Don’t bore your cores!

Anders Ahlgren
Mercur Solutions AB

mercur
Know your numbers

Hardware? We don’t need no stinkin’ hardware!

- Java ”protects” us from hardware details
- We usually use a simplified mental model of execution
- Most of the time, this is good
- Sometimes, it can mislead us, and hold us back

Unless stated otherwise, examples were measured on Intel Core i7 mobile ‘Skylake' 2.6 GHz / 2133 MHz memory, Oracle JVM 1.8, server 64-bit, compressed oops

By Source, Fair use,
https://en.wikipedia.org/w/index.php?curid=52517236

Test mental model #1: parallel stream speedup

```
long[] a;    // large array
...
long s = Arrays.stream(a).sum();
```

— *vs* —

```
long s = Arrays.stream(a).parallel().sum();
```

Test mental model #1: parallel stream speedup

```
long[] a;    // large array
...
long s = Arrays.stream(a).sum();
```

— *vs* —

```
long s = Arrays.stream(a).parallel().sum();
```

The speedup is limited by memory transfer speed

Core i7 (1-4-2): ≈ **1.9** times faster

40 vCPU Xeon (2-12-2): ≈ **5-8** times faster

(Xeon memory in performance mode)

Test mental model #2: ordering of operations

```
static int fun(int[] a) {
    int result = 1;
    for (int i = 0; i < a.length; i++)
        result = result * i + a[i]; // variation (a)
    return result;
}
```

How does performance change if loop body is replaced by:

```
        result = result + i * a[i]; // variation (b)
```

Test mental model #2: ordering of operations

```
static int fun(int[] a) {
    int result = 1;
    for (int i = 0; i < a.length; i++)
        result = result * i + a[i]; // variation (a)
    return result;
}
```

How does performance change if loop body is replaced by:

```
        result = result + i * a[i]; // variation (b)
```

(b) is faster than (a)

≈ 3 times faster

Test mental model #3: size of binary search

`Arrays.binarySearch(int[] a, int key)`

Assume many calls made, same `a`, different `key`-s, and compare performance of:

(a) Power of 2: `a.length` is 4,194,304 $= 2^{22}$

(b) Slightly larger: `a.length` is 4,198,399 $= 2^{22} + 4095$

Test mental model #3: size of binary search

```
Arrays.binarySearch(int[] a, int key)
```

Assume many calls made, same `a`, different `key`-s, and compare performance of:

(a) Power of 2: `a.length` is $4{,}194{,}304 = 2^{22}$

(b) Slightly larger: `a.length` is $4{,}198{,}399 = 2^{22} + 4095$

(b) is faster than (a)

$\approx$ 1.4 times faster

mercur
Know your numbers

Multi-core world, or monster-core world?

- Intel Xeon Broadwell-E5 (22 cores), 2016:

