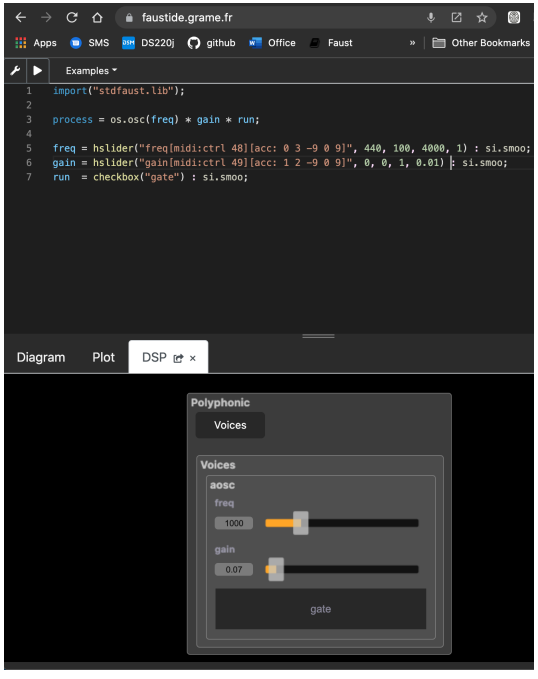
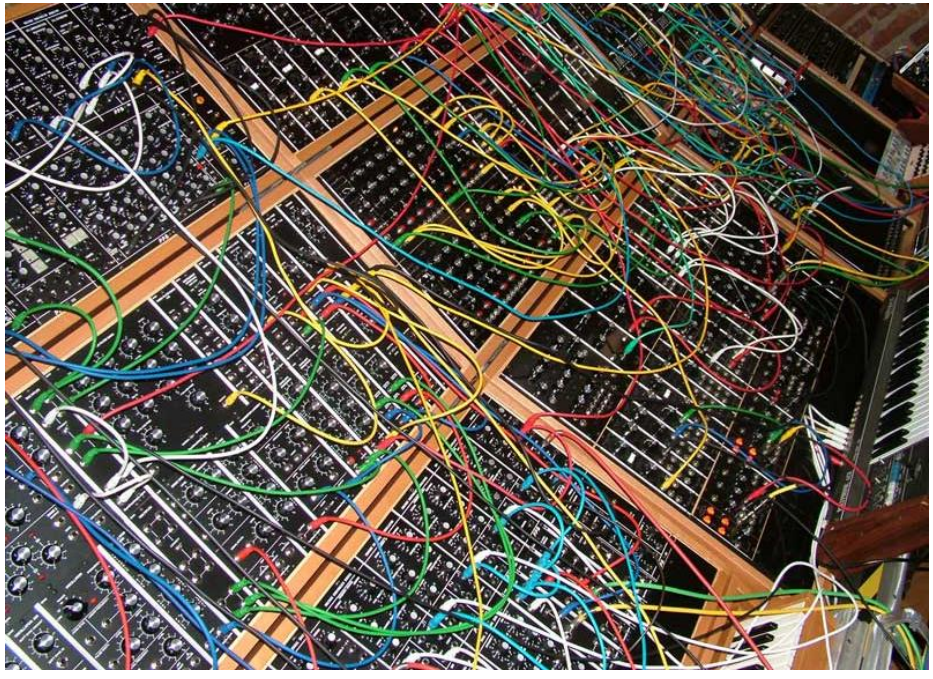
The Faust Programming Language

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."*

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/C++, LLVM IR, WebAssembly, Rust, Cmajor, etc.
- Code runs on most platforms: embedded systems, web pages, mobile devices, plug-ins, standalone applications, FPGAs, etc.



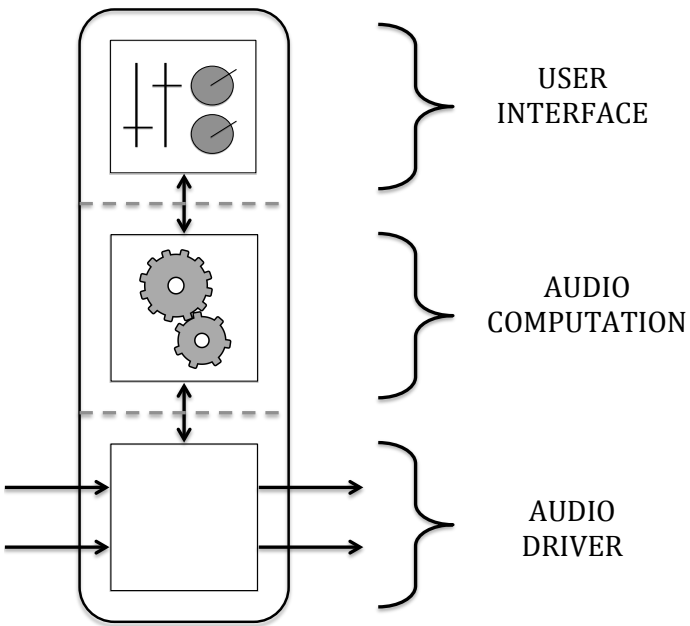


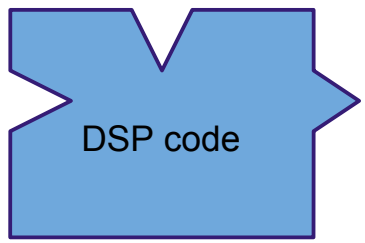


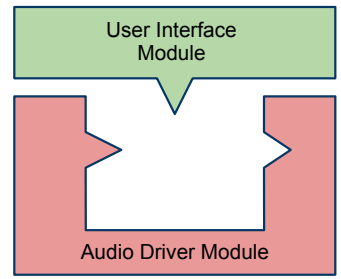
Architectures and Deployment

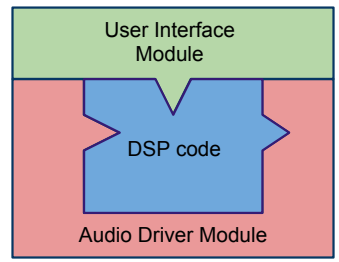
Separation of concern

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









The Faust Ecosystem

Compilers

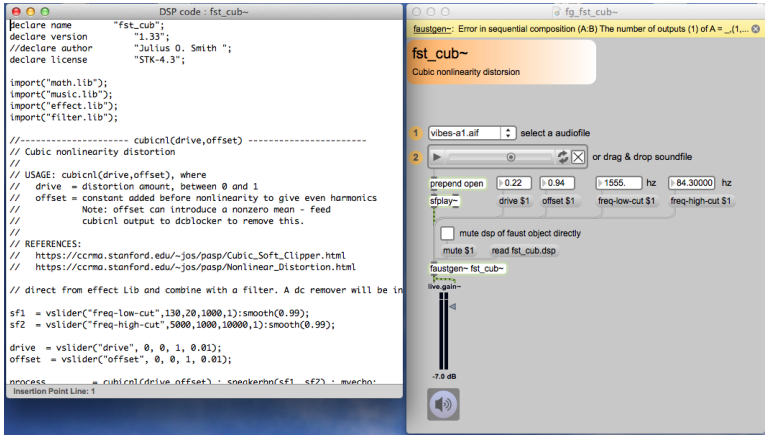
- Command Line Compilers
  - `faust` command line
  - `faust2xxx` command line
  - FaustWorks (IDE)
- Embedded Compilers (libfaust with the LLVM backend)
  - FaustLive (self contained)
  - Faustgen for Max/MSP
  - Faustcompile, etc. for Csound (V. Lazzarini)
  - Faust4processing
  - Antescofo (IRCAM's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

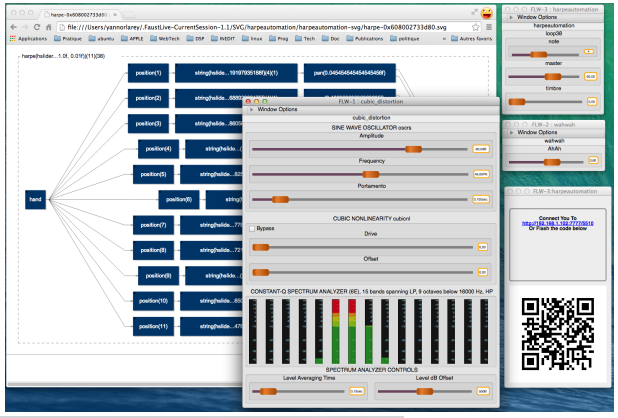
The Faust Ecosystem

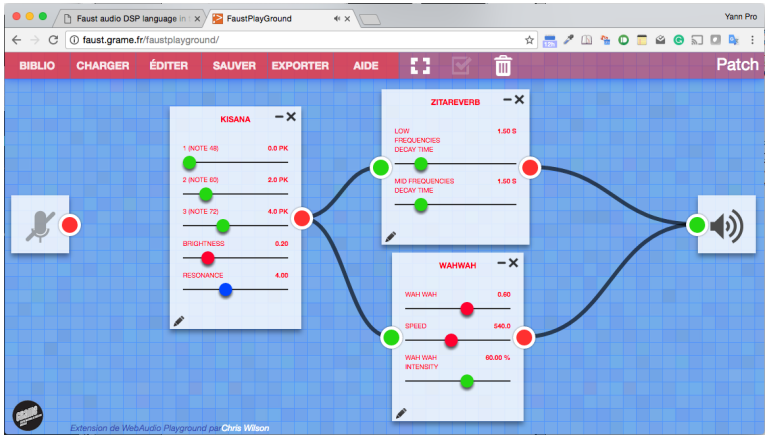
Libraries

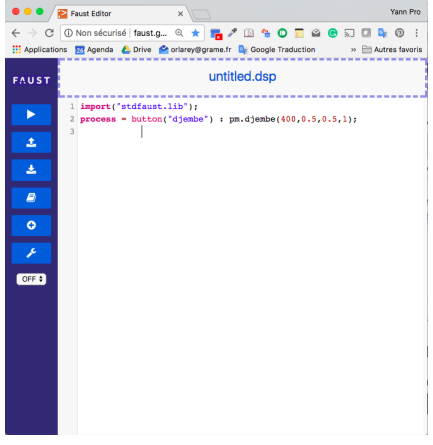
- The Faust libraries implement hundreds of DSP functions for audio processing and synthesis. They are organized by types in a set of .lib files (e.g., envelopes.lib, filters.lib, etc.).
- All documented at: `https://faustlibraries.grame.fr`
- And community contributions here: `https://faustlibraries.grame.fr/community`

