Writing C++ Wrappers for SIMD Intrinsics (3)

Oct 10th, 2014

2. First version of wrappers

Now that we know a little more about SSE and AVX, let's write some code; the wrappers will have a data vector member and provide arithmetic, comparison and logical operators overloads. Throughout this section, I will mainly focus on vector4f, the wrapper around __m128, but translating the code for other data vectors should not be difficult thanks to the previous section. Since the wrappers will be used as numerical types, they must have value semantics, that is they must define copy constructor, assignment operator and non-virtual destructor.

2.1 Initialization and assignment

SSE and AVX data vectors can be initialized from different inputs: a single value for all elements, a value per element, or another data vector.

simd_sse.hpp

```
1  class vector4f
2  {
3  public:
4
5      inline vector4f() {}
6      inline vector4f(float f) : m_value(_mm_set1_ps(f)) {}
7      inline vector4f(float f0, float f1, float f2, float f3) : m_value(_mm_setr_ps(f0,f1,f2,f3)) {}
8      inline vector4f(const __m128& rhs) : m_value(rhs) {}
9
10     inline vector4f& operator=(const __m128& rhs)
11     {
12         m_value = rhs;
13         return *this;
14     }
15
16     inline vector4f(const vector4f& rhs) : m_value(rhs.m_value) {}
17
18     inline vector4f& operator=(const vector4f& rhs)
19     {
20         m_value = rhs.m_value;
21         return *this;
22     }
```

```
23
24 private:
25
26     __m128 m_value;
27 };
28
```

2.2 Implicit conversion

The operators overloads have to access the m_value member of the wrapper so they can pass it as an argument to the intrinsic functions:

overload sample

```
1
2 vector4f operator+(const vector4f& lhs, const vector4f& rhs)
3 {
4     return _mm_add_ps(lhs.m_value,rhs.m_value);
5 }
```

We could declare the operator overloads as friend functions of the wrapper class, or provide a get method returning the internal m_value. Both of these solutions work, but aren't elegant: the first requires a huge amount of friend declarations, the second produces heavy code unpleasant to read.

A more elegant solution is to provide a conversion operator from vector4f to __m128; since vector4f can be implicitly converted from __m128, we can now use vector4f or __m128 indifferently. Moreover we can save the vector4f copy constructor and assignment operator:

simd_sse.hpp

```
1
2  class vector4f
3  {
4  public:
5
6      inline vector4f() {}
7      inline vector4f(float f) : m_value(_mm_set1_ps(f)) {}
8      inline vector4f(float f0, float f1, float f2, float f3) : m_value(_mm_setr_ps(f0,f1,f2,f3)) {}
9      inline vector4f(const __m128& rhs) : m_value(rhs) {}
10
11     inline vector4f& operator=(const __m128& rhs)
12     {
13         m_value = rhs;
14         return *this;
15     }
16
17     inline operator __m128() const { return m_value; }
18
19     // vector4f(const vector4f&) and operator=(const vector4f&) are not required anymore:
20     // the conversion operator will be called before calling vector4f(const __m128&)
21     // or operator=(const __m128&)
22
23 private:
24
25     __m128 m_value;
26 };
```

2.3 Arithmetic operators overloads

Next step is to write the arithmetic operators overloads. The classic way to do this is to write computed assignment operators and to use them in operators overloads, so they don't have to access private members of vector4f; but since vector4f can be implicitly converted to __m128, we can do the opposite and avoid using a temporary (this won't have any impact on performance since the compiler can optimize it, but produces shorter and more pleasant code to read):

simd_sse.hpp

```
class vector4f
{
public:

    // ...

    inline vector4f& operator+=(const vector4f& rhs)
    {
        *this = *this + rhs;
        return *this;
    }
};

inline vector4f operator+(const vector4f& lhs, const vector4f& rhs)
{
    return _mm_add_ps(lhs,rhs);
}
```

2.4 The need for a base class

We could go ahead and write the remaining arithmetic operators overloads, just as we did before:

simd_sse.hpp

```
vector4f operator+(const vector4f&, const vector4f&);
// Adds the same float value to each data vector member
vector4f operator+(const vector4f&, const float&);
vector4f operator+(const float&, const vector4f&);

// Similar for operator-, operator* and operator/
// ...

vector4f operator-(const vector4f&);

vector4f& operator++();
vector4f operator++(int);

// Similar for operator--
// ...
```

But wait! Whenever you add a new wrapper, you'll have to write these operators overloads again. Besides the fact that you will need to type a lot of boilerplate code, computed assignment operators will be the same as those of vector4f (that is, invoke the corresponding operator overload and return the object), and even some operators overloads will have the same code as the one of vector4f operators. Code duplication is never good, and we should look for ways to avoid it.