  7,200 M transistors (14 nm), or

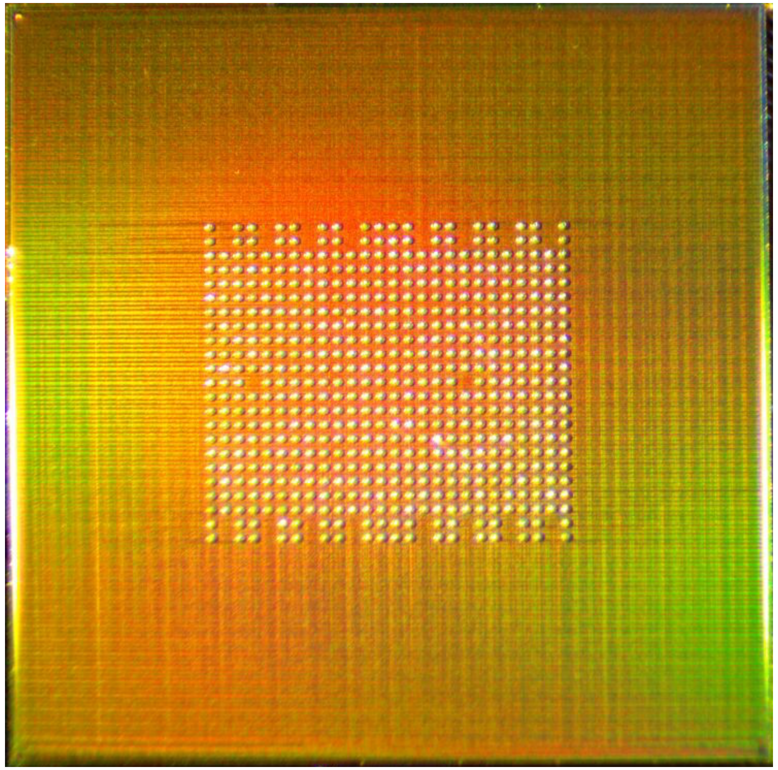
  327 M transistors / core

- UC Davis KiloCore (1,000 cores), 2016:

  621 M transistors (32 nm*), or

  0.6 M transistors / core

* For context, **32 nm** Intel Core was released January **2010**



KiloCore die
(Image source UC Davis)

Ordering of operations revisited

(a) `r = r * i + a[i];`

(b) `r = r + i * a[i];`

CPU can execute instructions in parallel

Can't start computation until inputs available (duh!)

JVM unrolls loops, body repeated 8 times (or 4, or…)

(a) basically multiply / add (≈ 4.3 cycles / iter if 8-way)

(b) unrolled body: load / multiply / add / drain pipeline
(≈ 1.4 cycles / iter if 8-way, ≈ 2.1 if 4-way)

Image by Patrick Bell from Haddonfield, NJ, USA - new 6-5-06 064,
CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=2187985

DIY unrolling: the ”before” picture

```java
// Taken from java.util.Arrays
public static int hashCode(int a[]) {
    if (a == null)
        return 0;

    int result = 1;
    for (int element : a)
        result = 31 * result + element;

    return result;
}
```

DIY unrolling: the ”before” picture

```
// Taken from java.util.Arrays
public static int hashCode(int a[]) {
    if (a == null)
        return 0;

    int result = 1;
    for (int element : a)
        result = 31 * result + element;

    return result;
}
```

JVM will optimize

`31 * r`

to

`(r << 5) - r`

Gain on x86 is moderate, ≈ **1.3** times

DIY unrolling: the expansion

$$r_8$$

$$31 \cdot r_7 + a_7$$

$$31 \cdot (31 \cdot r_6 + a_6) + a_7$$

$$31 \cdot (31 \cdot (31 \cdot r_5 + a_5) + a_6) + a_7$$

$$31 \cdot (31 \cdot (31 \cdot (31 \cdot r_4 + a_4) + a_5) + a_6) + a_7$$

$$31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot r_3 + a_3) + a_4) + a_5) + a_6) + a_7$$

$$31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot r_2 + a_2) + a_3) + a_4) + a_5) + a_6) + a_7$$

$$31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot r_1 + a_1) + a_2) + a_3) + a_4) + a_5) + a_6) + a_7$$

$$31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot (31 \cdot r_0 + a_0) + a_1) + a_2) + a_3) + a_4) + a_5) + a_6) + a_7$$

