Compiling Faust audio DSP code to WebAssembly

Stéphane Letz
GRAME
letz@grame.fr

Yann Orlarey
GRAME
orlarey@grame.fr

Dominique Fober
GRAME
fober@grame.fr

ABSTRACT

After a first version based on asm.js [4], we show in this paper how the FAUST audio DSP language can be used to generate efficient Web Audio nodes based on WebAssembly. Two new compiler backends have been developed. The *libfaust* library version of the compiler has been compiled for the Web, thus allowing to have an efficient compilation chain from FAUST DSP sources and libraries to audio nodes directly available in the browser.

Keywords

Web Audio API, WebAssembly, FAUST, Domain Specific Language, DSP, audio, real-time.

1. WEB AUDIO PROGRAMMING

1.1 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 wanted 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[1] aims in 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 is not yet officially implemented in any browser [2].

[1]https://webaudio.github.io/web-audio-api/#rendering-loop
[2]Although development seems to move on in Firefox, see https://bugzilla.mozilla.org/show_bug.cgi?id=1062849

Web Audio Conference WAC-2017, August 21–23, 2017, London, UK.

1.2 Extending the Web Audio API

Various JavaScript DSP libraries or musical languages, have been developed over the years [1], 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 [2]. Using the C/C++ to JavaScript Emscripten [3] compiler, the complete C written Csound runtime and DSP language is now available in the context of the Web Audio API.

2. FAUST DSP AUDIO LANGUAGE

In this paper we demonstrate a third alternative based on FAUST, a compiled Domain Specific Language (DSL). FAUST [Functional Audio Stream] [5] is a functional, synchronous, domain-specific programming language specifically designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing music languages like Max/MSP[3], PureData, Supercollider etc., is that programs are not interpreted, but fully compiled. FAUST provides a high-level alternative to C/C++ to implement efficient sample-level DSP algorithms. *Architecture files* have to be used to wrap the compiler generated code to connect it to the external world, that is different kind of controllers (OSC, MIDI, GUI), and the audio drivers to render the audio stream[6].

The FAUST compiler currently exists in two flavors. The official master branch only generates C++ code, the Faust2 branch can generate several other target languages, using an intermediate FIR representation (FAUST Imperative Representation), to be translated in the destination 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.). The *language of signals* (internal to the compiler) is compiled into the FIR. Several backends have been developed (Figure 1) to translate the FIR in C, C++, Java, JavaScript, asm.js, WebAssembly and LLVM IR[4].

[3]the *gen* object added in Max6 now creates compiled code from a patch-like representation, using the LLVM based technology.
[4]Low Level Virtual Machine Intermediate Representation.

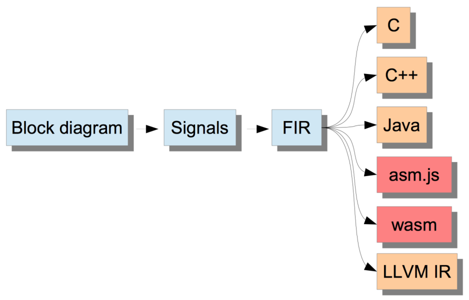

Figure 1: Faust2 compilation chain

Furthermore, the FAUST compiler has been packaged as an embeddable library called *libfaust*, published with an associated API [7] based on a factory/instance model.

3. COMPILING FOR THE WEB

3.1 From asm.js to WebAssembly

Mozilla developers have started in 2011 the Emscripten compiler project [3], based on LLVM technology, that generates from C/C++ sources, 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[5] and asm.js projects have designed WebAssembly[6], 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.

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 recently finalized with a binary format, as well as a textual format that looks like traditional assembly languages.

3.2 Compiling to WebAssembly

Two new *wast* and *wasm* FAUST backend 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 equivalent binary format to be directly loaded and executed in browsers.

3.2.1 Module definition

A module is a distributable, loadable, and executable unit of code in WebAssembly, instantiated at runtime with a set of import values to produce instances. The backends have to translate the intermediate FIR code to follow the required module format. All FIR functions are translated in a set of exported functions with their definition. Prototypes of required mathematical functions not part of the WebAssembly specification are generated in the *import* section. Code for 32 or 64 bits float format can be generated, with the adapted version of mathematical functions and memory access code.

