E.V.E., AN OBJECT ORIENTED SIMD LIBRARY

JOEL FALCOU AND JOCELYN SEROT*

Abstract. This paper describes the EVE (Expressive Velocity Engine) library, an object oriented C++ library designed to ease the process of writting efficient numerical applications using AltiVec, the SIMD extension designed by Apple, Motorola and IBM. AltiVec-powered applications typically show off a relative speed up of 4 to 16 but need a complex and awkward programmation style. By using various template metaprogramming techniques, E.V.E. provides an easy to use, STL-like, interface that allows developer to quickly write efficient and easy to read code. Typical applications written with E.V.E. can benefit from a large fraction of theorical maximum speed up while being written as simple C++ arithmetic code.

1. Introduction.

1.1. The AltiVec Extension.

Recently, SIMD enhanced instructions have been proposed as a solution for delivering higher microprocessor hardware utilisation. SIMD (Single Instruction, Multiple Data) extensions started appearing in 1994 in HP's MAX2 and Sun's VS extensions and can now be found in most of microprocessors, including Intel's Pentiums (MMX/SSE/SSE2) and Motorola/IBM's PowerPCs (Altivec). They have been proved particularly useful for accelerating applications based upon data-intensive, regular computations, such as signal or image processing.

AltiVec [4] is an extension designed to enhance PowerPC[1] processor performance on applications handling large amounts of data. The AltiVec architecture is based on a SIMD processing unit integrated with the PowerPC architecture. It introduces a new set of 128 bit wide registers distinct from the existing general purpose or floating-point registers. These registers are accessible through 160 new "vector" instructions that can be freely mixed with other instructions (there are no restriction on how vector instructions can be intermixed with branch, integer or floating-point instructions with no context switching nor overhead for doing so). Altivec handles data as 128 bit vectors that can contain sixteen 8 bit integers, eight 16 bit integers, four 32 bit integers or four 32 bit floating points values. For example, any vector operation performed on a `vector char` is in fact performed on sixteen `char` simultaneously and is theoretically running sixteen times faster as the scalar equivalent operation. AltiVec vector functions cover a large spectrum, extending from simple arithmetic functions (additions, subtractions) to boolean evaluation or lookup table solving.

Altivec is natively programmed by means of a C API [2]. Programming at this level can offer significant speedups (from 4 to 12 for typical signal processing algorithms) but is a rather tedious and error-prone task, because this C API is really "assembly in disguise". The application-level vectors (arrays, in variable number and with variable sizes) must be explicitly mapped onto the Altivec vectors (fixed number, fixed size) and the programmer must deal with several low-level details such as vector padding and alignment. To correctly turn a scalar function into a vector-accelerated one, a large part of code has to be rewritten.

*LASMEA, UMR 6602 CNRS / U. Clermont Ferrand, France (`falcou, jserot@lasmea.univ-bpclermont.fr`).

[1] PPC 74xx (G4) and PPC 970 (G5).

Consider for example a simple 3x1 smoothing filter (Fig. 1.1):

```
void C_filter( char* d, short* r)
{
  for(int i=1; i<SIZE-1; i++ )
    r[i] = (d[i-1]+2*d[i]+d[i+1])/4;
}
```

FIGURE 1.1. *A simple 3x1 gaussian filter written in standard C.*

This code can be rewritten ("vectorized") using Altivec vector functions. However, this rewriting is not trivial. We first have to look at the original algorithm in a *parrallel* way. The `C_filter` function is based on an iterative algorithm that runs trough each item of the input data, applies the corresponding operations and writes the result into the output array. By contrast, AltiVec functions operate on a bunch of data simulteaneously. We have to recraft the algorithm so that it works on vectors instead of single scalar values. This is done by loading data into AltiVec vectors, shifting these vectors left and right, and performing vector multiplication and addition. The resulting code - which is indeed significantly longer than the original one - is given in Appendix A. We have benchmarked both the scalar and vectorized implementation on a 2 GHz PowerPC G5 and obtained the results shown in Table 1.1. Both code were compiled using `gcc 3.3` using `-O3`. On this example, a ten fold acceleration can be observed with the AltiVec extension. However, the time spent to rewrite the algorithm in a "vectorized" way and the somehow awkward Altivec API can hinder the developement of larger scale applications.

