FAUST Domain Specific Audio DSP Language Compiled to WebAssembly

Stéphane Letz
GRAME, France
letz@grame.fr

Yann Orlarey
GRAME, France
orlarey@grame.fr

Dominique Fober
GRAME, France
fober@grame.fr

ABSTRACT

This paper demonstrates how FAUST, a functional programming language for sound synthesis and audio processing, can be used to develop efficient audio code for the Web. After a brief overview of the language, its compiler and the architecture system allowing to deploy the same program as a variety of targets, the generation of WebAssembly code and the deployment of specialized WebAudio nodes will be explained. Several use cases will be presented. Extensive benchmarks to compare the performance of native and WebAssembly versions of the same set of DSP have be done and will be commented.

CCS CONCEPTS

• **Applied computing → Sound and music computing**; • **Software and its engineering → Functional languages**; **Data flow languages**; **Compilers**; **Domain Specific Languages**;

KEYWORDS

Signal processing; Domain Specific Language; audio; Faust; DSP; compilation; WebAssembly; WebAudio

ACM Reference Format:
Stéphane Letz, Yann Orlarey, and Dominique Fober. 2018. FAUST Domain Specific Audio DSP Language Compiled to WebAssembly. In *WWW '18 Companion: The 2018 Web Conference Companion, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3184558.3185970

1 INTRODUCTION

FAUST [7], [8], [9], is a Domain Specific Language for sound synthesis and audio processing. It is used by audio DSP developers as an alternative to C/C++ to design and implement efficient DSP code as native applications or plugins. Built as a multi-languages code generator, the FAUST compiler can be easily extended to target new audio platforms (section 2 and 3).

With the advent of both HTML5 and the Web Audio API (a high-level JavaScript API for audio processing and synthesis), complex audio applications can now be developed for the Web. The Web Audio API even offers developers the possibility to add specialized and efficient audio nodes, to be used with the Web Audio API natively defined ones (sections 4.1 to 4.3).

WWW '18 Companion, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3185970

1.1 Extending the Web Audio API

Various JavaScript DSP libraries or musical languages have been developed over the years [2], [10], to extend, abstract and empower the capabilities of the official API. They offer users a richer set of audio DSP algorithms and sound models to be directly used in JavaScript code. In this case, developments have to be restarted from scratch, or by adapting already written code (often in more real-time friendly languages like C/C++) into JavaScript.

An interesting alternative has been developed by the Csound team [5]. Using the C/C++ to JavaScript Emscripten compiler, the complete C written Csound runtime and DSP language is now available in the context of the Web Audio API.

1.2 Extending the Web Audio API with FAUST

The paper shows how the FAUST compilation chain has been moved to the Web. After a general introduction of the language and its compiler organization, developments done to target the WebAudio API and the recently defined WebAssembly low-level format will be detailed (section 4.3 to 4.5). As a Domain Specific Language targeting highly demanding mathematical code, generating WebAssembly code from the FAUST language shows specific code generation issues that compiler writers in related domains could be interested in.

Two main applications will be exposed: creating ready to use Web Audio nodes and HTML pages using the FAUST compiler in an *offline* manner, or even more interesting and flexible, *embedding* and *using* the compiler itself in Web pages (section 5).

Several benchmarks to compare the performances of native and WebAssembly versions of the same set of DSP, and the performance of the different WebAssembly aware browsers will be presented (section 6).

2 FAUST LANGUAGE OVERVIEW

FAUST is a functional programming language for sound synthesis and audio processing with a strong focus on the design of synthesizers, musical instruments, audio effects, etc. Its main sources of inspiration are *lambda-calculus*, *combinatory logic*, John Backus' *FP* [1], and Stefanescu's *Algebra of Flownomials* [11].

2.1 Introduction

FAUST is used on stage for concerts and artistic productions, in education and research, in open-source projects as well as in commercial applications. Typical users are musicians, sound engineers, researchers, musical assistants, etc. They often have a background in signal processing or at least a clear idea of how audio effects and sound synthesis systems should work or sound. But users are not necessarily computer scientists or professional developers. The

development of real-time audio software in C is usually out of reach for most of them. The ambition of FAUST is to offer them a viable and efficient high-level alternative. The FAUST compiler can generate optimized code for a variety of languages: C++, C, Java, JavaScript, asm.js, LLVM, and WebAssembly.

2.2 Syntax and semantic

As we will see through various examples, programming in FAUST is essentially combining *signal processors* using a set of binary operations that form the Block-Diagram Algebra. The functional programming approach is particularly suited for this purpose. *Signals* are discrete-time functions. *Signal processors* are second-order functions operating on *signals*. The *block-diagram algebra* used to combine signal processors together is a set of third-order composition operations on *signal processors*. Finally, user-defined functions are higher-order functions on block-diagram expressions. Powerful means, like pattern matching, are available to algorithmically generate complex audio circuits.

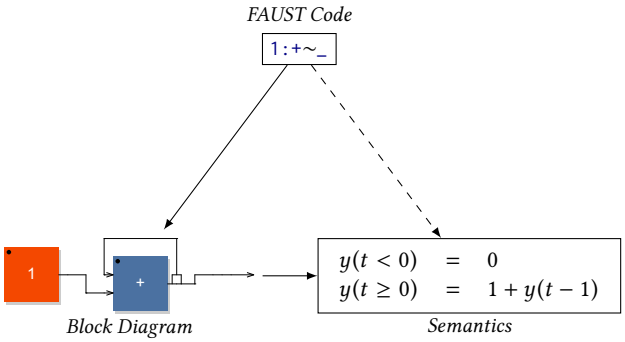

Figure 1: FAUST programs have a straightforward visual representation as block-diagrams, as well as a simple and well defined formal semantics. The block-diagram is a useful intermediate step to compute the semantics of a program.

