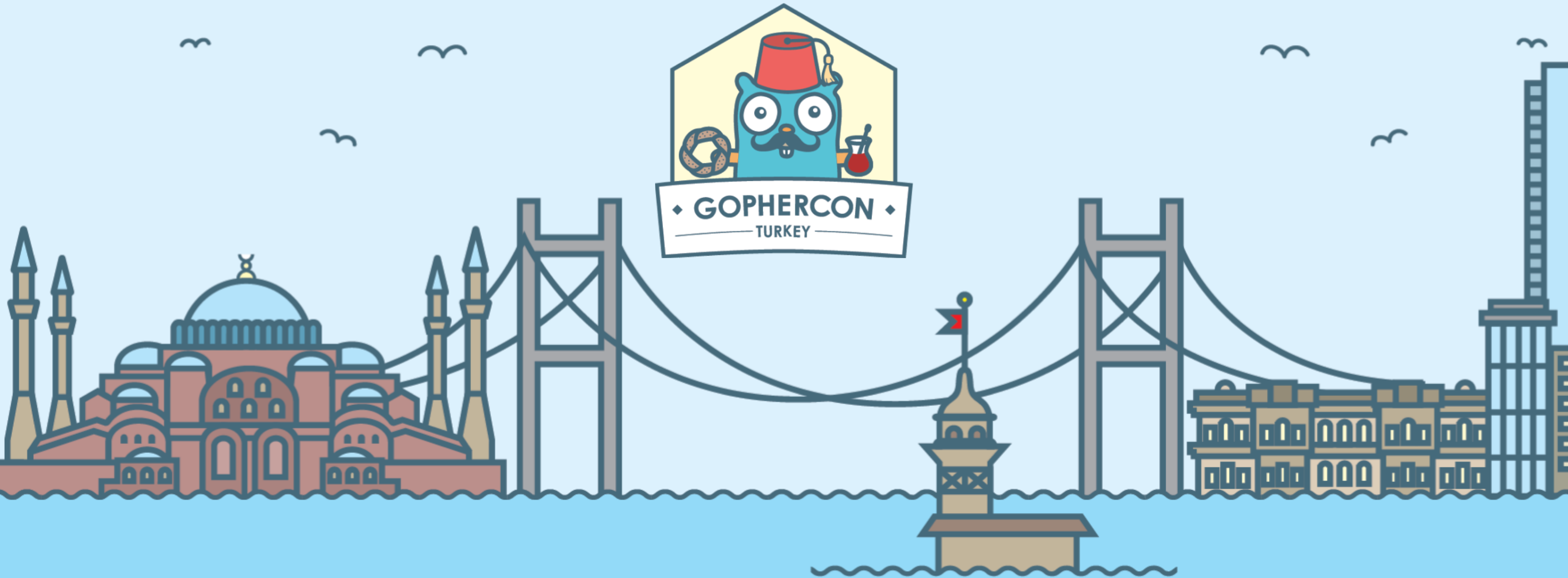


# Mechanical Sympathy in Go





You don't have to be an engineer to be a racing driver,  
but you do have to have **mechanical sympathy**

Sir Jackie Stewart - three-time F1 world champion



# Mechanical Sympathy Applied to IT

- Concept applied to software by Martin Thompson
- As developers, we **don't need** to be hardware engineers
- Yet, having an understanding of how does a machine work can make us a **better developer** (algorithms, data structures)
  
- Today: How to be a **better Go developer** by understanding how CPUs are working





Teiva Harsanyi  
 teivah

Software Engineer - Beat





HIRING

# SOFTWARE ENGINEERS

ON CUTTING-EDGE TECHNOLOGIES

DISCOVER YOUR NEXT CHALLENGE  
IN AMSTERDAM, ATHENS OR REMOTE

**BEAT**



beat.careers



# CPU Architecture

Locality of Reference

Data-Oriented Design

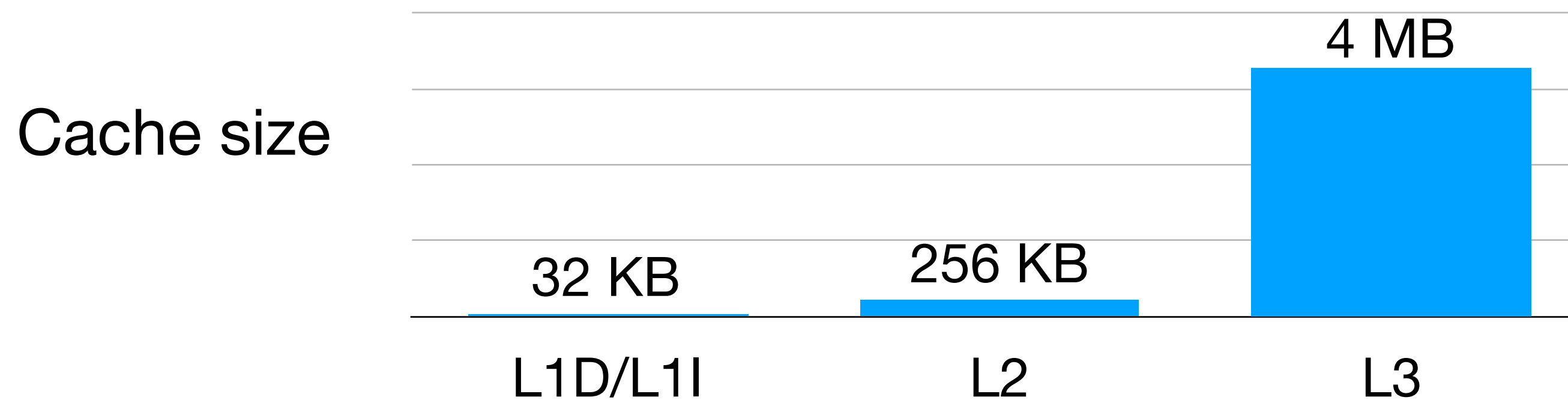
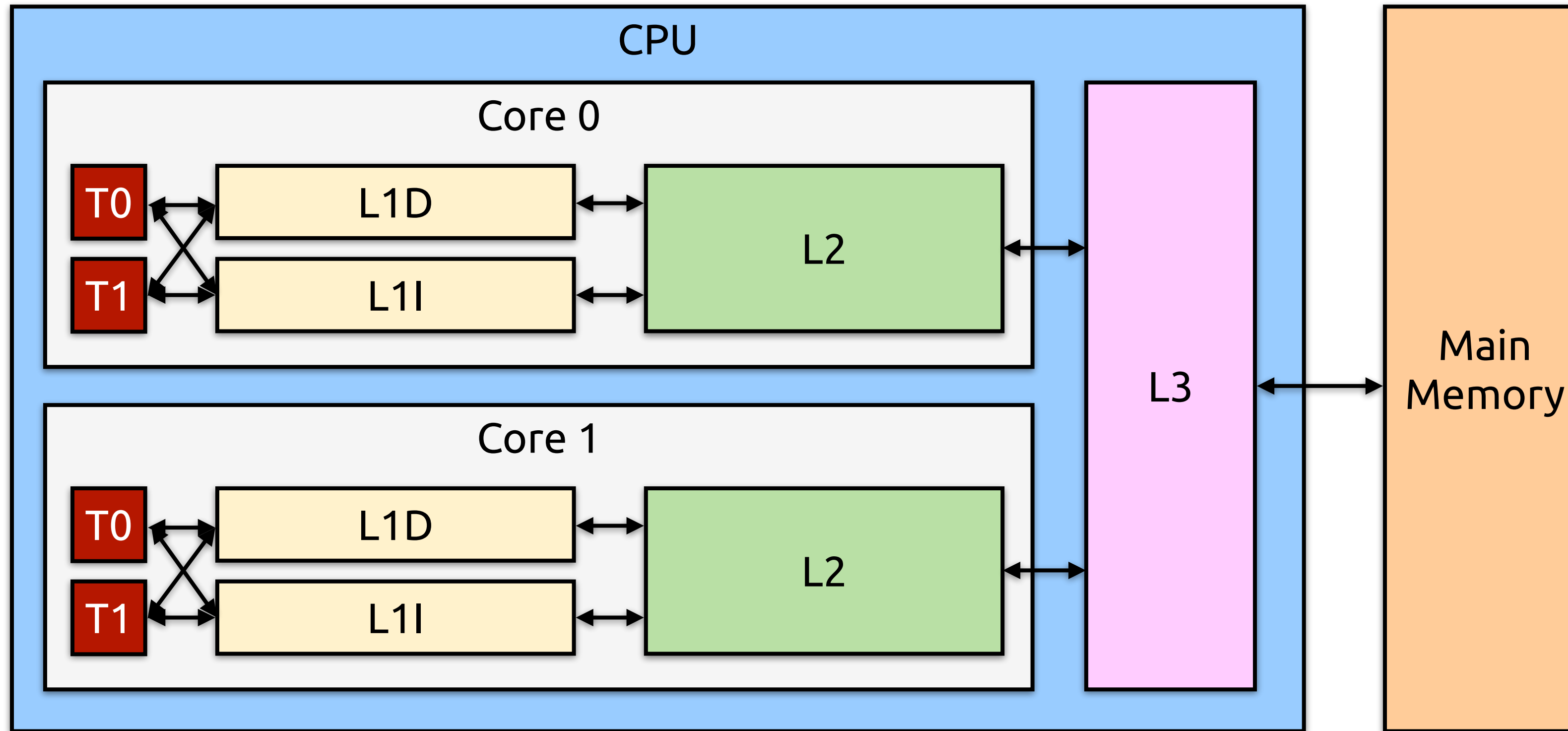
Caching Pitfall

Concurrency

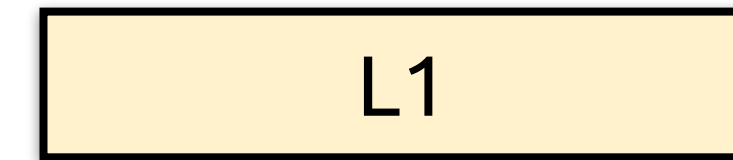




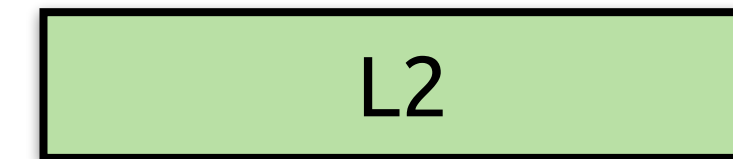
# CPU Architecture - Intel Core i5-7300



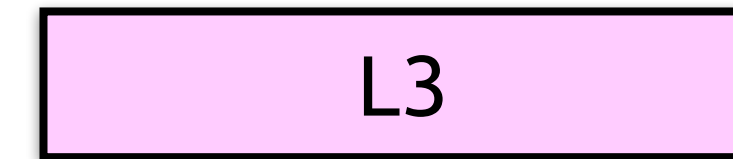
# CPU Architecture - Intel Core i5-7300



~1 ns



~3 times slower than L1



~10 times slower than L1



~50/100 times slower than L1



As a developer,  
I would like my application to leverage CPU caches





CPU Architecture

**Locality of Reference**

Data-Oriented Design

Caching Pitfall

Concurrency



# Locality of Reference

If a **particular memory location** is referenced, it is **likely** that...

Temporal  
Locality



The **same location** will be referenced again in a near future

Spatial  
Locality



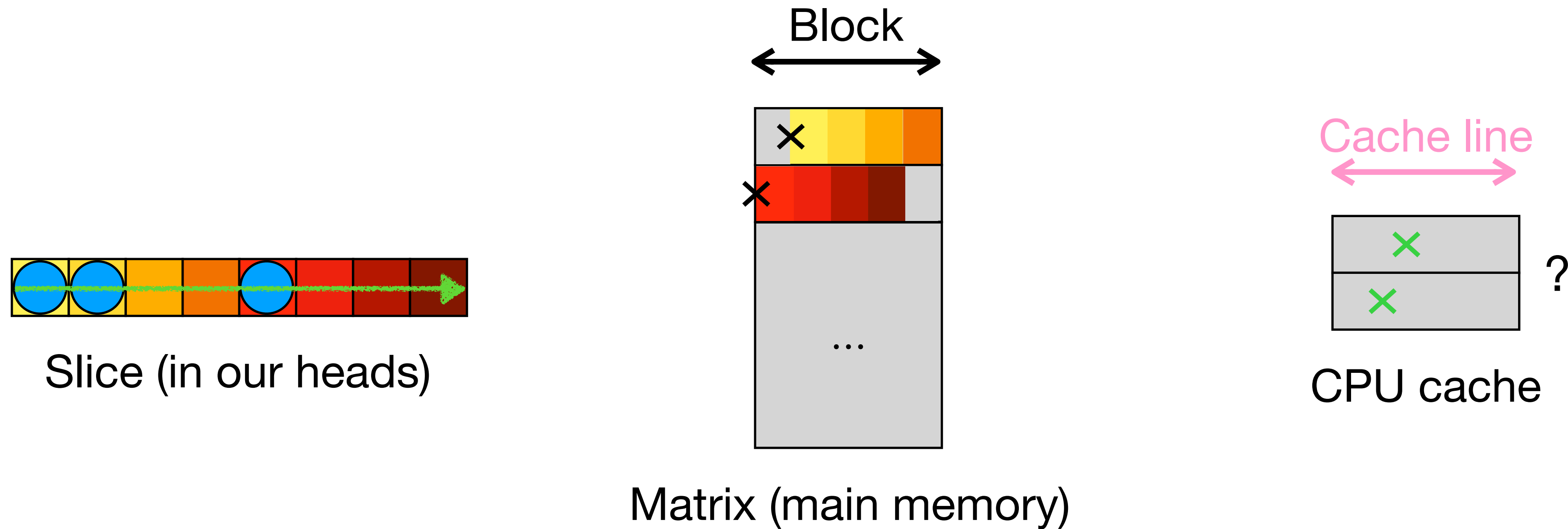
**Nearby memory locations** will be referenced in a near future

```
sum := 0
s := initSliceOfInts()
length := len(s)
for i := 0; i < length; i++ {
    sum += s[i]
}
```





# Spatial Locality



- Instead of copying a single variable, the processor will copy a **cache line**
- Cache line: **contiguous** segment of memory of a **fixed size** (usually 64 bytes)
- Limited number of cache miss (compulsory miss)
- Theory (other applications can run at the same time on the same core)
- Cache placement policy (L1, L2 or L3?)