SIZE value	`C_filter`	`AV_filter`	Speed Up
16 K	0.209 ms	0.020 ms	10.5
64 K	0.854 ms	0.075 ms	11.4
256 K	3.737 ms	0.322 ms	11.6
1024 K	16.253 ms	1.440 ms	11.3

TABLE 1.1
Execution time and relative speed-up for 3x1 filters.

2. AltiVec in high level API. As evidenced in the previous section, writting AltiVec-based applications can be a tedious task. A possible approach to circumvent this problem is to encapsulate Altivec vectors and the associated operations within a C++ class. Instanciating this class and using classic infix notations will produce the AltiVec code. We actually built such a class (`AVector`) and used it to encode the filtering example introduced in section 1.1. The resulting code is shown below.

```
AVector<char>  img(SIZE);
AVector<short> res(SIZE);
res = (img.sr(1)+2*img+img.sl(1))/4;
```

In this formulation, explicit iterations have been replaced by application of overloaded operators on `AVector` objects. The `sr` and `sl` methods implements the shifting operations. The performance of this code, however, is very disapointing. With the array sizes shown in Table 1.1, the measured speed-ups never exceed 1. The reasons for such behavior are given below.

Consider a simple code fragment using overloaded operators as shown below:

```
AVector<char> r(SIZE),x(SIZE),y(SIZE),z(SIZE);
r = x + y + z;
```

When a C++ compiler analyzes this code, it reduces the successive operator calls iteratively, resolving first `y+z` then `x+(y+z)` where `y+z` is in fact stored in a temporary object. Moreover, to actually compute `x+y+z`, the involved operations are carried out within a loop that applies the `vec_add` function to every single vector element of the array. An equivalent code, after operator reduction and loop expansion is:

```
AVector<char> r(SIZE),x(SIZE),y(SIZE),z(SIZE);
AVector<char> tmp1(SIZE),tmp2(SIZE);
for(i=0;i<SIZE/16);i++)  tmp1[i] = vec_add(y[i],z[i]);
for(i=0;i<SIZE/16);i++)  tmp2[i] = vec_add(x[i],tmp1[i]);
for(i=0;i<SIZE/16);i++)  r[i] = tmp2[i];
```

FIGURE 2.1. *Expanded code for overloaded operator compilation*

This code can be compared to an "optimal", *hand-written* Altivec code like the one shown on figure 2.2. The code generated by the "naive" AltiVec class clearly exhibits unnecessary loops and copies. When expressions get more complex, the situation gets worse. The time spent in loop index calculation and temporary object copies quickly overcomes the benefits of the SIMD parallelisation, resulting in poor performances. Almost all standard C++ compiler do the same thing. Such expression can't be optimized due to the dyadic reduction scheme used here. If some brute force optimisations can reduce theis overload, the compiler quickly lose all his ways of optimisation when the expression grow larger and larger.

```
AVector<char> r(SIZE),x(SIZE),y(SIZE),z(SIZE);
for(i=0;i<SIZE/16);i++)  r[i] = vec_add(x[i],vec_add(y[i],z[i]));
```

FIGURE 2.2. *Optimal, hand written AltiVec code for x+y+z computation*

This can be explained by the fact that all C++ compilers use a dyadic reduction scheme to evaluate operators composition. Some compilers[2] can output a slightly better code when certain optimisations are turned on. However, large expressions or complex functions call can't be totally optimised. Another factor is the impact of the order of AltiVec instructions. When writing AltiVec code, one have to take in account the fact that cache lines have to be filled up to their maximum. The typical way for doing so is to pack the loading instructions together, then the operations and finally the storing instructions. When loading, computing and storing instructions are mixed in an unordered way, AltiVec performances generally drop.

The aforementionned problem has already been identified – in [7] for example – and is the major inconvenient of the C++ language when it is used for high-level scientific computations. In the domain of C++ scientific computing, it has led to the development of the so-called *Active Libraries* [13, 12, 8, 9], which both provide domain-specific abstractions and dedicated code optimization mechanisms. This paper describes how this approach can be applied to the specific problem of generating efficient Altivec code from a high-level C++ API.

