Writing Fast Code

code::dive Conference, Wrocław, Poland

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

Nov 5, 2015

The Art of Benchmarking

Mind Amdahl's Law

- Improvement from a component in a system is limited by the component's participation to the system

- Collect app profile data before optimizing
- Focus on optimizing the top time-spenders

Mind Amdahl's Law

Choose hotspots from whole application

Mind Lhadma's Law

Optimize hotspots outside whole application; profile again within application

Discussion

- Edit/benchmark cycle slow on whole app
- But optimizations have global effects
  - Cache effects
  - Memory allocation/use
- Microbenchmarks' effectiveness decreases with margin

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

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

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

Deep Thought

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

Benchmarking for Speed

- Goal: estimate the speed of some specific algorithm
- Non-goals:
  - Framework overheads (looping, measurement)
  - Clock resolution noise
  - Vagaries during measurement (unrelated system activity etc)
  - Unrelated activity overheads: setup/cleanup work

Measuring

- Measured time: $t_m = t + t_q + t_n + t_o$

  where

- t_m is measured (observed) time
- t is the actual time of interest
- t_q is time added by quantization noise
- t_n is time added by various sources of noise
- t_o is overhead time (measuring, looping, calling functions)

1. Quantization noise

- Especially visible for very fast operations
- Solution: *repeat experiment many times, measure once*
- Adjust total time spent to make noise negligible

```
startClock();
for (unsigned i = 0; i < iMax; ++i) {
   ... experiment code ...
}
measureTime();
```

2. Other noise

- All fluctuations
- Gaussian, but always additive
- Solution: *measure many times, take the mode*

```
auto minTime = numeric_limits<unsigned>::max();
for (unsigned epoch = 0; epoch < eMax; ++epoch) {
    startClock();
    for (unsigned i = 0; i < iMax; ++i) {
       ... code to benchmark ...
    }
    auto t = measureTime();
    if (minTime > t) minTime = t;
}
```


Mode vs. Minimum

- Mode of a distribution: maximum density of measured values
- Practically: Mode slowly converges to minimum
  - No “subtractive” speed noise

- Exceptions: Networking, (live|dead)locking
- Sometimes worst case *does* matter

3. Overhead noise

- All code surrounding the actual work
- Solution: *measure and subtract the time for the loop only*

```
startClock();
for (unsigned i = 0; i < iMax; ++i) {
}
measureTime();
```


Wait a Minute

- Compiler optimizes away non-observable behavior
- Empty loops are not observable
- Solution: make them observable

```
startClock();
for (volatile unsigned i = 0;
      i < iMax; ++i) {
}
measureTime();
```

Better, but less portable

```
startClock();
for (unsigned i = 0; i < iMax; ++i) {
   asm volatile("");
}
measureTime();
```


Speaking of observable

- Frequently benchmarks don't use their results

```
// Let's time this
unsigned n = 1000000;
while (n-- > 0) {
   asciiToInteger("1234567890");
}
```

- Compiler may call `asciiToInteger` once or not at all!

Solution

- Define function that “uses” its parameter

```
template <class T>
void doNotOptimizeAway(T&& datum) {
   asm volatile("" : "+r" (datum));
}
```


Touching entire memory

```
void clobber() {
   asm volatile("" : : : "memory");
}
```

- Useful to insert at the end of a measurement

More Portable

- Use I/O guarded by an non-provable false test

```
template <class T>
void doNotOptimizeAway(T&& datum) {
   if (getpid() == 1) {
      const void* p = &datum;
      putchar(*static_cast<const char*>(p));
   }
}
```


Use

- Define function that “uses” its parameter

```
// Let’s time this
unsigned n = 1000000;
while (n-- > 0) {
   doNotOptimizeAway(
      asciiToInteger("1234567890"));
}
```

Which Counter to Use?

- Not a solved problem!
- High resolution, high fluctuation
  - Windows: `QueryPerformanceCounter`
  - Linux: `clock_gettime(CLOCK_REALTIME)`
    - `clock_getres` broken on older kernels, upgrade
- Low-resolution, low fluctuation
  - Windows: `timeGetTime`
  - Un*x: `gettimeofday`
- CPU-specific (RDTSC)—beware CPU jumping

Baselines

Baseline, Defined

- "Classic", "naïve", "established", "commonly-accepted" solution to the target of optimization
- Your own production version of a complex function
- Slower than baseline → no good
- `std::sort` for sorting
- `iostream`s for I/O (easy to beat)
- `scanf` for parsing numbers

