APPLICATION OF TEMPLATE-BASED METAPROGRAMMING COMPILATION TECHNIQUES TO THE EFFICIENT IMPLEMENTATION OF IMAGE PROCESSING ALGORITHMS ON SIMD-CAPABLE PROCESSORS

Joel Falcou and Jocelyn Sérot

falcou,jserot@lasmea.univ-bpclermont.fr
LASMEA, UMR 6602 CNRS/UBP, Clermont-Ferrand, France

ABSTRACT

Microprocessors with SIMD enhanced instruction sets have been proposed as a solution for delivering higher hardware utilization for the most demanding media-processing applications. But direct exploitation of these SIMD extensions places a significant burden on the programmer since it generally involves a complex, low-level and awkward programming style. In this paper, we propose a high-level C++ class library dedicated to the efficient exploitation of these SIMD extensions. We show that the weaknesses traditionally associated with library-based approaches can be avoided by resorting to sophisticated template-based metaprogramming techniques at the implementation level. Benchmarks for a prototype implementation of the library, targeting the PowerPC G4 processor and its Altivec extension are presented. They show that performances close to the one obtained with hand-crafted code can be reached from a formulation offering a significant increase in expressivity.

1. INTRODUCTION

Recently, SIMD enhanced instructions have been proposed as a solution for delivering higher microprocessor hardware utilization. SIMD (Single Instruction, Multiple Data) extensions started appearing in 1994 in HP’s MAX2 and Sun’s VIS 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.

However, to benefit from this acceleration, algorithms have to be recreated so that operations can be applied in parallel to groups of values (vectors) and programs have to be rewritten to make use of the specific SIMD instruction set. This parallelization process can be carried out by the programmer or by a dedicated compiler.

Explicit parallelization (by the programmer) generally offers the best performances. But it is a difficult and error-prone task since it involves identifying SIMD parallelism within loops, rewriting these loops so that they operate on vectors, explicitly handling vector registers allocation and mapping the operations of the algorithm onto the SIMD instruction set. For this reason, this approach cannot be reasonably applied at a large scale and is reserved to simple algorithms and applications.

Dedicated compilers – such as the Intel C compiler for the Pentium SSE – also rely on loop vectorization and are able to automatically extract parallelism from programs written with a conventional sequential language. Although performance may become comparable with the ones obtained with hand-crafted code for some simple, highly regular program patterns, these compilers have difficulties in generating efficient code for more complex applications, involving less regular data access and processing patterns. Moreover, these compilers are generally devoted to one specific processor and are not easily retargeted.

An intermediate solution for taking advantage of SIMD extensions is to rely on a class library providing a high-level view of parallel vectors and a related set of operations, and encapsulating all the low-level details related to actual implementation and manipulation by the SIMD processor extension. With this approach, the programmer is responsible for identifying parallelism in programs but not for the mapping of this parallelism onto the actual SIMD mode of operation. For example, he will write code in the style of Fig. 1.

Vector<float> a(4000), b(4000), c(4000);
Vector<float> r(4000);
r = a/b + c;

Figure 1: Vector processing with a high-level API

It is well known, however, that the code generated by such “naive” class libraries is often very inefficient, due to the unwanted copies caused by temporaries. This has led in turn to the development of Active Libraries [1], 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 SIMD code from a high-level C++ Application Programming Interface (API).
It is organized as follows. Sect. 2 explains why generating efficient code for vector expressions is not trivial and introduces the concept of template-based meta-programming. Sect. 3 explains how this can used to generate optimized SIMD code. Sect. 4 rapidly presents the API of the library we built upon these principles. Sect. 5 gives simple benchmark results while Sect. 6 describes the implementation of a complete image processing application. Sect. 7 is a brief survey of related work and Sect. 8 concludes.