# Helping the CPU

- To help the CPU, an application needs to leverage **locality of reference**
- ... and **predictability**



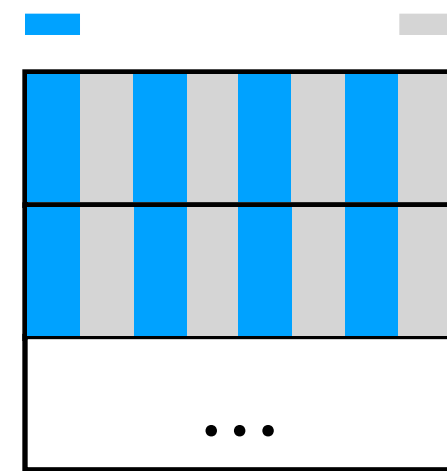


# Linked List Iteration

- Iterating on a **linked list** that **should be allocated contiguously** should be decent

Linked list iteration

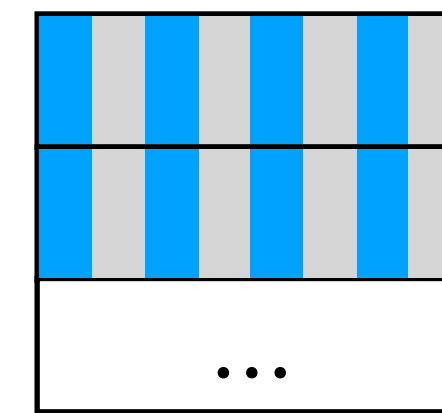
```
type Node struct {  
    Value int64  
    Next  *Node  
}
```



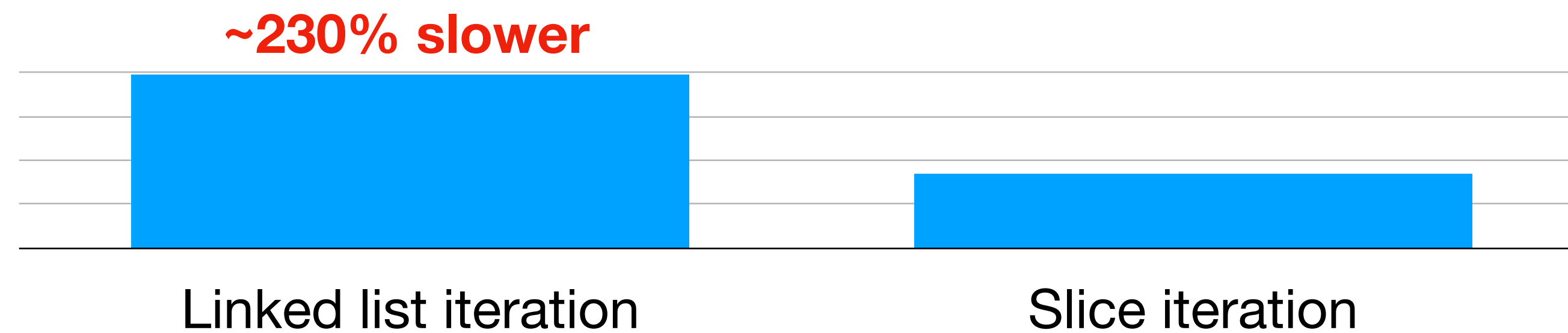
Main memory

Slice iteration: one element out of two

```
for i := 0; i < len(s); i+=2 {  
    sum += s[i]  
}
```



Main memory

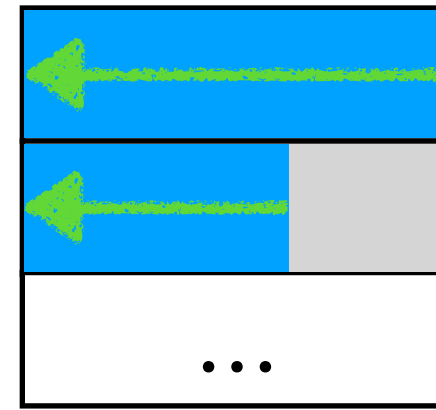


- Possible** spatial locality
- But **not predictable** for the CPU (no line fetching)



# Backwards Iteration

- What if we iterate **backwards** on a slice?



Main memory

Forwards

```
for i := 0; i < length; i++ {  
    r = s[i]  
}
```

Backwards

```
for i := length - 1; i >= 0; i-- {  
    r = s[i]  
}
```



Forwards

Backwards

- Spatial locality
- The CPU was able to **predict** that we iterate backwards
- Slightly faster because the bound check is faster



# How to Make Things Predictable?

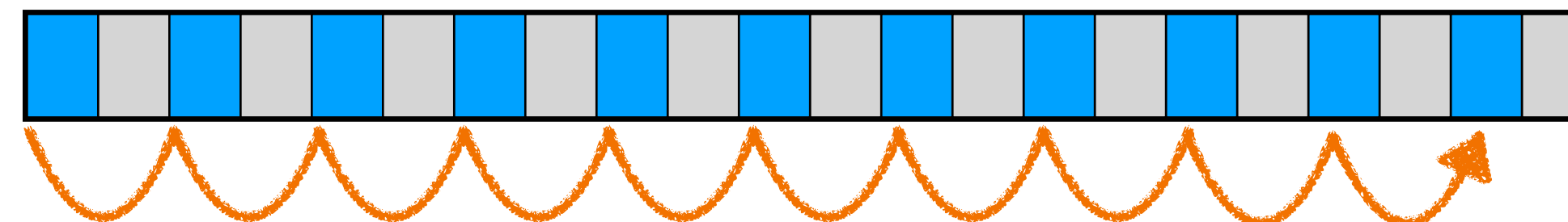
- Striding: how does a CPU work through our data?

- Unit stride: each element



Predictable

- Constant stride: each x element (e.g. one out of two)



Predictable but less efficient

- Non-unit stride: *might* be spread across memory (linked list)



Not predictable







- CPU caches are **extremely fast**
- A CPU doesn't cache a single variable but a **cache line**
- I can **help** the CPU if my application leverages:
  - **Locality of reference**
  - **Predictability**



CPU Architecture

Locality of Reference

**Data-Oriented Design**

Caching Pitfall

Concurrency



# Data-Oriented Design

- “The purpose of all programs and all parts of those programs is to **transform data** from one form to another” - Mike Acton
- Object-Oriented design is a way to mirror how we interact with the real world
- Yet, hardware does not like objects
- Data-Oriented design is about organising data in a way to get the **most value out of each cache line**





# Data-Oriented Design

- 2 concrete examples:
  - Structure alignment
  - Slice of structures vs structure of slices



# Structure Alignment

```
type I1 struct {  
    b1 bool  
    i  int64  
    b2 bool  
}
```

```
func BenchmarkI1(b *testing.B) {  
    s := make([]I1, it)  
    var r int64  
    b.ResetTimer()  
    for j := 0; j < it; j++ {  
        r += s[j].i  
    }  
    result = r  
}
```

```
type I2 struct {  
    i  int64  
    b1 bool  
    b2 bool  
}
```

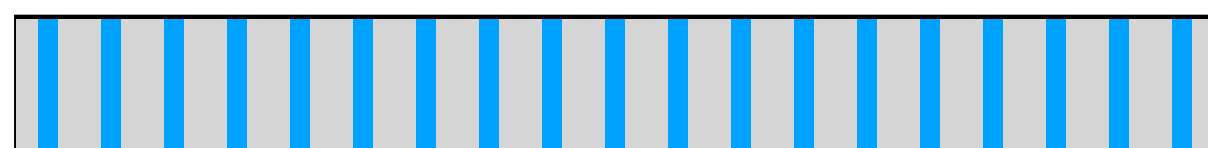
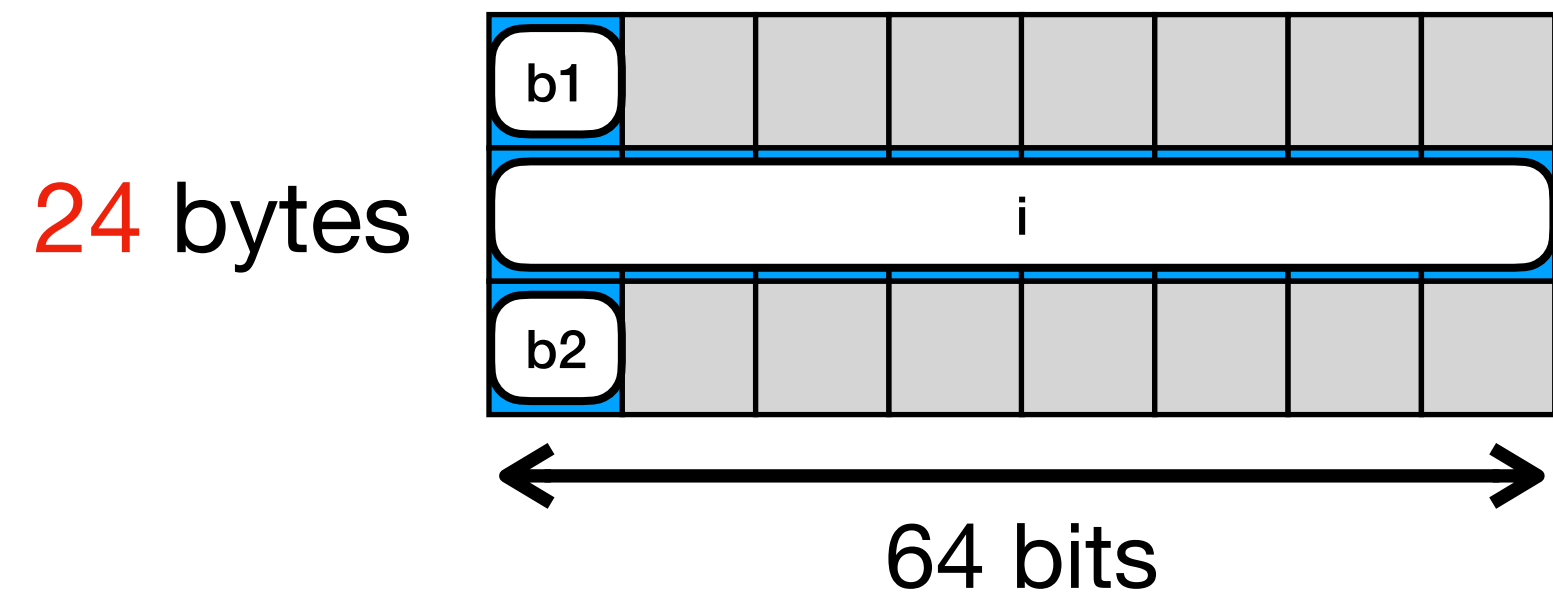
```
func BenchmarkI2(b *testing.B) {  
    s := make([]I2, it)  
    var r int64  
    b.ResetTimer()  
    for j := 0; j < it; j++ {  
        r += s[j].i  
    }  
    result = r  
}
```



# Structure Alignment

- The size of a structure is a **multiple of the word size** (64 bits on a 64-bit, etc.)

```
type I1 struct {  
    b1 bool  
    i int64  
    b2 bool  
}
```

