Performance Considerations About SIMD Wrappers

Nov 20th, 2014

When I posted a link to this blog on reddit, I had comments from people who were skeptical of the SIMD Wrappers performances. They raised many possible performance hits in the implementation:

- Arguments passed by const references instead of values, introducing a useless indirection and preventing the compiler from keeping the variable into registers
- Indirection due to the wrapping of __mXXX types into objects
- Operator overloads preventing the compiler from proper instruction reordering during optimization

I've always thought the compiler was smart enough to handle registers and optimizations, whatever the type of the functions arguments (const references or values); and I don't understand why operators overloads shouldn't be considered as classical functions by the compiler. But well, maybe I am too optimistic about the capabilities of the compiler? I was suggested a solution based on pure functions that should be simpler and faster, but I was not given any evidence. Let's take a closer look at both implementations and the assembly code they generate so we can determine whether or not the wrappers introduce performance hits.

Before we go further, here are some technical details: the compiler used in this article is gcc 4.7.3, results may be different with another compiler (and I am interested in seeing these results). The SIMD wrappers used are those of the article series mentioned above, and the implementation based on stateless pure functions looks like:

simd_function.hpp

```
typedef __m128 vector4f2;

inline vector4f2 add(vector4f2 lhs, vector4f2 rhs)
{
    return _mm_add_ps(lhs,rhs);
}

inline vector4f mul(vector4f2 lhs, vector4f2 rhs)
{
    return _mm_mul_ps(lhs,rhs);
}

inline vector4f2 load_a(const float* src)
{
    return _mm_load_ps(src);
}

inline vector4f2 store_a(float* dst, vector4f2 src)
{
    _mm_store_ps(dst,src);
}
```

1. Pure function vs SIMD wrappers

Let's see the assembly code generated by the following functions:

simd_test.cpp

```
vector4f test_sse_a(vector4f a, vector4f b)
{
    return a + b;
}


vector4f2 test_sse_a2(vector4f2 a, vector4f2 b)
{
    return add(a,b);
}
```

The generated assembly code is:

simd_test.asm

```
// test_sse_a
   0:    c5 f8 28 06              vmovaps (%rsi),%xmm0
   4:    48 89 f8                 mov    %rdi,%rax
   7:    c5 f8 58 02              vaddps (%rdx),%xmm0,%xmm0
   b:    c5 f8 29 07              vmovaps %xmm0,(%rdi)
   f:    c3                       retq
// test_sse_a2
   0:    c5 f8 58 c1              vaddps %xmm1,%xmm0,%xmm0
   4:    c3                       retq
   5:    66 66 2e 0f 1f 84 00     data32 nopw %cs:0x0(%rax,%rax,1)
   c:    00 00 00 00
```

If you're not familiar with assembler, vaddps is the assembly for _mm_add_ps (strictly speaking for _m256_add_ps, but this doesn't make a big difference), vmovaps is a transfer instruction from memory to SIMD register (load) or from SIM register to memory (store) depending on its arguments, and %xmmX are the SIMD registers. Do not worry about the last line of the test_sse_a2 function, this is a "do-nothing" operation, used for padding, and does not concern us here.

So what can we tell at first sight ? Well, it seems SIMD wrappers introduce an overhead, using transfer instructions, while the implementation based on stateless functions directly uses register. Now the question is why. Is this due to constant reference

arguments ?

2. Constant reference argument vs value argument

If we change the code of the SIMD wrappers operator overloads to take their arguments by value rather than by constant reference, the generated assembly code doesn't change:

simd_sse.hpp

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

// test_sse_a asm:
   0:    c5 f8 28 06             vmovaps (%rsi),%xmm0
   4:    48 89 f8                mov    %rdi,%rax
   7:    c5 f8 58 02             vaddps (%rdx),%xmm0,%xmm0
   b:    c5 f8 29 07             vmovaps %xmm0,(%rdi)
   f:    c3
```

Moreover, if we change the functional implementation so it takes arguments by constant reference instead of value, the generated assembly code for test_sse_a2 is exactly the same as in the previous section:

simd_test.cpp

```
inline vector4f2 add(const vector4f2& lhs, const vector4f2& rhs)
{
    return _mm_add_ps(lhs,rhs);
}