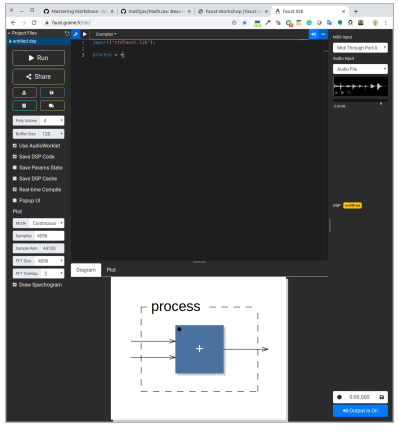
The Faust Ecosystem











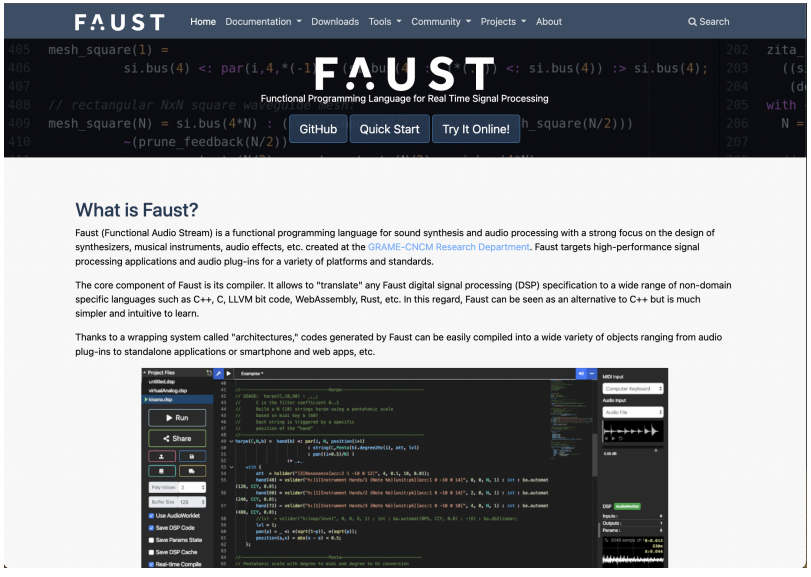
Quick Demo

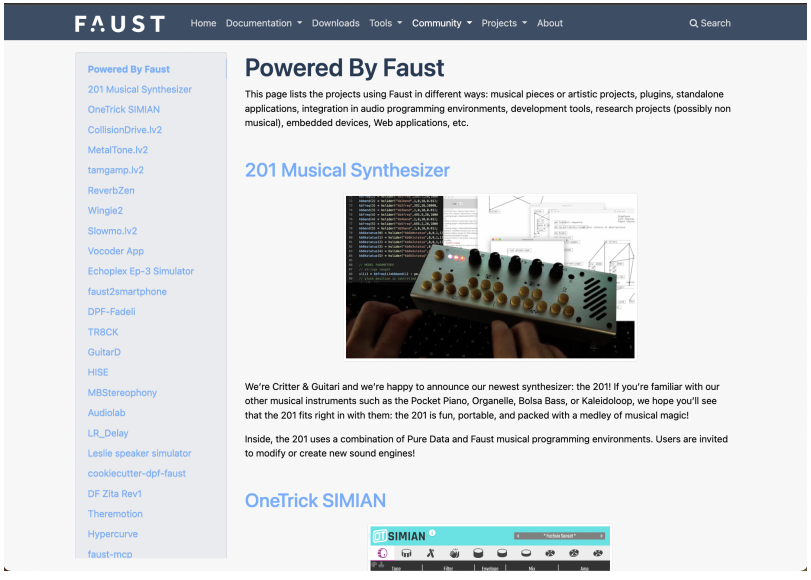
Scan the QR code with your smartphone

- `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





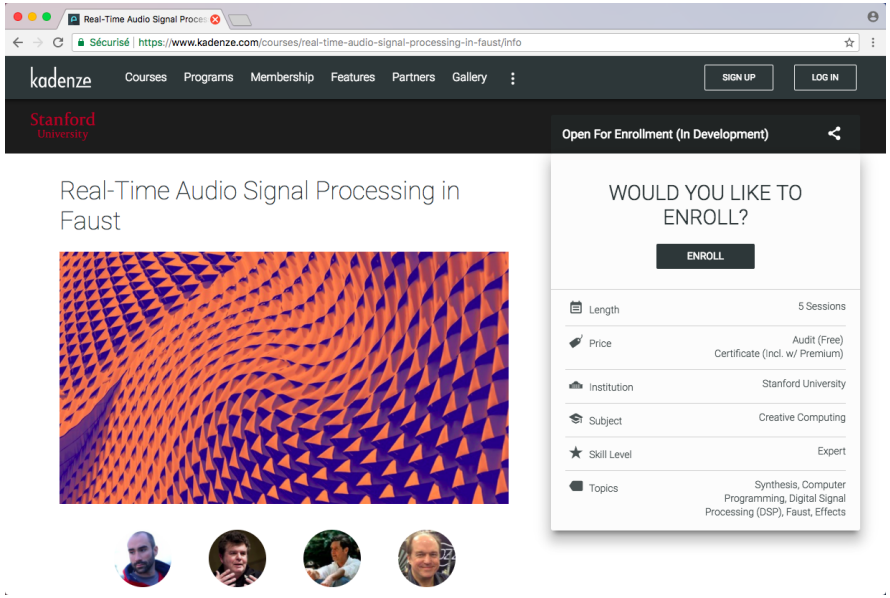
Learn Faust

International:

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

France:

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



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

Purely Functional Approach

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

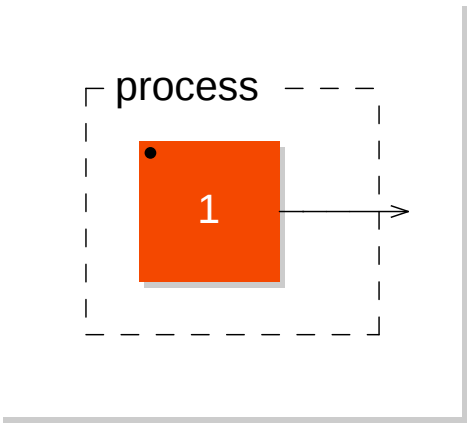
What is Faust used for?

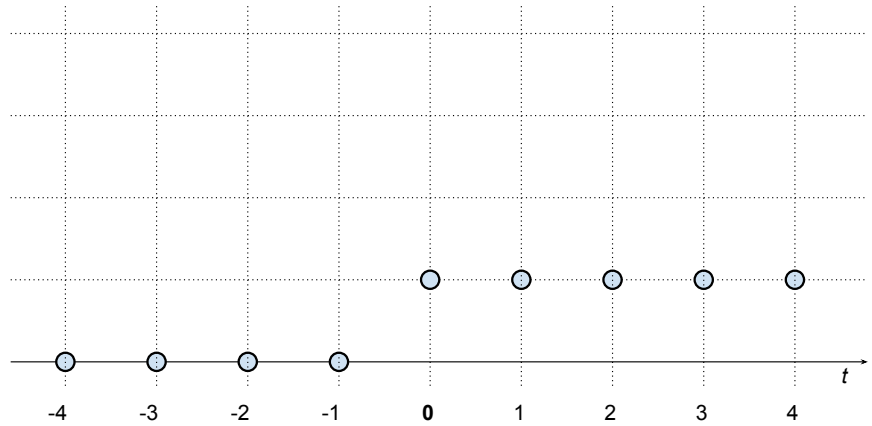
- Faust offers end-users a high-level alternative to C/C++ to develop audio applications for a large variety of platforms.
- Faust is used on stage for concerts and artistic productions, for education and research, for open sources projects and commercial applications
- The role of the Faust compiler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

Signal generators

Numbers

```
process = 1;
```



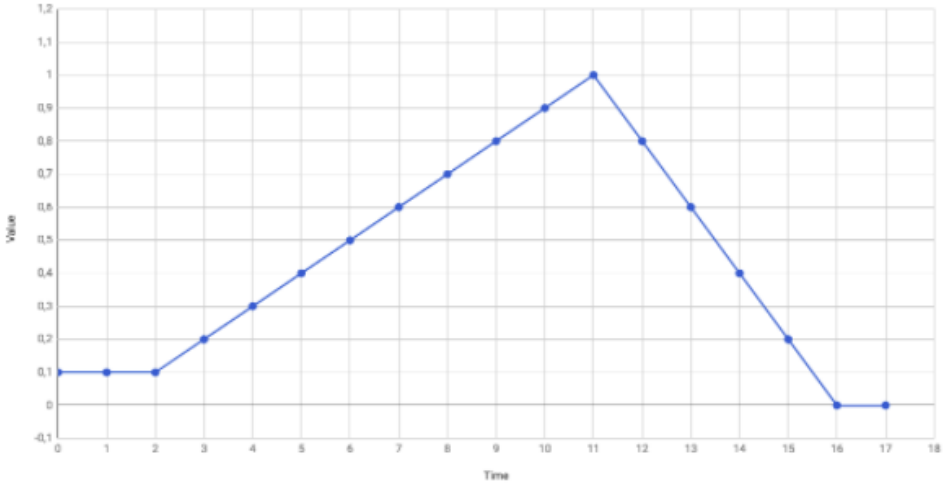


$$y(t) = \begin{cases} 1 & t \geq 0 \\ 0 & t < 0 \end{cases}$$

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: $<>, x(t) isignored$

Faust Primitives

Arithmetic operations

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

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: $y(t) = x_1(t) < x_2(t)$
<=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less or equal: $y(t) = x_1(t) <= x_2(t)$
>	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater than: $y(t) = x_1(t) > x_2(t)$
>=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater or equal: $y(t) = x_1(t) >= x_2(t)$
==	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	equal: $y(t) = x_1(t) == x_2(t)$
!=	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	different: $y(t) = x_1(t) != x_2(t)$

Faust Primitives

Trigonometric functions

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

Faust Primitives

Other Math operations

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

Faust Primitives

Delays and Tables

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

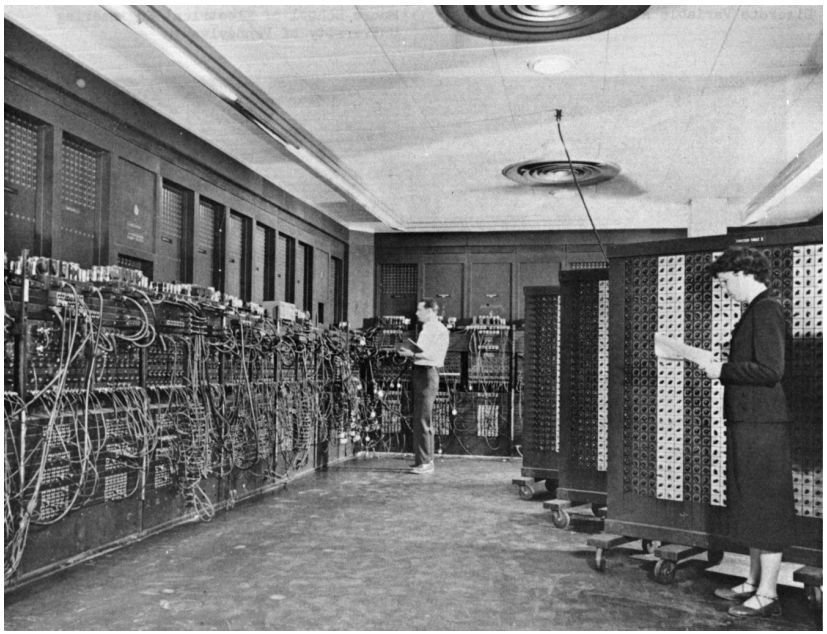
Faust Primitives

Other User Interface Primitives

Syntax	**Example**
`vgroup(`str`,`$block\text{-}diagram$`)`	`vgroup("reverb", ...)`
`hgroup(`str`,`$block\text{-}diagram$`)`	`hgroup("mixer", ...)`
`tgroup(`str`,`$block\text{-}diagram$`)`	`vgroup("parametric", ...)`
`vbargraph(`str`,`min`,`max`)`	`vbargraph("input",0,100)`
`hbargraph(`str`,`min`,`max`)`	`hbargraph("signal",0,1.0)`

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:

