Proceedings of the 1st International Faust Conference (IFC-18), Mainz, Germany, July 17–18, 2018

AN OVERVIEW OF THE FAUST DEVELOPER ECOSYSTEM

Stéphane Letz
GRAME
Lyon, France
letz@grame.fr

Yann Orlarey
GRAME
Lyon, France
orlarey@grame.fr

Dominique Fober
GRAME
Lyon, France
fober@grame.fr

ABSTRACT

The FAUST language has been designed to provide developers an alternative to C/C++ code, to easily develop and deploy DSP algorithms, effects, instruments etc. The ecosystem is composed of the language and its compiler, as well as different components that help test, benchmark and optimize, and run the resulting code on a large variety of platforms.

In this paper we present various architectures files, optimization and testing tools, that have been developed over the years as part of the FAUST ecosystem, in order to expand the use of the compiler on various targets, and help developers optimize their DSP code. Some of them were publicly announced and can help when deploying DSPs, some are more experimental to be tested by more adventurous developers.

1. INTRODUCTION

The FAUST compiler was initially developed and distributed as a standalone program, producing a C++ class from a given DSP source, to be wrapped by *architecture files* [1], containing the audio driver and controller part, and deployed as ready to use standalone applications or plug-ins, using so-called *faust2xx* scripts. As more deployment targets were added, new architecture files and scripts were progressively written.

At the same time, the compiler was reworked to be usable as an embeddable library called *libfaust*, associated with an LLVM IR backend. LLVM is a compiler infrastructure project, as a collection of modular and reusable compiler and toolchain technologies, that can be used to develop compiler front ends and back ends. Linked with components of the LLVM compiler toolchain, in particular its JIT LLVM IR to executable code module, the *libfaust* library allows to deploy a complete dynamic compilation chain, from DSP source to executable code, in applications or plug-ins.

Backends for additional target languages (Java, asm.js, Web Assembly, Interpreter) have been developed. The interpreter backend gives an alternate way to deploy the dynamic compilation chain, which can be useful in some very specific situations. The Web as a universal platform has been more recently targeted with the WebAssembly backend and associated JavaScript wrapper files [2]. Finally tools to help deploying the compiled code have been written.

The paper presents the most useful tools, and is divided in four parts: section 2 explains how the dynamic compilation chain works, section 3 demonstrates several optimisation tools, section 4 details new developments done in the *libfaustremote* library, and section 5 presents more experimental developments.

2. DYNAMIC COMPILATION

Dynamic compilation allows more flexible deployment of the compilation chain, as already demonstrated in the FaustLive application [3] or the *faustgen~* Max/MSP external. To further extend the same idea, new backends besides the LLVM one have been developed.

2.1. The interpreter backend in libfaust

The interpreter backend has been first written to allow dynamical compilation on iOS, where Apple does not allow LLVM based JIT compilation to be deployed, but can also be used to develop testing tools (see section 3.4). It has been defined as a typed language and a virtual machine to execute it.

2.1.1. Interpreter 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 to several target languages. The FIR language describes the computation performed on the samples in a generic manner. It contains primitives to read and write scalar variables and arrays, do arithmetic operations, and defines the necessary control structures (for and while loops, if or select statements for branching etc.).

The FIR language is simple enough to be easily translated in the typed bytecode for an interpreter, which contains the following kind of instructions:

- memory load/store operations using indexes as offset in global *real* (float or double) and *integer* heaps, to be used for scalar or array access
- mathematical operations taking their arguments from the stack or from the heap
- control operations (select, if and for loop).

The bytecode is generated by a *FIR to bytecode* compilation pass. The virtual machine then executes the bytecode on a stack based machine, with a *real* typed stack and an *integer* typed stack. Heap memories for reals and integers contain the DSP state with delay lines, waveforms, control parameters, to be initialized at init time (for instance with default controller values). Typed (as integer or real) intermediate results are pushed on the corresponding stack. Cast operations typically operate on a value from one type read on the corresponding stack, then casted and pushed on the other one.

