Intrinsic Hoisting - Examples

Hongpu Ma (hongpum@sfu.ca)
Dong Xue (xuedongx@sfu.ca)
Feb 6, 2016

Introduction

As mentioned in class, our term project would be intrinsic hoisting based on LLVM IR. The general idea would be writing a libllvm-based tool or a custom LLVM pass to process LLVM IR (e.g. icgrep-generated IR, or the intermediate result from clang), eliminate hard-coded intrinsics, and feed back into the LLVM backend. Possible outcomes include the portability of Intel-specific code, automatic migration from legacy SSE code to newer SIMD extensions and the "free" performance boost coming along.
It would also be a chance to investigate: 1) how programmers manually make use of SIMD; 2) if LLVM generates some sub-optimal instructions in certain cases. As for non-X86 platforms, ARM might be our top interest.

Examples

In this experiment, we wrote some short C programs containing different intrinsics, compiled to LLVM IR using clang, manually hoisted the intrinsics in the IR, and compiled to assembly using llc. Here are some useful commands:

```
clang -S -emit-llvm file.c  # generates file.ll (LLVM IR)
llc [-mattr=+avx] file.ll   # generates file.s (assembly)
clang file.s -o file        # generates executable file
```

We will give three examples in the following order: a simple pure vertical instruction (padd) which can be hoisted by clang automatically, a more complex vertical instruction which does not work well with llvm, and a horizontal instruction whose manually-hoisted form triggers a llvm bug.
All the results were tested under llvm and clang 3.7.1 (Arch Linux x86_64).

1)paddd (_mm_add_epi32)

Pure vertical SIMD instructions are a good starting point, among which “paddd” is probably the simplest. It simply divides the 128-bit vectors into four 32-bit ints and carrys out addition in each lane. Here is its semantic from Intel Intrinsics Guide [1]:

```
FOR j := 0 to 3
       i := j*32
       dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
```

The example program is as below:

```c
#include "emmintrin.h"
#include <stdio.h>
#include <stdint.h>

union M128i {
    __m128i v;
    int32_t i[4];
};

union M128i a, b, c;

int main() {
    a.v = _mm_set_epi32(1, 2, 3, 4);
    b.v = _mm_set_epi32(-4, 3, 2, -1);
    c.v = _mm_add_epi32(a.v, b.v);
    /* Expected: -3 5 5 3 */
    printf("%d %d %d %d\n", c.i[3], c.i[2], c.i[1], c.i[0]);
    return 0;
}
```

Surprisingly, when compiling this program to LLVM IR, clang will automatically replace the addition intrinsic with the native vector addition in IR (unrelated sections are omitted):

```
  %38 = load <2 x i64>, <2 x i64>* %6, align 16
  %39 = bitcast <2 x i64> %38 to <4 x i32>
  %40 = load <2 x i64>, <2 x i64>* %7, align 16
  %41 = bitcast <2 x i64> %40 to <4 x i32>
  %42 = add <4 x i32> %39, %41
```

In fact, even the assignment intrinsics are hoisted by clang. There is no need for extra hoisting at all. Yet we can still explore how the LLVM backend compiles this IR. By default, llc only uses SSE2 on X86_64 targets, and thus generates “paddd”. If we turn on AVX support, llc will use the AVX-prefixed instruction “vpaddd” instead. It will use some AVX horizontal instructions to simplify the assignment process, too. Although not shown in this example, AVX also excels at its three-operand form. Unlike the two-operand SSE, an AVX instruction can execute without destroying one of its sources.
Pure vertical instructions like “paddd” have direct correspondance in LLVM IR. clang and llvm have already incorporated this coherency, which gives the advantage of exploiting new ISA extensions when possible.

2)pmuludq (_mm_mul_epu32)

“pmuludq” is a vertical multiplication, but it has different field lengths in the source and the destination. It takes the first and the third 32-bit fields (starting from LSB) from each of the source vectors and stores the result of multiplication as 64-bit fields [1]:

```
FOR j := 0 to 1
       i := j*64
       dst[i+63:i] := a[i+31:i] * b[i+31:i]
ENDFOR
```

The example program:

```
#include "emmintrin.h"
#include <stdio.h>

__m128i a, b, c;

int main() {
    a = _mm_set_epi32(1, 2, 3, 4);
    b = _mm_set_epi32(5, 3, 2, 1);
    c = _mm_mul_epu32(a, b);
    /* __m128i is also an uint64_t[2]
     * Expected: 6 4
     */
    printf("%llu %llu\n", c[1], c[0]);
    return 0;
}
```

Note that in LLVM IR, vector multplications must have the same input width and output width by definition. Any overflow will be truncated. [2] Unfortunately, all multiplications in SSE2 are either “full product” with the double-length result or saturated overflow. Hence, none of them have direct analogy in LLVM IR. clang does not bother hoisting these intrinsics and generates the IR below:

```
  %36 = load <2 x i64>, <2 x i64>* %6, align 16
  %37 = bitcast <2 x i64> %36 to <4 x i32>
  %38 = load <2 x i64>, <2 x i64>* %7, align 16
  %39 = bitcast <2 x i64> %38 to <4 x i32>
  %40 = call <2 x i64> @llvm.x86.sse2.pmulu.dq(<4 x i32> %37, <4 x i32> %39) #3
```

The IR above will compile down to “pmuludq” instruction with nothing surprising. Now let’s try to hoist the intrinsic manually.

