Mechanical Sympathy in Go


8
MATRA F1
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

GOPHERCON
TURKEY

Teiva Harsanyi

teivah

Software Engineer - Beat

GOPHERCON TURKEY

HIRING

SOFTWARE ENGINEERS

ON CUTTING-EDGE TECHNOLOGIES

DISCOVER YOUR NEXT CHALLENGE
IN **AMSTERDAM, ATHENS** OR **REMOTE**

BEAT

beat.careers

CPU Architecture

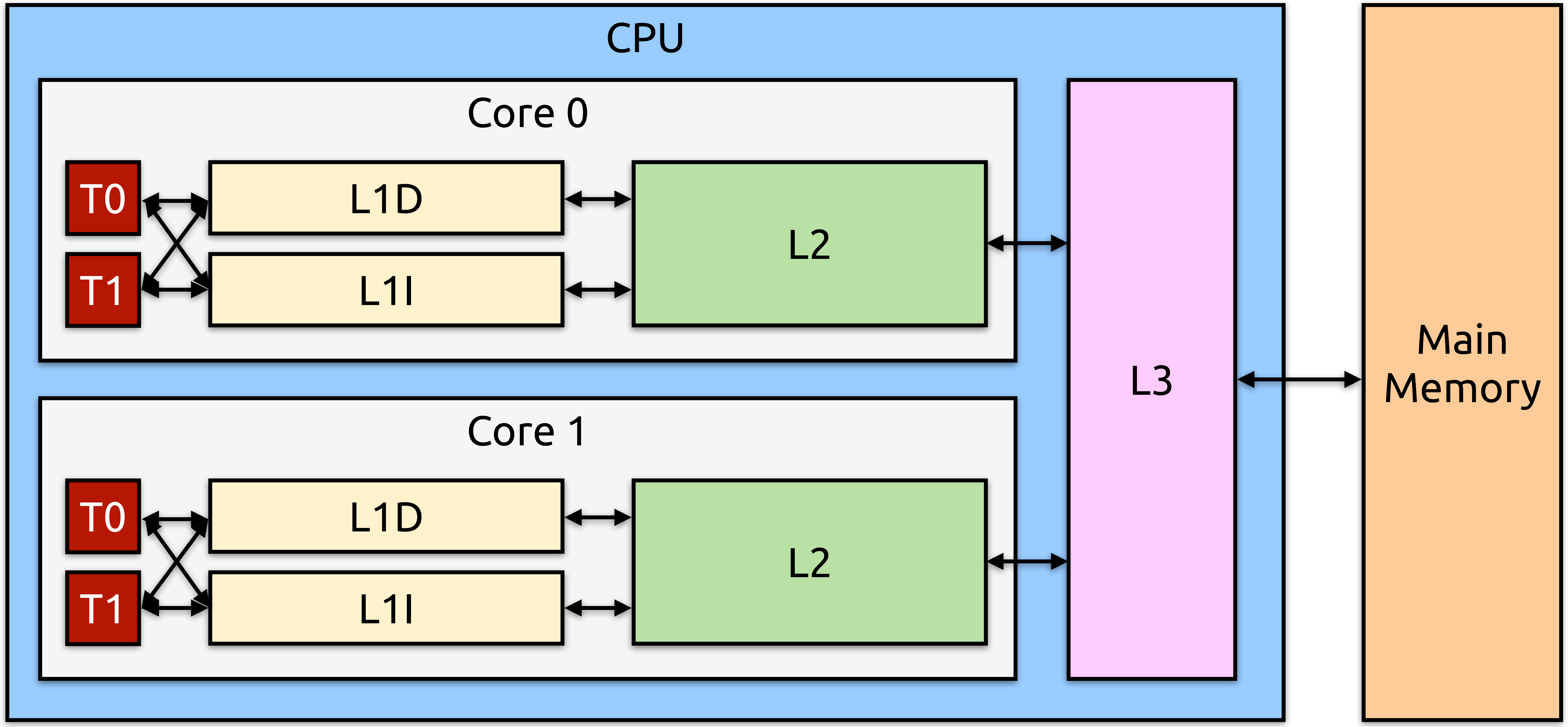
Locality of Reference

Data-Oriented Design

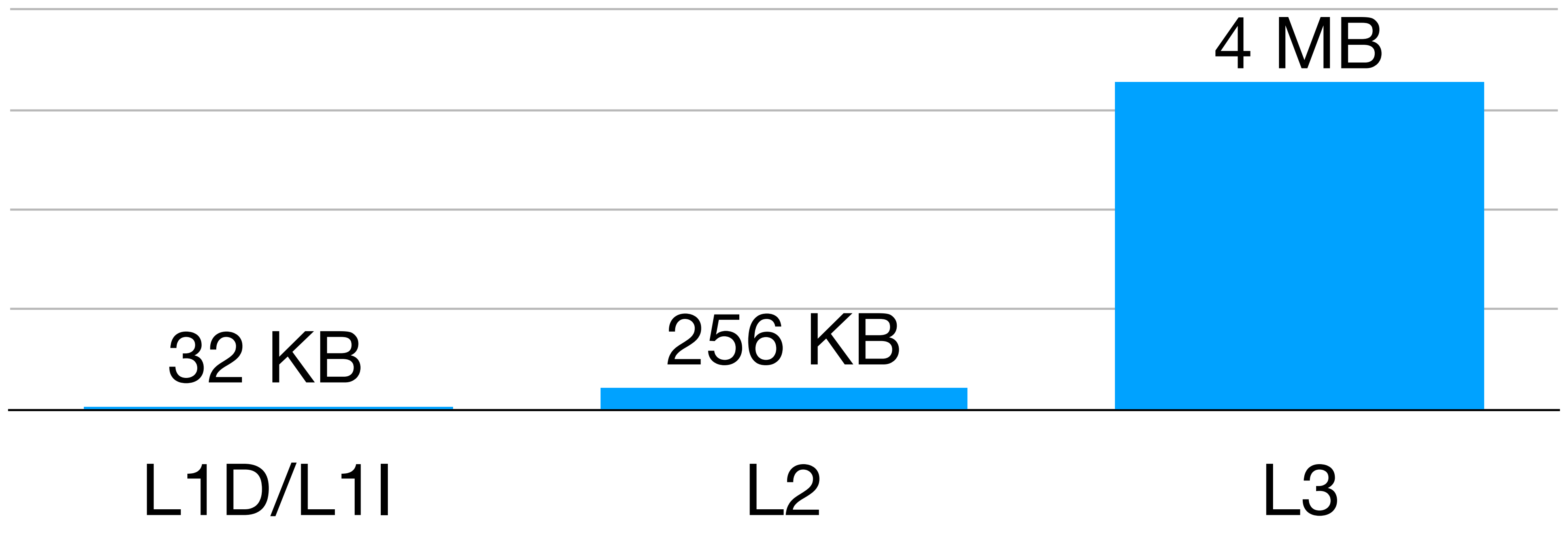
Caching Pitfall

Concurrency

CPU Architecture - Intel Core i5-7300



Cache size



CPU Architecture - Intel Core i5-7300

Thread

L1	~1 ns
L2	~3 times slower than L1
L3	~10 times slower than L1
Main Memory	~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

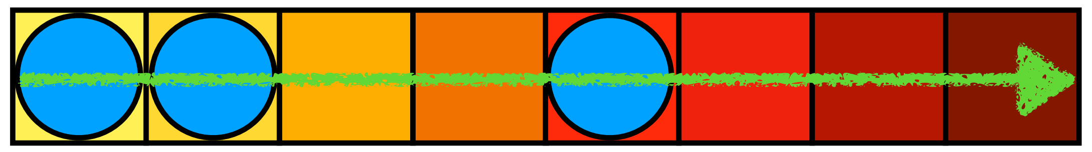
If a **particular memory location** is referenced, it is **likely** that…

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

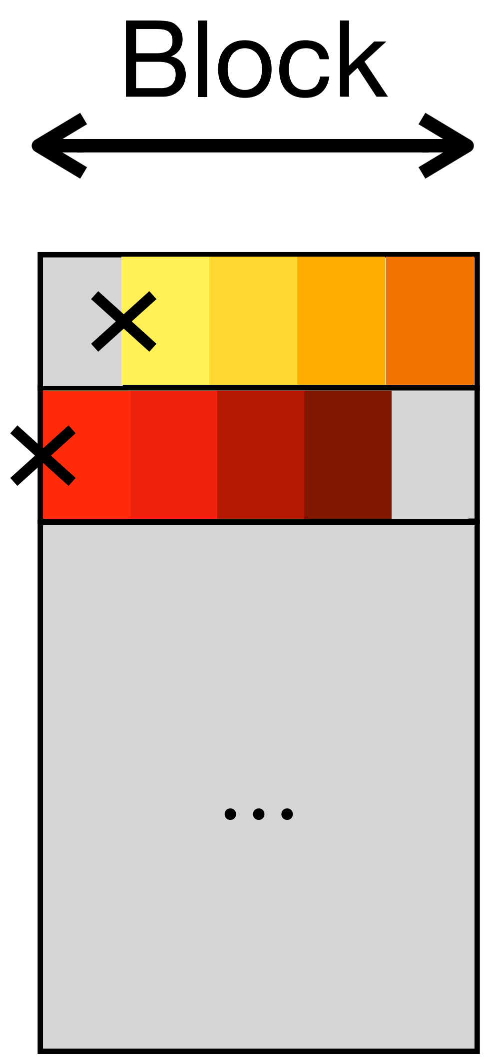
Spatial Locality — **Nearby memory locations** will be referenced in a near future

