PARALLELIZATION OF AUDIO APPLICATIONS WITH FAUST

Yann Orlarey
Grame
orlarey@grame.fr

Dominique Fober
Grame
fober@grame.fr

Stephane Letz
Grame
letz@grame.fr

ABSTRACT

Faust 0.9.9.6 introduces new compilation options to automatically parallelize audio applications. This paper explains how the automatic parallelization is done and presents some benchmarks.

1 INTRODUCTION

Faust is a programming language for real-time signal processing and synthesis designed from scratch to be a compiled language. Being efficiently compiled allows Faust to be complementary to existing audio languages and to provide a viable high-level alternative to C/C++ to develop high-performance signal processing applications, libraries or audio plug-ins.

Until recently the computation code generated by the compiler was organized quite traditionally as a single sample processing loop. This scheme works very well but it doesn't take advantages from multicore architectures. Moreover it can generate code that exceeds the autovectorization capabilities of current C++ compilers.

We have recently extended the compiler with two new schemes : the *vector* and the *parallel* schemes. The *vector* scheme simplifies the autovectorization work of the C++ compiler by splitting the sample processing loop into several simpler loops. The *parallel* scheme analyzes the dependencies between these loops and adds OpenMP pragmas to indicate those that can be computed in parallel.

These new schemes can produce interesting performance improvements. The goal of this paper is to present these new compilation schemes and to provide some benchmarks comparing their performances. The paper is organized as follows : next section will give a brief overview of Faust language, The third section will present the three code generation schemes and the last section will introduce the benchmarks used and the results obtained.

SMC 2009, July 23-25, Porto, Portugal

2 FAUST OVERVIEW

In this section we give a brief overview of Faust with some examples of code.

A Faust program describes a *signal processor*, something that transforms some input signals and produces some output signals. The programming model used combines a *functional programming approach* with a *block-diagram syntax*. The functional programming approach provides a natural framework for signal processing. Digital signals are modeled as discrete functions of time, and signal processors as second order functions that operate on them. Moreover Faust's block-diagram *composition operators*, used to combine signal processors together, fit in the same picture as third order functions.

The Faust compiler translates Faust programs into equivalent C++ programs. It uses several optimization techniques in order to generate the most efficient code. The resulting code can usually compete with, and sometimes outperform, DSP code directly written in C/C++. It is also self-contained and doesn't depend on any DSP runtime library.

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 1) and new ones can be easily added.

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
supercollider.cpp	Supercollider plug-in
vst.cpp	VST plug-in
q.cpp	Q language plug-in

Table 1. Some architecture files available for Faust

In the following subsections we are giving a short and informal introduction to the language through the example of

a simple noise generator. Interested readers can refer to [1] for a more complete description.

2.1 A simple noise generator

A Faust program describes a signal processor by combining primitive operations on signals (like $+, -, *, /, \sqrt{}, \sin, \cos, \ldots$) using an algebra of high level *composition operators* [2] (see Table 2). You can think of these composition operators as a generalization of mathematical function composition $f \circ g$.

f ~ g	recursive composition
f , g	parallel composition
f : g	sequential composition
f <: g	split composition
f :> g	merge composition

Table 2. The five high level block-diagram *composition operators* used in Faust

A Faust program is organized as a set of *definitions* with at least one for the keyword `process` (the equivalent of `main` in C).

Our noise generator example `noise.dsp` only involves three very simple definitions. But it also shows some specific aspects of the language:

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

The first definition describes a (pseudo) random number generator. Each new random number is computed by multiplying the previous one by `1103515245` and by adding to the result `12345`.

The expression `+(12345)` denotes the operation of adding `12345` to a signal. It is an example of a common technique in functional programming called *partial application*: the binary operation `+` is here provided with only one of its arguments. 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)`.

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.

2.2 Invoking the compiler

The role of the compiler is to translate Faust programs into equivalent C++ programs. The key idea to generate efficient code is not to compile the block diagram itself, but *what it computes*.

Driven by the semantic rules of the language the compiler starts by propagating symbolic signals into the block diagram, in order to discover how each output signal can be expressed as a function of the input signals.

