CS-118-02 Week 9

Varun Narravula

2022-03-28

Working With Non-Scalar Instructions

Up until now, you've worked with scalar math and rudimentary arithmetic. What about non-scalar math? Assembly is not all scalar math, and doing math fast requires working with more data than single values, such as arrays and other types of collections (i.e. vectors, maps, etc.).

Enter SIMD and SSE. **S**ingle **I**nstruction, **M**ultiple **D**ata. and **S**treaming **S**IMD **E**xtensions. Let's take the instruction `mul`; it takes two numbers, and multiples them together. What if you need to multiply two lists of numbers together? You'd use a `for` loop in a high-level programming language. But this isn't efficient; it can operate in $O(n)$ time and no less.

There is a set of 128-bit registers that allows us to store lists of numbers, ranging from `xmm0` to `xmm15`, which allow you to load 4 numbers at once into one. To compute the sum of two lists of 4 numbers across two register, you can use two registers (say, `xmm0` and `xmm1`) and add them together, and it's much more efficient than using a for loop; it has much less time complexity.

The more interesting part here, however, is that SSE registers can hold floating-point numbers as well as integers, and will interpret the numbers it holds however you want to. This allows for much more complex operations; for example, you can now average numbers and get a floating-point result, instead of getting an integer quotient and remainder like a normal `div` instruction would have you do.

Let's look at the anatomy of an SSE register `xmm0`. It works similarly to how a 64-bit general-purpose register such as `rax` is split into `eax` for 32 bits, `ax` for 16 bits, and `al/ah` for 8-bits. Since xmm0 is 128 bits long, it can hold multiple different numbers and reference them simultaneously. It can hold four `dword` (32-bit) integers or four single-precision floating-point numbers (the `float` type in C), eight `word` (16-bit) integers, two `qword` (64-bit) integers or two double-precision floating-point numbers (the `double` type in C), and pretty much whatever combinatino you would like. You can choose to interpret these numbers in whatever way you would like; check out the `gdb` representation of `xmm0` from this code (you don't need to know what it does in depth for right now, it moves an array of 32-bit floats into xmm0):

```
section .data
a dd 1.1, 1.2, 1.3, 1.4; an array of 32-bit floats

section .text
global _start

_start:
    ; move aligned packed single-precision-number array into xmm0
    movaps xmm0, [a]

; exit stuff
```

And the `gdb` representation:

```
(gdb) p/f $xmm0
$5 = {v8_bfloat16 = {-1.075e+08, 1.094, -1.592e-23, 1.195, 2.715e+23, 1.297,
      4.168e-08, 1.398}, v4_float = {1.10000002, 1.20000005, 1.29999995,
      1.39999998}, v2_double = {0.025000009659561329, 0.075000002898741508},
      v16_int8 = {-51, -52, -116, 63, -102, -103, -103, 63, 102, 102, -90,
      63, 51, 51, -77, 63}, v8_int16 = {-13107, 16268, -26214, 16281, 26214,
      16294, 13107, 16307}, v4_int32 = {1.10000002, 1.20000005, 1.29999995,
      1.39999998}, v2_int64 = {0.025000009659561329, 0.075000002898741508},
      uint128 = <invalid float value>}
(gdb)
```

You'll see that `gdb` cannot infer the type by itself, and instead attempts to interpret the register in every single way that you can store numbers in it. The vast majority of them are garbage except for `v4_float` (aka storing 4 floating-point numbers, which is what we did) and `v4_int32` (which is usually the same thing, but do not rely on that behavior, as shown by the names). You must be careful when using SSE registers to not use the wrong type for instructions, because your results will be complete garbage if you do and will almost certainly fail.

Other registers exist for other SIMD instruction sets, called `ymm0-ymm15` (for AVX instructions, which can work with 256 bits instead of 128 bits), and `zmm0-zmm15` (for AVX-512, which is self-explanatory). However, you'll only be seeing up to SSE3 instructions and will not need to use these registers in these notes; you'll only see `xmm0-xmm15`.

