FAUST : an Efficient Functional Approach to DSP Programming

Yann Orlarey, Dominique Fober and Stephane Letz

FAUST is a programming language that provides a purely functional approach to signal processing while offering a high level of performance. FAUST aims at being complementary to existing audio languages by offering a viable and efficient alternative to C/C++ to develop signal processing libraries, audio plug-ins or standalone applications.

The language is based on a simple and well defined formal semantics. A FAUST program denotes a *signal processor*, a mathematical function that transforms input signals into output signals. Being able to know precisely what a program computes is important not only for programmers, but also for compilers needing to generate the best possible code. Moreover these semantics questions are crucial for the long-term preservation of music programs [1].

The following paragraphs will give an overview of the language as well as a description of the compiler, including the generation of parallel code.

Introduction

From Music III [2] to Max [1], from MUSICOMP [3] to OpenMusic [2] and Elody [3], research in music programming languages has been very active since the early 60's. The Computer Music community has pioneered the field of End-User programming as well as the idea of using a programming language as a creative tool to invent complex objects like sounds, musical structures or realtime interactions. Nowadays the practice of Live Coding, with languages like ChucK [4], pushes even further the boundaries by positioning the computer programming activity as a Performing Art. Moreover, with the convergence of Digital Arts, visual programming languages like Max have gained a large audience well outside the Computer Music community. Within this context, FAUST's objective is to provide a purely functional approach to signal processing while offering the highest level of performance. FAUST aims at being complementary to existing audio languages

[1] Long-term preservation of music programs and patches is the question at the heart of the ASTREE project (ANR 08-CORD-003) in which FAUST will have a central role.

[2] MUSIC III developed by Max Mathews in 1959 at the Bell Labs is the first language for digital audio synthesis.

[3] MUSICOMP (Music Simulator Interpreter for Compositional Procedures.) developed by Lejaren Hiller and Robert Baker in 1963 is considered one of the very first music composition language

by providing a viable and efficient alternative to C/C++ to develop signal processing libraries, audio plug-ins or standalone applications.

Languages like Max and Puredata [5] are basically interpreters of *dataflow* inspired languages (see [6], [7], [8] or [9]). In order to achieve high perfomance compatible with the needs of real time audio applications, they minimize interpretation overhead by working on vectors of samples. The drawback of this approach is that sample level computations like recursive filters have to be provided as primitives or external plugins. Moreover, it makes Max or PD programs dependent on a runtime environment. Therefore embedding Max or PD programs in other systems, while not impossible, is less convenient than say DSP code written in C/C++.

Another problem is the fact that, due to their generality, the semantics of dataflow models can be quite complex, depending on many technical choices like: synchronous or asynchronous computations, deterministic or non-deterministic behavior, bounded or unbounded communication FIFOs, firing rules, etc. Because of this complexity, the vast majority of dataflow inspired music languages have no *explicit* formal semantic. The semantic is hidden in the dataflow engine and the global behavior of a block-diagram can be difficult to understand without a good knowledge of the implementation. This is obviously a problem in terms of heritage and long term preservation of the musical pieces that relie on these languages.

FAUST [10] [11] tries to address these problems. First of all FAUST is a compiled language. The FAUST compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The result can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers.

The generated code works at the sample level. It is therefore suited to implement low-level DSP functions like recursive filters. Moreover the code can be easily embedded. It is self-contained and doesn't depend of any DSP library or runtime system. It has a very deterministic behaviour and a constant memory footprint.

The semantic of FAUST is simple and well defined. This is not just of academic interest. It allows the FAUST compiler to be *semantically driven*. Instead of compiling a program literally, it compiles the mathematical function it denotes. This feature is useful for example to promote components reuse while preserving optimal perfomance. Moreover having access to the exact semantics of a FAUST program can simplify preservation issues. It will allow future versions of the compiler to generate automatic documentation with all the mathematical equations involved.

FAUST is a textual language but nevertheless block-diagram oriented. It actually combines two approaches: functional programming and algebraic block-diagrams. The key idea is to view block-diagram construction as function composition. For that, FAUST relies on a *block-diagram algebra* of five composition operations.

The next section will give an overview of FAUST's syntax and principles via some very simple DSP examples. Section 3 will give a detailled account of the compiler and describe the code generation strategies it implements. Section 4 will provide some performance benchmarks comparing these strategies. We will conclude with some perspectives on future evolutions of FAUST.

Overview

As said in the introduction, the FAUST programming model combines a *functional programming approach* with a *block-diagram syntax*:

- The functional programming approach [12] provides a natural framework for signal processing. Digital signals are modeled as discrete functions of time, signal processors as second order functions that operate on them, and FAUST's block-diagram *composition operators*, used to combine signal processors together, as third order functions, etc. .
- Block-diagrams, even if purely textual like in FAUST, promote a modular approach of signal processing that suits very well the sound engineers' and audio developers' habits, while providing a powerful and expressive syntax.

A FAUST program doesn't describe a sound or a group of sounds, but a *signal processor*, something that transforms input signals and produces output signals. The program source is organized as a set of *definitions* with at least the definition of the keyword `process` (the equivalent of `main` in C):

```
process = ...;
```

Some very simple examples

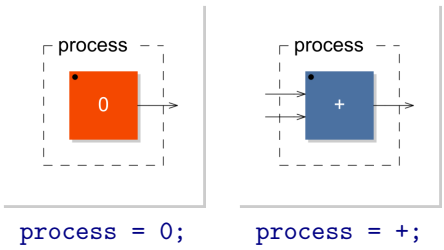


Figure 1. Some very simple FAUST programs

Let's start with some really simple one-line examples of FAUST program. Here is a first example (Figure 1) that produces silence:

```
process = 0;
```

The second example is a little bit more sophisticated and copies the input signal to the output signal. It involves the `_` (underscore) primitive that denotes the identity function on signals (that is a simple audio cable for a sound engineer):

```
process = _;
```

Another very simple example is the conversion of a two-channel stereo signal into a one-channel mono signal using the `+` primitive that adds two signals together:

```
process = +;
```

Most FAUST primitives are analogue to their C counterpart on numbers, but lifted to signals. For example the FAUST primitive `sin` operates on a signal X by applying the C function *sin* to each sample $X(t)$ of X. In other words `sin` transforms an input signal X into an output signal Y such that $Y(t) = sin(X(t))$. All C numerical functions have their counterpart in FAUST.

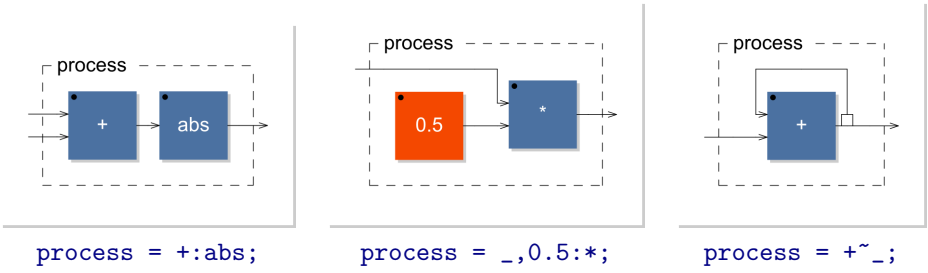
Some signal processing primitives are specific to FAUST. For example the delay operator `@` takes two input signals: X (the signal to be delayed) and D (the delay to be applied), and produces an output signal Y such that $Y(t) = X(t - D(t))$.

