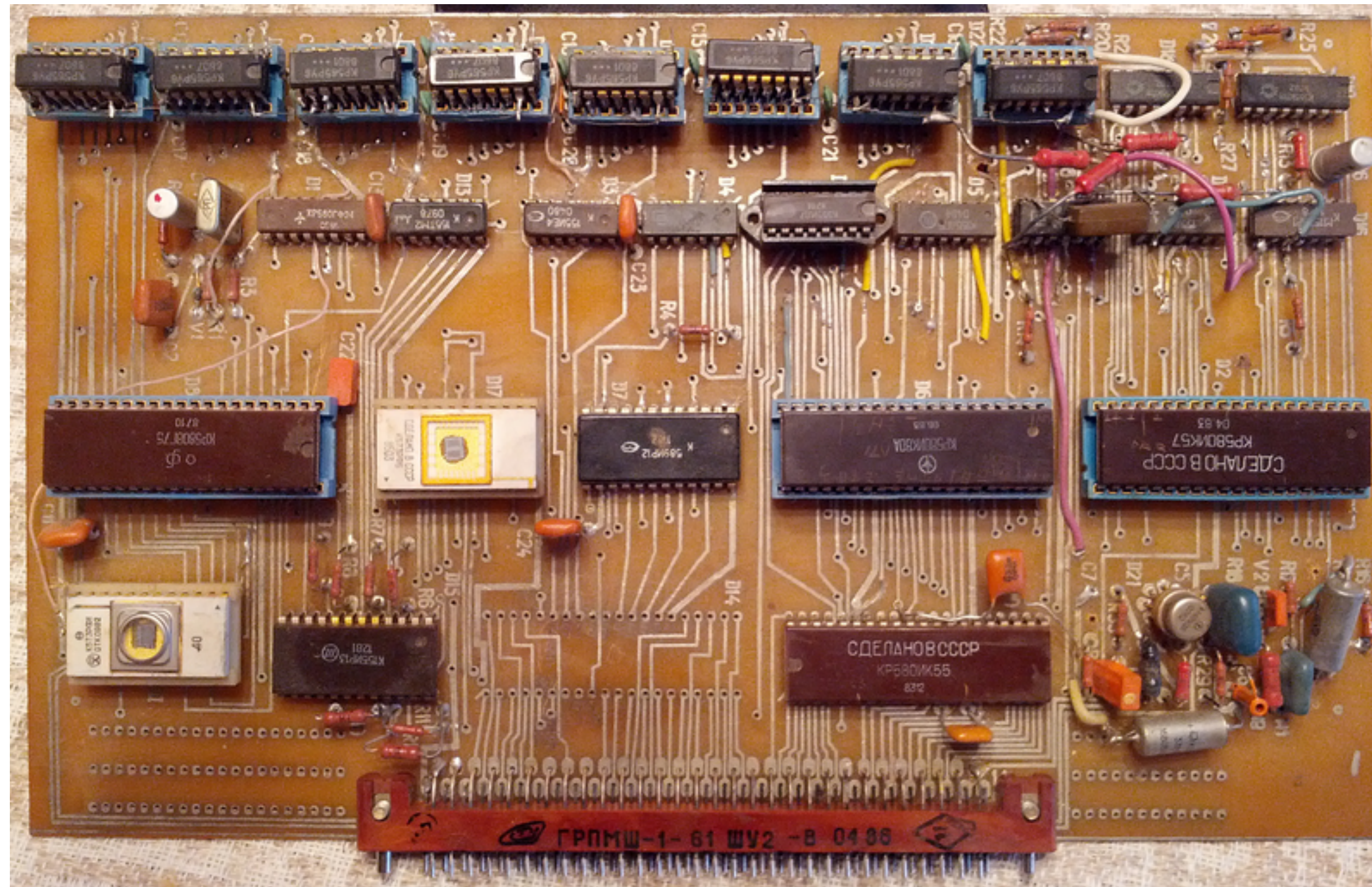


# Performance

## Physics Optimization Strategies

Sergiy Migdalskiy  
Valve

# My first computer



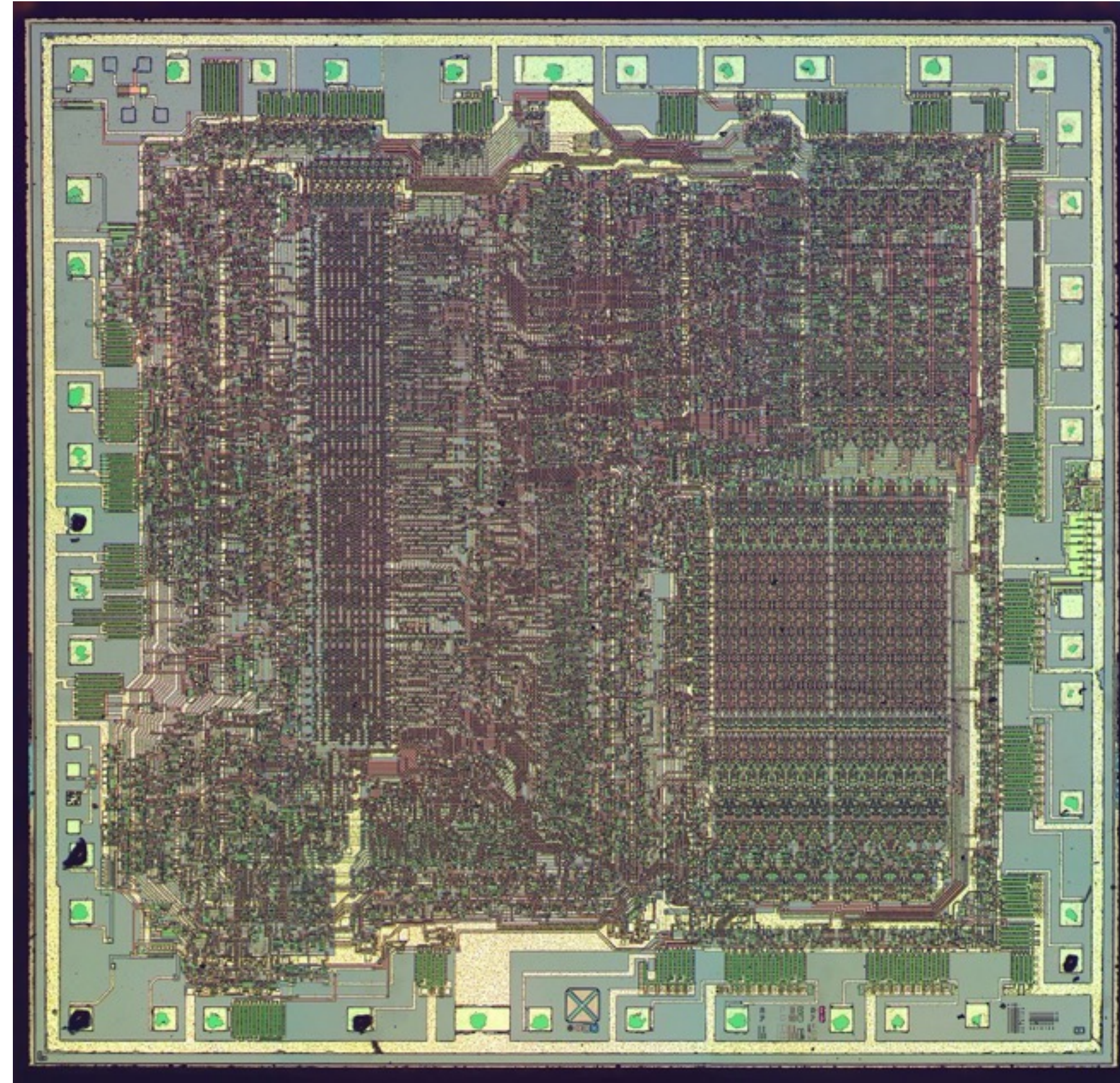
Do you remember yours?

