CPU Caches: Trust but Verify

Tommi Jalkanen - Kodan

Rust Finland

- First meetup in February 3, 2017
- 15 meetups organized, 277 members on meetup.com and 64 on matrix
- Chat with others: #rust-finland:matrix.org
- Website: https://www.rust-finland.org
- Github organization: https://github.com/rust-finland

Motivation

- Sharing knowledge
- Bad takes on the Internet
- Time + money + environment

Bottleneck for Computing

Bottleneck for Computing

- Speed of light

Bottleneck for Computing

- Speed of light
- 2 GHz CPU can do ~2 instructions/ns

Bottleneck for Computing

- Speed of light
- 2 GHz CPU can do ~2 instructions/ns
- Light travels ~0.3 m/ns

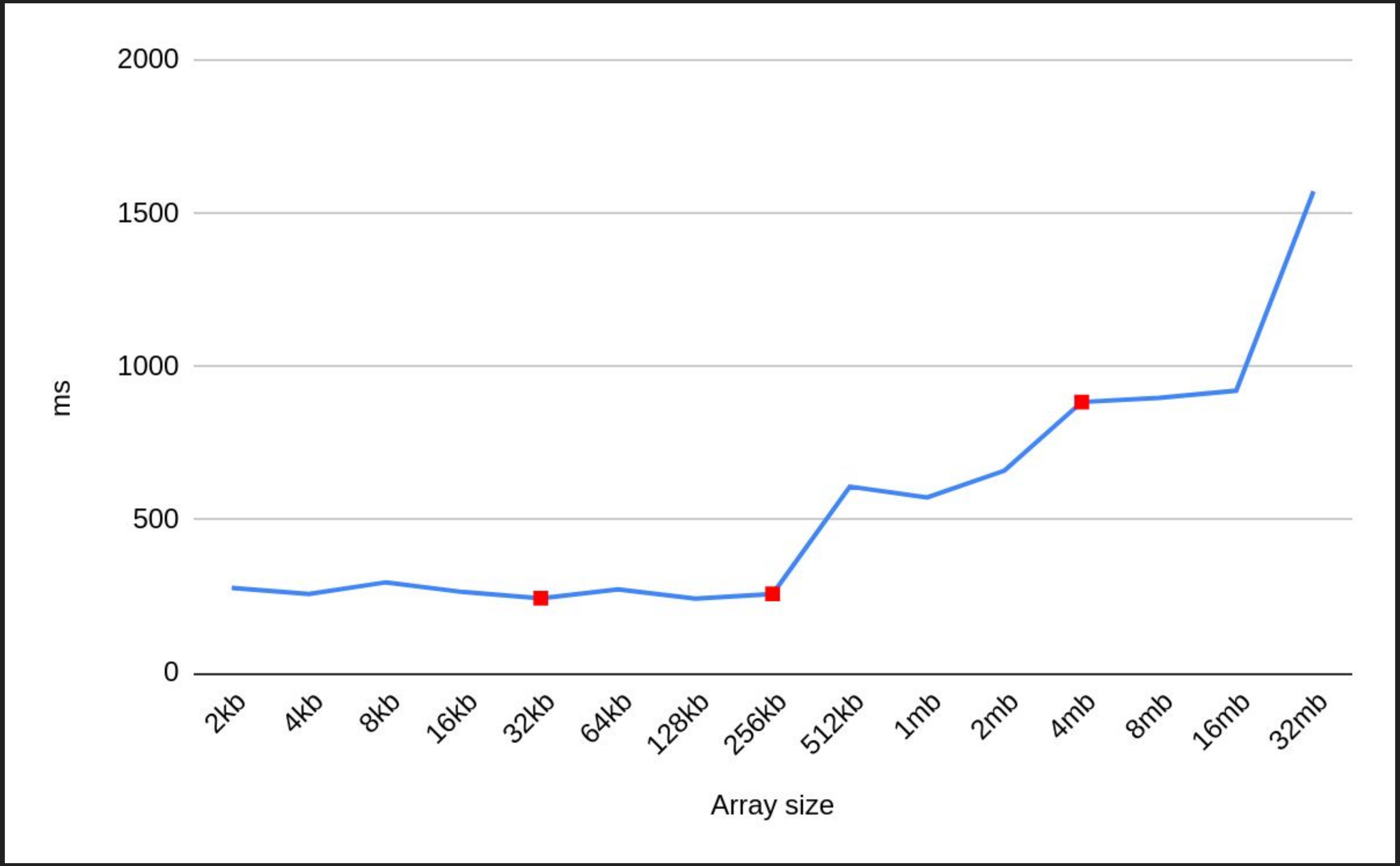
Caches on my Thinkpad

```
> getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  4194304
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Verifying Specs