// test_sse_a2 asm:
   0:    c5 f8 58 c1             vaddps %xmm1,%xmm0,%xmm0
   4:    c3                      retq
   5:    66 66 2e 0f 1f 84 00    data32 nopw %cs:0x0(%rax,%rax,1)
   c:    00 00 00 00
```

As I supposed, the compiler (at least gcc) is smart enough to optimize and keep in register arguments passed by constant reference (if they fit into registers of course). So it seems the overhead comes from the indirection of the wrapping, but this is really hard to believe.

3. And the culprit is …

To confirm this hypothesis, let's simplify the code of the wrapper so we only test the indirection. Inheritance from the **simd_vector** base class is removed:

simd_sse.hpp

```
class vector4f
{
public:

    inline vector4f2() {}
    inline vector4f2(__m128 rhs) : m_value(rhs) {}

    inline vector4f2& operator=(__m128 rhs)
        {
            m_value = rhs;
            return *this;
        }

    inline operator __m128() const { return m_value; }

```

```
16 private:
17
18         __m128 m_value;
19 };
20
21 inline vector4f2 operator+(vector4f2 lhs, vector4f2 rhs) { return _mm_add_ps(lhs,rhs); }
22
```

Now if we dump the assembly code of the test_sse_add function we defined in the beginning, here is what we get:

simd_test.cpp

```
1
2 // test_sse_a asm:
3    0:    c5 f8 58 c1              vaddps %xmm1,%xmm0,%xmm0
4    4:    c3                       retq
5    5:    66 66 2e 0f 1f 84 00     data32 nopw %cs:0x0(%rax,%rax,1)
6    c:    00 00 00 00
```

That's exactly the same code as the one generated by pure stateless functions. So the indirection of the wrapper doesn't introduce any overhead. Since the only change we've made from the previous wrapper is to remove the CRTP layer, we have the culprit for the overhead we noticed in the beginning: the CRTP layer.

I first thought of a Empty Base Optimization problem, but printing the size of both implementations of the wrapper proved me wrong: in both case, the size of the wrapper is 16, so it fits in the XMM registers. So I must admit, I still have no explanation for this problem.

In the next section, I will consider the wrapper implementation that doesn't use CRTP. Now that we've fixed this issue, let's see if operators overload prevents the compiler from proper instructions reordering during optimization.

4. Operators overload

For this test, I used the following functions:

simd_test2.cpp

```
1 vector4f2 test_sse_b2(vector4f2 a, vector4f2 b, vector4f2 c, vector4f2 d)
2 {
3     return add(mul(a,b),mul(c,d));
4 }
5
6 vector4f2 test_sse_c2(vector4f2 a, vector4f2 b, vector4f2 c, vector4f2 d)
7 {
8     return add(add(mul(a,b),div(c,d)),sub(div(c,b),mul(a,d)));
9 }
10
11 vector4f2 test_sse_d2(vector4f2 a, vector4f2 b, vector4f2 c, vector4f2 d)
12 {
13     return mul(test_sse_c2(a,b,c,d),test_sse_b2(a,b,c,d));
14 }
15
```

And the equivalent functions for wrappers:

simd_test.cpp

```
1 vector4f test_sse_b(vector4f a, vector4f b, vector4f c, vector4f d)
2 {
3     return a*b + c*d;
4 }
5
6 vector4f test_sse_c(vector4f a, vector4f b, vector4f c, vector4f d)
7 {
8     return (a*b + c/d) + (c/b - a*d);
9 }
```

```
10
11 vector4f test_sse_d(vector4f a, vector4f b, vector4f c, vector4f d)
12 {
13     return test_sse_c(a,b,c,d) * test_sse_b(a,b,c,d);
14 }
15
```

Here the parenthesis in **test_sse_c** ensure the compiler generates the same syntactic tree for both implementations; indeed, if we omitted the brackets, the code would have been almost equivalent to:

simd_test2_bis.cpp

```
1
2 // same code for test_sse_b2 and test_sse_d2
3
4 vector4f2 test_sse_c2(vector4f2 a, vector4f2 b, vector4f2 c, vector4f2 d)
5 {
6     return sub(add(div(c,b),add(mul(a,b),div(c,d))),mul(a,d));
7 }
```

Here is the generated assembly code with explanations in comments:

simd_test.asm