DIY unrolling: the ”after” picture

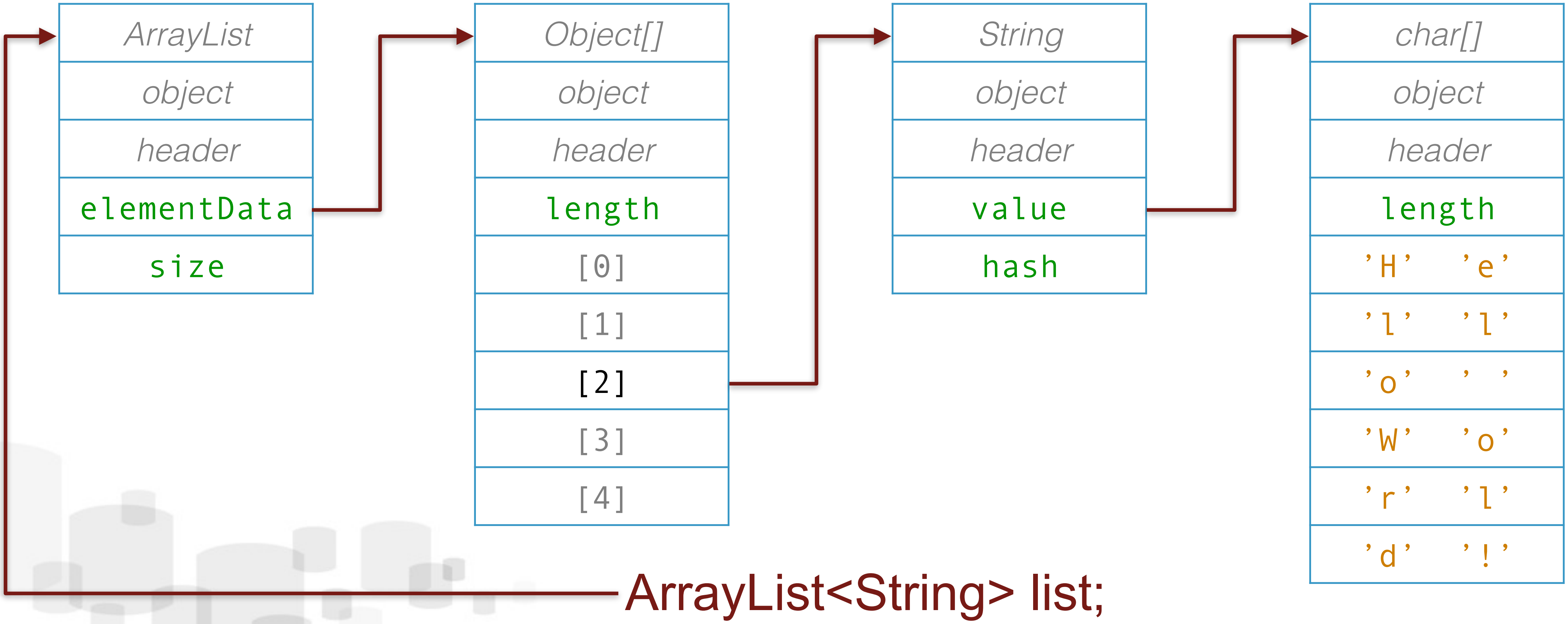
```
int result = 1;
for (int i = 0; i < a.length - 7; i += 8) {
    result = 0x94446F01 * result
           + 0x67E12CDF * a[i]   + 887503681 * a[i+1]
           +   28629151 * a[i+2] +    923521 * a[i+3]
           +      29791 * a[i+4] +       961 * a[i+5]
           +         31 * a[i+6] +             a[i+7];
}
for (int i = a.length & ~7; i < a.length; i++) {
    result = 31 * result + a[i];
}
return result;
```

DIY unrolling: the ”after” picture

```
int result = 1;
for (int i = 0; i < a.length - 7; i += 8) {
    result = 0x94446F01 * result
           + 0x67E12CDF * a[i]   + 887503681 * a[i+1]
           +   28629151 * a[i+2] +    923521 * a[i+3]
           +      29791 * a[i+4] +       961 * a[i+5]
           +         31 * a[i+6] +             a[i+7];
}
for (int i = a.length & ~7; i < a.length; i++) {
    result = 31 * result + a[i];
}
return result;
```

≈ 1.9 times faster

Chasing references



Location, Location, Location

```
String[] a = Files.lines(file.toPath())
                  .toArray(n -> new String[n]);
```

… *This is the part we are measuring:*

```
long c = Arrays.stream(a)
               .filter(s -> s.endsWith("?"))
               .count();
```

Location, Location, Location

```
String[] a = Files.lines(file.toPath())
                  .toArray(n -> new String[n]);
```

- No guarantees about location of objects
- In practice, starts in order: `a[0]`, `a[0].value`, `a[1]`, `a[1].value`, …
- GC fragments, here each consecutive run averages ≈15 strings

… *This is the part we are measuring:*

```
long c = Arrays.stream(a)
               .filter(s -> s.endsWith("?"))
               .count();
```

Location, Location, Location

```
String[] a = Files.lines(file.toPath())
                  .toArray(n -> new String[n]);
```

- No guarantees about location of objects
- In practice, starts in order: `a[0]`, `a[0].value`, `a[1]`, `a[1].value`, …
- GC fragments, here each consecutive run averages ≈15 strings

… *This is the part we are measuring:*

```
long c = Arrays.stream(a)
               .filter(s -> s.endsWith("?"))
               .count();
```