# *User-friendly* computer



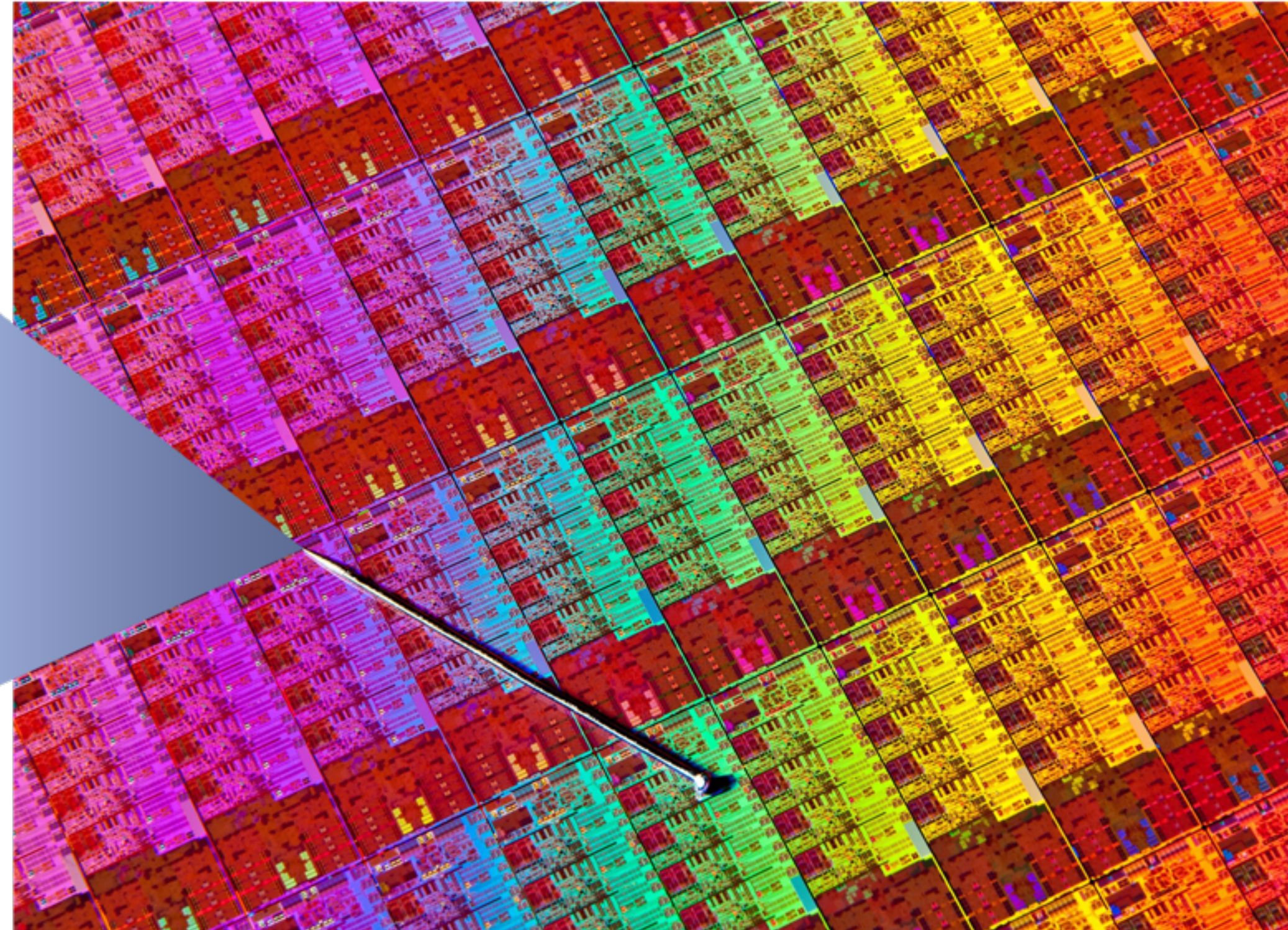
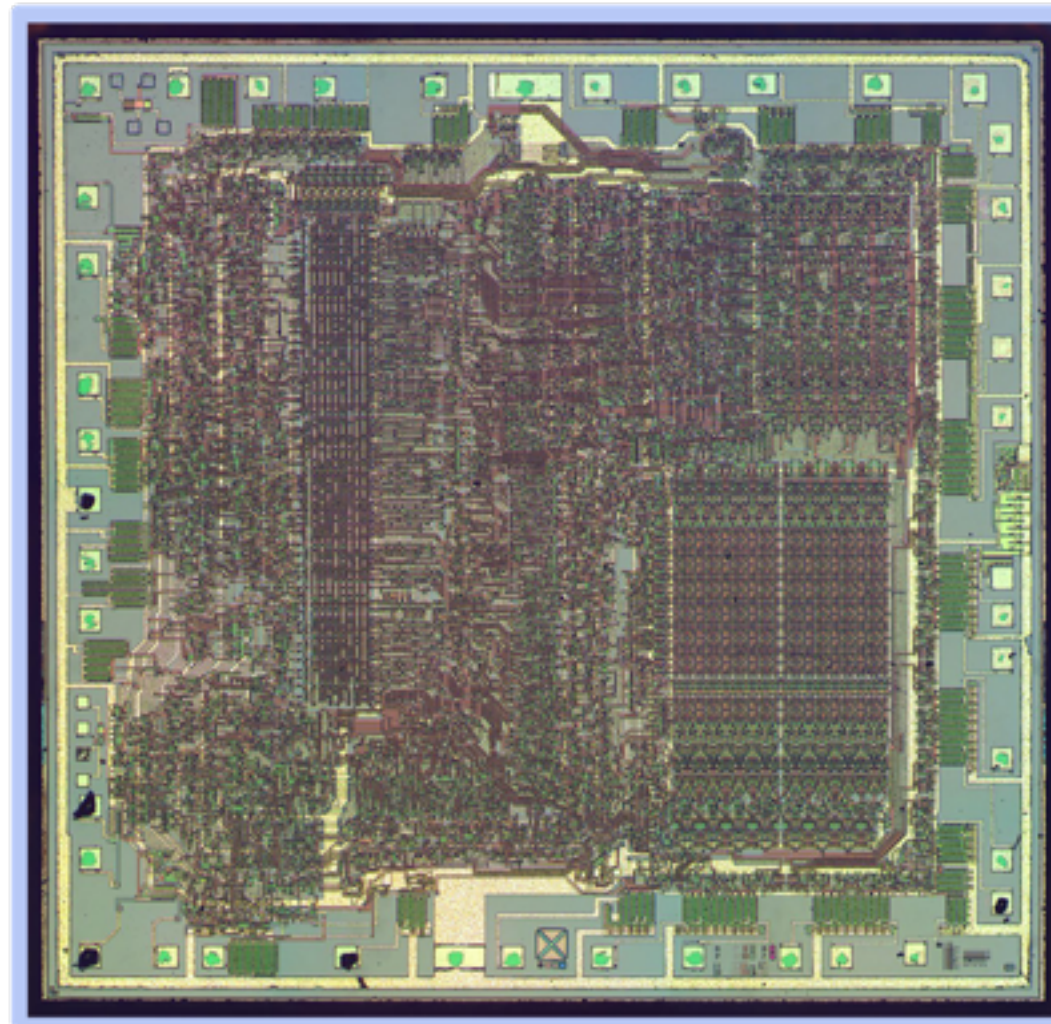
ZX Spectrum

# 8-bit CPU



A CPU most fondly remembered

# Back Then vs Now



# Golden Oldies

```

Float24Mul:
;BHL*CDE ->
AHL
  ld a,b \ xor c
  and 80h
  push af
  sla b \ sra b
  sla c \ sra c
  ld a,b
  add a,c
  pop bc
  jp m,$+10
  cp 64
  jr nc,SetInf-2
  jp $+7
  cp -64
  jr c,SetInf-2
;B has the right
sign
  and 7Fh
  or b
  push af
  ld b,h
  ld c,l

```

```

  call
BC_Times_DE
  bit 7,l
  ld l,h
  ld h,b
  jr z,$+3
  inc hl
  pop af
  ret
BC_Times_DE:
;BHLA is the
result
  ld a,b
  or a
  ld hl,0
  ld b,h
;1
  add a,a
  jr nc,$+4
  ld h,d
  ld l,e
;2
  add hl,hl
  rla

```

```

  jr nc,$+4
  add hl,de
  adc a,b
;227+10b-7p
  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,b

  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,b

  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,b

```

```

  jr nc,$+4
  add hl,de
  adc a,b

  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,b

  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,b

;===
;AHL is the result
of B*DE*256
  push hl
  ld h,b
  ld l,b
  ld b,a
  ld a,c

```

```

  ld c,h
;1
  add a,a
  jr nc,$+4
  ld h,d
  ld l,e
;2
  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,c
;227+10b-7p
  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,c

  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,c

```

```

  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,c

  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,c

  add hl,hl
  rla
  jr nc,$+4
  add hl,de
  adc a,c

```

```

  pop de
;Now
BDE*256+AHL
  ld c,a
  ld a,l
  ld l,h
  ld h,c
  add hl,de
  ret nc
  inc b
;BHLA is the 32-
bit result
  ret

```

# Brave New World

mulss	1	float
mulps	4	floats
vmulps	8	floats

# I've got the Power!

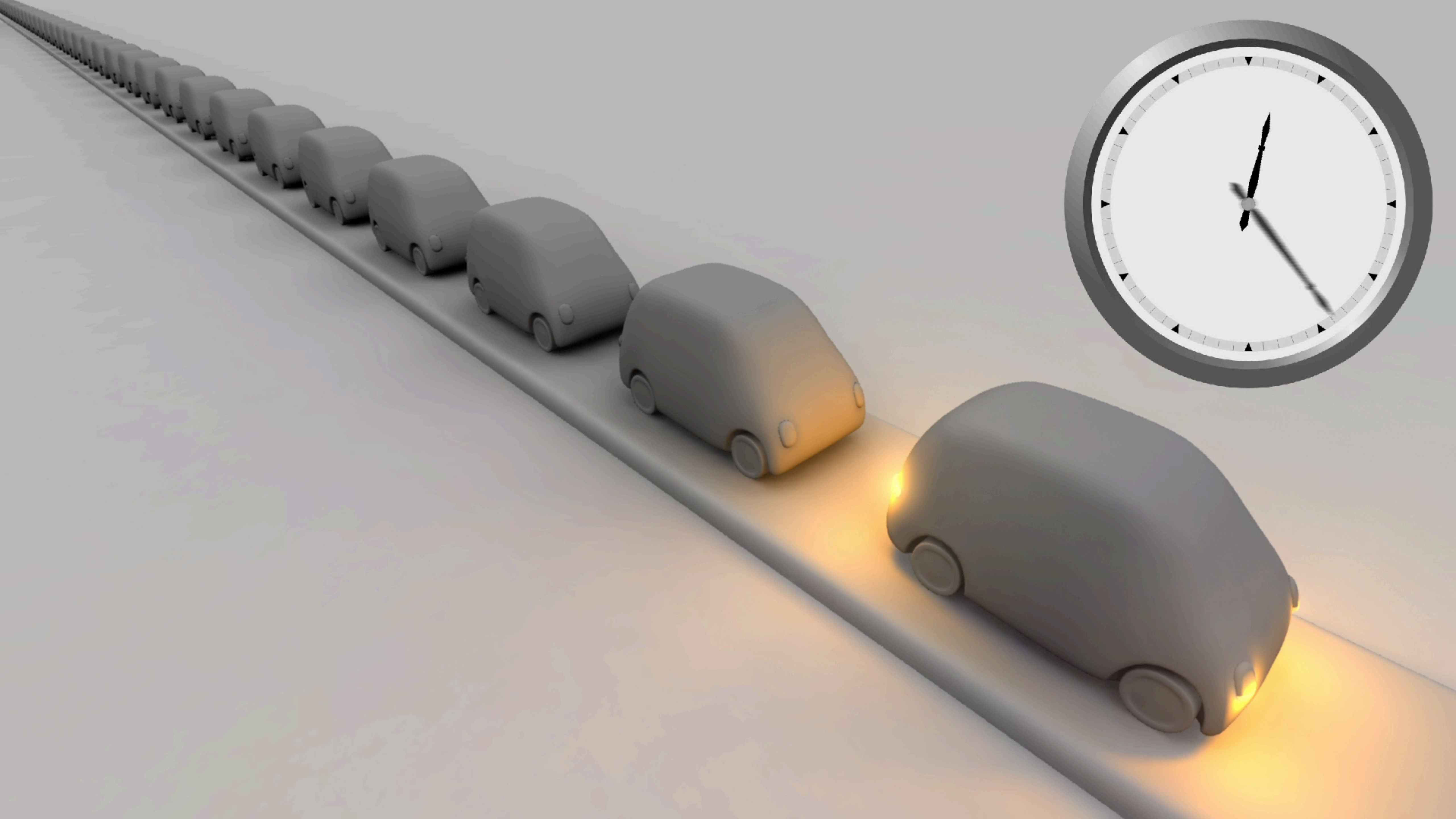
```
virtual void Foo();
```

...

```
pObject->Foo();
```

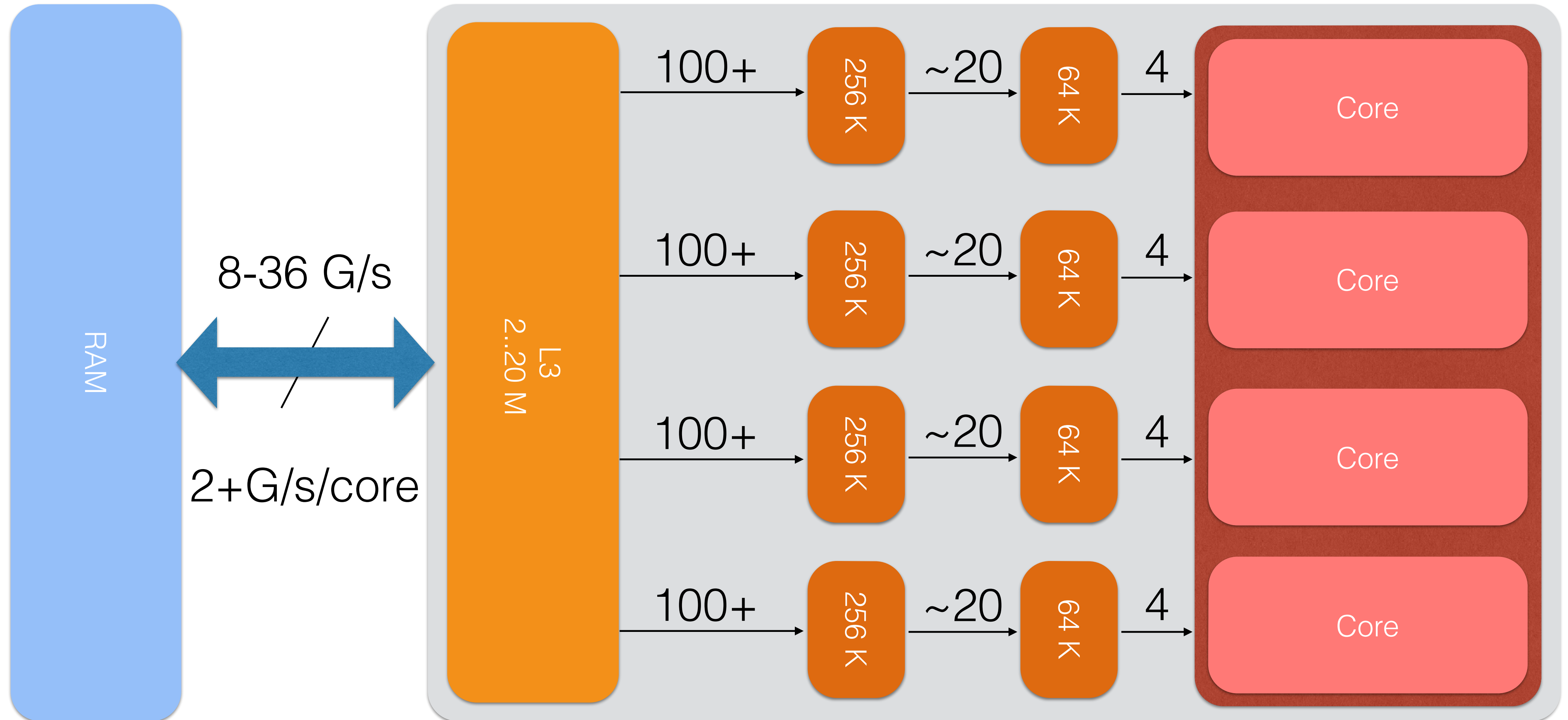
- Fetch object pointer
- Fetch Vtable pointer
- Fetch Vtable entry