```
1
2  // test_sse_d
3    40:    c5 f8 59 e1             vmulps %xmm1,%xmm0,%xmm4 // a*b in xmm4
4    44:    c5 e8 5e c9             vdivps %xmm1,%xmm2,%xmm1 // c/b in xmm1
5    48:    c5 f8 59 c3             vmulps %xmm3,%xmm0,%xmm0 // a*d in xmm0
6    4c:    c5 e8 59 eb             vmulps %xmm3,%xmm2,%xmm5 // c*d in xmm5
7    50:    c5 d8 58 ed             vaddps %xmm5,%xmm4,%xmm5 // a*b + c*d in xmm5
8    54:    c5 f0 5c c8             vsubps %xmm0,%xmm1,%xmm1 // c/b - a*d in xmm1
9    58:    c5 e8 5e c3             vdivps %xmm3,%xmm2,%xmm0 // c/d in xmm0
10   5c:    c5 d8 58 c0             vaddps %xmm0,%xmm4,%xmm0 // a*b + c/d in xmm0
11   60:    c5 f8 58 c1             vaddps %xmm1,%xmm0,%xmm0 // a*b + c/d + c/b - a*d in xmm0
12   64:    c5 f8 59 c5             vmulps %xmm5,%xmm0,%xmm0 // (a*b + c*d) * xmm0 in xmm0
13   68:    c3                      retq
14   69:    0f 1f 80 00 00 00 00    nopl   0x0(%rax)
15
16 // test_sse_d2
17   40:    c5 f8 59 e1             vmulps %xmm1,%xmm0,%xmm4
18   44:    c5 e8 5e c9             vdivps %xmm1,%xmm2,%xmm1
19   48:    c5 f8 59 c3             vmulps %xmm3,%xmm0,%xmm0
20   4c:    c5 e8 59 eb             vmulps %xmm3,%xmm2,%xmm5
21   50:    c5 d8 58 ed             vaddps %xmm5,%xmm4,%xmm5
22   54:    c5 f0 5c c8             vsubps %xmm0,%xmm1,%xmm1
23   58:    c5 e8 5e c3             vdivps %xmm3,%xmm2,%xmm0
24   5c:    c5 d8 58 c0             vaddps %xmm0,%xmm4,%xmm0
25   60:    c5 f8 58 c1             vaddps %xmm1,%xmm0,%xmm0
26   64:    c5 f8 59 c5             vmulps %xmm5,%xmm0,%xmm0
27   68:    c3                      retq
28   69:    0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

The generated assembly codes for **test_sse_d** and **test_sse_d2** are exactly the sames. Operators overloads and equivalent stateless functions generally produces the same assembly code provided that the syntax tree is the same in both implementations. Indeed, the evaluation order of operators arguments and functions arguments may differ, making it impossible to have the same syntax tree in both implementations when using non-commutative operators.

Now what if we mix computation instructions with loop, load and store ? Consider the following piece of code:

simd_test.cpp

```
1 void test_sse_e(const std::vector<float>& a,
2                 const std::vector<float>& b,
3                 const std::vector<float>& c,
4                 const std::vector<float>& d,
5                 std::vector<float>& e)
```

```
{
    // typedef vector4f2 for test_sse_e2 implementation
    typedef vector4f vec_type;
    size_t bound = a.size()/4;
    for(size_t i = 0; i < bound; i += 4)
    {
        vec_type av = load_a2(&a[i]);
        vec_type bv = load_a2(&b[i]);
        vec_type cv = load_a2(&c[i]);
        vec_type dv = load_a2(&d[i]);

    // vec_type ev = test_sse_d2(av,bv,cv,dv); for test_sse_e2 implementation
        vec_type ev = test_sse_d(av,bv,cv,dv);
        store_a(&e[i],ev);
    }
}

