A NEW INTERMEDIATE REPRESENTATION FOR COMPILING AND OPTIMIZING FAUST CODE

Yann Orlarey, Stéphane Letz, Dominique Fober, Romain Michon

Proceedings of the 2nd International Faust Conference (IFC-20), Maison des Sciences de l'Homme Paris Nord, France, December 1-2, 2020

A NEW INTERMEDIATE REPRESENTATION FOR COMPILING AND OPTIMIZING FAUST CODE

Yann Orlarey,[a] *Stéphane Letz,*[a] *Dominique Fober,*[a] *and Romain Michon,*[a,b]

[a]GRAME – Centre National de Création Musicale, Lyon, France
[b]Center for Computer Research in Music and Acoustics, Stanford University, USA
orlarey@grame.fr

ABSTRACT

The Faust compiler relies on several intermediate representations to translate a Faust program. One step, in particular, consists of moving from a functional representation of computations on infinite signals to an imperative (stateful) representation of computations on samples. This translation phase is complex, as it combines the recursive tree traversal, the division of the computations into instructions, the scheduling, and the code generation. As a result, the implementation of new code generation strategies is difficult to achieve.

In this paper, we propose a new intermediate representation in the form of a graph whose nodes represent computations on infinite signals and the edges time dependencies between these computations. The graph structure makes it much easier to handle recursive dependencies as well as to experiment with all kinds of scheduling strategies. We will present several of them whose performance have been tested with the examples of the Faust distribution. Performance gains can sometimes be quite significant compared to the current compiler.

1. INTRODUCTION

A FAUST program denotes a mathematical function [1] that operates on audio signals. The FAUST compiler aims at producing the most efficient implementation of this function in one of the supported target languages, e.g. C, C++, Java, Rust, WebAssembly, LLVM IR, SOUL, etc. Currently, all these target languages are imperative. We can therefore see the Faust compilation as a translation of a *stateless function on audio signals* into *a stateful imperative program on samples*. For this purpose, we have a specific intermediate representation: the FAUST Imperative Representation (FIR) which is used before generating the actual code.

In this article, we propose a new intermediate representation, located just before the FIR, which makes it possible to specify the order and organization in memory of the computations regardless of the FIR translation. Because of the importance of memory caches [2], the memory layout and the order of the computations have indeed a lot of influence on the performance of the code. It is therefore interesting to be able to propose different compilation strategies in this respect. This is the purpose of the new intermediate representation that we are proposing.

2. FROM SIGNALS TO INSTRUCTIONS GRAPHS

The compilation of a FAUST program is currently based on five steps:

1. The first step, the *lambda-calculus* phase, allows to pass from the algorithmic description of an audio circuit to its evaluated form: a flat circuit that contains only primitive operations. It is the result of this first phase which is represented in SVG diagrams.
2. The second step, called *symbolic propagation*, consists in symbolically propagating signals into the circuit in order to obtain, for each of its outputs, an expression describing how it is computed.
3. Then comes a phase of *symbolic calculation* which aims at simplifying these expressions and putting them in *normal form*. During this phase a number of rewriting rules are applied, e.g., $0 * S$ becomes 0, $S + S$ becomes $2 * S$, etc.
4. These expressions are then translated into another intermediate representation, the *Faust Imperative Representation* (FIR), a kind of idealized imperative programming language.
5. Finally, the FIR representation is translated into the chosen target language, e.g., C++.

The new representation is between phase 3 and phase 4. The signal expressions resulting from symbolic propagation are split into a set of instructions and organized in an oriented graph reflecting the dependencies between these instructions.

2.1. Signals Expressions

The set $\mathbb{S}$ of signal expressions is defined by the following (simplified) rules:

$$\mathbb{S} ::= k \mid \texttt{input}(n) \mid \texttt{op}(s_1, s_2, \ldots) \mid s_1 @ s_2 \mid X_i$$

A signal s is either:

- a constant signal k
- an input signal $\texttt{input}(n)$
- a primitive operation on signals $\texttt{op}(s_1, s_2, \ldots)$
- a delayed signal $s_1 @ s_2$
- an element X_i of a group of mutually recursive signals. The group X has an associated list of definitions $\mathcal{D}(X) = (s_1, s_2, \ldots)$. By extension we will write $D(X_i) = s_i$ the ith definition associated with the symbol X.

