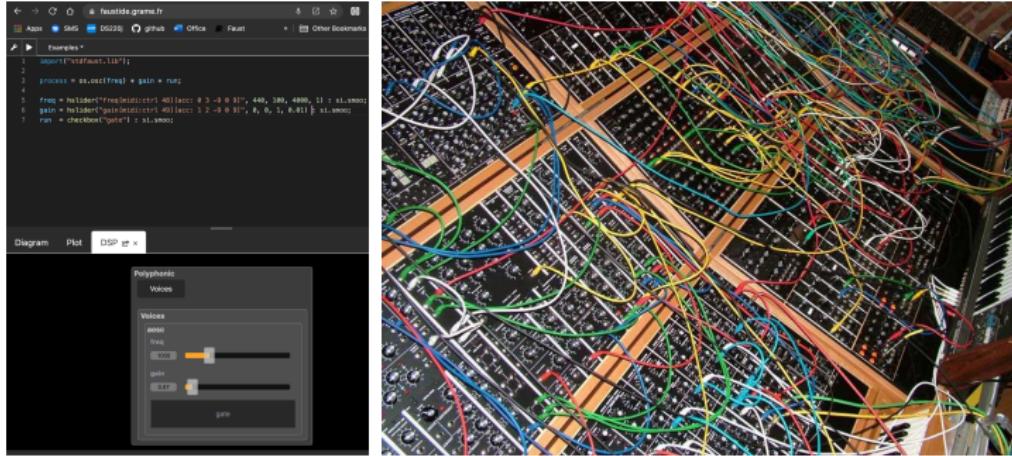




Functional  
Audio  
Stream

## The Faust Programming Language

# What is Faust (<https://faust.grame.fr>) ?



Faust is a programming language for signal processing, sound synthesis, electronic music instruments, etc.

# Making Realtime Audio Programming Accessible

- *"Faust is amazing! Without Faust, I could never have started making plugins. Community is fantastic!"*
- *"Faust is a really efficient and versatile language, with lots of tools and supported targets. The syntax makes the use of the language easier, also for people with no strong background in programming."*

The screenshot shows a news article from the French government's website. The URL in the address bar is <http://www.education.gouv.fr/pid/1502/2022>. The page title is "Remise des prix science ouverte du logiciel libre de la recherche". The text on the page discusses the awarding of the prize to various open-source software projects for their contributions to scientific research. The sidebar on the left includes links for "Sommaire", "The Coq proof assistant", "Faust", "Gammapy", and "Jury".

Faust awarded with the 2022  
Open Source Software for Open Science Research Prize

# Faust is Fully Compiled

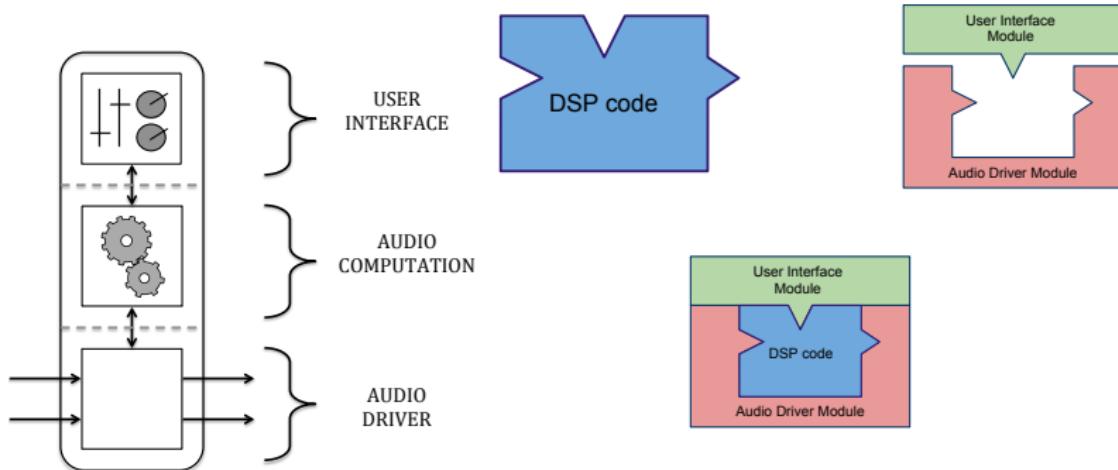
- Fully compiled to native code
- Sample level semantics
- Multiple backends: C++, WebAssembly, Rust, LLVM, etc.
- Code runs on most platforms: embedded systems, web pages, mobile devices, plug-ins, standalone applications, FPGAs, etc.



# Architectures and Deployment

## Separation of concern

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

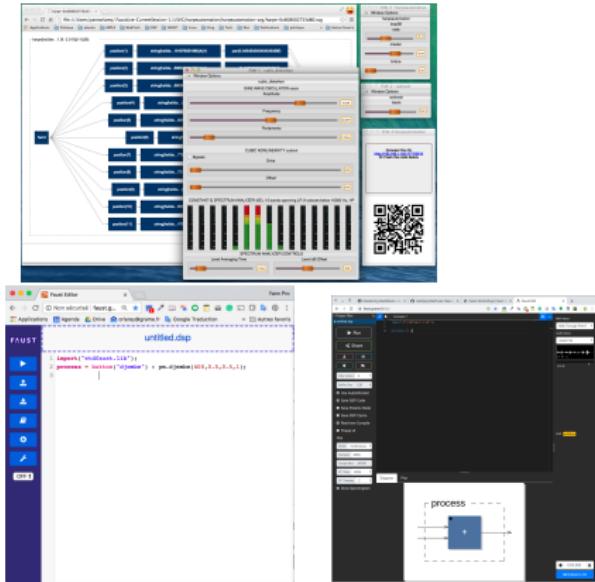
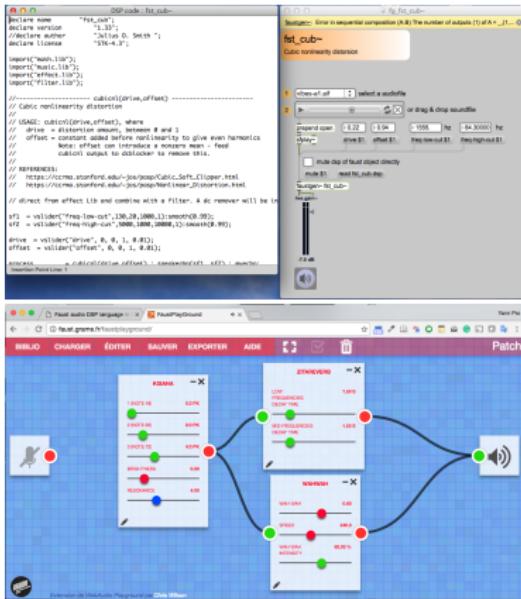


# The Faust Ecosystem

## Compilers

- Command Line Compilers
  - ▶ `faust` command line
  - ▶ `faust2xxx` command line
  - ▶ FaustWorks (IDE)
- Embedded Compilers (libfaust)
  - ▶ FaustLive (self contained)
  - ▶ Faustgen for Max/MSP
  - ▶ Faustcompile, etc. for Csound (V. Lazzarini)
  - ▶ Faust4processing
  - ▶ Antescofo (IRCAM's score follower)
- Web Based Compilers
  - ▶ Online documentation (<https://faustdoc.grame.fr>)
  - ▶ Faustplayground (<https://faustplayground.grame.fr>)
  - ▶ Online IDE (<https://faustide.grame.fr>)

# The Faust Ecosystem



# Quick Demo

Scan the QR code with your smartphone



- <https://tinyurl.com/45sxjt5m>
- <https://faustide.grame.fr/?autorun=1&code=https://raw.githubusercontent.com/orlarey/wahoo/master/examples/wahoo.dsp>

# Documentation, Examples, etc.

<https://faust.grame.fr>



```
FAUST Home Documentation Downloads Tools Community Projects About Q Search
1 mesh.square[1] = 202 zita
2   sl.bus(4) <- par(1,4,*(-1)) <- sl.bus(4)) :> sl.bus(4); 203 {(
3 // dequantizer with square wave
4   204 with
5 mesh.square[0] = sl.bus(4**N); 205
6   206 || 207
7   ->fsmc, fsmcheck(N/2)); 208
8 }
```

## What is Faust?

Faust (Functional Audio Script) is a functional programming language for sound synthesis and audio processing with a strong focus on the design of signal processing components, audio effects, etc. created at the [Graeme Charles Research Department](#). Faust targets high-performance signal processing applications and audio plug-ins for a variety of platforms and standards.

The core component of Faust is its compiler. It allows to "translate" any Faust digital signal processing (DSP) specification to a wide range of non-domain specific languages such as C++, C, LLVM IR code, WebAssembly, JavaScript, etc. In this regard, Faust can be seen as an alternative to C++ but to much simpler and intuitive to learn.

Thanks to a wrapping system called "architectures," codes generated by Faust can be easily compiled into a wide variety of objects ranging from audio plug-ins to standalone applications or smartphone and web apps, etc.



## Powered By Faust

This page lists the projects using Faust in different ways: musical pieces or artistic projects, plugins, standalone applications, integration in audio programming environments, development tools, research projects (possibly non-musical), embedded devices, Web applications, etc.

