

# Compiling Faust programs

## GRAME – CNCM

Y. Orlarey, S. Letz, S. Denoux

Journée Streaming, 14 avril 2014

# What is Faust ?



Faust is a *Domain-Specific Language* for real-time signal processing and synthesis. A Faust program denotes a *signal processor* :

- A (periodically sampled) *signal* is a *time to samples* function:
  - $S = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a mathematical function that maps a group of  $n$  input *signals* to a group of  $m$  output *signals* :
  - $P = S^n \rightarrow S^m$
- Everything in FAUST is a *signal processor* :
  - $+: S^2 \rightarrow S^1 \in P$ ,
  - $3.14 : S^0 \rightarrow S^1 \in P, \dots$
- Programming in FAUST is essentially combining signal processors :
  - $\{:, <:, :>, \sim\} \subset P \times P \rightarrow P$

# What is Faust ?



Faust is a *Domain-Specific Language* for real-time signal processing and synthesis. A Faust program denotes a *signal processor* :

- A (periodically sampled) *signal* is a *time to samples* function:
  - ▶  $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a mathematical function that maps a group of  $n$  input *signals* to a group of  $m$  output *signals* :
  - ▶  $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - ▶  $+: \mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$ ,
  - ▶  $3.14 : \mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \dots$ ,
- Programming in FAUST is essentially combining signal processors :
  - ▶  $\{:, <:, :> \sim\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

# What is Faust ?



Faust is a *Domain-Specific Language* for real-time signal processing and synthesis. A Faust program denotes a *signal processor* :

- A (periodically sampled) *signal* is a *time to samples* function:
  - ▶  $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a mathematical function that maps a group of  $n$  input *signals* to a group of  $m$  output *signals* :
  - ▶  $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - ▶  $+: \mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$ ,
  - ▶  $3.14 : \mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \dots$ ,
- Programming in FAUST is essentially combining signal processors :
  - ▶  $\{:, <:, :> \sim\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

# What is Faust ?



Faust is a *Domain-Specific Language* for real-time signal processing and synthesis. A Faust program denotes a *signal processor* :

- A (periodically sampled) *signal* is a *time to samples* function:
  - ▶  $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a mathematical function that maps a group of  $n$  input *signals* to a group of  $m$  output *signals* :
  - ▶  $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - ▶  $+: \mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$ ,
  - ▶ **3.14** :  $\mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \dots$ ,
- Programming in FAUST is essentially combining signal processors :
  - ▶  $\{:, <:, :> \sim\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

# What is Faust ?

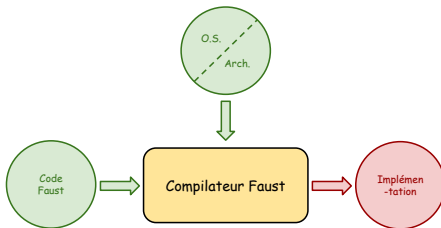


Faust is a *Domain-Specific Language* for real-time signal processing and synthesis. A Faust program denotes a *signal processor* :

- A (periodically sampled) *signal* is a *time to samples* function:
  - ▶  $\mathbb{S} = \mathbb{N} \rightarrow \mathbb{R}$
- A *signal processor* is a mathematical function that maps a group of  $n$  input *signals* to a group of  $m$  output *signals* :
  - ▶  $\mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$
- Everything in FAUST is a *signal processor* :
  - ▶  $+: \mathbb{S}^2 \rightarrow \mathbb{S}^1 \in \mathbb{P}$ ,
  - ▶  $3.14 : \mathbb{S}^0 \rightarrow \mathbb{S}^1 \in \mathbb{P}, \dots$ ,
- Programming in FAUST is essentially combining signal processors :
  - ▶  $\{:, <:, :> \sim\} \subset \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$

# Role of the Faust Compiler

Generate efficient implementations. . .



