Three Optimization Tips for C++

Andrei Alexandrescu, Ph.D.
Research Scientist, Facebook
andrei.alexandrescu@fb.com

This Talk

- Basics
- Reduce strength
- Minimize array writes

Things I Shouldn’t Even

Today's Computing Architectures

- Extremely complex
- Trade reproducible performance for average speed
- Interrupts, multiprocessing are the norm
- Dynamic frequency control is becoming common
- Virtually impossible to get identical timings for experiments

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”
- “Computation is faster than data”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”
- “Computation is faster than data”

Intuition

- Ignores aspects of a complex reality
- Makes narrow/obsolete/wrong assumptions

- “Fewer instructions = faster code”
- “Data is faster than computation”
- “Computation is faster than data”

- The only good intuition: *“I should time this.”*

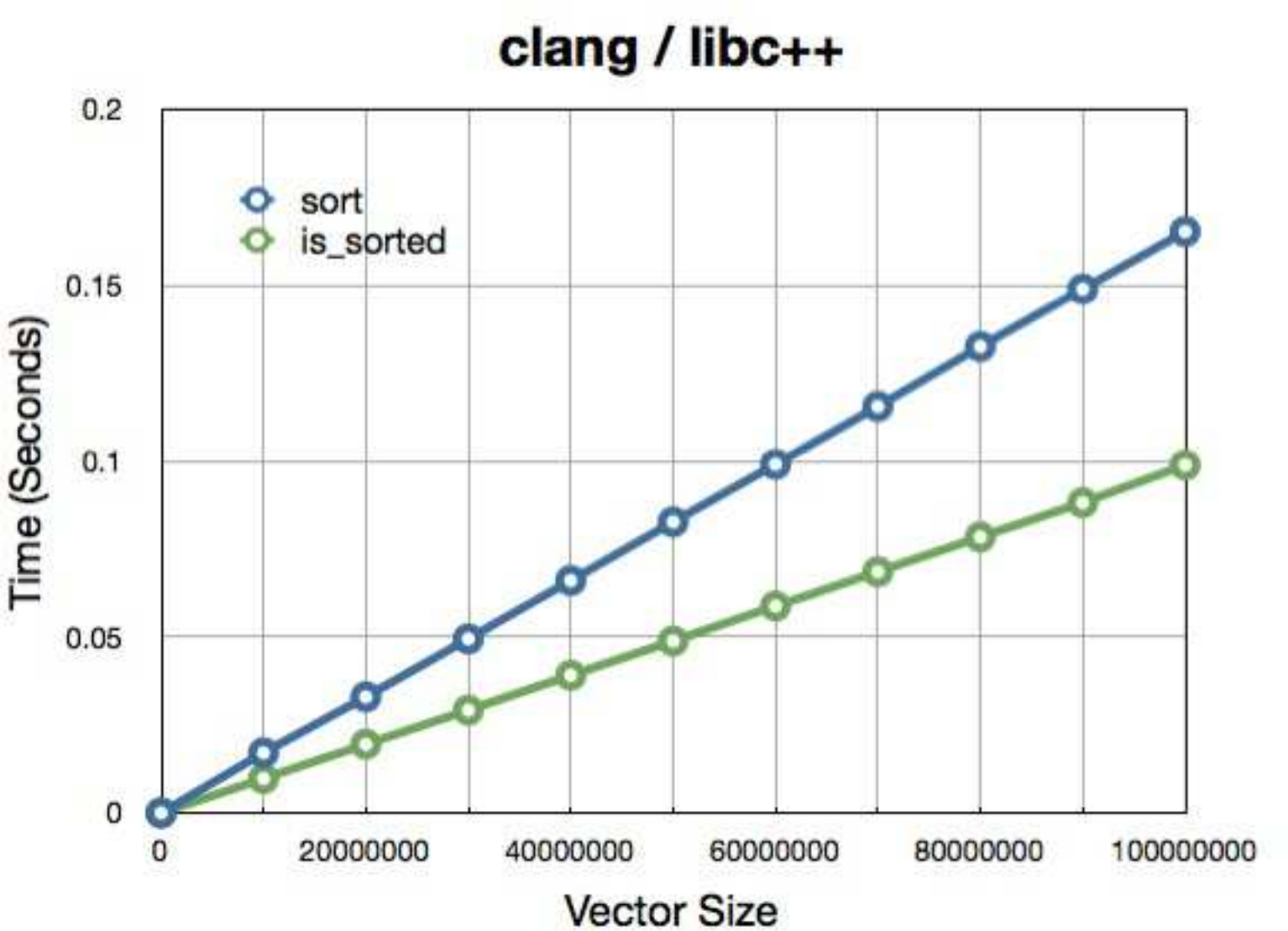
Paradox

Measuring gives you a leg up on experts who don’t need to measure

Common Pitfalls

- Measuring speed of debug builds
- Different setup for baseline and measured
  - Sequencing: heap allocator
  - Warmth of cache, files, databases, DNS
- Including ancillary work in measurement
  - `malloc`, `printf` common
- Mixtures: measure $t_a + t_b$, improve t_a, conclude t_b got improved
- Optimize rare cases, pessimize others

Optimizing Rare Cases



More generalities

