CS-118-02 Week 10

Varun Narravula

2022-04-04

Shift vs. Rotate vs. Arithmetic Shift

I have no idea why this was gone over, but I'm including it anyway.

You've done bitwise shifts before in C and C++ using `<<` and `>>`. In assembly, their equivalents are `shl` (*shift left*) and `shr` (*shift right*). `rax >> 1` If you want to perform the equivalent of `$ax << 2` in assembly, you would write `shl ax, 2`; if you wanted to shift right by 2, then you do `shr ax, 2`.

However, what about rotating? Rotating is a bit like shifting, except that if any values go off the sides on a shift, they will be wrapped onto the other side. If you rotate a number `0b00001011` right by 1, then it would result in `0b10000101`, where the 1 that got shifted got put back on the other side instead of getting discarded.

Another instruction that looks similar to the normal shifts is `sal` (*shift arithmetic right*) and `sar` (*shift arithmetic left*). What does this do? Normal shift instructions only work on unsigned values, and they ignore the sign bit if the number is stored in two's complement notation. Arithmetic shifts like these will make sure to preserve the sign (this is called sign extension, and is seen in instructions like `movsx` that have the same behavior).

`test` vs. `cmp`

There is another instruction called `test`, which works similarly to `cmp`. However, it performs a bitwise `and` (i.e. `test eax, eax` would compute `and eax, eax`) instead of subtracting (i.e. `cmp, eax, 0` would compute `sub eax, 0`) and set flags depending on the result. You can use this to compare as well and jump based on flags set based on those comparisons.

Intrinsics

What are intrinsics? You already know how SSE can make things much more efficient because it is a SIMD instruction set. However, it's extremely difficult to do this using assembly; most people use intrinsics instead. It is the same instruction set, but it uses C, which is much easier than assembly because of the

access to much more robust C tooling. Intel provides an SDK that interfaces with SSE instructions and looks similar to actual assembly code; all you need to do is include a single header. There is no need to deal with compliated compile instructions anymore; all you need is a C compiler and the `-msse4` flag in your compile command.

Every SSE instruction has a corresponding C function that is described on Intel's website with comprehensive documentation on what parameters it takes, what architectures and microcode and chipsets it is supported by, and other things. To look it up, you can type whatever instruction or method you need in the search bar and it will give you a description of what the function does, its corresponding assembly mnemonic, as well as a few other things such as the parameters it can take. You can select from a few different instruction sets, like AVX and whatnot, but don't worry about these for now. Refer to Intel's assembly guide if you don't know what to search for.

Raw link: [https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html]

Standard Deviation Using Intrinsics

A standard deviation, if you do not know what it is from statistics, is a description of the amount of difference from an average for a given dataset. The mathematical formula for calculating standard deviation is this:

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

Where:

- σ = standard deviation
- N = size of dataset
- x_i = each value from the dataset
- μ = mean of dataset

In English, this is the square root of the sum of squared differences from the mean divided by the size of the data set.

This isn't a math class, what does this have to do with assembly? Well, it has nothing directly, but calculating the standard deviation using SSE is a good example of why using intrinsics can be much simpler than using straight assembly.