These resulting signal expressions are then simplified and normalized, and common subexpressions are factorized. Finally these expressions are translated into a self contained C++ class that implements all the required computation.

To compile our noise generator example we use the following command :

```
$ faust noise.dsp
```

This command generates the following C++ code on the standard output :

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

The generated class contains seven 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. `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.

2.3 Generating a full application

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 generate 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. The figure 1 is a screenshot of our noise application running.

Figure 1. Screenshot of the noise example generated with the `jack-qt.cpp` architecture

2.4 Generating a block-diagram

Another interesting option is `-svg` that generates one or more SVG graphic files that represent the block-diagram of the program as in Figure 2.

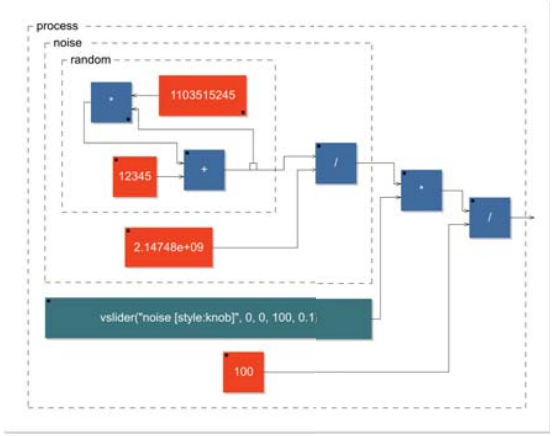

Figure 2. Graphic block-diagram of the noise generator produced with the -svg option

It is interesting to note the difference between the block diagram and the generated C++ code. The block diagram involves one addition, two multiplications and two divisions. The generated C++ program only involves one addition and two multiplications per sample. The compiler has been 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 itself, but *what it computes*.

3 CODE GENERATION

In this section we describe how the Faust compiler generates its code. We will first introduce the so called *scalar* generation of code which was the only one until version 0.9.9.5. Then, we will present the *vector* generation of code where the code is organized into several loops that operates on vectors, and finally the *parallel* generation of code where these vector loops are parallelized using OpenMP directives.

3.1 Preliminary steps

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

3.1.1 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.

3.1.2 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 1. 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*.

3.1.3 Discovering the mathematical equations

Faust doesn't compile a block-diagram directly. It uses a phase of symbolic propagation to first discover its mathematical semantic (what it computes). 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, but computing 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.

3.1.4 Typing the mathematical equations

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 samples.
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.

3.1.5 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 = <common subexpression code>;` is generated, and the cache variable `fTemp` is used in its enclosing expressions. Otherwise the subexpression code is used inlined.

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. 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 also need to 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 require to be cached.

Only after this phase can the generation of the C++ code start.

3.2 Scalar Code generation

The generation of the C++ code is made by populating a *klass* object (representing a C++ class), with strings representing 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.

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

We don't have enough room to go in too much details but here is the translation rule for the addition of two signal expressions:

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

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

Let's say that the string passed to the cache function $\mathbf{C}()$ is `(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 and return `fTemp0` as a string to be used when compiling enclosing expressions. If the expression is not shared it will simply return the string `(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;
  }
}
```

3.3 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 x4. 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 new vector code generation 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 communicates 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)
mean(n) = float2fix : integrate(n) :
          fix2float : /(n);

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

// Convertion 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) ;
```

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 performance of 129.144 MB/s, while the vector version achieves 359.548 MB/s, a speedup of x2.8 !

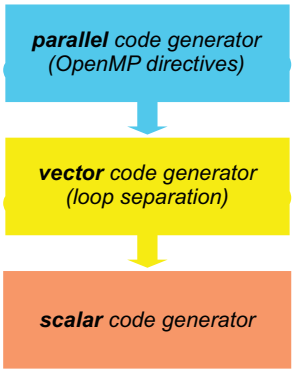

Figure 3. Faust's stack of code generators

The vector code generation is built on top of the scalar code generation (see figure 3). 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 (see Figure 4). This graph is stored in the klass object and a topological sort is applied to it before printing the code.

