

The Need for Speed of AI Applications

Performance Comparison of Native vs. Browser-based Algorithm Implementations

Bernd Malle^{1,2}

Nicola Giuliani¹

Peter Kieseberg^{1,2,3}

Andreas Holzinger¹

B.MALLE@HCI-KDD.ORG

N.GIULIANI@HCI-KDD.ORG

PKIESEBERG@SBA-RESEARCH.ORG

A.HOLZINGER@HCI-KDD.ORG

¹ Holzinger Group HCI-KDD, Institute for Medical Informatics, Statistics & Documentation, Medical University Graz, Austria

² SBA Research, Vienna, Austria

³ University of Applied Sciences St.Pölten, Austria

Abstract

AI applications pose increasing demands on performance, so it is not surprising that the era of client-side distributed software is becoming important. On top of many AI applications already using mobile hardware, and even browsers for computationally demanding AI applications, we are already witnessing the emergence of client-side (federated) machine learning algorithms, driven by the interests of large corporations and startups alike. Apart from mathematical and algorithmic concerns, this trend especially demands new levels of computational efficiency from client environments. Consequently, this paper deals with the question of state-of-the-art performance by presenting a comparison study between native code and different browser-based implementations: *JavaScript*, *ASM.js* as well as *WebAssembly* on a representative mix of algorithms. Our results show that current efforts in runtime optimization push the boundaries well towards (and even beyond) native binary performance. We analyze the results obtained and speculate on the reasons behind some surprises, rounding the paper off by outlining future possibilities as well as some of our own research efforts.

Keywords — Federated machine learning, Client side computing, distributed computing, In-browser computing, JavaScript Virtual Machine (JSVM), VM execution cycle, VM optimizations, parallelization, SIMD, WebWorkers, ASM.js, Web Assembly, GPGPU

1. Introduction & Motivation

For some years now, distributed & cloud computing as well as virtualized and container-based software have been buzzwords in the software engineering community. However, although these trends have helped making software infrastructure generally and AI applications specifically more stable, fault-tolerant, easy to control, scale, monitor and bill - one characteristic has never changed: the computationally most demanding parts of our software's logic, may it be recommenders, intelligent social network analysis or deep learning models, are still handled by enormous data-centers or GPU-clusters of incredible capacity (Armbrust et al., 2010).

On the other hand, exponential hardware evolution has given us small mobile devices more capable than many servers ten years ago, enabling to run successful AI applications (Jordan and Mitchell, 2015), which increasingly interact with the environment and with other smart devices (Moritz et al., 2017) demanding even more performance (Stoica et al., 2017). Following two questions arise:

1. Could it be possible to create a *new* internet, based not on proprietary data access and computation, but purely distributed computing models?
2. Just like data-centers have shifted from specialized supercomputers to grids of commodity hardware, could the logical next step be the involvement of every laptop, tablet, phone and smart device on the planet into our future business models?

The economic incentive is obvious: because access to powerful, centralized hardware will often be out of reach of a small organization, outsourcing the computational costs of an application to a network of clients is reasonable and would furthermore contribute to **vastly superior scalability** of the system, as any centralized server infrastructure could be reduced to its absolute minimum duties: 1) distributing the client-side code, as well as 2) routing, and 3) storing a certain amount of (global) data. Most of all such approaches are of greatest importance for the health domain, where

In order to achieve such grand goals, we need:

- new algorithms that can collaborate without relying too much on global (centralized) data;
- new communication protocols to make such collaboration efficient;
- new mathematical / machine learning models which are statistically robust under distributed uncertainty – but most of all:
- raw power on the client side, not only in hardware but especially in software & its underlying execution environments.

Ironically, one of the most progressive software execution and distribution environments to implement such a future solution is a technology straight from the 90's - the Web Browser. Code written for this universally available runtime carries several advantages: It is highly portable out-of-the-box, any update in the future is trivially easy to deploy, and the hurdle for entry is practically zero as Browsers come pre-installed on every consumer device and operating system.

In addition, the browser's *sandbox model* (code running within has no access to data on the rest of the filesystem) guarantees better privacy than mobile Apps or traditional, installed software; distributed algorithms at the same time force developers to design new models which can efficiently learn on limited data. All of this will make it easier to comply with the ever-increasing demands of data protection regulations on all levels, exemplified by the new European General Data Protection Regulation (GDPR), the right to be forgotten (Malle et al., 2016) and the need for trust (Holzinger et al., 2018).

Finally – due to modern platforms including Apache Cordova – we are able to package web applications into practically any format, may it be iOS or Android Apps or desktop applications. Therefore, in this paper we concentrate on performance tests within the browser’s Virtual Machine. Traditionally this implied JavaScript as the language of the (only) choice; in recent years however alternatives gained attraction - in many cases promising whole new levels of performance and efficiency.

2. Selected Related Work

Fortuna et al. (2010) were pointing out that JS would be suitable as a target language for future compute-intensive applications, given the contemporary rapid evolution of JS virtual machines during the browser wars of the late 2000s. As JS offers no native concurrency, they implemented parallelism on a "task" level, where task is defined as at least one JS OpCode. They explored and contrasted loop-level parallelism to inter- and intra-function parallelism and measured speedups of 2.19 to 45 times (average), with significantly higher factors achieved on function parallelism. Another attempt on dynamic parallelization of JS Applications using a speculation mechanism was outlined in (Mehrara et al., 2011), where the authors used data flow analysis in addition to memory dependence analysis and memory access profiling. However, in real-world, web-based interaction-intensive applications, they were only able to measure a meek speedup of 2.19, in-line with the findings mentioned earlier.

Another attempt at automatic parallelization of JS code was made by (Wook Kim and Han, 2016), who also point out the economic advantage of such an approach for startups and small development shops, as they are usually not well-equipped to write parallel code-bases themselves. Their implementation utilizing static compiler analysis detecting DOALL (loop-level) parallelism enabled multi-CPU execution by emitting JS LLVM IR (intermediate representation), resulting in a speedup of 3.02 on the Pixastic benchmarks as well as 1.92 times on real-world HTML5 benchmarks, requiring a compilation overhead of only 0.215%.

