Automatic vectorization in Faust

Nicolas Scaringella, Yann Orlarey, Stéphane Letz, Dominique Fober

Automatic vectorization in Faust

Nicolas Scaringella, Yann Orlarey, Dominique Fober, Stéphane Letz
Grame, Centre National de Création Musicale
9 rue du Garet, BP 1185
69202 Lyon Cedex, France

March 2003

Abstract

Faust is a *Block-Diagram* language for sound signal processing and synthesis. It implements a new algebraic representation of block-diagrams and adopts a functional model of semantics instead of a data flow model [12]. Based on these elements, a compiler able to translate DSP block diagram specification into C code is briefly presented. The code produced proves to be efficient and can compete with a hand written code.

The optimization process is even pushed further: the C code produced can be automatically vectorized to address Altivec extension for PowerPC ([7]) and SSE and SSE2 extensions for Intel architecture ([4]).

A method is proposed to determine whether or not a *Faust* expression can be vectorized by crossing a *type* information (synthesized during an upward run-around in the syntactic tree to be compiled) and a *contextual* information (inherited during a downward run-around in the syntactic tree). Thanks to this method, we are able to find expressions that can be vectorized inside recursive expressions that are not supposed to be vectorizable.

The quality of the code produced by *Faust* is evaluated. On one hand, scalar code produced by *Faust* is compared to vector code produced by *Faust*, on the other hand, scalar and vector code are compared to code optimized by hand.

In the end, we briefly present code transformations to vectorize the expressions classed as non-vectorizable by the previous method so that even better performances can be achieved in the future.

1 The compilation process: from block-diagrams to efficient C code

Before going into the details of automatic vectorization, the compilation of a *Faust* program is briefly presented so that internal representation of signals can be understood.

1.1 Block diagram representation

A graphical specification of a *Faust*'s block diagram is directly equivalent to its textual representation (see [12] for detailed explanations). During the parsing of the source code, an internal block diagram representation can thus be directly built.

The block diagram algebra makes it easy then to analyse formally this internal representation. The analysis is lead through the evaluation of the block diagram. Evaluation consists especially in the application of abstractions (abstractions being the equivalent of functions in the lambda-calculus vocabulary). Abstractions in our case are block diagrams with variable inputs. A larger block diagram is obtained from the evaluation since each function is replaced with its body with the right arguments applied to it. Yet the resulting block diagram is now a construction of elementary blocks of the language.

At this point, the correctness of the source code is ensured both on lexical and syntactical points of view.

1.2 Propagation of signals in the block-diagram

Thanks to lambda-calculus, we have built, from a functionnal description of the program, a single block diagram standing for the complete treatment with n inputs and m outputs.

The goal of a treatment is to produce m signals standing for audio outputs. The C code to produce is therefore the computation of these m outputs. Consequently, we need now an internal representation in terms of signals to generate coding.

A list of n input signals is taken and propagated into the block diagram. A list of m output signals is obtained as a result. The propagation process builds internal trees standing for each signals. The leafs of these trees are the audio inputs, the control inputs (linked to the graphical interface of the application to produce), and the numerical constants. The nodes are the operations or the basic block that needs arguments (such as the data tables or the external functions).

This process builds hence a list of m trees or signals describing the computations to be done for each output of the system. Common subtrees are naturally shared within this process.

1.3 Type of signals

A signal is modeled by a discrete function of time that gives for each instant of time the value that is associated with it. We could base the whole system on this definition, but in an actual implementation, for ease of efficiency and calculation optimisation, it is beneficial to refine this definition.

A typing system is thus proposed to annotate each node of the trees standing for the signals to output. Three characteristics are taken into account:

1. the *nature* of the signal: *integer* or *real.*

2. the *variability* of the signal: *constant* or *evolving* in time. Among signals evolving in time, *low frequency* signals that can be considered constant for the duration of the computation of one block of samples, are distinguished from *high frequency* signals that evolve inside a block of samples.

3. the *computability* of a signal: *deferred timing* (not depending on real-time inputs of the system and that can thus be pre-computed) or *real-time* (depending on real-time inputs of the system). Among *deferred timing* signals, one can distinguish the signals that can be computed at compilation time and the signals that will be computed at the initialisation of the program.

4. the *vectorability* of the signal: *vector* or *scalar.* We focus on the vectorization process from part 2. Part 4 details the rules that determine if a signal is of the *vector* type or of the *scalar* type.

On one hand typing rules aim at checking the validity of expressions and on the other hand at determining certain properties so that more efficient coding can be produced.

1.4 Optimization based on the tree representation

Thanks to the *computability* property, one can detect computation that can be done at compilation time so that the corresponding subtrees can be directly replaced by their result in the tree representation.

Some arithmetic simplifications can also be performed on trees. Among them, operation involving neutral elements or identity elements are simplified.

Sharing of common subexpressions is performed especially in the case of recursive expressions.

1.5 Code generation

Coding is produced during an in-depth wandering in the tree representing the signals. A C++ object is produced to stand for the treatments described in the *Faust* language. Each node of the tree being annotated with a type, the code generation can be more specific:

1. The *nature* of a signal allows the determination of the C type to be affected to the production, that is 32 bits signed integer or 32 bits floating point number.