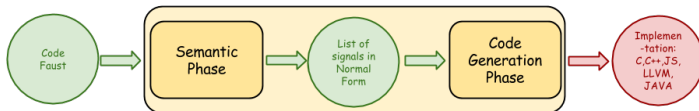
FAUST file name	STK	FAUST	Difference
blowBottle.dsp	3.23	2.49	22.91
blowHole.dsp	2.70	1.75	35.19
bowed.dsp	2.78	2.28	17.99
brass.dsp	10.15	2.01	80.20
clarinet.dsp	2.26	1.19	47.35
flutestk.dsp	2.16	1.13	47.69
saxophony.dsp	2.38	1.47	38.24
sitar.dsp	1.59	1.11	30.19
tibetanBowl.dsp	5.74	2.87	50

Table 2: Comparison of the performance of Pure Data plug-ins using the STK C++ code with their FAUST generated equivalent. Values in the "STK" and "FAUST" columns are CPU loads in percents. The "difference" column give the gain of efficiency in percents.

... for a wide range of audio architectures and platforms



# Structure of the Faust Compiler



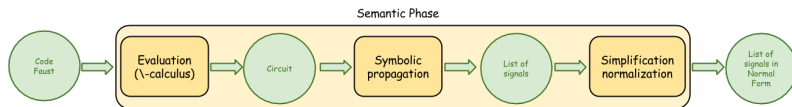
- *Semantic phase* : transforms the signal processor denoted by a Faust program into the list of signals expressions it computes.
- *Code generation* : generates the best possible implementation for this list of signal expressions



# Semantic Phase

The main strategy of the Faust compiler is the idea of *semantic compilation*:

- A Faust program is not compiled as such. What is compiled is its «*mathematical meaning*».
- Therefore two different Faust programs, but with the same semantic, should result in the same implementation.



1. The Faust program is translated into a «flat» circuit.'
2. Symbolic signals are propagated into the circuit.
3. Common sub-expressions are shared by hash-consing.
4. The resulting signals are simplified, normalized and type annotated.

# Semantic Phase

## Moog filter example

Why not compile a Faust program as such ? To increase expressivity, modularity, reusability and clarity...

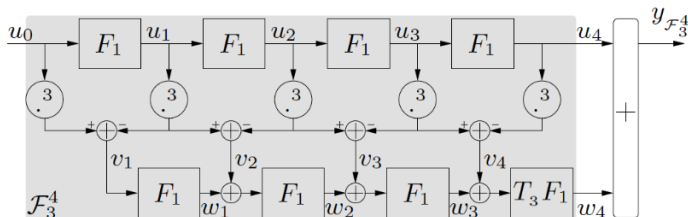
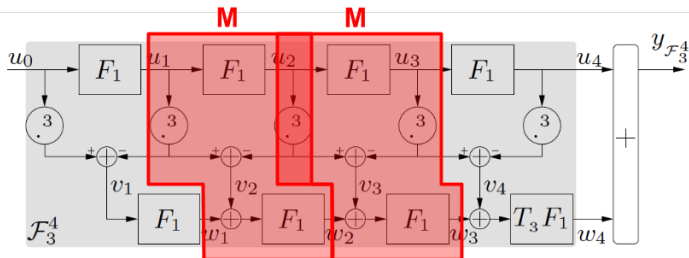
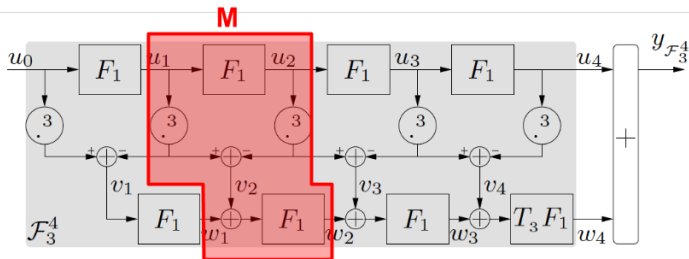


Figure: Digital model of a Moog analog filter (T. Helie/IRCAM)