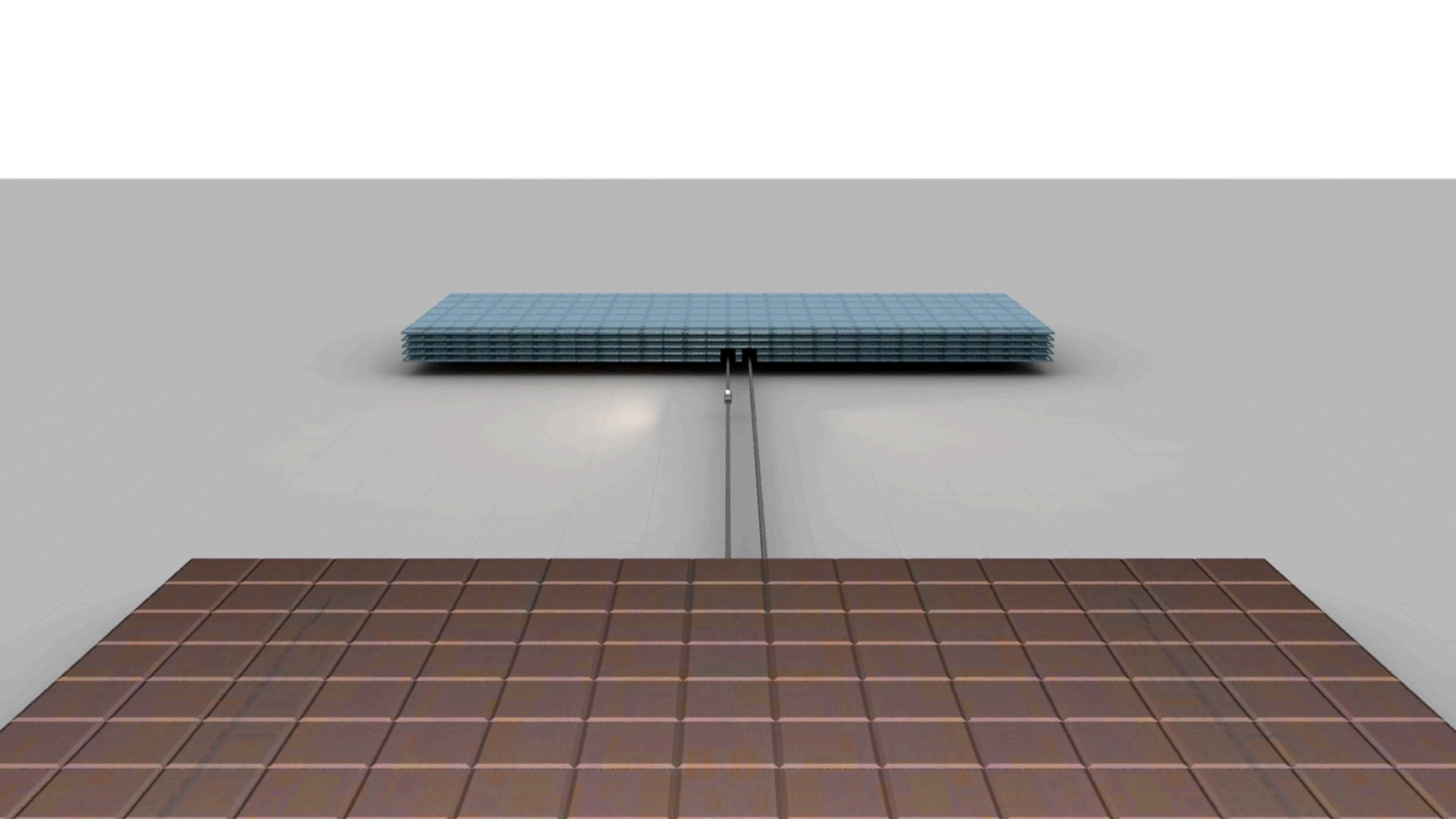




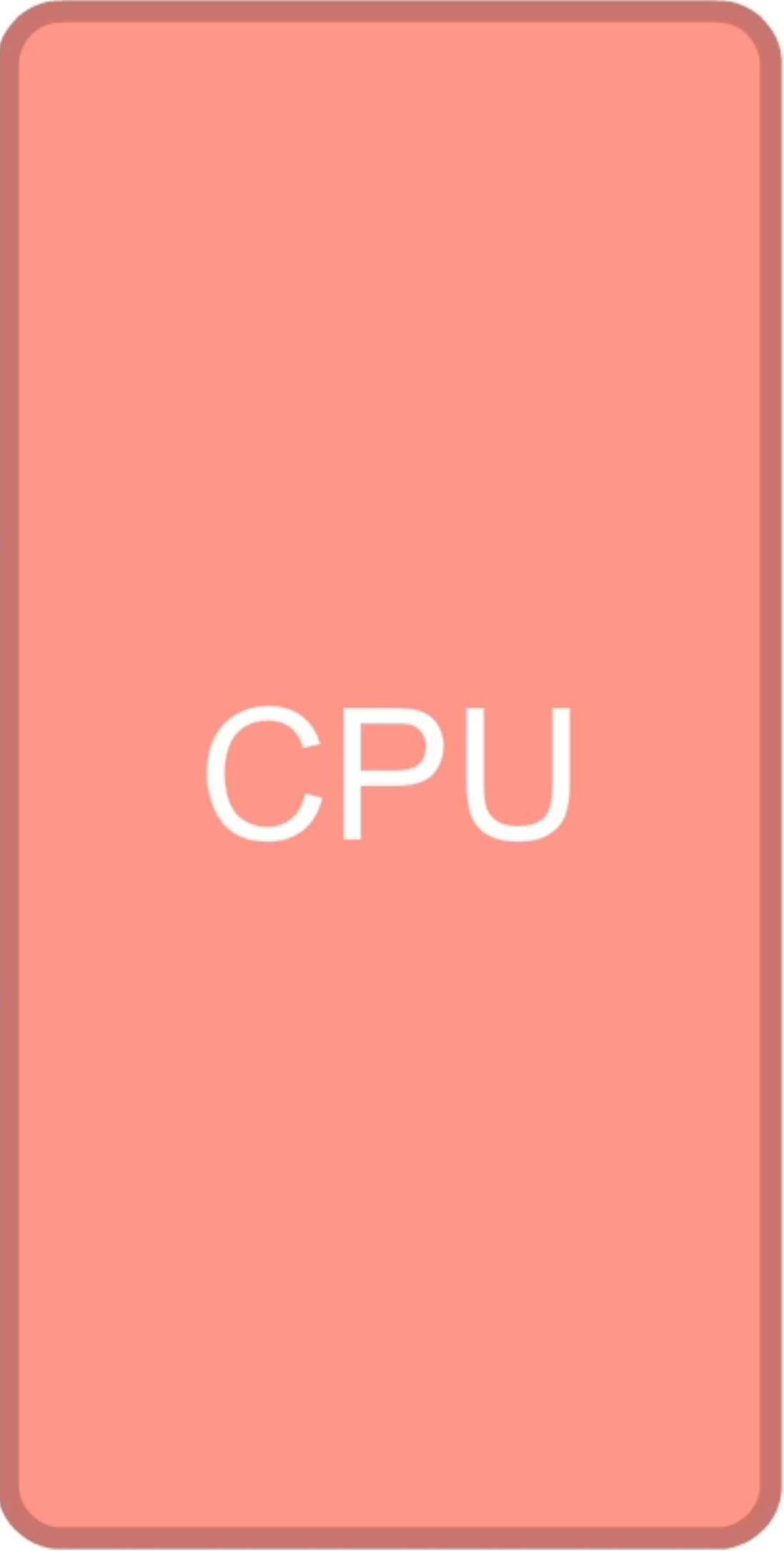
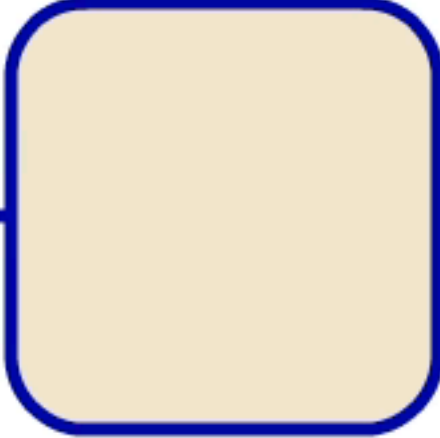
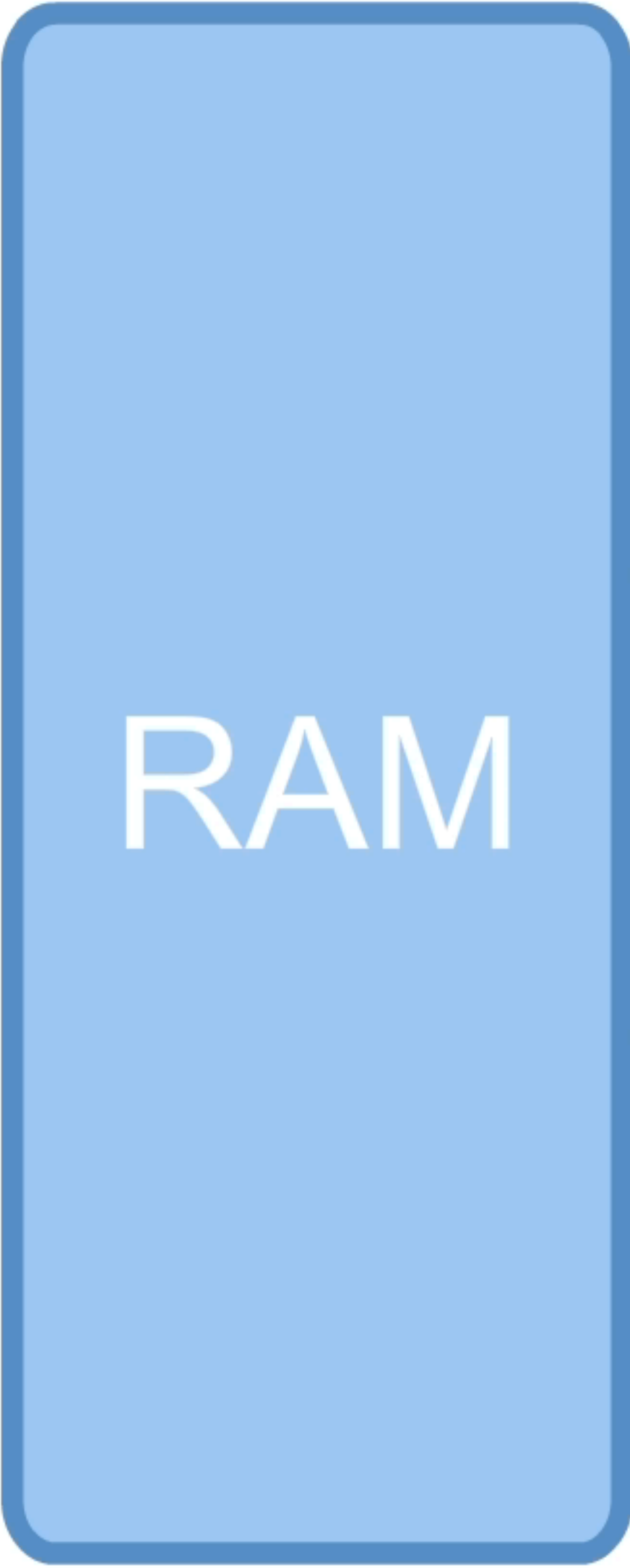