Fragmentation from GC is moderate, performance is OK

Location, Location, Location

```
String[] a = Files.lines(file.toPath())
                  .toArray(n -> new String[n]);


Arrays.sort(a);  // Seems innocent enough?
```

… *This is the part we are measuring:*

```
long c = Arrays.stream(a)
               .filter(s -> s.endsWith("?"))
               .count();
```

Location, Location, Location

```
String[] a = Files.lines(file.toPath())
                  .toArray(n -> new String[n]);


Arrays.sort(a);  // Seems innocent enough?
```

… *This is the part we are measuring:*

```
long c = Arrays.stream(a)
               .filter(s -> s.endsWith("?"))
               .count();
```

≈ 4 times slower

mercur
Know your numbers

Location, Location, Location

```
String[] a = Files.lines(file.toPath())
                  .toArray(n -> new String[n]);
for (int i = 0; i < a.length; i++)
   a[i] = new String(a[i]);  // Ewww!
Arrays.sort(a);
```

… *This is the part we are measuring:*

```
long c = Arrays.stream(a)
               .filter(s -> s.endsWith("?"))
               .count();
```

Location, Location, Location

```
String[] a = Files.lines(file.toPath())
                  .toArray(n -> new String[n]);
for (int i = 0; i < a.length; i++)
   a[i] = new String(a[i]);  // Ewww!
Arrays.sort(a);
```

… *This is the part we are measuring:*

```
long c = Arrays.stream(a)
               .filter(s -> s.endsWith("?"))
               .count();
```

≈ 7 times slower

Experimenting with locations: Chasing `int`

- Want to investigate location effects details
- Clearly, quite tricky using object references…
- Instead, array of `int` (assuming compressed oops; otherwise `long`)

```
static int chaseInts(int[] a) {
    int t = 0;
    for (int i = 0; i < N; i++)
        t = a[t];
    return t; // beware dead code in micro benchmarks
}
```

You won't *believe* how slow memory can be

Running `chaseInts` on a large array with random "jumping around":

1,000,000,000 iterations ≈ 130 s

≈ 130 ns; ≈ 340 cycles per iteration (test system is 2.6 GHz)

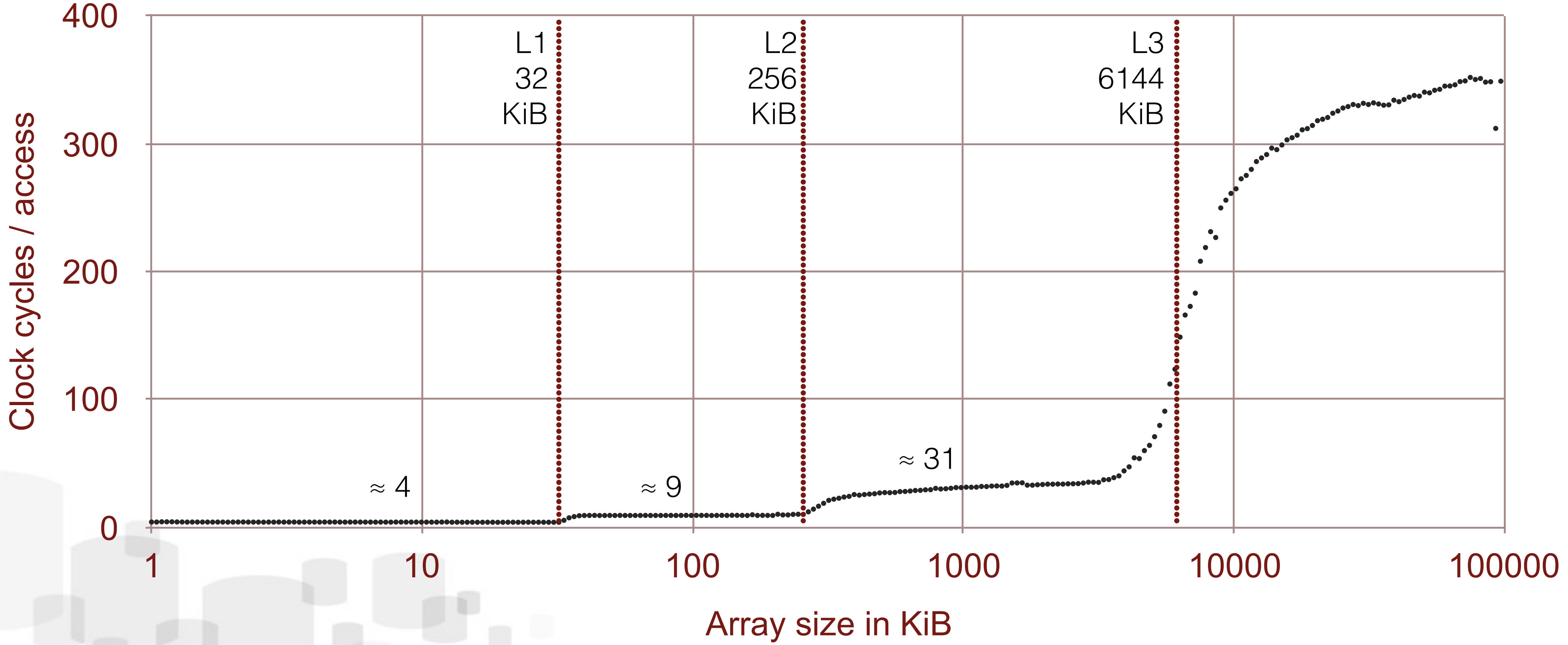
Each iteration reads 4 bytes from memory