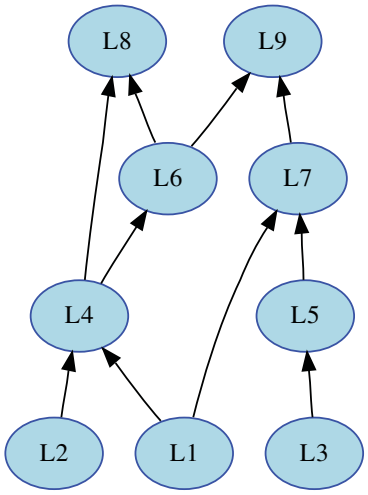

Figure 4. The result of the -vec option is a directed acyclic graph (DAG) of small computation loops

3.4 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.

3.4.1 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 5). 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
{
  // 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, OpemMP 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:

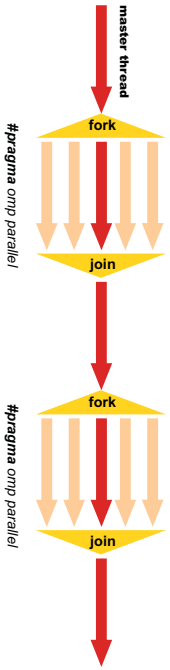

Figure 5. OpenMP is based on a fork-join model

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

  ...
}
```

3.4.2 Adding OpenMP directives

As said before 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 S_0 (loops $L1$, $L2$ and $L3$ in the DAG of Figure 4)

contains the loops that don't depend on any other loops, the set S_1 contains the loops that only depend on loops of S_0, (that is loops $L4$ and $L5$), etc..

As all the loops of a given set S_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 constains 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
}
```

3.4.3 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 6 the corresponding block-diagram):

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

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];
      }
    }
  }
}
```

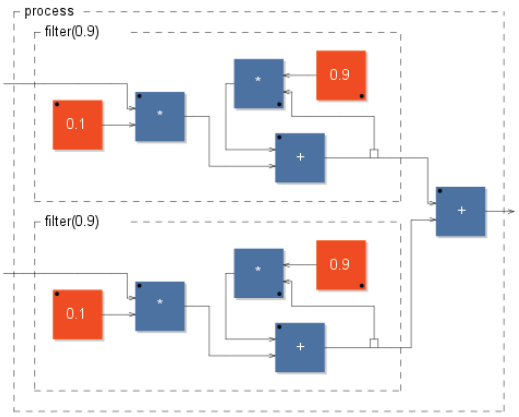

Figure 6. two filters in parallel connected to an adder

This code appeals for 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 `firstprivate(fRec0,fRec1)` clause 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 quite 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. Please 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 and the other between count/2 and count.

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

4 BENCHMARKS

To compare the performances of these three types of code generation in a realistic situation we have 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);
}
```

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 processors cycles, is first converted into a duration, and then into 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 uses all the available memory bandwidth, there is no room for improvement. In this case a parallel version can only perform worth.

4.1 Machines and compilers used

In order to compare the scalar code generation with the new vector and parallel code generation, we have 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 have also used two different C++ compilers, GNU GCC and Intel ICC :

- GCC version 4.3.2 with options : `-O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize.` (`-fopenmp` added for OpenMP).

- ICC version 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

4.2 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 have obtained are summarized figure 7. The horizontal axe corresponds to the three faust compilation schemes : *scalar* , *vector* and *parallel*, combined with the two C++ compilers : *gcc* and *icc*. The vertical axe is the throughput : how many bytes of samples each tested program is able to produce per second (higher values are the 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 8 compares the performances of copy1 (compiled with icc) on the Macpro on 5 different runs. As we can see the stability is reasonably good.