2. The property of *variability* of a signal determines where in the class to produce its corresponding coding:

- if a signal is constant, then it can be produced directly in the expression that uses it for direct addressing rather than using a temporary variable;
- if a signal is constant at the buffer level, then it is produced out of the loop performing the treatment;
- if it varies for each sample, it is produced inside the loop.

3. The *computability* of a signal determines when to compute a signal:
   - if a signal is computable at compilation time, then the resulting signal is simplified in the tree representation;
   - if it is computable at initialisation time, then it is produced at the initialisation of the program and never again;
   - if it is computable at execution time, then it depends on the system's real-time inputs and will be produced before the loop if it depends on control inputs or inside the loop if it depends on audio inputs.
4. The *vectorability* of a signal combined with the *context* of an expression determine if *vector* code can be produced.

We focus now on the vectorization process.

2 Vectorization: fundamentals

2.1 Definition

A vector can be defined as an ordered list of scalar values (an array can therefore be considered as a vector). A vector instruction applies uniformly the same operation on a set of data.

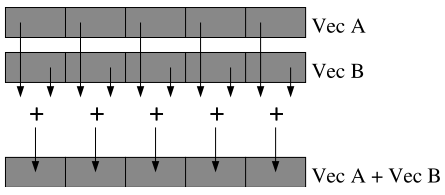

Figure 1: Vector addition

Since in a vector algorithm, parallelism comes from simultaneous operations across a set of data rather than from multiple threads, these algorithms are called *data-parallel* algorithms and we talk about *data-parallel* programming. An *SIMD* architecture (*Single Instruction Multiple Data*) is adapted to that kind of programming. A vectorizing compiler transforms a sequential (or scalar) code into a *data-parallel* code using vector instructions of an *SIMD* architecture.

2.2 The vectorizing process in *Faust*

Faust handles *signals*. The domain $\mathbb{S}$ of *signals* can be defined as the set of functions of time $s : \mathbb{N} \to \mathbb{R}$. We will write $s(t)$ the value of s at time t. *Faust* terms denote *signal processors*, functions transforming n-uples of input signals into m-uples of output signals. The domain $\mathbb{P}$ of *signal processors* is the set of functions $p : \mathbb{S}^n \to \mathbb{S}^m$ (where $n, m \in \mathbb{N}$) from n-uples of input signals to m-uples of output signals.

Let's consider for simplicity's sake a *signal processor* transforming a single input signal into a single output signal ($f : \mathbb{S}^1 \to \mathbb{S}^1$).

$$f(s) \to s' \quad \text{where} \quad s \quad \text{and} \quad s' \in \mathbb{S} \qquad (1)$$

To lighten the writing, we will write $f_t(s_t) \to s'_t$ the processing at instant t ($f_t : \mathbb{R} \to \mathbb{R}$).

The vectorization process transforms the processor $f : \mathbb{S}^1 \to \mathbb{S}^1$ into a processor $fv : \mathbb{S}^N \to \mathbb{S}^N$ where N is the vector size. fv computes N elements at a time and requires therefore N times less iterations.

$$fv_{t..t+N-1} \begin{pmatrix} s_t \\ \vdots \\ s_{t+N-1} \end{pmatrix} \to \begin{pmatrix} s'_t \\ \vdots \\ s'_{t+N-1} \end{pmatrix} \qquad (2)$$

The transformation of f into fv is obvious for basic operations (such as additions or multiplications) that do not involve any *data dependence*. A *data dependence* between two sections of a program indicates that those two sections of code must be run in the order indicated by the dependence (see [5]). This prevent the sections from running in parallel because the results may be different from those of the serial code. This is essentially the case

of recursive definition. Various more or less complex technics exist to vectorize recurrent equations (see [8], [1], [11] ...). As of now, a simple strategy is choosen: in the case of a fragment including a data dependence, the code is unrolled a number of times equal to the length of the vector so that it simulates a vector operation.

Let's consider the following recursive processor ($f : \mathbb{S}^2 \to \mathbb{S}^1$):

$$f_t(s_t, s'_{t-1}) \to s'_t \quad \text{where} \quad s'_t = 0 \quad \text{for} \quad t < 0 \quad (3)$$

By unrolling it N times, we obtain:

$$\begin{array}{rcl} f_t(s_t, s'_{t-1}) & \to & s'_t \\ f_{t+1}(s_{t+1}, s'_t) & \to & s'_{t+1} \\ & \cdots & \\ f_{t+N-1}(s_{t+N-1}, s'_{t+N-2}) & \to & s'_{t+N-1} \end{array} \quad (4)$$

For a recursive equation, we will produce scalar code corresponding to equation (4). Yet, for a better analogy with equation (2), equation (4) can be rewritten as:

$$\begin{array}{rcl} f_t(s_t, s'_{t-1}) & \to & s'_t \\ f^2_{t+1}(s_{t+1}, s'_{t-1}) & \to & s'_{t+1} \\ & \cdots & \\ f^N_{t+N-1}(s_{t+N-1}, s'_{t-1}) & \to & s'_{t+N-1} \end{array} \quad (5)$$

where $f^2_t = f_{t-1} \circ f_t$ stands for the composition of functions f_t.