If we had encountered this problem for classes with entity semantics, we would have captured the common code into a base class, and delegate the specific behavior to virtual methods, a typical use of classical dynamic polymorphism. What we need here is an equivalent architecture for classes with value semantics and no virtual methods (since virtual assignment operators are nonsense). This equivalent architecture is the CRTP (Curiously Recurring Template Pattern). A lot has been written about CRTP and I will not dwell on it. If you don't know about this pattern, the most important

thing to know is CRTP allows you to invoke methods of inheriting classes from the base class just as you would do through virtual methods, except the target methods are resolved at compile time.

Let's call our base class simd_vector, it will be used as base class for every wrapper; here is what it should look like:

simd_base.hpp

```
template <class X>
    struct simd_vector_traits;

template <class X>
    class simd_vector
    {
    public:

        typedef typename simd_vector_traits<X>::value_type value_type;

        // downcast operators so we can call methods in the inheriting classes
        inline X& operator()() { return *static_cast<X*>(this); }
        inline const X& operator()() const { return *static_cast<const X*>(this); }

        // Additional assignment operators
        inline X& operator+=(const X& rhs)
        {
            (*this)() = (*this)() + rhs;
            return (*this)();
        }

        inline X& operator+=(const value_type& rhs)
        {
            (*this)() = (*this)() + X(rhs);
            return (*this)();
        }

        // Same for operator-=, operator*=, operator/= ...
        // ...

        // Increment operators
        inline X operator++(int)
        {
            X tmp = (*this)();
            (*this) += value_type(1);
            return tmp;
        }

        inline X& operator++()
        {
            (*this)() += value_type(1);
            return (*this)();
        }

        // Similar decrement operators
        // ...

    protected:

        // Ensure only inheriting classes can instantiate / copy / assign simd_vector.
        // Avoids incomplete copy / assignment from client code.
        inline simd_vector() {}
        inline ~simd_vector() {}

        inline simd_vector(const simd_vector&) {}
        inline simd_vector& operator=(const  simd_vector&) { return *this; }
    };

template <class X>
    inline simd_vector<X> operator+(const simd_vector<X>& lhs,
```

```
61                                      const typename simd_vector_traits<X>::type& rhs)
62      {
63          return lhs() + X(rhs);
64      }
65
66  template <class X>
67      inline simd_vector<X> operator+(const typename simd_vector_traits<X>::type& lhs,
68                                      const simd_vector<X>& rhs)
69      {
70          return X(lhs) + rhs();
71      }
72
73  // Same for operator-, operator*, operator/
74  // ...
75
```

Now, all vector4f needs to do is to inherit from simd_vector and implement the traditional operator+, and it will get += and ++ operators overloads for free (and the same for other arithmetic operators):

simd_sse.hpp

```
1   class vector4f : public simd_vector<vector4f>
2   {
3   public:
4
5       inline vector4f() {}
6       inline vector4f(float f) : m_value(_mm_set1_ps(f)) {}
7       inline vector4f(float f0, float f1, float f2, float f3) : m_value(_mm_setr_ps(f0,f1,f2,f3)) {}
8       inline vector4f(const __m128& rhs) : m_value(rhs) {}
9
10      inline vector4f& operator=(const __m128& rhs)
11      {
12          m_value = rhs;
13          return *this;
14      }
15
16      inline operator __m128() const { return m_value; }
17
18      // No more operator+= since it is implemented in the base class
19
20  private:
21
22      __m128 m_value;
23  };
24
25  // Based on this operator implementation, simd_vector<vector4f> will generate
26  // the following methods and overloads:
27  // vector4f& operator+=(const vector4f&)
28  // vector4f operator++(int)
29  // vector4f& operator++()
30  // vector4f operator+(const vector4f&, ocnst float&)
31  // vector4f operator+(const float&, const vector4f&)
32  inline vector4f operator+(const vector4f& lhs, const vector4f& rhs)
33  {
34      return _mm_add_ps(lhs,rhs);
35  }
36
37  inline vector4f operator-(const vector4f& lhs, const vector4f& rhs)
38  {
39      return _mm_sub_ps(lhs,rhs);
40  }
41
42  inline vector4f operator*(const vector4f& lhs, const vector4f& rhs)
43  {
44      return _mm_mul_ps(lhs,rhs);
45  }
```

```
46
47 inline vector4f operator/(const vetcor4f& lhs, const vector4f& rhs)
48 {
49     return _mm_div_ps(lhs,rhs);
50 }
51
```

Looks good, doesn't it ? Every time we want to implement a new wrapper, we only have to code 4 operators and make our class inherit from simd_vector, and all overloads will be generated for free!