### 201 Musical Synthesizer



We're Citizen & Dulari and we're happy to announce our newest synthesizer: the 201! If you're familiar with our other modular instruments such as the Podust Para, Dipsolar, Bokta Bass, or Kafelabido, we hope you'll see that the 201 fits right in with them: the 201 is fun, portable, and packed with a medley of musical magic!

Inside, the 201 uses a combination of Pure Data and Faust musical programming environments. Users are invited to modify or create new sound engines!

### OneTrick SIMIAN



# Learn Faust

## International:

- Stanford U./CCRMA
- Maynooth U.
- Louisiana State U.
- Aalborg U.

## France:

- Jean Monnet U.
- IRCAM, ATIAM
- PARIS 8 U.

The screenshot shows a web browser displaying a course page on Kadenze. The title of the course is "Real-Time Audio Signal Processing in Faust". On the right side, there is a sidebar titled "WOULD YOU LIKE TO ENROLL?" with a large "ENROLL" button. The sidebar includes fields for "Length" (5 Sessions), "Price" (Audit (Free) Certificate (incl. w/ Premium)), "Institution" (Stanford University), "Subject" (Creative Computing), "Skill Level" (Expert), and "Topics" (Synthesis, Computer Programming, Digital Signal Processing (DSP), Field Effects). Below the sidebar, there are four small circular profile pictures of people.

<https://www.kadenze.com/courses/real-time-audio-signal-processing-in-faust/info>

# Purely Functional Approach

- Signals are functions:  $\mathbb{S} = \text{time} \rightarrow \text{sample}$ ,
- Faust primitives are signal processors:  $\mathbb{P} = \mathbb{S}^m \rightarrow \mathbb{S}^n$ ,
- Faust composition operations (`<:` `:` `:` `,` `~`) are binary functions on signal processors:  $\mathbb{A} = \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ ,
- User defined functions are higher order functions on signal processors:  $\mathbb{U} = \mathbb{P}^n \rightarrow \mathbb{P}$ ,
- A Faust program denotes a signal processor.

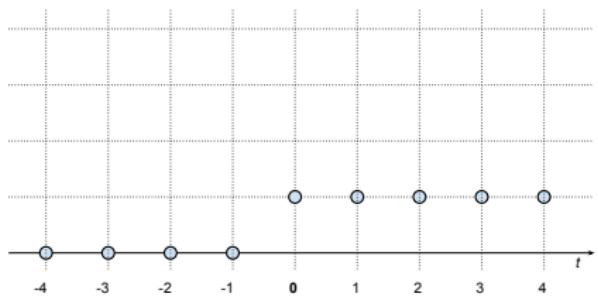
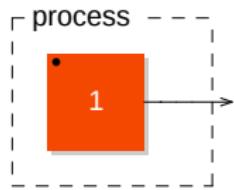
## What is Faust used for?

- Faust is used on stage for concerts and artistic productions, for education and research, for open sources projects and commercial applications :
- Faust offers end-users a high-level alternative to C to develop audio applications for a large variety of platforms.
- The role of the Faust compiler is to synthesize the most efficient implementations for the target language (C, C++, LLVM, Javascript, etc.).

# Signal generators

## Numbers

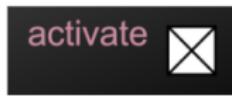
```
process = 1;
```



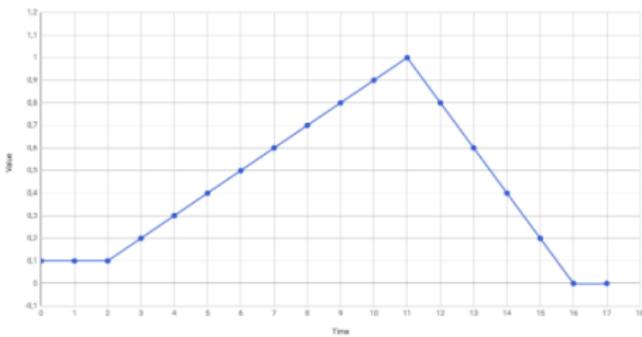
$$y(t) = \begin{cases} 1 & t \geq 0 \\ 0 & t < 0 \end{cases}$$

# Signal generators

## UI widgets



```
process = vslider("level", 0.1, 0, 1, 0.01);
```



# Faust Primitives

## Wire and Cut

Syntax	Type	Description
-	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	wire: $y(t) = x(t)$
!	$\mathbb{S}^1 \rightarrow \mathbb{S}^0$	cut: $\langle \rangle$ , $x(t)$ is ignored

# Faust Primitives

## Arithmetic operations

Syntax	Type	Description
+	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	addition: $y(t) = x_1(t) + x_2(t)$
-	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	subtraction: $y(t) = x_1(t) - x_2(t)$
*	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	multiplication: $y(t) = x_1(t) * x_2(t)$
$\wedge$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	power: $y(t) = x_1(t)^{x_2(t)}$
/	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	division: $y(t) = x_1(t)/x_2(t)$
%	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	modulo: $y(t) = x_1(t)\%x_2(t)$
int	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an int signal: $y(t) = (\text{int})x(t)$
float	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an float signal: $y(t) = (\text{float})x(t)$

# Faust Primitives

## Bitwise operations

Syntax	Type	Description
&	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical AND: $y(t) = x_1(t) \& x_2(t)$
	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical OR: $y(t) = x_1(t)   x_2(t)$
xor	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	logical XOR: $y(t) = x_1(t) \wedge x_2(t)$
<<	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arith. shift left: $y(t) = x_1(t) << x_2(t)$
>>	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arith. shift right: $y(t) = x_1(t) >> x_2(t)$

# Faust Primitives

## Comparison operations

Syntax	Type	Description
<	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less than: $y(t) = x_1(t) < x_2(t)$
$\leq$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less or equal: $y(t) = x_1(t) \leq x_2(t)$
>	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater than: $y(t) = x_1(t) > x_2(t)$
$\geq$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater or equal: $y(t) = x_1(t) \geq x_2(t)$
$\equiv$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	equal: $y(t) = x_1(t) \equiv x_2(t)$
$\neq$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	different: $y(t) = x_1(t) \neq x_2(t)$

# Faust Primitives

## Trigonometric functions

Syntax	Type	Description
acos	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc cosine: $y(t) = \text{acosf}(x(t))$
asin	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc sine: $y(t) = \text{asinf}(x(t))$
atan	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc tangent: $y(t) = \text{atanf}(x(t))$
atan2	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arc tangent of 2 signals: $y(t) = \text{atan2f}(x_1(t), x_2(t))$
cos	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cosine: $y(t) = \text{cosf}(x(t))$
sin	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	sine: $y(t) = \text{sinf}(x(t))$
tan	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	tangent: $y(t) = \text{tanf}(x(t))$

# Faust Primitives

## Other Math operations

Syntax	Type	Description
exp	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-e exponential: $y(t) = \text{expf}(x(t))$
log	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-e logarithm: $y(t) = \text{logf}(x(t))$
log10	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-10 logarithm: $y(t) = \text{log10f}(x(t))$
pow	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	power: $y(t) = \text{powf}(x_1(t), x_2(t))$
sqrt	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	square root: $y(t) = \text{sqrtf}(x(t))$
abs	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	absolute value (int): $y(t) = \text{abs}(x(t))$ absolute value (float): $y(t) = \text{fabsf}(x(t))$
min	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	minimum: $y(t) = \text{min}(x_1(t), x_2(t))$
max	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	maximum: $y(t) = \text{max}(x_1(t), x_2(t))$
fmod	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	float modulo: $y(t) = \text{fmodf}(x_1(t), x_2(t))$
remainder	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	float remainder: $y(t) = \text{remainderf}(x_1(t), x_2(t))$
floor	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	largest int $\leq$ : $y(t) = \text{floorf}(x(t))$
ceil	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	smallest int $\geq$ : $y(t) = \text{ceilf}(x(t))$
rint	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	closest int: $y(t) = \text{rintf}(x(t))$

# Faust Primitives

## Delays and Tables

Syntax	Type	Description
mem	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	1-sample delay: $y(t + 1) = x(t), y(0) = 0$
@	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	delay: $y(t + x_2(t)) = x_1(t), y(t < x_2(t)) = 0$
rdtable	$\mathbb{S}^3 \rightarrow \mathbb{S}^1$	read-only table: $y(t) = T[r(t)]$
rwtable	$\mathbb{S}^5 \rightarrow \mathbb{S}^1$	read-write table: $T[w(t)] = c(t); y(t) = T[r(t)]$
select2	$\mathbb{S}^3 \rightarrow \mathbb{S}^1$	between 2 sig: $T[] = \{x_0(t), x_1(t)\}; y(t) = T[s(t)]$
select3	$\mathbb{S}^4 \rightarrow \mathbb{S}^1$	between 3 sig: $T[] = \{x_0(t), x_1(t), x_2(t)\}; y(t) = T[s(t)]$