Immutable Rule

Always choose relevant baselines

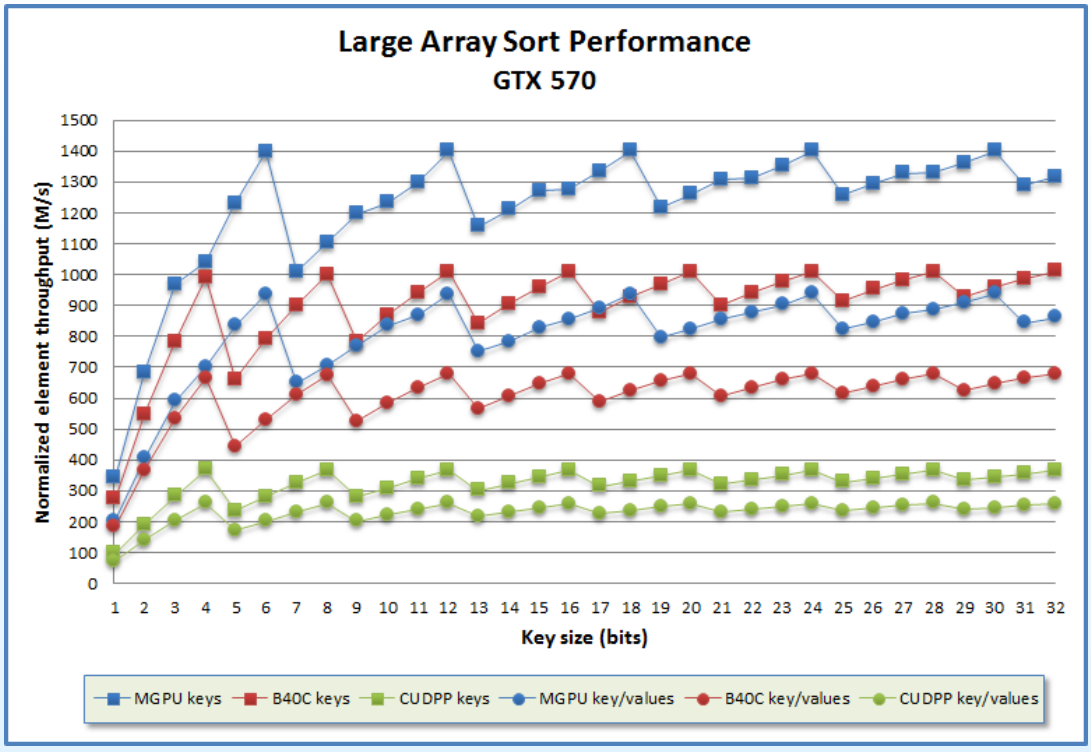
Bad

“I sorted 1M floats in 8 milliseconds!”

Good

“Quicksort is 25% faster than heapsort on 1M Guassian floats”

Splendid



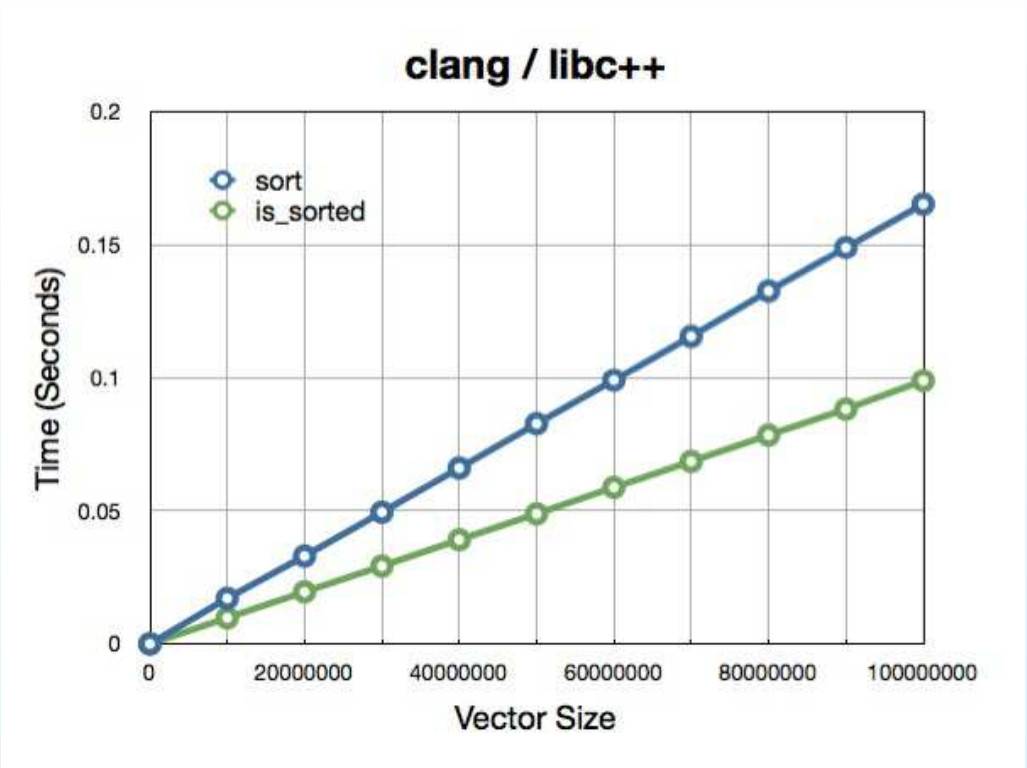
Differential Timing

- Traditional:
  - Run baseline n times, measure t_a
  - Run contender n times, measure t_b
  - relative improvement $r = \frac{t_a}{t_b}$
- Differential:
  - Run baseline $2n$ times, measure t_{2a}
  - Run baseline n times *and* contender n times, measure t_{a+b}
  - Relative improvement $r = \frac{t_{2a}}{2t_{a+b} - t_{2a}}$
- Some overhead noises canceled

Common Benchmarking 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



Almost Portable Optimization Techniques

Categories

- Generalities
- Storage
- Numbers
- Strength reduction
- Minimize indirect (e.g. array) writes

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 `static const` for all immutables
  - Beware cache issues
- Use stack for most variables
  - Hot
  - 0-cost addressing, like `struct`/`class` fields

- Globals: aliasing issues
- `thread_local` slowest, use local caching
  - 1 instruction in Windows, Linux
  - 3-4 in OSX

Integrals

- Prefer 32-bit ints to all other sizes
  - 64 bit may make some code 20x 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