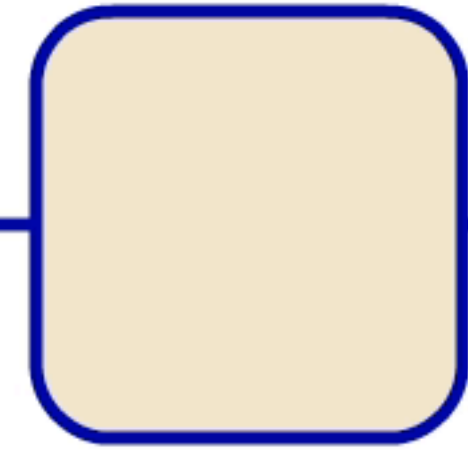
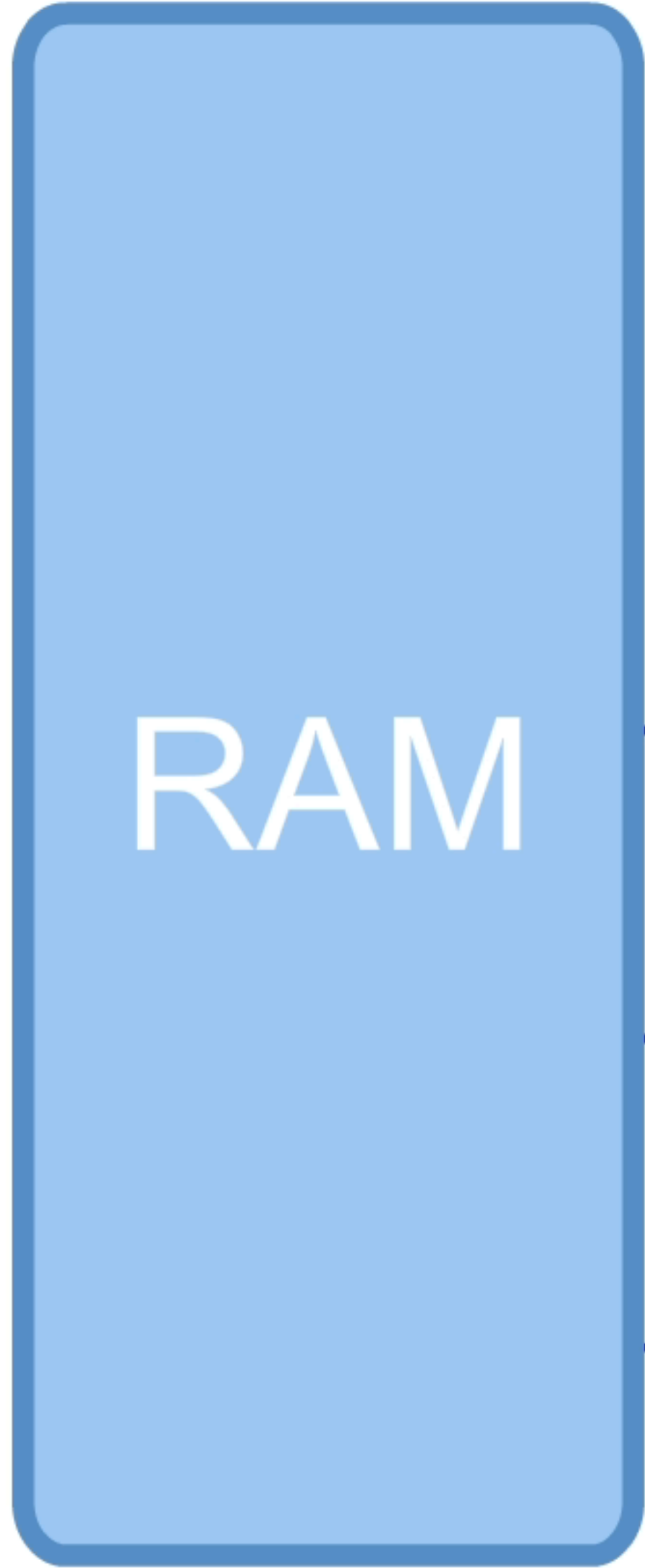
# Main Memory Is Not Slow



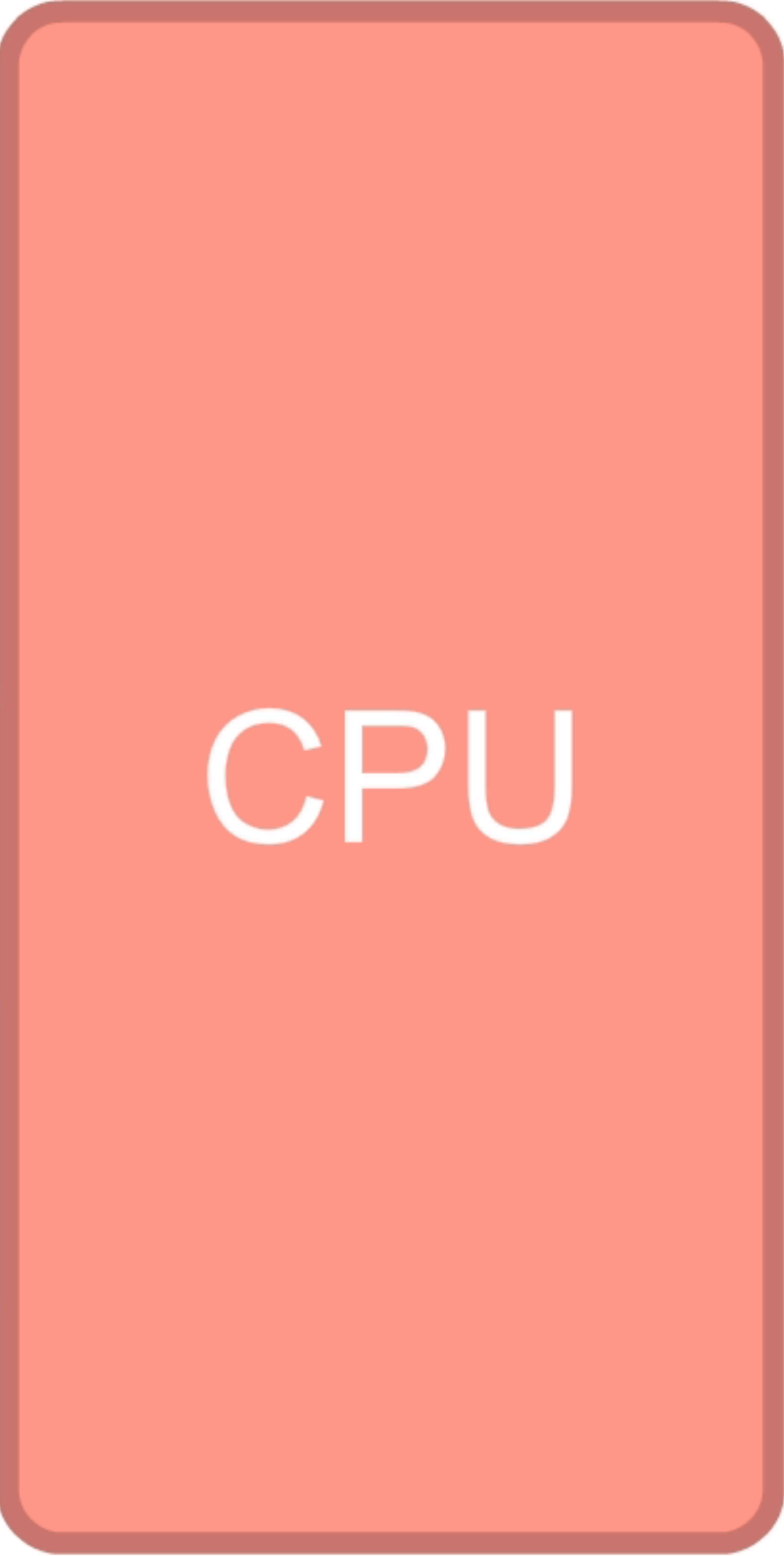
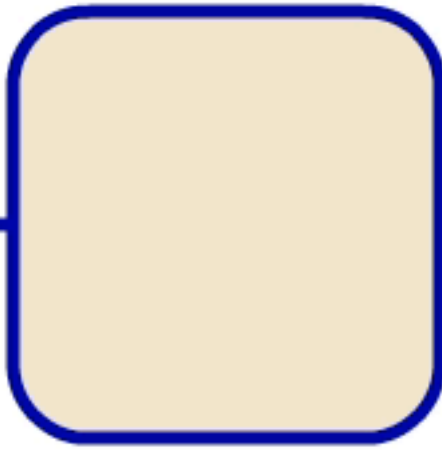


a -> b -> c -> d -> . . .





...