Block-Diagram Composition

Contrary to Max like visual programming languages where the user does manual connections, FAUST primitives are assembled in block-diagrams by using a set of high-level block-diagram composition operations (see Table 1). You can think of these composition operators as a generalization of the mathematical function composition operator $\circ$.

f`~`g	recursive composition (precedence 4)
f`,`g	parallel composition (precedence 3)
f`:`g	sequential composition (precedence 2)
f`<:`g	split composition (precedence 1)
f`:>`g	merge composition (precedence 1)

Table 1. The block-diagram *composition operators* used in FAUST



`process = +:abs;` `process = _,0.5:*;` `process = +~_;`

Figure 2. Simple examples of block-diagram composition

Let's say that we want to connect the output of `+` to the input of `abs` in order to compute the absolute value of the output signal (see Figure 2). This connection can be done using the *sequential composition* operator '`:`' (colon):

```
process = + : abs;
```

Here is an example of *parallel composition* (a stereo cable) using the operator '`,`' that puts in parallel its left and right expressions:

```
process = _,_;
```

These operators can be arbitrarily combined. For example to multiply the input signal by 0.5 one can write:

```
process = _,0.5 : *;
```

Taking advantage of some syntactic sugar the above example can be rewritten (using what functional programmmers know as curryfication):

```
process = *(0.5);
```

The recursive composition operator '~' can be used to create block-diagrams with cycles (that include an implicit one-sample delay). Here is the example of an integrator that takes an input signal X and computes an output signal Y such that $Y(t) = X(t) + Y(t-1)$:

```
process = + ~ _;
```

A Noise Generator

Here is a more involved 3-lines program representing a noise generator.

```
random  = +(12345) ~ *(1103515245);
noise   = random/2147483647.0;
process = noise*vslider("noise[style:knob]",0,0,100,0.1)/100;
```

Listing 1. noise.dsp

The first definition describes a (pseudo) random number generator that produces a signal R such that $R(t) = 12345 + 1103515245 * R(t-1)$. The expression `+(12345)` denotes the operation of adding `12345` to a signal. In the same way `*(1103515245)` denotes the multiplication of a signal by `1103515245`.

The two resulting operations are *recursively composed* using the `~` operator. This operator connects in a feedback loop the output of `+(12345)` to the input of `*(1103515245)` (with an implicit 1-sample delay) and the output of `*(1103515245)` to the input of `+(12345)` (see the corresponding block-diagram Figure 6)

The second definition transforms the random signal into a noise signal by scaling it between -1.0 and +1.0.

Finally, the definition of `process` adds a simple user interface to control the production of the sound. The noise signal is multiplied by the value delivered by a slider to control its volume.

This is obviously a very simplified example of sound synthesis. For more sophisticated descriptions of virtual instruments using FAUST we refer the reader to [13], while for a general introduction to Digital Filters with some examples in FAUST see [14].

Invoking the Compiler

We can compile our noise generator example using the following command (see Table 2):

```
  $ faust noise.dsp
```

Since we have not defined any output file, the corresponding C++ code is generated on the standard output:

```
class mydsp : public dsp {
 private:
  int   iRec0[2];
  float  fslider0;
```

short	*long*	*description*
`-a` *file*		C++ architecture file
`-o` *file*		C++ output file
`-h`	`--help`	print the help message
`-v`	`--version`	print version information
`-svg`	`--svg`	generate block-diagram svg files
`-xml`	`--xml`	generate an additional description file in xml format
`-vec`	`--vectorize`	generate easier to vectorize code
`-vs` *n*	`--vec-size` *n*	size of the vector (default 32 samples)
`-omp`	`--openMP`	generate openMP pragmas, activates –vectorize option

Table 2. Faust main compiler options

```
public:
 static void metadata(Meta* m)  {
 }

 virtual int getNumInputs() { return 0; }
 virtual int getNumOutputs() { return 1; }
 static void classInit(int samplingFreq) {
 }
 virtual void instanceInit(int samplingFreq)
 {
   fSamplingFreq = samplingFreq;
   for (int i=0; i<2; i++) iRec0[i] = 0;
   fslider0 = 0.0f;
 }
 virtual void init(int samplingFreq)
 {
   classInit(samplingFreq);
   instanceInit(samplingFreq);
 }
 virtual void buildUserInterface(UI* interface)
 {
   interface->openVerticalBox("noise");
   interface->declare(&fslider0,"style","knob");
   interface->addVerticalSlider("noise",
         &fslider0, 0.0f, 0.0f, 100.0f, 0.1f);
   interface->closeBox();
 }
 virtual void compute (int count, float** input,
                                  float** output)
 {
   float  fSlow0 = (4.656613e-12f * fslider0);
   float* output0 = output[0];
```

```
    for (int i=0; i<count; i++) {
      iRec0[0] = 12345+1103515245*iRec0[1];
      output0[i] = fSlow0*iRec0[0];
      // post processing
      iRec0[1] = iRec0[0];
    }
  }
};
```

Listing 2. C++ code of the noise generator

The generated class contains eight methods. Among these methods `getNumInputs()` and `getNumOutputs()` return the number of input and output signals required by our signal processor. `init()` initializes the internal state of the signal processor. The method `buildUserInterface()` can be seen as a list of high level commands, independent of any toolkit, to build the user interface. The method `compute()` does the actual signal processing. It takes 3 arguments: the number of frames to compute, the addresses of the input buffers and the addresses of the output buffers, and computes the output samples according to the input samples.

Generating a Full Application

Thanks to specific *architecture files*, a single FAUST program can be used to produce code for a variety of platforms and plug-in formats. These architecture files act as wrappers and describe the interactions with the host audio and GUI system. Currently more than 10 architectures are supported (see Table 3) and new ones can be easily added. For the interested reader, the interfaces with Supercollider and Puredata are described in details in [15] and [16].

alsa-gtk.cpp	ALSA application + GTK
alsa-qt.cpp	ALSA application + QT4
jack-gtk.cpp	JACK application + GTK
jack-qt.cpp	JACK application + QT4
ca-qt.cpp	CoreAudio application + QT4
ladspa.cpp	LADSPA plug-in
max-msp.cpp	Max MSP plug-in
pd.cpp	Puredata plug-in
q.cpp	Q language plug-in
supercollider.cpp	Supercollider plug-in
vst.cpp	VST plug-in

Table 3. Some architecture files available for FAUST

The `faust` command accepts several options to control the generated code. Two of them are widely used. The option `-o outputfile` specifies the output file to be used instead of the standard output. The option `-a architecturefile` defines the architecture file used to wrap the generated C++ class.

For example the command `faust -a jack-qt.cpp -o noise.cpp noise.dsp` generates a full jack application using QT4.4 as a graphic toolkit. Figure 3 is a screenshot of our noise application running.

Figure 3. Screenshot of the noise example generated with the `jack-qt.cpp` architecture

Describing the User Interface

A very convenient aspect of Max is how easy it is to build a graphical user interface. For that purpose FAUST integrates powerful means to describe a user interface in a very abstract manner, independently of any GUI toolkit. The actual implementation is the responsibility of each architecture file. Implementations are free to ignore some details. For example a command line interface or an OSC interface will ignore all the graphical aspects and only retain the names and ranges of the parameters, while a QT interface will take great care of the visual details as in Figure 4.

Figure 4. Screenshot of mixer.dsp using the jack-qt architecture

A FAUST interface involves active widgets: *buttons*, *checkboxes*, *sliders* and *numerical entries*; passive widgets: *bargraphs*; as well as *vertical*, *horizontal* and *tabular* grouping schemes (see table 4).

All these widgets produce signals and can be freely mixed with other elements of the language. A button for example (see Figure 5) produces a signal that is 1 when the button is pressed and 0 otherwise. A slider produces a signal that varies between two values according to the user's actions.