The virtual machine can execute the simple version of the bytecode directly produced by the FIR to bytecode pass, which assumes for instance that arguments to mathematical operations are always to be taken from the stack. Bytecode optimization passes can be used to group successive operations, and generate faster

ones, like variants of mathematical operations that directly take their arguments at a given heap location, instead of the stack.

2.1.2. The interpreter API

The interpreter backend API is similar to the LLVM backend API [4]. The compilation step is done through the `createInterpreterDSPFactory` function. Given a FAUST source code (as a file or a string), the compilation chain (FAUST + interpreter backend) generates the "prototype" of the class, as an `interpreter_dsp_factory` pointer. This factory actually contains the compiled bytecode for the given DSP.

Next, the `createDSPInstance` method of the factory class, corresponding to the `new className` of C++, instantiates an `interpreter_dsp` pointer, to be used as any regular FAUST compiled DSP object, run and controlled through its interface. The instance contains the interpreter virtual machine loaded with the compiled bytecode, to be executed for each method.

After the DSP factory has been compiled, the application or plugin may want to save/restore it in order to save FAUST to interpreter bytecode compilation at next use. To get the internal factory bytecode and save it, two functions are available:

- `writeInterpreterDSPFactoryToMachine` allows to get the interpreter bytecode as a string
- `writeInterpreterDSPFactoryToMachineFile` allows to save the interpreter bytecode in a file

To re-create a DSP factory from a previously saved code, two functions are available:

- `readInterpreterDSPFactoryFromMachine` allows to create a DSP factory from a string containing the interpreter bytecode
- `readInterpreterDSPFactoryFromMachineFile` allows to create a DSP factory from a file containing the interpreter bytecode

The complete API is available and documented in the installed *faust/dsp/interpreter-dsp.h* header [1]. Note that only the scalar compilation mode is supported.

2.1.3. Performance

The generated code is obviously much slower than LLVM generated native code. Measurements on various DSPs examples have been done, and the code is between 3 and more than 10 times slower than the LLVM native code.

2.1.4. Use cases

The typical use-case is an extension of the *FaustLive* application, where DSP source is edited and tested in FaustLive. The DSP source is sent to an iOS based mobile machine using the HTTP protocol [2], dynamically compiled and executed, possibly at a lower sample-rate to compensate for the higher DSP CPU usage. Another possibility is to cross-compile the DSP source to the interpreter bytecode on the FaustLive hosting machine, to be sent and executed on the iOS one, where the virtual machine part only has to be deployed. When finished, the application can be compiled to fast and native code using the standard *faust2ios* script.

Another idea that needs further exploration: since the virtual machine is under complete control, instrumenting the DSP code (adding signal sensors to look at internal signals at specific locations in the generated code) could possibly be done.

2.2. Using libfaust and the WebAssembly backend

WebAssembly [3] is a new Web standard that defines a binary format and a corresponding assembly-like text format for executable code in Web pages. It allows to execute code nearly as fast as native code. It is envisioned to complement JavaScript to speed up performance-critical parts of Web applications and later on to enable web development in other languages than JavaScript.

The WebAssembly backend allows to generate textual or binary code, to be usually deployed on the Web and executed in browsers. Standalone WebAssembly supporting runtimes start to appear, promising to help deploying dynamically generated code in various contexts, outside of the Web. So the wasm format will presumably become a cross-platform format, quite interesting to use in the audio domain [5]. Here are three solutions, two of them can already be used, the last one is still in progress and will need additional glue code to be written.

2.2.1. Using Node.js

Node.js [4] is an open-source, cross-platform JavaScript run-time environment, built on Chrome's V8 JavaScript engine, typically for executing JavaScript code on the server-side. Since recent versions now support WebAssembly, FAUST generated wasm modules can now be loaded, compiled using the `WebAssembly.compile` function, instantiated using the function `WebAssembly.Instance`, and finally driven by JavaScript code.