[2]Like Code Warrior or gcc.

It is organized as follows. Sect. 3 explains why generating efficient code for vector expressions is not trivial and introduces the concept of template-based metaprogramming. Sect. 4 explains how this concept can used to generate optimized Altivec code. Sect. 5 rapidly presents the API of the library we built upon these principles. Performance results are presented in Sect. 6. Sect. 7 is a brief survey of related work and Sect. 8 concludes.

3. Template based Meta Programming. The evaluation of any arithmetic expression can be viewed as a two stages process:

- A first step, performed at compile time, where the structure of the expression is analyzed to produce a chain of function calls.
- A second step, performed at run time, where the actual operands are provided to the sequence of function calls and the afferent computations are carried out.

When the expression structure matches certain pattern or when certain operands are known at compile time, it is often possible to perform a given set of computations at compile time in order to produce an optimized chain of function calls. For example, if we consider the following code:

```
for( int i=0;i<SIZE;i++)
{
  table[i] = cos(2*i);
}
```

If the size of the table is known at compile time, the code could be optimized by removing the loop entirely and writing a linear sequence of operations:

```
table[0] = cos(0);
table[1] = cos(2);

// later ...

table[98] = cos(196);
table[99] = cos(198);
```

Furthermore, the value `cos(0)`, ..., `cos(198)` can be computed once and for all at compile-time, so that the runtime cost of such initialisation boils down to 100 store operations.

Technically speaking, such a "lifting" of computations from runtime to compile-time can be implemented using a mechanism known as *template-based metaprogramming*. The sequel of this section gives a brief account of this technique and of its central concept, *expressions templates*. More details can be found, for example, in Veldhuizen's papers [5, 6, 7]. We focus here on how this technique can be used to remove unnecessary loops and object copies from the code produced for the evaluation of vector based expressions.

The basic idea behing *expressions templates* is to encode the abstract syntax tree (AST) of an expression as a C++ recursive template class and use overloaded operators to build this tree. Combined with an array-like container class, it provides

a way to build a static representation of an array-based expression. For example, if we consider an float `Array` class and an addition functor `add`, the expression `D=A+B+C` could be represented by the following C++ type:

```
Xpr<Array,add,Xpr<Array,add,Array>>
```

Where `Xpr` is defined by the following type:

```
template<class LEFT,class OP,class RIGHT>
class Xpr
{
   public:
   Xpr( float* lhs, float* rhs ) : mLHS(lhs), mRHS(rhs) {}

   private:
   LEFT  mLHS;
   RIGHT mRHS;
};
```

The `Array` class is defined as below :

```
class Array
{
   public:
   Array( size_t s ) { mData = new float[s]; mSize = s;}
   ~Array() {if(mData) delete[] mData; }
   float* begin() { return mData; }

   private:
   float *mData;
   size_t mSize;
};
```

This type can be automatically built from the concrete syntax of the expression using an overloaded version of the '+' operator that takes an `Array` and an `Xpr` object and returns a new `Xpr` object:

```
Xpr< Array,add,Array> operator+(Array a, Array b)
{
  return Xpr<T,add,Array>(a.begin(),b.begin());
}
```

Using this kind of operators, we can simulate the parsing of the above code ("A+B+C") and see how the classes get combined to build the expression tree:

```
Array A,B,C,D;
D = A+B+C;
D = Xpr<Array,add,Array> + C
D = Xpr<Xpr<Array,add,Array>,add,Array>
```

Following the classic C++ operator resolution, the A+B+C expression is parsed as (A+B)+C. The A+B part gets encoded into a first template type. Then, the compiler reduce the X+C part, producing the final type, encoding the whole expression.