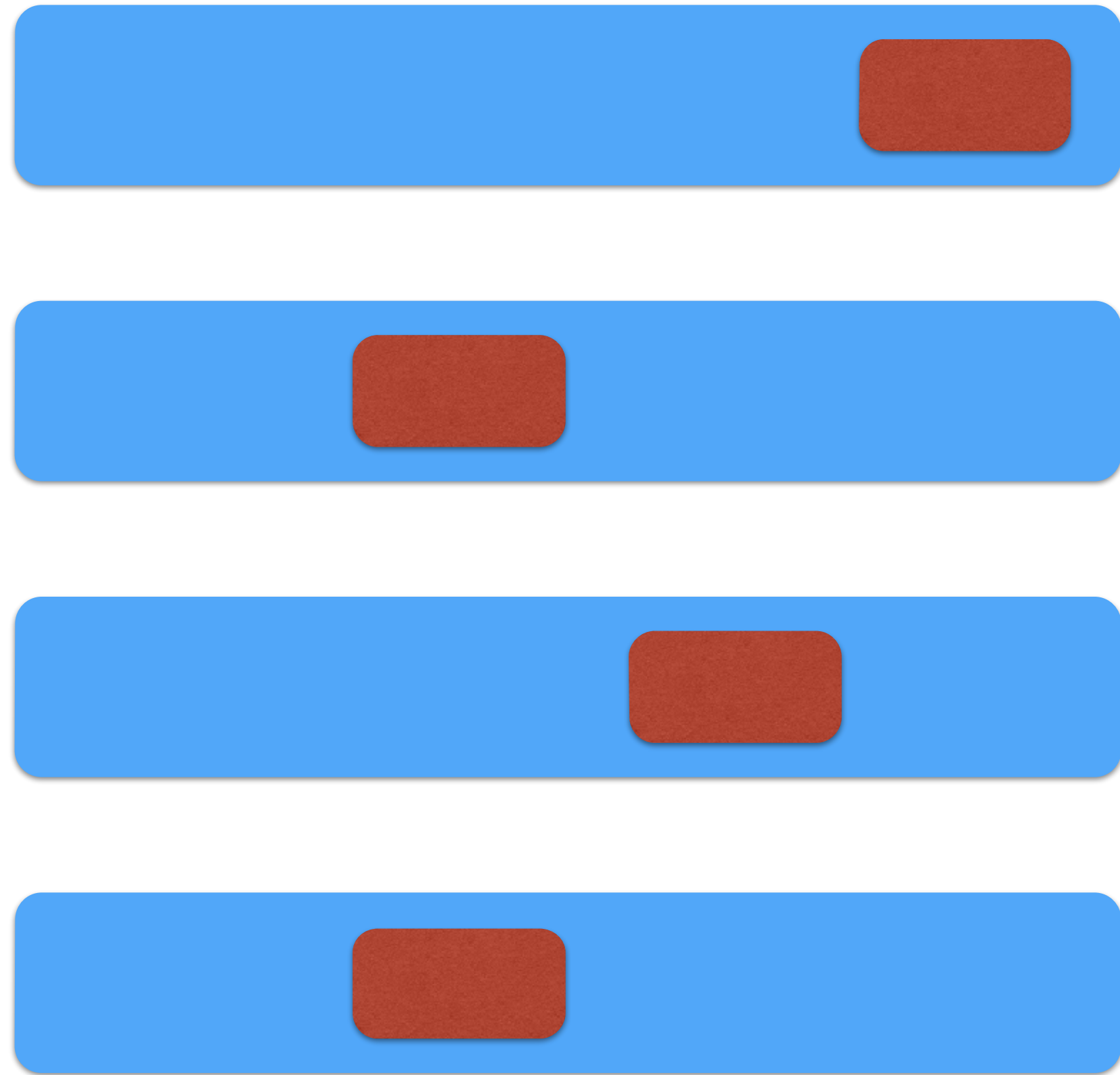
p[0]

p[1]

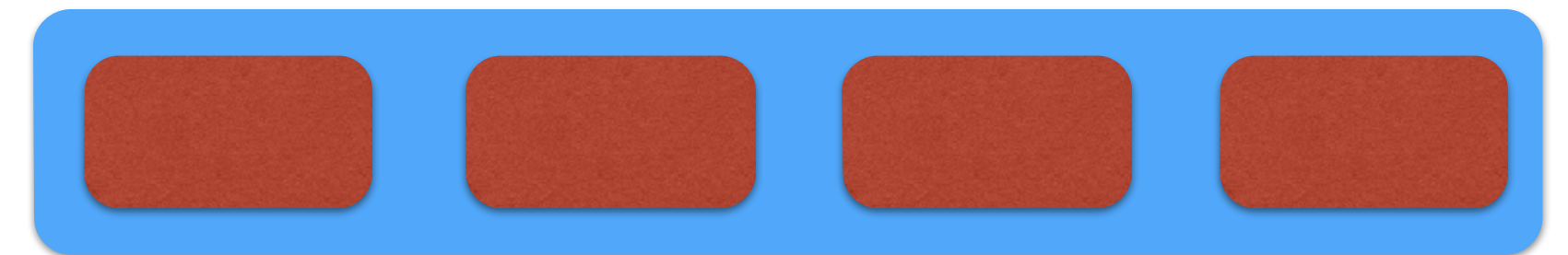
p[2]

p[3]

