Overview

- SIMD = single instruction multiple data
- Special instructions which can operate on multiple values at once
- Not really a "modern" feature
  - SSE released 1999 (Pentium III)
  - SSE2 (current reg. set) in 2001. (Pentium 4)
- Exist in ARM as "NEON"
  - Today → x86 SSE/AVX.

Use cases

- Iterating over an array and doing something to each element.

1	2	3	4	5

} add 1 to each → easily vectorizable!

- Performing a reduction (summing elems in a vector)
- Complex math (trig, sqrt, recipricol)
- Lots of random instructions in x86!
  - Show Intel intrinsics website.
    - sqrt.
    - add
    - swizzle

- SIMD can be used to parallelize on a single core.
  - It's not scary, and it's not a micro-optimization! Can give ~8x speedups*

* If you're compute bound. Depends on application.

What Can Your Compiler Do?

- I'll talk about Clang, but GCC similar
  - ICC can do even fancier stuff!

- -ftree-vectorize.
  - -O3 automatically does this.

- -Rpass = ...
  for diagnostics

Demo.

- All but sqrt-each ~~+e~~ and sum-large-floats vectorizes.
- Enable -ffast-math

  Aside: you should use -ffast-math
  - Breaks IEEE 754 Standard (allows reordering of float + and *)
  - Enables certain approx.
  - Prevents NaN check
  - Disables errno
- Uncomment if. Float (sum-large-floats) doesn't vectorize.
  - Why? Don't know...

Show aliasing code (memory check in update)

- restrict definition.

So your compiler can actually do a lot, but it can be fickle. Use the diagnostic messages and check the assembly to see if it's doing what you want before hand optimizing.

Some tips

- Use restrict whenever possible (__restrict__) ← C++
- Avoid branches if you want auto-vectorization
- Use unit stride (innermost loop does i++).

③

← Header

```
#include <immintrin.h>

// Sums all the values in v which are greater than 100.
double sum_large_floats_vectorized(const float *v, unsigned length, unsigned
    trials) {

    double result = 0.0f;

    for (unsigned t = 0; t < trials; t++) {

        __m128 cutoff = _mm_set1_ps(10.0f);
        __m128 sum = _mm_setzero_ps();

        float final[4] __attribute__((aligned(16)));

        unsigned i;

        for (i = 0; i+4 <= length; i+=4) {
            __m128 data = _mm_loadu_ps(v + i);
            __m128 mask = _mm_cmpge_ps(data, cutoff);
            __m128 masked = _mm_and_ps(mask, data);

            // Update the sum.
            sum = _mm_add_ps(sum, masked);
        }

        // Handle the fringe case.
        for (; i < length; i++) {
            float val = v[i];
            if (val >= 10.0f) {
                result += val;
            }
        }

        // Sum up the sum vector.
        // TODO there are faster ways of doing this! See hadd, etc.
        _mm_store_ps(final, sum);
        result += (final[0] + final[1] + final[2] + final[3]);
    }
    return result;
}
```

128 bit type.

to use store instead of storeu.

In case not divisible by 4

Other techniques for best performance:

- Unrolling (check assembly! Might happen automatically).
- Understanding the ISA
    - which instructions can be run in parallel? How to avoid data dependencies? etc. etc.

Conclusions

- SIMD is the core principle behind GPUs
  - Just many more lanes, and some stuff we need to do manually (e.g. computing the mask) GPUs can do automatically
  - AVX-512 has masks built-in: more GPU like.

- If SIMD is so accessible (it's in every laptop, phone, tablet, etc.) and reasonably impactful, why doesn't everyone use it?
  - Maybe this is where DAWN comes in :)

  How can we make writing vector code easier?
  - Better compilers?
  - Better programmers?
  - Weld or Delite or other DSLs?