Strength reduction

- Don’t waste time replacing `a /= 2` with `a >>= 1`
- Speed hierarchy:
  - comparisons
  - (u)int add, subtract, bitops, shift
  - FP add, sub (separate unit!)
  - (u)int32 mul; FP mul
  - FP division, remainder
  - (u)int division, remainder

Strength reduction: Example

- Compute digit count in base-10 representation

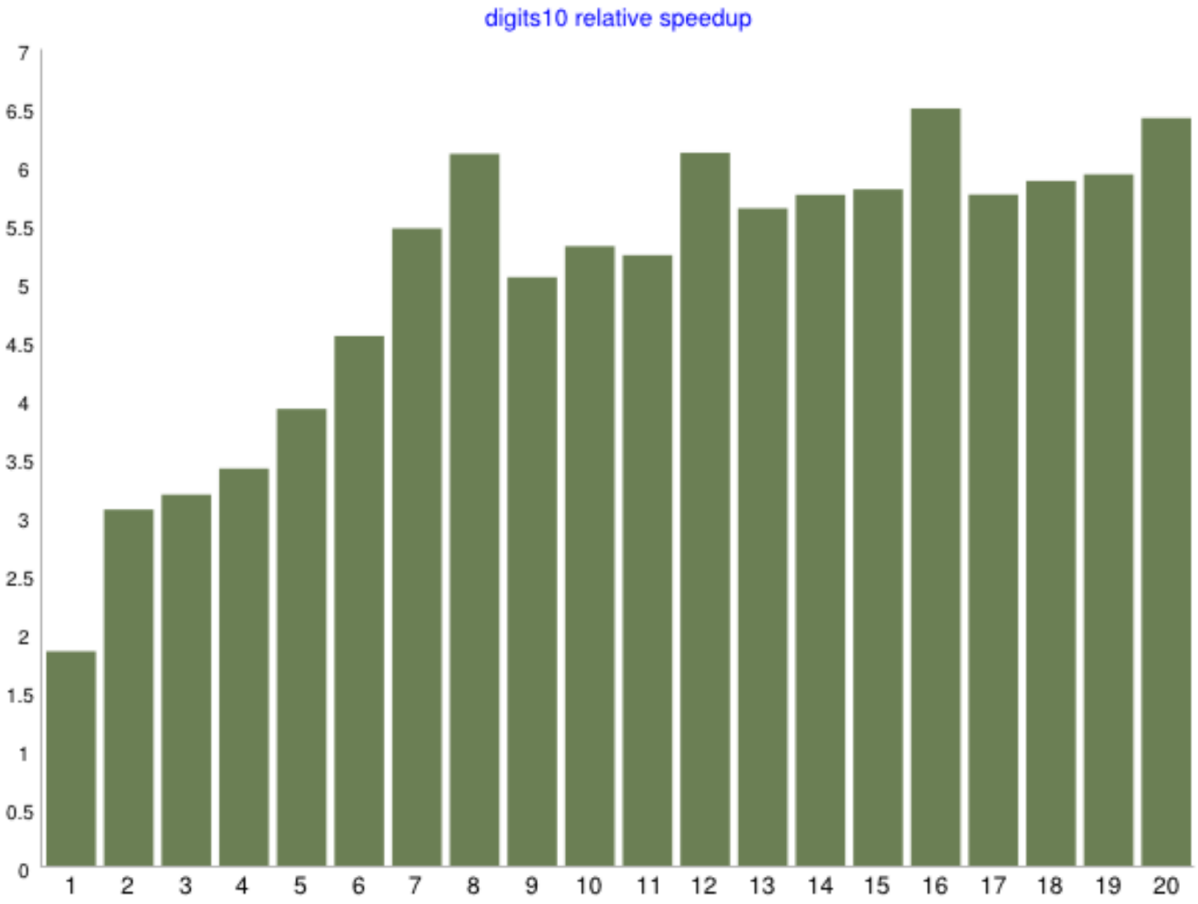
```
uint32_t digits10(uint64_t v) {
   uint32_t result = 0;
   do {
      ++result;
      v /= 10;
   } while (v);
   return result;
}
```

- Uses integral division extensively

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!)



Pro version

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

- 102 vs. 149 bytes in size
- Does not improve execution time!

LIKELY

- Uses static hinting for the compiler

```
#if defined(__GNUC__) && __GNUC__ >= 4
#define LIKELY(x)    (__builtin_expect((x), 1))
#define UNLIKELY(x) (__builtin_expect((x), 0))
#else
#define LIKELY(x)    (x)
#define UNLIKELY(x) (x)
#endif
```

Minimize Indirect Writes: Why?

- Disables enregistering
- A write is really a read and a write
- Maculates the cache