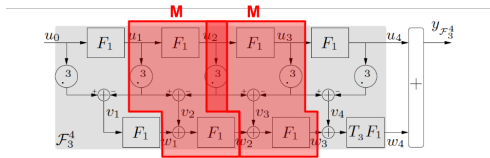
# Semantic Phase

## Moog filter example



# Semantic Phase

## Moog filter example



```
M = (_ <: _, (_^3, F1^3 : -)), _ : _, + : F1, F1;
process = _, 0 : M : M : M : S;
```

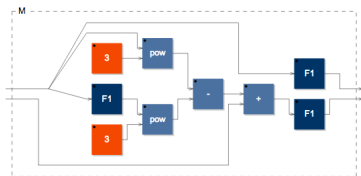


Figure: Module M

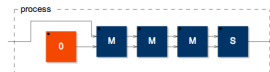
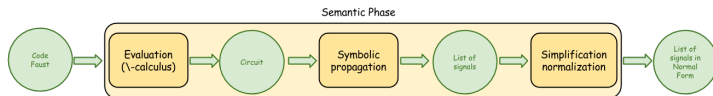


Figure: Moog filter

# Semantic Phase

## Evaluation to circuits


$$\begin{array}{lcl} \text{Circuit } C & ::= & P \quad \mid \quad C1 : C2 \quad \mid \quad C1 , C2 \\ & & C1 <: C2 \quad \mid \quad C1 :> C2 \quad \mid \quad C1 \sim C2 \end{array}$$

Where  $P \in \{*, +, -, \dots, \sin, \cos, e^x, \dots\}$  is a *primitive* operation on signals;  $C1 : C2$  the *sequential* composition of two circuits;  $C1, C2$  the *parallel* composition;  $C1 <: C2$  and  $C1 :> C2$  the *split* and *merge* compositions; and  $C1 \sim C2$  the *recursive* composition.

Set  $\mathbb{W}$  of well formed circuits:

$$P \in \mathbb{W}$$

$$(C1:C2) \in \mathbb{W} \Leftrightarrow C1 \in \mathbb{W}, C2 \in \mathbb{W}, O(C1) = I(C2)$$

$$(C1, C2) \in \mathbb{W} \Leftrightarrow C1 \in \mathbb{W}, C2 \in \mathbb{W}$$

$$(C1<:C2) \in \mathbb{W} \Leftrightarrow C1 \in \mathbb{W}, C2 \in \mathbb{W}, O(C1).k = I(C2)$$

$$(C1:>C2) \in \mathbb{W} \Leftrightarrow C1 \in \mathbb{W}, C2 \in \mathbb{W}, O(C1) = I(C2).k$$

$$(C1 \sim C2) \in \mathbb{W} \Leftrightarrow C1 \in \mathbb{W}, C2 \in \mathbb{W}, O(C1) \geq I(C2), I(C1) \geq O(C2)$$

A circuit  $C \in \mathbb{W}$  has  $I(C)$  *input signals* and  $O(C)$  *output signals*:

$$I(P) \in \mathbb{N}$$

$$O(P) \in \mathbb{N}$$

$$I(C1:C2) = I(C1)$$

$$O(C1:C2) = O(C2)$$

$$I(C1, C2) = I(C1) + I(C2)$$

$$O(C1, C2) = O(C1) + O(C2)$$

$$I(C1<:C2) = I(C1)$$

$$O(C1<:C2) = O(C2)$$

$$I(C1:>C2) = I(C1)$$

$$O(C1:>C2) = O(C2)$$

$$I(C1 \sim C2) = I(C1) - O(C2)$$

$$O(C1 \sim C2) = O(C1)$$

# Semantic Phase

## Evaluation to circuits

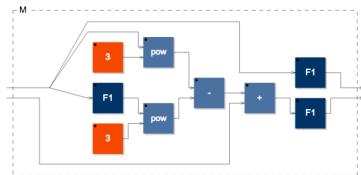


Figure: Module M

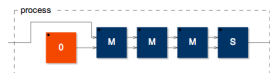


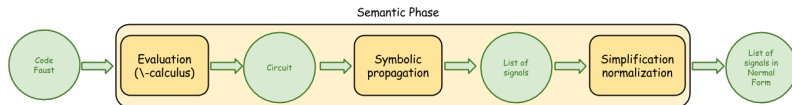
Figure: Moog filter



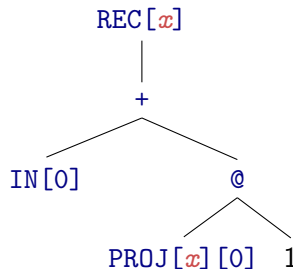
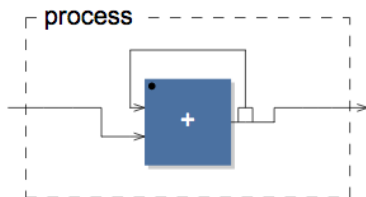
Figure: Flat Moog circuit after evaluation