The *architecture/webaudio/wasm-standalone-node-wrapper.js* loader file example can be used. It implements a wrapper of the wasm module to expose the DSP API as JavaScript functions.

2.2.2. Using the WAVM runtime

The WAVM project, developed in C++, compiles WebAssembly to native code using the LLVM technology [5]. Compared to native code, performances are already good, and can possibly be further improved by adapting the LLVM JIT generated code to the specificities of audio code [5].

A `wasm_dsp` class, which is a subclass of the base `dsp` class has been developed. It contains the glue code to load the wasm module, compile it with the WAVM machinery, get pointers to the JIT compiled functions, and make them accessible as methods of the `wasm_dsp` class. A newly allocated `wasm_dsp` object can then be used with audio drivers or UI classes as usual.

The FAUST related code has been developed on a WAVM fork here: `https://github.com/sletz/WAVM`.

[1] Note that the whole *faust/xxx/yyy* header hierarchy will be installed in the system after the standard `make install` step

[2] Using a libmicrohttpd (`https://www.gnu.org/software/libmicrohttpd/`) based solution for instance.

[3] `https://webassembly.org`

[4] `https://nodejs.org`

[5] `https://github.com/AndrewScheidecker/WAVM`

2.2.3. Using cretonne and wasmstandalone

Cretonne [6] is a Rust written low-level retargetable code generator, designed to be a code generator for WebAssembly. It translates a target-independent intermediate language into executable machine code. Associated with the wasmstandalone project [7], it will allow to deploy FAUST DSP code compiled to wasm [8].

2.3. Testing libfaust and LLVM/interpreter backends

The *libfaust* API is published and documented in the *faust/dsp/llvm-dsp.h* public header file for its LLVM backend, and *faust/dsp/interpreter-dsp.h* public header file for its interpreter backend.

The *dynamic-jack-gtk* tool has been developed to test them. It uses the dynamic compilation chain, compiles a FAUST DSP source, and runs it with the LLVM or interpreter backend. Command line parameters allow to choose between LLVM or interpreter backends, activate polyphonic and MIDI modes, OSC and httpd controllers:

- `-llvm|interp` to choose either LLVM or interpreter backend
- `-nvoices N` to start the DSP in polyphonic mode with N voices
- `-midi` to activate MIDI control
- `-osc` to activate OSC control
- `-httpd` to activate HTTPD control

The *dynamic-jack-gtk* tool is located in tools/benchmark and has to be compiled separately. After having prepared the needed dependancies (*libfaust* with embedded LLVM and Interpreter backends), use `make & sudo make install` to compile and install it.

3. OPTIMISATION TOOLS

The FAUST compiler has a lot of different compilation parameters to explore. Code can be generated in the so-called *scalar mode* (one big loop), or as a *graph of smaller loops* connected with buffers [9] and more amenable to auto-vectorization or even parallelisation. Size of internal buffers in vector mode and handling of delays lines can be changed. The typical parameters to explore are:

- `-vec` generate easier to vectorize code with the associated *-lv* [0:fastest (default), 1:simple]
- `-g` group single-threaded sequential tasks together if -omp or -sch is used
- `-fun` separate tasks code as separated functions (in -vec, -sch, or -omp mode)
- `-mcd` threshold between copy and ring buffer implementation (default 16 samples), used for delay lines

[6] `https://github.com/stoklund/cretonne`

[7] `https://github.com/sunfishcode/wasmstandalone`

[8] At the time of writing, the code is not yet ready to be tested.

[9] Using with vector (-vec) and parallel (-omp, -sch) code generation modes

Discovering the best combination of parameters to get maximum CPU throughput for a given DSP program can be automated. Two `measure_dsp` and `dsp_optimizer` classes are available for developers to measure DSP CPU use directly in their code. Using them, two *faustbench* and *faustbench-llvm* tools have been developed. They allow to discover the best FAUST compiler parameters, to be used later on with *faust2xx* scripts, *faustgen~* Max/MSP external or *FaustLive*.