Minimize Indirect 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;
}
```

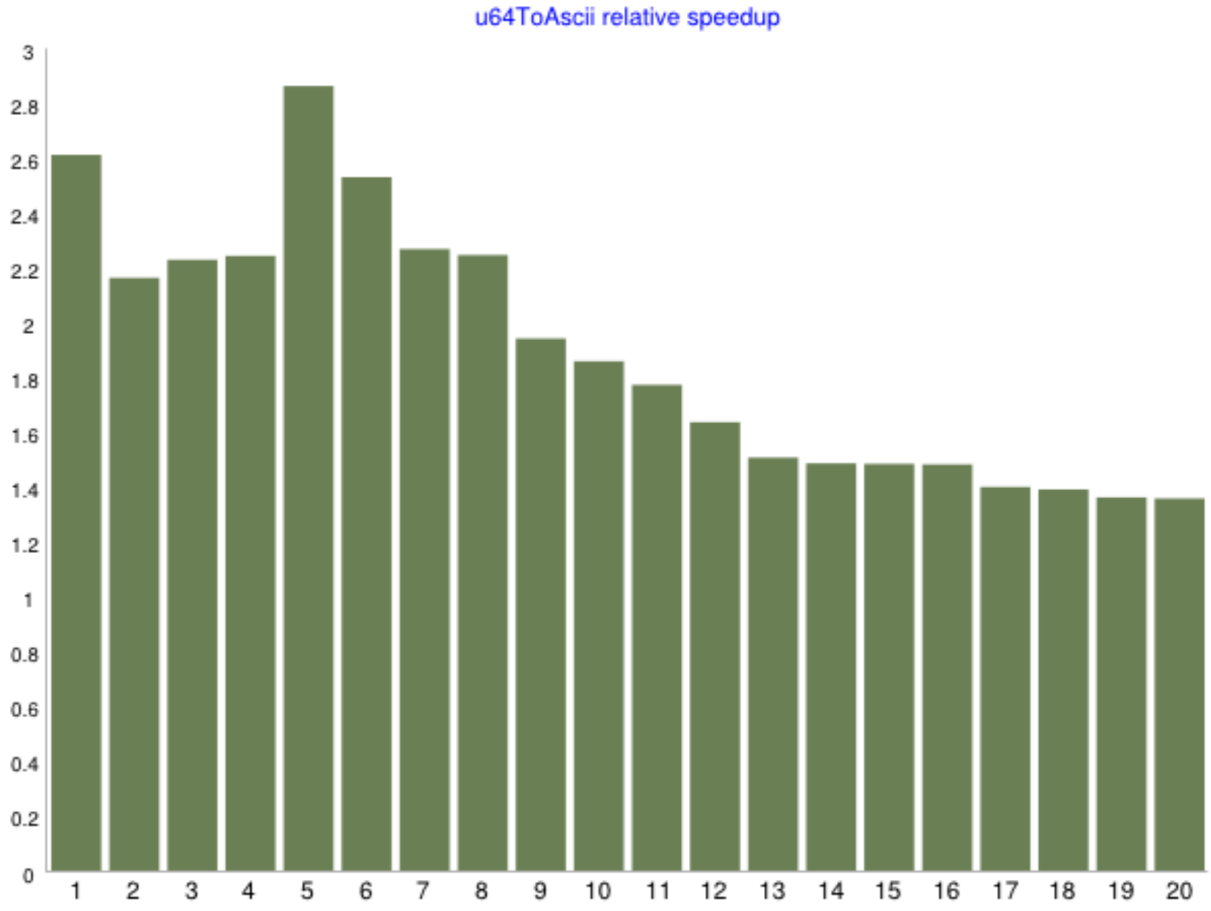
Minimize Indirect Writes

- 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

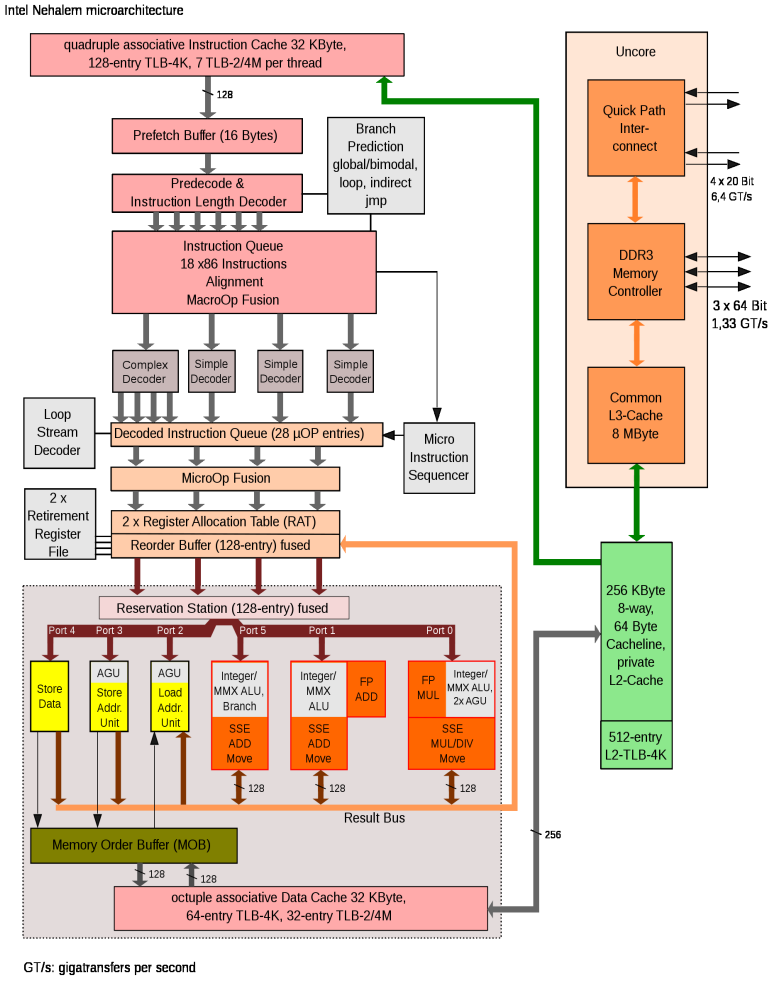
- Fewer indirect writes
- Regular access patterns
- Fast on small numbers
- Data dependencies reduced



Checkpoint

- You can’t improve what you can’t measure
  - Pro tip: You can’t measure what you don’t measure
- Always choose good baselines
- Optimize code judiciously for today’s dynamics

The Forgotten Parallelism



How to use those ALUs?

Instruction-Level Parallelism (ILP)

- Pipelining
- Superscalar execution
- Out-of-order execution
- Register renaming
- Speculative execution
- . . . and many more

better instruction-level parallelism

=

fewer data dependencies

Automatic?

Important?

Minimizing Data Dependencies

Study: string to integral