4 B · 1,000,000,000 / (130 s) ≈ **30 MB/s**

***20 times** slower than SATA 3 SSD…!*

Reading SSD *sequentially*, so unfair comparison

That’s where the caches come in…



Cache lines, pre-fetching, and dependencies

Caches handle data in blocks (64 byte on x86), called *cache lines*

⇒ cheaper to access several locations in the same cache line

Sequential access makes processor *pre-fetch* next cache line into L1

(unless crossing memory page boundary)

The `chaseInts` method is bounded by L1 latency (4 cycles), because the loads are data dependent — can't be done in parallel

Compare `Arrays.stream(a).sum()` which is ≈ 4 times faster

TMI…?

”64 byte cache lines”
”32KiB L1 data cache”

- Should we really hard-wire assumptions about size of cache lines, and sizes of the different levels of caches into Java code…?!
- Computer science answer: The class of cache-oblivious algorithms will work regardless of details about the caches
- Practical answer: Getting the sizes wrong is much better than ignoring the existence of caches

So — how does all this change how we code?

- Some techniques are generally applicable good practice:
  - Move fields to class that actually use them
  - Don't design data structure first and adapt code to fit them; instead evolve both iteratively to harmonize with each other
- Don't break things down in "systematical" order, but rather in frequency order — focus on the *happy-path*
  - The real power of this comes from synergy effects

The bad news: no big gain without big pain

Serious cache-optimization on JVM require* low-level, time-consuming, and perhaps ugly, rewrites.

* There are ObjectLayout project (and of course Project Valhalla)

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that ***critical 3%.***

Donald E. Knuth

Don’t *have* a critical 3%? More of 80-20 rule, perhaps?
That might be fixable by designing differently

mercur
Know your numbers

Going primitive

```
class PointArr { // JVM supported value types, anyone?
    private final int[] xs;
    private final int[] xs; // or interleave x-y-x-y
    …                      // or pack hi-lo into longs
    int getX(int index) { … }
    int getY(int index) { … }
}
```

- Not *that* painful. Really. Remember, just in the 3%.

Sort? Why would you want to sort?
What, you want strings too? No-one uses strings anymore!

The back-and-forth compromise

- Translate back-and-forth between arrays-of-primitives and array-of-object!
- Could be "original" object (like `Point`), or represented by an inner class:

```
private class Pointy {
      final int index; … // hashCode, equals, …
```

- Most important use cases work directly on the arrays of primitives, less important can take the detour (in particular sorting and hashing)
- Bored processor has computational resources to spare
- Arrays-of-primitive pre-fetches, array-of-object are done in blocks fitting cache
  - Even "Location, Location, Location" example only $\approx$ 1.4 times slower

mercur
Know your numbers

Bit-twiddling for fun and profit

- Algorithms depending on tricky bit-twiddling are often fast (and a lot of fun!) but not all that common
- Simple, light-weight packing of data is nearly always a gain, and very frequently applicable
  - Often, you can fit quite a lot into a handful of `long`
- `BitSet` (and similar) are sadly under-utilized — make a habit of looking for new use-cases for them

Complexity cache clues

Linear: exploit pre-fetching (duh!)