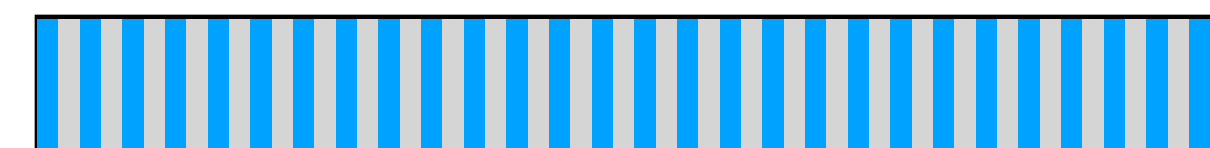
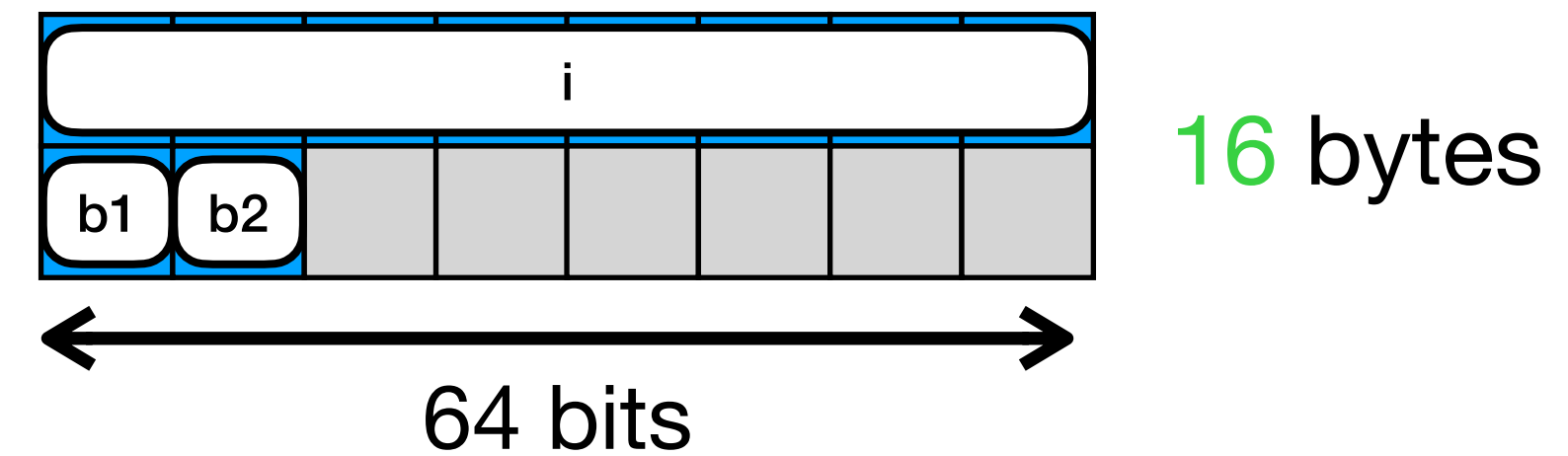


10k slice: 3750 cache lines  
(cache line: 64 bytes)

Structure  
alignment

In  
memory

```
type I2 struct {  
    i int64  
    b1 bool  
    b2 bool  
}
```



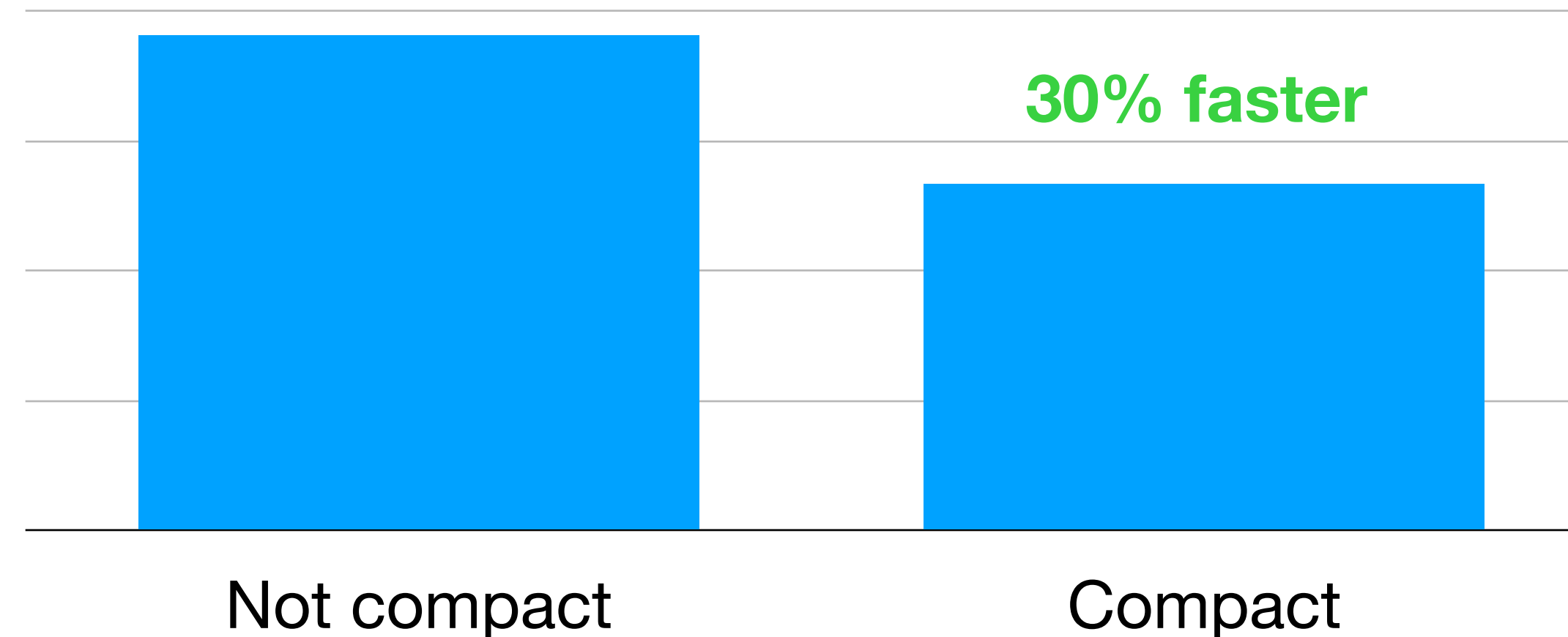
10k slice: 2500 cache lines



# Structure Alignment

```
type I1 struct {  
    b1 bool  
    i  int64  
    b2 bool  
}
```

```
type I2 struct {  
    i  int64  
    b1 bool  
    b2 bool  
}
```



- Memory footprint (GC pressure)
- Iterating over a **compact** data structure is more efficient as it requires **less caches lines**





# Slice of Structures vs Structure of Slices

```
type Struct1 struct {  
    a int32  
    b int64  
}
```

```
func BenchmarkSliceOfStructures(b *testing.B) {  
    s := make([]Struct1, it)  
    var r int32  
    b.ResetTimer()  
    for i := 0; i < b.N; i++ {  
        for i := 0; i < it; i++ {  
            r = s[i].a  
        }  
    }  
    result = r  
}
```

```
type Struct2 struct {  
    a []int32  
    b []int64  
}
```

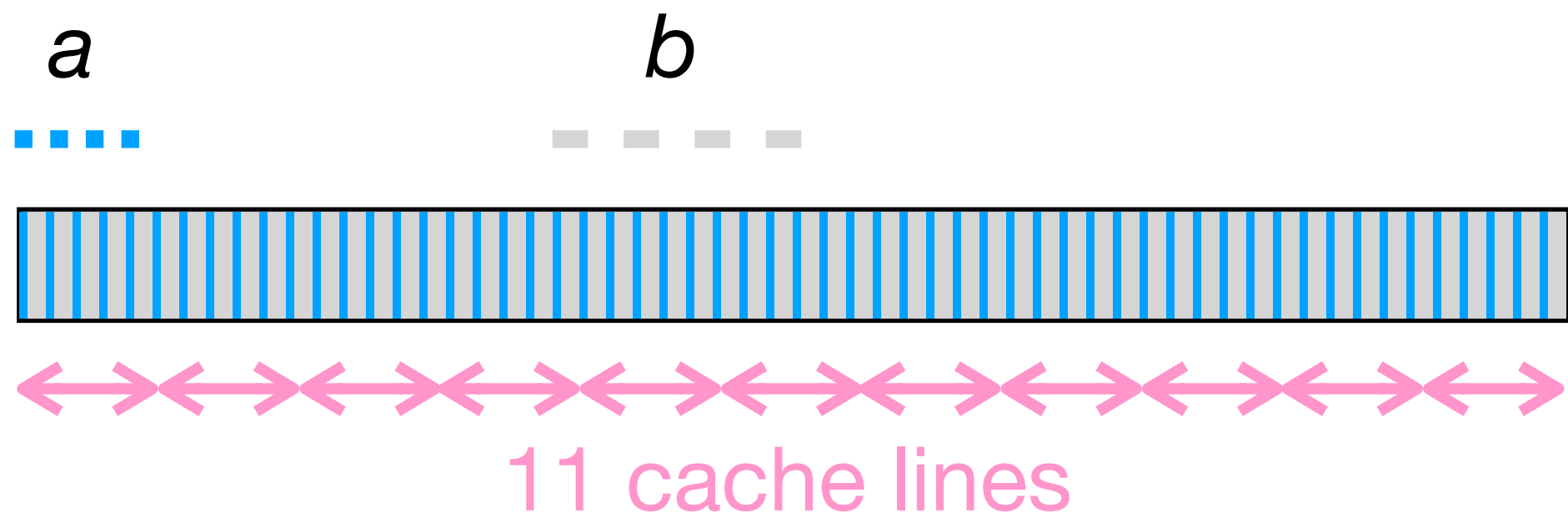
```
func BenchmarkStructureOfSlices(b *testing.B) {  
    s := Struct2{  
        a: make([]int32, it),  
        b: make([]int64, it),  
    }  
    var r int32  
    b.ResetTimer()  
    for i := 0; i < b.N; i++ {  
        for i := 0; i < it; i++ {  
            r = s.a[i]  
        }  
    }  
    result = r  
}
```



## Slice of structs

```
type Struct1 struct {  
    a int32  
    b int64  
}  
s := make([]Struct1, it)
```

Constant  
stride

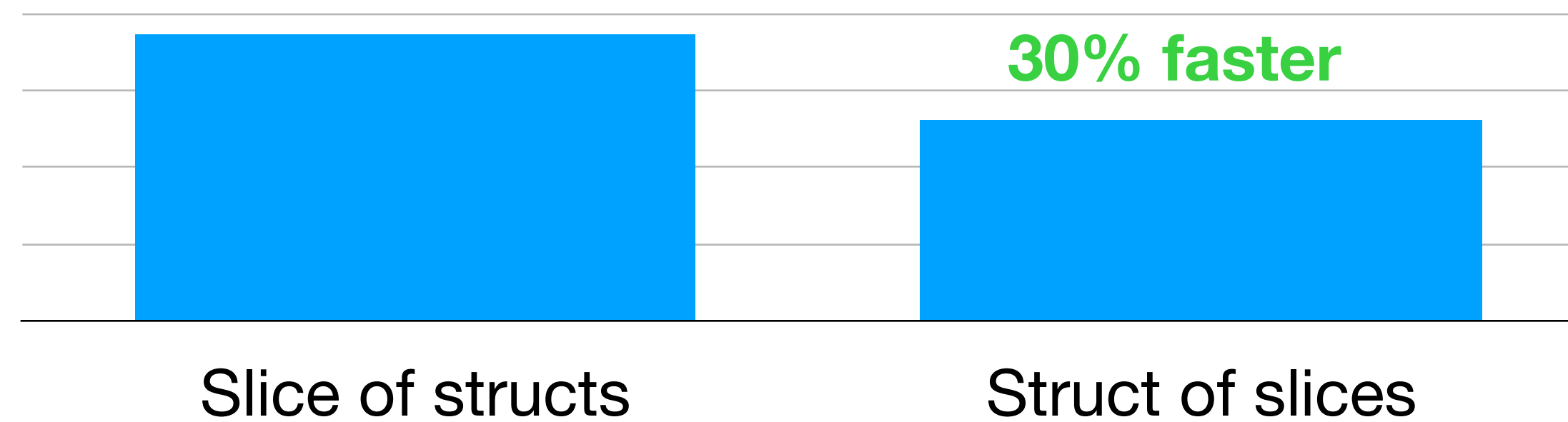


## Struct of slices

```
type Struct2 struct {  
    a []int32  
    b []int64  
}  
s := Struct2{  
    a: make([]int32, it),  
    b: make([]int64, it),  
}
```

Unit  
stride

In  
memory



# Slice of Structures vs Structure of Slices

- A concrete example: Go standard **flate package**
- Flate is a **compression** algorithm based on two other algorithms: huffman encoding and LZ77 compression





# Slice of Structures vs Structure of Slices

Go flate package

```
type hcode struct {  
    code, len uint16  
}
```

```
type huffmanEncoder struct {  
    codes []hcode  
    freqcache []literalNode  
    bitCount [17]int32  
    lns byLiteral // stored to avoid repeated allocation in generate  
    lfs byFreq // stored to avoid repeated allocation in generate  
}
```

Go flate package modified

```
type hcodes struct {  
    code []uint16  
    len []uint16  
}
```

```
type huffmanEncoder struct {  
    codes hcodes  
    freqcache []literalNode  
    bitCount [17]int32  
    lns byLiteral // stored to avoid repeated allocation in generate  
    lfs byFreq // stored to avoid repeated allocation in generate  
}
```

[https://github.com/golang/go:src/compress/flate/huffman\\_code.go](https://github.com/golang/go:src/compress/flate/huffman_code.go)