- Prefer static linking and PDC
- Prefer 64-bit code, 32-bit data
- Prefer (32-bit) array indexing to pointers
  - Prefer `a[i++]` to `a[++i]`
- Prefer regular memory access patterns
- Minimize flow, avoid data dependencies

Storage Pecking Order

- Use `enum` for integral constants
- Use `static const` for other immutables
  - Beware cache issues
- Use stack for most variables
- Globals: aliasing issues
- `thread_local` slowest, use local caching
  - 1 instruction in Windows, Linux
  - 3-4 in OSX

Reduce Strength

Strength reduction

- Speed hierarchy:
  - comparisons
  - (u)int add, subtract, bitops, shift
  - FP add, sub (separate unit!)
  - Indexed array access
  - (u)int32 mul; FP mul
  - FP division, remainder
  - (u)int division, remainder

Your Compiler Called

I get it. `a >>= 1` is the same as `a /= 2`.

Integrals

- Prefer 32-bit ints to all other sizes
  - 64 bit may make some code slower
  - 8, 16-bit computations use conversion to 32 bits and back
  - Use small ints in arrays
- Prefer unsigned to signed
  - Except when converting to floating point
- “Most numbers are small”

Floating Point

- Double precision as fast as single precision
- Extended precision just a bit slower
- Do not mix the three
- 1-2 FP addition/subtraction units
- 1-2 FP multiplication/division units
- SSE accelerates throughput for certain computation kernels
- ints→FPs cheap, FPs→ints expensive