2.2. Instructions

An instruction indicates that a signal must be written to memory. This is usually the case for output signals that need to be written to output buffers, delayed signals, and recursive signals that need to be stored in delay lines. It can also be the case for shared intermediate signals that are stored in memory to avoid needless

recalculations. The set $\mathbb{I}$ of instructions is defined by the following rule:

$$\mathbb{I} ::= \texttt{output}(n, s') \mid \texttt{dwrite}(M, s') \mid \texttt{vwrite}(M, s')$$

An instruction $i \in \mathbb{I}$ either:

- writes a signal into an output buffer: $\texttt{output}(n, s')$
- writes a signal into a delay line: $\texttt{dwrite}(M, s')$
- writes a signal into a vector: $\texttt{vwrite}(M, s')$

Here M indicates a memory zone and $s' \in \mathbb{S}'$ is a slightly transformed signal expression where delayed and recursive signals are replaced by signals read from memory:

$$\mathbb{S}' ::= k \mid \texttt{input}(n) \mid \texttt{op}(s'_1, s'_2, \ldots) \mid \texttt{vread}(M) \mid \texttt{dread}(M, s')$$

A transformed signal s' is either:

- a constant signal k
- an input signal $\texttt{input}(n)$
- a primitive operation on signals $\texttt{op}(s'_1, s'_2, \ldots)$
- a read access to a signal in memory $\texttt{vread}(M)$
- an indexed read access to a signal in memory $\texttt{dread}(M, s')$

2.3. From Signals to Instructions

We first introduce the function $\mathcal{T} : \mathbb{S} \to \mathbb{S}' \times \mathcal{P}(\mathbb{I})$ that transforms a signal into a simplified signal and an associated set of instructions. Instructions are introduced when transforming recursive expressions and delays. The function $\mathcal{M}$ assign to a signal s a unique memory area $\mathcal{M}(s) \to M$ that can be a vector or a scalar depending of the context.

$$\text{(const)}\ \frac{}{\mathcal{T}(k) \to k \times \emptyset}$$

$$\text{(input)}\ \frac{}{\mathcal{T}(\texttt{input}(n)) \to \texttt{input}(n) \times \emptyset}$$

$$\text{(fun)}\ \frac{\mathcal{T}(s_1) \to s'_1 \times I_1 \quad \mathcal{T}(s_2) \to s'_2 \times I_2 \quad \ldots}{\mathcal{T}(\texttt{op}(s_1, s_2, \ldots) \to \texttt{op}(s'_1, s'_2, \ldots) \times I_1 \cup I_2 \cup \ldots}$$

$$\text{(rec1)}\ \frac{\begin{array}{cc}\mathcal{D}(X_i) \to s_i & \mathcal{M}(s_i) \to M \\ \mathcal{T}(s_i) \to s'_i \times I_i & \mathcal{T}(s_d) \to s'_d \times I_d\end{array}}{\mathcal{T}(X_i @ s_d) \to \texttt{dread}(M, s'_d) \times \{\texttt{dwrite}(M, s'_i)\} \cup I_i \cup I_d}$$

$$\text{(rec2)}\ \frac{\mathcal{D}(X_i) \to s_i \quad \mathcal{M}(s_i) \to M \quad \mathcal{T}(s_i) \to s'_i \times I_i}{\mathcal{T}(X_i) \to \texttt{dread}(M, 0) \times \{\texttt{dwrite}(M, s'_i)\} \cup I_i}$$

$$\text{(del)}\ \frac{\mathcal{M}(s_a) \to M \quad \mathcal{T}(s_a) \to s'_a \times I_a \quad \mathcal{T}(s_d) \to s'_d \times I_d}{\mathcal{T}(s_a @ s_d) \to \texttt{dread}(M, s'_d) \times \{\texttt{dwrite}(M, s'_a)\} \cup I_a \cup I_d}$$