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

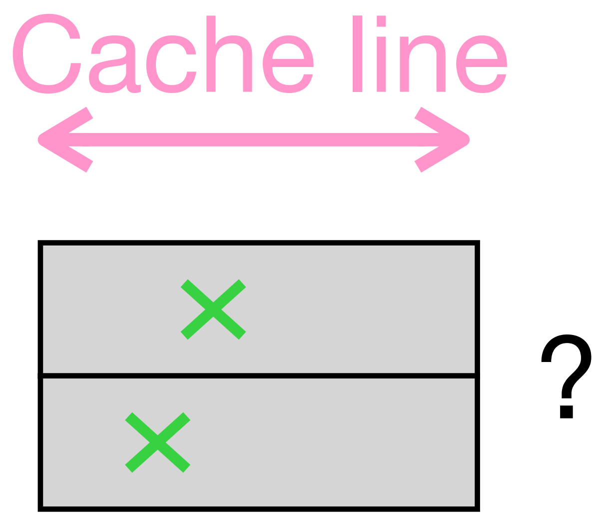
Spatial Locality



Slice (in our heads)



Matrix (main memory)



CPU cache

- 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**

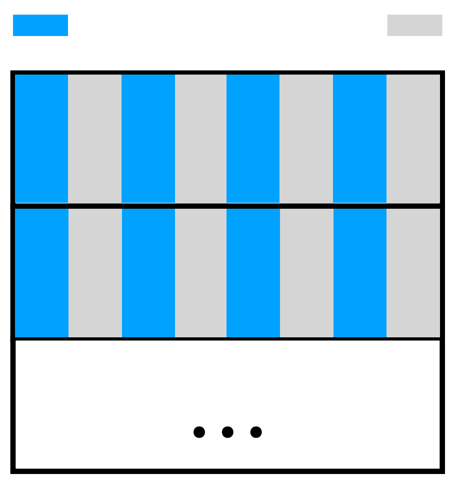
- … and **predictability**

Linked List Iteration

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

Linked list iteration

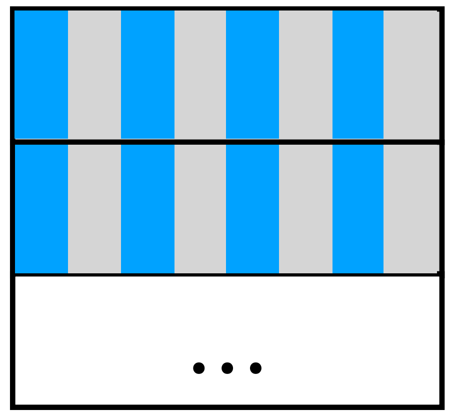
```go
type Node struct {
    Value int64
    Next  *Node
}
```



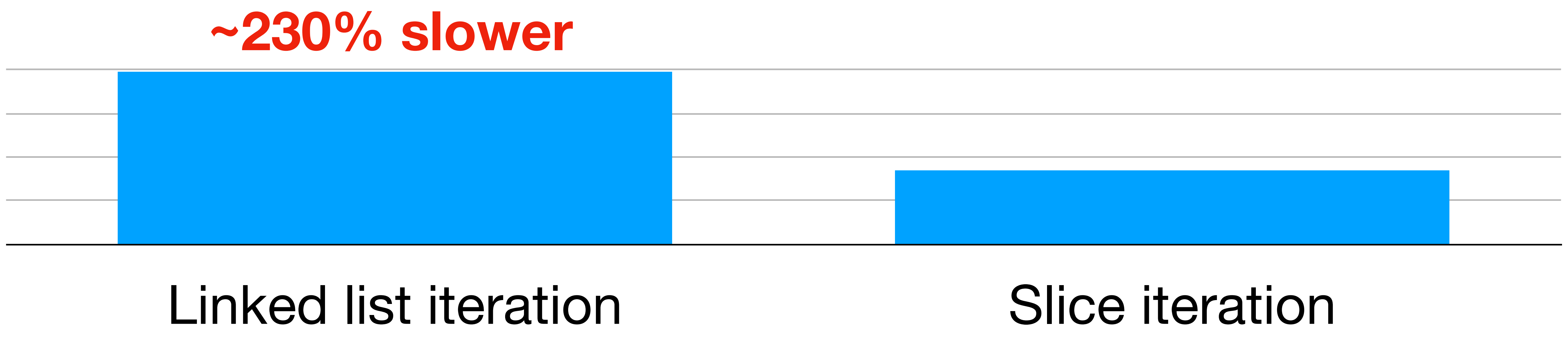
Main memory

Slice iteration: one element out of two

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



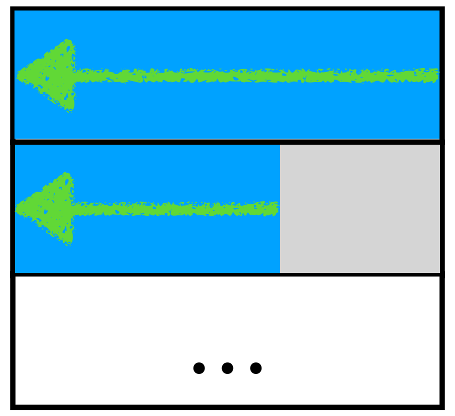
Main memory



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

Backwards Iteration

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



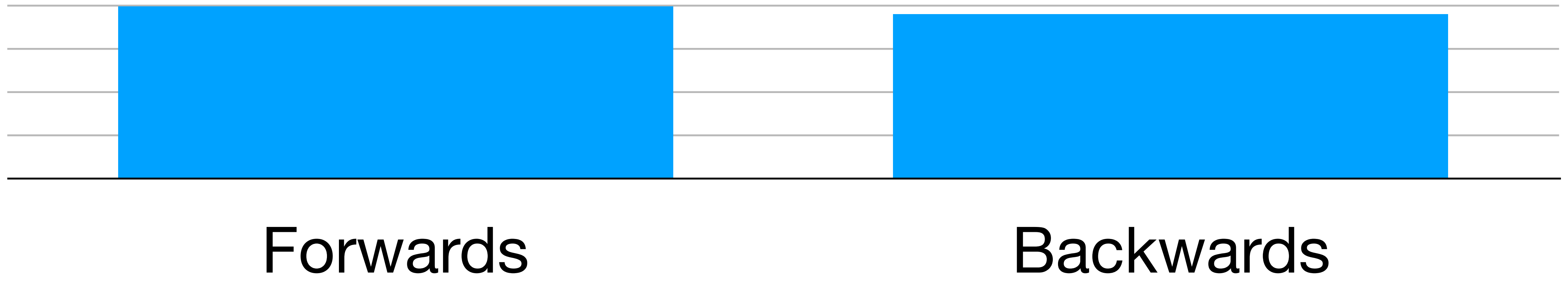
Main memory

Forwards

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

Backwards

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

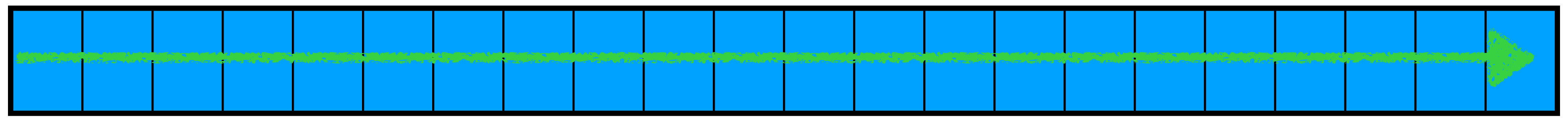


- Spatial locality
- The CPU was able to **predict** that we iterate backwards
- Slightly faster because the bound check is faster

How to Make Things Predictable?

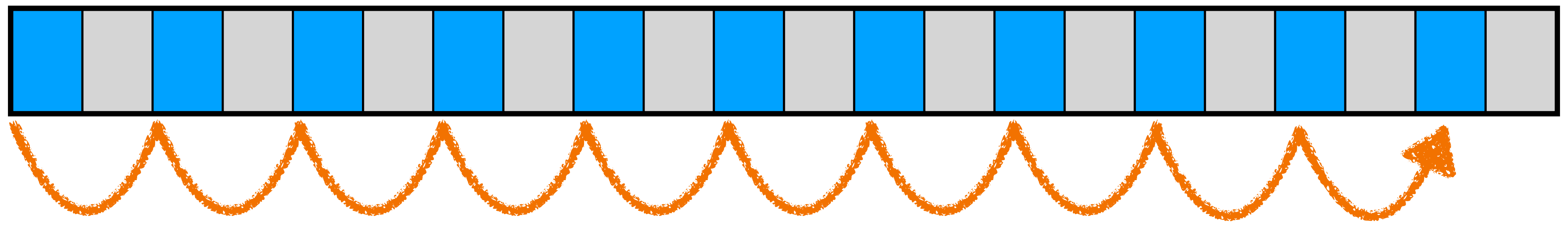
- Striding: how does a CPU work through our data?