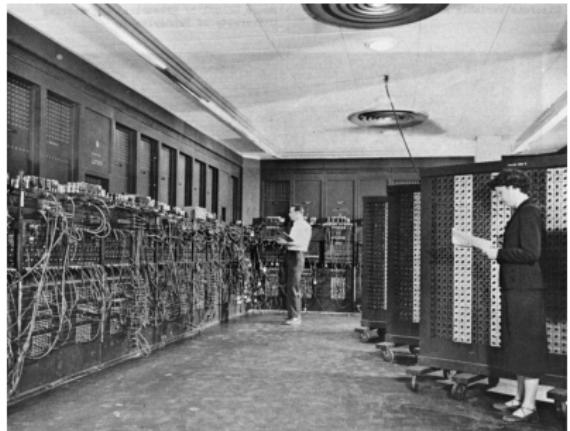
# Faust Primitives

## Other User Interface Primitives

Syntax	Example
<code>vgroup(str, block-diagram)</code>	<code>vgroup("reverb", ...)</code>
<code>hgroup(str, block-diagram)</code>	<code>hgroup("mixer", ...)</code>
<code>tgroup(str, block-diagram)</code>	<code>vgroup("parametric", ...)</code>
<code>vbargraph(str, min, max)</code>	<code>vbargraph("input", 0, 100)</code>
<code>hbargraph(str, min, max)</code>	<code>hbargraph("signal", 0, 1.0)</code>

# Block-Diagram Algebra

Programming by patching is familiar to musicians :



# Block-Diagram Algebra

Today programming by patching is widely used in Visual Programming Languages like Max/MSP:

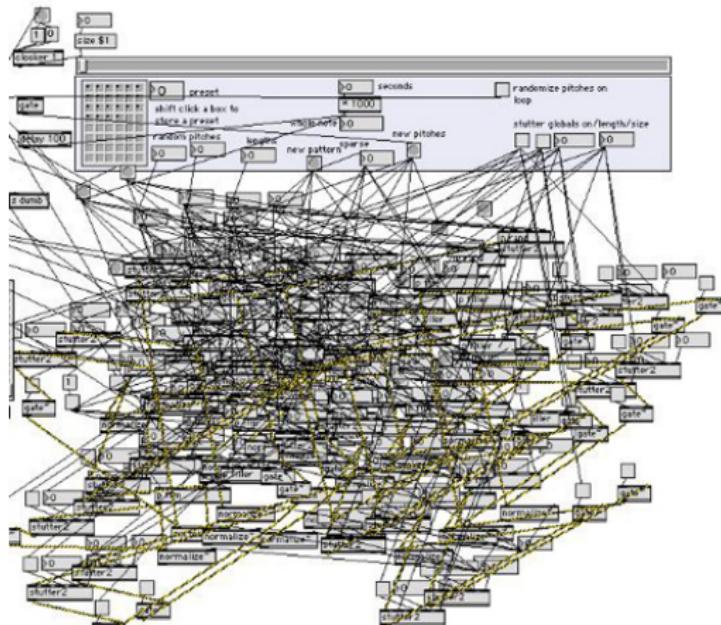


Figure: Block-diagrams can be a mess

# Block-Diagram Algebra

Faust allows structured block-diagrams

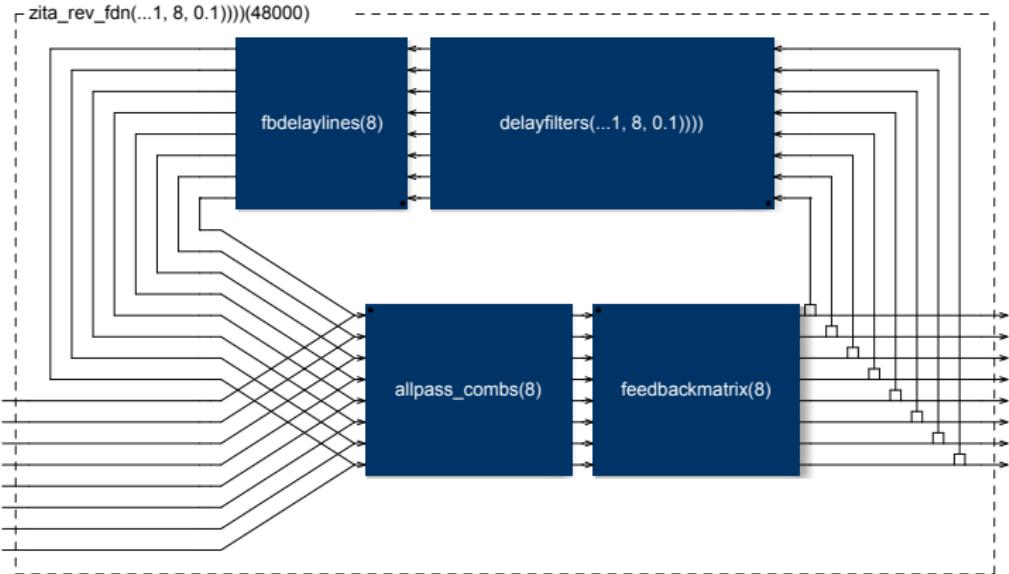


Figure: A complex but structured block-diagram

# Block-Diagram Algebra

Faust syntax is based on a *block diagram algebra*

## 5 Composition Operators

- $(A \sim B)$  recursive composition (priority 4)
- $(A, B)$  parallel composition (priority 3)
- $(A : B)$  sequential composition (priority 2)
- $(A <: B)$  split composition (priority 1)
- $(A :> B)$  merge composition (priority 1)

## 2 Constants

- $!$  cut
- $_$  wire

# Block-Diagram Algebra

## Parallel Composition

The *parallel composition* ( $A, B$ ) is probably the simplest one. It places the two block-diagrams one on top of the other, without connections.

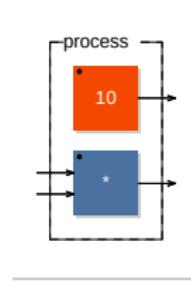


Figure: Example of parallel composition  $(10, *)$

# Block-Diagram Algebra

## Sequential Composition

The *sequential composition* ( $A : B$ ) connects the outputs of  $A$  to the inputs of  $B$ .  $A[0]$  is connected to  $[0]B$ ,  $A[1]$  is connected to  $[1]B$ , and so on.

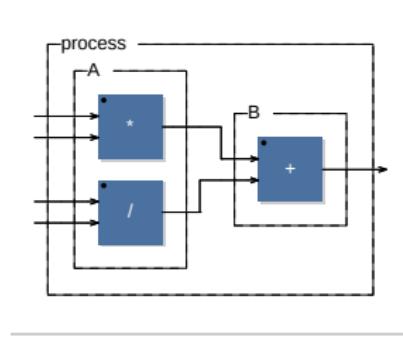


Figure: Example of sequential composition  $((*,/):+)$

Note that the number of outputs of  $A$  must be equal to the number of inputs of  $B$ .

# Block-Diagram Algebra

## Split Composition

The *split composition* ( $A <: B$ ) operator is used to distribute  $A$  outputs to  $B$  inputs.

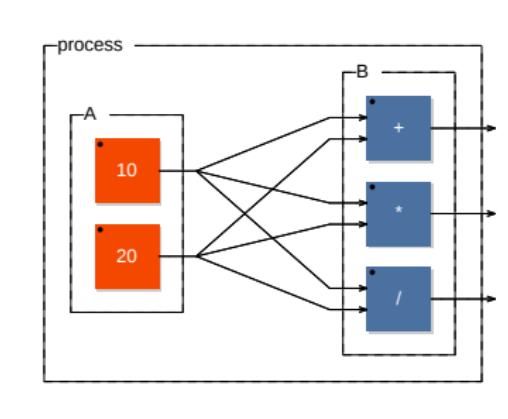


Figure: example of split composition  $((10,20) <: (+,*,/))$

# Block-Diagram Algebra

## Merge Composition

The *merge composition* ( $A :> B$ ) is used to connect several outputs of  $A$  to the same inputs of  $B$ .

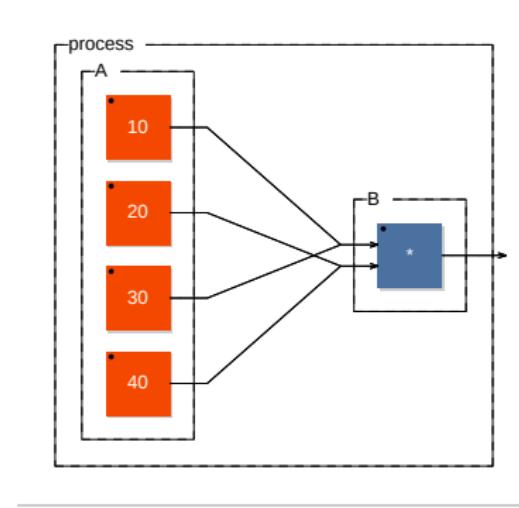


Figure: example of merge composition  $((10, 20, 30, 40) :> *)$

# Block-Diagram Algebra

## Recursive Composition

The *recursive composition* ( $A^\sim B$ ) is used to create cycles in the block-diagram in order to express recursive computations.