4.3 Benchmark: freeverb.dsp

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

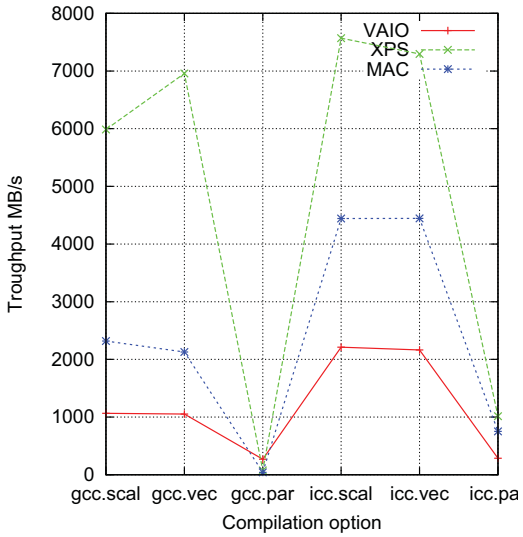

Figure 7. Copy1.dsp benchmark

The results are given figure 9. 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 version the 8 3GHz cores of the macpro were slower than 4 2.5Ghz cores of the XPS !

4.4 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 10 gives the results. Again we can see excellent performances of gcc in scalar mode, good progression of the performances in vector mode as well as in parallel mode for icc.

4.5 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");

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

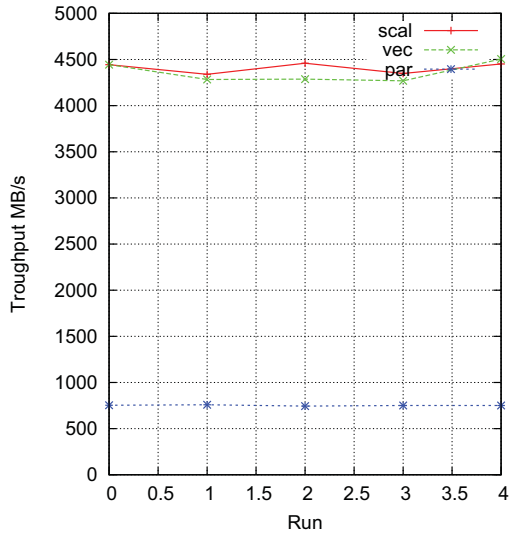

Figure 8. Stability of measures (copy1 on macpro, icc version)