  - Unit stride: each element



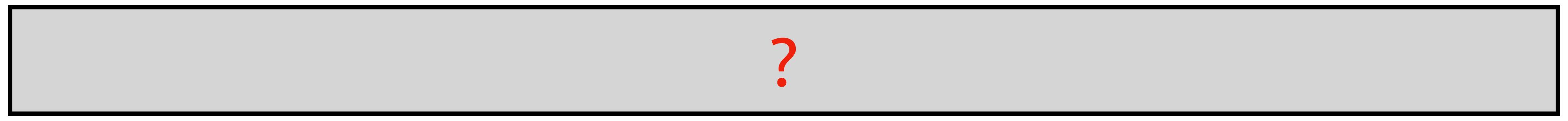
Predictable

  - 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

```go
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
}
```

```go
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
}
```

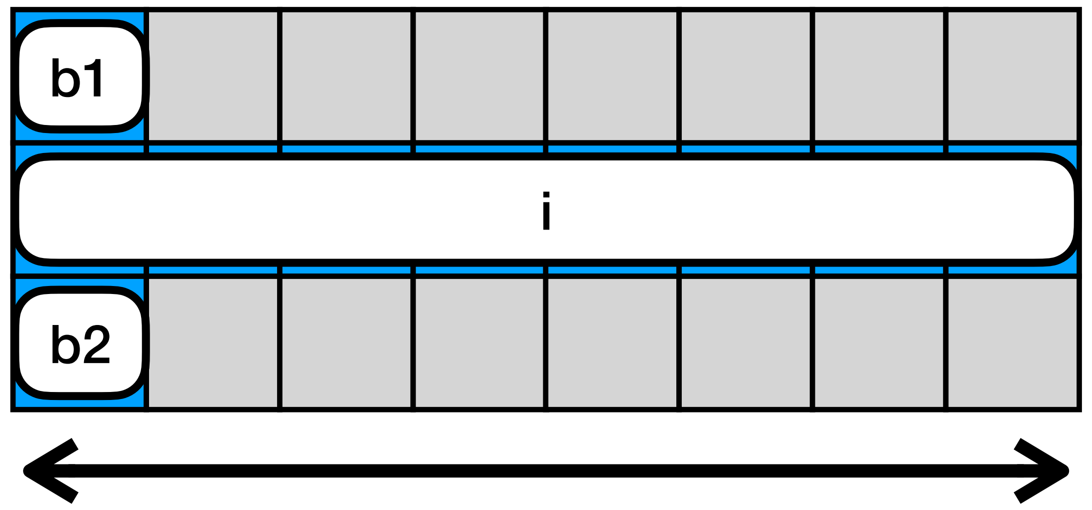
Structure Alignment

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

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

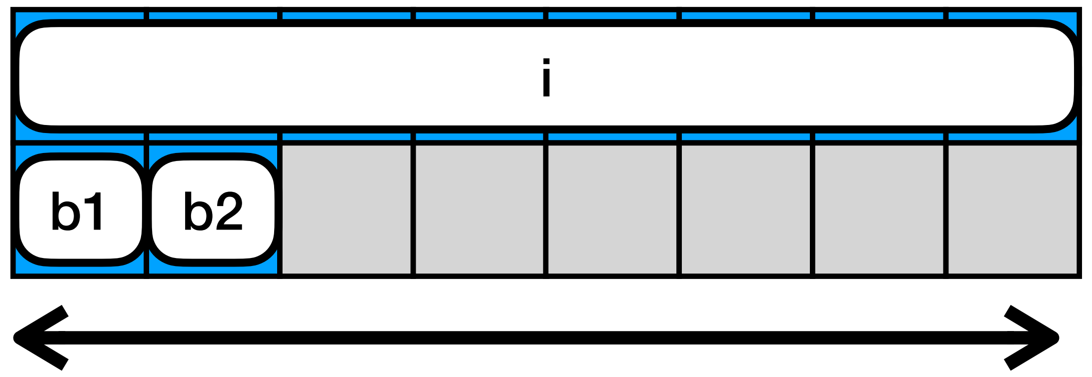
```go
type I2 struct {
    i  int64
    b1 bool
    b2 bool
}
```

24 bytes



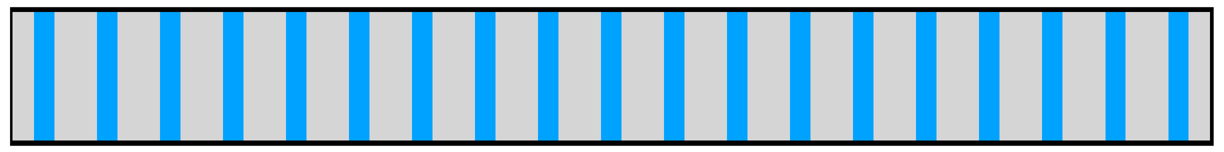
64 bits

Structure alignment

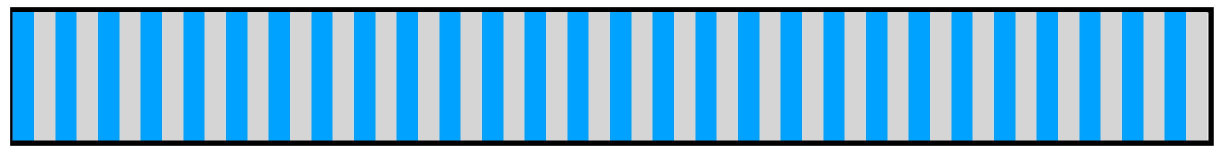


16 bytes

64 bits



In memory



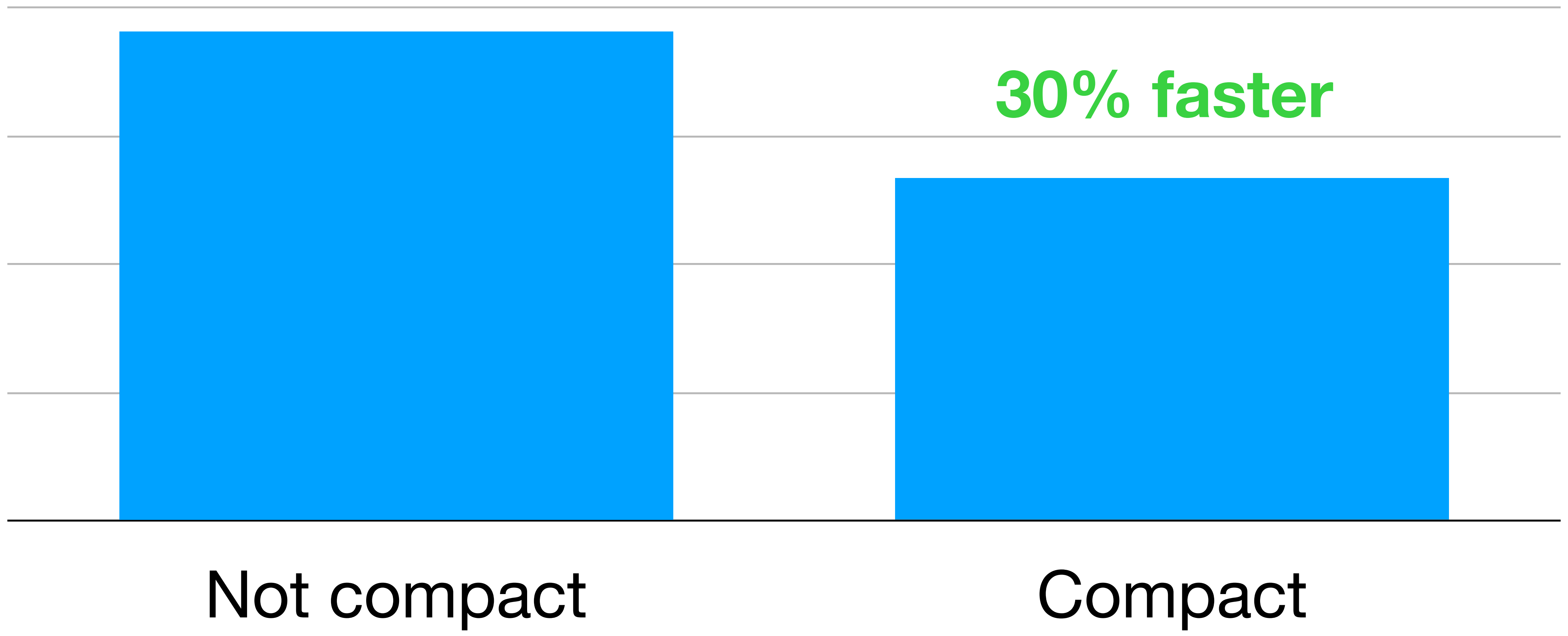
10k slice: 3750 cache lines
(cache line: 64 bytes)

10k slice: 2500 cache lines

Structure Alignment

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

```go
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