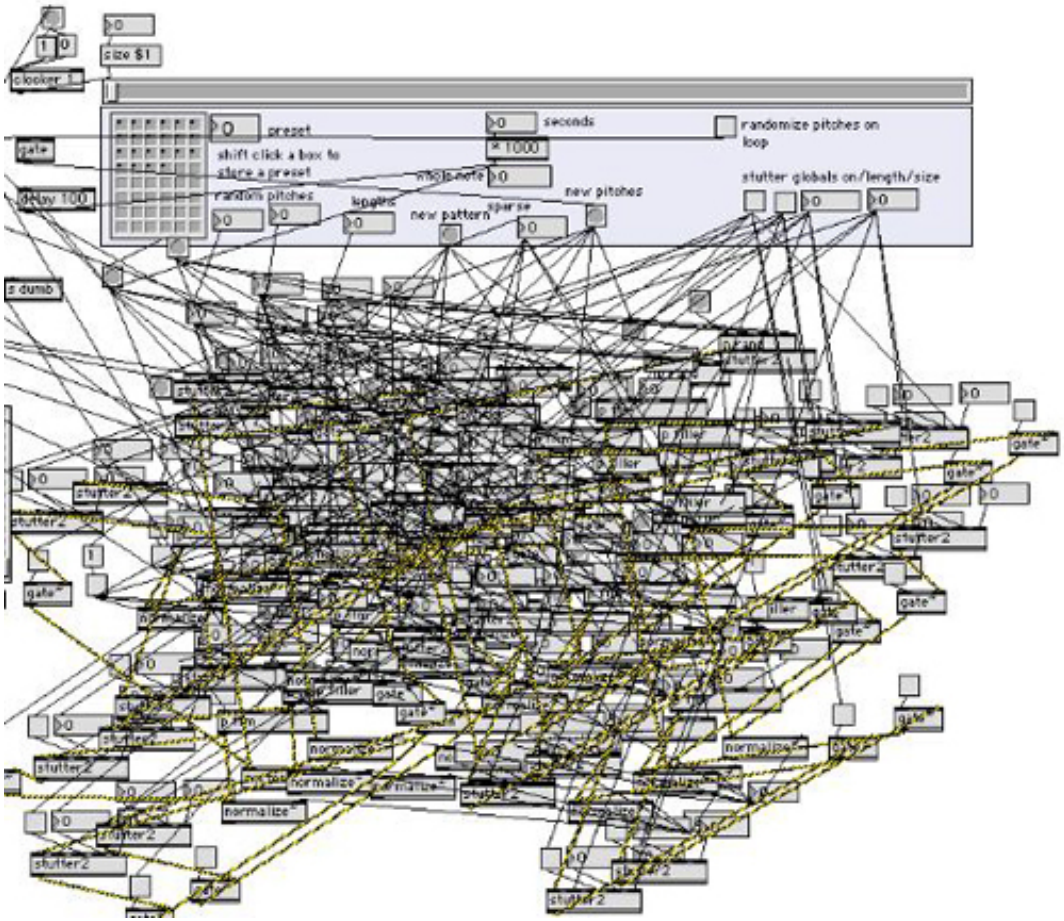


Figure: Block-diagrams can be a mess

Block-Diagram Algebra

Faust allows structured block-diagrams

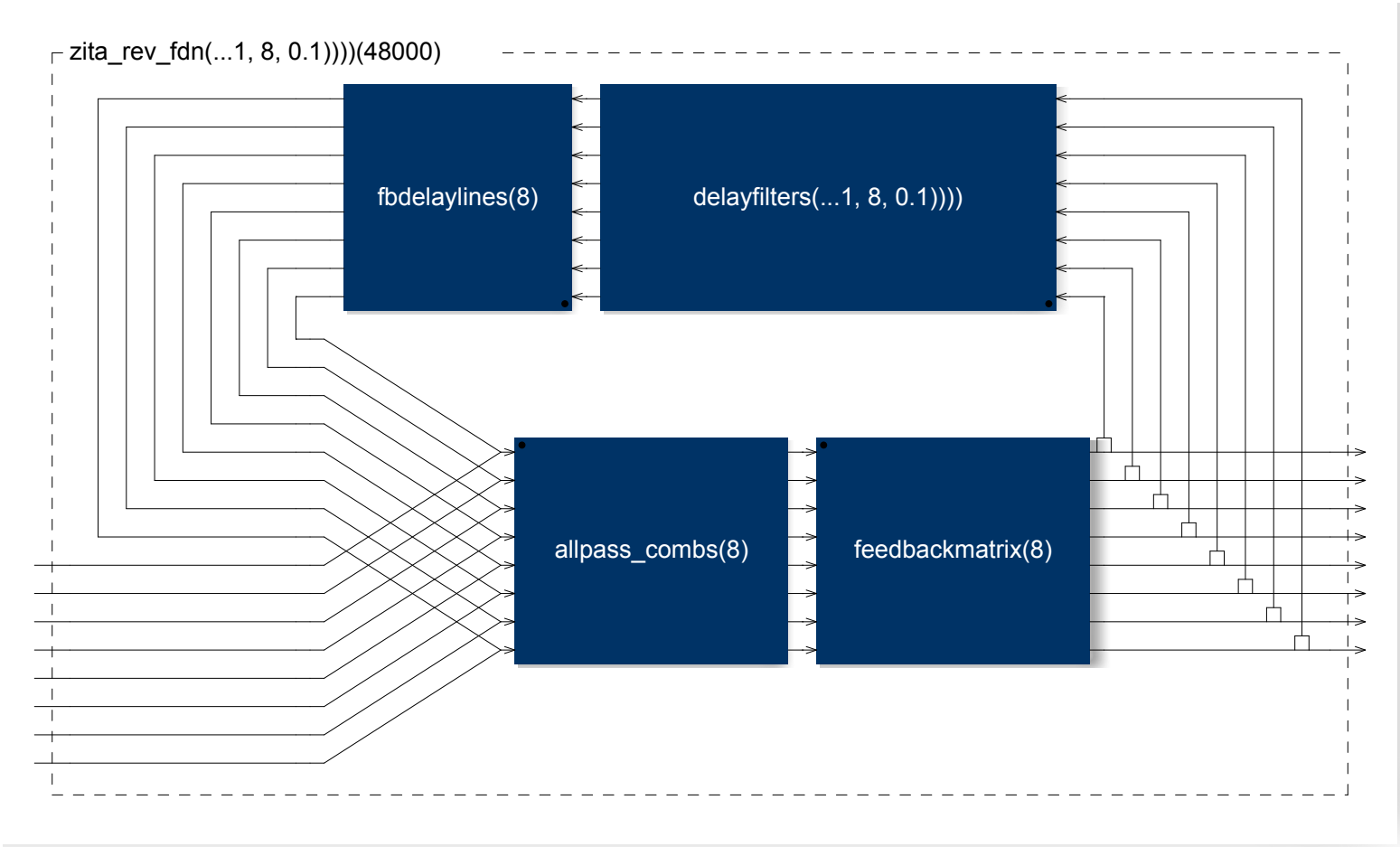


Figure: A complex but structured block-diagram

Block-Diagram Algebra

Faust syntax is based on a *block diagram algebra*

5 Composition Operators

- `(A~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.

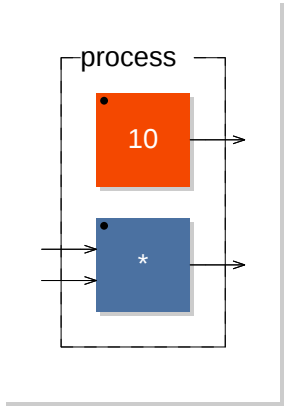


Figure: Example of parallel composition `(10,*)`

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.

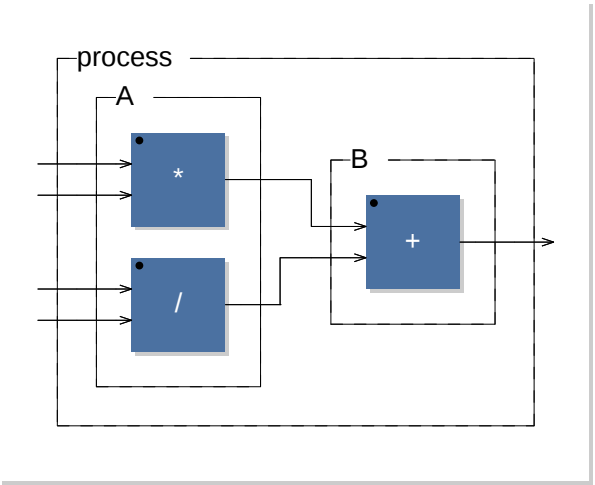


Figure: Example of sequential composition `((*,/):+)`

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.

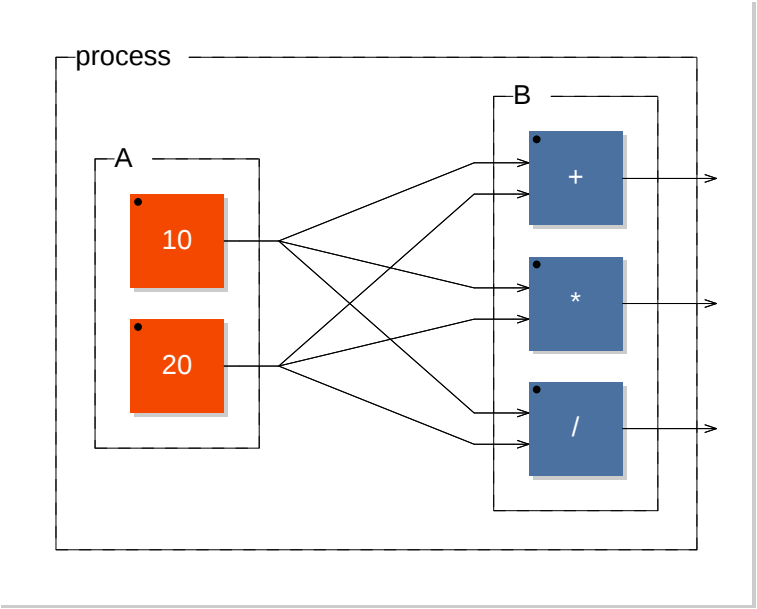


Figure: example of split composition `((10,20) <: (+,*,/))`

Block-Diagram Algebra

Merge Composition

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

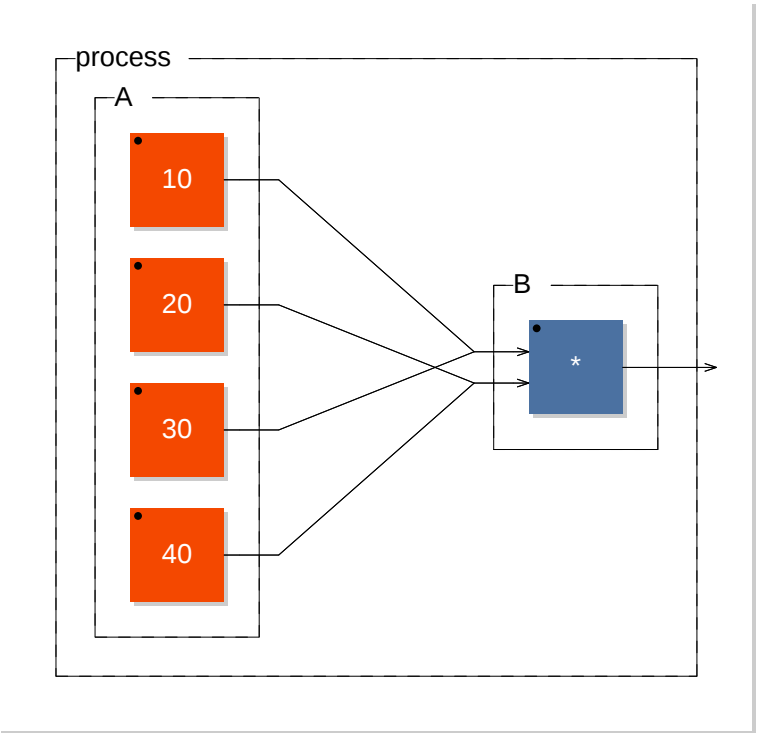


Figure: example of merge composition `((10,20,30,40) :> *)`

Block-Diagram Algebra

Recursive Composition

The *recursive composition* `(A~B)` is used to create cycles in the block-diagram in order to express recursive computations.

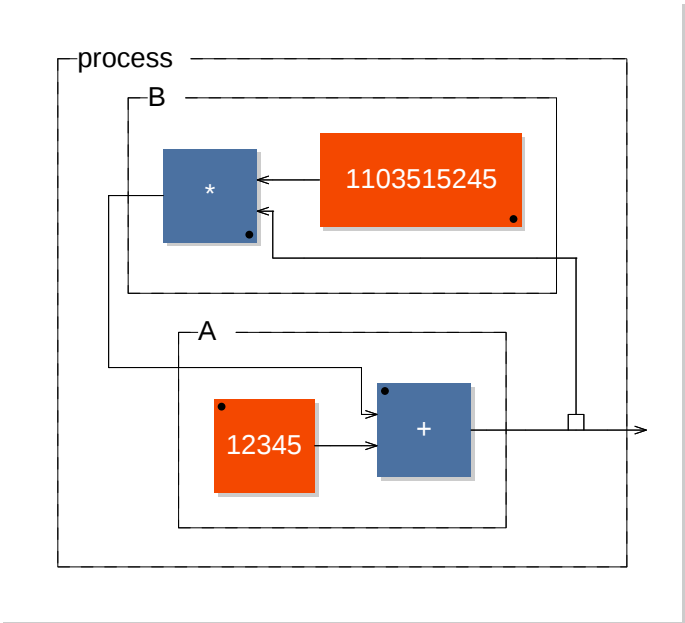


Figure: example of recursive composition `+(12345) ~ *(1103515245)`

Language Expressivity

- Function definition and composition
- Anonymity
- Faust programs as components, libraries, namespaces
- Partial application
- Lexical environments as first class citizen
- Pattern Matching (meta programmation = algorithmic description of circuits)

Language Expressivity

- Function definition and composition
- Anonymity
- Faust programs as components
- Libraries, namespaces
- Partial application
- Lexical environments as first class citizen
- Pattern Matching (meta programmation = algorithmic description of circuits)

