Vector Processing in Ada

Franco Gasperoni

ACT Europe, 8 rue de Milan, 75009 Paris, France
gasperon@act-europe.com

Abstract. To handle signal processing algorithms such as the Fast Fourrier Transform (FFT) or the Discrete Cosine Transform (DCT) system designers have traditionally resorted to specialized hardware with built-in vector-processing capabilities. With the advent of the Altivec vector extensions for the general-purpose PowerPC processor, developers have the ability to write efficient signal processing code in C and C++. Using Altivec-enabled PowerPCs as an example, this paper explains what can and should be done to write not only efficient, but also clean and reliable vector processing code in Ada.

1 Introduction

In the 80s fast execution of signal processing algorithms such as the Fast Fourrier Transform (FFT) [1] or the Discrete Cosine Transform (DCT) was relegated to expensive vector supercomputers such as the Crays. The 90s have seen the growth of specialized Digital Signal Processing (DSP) chips to lower the accessibility and cost of these algorithms in general-purpose embedded and non-embedded systems. The advent of modern video/audio signal compression algorithms in MPEG-1, MP3, MPEG-2, and their ubiquitous use in general purpose PCs have fostered the adjunction of vector-like capabilities in conventional general purpose processors such as the PowerPC which is used both in Apple Macs and embedded systems. QuickTime, for instance, has been modified to take advantage of AltiVec extensions that come with the G4 PowerPC microprocessor on the Mac.

To allow developers to code in high-level languages (instead of assembly), Motorola has proposed a C/C++ language interface for Altivec-enabled PowerPCs as well as an Altivec API (Application Programming Interface). With this interface one can write efficient vector processing code in C/C++ but is rather low-level and error-prone. Can Ada help? Can Ada provide an efficient, clean, and reliable way to write vector processing software?

2 The Altivec PowerPC Vector Extensions

The Altivec extension adds Single-Instruction Multiple-Data (SIMD) computing capabilities to the PowerPC, namely:

- 32 vector registers of 128-bits;

– Each vector register can be used as a vector of: 16 signed/unsigned 8-bit integers, or 8 signed/unsigned 16-bit integers, or 4 signed/unsigned 32-bit integers, or 4 IEEE-754 32-bit floats, plus a couple of additional data types such as pixel;
– Well over 70 generic vector processing instructions have been added to manipulate the above Altivec vectors.

The Altivec extensions are described in [2] and their C/C++ interface is summarized below.

3 Summary of Altivec C/C++ Language Interface

3.1 Altivec Data Types Available in C/C++

The Altivec data types that are made available in Motorola's C/C++ programming language extensions are descrived in the following table:

New C/C++ Type	Interpretation	Component Values
vector signed char	16 signed char	-128 .. 127
vector signed short	8 signed short	-32768 .. 32767
vector signed int	4 signed int	-2**31 .. 2**31-1
vector unsigned char	16 unsigned char	0 .. 255
vector unsigned short	8 unsigned short	0 .. 65535
vector unsigned int	4 unsigned int	0 .. 2**32-1
vector bool char	16 unsigned char	0 (False), 255 (True)
vector bool short	8 unsigned short	0 (False), 65535 (True)
vector bool int	4 unsigned int	0 (False), 2**32-1 (True)
vector float	4 float	IEEE-754 values
vector pixel	8 unsigned short	1/5/5/5 pixel

3.2 Alignment Considerations

An Altivec vector object should be aligned on 16-byte boundaries. However, the Altivec technology does not generate alignment exceptions. If the address of an Altivec vector object does not align on a 16-byte boundary, a vector load/store ignores the low-order bits of the address. As a result developers must determine whether and when vector data becomes unaligned.

Fortunately, the Altivec extensions to the PowerPC ABI (Application Binary Interface) provide a number of guarantees to make developers' life easier.

The purpose of the PowerPC ABI is to establish a standard binary interface (register usage conventions, calling conventions, stack frame layout, `setjmp()`, `longjmp()`, etc.) for applications on PowerPC systems.

In the context of Altivec-enabled PowerPCs, some useful extensions to the core PowerPC ABI are:

– The compiler is responsible for aligning vector data types on 16-byte boundaries;

– Records, arrays and unions containing vector types must be aligned on a 16-byte boundaries and their internal organization padded, if necessary, so that each internal vector type is aligned on a 16-byte boundary.