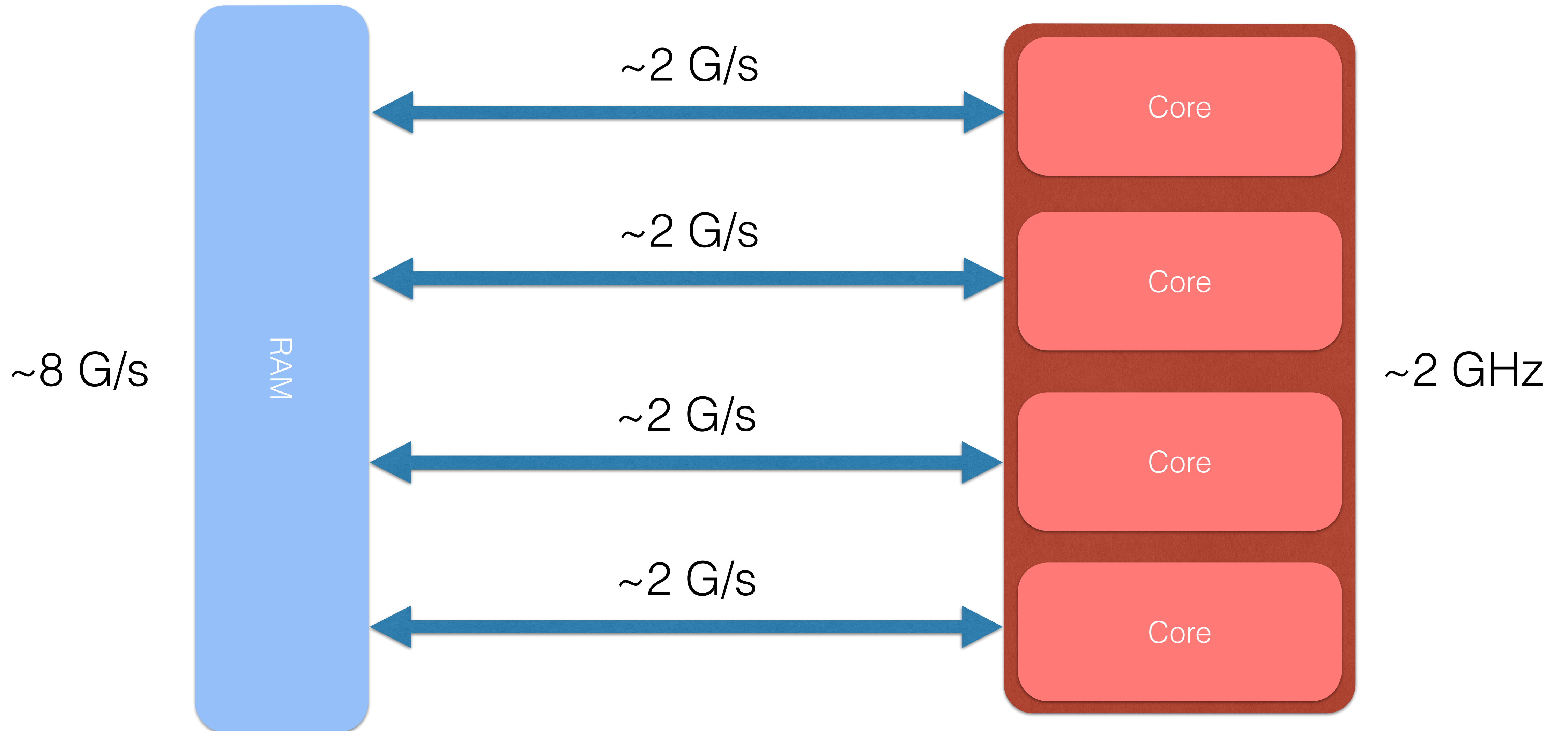
# Cache lines are indivisible



Pack & Align



# One Cycle per Byte





# Measure



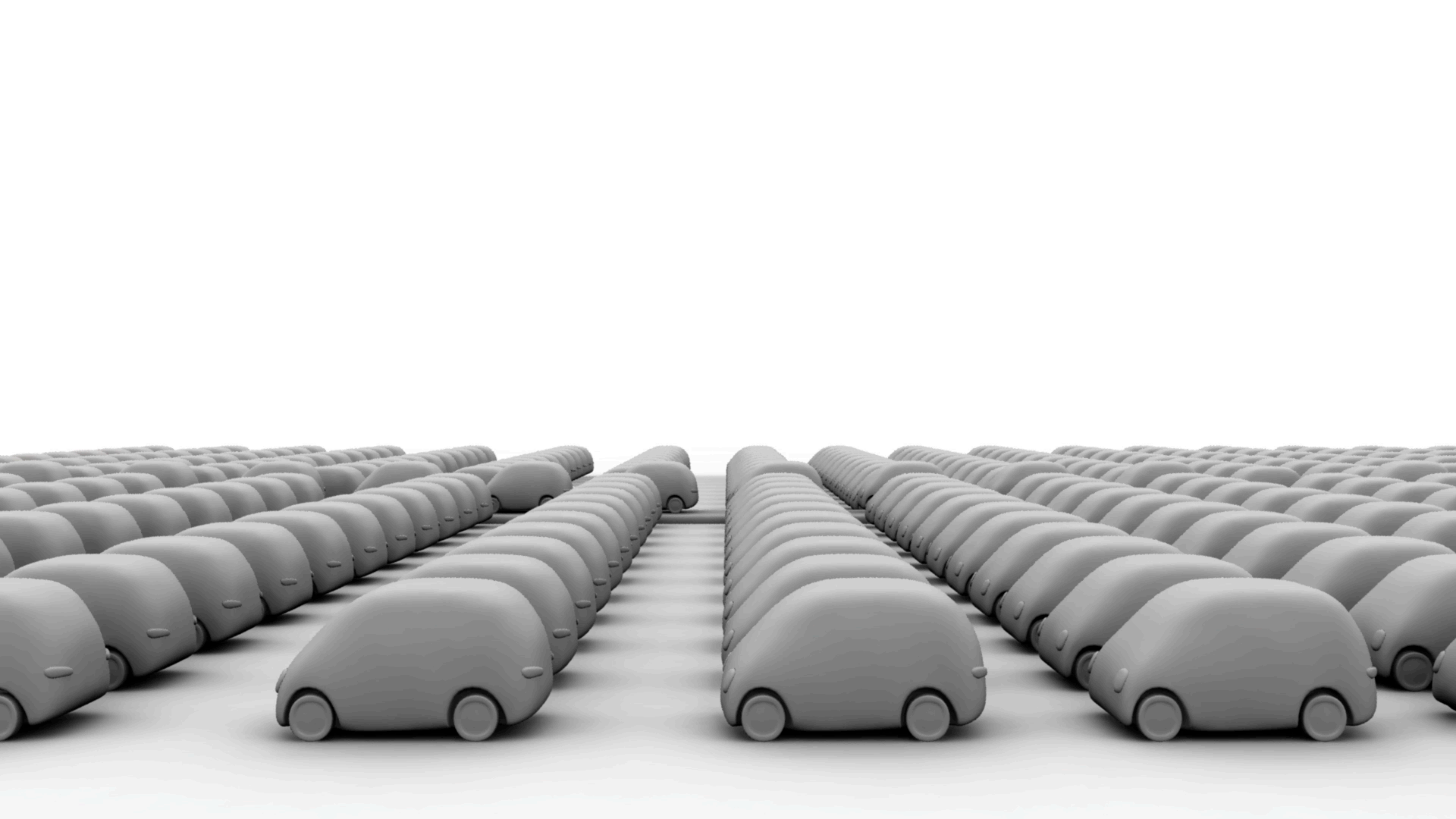
# Repeat

# Compute-Bound

Predictable

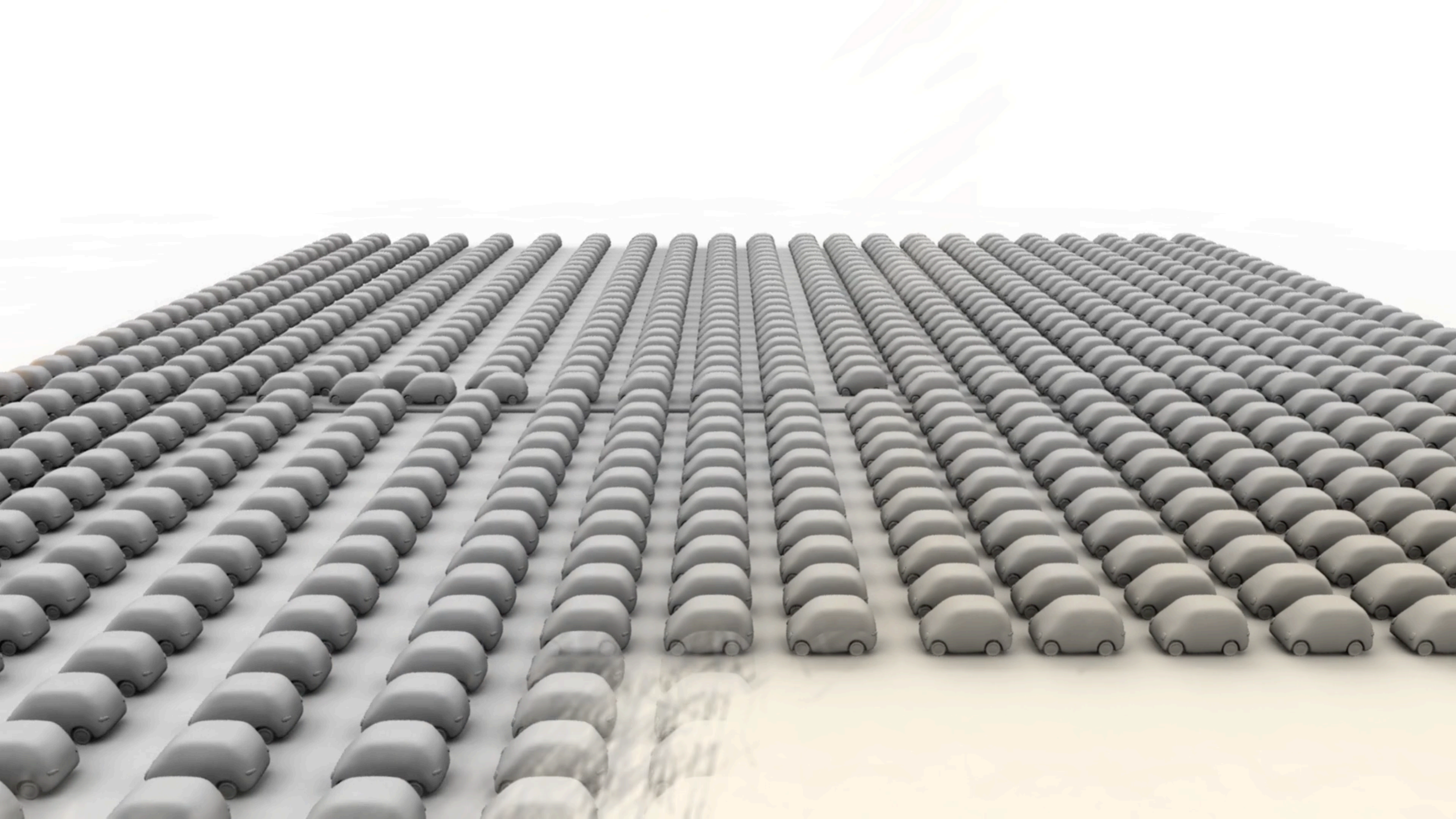
We are trained for this

Hard to achieve

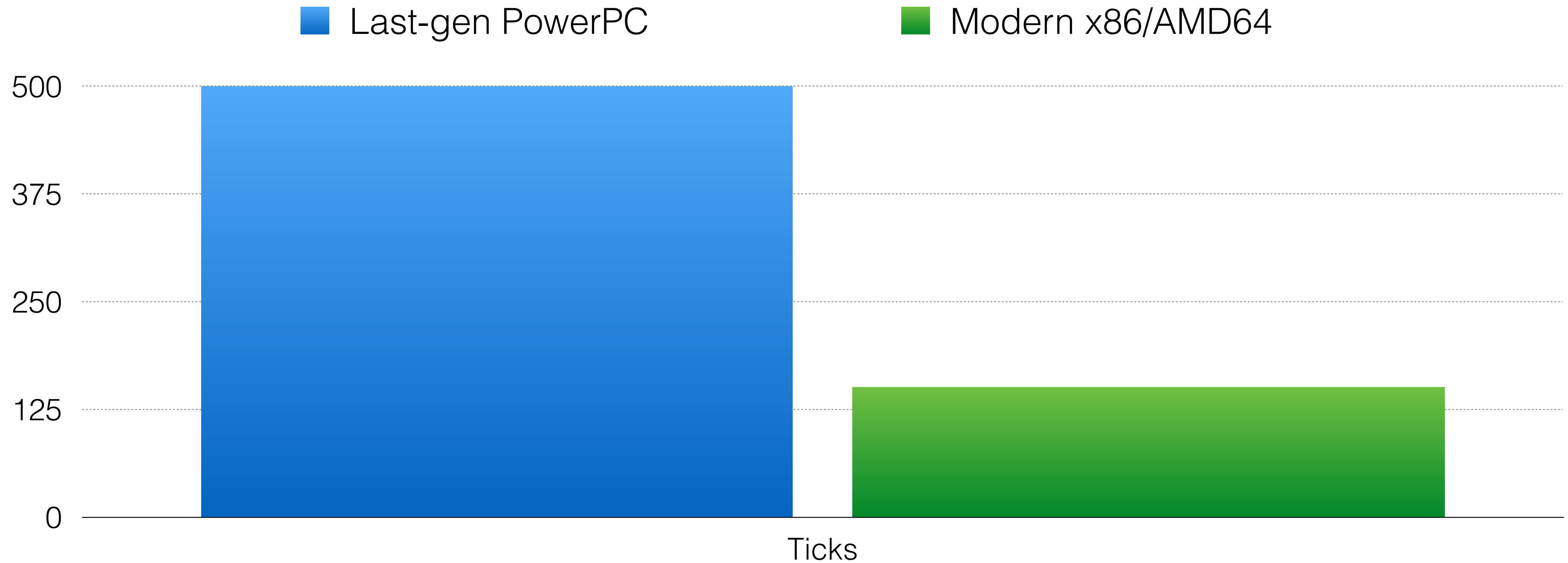


# Cache Misses...

...they're bad



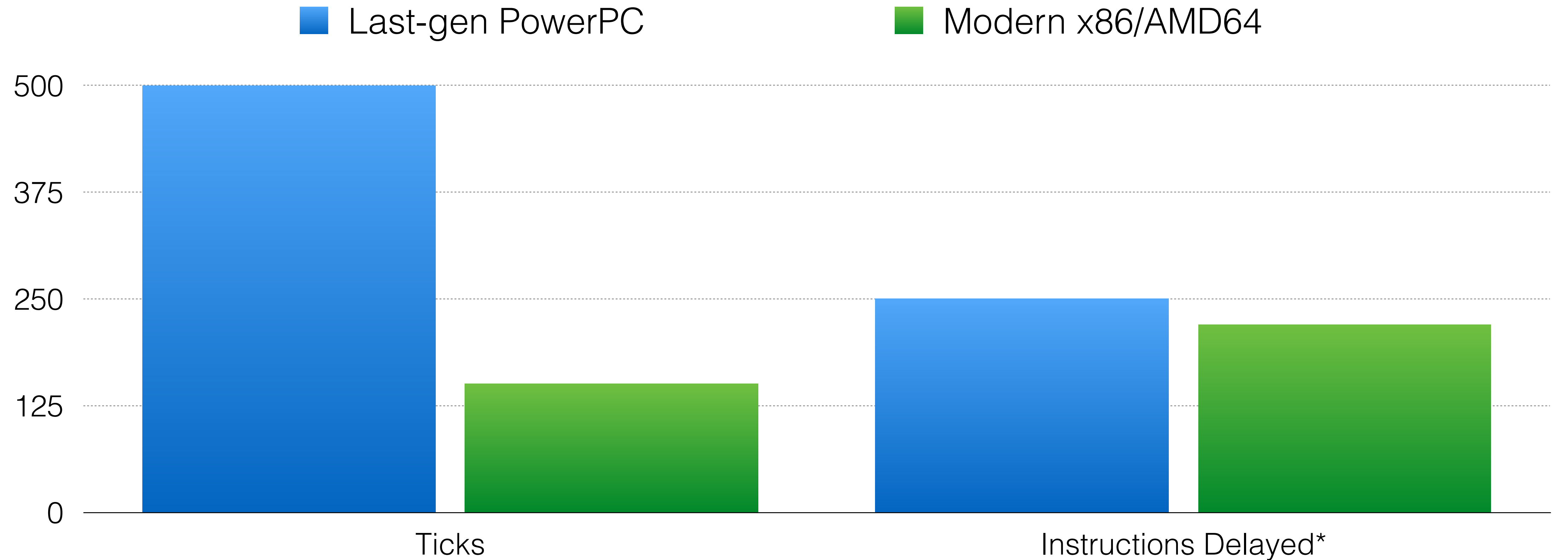
# Cache Miss: Ticks



# 3+ IPC and beyond

1 tick {  
  mov %r11, %r10  
  mov %rsi, %rcx  
  mov %r13, %r12  
  mov %r15, %r14  
  inc %r11  
  inc %r10  
  inc %rsi } 1 tick

