Writing C++ Wrappers for SIMD Intrinsics (2)

Oct 10th, 2014

1. SSE/AVX intrinsics

Before we start writing any code, we need to take a look at the instrinsics provided with the compiler. Henceforth, I assume we use an Intel processor, recent enough to provide SSE 4 and AVX instruction sets; the compiler can be gcc or MSVC, the instrinsics they provide are almost the same.

If you already know about SSE / AVX intrinsics you may skip this section.

1.1 Registers

SSE uses eight 128 bits registers, from xmm0 to xmm7; Intel and AMD 64 bits extensions adds eight more registers, from xmm8 to xmm15; thus SSE intrinsics can perform on 4 packed float, 2 packed double, 4 32-bits integers, etc …

With AVX, the width of the SIMD registers is increased from 128 to 256 bits; the register are renamed from xmm0-xmm7 to ymm0-ymm7 (and from xmm8-xmm15 to ymm8 to ymm15); however legacy sse instructions still can be used, and xmm registers can still be addressed since they're the lower part of ymm registers.

AVX512 will increase the width of the SIMD registers from 256 to 512 bits.

1.2 Files to include

Intrinsic functions are made available in different header files, based on the version of the SIMD instruction set they belong to:

- <xmmintrin.h> : SSE, operations on 4 single precision floating point numbers (float).
- <emmintrin.h> : SSE 2, operations on integers and on 2 double precision floating point numbers (double).
- <pmmintrin.h> : SSE 3, horizontal operations on SIMD registers.
- <tmmintrin.h> : SSSE 3, additional instructions.
- <smmintrin.h> : SSE 4.1, dot product and many operations on integers

- <nmmintrin.h> : SSE 4.2, additional instructions.
- <immintrin.h> : AVX, operations on integers, 8 float or 4 double.

Each of these files includes the previous one, so you only have to include the one matching the highest version of the SIMD instruction set available in your processor. Later we will see how to detect at compile time which version on SIMD instruction set is available and thus which file to include. For now, just assume we're able to include the right file each time we need it.

1.3 Naming rules

Now if you take a look at these files, you will notice provided data and functions follow some naming rules :

- data vectors are named **__mXXX(T)**, where :
  - XXX is the number of bits of the vector (128 for SSE, 256 for AVX)
  - is T a character for the type of the data; T is omitted for float, i fot integers and d for double; thus __m128d is the data vector to use when performing SSE instructions on double.
- intrinsic functions operating on floating point numbers are usually named **_mm(XXX)_NAME_PT**, where :
  - XXX is the number of bits of the SIMD registers; it is omitted for 128 bits registers
  - NAME is the short name of the function (add, sub, cmp, …)
  - P indicates whether the functions operates on a packed data vector (p) or on a scalar only (s)
  - T indicates the type of the floating point numbers : s for single precision, d for double precision
- intrinsic functions operating on integers are usually named **_mm(XXX)_NAME_EPSYY**, where :
  - XXX is the number of bits of the SIMD registers; it is omitted for 128 bits registers
  - NAME is the short name of the function (add, sub, cmp)
  - S indicates whether the integers are signed (i) or unsigned (u)
  - YY is the number of bits of the integer

1.4 Intrinsics categories

Intrinsics encompass a wide set of features; we can distinguish the following categories (not exhausive) :

- Arithmetic : _mm_add_xx, _mm_sub_xx, _mm_mul_xx, _mm_div_xx, …
- Logical : _mm_and_xx, _mm_or_xx, _mm_xor_xx, …
- Comparison : _mm_cmpeq_xx, _mm_cmpneq_xx, _mm_cmplt_xx, …
- Conversion : _mm_cvtepixx, …
- Memory move : _mm_load_xx, _mm_store_xx, …
- Setting : _mm_set_xx, _mm_setzero_xx, …

I will not provide wrappers for all intrinsics, and some of them will be used only to build higher level functions in the wrappers.

1.5 Sample code

Now you know a little more about SSE and AVX intrinsics, you may reconsider the need for wrapping them; indeed, if you don't need to handle other instructions set, you could think of using SSE/AVX intrinsics directly. I hope this sample code will make you change your mind :

SSE_sample.cpp

```
// computes e = a*b + c*d using SSE where a, b, c, d and e are vector of floats
for(size_t i = 0; i < e.size(); i += 4)
{
    __m128 val = _mm_add_ps(_mm_mul_ps(_mm_load_ps(&a[i]),_mm_load_ps(&b[i])),
                            _mm_mul_ps(_mm_load_ps(&c[i]),_mm_load_ps(&d[i])));
    _mm_store_ps(&e[i],val);
}

```

Quite hard to read, right ? And this is just for two multiplications and one addition; imagine using intrinsics in a huge amount of code, and you will get code really hard to understand and to maintain. What we need is a way to use __m128 with traditional arithmetic operators, as we do with float :

wrapped_sample.cpp

```
// computes e = a*b + c*d using SSE where a, b, c, d and e are vector of floats
for(size_t i = 0; i < e.size(); i += 4)
{
    __m128 val = load(&a[i]) * load(&b[i]) + load(&c[i]) * load(&d[i]);
    store(&e[i],val);
}
```

That's the aim of the wrappers we start to write in the next section.

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

« Writing C++ wrappers for SIMD intrinsics (1) Writing C++ Wrappers for SIMD Intrinsics (3) »

Comments

FL Jia
6 years ago

Useful!