2. TEMPLATE BASED META-PROGRAMMING

The problem of generating efficient code for vector expressions can be understood, in essence, by starting with the code sample given in Fig. 1. Ideally, this code should be inlined as:

```c
for (i=0; i<4000; i++)
    r[i] = a[i] / b[i] + c[i];
```

In practice, due to the way overloaded operators are handled in C++, it is developed as:

```c
Vector<float> _t1(4000), _t2(4000);
for (int i=0; i < 4000; ++i)
    _t1[i] = a[i] / b[i];
for (i=0; i < 4000; ++i)
    _t2[i] = _t1[i] + c[i];
for (i=0; i < 4000; ++i)
    r[i] = _t2[i];
```

The allocation of temporary vectors and the redundant loops result in poor efficiency. For more complex expressions, the performance penalty can easily reach one order of magnitude. In this case, it is clear that expressiveness is obtained at a prohibitive cost. This problem can be overcome by using an advanced C++ technique known as expression templates [2, 3]. The basic idea of expression templates is to create parse trees of vector expressions at compile time and to use these parse trees to generate customized loops. This is actually done in two steps. In the first step, the parse trees – represented as C++ types – are constructed using recursive template instantiations and overloaded versions of the vector operators (\(/\), \(+\), etc.). In the second step, these parse trees are “evaluated” using overloaded versions of the assignment (=) and indexing ([index]) operators to compute the left-hand side vector in a single pass, with no temporary. Both steps rely on the definition of two classes:

- a `Vector` class, for representing application-level vectors; this class has a member `data_[index]` where the array elements are stored and an operator `[]` for accessing these elements:

```c
float Vector::operator[](int index) {
    return data_[index];
}
```

- an `Xpr` class, for representing (encoding) vector expressions:

```c
template<class LHS, class OP, class RHS>
class Xpr {};
```

Consider for example, the statement `r=a/b+c` given in the code sample above. Its right-hand side expression `(a/b+c, where a, b and c have type Vector)` will be encoded with the following C++ type:

```c
Xpr<Xpr<Vector,div,Vector>,add,Vector>
```

This type will be automatically built from the expression syntax `a/b+c` using overloaded versions of the `/` and `+` operators:

```c
template<class T> Xpr<T,div,Vector> operator/(T, Vector) {
    return Xpr<T,div,Vector>();
}
template<class T> Xpr<T,add,Vector> operator+(T, Vector) {
    return Xpr<T,add,Vector>();
}
```

The “evaluation” (at compile time) of the encoded vector expression is carried out by an overloaded version of the assignment operator (=):

```c
template<class T> Vector& Vector::operator=(const T& xpr) {
    for(int i=0;i<size;i++)
        data_[i] = xpr[i];
    return *this;
}
```

For this, the `Xpr` class provides an `operator[]` method, so that each element of the result vector (`data_[i]`) gets the value `xpr[i]`:

```c
template<class LEFT, class OP, class RIGHT>
float X<LEFT,OP,RIGHT>::operator[](int i) {
    return OP::eval(left_[i],right_[i]);
}
```

where `left_` and `right_` are the members storing the left and right sub-expressions of an `Xpr` object and `eval` the static method of the C++ functor associated with the vector operation `OP`. Such a functor will be defined for each possible operation. For example, the `div` and `add` functors associated with the `/` and `+` vector operators are defined as:

```c
class div {
    static float eval(float x, float y) {
        return x/y;
    }
}
```
class add {
    static float eval(float x, float y) {
        return x + y;
    }
};

Using the mechanism described above, a standard C++ compiler can reduce the statement \( r = a/b + c \) to the following "optimal" code:

```cpp
for (i = 0; i < 4000; ++i)
    r[i] = a[i]/b[i] + c[i];
```

### 3. APPLICATION TO THE GENERATION OF EFFICIENT SIMD CODE

The template-based meta-programming technique described in the previous section can readily be adapted to support the generation of efficient code for processors with SIMD extensions. Suppose that:

- `vec_t` is the type of SIMD registers, as viewed by the compiler\(^2\).
- `vec_ld(addr)` is the operation for loading a SIMD register with a vector of data located at address `addr` in memory.
- `vec_st(r, addr)` is the operation for storing a SIMD register in memory at address `addr`.
- `vec_add(r1, r2), vec_mul(r1, r2), etc.` are the SIMD operations for performing addition, multiplication, etc. between SIMD registers.

