Writing C++ Wrappers for SIMD Intrinsics (4)

Oct 13th, 2014

3. Plugging the wrappers into existing code

3.1 Storing vector4f instead of float

Now that we have nice wrappers, let’s see how we can use them in real code. Consider the following loop:

sample.cpp

```
std::vector<float> a, b, c, d, e;
// somewhere in the code, a, b, c, d and e are
// resized so they hold n elements
// ...
for(size_t i = 0; i < n; ++i)
{
    e[i] = a[i]*b[i] + c[i]*d[i];
}
```

A first solution could be to store vector of vector4f instead of vector of float:

sample.cpp

```
std::vector<vector4f> a, b ,c, d, e;
// somewhere in the code, a, b, c, d and e are
// resized so they hold n/4 vector4f
// ...
for(size_t i = 0; i < n/4; ++i)
{
    e[i] = a[i]*b[i] + c[i]*d[i];
}
```

Not so bad, thanks to the operators overloads, the code is exactly the same as the one for float, but the operations are performed on four floats at once. If n is not a multiple of four, we allocate an additional vector4f in each vector and we initialize the useless elements with 0.

The problem is you could need to work with the scalar instead of the vector4f, for instance if you search for a specific element in the vector or if you fill your vector pushing back elements one by one. In this case, you would have to recode any piece of algorithm that works on single elements (and that includes a lot of STL algorithms) and then add special code for working on scalars within a vector4f. Working on scalars within vector4f is possible (we will see later how to modify our wrappers so that we can do it), but is slower than working directly on scalars, thus you could lose the benefits of using vectorization.

3.2 Initializing vector4f from container of float

Another solution could be to initialize the wrapper from values stored in a vector:

sample.cpp

```
std::vector<float>a, b, c, d, e;
// somewhere in the code, a, b, c, d and e are
// resized so they hold n elements
/ ...
for(size_t i = 0; i < n/4; i += 4)
{
    vector4f av(a[i],a[i+1],a[i+2],a[i+3]);
    vector4f bv(b[i],b[i+1],b[i+2],b[i+3]);
    vector4f cv(c[i],c[i+1],c[i+2],c[i+3]);
    vector4f dv(d[i],d[i+1],d[i+2],d[i+3]);

    vector4f ev = av*bv + cv*dv;
    // how do we store ev in e[i],e[i+1],e[i+2],e[i+3] ?
}
for(size_t i = n/4; i < n; ++i)
{
    e[i] = a[i]*b[i] + c[i]*d[i];
}
```

The first problem is that we need a way to store a vector4f into 4 floats; as said in the previous paragraph, we can add to our wrappers a method that returns a scalar within the vector4f and invoke it that way:

sample.cpp

```
e[i]   = ev[0];
e[i+1] = ev[1];
e[i+2] = ev[2];
e[i+3] = ev[3];
```

The second problem is that this code is not generic; if you migrate from SSE to AVX, you’ll have to update the initialization of your wrapper so it takes 8 floats; the same for storing your vector4f in scalar results.

What we need here is a way to load float into vector4f and to store vector4f into floats that doesn’t depend on the size of vector4f (that is, 4). That’s the aim of the load and store intrinsics.

3.3 Load from and store to memory

If you take a look at the xmmintrin.h file, you'll notice the compiler provides two kinds of load and store intrinsics:

- _mm_load_ps / _mm_store_ps: these functions require the source / destination memory buffer to be aligned; the alignment boundary depends on the version of the SIMD you're using: 16 bits for SSE2, 32 bits for AVX.
- _mm_loadu_ps / _mm_storeu_ps: these functions don't require any alignment of the source / destination memory buffer.

Intrinsics with alignment constraints are faster, and should be used by default; however, even if memory allocations are aligned, you can't guarantee that the memory buffer you pass to the load / store function is aligned. Indeed, consider the matrix product C=AxB, where A is a 15x15 matrix of floats with linear row storage and B a vector that holds 15 float elements. The computation of C[1] starts with:

sample.cpp

```
vector4f tmp(0);
for(size_t k = 0; k < 12; k+=4)
{
    tmp += loadu(a+15+k) * load(b+k);
}
// ...
```

Here, if A is 16-byte aligned, since the size of a float is 4 bytes, a[15], a[19] and a[23] aren't 16-byte aligned, and you have to use the unaligned overload of the intrinsics (designated by the generic *loadu* function in the sample code).