3.1. The measure_dsp and dsp_optimizer DSP classes

The `measure_dsp` class defined in the *faust/dsp/dsp-bench.h* file allows to decorate a given DSP object and measure its compute method CPU consumption. Results are given in Megabytes/seconds (higher is better). Here is a C++ code example of its use:

```
void bench(dsp* dsp, const string& name)
{
    // Init the DSP
    dsp->init(48000);
    // Wraps it with a 'measure_dsp' decorator
    measure_dsp mes(dsp, 1024, 5);
    // Measure the CPU use
    mes.measure();
    // Print the stats
    cout << mes.getStats() << endl;
}
```

Defined in the *faust/dsp/dsp-optimizer.h* file, the `dsp_optimizer` class uses the libfaust library and its LLVM backend to dynamically compile DSP objects produced with different FAUST compiler options, and then measure their DSP CPU. Here is a C++ code example of its use:

```
void dynamic_bench(const string& dsp_source)
{
    // Init the DSP optimizer with the dsp_source
    dsp_optimizer optimizer(dsp_source, "/usr/
        local/share/faust", "", 1024);
    // Discover the best set of parameters
    pair<double, vector<string>> res = optimizer.
        findOptimizedParameters();
}
```

Starting from this C++ code, several tools have been developed.

3.2. The faustbench tool

The *faustbench* tool uses the C++ backend to generate a set of C++ files produced with different FAUST compiler options. All files are then compiled to a unique binary that will measure DSP CPU of all versions of the compiled DSP, and find the best set of parameters.

When used on a standard machine, the tool is supposed to be launched in a terminal. To facilitate testing on the iOS mobile platform, it can also be used to generate an Xcode project, ready to be launched and tested, with results simply printed on the console.

3.3. The faustbench-llvm tool

The *faustbench-llvm* tool uses the libfaust library and its LLVM backend to dynamically compile DSP objects produced with different FAUST compiler options, and then measure their DSP CPU. Since the dynamic compilation chain is used, additional compiler options can be added, beside the ones that will be automatically explored by the tool.

3.4. The interp-tracer tool

The interpreter backend allows to execute the DSP code on a virtual machine, that can be easily instrumented to test various runtime code characteristics.

The *interp-tracer* tool runs and instruments the compiled program using the interpreter backend. Various statistics on floating-point and integer numbers computation are collected and displayed while running and/or when closing the application:

- *FP_SUBNORMAL* indicates that the value is subnormal
- *FP_INFINITE* indicates that the value is not representable by the underlying type (positive or negative infinity)
- *FP_NAN* indicates that the value is not-a-number (NaN)
- *INTEGER_OVERFLOW* indicates a overflow in integer computation
- *DIV_BY_ZERO* indicates a division by zero

Some of them can indicate a real issue in the generated code (still not detected at compile time by the FAUST compiler), some like the *INTEGER_OVERFLOW* are expected behaviors for some algorithms like noise generation for instance. Several modes can be used:

```
interp-trace -trace <1-5> -control [additional
    Faust options (-ftz xx)] foo.dsp
```

Mode 4 and 5 allow to display the stack trace of the running code when *FP_INFINITE*, *FP_NAN* or *INTEGER_OVERFLOW* values are produced. The *-control* mode allows to check control parameters, by explicitly setting their min and max values, to possibly detect out-of range generated values (still experimental).

All these tools are located in tools/benchmark and have to be compiled separately. After having prepared the needed dependancies (*libfaust* with embedded LLVM and Interpreter backends), use `make & sudo make install` to compile and install them.

4. REMOTE DSP DEPLOYMENT

4.1. The libfaustremote library

With the availability of dynamic compilation chains, DSP code can be remotely deployed, possibly migrating too heavy CPU needs on more powerful machines, or separating control, compute, and rendering steps in more flexible setups.

Following a client/server model, the *libfaustremote* allows to develop distributed compilation or execution setups. Using a proxy like API, DSPs can be compiled and executed on distant machines. Audio streams are sent, processed, and received using NetJack, as in the following typical use-case (Figure 1).