FAUST is a textual language, but programs have straightforward translations into visual block-diagrams as well as mathematical descriptions. The relation between the FAUST code, and its translations is represented (Figure 1). The FAUST compiler is able to automatically produce these diagrams and the mathematical semantics of a program. This feature is used in particular for preservation purposes, an important concern for music pieces relying on real time programs. But these features are also very useful when learning FAUST to better understand the meaning of expressions and programs.

2.3 DSP source code example

Here is a simple organ instrument coded in FAUST (Figure 2). An oscillator signal is first defined using a *phasor* periodic signal connected to the *sin* function. The organ timbre is defined as the summation of three oscillator signals at different frequencies and volumes. An *envelop* is then defined to be applied on the continuous timbre to build a *voice*. An finally control items to be displayed as buttons or sliders to start the note, change its frequency and volume are defined.

```
// Simple Organ
import("stdfaust.lib");

// MIDI keyon-keyoff
midigate = button ("gate");
// MIDI keyon key
midifreq = hslider("freq[unit:Hz]", 440, 20, 5000, 1);
// MIDI keyon velocity
midigain = hslider("gain", 0.5, 0, 10, 0.01);

process = voice(midigate, midigain, midifreq)
    * hslider("volume", 0, 0, 1, 0.01);

// Implementation
phasor(f) = f/ma.SR : (+,1.0:fmod) ~ _ ;
osc(f) = phasor(f) * 6.28318530718 : sin;

timbre(freq)= osc(freq) + 0.5*osc(2.0*freq) + 0.25*osc(3.0*freq);

envelop(gate, gain) = gate * gain : smooth(0.9995)
                with { smooth(c) = * (1-c) : + ~ * (c) ; } ;

voice(gate, gain, freq) = envelop(gate, gain) * timbre(freq);
```

Figure 2: Example of a simple organ instrument coded in FAUST

3 COMPILER INTERNALS

The FAUST compiler is organized in successive stages, from the DSP block diagram to signals, and finally to the FIR (FAUST Imperative Representation) which is then translated in several target languages. The FIR language describes the computation performed on the samples in a generic manner. It contains primitives to read and write variables and arrays, do arithmetic operations, and defines the necessary control structures (*for* and *while* loops, *if* structure etc.).

Furthermore, the FAUST compiler itself has been packaged as an embeddable library called *libfaust*, published with an associated API [6] based on a factory/instance model. This model allows to develop standalone IDE like the FaustLive [3] application for instance.

3.1 FIR backends

Several backends have been developed (Figure 3) to translate the FIR in C, C++, Java, asm.js, WebAssembly and LLVM IR[1].

A generated DSP object has a compiled time known memory footprint and a bounded CPU usage. Its memory is statically allocated by the architecture at initialization time. Memory zones [2] corresponding to the controller value (sliders, buttons, bargraph...) are shared between the external control code and the DSP itself.

Depending of the target language, each backend has to adapt its code generation strategy so that the resulting code can be deployed and run in an execution context. For instance memory can be accessed using variable names or just indexes. Some needed mathematical functions will be available in the target langage, other will have to be externally found and *linked* with the generated code.

[1] Low Level Virtual Machine Intermediate Representation.
[2] As float* or double* memory addresses

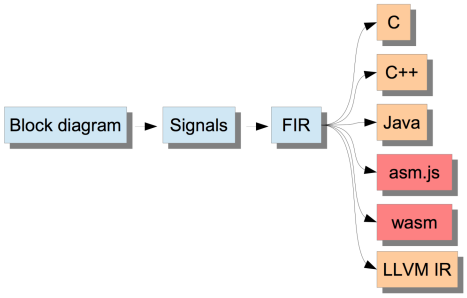

Figure 3: Compilation chain: from DSP source code to output language

```
virtual void compute(int count,
                     FAUSTFLOAT** inputs,
                     FAUSTFLOAT** outputs) {
    FAUSTFLOAT* output0 = outputs[0];
    float fSlow0 = float(fHslider0);
    float fSlow1 = (0.000500000024f
                    * (float(fButton0)
                    * float(fHslider1)));
    float fSlow2 = float(fHslider2);
    float fSlow3 = (fConst1 * fSlow2);
    float fSlow4 = (fConst2 * fSlow2);
    float fSlow5 = (fConst3 * fSlow2);
    for (int i = 0; (i < count); i = (i + 1)) {
        fRec0[0] = (fSlow1 + (0.999499977f * fRec0[1]));
        fRec1[0] = fmodf((fSlow3 + fRec1[1]), 1.0f);
        fRec2[0] = fmodf((fSlow4 + fRec2[1]), 1.0f);
        fRec3[0] = fmodf((fSlow5 + fRec3[1]), 1.0f);
        output0[i] = FAUSTFLOAT((fSlow0 * (fRec0[0] *
                ((sinf((6.28318548f * fRec1[0]))
                + (0.5f * sinf((6.28318548f * fRec2[0])))))
                + (0.25f * sinf((6.28318548f * fRec3[0])))))));
        fRec0[1] = fRec0[0];
        fRec1[1] = fRec1[0];
        fRec2[1] = fRec2[0];
        fRec3[1] = fRec3[0];
    }
}
```

Figure 4: The C++ compute function compiled from the organ instrument

Their prototype may have to be generated in a so called *import section*. Some primitives (like *min/max*) functions are possibly available in the execution context, or will have to be explicitly generated.