We have thus a function $fvrec$ that simulates a vector function of the type of equation (2):

$$fvrec_{t..t+N-1} \begin{pmatrix} s_t \\ \vdots \\ s'_{t+N-1} \end{pmatrix}, s'_{t-1} \to \begin{pmatrix} s'_t \\ \vdots \\ s'_{t+N-1} \end{pmatrix} \quad (6)$$

2.3 Recognition of vectorizable code fragments

We have to deal on the one hand with vectorizable expressions that do not involve any data dependence and on the other hand with non-vectorizable expressions for which we need to simulate a vector operation. Consequently, we need a method to separate vectorizable expressions from non-vectorizable ones.

To determine whether an expression is vectorizable, we cross a type information of the considered expression and a context information in which the expression appears. We assume two possible contexts: *vector* context when a subexpression is included inside an expression vectorized and *scalar* context when a subexpression is included inside an expression that is unrolled. In the same way, we assume two possible types: *vector* type for a vectorizable expression and *scalar* type for a non-vectorizable one.

The context in which an expression appears is determined, or *inherited*, during a downward run-around of the tree-expression to be compiled. It is a function of the context of including expressions and the type of the expression considered.

The type of an expression is determined, or *synthesized*, under certain rules and from the type of the sub-expressions it is composed of (typing during an upward wandering of the tree-expression to be compiled). That type depends on the very nature of the object to type (the type is intrinsic to the object in simple cases).

The crossing of these two informations allows distinction between vectorizable fragment and non vectorizable ones.

3 Determination of the context

The context is determined during the code generation. Depending on the context, either vector or unrolled scalar codes will be produced.

The context is *à priori* vectorial as we want to produce vector results. The in depth wandering of the tree to be compiled is initialised with the vector context.

The context evolves at each node according to the following rules:

- if the context is vectorial and the considered expression is of the vector type, then the context stays vectorial (vector code is produced for this expression);
- if the context is vectorial and the considered

expression is of the scalar type, then the context becomes scalar (scalar code is produced for this expression and the intermediate results of each unrolled iteration build a vector interface towards the including expression);

- if the context is scalar and the expression is of the scalar type, then the context stays scalar (scalar code is produced for this expression);

- if the context is scalar and the expression is of the vector type, the context becomes vectorial (vector code is produced for this expression and each result of the vector feeds the corresponding iteration of the including scalar expression).

The possible cases are sumed up in the following table:

	Scalar context	Vector context
Scalar type	Scalar production	Context becomes scalar + Scalar production
Vector type	Context becomes vectorial + Vector production	Vector production

Some simplifications are added to these rules. For example in a scalar context, for a numerical constant taking part in any operation, we produce a constant scalar rather than a constant vector in which we would use only one element (knowing that a numerical constant is always of the vector type because it does not prevent the vectorization).

As a matter of fact, these typing and change of context rules are essentially used to determine vectorizable expressions inside scalar expressions.

4 Typing rules for the vectorability

Typing indicates if an expression is potentially vectorizable. A vector type is a necessary condition to vectorization but whether or not the expression will be vectorized depends on the context.

Each rule indicates how to compute the type of an expression depending on the type of its subexpressions. Rules are presented under the form of an environment $\Gamma_i \vdash \Im_i$ (placed above a horizontal line) and a judgement $\Gamma \vdash \Im$ placed below. The environment Γ indicates a lexical context of an expression and is used for the typing of recursive expressions. It remembers the types associated with the identifiers of recursive groups including this expression (this is the notation of simply typed lambda-calculus, see [2]).

4.1 Relations of inclusion between types

If one consider types as sets, then, saying that type t_1 is more specific than type t_2 (in that t_1 would have particularities that t_2 does not have) is equivalent to saying that t_1 is a subset of t_2 : $t_1 \subset t_2$.

Let $\mathcal{V}$ be the vector type and $\mathcal{S}$ the scalar type.

Because a fragment of vector code can be written as a fragment of scalar code unrolled a number of times equal to the size of the vector, the relation $\mathcal{V} \subset \mathcal{S}$ is true.

4.2 Rules related to primitive signals

Numerical constants and real-time inputs of the system (that is audio inputs and control inputs) are vectorizable.

4.2.1 Constants

Numerical constants are vectorizable beceause there are known for every iteration.

$$\frac{n \in \mathbb{Z}}{\varnothing \vdash n : \mathcal{V}} \qquad \frac{n \in \mathbb{R}}{\varnothing \vdash n : \mathcal{V}} \tag{7}$$

4.2.2 Inputs

Audio inputs are vectorizable because the audio flow to be computed is cut and processed by a buffer of a given size. Therefore the samples for a complete buffer including the sample at the instant t are known when the result of the instant t is being computed.

$$\frac{}{\Gamma \vdash \mathbf{input}_i : \mathcal{V}} \tag{8}$$

Control inputs are also vectorizable because they are updated for each buffer treated (therefore they are constant at the buffer level).

$$\frac{}{\Gamma \vdash \mathbf{ctrl}_i : \mathcal{V}} \tag{9}$$

4.3 Rules related to the operations on the signals

The type of an operation is the union of the type of its arguments.

$$\frac{\begin{array}{l}\Gamma \vdash s_1 : t_1 \\ \vdots \\ \Gamma \vdash s_m : t_m \\ \Gamma \vdash p : t'_1 \times \cdots \times t'_m \to t'_0 \qquad \forall i, t_i \subseteq t'_i\end{array}}{\Gamma \vdash p(s_1 \cdots s_m) : \bigcup_{i=1}^{m} t_i} \tag{10}$$