2.4. Transforming the List of Output Signals into a Set of Instructions

The next step is to transform the list of signals $(s_1, s_2, ...) \in \mathbb{S}$ resulting from the symbolic propagation phase, into a set of instructions $I = \{i_1, i_2, ...\} \in \mathcal{P}(\mathbb{I})$.

$$\frac{\mathcal{T}(s_1) \to s'_1 \times I_1 \quad \mathcal{T}(s_2) \to s'_2 \times I_2 \quad \ldots}{(s_1, s_2, ...) \to \{\texttt{output}(1, s'_1), \texttt{output}(2, s'_2), \ldots\} \cup I_1 \cup I_2 \cup \ldots}$$

2.5. Directed Graph of Instructions

Once we have the set of instructions, we can create a directed graph whose nodes are the instructions of this set. Informally, there is an edge from instruction i to an instruction $j = \texttt{dwrite}(M, s')$ if and only if the signal associated with instruction i contains at least one occurrence of $\texttt{dread}(M, s'_d)$. The value associated with this edge will be the lowest value of the s'_d signal.

There is an edge from instruction i and instruction $j = \texttt{vwrite}(M, s')$ if and only if the signal associated with instruction i contains at least one occurrence of $\texttt{vread}(M)$. In this case, the value associated with this edge will be 0.

Output instructions $\texttt{output}(n, s')$ are never destination on any edge. The graph may have cycles due to recursive signals. But if all edges with a value greater than 0 are removed, the resulting graph is guaranteed to be a DAG.

An Example. The instruction graph is the result of several substeps that we will briefly illustrate using an example, a simple Karplus-Strong string triggered by a button. Here is the Faust code:

```
process = noise*button("play")
        : resonator(80, 1)
with {
 resonator(d,a) = (+ : @(d-1))
                ~ (average : *(a));
 average(x      = x+x':*(0.5);
 random         = +(12345)~*(1103515245);
 noise          = random/2147483647.0;
};
```

Step 1. The first step consists in transforming the signals coming from the symbolic propagation phases into a first set of instructions. The resulting graph, Figure 1, represents the dependencies between instructions.

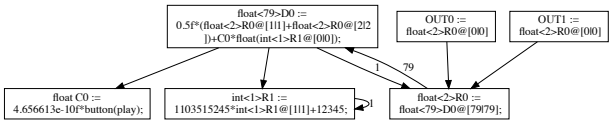

Figure 1: Output signals, delay lines and recursive signals are transformed into instructions

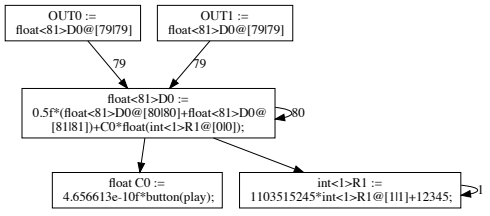

Figure 2: Mutually defined delay lines are merged

As indicated in 2.3, each output signal, delay line or recursive signal is a natural candidate to be an instruction, because it will have to be written in memory at some point. We also take advantage of this phase to factorize control signals so that they are calculated only once per block.

Step 2. The second step is to merge delay lines that can be merged as here R0 and D0. As we can see Figure 2, R0, which is itself a delay on D0, is absorbed by D0.

Step 3. The third step Figure 3 consists in transforming the delay lines into read-write entries in tables dimensioned to the power of two immediately above the maximum delay.