The C++ backend will typically generate a class, with named fields (as integer or float scalar or arrays) to describe the DSP object internal state (made up from delay lines and controller values), as well as a set of methods to get/set the DSP state, initialize it, and process samples at each cycle.

An extract of the C++ class generated from the previously described organ instrument shows the *compute* method (Figure 4), which will be called with input/output audio buffers at each cycle to produce samples.

3.2 Deploying the code

The generated code is then combined with an *architecture file* describing how to relate the audio computation with the external world.

Native audio drivers are developed as subclasses of a base audio class, controllers as subclasses of a base UI class. Typical Graphical

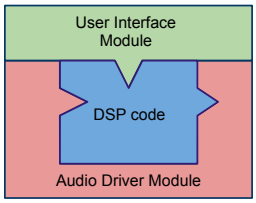

Figure 5: DSP code is generated by the compiler, audio and UI codes are added from the generic architecture files.

User Interface architectures are based on well established frameworks like QT or JUCE, and allow to display a ready to use window with sliders, text zones and buttons. Audio and UI parts are finally combined with the actual DSP computation to produce the final audio application or plugin (Figure 5).

Thanks to this approach, the exact same source code can be used to generate native applications and plugins for more than 20 different targets, from VST and Unity plugins to Android applications, from embedded systems to Web Audio applications.

4 WEB AUDIO API

The Web Audio API specification describes a high-level JavaScript API for processing and synthesizing audio in Web applications. The design model is based on an audio graph, where a set of AudioNode objects are created and connected together to describe the desired audio computation.

The actual processing is usually executed in the underlying implementation (typically optimized Assembly/C++ code), and direct JavaScript processing and synthesis is also supported using the *ScriptProcessorNode* interface, in a non-real-time rendering context, thus possibly causing annoying audio glitches.

The AudioWorklet specification[3] aims at improving the situation, having the audio graph definition done in the main thread, but rendering it (including user-defined nodes coded in pure JavaScript or WebAssembly) in a separated real-time thread. This new specification can now be tested in Chrome Canary development version.

4.1 From asm.js to WebAssembly

Mozilla developers have started in 2011 the Emscripten compiler project [12], based on LLVM technology. From C/C++ sources, it allows to generate a statically compilable and garbage-collection free typed subset of JavaScript named *asm.js*. This approach has been successful, demonstrating that near native code performance could be achieved on the Web.

Starting from this asm.js experience, core developers of the PNaCL[4] and asm.js projects have designed WebAssembly[5], a new efficient low-level programming language for in-browser client-side scripting. As a portable stack machine model, it aims to be faster than JavaScript to parse and execute.

[3] https://webaudio.github.io/web-audio-api/#rendering-loop
[4] Google Portable Native Client (PNaCl) is a sandboxing technology for running a subset of Intel x86, ARM, or MIPS native code in a sandbox.
[5] As asm.js model done correctly, see http://webassembly.org

WebAssembly initial goal is to support compilation from C/C++ using specialized compilers like Emscripten, or as a compilation target for other high level or Domain Specific Languages. The minimum viable product (MVP) specification has been finalized early 2017, with a binary format, as well as a textual format that looks like traditional assembly languages.

WebAssembly is now officially supported in all major browsers. Porting well established C/C++ codebase with Emscripten [6], like the Csound [7] framework as an example, or using DSL languages like FAUST, will then naturally benefit from improved and more stable performances.

4.2 Compiling to WebAssembly

The WebAssembly specification precisely defines the semantic of the language as well as the module format. With the previous experience of the asm.js format, the designers decided to create a binary encoding with a dense representation of module information, that enables small files, fast decoding, and reduced memory usage, aimed to be *streamed and decoded on the fly*.

Thus the binary format requires the module to be structured so that relevant information is always available while decoding the file. For developers, the textual format is simpler to deal with, can be read, understood, and possibly manually written, while working on the tools.

Two new *wast* and *wasm* FAUST backends have been developed to generate these formats. The *wast* one has been done first and generates the textual human-readable code, easier to test and debug. The *wasm* one generates the binary format to be directly loaded and executed in browsers or wasm aware standalone runtimes. The equivalent of the C++ class (generated by the C++ backend), is generated using the WebAssembly module model.

4.2.1 Module definition. A module is a distributable, loadable, and executable unit of code in WebAssembly, instantiated at runtime with a set of imported values (like JavaScript functions or memory segments) to produce instances. The two WebAssembly backends must translate the intermediate FIR code to comply to the required module format.

Some WebAssembly focused projects like Binaryen [8] define and implement their own abstract syntax tree model, and a client API to build, manipulate it, and save if as textual or binary formats. To avoid having to work on a new memory structure, and adding an external library dependency to the FAUST compiler, it was chosen not to use them, and directly work on the FIR format.

4.2.2 Memory management. Modules work on *linear memory* blocks, contiguous, byte-addressable range of memory spanning from offset 0 and extending up to a varying memory size. Memory segments are either defined internally in the module, or imported from the JavaScript context. Since the DSP memory size is known at compile time, it is easy to define the proper memory layout to represent the DSP state.