In the case of *Faust*, some particular cases of this general rule are especially used for:

- arithmetic operations with two arguments, this leads to:

$$\frac{\Gamma \vdash s_1 : t_1 \qquad \Gamma \vdash s_2 : t_2}{\Gamma \vdash (s_1 \star s_2) : (t_1 \cup t_2)} \tag{11}$$

- the `mem()` primitive of *Faust* which is a one sample delay of a signal ($mem(x_i) \to x_{i-1}$):

$$\frac{\Gamma \vdash s : t}{\Gamma \vdash \mathbf{mem}(s) : t} \tag{12}$$

The external C functions follow the general rule. In the particular case of functions without arguments such as `random()` in C, vectorization is always possible:

$$\frac{}{\Gamma \vdash p() : \mathcal{V}} \tag{13}$$

4.4 Rules related to groups of mutually recursive signals

A term (which is equivalent in lambda-calculus to an abstraction of a function) standing for a recursive group is of the vector type if it is closed, that is if does not contain any free variable. In the other case it is of the scalar type. This result is true in *Faust* where the only variables that can be free in a term are the references to recursive definitions.

$$\frac{FV(\mu x.(s_1, \cdots, s_n)) = \varnothing}{\Gamma \vdash \mu x.(s_1, \cdots, s_n) : \mathcal{V}} \tag{14}$$

$$\frac{FV(\mu x.(s_1, \cdots, s_n)) \neq \varnothing}{\Gamma \vdash \mu x.(s_1, \cdots, s_n) : \mathcal{S}} \tag{15}$$

where $FV(M)$ denotes the set of free variables in M.

For example, let μx and μy be two recursive groups. In the following term:

$$\mu x.(s_1, \cdots, \mu y.A, \cdots, s_n) \tag{16}$$

where x occurs in A.

x occurs in A which is the *body* of the expression $\mu y.A$. The variable x is free in the term $\mu y.A$, therefore μy is of the scalar of type. On the other hand, μx is of the vector type because it is closed.

One can interpret this result through syntactic trees: a subtree is vectorizable in a recursive tree if there is no path from the recursive definition to the recursive reference passing through the considered subtree.

The projection operator π_i allows to select one of the components of a group of mutually recursive signals.

$$\frac{\Gamma \vdash \mu x.(s_1, \cdots, s_n) : (t_1, \cdots, t_n) (0 < i \leq n)}{\Gamma \vdash \pi_i(\mu x.(s_1, \cdots, s_n)) : t_i} \tag{17}$$

The term x_i is a reference, inside the definition of a recursive group of label x, to the i-th component of that group. A reference is inevitably scalar to reference the state of the recursive variable at the correct iteration.

$$\frac{}{\Gamma, x : (t_1, \cdots, t_n) \vdash x_i : \mathcal{S}} \tag{18}$$

4.5 Rules related to to data arrays

One needs an index signal for reading an array and another for writing so that the desired element can be accessed. This signal is *à priori* independant from the table and follows the typing rules of any other signal. In the case of an index signal function of the content of the array that it is indexing, the type of the index will be scalar as it will be necesseraly included in a recursive loop.

In the case of a read-only table, the reading can therefore be vectorized depending on the type of its index (the read-only table can also be seen as a one argument function, the argument being the index signal, see rule 10).

$$\frac{\Gamma \vdash index : t}{\Gamma \vdash a_{readonly}(index) : t} \tag{19}$$

The case of read-write tables is more complex. Writing before reading (or reading before writing) implies a data-dependence that prevents parallelization. N writings followed by N readings in a table is not equivalent to N writings and readings. If a reading and a writing can occur at the same storage location in different iterations then a data dependence exists within the loop. The easiest way to bypass this issue is to unroll the corresponding code so that a vector operation is simulated. This technic is all the more justified as the vector instruction sets of both SSE, SSE2 and Altivec technologies only propose vectorized reading or writing for consecutive elements. Therefore, to access any location of a table, we need to use scalar instructions.

Because reading and writing operations need to be unroll, the operation of writing in a table is typed as scalar. Yet the the signal s being written can still be of the vector type.

$$\frac{\begin{array}{c}\Gamma \vdash index : t_1 \\ \Gamma \vdash s : t_2\end{array}}{\Gamma \vdash a_{readwrite}(index) := s : \mathcal{S}} \tag{20}$$

On the other hand, the reading is not necessarily typed as scalar. In fact, the type associated with the reading is the union of the types of the index and of the signal s to be written in the table (just like rule 10).

$$\frac{\begin{array}{c}\Gamma \vdash index : t_1 \\ \Gamma \vdash s : t_2\end{array}}{\Gamma \vdash a_{readwrite}(index) : t_1 \cup t_2} \tag{21}$$

5 Other possible optimizations

We have seen that some expressions can't be vectorized and in that case they are unrolled so that the rhythm of treatment is preserved (N results per iteration, N being the size of the considered vectors). Some other strategies can be used to vectorize the code left as scalar by the previous method and will soon be implemented in our compiler.