Note that an array of components to be loaded into vector registers need not be aligned but will have to be accessed taking alignment into consideration. Special instructions in the Altivec API are provided to this end. As an example, suppose you have an arbitrary, and potentially non 16-byte aligned array called `input` of `signed char` (i.e. integers spanning -128 .. 127), that needs to be loaded into a vector `v`. That needs to be written as:

```
signed char *input;
//  Possibly unaligned array of signed char

vector signed char v;
//  Output vector containing input[0] through input[15]

vector signed char *temp;
vector signed char shift;
//  Temporaries used to align input's data into v

//  Figure out the alignment correction. 'vec_lvsl' below
//  computes the number of bytes we need to shift the data
//  pointed  by 'temp' so that it becomes 16-byte aligned.

temp  = (vector signed char *) input;
shift = vec_lvsl (0, temp);

//  Here 'shift' contains the permutation vector to do
//  the appropriate shifting so that vector 'v' is
//  16-byte aligned and contains a copy of 'input'.
//  'vec_perm' does the actual permutation by taking two
//  16-byte memory chunks so that the actual vector is
//  inside the union of the two memory chunks.

v = vec_perm (temp[0], temp[1], shift);
```

3.3 Motorola's Altivec C/C++ Language Extensions

Motorola has extended C/C++ as described below. In all the following cases the compiler is required to generate the appropriate Altivec vector handling instructions. Motorola's C/C++ extensions are:

– Assigning two vectors of the same type is allowed. For instance the following code snipet is valid:

```
vector signed int a, b;
a = b;
```

- The comparison operator (`==`), however, is not available for vector types.
- It is possible to initialize a vector with a vector literal. For instance in GCC one can write:

```
vector signed int a = (vector signed int) {1,2,3,4};
```

- One can take the address of a vector object (`&a`) and can dereference a pointer to a vector object. A vector pointer dereference `*p` implies a 128-bit vector load or store from the address obtained by clearing the low order bits of `p`. Access to unaligned memory must be carried out explicitely by the programmer using routines defined in the Altivec API.
- Casts between vector types are allowed. None of the casts performs a conversion: the bit pattern of the result is the same as the bit pattern of the argument that is cast. Casts between vector types and scalar types are illegal.

3.4 Motorola's Altivec API

Motorola has defined an intrinsic API to access all the functionalities provided by the Altivec architecture. There are about 70 generic functions defined in this API which give rise to over 200 unique function names and over 900 overloaded C++ functions once all possible parameter combinations are factored in.

The key point about this API is that although its routines look like regular function calls, in reality these operations are intrinsic to the compiler and generate direct Altivec processor instructions. The mapping is predominantly 1-to-1. As an example the following code snipet gives the profiles of the "Vector Absolute Value" routine available in the intrinsic Altivec API for C++:

```
inline vector signed char  vec_abs (vector signed char a1);
inline vector signed short vec_abs (vector signed short a1);
inline vector signed int   vec_abs (vector signed int a1);
inline vector float        vec_abs (vector float a1);
```

4 Programming the Altivec in Ada

The purpose of this section is to propose a semantic model to make the Altivec PowerPC extensions directly available to Ada developers. Several options are available:

1. Make available in Ada Motorola's C/C++ programming language extensions.

2. Have the compiler recognize computation-intensive loops involving arrays and by using the Altivec vector instructions implement optimizations found in vectorizing compilers as described, for instance, in [4].
3. Create a high-level Ada package that provides general vector processing functionalities such as vector add, multiply, permute, etc. with which developers could easily write FFTs, DCTs, ... that use the underlying vector processing hardware efficiently.

Given what is involved in vectorizing compilers and the amount of work and the limitations involved in approach 2 we will discuss approach 1 in the following sub-sections. Approach 3 will be addressed in the conclusion of this paper.

4.1 Altivec Data Types in Ada

It is tempting to define the Altivec data types as public array types in Ada. This is not the model that is put forth by Motorola and it would mean that all array operations, parameter passing, and attributes would have to be handled explicitely in the Ada compiler which would, for instance, have to convert array indexing operations into a sequence of Altivec calls. This would not only sprinkle Ada front-ends with Altivec specific code, it would be a large undertaking that would go beyond the programming model put forth by Motorola for C/C++.