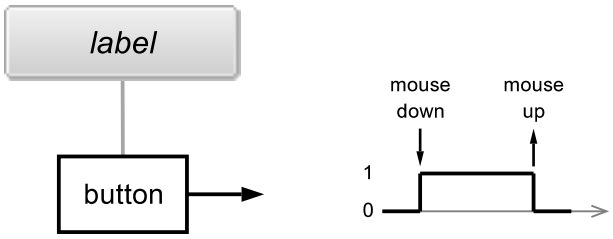


Figure 5. User Interface Button

Syntax	Example
`button(`*str*`)`	`button("play")`
`checkbox(`*str*`)`	`checkbox("mute")`
`vslider(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`vslider("vol",50,0,100,1)`
`hslider(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`hslider("vol",0.5,0,1,0.01)`
`nentry(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`nentry("freq",440,0,8000,1)`
`vgroup(`*str*`,`*block-diagram*`)`	`vgroup("reverb", ...)`
`hgroup(`*str*`,`*block-diagram*`)`	`hgroup("mixer", ...)`
`tgroup(`*str*`,`*block-diagram*`)`	`vgroup("parametric", ...)`
`vbargraph(`*str*`,`*min*`,`*max*`)`	`vbargraph("input",0,100)`
`hbargraph(`*str*`,`*min*`,`*max*`)`	`hbargraph("signal",0,1.0)`

Table 4. FAUST's User Interface widgets and groups

Generating a Block-Diagram

Another useful option is `-svg`. It generates the block diagram representation of the program as one or more SVG graphic files like in Figure 6.

It is interesting to note the difference between the block diagram and the generated C++ code. The block diagram Figure 6 involves one addition, two multiplications and two divisions. The generated C++ program only involves one addition and two multiplications per samples. The compiler was able to optimize the code by factorizing and reorganizing the operations.

As already said, the key idea here is not to compile the block diagram literally, but the mathematical function it denotes. Modern C/C++ compilers too don't compile programs literally. But because of the complex semantic of C/C++ (due to side effects, pointer aliasing, etc.) they can't go very far in that direction. This is a distinctive advantage of a purely functional language: it allows compilers to do very advanced optimisations. The next section will explain how these optimizations are done.

Code Generation

Starting with version 0.9.9.5, the FAUST compiler implements several different strategies to generate C++ code. We will first introduce the so called *scalar* generation of code organized around a single sample computation loop. Then we will present the *vector* generation of code where the code is organized into several loops that operate on vectors,

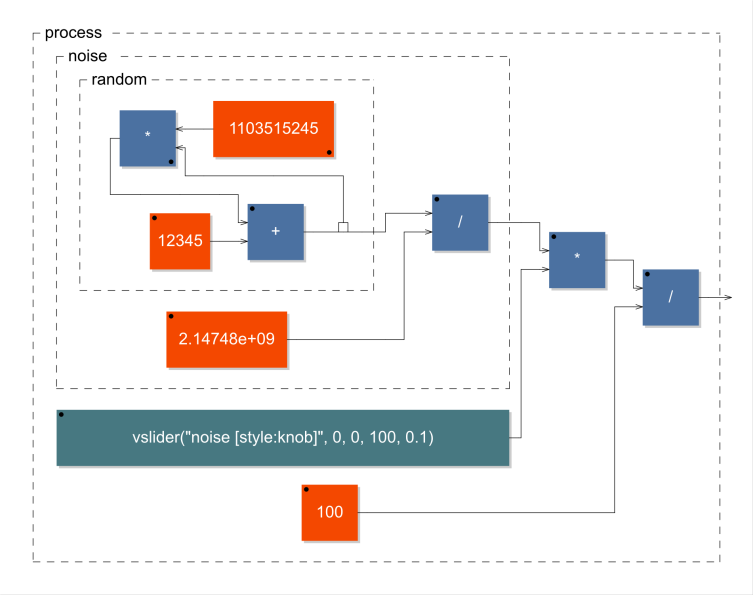


Figure 6. Graphic block-diagram of the noise generator produced with the `-svg` option

and finally the *parallel* generation of code where these vector loops are parallelized using OpenMP directives.

Preliminary Steps

Before reaching the stage of the C++ code generation, the FAUST compiler has to carry on several steps that we describe briefly here.

Parsing Source Files

The first one is to recursively parse all the source files involved. Each source file contains a set of definitions and possibly some *import* directives for other source files. The result of this phase is a list of definitions: $[(name_1 = definition_1), (name_2 = definition_2), \ldots]$. This list is actually a set, as redefinitions of symbols are not allowed.

Evaluating Block-Diagrams

Among the names defined, there must be *process*, the analog of main in C/C++. This definition has to be evaluated as FAUST allows algorithmic block-diagram definitions.

For example the algorithmic definition:

```
foo(n) = *(10+n);
process = par(i,3, foo(i));
```

Listing 3. Example of algorithmic definition

will be translated in a *flat* block-diagram description that contains only primitive blocks:

```
process = (_,10:*),(_,11:*),(_,12:*);
```

This description is said to be in *normal form*.

Discovering the Mathematical Equations

FAUST uses a phase of symbolic propagation to first discover the semantics (the mathematical function it denotes) of a block-diagram. The principle is to propagate symbolic signals through the inputs of the block-diagram in order to get, at the other end, the mathematical equation of each output signal.

These equations are then normalized so that different block-diagrams that compute mathematically equivalent signals result in the same output equations.

Here is a very simple example where the input signal is divided by 2 and then delayed by 10 samples:

```
process =  /(2) : @(10);
```

This is equivalent to having the input signal first multiplied by 2, then delayed by 7 samples, then divided by 4 and then delayed by 3 samples.

```
process = *(2) : @(7) : /(4):  @(3);
```

Both lead to the following signal equation:

$$Y(t) = 0.5 * X(t - 10)$$

FAUST applies several rules to simplify and normalize output signal equations. For example, one of theses rules says that it is better to multiply a signal by a constant after a delay than before. It gives the compiler more opportunities to share and reuse the same delay line. Another rule says that two consecutive delays can be combined into a single one.

Assigning Types

The next phase is to assign *types* to the resulting signal equations. This will not only help the compiler to detect errors but also to generate the most efficient code. Several aspects are considered:

1. the *nature* of the signal: *integer* of *float*.
2. *interval of values* of the signal: the minimum and maximum values that a signal can take
3. the *computation time* of the signal: the signal can be computed at *compilation time*, at *initialization time* or at *execution time*.
4. the *speed* of the signal: *constant* signals are computed only once, *low speed* user interface signals are computed once for every block of samples, *high speed* audio signals are computed every sample.[4]

[4]This distinction is purely internal to the compiler and only for optimisation purposes. From the user point of view everything is a full-rate audio signal. He doesn't have, like in CSOUND for example, to separate a-rate and k-rate signals.

5. parallelism of the signal: *true* if the samples of the signal can be computed in parallel, *false* when the signal has recursive dependencies requiring its samples to be computed sequentially.

Occurrence Analysis

The role of this last preparation phase is to analyze in which context each subexpression is used and to discover common subexpressions. If an expensive common subexpression is discovered, an assignment to a *cache variable* `float fTemp = <subexpression code>;` is generated, and the cache variable `fTemp` is used in its enclosing expressions. Otherwise the subexpression code is in-lined.

The occurrence analysis proceeds by a top-down visit of the signal expression. The first time a subexpression is visited, it is annotated with a counter. The next time the counter will be increased and its visit skipped.