Language Expressiveness

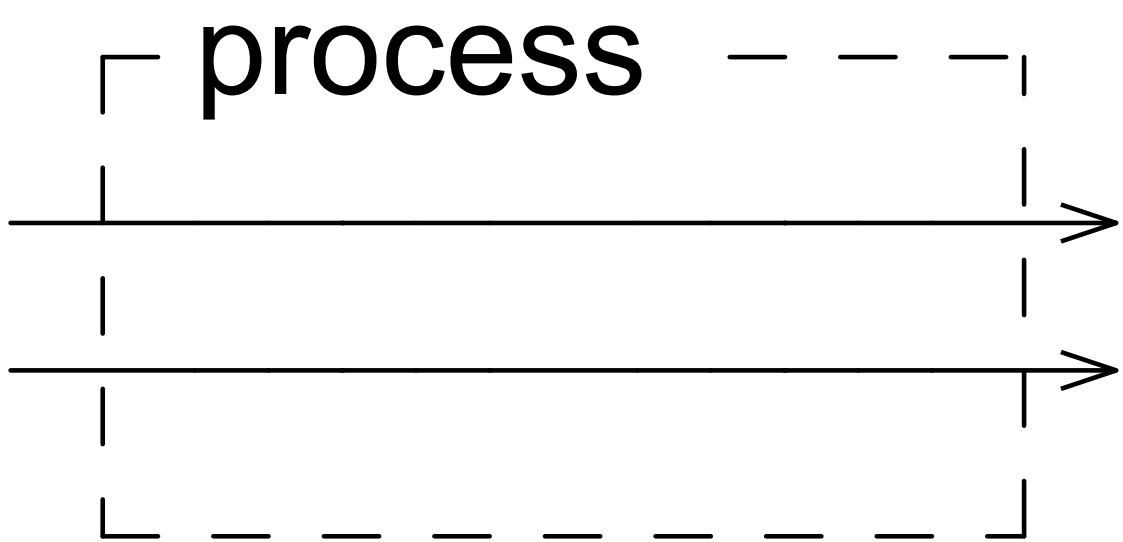
Fast Fourier Transform

```
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

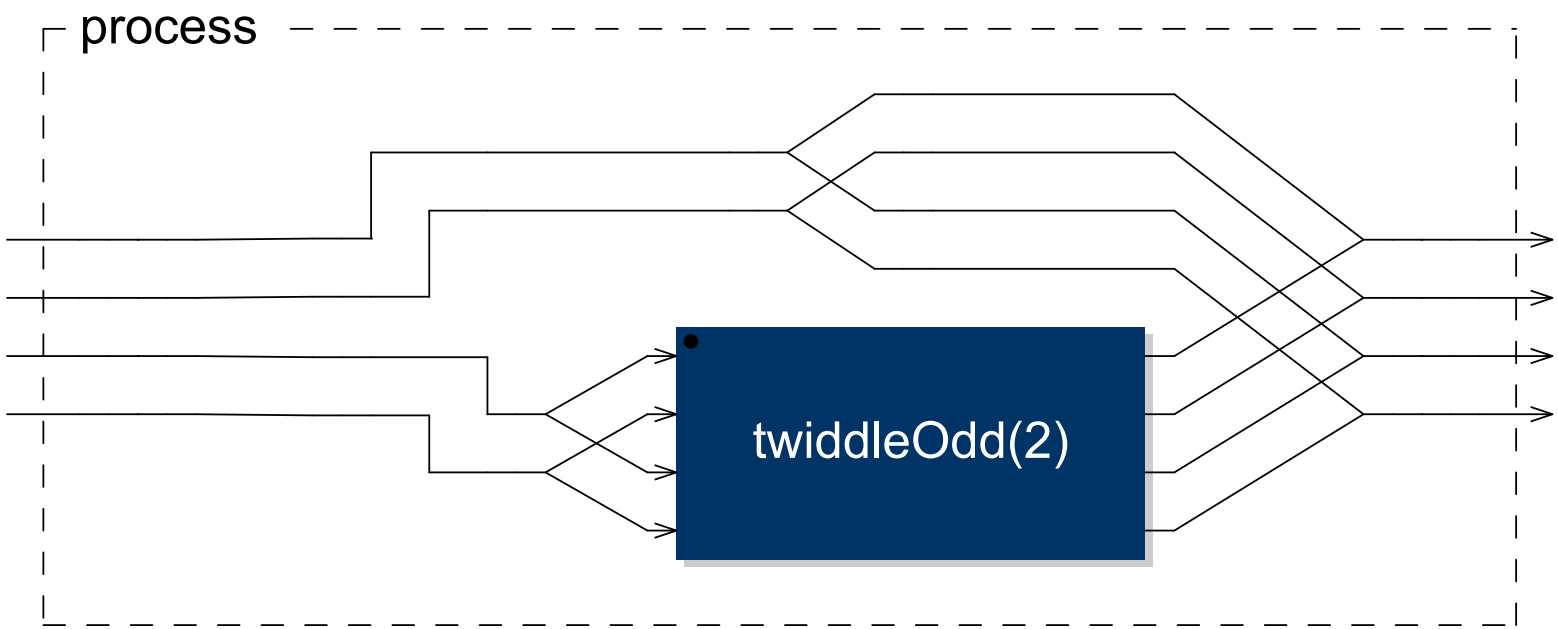
```
fft(1)
```



Language Expressiveness

Fast Fourier Transform

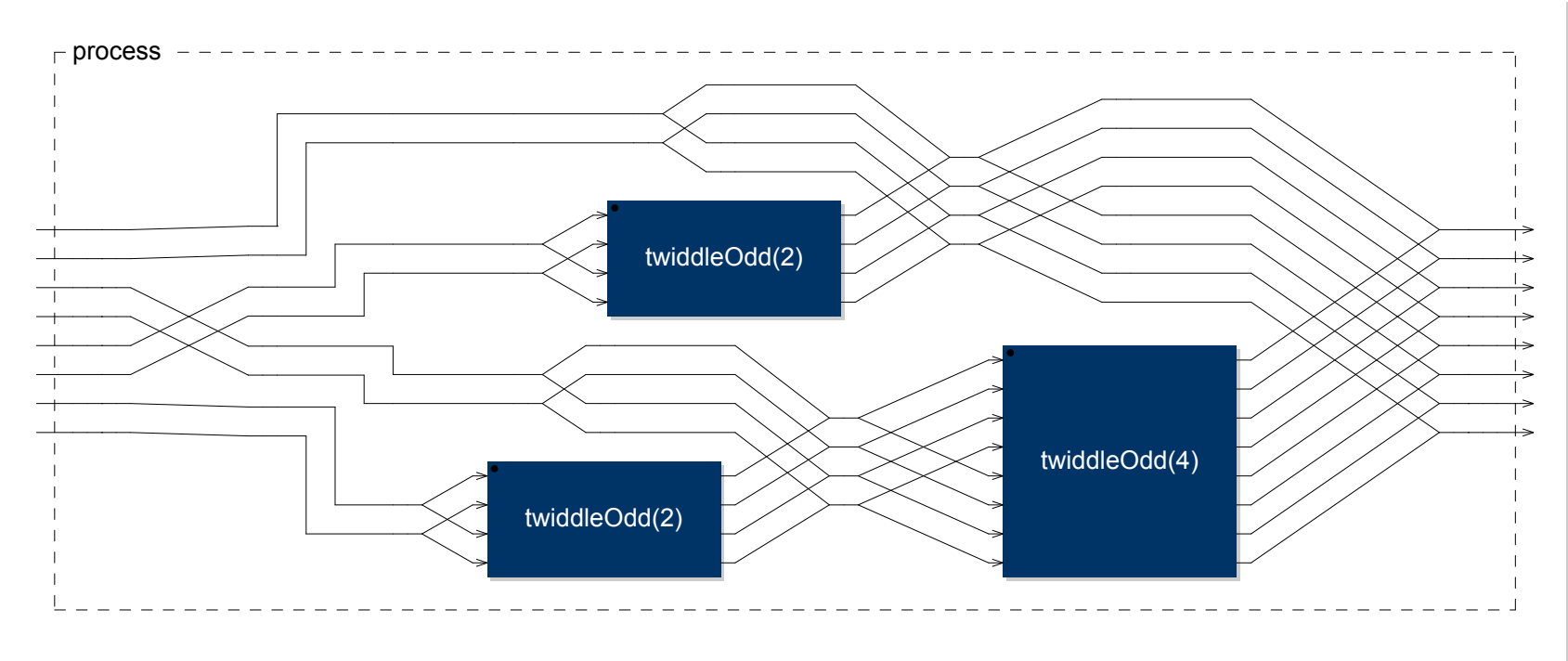
```
fft(2)
```



Language Expressiveness

Fast Fourier Transform

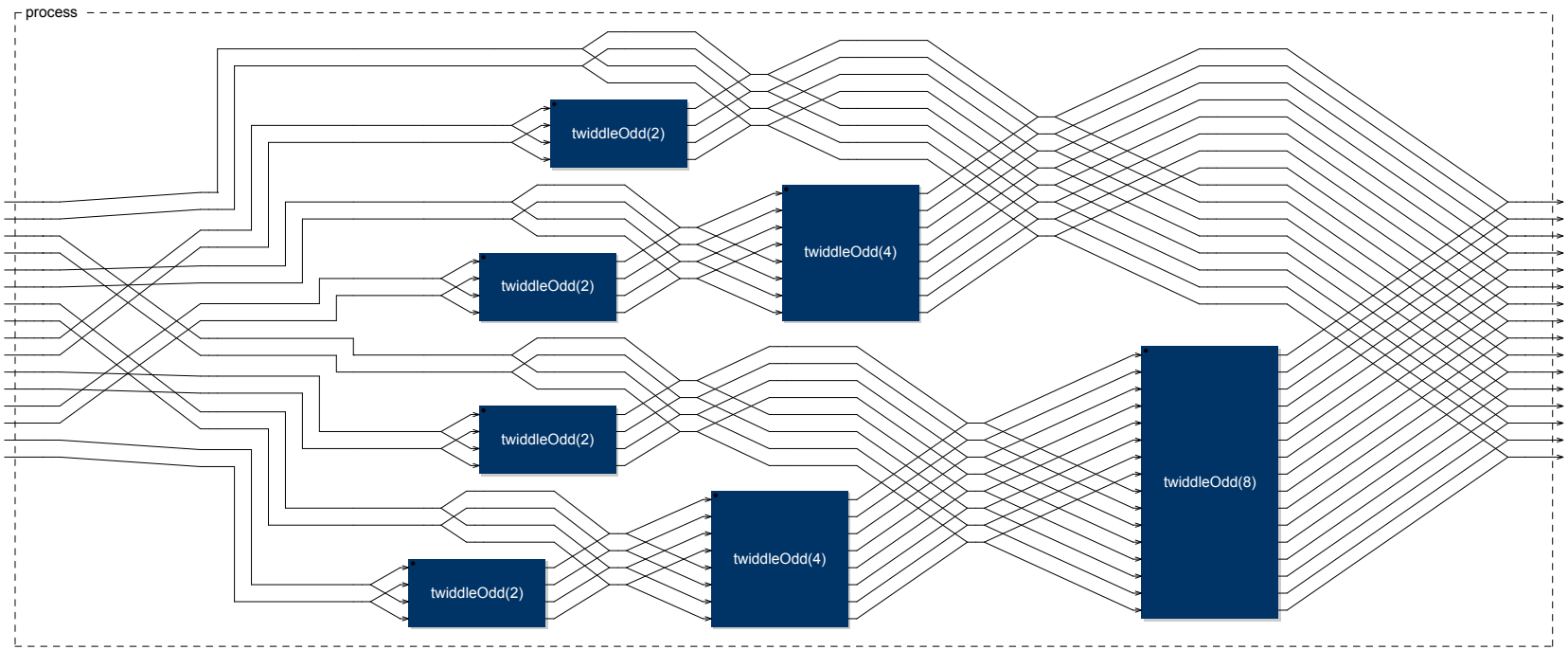
```
fft(4)
```



Language Expressiveness

Fast Fourier Transform

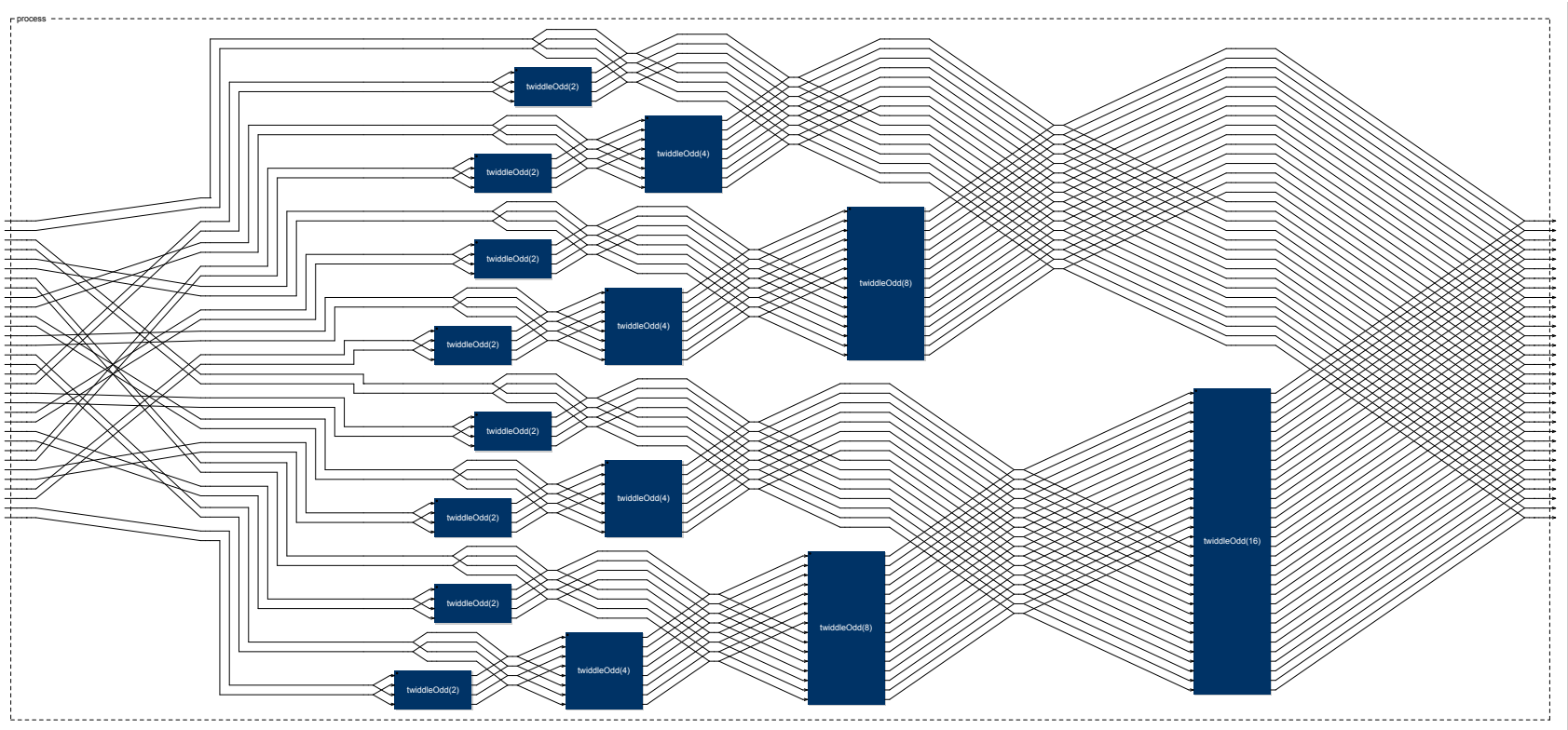
```
fft(8)
```



Language Expressiveness

Fast Fourier Transform

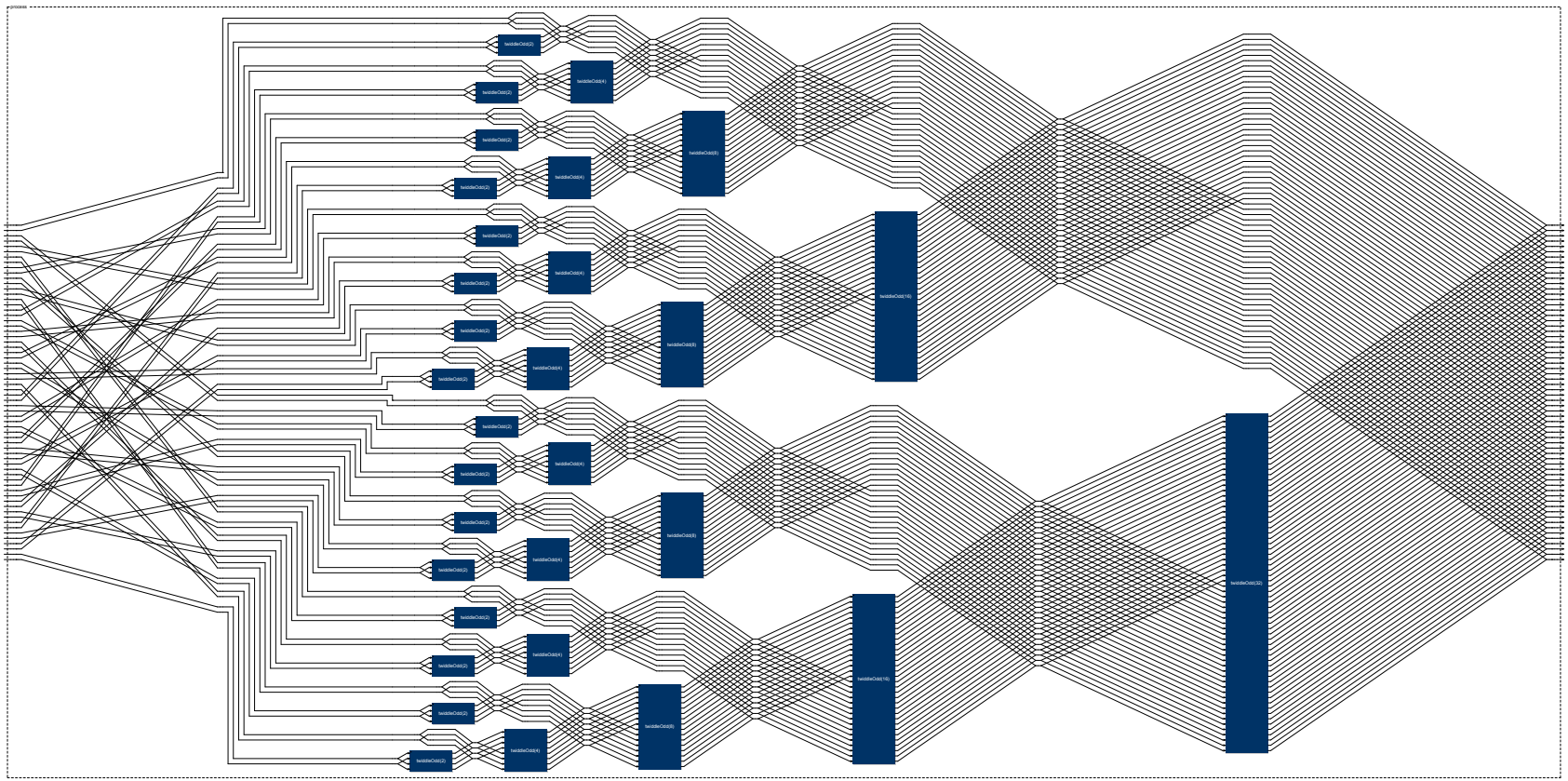
```
fft(16)
```



Language Expressiveness

Fast Fourier Transform

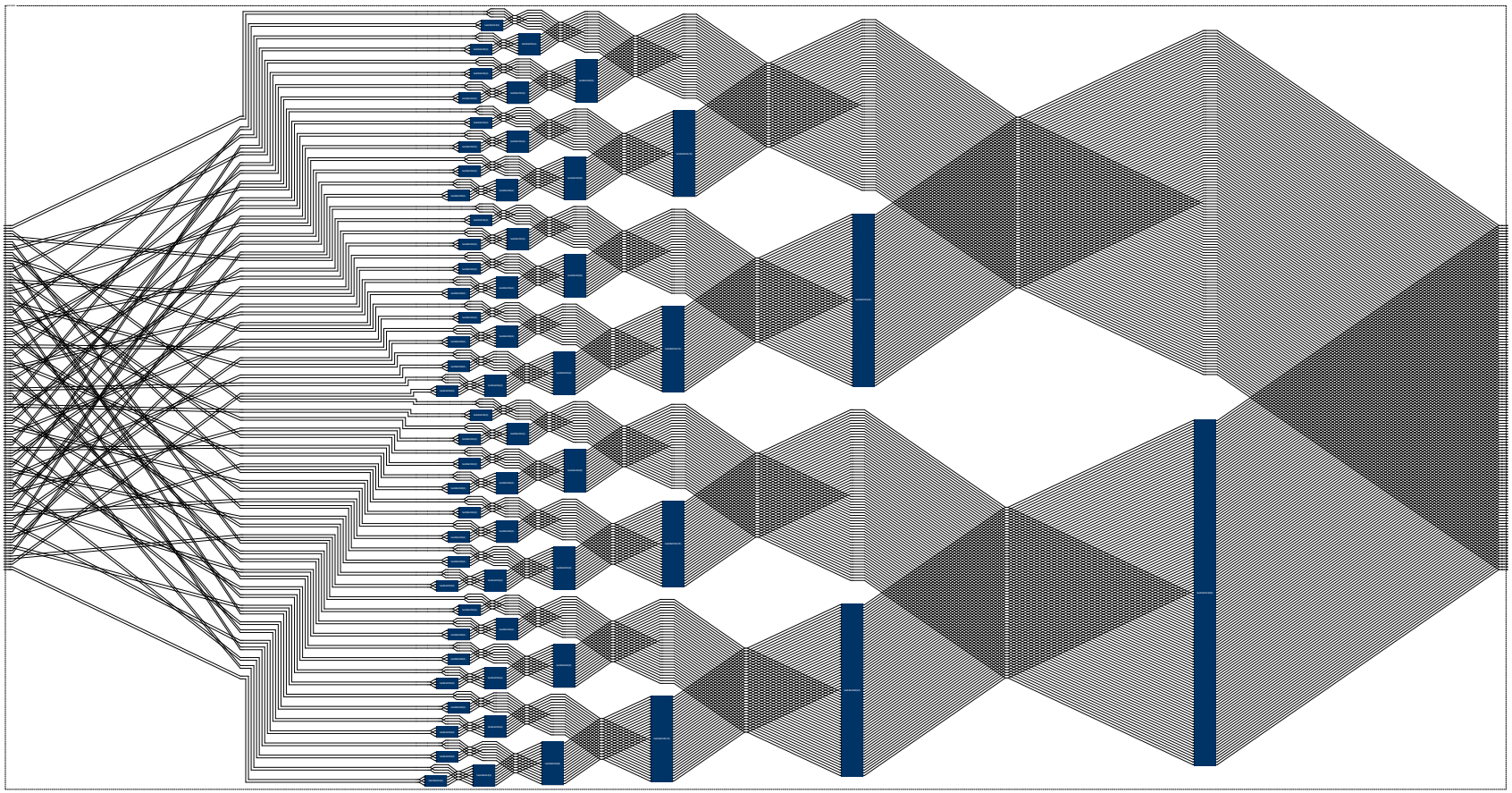
```
fft(32)
```



Language Expressiveness

Fast Fourier Transform