```go
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
}
```

```go
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

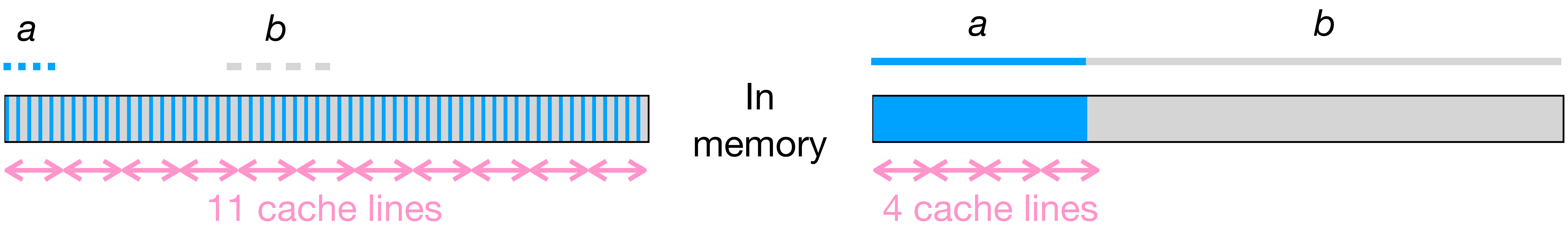
Constant stride

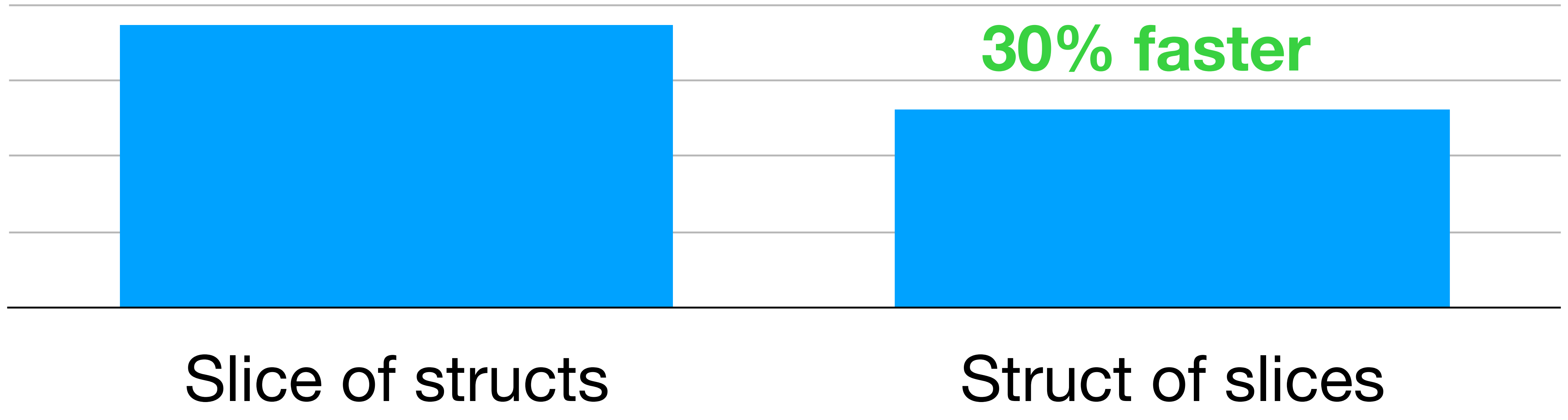
```go
type Struct1 struct {
    a int32
    b int64
}
s := make([]Struct1, it)
```

Struct of slices

Unit stride

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





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 an LZ77 compression

Slice of Structures vs Structure of Slices

Go flate package

```go
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

```go
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

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

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

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

- Metrics?

Xcode Instruments

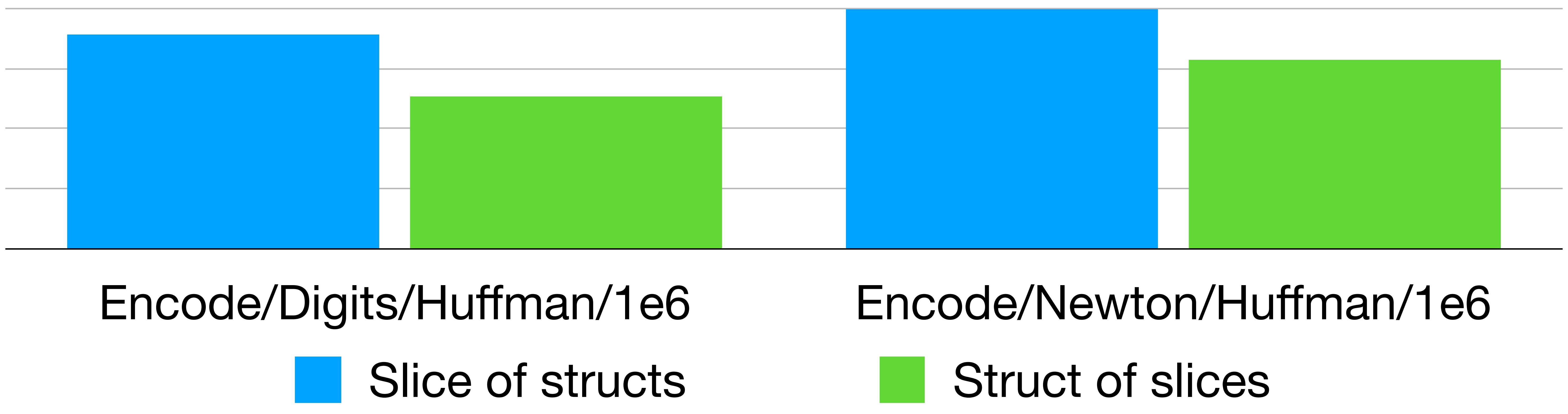
Slice of Structures vs Structure of Slices