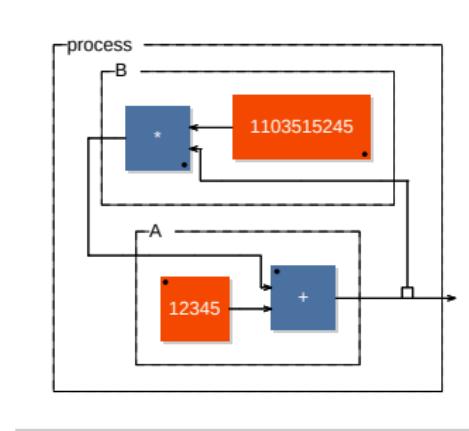


Figure: example of recursive composition  $+(12345) \sim *(1103515245)$

# Language Expressivity

- Function Composition
- Anonymity
- Faust programs as components
- Partial application
- Lexical environments as first class citizen
- Pattern Matching

# Language Expressiveness

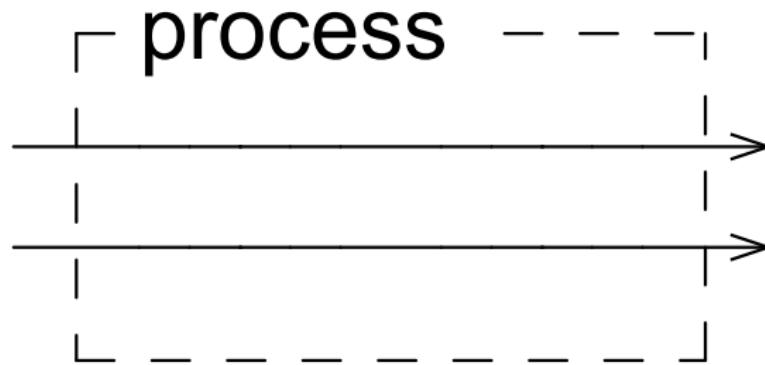
## Fast Fourier Transform

```
fft(N) = si.cbus(N) : an.c_bit_reverse_shuffle(N) : fftb(N)
with {
    fftb(1) = _,_;
    fftb(N) = si.cbus(N)
        : (fftb(No2)<:(si.cbus(No2), si.cbus(No2))), 
        (fftb(No2) <: (si.cbus(N):twiddleOdd(N)))
        :> si.cbus(N)
    with {
        No2 = int(N)>>1;
        twiddleOdd(N) = par(k,N,si.cmul(cos(w(k)),0-sin(w(k))));
        w(k) = 2.0*ma.PI*float(k)/float(N);
    };
};
```

# Language Expressiveness

## Fast Fourier Transform

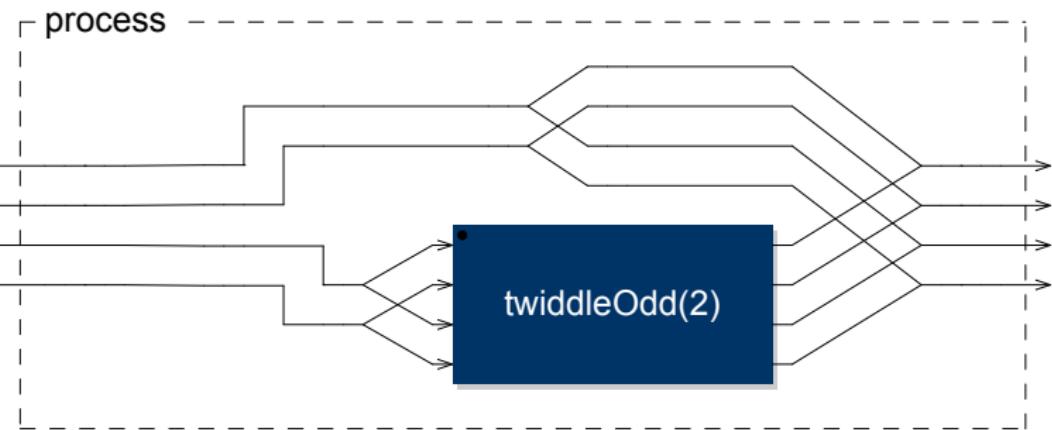
fft(1)



# Language Expressiveness

## Fast Fourier Transform

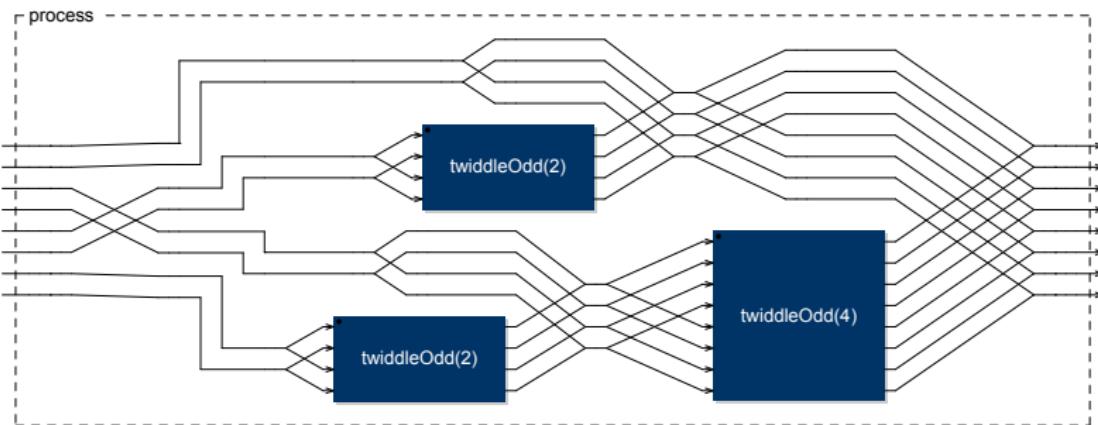
fft(2)



# Language Expressiveness

## Fast Fourier Transform

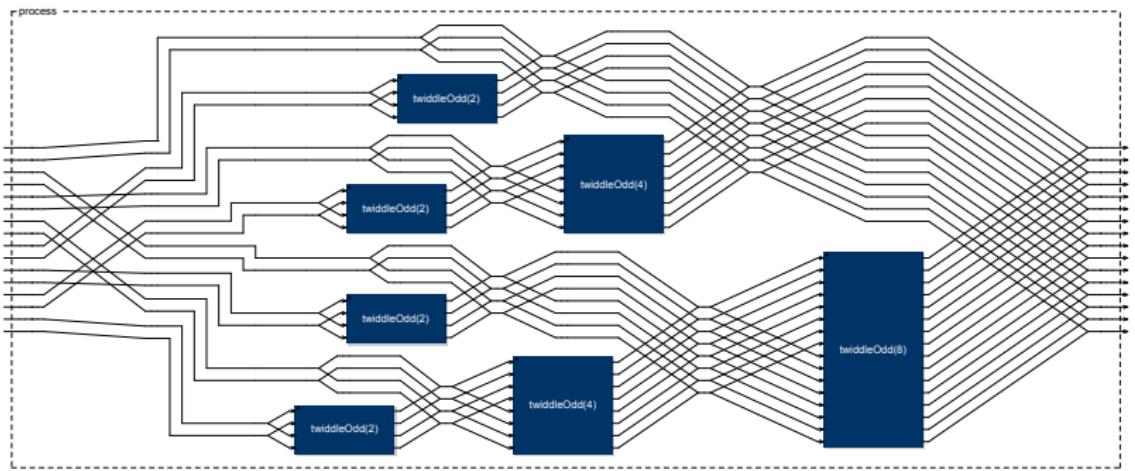
fft(4)



# Language Expressiveness

## Fast Fourier Transform

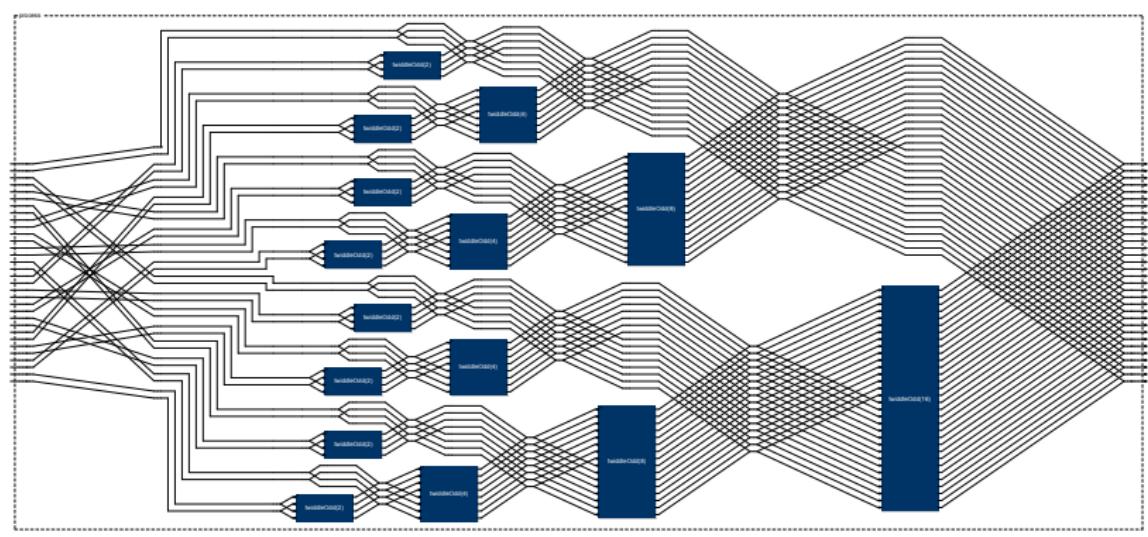
fft(8)