[6] http://kripken.github.io/emscripten-site/
[7] https://www.mansoft.nl/csound/
[8] https://github.com/WebAssembly/binaryen

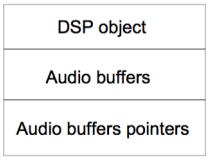

Figure 6: Wasm memory block layout for a monophonic DSP

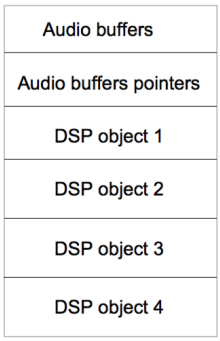

Figure 7: Wasm memory block layout for a polyphonic DSP (here with 4 voices)

The memory block of the generated DSP contains the main DSP object, *inlined* sub-objects [9], as well as audio buffers and their corresponding pointers [10]. In C++, fields in the C++ object would be accessed using their name. In WebAssembly, memory has to be accessed using an index in the memory block. All fields indexes are thus computed in a preliminary compilation pass, as offsets from the DSP object base address, to be used later in the code for load/store operations.

When a single monophonic DSP object is generated, the module internal memory is used. In our own defined memory layout, we can decide to locate the DSP object at address 0, thus simplifying and speeding up access of the DSP fields, since they can simply be defined as *offsets in the DSP object*, without having to generate the code to add the DSP base address. Audio buffers and their pointers are placed after the DSP (Figure 6).

If the DSP object is going to be used in a more complex memory layout (like when allocating several DSP objects in a polyphonic instrument for instance), a JavaScript created WebAssembly memory block is imported. Audio buffers and their pointers are placed starting at address 0 followed by DSP objects. Since the base address of each DSP will change, more complex field access code which adds the DSP base address and the field offset will be generated (Figure 7).

4.2.3 Functions generation. All FIR functions are compiled and exported in the module *export* section. Prototypes of required mathematical functions which are not part of the WebAssembly

[9] The FAUST backend for a more structured langage like C++ typically generates sub-classes in this case.
[10] At each cycle, audio buffers will be copied with data coming from the audio wrapper

specification are generated in the module *import* section, to be retrieved from the enclosing JavaScript environment at execution time. Code for 32 or 64 bits float format can be generated, with the adapted version of mathematical functions and memory access code.

Some specific functions like *integer version of min/max functions* (which are not yet part of WebAssembly specification) are internally generated and inlined in the resulting code. Specialized FIR passes [11] must be written: to move all local variables definition at the beginning of the functions, or to count local variables (stack-/loop) with their types, etc., as required by the binary encoding specification.

The code generation requires a first pass to compute all scalar and array fields offset in memory, then possibly additional FIR passes to transform the FIR, or compute needed informations. The *wast* backend is easier to write since part of this information (like for instance the number of local variables with their types) does not need to be explicitly generated. On the contrary, the *wasm* backend is much more demanding, and several preliminary passes have to be done before the final code can be generated.

4.2.4 Additional code. In both cases, a full description of the DSP object state as a JSON string is generated in the module data segment [12] (including memory indexes of all controllers). Glue code will get and decode this JSON description, and use whatever parts of the description it needs to run the DSP code. In particular, control memory zones (corresponding to the UI items like buttons, sliders, bargraph...) can be directly read/written by the wrapper code.

4.2.5 Float denormals handling. A specific problem occurs when audio computation produces *denormal* [13] float values, which is quite common with recursive filters. Denormals are very small numbers, close to zero, that doesn't follow the format of normal floating point numbers. The problem is that denormal computations take much longer to calculate than normal computations on some processors, like the Intel family for instance.

Since audio DSP algorithms can usually afford to approximate computations with very small numbers and replace denormals with 0, a Flush To Zero (FTZ) mode for denormals can usually be set at hardware level, in order to completely avoid those costly computations.

The hardware FTZ mode is not yet available in WebAssembly MVP version, which strictly conforms to the IEEE 754 norm [14]. An automatic software strategy which consists in adding FTZ code in all recursive loops has been implemented in the FAUST compiler. To activate it, the *-ftz* compilation parameter must be used at compilation time:

- the *-ftz 1* mode adds a test in each recursive loop which uses the *fabs* function and a threshold to detect denormal samples (slower).
- the *-ftz 2* mode adds a test in each recursive loop which uses a mask to detect denormal samples (faster).

[11] The FIR tree can be traversed using a visitor like pattern, possibly doing *FIR to FIR* kind of transformations.
[12] http://webassembly.org/docs/modules/
[13] https://en.wikipedia.org/wiki/Denormal_number
[14] https://github.com/WebAssembly/design/issues/148

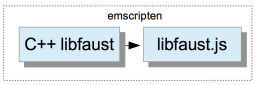

Figure 8: Compiling C++ libfaust to libfaust.js with Emscripten

Even if this strategy is not perfect, this additionally generated code [15] will avoid the production of most of denormals values with their associated CPU consumption peaks.

4.3 WebAssembly code in Web Audio nodes

JavaScript code is used to load the wasm file into a typed array, compile it to a module with *WebAssembly.compile*, then instantiate it using *WebAssembly.Instance* function, and finally get the callable exported functions. The DSP memory is either allocated inside the wasm module, or externally in the wrapping JavaScript code, and given as parameter when creating the module.

An extended *AudioNode* object [16] with some additional methods is created. As an AudioNode type it will be usable like a regular AudioNode, possibly connected to other nodes, etc.