Then, this adaptation only requires:

- the addition of a method for loading a part of a vector into a SIMD register using the `vec_ld` instruction,
- a modification of the assignment operator, which now uses the `vec_st` instruction to store the result,
- a modification of the `add` (resp. `mul`, etc.) functions, which now use the available `vec_add` (resp. `vec_mul`, etc.) instructions.

```cpp
vec_t Array::load(int index) {
    return vec_ld(data_, index*ELT_SIZE);
}
```

```cpp
template<class T>
Array& Array::operator=(T xpr) {
    for (int i = 0; i < size/4; i++)
        vec_st(xpr.load(i), 0, data);
    return *this;
}
```

```cpp
class add { eval(vec_t x, vec_t y) {
    return vec_add(x,y);
} 
class mul { eval(vec_t x, vec_t y) {
    return vec_mul(x,y);
} 
...
```

With this approach, the compilation of the previous example \( r = a/b + c \) now gives a code equivalent to:

```cpp
for (int i = 0; i < size/4; i++)
    { vec_st(vec_div(vec_ld(a, 16*i),
                   vec_add(vec_ld(b, 16*i),
                            vec_ld(c, 16*i)),
                            0, data);
    }
```

This code is very similar to a hand-crafted one. It is optimal in the sense that it contains the minimum of vector load/store and operations: three vector load operations, one vector addition, one vector multiplication and one vector store.

### 4. THE EVE LIBRARY

As a proof of concept, we have implemented a high-level C++ vector library for exploiting the Altivec extension of the PowerPC processors. This library, called EVE (for Expressive Velocity Engine) can be used to accelerate programs manipulating 1D and 2D arrays and provides specific operations dedicated to signal and image processing. A brief survey of the Altivec is given in Sect. 4.1. The library itself is sketched in Sect. 4.2.

#### 4.1. Altivec

Altivec \(^4\) is an extension designed to enhance PowerPC\(^5\) processor performance. Its 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 or 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

\(^2\) We will make the assumption here that all SIMD operations are accessible through a C API (as for the Altivec extension of PowerPC processors). However, the technique described here remains applicable, though less readable, when these operations must be inserted as inline assembly codes.

\(^4\) PPC 74xx (G4) and PPC 970 (G5).
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 [5].

4.2. EVE API

EVE basically provides two classes, array and matrix, for dealing with 1D and 2D arrays respectively. Its API can be roughly divided in four families:

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

\[
\text{array}\langle\text{char}\rangle\ a(64), b(64), c(64), d(64);
d = (a+b)/c;
\]

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

\[
\text{array}\langle\text{char}\rangle\ a(64), b(64), c(64);
c = \text{where}(a < b, a, b);
\]

// \(c[i] = a[i] < b[i] \ ? \ a[i] : b[i]\)

3. Mathematical and STL functions. These functions work like their STL or \texttt{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. This allows algorithms developed with the STL to be ported (and accelerated) with a minimum effort on a PowerPC platform with EVE. Example:

\[
\text{array}\langle\text{float}\rangle\ a(64), b(64);
b = \tan(a);
\text{float} \ r = \text{inner\_product}(a, b);
\]

// \(r = a[0]*b[0] + \ldots + a[63]*b[63]\)

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

\[
\text{array}\langle\text{float}\rangle\ a(64), b(64);
\text{filter}\langle3, \text{horizontal}\rangle\ \text{gaussian} = 1, 2, 1;
\text{res} = \text{gaussian}(\text{image});
\]

5. BENCHMARKS

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 performance. They have been conducted on a 1.2GHz PowerPC G4 with gcc 3.3.1\textsuperscript{5}. A selection of performance results is given in Table 1. For each test, four numbers are given: the maximum theoretical speedup\textsuperscript{6} (TM), the measured speedup for a hand-coded version of the test using the ALtivec native C API (NC), the measured speedup with a “naive” vector library (NV) – which does not use the expression template mechanism described in Sect. 2, and the measured speedup with the EVE library. 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. Tests 1-2 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 maximum. Tests 3-6 correspond to more complex operations, involving several AltiVec instructions\textsuperscript{7}. Here again, the observed speed-ups are close to the ones obtained with hand-crafted code. By contrast, the performance of the “naive” class library are here disappointing. This clearly demonstrates the effectiveness of the metaprogramming-based optimization.

6. REALISTIC CASE STUDY

In this section we apply EVE to the parallelization of a complete image processing (IP) application. This application, a classic one in the field, aims at detecting point of interest (POIs) using the Harris filter \textsuperscript{6}. It is often used in applications performing 3D video reconstruction, where it provides a set of points for the 3D matching algorithms. Its high demand for computing power – especially when it must be performed in real-time on digital video stream – together with the relatively regular nature of the data

\textsuperscript{4}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.

\textsuperscript{5}With: -faltivec -ftemplate-deph-128 -O3

\textsuperscript{6}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.