# Language Expressiveness

## Fast Fourier Transform

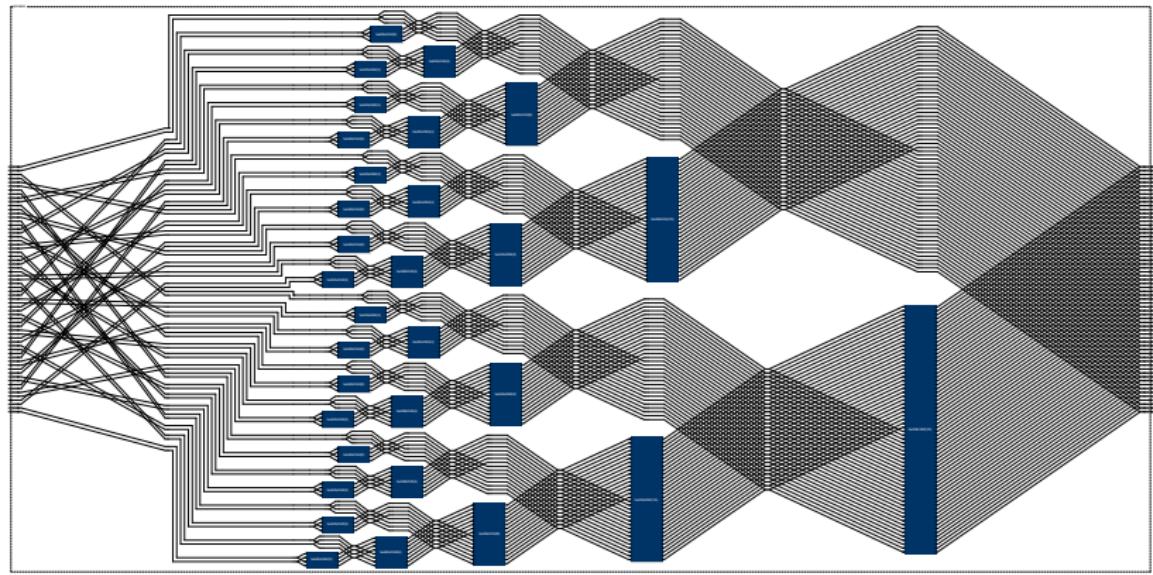
fft(16)



# Language Expressiveness

## Fast Fourier Transform

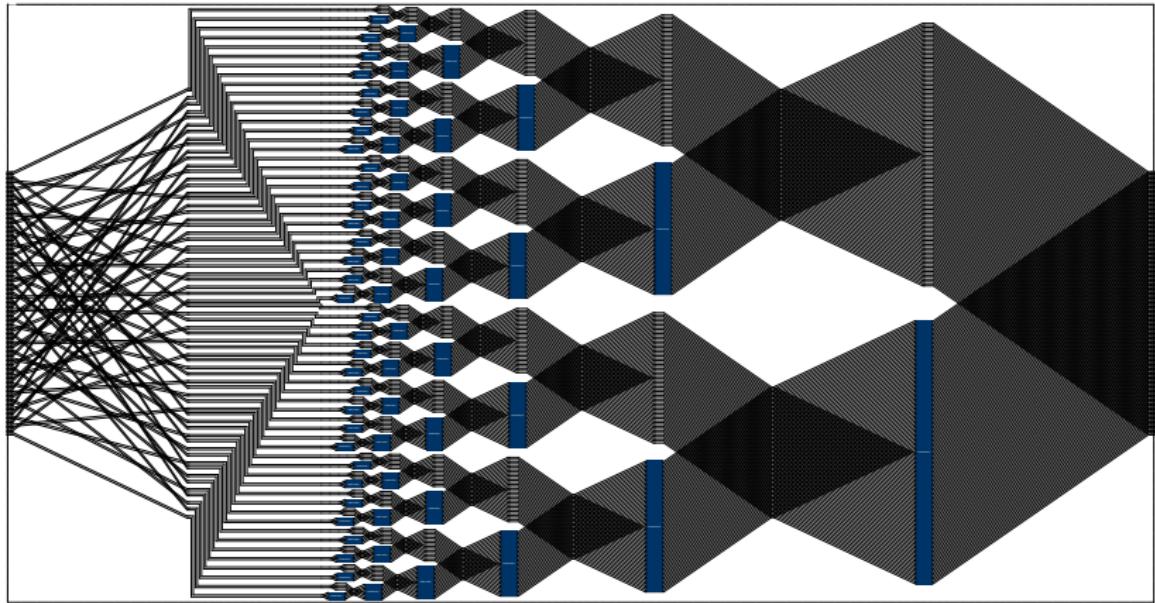
fft(32)



# Language Expressiveness

## Fast Fourier Transform

fft(64)



# Performance Quest

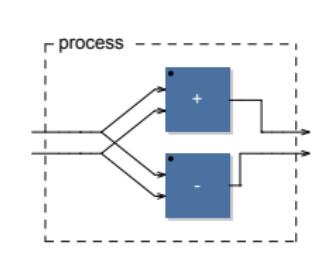
- Fully compiled to native code
- Sample level semantics
- Specification language
- Automatic parallelization

# Fully compiled to native code

Faust code:

```
process = _,- <: +,-;
```

Block-diagram:



C++ translation:

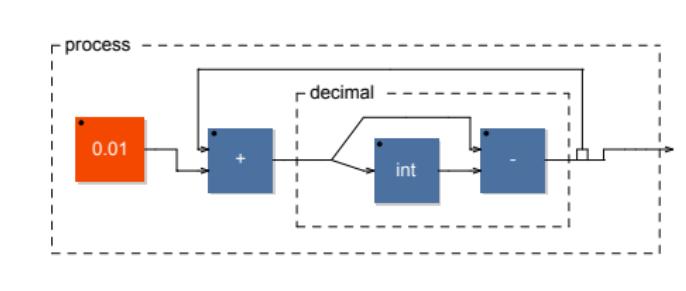
```
for (int i = 0; (i < count); i = (i + 1)) {  
    float fTemp0 = input0[i];  
    float fTemp1 = input1[i];  
    output0[i] = fTemp0 + fTemp1;  
    output1[i] = fTemp0 - fTemp1;  
}
```

# Sample level semantics

Sawtooth signal by step of 0.01:

```
decimal = _ <: _, int : -;  
process = 0.01 : (+:decimal) ~ _;
```

Block-diagram:



Signal equation:

$$y(t < 0) = 0$$

$$y(t \geq 0) = decimal(y(t-1) + 0.01)$$

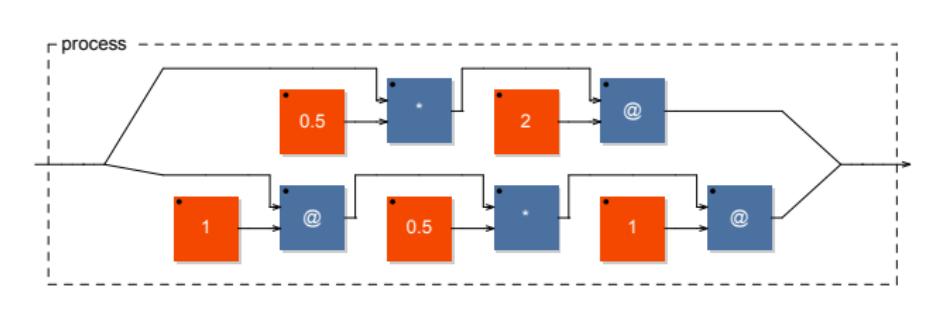
# Specification Language

Leave the implementation to the compiler

User's code:

```
process = _<:(*(0.5):@(2)),(@(1):*(0.5):@(1)):>_;
```

Block-diagram:

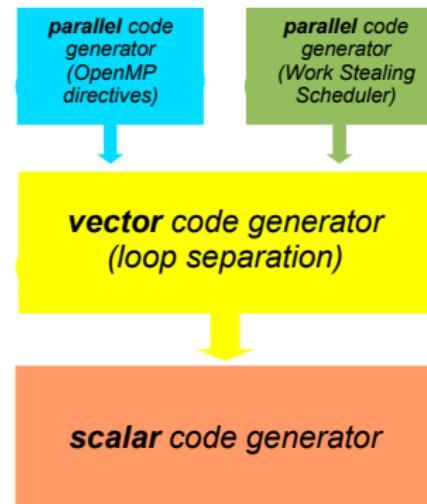


Equivalent, more efficient code

```
process = @(2);
```

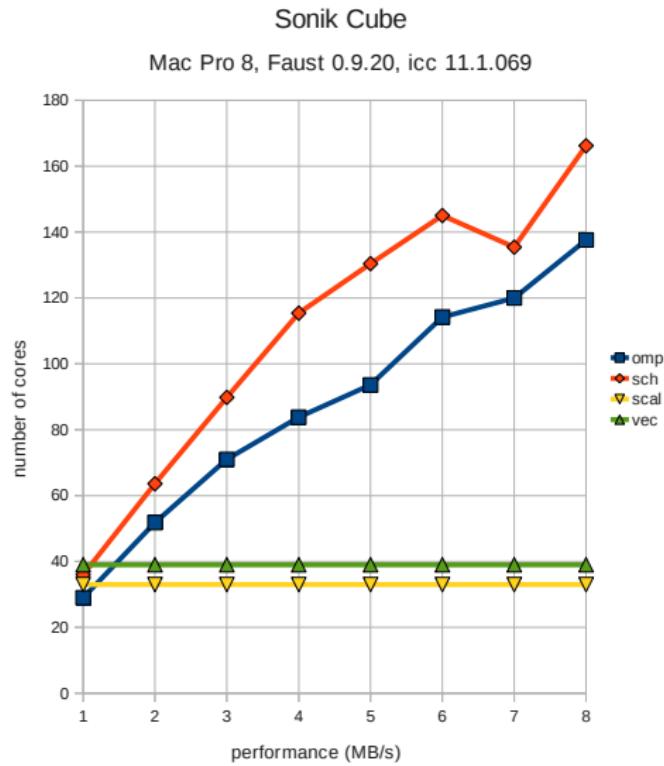
# Automatic Parallelization

## Code Generators



# Automatic Parallelization

## Performances



# Multiple Languages Backends

- **C++**
- C
- Rust
- Java
- Javascript
- **LLVM**
- **WebAssembly**
- **SOUL**
- ...