The design and associated API has been previously presented in [6]. Extensions have been developed since, allowing to simply send and execute DSPs on the remote machine, using its own audio driver:

- `createRemoteAudioInstance` allows to create a remote DSP audio instance (a DSP instance wrapped with an audio driver), from a given remote factory, with `start` and `stop` methods
- `deleteRemoteAudioInstance` destroys the remote DSP audio instance

On the server side, the following functions can be used to control the server:

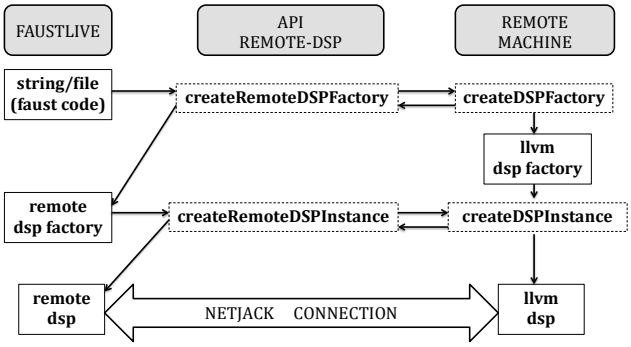

Figure 1: Remote compilation and processing: audio streams are sent, processed, and received on the client machine

- `createRemoteDSPServer` creates a FAUST remote DSP server
- `deleteRemoteDSPServer` destroys the server

The server can then be started and stopped. Callbacks can be set up to get notifications when new DSP factories and instances are created or destroyed from the client side.

The complete (but still experimental) API is available and documented in the *embedded/faustremote/remote-dsp.h* header.

5. EXPERIMENTAL DEVELOPMENTS

5.1. The Rust backend

Rust is a system programming language initially developed by Mozilla Research, which describes it as a "safe, concurrent, practical language." Rust is syntactically similar to C++, with better memory safety while maintaining performance. In the audio domain where precise control of memory, thread usage and data integrity in a real-time context is critical, it will probably take a place quite rapidly. Since the language is still new, only few audio applications or libraries have been developed [10].

An experimental Rust backend has been added in the FAUST compilation chain. It is only able to produce scalar code for now. A public data structure containing typed named fields, with a set of associated functions is generated, following the regular C++ class model used in the C++ backend.

A very basic JACK based architecture file has been developed, to be used with the *faust2jackrust* script. It allows to generate and test standalone programs, but the controller interface is not yet connected to the compiled DSP and parameters cannot be changed dynamically yet.

6. CONCLUSION

Tools to expand the deployment of the FAUST compiler, to test the API, optimize and benchmark the DSP code, have been presented. We invite developers to test and extend them, and contribute their improvements back to the FAUST ecosystem.

[10] `https://github.com/RustAudio`

7. REFERENCES

[1] Dominique Fober, Yann Orlarey, and Stéphane Letz, "Faust architectures design and osc support," in *14th Int. Conference on Digital Audio Effects (DAFx-11)*, 2011, pp. 231–216.

[2] Stéphane Letz, Yann Orlarey, and Dominique Fober, "Compiling Faust audio DSP code to WebAssembly," in *Proceedings of the Web Audio Conference 2017, Queen Mary University, London, England*, 2017.

[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] Yann Orlarey, "Version librairie du compilateur Faust," in *INEDIT Project, ANR-12-CORD-0009, Programme ContInt*, 2014.

[5] Stéphane Letz, Yann Orlarey, and Dominique Fober, "FAUST Domain Specific Audio DSP Language Compiled to WebAssembly," in *Proceedings of TheWebConf 2018, Lyon, France*, 2018.

[6] Stéphane Letz, Yann Orlarey, and Dominique Fober, "Audio Rendering/Processing and Control Ubiquity? A Solution Built Using the Faust Dynamic Compiler and JACK/NetJack," in *Proceedings of the International Computer Music Conference 2014, Athens, Grece*, 2014.