Subexpressions with several occurrences are candidates to be cached in variables. However, in some circumstances expressions with a single occurrence need also be cached if they occur in a faster context. For example a constant expression occurring in a low speed user interface expression or a user interface expression occurring in a high speed audio expression will generally be cached.

It is only after this phase that the generation of the C++ code can start.

Scalar Code Generation

The generation of the C++ code is made by populating a *klass* object (representing a C++ class), with C++ declarations and lines of code. In *scalar* mode these lines of code are organized in a single sample computation loop, while they can be splitted in several loops with the new *vector* and *parallel* schemes (see below).

The code generation relies basically on two functions: a translation function $[\![\ ]\!]$ that translate a signal expression into C++ code, and a cache function $\mathbf{C}()$ that checks if a variable is needed.

We do not have the space to go in too much details, but here is the translation rule for the addition of two signal expressions:

$$\frac{\begin{array}{c}[\![E_1]\!] \rightarrow \mathtt{S}_1 \\ [\![E_2]\!] \rightarrow \mathtt{S}_2\end{array}}{[\![E_1 + E_2]\!] \rightarrow \mathbf{C}(''(\mathtt{S}_1 + \mathtt{S}_2)'')}$$

It says that to compile the addition of two signals we compile each of these signals and concat the resulting codes with a + sign in between. The code obtained is passed to the cache function that will check if the expression is shared or not.

Let's say that the code passed to the cache function $\mathbf{C}()$ is the C++ expression `(input0[i] + input1[i])`. If the expression is shared, the cache function will:

- allocate a fresh variable name `fTemp0`;
- add the line of code `float fTemp0 = (input0[i] + input1[i]);` to the *klass* object;
- return `fTemp0` to be used when compiling enclosing expressions.

If the expression is not shared it will simply return `(input0[i] + input1[i])` unmodified.

To illustrate this, let's take two simple examples. The first one converts a stereo signal into a mono signal by adding the two input signals:

```
process = +;
```

In this case `(input0[i] + input1[i])` is not shared and the generated C++ code is the following:

```
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++) {
    output0[i] = (input0[i] + input1[i]);
  }
}
```

But when the sum of the two input signals is duplicated on two output signals as in:

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

then `(input0[i] + input1[i])` will be cached in a temporary variable:

```
virtual void compute (int count,
                      float** input,
                      float** output)
{
  float* input0 = input[0];
  float* input1 = input[1];
  float* output0 = output[0];
  float* output1 = output[1];
  for (int i=0; i<count; i++) {
    float fTemp0 = (input0[i] + input1[i]);
    output0[i] = fTemp0;
    output1[i] = fTemp0;
  }
}
```

Vector Code Generation

Modern C++ compilers are able to do autovectorization, that is to use SIMD instructions to speedup the code. These instructions can typically operate in parallel on short vectors of 4 simple precision floating point numbers thus leading to a theoretical speedup of 4. Autovectorization of C/C+ programs is a difficult task. Current compilers are very sensitive to the way the code is arranged. In particular too complex loops can prevent autovectorization. The goal of the FAUST vector code generator is to rearrange the C++

code in a way that facilitates the autovectorization job of the C++ compiler. Instead of generating a single sample computation loop, it splits the computation into several simpler loops that communicate by vectors.

The vector code generation is activated by passing the `--vectorize` (or `-vec`) option to the FAUST compiler. Two additional options are available: `--vec-size <n>` controls the size of the vector (by default 32 samples) and `--loop-variant 0/1` gives some additional control on the loops.

To illustrate the difference between scalar code and vector code, let's take the computation of the RMS (Root Mean Square) value of a signal. Here is the FAUST code that computes the Root Mean Square of a sliding window of 1000 samples:

```
// Root Mean Square of n consecutive samples
RMS(n) = square : mean(n) : sqrt ;

// Square of a signal
square(x) = x * x ;

// Mean of n consecutive samples of a signal
// (uses fixpoint to avoid the accumulation of
// rounding errors with denormals)
mean(n) = float2fix : integrate(n) :
          fix2float : /(n);

// Sliding sum of n consecutive samples
integrate(n,x) = x - x@n : +~_ ;

// Conversion between float and fix point
float2fix(x) = int(x*(1<<20));
fix2float(x) = float(x)/(1<<20);

// Root Mean Square of 1000 consecutive samples
process = RMS(1000) ;
```

Listing 4. rms.dsp

The `compute()` method generated in scalar mode is the following:

```
virtual void compute (int count,
                      float** input,
                      float** output)
{
  float* input0 = input[0];
  float* output0 = output[0];
  for (int i=0; i<count; i++) {
    float fTemp0 = input0[i];
    int iTemp1 = int(1048576*fTemp0*fTemp0);
    iVec0[IOTA&1023] = iTemp1;
    iRec0[0] = ((iVec0[IOTA&1023] + iRec0[1])
                    - iVec0[(IOTA-1000)&1023]);
    output0[i] = sqrtf(9.536744e-10f *
```

```
                        float(iRec0[0]));
    // post processing
    iRec0[1] = iRec0[0];
    IOTA = IOTA+1;
  }
}
```

The -vec option leads to the following reorganization of the code:

```
virtual void compute (int fullcount,
                      float** input,
                      float** output)
{
  int     iRec0_tmp[32+4];
  int*    iRec0 = &iRec0_tmp[4];
  for (int index=0; index<fullcount; index+=32)
  {
    int count = min (32, fullcount-index);
    float* input0 = &input[0][index];
    float* output0 = &output[0][index];
    for (int i=0; i<4; i++)
      iRec0_tmp[i]=iRec0_perm[i];
    // SECTION : 1
    for (int i=0; i<count; i++) {
      iYec0[(iYec0_idx+i)&2047] =
               int(1048576*input0[i]*input0[i]);
    }
    // SECTION : 2
    for (int i=0; i<count; i++) {
      iRec0[i] = ((iYec0[i] + iRec0[i-1]) -
               iYec0[(iYec0_idx+i-1000)&2047]);
    }
    // SECTION : 3
    for (int i=0; i<count; i++) {
      output0[i] = sqrtf((9.536744e-10f *
                 float(iRec0[i])));
    }
    // SECTION : 4
    iYec0_idx = (iYec0_idx+count)&2047;
    for (int i=0; i<4; i++)
      iRec0_perm[i]=iRec0_tmp[count+i];
  }
}
```

While the second version of the code is more complex, it turns out to be much easier to vectorize efficiently by the C++ compiler. Using Intel icc 11.0, with the exact same compilation options (`-O3 -xHost -ftz -fno-alias -fp-model fast=2`), the scalar version leads to a throughput of 129.144 MB/s, while the vector version achieves 359.548 MB/s, a speedup of 2.8!

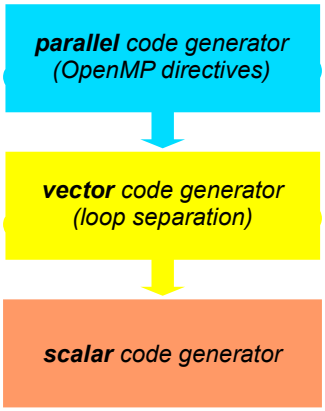


Figure 7. FAUST's stack of code generators

The vector code generation is built on top of the scalar code generation (see Figure 7). Every time an expression needs to be compiled, the compiler checks to see if it needs to be in a separate loop or not. It applies some simple rules for that. Expressions that are shared (and are complex enough) are good candidates to be compiled in a separate loop, as well as recursive expressions and expressions used in delay lines.

The result is a directed graph in which each node is a computation loop. This graph is stored in the klass object and a topological sort is applied to it before outputting the code.