SSE Support

Not all x86_64 CPUs have SSE support. To find out if a CPU has SSE support, you can read a control register, which you can access using different instructions. Control registers are registers that you do not normally access, and they contain different flags (like `eflags`) that describe what features the CPU supports and the different capabilities it has. Take this snippet below, which uses `cpuid` to find out if SSE is enabled:

```
section .data
ymsg    db "SSE supported! :)", 10
nmsg    db "SSE not supported! :(", 10

section .text
global  _start

_start:
    mov  eax, 1
    cpuid
    test edx, 1<<25
    jz   .no
    jnz  .yes

.no:
    mov rdi, 1
    mov rsi, nmsg
    mov rdx, 22
    mov rax, 1
    syscall
    jmp exit

.yes:
    mov rdi, 1
    mov rsi, ymsg
    mov rdx, 18
    mov rax, 1
    syscall
    jmp exit

exit:
    mov rax, 60
    mov rdi, 1
    syscall
```

The vast majority of modern x86_64 CPUs enable SSE by default, so you should not have to enable it. However, if you do need to enable it, you can do so using this snippet at the beginning of your assembly file.

```
mov eax, cr0
and ax, 0xFFFB; clear coprocessor emulation CR0.EM
or ax, 2; set coprocessor monitoring  CR0.MP
mov cr0, eax
mov eax, cr4
or ax, 3 << 9; set CR4.OSFXSR and CR4.OSXMMEXCPT at the same time
mov cr4, eax
ret
```

Memory Alignment

Remember how the stack works? Regardless of how large your data is, 32, 8 bits even, the stack always allocates 64 bits for it on a 64-bit system. This is memory alignment. For performance reasons, SSE instructions work much better when using 16-byte-aligned memory. Why? This is because it does not have to manipulate memory to get to addresses, because it will always exist within a set 16 bytes.

Why is this important? There are numerous instructions that rely on aligned memory that are much more efficient than their unaligned variants, if those even exist. Take `movaps`, which moves aligned packed memory to/from SSE registers, and `movups`, which does the same but with unaligned memory. The `movups` instruction is much slower than the `movaps` command because it has to perform bounds checking for alignment, and `movaps` has some additional magic to move much faster because it can guarantee alignment. However, you have to guarantee that memory is aligned before using aligned-memory-only instructions; otherwise, something called a general protection fault (GPF) will be thrown and your program will stop immediately. You've already seen a GPF before; it's the infamous `segmentation fault: core dumped` message you've seen when accessing memory outside of the program's allocated memory that confuses the living daylight out of programmers because it provides no other useful information. A segfault is a type of GPF.

Well, if an instruction needs aligned memory, how do you create memory that is aligned? Most SSE instructions that operate on aligned memory require memory aligned to 16 bytes (aka 128 bits, the length of an `xmm` register). You can do this using `malloc` in a cool way. Let's look at this code that generates 1024 bytes of 16-byte-aligned memory:

```
void *mem = malloc(1024 * 15);
void *ptr = ((uintptr_t)mem + 15) & ~(uintptr_t)0xF;
// Do whatever you need aligned with *ptr instead of *mem
free(mem)
```

Wait, why allocate $1024 + 15$ bytes? And why are there two pointers? This is because somewhere within those $1024 + 15$ bytes of memory, there is a memory location that is 16-byte-aligned that we can use. To get this pointer, you need to do some pointer arithmetic and use an inverse mask. You add 15 to this `mem` pointer again, and then mask off the last hex digit of the address using `not 0xF` (an inverse mask that clears the last bits). And there you go! There are `uintptr_t` casts to avoid GCC warnings and to ensure the correct size for the mask. You can technically do this with void pointers but it throws a warning. At the end, you'll notice that instead of freeing `ptr`, `mem` gets freed. This is `free` can ONLY free values assigned by `malloc`; any other value provided to `free` is a huge disaster and will always lead to undefined behavior of some sort or even stop executing in an indeterminate state.

SSE Instruction List

A lot of SSE instructions exist. They're used for everything from machine learning to neural networks to even the regular audio and video that you use today (this is where SSE and AVX get their names from). Here is a list of them sourced from the Intel x86 assembly manual; this is definitely incomplete, but it has most of the instructions you will use for this class.