Advice

Design algorithms to use minimum operation strength

Strength reduction: Example

- Digit count in base-10 representation

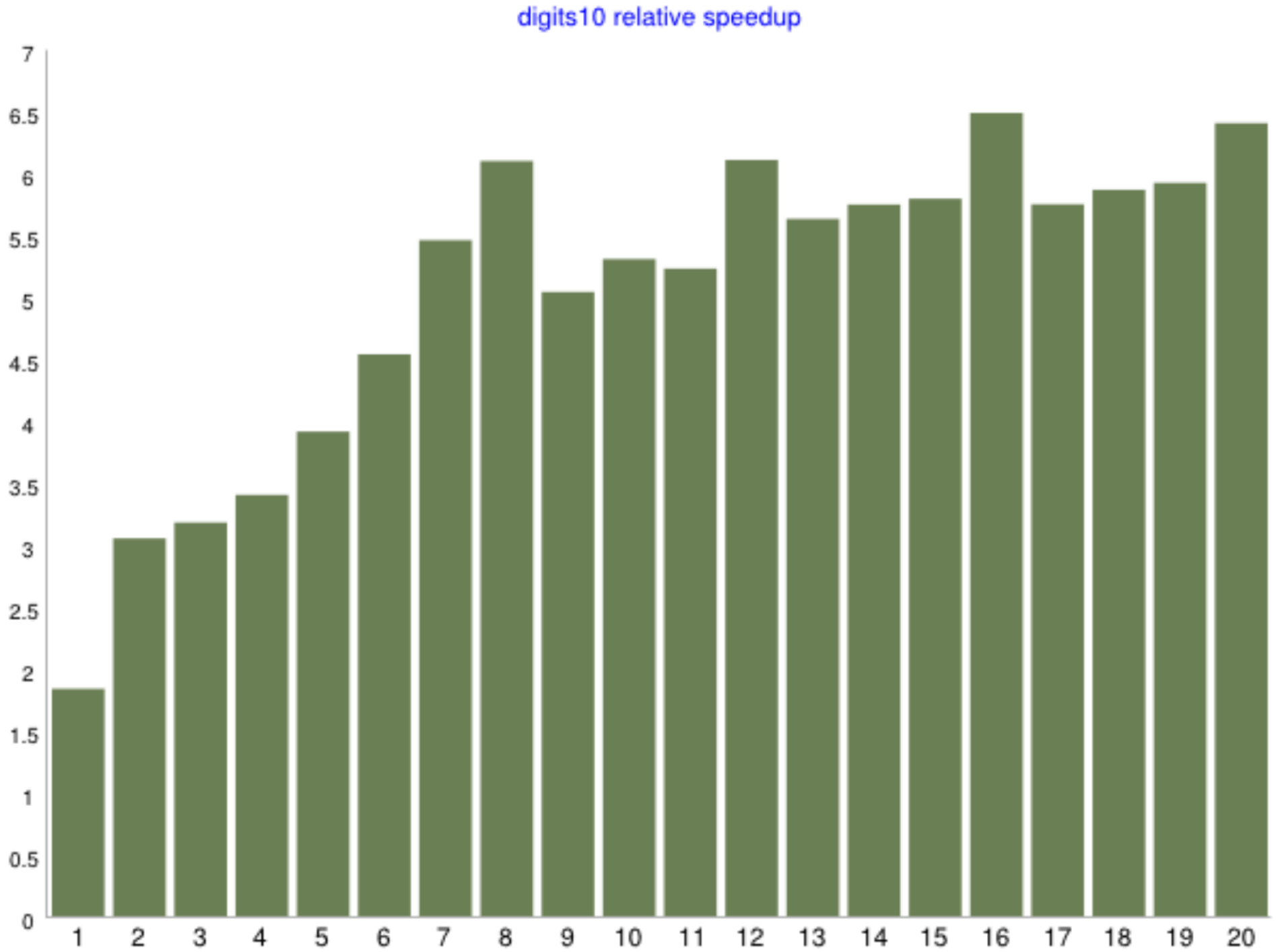
```
uint32_t digits10(uint64_t v) {
   uint32_t result = 0;
   do {
      ++result;
      v /= 10;
   } while (v);
   return result;
}
```

- Uses integral division extensively
  - (Actually: multiplication)

Strength reduction: Example

```
uint32_t digits10(uint64_t v) {
    uint32_t result = 1;
    for (;;) {
        if (v < 10) return result;
        if (v < 100) return result + 1;
        if (v < 1000) return result + 2;
        if (v < 10000) return result + 3;
        // Skip ahead by 4 orders of magnitude
        v /= 10000U;
        result += 4;
    }
}
```

- More comparisons and additions, fewer `/=`
- (This is not loop unrolling!)


digits10 relative speedup
7
6.5
6
5.5
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

Minimize Array Writes

Minimize Array Writes: Why?

- Disables enregistering
- A write is really a read and a write
- Aliasing makes things difficult
- Maculates the cache

- Generally just difficult to optimize

Minimize Array Writes

```
uint32_t u64ToAsciiClassic(uint64_t value, char* dst) {
   // Write backwards.
   auto start = dst;
   do {
      *dst++ = '0' + (value % 10);
      value /= 10;
   } while (value != 0);
   const uint32_t result = dst - start;
   // Reverse in place.
   for (dst--; dst > start; start++, dst--) {
      std::iter_swap(dst, start);
   }
   return result;
}
```