```

Again, the generated assembly code is the same for both implementations:

simd_test.asm

```
// test_sse_e:
  70:    4c 8b 0f                mov    (%rdi),%r9
  73:    48 8b 7f 08             mov    0x8(%rdi),%rdi
  77:    4c 29 cf                sub    %r9,%rdi
  7a:    48 c1 ff 02             sar    $0x2,%rdi
  7e:    48 c1 ef 02             shr    $0x2,%rdi
  82:    48 85 ff                test   %rdi,%rdi
  85:    74 5d                   je     e4 <_ZN4simd11test_sse_eERKSt6vectorIfSaIfEES4_S4_S4_RS2_+0x74>
  87:    4c 8b 16                mov    (%rsi),%r10
  8a:    31 c0                   xor    %eax,%eax
  8c:    48 8b 32                mov    (%rdx),%rsi
  8f:    48 8b 09                mov    (%rcx),%rcx
  92:    49 8b 10                mov    (%r8),%rdx
  95:    0f 1f 00                nopl   (%rax)
  98:    c5 f8 28 0c 86          vmovaps (%rsi,%rax,4),%xmm1
  9d:    c5 f8 28 04 81          vmovaps (%rcx,%rax,4),%xmm0
  a2:    c4 c1 78 28 24 81       vmovaps (%r9,%rax,4),%xmm4
  a8:    c4 c1 78 28 1c 82       vmovaps (%r10,%rax,4),%xmm3
  ae:    c5 f0 59 e8             vmulps %xmm0,%xmm1,%xmm5
  b2:    c5 d8 59 d3             vmulps %xmm3,%xmm4,%xmm2
  b6:    c5 d8 59 e0             vmulps %xmm0,%xmm4,%xmm4
  ba:    c5 f0 5e db             vdivps %xmm3,%xmm1,%xmm3
  be:    c5 e8 58 ed             vaddps %xmm5,%xmm2,%xmm5
  c2:    c5 f0 5e c0             vdivps %xmm0,%xmm1,%xmm0
  c6:    c5 e0 5c dc             vsubps %xmm4,%xmm3,%xmm3
  ca:    c5 e8 58 d0             vaddps %xmm0,%xmm2,%xmm2
  ce:    c5 e8 58 d3             vaddps %xmm3,%xmm2,%xmm2
  d2:    c5 e8 59 d5             vmulps %xmm5,%xmm2,%xmm2
  d6:    c5 f8 29 14 82          vmovaps %xmm2,(%rdx,%rax,4)
  db:    48 83 c0 04             add    $0x4,%rax
  df:    48 39 c7                cmp    %rax,%rdi
  e2:    77 b4                   ja     98 <_ZN4simd11test_sse_eERKSt6vectorIfSaIfEES4_S4_S4_RS2_+0x28>
  e4:    f3 c3                   repz retq

// test_sse_e2
  70:    4c 8b 0f                mov    (%rdi),%r9
  73:    48 8b 7f 08             mov    0x8(%rdi),%rdi
  77:    4c 29 cf                sub    %r9,%rdi
  7a:    48 c1 ff 02             sar    $0x2,%rdi
  7e:    48 c1 ef 02             shr    $0x2,%rdi
  82:    48 85 ff                test   %rdi,%rdi
  85:    74 5d                   je     e4 <_ZN4simd11test_sse_e2ERKSt6vectorIfSaIfEES4_S4_S4_RS2_+0x74>
  87:    4c 8b 16                mov    (%rsi),%r10
  8a:    31 c0                   xor    %eax,%eax
  8c:    48 8b 32                mov    (%rdx),%rsi
  8f:    48 8b 09                mov    (%rcx),%rcx
  92:    49 8b 10                mov    (%r8),%rdx
  95:    0f 1f 00                nopl   (%rax)
  98:    c5 f8 28 0c 86          vmovaps (%rsi,%rax,4),%xmm1
```

```
50   9d:    c5 f8 28 04 81          vmovaps (%rcx,%rax,4),%xmm0
51   a2:    c4 c1 78 28 24 81       vmovaps (%r9,%rax,4),%xmm4
52   a8:    c4 c1 78 28 1c 82       vmovaps (%r10,%rax,4),%xmm3
53   ae:    c5 f0 59 e8             vmulps %xmm0,%xmm1,%xmm5
54   b2:    c5 d8 59 d3             vmulps %xmm3,%xmm4,%xmm2
55   b6:    c5 d8 59 e0             vmulps %xmm0,%xmm4,%xmm4
56   ba:    c5 f0 5e db             vdivps %xmm3,%xmm1,%xmm3
57   be:    c5 e8 58 ed             vaddps %xmm5,%xmm2,%xmm5
58   c2:    c5 f0 5e c0             vdivps %xmm0,%xmm1,%xmm0
59   c6:    c5 e0 5c dc             vsubps %xmm4,%xmm3,%xmm3
60   ca:    c5 e8 58 d0             vaddps %xmm0,%xmm2,%xmm2
61   ce:    c5 e8 58 d3             vaddps %xmm3,%xmm2,%xmm2
62   d2:    c5 e8 59 d5             vmulps %xmm5,%xmm2,%xmm2
63   d6:    c5 f8 29 14 82          vmovaps %xmm2,(%rdx,%rax,4)
64   db:    48 83 c0 04             add    $0x4,%rax
65   df:    48 39 c7                cmp    %rax,%rdi
66   e2:    77 b4                   ja     98 <_ZN4simd11test_sse_e2ERKSt6vectorIfSaIfEES4_S4_S4_RS2_+0x28>
67   e4:    f3 c3                   repz retq
68
```