Handling the assignation of A+B+C to D can then be done using an overloaded version of the assignment operator:

```
template<class XPR> Array& Array::operator=(const XPR& xpr)
{
  for(int i=0;i<mSize;i++) mData[i] = xpr[i];
  return *this;
}
```

The `Array` and `Xpr` classes have to provide a `operator[]` method to be able to evaluate `xpr[i]` :

```
int Array::operator[](size_t index)
{
    return mData[index];
}

template<class L,class OP,class R>
int Xpr<L,OP,R>::operator[](size_t index)
{
  return OP::eval(mLHS[i],mRHS[i]);
}
```

We still have to define the `add` class code. Simply enough, `add` is a functor that exposes a static method called `eval` performing the actual computation. Such functors can be freely extended to include any other arithmetic or mathematical functions.

```
class add
{
  static int eval(int x,int y) { return x+y; }
}
```

With these methods, each reference to `xpr[i]` can be evaluated. For the above example, this gives:

```
data[i] = xpr[i];
data[i] = add::eval(Xpr<Array,add,Array>,C[i]);
data[i] = add::eval(add::apply(A[i],B[i]),C[i]);
data[i] = add::eval(A[i]+B[i],C[i]);
data[i] = A[i]+B[i]+C[i];
```

4. Application to efficient AltiVec code generation. At this stage, we can add AltiVec support to this meta-programming engine. If we replace the scalar computations and the indexed accesses by vector operations and loads, we can write an AltiVec template code generator. These changes affect all the classes and functions shown in the previous sections.

The Array class now provides a `load` method that return a vector instead of a scalar:

```
int Array::load(size_t index) { return vec_ld(data_,index*16); }
```

The add functor now use `vec_add` functions instead of the standard `+` operator:

```
class add
{
  static vector int eval(vector int x,vector int y)
    { return vec_add(x,y); }
}
```

Finally, we use `vec_st` to store results:

```
template<class XPR> Array& Array::operator=(const XPR& xpr)
{
    for(size_t i=0;i<mSize/4;i++)   vec_st(xpr.load(i),0,mData);
    return *this;
}
```

The final result of this code generation can be observed on figure 4.1.b for the A+B+C example. Figure 4.1.a gives the code produced by `gcc` when using the `std::valarray` class.

(a) std::valarray code

```
L253:
lwz r9,0(r3)
slwi r2,r12,2
lwz r4,4(r3)
addi r12,r12,1
lwz r11,4(r9)
lwz r10,0(r9)
lwz r7,4(r11)
lwz r6,4(r10)
lfsx f0,r7,r2
lfsx f1,r6,r2
lwz r0,4(r4)
fadds f2,f1,f0
lfsx f3,r2,r0
fadds f1,f2,f3
stfs f1,0(r5)
addi r5,r5,4
bdnz L253
```

(b) optimized code

```
L117:
slwi r2,r9,4
addi r9,r9,1
lvx v1,r5,r2
lvx v0,r4,r2
lvx v13,r6,r2
vaddfp v0,v0,v1
vaddfp v1,v0,v13
stvx v1,r2,r8
bdnz L117
```

FIGURE 4.1. *Assembly code for a simple vector operation*

For this simple task, one can easily see that the minimum number of loads operation is three and the minimum number of store operations is one. For the standard

code, we have seven extraneous `lwz` instructions to load pointers, three `lsfx` to load the actual data and one `stfs` to store the result. For the optimized code, we have replaced the scalar `lsfx` with the AltiVec equivalent `lvx`, the scalar `fadds` with `vaddfp` and `stfsx` with the vector `stvx`. Only three load instructions and one store instructions, reducing opcode count from 17 to 9.

One can ask for a code proof of this expansion, but the best we can make is having the compiler outputs a trace of the template tree recursion. But a trace is not a proof. The fact that C++ code proof is highly problematic due to a lack of C++ strict semantic and the differences between compiler implementation makes any further approach on this part very fuzzy.