Parallel Code Generation

The parallel code generation is activated by passing the `--openMP` (or `-omp`) option to the FAUST compiler. It implies the `-vec` options as the parallel code generation is built on top of the vector code generation by inserting appropriate OpenMP directives in the C++ code.

The OpenMP API

OpenMP (http://wwww.openmp.org) is a well established API that is used to explicitly define direct multi-threaded, shared memory parallelism. It is based on a fork-join model of parallelism (see Figure 8). Parallel regions are delimited by using the `#pragma omp parallel` construct. At the entrance of a parallel region a team of parallel threads is activated. The code within a parallel region is executed by each thread of the parallel team until the end of the region.

```
#pragma omp parallel
{
```

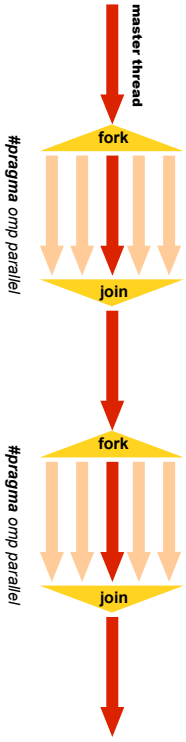


Figure 8. OpenMP is based on a fork-join model

```
  // the code here is executed simultaneously by
  // every thread of the parallel team
  ...
}
```

In order not to have every thread doing redundantly the exact same work, OpenMP provides specific *work-sharing* directives. For example `#pragma omp sections` allows to break the work into separate, discrete sections, each section being executed by one thread:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      // job 1
    }
    #pragma omp section
    {
      // job 2
```

```
    }
    ...
  }

  ...
}
```

Adding OpenMP Directives

As already said the parallel code generation is built on top of the vector code generation. The graph of loops produced by the vector code generator is topologically sorted in order to detect the loops that can be computed in parallel. The first set L_0 contains the loops that don't depend on any other loop; the set L_1 contains the loops that only depend of loops of L_0, etc..

As all the loops of a given set L_n can be computed in parallel, the compiler will generate a `sections` construct with a `section` for each loop.

```
#pragma omp sections
{
  #pragma omp section
  for (...) {
    // Loop 1
  }
  #pragma omp section
  for (...) {
    // Loop 2
  }
  ...
}
```

If a given set contains only one loop, then the compiler checks to see if the loop can be parallelized (no recursive dependencies) or not. If it can be parallelized, it generates:

```
#pragma omp for
for (...) {
 // Loop code
}
```

otherwise it generates a `single` construct so that only one thread will execute the loop:

```
#pragma omp single
for (...) {
 // Loop code
}
```

Example of Parallel Code

To illustrate how FAUST uses the OpenMP directives, here is a very simple example: two 1-pole filters in parallel connected to an adder (see Figure 9 for the corresponding block-diagram):

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

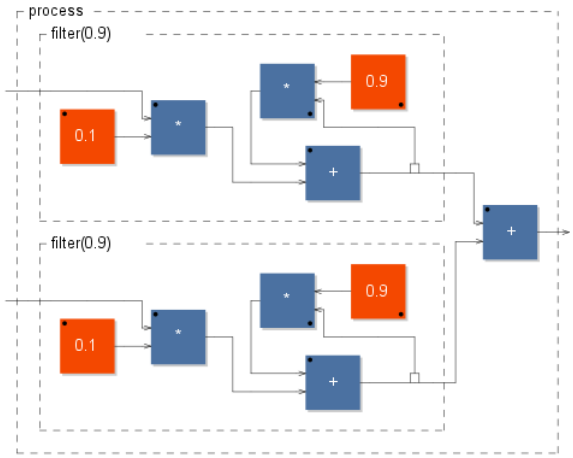


Figure 9. Two filters in parallel connected to an adder

The corresponding `compute()` method obtained using the `-omp` option is the following:

```
virtual void compute (int fullcount,
                      float** input,
                      float** output)
{
  float   fRec0_tmp[32+4];
  float   fRec1_tmp[32+4];
  float*  fRec0 = &fRec0_tmp[4];
  float*  fRec1 = &fRec1_tmp[4];
  #pragma omp parallel firstprivate(fRec0,fRec1)
  {
    for (int index = 0; index < fullcount;
                                index += 32)
    {
      int count = min (32, fullcount-index);
      float* input0 = &input[0][index];
      float* input1 = &input[1][index];
      float* output0 = &output[0][index];
      #pragma omp single
      {
        for (int i=0; i<4; i++)
          fRec0_tmp[i]=fRec0_perm[i];
        for (int i=0; i<4; i++)
          fRec1_tmp[i]=fRec1_perm[i];
```

```
      }
      // 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]);
      }
      // SECTION : 3
      #pragma omp single
      {
        for (int i=0; i<4; i++)
          fRec0_perm[i]=fRec0_tmp[count+i];
        for (int i=0; i<4; i++)
          fRec1_perm[i]=fRec1_tmp[count+i];
      }
    }
  }
}
```

This code requires some comments:

1. The parallel construct `#pragma omp parallel` is the fundamental construct that starts parallel execution. The number of parallel threads is generally the number of CPU cores but it can be controlled in several ways.

2. Variables external to the parallel region are shared by default. The OpenMP clause `firstprivate(fRec0,fRec1)` indicates that each thread should have its private copy of `fRec0` and `fRec1`. The reason is that accessing shared variables requires an indirection and is very inefficient compared to private copies.

3. The top level loop `for (int index = 0;...)...` is executed by all threads simultaneously. The subsequent work-sharing directives inside the loop will indicate how the work must be shared between the threads.

4. Note that an implied barrier exists at the end of each work-sharing region. All threads must have executed the barrier before any of them can continue.

5. The work-sharing directive `#pragma omp single` indicates that this first section will be executed by only one thread (any of them).

6. The work-sharing directive `#pragma omp sections` indicates that each corresponding `#pragma omp section`, here our two filters, will be executed in parallel.

7. The loop construct `#pragma omp for` specifies that the iterations of the associated loop will be executed in parallel. The iterations of the loop are distributed across the parallel threads. For example, if we have two threads, the first one can compute indices between 0 and $count/2 - 1$ and the other between $count/2$ and $count - 1$.

8. Finally `#pragma omp single` in section 3 indicates that this last section will be executed by only one thread (any of them).

Benchmarks

To compare the performances of these three types of code generation in a realistic situation, we implemented a special *alsa-gtk-bench.cpp* architecture file that measures the duration of the `compute()` method. Here is a fragment of this architecture file:

```
while(running) {
  audio.read();
  STARTMESURE
  DSP.compute(audio.buffering(),
              audio.inputSoftChannels(),
              audio.outputSoftChannels()
             );
  STOPMESURE
  audio.write();
  running = mesure <= (KMESURE + KSKIP);
}
```

Listing 5. alsa-gtk-bench.cpp (part)

The methodology is the following. The duration of the compute method is measured by reading the TSC (Time Stamp Counter) register. A total of 128+2048 measures are made by run. The first 128 measures are considered a warm-up period and are skipped. The median value of the following 2048 measures is computed. This median value, expressed in processor cycles, is first converted in a duration, and then in a number of bytes produced per second considering the audio buffer size (in our test, 2048) and the number of output channels.

This *throughput performance* is a good indicator. The memory bandwidth is a strong limiting factor for today's processors, and it has to be shared among the processors. In other words, on a SMP machine a realtime audio program can never go faster than the memory bandwidth. And if a sequential program already utilises all the available memory bandwidth, there is no room for improvement. In this case a parallel version can only perform worse.

Machines and Compilers Used

In order to compare the scalar code generation with the new vector and parallel code generation we compiled with FAUST 0.9.9.5b2 a series of tests in three different versions. The following commands were used:

- **scal**: `faust -a alsa-gtk-bench.cpp test.dsp -o test.cpp`
- **vec**: `faust -a alsa-gtk-bench.cpp -vec -vs 3968 test.dsp -o test.cpp`
- **par**: `faust -a alsa-gtk-bench.cpp -omp -vs 3968 test.dsp -o test.cpp`

We also used two different C++ compilers, GNU GCC and Intel ICC:

- GCC ver 4.3.2 with options: `-O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize` (`-fopenmp` added for OpenMP).
- ICC ver 11.0.074 with options: `-O3 -xHost -ftz -fno-alias -fp-model fast=2` (`-openmp` is added for OpenMP).

All the tests were run on three different machines:

- **vaio**: a Sony Vaio SZ3VP laptop, with an Intel T7400 dual core processor at 2167 MHz, 2GB of Ram, running an Ubuntu 7.10 distribution with a 2.6.22-15-generic kernel.
- **xps**: a Dell XPS machine with an Intel Q9300 quad core processor at 2500 MHz, 4GB of Ram, running an Ubuntu 8.10 distribution with a 2.6.22-15-generic kernel.
- **macpro**: an Apple Macpro with two Intel Xeon X5365 quad core processors at 3000 MHz, 2GB of Ram, running an Ubuntu 8.10 distribution with a 2.6.27-12-generic kernel

Benchmark: copy1.dsp

The goal of this first test is to measure the memory bandwidth. We use a very simple FAUST program copy1.dsp that simply copies the input signal to the output signal:

```
process = _;
```

The results we obtained are summarized in figure 10. The horizontal axis corresponds to the three faust compilation schemes: *scalar* , *vector* and *parallel*, combined with the two C++ compilers: *gcc* and *icc*. The vertical axis is the throughput: how many bytes of samples each tested program is able to produce per second (higher values are better).

It is interesting to note how catastrophic the performances of the parallel versions are. The scalar and vector versions are quite similar, with a little advantage to the scalar version. The code generated by icc performs better. The memory bandwidth of the Macpro is disappointing, especially considering that it has to be shared by 8 cores.

How stable are these measures? Figure 11 compares the performances of copy1 (compiled with icc) on the Macpro on 5 different runs. As we can see the stability is reasonably good.

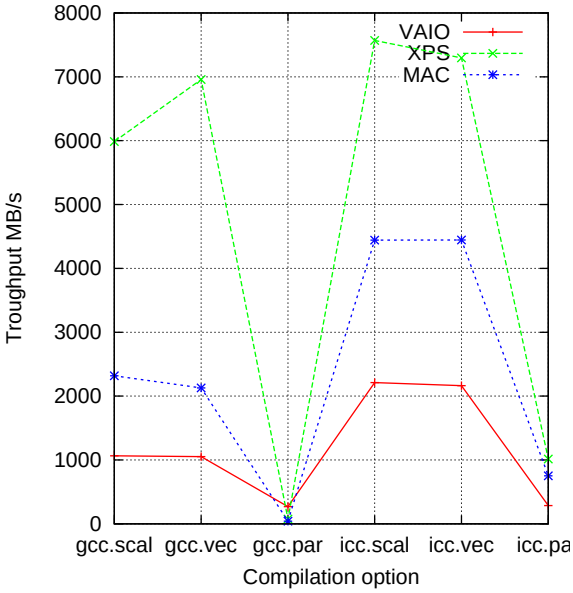


Figure 10. Copy1.dsp benchmark

Benchmark: freeverb.dsp

The second test is freeverb.dsp, a FAUST implementation of the Freeverb (the source can be found in the FAUST distribution).

The results are given figure 12. Here gcc gives very good results in scalar code and outperforms icc in 2 of the 3 cases. But the performances of gcc are still very poor on vector and parallel code.

Despite the fact that freeverb has a limited amount of parallelism, icc gives quite convincing results with a reasonable speedup on vector and parallel code on the Vaio and the XPS machines. It is also interesting to note that on parallel versions the 8 3GHz cores of the macpro were slower than 4 2.5Ghz cores of the XPS !

Benchmark: karplus32.dsp

Karplus32.dsp is a generalized version of Karplus-Strong algorithm with 32 slightly detuned strings in parallel (the source can be found in the FAUST distribution). Figure 13 gives the results. Again excellent performances of gcc in scalar mode. Good performance progression in vector mode as well as in parallel mode is obtained for icc.

Benchmark: mixer.dsp

This is the implementation of a simple 8 channels mixer. Each channel has a mute button, a volume control in dB, a vumeter and a stereo pan control. The mixer has also a volume control of the stereo output.

```
import("music.lib");
```

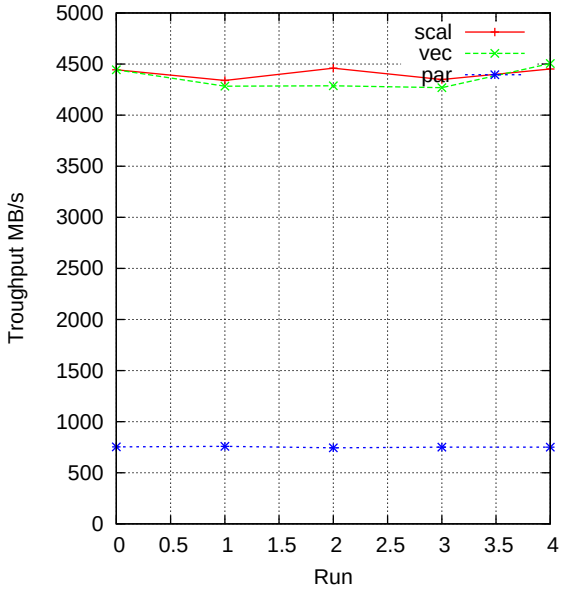


Figure 11. Stability of measures (copy1 on macpro, icc version)

```
smooth(c) = *(1-c) : +~*(c);

vol   = *( vslider("fader", 0, -60, 4, 0.1)
         : db2linear : smooth(0.99) );

mute  = *(1 - checkbox("mute"));

vumeter(x) = attach(x, env(x) :
                    vbargraph("",0,1))
  with {
       env = abs:min(0.99):max ~ -(1.0/SR);
      };

pan     = _ <: *(sqrt(1-c)), *(sqrt(c))
  with {
       c = ( nentry("pan",0,-8,8,1)-8)/-16 :
           smooth(0.99 );
      };

voice(v)  = vgroup("voice␣%v",
            mute :
            hgroup("", vol : vumeter) :
            pan );
```

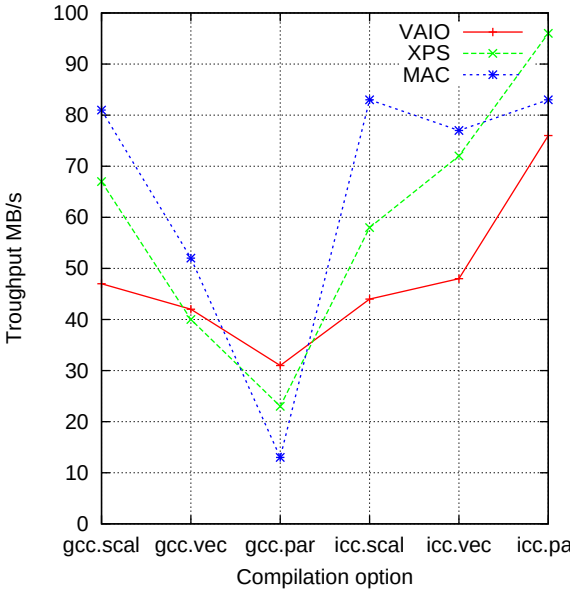


Figure 12. Freeverb.dsp benchmark

```
stereo     = hgroup("stereo␣out", vol, vol);

process    = hgroup("mixer",
                par(i,8,voice(i)) :> stereo);
```

Listing 6. mixer.dsp

The results of figure 14 show a real benefit for the vectorized version with a speedup exceeding 2 on the 3 machines. There is also a positive impact of the parallelization, even if more limited. As usual, gcc delivers good scalar code but poor results on vectorized and OpenMP code.

Benchmark: fdelay8.dsp

This test implements an 8-channel fractional delay. Each channel has a volume control in dB as well as a delay control in fractions of samples. The interpolation is based on a fifth-order Lagrange interpolation from Julius Smith's FAUST filter library.