```
fft(64)
```



Performance Quest

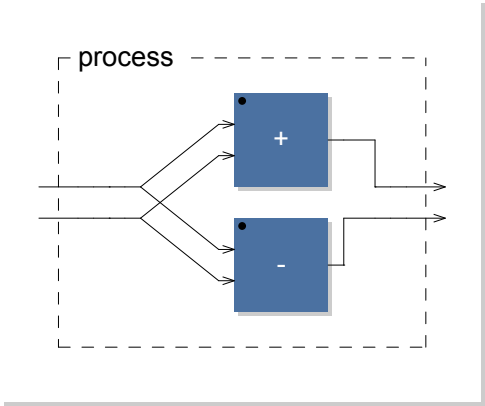
- Fully compiled to native code
- Sample level semantics
- Specification language
- Automatic parallelization

Fully compiled to native code

Faust code:

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

Block-diagram:



C++ translation:

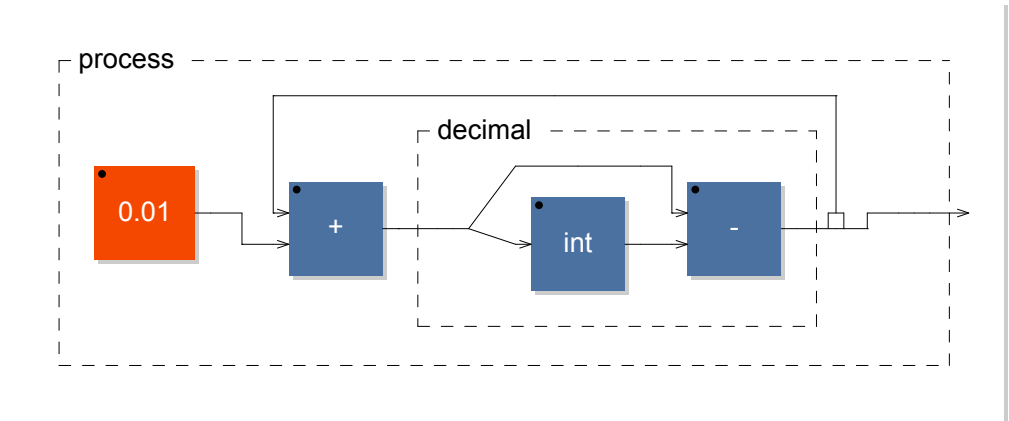
```
for (int i = 0; (i < count); i = (i + 1)) {
    float fTemp0 = input0[i];
    float fTemp1 = input1[i];
    output0[i] = fTemp0 + fTemp1;
    output1[i] = fTemp0 - fTemp1;
}
```

Sample level semantics

Sawtooth signal by step of 0.01:

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

Block-diagram:



Signal equation:

$$
\begin{aligned}
y(t < 0) &= 0 \\
y(t \geq 0) &= \mathit{decimal}(y(t-1) + 0.01)
\end{aligned}
$$

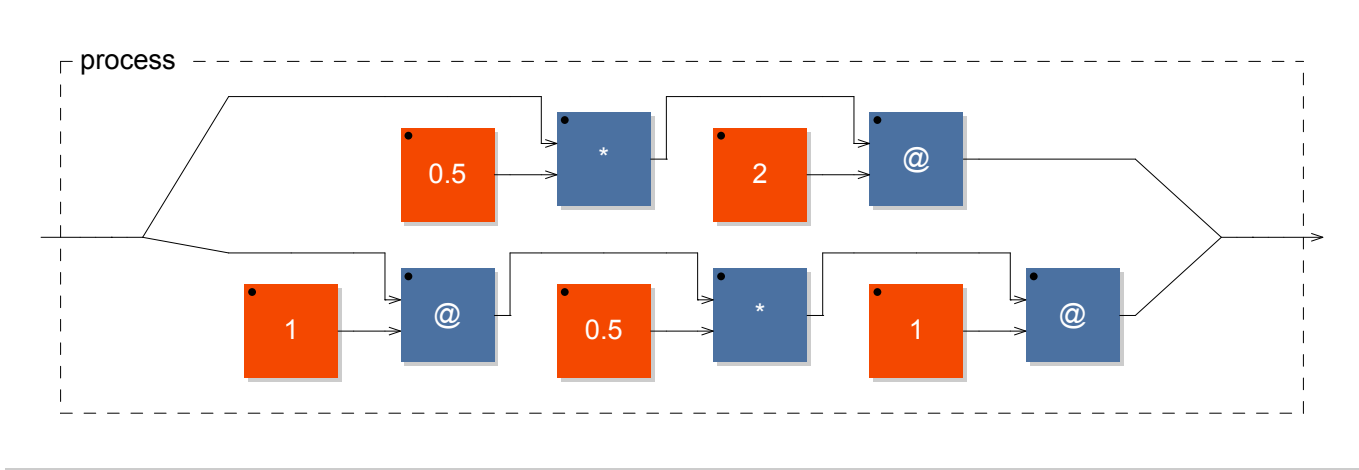
Specification Language

Leave the implementation to the compiler

User's code:

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

Block-diagram:

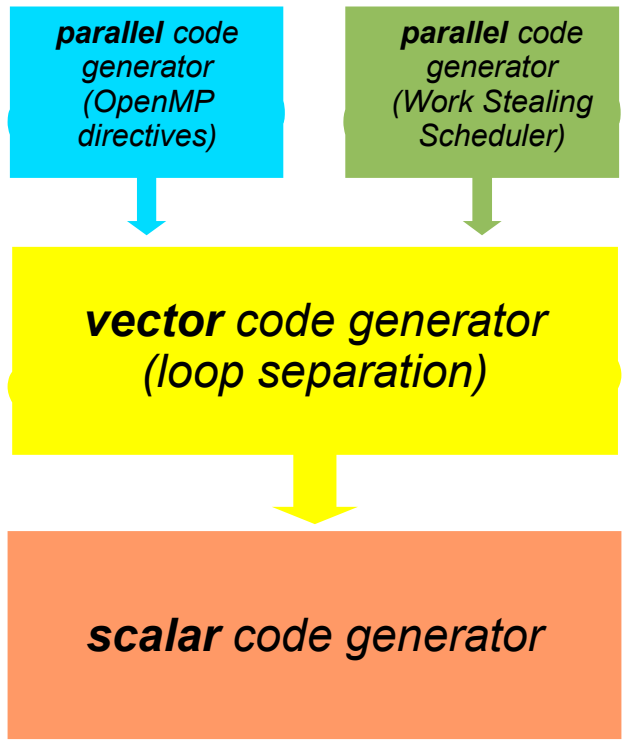


Equivalent, more efficient code

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

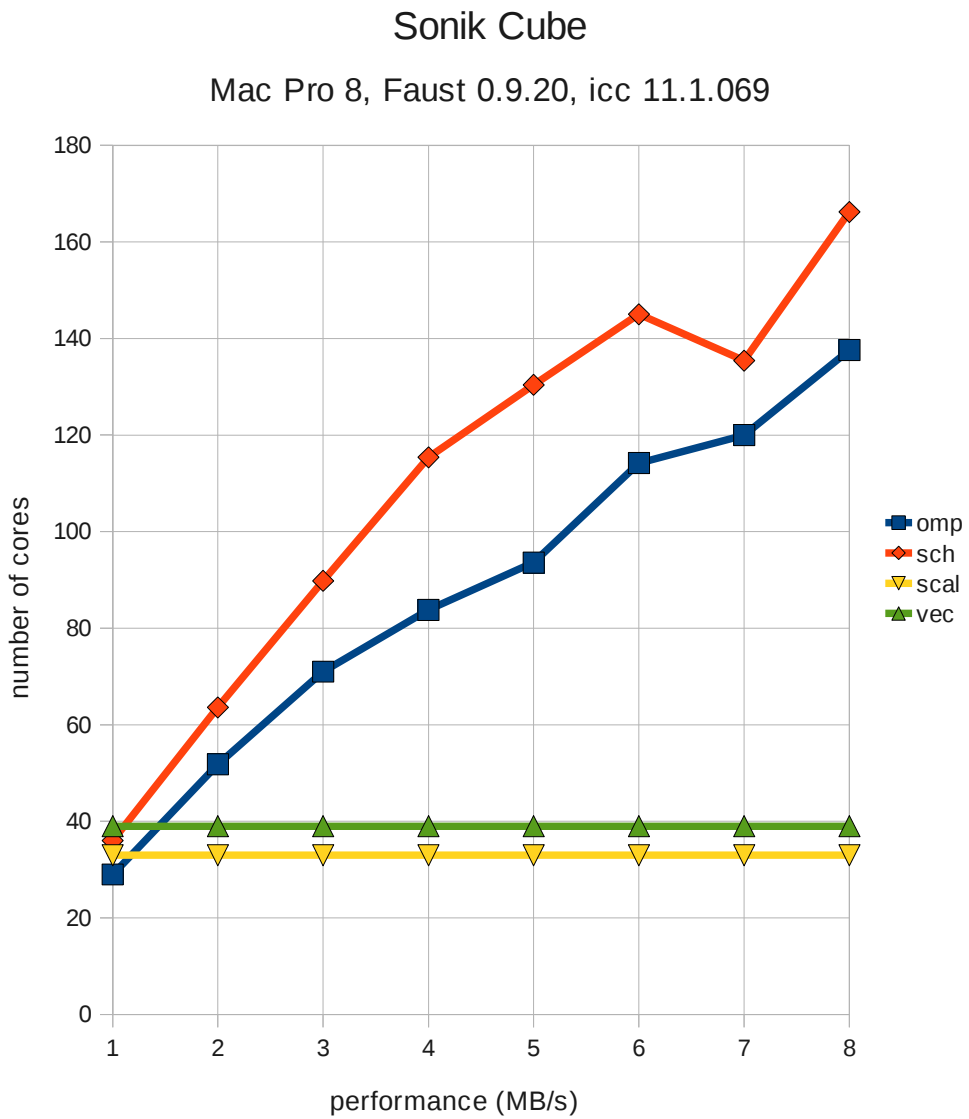
Automatic Parallelization

Code Generators



Automatic Parallelization

Performances



Multiple Languages Backends

- **C++**
- C
- Rust
- **LLVM**
- **WebAssembly**
- Cmajor
- Julia
- . . .

Multiple Languages Backends

Echo in Faust