Lower-level (non-thread) parallelism can be achieved via Single Instruction Multiple Data (SIMD), which is a technique to perform one CPU instruction on several pieces of data (streams) simultaneously. Jensen et al. (2015), who also describe the spiral of increasing performance of JS (the Web platform in general) leading to ever more use cases, which in turn lead to higher performance requirements, implemented SIMD.js, introducing CPU-native vector datatypes such as $\text{float32} \times 4$, $\text{float64} \times 2$, and $\text{int32} \times 4$ to the JavaScript language. They were able to achieve speedups between $2\times$ and $8\times$ with an average around $4\times$ (unsurprisingly for 4-wide vectorization); however development on SIMD.js has since then been halted in favor of the upcoming WebAssembly’s native SIMD capabilities.

Parallelizing JS applications can also be achieved by employing WebWorkers - light-weight execution units which can spread over multiple CPUs and do not share the same memory space, but communicate via message-passing. As modern browsers enable retrieving the number of logical cores but give no information about resource sharing / utilization, the authors of (Adhianto et al., 2010) suggested using a web worker pool to balance the amount of web workers dynamically depending on current resource utilization. In addition, thread-level speculation can extract parallel behavior from single-threaded JavaScript programs (akin to the methods already described). Their results show a predictable maximum

speedup at 4 parallel Workers on a 4-core (8 virtual core) machine in the range of a $3\times\text{--}5\times$ factor, with different browsers behaving within a $\sim 30\%$ margin.

A completely different approach and real breakthrough in speeding up web-based applications was achieved with the introduction of Emscripten (?), which actually uses LLVM as its input format and outputs a statically typed subset of JS called ASM.js, enabling the JSVM to optimize code to a much higher degree than dynamic JS. In addition, it uses a flat memory layout like compiled binaries do, simulating pointer arithmetic etc., which makes compilation from C-like languages possible in the first place. Although criticized for opening up additional security holes by introducing potential risks like buffer-overflows which were already "fixed" by the JS object memory layout model, Emscripten has been extremely successful in speeding up JS to almost native speeds, while (in theory) enabling running any language in a browser by compiling its whole underlying runtime to ASM.js, as long as the runtime itself is written in a language compiling to LLVM.

Borins (2014) used Emscripten to compile the Faust libraries C++ output to JS; Faust is a code generation engine which forms a functional approach to signal processing tasks; it is generally used to deploy a signal processor to various languages and platforms. In this work specifically it was employed to target the Web Audio API in order to create audio visualizations, add effects to audio etc. The whole pipeline encompassed code generation from Faust to C++ as well as compiling to ASM.js. Preliminary experimentation showed that the pipeline was functional, although exact time measurements were apparently not possible in 2014 since Emscripten was not able to translate C++ timing code to ASM properly (this is working as of 2018).

The third general approach of speeding up browser-based code lies in transferring it to the GPU (mostly via WebGL which in its current form is a port of OpenGL ES 2.1), thus making use of modern graphic hardware's great parallelization capabilities. This is usually done for image processing or video (games), but can also work in non-obvious cases - (Ingole and Nasreh, 2015) report on experiments on dynamic graph data-structures, which is a better indicator of how GPU-parallelization will behave on general code-bases. They used *Parallel.js* (and other libraries like Rivertrail (Barton et al., 2013) which are nowadays deprecated) to compute dynamic shortest paths on growing, shrinking and fully dynamic, directed, weighted graphs with positive edge weights. Their results show that for up to a few percent of edge addition / deletion, the dynamic version usually outperforms the static version, although the margins differ between graph structures and depend on the random choice of edges added / deleted.

As a general reflection on VMs, Würthinger (2014) point out that generic bytecode-VMs are usually slower than ones focused on a specific guest language, for their parsing- and optimization routines can be specifically tailored to the syntactic idiosyncrasies of one language (and patterns / idioms usually employed by its programmers). As a remedy they propose an architecture where guest language semantics are communicated to the host runtime via specific interpreters, then the host compiler optimizes the obtained intermediate code - only this host compiler as well as the garbage collector remain the same for all languages. They implemented this idea at Oracle Corporation in the form of *Truffle* as guest runtime framework on top of the *Graal* compiler running in the *Hotspot* VM; at the time of their publication, they supported C, Java, Python, Javascript, R, Ruby and Smalltalk; performance measures had not yet been published.

Recently, the introduction of WebAssembly (WASM, (Haas et al., 2017)) as a new low-level, bytecode-like standard including static types, a structured control flow as well as compact representation opens up new opportunities, as it is simply an abstraction over modern hardware, making it language-, hardware-, and platform-independent.

Moreover, it will be able to utilize CPU-based SIMD Vectorization as well as offer a thread-based model for concurrency on shared memory, which the JS-based WebWorker model was not able to deliver (it offered concurrency via message-passing only). As an entirely independent language, it can also be extended at any time without having to extend the underlying runtime, as is the case with ASM.js & JS.

Nevertheless, it can be included in JS programs easily and even compiled from LLVM-based bytecode via Emscripten, making it a breeze to experiment with by simply using existing (C/C++) code.

3. Research goal

We posit that recent developments have made it feasible for browser-based solutions of non-trivial computational complexity to be competitive against native implementations.

In order to show this, we selected a mix of algorithms which we implemented (or took implementations from Rosetta Code) in C++ as well as JavaScript and injected additional code for run-time measurement (wall time). We then compiled the C++ version via Emscripten (Zakai, 2011) to ASM.js as well as WebAssembly (WASM) and tested the results on native Linux (GCC-compiled binary), NodeJS (native JS, ASM, WASM) as well as Chrome, Firefox and Edge (ASM, WASM). Our main interests in performing these experiments were:

- to establish a baseline performance measure by executing binary (C++) code.
- to compare the performance of binary code to JS as well as Emscripten-compiled ASM.js / WASM.
- to establish an understanding of the different optimization possibilities for JS, ASM, WASM and develop insights as to which kind of computational tasks (numeric performance, memory-management, function-call efficiency, garbage-collection etc.) they would affect.
- to test those hypotheses on the performance measures we empirically observed and describe as well as speculate on any inexplicable deviation.
- to provide an outlook on future challenges and opportunities, predicated on our lessons learned.