Go flate package

```go
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

```go
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

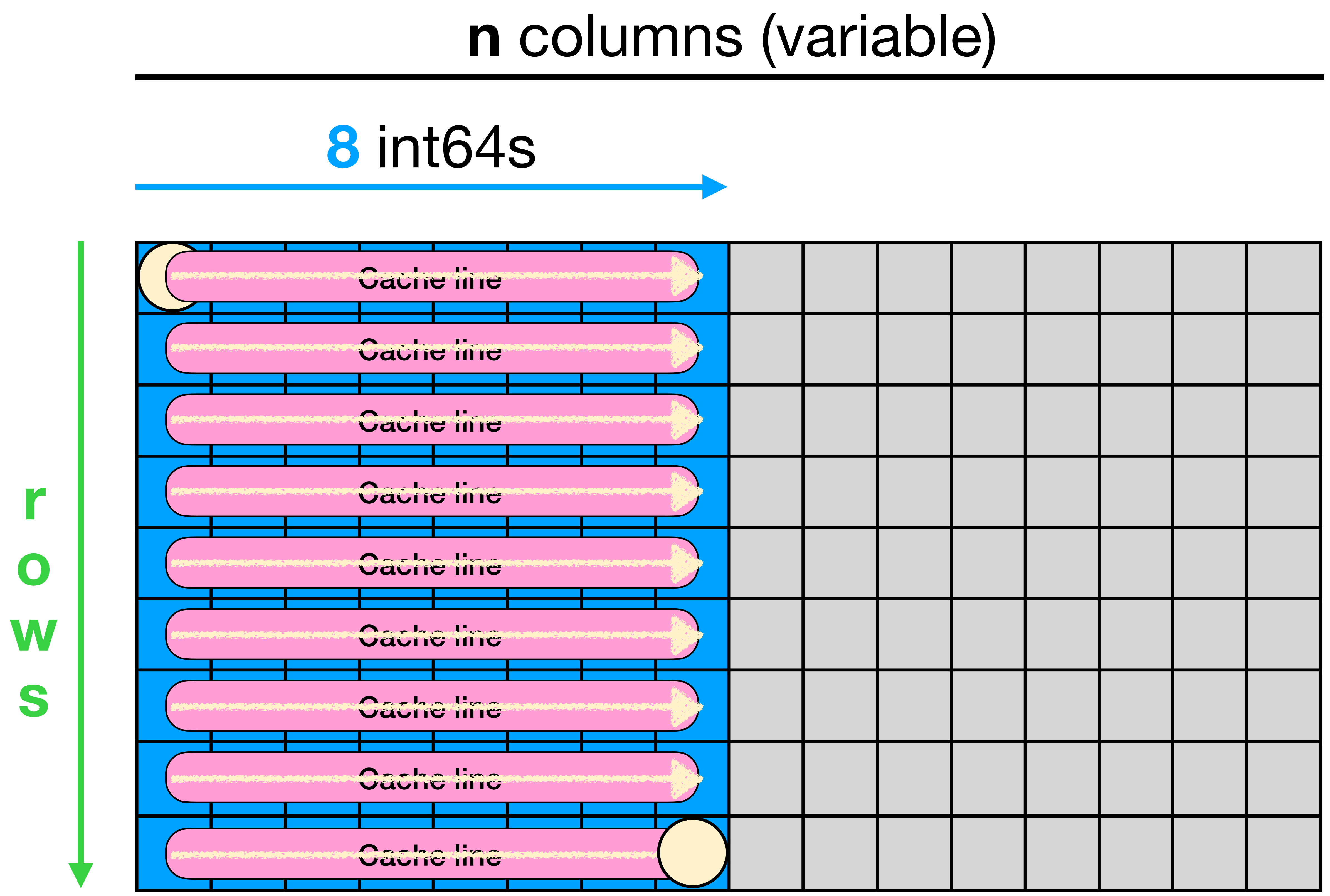
Concurrency

GOPHERCON TURKEY

- 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]
        }
    }
}
```



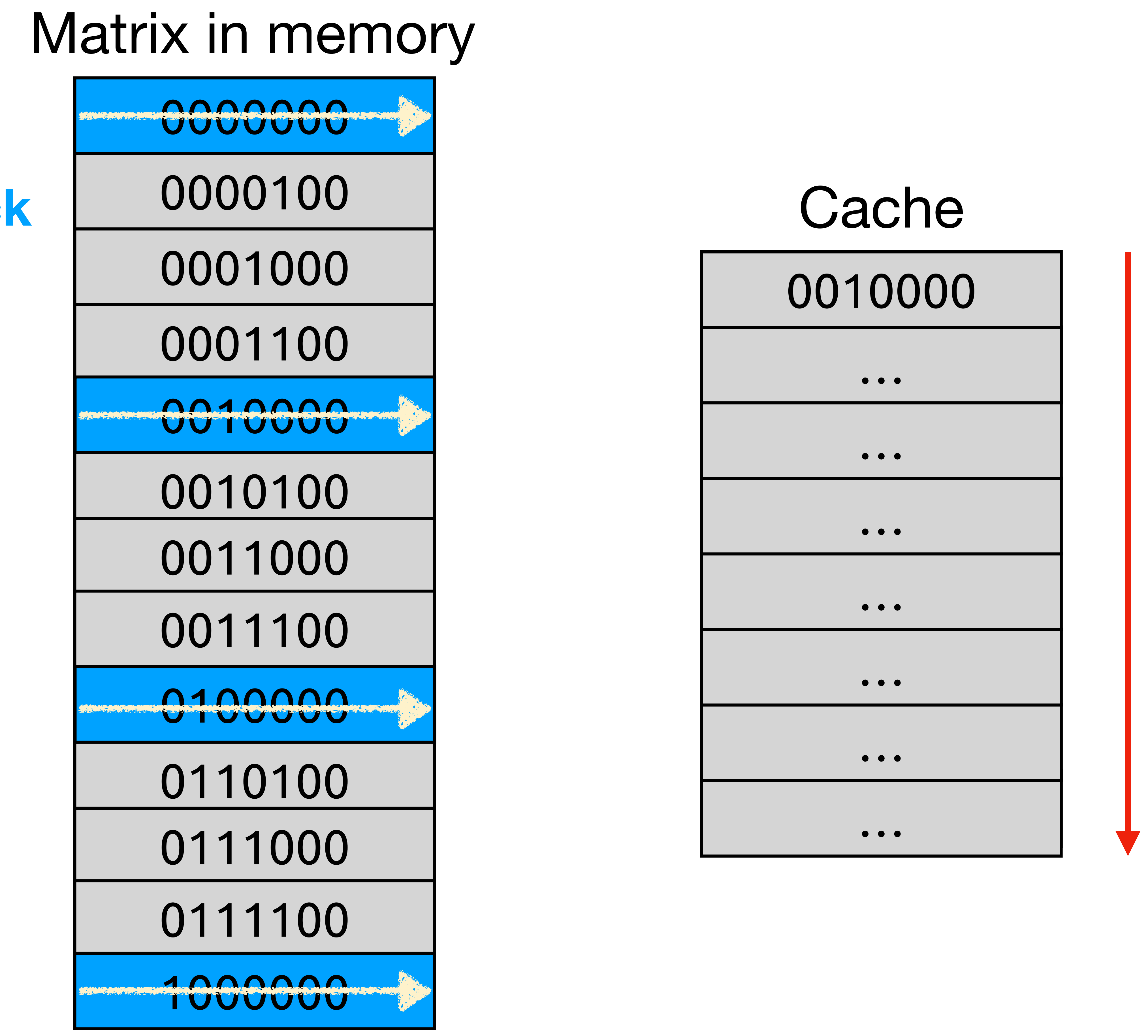
- rows is small enough so that each line should fit in the cache

- 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**



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

E.g Block size: 4 bits

$4 = 2^{2}$

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

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

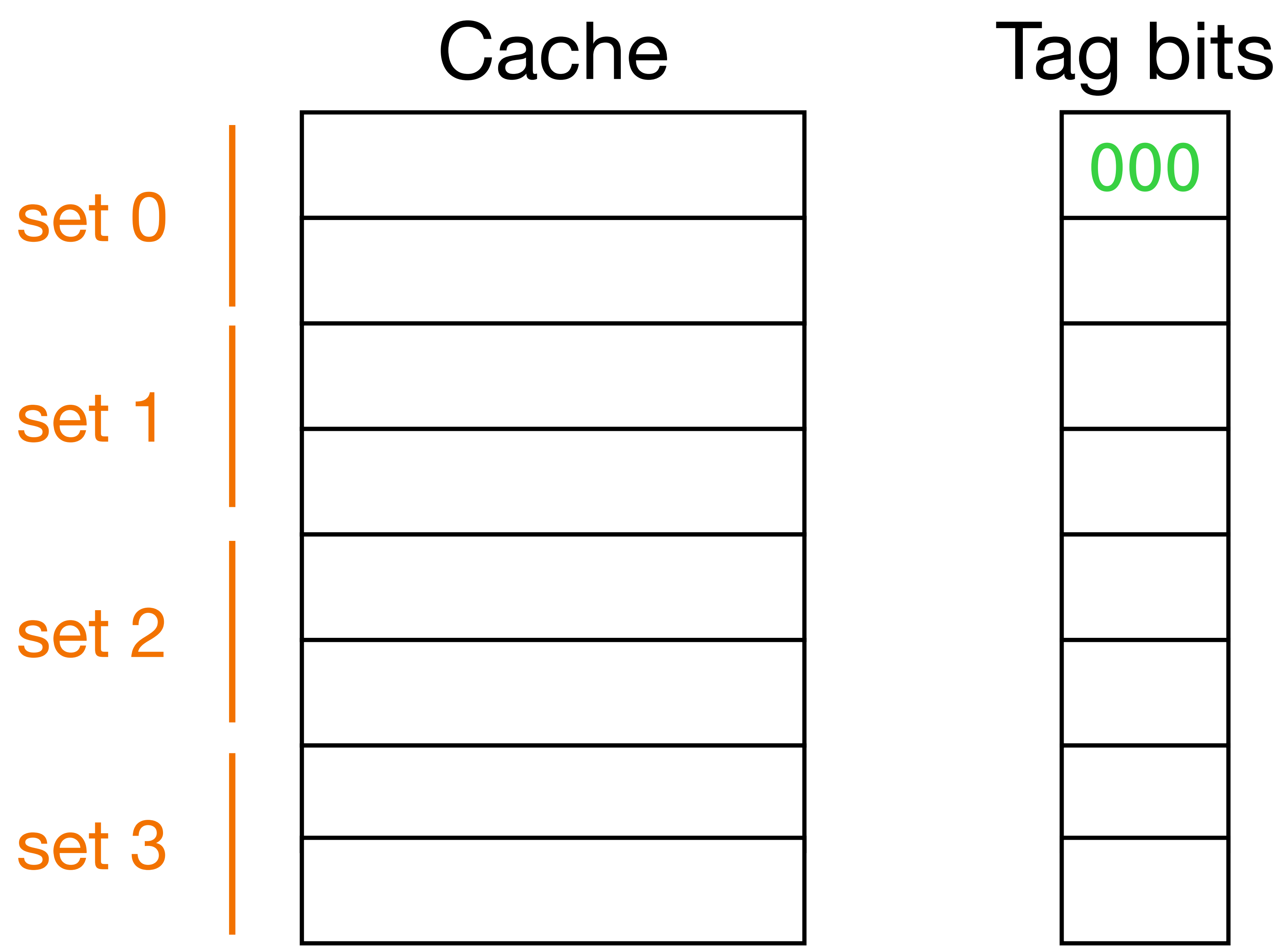