5.1 Naturally parallel blocks

The syntax of *Faust* contains a parallel composition (A, B means that the block diagrams A and B execute in parallel). In the case of several parallel block-diagrams that can't be vectorized independently, we can vectorize together each unrolled iterations of A and B.

For example let f be the processor of A and g the processor of B (with the notations of 2.2):

$$\begin{array}{l} f_t(a_t, a'_{t-1}) \rightarrow a'_t \quad \text{where} \quad a'_t = 0 \quad \text{for} \quad t < 0 \\ g_t(b_t, b'_{t-1}) \rightarrow b'_t \quad \text{where} \quad b'_t = 0 \quad \text{for} \quad t < 0 \end{array} \tag{22}$$

Unrolling N times and making appear the parallellisable treatments leads us to:

$$\begin{array}{rcl} f_t(a_t, a'_{t-1}) & \rightarrow & a'_t \\ g_t(b_t, b'_{t-1}) & \rightarrow & b'_t \\ & & \\ f_{t+1}(a_{t+1}, a'_t) & \rightarrow & a'_{t+1} \\ g_{t+1}(b_{t+1}, b'_t) & \rightarrow & b'_{t+1} \\ & \ldots & \\ f_{t+N-1}(a_{t+N-1}, a'_{t+N-2}) & \rightarrow & a'_{t+N-1} \\ g_{t+N-1}(b_{t+N-1}, b'_{t+N-2}) & \rightarrow & b'_{t+N-1} \end{array} \tag{23}$$

We can then find vector operations to share between blocks A and B, that is between processors f and g. This strategy is optimal when f and g do the same computations (for example band-pass filters in parallel) as every operations can then be

vectorized. The best results are obtained when the number of parallel blocks is equal to the size of the vector used.

5.2 Succession of non vectorizable blocks

Let's now consider a succession of non-vectorizable blocks, for example block A followed by block B, f and g standing for the corresponding recursive processors:

$$\begin{array}{l} f_t(a_t, a'_{t-1}) \rightarrow a'_t \quad \text{where} \quad a'_t = 0 \quad \text{for} \quad t < 0 \\ g_t(a'_t, b'_{t-1}) \rightarrow b'_t \quad \text{where} \quad b'_t = 0 \quad \text{for} \quad t < 0 \end{array} \tag{24}$$

Notice that a'_t, output of f_t, feeds g_t. Again, the treatments are unrolled N times.

$$\begin{array}{rcl} f_t(a_t, a'_{t-1}) & \rightarrow & a'_t \\ & & \\ g_t(a'_t, b'_{t-1}) & \rightarrow & b'_t \\ f_{t+1}(a_{t+1}, a'_t) & \rightarrow & a'_{t+1} \\ & & \\ & \cdots & \\ & & \\ g_{t+N-2}(a'_{t+N-2}, b'_{t+N-3}) & \rightarrow & b'_{t+N-2} \\ f_{t+N-1}(a_{t+N-1}, a'_{t+N-2}) & \rightarrow & a'_{t+N-1} \\ & & \\ g_{t+N-1}(a'_{t+N-1}, b'_{t+N-2}) & \rightarrow & b'_{t+N-1} \end{array} \tag{25}$$

Here vector operations to share between f and g can also be found (for iteration t of g and $t-1$ of f). If a number of blocks in serial greater than or equal to the size of the considered vector is found, then the whole width of the vector can be used.

5.3 Vectorizable motifs in a non-vectorizable expression

Another approach is possible. Let's consider a non-vectorizable expression for which the code is unrolled. In this expression, we may find subexpressions that share identical operations with different arguments not bound together. The idea is to built vector operations from these independent subtrees that have the same structure but not the same leaves.

Let's consider for example a numeric filter with N zeros and M poles. Its input-output relation is:

$$y_n = \sum_{i=0}^{N} a_i x_{n-i} + \sum_{j=1}^{M} b_j y_{n-j} \tag{26}$$

The recursive part ($\sum_{j=1}^{M} b_j y_{n-j}$) is *à priori* not vectorizable. Yet it makes use of repetitive operations: M multiplications and $M-1$ additions (or M multiplication-accumulations). A *horizontal* vectorization can be produced and consists in performing the repetitive multiplication and accumulation in parallel until the point where there is no more parallelisation to be found. The scheme of operations for $M = 8$ is shown in figure 2.

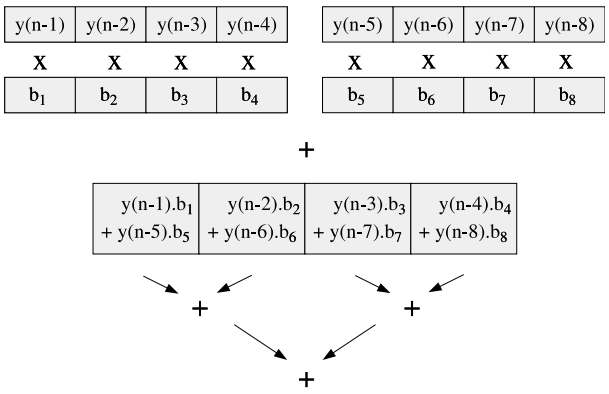

Figure 2: *Horizontal* vectorization

In this example, there are still 3 scalar additions to compute but if a transposed structure is used for the filter (see part 5.4), it is possible to avoid any scalar operations.