4. Algorithms

In order to gain relevant insights into the performance of our chosen execution environments, we gathered a mixture of algorithms from very simple toy examples to a real-world graph problem employed in modern computer vision.

4.1 Base tests

Our toy examples consist of three test cases designed to elicit algorithmic performance w.r.t. 3 specific use-cases: 1) Basic memory management: Fill an array of length 1 million with random integers, 2) Function calls: Recursive Fibonacci of $n = 40$ and 3) Numeric computations: Compare 10 million pairs of random integers.

4.2 Floyd-Warshall

The Floyd-Warshall algorithm (Floyd, 1962) is an APSP (All-pair shortest-path) graph algorithm; given a graph $G(V, E)$ with V being the set of vertices and E being the set of edges, it works by choosing a vertex k in the graph, then iterating over all possible pairs of vertices (i, j) , at each point deciding if there exists a shorter route between (i, j) were they connected via k :

Algorithm: Floyd-Warshall APSP

```

1 function FWdense(graph) {
2     i, j, k <- graph.vertices.length
3     for (k = 0; k < V; k++) {
4         for (i = 0; i < V; i++) {
5             for (j = 0; j < V; j++) {
6                 if (graph[i][j] > graph[i][k] + graph[k][j]) {
7                     graph[i][j] = graph[i][k] + graph[k][j];
8                 }
9             }
10        }
11    }
12 }
```

Figure 1: Floyd-Warshall APSP.

We chose this algorithm due to its simple implementation which relies on pure iterative performance, thus testing data structure as well as numerical efficiency.

4.3 Huffman Coding

Huffman coding/encoding is a particular form of entropy encoding that is often used for lossless data compression (Han et al., 2015). A prefix-free binary code with minimum expected codeword length is sought from a given set of symbols and their corresponding weights. The weights are usually proportional to probabilities. Let

$$L(C) = \sum_{i=1}^n w_i * \text{length}(c_i)$$

be the weighted path length of code C . The goal consists of finding a code C such that $L(C) \leq L(T)$ for all $T(A, W)$ where C is defined as:

$$C(A, W) = (c_1, c_2, c_3, \dots, c_n),$$

with the alphabet $A = \{a_1, a_2, a_3, \dots, a_n\}$ and the set of symbol weights $W = \{w_1, w_2, w_3, \dots, w_n\}$

Algorithm: Huffman(C)

```
1 n := |C|;
2 Q := C;
3 for i := 1 to n - 1 do
4     allocate a new node z
5     z.left := x := Extract-Min(Q);
6     z.right := y := Extract-Min(Q);
7     z.freq := x.freq + y.freq;
8     Insert(Q, z);
9 end for
10 return Extract-Min(Q); {return the root of the tree}
```

Figure 2: Huffman(C) taken from (Goemans, 2015)

As Huffman's implementation is heavily dependent on its binary heap, whose operations depend on memory implementation but will be much faster than heavy copying / deletion of deep data structures, we suspected that JavaScript-based runtimes would fare relatively well in their own right.

4.4 Permutation

The problem of permutation is simple to solve by dissecting a given input and combining its "head" with the inner permutations of its "rest", which can be done in a recursive fashion. For instance, given the input *abc*, the Function *getPermutations* depicted below would first extract the letter *a* and then combine it with all the permutations of the remaining string *bc*.

Algorithm: Permutation

```
1 function getPermutations(string text)
2     define results as string[]
3     if text is a single character
4         add the character to results
5         return results
6     foreach char c in text
7         define innerPermutations as string[]
8         set innerPermutations to getPermutations (text without c)
9         foreach string s in innerPermutations
10            add c + s to results
11    return results
```

Figure 3: String permutation

Since there are $n!$ possible permutations, this is also the expected runtime of the algorithm (without printing each permutation, which would result in $O(n * n!)$). As far as different language runtimes are concerned, we would suspect that those with efficient memory operations will have a significant advantage over others, since subset copying and joining will be the central operation.

4.5 Fast Fourier Transform

The Fast Fourier Transformation FFT Algorithm is a numerically very efficient algorithm to compute the Discrete Fourier Transformation (DFT) of a series of data samples (time series

data). It goes back to Gauss (1805) and was quasi re-invented by Cooley & Tukey (1965) (Cooley and Tukey, 1965). The huge advantage is that the calculation of the coefficients of the DFT can be carried out iteratively and this reduces computational time dramatically Cochran et al. (1967). A detailed description can be found in Puschel and Moura (2008)

Here a rapid explanation of the basics:

Let $f : \mathbb{R} \rightarrow \mathbb{C}$ a continuously function, where \mathbb{R} represents the set of real numbers and \mathbb{C} the complex numbers. The *Fourier Transform* of f is given by

$$H(\nu) = \int_{-\infty}^{\infty} f(t)e^{-2\pi i \nu t} dt, \quad (1)$$

where i is the imaginary unity and ν the frequency.

In most of practical situations, the function f is given in discrete form as a finite values collection $f(x_0), f(x_1), \dots, f(x_{N-1})$ with $N \in \mathbb{N}$, where \mathbb{N} is the set of natural numbers and $\{x_0, x_1, \dots, x_{N-1}\}$ is one partition on an real interval $[a, b]$ and $x_k = a + \frac{b-a}{N}k$, for $0 \leq k \leq N - 1$. In problems that imply numerical calculation, instead the equation (1) we use the sum partial

$$H(k) = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-2\pi i k j / N}, \quad (2)$$

designated *Discrete Fourier Transform* (DFT) of f over the interval $[a, b]$. If f is a function defined on the interval $[0, 2\pi]$ of real value with period 2π , the values $H(k)$, by $k = 0, \dots, N-1$, can be interpreted as the coefficients c_k

$$c_k = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-i \frac{2\pi k}{N} j}, \quad 0 \leq k \leq N-1 \quad (3)$$

of a *exponential polynomial*

$$p(x) = \sum_{k=0}^{N-1} c_k e^{ikx}, \quad (4)$$

where $x \in \mathbb{R}$, $c_k \in \mathbb{C}$, $y k = 0, 1, \dots, N-1$, which interpolate f in the values $f(x_0), f(x_1), \dots, f(x_{N-1})$.