```
unsigned atoui(const char* b, const char* e) {
   enforce(b < e);
   unsigned result = 0;
   for (; b != e; ++b) {
      enforce(*b >= '0' && *b <= '9');
      result = result * 10 + (*b - '0');
   }
   return result;
}
```


How does it work?

$$523924 = (((((0 * 10 + 5) * 10 + 2) * 10 + 3) * 10 + 9) * 10 + 2) * 10 + 4$$

Divide & Conquer?

"523924" : "523" then "924"

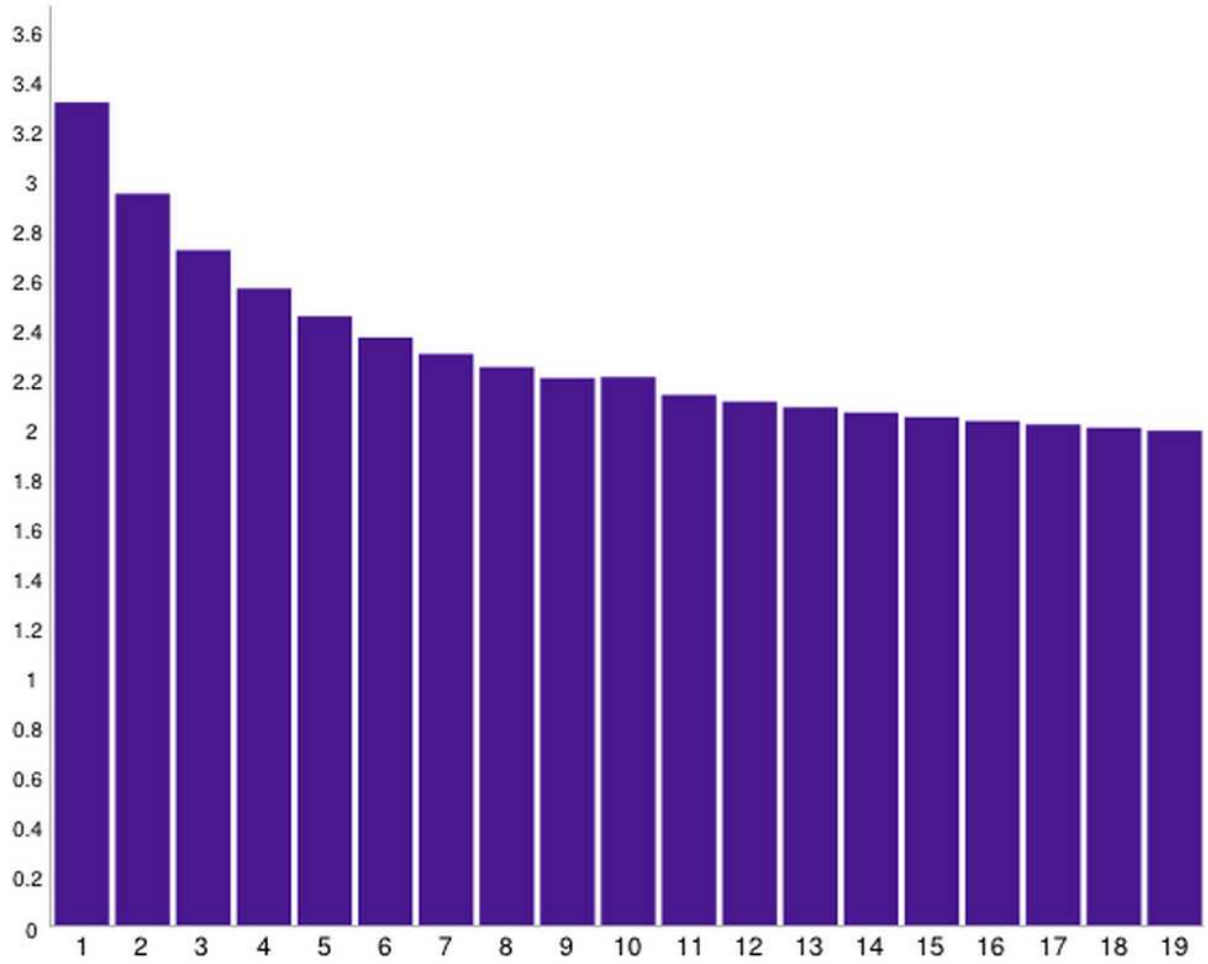
$523924 = 523 * 1000 + 924$

Key: Multiply, then add

$$
\begin{aligned}
523924 &= 5 * 100{,}000 \\
&+ 2 * 10{,}000 \\
&+ 3 * 1{,}000 \\
&+ 9 * 100 \\
&+ 2 * 10 \\
&+ 4
\end{aligned}
$$

Reducing dependencies

```
unsigned atoui(const char* b, const char* e) {
   static const unsigned pow10[20] = {
      10000000000000000000UL,
      ...
      1
   };
   enforce(b < e);
   unsigned result = 0;
   auto i = sizeof(pow10) / sizeof(*pow10) - (e - b);
   for (; b != e; ++b) {
      enforce(*b >= '0' && *b <= '9');
      result += pow10[i++] * (*b - '0');
   }
   return result;
}
```

associative
means
parallelizable

Strength reduction, again