Starting from a `karplus.dsp` FAUST source file for example, the following function has to be used, taking as parameters: the wasm filename, the Web Audio context, the buffer size, and a callback to use the extended WebAudio node:

```
faust.createkarplus(file, context, bs, cb);
```

Assuming a *karplus* variable finally contains the created object, the user interface can be retrieved as a JSON description:

```
var jd = karplus.getJSON();
```

The WebAudio node can be controlled with the following kind of code:

```
karplus.connect(context.destination);
karplus.setParamValue("/path/to/control", 0.5);
```

4.4 Embedding the JavaScript FAUST compiler in the browser

Since the Emscripten compiler helps deploying any C++ code on the Web, it becomes possible to *compile the FAUST compiler itself with its embedded wasm backend* in pure JavaScript and WebAssembly (Figure 8).

It has been done by compiling the C++ *libfaust* library in a *libfaust.js* library combined with a *libfaust.wasm* file. A unique low-level `createWasmCDSPFactoryFromString` entry point has been defined, compiling the DSP source code using the wasm backend, and producing the module as an array of bytes and the helper JavaScript function as a string (Figure 9).

Using WebAssembly API again and JavaScript *eval* function, allows to deploy it in the JavaScript context. Some additional glue code has been written, so that from the JavaScript side, a DSP factory will be created from the DSP source code with the following code:

```
faust.createDSPFactory(dsp_code, argv, cb);
```

[15] which is actually a bit more costly to compute
[16] https://webaudio.github.io/web-audio-api/#the-audionode-interface

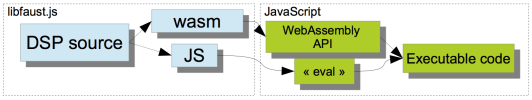

Figure 9: libfaust.js + wasm dynamic compilation chain

Here *argv* is a array of possible additional compilation arguments, and *cb* a callback to use the created DSP factory.

A fully working DSP instance as an extended Web Audio node is then created with the following code;

```
faust.createDSPInstance(fact, context, bs, cb);
```

and can be controlled with the API described in section 4.3.

5 USE CASES

Using the previously explained technologies, three different use cases have been experimented:

- compiling self-contained ready to use Web Audio nodes
- using FAUST static compilation chain to produce HTML pages with Web Audio nodes
- using the FAUST dynamic compilation chain to directly *program DSP* on the Web.

5.1 Programming Web Audio nodes with FAUST

Self contained ready to use Web Audio nodes can be produced from a DSP source using the *faust2wasm* script, which basically calls the FAUST compiler targeting the wasm backend, then wraps the produced code with a generic JavaScript API to be usable in the Web Audio context.

Audio nodes are created and activated. The full JSON description of the control parameters and their layout is available and can be used to create customized Graphical User Interfaces. Control parameters can then be read and written. This model has to be used when a custom control or Graphical User Interface is developed later on.

5.2 Deploying FAUST DSP examples on the Web

Using the *faust2webaudiowasm* script, a DSP source file can be compiled to a self-contained ready to run HTML page, and wrapping the wasm/JavaScript generated code in a HTML CSS/SVG based Graphical User Interface (Figure 10).

5.3 Web embedded compiler

Having the FAUST compiler itself as a library in the browser opens interesting capabilities, experimented in two different tools.

5.3.1 The FAUST Editor . The *FAUST Editor* application[17] can be used to edit, compile and run FAUST code from any recent Web Browser with WebAssembly support (Figure 11). This editor

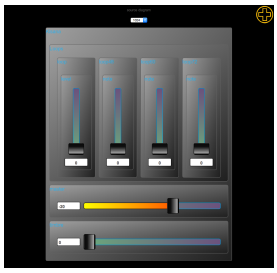

Figure 10: Self-contained HTML page loading the wasm module

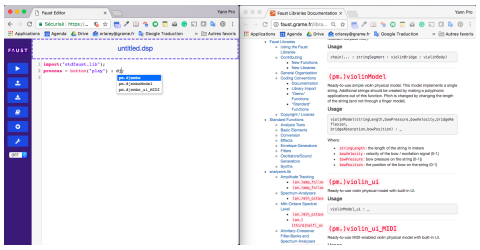

Figure 11: FAUST Editor online tool

completely works on the client side and it is therefore very convenient for situations with many simultaneous users (workshops, classrooms, etc.).

It embeds the latest version of the FAUST compiler with the wasm backend and offers polyphonic MIDI support. The editor engine is based on codemirror. It offers syntax highlighting, auto completion and direct access to the online documentation.

5.3.2 The FaustPlayground . The *FaustPlayground* application[18] lets the user develop an audio application by graphically connecting high-level modules written in FAUST. The source code can be dropped as a string, a file, or a Web URL, or loaded from a library of predefined modules included in the platform (Figure 12).

Using libfaust.js, the DSP is compiled in the browser on the client machine, to become a functional Web Audio node that can be connected to others. At any time, the node source code can be edited and recompiled.

The user can then export his work to all the platforms supported by the online compilation service, from pure standalone applications, various plugins formats (VST, Max/MSP...), projects in source code form (JUCE, iOS...) [19]. In order to perform this export, the graph must first be transformed into a single FAUST source code obtained by collecting the FAUST implementations of each node of the graph [4].