```
let steps = 256 * 1024 * 2048;
let length_mod = arr.len() - 1;
for i in 0..steps {
    arr[(i * 16) & length_mod] += 1;
}
```

Verifying Specs



Step Sizes Example

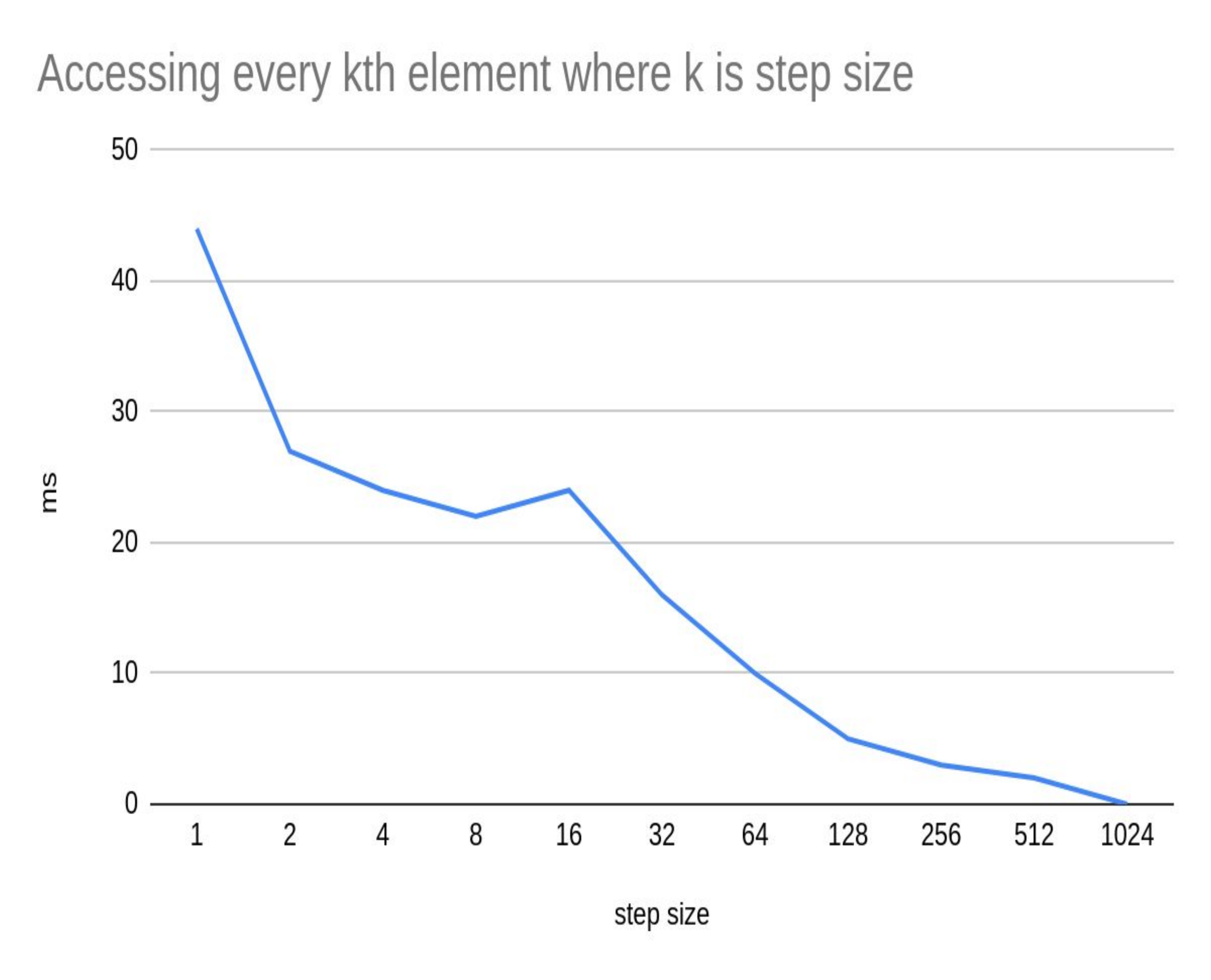
Accessing every element

```
for i in 0..arr.len() {
    arr[i] *= 3;
}
```

Accessing every 16th element

```
for i in (0..arr.len()).step_by(16) {
    arr[i] *= 3;
}
```

Cachelines



Accessing every kth element where k is step size
50
40
30
20
10
0
ms
1
2
4
8
16
32
64
128
256
512
1024
step size

Rows vs Columns

