Automatic Parallelization of Audio Applications with Faust

Yann Orlarey, Stéphane Letz, Dominique Fober

10ème **C**ongrès **F**rançais d'**A**coustique

Lyon, 12-16 Avril 2010

Automatic Parallelization of Audio Applications with Faust

Yann Orlarey, Stéphane Letz, Dominique Fober

Grame, centre national de création musicale, BP 1185, 69202 Lyon Cedex 01

1 Introduction

FAUST (Functional AUdio STreams) [1] stands for both a programming language and its compiler. Being fully compiled allows FAUST to be used as an alternative to C/C++ to develop high-performance audio signal processing applications, DSP libraries and plug-ins for a variety of audio platforms and standards.

Several principles have guided the design of FAUST :

- FAUST is a *specification language*. It aims at providing an adequate notation to describe *signal processors* from a mathematical point of view. It is, as much as possible, free from implementation details.
- Audio programming languages are generally interpreted. FAUST is the first to be fully compiled. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code.
- 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.
- 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.
- 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.
- Thanks to the notion of *architecture*, FAUST programs can be easily deployed on a large variety of audio platforms and plugin formats without any change to the FAUST code.

In the following section we will briefly describe the FAUST compiler and how it generates C++ code. In particular we will introduce the new compilation techniques used to generate parallel C++ code.

2 The Faust compiler

Being efficiently compiled is essential for FAUST to be used as an alternative to C/C++ to write high performance audio applications and plugins. Several steps are involved to compile a Faust program into C++. Before describing the C++ code generation itself, let's first see a crucial step that translates a FAUST program into mathematical equations.

2.1 The math behind a Faust program

The FAUST compiler is *semantically driven*. It doesn't compile a FAUST program directly, but its *mathematical meaning*. This approach allows two very different FAUST programs, but with the same mathematical meaning, to result in the exact same C++ implementation. In other words, the way a FAUST program is written doesn't matter (in theory), only counts its mathematical meaning. It is the role of the FAUST compiler, not of the user, to provide the best implementation of this mathematical meaning.

In order to discover the mathematical meaning of a program, the FAUST compiler uses a phase of symbolic propagation. The principle is to propagate symbolic signals through the components of the block-diagram in order to get, at the other end, the mathematical equation of the produced signals. These equations are then normalized so that different block-diagrams, but computing mathematically equivalent signals, result in the exact same output equations.

Here is a very simple example of normalization, where two different FAUST programs will result in the same equation. In the first program :

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

the input signal is divided by 2 (`/(2)`) and then delayed by 10 samples (`@(10)`). The `:` operator indicates a sequential composition, the outputs of the left expression are connected to the inputs of the right expression.

In the second program:

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

the input signal first multiplied by 2, then delayed by 7 samples, then divided by 4 and then delayed by 3 samples.

Both programs are different but lead to the same signal equation:

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

and therefore will result in the same C++ program.

Faust applies several rules in order 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. etc.

Once the compiler has the mathematical equations of a program, it goes through a phase of type inference that will help not only to detect errors but also to generate the most efficient C++ code. Then the equations are passed to the code generation stage. Three code generation modes are available : *scalar*, *vector* and *parallel*.

2.2 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 (the default code generation mode) these lines of code are organized in a single sample computation loop, while they can be splitted in several loops with the *vector* and *parallel* schemes.

To illustrate scalar code generation, 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 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]);
  }
}
```

In the next example, the sum of the two input signals is duplicated on two output signals :

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

In this case the expression `(input0[i] + input1[i])` will not be duplicated but 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;
  }
}
```

2.3 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 $\times 4$. 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 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) = ^(2) : mean(n) : sqrt ;

// 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) ;
```

Listing 1: RMS implementation in FAUST

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;
  }
}
```

Listing 2: RMS example, scalar C++ code

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];
  }
}
```

Listing 3: RMS example, vectorized C++ code

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 !

Technically the vector code generation is built on top of the scalar code generation. 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 1). This graph is stored in the klass object and a topological sort is applied to it before printing the code in sequential order (see listing 3).

2.4 Parallel Code generation

The parallel code generator is build on top of the vector code generator. In particular it uses the same DAG of computation loops, because it expresses all the potential parallelism. It is easy to see from figure 1 that L5 and L6 can be computed in parallel, or L4 and L7, but not L1 and L6.

There are two approaches to execute this graph in parallel : *static scheduling* and *dynamic scheduling*. With *static scheduling*, parallel executions are decided at compile time, while with *dynamic scheduling*, they are decided at run time.

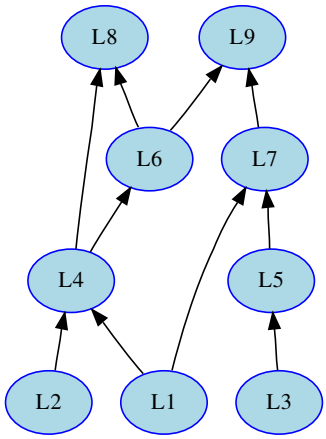

Figure 1: The result of the -vec option is a directed acyclic graph (DAG) of small computation loops

The FAUST compiler provides both options. Static scheduling based on OpenMP is activated with the `--openMP` option, while dynamic scheduling is activated with the `--scheduler` one.

2.4.1 The OpenMP code generator

The `--openMP` (or `-omp`) option given to the FAUST compiler will insert appropriate OpenMP directives in the C++ code. 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. 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:

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

2.4.2 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 1) 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 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
}
```