[17] http://faust.grame.fr/editor

[18] http://faust.grame.fr/faustplayground

[19] http://faustservice.grame.fr

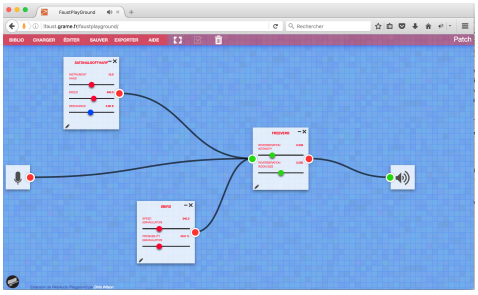

Figure 12: FaustPlayground dynamic compilation platform

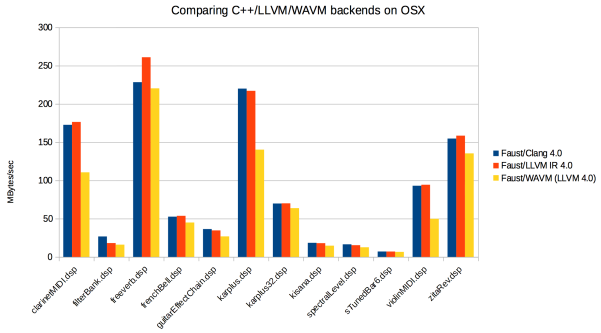

Figure 13: MBytes/sec for a test set of DSPs (higher is better)

6 BENCHMARKS

6.1 Comparing the FAUST C++, LLVM IR and wast/wasm backends

The WebAssembly approach promises *near native* performances for C/C++ written code compiled to WebAssembly using the Emscripten tool chain. Other languages like Rust (using the *mir2wasm* [20] tool) experiment with direct WebAssembly generation. It seems clear that as the WebAssembly specification and its implementation stabilizes, more and more languages will directly generate wasm to be deployed in browsers. The question of the *code generation quality at each step of the compilation chain* will rapidly emerge.

Since our compiler is generating code for the quite focused audio domain, which is caracterized by a lot of memory access and mathematical operations, we can possibly expect to generate high quality code, even beating in some cases the generic Emscripten compilation chain based generated code.

While WebAssembly is initially designed to run on the Web, it may be deployed in non Web environnement like nodejs [21], or even in standalone Virtual Machines like WAVM, developed in C++, which JIT compile WebAssembly to native code using the LLVM technology [22]. Thus WebAssembly becomes a portable binary format that can be used in a large variety of situations. This is especially of interest for a DSL language like FAUST.

6.2 Benchmark of C++, LLVM IR and wast/wasm generated code

Since FAUST already generates C++ or LLVM IR code, the performances of those two backends can be compared with the new wasm one. Using the WAVM C++ written machine allows to deploy the same measuring code [23]. The first benchmark compares the speed of C++, LLVM IR and wasm backends, running a set of DSP on a MacBook Pro 2,2 GHz Core I7 with OSX El capitan. The same 4.0 version of LLVM toolchain has been used with the three backends.

C++ and LLVM IR code has been compiled with the *-Ofast* optimization flag, the WAVM runtime is the standard version one (without any specific audio optimization, see later) (Figure 13).

The diagram clearly shows that the wasm code is still slower than C++ or LLVM IR code, but speed difference is not so high in most cases. The poor performances of some DSP need to be analyzed in more detail.

6.3 Optimizing the WAVM runtime for audio code

The WAVM runtime strictly conforms to the WebAssembly specification, thus behaving as an interesting base reference. In the audio domain, the C/C++ generated code is usually compiled with specific optimization flags [24]. Since C++ WAVM runtime can be quite easily modified, we did several changes into the reference implementation to improve the generated code performances (Figure 14).

- removing the *atomic* flag in all load/store [25] that are added to pass all spec WebAssembly tests
- adding the equivalent of *-fast-math* compilation flag that has to be done at LLVM IR and JIT (= native) [26] generation steps
- and finally simplifying some mathematical operators, using their standard definition [27] instead of the specific WAVM coded ones that strictly implement the WebAssembly official semantic [28] (see for instance the definition of *f32.min/f32.max* operations with their handling of NaN values)

The diagram shows the difference between a runtime strictly conforming to the WebAssembly specification, and the same runtime optimized for audio code. This somewhat characterizes the *incompressible speed difference* that will always exist between the C++ or LLVM IR version of FAUST generated code, and the wasm generated one.

Note that after generating the LLVM IR code, the WAVM runtime runs a set of LLVM IR to IR optimizations passes. It remains to be tested if adding more optimization passes (especially the *auto-vectorization* ones) could help producing even better code.

[20] https://github.com/brson/mir2wasm
[21] http://webassembly.org/docs/non-web/
[22] https://github.com/AndrewScheidecker/WAVM
[23] http://faust.grame.fr/news/2017/04/26/optimizing-compilation-parameters.html

[24] like *-fast-math* which lets the compiler generate faster (but less precise) mathematical operations
[25] https://github.com/sletz/WAVM/commit/cf6011026aa75dfd0f88e051da271ce0c0d525a9
[26] https://github.com/sletz/WAVM/commit/1aa96a2088ed1c6eb918b7f292f4571aecdfc6da
[27] https://github.com/sletz/WAVM/commit/a9e2a91c53e79168fb7e193beb36e99d81d0be21
[28] http://webassembly.org/docs/semantics/