```
for i in 0..A[0].len() {
    for j in 0..A.len() {
        count += A[i][j];
    }
}
```

VS

```
for i in 0..A[0].len() {
    for j in 0..A.len() {
        count += A[j][i];
    }
}
```

Rows vs Columns

```
for i in 0..A[0].len() {
    for j in 0..A.len() {
        count += A[i][j];
    }
}
```

VS

```
for i in 0..A[0].len() {
    for j in 0..A.len() {
        count += A[j][i];
    }
}
```

```
Benchmark #1: ./target/release/row
  Time (mean ± σ):      284.5 ms ±  25.2 ms    [User: 133.0 ms, System: 150.4 ms]
  Range (min … max):   263.7 ms … 327.4 ms    10 runs

Benchmark #1: ./target/release/column
  Time (mean ± σ):      925.8 ms ±  62.3 ms    [User: 795.6 ms, System: 128.5 ms]
  Range (min … max):   872.5 ms … 1019.6 ms    10 runs
```

More examples

```
for _i in 0..steps {
        arr[0] += 1;
        arr[0] += 1;
}
```

vs

```
for _i in 0..steps {
    arr[0] += 1;
    arr[1] += 1;
}
```

Parallel Instructions

```
for _i in 0..steps {
        arr[0] += 1;
        arr[0] += 1;
}
```

VS

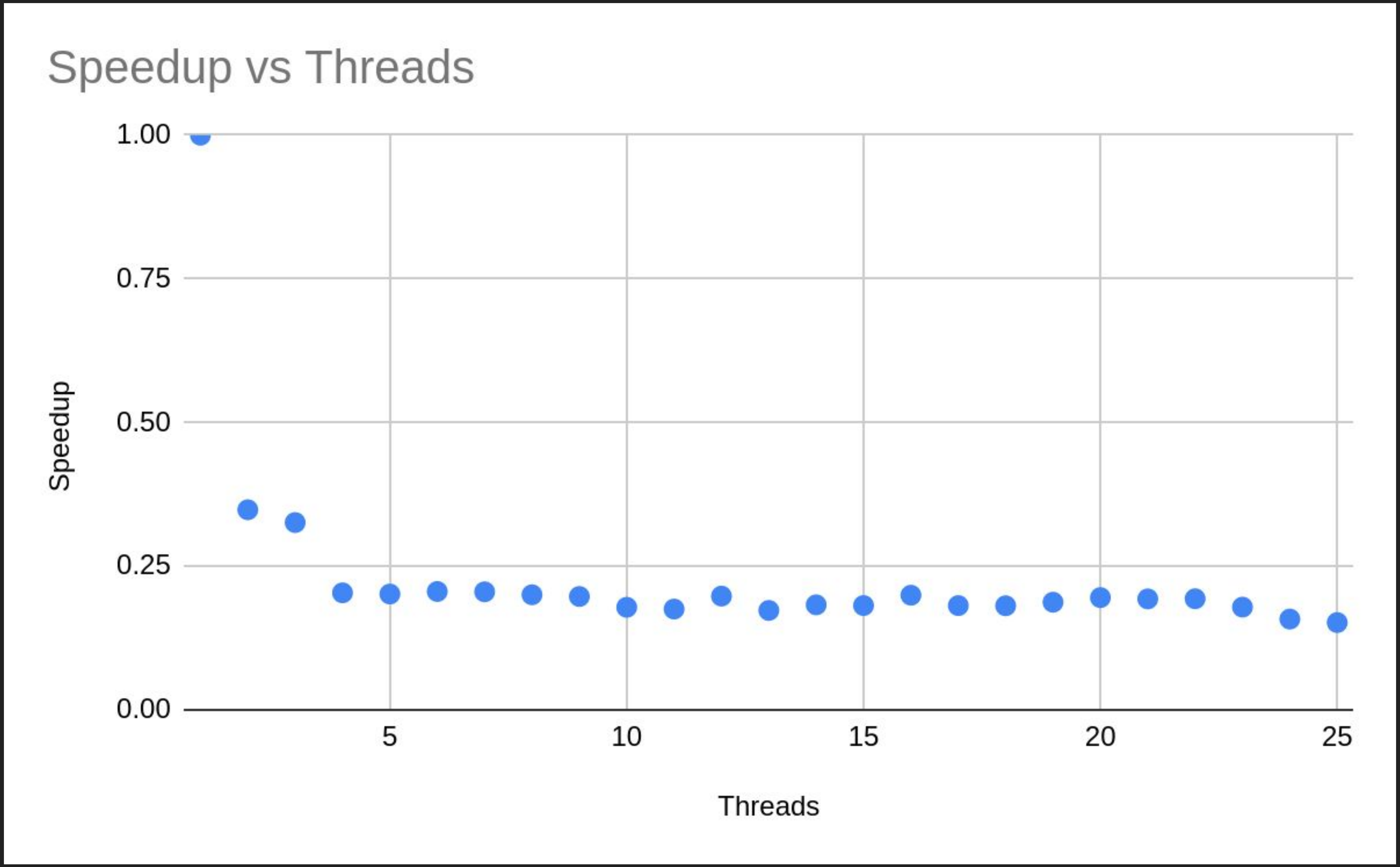
```
for _i in 0..steps {
    arr[0] += 1;
    arr[1] += 1;
}
```

Code snippet on the right is ~2x faster

Yet Another Example