# The Semantic Phase

## Symbolic propagation


$$\text{Signal } S ::= \begin{array}{l} P(S1..Sn) \\ \text{REC}[x](S1..Sn) \end{array} \quad \bigg| \quad \begin{array}{l} \text{IN}[i] \\ \text{PROJ}[x][i] \end{array} \quad \bigg| \quad n$$

process = +~\_;





# The Semantic Phase

## Hash-consing



The Faust compiler heavily relies on *hash-consing* to discover common sub-expressions, and on *memoization* to speedup compilation.

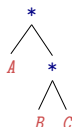
$$\begin{aligned} T_1 = T_2 &\Rightarrow \text{Addr}(T_1) = \text{Addr}(T_2) \\ \text{Addr}(T_1) \neq \text{Addr}(T_2) &\Rightarrow T_1 \neq T_2 \end{aligned}$$

Problem : Hash-consing can miss potential sharings. Simplification and normalization are used to improve sharing and common subexpression elimination.

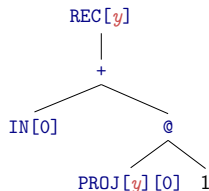
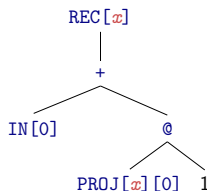
# The Semantic Phase

Hash-consing

Miss due to associativity or commutativity :



Miss due to  $\alpha$ -equivalence :



# The Semantic Phase

## Simplification and normalization



Polynomial expressions are rewritten as sum of products :  
 $k_1 A^m B^n \dots + k_2 C^i D^j \dots$  and then factorized.

$$0 * A \rightarrow 0$$

$$1 * A \rightarrow A$$

$$A * k \rightarrow k * A$$

$$(k * A) * (k' * B) \rightarrow (k * k') * (A * B)$$

$$B * A \rightarrow A * B$$

$$A^n * A^m \rightarrow A^{n+m}$$

$$0 + A \rightarrow A$$

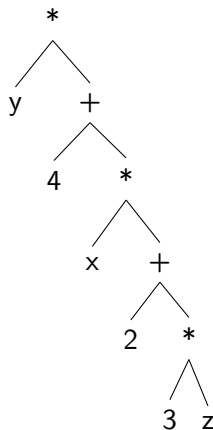
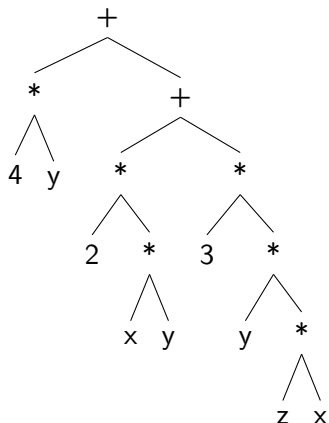
$$B + A \rightarrow A + B$$

$$A * B + A * C \rightarrow A * (B + C)$$

# The Semantic Phase

## Simplification and normalization

Example :  $4y + 2xy + 3xyz \rightarrow y(4 + x(2 + 3z))$



# The Semantic Phase

## Simplification and normalization



Reorganization of delay lines moved towards inputs :

$$0z^{-n} \rightarrow 0$$

$$Az^0 \rightarrow A$$

$$(k * A)z^{-n} \rightarrow k * Az^{-n}$$

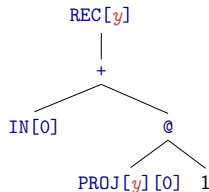
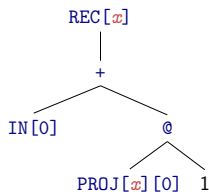
$$Az^{-n}z^{-m} \rightarrow Az^{-(n+m)}$$

# The Semantic Phase

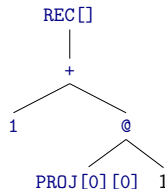
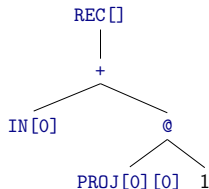
## Sharing of recursive terms