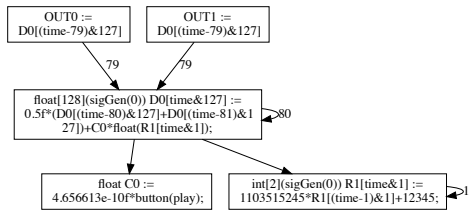

Figure 3: Delay lines are implemented using read-write tables

Step 4. The last step, Figure 4, finally consists in factoring common sub-expressions created by the previous transformations. Here, two time-based index operations are stored in V0 and V1.

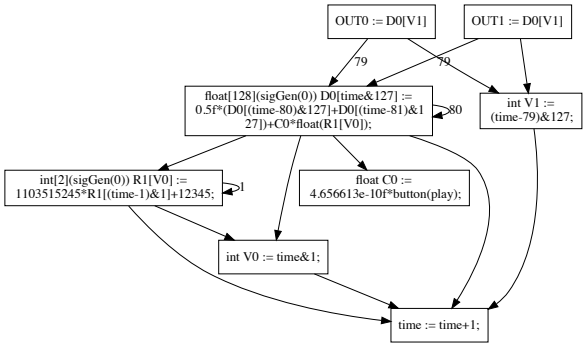

Figure 4: Previously created common sub-expressions are factorized

2.6. Scheduling

To build and manipulate the graph we use the *Digraph* library (https://github.com/grame-cncm/digraph), a very simple, C++-11 template-based, directed graph library. It offers several useful functions for our purpose:

- `graph2dag` transforms a graph with cycles into a directed acyclic graphe (DAG) of cycles using Tarjan's algorithm [3],
- `parallelize` transforms a DAG into a sequence of parallel nodes,
- `serialize` transforms a DAG into a sequence of nodes,
- `cut`(n) transforms a graph by removing all connections with a value greater or equal to n

The scheduling of an instruction graph consists of assigning to each instruction i of the graph an order $O(i)$ which indicates when an instruction must be computed and which respects time dependencies. In other words, if there is a 0-time dependency relationship between two instructions i and j, then $O(i) > O(j)$. In other words, if there is a 0-time dependency relationship between two i and j statements, then i must be computed after j because the computation of i immediately depends on the computation of j.

We have implemented 6 scheduling modes:

1. *mode 0*: transforms the graph into a DAG of cycles, and serialize it. For each node (a graph representing a cycle), cut all connections with values > 1, serialize-it and emit the FIR code,
2. *mode 1*: cut all connections with values > 1, serialize-it and emit the FIR code,
3. *mode 2*: cut all connections with values > 1, parallelize-it and emit the FIR code in a breadth-first manner,
4. *mode 3*: cut all connections with values > 1, serialize-it with priority on output instructions and emit the FIR code,
5. *mode 4*: cut all connections with values > 1, serialize-it with priority on delay lines and emit the FIR code,
6. *mode 5*: Use a deep-first traversal with priority on output instructions and immediate dependencies and emit the FIR code.

3. BENCHMARKS

We tested these 6 modes on a fairly representative set of 26 FAUST programs (see Figure 5). Compared to the current scalar mode, we have the following improvement on average:

- *mode 0* average improvement: 22%,
- *mode 1* average improvement: 25%,
- *mode 2* average improvement: 29%,
- *mode 3* average improvement: 23%,
- *mode 4* average improvement: 25%,
- *mode 5* average improvement: 25%

Now, considering the mode giving the best result for each test (it is not necessarily the same one), we get an average improvement of $+41\%$. But all modes failed to improve `zitaRev.dsp` over the current scalar mode, with a loss of performances of at least -7%. The most spectacular improvement is obtained by mode 1 on `karplus32.dsp`: $+237\%$! Surprisingly, mode 2 is the best mode for 11 of the 26 tests.