\textsuperscript{7}For test 5, 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.
Table 1: Selected performance results

<table>
<thead>
<tr>
<th>Test</th>
<th>Vector type</th>
<th>TM</th>
<th>NC</th>
<th>NV</th>
<th>EVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. v3=v1+v2</td>
<td>8 bit integer</td>
<td>16</td>
<td>15.7</td>
<td>8.0</td>
<td>15.4</td>
</tr>
<tr>
<td>2. v2=tan(v1)</td>
<td>32 bit float</td>
<td>4</td>
<td>3.6</td>
<td>2.0</td>
<td>3.5</td>
</tr>
<tr>
<td>3. v3=inner_prod(v1,v2)</td>
<td>8 bit integer</td>
<td>8</td>
<td>7.8</td>
<td>4.5</td>
<td>7.2</td>
</tr>
<tr>
<td>4. 3x1 FIR</td>
<td>8 bit integer</td>
<td>8</td>
<td>7.9</td>
<td>0.1</td>
<td>7.8</td>
</tr>
<tr>
<td>5. 3x1 FIR</td>
<td>32 bit float</td>
<td>4</td>
<td>3.7</td>
<td>0.1</td>
<td>3.7</td>
</tr>
<tr>
<td>6. v5=sqrt(tan(v1+v2)/cos(v3*v4))</td>
<td>32 bit float</td>
<td>4</td>
<td>3.9</td>
<td>0.04</td>
<td>3.9</td>
</tr>
</tbody>
</table>

access patterns makes it a good candidate for SIMD parallelization.

With this experience, our goals were twofold: first to show that the presence of a few IP-specific operators can greatly ease the writing of these applications; second to demonstrate that this increase in expressivity is not obtained at the expense of efficiency, even when many vector computations are composed within the same application.

6.1. Algorithm

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

\[
M(x, y) = \begin{vmatrix}
\frac{\partial I}{\partial x}^2 & \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} \\
\frac{\partial I}{\partial x} \frac{\partial I}{\partial y} & \frac{\partial I}{\partial y}^2
\end{vmatrix}
\]

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:

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

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.

6.2. Implementation

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 figure 2. 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 supports an overloaded * operator that is semantically used as the composition operator between filters.

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

6.3. Performance

The performance 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 1.2GHz Power PC G4 and compiled with gcc 3.3.1. As the two steps of the algorithm (filtering and detection) use two different parts of the EVE API, we give the execution times for each step along with the total execution time.

The performance of the first step are very satisfactory (82% of the theoretical maximum speed-up). The observed speed-up observed for the second step is more disappointing.
// 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;

// Evaluation of coefficient of M
  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);

// Evaluation of H(x,y)
  h = (a*b-c*c)-k*(a+b)*(a+b);

Figure 2: The Harris detector, coded with EVE

<table>
<thead>
<tr>
<th>Step</th>
<th>Execution Time</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filtering</td>
<td>3.03ms</td>
<td>3.32</td>
</tr>
<tr>
<td>Evaluation</td>
<td>1.08ms</td>
<td>1.18</td>
</tr>
<tr>
<td>Total Time</td>
<td>4.11ms</td>
<td>2.75</td>
</tr>
</tbody>
</table>

Reasons for this have been identified: this is due to multiple redundant vector load and store operations\(^8\). We are currently working in order to fix this. The global performance of the algorithm, however remain satisfactory.

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, [7]) 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 (S\text{CC}) 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 [8]. They describe a compiler for a parallel C dialect (\text{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 \text{VAST} code optimizer [9] has a specific back-end for generating Altivec/PowerPC code. This compiler offers automatic vectorization and parallelization from conventional C source code, automatically replacing loops with inline vector extensions. The speedups obtained with \text{VAST} are claimed to be closed to those obtained with hand-vectorized code. \text{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 \text{VECLIB} [10], 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 \text{EVE} in goals and design principles. This C++ class library provides a fast \text{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 \text{EVE} offers additional domain-specific functions for signal and image processing.

8. CONCLUSION

We have shown how a classical technique – template-based metaprogramming – can be applied to the design and im-

\(^8\)For the evaluation of $H$, the current code generator actually issues eight vector load instructions (three for $a$, three for $b$ and two for $c$). These redundant loads have a very negative effect on Altivec performance, for which the best strategy is to load all at once, then compute and finally write all.
plementation 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 traditionally 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. A prototype version of the library can be downloaded from the following URL: http://lasmea.univ-bpclermont.fr/Personnel/Joel.Falcou/eng/ eve

We are currently working on improving the performance 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.

9. REFERENCES