Just one remark before we continue with comparison operators; if you have noticed, the base class simd_vector defines a type named value_type, depending on the nature of the inheriting class (float for vector4f, double for vector2d, …). However, this type is not defined by the inheriting class, but by a traits class instead. This is a constraint of the CRTP pattern: you can access the inheriting class as long the compiler doesn't instantiate the code; if you call a method defined in the inheriting class, the compiler will assume it exists until it has to instantiate the code. But type resolution is different and you have to define it out of the inheriting class. This is one reason for the existence of the simd_vector_traits class. Other reasons will be discussed in a later section. Note the class containing the type definition doesn't have to be fully defined at this point: a simple forward declaration is sufficient.

EDIT 20/11/2014: it seems the CRTP layer introduces a slight overhead (at least with GCC), see this article for more details and an alternative solution.

2.5 Comparison operators

Since ordinary comparison operators return boolean value, we need to implement SIMD wrappers for booleans. The number of boolean elements of the wrappers will be directly related to the number of floating values wrapped by our arithmetic wrappers.

In order not to duplicate code, we'll use the same architecture as for arithmetic wrappers: a CRTP with base class for common code, and inheriting classes for specific implementation. Here is the implementation of the simd_vector_bool class, the base used to generate bitwise assignment operators and logical operators overloads in inheriting classes:

simd_base.hpp

```
1  template <class X>
2      class simd_vector_bool
3      {
4      public:
5
6          inline X& operator()() { return *static_cast<X*>(this); }
7          inline const X& operator()() const { return *static_cast<const X*>(this); }
8
9          inline X& operator&=(const X& rhs)
10         {
11             (*this) = (*this) && rhs;
12             return (*this)();
13         }
14
15         inline X& operator|(const X& rhs)
16         {
17             (*this)() = (*this) || rhs;
18             return (*this)();
19         }
20
21         inline X& operator^=(const X& rhs)
22         {
23             (*this)() = (*this)() ^ rhs;
24             return (*this)();
25         }
26
27     protected:
```

```
28
29          inline simd_vector_bool() {}
30          inline ~simd_vector_bool() {}
31
32          inline simd_vector_bool(const simd_vector_bool&) {}
33          inline simd_vector_bool& operator=(const simd_vector_bool&) { return *this; }
34      };
35
36  template <class X>
37      inline X operator&&(const simd_vector_bool<X>& lhs, const simd_vector_bool<X>& rhs)
38      {
39          return lhs() & rhs();
40      }
41
42  template <class X>
43      inline X operator&&(const simd_vector_bool<X>& lhs, bool rhs)
44      {
45          return lhs() & rhs;
46      }
47
48  template <class X>
49      inline X operator||(bool lhs, const simd_vector_bool<X>& rhs)
50      {
51          return lhs & rhs();
52      }
53
54  // Similar for operator|| overloads
55  // ...
56
57  template <class X>
58      inline X operator!(const simd_vector_bool<X>& rhs)
59      {
60          return rhs() == 0;
61      }
62
```

The inheriting class vector4fb only has to provide bitwise operators and equality/inequality operators:

simd_sse.hpp

```
1   class vector4fb : public simd_vector_bool<vector4fb>
2   {
3   public:
4
5       inline vector4fb() {}
6       inline vector4fb(bool b) : m_value(_mm_castsi128_ps(_mm_set1_epi32(-(int)b))) {}
7       inline vector4fb(bool b0, bool b1, bool b2, bool b3)
8       : m_value(_mm_castsi128_ps(_mm_setr_epi32(-(int)b0,-(int)b1,-(int)b2,-(int)b3))) {}
9
10      inline vector4fb(const __m128& rhs) : m_value(rhs) {}
11
12      inline vector4fb& operator=(const __m128& rhs)
13      {
14          m_value = rhs;
15          return *this;
16      }
17
18      inline operator __m128() const { return m_value; }
19
20  private:
21
22      __m128 m_value;
23  };
24
25  inline vector4fb operator&(const vector4fb& lhs, const vector4fb& rhs)
26  {
```

```
27      return _mm_and_ps(lhs,rhs);
28  }
29
30  inline vector4fb operator|(const vector4fb& lhs, const vector4fb& rhs)
31  {
32      return _mm_or_ps(lhs,rhs);
33  }
34
35  inline vector4fb operator^(const vector4fb& lhs, const vector4fb& rhs)
36  {
37      return _mm_xor_ps(lhs,rhs);
38  }
39
40  inline vector4fb operator~(const vector4fb& rhs)
41  {
42      return _mm_xor_ps(rhs,_mm_castsi128_ps(_mm_set1_epi32(-1)));
43  }
44
45  inline vector4fb operator==(const vector4fb& lhs, const vector4fb& rhs)
46  {
47      return _mm_cmeq_ps(lhs,rhs);
48  }
49
50  inline vector4fb operator!=(const vector4f& lhs, const vector4fb& rhs)
51  {
52      return _mm_cmpneq_ps(lhs,rhs);
53  }
54
```

Now that we have wrappers for boolean, we can add the comparison operators to the vector4f class; again, to avoid code duplication, some operators will be implemented in the base class and will be based on specific operators implemented in the inheriting class. Let's start with the vector4f comparison operators:

simd_sse.hpp

```
1
2   // Definition of vector4f and arithmetic overloads
3   // ...
4   inline vector4fb operator==(const vector4f& lhs, const vector4f& rhs)
5   {
6       return _mm_cmpeq_ps(lhs,rhs);
7   }
8
9   inline vector4fb operator!=(const vector4f& lhs, const vector4f& rhs)
10  {
11      return _mm_cmpneq_ps(lhs,rhs);
12  }
13
14  inline vector4fb operator<(const vector4f& lhs, const vector4f& rhs)
15  {
16      return _mm_cmplt_ps(lhs,rhs);
17  }
18
19  inline vector4fb operator<=(const vector4f& lhs, const vector4f& rhs)
20  {
21      return _mm_cmple_ps(lhs,rhs);
22  }
```

Before we implement operator> and operator>= for the base class, we have to focus on their return type. If these operators were implemented for vector4f, we would have return vector4fb; but since they are implemented for the base class, they need to return the boolean wrapper related to the arithmetic wrapper, i.e the inheriting class. What we need here is to provide a mapping between arithmetic wrapper type and boolean wrapper type somewhere. Remember the simd_vector_traits structure we declared to define our value_type ? It would be the perfect place for defining that mapping:

simd_sse.hpp

```
// simd_vector_traits<vector4f> must be defined before vector4f so simd_vector can compile
// (remember we use simd_vector_traits<X>::value_type in the definition of simd_vector).
class vector4f;

// Full specialization of the template simd_vector_traits declared in simd_base.hpp
template <>
    struct simd_vector_traits<vector4f>
    {
        typedef float value_type;
        typedef simd_vector4fb vector_bool;
    };

class vector4f
{
    // ...
```

A last remark before we add the last comparison operators: since the template simd_vector_traits will never be defined but fully specialized instead, there is no risk we forget to define it when we add a new wrapper, we'll have a compilation error.

Finally, we can add the missing operators for the base class:

simd_base.hpp

```
// Declaration of simd_vector and operators
//...

template <class X>
    inline typename simd_vector_traits<X>::vector_bool
    operator>(const simd_vector<X>& lhs, const simd_vector<X>& rhs)
    {
        return rhs() <= lhs();
    }

template <class X>
    inline typename simd_vector_traits<X>::vector_bool
    operator>=(const simd_vector<X>& lhs, const simd_vector<X>& rhs)
    {
        return rhs() < lhs();
    }
```

2.6 Logical operators

Since float provides logical operators, our wrapper should do so. The implementation is the same as for the simd_vector_bool class, that is logical assignment operator in the simd_vector base class, and operator overloads for the inheriting classes. The implementation of operator|, operator&, operator^ and operator~ is the same as the one for vector4fb, so I don't repeat it here.

2.7 Next step

Next step is to implement wrapper for 2 double, then wrapper for 8 float and 4 double if you want to support AVX. You can also implement wrappers for int if you aim to do integre computation. The implementation is similar to what has been done in this section.

Now we have nice wrappers, we'll see in the next section how to plug them into existing code.

Posted by Johan Mabille Oct 10th, 2014 SIMD, Vectorization