4. CONCLUSION

The new intermediate representation that we have proposed in this article aims at simplifying the code generation phase of the Faust compiler which is currently very complex and not very convenient

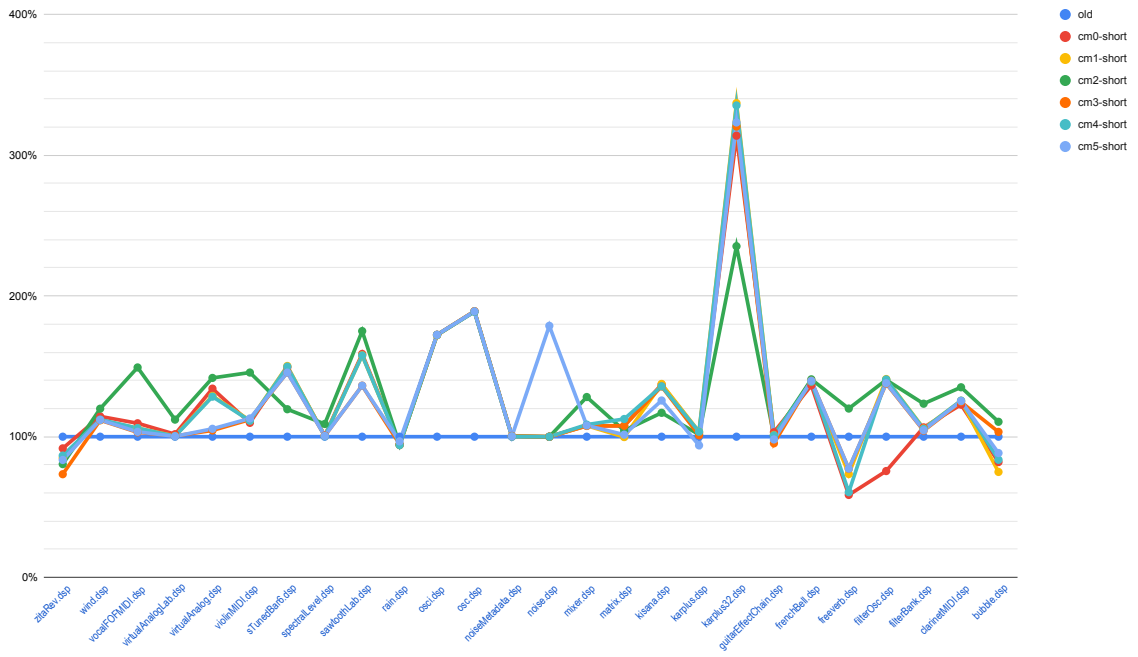

Figure 5: Relative performances of the new compilation modes compared to the current scalar mode on 26 FAUST programs (tested on a 4-cores AMD A10-7700K Radeon R7, Ubuntu 16.04 and GCC 9.2.1)

for experimentation. The idea is to decouple the mapping in memory of the signals from the other operations necessary to generate the code. The latter can thus rely on an explicit graph structure, which is much easier to manipulate and navigate.

The mapping in memory of signals is defined by a notion of instruction that associates a symbolic memory label and a signal. We thus remain in the domain of infinite signals, but we have specified the signals that will have to be written in memory.

The set of instructions thus forms the vertices of a graph whose edges indicate temporal dependencies. One can traverse and transform this graph in many ways. For example, in vector or parallel mode, one can cut all time-dependencies greater than the vector size. This will make it possible to vectorize or parallelize certain recursive computations that can currently only be compiled in scalar mode. It is also very easy to experiment with different scheduling strategies, i.e. different ways of traversing the graph while respecting time dependencies. We have presented several of them.

It is interesting to note that they are simple code reorganizations, which do not involve either algorithm changes or changes in complexity. However, due to the importance of memory caches on modern processors, the differences in performance can be quite significant.

It should be noted that, in all the tests we have performed, none of the strategies is systematically superior to the others. This shows the interest of having, at the compiler level, options to easily choose among these different code organization strategies, especially since manual exploration of this set of variants seems out of reach for a human programmer writing directly in C++.

This is a work in progress, the vector and parallel modes have not yet been implemented and a lot of effort has to be made to extend these new options to all backends. But we think the approach presented here is very promising.

5. REFERENCES

[1] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[2] Alan Jay Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sept. 1982.

[3] Robert Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.