Here are some terms for clarification:

- *packed*: multiple numbers stored into a single register (i.e. packed together)
- *scalar*: a single number, as compared to packing multiple numbers
- *single-precision floating-point*: equivalent to C/C++ `float` type (32 bits)
- *double-precision floating-point*: equivalent to C/C++ `double` type (64 bits)

Now that you know these, here's the list of instructions. They are sourced from Section 5.5 of the Intel x86 instruction reference.

- **Data Transfer Instructions**
  - `movaps` :: Move 4 aligned packed single-precision floating-point values between XMM registers and/or memory
  - `movups` :: Move 4 unaligned packed single-precision floating-point values between XMM registers and/or memory (much slower than `movaps`, but it does work for both aligned and unaligned memory)
  - `movss` :: Move scalar single-precision floating-point value between XMM registers and/or memory
  - Other interesting variants: `movhps`, `movhlps`, `movlps`, `movlhps`, `movmskps` (I do not believe that you will use these in the class, but you can refer to their sections in the manual).
- **SSE Arithmetic Instructions**
  - `addps` :: Add packed single-precision floating-point values
  - `addss` :: Add scalar single-precision floating-point values
  - `subps` :: Subtract packed single-precision floating-point values
  - `subss` :: Subtract scalar single-precision floating-point values
  - `mulps` :: Multiply packed single-precision floating-point values
  - `mulss` :: Multiply scalar single-precision floating-point values
  - `divps` :: Divide packed single-precision floating-point values
  - `divss` :: Divide scalar single-precision floating-point values
  - `rcpps` :: Reciprocals of packed single-precision floating-point values
  - `rcpss` :: Reciprocal of scalar single-precision floating-point values
  - `sqrtps` :: Square roots of packed single-precision floating-point values
  - `sqrtss` :: Square root of scalar single-precision floating-point values
  - `rsqrtps` :: Reciprocals of square roots of packed single-precision floating-point values
  - `rsqrtss` :: Reciprocal of square root of scalar single-precision floating-point values
  - `maxps` :: Maximum of packed single-precision floating-point values

  - `maxss` :: Maximum of scalar single-precision floating-point values
  - `minps` :: Minimum of packed single-precision floating-point values
  - `minss` :: Minimum of scalar single-precision floating-point values
- **SSE Comparison Instructions**
  - `cmpps` :: Compare packed single-precision floating-point values
  - `cmpss` :: Compare scalar single-precision floating-point values
- **SSE Logical Instructions**
  - `andps` :: Bitwise AND of packed single-precision floating-point values
  - `andnps` :: Bitwise AND NOT of packed single-precision floating-point values
  - `orps` :: Bitwise OR of packed single-precision floating-point values
  - `xorps` :: Bitwise XOR of packed single-precision floating-point values
- **Other SSE Instructions** (*these might come up, maybe, maybe not*)
  - `shufps` :: Shuffle values in packed single-precision floating-point values
  - `haddps` :: Performs a single-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand
  - `hsubps` :: Performs a single-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the fourth element of the first operand from the third element of the first operand; the third by subtracting the second element of the second operand from the first element of the second operand; and the fourth by subtracting the fourth element of the second operand from the third element of the second operand
  - `phaddw` :: Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand
  - `phaddsw` :: Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand
  - `phaddd` :: Adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand
  - `phsubw` :: Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed 16-bit results are packed and written to the destination operand.
  - `phsubsw` :: Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from

the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed and written to the destination operand

- `phsubd` :: Performs horizontal subtraction on each adjacent pair of 3-bit signed integers by subtracting the most significant doubleword from the least significant double word of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand

Note: Variants of the arithmetic and logical instructions all exist for double-precision floating-point values except for the reciprocal instructions (I have no explanation as to why). Replace the last `s` in each instruction with `d` to get double-precision variants of the same commands.

There are many, many variants of these commands. I don't want to copy paste all of them here manually and format them, so if you by any chance need other instructions, then you can refer to Intel's website for the assembly reference and look in section 5.5 to see the rest.