The Discrete Fourier Transform of f over the partition $\{x_0, x_1, \dots, x_{N-1}\}$, is defined as the operator

$$DFT : \mathbb{C}^N \longrightarrow \mathbb{C}^N$$

such that

$$[c_0, c_1, \dots, c_{N-1}]^T = DFT([f(x_0), f(x_1), \dots, f(x_{N-1})]^T)$$

Technically, this is a divide & conquer algorithm based on multi-branched recursion, i.e. it breaks down a problem into sub-problems until the problem becomes so simple that it can be solved directly.

Algorithm: Fast Fourier Transform, recursive

```

1 function y = fft_rec(x)
2 n = length(x);
3 if n == 1
4   y = x;
5 else
6   m = n/2;
7   y_top = fft_rec(x(1:2:(n-1)));
8   y_bottom = fft_rec(x(2:2:n));
9   d = exp(-2 * pi * i / n) .^ (0:m-1);
10  z = d .* y_bottom;
11  y = [ y_top + z , y_top - z ];
12 end

```

Figure 4: FFT, taken from (Wörner, 2008)

Since FFT is mostly about numerical computations we suspected that JS runtimes would handle this scenario relatively well, even compared to compiled code.

4.6 Min Cut Max Flow

In order to obtain meaningful, relevant results from our experiments, we choose to add a non-trivial problem encompassing enough interesting 'moving parts' - memory management, computation intensity as well as a variety of (conditional) function call sequences - so that the resulting performance measures convey information applicable to real-world scenarios. We therefore decided on a theoretically simple graph problem (max-flow min-cut) which consists of only three computing stages, yet comprising enough internal complexity to make it interesting beyond toy samples.

Theory Assuming a graph $G = \langle V, E \rangle$, where V is a set of nodes and E is the set of edges with positive edge weights connecting them, a cut is defined as the partition of the vertices of the graph G into two disjoint sets. If the graph G contains two distinct vertices s and t , an $s - t - \text{cut}$ can be defined, which is a partition into two disjoint sets S and T where s is in S and t is in T . The cost of the $s - t - \text{cut}$ C is the sum of weights of all edges that are connecting the two disjoint sets.

$$c(S, T) = \sum_{\substack{e_{\{i,j\}} \in E, \\ |\{i,j\} \cap S| = 1}} c_e$$

The minimum cut is then defined as the cut $c(S, T)$ amongst all possible cuts on G with minimum cost.

If one considers a graph with two terminals s, t , one usually refers to these two as *source* and *sink* respectively. The max-flow problem asks how much *flow* can be transferred from the *source* to the *sink*. This can be envisioned as the edges of the graphs being pipes, with their cost representing throughput capacity.

Ford and Fulkerson (1962) stated that a maximum flow from s to t would saturate a set of edges in the graph, dividing it into two disjoint parts corresponding to a minimum cut. Thus, these two problems are equivalent, and the *cost* of a min-cut is simultaneously the maximum flow.

Graph Cuts in Computer Vision

In computer vision a lot of problems can be formulated in terms of energy minimization - a labeling f minimizing a given energy equation of the form $E(f) = \sum_{\{p,q\} \in N} V_{p,q}(f_p, f_q) + \sum_{p \in P} D_p(f_p)$ is sought. In the case of image segmentation the labels represent pixel intensities.

α -Expansion Algorithm

One algorithm for solving such energy equations is the α - Expansion algorithm by Boykov et al. (Boykov et al., 2001) An overview of the algorithm is given in figure 5. It is based on the computation of several minimum-cuts on specifically designed graphs G_α , where each graph G_α corresponds to a labeling f . The cost of the cut on G_α corresponds to the energy $E(f)$.

The graph G_α changes for each label α . Its set of vertices consists of a source (α) and a sink ($\bar{\alpha}$) vertex, all image pixels p in P and all auxiliary vertices $a_{\{p,q\}}$. Auxiliary vertices are added for each pair of neighboring pixels $\{p, q\} \in N$ with $f_p \neq f_q$.

Algorithm: Boykov α -expansion

```

1 | set an arbitrary labeling f
2 | set success := 0
3 | for each label  $\alpha \in \mathcal{L}$ 
4 |   \text{} \quad 3.1 Find  $\hat{f} = \arg \min E(\hat{f})$  among  $\hat{f}$  within one  $\alpha$ -expansion move of
|   f
5 |   \text{} \quad 3.2 If  $E(\hat{f}) < E(f)$ , set  $f = \hat{f}$  and success \text{ } := 1
6 | end for
7 | if success == true goto 2
8 | return f

```

Figure 5: α -expansion algorithm (Boykov et al., 2001)

Minimum-cut Algorithm The core of the previously presented α -expansion algorithm consists of computing minimum-cuts. Boykov et al. presented a suitable algorithm for solving the minimum-cut problem in (Boykov and Kolmogorov, 2004). It is based on finding augmenting paths and is comprised of three consecutive stages: **growth**, **augment** and **adoption**. During the computation four sets of vertices are maintained: the search trees S and T , a set of active vertices A and a set of orphan vertices O . Additionally the parent / child relations are kept.

Figure 6 shows the detail of the growth stage. $TREE(p)$ denotes the tree of vertex p , $PARENT(q)$ means the parent of p .

After an augmenting path was found during the growth stage, the path gets augmented as described in figure 7. In the adoption stage new parents are sought for the orphan nodes.

Preprocessing We conducted our experiments on electron microscopy images from human skin. These images were first converted to gray-scale and thresholded to obtain a binary segmentation. Subsequently a graph was extracted from the binary image for label 255 as described above. Computing a minimum-cut on this graph would correspond to an α -move for label 255. This graph was then saved and used for the following experiments.