*Problem with hash-consing* :  $\alpha$ -equivalent recursive terms in standard notation are not shared:



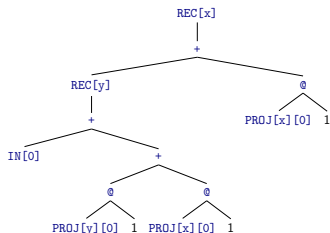
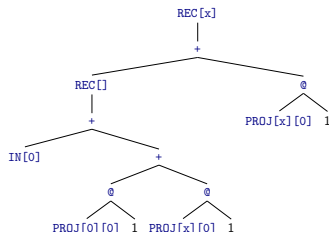
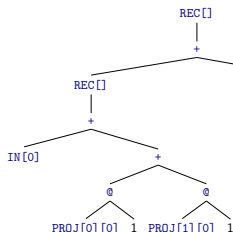
Open terms in de Bruijn notation are incorrectly shared:



# The Semantic Phase

## Sharing of recursive terms

*Solution* : start in de Bruijn notation and progressively transform de Bruijn terms into standard terms.



# The Semantic Phase

## Type Annotation and interval computation



Signals are annotated with various information used to guide the code generation :

- *nature*: integer or floating point values
- *boolean*: when a signal stands for a boolean value
- *variability*: how fast values change (constant, at each block or at each sample)
- *computability*: when values are available (at compile time, at initialization time, at execution time)
- *vectorability*: when a signal computation can be vectorized
- *interval*: minimal and maximal values a signal can take
- *occurrence context*: maximal delay, number of occurrences



# The Semantic Phase

Resulting signals for the Moog example



Figure: Flat circuit after evaluation of the Faust program

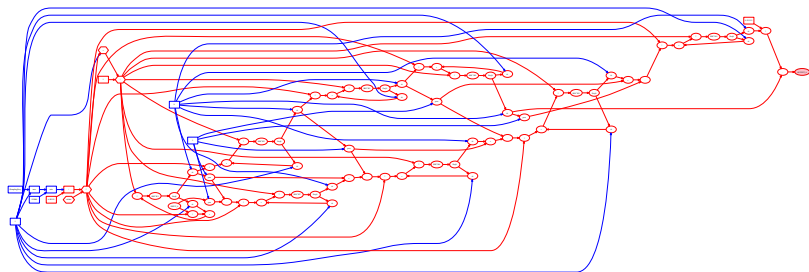
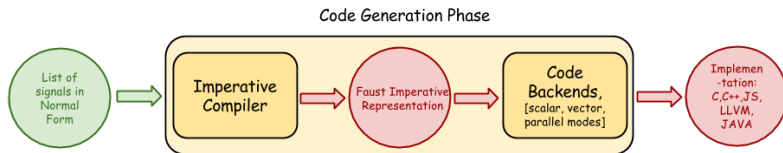


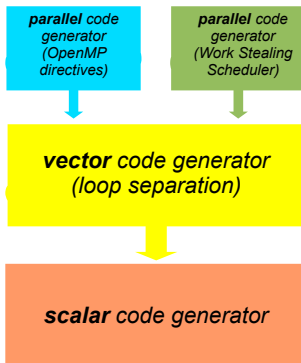
Figure: Resulting signal after symbolic propagation in the circuit and normalization

# Code Generation Phase



# Code Generation Phase

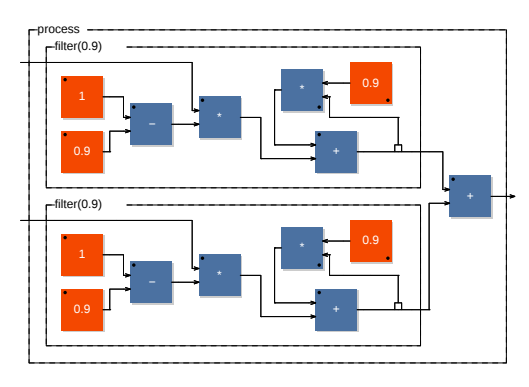
## Four Code generation modes



# Code Generation Phase

two 1-pole filters in parallel connected to an adder

```
filter(c) = *(1-c) : + ~ *(c);  
process = filter(0.9), filter(0.9) : +;
```