```
process = +~(@(4410):*(0.8));
```

Multiple Languages Backends

C++: `faust -lang cpp`

```
virtual void compute(int count, FAUSTFLOAT** RESTRICT inputs, FAUSTFLOAT** RESTRICT outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    for (int i0 = 0; i0 < count; i0 = i0 + 1) {
        fRec0[IOTA0 & 8191] = float(input0[i0]) + 0.8f * fRec0[(IOTA0 - 4411) & 8191];
        output0[i0] = FAUSTFLOAT(fRec0[IOTA0 & 8191]);
        IOTA0 = IOTA0 + 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 $i0 i32)
    (local.set $input0 (i32.const 0))
    (local.set $output0 (i32.const 0))
    (local.set $i0 (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 $i0 (i32.const 0))
    (loop $for-in-i0
        (block $for-out-i0
            (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 $i0))) (f32.mul (f32.const 0.8)
            (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 $i0)) (f32.load (i32.add (i32.const 4)
            (i32.shl (i32.and (i32.load (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 $i0 (i32.add (local.get $i0) (i32.const 4)))
            (if (i32.lt_s (local.get $i0) (i32.mul (i32.const 4) (local.get $count))) (br $for-in-i0) (br $for-out-i0))
        )
    )
)
```

Multiple Languages Backends

Rust code: `faust -lang rust`

```rust
fn compute(&mut self, count: i32, inputs: &[&[Self::T]], outputs: &mut[&mut[Self::T]]) {
    let (inputs0) = if let [inputs0, ..] = inputs {
        let inputs0 = inputs0[..count as usize].iter();
        (inputs0)
    } else {
        panic!("wrong number of inputs");
    };
    let (outputs0) = if let [outputs0, ..] = outputs {
        let outputs0 = outputs0[..count as usize].iter_mut();
        (outputs0)
    } else {
        panic!("wrong number of outputs");
    };
    let zipped_iterators = inputs0.zip(outputs0);
    for (input0, output0) in zipped_iterators {
        self.fRec0[(self.IOTA0 & 8191) as usize] = *input0 + 0.8
        * self.fRec0[((i32::wrapping_sub(self.IOTA0, 4411)) & 8191) as usize];
        *output0 = self.fRec0[(self.IOTA0 & 8191) as usize];
        self.IOTA0 = i32::wrapping_add(self.IOTA0, 1);
    }
}
```

Multiple Languages Backends

Cmajor code: `faust -lang cmajor`

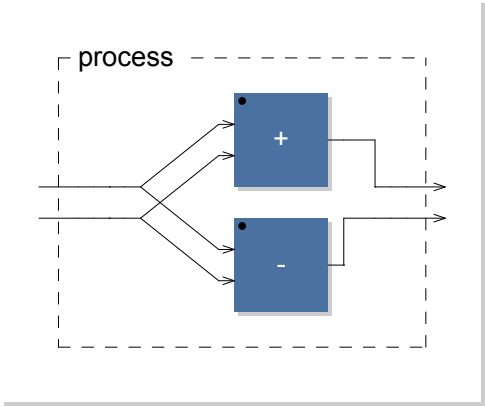
```
void main()
{
    // DSP loop running forever...
    loop
    {
        if (fUpdated) { fUpdated = false; control(); }

        // Computes one sample
        fRec0.at (IOTA0 & 8191) = input0 + 0.8f * fRec0.at ((IOTA0 - 4411) & 8191);
        output0 <- fRec0.at (IOTA0 & 8191);
        IOTA0 = IOTA0 + 1;

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

Fully compiled to native code

Faust code:

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

Block-diagram:



C++ translation:

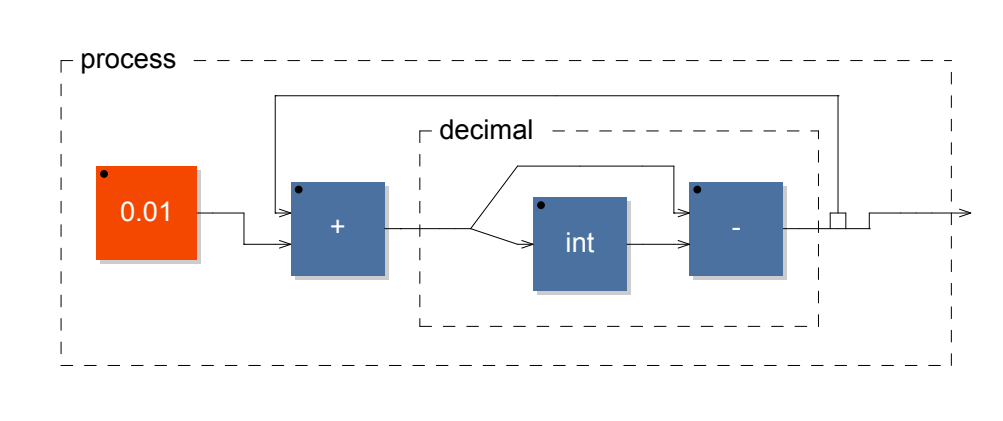
```
for (int i = 0; (i < count); i++)) {
    float fTemp0 = input0[i];
    float fTemp1 = input1[i];
    output0[i] = fTemp0 + fTemp1;
    output1[i] = fTemp0 - fTemp1;
}
```

Sample level semantics

Sawtooth signal by step of 0.01:

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

Block-diagram:



Signal equation:

$$
\begin{aligned}
y(t < 0) &= 0 \\
y(t \geq 0) &= \mathit{decimal}(y(t-1) + 0.01)
\end{aligned}
$$

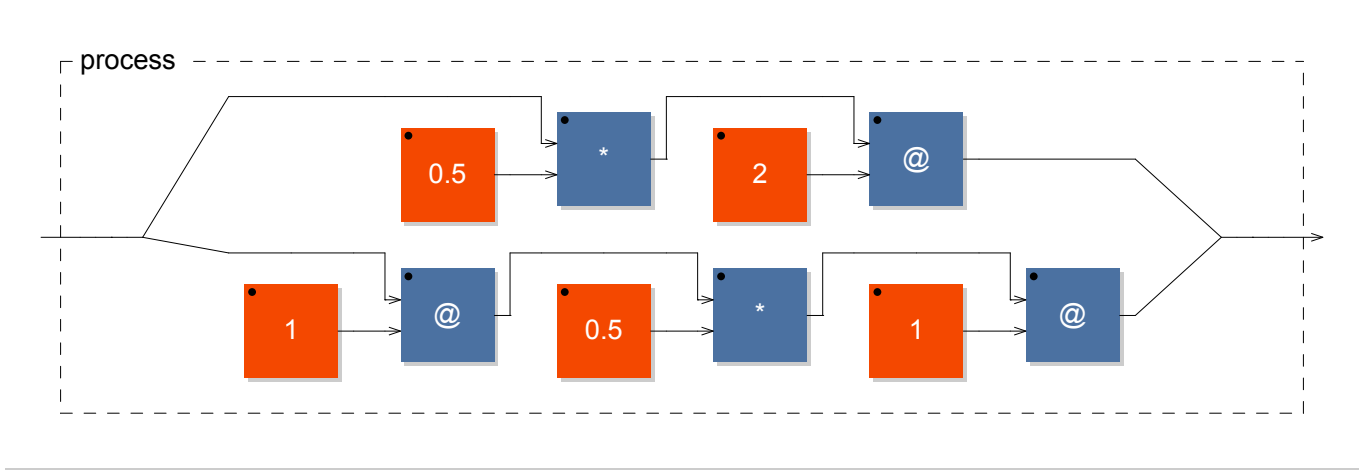
Specification Language

Leave the implementation to the compiler

User's code:

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

Block-diagram:

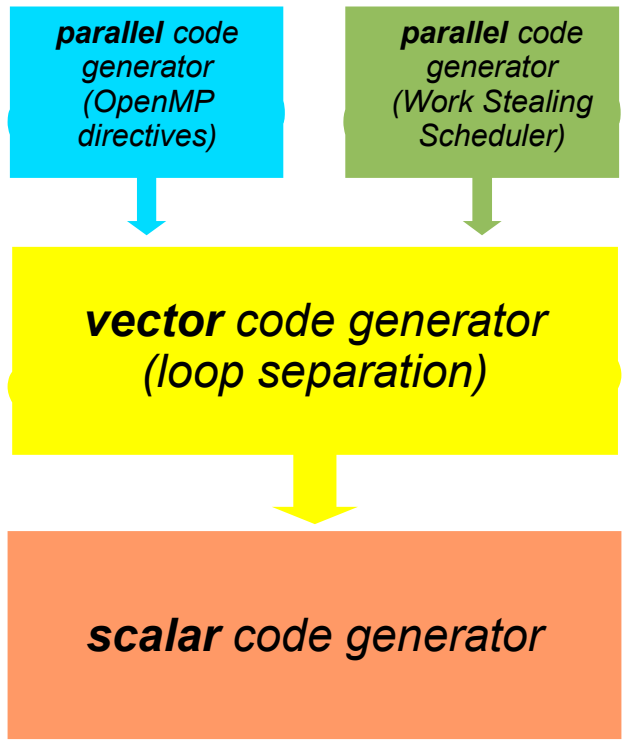


Equivalent, more efficient code

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

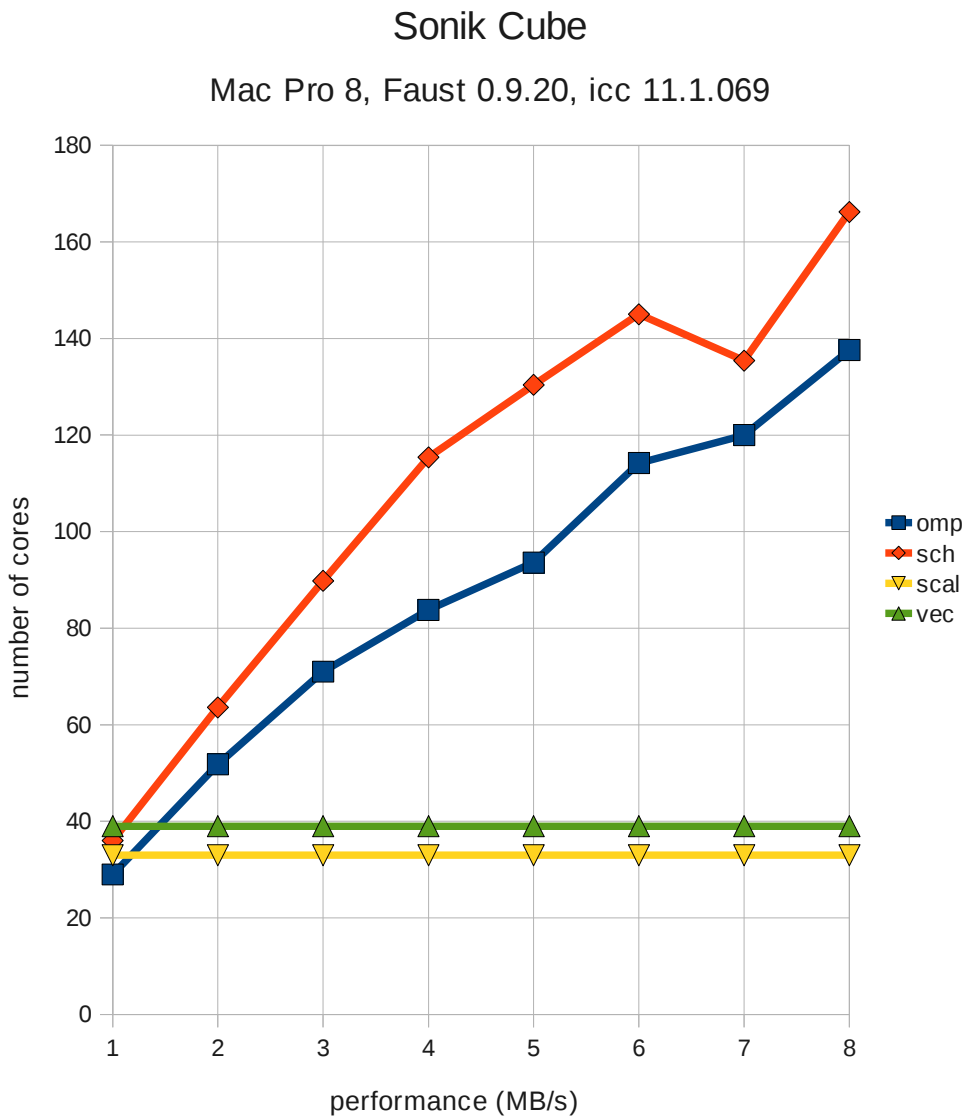
Automatic Parallelization

Code Generators

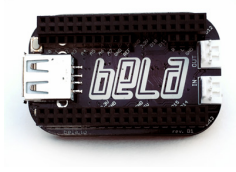


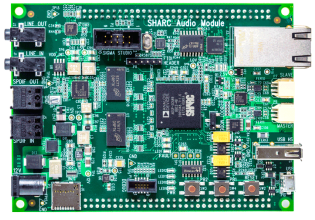
Automatic Parallelization

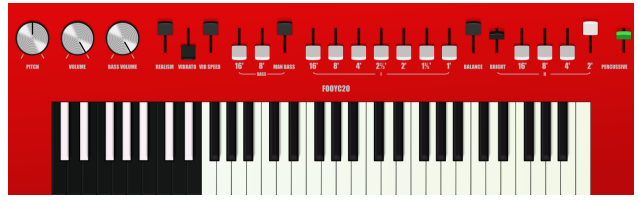
Performances



Easy Deployment



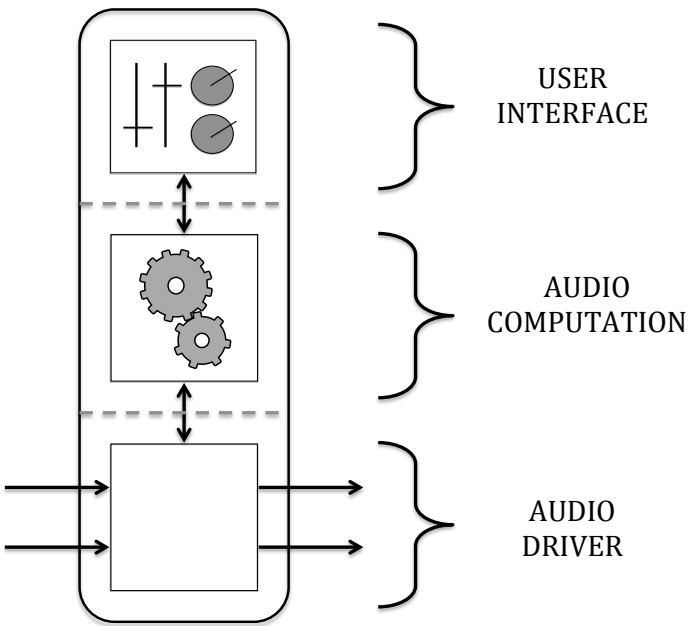


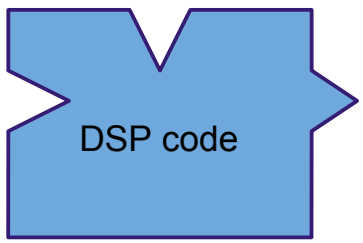


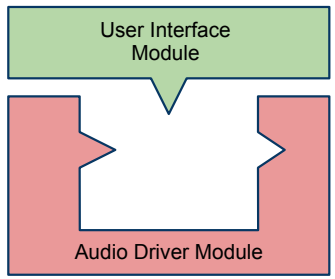
Easy Deployment

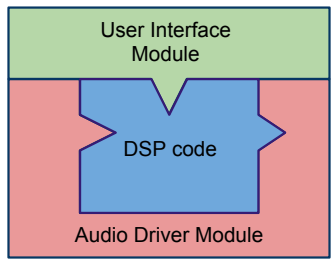
Separation of concern

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

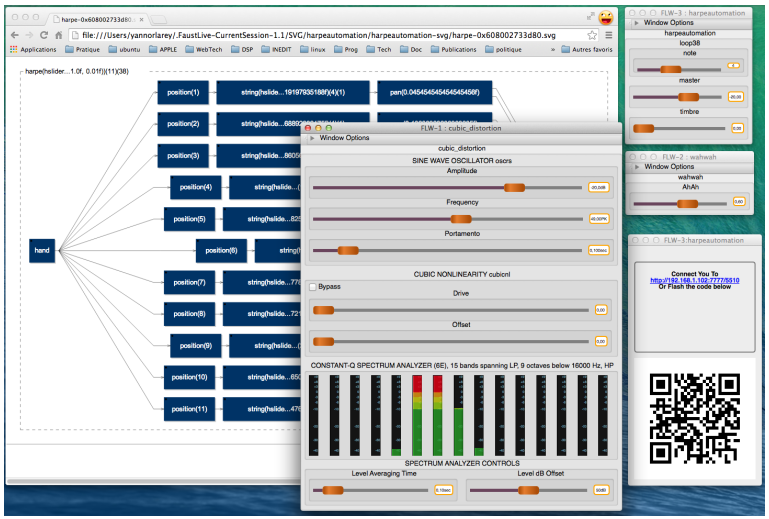
- Libfaust: embeddable version of the Faust compiler coupled with LLVM
- Libfaust.js: embeddable Javascript version of the Faust compiler

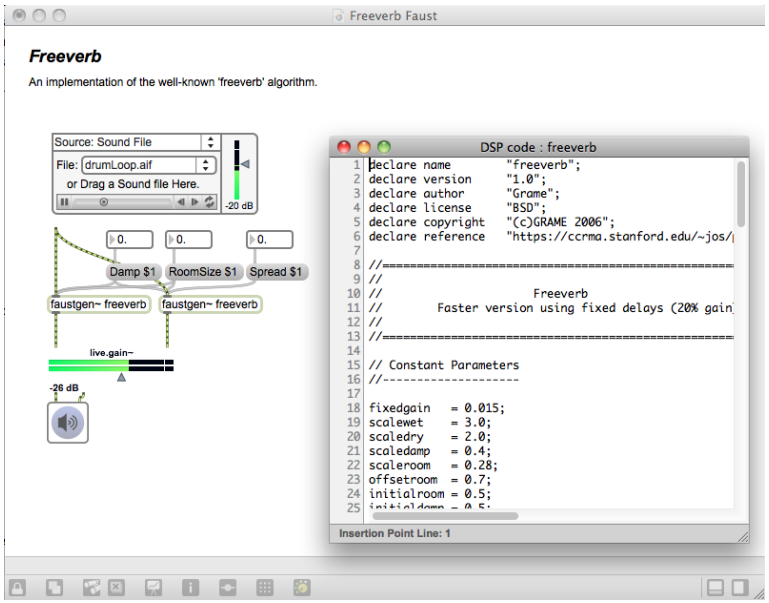
Embedded Faust Compilers

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





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

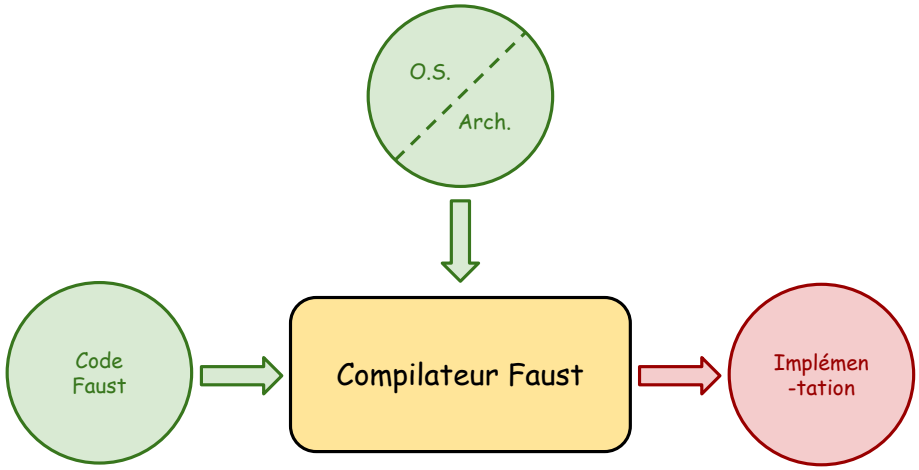
- Libfaust: embeddable version of the Faust compiler coupled with LLVM
- Libfaust.js: embeddable Javascript version of the Faust compiler

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. . .

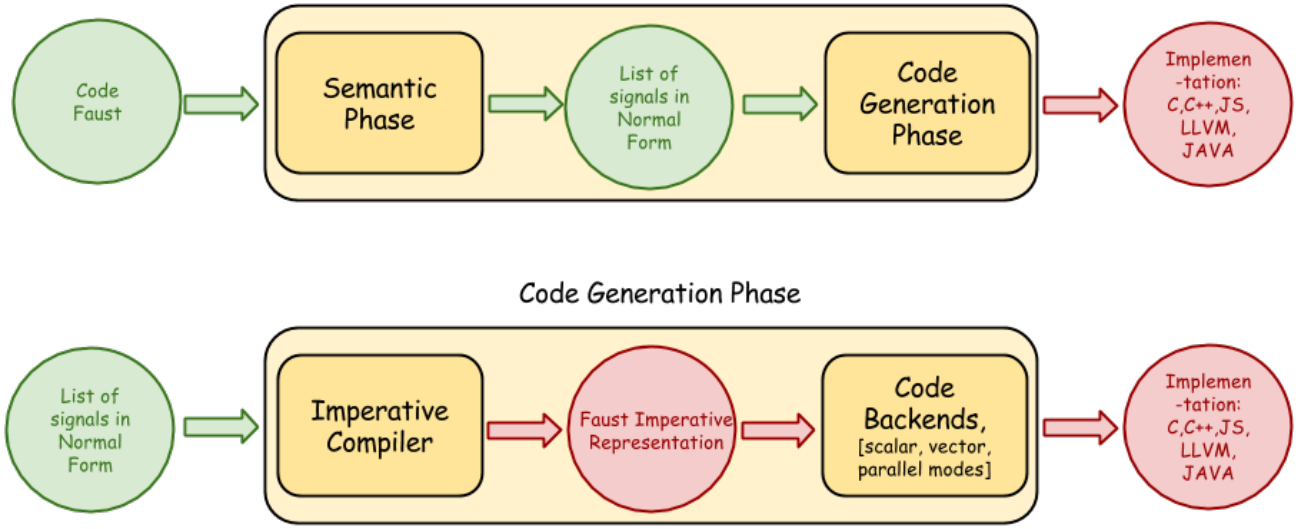


FAUST file name	STK	FAUST	Difference
blowBottle.dsp	3.23	2.49	22.91
blowHole.dsp	2.70	1.75	35.19
bowed.dsp	2.78	2.28	17.99
brass.dsp	10.15	2.01	80.20
clarinet.dsp	2.26	1.19	47.35
flutestk.dsp	2.16	1.13	47.69
saxophony.dsp	2.38	1.47	38.24
sitar.dsp	1.59	1.11	30.19
tibetanBowl.dsp	5.74	2.87	50

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.*

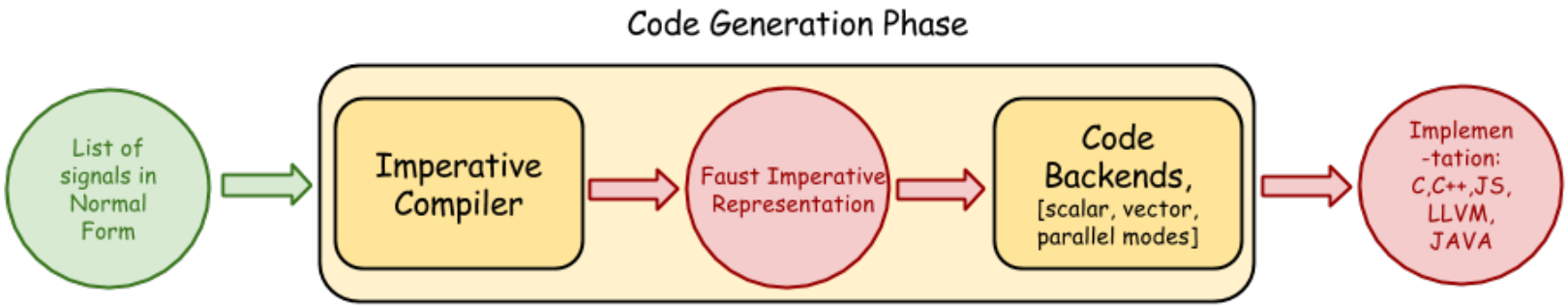
. . . for a wide range of audio architectures and plateforms