k-way associative cache: k lines per set

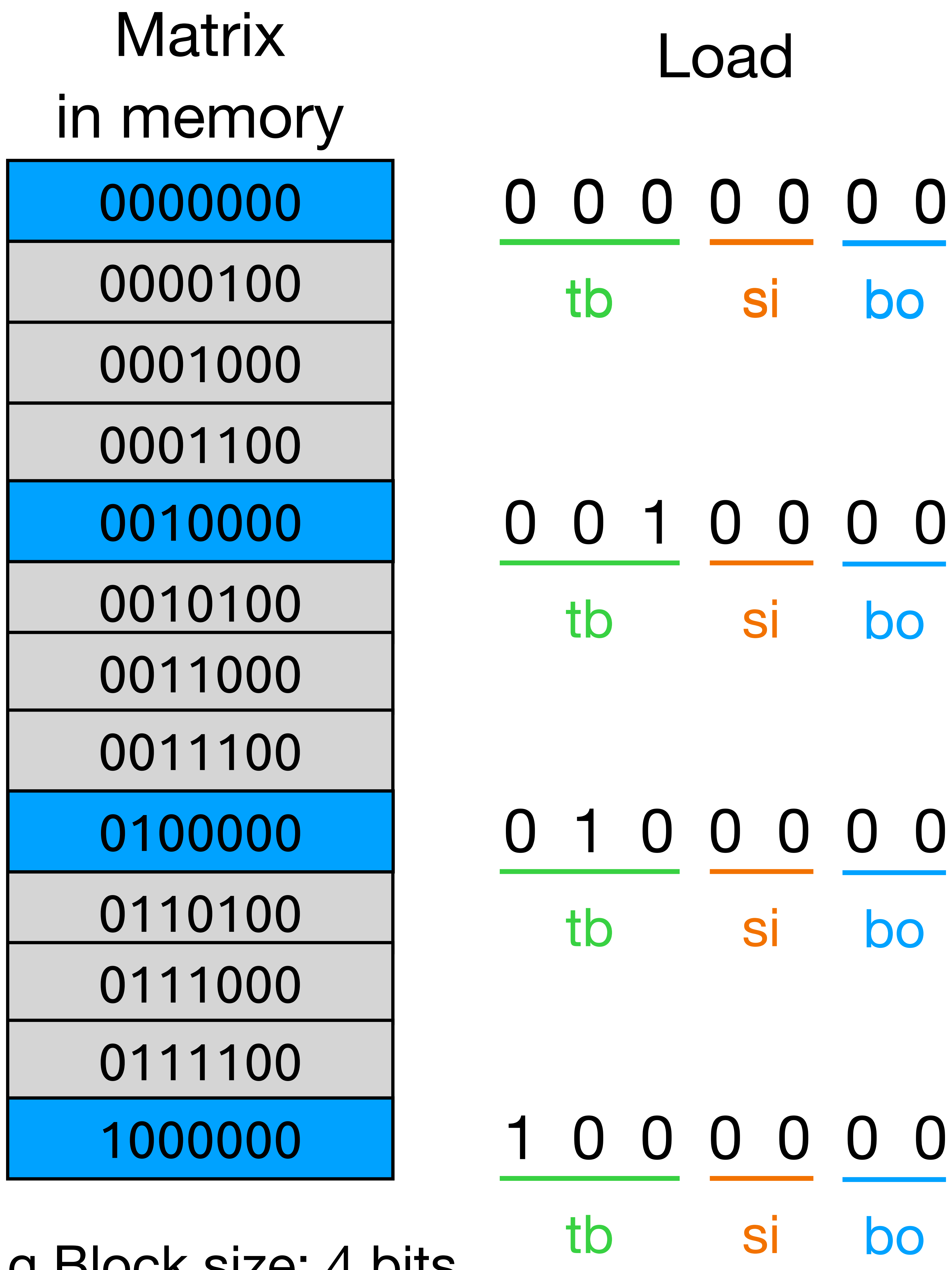
E.g. 8 lines, 2-way associative

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)

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

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

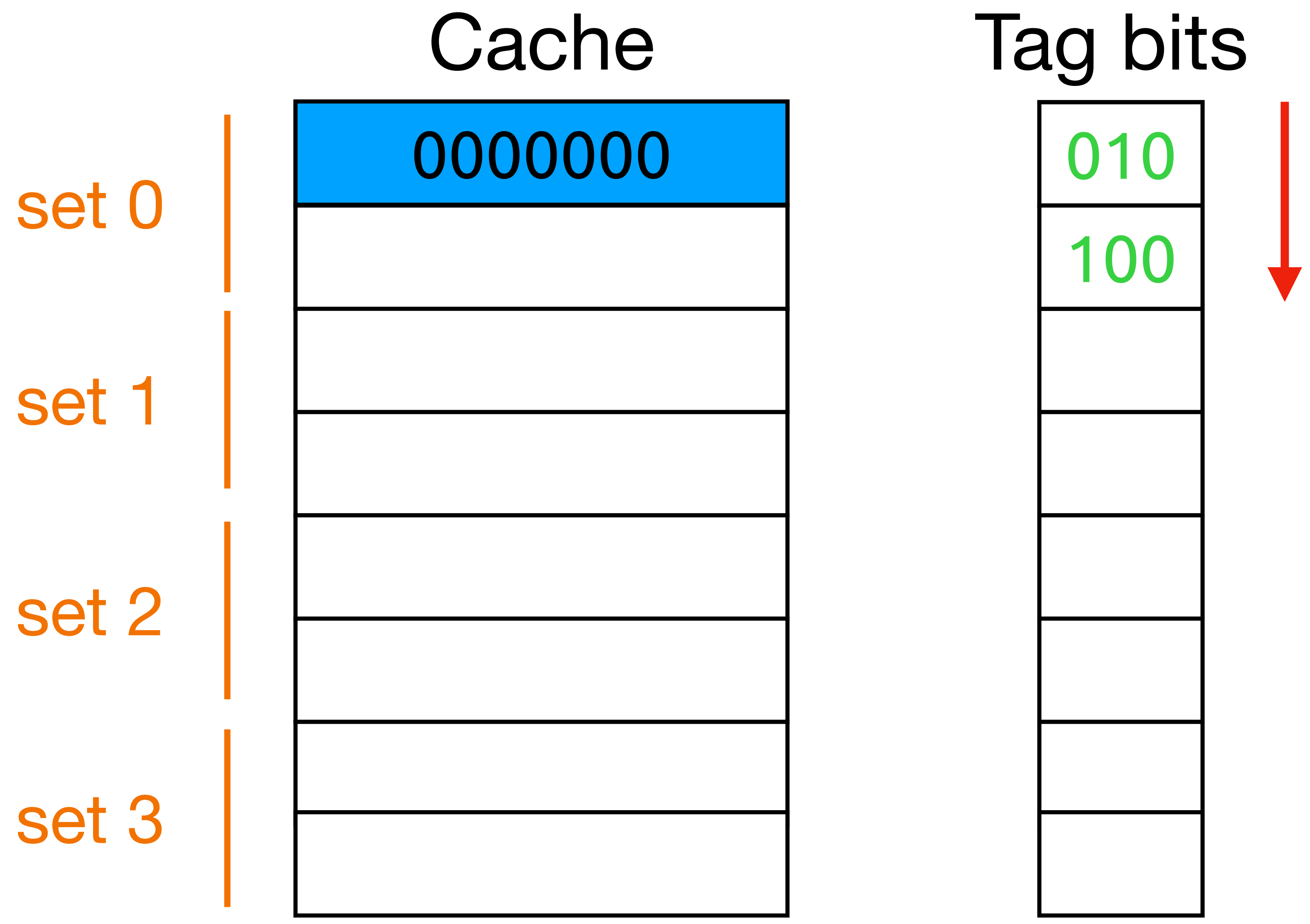
k-way associative cache: k lines per set

E.g. 8 lines, 2-way associative

nb of sets = 8 / 2 = 4

$4 = 2^2$

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



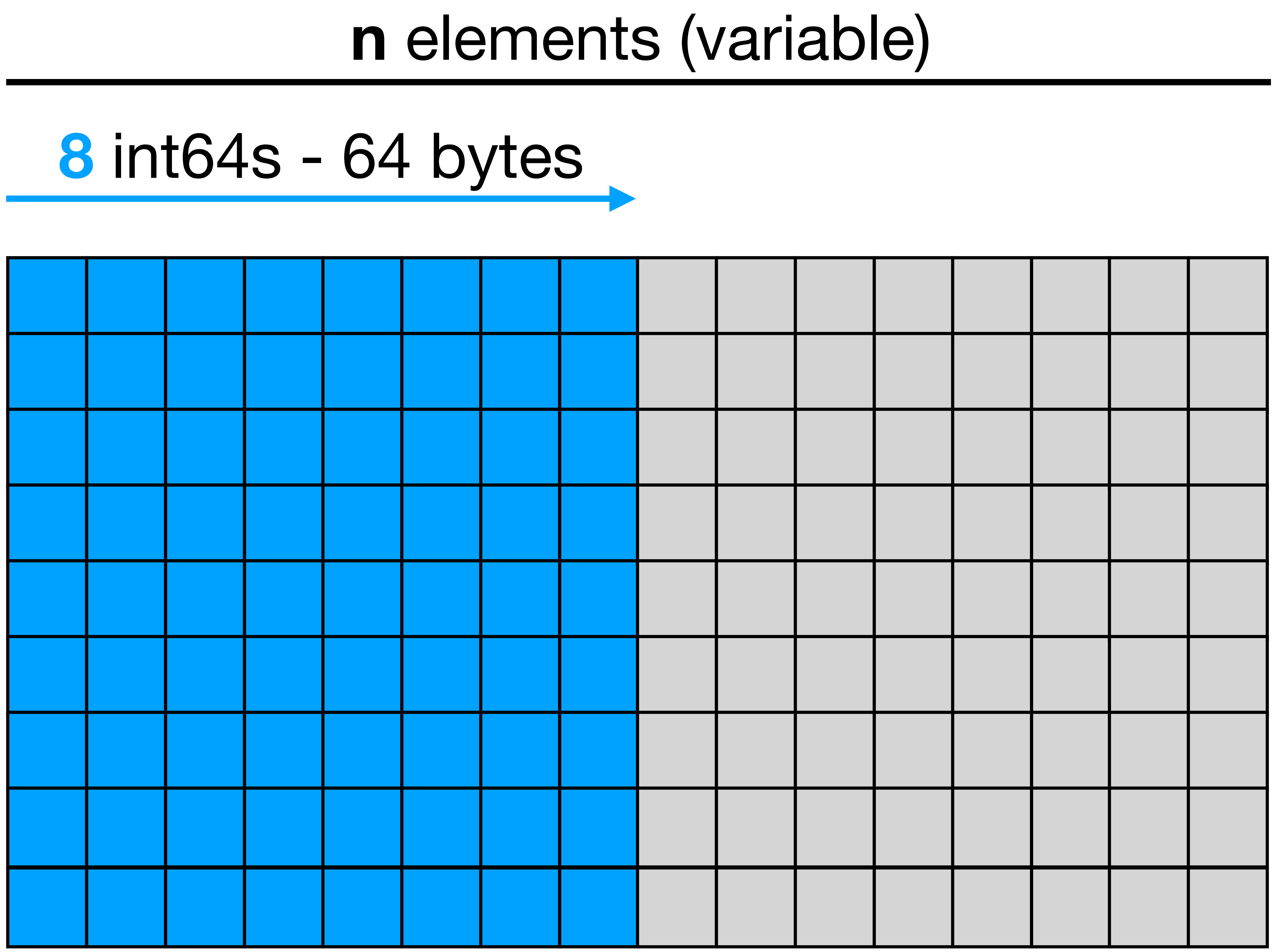
The distribution is not even, we used **only one set**

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

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