```
let threads: Vec<_> = (0..*P)
    .collect::<Vec<usize>>()
    .iter()
    .map(|&p| {
        let result_ = Arc::clone(&result);
        let P_ = Arc::clone(&P);
        thread::spawn(move || {
            let chunk_size = DIM / *P_ + 1;
            let my_start = p * chunk_size;
            let my_end = std::cmp::min(my_start + chunk_size, DIM);
            for i in my_start..my_end {
                for j in 0..DIM {
                    if MATRIX[i][j] % 2 != 0 {
                        let mut result__ = result_.lock().unwrap();
                        result__[p] += 1;
                        drop(result__);
                    }
                }
            }
        })
   })
   .collect();
for handle in threads {
    handle.join().unwrap();
}
```

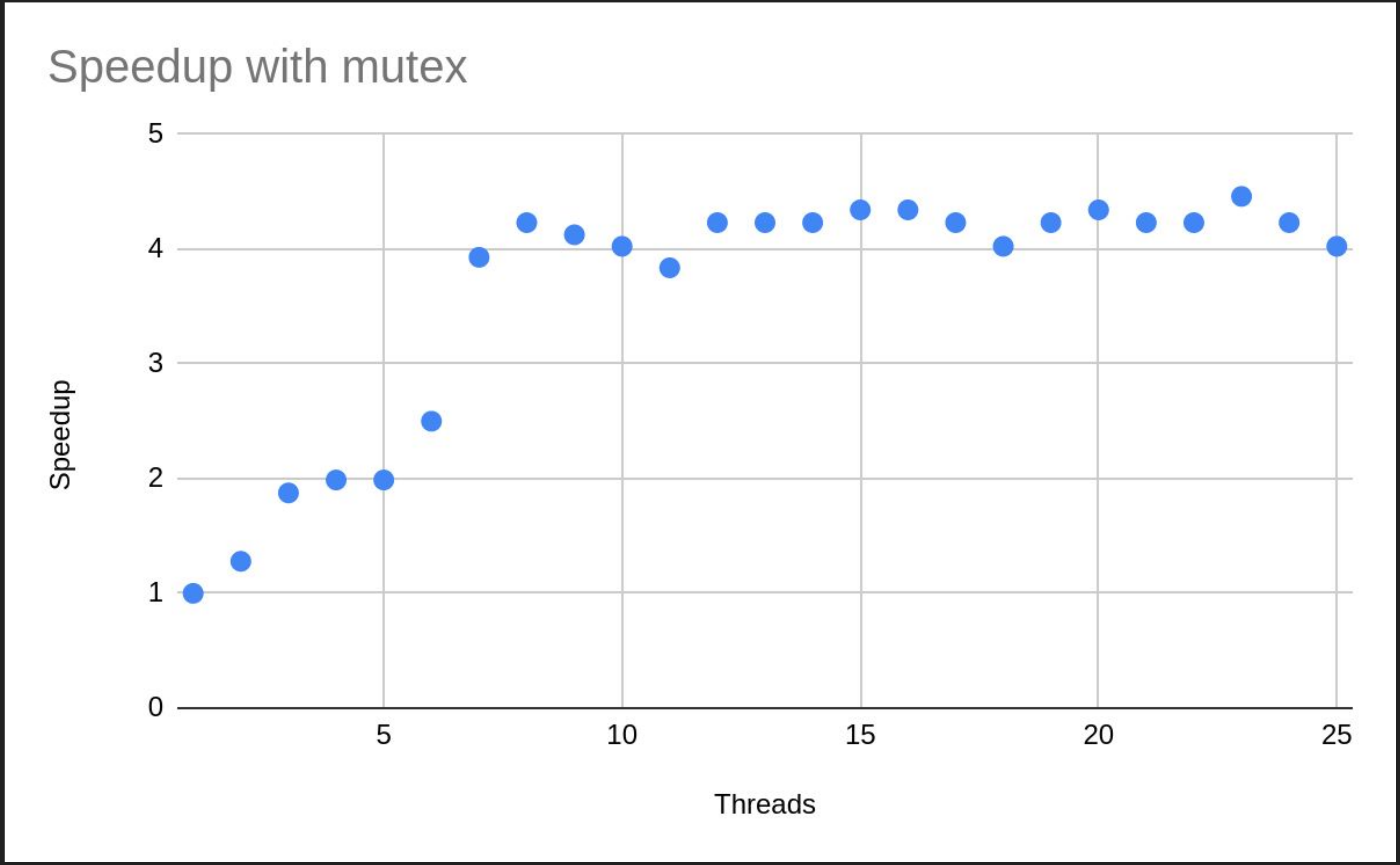
Yet Another Example



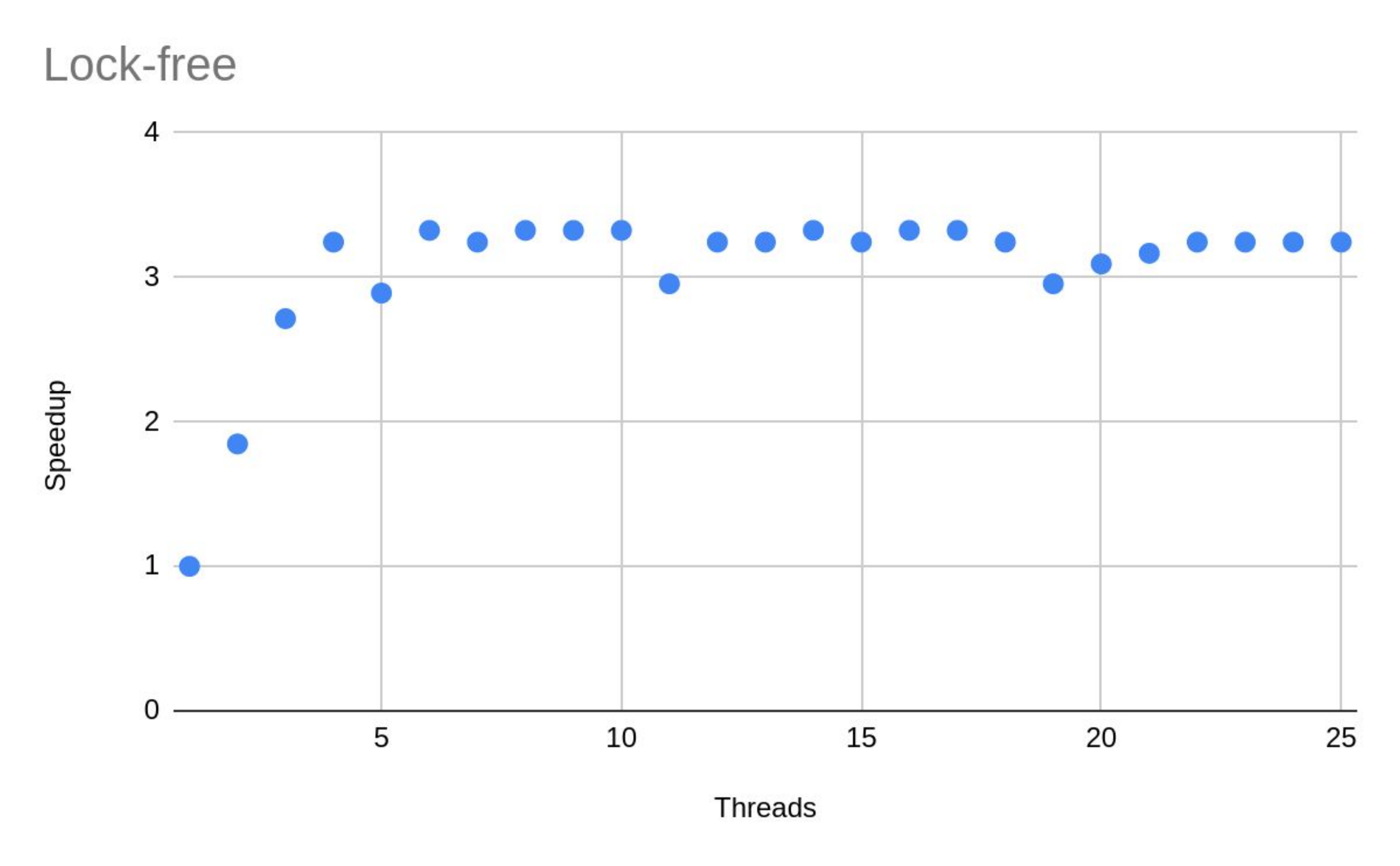
Yet Another Example