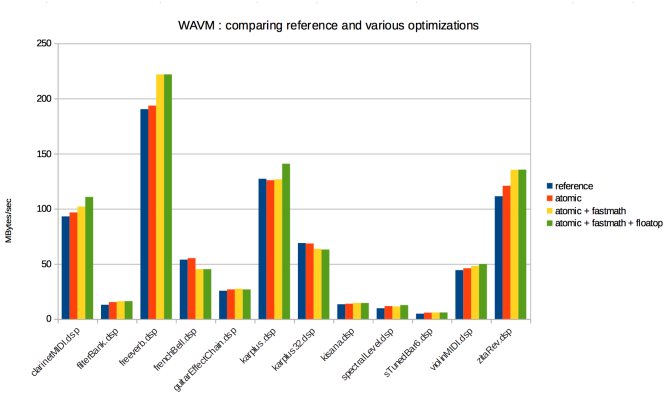

Figure 14

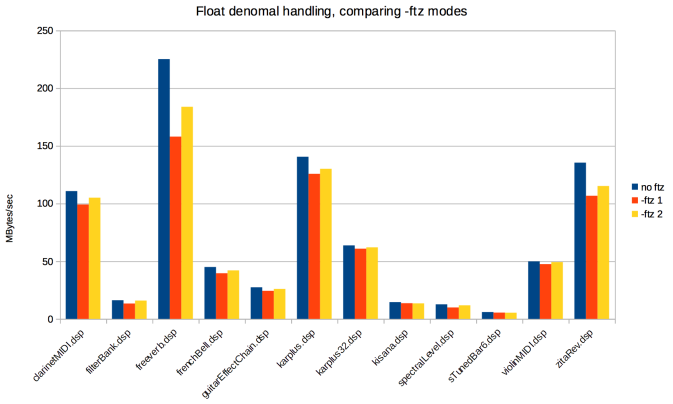

Figure 16

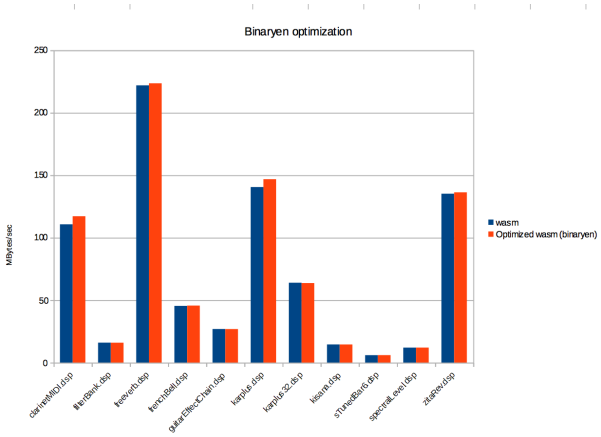

Figure 15

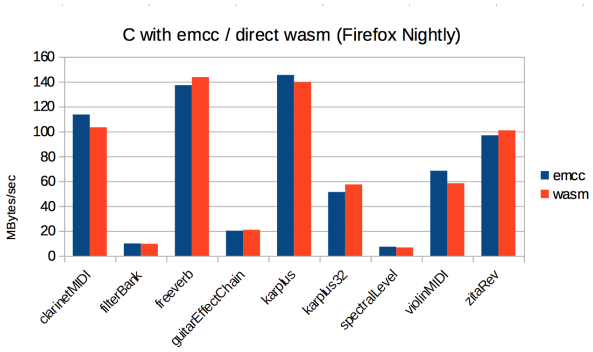

Figure 17

6.4 Module optimization with Binaryen

Binaryen is a compiler and toolchain infrastructure library for WebAssembly [29], written in C++. Its contains *wasm to wasm* optimisations passes, possibly allowing to even improve the speed of the FAUST generated wasm code. We tested the *wasm-opt* tool at *-O3* level on FAUST generated wasm modules to estimate which speedup we can expect. The diagram shows limited gains, with no more than 5% in some of the tested cases (Figure 15).

6.5 Float denormal handling

The software float denormal code can be easily tested in the WAVM machine [30]. Here is the result of code generated with -ftz from 0 to 2 (Figure 16). The code is continuously a bit slower when software FTZ is activated, the point being obviously to avoid pathological cases where the CPU consumption would raise dramatically in the presence of denormals values. The *-ftz 2* mode appears to be the most efficient one and will preferably be used.

[29] https://github.com/WebAssembly/binaryen

[30] by explicitly deactivating hardware FTZ mode on the CPU running in this native runtime.

6.6 Emscripten versus FAUST direct compilation

Wasm code directly generated by the FAUST wasm backend can be compared to wasm code generated by compiling (with Emscripten) the code generated by the FAUST C backend. All sophisticated optimizations passes the LLVM based Emscripten compiler can possibly be used on the C side, a simpler *FIR to wasm* generation model, but coupled with some specific optimizations (for instance the optimized memory layout one described in section 4.2.2) on the direct wasm side. Compilation of FAUST generated C has been done using *-s SIDE_MODULE=1* mode to produce a light wasm module, without any runtime, to be loaded and activated by additional JavaScript glue code.

Done under Firefox Nightly, the diagram shows that direct wasm code generation can even beat Emscripten generated wasm code (Figure 17).

6.7 Comparing three browsers

HTML test pages [31] were prepared to compare the performances of the three main browsers. The DSP code is compiled with float denormal protection on. The generated wasm module's *compute*