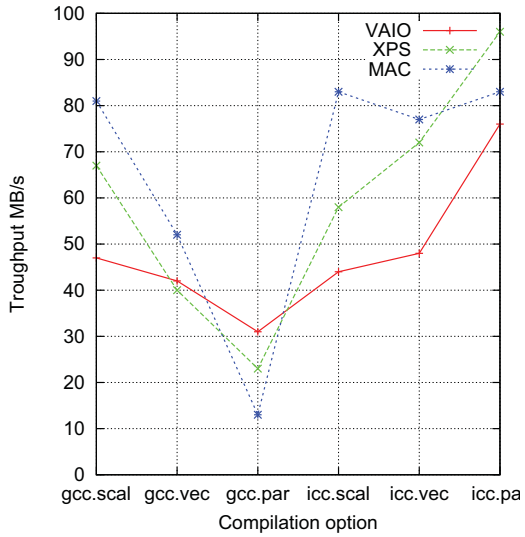

Figure 9. Freeverb.dsp benchmark

```
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 );

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

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

The results of figure 11 show a real benefit for the vectorized version with a speedup exceeding x2 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.

4.6 Benchmark: fdelay8.dsp

This test implements an 8-channels 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);
      };

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

The results are presented figure 12. The Macpro exhibits a good speedup of x2.5 for its parallel version. The parallel speedup for the XPS machine is more limited and there is no speedup at all on the Vaio.

4.7 Benchmark: rms.dsp

The Faust source of rms.dsp was presented section 3.3. It is a purely sequential algorithm therefore the performances of the parallel versions are very bad. But, as figure 13 indicates, the vectorization gives a real boost to the performances, particularly on the vaio.

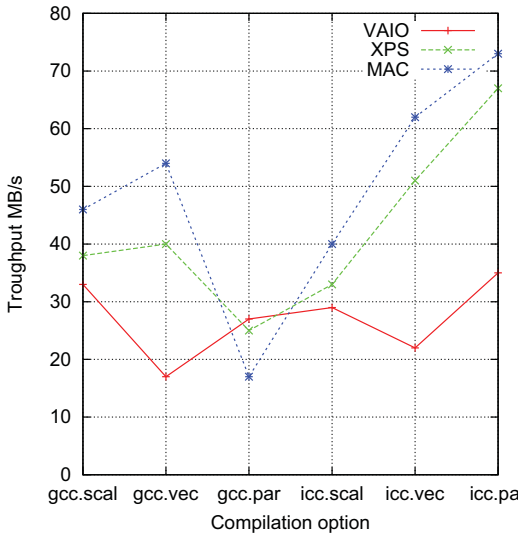

Figure 10. Karplus32.dsp benchmark

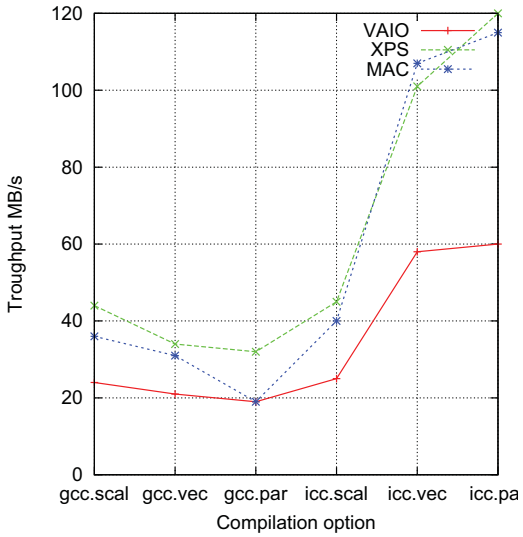

Figure 11. mixer.dsp benchmark

4.8 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")) ;
```

We obviously have a good amount of parallelism here that icc is able to exploit as indicated by the results figure 14. Compared to the scalar performances, the parallel version exhibits a speedup of nearly x3 on the Mac, while the speedup for the XPS exceed x2.5. But the record is for the Vaio with a speedup of x2.2 !

5 CONCLUSION

We have presented two new compilation schemes recently introduced in the Faust compiler. The *vector* scheme simplifies the autovectorization work of the C++ compiler by splitting the sample processing loop into several simpler loops. The *parallel* scheme analyzes the dependencies between these loops and add OpenMP pragmas to indicate those that can be computed in parallel.

Figure 15 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 a degradation of the performances. We therefore highly recommend icc to compile vectorized code, that 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 on the quantity of parallelism available (see figure 16. On purely parallel programs like fdelay8 and rms8 a speedup exceeding x2.5 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.

All these results are dependent on many choices and settings, in particular on compiler's options. The options we have retained were the best we could find, but the parameters space is huge and we have only explored a small 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 the whole potential parallelism of a Faust program [1], generating efficient OpenMP programs is much more difficult due to the overheads introduced and the additional pressure on the shared memory.

The trade-off between parallelism and overhead + memory pressure is something that we will have to improve in future versions. The fixed scheduling of the parallel tasks is also probably far from optimal in many cases. It will be also

[1] parallel programming is probably the chance of functional programming languages compared to imperative languages