Algorithm: Min-Cut growth stage

```

1 while  $A \neq \emptyset$ 
2   pick an active node  $p \in A$ 
3   for every neighbor  $q$  such that  $\text{tree\_cap}(p \rightarrow q) > 0$ 
4     if  $\text{TREE}(q) = \emptyset$  then add  $q$  to search tree as an active node
5      $\text{TREE}(q) := \text{TREE}(p), \text{PARENT}(q) := p, A := A \cup \{q\}$ 
6     if  $\text{TREE}(q) \neq \emptyset$  and  $\text{TREE}(q) \neq \text{TREE}(p)$  return  $P = \text{PATH}_{s \rightarrow t}$ 
7   end for
8   remove  $p$  from  $A$ 
9 end while
10 return  $P = \emptyset$ 

```

Figure 6: Growth stage of the Min-Cut/Max-Flow algorithm (Boykov and Kolmogorov, 2004)

Algorithm: Min-Cut augmentation stage

```

1 find the bottleneck capacity  $\Delta$  on  $P$ 
2 update the residual graph by pushing flow  $\Delta$  through  $P$ 
3 for each edge  $(p, q)$  in  $P$  that becomes saturated
4   if  $\text{TREE}(p) = \text{TREE}(q) = S$  then set  $\text{PARENT}(q) := \emptyset$  and  $O := P \cup \{q\}$ 
5   if  $\text{TREE}(p) = \text{TREE}(q) = T$  then set  $\text{PARENT}(p) := \emptyset$  and  $O := P \cup \{p\}$ 
6 end for

```

Figure 7: Augmentation stage of the Min-Cut / Max-Flow algorithm in Boykov and Kolmogorov (2004)

Algorithm: Min-Cut adoption stage

```

1 for all neighbors  $q$  of  $p$  such that  $\text{TREE}(q) = \text{TREE}(p)$ :
2   if  $\text{tree\_cap}(q \rightarrow p) > 0$  add  $q$  to the active set  $A$ 
3   if  $\text{PARENT}(q) = p$  add  $q$  to the set of orphans  $O$  and set  $\text{PARENT}(q) := \emptyset$ 
4  $\text{TREE}(p) := \emptyset, A := A - \{p\}$ 

```

Figure 8: Adoption stage of the Min-Cut / Max-Flow algorithm in Boykov and Kolmogorov (2004)

5. Experimental Setup

5.1 Testing equipment

All tests were conducted on a 2017 Lenovo Carbon X1, 5th generation, with a Core i7-7500U, 16GB of LPDDR3 RAM and a PCIe-NVMe SSD. As C-compiler we utilized *gcc (GCC) 7.2.1* with standard option flags ”-O3 -std=c++11”. For ASM.js / WASM we used *Emscripten 1.37.32* with the same option flags ”-O3 -std=c++11” (plus ”-s WASM=1” for WASM). The string permutation sample had to be compiled with Emscripten option *-s TOTAL_MEMORY=268435456 -s ALLOW_MEMORY_GROWTH=1* in order to work at all, as simply using *-s TOTAL_MEMORY=XYZ* (no matter how great the amount) showed no effect in our experiments.

The Node ASM / WASM runs were conducted directly on the respective .asm.js / .wasm.js output files, whereas the browser tests were conducted on the respective asm.html / wasm.html files under a standard Apache 2.24 local document root. The underlying operating system was *Antergos (Arch) Linux with a 4.14.15-1-ARCH kernel*. For the Chrome runs, we used *Chromium 64.0.3282.119 (Developer Build) (64-bit)* as provided by the Antergos (Arch) Linux distribution, for the Firefox benchmarks we used the Linux version *58.0.1 (64-bit)*; finally, the Edge performance tests were conducted on the latest Edge Browser on *Windows 10 Education, Build 1709*, on the same hardware as described above.

5.2 Time measurement

All timing was directly inserted into the code of our examples, to measure "wall time" from certain points of execution to others. For procedures that take too little time to conduct accurate measurements, we executed the procedure n times - e.g. both the Fast Fourier Transformation as well as the Huffman Coding were done 100k times to arrive at a measurement. Each procedure was then repeated 10 times and the average taken to be the final measure as reported in Figure 9.

6. Results and discussion

In order to understand the results better and give a starting point for their interpretation, we need to look at what happens when JS / ASM.js / WASM are executed ((Clark, 2017)) in the JSVM (binary execution does not need further explanation):

6.1 The interpretation / optimization / execution cycle:

1. **Fetching** means the downloading of the actual code representation. WASM has a slightly more compact form than JS, thus should load slightly faster.
2. **Parsing / Validating** means "reading" and "translating" the code to an executable format, in our case either JS (incl. ASM.js) or WASM bytecode. WASM offers some advantages here, since it is already presented in a format much closer to the actual hardware-determined assembly (the virtual assembly format).
3. **Monitoring** occurs during execution of JS/ASM.js (not WASM) in order to determine if an interpreted snippet of code (*cold state*) is run frequently, in which case it is compiled via the base compiler into assembly (*warn state*). For dynamically typed JS this state potentially produces a whole slew of different stumps (one for each combination of possible input value types) which then have to be chosen each time that snippet is called (depending on the "current" input value types). In case the stump is called sufficiently often, it enters the *hot* phase.
4. **Execution** is the main phase where code is actually run and produces effects. WASM is supposed to be slightly faster in this phase, since it's lower-level than (un-optimized) JS / ASM.js.
5. **Optimization** For *hot* snippets of code, the bytecode is further optimized making more or less aggressive assumptions (mostly about types again). This leads to less

querying for each call (e.g. each iteration in a loop) but generates some overhead as optimizations need to be computed during runtime.

6. **De-(Re)-optimizing / Bail-out.** In case assumptions do not hold - which can only be checked when a snippet of code is executed and some error occurs - the VM falls back to a baseline compilation or pure interpretation. This phase never occurs for WASM.
7. **Garbage-collection.** Memory management; since WASM requires manual memory management, this phase is only necessary for JS/ASM.js. This phase can have a great effect on (absolute) runtime in case memory is heavily / carelessly freed. Especially the "delete" operator in JS is notoriously slow, but recursive algorithms operating on substrings may also be...