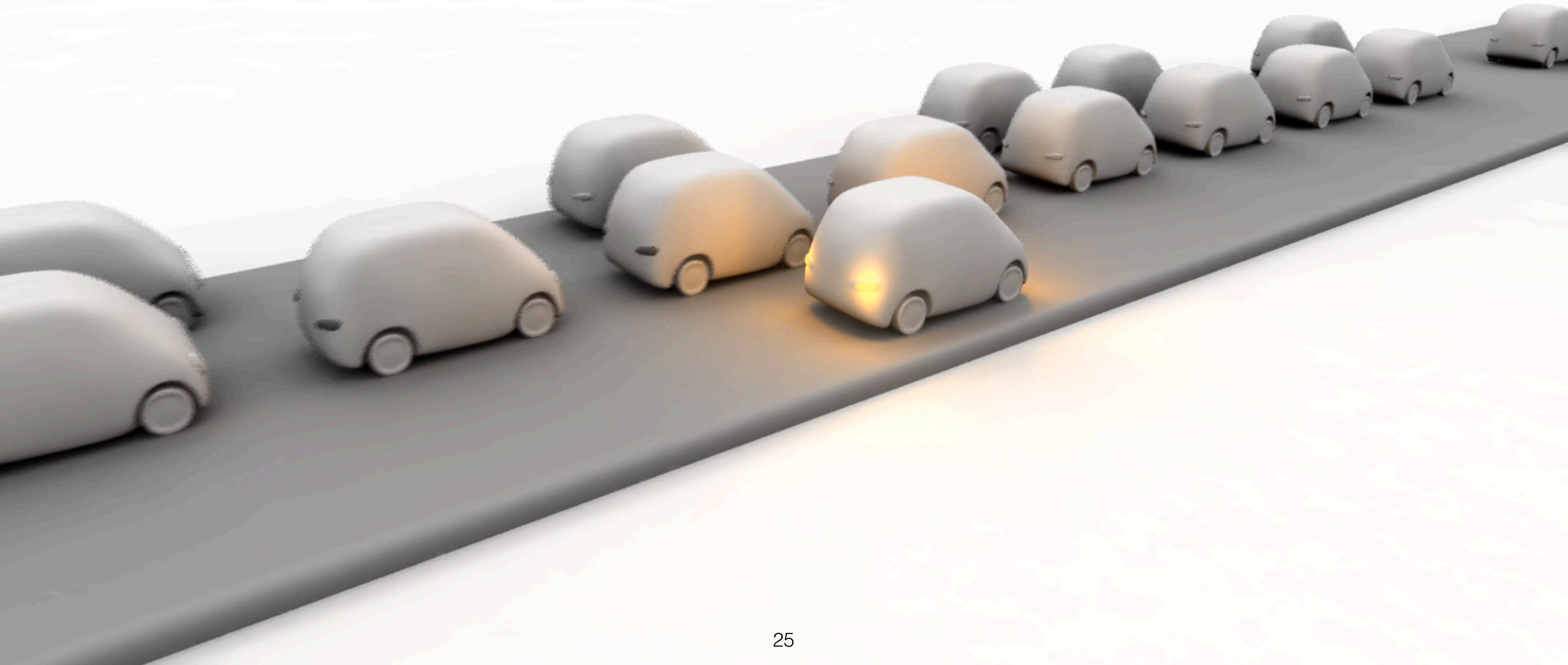
# Cache miss: RealCost (tm)



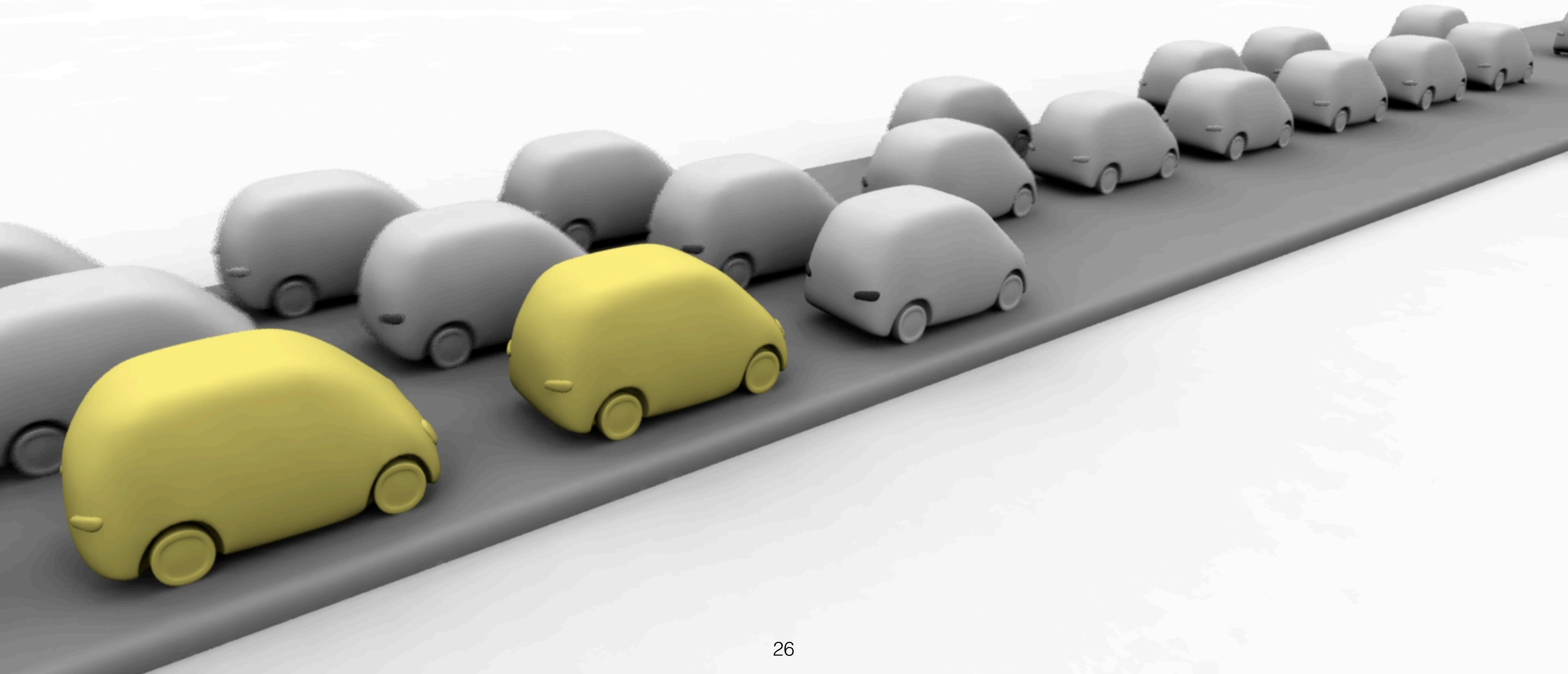
\* Not-So-Rigorous Analysis (tm)



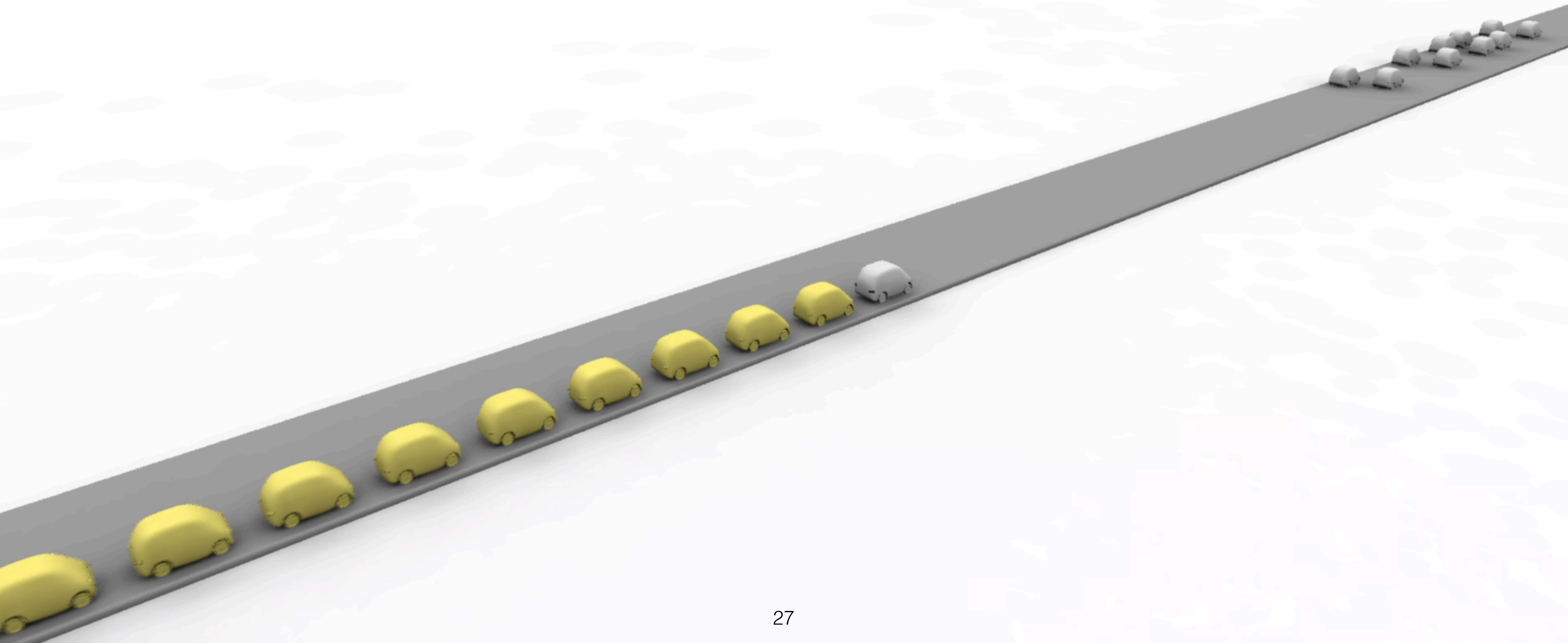
000 FTW!



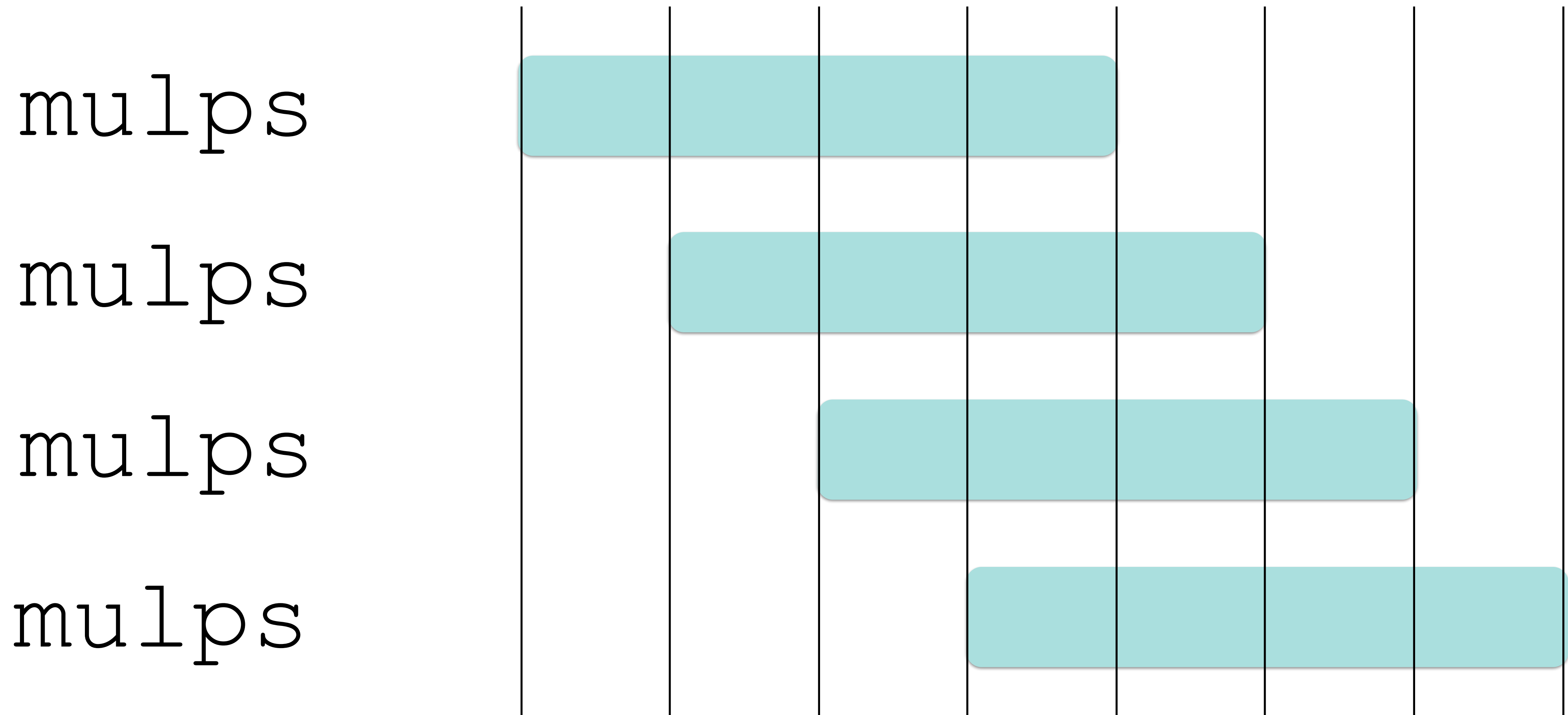
# 000 Limits



# 000 Limits IRL



# Latency and Throughput



$$\text{Solve : } \begin{pmatrix} m_{00} & 0 & 0 \\ m_{10} & m_{11} & 0 \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_x \\ \mathbf{b}_y \\ \mathbf{b}_z \end{pmatrix}$$