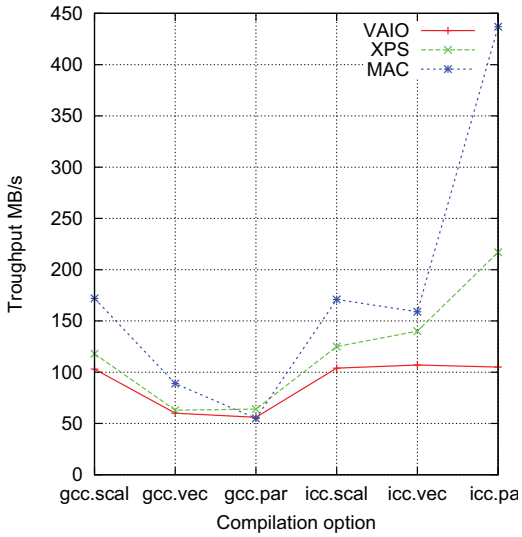

Figure 12. fdelay8.dsp benchmark

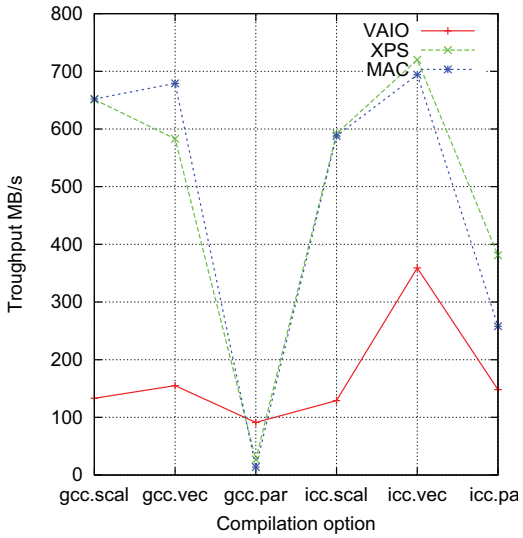

Figure 13. rms.dsp benchmark

interesting to explore the possibilities of GPGPU and their high-level programming languages as an alternative to C++ and OpenMP.

Resources

1. http://openmp.org/
2. http://faust.grame.fr
3. http://www.intel.com/cd/software/products/asmo-na/eng/277618.htm

6 REFERENCES

[1] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9):623–632, 2004.

[2] 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.

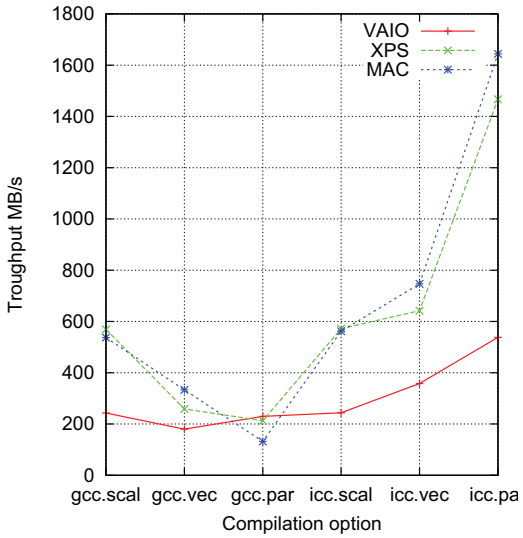

Figure 14. rms8.dsp benchmark

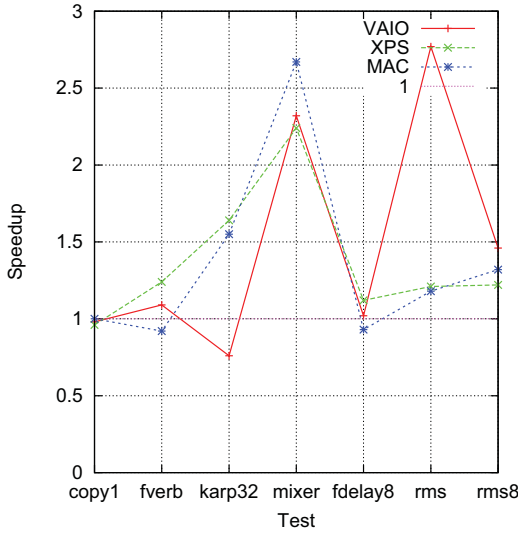

Figure 15. Speedup ratio between vector and scalar code (using icc)

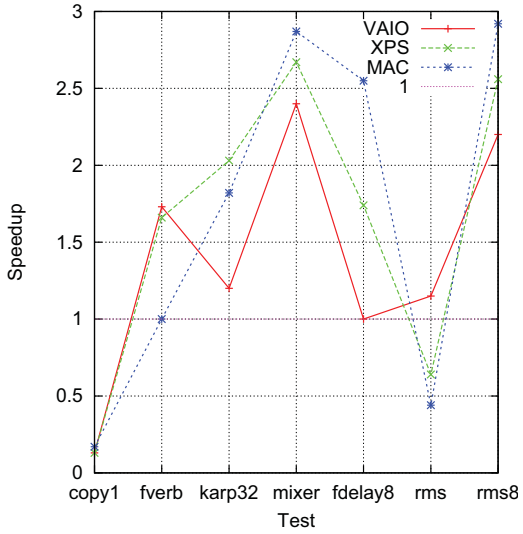

Figure 16. Speedup ratio between parallel and scalar code (using icc)