6.2 Results

RESULTS	FillArray	RecFib	IntComp.	Floyd-W.	Perm.	FFT	Huffman	MC 50	MC 100	MC 150
Native C++ GCC	10.3	272	149	1045	92	307	547	619	11068	65227
native JS Node	18.2	1102.7	156	3204.7	3475.8	827.2	3698	n/a	n/a	n/a
ASM.js Node	14	740	173	1374	321	258	1297	1143	3249	119323
WASM Node	3.67	723	44	1364	252	218	1133	n/a	n/a	n/a
ASM.js Chrome	17	750	221	2356	407	347	1834	809	2624	120450
WASM Chrome	4	730	44	1989	352	272	1447	682	2400	112353
ASM.js Firefox	13	770	220	1567	354	467	1027	708	2406	117550
WASM Firefox	3	780	43	1538	147	179	896	610	2074	108537
ASM.js Edge	29	1670	545	1701	501	378	1828	987	3267	132141
WASM Edge	7	1630	44	1570	245	328	1642	1106	3666	148301

Figure 9: Overview table of detailed experimental results (runtime in milliseconds). Unsurprisingly, C++ compiled to binary wins in 50% of experiments; the remaining 50% go to Firefox's WASM implementation, which is comprehensible since the Mozilla foundation was first to introduce ASM.js / Emscripten and therefore has the most experience in runtime optimizations on the platform. It should be noted that WASM performance spread over different implementations was in no case dramatic (roughly within a factor of 2x), which is a testimony to the industry's good cooperation & a positive sign for the future integrity of the platform.

6.3 Discussion

We first note that our results for Recursive Fibonacci (of $n = 40$), the Floyd-Warshall APSP Algorithm (on a graph of $|vertices| = 1k$) as well as 100k iterations of Huffman Coding (on a paragraph of lorem ipsum-based strings) yielded the expected results: Compiled C++ code came in fastest, with JS showing significantly worse performance (although easily within an order of magnitude). In all three cases, ASM.js performed significantly better than native JS, but worse than WASM which allows for higher optimization. We note that those algorithms are mostly computational and function-call oriented, meaning they do not heavily rely on memory management. In the only case where heap-operations are heavily involved (Huffman Coding) - we see that JS performance worsens comparatively - a sign that memory management in the VM is still slower than natively.

The results for String Permutations followed the expected patterns, but dramatically showed the impact of standard JS garbage collection on an exponential amount of substrings created and released: the performance dropped to about 35x the time spent by the GCC compiled binary. ASM / WASM performed drastically better, the reason for which can be found in the flat-memory model of those implementations - chunks in the fixed-sized memory space are never really deallocated, but simply set inactive (undefined).

The first perplexing results lie in the run-times of filling an integer array with a million random numbers (FillArrayRand 1e6) as well as comparing ten million random integers pairs with one another (IntCompare 1e7). Here native (GCC-produced CPP) code is predictably beating ASM - although only by a slight margin when using the most recent version of NodeJS in ASM mode (factor 1.2) - but blown away by WASM by up to a factor of 3. This was so surprising that our first suggestion was Emscripten/Binaryen might be optimizing the whole procedure away; however subsequently added randomized access to the arrays (including output) alleviated this suspicion. The same behavior, albeit to a lesser extent, can be observed in Fast Fourier Transformations (FFT 100k), where WASM manages to slightly beat the native binary. We note that all of those examples are heavily computation oriented, almost purely consisting of one giant number-crunching routine.

Therefore a possible interpretation for the WASM speedup could be SIMD Vectorization happening within the Emscripten pipeline; capabilities for SIMD are being actively developed for Chrome and Firefox, and would therefore also find their way into NodeJS. On the other hand, according to current sources (Mozilla-Foundation, 2018) support for SIMD is not yet enabled in any standard build of any major JSVM. However, (Dehesa-Azuara and Chittenden) mention that their library *Vectorize.js*, which can enhance normal JavaScript by applying 4-wide vectorization on CPU-heavy loops, is actually being used by Emscripten. While this does not guarantee that current ASM.js/WASM output is using vectorization, it is a strong hint that optimizations along those lines have been an effort for years. We will need to further investigate into possible intra-JSVM / WASM optimizations that might catapult performance beyond the CPP / binary baseline.

As far as our real-world scenario is concerned, we did not run the unoptimized JS version of this algorithm, since we found the sample code to be different enough between implementations as to make a fair comparison all but impossible. As for the GCC binary and (W)ASM, the two runs on a 50x50 / 150x150 pixel input image (graph) behaved as expected, although WASM on the latter was slightly outperformed by ASM. We can only assume that

in this real-world code, optimization advantages regarding memory management (the graph was completely instantiated prior to the traversal) as well as static type assumptions did not play as great a role as expected. Furthermore, the task given most closely resembles the Floyd-Warshall (as in graph traversal) as well as RecFib40 (as in function-call centered) examples, in both of which ASM/WASM displayed similar behavior. All in all, we are satisfied to see results that 1) more or less matched our expectations and 2) behaved similarly to much smaller toy examples, since this indicates the technology is mature enough for real-world deployment.

The only case remaining inexplicable to us is that of the MinCut 100x100, in which GCC was outperformed by both ASM/WASM by a factor of more than 3x. A possible, yet unlikely, explanation could lie in a stark input abnormality in such a way that the resulting graph would allow traversal in a much different fashion from the other 2 samples: By accidentally presenting us with a graph in which the number of vertex discovery operations (involving a heap or other data-structure) are greatly reduced versus sheer numeric computations, we might get a result akin to the IntCompare one; however, the relatively good ASM.js performance speaks clearly against that; moreover, one could not attribute that behavior to implicit vectorization. In future work, we will strive to design input data to more complex algorithms in such a way as to uncover the explanatory factors hidden to us for now.