- 5 iteration loops on either *hcode.code* or *hcode.len*

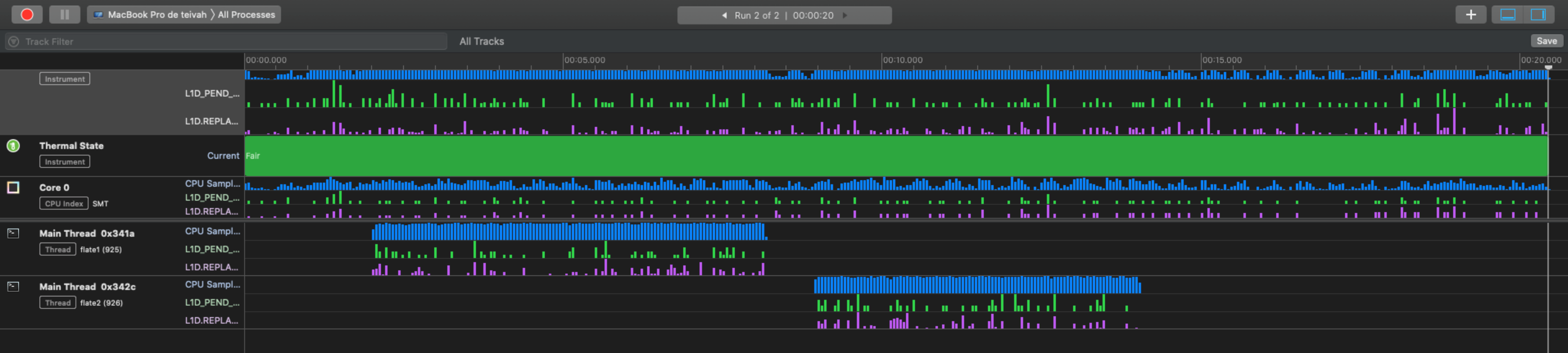
- Example:

```
for i := 0; i < numCodegens; i++ {  
    value := uint(w.codegenEncoding.codes[codegenOrder[i]].len)  
    w.writeBits(int32(value), nb: 3)  
}
```

```
for i := 0; i < numCodegens; i++ {  
    value := uint(w.codegenEncoding.codes.len[codegenOrder[i]])  
    w.writeBits(int32(value), nb: 3)  
}
```

- Metrics?





Counters > Call Tree > Call Tree

Total	Running Time	Self (ms)	L1D_PEND_MISS.PENDING	L1D.REPLACEMENT	Symbol Name
103	103.0ms	0.3%	0,0	332 042 519 206 384 200 000	332 041 393 326 658 750 000 ▶firefox (332)
5944	5944.0ms	21.9%	0,0	22 634 155 259 928 865 000 0...	22 634 154 978 442 250 000 000 ▼flate1 (925)
4	4.0ms	0.0%	0,0	36 892 925 199 884 853 000	36 893 488 147 244 220 000 ▶_dyld_start dyld
2	2.0ms	0.0%	0,0	281 474 072 705 310	93 445 947 ▶dyld_get_min_os_version libdyld.dylib
1	1.0ms	0.0%	0,0	10 253	0 ▶runtime.bgscavenge flate1
5934	5934.0ms	21.9%	2,0	22 578 815 309 178 397 000 0...	22 578 814 746 220 986 000 000 ▼runtime.main flate1
5932	5932.0ms	21.9%	5,0	22 578 815 309 178 397 000 0...	22 578 814 746 220 986 000 000 ▼main.main flate1
1	1.0ms	0.0%	1,0	1 491	0 github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*Writer).Reset flate1
5906	5906.0ms	21.8%	0,0	22 523 475 358 404 820 000 0...	22 523 474 513 998 427 000 000 ▼github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*compressor).close flate1
5900	5900.0ms	21.8%	2817,0	22 505 028 614 335 974 000 0...	22 505 027 769 925 007 000 000 ▼github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*compressor).storeHuff flate1
3083	3083.0ms	11.4%	1398,0	12 248 628 494 795 629 000 0...	12 248 638 064 937 155 000 000 ▼github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*huffmanBitWriter).writeBlockHuff flate1
45	45.0ms	0.1%	45,0	221 360 928 872 791 280 000	221 360 928 883 581 350 000 github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*huffmanBitWriter).dynamicSize flate1
26	26.0ms	0.0%	26,0	92 233 157 434 736 660 000	92 233 720 369 331 080 000 github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*huffmanBitWriter).writeDynamicHeader flate1
1614	1614.0ms	5.9%	933,0	6 124 313 965 952 697 000 000	6 124 319 032 469 608 000 000 ▶github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*huffmanEncoder).generate flate1
6	6.0ms	0.0%	4,0	18 446 744 068 843 934 000	18 446 744 073 418 738 000 ▶github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*huffmanBitWriter).writeStoredHeader flate1
20	20.0ms	0.0%	20,0	36 893 206 696 542 073 000	36 893 488 148 606 350 000 github.com/teivah/mechanical-sympathy-in-go/cmd/flate.(*compressor).write flate1
2	2.0ms	0.0%	0,0	281 476 883 776 767	308 653 940 ▶runtime.mcall flate1
1	1.0ms	0.0%	0,0	18 446 462 599 628 040 000	18 446 744 073 615 712 000 ▶thread_start libsystem_pthread.dylib
4944	4944.0ms	18.2%	0,0	13 982 632 007 901 488 000 0...	13 982 632 007 873 746 000 000 ▼flate2 (926)
2	2.0ms	0.0%	0,0	18 446 462 598 960 579 000	18 446 744 073 584 247 000 ▶_dyld_start dyld
1	1.0ms	0.0%	0,0	1 239 265	40 778 ▶dyld_get_min_os_version libdyld.dylib
1	1.0ms	0.0%	1,0	281 474 758 259 731	125 616 097 runtime.asmcgocall flate2
4938	4938.0ms	18.2%	4,0	13 964 185 263 827 763 000 0...	13 964 185 263 800 037 000 000 ▼runtime.main flate2
4934	4934.0ms	18.2%	7,0	13 945 738 519 762 044 000 0...	13 945 738 519 726 745 000 000 ▼main.main flate2
2	2.0ms	0.0%	1,0	18 446 462 598 382 694 000	18 446 744 073 570 361 000 ▶github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*Writer).Reset flate2
4905	4905.0ms	18.1%	6,0	13 835 058 899 743 300 000 0...	13 835 058 055 284 682 000 000 ▼github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*compressor).close flate2
4895	4895.0ms	18.1%	2523,0	13 835 058 899 743 300 000 0...	13 835 058 055 284 682 000 000 ▼github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*compressor).storeHuff flate2
2372	2372.0ms	8.7%	787,0	6 714 611 183 731 171 000 000	6 714 614 842 831 794 000 000 ▼github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*huffmanBitWriter).writeBlockHuff flate2
33	33.0ms	0.1%	33,0	92 233 720 357 337 330 000	92 233 720 367 897 770 000 github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*huffmanBitWriter).dynamicSize flate2
34	34.0ms	0.1%	34,0	166 020 133 688 638 500 000	166 020 696 661 746 650 000 github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*huffmanBitWriter).writeDynamicHeader flate2
1518	1518.0ms	5.6%	911,0	4 039 836 670 803 177 400 000	4 039 836 952 149 304 500 000 ▶github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*huffmanEncoder).generate flate2
4	4.0ms	0.0%	4,0	937 650	47 947 github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*huffmanBitWriter).writeStoredHeader flate2
20	20.0ms	0.0%	20,0	73 786 413 347 266 200 000	73 786 976 294 798 580 000 github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(*compressor).write flate2
1	1.0ms	0.0%	0,0	450 604	7 391 ▶runtime.schedinit flate2
1	1.0ms	0.0%	0,0	1 905 519	32 467 ▶thread_start libsystem_pthread.dylib

Heaviest Stack Trace

- 4944.0 flate2 (926)
- 4938.0 runtime.main
- 4934.0 main.main
- 4905.0 github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(\*compressor).close
- 4895.0 github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(\*compressor).storeHuff
- 2372.0 github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(\*huffmanBitWriter).writeBlockHuff
- 1518.0 github.com/teivah/mechanical-sympathy-in-go/cmd/flate2.(\*huffmanEncoder).generate
- 445.0 sort.Sort
- 436.0 sort.quickSort
- 204.0 sort.quickSort
- 63.0 sort.quickSort
- 31.0 sort.insertionSort

Xcode Instruments



# Slice of Structures vs Structure of Slices

Go flate package

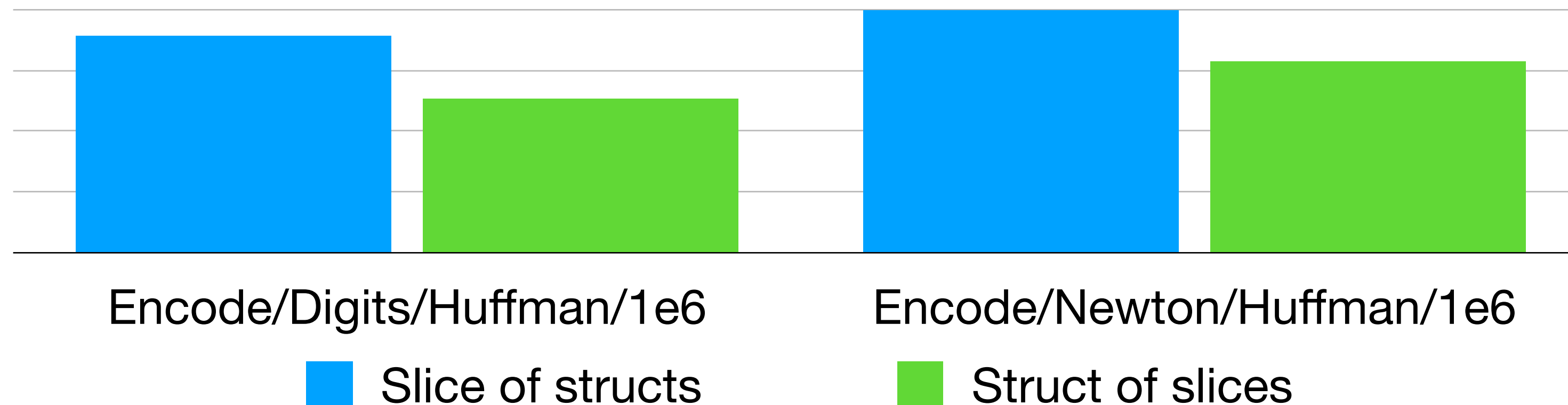
```
type hcode struct {
    code, len uint16
}

type huffmanEncoder struct {
    codes []hcode
    freqcache []literalNode
    bitCount [17]int32
    lns byLiteral // stored to avoid repeated allocation in generate
    lfs byFreq // stored to avoid repeated allocation in generate
}
```

Go flate package modified

```
type hcodes struct {
    code []uint16
    len []uint16
}

type huffmanEncoder struct {
    codes hcodes
    freqcache []literalNode
    bitCount [17]int32
    lns byLiteral // stored to avoid repeated allocation in generate
    lfs byFreq // stored to avoid repeated allocation in generate
}
```



Between **21% and 28% faster**





- I can **design algorithms** to leverage CPU caches
- I can also **organise my data** to get the most value out of cache lines
- **Unit stride** > Constant stride > Non-unit stride





CPU Architecture  
Locality of Reference  
Data-Oriented Design  
**Caching Pitfall**  
Concurrency