5.4 Block diagram simplification

Optimizations based on the equivalence between 2 block-diagrams are possible. One block-diagram may present a lower cost of implementation. For example, a typical equivalence exists between the direct form I structure (figure 3) and the direct form II structure of a filter (figure 4) [9].

Since linear and time invariant systems are commutative, we may reverse the order of the direct form I and implement block A after block B. Then it is obvious that the delay elements can be shared. A canonical form with respect to delay is obtained.

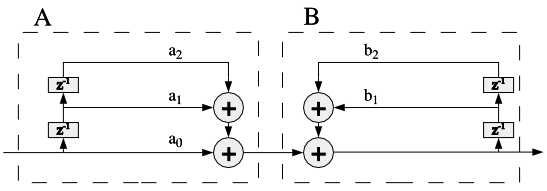

Figure 3: Filter: direct form I (2 zeros, 2 poles)

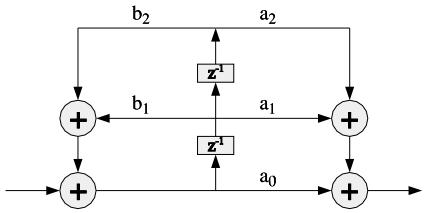

Figure 4: Filter: direct form II (2 zeros, 2 poles)

An equivalence can then be found with the transposed structure (figure 5).

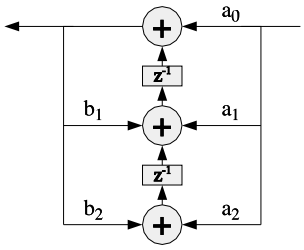

Figure 5: Filter: transposed form (2 zeros, 2 poles)

This equivalence can be proved by an approach based on signal flow-graphs and Mason's theorem [10] (knowing that there is a direct transformation from block-diagram to signal flow-graph). The transposition theorem states that inverting the direction of the branches and inverting input and ouput of the signal flow-graph does not change the system. The signal flow-graphs give a graphical representation of the equations that rule the analysed system. The system can then be simplified as a graph rather than with its corresponding equations. This transposed implementation requires the same operations as the direct form II. Yet, if the multiply-accumulate operation can be used as a single instruction, the cost of implementation is lowered. Indeed, the direct form II needs 2 multiplications, 1 addition and 3 multiply-accumulations while the tranposed structure needs only 1 multiplication and 4 multiply-accumulations.

6 Performance evaluation

In the following tests, scalar and vectorized versions of *Faust* programs are produced and the generated code is compiled with GCC (with the command line `g++ -msse -O3` on PC and `g++ -faltivec -O3` on PowerPC). Notice that the tests presented here only involve floating point vector operations as GCC does not support SSE2 intrinsics when this article is being written (SSE2 intrinsics are supported from the 3.3 release of GCC).

The evaluation consists in comparing the speed of execution of the programs in both scalar and vector mode. The ratio of the scalar time and the vector time gives an evaluation of the acceleration brought by automatic vectorization on a given architecture. The execution times are mesured with *lmbench* [6]. Time needed to treat 100 times blocks of 1024 32 bits floating point samples (one such block per input of the system) is mesured. The treatment for the 1024 samples inputs is divided into smaller loops that treat 16,32,64,128,256,512 or 1024 samples.

The results are presented for an Athlon 1.5 GHz with 512 Mo of RAM and a PowerPC G4 1,0 GHz with 512 Mo of RAM (both of these machines are biprocessors but the tests were not compiled to benefit from this particularity).

6.1 8 tracks mixing

This program takes 8 audio inputs (that means 8 blocks of 1024 floating point numbers in our case), multiplies each input by a gain fixed by a cursor and sums the 8 tracks (appendix A). The code is fully vectorizable.

Accelerations around 2 for PC architecture and around 4.5 on PowerPC architecture are obtained. This can be explained by the fact that the Altivec unit of PowerPC has 4 times more vector registers than the SSE unit of PC. Thus on the PowerPC a lot of intermediate results and all vector con-