GOPHERCON TURKEY

- 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 x 64 = **4 KB**
- We reach a critical stride with n = **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

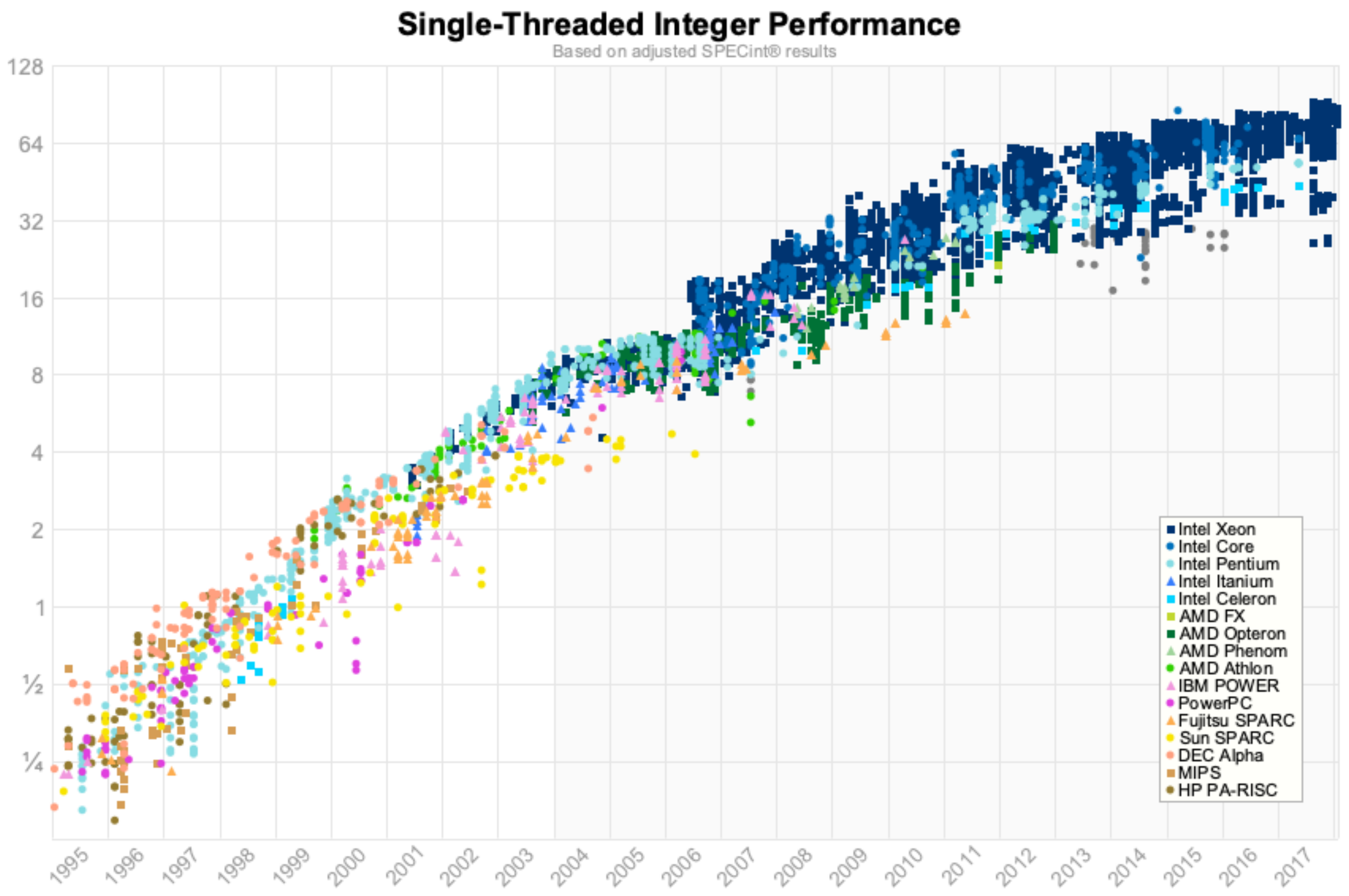
Locality of Reference

Data-Oriented Design

Caching Pitfall

Concurrency

Why Concurrency?

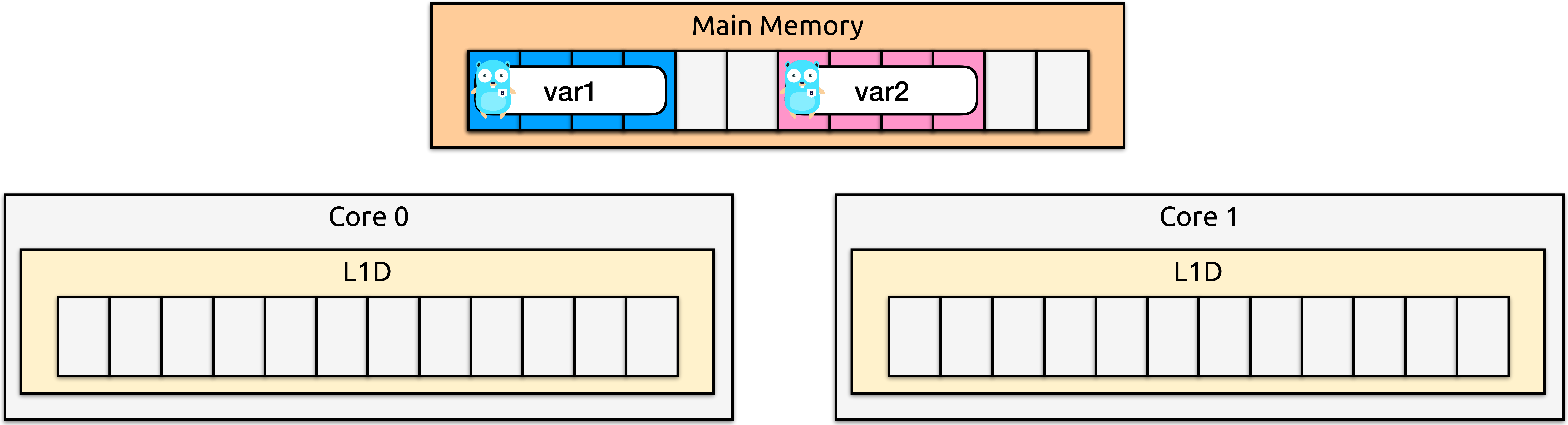


- Instead of focusing on clock speed, vendors focus on **multicores** and **hyperthreading architectures**
- The free lunch is over - Herb Sutter, 2005
- We cannot rely solely on the **hardware** to make our programs faster

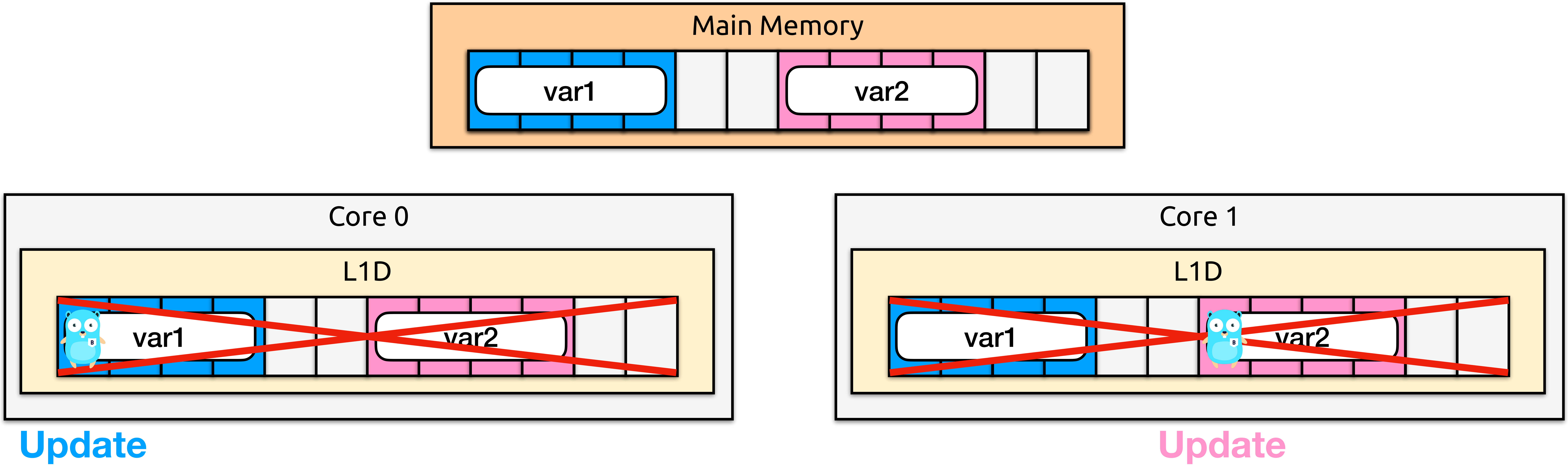
Concurrency is the next major revolution in how we write software