Given an dataset of numbers (say an array of type float), how would you calculate the standard deviation of it using intrinsics? The solution is here (try it yourself first!):

```
float sum(float *array, size_t size) {
  __m128 accumulator = _mm_setzero_ps();
```

```
  size_t i = 0;

  for (; i < ((size) & ~((4) - 1)); i += 4) {
    accumulator = _mm_add_ps(accumulator, _mm_loadu_ps(array + i));
  }

  accumulator = _mm_hadd_ps(accumulator, accumulator);
  accumulator = _mm_hadd_ps(accumulator, accumulator);

  for (; i < size; i++) {
    accumulator = _mm_add_ss(accumulator, _mm_load_ss(array + i));
  }

  return _mm_cvtss_f32(accumulator);
}

float standard_deviation(float *array, size_t size) {
  float average = sum(array, size) / size;
  const __m128 diff = _mm_set1_ps(average);

  __m128 accumulator = _mm_setzero_ps();
  size_t i = 0;

  for (; i < ((size) & ~((4) - 1)); i += 4) {
    __m128 sub = _mm_sub_ps(_mm_loadu_ps(array + i), diff);
    accumulator = _mm_add_ps(accumulator, _mm_mul_ps(sub, sub));
  };

  accumulator = _mm_hadd_ps(accumulator, accumulator);
  accumulator = _mm_hadd_ps(accumulator, accumulator);

  float deviations = _mm_cvtss_f32(accumulator);

  for (; i < size; i++) {
    float sub = array[i] - average;
    deviations += sub * sub;
  }

  return sqrt(deviations / size); // #include <math.h>
}
```

I separated out the `sum` function to make it easier to read and less of a monolith. But hold on, what are all those underscores? And where's the assembly? All of these functions can be used by including specific header files that are described in Intel's intrinsics documentation from above. Some of them have slightly different names, but roughly speaking, here are the instructions used and what

they correspond to (if they have a corresponding instruction):

- `_mm_setzero_ps` :: `xorps <reg>, <reg>` (Clears a register to zeros)
- `_mm_loadu_ps` :: `movups`
- `_mm_load_ss` :: `movss`
- `_mm_set1_ps` :: ***intrinsics-unique*** (Broadcast 32-bit float to all parts of xmm register)
- `_mm_add_ps` :: `addps`
- `_mm_add_ss` :: `addss`
- `_mm_hadd_ps` :: `haddps`
- `_mm_sub_ps` :: `subps`
- `_mm_mul_ps` :: `mulps`
- `_mm_cvtss_f32` :: `movss <m32>, <xmm>` (Moves lower word of xmm register to location)

You'll notice that there are some extra convenience functions included in the intrinsics that Intel provides that are extremely useful. `set1_ps` is much more intuitive than `movss` and shuffling a value around, and `setzero_ps` is explicit in what it does, unlike `xorps <reg> <reg>`. Right away, you see that intrinsics can be more readable than straight assembly, despite the underscores.

Another great advantage to this is that you do not have to take into account the fact that only `xmm0-xmm15` exists. You can now assign however many registers you need, and the compiler will do some magic to only use the 16 registers the CPU has! Now you don't have to take into account yet another limit that straight assembly places on you.

Well, how does this work? Essentially, it sums the array using an external counter in `sum`. To sum up the remaining <4 elements, instead of attempting to push it into another SSE register (that would cause a segfault), you would normally loop over it using the same external counter for the rest of the array. `xmm` registers are now of a type `__m128` (aka 128-bit register), and you can bind them to C variables. Here, an accumulator register is used to sum up the values using SSE, and then it is horizontally summed twice with itself to get the result of the SSE summations. In the `sum` function, `addss` is used to add scalars to the bottom dword of the register (now all other parts of the register are ignored). This is then extracted as a float in the return value using `_mm_cvtss_f32`.

For the part that calculates the standard deviation, it is similar. It calculates the average using the `sum` function (this is μ in the standard deviation equation), broadcasts (fills) this average into all the `f32` slots in another register so that it can simultaneously take differences using SSE. The loop logic is the same, where it sums up the differences from the mean using this register and an accumulator, and horizontally adds this accumulator with itself twice to sum it all up. Then it gets the remaining deviations from the mean; now you have the value of $\sum (x_i - \mu)^2$. I included `<math.h>` and got the square root with it, because I exracted the deviations and used a normal C loop without intrinsics for the last part unlike what I used in `sum` with the scalar instructions, but it should work

either way. Try not to copy the example though, then you won't learn anything.

And there you go! You can now pass in an array of floats of any size, and you should be able to calculate the standard deviation from it. That's all for the notes for this week, the prof didn't go into depth about much of anything.