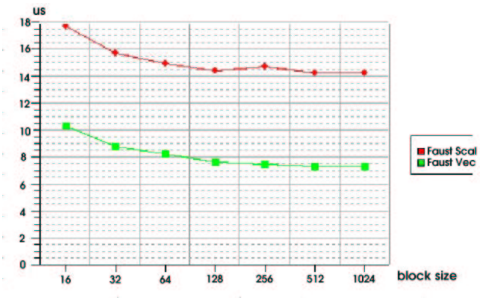

Figure 6: 8 tracks mixing: Athlon 1,5 GHz

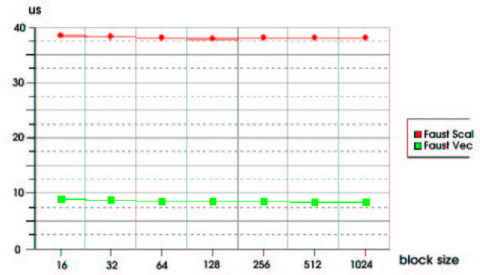

Figure 7: 8 tracks mixing: G4 1,0 GHz

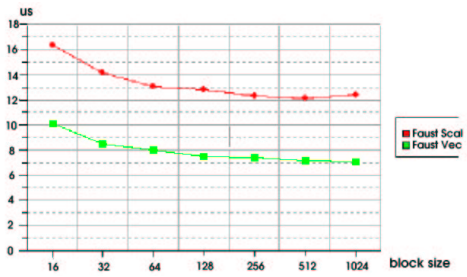

Figure 8: Matrix multiplication: Athlon 1,5 GHz

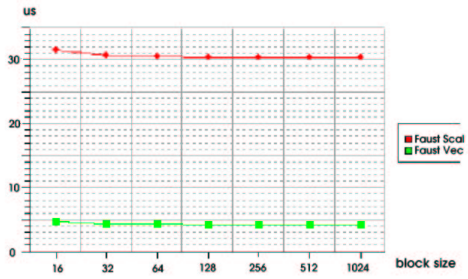

Figure 9: Matrix multiplication: G4 1,0 GHz

stants can be kept in registers minimizing memory traffic. The SSE unit having only 8 such registers need to store more intermediate results for a mixing of 8 tracks. Besides, Altivec proposes multiplication-accumulation in a single instruction for floating point numbers which fits perfectly for mixing (whereas SSE and SSE2 have no such instructions). Furthermore, acceleration seems all the more important on PowerPC because the scalar floating point unit is quite slow compared to the vector unit (the PC architecture is more homogeneous).

6.2 Matrix multiplication

This program takes 3 audio inputs (three blocks of 1024 floating point numbers) considered as a vector (we should say a matrix 3×1 to avoid any confusion) and multiplies it by a 3×3 matrix to produce 3 outputs (each element of the resulting 3×1 matrix, see appendix B). This code is fully vectorizable.

An acceleration around 1.75 on PC and around 7 on PowerPC is obtained for the same reason as stated in the previous example. It is interesting to notice that scalar code is more than 3 times slower on the PowerPC (while the the PC clock is only 1.5 faster) but that the vector code is faster on the PowerPC surpassing its slower clock.

6.3 IIR Filter

A numeric filter with N zeros and M poles has an input-output relation of the following form ([9]):

$$y_n = \sum_{i=0}^{N} a_i x_{n-i} + \sum_{j=1}^{M} b_j y_{n-j} \qquad (27)$$

This program implements an Infinite Impulse Response (IIR) filter with 10 zeros and 11 poles. Two implementations in Faust are proposed:

- a direct-form I implementation (appendix C.1) which is the direct implementation of the previous input-output relation;

- a transposed implementation (appendix C.2) which uses an optimization that we plan to use for *Faust* (see part 5.4).

With the proposed *Faust* implementations, one can build a filter of any size by using the right number of elementary cells. Furthermore, this implementation can represent various kind of filters (finite impulse response filter if all the b_j are equal to 0, purely recursive filter if all the a_i are equal to 0 and all multiplication by 0 will be removed ensuring the optimal code is produced, in the case of a finite impulse response filter, the resulting code is thus completely vectorized).

By way of example, we include on these graphs speed results obtained for the same filter with Intel's implementations (C source code of these implementations can be found in [3]):

- a basic implementation with a delay line;

- an optimized implementation with unrolled coding and a circular buffer (notice that these optimizations depend on the *à priori* knowledge of the size of the filter).

For the G4 graph, results for a Faust code fully vectorized are also added: it was actually vectorized by hand but it implements the future automatic optimizations that we plan to add to the *Faust* compiler (see part 5). Results of this version of the code for the PC were left off the graph as it does not accelerate significantly compared to the *classic* vectorized version.

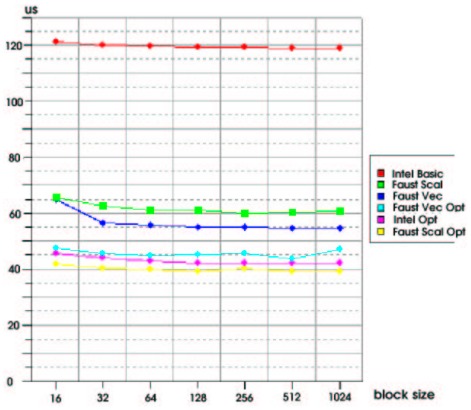

Figure 10: IIR filtering: Athlon 1,5 GHz

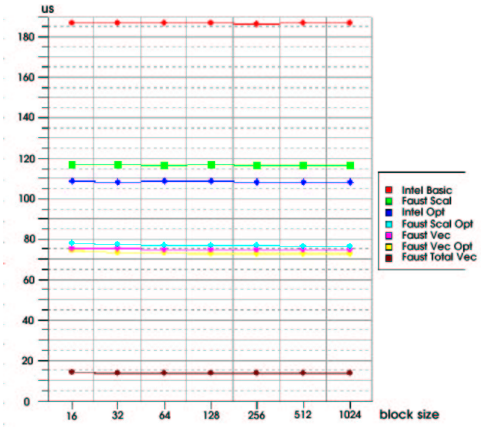

Figure 11: IIR filtering: G4 1,0 GHz

The vectorized versions are not really faster than their scalar versions on PC because this filter is not very vectorizable (for the moment). On PowerPC on the contrary, we observe a significant acceleration because of the slowness of scalar code compared to vectorized code. It is interesting to notice that because of the proportion of scalar code even in vectorized version, the PC is faster than the PowerPC.