Semantically Altivec vectors are really 128-bit "blobs" than can only be manipulated through the series of routines provided in the Motorola API. This means that vector types should all be private types. For example:

```
type Vector_Float is private;
--  Corresponds to C/C++ "vector float"
```

These private vector types do not have to be limited since the Motorola spec allows assignment. Direct vector equality could be made abstract to mimic Motorola spec.

For equality Motorola provides a number of vector comparaison routines that should be used for equality depending on whether the vector comparison should return a vector of booleans or a single boolean.

Because of C++'s inability to overload functions on the return type Motorola has probably decided that it would be less confusing that all vector comparisons appear explicitely by means of the corresponding function call in the Altivec API. Thus, in Ada, we could have

```
type Vector_Float is private;
--  Corresponds to C/C++ "vector float"

function "=" (X, Y: Vector_Float) return Boolean is abstract;
```

It would, however, be perfectly reasonable and actually desireable to provide "`=`" in a higher-level Altivec Ada package, where "`=`" would be implemented in terms of the vector routines provided by Motorola. For instance one could write:

```
function "=" (X, Y: Vector_Float) return Boolean is
begin
   return Vec_All_Eq (X, Y);
   --     Vec_All_Eq is intrinsic. The compiler maps it
   --     directly to an Altivec machine instruction.
end "=";

function "=" (X, Y: Vector_Float) return Vector_4_Boolean is
begin
   return Vec_Cmpeq (X, Y);
end "=";
```

where `Vector_4_Boolean` corresponds to C/C++'s type "`vector bool int`". The above is a good example of how Ada's powerful semantic model offers a clear advantage over what is currently available in C/C++.

4.2 Alignment Considerations

Ada's alignment clauses provide a big help and relief to the programmer. Yet another area where, unlike their C/C++ colleagues, Ada programmers get help from the language. In the definition of each Ada vector type the alignment clause will be set to 16. For example we would have:

```
type Vector_Float is private;
for Vector_Float'Alignment use 16;
```

Furthermore every time an Ada developer desires to go back and forth betwen an Altivec vector and a corresponding array it can use a representation clause on the array type and be relieved from tedious alignment fiddling. As an example one could write:

```
type Arr_Float is array (Integer range 0 .. 3) of Float;
for Arr_Float'Alignment use 16;

function UC is new Ada.Unchecked_Conversion (Arr_Float, Vector_Float);
VF : Vector_Float := UC (Arr_Float'(0 .. 3 => 3.141));
```

4.3 Motorola's Altivec API

Doing a clean Ada translation of Motorola's C/C++ API is not completely straightforward since, for instance, to load an array of components into an Altivec vector in C/C++ pointers are used systematically. As an example:

```
inline vector float vec_ld (int a1, float *a2);
```

is used in C++ to load an array of 4 single-precision floats into a vector which is then returned. `a2` is a pointer to a stream of floats and `a1` is an index in this stream from which the copy operation is started. Before doing the copy, address `a2+a1` is 16 byte aligned, which means that if it is not, low-order bits are silently dropped. A similar function in Ada could look like:

```
function Vec_Ld (A1 : Integer; A2 : System.Address)
  return Vector_Float;
pragma Import (Intrinsic, Vec_Ld, "__builtin_altivec_lvx");
```

It would be helpful to also have a version with an array as a parameter, something like:

```
type Arr_Float is array (Integer range <>) of Float;
for Arr_Float'Alignment use 16;

function Vec_Ld (A : Arr_Float; I : Integer)
  return Vector_Float;
function Vec_Ld (A : Arr_Float) return Vector_Float;
pragma Inline (Vec_Ld);
```

In the second version the first parameter has been dropped because one can pass a slice. Of course when doing the above care must be taken that array bounds do not get in the way when making the actual call to the actual Altivec routine. The second routine could for instance be implemented as follows:

```
function Vec_Ld (A : Arr_Float) return Vector_Float is
begin
   pragma Assert (A'Address mod 16 = 0);
   --  No code is generated for this pragma when assertion
   --  checking is off.

   return Vec_Ld (0, A'Address);
   --     Vec_Ld is intrinsic. The compiler maps it
   --     directly to an Altivec machine instruction.
end Vec_Ld;
```

