Writing C++ Wrappers for SIMD Intrinsics (5)

Oct 25th, 2014

4. Making the code more generic

In the previous section we saw how to plug the wrappers into existing code and ended up with the following loop:

sample.cpp

```
1  std::vector<float> a, b, c, d, e
2  // Somewhere in the code the vectors are resized
3  // so they hold n elements
4  for(size_t i = 0; i < n/4; i+=4)
5  {
6      vector4f av; av.load_a(&a[i]);
7      vector4f bv; bv.load_a(&b[i]);
8      vector4f cv; cv.load_a(&c[i]);
9      vector4f dv; dv.load_a(&d[i]);
10
11     vector4f ev = av*bv + cv*dv;
12     ev.store_a(&e[i]);
13 }
14 // Remaining part of the loop
15 // ...
16
```

As said in the previous section, the first problem of this code is its lack of genericity; we are highly coupled with the SIMD instruction set wrapped, and replacing it with another one requires code changes we should avoid. If we want to make the code independant from the SIMD instruction set and the related wrapper, we need to hide the specifics of this instruction set, that is, the vector type and its size (the number of scalars it holds).

4.1 Hiding the wrapper type

We want to be able to select the right wrapper depending on the scalar type and the instruction set used. When talking about selecting a type depending on another one, the first thing that comes to mind is type traits. Here our traits must contain the wrapper type and its size associated with the scalar type used:

simd_traits.hpp

```
1 template <class T>
2     struct simd_traits
```

```
    {
        typedef T type;
        static const size_t size = 1;
    };

```

The general definition of the traits class allows us to write code that works even for types that don't have related wrappers (numerical types defined by another user for instance). Then we need to specialize these definitions for float and double, depending on the considered instruction set. Assume we can detect the instruction set available on our system and save this information in a macro (we'll see how to do that in a later section). The specialization of the traits class will look like:

simd.hpp

```
#ifdef USE_SSE
template <>
    struct simd_traits<float>
    {
        typedef vector4f type;
        static const size_t size = 4;
    };

template <>
    struct simd_traits<double>
    {
        typedef vector2d type;
        static const size_t size = 2;
    };
#elif USE_AVX
template <>
    struct simd_traits<float>
    {
        typedef vector8f type;
        static const size_t size = 8;
    };

template <>
    struct simd_traits<double>
    {
        typedef vector4d type;
        static const size_t size = 4;
    };
#endif
```

Now we can adapt the loop so it doesn't explicitly refer to the vector4f type:

sample.cpp

```
std::vector<float> a,b,c,d,e;
// ... resize a, b, c, d, and e so they hold n elements
typedef simd_traits<float>::type vec_type;
size_t vec_size = simd_traits<float>::size;
for(size_t i = 0; i < n/vec_size; i += vec_size)
{
    vec_type av; av.load_a(&a[i]);
    vec_type bv; bv.load_a(&b[i]);
    vec_type cv; cv.load_a(&c[i]);
    vec_type dv; dv.load_a(&d[i]);

    vec_type ev = av*bv + cv*dv;
    ev.store_a(&e[i]);
}
// Remaining part of the loop
// ...
```

That's it! If we need to compile this code on a system where AVX is available, we have nothing to do. The macro USE_AVX will be defined, the specialization of simd_traits with vector8f as inner type will be instantiated, and the loop will use the vector8f wrapper and the AVX intrinsics. However, there's still a problem: we can migrate to any SIMD instruction set for which a wrapper is available, but we can't use types that don't have related wrappers. The simd_traits works fine even for user defined types, but the load and store functions are available for wrappers only. We need to provide generic versions of these functions that work with any type.

4.2 Generic load an store functions

Actually, all we have to do is to provide two versions of these functions: one for types that don't have related wrappers, and one that works with wrappers. Template specialization can be of help here, but since partial specialization is not possible for functions, let's wrap them into a simd_functions_invoker class:

simd.hpp

```
// Common implementation for types that support vectorization
template <class T, class V>
    struct simd_functions_invoker
    {
        inline static V
        set1(const T& a) { return V(a); }

        inline static V
        load_a(const T* src) { V res; res.load_a(src); return res; }

        inline static V
        load_u(const T* src) { V res; res.load_u(src); return res; }

        inline static void
        store_a(T* dst, const V& src) { src.store_a(dst); }

        inline static void
        store_u(T* dst, const V& src) { src.store_u(dst); }
    };

// Specialization for types that don't support vectorization
template <class T>
    struct simd_functions_invoker<T,T>
    {
        inline static T
        set1(const T& a) { return T(a); }

        inline static T
        load_a(const T* src) { return *src; }

        inline static T
        load_u(const T* src) { return *src; }

        inline static void
        store_a(T* dst, const T& src) { *dst = src; }

        inline static void
        store_u(T* dst, const T& src) { *dst = src; }
    };
```