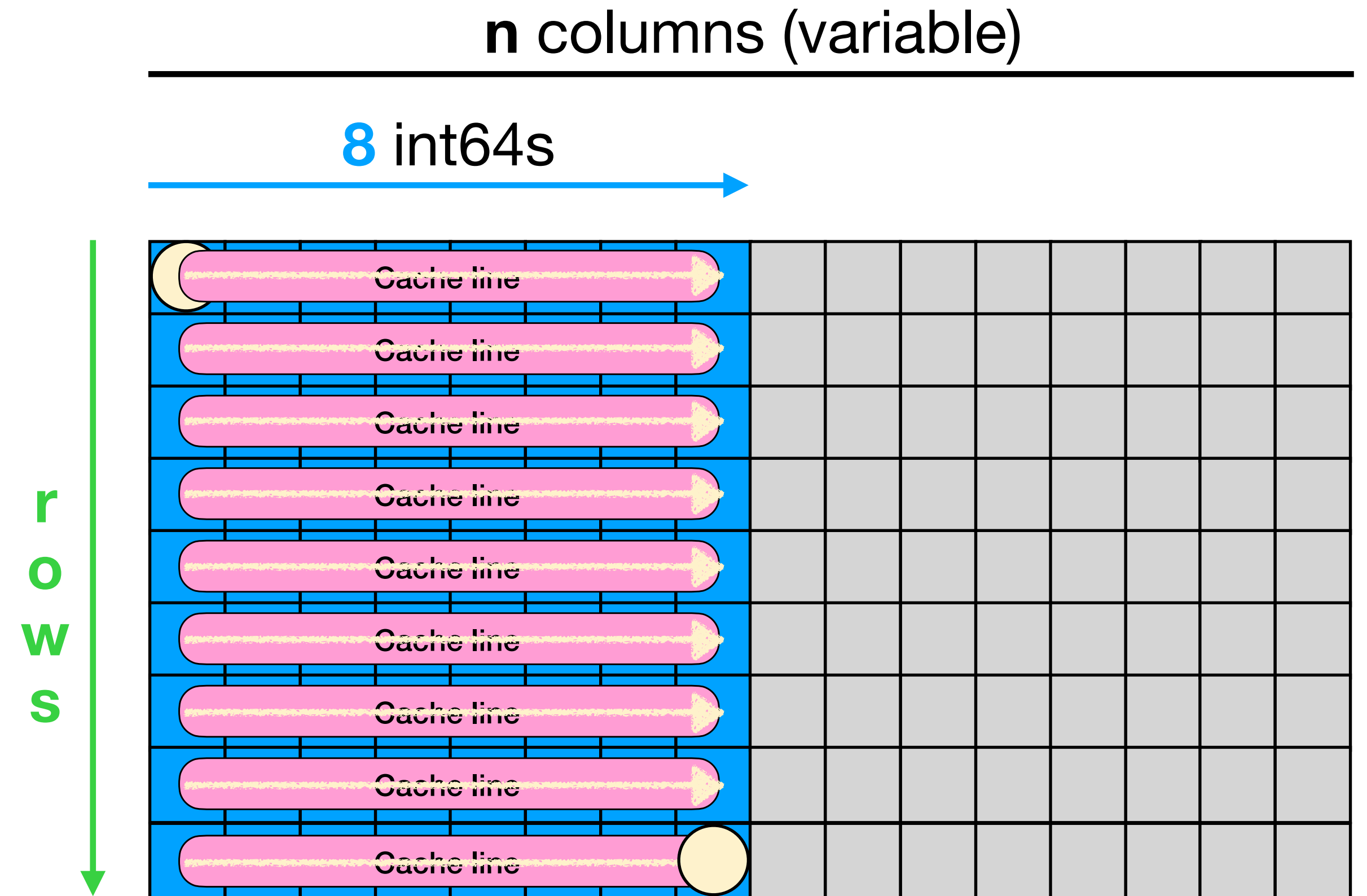
- Two-dimensional array of int64s  
64 bytes cache line (8 elements)
- Traverse **each row multiple times** the first **8 columns** only

```

for 0..k {
  for i in 0..rows {
    for j in 0..8 {
      sum += a[i][j]
    }
  }
}

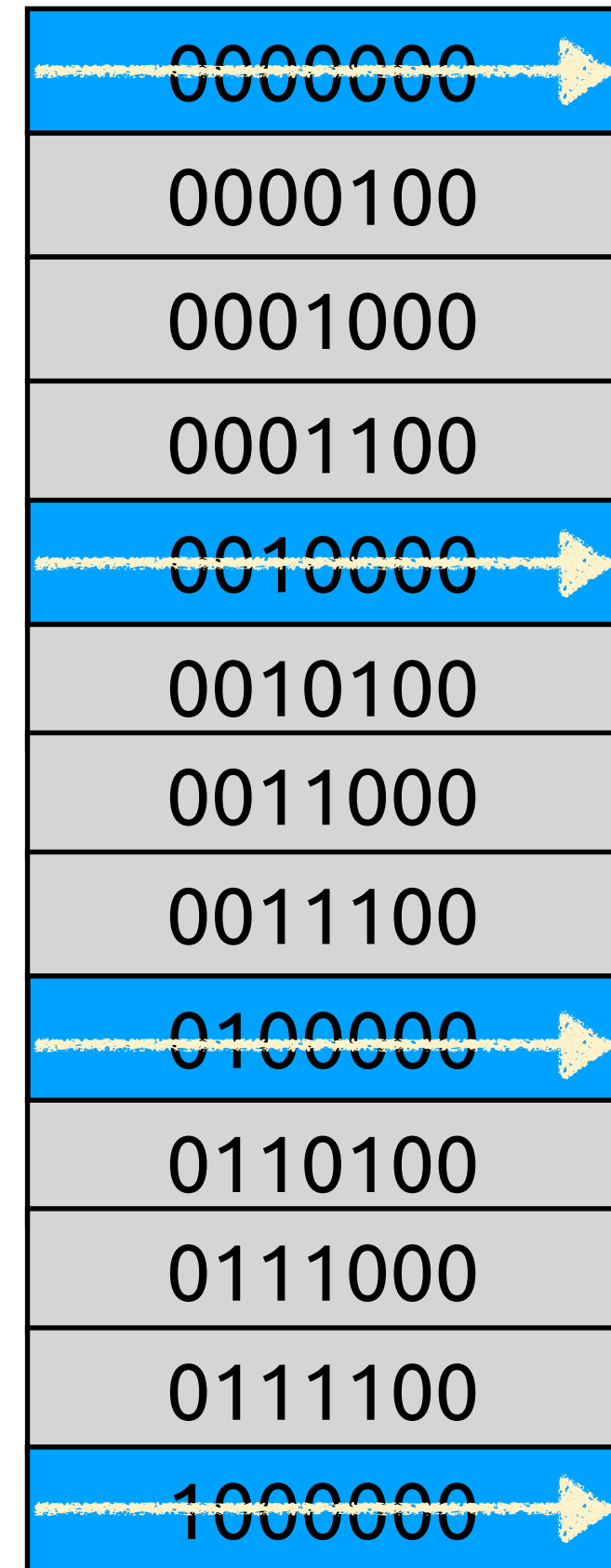
```

- **rows** is small enough so that each line should fit in the cache
- The execution time depends on **n** (?)
- Depending on **n**, the execution can be up to **100% slower**

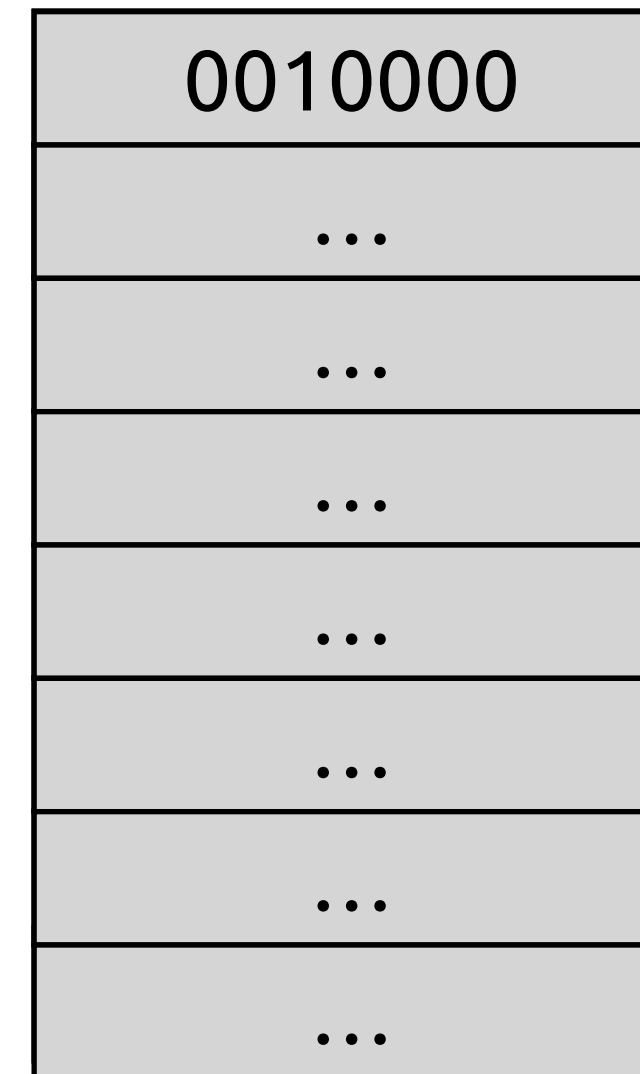


A block is referenced by an address  
We want to iterate on each **blue block**

Matrix in memory



Cache



Program:

```
...  
load address 0000000  
...
```

- In a **fully associative cache**, we may have to traverse the **whole** cache to check if an address is present
- Example on an Intel Core i5-7300 L1D: we need to iterate on **512 lines**
- Solution: **partitioning**



# Matrix in memory

0000000
0000100
0001000
0001100
0010000
0010100
0011000
0011100
0100000
0110100
0111000
0111100
1000000

# Load

This address belongs to **set 0**

0 0 0 0 0 0 0

tb      si      bo

A cache is **partitioned** into **sets**

A block can belong to **only one** set

**k-way associative** cache: k lines per set

E.g. 8 lines, 2-way associative

nb of sets =  $8 / 2 = 4$

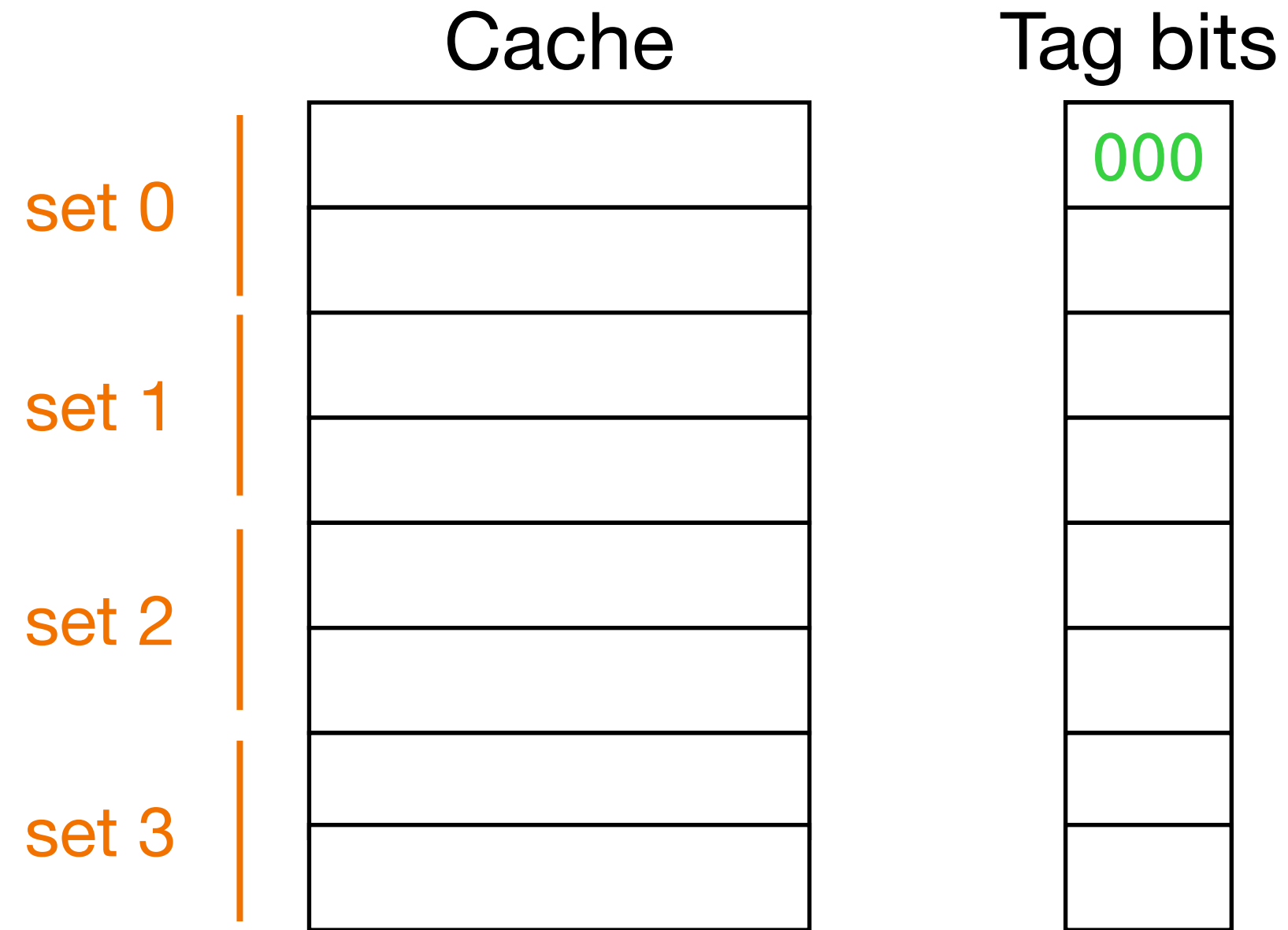
$4 = 2^2$

**2** represents the **set index (si)**

E.g Block size: 4 bits

$4 = 2^2$

**2** represents the **block offset (bo)**





Matrix  
in memory

0000000
0000100
0001000
0001100
0010000
0010100
0011000
0011100
0100000
0110100
0111000
0111100
1000000