Note that Ada offers the opportunity to use explicit operators such as "`+`" or "`<=`" to define an alias for the corresponding routines in the API. Thus while the API will contain:

```
function Vec_Add (A1, A2: System.Address) return Vector_Float;
pragma Import (Intrinsic, Vec_Add, "__builtin_altivec_vaddfp");

function Vec_Add (A1, A2: Vector_Float) return Vector_Float;
pragma Inline (Vec_Add);
```

```
function Vec_Add (A1, A2: Vector_Float) return Vector_Float is
begin
   pragma Assert (A1'Address mod 16 = 0 and
                  A2'Address mod 16 = 0 );
   --  No code is generated for this pragma when assertion
   --  checking is off.

   return Vec_Add (A1'Address, A2'Address);
   --     Vec_Add is intrinsic. The compiler maps it
   --     directly to an Altivec machine instruction.
end Vec_Ld;
```

the Ada API should also contain:

```
function "+" renames Vec_Add;
```

The best way to organize the Ada implementation of the Altivec API would be to have a low-level Altivec package which is a one to one mapping to Motorola's API and have a higher-level package containing the Ada routines and renamings mentioned above that are not directly required by Motorola's API but are a convenient abstraction of the low-level Altivec routines provided by Motorola.

4.4 Ada 05 Unions

To have the ability to view a vector as both a vector and an array of components it is possible to use Ada unions. Unions are currently supported in GNAT and will probably be available in Ada's future revision: Ada 05. As an example a developer could write:

```
type Arr_Float is array (Integer range 0 .. 3) of Float;
for Arr_Float'Alignment use 16;

type Vector_Rec (Dont_Care : Boolean := False) is record
   case Dont_Care is
      when False =>
         A : Arr_Float;
      when True =>
         V : Vector_Float;
   end case;
end record;
pragma Unchecked_Union (Vector_Rec);

X : Vector_Rec := (A => (10.0, 11.0, 12.0, 13.0));

VF : Vector_Float := X.V;
```

Another way to achive the same effect would be to write something like:

```
A : Arr_Float;
V : Vector_Float;
for V'Address use A'Address;
```

Which is definitely in the clever-but-not-suggested category.

4.5 Vector Initialization

To initialize vector types we can either use the unchecked conversion approach demonstrated in a previous example, or the unchecked union, or directly provide an initialization routine. For instance we could have:

```
function Vec_Init (E1, E2, E3, E4: Float) return Vector_Float;
pragma Inline (Vec_Init);
```

and then write:

```
VF : Vector_Float := Vec_Init (10, 11, 12, 13);
```

4.6 Performance Issues

The main reason for using the Altivec extension in a PowerPC is to improve computational performance. As a result an Ada implementation of the PowerPC API must preserve efficiency. The mapping described above achieves that since the Altivec primitives are wrapped in a layer of Ada code that performs alignment checks only when assertions are enabled and otherwise maps the routine directly to the corresponding Altivec machine instruction. For example when a programmer writes:

```
procedure Proc (A, B : Vector_Float) is
  C : Vector_Float;
begin
  if not A = B then
     C = A + B;
     ...
  end if;
  ...
end Proc;
```

the comparison and the addition are compiled straight into Altivec machine instructions (when assertion cheks are off).

Thus programmers can objtain the same performance for the Altivec in Ada as they do in C, while retaining all of Ada's advantages mentioned in the previous sections.

5 Conclusion

The previous section explains how the Motorola Vector processing specification can be written in Ada to provide a safe and higher-level way to program Altivec-enabled PowerPCs which is as efficient as C.