3.2.2 Memory management

Modules may define an internal *linear memory* area, or can import it from the JavaScript context. The memory zone of the generated DSP contains the main DSP object, as well as *inlined* sub-objects or *waveforms* [7]. Fields of the DSP object are addressed with their computed index.

When a single DSP object is generated, the module internal memory is used. On the contrary 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), an externally defined memory zone from JavaScript context is imported.

3.2.3 Denormals handling

A specific problem occurs when audio computation produces *denormal* float values, which is quite common with recursive filters, and can be extremely costly to compute on some processors like the Intel family for instance. A Flush To Zero (FTZ) mode for denormals can usually be set at hardware level, but it not yet available in the MVP version, which strictly conform to the IEEE 754 norm [8]. Thus an automatic software strategy which consists in adding FTZ code in all recursive loops has been implemented [9].

3.2.4 Additional JavaScript glue code

An additional JavaScript file containing helper functions (to get the DSP JSON representation, table of controller paths etc.) is also generated.

3.3 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 to the module.

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 wasm compiled instance:

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

The user interface can be retrieved as a JSON description:

```
var jd = dsp.json();
```

The instance can be used with the following kind of code:

```
dsp.connect(context.destination);
dsp.setParamValue("path_to_control", 0.5);
```

[5]Google Portable Native Client (PNaCl) is a sandboxing technology for running a subset of Intel x86, ARM, or MIPS native code in a sandbox.

[6]As asm.js model done correctly, see http://webassembly.org

[7]The FAUST backend for a more structured langage like C++ typically generates sub-classes in this case.

[8]https://github.com/WebAssembly/design/issues/148

[9]Using the -ftz 1 or -ftz 2 parameters in the compiler.

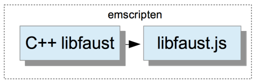

Figure 2: Compiling C++ libfaust to libfaust.js with Emscripten

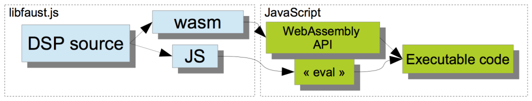

Figure 3: libfaust.js + wasm dynamic compilation chain

3.4 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 in pure JavaScript and WebAssembly (Figure 2).

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, using the wasm backend, compiling the DSP source code, and producing the module as an array of bytes and helper JavaScript functions as a string (Figure 3).