```go
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)



- 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**

```go
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**

GOPHERCON TURKEY

False Sharing

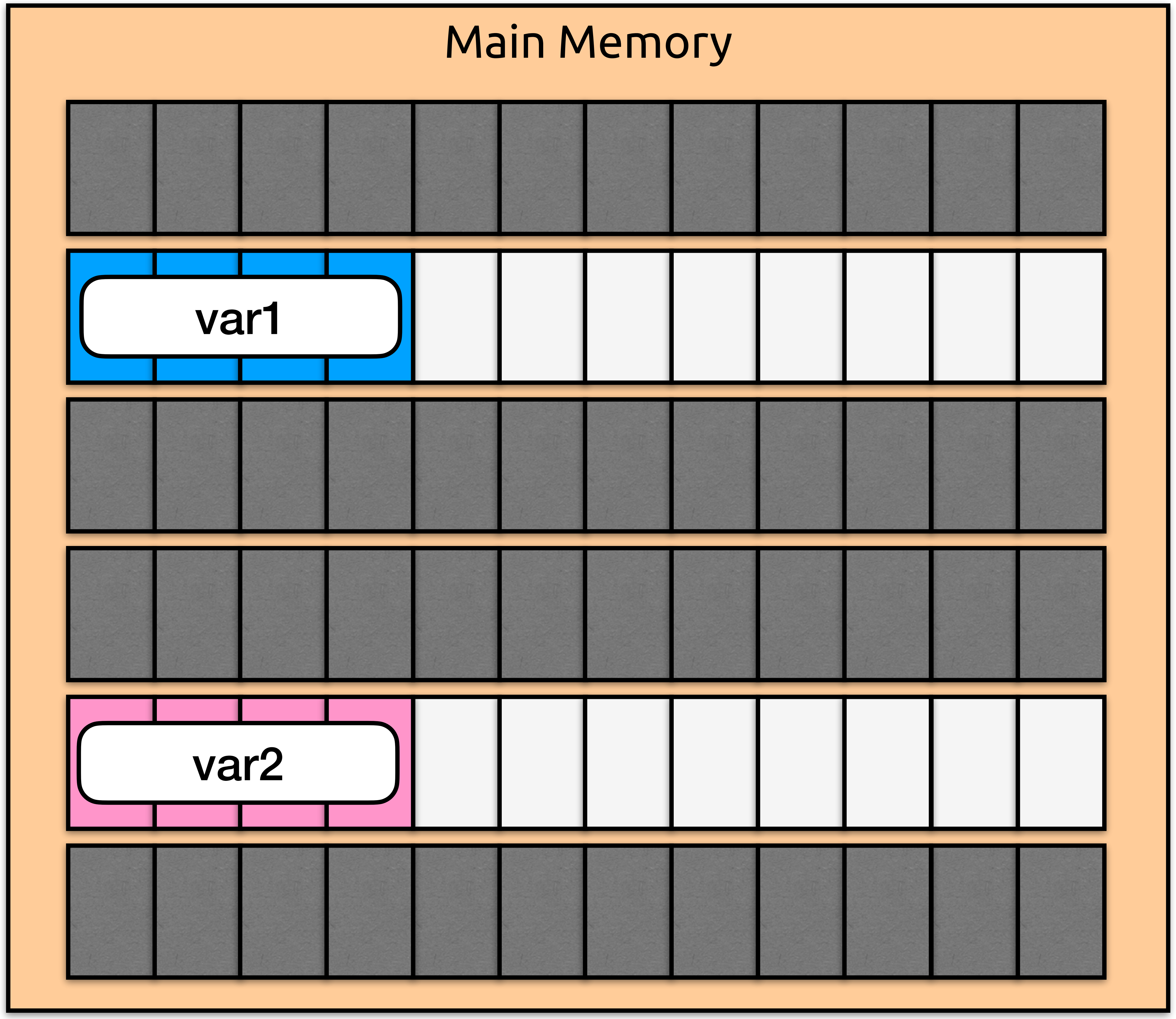
- How to prevent false sharing?

- Solution 1:
  Do not communicate by sharing memory; instead, share memory by **communicating**


```go
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**



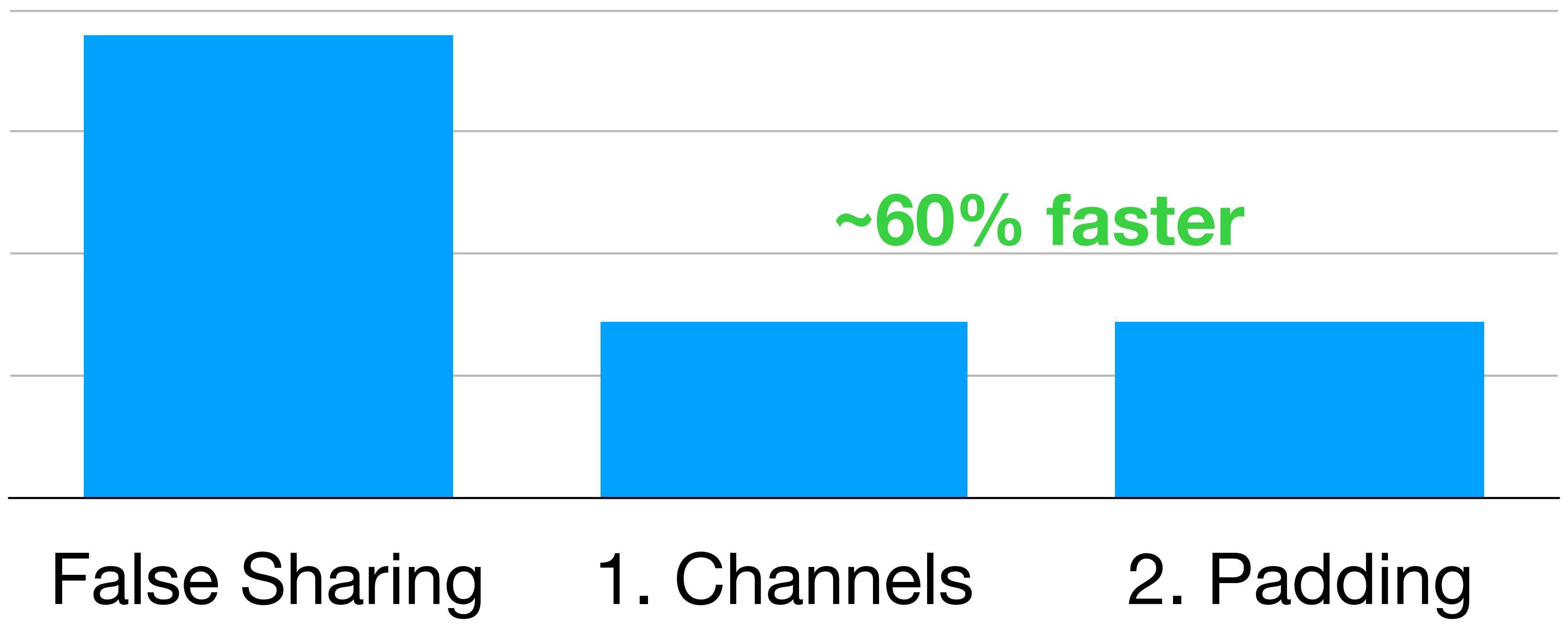

```go
type PaddedStruct struct {
    _ cpu.CacheLinePad  // 64 bytes
    n int
    _ cpu.CacheLinePad  // 64 bytes
}
```

```go
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
    }
}
```

GOPHERCON TURKEY

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

teivah