However, depending on the signal procesing algorithm, the previous approach will still require a fairly low-level of coding for algorithms such as FFTs or DCTs which combine vector data in a non-local fashion. In the FFT, for instance, the famous butterfly permutation combines, in the first iteration, the 0-th element with the $n/2$-th element. Because vector processing hardware basically chops an n-element array into an array of n/s components, where each component fits into a vector register holding s elements, efficiently combining the 0-th array element with the $n/2$-th becomes a low-level programming feat. As an example the last two stages of an FFT must be handcrafted as follows [3]:

```
Data : array (0 .. N) of Vector_Float;

V_11, V_12, V_13, V_14, V_15 : Vector_Float;
V_21, V_22, V_23, V_24, V_25 : Vector_Float;
V_31, V_32, V_33, V_34, V_35 : Vector_Float;
V_41, V_42, V_43, V_44, V_45 : Vector_Float;

Permutation_0101 : constant Vector_Byte
  := Vec_Init (0,1,2,3,4,5,6,7, 0,1,2,3,4,5,6,7);
Permutation_2301 : constant Vector_Byte
  := Vec_Init (8,9,10,11,12,13,14,15, 0,1,2,3,4,5,6,7);
Permutation_3232 : constant Vector_Byte
  := Vec_Init (12,13,14,15,8,9,10,11, 12,13,14,15,8,9,10,11);

X : constant Vector_Float := Vec_Init (1.0,  1.0, -1.0, -1.0);
Y : constant Vector_Float := Vec_Init (1.0, -1.0, -1.0,  1.0);

Block : Unsigned_Integer := 0;
...
loop
   exit when Block >= (N + 1)/2;

   V_13 := Data (Block  ) - Data (Block+1);
   V_11 := Data (Block  ) + Data (Block+1);
   V_23 := Data (Block+2) - Data (Block+3);
   V_21 := Data (Block+2) + Data (Block+3);
   V_33 := Data (Block+4) - Data (Block+5);
   V_31 := Data (Block+4) + Data (Block+5);
   V_43 := Data (Block+6) - Data (Block+7);
   V_41 := Data (Block+6) + Data (Block+7);
```

```
   V_15 := Vec_Permute (V_13, V_13, Permutation_3232);
   V_14 := Vec_Permute (V_13, V_13, Permutation_0101);
   V_12 := Vec_Permute (V_11, V_11, Permutation_2301);
   V_25 := Vec_Permute (V_23, V_23, Permutation_3232);
   V_24 := Vec_Permute (V_23, V_23, Permutation_0101);
   V_22 := Vec_Permute (V_21, V_21, Permutation_2301);
   V_35 := Vec_Permute (V_33, V_33, Permutation_3232);
   V_34 := Vec_Permute (V_33, V_33, Permutation_0101);
   V_32 := Vec_Permute (V_31, V_31, Permutation_2301);
   V_45 := Vec_Permute (V_43, V_43, Permutation_3232);
   V_44 := Vec_Permute (V_43, V_43, Permutation_0101);
   V_42 := Vec_Permute (V_41, V_41, Permutation_2301);

   Data (Block  ) := Vec_Multiply_Add (V_11, X, V_12);
   Data (Block+1) := Vec_Multiply_Add (V_15, Y, V_14);
   Data (Block+2) := Vec_Multiply_Add (V_21, X, V_22);
   Data (Block+3) := Vec_Multiply_Add (V_25, Y, V_24);
   Data (Block+4) := Vec_Multiply_Add (V_31, X, V_32);
   Data (Block+5) := Vec_Multiply_Add (V_35, Y, V_34);
   Data (Block+6) := Vec_Multiply_Add (V_41, X, V_42);
   Data (Block+7) := Vec_Multiply_Add (V_45, Y, V_44);

   Block := Block + 8;
end loop;
```

This code is more readable than the corresponding C code, but remains low-level. As a result an interesting research question is whether it is possible to write an Ada vector package with a number of high-level vector operators which, when combined, allow the simple writing of *efficient* signal processing algorithms such as FFTs, and DCTs on the Altivec.

References

1. Cormen, Leiserson, Rivest, *Introduction to Algorithms*, the MIT Press, 1990.
2. Motorola, *AltiVec Technology - Programming Interface Manual*, http://e-www.motorola.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
3. Motorola, *Complex Floating Point Fast Fourier Transform Optimization for AltiVec*, Application Note AN2115/D Rev. 2.1, 6/2003.
4. Wolfe, *Optimizing Supercompilers for Supercomputers*, the MIT Press, 1989.