2.4.3 Example of parallel OpenMP 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 2 the corresponding block-diagram):

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

Listing 4: 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);
```

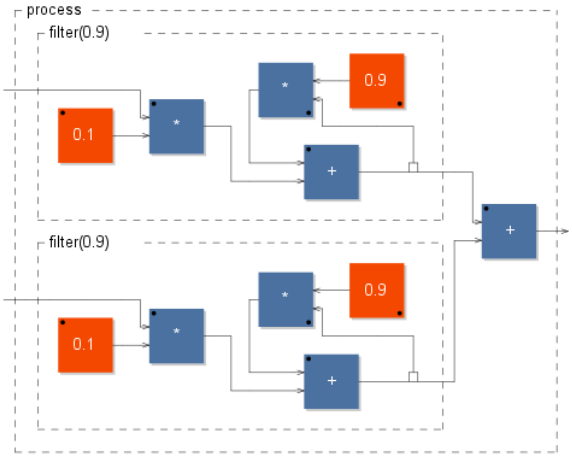

Figure 2: two filters in parallel connected to an adder

```
      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 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 pragma `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 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).

2.4.4 The scheduler code generator

With the `--scheduler` option, the FAUST compiler uses a very different approach. It generates parallel C++ code embedding a *work stealing* scheduler [2] based on pthreads. The idea of *work stealing* is to try to keep the memory cache hot while trying to minimize the interactions between threads.

The C++ code generated contains a static description of the computation DAG (see listing 5). At the beginning of the execution, a pool of worker threads is created. Each thread has it's own local WSQ (Work Stealing Queue), a special LIFO queue were each thread places the runnable tasks he will have to run. Whenever a task is finished, the thread push in the WSQ all the tasks that become runnable (if any) and then pop the next task to run, etc. If the WSQ of a thread becomes empty, the thread is allowed to "steal" a task from another WSQ. But in this case, instead of a LIFO-pop, it does a FIFO-pop in order to select the oldest task of the WSQ.

The local LIFO Pop operation allows better cache locality and the FIFO steal Pop "larger chuck" of work to be done. The reason for this is that many work stealing workloads are divide-and-conquer in nature, stealing one of the oldest task implicitly also steals a (potentially) large subtree of computations that will unfold once that piece of work is stolen and run.

2.4.5 Example of parallel scheduler code

Listing 5 illustrates the scheduler code generated for the parallel filters example of figure 2:

```
virtual void compute (int fullcount,
                      float** input,
                      float** output)
{
  GetRealTime();
  this->input = input;
  this->output = output;
  StartMeasure();
  for (fIndex=0; fIndex<fullcount; fIndex+=32){
    fFullCount = min (32, fullcount-fIndex);
    TaskQueue::Init();
    // Initialize end task
    fGraph.InitTask(1,1);
    // Only initialize tasks with inputs
    fGraph.InitTask(4,2);
    fIsFinished = false;
    fThreadPool.SignalAll(fDynamicNumThreads-1);
    computeThread(0);
    while (!fThreadPool.IsFinished()) {}
  }
  StopMeasure(fStaticNumThreads,
    fDynamicNumThreads);
}
void computeThread (int cur_thread) {
  float*        fRec0 = &fRec0_tmp[4];
  float*        fRec1 = &fRec1_tmp[4];
  // Init graph state
  {
    TaskQueue taskqueue;
    int tasknum = -1;
    int count = fFullCount;
    // Init input and output
    FAUSTFLOAT* input0 = &input[0][fIndex];
    FAUSTFLOAT* input1 = &input[1][fIndex];
    FAUSTFLOAT* output0 = &output[0][fIndex];
    int task_list_size = 2;
    int task_list[2] = {2,3};
    taskqueue.InitTaskList( task_list_size,
            task_list, fDynamicNumThreads,
            cur_thread, tasknum );
    while (!fIsFinished) {
      switch (tasknum) {
        case WORK_STEALING_INDEX: {
          tasknum = TaskQueue::GetNextTask(
                                    cur_thread);
          break;
        }
        case LAST_TASK_INDEX: {
          fIsFinished = true;
          break;
        }
        case 2: {
          for (int i=0; i<4; i++)
            fRec0_tmp[i]=fRec0_perm[i];
          for (int i=0; i<count; i++) {
            fRec0[i] = ((1.0e-01f * input1[i])
                     + (0.9f * fRec0[i-1]));
          }
          for (int i=0; i<4; i++)
            fRec0_perm[i]=fRec0_tmp[count+i];

          fGraph.ActivateOneOutputTask(
                       taskqueue, 4, tasknum);
          break;
        }
        case 3: {
          for (int i=0; i<4; i++)
            fRec1_tmp[i]=fRec1_perm[i];
          for (int i=0; i<count; i++) {
            fRec1[i] = ((1.0e-01f * input0[i])
                     + (0.9f * fRec1[i-1]));
          }
          for (int i=0; i<4; i++)
            fRec1_perm[i]=fRec1_tmp[count+i];
```

```
          fGraph.ActivateOneOutputTask(
                          taskqueue, 4, tasknum);
          break;
        }
        case 4: {
          for (int i=0; i<count; i++) {
            output0[i] = (FAUSTFLOAT)(fRec1[i]
                                      + fRec0[i]);
          }

          tasknum = LAST_TASK_INDEX;
          break;
        }
      }
    }
  }
}
```