# Multiple Languages Backends

## Echo in Faust

```
process = + - (@(4410)*(0.8));
```

# Multiple Languages Backends

C++: faust -lang cpp

```
pub fn compute(&mut self, count: i32, inputs: &[f32], outputs: &mut[&mut[f32]]) {
    for i in 0..count {
        self.fRec0[(self.IOTA & 8191) as usize] = ((inputs[0][i as usize] as f32) + (0.00000012 * self.fRec0[((self.IOTA - 4411) & 8191) as usize]));
        outputs[0][i as usize] = (self.fRec0[((self.IOTA - 0) & 8191) as usize] as f32);
        self.IOTA = (self.IOTA + 1);
    }
}
```

# Multiple Languages Backends

## Web Assembly code: faust -lang wast

```
(func $compute (param $dsp i32) (param $count i32) (param $inputs i32) (param $outputs i32)
  (local $input0 i32)
  (local $output0 i32)
  (local $i i32)
  (local.set $input0 (i32.const 0))
  (local.set $output0 (i32.const 0))
  (local.set $i (i32.const 0))
  (local.set $input0 (i32.load (i32.add (local.get $inputs) (i32.const 0))))
  (local.set $output0 (i32.load (i32.add (local.get $outputs) (i32.const 0))))
  (local.set $i (i32.const 0))
  (loop $for-in-i
    (block $for-out-i
      {f32.store (i32.add (i32.const 4) (i32.shl (i32.and (i32.load (i32.const 0)) (i32.const 8191)) (i32.const 2))) (f32.add (f32.load (i32.add
        (local.get $input0) (local.get $i))) (f32.mul (f32.const 0.000000012) (f32.load (i32.add (i32.const 4) (i32.shl (i32.and (i32.sub (i32.load
          (i32.const 0)) (i32.const 4411)) (i32.const 8191)) (i32.const 2)))))))
      {f32.store (i32.add (local.get $output0) (local.get $i)) (f32.load (i32.add (i32.const 4) (i32.shl (i32.and (i32.sub (i32.load (i32.const 0))
        (i32.const 0)) (i32.const 8191)) (i32.const 2))))}
      {i32.store (i32.const 0) (i32.add (i32.load (i32.const 0)) (i32.const 1)))
      (local.set $i (i32.add (local.get $i) (i32.const 4)))
      {if (i32.lt_s (local.get $i) (i32.const 4)) (local.get $count)) (br $for-in-i) (br $for-out-i)}
    )
  )
}
```

# Multiple Languages Backends

Rust code: faust -lang rust

```
pub fn compute(&mut self, count: i32, inputs: &[f32], outputs: &mut [f32]) {
    for i in 0..count {
        self.fRec0[(self.IOTA & 8191) as usize] = ((inputs[0][i as usize] as f32) + (0.800000012 * self.fRec0[((self.IOTA - 4411) & 8191) as usize]));
        outputs[0][i as usize] = (self.fRec0[((self.IOTA - 0) & 8191) as usize] as f32);
        self.IOTA = (self.IOTA + 1);
    }
}
```

# Multiple Languages Backends

## SOUL code: faust -lang soul

```
void run()
{
    // DSP loop running forever...
    loop
    {
        // Computes one sample
        fRec0.at ((int (IOTA) & int (8191))) = (float32 (input0) + (0.800000012f * fRec0.at ((int ((IOTA - 4411)) & int (8191)))));
        output0 << float32 (fRec0.at ((int ((IOTA - 0)) & int (8191))));
        IOTA = (IOTA + 1);

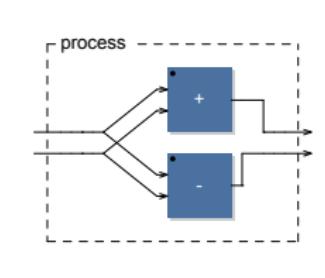
        // Moves all streams forward by one 'tick'
        advance();
    }
}
```

# Fully compiled to native code

Faust code:

```
process = _,_ <: +,-;
```

Block-diagram:



C++ translation:

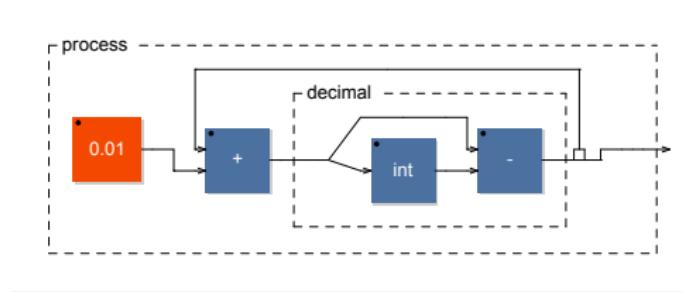
```
for (int i = 0; (i < count); i = (i + 1))  
    float fTemp0 = input0[i];  
    float fTemp1 = input1[i];  
    output0[i] = fTemp0 + fTemp1;  
    output1[i] = fTemp0 - fTemp1;  
}
```

# Sample level semantics

Sawtooth signal by step of 0.01:

```
decimal = _ <: _, int : -;  
process = 0.01 : (+:decimal) ~ _;
```

Block-diagram:



Signal equation:

$$y(t < 0) = 0$$

$$y(t \geq 0) = decimal(y(t-1) + 0.01)$$

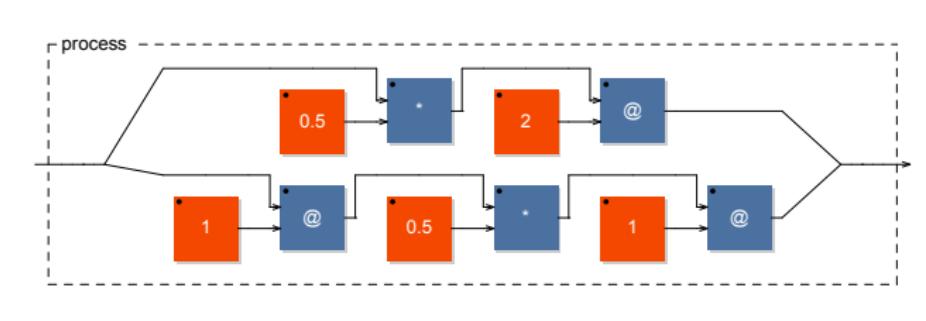
# Specification Language

Leave the implementation to the compiler

User's code:

```
process = _<:(*(0.5):@(2)),(@(1):*(0.5):@(1)):>_;
```

Block-diagram:

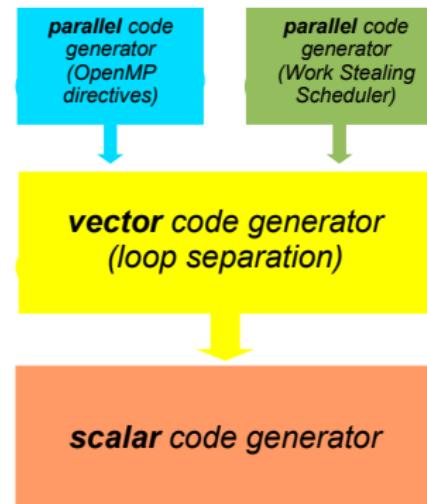


Equivalent, more efficient code

```
process = @(2);
```

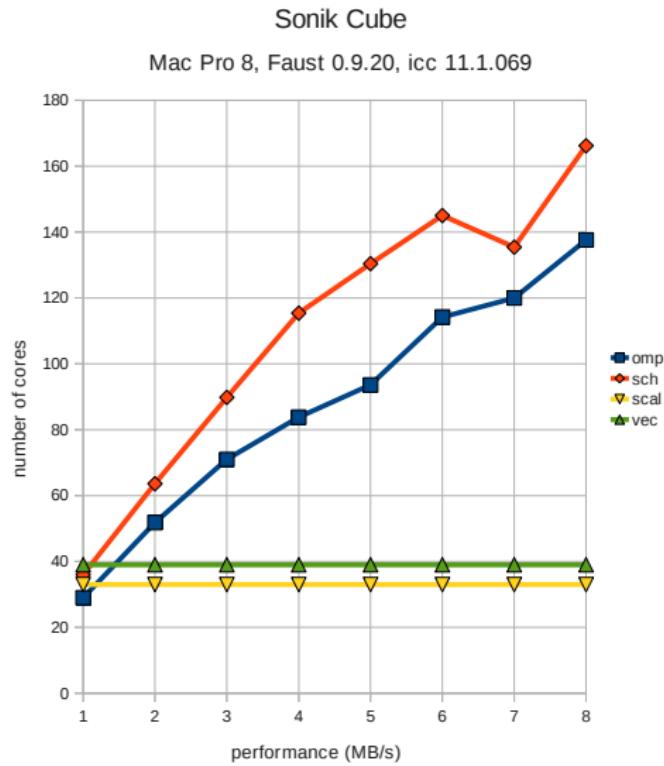
# Automatic Parallelization

## Code Generators

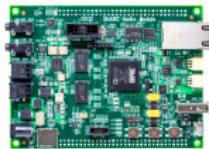


# Automatic Parallelization

## Performances



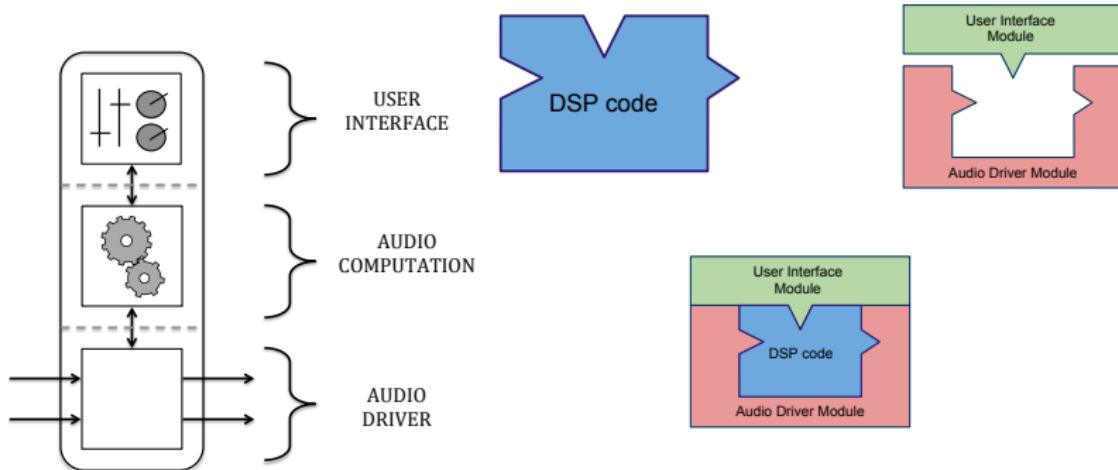
# Easy Deployment



# Easy Deployment

## Separation of concern

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



# Easy Deployment

## Examples of supported architectures

- Audio plugins :

- ▶ AudioUnit
- ▶ LADSPA
- ▶ DSSI
- ▶ LV2
- ▶ Max/MSP
- ▶ VST
- ▶ PD
- ▶ Csound
- ▶ Supercollider
- ▶ Pure
- ▶ Chuck
- ▶ JUCE
- ▶ Unity

- Devices :

- ▶ OWL
- ▶ MOD
- ▶ BELA
- ▶ SAM

- Audio drivers :

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

- Graphic User Interfaces :

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

- Other User Interfaces :

- ▶ MIDI
- ▶ OSC
- ▶ HTTPD

# Embeddable Compilers

## Libfaust

- Libfaust: embeddable version of the Faust compiler coupled with LLVM
- Libfaust.js: embeddable Javascript version of the Faust compiler

# Embedded Faust Compilers

Based on libfaust

- FaustLive (self contained)
- Faustgen for Max/MSP
- Faustgen for PD
- Faustcompile, etc. for Csound (V. Lazzarini)
- Faust4processing
- Antescofo (IRCAM)

# Embedded Faust Compilers



This screenshot shows a graphical interface for a Faust compiler. On the left, a window titled "Freeverb" displays the DSP code for the Freeverb algorithm. The code is a C-like script with declarations for names, version, author, license, copyright, and reference. It includes comments indicating it's a faster version using fixed delays (28k gain). On the right, a window titled "Freeverb Faust" shows a block diagram corresponding to the code. The diagram includes nodes for "Source Sound File" (set to "drum.loop.aif"), "Delay \$1", "RoomSize \$1", "Spread \$1", "Biquad (freeverb)", and "Biquad (freeverb)". Below the diagram is a "Live Gain" slider and a QR code.

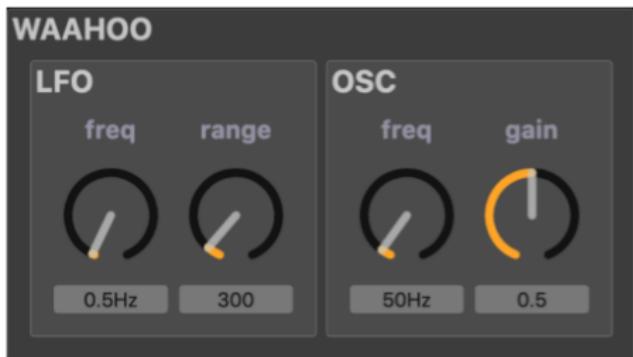
```
declare name      "Freeverb";
declare version   "1.0";
declare author    "Gromé";
declare license   "BSD";
declare copyright "CC-MN, Stanford University, 2006";
declare reference "https://cc-mn.stanford.edu/~jos/";

// Freeverb
// Faster version using fixed delays (28k gain)

// Constant Parameters
fixedgain = 0.815;
scallowet = 1.8;
scaldry = 2.0;
scaldamp = 0.4;
scalq = 0.74;
offsetres = 0.7;
initialroom = 0.5;
wetdryAmax = 0.8;
```

# Quick Demo

Scan the QR code with your smartphone



- <https://tinyurl.com/45sxjt5m>
- <https://faustide.grame.fr/?autorun=1&code=https://raw.githubusercontent.com/orlarey/wahoo/master/examples/wahoo.dsp>

# Compiling Everywhere

## Language Backends

- C++
- C
- Rust
- Java
- Javascript
- Asm.js
- LLVM
- WebAssembly
- ...

# Compiling Everywhere

## Libfaust

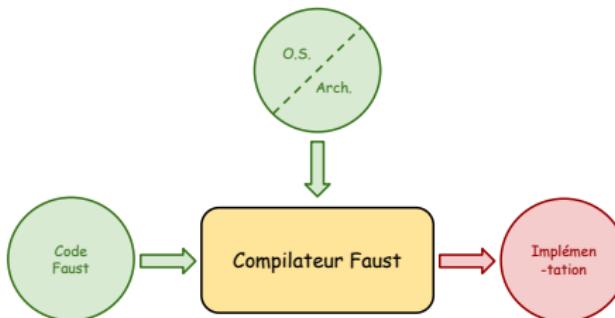
- Libfaust: embeddable version of the Faust compiler coupled with LLVM
- Libfaust.js: embeddable Javascript version of the Faust compiler

# Compiling Everywhere

- Command Line Compilers
  - ▶ `faust` command line
  - ▶ `faust2xxx` command line
  - ▶ FaustWorks (IDE)
- Embedded Compilers (libfaust)
  - ▶ FaustLive (self contained)
  - ▶ Faustgen for Max/MSP
  - ▶ Faust for PD
  - ▶ Faustcompile, etc. for Csound (V. Lazzarini)
  - ▶ Faust4processing
  - ▶ Antescofo (IRCAM's score follower)
- Web Based Compilers
  - ▶ Faustweb API (<https://faustservice.grame.fr>)
  - ▶ Online Development Environment  
(<https://faustide.grame.fr/>)
  - ▶ Online Editor (<https://fausteditor.grame.fr/>)
  - ▶ Faustplayground (<https://faustplayground.grame.fr/>)

# Role of the Faust Compiler

Generate efficient implementations . . .

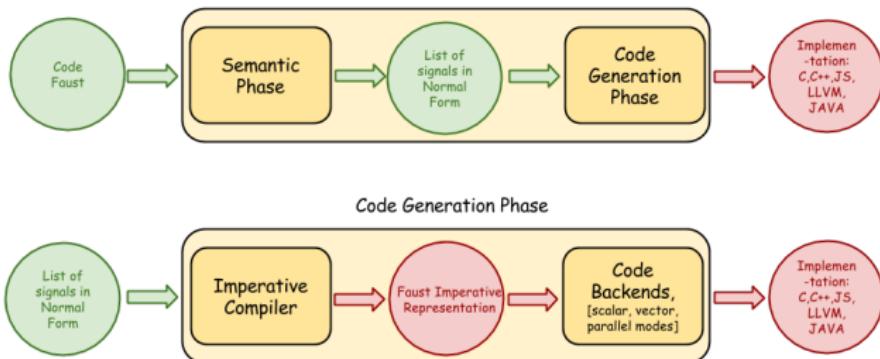


FAUST file name	STK	FAUST	Difference
blowBottle.dsp	3.23	2.49	22.91
blowHole.dsp	2.70	1.75	35.19
bowed.dsp	2.78	2.28	17.99
brass.dsp	10.15	2.01	80.20
clarinet.dsp	2.26	1.19	47.35
flutestk.dsp	2.16	1.13	47.69
saxophony.dsp	2.38	1.47	38.24
sitar.dsp	1.59	1.11	30.19
tibetanBowl.dsp	5.74	2.87	50

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.