Structure of the Faust Compiler



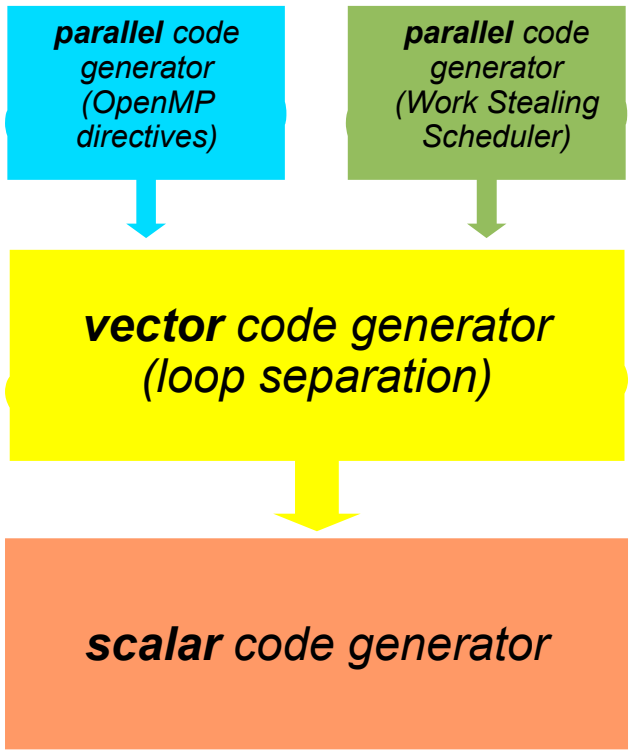
- *Semantic phase*: transforms the signal processor denoted by a Faust program into the list of signals expressions it computes.
- *Code generation*: generates the best possible implementation for this list of signal expressions

Code Generation Phase



Code Generation Phase

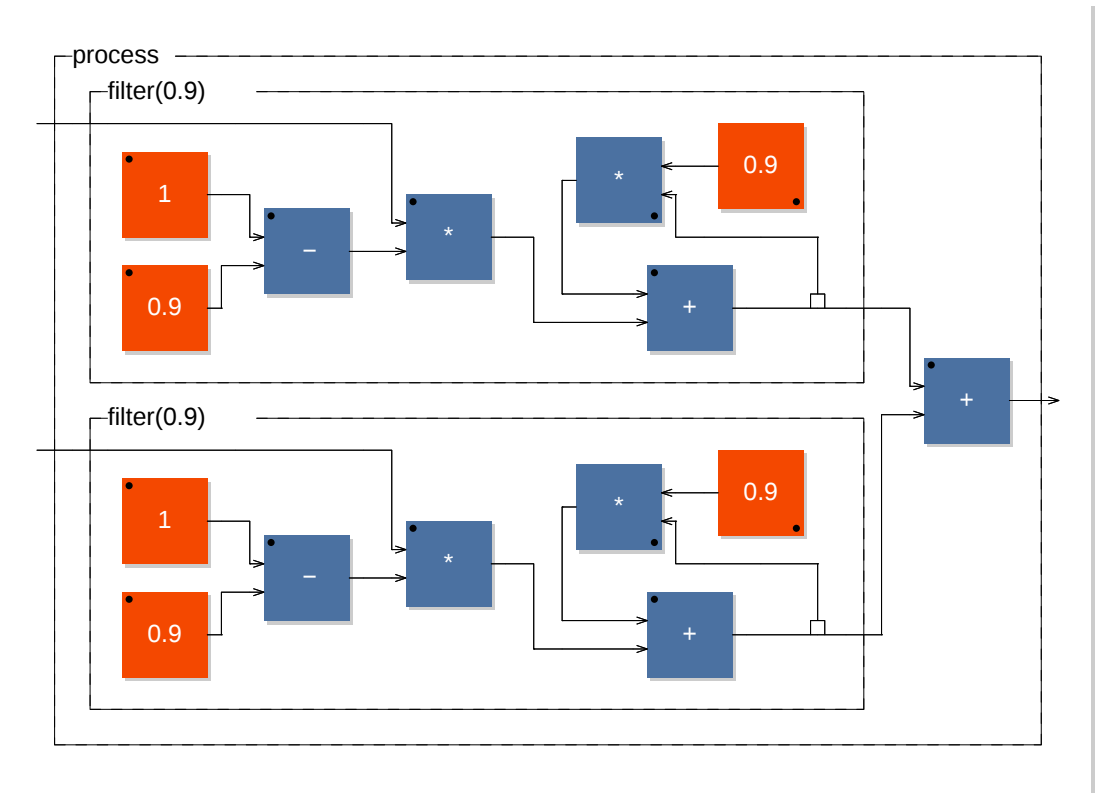
Four Code generation modes



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

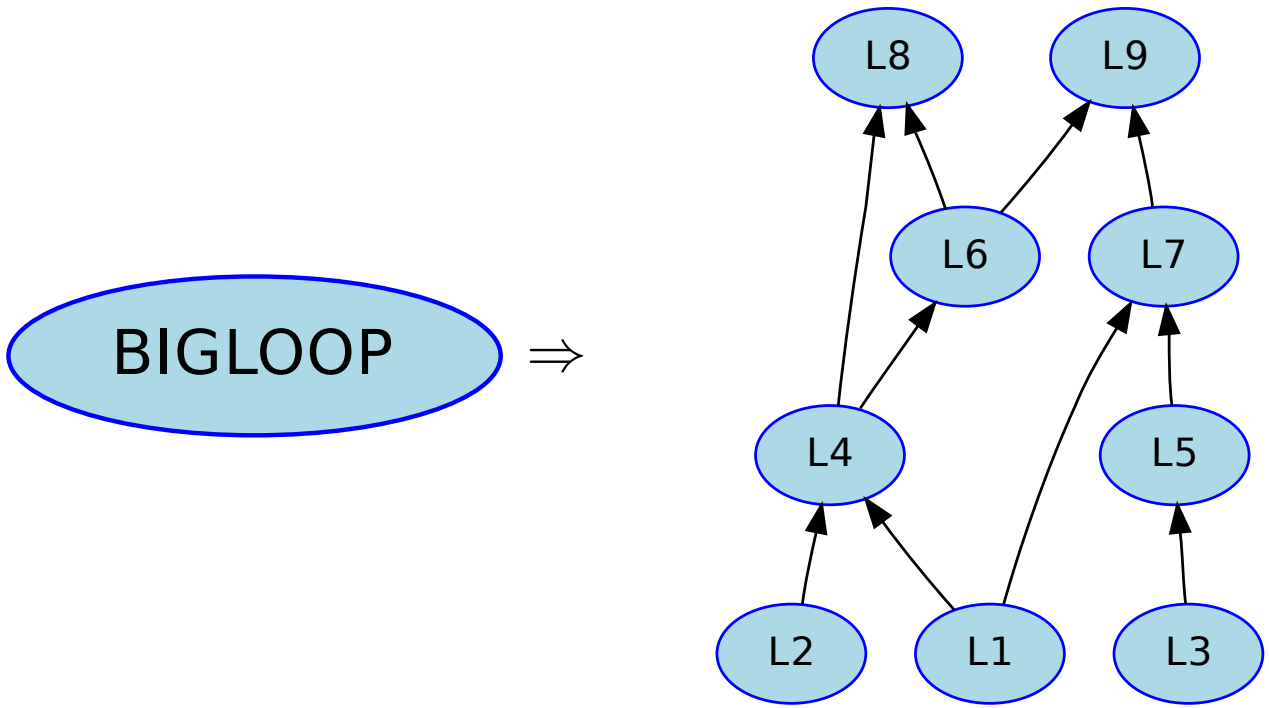
Scalar Code

```
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

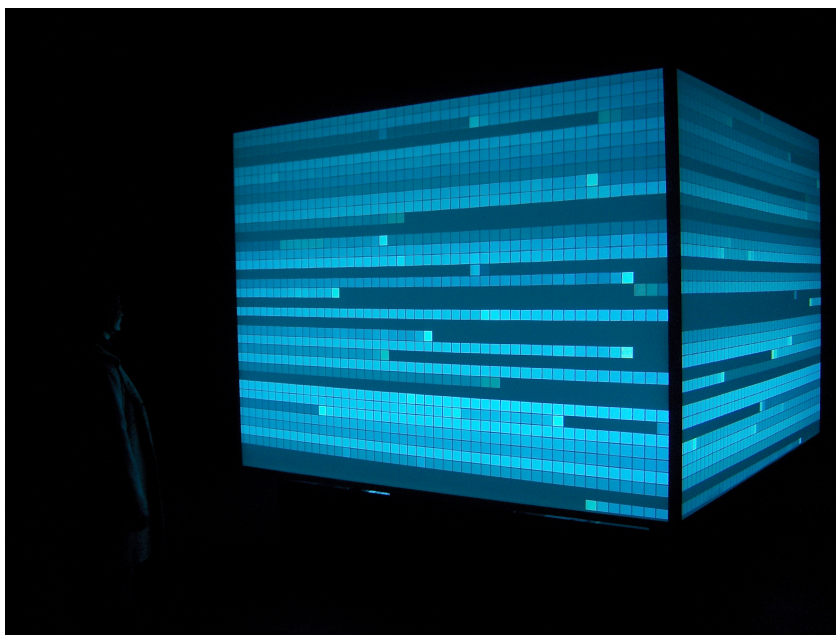
```
taskqueue.InitTaskList(task_list_size, task_list, fDynamicNumThreads,
                                                  cur_thread, tasknum);
while (!fIsFinished) {
  switch (tasknum) {
    case WORK_STEALING_INDEX: {
      tasknum = TaskQueue::GetNextTask(cur_thread, fDynamicNumThreads);
      break;
    }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;
    }
  }
};
```

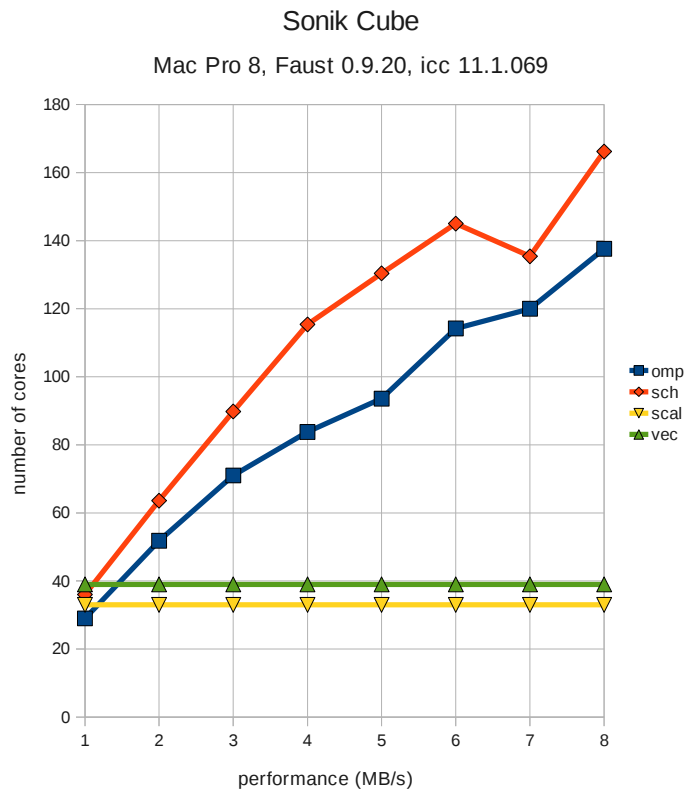
Code Generation Phase

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).

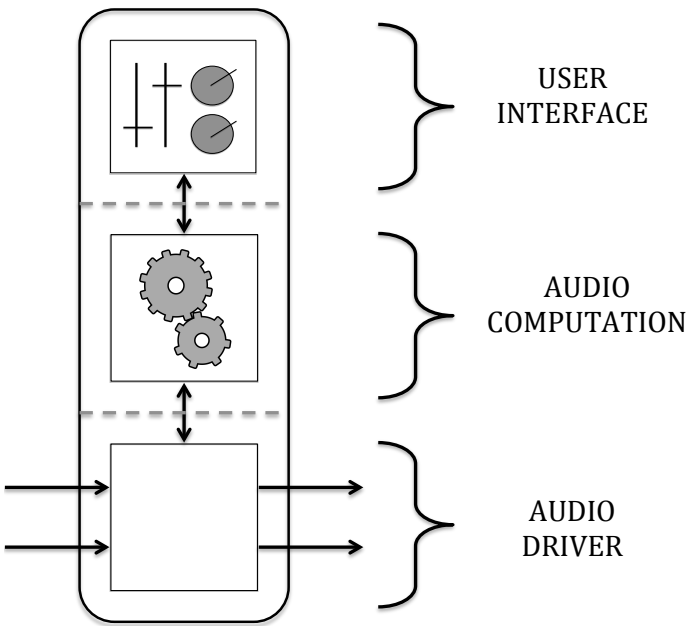


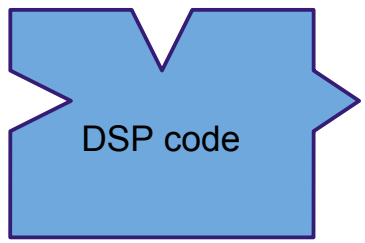


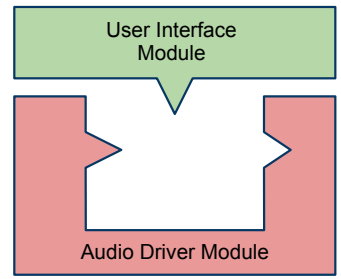
Architectures and Deployment

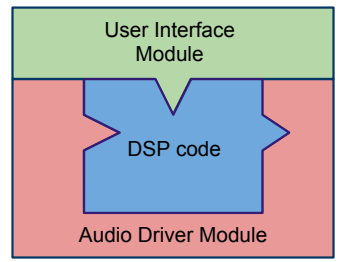
Separation of concern

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

$\rightarrow$ 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