# Code Generation Phase

## Scalar Code

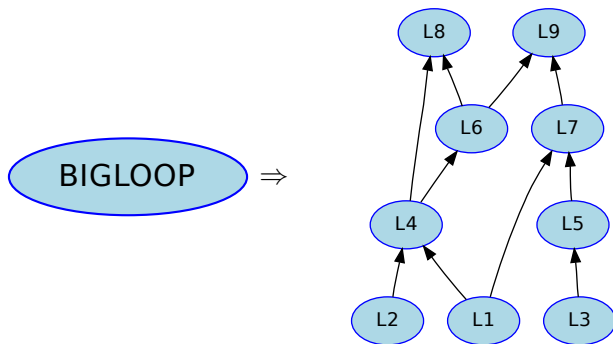


```
virtual void compute (int count, float** input, float** output) {  
    float* input0 = input[0];  
    float* input1 = input[1];  
    float* output0 = output[0];  
    for (int i=0; i<count; i++) {  
        fRec0[0] = (0.1f * input1[i]) + (0.9f * fRec0[1]);  
        fRec1[0] = (0.1f * input0[i]) + (0.9f * fRec1[1]);  
        output0[i] = (fRec1[0] + fRec0[0]);  
        // post processing  
        fRec1[1] = fRec1[0];  
        fRec0[1] = fRec0[0];  
    }  
}
```

# Code Generation Phase

## Loop Separation

The *Vector* Compilation Backend simplifies the autovectorization work of the C++ compiler by splitting the sample processing loop into several simpler loops.



# Code Generation Phase

## Vector Code



```
...  
// SECTION : 1  
for (int i=0; i<count; i++) {  
    fRec0[i] = (0.1f * input1[i]) + (0.9f * fRec0[i-1]);  
}  
for (int i=0; i<count; i++) {  
    fRec1[i] = (0.1f * input0[i]) + (0.9f * fRec1[i-1]);  
}  
// SECTION : 2  
for (int i=0; i<count; i++) {  
    output0[i] = fRec1[i] + fRec0[i];  
}  
...
```

# Code Generation Phase

## Parallel Code – OpenMP



```
...
// SECTION : 1
#pragma omp sections
{
    #pragma omp section
    for (int i=0; i<count; i++) {
        fRec0[i] = (0.1f * input1[i]) + (0.9f * fRec0[i-1]);
    }
    #pragma omp section
    for (int i=0; i<count; i++) {
        fRec1[i] = (0.1f * input0[i]) + (0.9f * fRec1[i-1]);
    }
}
// SECTION : 2
#pragma omp for
for (int i=0; i<count; i++) {
    output0[i] = (fRec1[i] + fRec0[i]);
}
...
```



# Code Generation Phase

## Parallel Code – Work Stealing



```
taskqueue.InitTaskList(task_list_size, task_list, fDynamicNumThreads,
                        cur_thread, tasknum);
while (!fIsFinished) {
    switch (tasknum) {
        case WORK_STEALING_INDEX: {
            tasknum = TaskQueue::GetNextTask(cur_thread, fDynamicNumThreads);
            break;
        }case LAST_TASK_INDEX: {
            fIsFinished = true;
            break;
        }case 2: {
            // LOOP 0x7fd873509e00
            ....
            fGraph.ActivateOneOutputTask(taskqueue,4,tasknum);
            break;
        }case 3: {
            // LOOP 0x7fd873703d70
            ....
            fGraph.ActivateOneOutputTask(taskqueue,4,tasknum);
            break;
        }case 4: {
            // LOOP 0x7fd873509d20
            ....
            tasknum = LAST_TASK_INDEX;
            break;
        }
    }
}
```

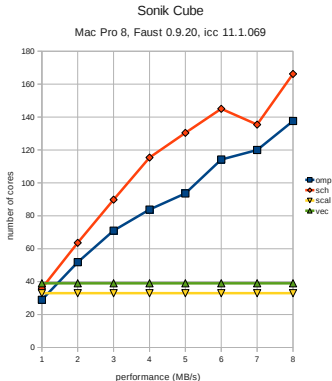
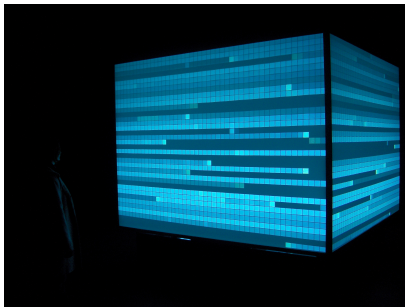
# Code Generation Phase

What improvements to expect from parallelized code ?