5. The EVE library. Using the code generation technique described in the previous section, we have produced a high-level array manipulation library aimed at scientific computing and taking advantage of the SIMD acceleration offered by the Altivec extension on PowerPC processors. This library, called EVE (for *Expressive Velocity Engine*) basically provides two classes, `vector` and `matrix` – for 1D and 2D arrays –, a way to approximate its speedup at compile time, and a rich set of operators and functions to manipulate them. This set can be roughly divided in four families:

1. **Arithmetic and boolean operators**, which are the direct vector extension of their C++ counterparts. For example:

   ```
   vector<char>  a(64),b(64),c(64),d(64);
   d = (a+b)/c;
   ```

2. **Boolean predicates**. These functions can be used to manipulate boolean vectors and use them as selection masks. For example:

   ```
   vector <char>  a(64),b(64),c(64);

   // c[i] = a[i] if a[i]<b[i], b[i] otherwise
   c = where(a < b, a, b);
   ```

3. **Mathematical and STL functions**. These functions work like their STL or `math.h` counterparts. The only difference is that they take an array (or matrix) as a whole argument instead of a couple of iterators. Apart from this difference, EVE functions and operators are very similar to their STL counterparts (the interface to the EVE `array` class is actually very similar to the one offered by the STL `valarray` class. This allows algorithms developed with the STL to be ported (and accelerated) with a minimum effort on a PowerPC platform with EVE. For example:

   ```
   vector <float>  a(64),b(64);
   b = tan(a);
   float r = inner_product(a, b);
   ```

4. **Signal processing functions**. These functions allow the direct expression (without explicit decomposition into sums and products) of 1D and 2D FIR filters. For example:

```
matrix<float>  image(320,240),res(320,240);
filter<3,horizontal>  gauss_x = 0.25, 0.5, 0.25;
res = gauss_x(image);
```

E.V.E can easily support any algorithm that can be expressed as a serie of global operations on vectors.

6. Performance. Two kinds of performance tests have been performed: basic tests, involving only one vector operation and more complex tests, in which several vector operations are composed into more complex expressions. All tests involved vectors of different types (8 bit integers, 16 bit integers, 32 bit integers and 32 bit floats) but of the same total length (16 Kbytes) in order to reduce the impact of cache effects on the observed performances[3]. They have been conducted on a 2GHz PowerPC G5 with `gcc 3.3.1` and the following compilation switches: `-faltivec -O3`. A selection of performance results is given in Table 6.1. For each test, four numbers are given: the maximum theoretical speedup[4] (TM), the measured speedup for a hand-coded version of the test using the native C Altivec API (N.C), the measured speedup with a "naive" vector library – which does not use the expression template mechanism described in Sect. 3 (N.V), and the measured speedup with the EVE library.

TABLE 6.1
Selected performance results

Test	Vector type	TM	N.C	N.V	EVE
1. `v3=v1+v2`	8 bit integer	16	15.7	8.0	15.4
2. `v2=tan(v1)`	32 bit float	4	3.6	2.0	3.5
3. `v3=v1/v2`	32 bit float	4	4.8	2.1	4.6
4. `v3=v1/v2`	16 bit integer	8(4)	3.0	1.0	3.0
5. `v3=inner_prod(v1,v2)`	8 bit integer	8	7.8	4.5	7.2
6. `v3=inner_prod(v1,v2)`	32 bit float	4	14.1	4.8	13.8
7. 3x1 Filter	8 bit integer	8	7.9	0.1	7.8
8. 3x1 Filter	32 bit float	4	4.12	0.1	4.08
9. `v5=sqrt(tan(v1+v2)/cos(v3*v4))`	32 bit float	4	3.9	0.04	3.9

It can be observed that, for most of the tests, the speedup obtained with EVE is close to the one obtained with a hand-coded version of the algorithm using the native C API. By contrast, the performances of the "naive" class library are very disappointing (especially for tests 7-10). This clearly demonstrates the effectiveness of the metaprogramming-based optimization.

Tests 1-3 correspond to basic operations, which are mapped directly to a single AltiVec instruction. In this case, the measured speedup is very close to the theoretical

[3]I.e. the vector size (in elements) was 16K for 8 bit integers, 8K for 16 bit integers and 4K for 32 bits integers or floats.

[4]This depends on the type of the vector elements: 16 for 8 bit integers, 8 for 16 bit integers and 4 for 32 bit integers and floats.

maximum. For test 3, it is even greater. This effect can be explained by the fact that on G5 processors, and even for non-SIMD operations, the Altivec FPU is already faster than the scalar FPU[5]. When added to the speedup offered by the SIMD parallelism, this leads to super-linear speedups. The same effect explains the result obtained for test 6. By contrast, test 4 exhibits a situation in which the observed performances are significantly lower than expected. In this case, this is due to the asymmetry of the Altivec instruction set, which does not provide the basic operations for all types of vectors. In particular, it does not include division on 16 bit integers. This operation must therefore be emulated using vector float division. This involves several type casting operations and practically reduces the maximum theoretical speedup from 8 to 4.