Load

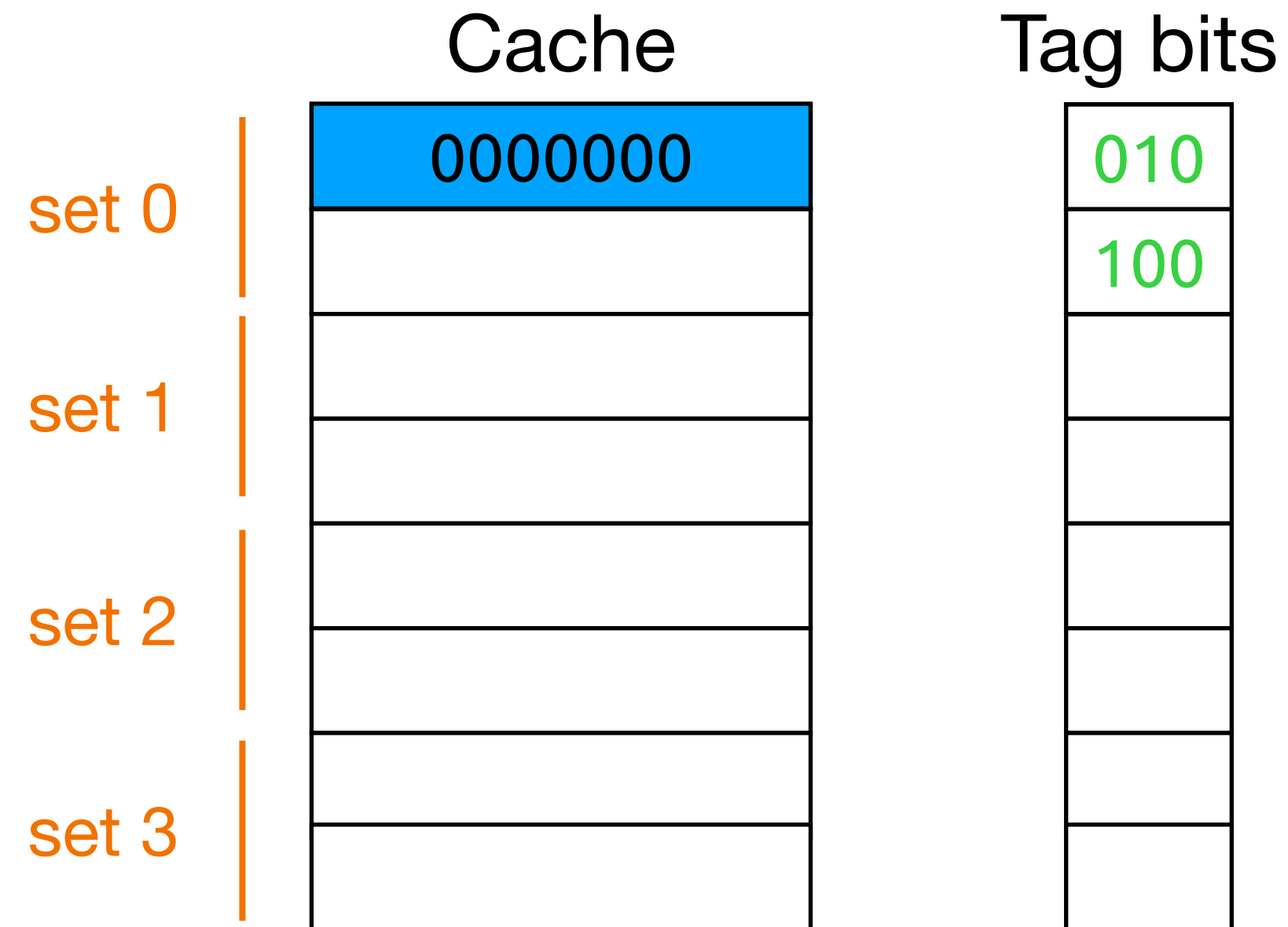
0 0 0	0 0	0 0
tb	si	bo
0 0 1	0 0	0 0
tb	si	bo
0 1 0	0 0	0 0
tb	si	bo
1 0 0	0 0	0 0
tb	si	bo

E.g Block size: 4 bits  
 $4 = 2^2$

2 represents the  
**block offset (bo)**

A cache is **partitioned** into **sets**  
A block can belong to **only one** set  
**k-way associative** cache: k lines per set

E.g. 8 lines, 2-way associative  
nb of sets =  $8 / 2 = 4$   
 $4 = 2^2$   
2 represents the **set index (si)**



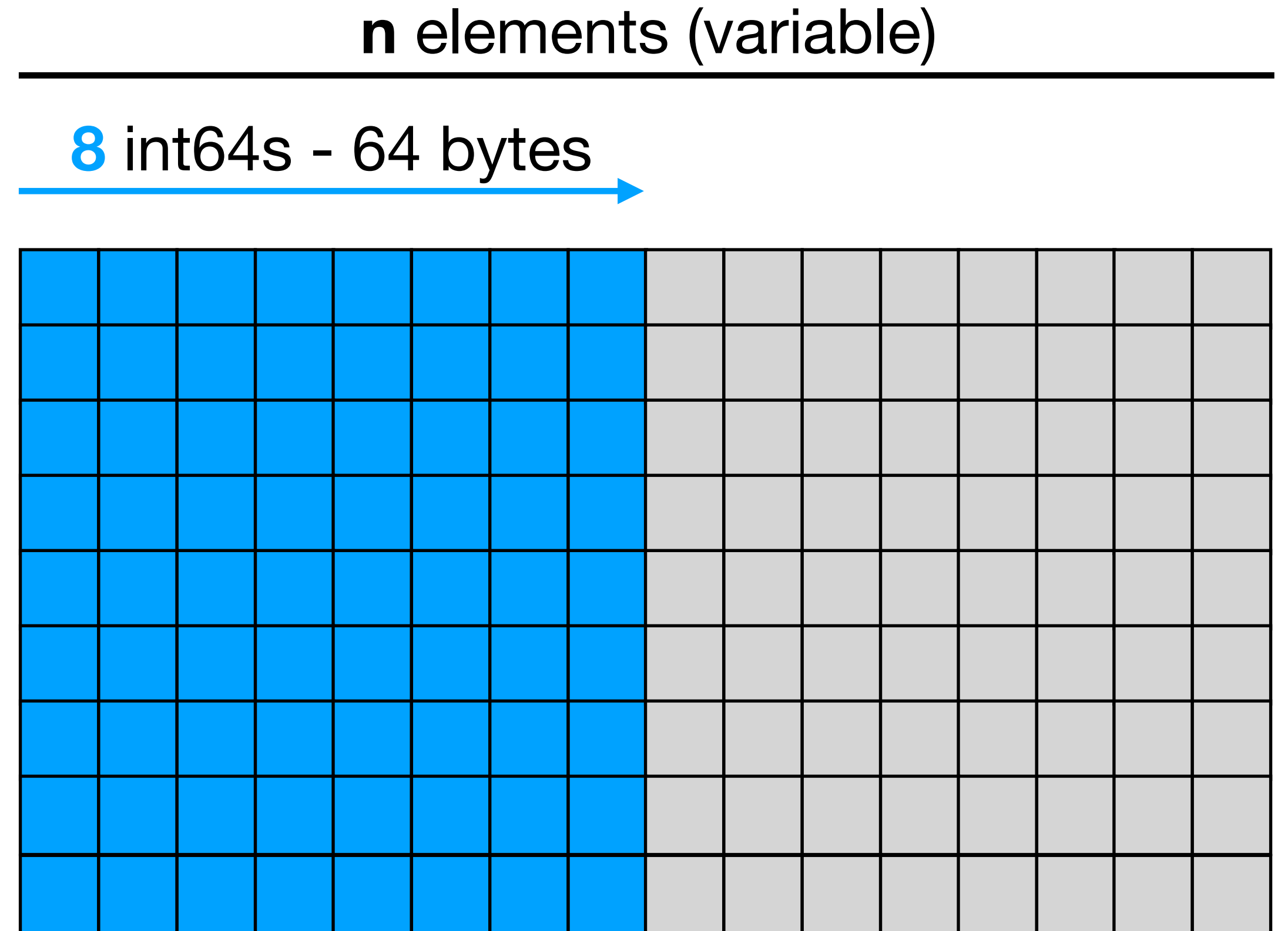
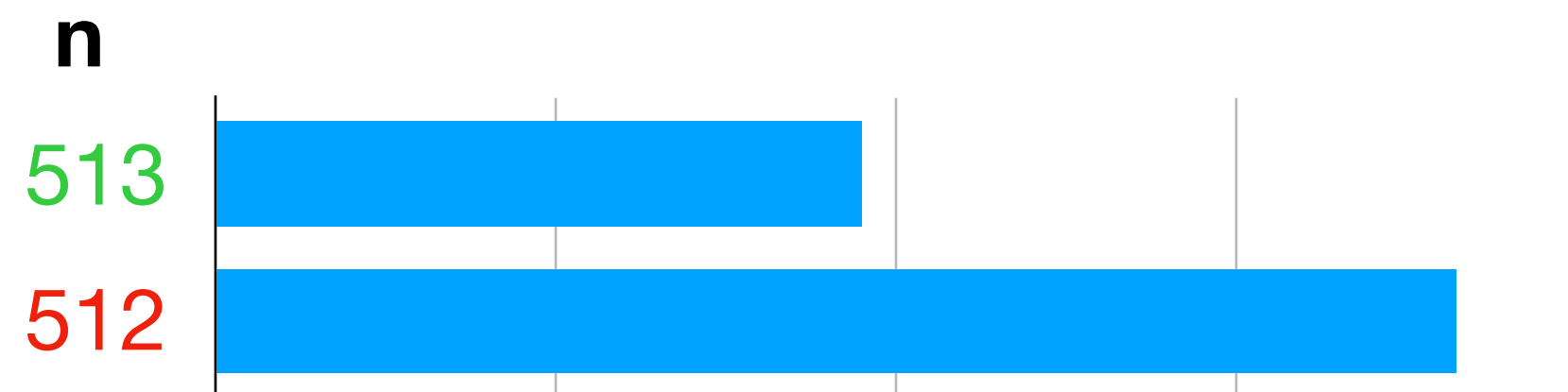
The distribution is not even, we used **only one set**

It will generate a lot of cache misses (conflict miss)

This constant stride is called the **critical stride**



- Critical stride = nb sets x cache line size
- Example with an Intel Core i5-7300:
  - Cache line = 64 bytes
  - 32 KB, 8-way set associative, **64 sets**
  - Critical stride =  $64 \times 64 = \mathbf{4\ KB}$
- We reach a critical stride with  $n = \mathbf{512\ elements}$
- If  $n = 512$ , we are going to use **1 set only**





- CPU caches are **partitioned**
- Depending on my data, my application can occupy a **fraction** of the cache only
- **Critical stride**

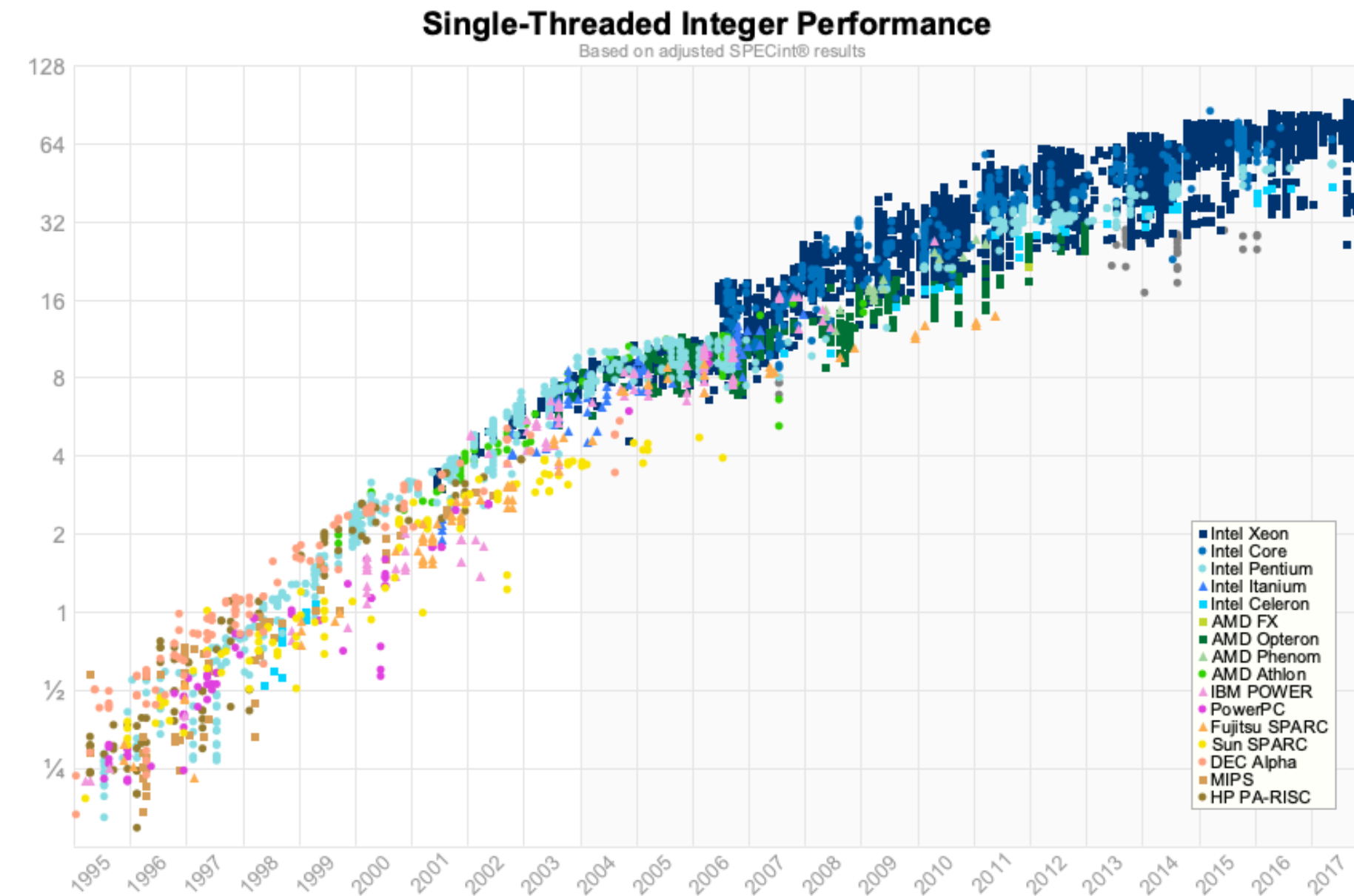


CPU Architecture  
Locality of Reference  
Data-Oriented Design  
Caching Pitfall  
**Concurrency**