Super-linear: pre-fetching + splitting data in cache-sized blocks (explicitly, or automagically through divide-and-conquer)

Merge sort and quicksort good, heapsort bad

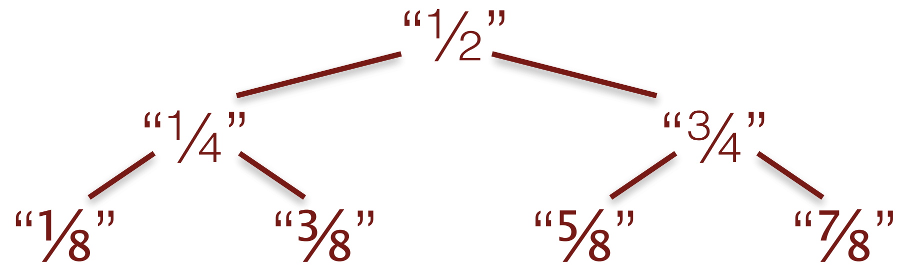
Sub-linear: exploit cache lines

Contrast B-trees with binary trees

Linear probing and K-V-K-V layout for hashing

Example: Binary search and cache lines

- Binary search works on an array, but treats it like a tree
  - Depth 0: nodes at:
  - Depth 1: nodes at:
  - Depth 2: nodes at:



- “Top of the tree” is small, so several levels fit into cache
  - But if $n = 2^{m+k}$, top k level indices share low m bits ⇒ cache conflicts
  - Only a single element is used in each cache line
- “Bottom of the tree” is consecutive locations; good for cache lines

mercur
Know your numbers

An even more cache-friendly alternative

- Imagine `Array.binarySearch(int[] a, int key)` is critical, on an array that doesn't fit cache (a bit far fetched; it's just an example…)
- Introduce a new class wrapping the array:

```
IntSearcher searcher = new IntSearcher(a);
…
int ip = searcher.binarySearch(key);
```

- IntSearcher adds second array encoding a 16-way tree
  - Size roughly `a.length/16 + a.length/256 + a.length/4096 +…`

mercur
Know your numbers

An even more cache-friendly alternative

- Imagine `Array.binarySearch(int[] a, int key)` is critical, on an array that doesn't fit cache (a bit far fetched; it's just an example…)
- Introduce a new class wrapping the array:

```
IntSearcher searcher = new IntSearcher(a);
…
int ip = searcher.binarySearch(key);
```

- IntSearcher adds second array encoding a 16-way tree
  - Size roughly `a.length/16 + a.length/256 + a.length/4096 +…`

≈ 1.5 times faster

The “string free zone”

- In bordered-off region, represent (relevant) strings with `int`
  - Say, content of specific column in a database shard
- When (relevant) string enters a region, it is translated to `int`
  - If `compareTo` is important, translation may reflect ordering
- When (relevant) string exits the region, translate back
- **Advantages:** cache locality, footprint, fast `compareTo`,…
- Needs few strings enter, especially (in ordered cases) those not already represented in the region (may invalidate mapping)

Typical optimizing cycle

1. Profile your program on important use case
2. Find the largest bottleneck, and address it
3. Repeat until you are happy, or optimization budget spent, or diminishing returns hurts too much

- Great for finding bugs, like accidental n^2 behavior
- Hill-climbing — gets you higher, but often miss the top
- 90-10 or 80-20 rule ⇒ quickly diminishing returns

“Distilling” code before normal optimize

1. Describe the absolute core of what needs to be done in a simple way, omit annoying parts
2. Can you do anything to make reality closer to description? Try to move (not necessarily remove) stuff from the core code
    - Pre- and post-processing
    - Transformation to other data structures
3. Re-apply happy path and data/code harmony ideas
4. Repeat and refine

May get you from Pareto’s 80-20 to Knuth’s “critical 3%”

Closing words

- Understanding modern hardware is the first step
- How much effort should *you* spend on performance?
- Which techniques applies to *your* problem?
    - You *will* have to adapt them to your situation
    - You *may* need to invent new ones
- If it were easy, *we* would be the ones bored, right?

Any questions?

?