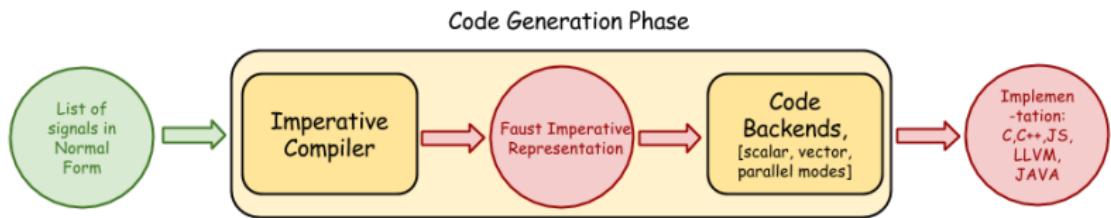
. . . for a wide range of audio architectures and platforms

# Structure of the Faust Compiler



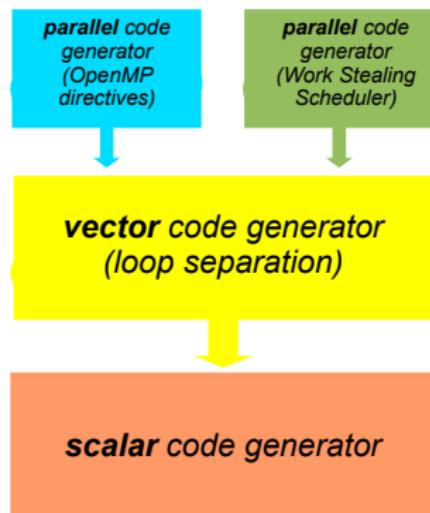
- *Semantic phase*: transforms the signal processor denoted by a Faust program into the list of signals expressions it computes.
- *Code generation*: generates the best possible implementation for this list of signal expressions

# Code Generation Phase



# Code Generation Phase

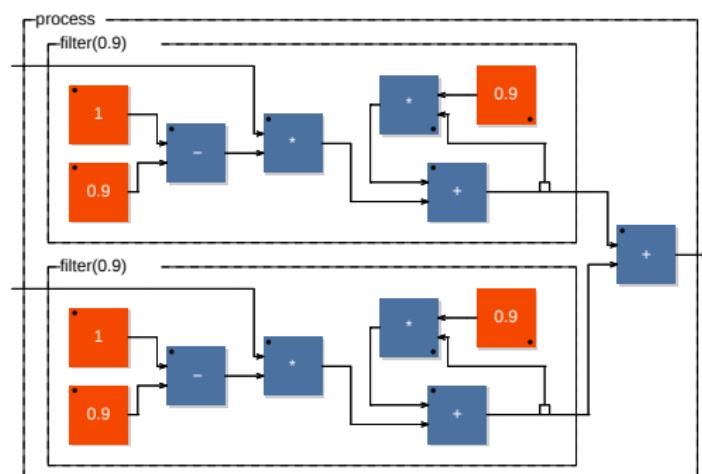
Four Code generation modes



# 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

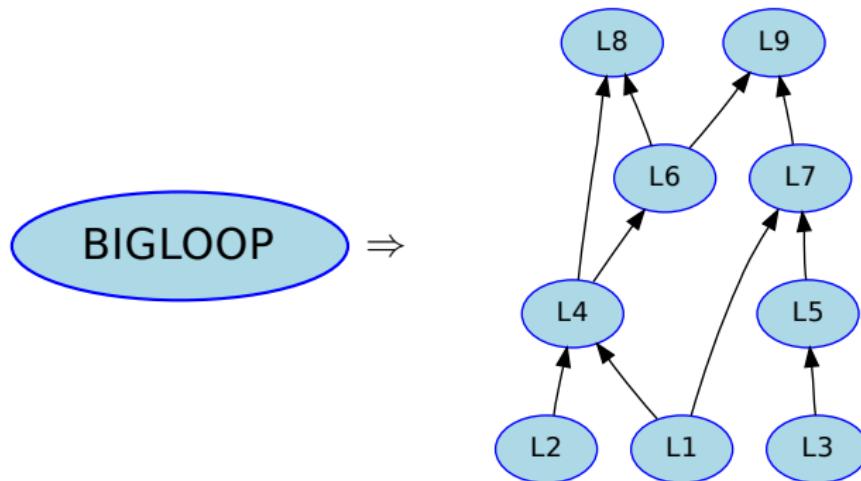
## Scalar Code

```
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);

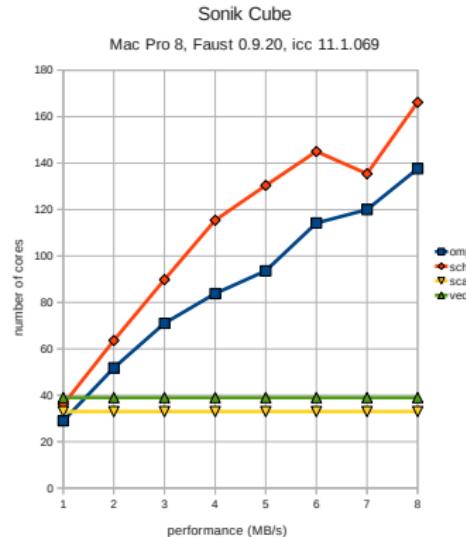
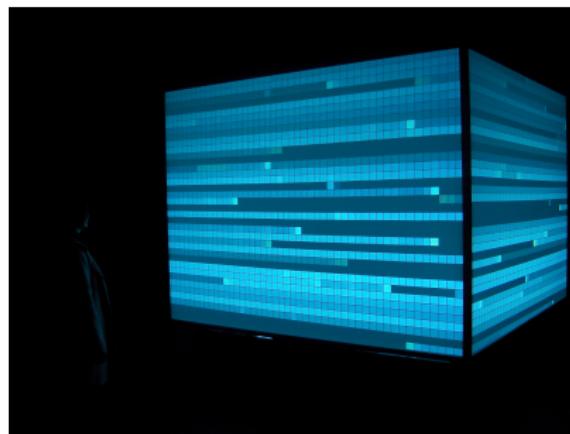
while (!fIsFinished) {
    switch (tasknum) {
        case WORK_STEALING_INDEX: {
            tasknum = TaskQueue::GetNextTask(cur_thread, fDynamicNumThreads);
            break;
        }
        case LAST_TASK_INDEX: {
            fIsFinished = true;
            break;
        }
        case 2: {
            // 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

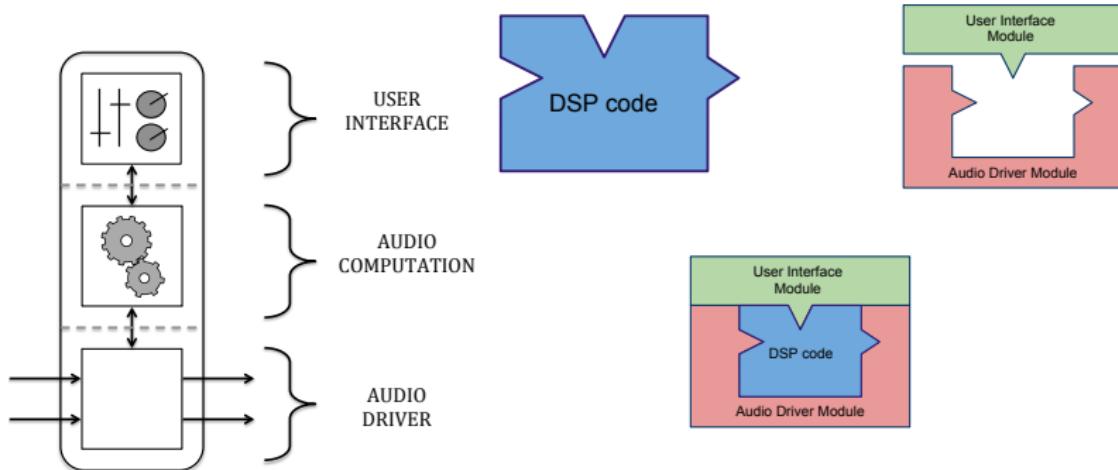
Audio-visual installation involving a cube of light, reacting to sounds, immersed in an audio feedback room (Trafik/Orlarey 2006).



# Architectures and Deployment

## Separation of concern

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



# Architectures and Deployment

## 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

# Architectures and Deployment

→ Deployment Demo

# Future Works

- Computation Model
  - ▶ Control Signals
  - ▶ Multi-Rate
  - ▶ Physical Modeling
  - ▶ Linear Algebra
- Backends
  - ▶ TensorFlow
  - ▶ FPGA
  - ▶ GPU
- Programming Approaches
  - ▶ Programming by Example
  - ▶ Programming by Machine Learning
  - ▶ Tangible Programming