To conclude, operators overloads don't prevent the compiler to reorder instructions during optimization, and thus they don't introduce any performance issue. Since they allow you to write code more readable and easier to maintain, it would be a shame not to use them.

5. Refactoring the wrappers without CRTP

Before we consider refactoring the wrappers, let's see the overhead of the CRTP layer in a more realistic code. Using the **test_sse_d** and **test_sse_e** functions of the previous section with the first version of the wrappers (the one with CRTP), here is the result of objdump:

test_sse.asm

```
1  // test_sse_d
2    70:    c4 c1 78 28 00          vmovaps (%r8),%xmm0
3    75:    48 89 f8                mov    %rdi,%rax
4    78:    c5 f8 28 09             vmovaps (%rcx),%xmm1
5    7c:    c5 f8 28 1a             vmovaps (%rdx),%xmm3
6    80:    c5 f8 28 26             vmovaps (%rsi),%xmm4
7    84:    c5 f0 59 e8             vmulps %xmm0,%xmm1,%xmm5
8    88:    c5 d8 59 d3             vmulps %xmm3,%xmm4,%xmm2
9    8c:    c5 d8 59 e0             vmulps %xmm0,%xmm4,%xmm4
10   90:    c5 f0 5e c0             vdivps %xmm0,%xmm1,%xmm0
11   94:    c5 e8 58 ed             vaddps %xmm5,%xmm2,%xmm5
12   98:    c5 f0 5e db             vdivps %xmm3,%xmm1,%xmm3
13   9c:    c5 e8 58 d0             vaddps %xmm0,%xmm2,%xmm2
14   a0:    c5 e8 58 d3             vaddps %xmm3,%xmm2,%xmm2
15   a4:    c5 e8 5c e4             vsubps %xmm4,%xmm2,%xmm4
16   a8:    c5 d8 59 e5             vmulps %xmm5,%xmm4,%xmm4
17   ac:    c5 f8 29 27             vmovaps %xmm4,(%rdi)
18   b0:    c3                      retq
19   b1:    66 66 66 66 66 66 2e    data32 data32 data32 data32 data32 nopw %cs:0x0(%rax,%rax,1)
20   b8:    0f 1f 84 00 00 00 00
21   bf:    00
22
23 // test_sse_e
24   c0:    4c 8b 0f                mov    (%rdi),%r9
25   c3:    48 8b 7f 08             mov    0x8(%rdi),%rdi
26   c7:    4c 29 cf                sub    %r9,%rdi
27   ca:    48 c1 ff 02             sar    $0x2,%rdi
28   ce:    48 c1 ef 02             shr    $0x2,%rdi
29   d2:    48 85 ff                test   %rdi,%rdi
30   d5:    74 5d                   je     134 <_ZN4simd10test_sse_eERKSt6vectorIfSaIfEES4_S4_S4_RS2_+0x74>
31   d7:    4c 8b 16                mov    (%rsi),%r10
32   da:    31 c0                   xor    %eax,%eax
33   dc:    48 8b 32                mov    (%rdx),%rsi
34   df:    48 8b 09                mov    (%rcx),%rcx
35   e2:    49 8b 10                mov    (%r8),%rdx
36   e5:    0f 1f 00                nopl   (%rax)
```

```
  e8:    c5 f8 28 0c 86          vmovaps (%rsi,%rax,4),%xmm1
  ed:    c5 f8 28 04 81          vmovaps (%rcx,%rax,4),%xmm0
  f2:    c4 c1 78 28 24 81       vmovaps (%r9,%rax,4),%xmm4
  f8:    c4 c1 78 28 1c 82       vmovaps (%r10,%rax,4),%xmm3
  fe:    c5 f0 59 e8             vmulps %xmm0,%xmm1,%xmm5
102:    c5 d8 59 d3             vmulps %xmm3,%xmm4,%xmm2
106:    c5 d8 59 e0             vmulps %xmm0,%xmm4,%xmm4
10a:    c5 f0 5e c0             vdivps %xmm0,%xmm1,%xmm0
10e:    c5 e8 58 ed             vaddps %xmm5,%xmm2,%xmm5
112:    c5 f0 5e db             vdivps %xmm3,%xmm1,%xmm3
116:    c5 e8 58 d0             vaddps %xmm0,%xmm2,%xmm2
11a:    c5 e8 58 d3             vaddps %xmm3,%xmm2,%xmm2
11e:    c5 e8 5c e4             vsubps %xmm4,%xmm2,%xmm4
122:    c5 d8 59 e5             vmulps %xmm5,%xmm4,%xmm4
126:    c5 f8 29 24 82          vmovaps %xmm4,(%rdx,%rax,4)
12b:    48 83 c0 04             add    $0x4,%rax
12f:    48 39 c7                cmp    %rax,%rdi
132:    77 b4                   ja     e8 <_ZN4simd10test_sse_eERKSt6vectorIfSaIfEES4_S4_S4_RS2_+0x28>
134:    f3 c3                   repz retq