7. Open problems & Future challenges

Although the results presented are very interesting and clearly point to areas of algorithmic programming in which a decisive speedup can be achieved employing ASM.js / WASM via Emscripten, not only are our current insights incomplete with respect to *why* certain speedup factors can be achieved, nor have we exploited all possibilities to accelerate Browser-based software in general. Therefore, we will strive to improve and expand upon our research in four vital areas:

- **Better profiling.** Deeper insights into the speedup of different parts of the execution pipeline for JS, ASM.js and WASM could help us target strengths as well as weaknesses of different platforms w.r.t. certain algorithmic requirements and programming styles a lot better; this would not only clear up the picture regarding explanatory factors but might also pave the way to deriving general rules as to what technology to utilize under certain conditions.
- **Future WASM features** like native SIMD vectorization and shared-memory concurrency can not been exploited as-of-date, at least not in a stable implementation. We will be closely monitoring the progress in the field and design experiments to optimally utilize any relevant upcoming features (e.g. parallel graph algorithms would be of great interest for client-side social network computations).
- **GPGPU.** Utilizing GPUs' massive parallelization capabilities even on modest hardware like integrated or mobile graphics could easily speed up parallelizable / vectorizable code beyond the speed limits of natively compiled C/C++. However, direct compilation from already (non-GPU enabled) code-bases is highly unlikely (except

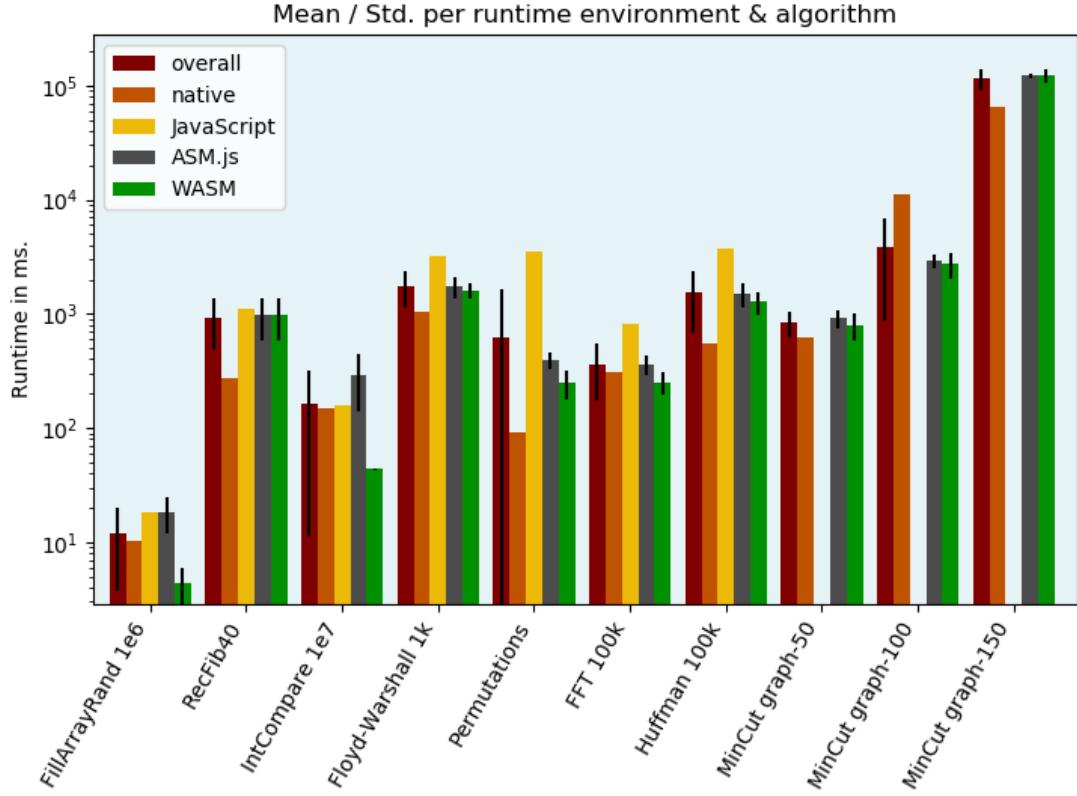


Figure 10: Overview plot of all results on one log scale including standard deviation. We note that overall performance deviation was most striking with FillArrayRand, IntCompare 1e7 as well as Permutations, but was less significant (although in no case negligible, as Figure 11 clearly shows) in the case of Floyd Warshall 1k, FastFourierTransform, as well as MinCuts 50 & 150

Emscripten's capacity for compiling OpenGL ES 2.1 code into WebGL, where the latter is a port of the former); therefore the additional effort of completely designing new code-bases for WebGL need to be taken into account. As a side-note, there exist libraries (like Tensorfire) who promise execution of Tensorflow models via WebGL - although this can not be regarded as 'speed-up' of traditional code, it would offer great potential for client-side classifiers once deep models have been obtained on more powerful hardware.

avg. runtime per algorithm & execution environment (in ms.)



Figure 11: Performance of different runtime environments on the 10 chosen algorithms. These are the same results as in Figure 10, but grouped per algorithm with each group on its own linear scale, thus relative runtime advantages can be depicted more intuitively.

- **Federated Learning.** Since client-side / Browser-based computations are quickly entering the performance-range in which they are useful for even Machine Learning Tasks, their utilization in a ML-grid as proposed by Google’s *Federated Learning* initiative (Bonawitz et al., 2016) makes great sense even and especially for startups or mobile App developers, but also for decentralized health environments (Holzinger et al., 2017), (O’Sullivan et al., 2017). We are already working on a demonstration of this concept applied to distributed graphs and will gladly include our insights in a follow-up of this paper.

Overall, we are greatly enthusiastic about the current & future possibilities of client-side algorithmic computations and are looking forward to seeing any combination of the technologies at our disposal implemented in future applications (see e.g. Malle et al. (2017)).

8. Conclusion

In this paper we presented a comparison study between the performance of native binary code, JavaScript, ASM.js as well as WASM on a selected set of algorithms we deemed representative for current challenges in client-side, distributed computing. Following a justification of this selection as well as some hypotheses on how different algorithms should perform in different execution environments, we presented experimental results and compared them to our prior expectations. We can conclude that although some scenarios played out exactly as we suspected, we were greatly surprised by others - especially WASM’s capacity of (sometimes) outperforming native code by impressive margins will need further investigation into the internals of JSVM optimization. Above and beyond that, we point to the need of subsequent studies in parallelization as well as GPU computing to further expand on the work presented.