We've added the set1 function so we can intialize wrappers and scalar type from a single value in an uniform way. Calling the generic functions would look like:

sample.cpp

```
typedef simd_traits<float>::simd_type vec_type;
vec_type va = simd_functions_invoker<float,vec_type>::load_a(a);
```

That's too much verbose. Let's add façade functions that deduce template parameters for us:

simd.hpp

```
template <class T> inline typename simd_traits<T>::type
set1(const T& a)
{ return simd_functions_invoker<T,typename simd_traits<T>::type>::set1(a); }

template <class T> inline typename simd_traits<T>::type
load_a(const T* src)
{ return simd_functions_invoker<T,typename simd_traits<T>::type>::load_a(src); }

template <class T> inline typename simd_traits<T>::type
load_u(const T* src)
{ return simd_functions_invoker<T,typename simd_traits<T>::type>::load_u(src); }

template <class T> inline void
store_a(T* dst, const typename simd_traits<T>::type& src)
{ simd_functions_invoker<T,typename simd_traits<T>::type>::store_a(dst,src); }

template <class T> inline void
store_u(T* dst, const typename simd_traits<T>::type& src)
{ simd_functions_invoker<T,typename simd_traits<T>::type>::store_u(dst,src); }
```

Now we can use these generic functions in the previous loop so it works with any type, even those that don't support vectorization:

sample.cpp

```
std::vector<float> a,b,c,d,e;
// ... resize a, b, c, d, and e so they hold n elements
typedef simd_traits<float>::type vec_type;
size_t vec_size = simd_traits<float>::size;
for(size_t i = 0; i < n/vec_size; i += vec_size)
{
    vec_type av = load_a(&a[i]);
    vec_type bv = load_a(&b[i]);
    vec_type cv = load_a(&c[i]);
    vec_type dv = load_a(&d[i]);

    vec_type ev = av*bv + cv*dv;
    store_a(&e[i],ev);
}
// Remaining part of the loop
// ...
```

Or, if you want to be more concise:

sample.cpp

```
std::vector<float> a,b,c,d,e;
// ... resize a, b, c, d, and e so they hold n elements
typedef simd_traits<float>::type vec_type;
size_t vec_size = simd_traits<float>::size;
for(size_t i = 0; i < n/vec_size; i += vec_size)
{
    vec_type ev = load_a(&a[i])*load_a(&b[i]) + load_a(&c[i])*load_a(&d[i]));
    store_a(&e[i], ev);
}
// Remaining part of the loop
// ...
```

We've reached our goal, we can use intrinsics almost like floats; in a real application code, it is likely that you initialize the wrappers through load functions, then perform the computations and finally store the results (like in the not concise version of the generic loop); thus the only difference between classical code and code with SIMD wrappers is the initialization and storing of wrappers (and eventually the functions signatures if you want to pass wrappers instead of scalars), the other parts should be exactly the same and the code remains easy to read and to maintain.

4.3 Detecting the supported instruction set

Until now, we've assumed we were able to detect at compile time the available instruction sets. Let's see now how to achieve this. Compilers often provide preprocessor tokens depending on the available instruction sets, but these tokens may vary from one compiler to another, so we have to standardize that. On most 64-bit compilers, the tokens look like **__SSE__** or **__SSE3__**, on 32-bit systems, Microsoft compilers set the preprocessor token **_M_IX86_FP** to 1 for SSE (vectorization of float) and 2 for SSE2 (vetorization of double and integers).