Here's how we need to update our wrappers to handle load and store functions:

simd_sse.hpp

```
class vector4f : public simd_vector<vector4f>
{
public:

    // ...

    inline vector4f& load_a(const float* src)
    {
        m_value = _mm_load_ps(src);
        return *this;
    }

    inline vector4f& load_u(const float* src)
    {
        m_value = _mm_loadu_ps(src);
        return *this;
    }

    inline void store_a(float* dst) const
    {
        _mm_store_ps(dst,m_value);
    }

    inline void store_u(float* dst) const
    {
        _mm_storeu_ps(dst,m_value);
    }
};
```

Assuming the memory buffer of std::vector is 16-bytes aligned, the sample code becomes:

sample.cpp

```
std::vector<float>a, b, c, d, e;
// somewhere in the code, a, b, c, d and e are
// resized so they hold n elements
/ ...
for(size_t i = 0; i < n/4; i += 4)
{
    vector4f av; av.load_a(&a[i]);
    vector4f bv; bv.load_a(&b[i));
    vector4f cv; cv.load_a(&c[i]);
    vector4f dv; dv.load_a(&d[i]);

    vector4f ev = av*bv + cv*dv;
    ev.store_a(&e[i]);
}
for(size_t i = n/4; i < n; ++i)
{
    e[i] = a[i]*b[i] + c[i]*d[i];
}
```

Now, if we migrate our code from SSE to AVX, all we have to do is to replace vector4f by vector8f! (Ok, we also have to deal with memory alignment issues, I come back to this in a few moments). We'll see in a future section how we can avoid the explicit usage of vector4f so we get full genericity. But for now, we have to face a last problem: in the sample code, we assumed the memory buffer wrapped by std::vector was 16-bytes aligned. How do we know a memory allocation is aligned, and how do we know the boundary alignment?

The answer is that it depends on your system and your compiler. On Windows 64 bits, dynamic memory allocation is 16-bytes aligned; in GNU systems, a block returned by malloc or realloc is always a multiple of 8 (32-bit systems) or 16 (64-bit system). So if we want to write code generic enough to handle many SIMD instruction sets, it is clear that we must provide a way to ensure memory allocation is always aligned, and is aligned on a given boundary.

The solution is to design an aligned memory allocator and to use it in std::vector:

sample.cpp

```
typedef aligned_allocator<16> simd_allocator;
std::vector<float,simd_allocator> a,b,c,d,e.
// ....
```

Now, we can handle any alignment boundary requirement through a typedef:

sample.cpp

```
typedef aligned_allocator<16> simd_allocator_sse; // SSE
typedef aligned_allocator<32> simd_allocator_avx; // AVX
```

3.4 Conditional branch

Another issue we have to deal with, when we plug our wrapper, is conditional branching: indeed the if-else statement evaluates a branch depending on the scalar condition, but the if statement works only for scalar condition, and we cannot directly override it to work with our wrappers. Consider the following code:

sample.cpp

```
std::vector<float> a,b,c,d,e;
// ... initialization of a, b, c and d
for(size_t i = 0; i < a.size(); ++i)
{
    if(a[i] > 0)
    {
        e[i] = a[i]*b[i] + c[i]*d[i];
    }
    else
    {
        e[i] = b[i] + c[i]*d[i];
    }
}
```

What we do here is selecting a value for e[i] depending on the sign of a[i]; the code could be written in a sub-optimal way:

sample.cpp

```
float select(bool cond, float v1, float v2)
{
    return cond ? v1 : v2;
}

for(size_t i = 0; i < a.size(); ++i)
{
    float e_tmp1 = a[i]*b[i] + c[i]*d[i];
    float e_tmp2 = b[i] + c[i]*d[i];

    e[i] = select(a[i] > 0, e_tmp1, e_tmp2);
}

```

Even though the “select” function is a bit overkill in the scalar case, it is exactly what we need for handling conditional branching with the SIMD wrappers. This means the two values (or “branches”) of the conditional statement will be evaluated before we choose the one to affect, but we can’t do better. And since you execute your conditional statement on 4 floats at once, it is still faster than the scalar version, even if suboptimal. The only case where the vectorized code could have a performance loss compared to the scalar code is if one of the conditional branch takes much more time to compute than the other and its result is seldom used.

Knowing this, let’s see how we can implement a select function taking SIMD wrapper parameters. Depending on the SSE version, the compiler may provide a built-in function we can directly use as ternary operator. If not, we have to handle it with old bitwise logical:

sample.cpp

```
vector4f select(const vector4fb& cond, const vector4f& a, const vector4f& b)
{
// Don't bother with the SSE_INSTR_SET preprocessor token, we'll be back ont it later
#if SSE_INSTR_SET >= 5 // SSE 4.1
    return _mm_blendv_ps(b,a,cond);
#else
    return _mm_or_ps(_mm_and_ps(cond,a),_mm_andnot_ps(cond,b));
#endif
}
```

That’s it! We can now write the previous loop using full vectorization:

sample.cpp

```
for(size_t i = 0; i < n/4; i+=4)
{
    vector4f av; av.load_a(&a[i]);
    vector4f bv; bv.load_a(&b[i));
    vector4f cv; cv.load_a(&c[i]);
    vector4f dv; dv.load_a(&d[i]);

    vector4f e_tmp1 = av*bv + cv*dv;
    vector4f e_tmp2 = bv + cv*dv;

    vector4f ev = select(av > 0, e_tmp1, e_tmp2);
    ev.store_a(&e[i]);
}
// scalar version for the last elements of the vectors
// ...
```

Although this code is far better than using intrinsics directly, it is still very verbose and, worse, not generic. If you want to update your code to take advantage of AVX instead of SSE, you need to replace every occurence of vector4f by vector8f, and to change the loop condition and increment so as to take into account the size of vector8f. Doing this in real code will quickly become painful.