[31] http://faust.grame.fr/bench/

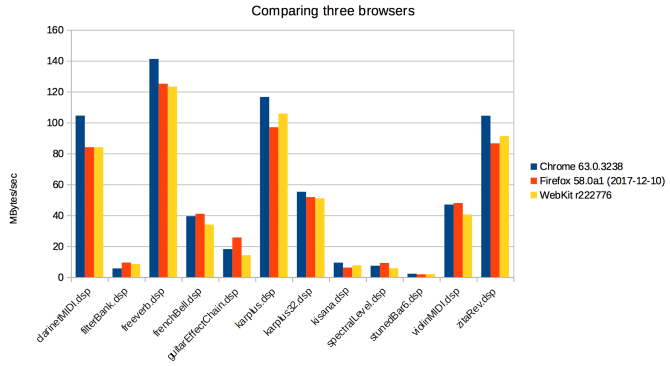

Figure 18

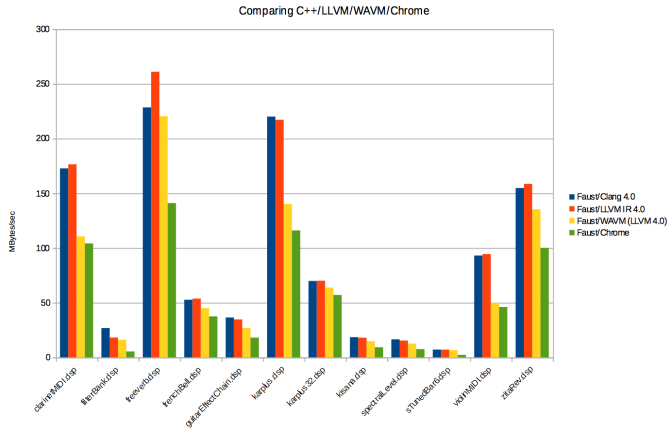

Figure 19

method is called repeatedly in a timed loop, using successive slices of a big allocated circular audio buffer to avoid cache effects. Here are the results (Figure 18).

The fasted one (Chrome for now) can be compared with C++, LLVM IR, WAVM native engines, all compiled with float denormal protection on (Figure 19).

7 COMMENTS ON BENCHMARKS

Testing wasm JIT machines across different browsers is not an easy task. Thus being able to use a simpler standalone WebAssembly aware runtime was a good starting point. The C++ WAVM runtime revealed to be an excellent tool to compare the FAUST C++, LLVM IR and wasm backends. Since its code can be easily adapted, what can be expected deploying wasm DSP modules in pure native environments can also be estimated (that is outside of the browser, where some audio specific optimizations may be considered).

Measurements done on a set of FAUST DSPs show that WebAssembly code still runs slower than C++ or LLVM IR generated code in most cases, up to almost 66% slower in the less favorable examples. This value will typically be a bit worse when deploying in browsers, since float denormal protection code has to be used.

Comparing the Chrome, Firefox and WebKit browsers on the tested machine shows that Chrome was the fastest engine in most cases (at the time of testing), with Firefox and WebKit quite similar (with a slight plus for WebKit).

Comparing the Chrome browser with native engines shows results from 4,8 times slower (filterBank.dsp), 2,8 times slower (sTunedBar6.dsp) up to much more favorable cases (karplus32.dsp). Note that filterBank.dsp example is a bit of a pathological case, since Chrome is significantly slower than Firefox and WebKit in this case, and filterBank.dsp uses a lot of pow(10, x) code that is rewritten and optimized as exp10(x) with the C++ backend path.

And finally it has to be said that all those benchmarks compare the browsers and the WAVM standalone runtime at time t, in a fast changing situation that hopefully will improve progressively.

8 CONCLUSION

The WebAssembly format promises to be an excellent compilation target for Domain Specific Like languages like FAUST. Performance in the different tested browsers is already quite good, and we can expect improvements as the wasm implementation matures.

The float denormals handling remains a serious issue, since adding software protection code cannot be considered a long terme and satisfactory solution. We hope that future versions of the WebAssembly specification will properly address this question.

The WAVM project shows that non-Web embeddings are perfectly possible, even with audio code aware customized runtimes, and opens interesting deployment possibilities.

In the context of the WebAudio API, the AudioWorklet implementation is still quite fresh and needs to be more thoroughly tested.

REFERENCES

[1] John Backus. Can Programming Be Liberated from the von Neumann Style ? *ACM Turing Award lecture*, pages 613–641, 1978.
[2] C. Clark and A. Tindale. Flocking: a framework for declarative music-making on the Web. In *Proceedings of the International Computer Music Conference*, 2014.
[3] Sarah Denoux, Stéphane Letz, Yann Orlarey, and Dominique Fober. FAUSTLIVE, Just-In-Time Faust Compiler... and much more. In *Proceedings of the Linux Audio Conference 2014, ZKM, Karlsruhe, Germany*, 2014.
[4] Sarah Denoux, Yann Orlarey, Stéphane Letz, and Dominique Fober. Calcul d'une expression Faust équivalente à partir d'un graphe d'applications. In *Journées d'Informatique Musicale*, 2016.
[5] V. Lazzarini, E. Costello, S. Yi, and J. Fitch. Csound on the Web. In *Proceedings of the Linux Audio Conference*, 2014.
[6] S. Letz, Y. Orlarey, and D. Fober. Comment embarquer le compilateur Faust dans vos applications? In *In Proceedings of the Journees d'Informatique Musicale*, 2013.
[7] Yann Orlarey, Dominique Fober, and Stephane Letz. An algebra for block diagram languages. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 542–547, 2002.
[8] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and Semantical Aspects of Faust. *Soft Computing*, 8(9):623–632, 2004.
[9] Yann Orlarey, Dominique Fober, and Stephane Letz. Parallelization of audio applications with Faust. In *Proc. of the 6th Sound and Music Computing Conference, Porto, PT*, pages 99–112, 2009.
[10] Charlie Roberts. Strategies for Per-Sample Processing of Audio Graphs in the Browser. In *Proceedings of the Web Audio Conference*, 2017.
[11] Gheorghe Stefanescu. Algebra of flownomials. *Institut für Informatik, Technical University Munich, Report TUM- I9437*, 1994.
[12] A. Zakai. Emscripten: an LLVM to JavaScript compiler. In *In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, 2011.