References

- Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John MellorCrummey, and Nathan R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. doi: 10.1002/cpe.1553.
- Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010. doi: 10.1145/1721654.1721672.
- Kyle Barton, Samuel Dodson, Danielle Fenske, Ben Whitehead, and Jens Mache. Learning Parallel JavaScript with a Visual Boids Simulation. pages 6–7, 2013.
- Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. *NIPS Workshop on Private Multi-Party Machine Learning*, arXiv:1611.04482, 2016.

Myles Borins. From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten. *Linux Audio Conference*, 2014. URL <http://lac.linuxaudio.org/2014/papers/27.pdf>.

Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on pattern analysis and machine intelligence*, 23(11):1222–1239, 2001.

Lin Clark. A cartoon intro to webassembly, 2017. URL <https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/>.

William T. Cochran, James W. Cooley, David L. Favin, Howard D. Helms, Reginald A. Kaenel, William W. Lang, George C. Maling, David E. Nelson, Charles M. Rader, and Peter D. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, 1967. doi: 10.1109/PROC.1967.5957.

James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. doi: 10.2307/2003354.

Mario Dehesa-Azuara and Tom Chittenden. Automatic vectorization of JavaScript using SIMD. *js.contrib.andrew.cmu.edu*.

Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962. doi: 10.1145/367766.368168.

Lester R. Ford and Delbert R. Fulkerson. *Flows in Networks*. Princeton University Press, New Jersey, 1962.

Emily Fortuna, Owen Anderson, Luis Ceze, and Susan Eggers. A limit study of JavaScript parallelism. *IEEE International Symposium on Workload Characterization, IISWC’10*, 2010. doi: 10.1109/IISWC.2010.5649419.

Michel Goemans. Huffman Codes, MIT 18.310A Lecture Notes Principles of Discrete Mathematics. 2015. URL <http://math.mit.edu/~goemans/18310S15/18310.html>.

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. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, pages 185–200, 2017. ISSN 03621340. doi: 10.1145/3062341.3062363.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv:1510.00149*, 2015.

- Andreas Holzinger, Bernd Malle, Peter Kieseberg, Peter M. Roth, Heimo Müller, Robert Reihs, and Kurt Zatloukal. Towards the augmented pathologist: Challenges of explainable-ai in digital pathology. *arXiv:1712.06657*, 2017.
- Katharina Holzinger, Klaus Mak, Peter Kieseberg, and Andreas Holzinger. Can we trust Machine Learning Results? Artificial Intelligence in Safety-Critical Decision Support. *ERCIM News*, 112(1):42–43, 2018.
- Anurag Ingole and Rupesh Nasreh. Dynamic Shortest Paths using JavaScript on GPUs. 2015.
- Peter Jensen, Ivan Jibaja, Ningxin Hu, Dan Gohman, and John McCutchan. SIMD in JavaScript via C++ and Emscripten. *Proceedings of the 2015 Workshop on Programming Models for SIMD/Vector Processing*, 2015. URL <http://www.cs.utexas.edu/users/ivan/pubs/simdjs.pdf>.
- Michael I. Jordan and Tom M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015. doi: 10.1126/science.aaa8415.
- Bernd Malle, Peter Kieseberg, Edgar Weippl, and Andreas Holzinger. The Right to be Forgotten: Towards Machine Learning on perturbed knowledge bases. In *Springer Lecture Notes in Computer Science LNCS 9817*, pages 251–256. Springer, Heidelberg, Berlin, New York, 2016. doi: 10.1007/978-3-319-45507-5_17.
- Bernd Malle, Nicola Giuliani, Peter Kieseberg, and Andreas Holzinger. The More the Merrier - Federated Learning from Local Sphere Recommendations. In *Machine Learning and Knowledge Extraction, IFIP CD-MAKE, Lecture Notes in Computer Science LNCS 10410*, pages 367–374. Springer, Cham, 2017. doi: 10.1007/978-3-319-66808-6_24.
- Mojtaba Mehrara, Po Chun Hsu, Mehrzad Samadi, and Scott Mahlke. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. *Proceedings - International Symposium on High-Performance Computer Architecture*, pages 87–98, 2011. ISSN 15300897. doi: 10.1109/HPCA.2011.5749719.
- Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. *arXiv:1712.05889*, 2017.
- Mozilla-Foundation. SIMD technical reference, 2018. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SIMD.
- Shane O’Sullivan, Andreas Holzinger, Kurt Zatloukal, Paulo Saldiva, Muhammad I. Sajid, and Wichmann Dominic. Machine learning enhanced virtual autopsy. *Autopsy Case Report*, 7(4):3–7, 2017. doi: 10.4322/acr.2017.037.
- Markus Puschel and Jose M.F. Moura. Algebraic signal processing theory: Cooleytukey type algorithms for dcts and dsts. *IEEE Transactions on Signal Processing*, 56(4):1502–1521, 2008. doi: 10.1109/TSP.2007.907919.

Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W. Mahoney, Randy Katz, Anthony D. Joseph, Michael Jordan, Joseph M. Hellerstein, and Joseph E. Gonzalez. A berkeley view of systems challenges for ai. *arXiv:1712.05855*, 2017.

Seon Wook Kim and Youngsun Han. JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–25, 2016. ISSN 15443973. doi: 10.1145/2846098. URL <http://dx.doi.org/10.1145/2846098>.

Stefan Wörner. Fast Fourier Transform. *Numerical Analysis Seminar ETH Zürich*, 2008.

Thomas Würthinger. Graal and Truffle : Modularity and Separation of Concerns as Cornerstones for Building a Multipurpose Runtime. page 2584663, 2014. doi: 10.1145/2584469. 2584663.

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 (OOPSLA 2011)*, pages 301–312. ACM, 2011. doi: 10.1145/2048147.2048224.