```
let threads: Vec<_> = (0..*P)
    .collect::<Vec<usize>>()
    .iter()
    .map(|&p| {
        let result_ = Arc::clone(&result);
        let P_ = Arc::clone(&P);
        thread::spawn(move || {
            let mut count = 0;
            let chunk_size = DIM / *P_ + 1;
            let my_start = p * chunk_size;
            let my_end = std::cmp::min(my_start + chunk_size, DIM);
            for i in my_start..my_end {
                for j in 0..DIM {
                    if MATRIX[i][j] % 2 != 0 {
                        count += 1;
                    }
                }
            }
            let mut result__ = result_.lock().unwrap();
            result__[p] = count;
            drop(result__);
       })
    })
    .collect();
for handle in threads {
    handle.join().unwrap();
}
```

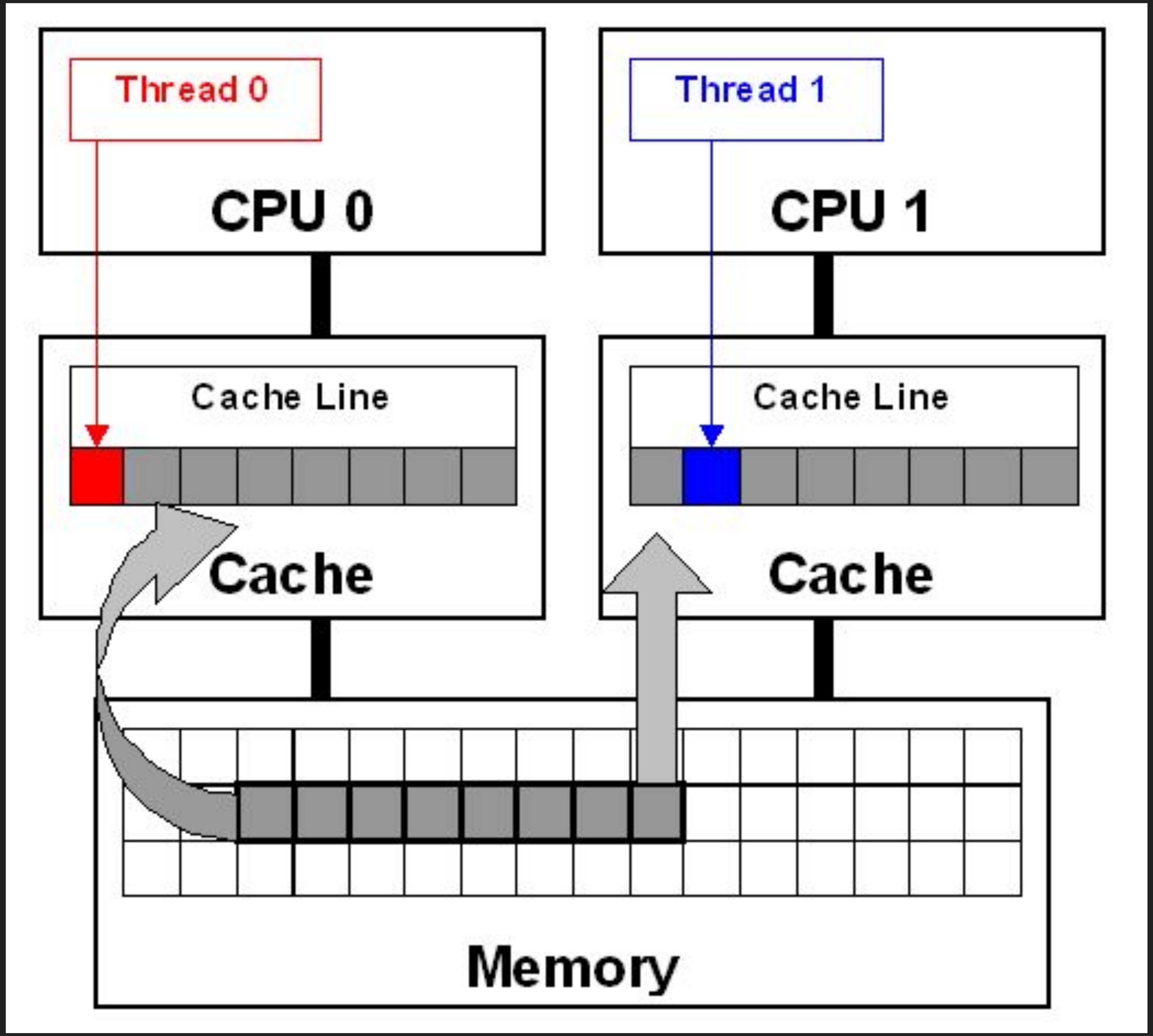
Yet Another Example



Yet Another Example



False Sharing



https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html

Cache Associativity (L1)

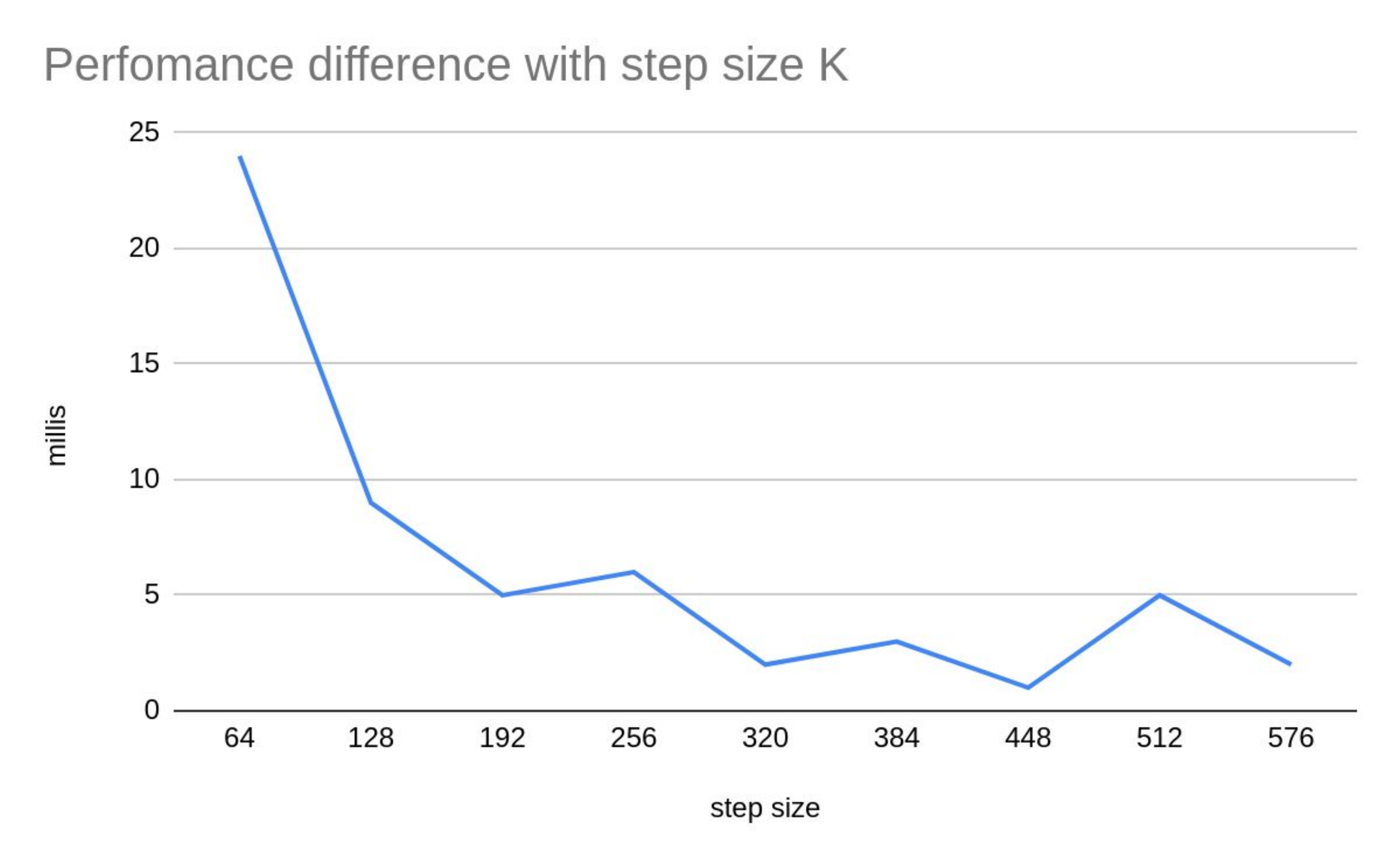
- Number of slots
  - L1: 512

- 64-byte chunks partioned into sets
  - lowest 6 bits determine to which of the 64 sets the cacheline belongs to
  - cache can hold at most 8 lines per any given set

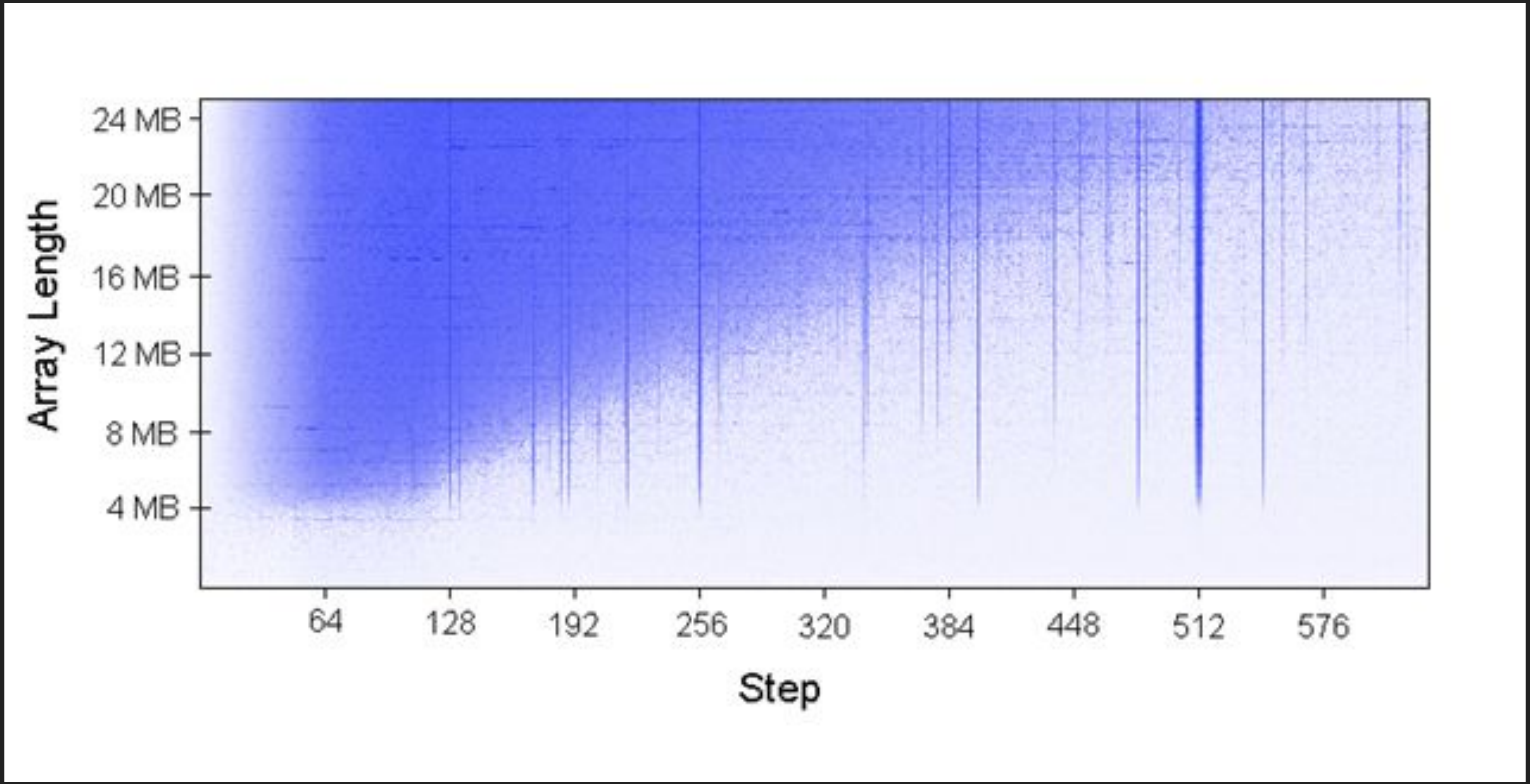
Cache Associativity

```
let rep = 1024 * 1024;
let mut p = 0;
for _i in 0..rep {
    arr[p] += 1;
    p += K;
    if p >= arr.len() {
        p = 0;
    }
}
```

Cache Associativity 8mb Array



Cache Associativity



http://igoro.com/archive/gallery-of-processor-cache-effects/

Conclusions

- **Performance behaviour can feel counterintuitive**

Conclusions

- **Performance behaviour can feel counterintuitive**
- **Trust the computer but verify**

Conclusions

- **Performance behaviour can feel counterintuitive**
- **Trust the computer but verify**
- **Devs can help computers do a better job**

Sources

- https://akkadia.org/drepper/cpumemory.pdf
- http://igoro.com/archive/gallery-of-processor-cache-effects/
- Scott Meyers: Cpu Caches and Why You Care: https://www.youtube.com/watch?v=WDIkqP4JbkE
- https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html

Questions?

Tommi Jalkanen
github: koura
twitter: @om_nommy