Listing 5: Parallel C++ code with an embedded work stealing scheduler

As we can see from the generated code, the structure of the DAG is hard coded in a switch statement. Each case correspond to a task and ends with code to activate the next task in the DAG. Two special cases are also coded, one calls the work stealing method, and the other terminates the code.

3 Performances

To compare the performances of these various compilation schemes in a realistic situation, we have modified an existing architecture file (Alsa-GTK) to continuously measure the duration of the `compute()` method (600 measures per run). We give here the results for two real-life applications : *Karplus32*, a 32 strings simulator based on the Karplus-Strong algorithm (figure 3), and *Sonik Cube* (figure 4), the audio part software of an audio-visual installation.

For every test, the generated C++ code is compiled in 64-bits using icc 11.1.059 and the same set of optimization options (-O3 -xHost -ftz -fno-alias -fp-model fast=2). The machine used to run the tests is a VGN-Z4 Vaio laptop, with a 2.80 GHZ Core 2 Duo P9700 processor and 4 GB of RAM. The operating system is a 64-bits Ubuntu 9.10 Linux system with a 2.6.31-19-generic kernel.

Instead of time, the results of the tests are expressed in MB/s of processed samples because memory bandwidth is a strong limiting factor for today's processors (an audio application can never go faster than the memory bandwidth). Moreover it facilitates comparisons between applications, independently of their number of input/output channels.

As we can see, in both cases the parallelization introduces a real gain of performances. The speedup for *Karplus32* was x2.1 for OpenMP and x2.08 for the WS scheduler. For *Sonik Cube* the speedup was of x1.66 for OpenMP and x1.79 for the WS scheduler.

It is obviously not always the case. Simple applications, with limited demands in terms of computing power, tend to perform usually better in scalar mode. More demanding applications usually benefit from parallelization.

The efficiency of OpenMP and WS scheduler are quite comparable, with an advantage to WS scheduler

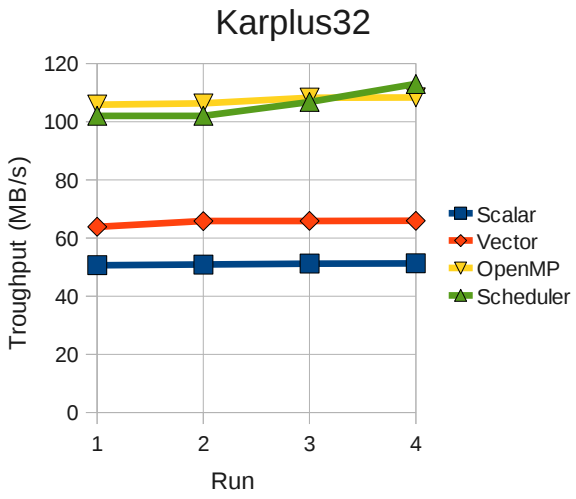

Figure 3: Compared performances of the 4 compilation schemes on karplus32.dsp

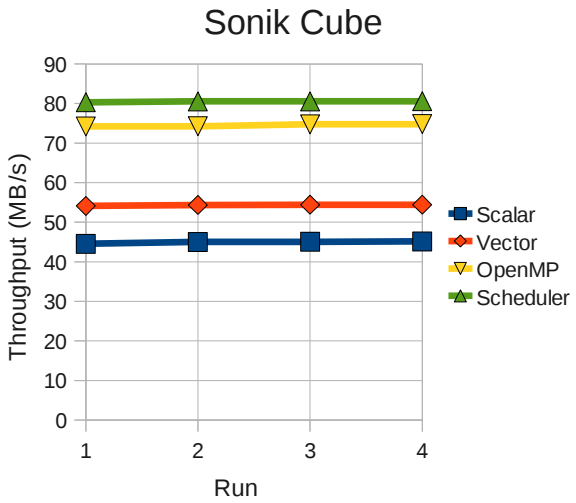

Figure 4: Compared performances of the 4 compilation schemes on Sonik Cube

with complex applications and more CPUs. Please note that not all implementations of OpenMP are equivalent. Unfortunately the GCC 4.4.1 version is still unusable for real time audio application. In this case the WS scheduler is the only choice. The efficiency is also dependent of the vector size used. Vector sizes of 512 or 1024 samples give usually the best results.

References

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

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.