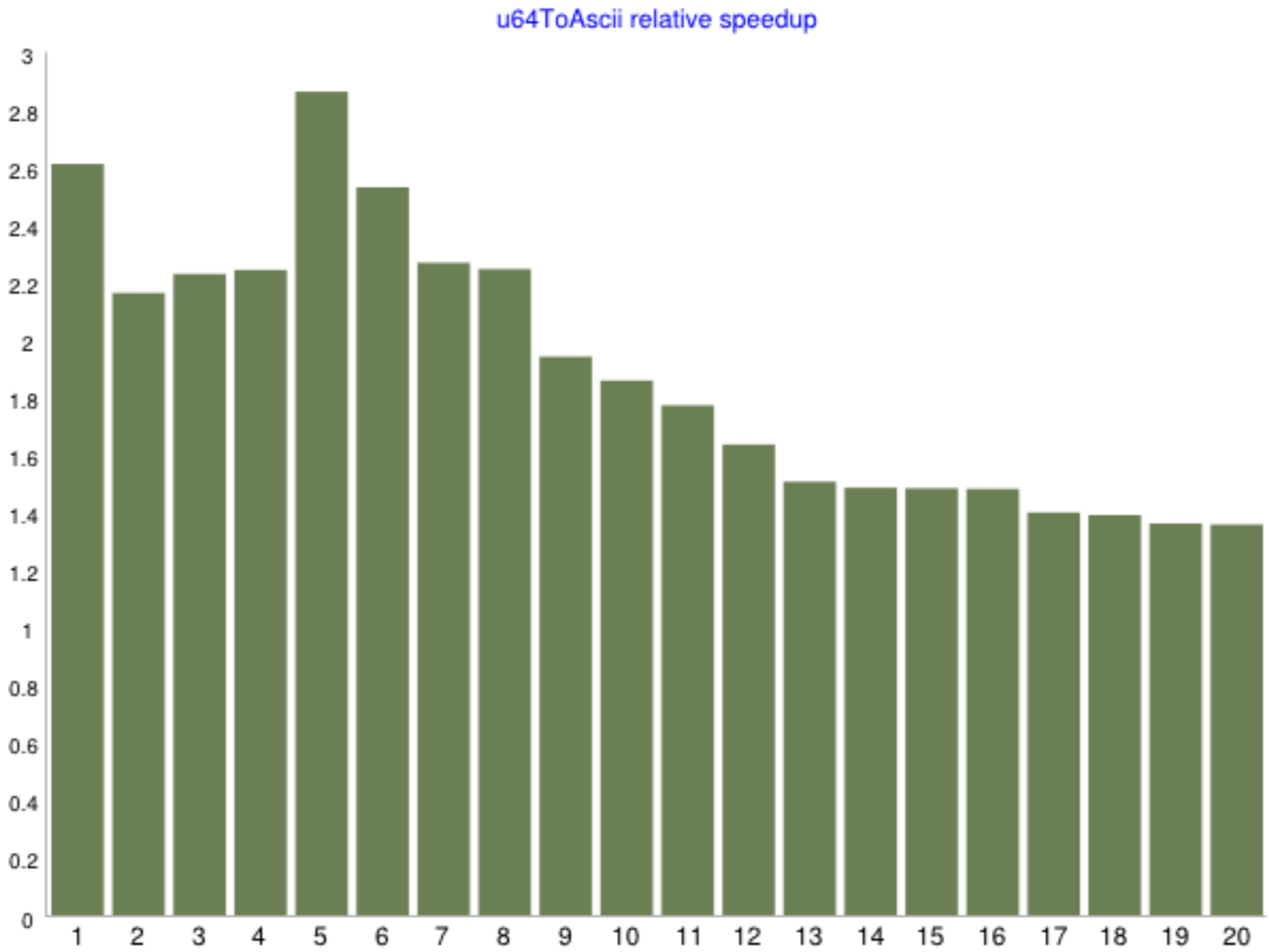
Minimize Array Writes

- Gambit: make one extra pass to compute length

```
uint32_t uint64ToAscii(uint64_t v, char *const buffer) {
   auto const result = digits10(v);
   uint32_t pos = result - 1;
   while (v >= 10) {
      auto const q = v / 10;
      auto const r = static_cast<uint32_t>(v % 10);
      buffer[pos--] = '0' + r;
      v = q;
   }
   assert(pos == 0);
   // Last digit is trivial to handle
   *buffer = static_cast<uint32_t>(v) + '0';
   return result;
}
```

Improvements

- Fewer array writes
- Regular access patterns
- Fast on small numbers
- Data dependencies reduced


u64ToAscii relative speedup
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

