, the fastest method was to copy to single-precision floating-point data on the JavaScript side (JS-32). This is considered the fastest method because its conversion is optimized on the browser even if the function is called in the JavaScript side. Usually, operations written in JavaScript tend to be slow; however, library functions have optimized binary for each naive environment, which binary is contained for each browser.

The slowest was the method that converts each time before interleaving (Redundant-8). This approach has redundant casting processes because each convolution requires overlapped regions. In the naive environment, usually, this approach is the fastest, because the process is the most cache-efficient implementation. Cache access efficiency gain is greater than the overhead of redundant casts. However, WebAssembly has a large overhead due to insufficient SIMD implementation of casts.

All-8 is the second first method, which is very similar to JS-32. The main difference is conversion operation is supported by WebAssembly or JavaScript. The simple operation of casting all data is supported by native operations which can use 256/512-bit SIMD operations; thus, JS-32 is faster.

5. CONCLUSION

This paper compared OpenCV.js with the implementation in WebAssembly converted from C language by evaluating separable Gaussian filtering. Experimental results showed that even the separable filter, a classical speed-up method for Gaussian filters, was faster than OpenCV.js by using WebAssembly and SIMD. Furthermore,

WebAssembly's SIMD did not support all instructions, and some instructions were emulated, which sometimes resulted in slower performance. Since the convert instruction was not fast in this experiment, it was found that the same idea as in native environments such as C language could not be used to accelerate the performance.

REFERENCES

[1] Sumiya, Y., Kamei, H., Ishikawa, K., Sumiya, Y., and Fukushima, N., "Vectorized computing for edge-avoiding wavelet," in [*International Workshop on Advanced Image Technology (IWAIT),*], **12177**, 23 – 28, International Society for Optics and Photonics, SPIE (2022).

[2] Maeda, Y., Fukushima, N., and Matsuo, H., "Taxonomy of vectorization patterns of programming for fir image filters using kernel subsampling and new one," *Applied Sciences* **8**(8) (2018).

[3] Maeda, Y., Fukushima, N., and Matsuo, H., "Effective implementation of edge-preserving filtering on cpu microarchitectures," *Applied Sciences* **8**(10) (2018).

[4] Fukushima, N., Tsubokawa, T., and Maeda, Y., "Vector addressing for non-sequential sampling in fir image filtering," in [*IEEE International Conference on Image Processing (ICIP)*], (2019).

[5] Fukushima, N., Sugimoto, K., and Kamata, S., "Complex coefficient representation for iir bilateral filter," in [*in Proc. International Conference on Image Processing (ICIP)*], (2017).

[6] Fukushima, N., Maeda, Y., Kawasaki, Y., Nakamura, M., Tsumura, T., Sugimoto, K., and Kamata, S., "Efficient computational scheduling of box and gaussian fir filtering for cpu microarchitecture," in [*Proc. Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2018.*], (2018).

[7] Otsuka, T., Fukushima, N., Maeda, Y., Sugimoto, K., , and Kamata, S., "Optimization of sliding-dct based gaussian filtering for hardware accelerator," in [*Proc. International Conference on Visual Communications and Image Processing (VCIP)*], (2020).

[8] Sumiya, Y., Otsuka, T., Maeda, Y., and Fukushima, N., "Gaussian fourier pyramid for local laplacian filter," *IEEE Signal Processing Letters* **29**, 11–15 (2022).

[9] Sugimoto, K., Fukushima, N., and Kamata, S., "200 fps constant-time bilateral filter using svd and tiling strategy," in [*IEEE International Conference on Image Processing (ICIP)*], (2019).

[10] Sumiya, Y., Fukushima, N., Sugimoto, K., and Kamata, S., "Extending compressive bilateral filtering for arbitrary range kernel," in [*Proc. International Conference on Image Processing (ICIP)*], (2020).

[11] Miyamura, T., Fukushima, N., Waqas, M., Sugimoto, K., and Kamata, S., "Image tiling for clustering to improve stability of constant-time color bilateral filtering," in [*Proc. International Conference on Image Processing (ICIP)*], (2020).

[12] Fujita, S., Fukushima, N., Kimura, M., and Ishibashi, Y., "Randomized redundant dct: Efficient denoising by using random subsampling of dct patches," in [*SIGGRAPH Asia 2015 Technical Briefs*], ACM (2015).

[13] Fukushima, N., Kawasaki, Y., and Maeda, Y., "Accelerating redundant dct filtering for deblurring and denoising," in [*IEEE International Conference on Image Processing (ICIP)*], (2019).

[14] Fukushima, N., Sugimoto, K., and Kamata, S., "Guided image filtering with arbitrary window function," in [*IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*], (2018).

[15] Otsuka, T. and Fukushima, N., "Vectorized implementation of k-means," in [*Proc. International Workshop on Advanced Image Technology (IWAIT)*], (2021).

[16] Rossberg, A., "Webassembly core specification," tech. rep., W3C (2019). https://www.w3.org/TR/wasm-core-1/, https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[17] Rossberg, A., "WebAssembly Core Specification," tech. rep., W3C (2022). https://www.w3.org/TR/wasm-core-2/, https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[18] Bhonsle, A., Patil, V., Valkunde, T., and Lotlikar, T., "Linear algebra in the browser powered by webassembly," in [*International Conference for Advancement in Technology (ICONAT)*], 1–7 (2022).

[19] "opencv.js." https://docs.opencv.org/4.6.0/opencv.js.

[20] Fukushima, N., Fujita, S., and Ishibashi, Y., "Switching dual kernels for separable edge-preserving filtering," in [*Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*], (2015).