Here is how we can standardize that:

simd_config.hpp

```
#if (defined(_M_AMD64) || defined(_M_X64) || defined(__amd64)) && ! defined(__x86_64__)
    #define __x86_64__ 1
#endif

// Find sse instruction set from compiler macros if SSE_INSTR_SET not defined
// Note: Not all compilers define these macros automatically
#ifndef SSE_INSTR_SET
    #if defined ( __AVX2__ )
        #define SSE_INSTR_SET 8
    #elif defined ( __AVX__ )
        #define SSE_INSTR_SET 7
    #elif defined ( __SSE4_2__ )
        #define SSE_INSTR_SET 6
    #elif defined ( __SSE4_1__ )
        #define SSE_INSTR_SET 5
    #elif defined ( __SSSE3__ )
        #define SSE_INSTR_SET 4
    #elif defined ( __SSE3__ )
        #define SSE_INSTR_SET 3
    #elif defined ( __SSE2__ ) || defined ( __x86_64__ )
        #define SSE_INSTR_SET 2
    #elif defined ( __SSE__ )
        #define SSE_INSTR_SET 1
    #elif defined ( _M_IX86_FP )           // Defined in MS compiler on 32bits system. 1: SSE, 2: SSE2
        #define SSE_INSTR_SET _M_IX86_FP
    #else
        #define SSE_INSTR_SET 0
    #endif // instruction set defines
#endif // SSE_INSTR_SET
```

Now we can use the **SSE_INSTR_SET** token to include the right file:

simd_config.hpp

```
#// Include the appropriate header file for intrinsic functions
#if SSE_INSTR_SET > 7                  // AVX2 and later
    #ifdef __GNUC__
        #include <x86intrin.h>         // x86intrin.h includes header files for whatever instruction
                                       // sets are specified on the compiler command line, such as:
                                       // xopintrin.h, fma4intrin.h
    #else
        #include <immintrin.h>         // MS version of immintrin.h covers AVX, AVX2 and FMA3
    #endif // __GNUC__
#elif SSE_INSTR_SET == 7
    #include <immintrin.h>             // AVX
#elif SSE_INSTR_SET == 6
    #include <nmmintrin.h>             // SSE4.2
```

```
#elif SSE_INSTR_SET == 5
    #include <smmintrin.h>            // SSE4.1
#elif SSE_INSTR_SET == 4
    #include <tmmintrin.h>            // SSSE3
#elif SSE_INSTR_SET == 3
    #include <pmmintrin.h>            // SSE3
#elif SSE_INSTR_SET == 2
    #include <emmintrin.h>            // SSE2
#elif SSE_INSTR_SET == 1
    #include <xmmintrin.h>            // SSE

```

Note that if you split the implementation of SSE wrappers and AVX wrappers into different files, you can also use the **SSE_INSTR_SET** token to include the implementation file in the simd.hpp file:

simd.hpp

```
#include "simd_config.hpp"
#if SSE_INSTR_SET > 6
    #include "simd_avx.hpp"
#endif
#if SSE_INSTR_SET > 0
    #include "simd_sse.hpp"
#endif

// Definition of traits and generic load and store functions
// ...
```

Now from the client code, the only file to include is simd.hpp, and everything will be available.

4.4 Going further

Now that we have nice wrappers providing basic functionalities, what could be the next step ? Well, first we could add a method to retrieve an element in the vector:

simd_base.hpp

```
template <class X>
    class simd_vector
    {
    public:

        typedef simd_traits<X>::value_type value_type;

        // ...

        value_type operator[](size_t index) const
        {
            size_t size = simd_traits<X>::size;
            value_type v[size];
            (*this)().store_u(v);
            return v[index];
        }
    };
```

We can add horizontal add function, useful for linear algebra products:

simd_sse.hpp

```
    inline float hadd(const vector4f& rhs)
    {
    #if SSE_INSTR_SET >= 3 // SSE3
        __m128 tmp0 = _mm_hadd_ps(rhs,rhs);
```

```
5           __m128 tmp1 = _mm_hadd_ps(tmp0,tmp0);
6       #else
7           __m128 tmp0 = _mm_add_ps(rhs,_mm_movehl_ps(rhs,rhs));
8           __m128 tmp1 = _mm_add_ss(tmp0,_mm_shuffle_ps(tmp0,tmp0,1));
9       #endif
10          return _mm_cvtss_f32(tmp1);
11      }
12
```

Another useful project would be to write overloads of standard mathematical functions (exp, log, etc) that work with the wrappers.

As you can see, writing the wrappers is just the beginning, you can then enrich them with whatever functionality you need but this goes beyond the topic of this first series of articles.

Posted by Johan Mabille Oct 25th, 2014 SIMD, Vectorization

« Writing C++ wrappers for SIMD intrinsics (4) Performance considerations about SIMD wrappers »

Comments