```
import("filter.lib");

line(i) =  vgroup("line␣%i",fdelay5(128,d):*(g))
  with{ g = vslider("gain␣(dB)",-60,-60,4,0.1)
          : db2linear : smooth(0.995);
        d = nentry("delay␣(samp)",10,10,128,0.1)
          : smooth(0.995);
      };
```

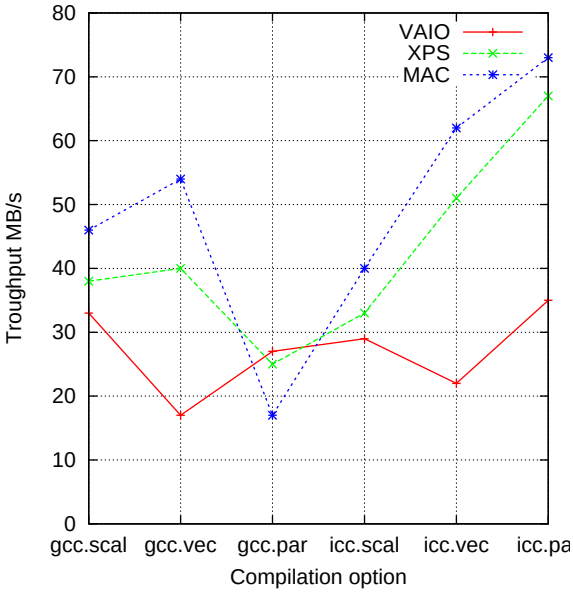


Figure 13. Karplus32.dsp benchmark

```
process = hgroup("", par(i, 8, line(i)) );
```

Listing 7. fdelay8.dsp

The results are presented figure 15. The Macpro exhibits a good speedup of 2.5 for its parallel version. The parallel speedup for the XPS machine is more limited and there is no speedup at all on the Vaio.

Benchmark: rms.dsp

The FAUST source of rms.dsp was presented listing 4. It is a purely sequential algorithm therefore the performances of the parallel versions are very bad. But, as figure 16 indicates, the vectorisation gives a real boost to the performances, particularly on the vaio.

Benchmark: rms8.dsp

This test computes the RMS value on 8 channels in parallel. The FAUST code is:

```
process = par(i,8,component("rms.dsp")) ;
```

Listing 8. rms8.dsp

We have obviously a good amount of parallelism here that icc is able to exploit as indicated by the results figure 17. Compared to the scalar performances, the parallel version exhibits a speedup of nearly 3 on the Mac, while the speedup for the XPS exceeds 2.5. But the record is for the Vaio with a speedup of 2.2!

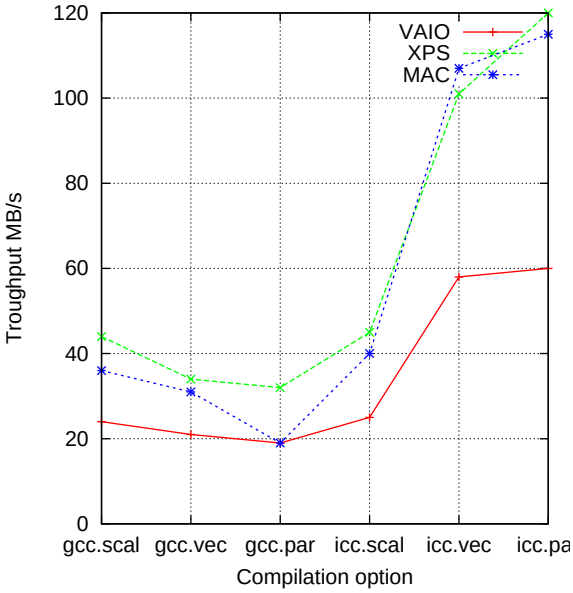


Figure 14. mixer.dsp benchmark

Overall Comparison

Figure 18 shows the speedup obtained with the vectorized code. With a good autovectorizing C++ compiler like Intel icc 11.0 we can obtain very significant improvements in many cases. On the contrary, gcc 4.3.2 was not able to generate SIMD instructions, leading to performance degradation. We therefore highly recommend icc to compile vectorized code, which is a pity considering the excellent results of gcc on scalar code.

Following the so called Amdahl's law, the speedup obtained with the parallelized code is highly dependent of the quantity of parallelism available (see figure 19). On purely parallel programs like fdelay8 and rms8, a speedup exceeding 2.5 was observed on the Mac. This is a little bit disappointing for a 8-cores machine, but in phase with its relatively limited memory bandwidth. Here too we recommend icc to compile OpenMP applications.

Obviously all these results are dependent of many choices and settings, in particular compiler's options. The options we have retained were the best we could find, but the parameter space is huge and we have only explored a little part of it. It may be the case that the gcc results could be improved by changing the settings. This would be good news and the authors are interested by any suggestions on that point.

There is also a lot of possible improvements in the code generated by FAUST. While it is easy to discover all the potential parallelism of a FAUST program, generating efficient OpenMP programs is much more difficult due to the overheads introduced and the additional pressure on the shared memory.

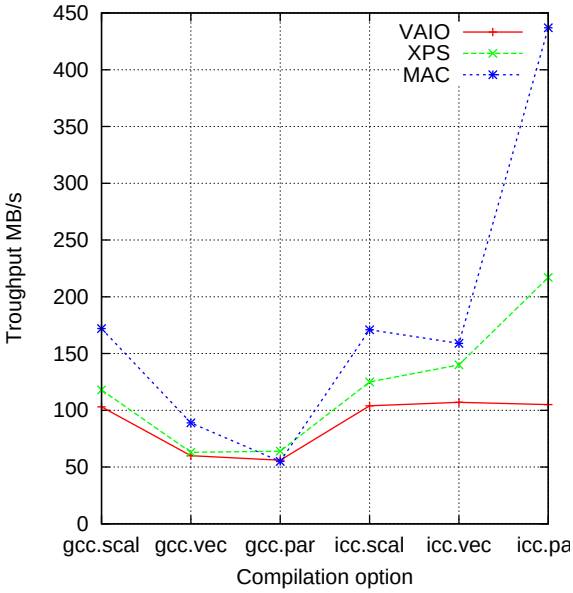


Figure 15. fdelay8.dsp benchmark

Conclusion

Functional programming languages have a reputation of being slow and inefficient. Although the DSP domain here is clearly delimited and highly specialized, the example of FAUST shows that this is not necessarily true. Having a simple and well defined mathematical semantics allows the FAUST compiler to perfom deep transformations and simplifications leading to very efficient code. It is also crucial in the long-term preservation of the *meaning* of music programs.

Semantics is also the key to allow a clear distinction of roles between the FAUST language and the FAUST compiler. The role of the language is a *specification* one. It is to provide the most adequate and expressive notation for signal processing *independently of any implementation consideration*. The role of the compiler is to provide the best possible implementation. Ideally this means that two very different, but semantically equivalent, FAUST programs should lead to exact same implementation.

The question of parallelism illustrates that point. FAUST is not a parallel language nor a dataflow language for the same reason that mathematical expressions like $3*x+2*y$ are not parallel or dataflow expressions. Parallelism is an implementation aspect and as such it is of the compiler responsibility, not of the language responsibility.

The FAUST compiler has no difficulties in discovering all the potential parallelism of a FAUST program[5]. What is much more difficult is to generate efficient parallel programs with current SMP architectures. The tradeoff between parallelism and overhead + memory pressure is something that is difficult to evaluate and highly dependent on

[5]Parallel programming is probably the chance of functional programming to go mainstream.

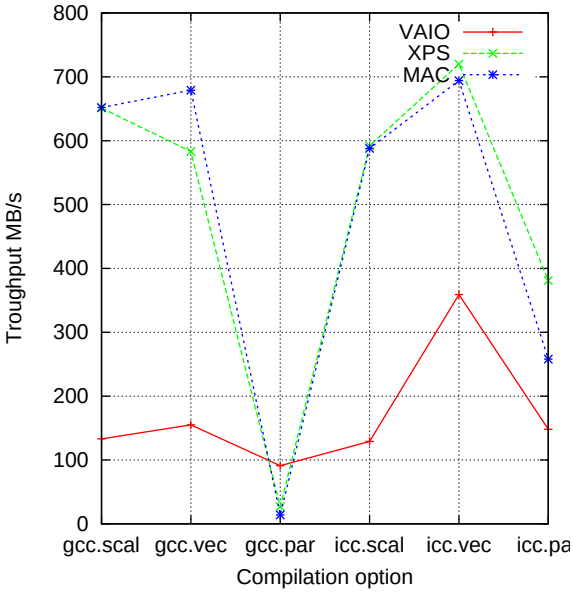


Figure 16. rms.dsp benchmark

hardware details. SMP architectures show clearly their limitations. It will be interesting to explore the possibilities of GPGPU and their high-level programming languages as an alternative to C++ and OpenMP.

Bibliography

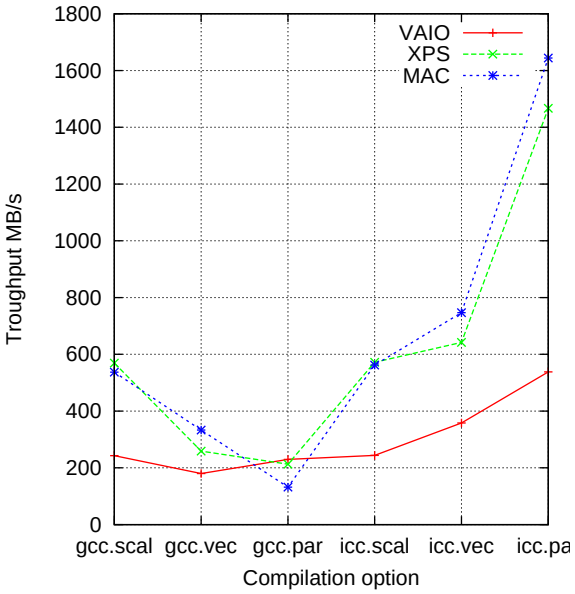


Figure 17. rms8.dsp benchmark

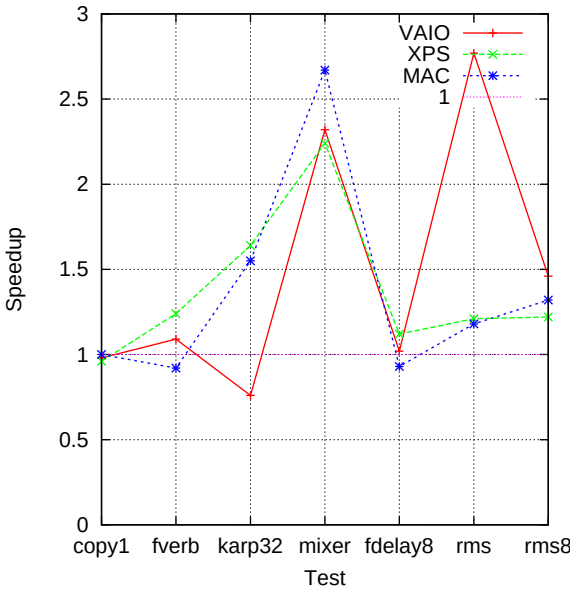


Figure 18. Speedup ratio between vector and scalar code (using icc)

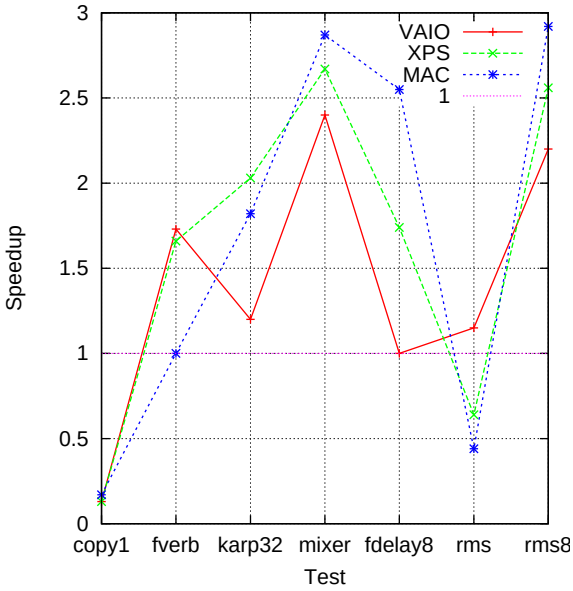


Figure 19. Speedup ratio between parallel and scalar code (using icc)

[1] M. Puckette. The patcher. In *Proceedings of the International Computer Music Conference*. ICMA, 1988.

[2] G. Assayag and C. Agon. OpenMusic Architecture. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 339–340, 1996.

[3] S. Letz, Y. Orlarey, and D. Fober. The Role of Lambda-Abstraction in Elody. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 377–384, 1998.

[4] Ge Wang and Perry R. Cook. Chuck: A concurrent, on-the-fly, audio programming language. In ICMA, editor, *Proceedings of International Computer Music Conference*, 2003.

[5] M. Puckette. Pure data. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 224–227, 1997.

[6] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*. North-Holland, 1974.

[7] J. B. Dennis and D. P. Misunas. A computer architecture for highly parallel signal processing. In *Proceedings of the ACM 1974 National Conference*, pages 402–409. ACM, November 1974.

[8] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[9] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.

[10] Y. Orlarey, D. Fober, and S. Letz. An algebra for block diagram languages. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 542–547, 2002.

[11] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9):623–632, 2004.

[12] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[13] J. O. Smith. Virtual Electric Guitars and Effects Using Faust and Octave. In LAC, editor, *Linux Audio Conference*, 2008.

[14] J. O. Smith. *Introduction to Digital Filters with Audio Applications*. W3K Publishing, 2007.

[15] A. Graef, S. Kersten, and Y. Orlarey. DSP Programming with Faust, Q and SuperCollider. In LAC, editor, *Linux Audio Conference*, 2006.

[16] A. Graef. Interfacing Pure Data with Faust. In LAC, editor, *Linux Audio Conference*, 2007.