Tests 5-9 correspond to more complex operations, involving *several* AltiVec instructions. Note that for tests 5 and 7, despite the fact that the operands are vectors of 8 bit integers, the computations are actually carried out on vectors of 16 bit integers, in order to keep a reasonable precision. The theoretical maximum speedup is therefore 8 instead of 16.

6.1. Realistic Case Study. In order to show that EVE can be used to solve realistic problems, while still delivering significant speedups, we have used it to vectorize several complete image processing algorithms. This section describes the implementation of an algorithm performing the detection of *points of interest* in grey scale images using the Harris filter [1].

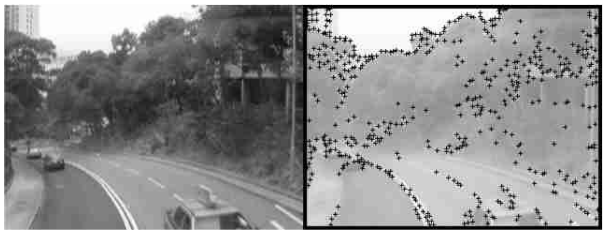
Starting from an input image $I(x, y)$, horizontal and vertical gaussian filters are applied to remove noise and the following matrix is computed:

$$M(x,y) = \begin{pmatrix} (\frac{\partial I}{\partial x})^2 & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & (\frac{\partial I}{\partial y})^2 \end{pmatrix}$$

Where $(\frac{\partial I}{\partial x})$ and $(\frac{\partial I}{\partial y})$ are respectively the horizontal and vertical gradient of $I(x, y)$. $M(x, y)$ is filtered again with a gaussian filter and the following quantity is computed:

$$H(x,y) = Det(M) - k.trace(M)^2, k \in [0.04; 0.06]$$

H is viewed as a measure of pixel interest. Local maxima of H are then searched in 3x3 windows and the n^{th} first maxima are finally selected. Figure 6.1 shows the result of the detection algorithm on a video frame picturing an outdoor scene.



[5]It has more pipeline stages and a shortest cycle time.

In this implementation, only the filtering and the pixel detection are vectorized. Sorting an array cannot be easily vectorized with the AltiVec instruction set. It's not worth it anyway, since the time spent in the final sorting and selection process only accounts for a small fraction (around 3%) of the total execution time of the algorithm. The code for computing M coefficients and H values is shown in Fig. 6.1. It can be split into three sections:

1. **A declarative section** where the needed `matrix` and `filter` objects are instantiated. `matrix` objects are declared as `float` containers to prevent overflow when filtering is applied on the input image and to speed up final computation by removing the need for type casting.

2. **A filtering section** where the coefficients of the M matrix are computed. We use EVE filter objects, instantiated for gaussian and gradient filters. Filter support an overloaded * operator that is semantically used as the composition operator.

3. **A computing section** where the final value of $H(x, y)$ is computed using the overloaded versions of arithmetics operators.

```
// Declarations
#define W 320
#define H 240
matrix<short> I(W,H),a(W,H),b(W,H);
matrix<short> c(W,H),t1(W,H),t2(W,H);
matrix<float> h(W,H);
float k = 0.05f;

filter<3,horizontal> smooth_x = 1,2,1;
filter<3,horizontal> grad_x = 1,0,1;
filter<3,vertical> smooth_y = 1,2,1;
filter<3,vertical> grad_y = -1,0,1;

// Computes matrix M :
//
//          | a c |
//     M =  | c b |

t1 = grad_x(I);
t2 = grad_y(I);
a  = (smooth_x*smooth_y)(t1*t1);
b  = (smooth_x*smooth_y)(t2*t2);
c  = (smooth_x*smooth_y)(t1*t2);

// Computes matrix H
H = (a*b-c*c)-k*(a+b)*(a+b);
```

FIGURE 6.1. *The Harris detector, coded with* EVE

The performances of this detector implementation have been compared to those of the same algorithm written in C, both using 320*240 pixels video sequence as input. The tests were run on a 2GHz Power PC G5 and compiled with `gcc 3.3`. As the two steps of the algorithm (filtering and detection) use two different parts of the E.V.E. API, we give the execution time for each step along with the total execution time.