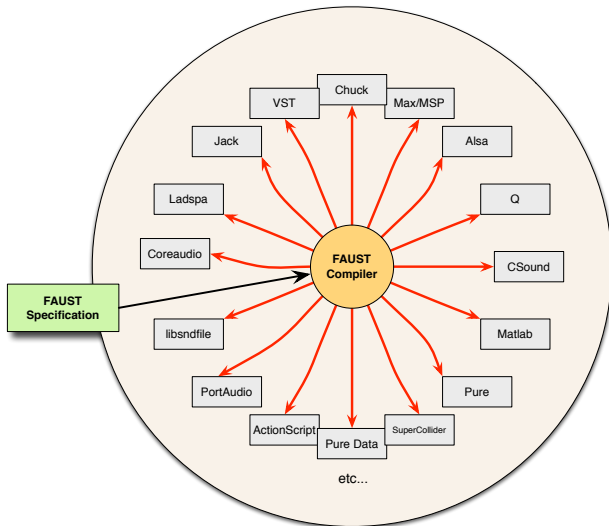
## Sonik Cube

Audio-visual installation involving a cube of light, reacting to sounds, immersed in an audio feedback room (Trafik/Orlarey 2006).



# Faust Architecture System

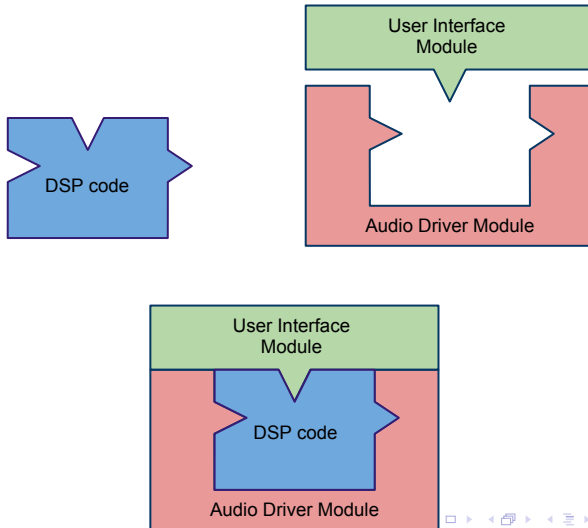
Easy deployment : one Faust code, multiple targets



# Faust Architecture System

## Separation of concern

The *architecture file* describes how to connect the audio computation to the external world.



# Faust Architecture System

## Examples of supported architectures



### ■ Audio plugins :

- ▶ AudioUnit
- ▶ LADSPA
- ▶ DSSI
- ▶ LV2
- ▶ Max/MSP
- ▶ VST
- ▶ PD
- ▶ CSound
- ▶ Supercollider
- ▶ Pure
- ▶ Chuck
- ▶ Octave
- ▶ Flash

### ■ Audio drivers :

- ▶ Jack
- ▶ Alsa
- ▶ CoreAudio
- ▶ Web Audio API

### ■ Graphic User Interfaces :

- ▶ QT
- ▶ GTK
- ▶ Android
- ▶ iOS
- ▶ HTML5/SVG

### ■ Other User Interfaces :

- ▶ OSC
- ▶ HTTPD

- Faust standalone compiler
- Faust online compiler (<http://faust.grame.fr>)
- LibFaust embedded compiler (FaustLive, MaxMSP, Csound, Antescofo, Open Music, iScore, Webaudio API) [ANR-INEDIT]
- Remote compiler service (<http://faustservice.grame.fr>) [ANR-FEEVER]
- Faustine (Faust Multirate interpreter) [ANR-FEEVER]
- Faust Multirate/Multidimension compiler [ANR-FEEVER]