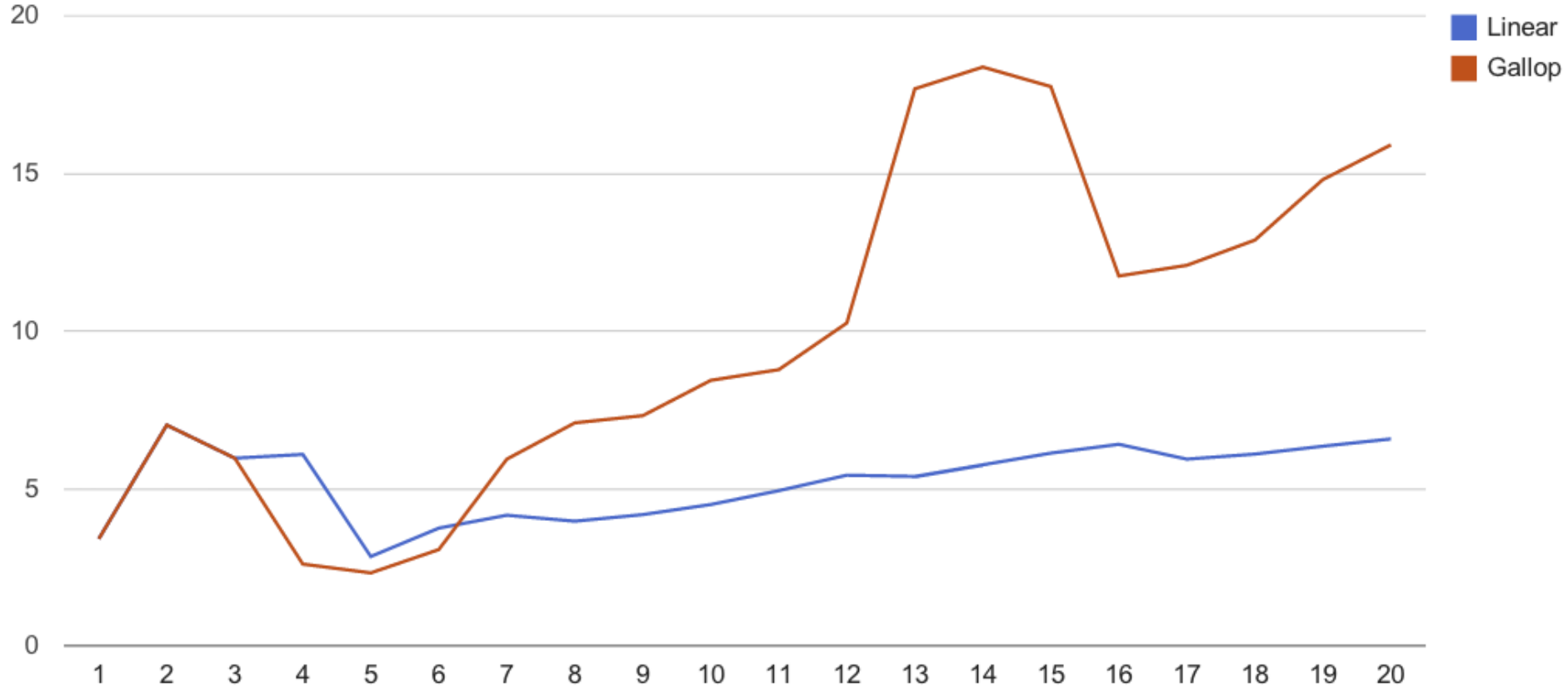
One More Pass

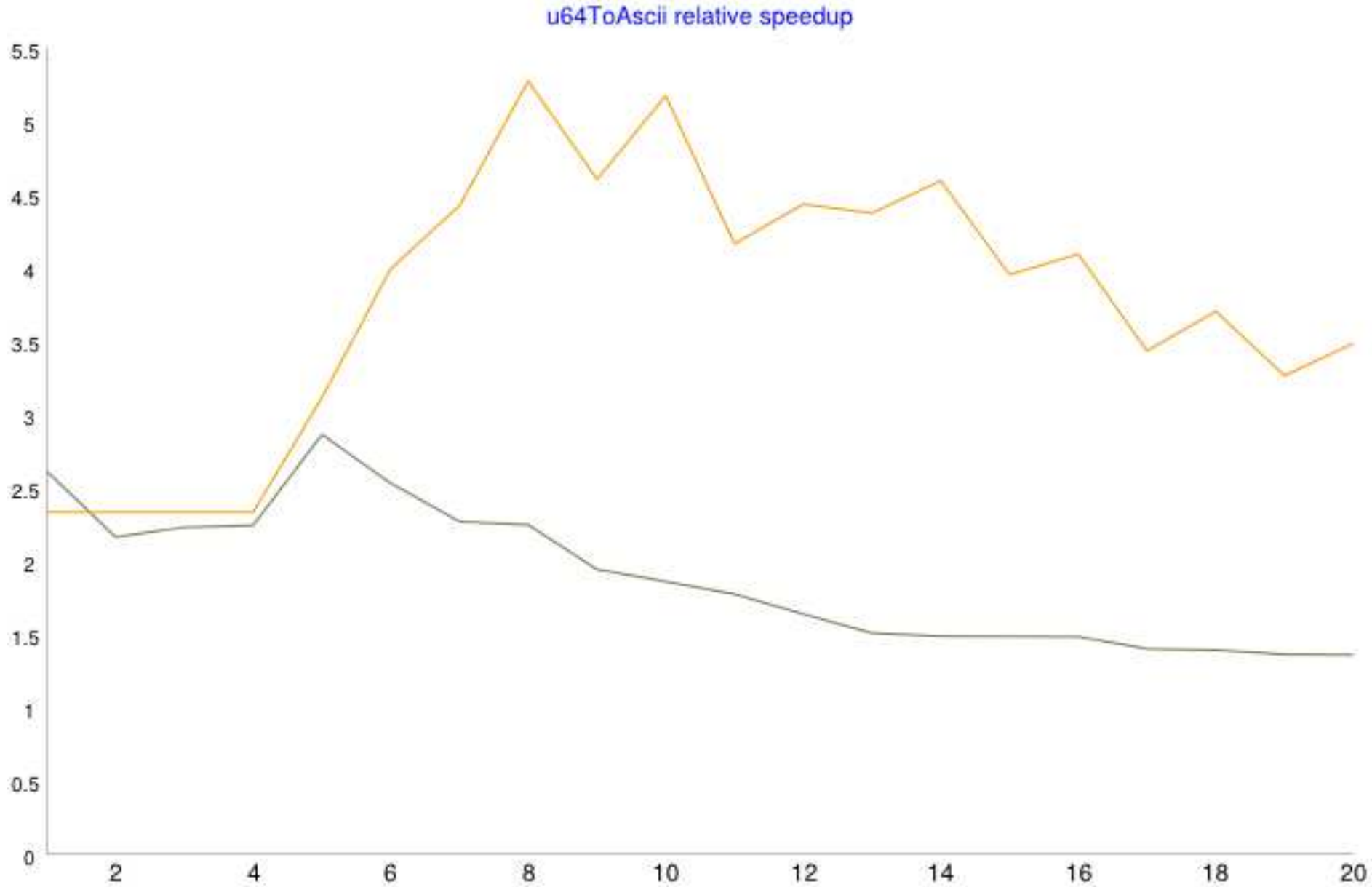
- Reformulate `digits10` as search
- Convert two digits at a time

```
uint32_t digits10(uint64_t v) {
    if (v < P01) return 1;
    if (v < P02) return 2;
    if (v < P03) return 3;
    if (v < P12) {
        if (v < P08) {
            if (v < P06) {
                if (v < P04) return 4;
                return 5 + (v < P05);
            }
            return 7 + (v >= P07);
        }
        if (v < P10) {
            return 9 + (v >= P09);
        }
        return 11 + (v >= P11);
    }
    return 12 + digits10(v / P12);
}
```

```
unsigned u64ToAsciiTable(uint64_t value, char* dst) {
   static const char digits[201] =
      "000102030405060708091011121314151617181920"
      "212223242526272829303132333435363738394041"
      "424344454647484950515253545556575859606162"
      "636465666768697071727374757677787980818283"
      "848586878889909192939495969798990";
   uint32_t const length = digits10(value);
   uint32_t next = length - 1;
   while (value >= 100) {
      auto const i = (value % 100) * 2;
      value /= 100;
      dst[next] = digits[i + 1];
      dst[next - 1] = digits[i];
      next -= 2;
   }
```

```
    // Handle last 1-2 digits
    if (value < 10) {
      dst[next] = '0' + uint32_t(value);
    } else {
      auto i = uint32_t(value) * 2;
      dst[next] = digits[i + 1];
      dst[next - 1] = digits[i];
    }
    return length;
}
```


20
15
10
5
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Linear
Gallop


u64ToAscii relative speedup
5.5
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
2
4
6
8
10
12
14
16
18
20

Summary

Summary

- You can’t improve what you can’t measure
  - Pro tip: You can’t measure what you don’t measure
- Reduce strength
- Minimize array writes