# Why Concurrency?



- Instead of focusing on clock speed, vendors focus on **multicores** and **hyperthreading architectures**
- The free lunch is over - Herb Sutter, 2005
- We cannot rely solely on the **hardware** to make our programs faster  
**Concurrency** is the next major revolution in how we write software 🐻



```
type Struct struct {  
    n int  
}
```

```
var result int
```

```
func BenchmarkIteration(b *testing.B) {
```

```
    structA := Struct{} // Initialization  
    structB := Struct{} // Initialization  
    wg := sync.WaitGroup{}  
    b.ResetTimer()
```

```
    for i := 0; i < b.N; i++ {
```

```
        wg.Add(delta: 2)
```

```
        go func() { // Spin up first goroutine  
            for j := 0; j < iteration; j++ {  
                structA.n += j  
            }  
            wg.Done()  
        }()
```

```
        go func() { // Spin up second goroutine  
            for j := 0; j < iteration; j++ {  
                structB.n += j  
            }  
            wg.Done()  
        }()
```

```
        wg.Wait() // Wait
```

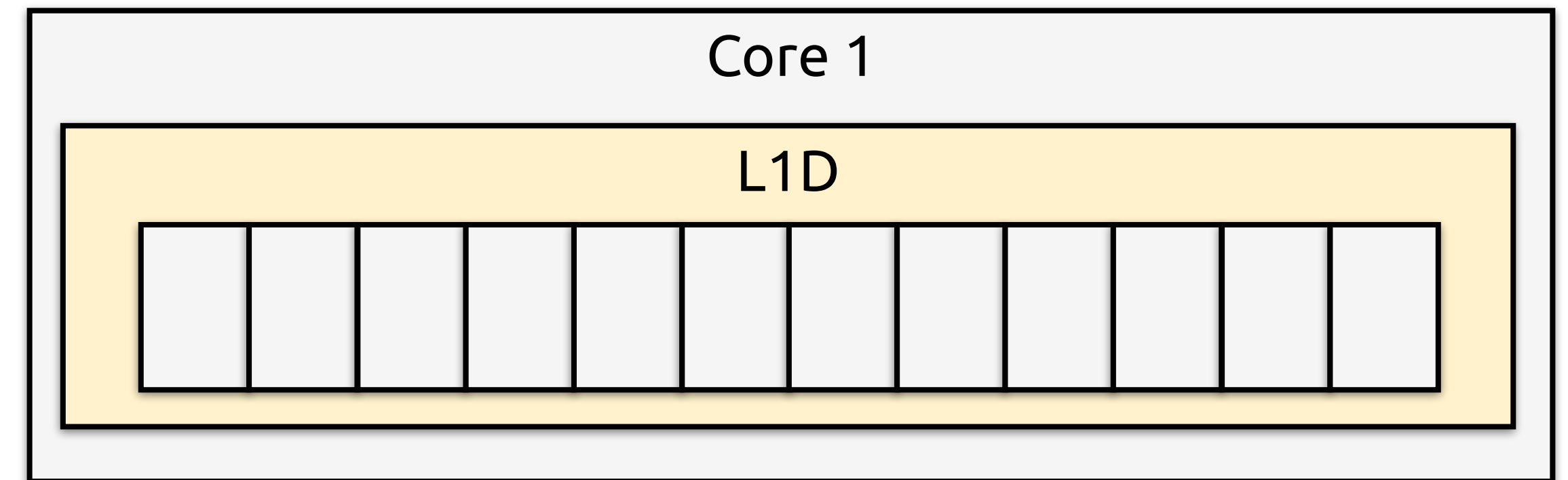
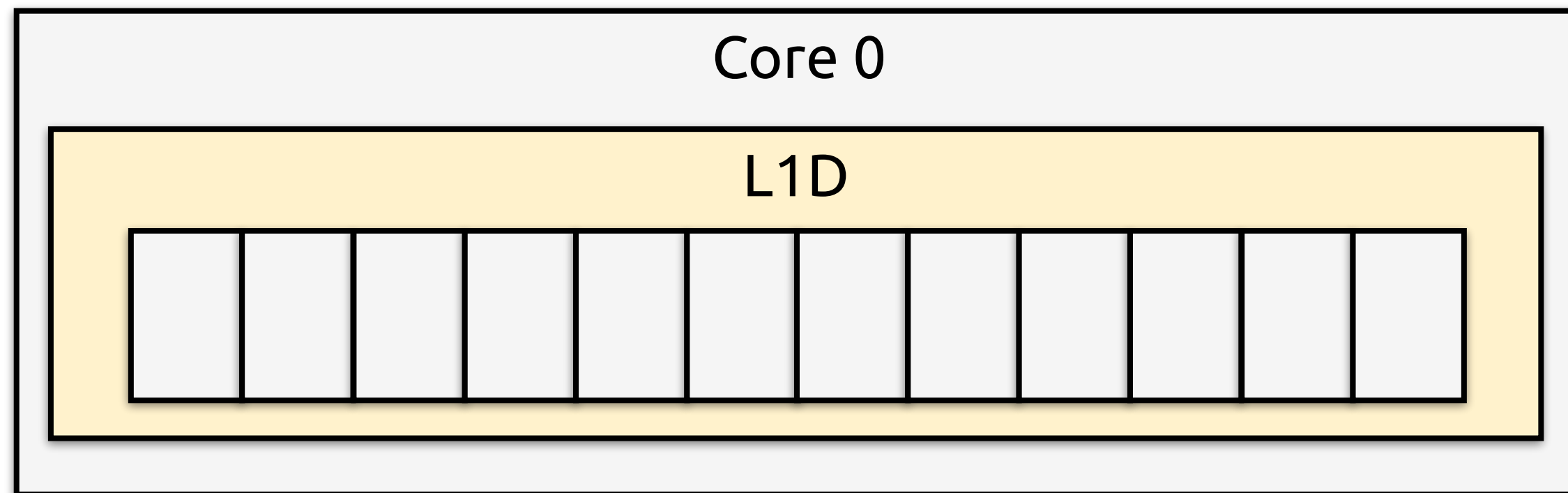
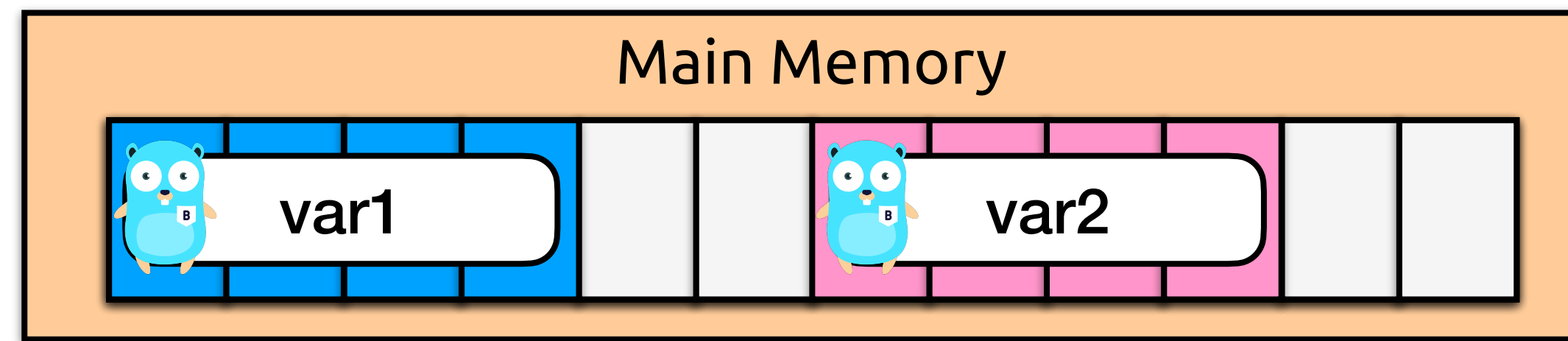
```
        result = structA.n + structB.n // Aggregate
```

```
    }
```

```
}
```

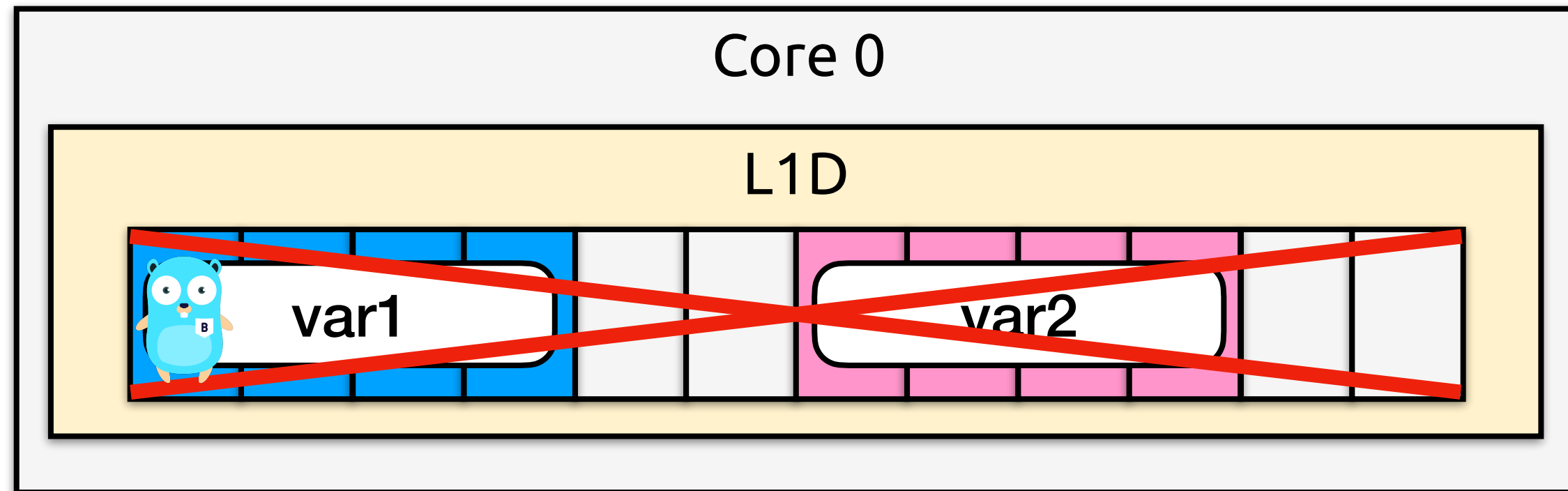
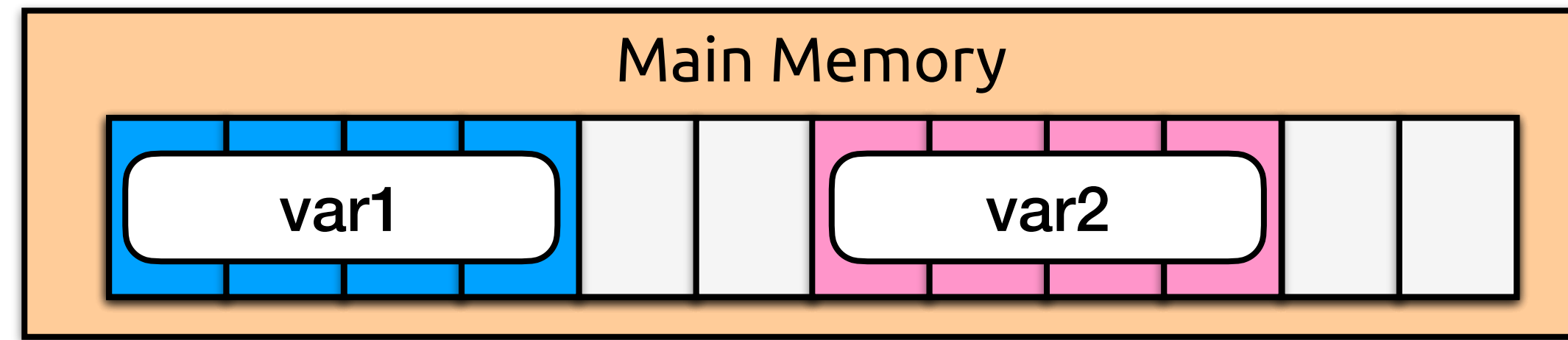
**Race-free  
implementation!**