```

In **test_sse_d**, we have six more instructions than in the previous version, these instructions are data transfer to the SIMD registers at the beginning of the function, and data transfer from the SIMD register at the end of the function. Now if we look at **test_sse_e**, we've got exactly the same code as in the previous section. The call to **test_sse_d** is inlined, and since the data transfer from and to SIMD registers is required by **load_a** and **store_a** functions, there is no need to keep the **movaps** instructions of **test_sse_d**. So if the functions working with wrappers are small enough to be inlined and if computation instructions are used between load and store functions, using the wrappers with CRTP should not introduce any overhead since the compiler will remove useless **movaps** instructions.

However, if you still want to refactor the wrappers but don't want to repeat the boilerplate implementation of operators overloads, the alternative is to use preprocessor macros:

simd_sse.hpp

```
#define DEFINE_OPERATOR+=(RET_TYPE,ARG_TYPE)\
    inline RET_TYPE& operator+=(const ARG_TYPE& rhs)\
    {\
        *this = *this + rhs;\
        return *this;\
    }
// ... etc for other computed assignment operators
#define DEFINE_ASSIGNMENT_OPERATORS(TYPE,SCALAR_TYPE)\
    DEFINE_OPERATOR+=(TYPE,TYPE)\
    DEFINE_OPERATOR+=(TYPE,SCALAR_TYPE)\
    DEFINE_OPERATOR-=(TYPE,TYPE)\
    DEFINE_OPERATOR-=(TYPE,SCALAR_TYPE)\
    // etc
```

This is much less elegant, but it comes with the guarantee that there won't be any performance issue.

Conclusion

Performance is not an intuitive domain; we have to check any assumption we make, because these assumptions can be legacy of time when compilers were inefficient or buggy, or a bias due to our misunderstanding of some mechanisms of the language. Here we've seen that neither operator overloads nor constant reference argument instead of value argument introduce any performance issue with GCC, but this might be different with another compiler.

Posted by Johan Mabille Nov 20th, 2014 SIMD, Vectorization

« Writing C++ Wrappers for SIMD Intrinsics (5) Aligned memory allocator »

Comments