Using WebAssembly API again and JavaScript "eval" function, allows to deploy it in JavaScript context. 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);
```

A fully working DSP "instance" as a Web Audio node is then created with the code:

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

The DSP instance can then be controlled with the API described in section 3.3. Note that the high-level JavaScript API stays the same with asm.js and wasm backends.

4. 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.

4.1 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. The *-comb* parameter can be added to compile several .dsp source files in a unique resulting JavaScript file.

Audio nodes can be created and activated. A 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.

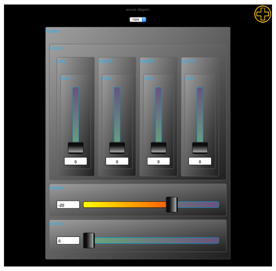

Figure 4: Self-contained HTML page loading the wasm module

4.2 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 4).

Adding the *-links* parameter to the script makes the HTML page also contains links to the original DSP textual file, as well as the block-diagram SVG representation. Thus it becomes quite simple to publish reusable DSP algorithms.

4.3 Web embedded compiler

Having the FAUST compiler itself as a library in the browser opens interesting capabilities experimented in the *FaustPlayground* application[10].

FaustPlayground (Figure 5) 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.

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 realization to all the platforms supported by the online compilation service[11]. 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.

FaustPlayground is still using the asm.js version of the libfaust.js FAUST compiler library. A WebAssembly version is in progress.

[10]http://faust.grame.fr/faustplayground
[11]http://faustservice.grame.fr

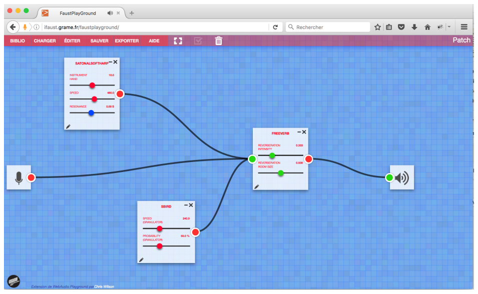
Figure 5: FaustPlayground dynamic compilation platform

5. TESTS AND BENCHMARKS

WebAssembly and asm.js backends have been tested on a 4 cores MacBook Pro Core i7 2,2 GHz using Firefox 54.0.1, Chrome 59.0 and WebKit (development version) browsers. All backends are generating code with the same default "scalar" generation model [12]. The benchmarks have been done using the Activity Monitor tool by simply comparing the application CPU use with two different DSP programs, running at 44.1kHz and 1024 frames per audio buffer.

This first one (Table 1) compares native code (deployed using the libfaust + LLVM chain) using hardware FTZ, and wasm code (using software FTZ protection code) running on three browsers.

DSP code	native	Chrome	Firefox	WebKit
STunedBar6	4.0 %	11.8 %	17.6 %	14.8 %
frenchBell	1.8 %	4.6 %	7.4 %	5.3 %

Table 1: Global CPU use of the application tested with native, asm.js and wasm backends

The second one (Table 2) compares asm.js and wasm under Chrome with FTZ protection code activated.

DSP code	Chrome asm.js	Chrome wasm
STunedBar6	15.4 %	11.6 %
frenchBell	5.0 %	4.6 %

Table 2: Global CPU use of the application tested with asm.js and wasm backends

The third one (Table 3) compares wasm under Firefox with and without FTZ protection code.

Even with this rather limited testing method, some interesting results emerge. With the first official version, WebAssembly based audio nodes already outperform asm.js ones, which is quite encouraging. Comparing the browsers shows that Chrome currently wins the race, but this may change since work is still done to optimize WebAssembly implementations on all of them. Until an hardware solution can be found, the use of FTZ software protection is almost mandatory to have good performances.

[12]In this mode, the compute method uses a single global DSP loop

DSP code	Firefox FTZ	Firefox no FTZ
STunedBar6	17.6 %	21.1 %
frenchBell	7.4 %	35.0 %

Table 3: Global CPU use of the application tested using wasm backends with and without FTZ protection code

Note that native code can benefit from compilation optimization steps like auto-vectorization, which is not yet the case for the WebAssembly model.

6. CONCLUSION

We have demonstrated how the Faust audio DSP language can be used to easily develop new audio nodes in the Web Audio model, and use them in an audio graph. Complete HTML pages with a working user interface can also be generated. Having the dynamic compilation chain directly available in the browser is also interesting to further explore.

The community still waits for the AudioWorklet specification to be implemented with its promise of real-time low latency audio rendering. Proper handling of float denormals has also to be done. And finally the Web MIDI API [13] is also a much needed addition to the ecosystem in order to develop professional level applications on the Web.

Acknowledgments

This work has been done under the FEEVER project [ANR-13-BS02-0008] supported by the "Agence Nationale de la Recherche".

7. REFERENCES

[1] C. Clark, A. Tindale, "Flocking: a framework for declarative music-making on the Web", *International Computer Music Conference*, 2014.

[2] V. Lazzarini, E. Costello, S. Yi and J. Fitch, "Csound on the Web", *Linux Audio Conference*, 2014.

[3] A. Zakai, "Emscripten: an LLVM to JavaScript compiler", *In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications, pages 301–312. ACM* , 2011.

[4] S. Letz, S.Denoux, Y. Orlarey and D. Fober, "Faust audio DSP language in the Web", *Linux Audio Conference*, 2015.

[5] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust", *Soft Computing*, 8(9), 2004, pp. 623–632.

[6] D. Fober, Y. Orlarey, and S. Letz, "Faust Architectures Design and OSC Support", *IRCAM*, (Ed.): Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11), pp. 231-216, 2011.

[7] S. Letz, Y. Orlarey and D. Fober, "Comment embarquer le compilateur Faust dans vos applications?", *Journees d'Informatique Musicale*, 2013.

[13]https://webaudio.github.io/web-midi-api/