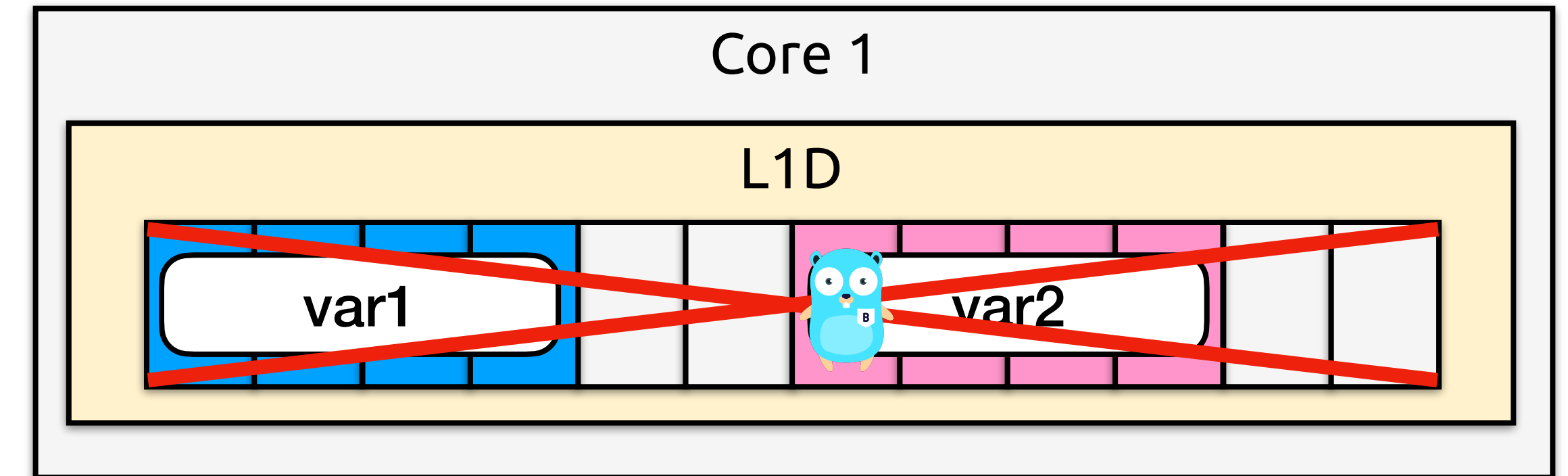


- What if both goroutines want to update their own lines
- The CPU must guarantee **cache coherency**
- MESI protocol (Modified, Exclusive, Shared, Invalid)





Update



Update

- Why does it matter?
- **False sharing** - a cache line is shared across two cores with at least one goroutine being a writer
- Sharing memory is an **illusion**





```

type Struct struct {
    n int
}

var result int

func BenchmarkIteration(b *testing.B) {
    structA := Struct{} // Initialization
    structB := Struct{} // Initialization
    wg := sync.WaitGroup{}
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        wg.Add(delta: 2)
        go func() { // Spin up first goroutine
            for j := 0; j < iteration; j++ {
                structA.n += j
            }
            wg.Done()
        }()
        go func() { // Spin up second goroutine
            for j := 0; j < iteration; j++ {
                structB.n += j
            }
            wg.Done()
        }()
        wg.Wait() // Wait
        result = structA.n + structB.n // Aggregate
    }
}

```

*structA.n* and *structB.n* belongs to the **same cache line**



# False Sharing

- How to prevent false sharing?
- Solution 1:  
Do not communicate by sharing memory;  
instead, share memory by **communicating**



```
func BenchmarkIterationCommunication(b *testing.B) {
    ch := make(chan int, 2)
    for i := 0; i < b.N; i++ {
        go func() { // Spin up first goroutine
            i := 0 // Local state
            for j := 0; j < iteration; j++ {
                i += j
            }
            ch <- i
        }()
        go func() { // Spin up second goroutine
            i := 0 // Local state
            for j := 0; j < iteration; j++ {
                i += j
            }
            ch <- i
        }()
        result = <-ch + <-ch // Wait and aggregate
    }
}
```

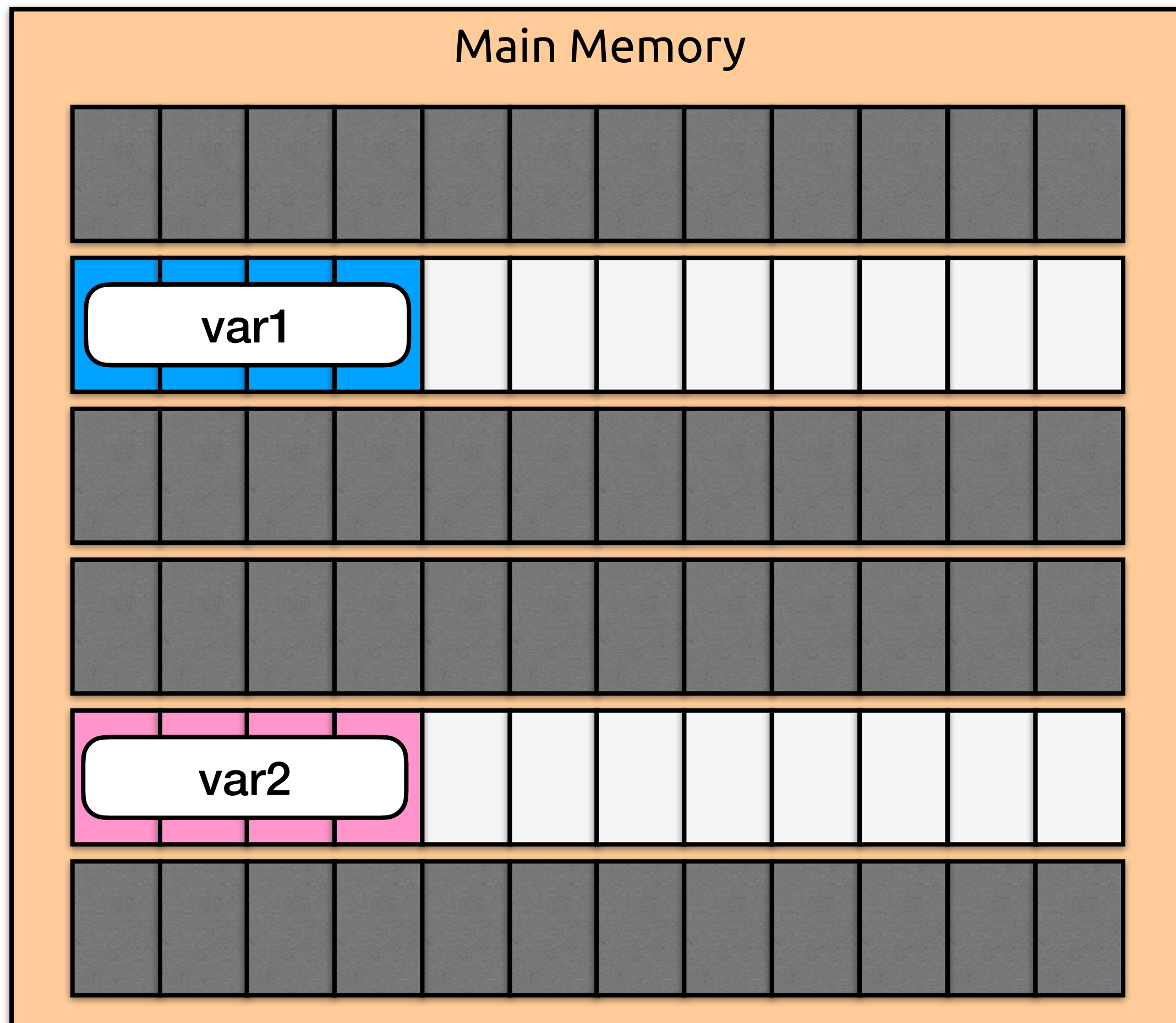


# False Sharing

- How to prevent false sharing?
- Solution 2: **padding**

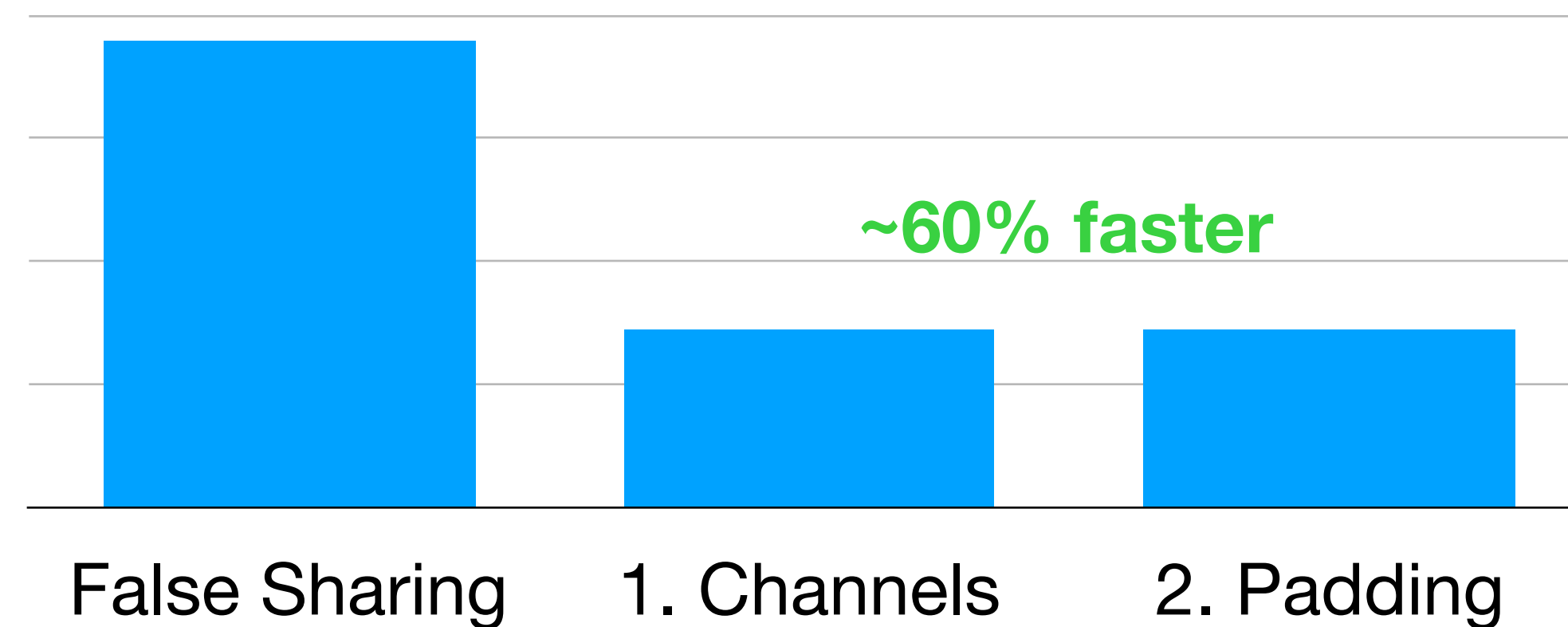
```
type PaddedStruct struct {  
    _ cpu.CacheLinePad // 64 bytes  
    n int  
    _ cpu.CacheLinePad // 64 bytes  
}
```

```
func BenchmarkIterationWithPadding(b *testing.B) {  
    structA := PaddedStruct{} // Initialization  
    structB := PaddedStruct{} // Initialization  
    wg := sync.WaitGroup{}  
    b.ResetTimer()  
  
    for i := 0; i < b.N; i++ {  
        wg.Add(delta: 2)  
        go func() { // Spin up first goroutine  
            for j := 0; j < iteration; j++ {  
                structA.n += j  
            }  
            wg.Done()  
        }()  
        go func() { // Spin up second goroutine  
            for j := 0; j < iteration; j++ {  
                structB.n += j  
            }  
            wg.Done()  
        }()  
        wg.Wait() // Wait  
    }  
}
```



# False Sharing

- Let's compare the results:



- Padding is hard - Dave Cheney
- Sometimes, padding is **necessary**. E.g. we are obliged to share memory and we want to prevent false sharing (library, etc.).



# Conclusion





# 3 Main Takeaways

- Sharing memory is an **illusion**
- A code that looks perfectly valid might still be **problematic** at CPU level:
  - Caching distribution
  - False sharing
- We can help the CPU with **locality of reference** and **predictability** (algorithms & data structures)

# What else?

- Watch out for premature:
  - Optimisations
  - Concurrency
- Mechanical sympathy goes beyond the very scope of CPUs



# Thank You