```
unsigned atoui(const char* b, const char* e) {
   static const unsigned pow10[20] = {
      10000000000000000000UL,
      ...
      1
   };
   enforce(b < e);
   unsigned result = 0;
   auto i = sizeof(pow10) / sizeof(*pow10) - (e - b);
   for (; b != e; ++b) {
      enforce(*b >= '0' && *b <= '9');
      result += pow10[i++] * (*b - '0');
   }
   return result;
}
```

Strength reduction, again

```
// Reduce the strength of this
if (*b < '0' || *b > '9')
      throw range_error("ehm");
result += pow10[i++] * (*b - '0');
```


Tip: Use Bitwise Ops

```
if ((*b < '0') | (*b > '9'))
      throw range_error("ehm");
result += pow10[i++] * (*b - '0');
```

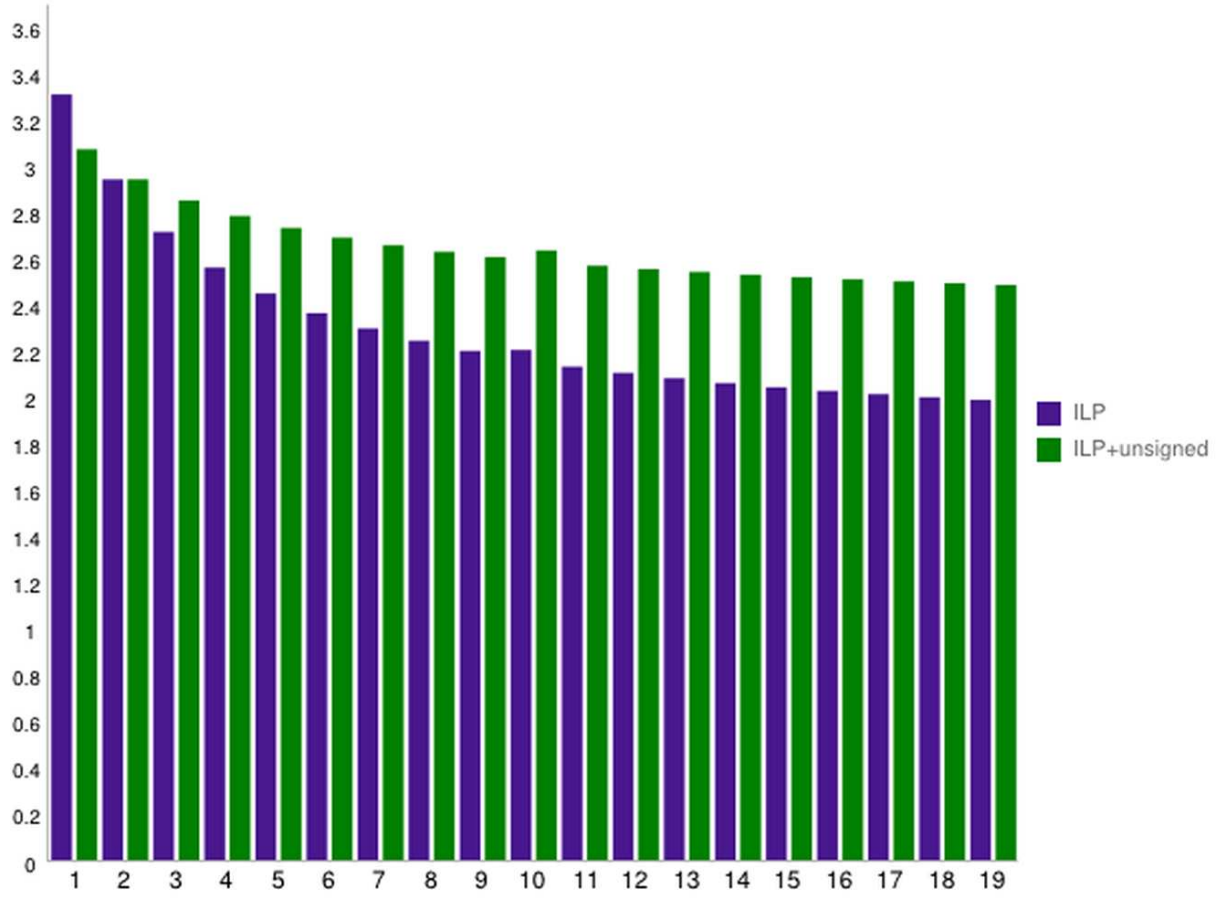
Tip: Use `isdigit`

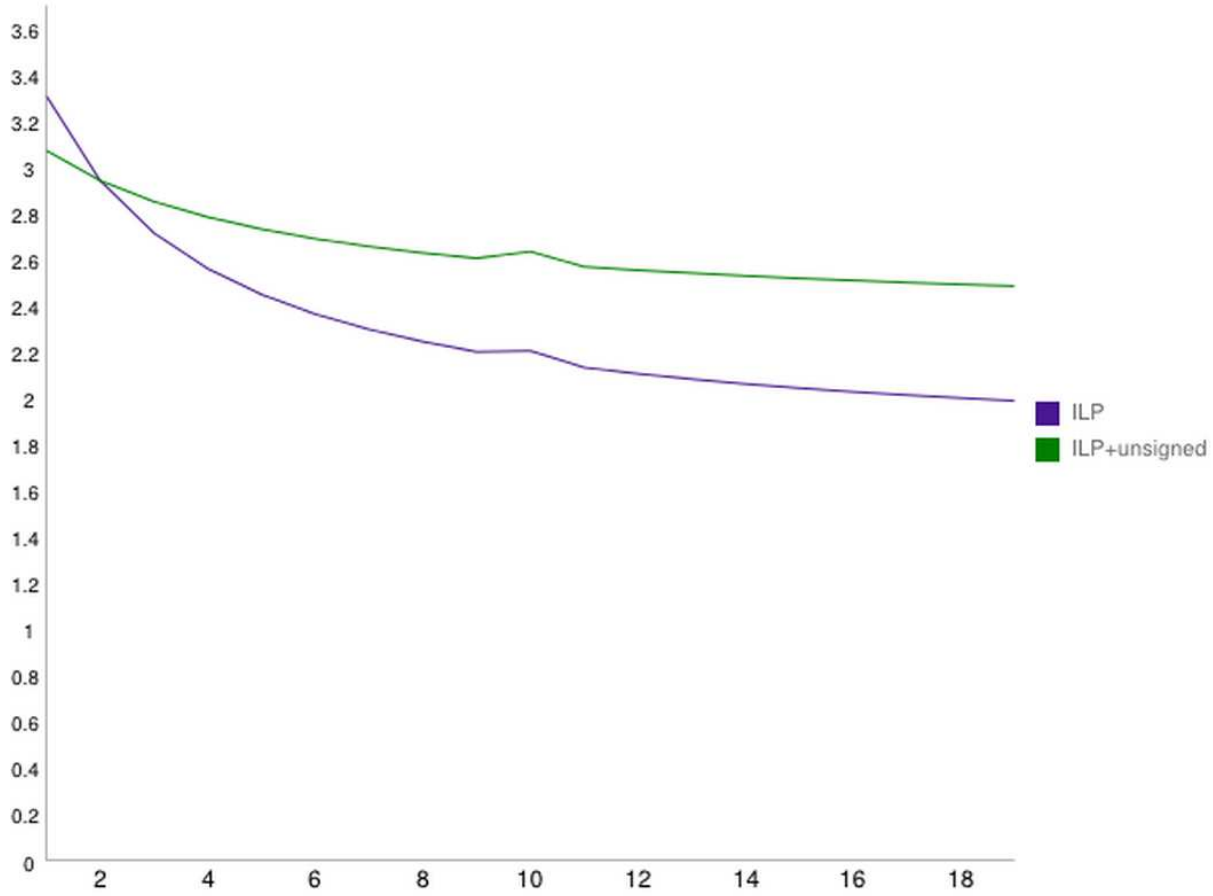
```
if (!isdigit(*b))
      throw range_error("ehm");
result += pow10[i++] * (*b - '0');
```

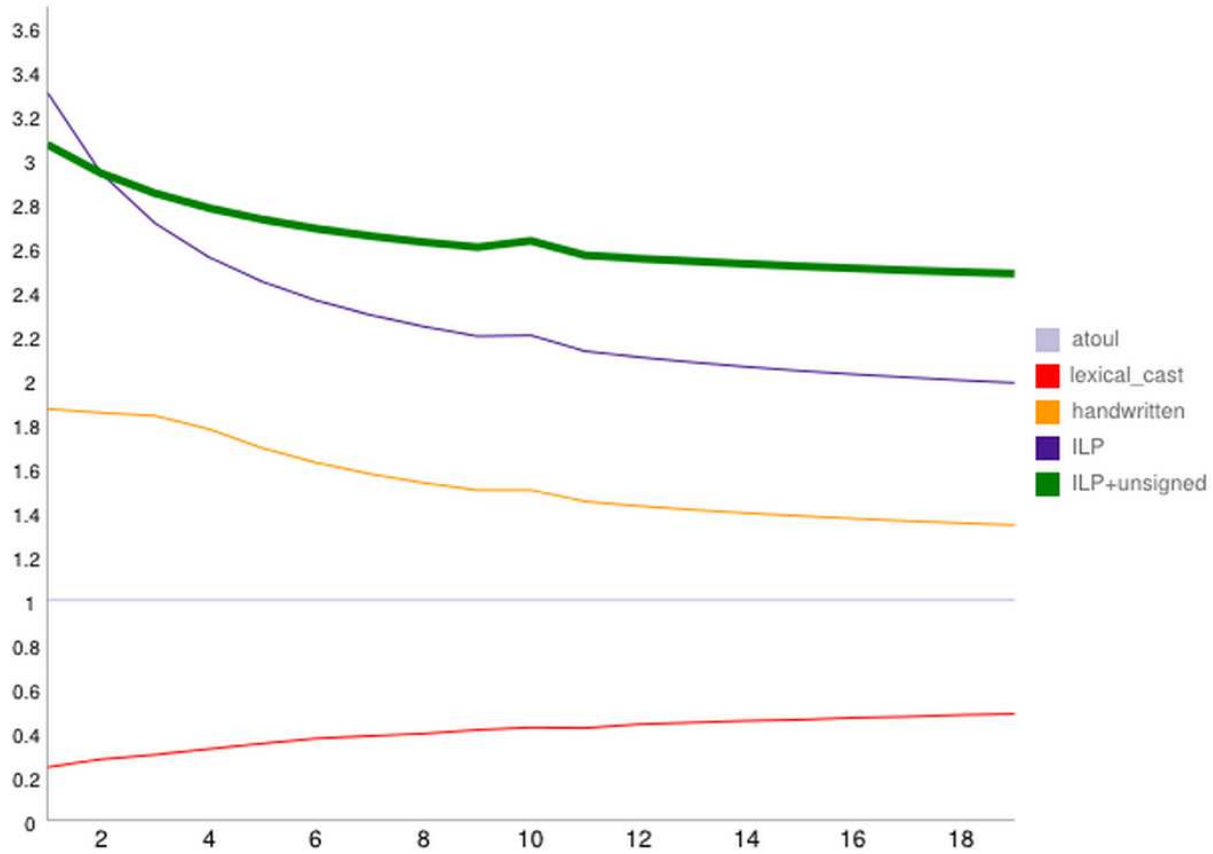

Pro Tip: Use unsigned wraparoo