According to LLVM IR specs, the only way to get a “full product” is to extend the input, so here we have two possible transformations. The first version is to clear out the higher 32 bits of each 64-bit field in the input, without the bitcast (the magic number is just a bitmask):

```
  %36 = load <2 x i64>, <2 x i64>* %6, align 16
  %37 = and <2 x i64> %36, <i64 4294967295, i64 4294967295>
  %38 = load <2 x i64>, <2 x i64>* %7, align 16
  %39 = and <2 x i64> %38, <i64 4294967295, i64 4294967295>
  %40 = mul <2 x i64> %37, %39
```

The second approach is to use shufflevector and zext:

```
  %36 = load <2 x i64>, <2 x i64>* %6, align 16
  %37 = bitcast <2 x i64> %36 to <4 x i32>
  %38 = shufflevector <4 x i32> %37, <4 x i32> undef, <2 x i32> <i32 1, i32 3>
  %39 = zext <2 x i32> %38 to <2 x i64>
  %40 = load <2 x i64>, <2 x i64>* %7, align 16
  %41 = bitcast <2 x i64> %40 to <4 x i32>
  %42 = shufflevector <4 x i32> %41, <4 x i32> undef, <2 x i32> <i32 1, i32 3>
  %43 = zext <2 x i32> %42 to <2 x i64>
  %44 = mul <2 x i64> %39, %43
```

Surprisingly, llc compiles these two versions to very similar assemblies (i.e., LLVM will automatically use the bitwise logic to clear out higher bits given the second version), both pretty bad. The core part, different from the original intrinsic version, of the assembly is:

```
     movdqa    .LCPI0_0(%rip), %xmm1   # xmm1 = [4294967295,4294967295]
     pand    %xmm1, %xmm0
     pand    -48(%rbp), %xmm1
     movdqa    %xmm1, %xmm2
     pmuludq    %xmm0, %xmm2
     movdqa    %xmm0, %xmm3
     psrlq    $32, %xmm3
     pmuludq    %xmm1, %xmm3
     psllq    $32, %xmm3
     paddq    %xmm2, %xmm3
     psrlq    $32, %xmm1
     pmuludq    %xmm0, %xmm1
     psllq    $32, %xmm1
     paddq    %xmm3, %xmm1
```

xmm1 is the final result. The sequence above is roughly equivalent to just one “pmuludq xmm0, xmm1”. In short, the lengthy assembly code is constructing “64-bit x 64-bit = 64-bit” truncated multiplications using the “32-bit x 32-bit = 64-bit” “pmuludq”, without noticing the higher 32 bits in the input are all zero. It is essentially a scalar calculation (taking 32 bits from the 128-bit SIMD multiplication at a time), despite the SSE2 instruction.

3)pmovmskb (_m_pmovmskb)

We spent some time investigating “pmovmskb” (taking the MSB of every byte) as an example of horizontal operations, but the manually-hoisted IR couldn’t generate the same output. We didn’t realize it was a LLVM bug until Dr. Cameron pointed out his previous experience [3].
Our example is shorter than the one in [3], so we list the C program and the IR sections (before and after hoisting) below for those who are interested.

```
#include "xmmintrin.h"
#include <stdio.h>

__m64 a = {0xF0F0F0F0F0F0F0F0u};

int main() {
    int r = _m_pmovmskb(a);
    /* Expected: ff */
    printf("%x\n", r);
    return 0;
}
```

IR with intrinsics:

```
%9 = load <1 x i64>, <1 x i64>* %2, align 8
%10 = bitcast <1 x i64> %9 to <8 x i8>
%11 = bitcast <8 x i8> %10 to x86_mmx
%12 = call i32 @llvm.x86.mmx.pmovmskb(x86_mmx %11) #3
```

Hoisted IR:

```
%9 = load <1 x i64>, <1 x i64>* %2, align 8
%10 = bitcast <1 x i64> %9 to <8 x i8>
%11 = icmp slt <8 x i8> %10, zeroinitializer
%12 = bitcast <8 x i1> %11 to i8
%13 = zext i8 %12 to i32
```

LLVM fails to compile the i1-bitcast correctly. It will print out “1” instead of “ff”. For more details about this issue, please refer to http://parabix.costar.sfu.ca/wiki/I2Result

Conclusion

The three examples show a potential dilemma: for some SIMD intrinsics, clang is already able to hoist them automatically; for the other complex ones where clang waives its hands, the LLVM backend seems to dislike the manually hoisted IR and gives bad or even wrong assembly. We will keep looking into the issues, and probably work with the code generation group to see if there is any chance of improvement in the LLVM backend as well.

References

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/
[2] http://llvm.org/docs/LangRef.html
[3] http://parabix.costar.sfu.ca/wiki/I2Result

http://parabix.costar.sfu.ca/wiki/SSE2_Hoisting
http://parabix.costar.sfu.ca/wiki/SSERemoval