Compared to Intel's implementations, we see that all *Faust* versions are at least two times faster than a *naïve* code. We see that *Faust* code can compete with a code optimized by hand and on PowerPC (because of the relative slowness of scalar code), *Faust* can really be faster. The optimized *Faust* version that minimizes the number of additions and memory cells proves to be quite fast (faster than Intel's implementation even in scalar mode).

The fully vectorized (by hand for the moment) version proves to be very efficient and confirms that we have to go further into the automatic vectorization process. This fully vectorized version does not bring a noticeable acceleration on PC nor goes slower so the efforts that have to be made to optimize for PowerPC architecture can also be applied to PC without slowing down. Furthermore, improvment of the vector unit on PC architecture can be expected for the future. Some other optimizations for PC architecture that have not been tried yet may further improve the results obtained with vectorization.

7 Conclusion

We have presented the compilation of *Faust*'s code and its automatic vectorization.

We have proposed a typing system and a rules system to separate vectorizable expressions from non-vectorizable ones in *Faust*. We have briefly presented technics to make vectorization possible in code fragments that are not vectorizable.

In the case of fully vectorizable programs, the vectorized code generated by *Faust* is 1.5 to 2.5 times faster than its scalar homologue for PC and 2.5 to 11.5 faster for PowerPC. In the case of applications more difficult to vectorize, vectorization as it stands now does not improve a lot the performances. Yet, tests using the coming improvements of *Faust* give accelerations with a factor from 5 to 10 on PowerPC.

The code produced by *Faust* in both scalar and vector modes proves to be competitive with code optimized by hand.

References

[1] Blelloch, Chatterjee, and Zagha. Solving linear recurrences with loop raking. *JPDC: Journal of Parallel and Distributed Computing*, 25, 1995.

[2] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[3] Intel. Application note 598: Fir and iir filtering using streaming simd extensions, 1999.

[4] Intel. Ia-32 intel architecture software developer's manual, 2002.

[5] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218. ACM Press, 1981.

[6] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294, Berkeley, January 22–26 1996. Usenix Association.

[7] Motorola. Altivec technology programming interface manual, 1999.

[8] Y. Okabe, M. Nakamura, and T. Tsuda. New fast algorithms for first-order linear recurrences on vector computers. In IFIP Working Group 2.5, editor, *Kyoto Workshop 1995: Current Directions in Numerical Software and High Performance Computing, 19–20 October 1995, Kyoto, Japan*, 1995.

[9] A. V. Oppenheim and R. W. Schafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

[10] John Owen Pliam. An algebraic approach to signal flow graph theory, 1992.

[11] Y. Tanaka, K. Iwasawa, S. Gotoo, and Y. Umetani. Compiling techniques for first-order linear recurrences on a vector computer. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 174–181. IEEE Computer Society Press, 1988.

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

A Faust: 8 tracks mixing

```
volume0 = hslider("Vol0",0.5,0,1,0.025);
volume1 = hslider("Vol1",0.5,0,1,0.025);
volume2 = hslider("Vol2",0.5,0,1,0.025);
volume3 = hslider("Vol3",0.5,0,1,0.025);
volume4 = hslider("Vol4",0.5,0,1,0.025);
volume5 = hslider("Vol5",0.5,0,1,0.025);
volume6 = hslider("Vol6",0.5,0,1,0.025);
volume7 = hslider("Vol7",0.5,0,1,0.025);

process = *(volume0),*(volume1),
          *(volume2),*(volume3),
          *(volume4),*(volume5),
          *(volume6),*(volume7)  +> _;
```

B Faust: Matrix multiplication

```
matrix3x3(a1,a2,a3,b1,b2,b3,c1,c2,c3)
   =  _,_,_  <:   *(a1), *(a2), *(a3),
                  *(b1), *(b2), *(b3),
                  *(c1), *(c2), *(c3)
     +> _;

process = matrix3x3( 0.5, 0.3, 0.9,
                     0.2, 0.8, 0.7,
                     0.3, 0.1, 0.5  );
```

C Faust: 10th order IIR filter

C.1 Direct-Form I

```
cellule(x,cel) = _ <: _,mem : *(x),cel;

tencell(x0,x1,x2,x3,x4,x5,x6,x7,x8,x9) =
           cellule(x0,
           cellule(x1,
           cellule(x2,
           cellule(x3,
           cellule(x4,
           cellule(x5,
           cellule(x6,
           cellule(x7,
           cellule(x8,
           cellule(x9,!))))))))))) +>_;

A = tencell(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9);
B = tencell(b0,b1,b2,b3,b4,b5,b6,b7,b8,b9);

process =  A : + ~ B;
```

C.2 Transposed form

```
cellule(cel,a,b) = cel,_,_ : mem,*(b),*(a)
                           : _,(_,_+>_)+> _;

A = _,_  <: cellule(
            cellule(
            cellule(
            cellule(
            cellule(
            cellule(
            cellule(
            cellule(
            cellule(
          ((*(b9),*(a9))+>_),
                a8,b8),
                a7,b7),
                a6,b6),
                a5,b5),
                a4,b4),
                a3,b3),
                a2,b2),
                a1,b1),
                a0,b0);

process = _ : A ~ _;
```