```
auto d = unsigned(*b) - '0';
if (d >= 10)
      throw range_error("ehm");
result += pow10[i++] * d;
```

modulo subtraction
is
injective




3.6
3.4
3.2
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
2
4
6
8
10
12
14
16
18
ILP
ILP+unsigned


3.6
3.4
3.2
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
2
4
6
8
10
12
14
16
18
atoul
lexical_cast
handwritten
ILP
ILP+unsigned

Love/money makes the world go round

Math makes strength reduction go round

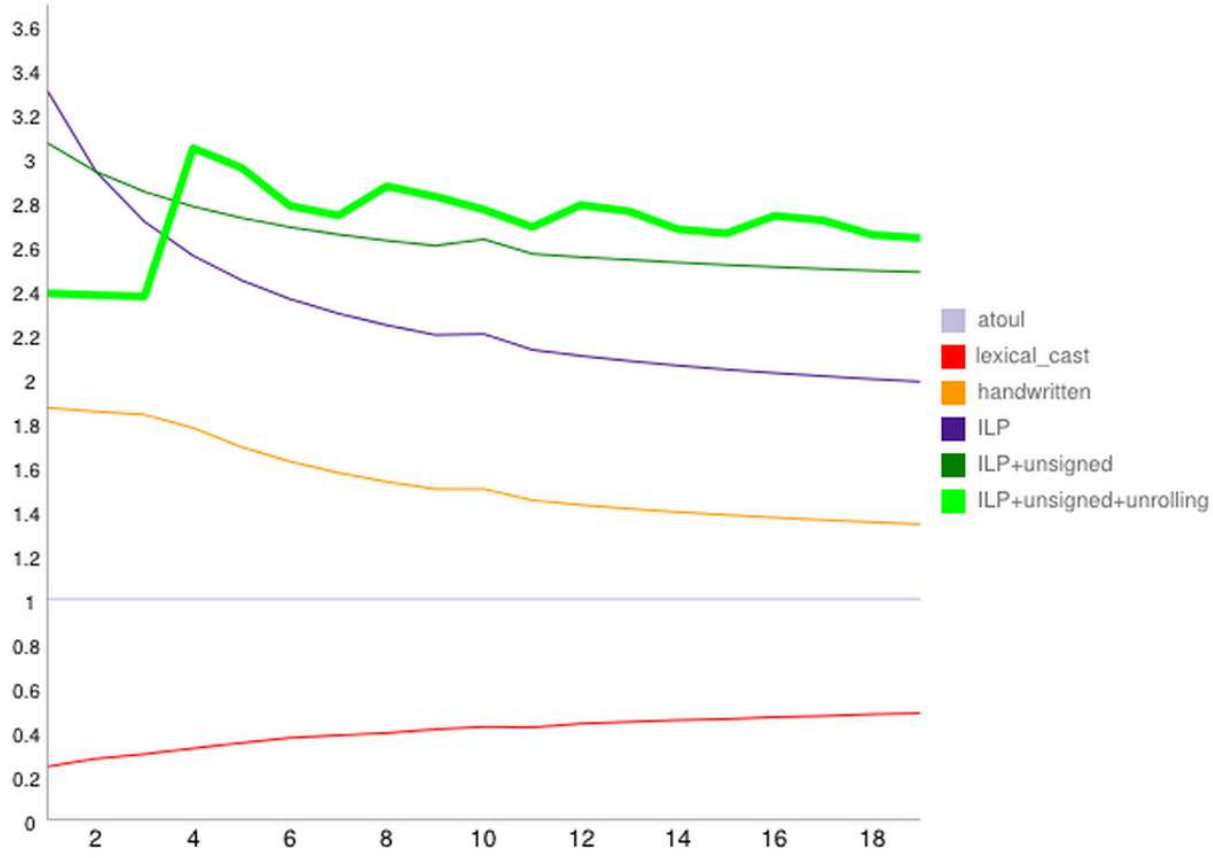
Think Outside the Loop

```
for (; b != e; ++b) {
   auto d = unsigned(*b) - '0';
   enforce(d < 10);
   result += pow10[i++] * d;
}
```

Think Outside the Loop

```
for (; e - b >= 4; b += 4, i += 4) {
      auto d0 = unsigned(*b) - '0';
      enforce(d0 < 10);
      auto s0 = pow10[i] * d0;
      auto d1 = unsigned(b[1]) - '0';
      enforce(d1 < 10);
      auto s1 = pow10[i + 1] * d1;
      auto d2 = unsigned(b[2]) - '0';
      enforce(d2 < 10);
      auto s2 = pow10[i + 2] * d2;
      auto d3 = unsigned(b[3]) - '0';
      enforce(d3 < 10);
      auto s3 = pow10[i + 3] * d3;

      result += s0 + s1 + s2 + s3;
}
```

Summary

- Measure
- Use good baselines
- Reduce strength
- Reduce dependencies
- Knowing math is knowing optimization

Q & A