Step	Execution Time	Speed Up
Filtering	$1.4ms$	5.21
Evaluation	$0.45ms$	4.23
Total Time	$1.85ms$	4.98

7. Related Work. Projects aiming at simplifying the exploitation of the SIMD extensions of modern micro-processors can be divided into two broad categories : compiler-based approaches and library-based approaches.

The SWAR (SIMD Within A register, [15]) project is an example of the first approach. Its goal is to propose a versatile data parallel C language making full SIMD-style programming models effective for commodity microprocessors. An experimental compiler (SCC) has been developed that extends C semantics and type system and can target several family of microprocessors. Started in 1998, the project seems to be in dormant state.

Another example of the compiler-based approach is given by Kyo *et al.* in [14]. They describe a compiler for a parallel C dialect (1DC, One Dimensional C) producing SIMD code for Pentium processors and aimed at the succinct description of parallel image processing algorithms. Benchmarks results show that speed-ups in the range of 2 to 7 (compared with code generated with a conventional C compiler) can be obtained for low-level image processing tasks. But the parallelization techniques described in the work – which are derived from the one used for programming linear processor arrays – seems to be only applicable to simple image filtering algorithms based upon sweeping a horizontal pixel-updating line row-wise across the image, which restricts its applicability. Moreover, and this can be viewed as a limitation of compiler-based approaches, retargeting another processor may be difficult, since it requires a good understanding of the compiler internal representations.

The VAST code optimizer [10] has a specific back-end for generating Altivec/Power PC code. This compiler offers automatic vectorization and parallelization from conventional C source code, automatically replacing loops with inline vector extensions. The speedups obtained with VAST are claimed to be closed to those obtained with hand-vectorized code. VAST is run prior to any compilation. It completely parses the code, extract far more informations than just the Abstract Syntax Tree and output a completely vectorized code. VAST can handle any type of input code and has a large spectrum of solution to vectorize it. At the opposite, E.V.E. is mainly focused on mathematic or image processing optimisation. VAST is a commercial product.

There have been numerous attempts to provide a library-based approach to the exploitation of SIMD features in micro-processors. Apple VECLIB [3], which provides a set of Altivec-optimized functions for signal processing, is an example. But most of these attempts suffer from the weaknesses described in Sect. 2; namely, they cannot handle complex vector expressions and produce inefficient code when multiple vector operations are involved in the same algorithm. MacSTL [11] is the only work we

are aware of that aims at eliminating these weaknesses while keeping the expressivity and portability of a library-based approach. MacSTL is actually very similar to EVE in goals and design principles. This C++ class library provides a fast `valarray` class optimized for Altivec and relies on template-based metaprogramming techniques for code optimization. The only difference is that it only provides STL-compliant functions and operators (it can actually be viewed as a specific implementation of the STL for G4/G5 computers) whereas EVE offers additional domain-specific functions for signal and image processing that can't be optimized as much as with EVE.

8. Conclusion. We have shown how a classical technique – template-based metaprogramming – can be applied to the design and implementation of an efficient high-level vector manipulation library aimed at scientific computing on PowerPC platforms. This library offers a significant improvement in terms of expressivity over the native C API traditionnaly used for taking advantage of the SIMD capabilities of this processor. It allows developers to obtain significant speedups without having to deal with low level implementation details. Moreover, The EVE API is largely compliant with the STL standard and therefore provides a smooth transition path for applications written with other scientific computing libraries. A prototype version of the library can be downloaded from the following URL: http://wwwlasmea.univ-bpclermont.fr/Personnel/Joel.Falcou/eng/eve. We are currently working on improving the performances obtained with this prototype. This involves, for instance, globally minimizing the number of vector load and store operations, using more judiciously Altivec-specific cache manipulation instructions or taking advantage of fused operations (e.g. multiply/add). Finally, it can be noted that, although the current version of EVE has been designed for PowerPC processors with Altivec, it could easily be retargeted to Pentium 4 processors with MMX/SSE2 because the code generator itself (using the expression template mechanism) can be made largely independent of the SIMD instruction set or of any particular parallelization scheme. Porting EVE onto multiprocessors systems or MPI based clusters is not out of question and will surely be a way to explore in the near future.

REFERENCES

[1] C. Harris and M. Stephens. A combined corner and edge detector. In 4th Alvey Vision Conference, 1988.

[2] Apple. The AltiVec Instructions References Page. `http://developer.apple.com/hardware/ve`.

[3] Apple.VecLib framework. `http://developer.apple.com/hardware/ve/vector_libraries.html`

[4] I. Ollman. AltiVec Velocity Engine Tutorial. `http://www.simdtech.org/altivec`. March 2001.

[5] T. Veldhuizen. Using C++ Template Meta-Programs. In C++ Report, vol. 7, p. 36-43,1995.

[6] T. Veldhuizen. Expression Templates. In C++ Report, vol. 7, p. 26-31, 1995.

[7] T. Veldhuizen.Techniques for Scientific C++.http://osl.iu.edu/ tveldhui/papers/techniques/—.

[8] T. Veldhuizen. Arrays in Blitz++. In Dr Dobb's Journal of Software Tools, p. 238-44, 1996.

[9] The BOOST Library. `http://www.boost.org/`.

[10] VAST. `http://www.psrv.com/vast_altivec.html/`.

[11] G. Low. Mac STL. `http://www.pixelglow.com/macstl/`.

[12] The POOMA Library. `http://www.codesourcery.com/pooma/`.

[13] T. Veldhuizen and D. Gannon. Active Libraries: Rethinking the roles of compilers and libraries Proc. of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing. SIAM Press, 1998

[14] S. Kyo and S. Okasaki and I. Kuroda An extended C language and a SIMD compiler for efficient implementation of image filters on media extended micro-processors in Proceedings of Acivs 2003 (Advanced Concepts for Intelligent Vision Systems), Ghent, Belgium, Sept. 1998

[15] The SWAR Home Page http://shay.ecn.purdue.edu/ swar Purdue University

Appendix A: Simple 3x1 gaussian filter written using Altivec C API

```
void AV_filter( char* img, short* res)
{
  vector unsigned char zu8,t1,t2,t3,t4;
  vector signed short x1h,x1l,x2h;
  vector signed short x2l,x3h,x3l;
  vector signed short zs16 ,rh,rl,v0,v1,shift;

  // Generate constants
  v0    = vec_splat_s16(2);
  v1    = vec_splat_s16(4);
  zu8   = vec_splat_u8(0);
  zs16  = vec_splat_s16(0);
  shift = vec_splat_s16(8);

  for( int j = 0; j< SIZE/16 ; j++ )
  {
    // Load input vectors
    t1 =  vec_ld(j*16, img); t2 =  vec_ld(j*16+16, img);

    // Generate shifted vectors
    t3 =  vec_sld(t1,t2,1); t4 =  vec_sld(t1,t2,2);

    // Cast to short
    x1h = vec_mergeh(zu8,t1); x1l = vec_mergel(zu8,t1);
    x2h = vec_mergeh(zu8,t3); x2l = vec_mergel(zu8,t3);
    x3h = vec_mergeh(zu8,t4); x3l = vec_mergel(zu8,t4);

    // Actual filtering
    rh = vec_mladd(x1h,v0,zs16);
    rl = vec_mladd(x1l,v0,zs16);
    rh = vec_mladd(x2h,v1,rh);
    rl = vec_mladd(x2l,v1,rl);
    rh = vec_mladd(x3h,v0,rh);
    rl = vec_mladd(x3l,v0,rl);
    rh = vec_sr(rh,shift);
    rl = vec_sr(rl,shift);

    // Pack and store result vector
    t1 = vec_packsu(rh,rl);
    vec_st(t1,j,out);
  }
}
```