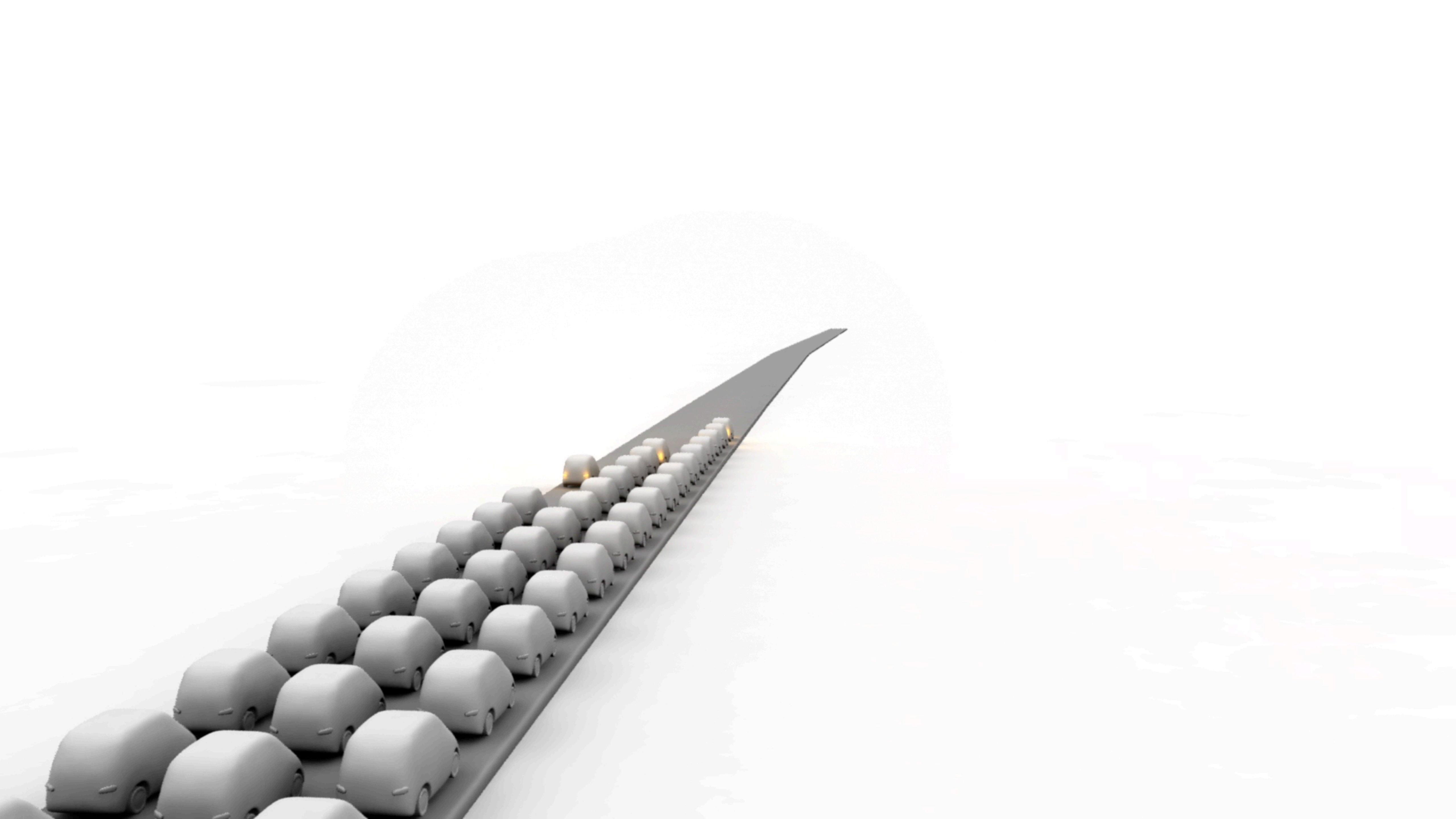
$$\mathbf{x} = \frac{1}{m_{00}} * \mathbf{b}_x ;$$

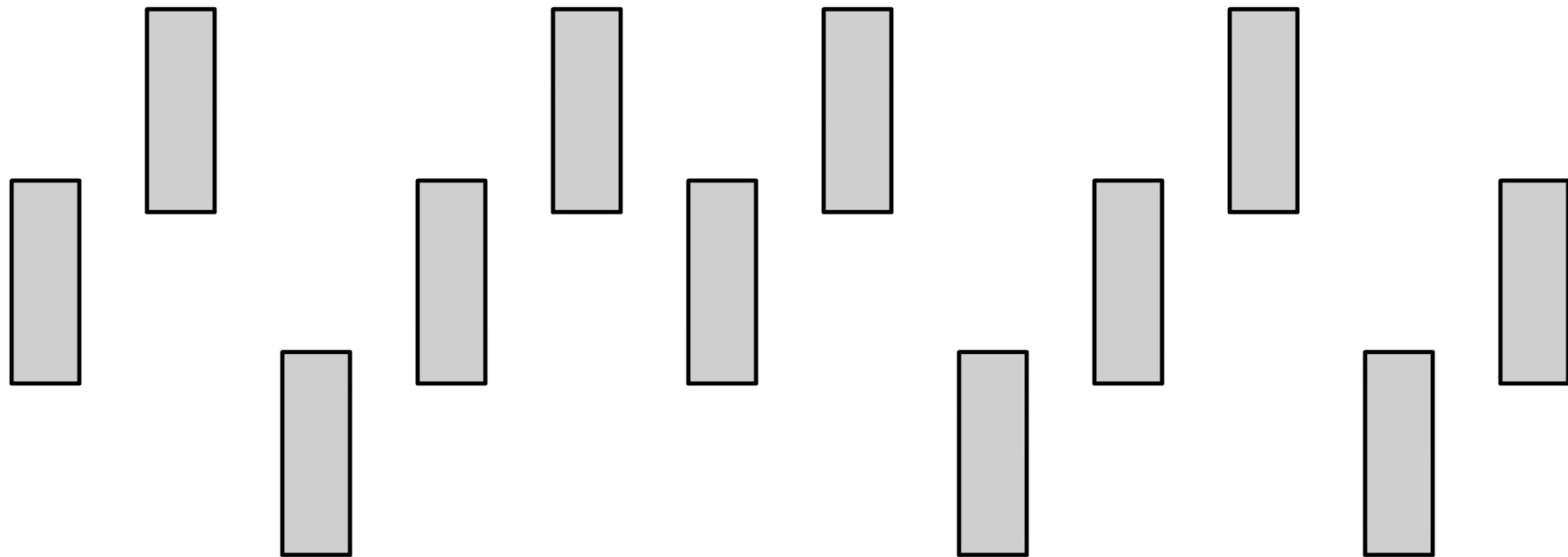
$$\mathbf{y} = \frac{1}{m_{11}} * (\mathbf{b}_y - m_{10} * \mathbf{x}) ;$$

$$\mathbf{z} = \frac{1}{m_{22}} * (\mathbf{b}_z - m_{20} * \mathbf{x} - m_{21} * \mathbf{y}) ;$$

# Think Short Chains

More opportunities to reorder  
Many cache misses resolve at once



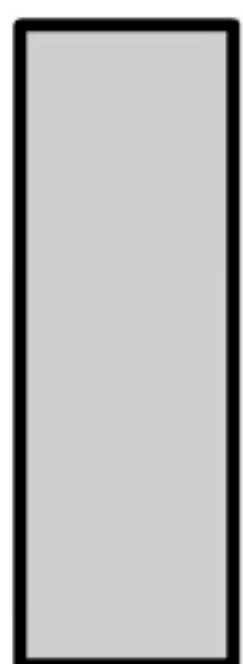
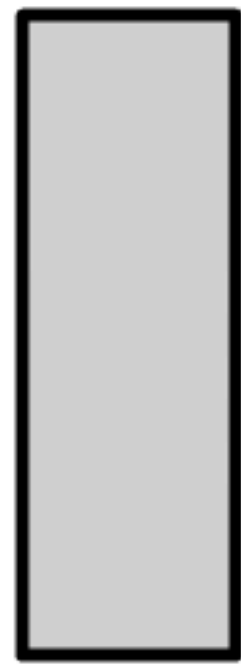
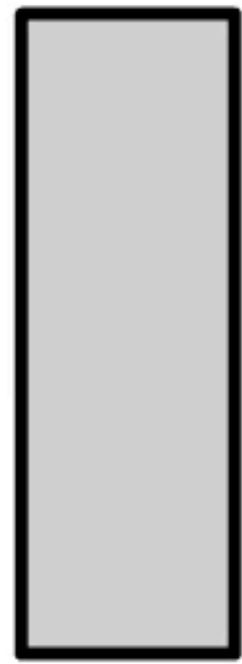
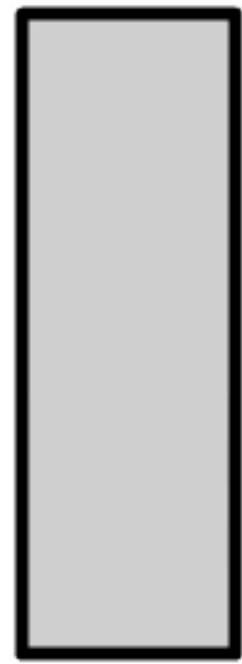
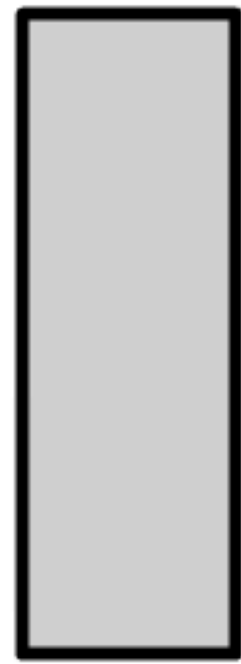
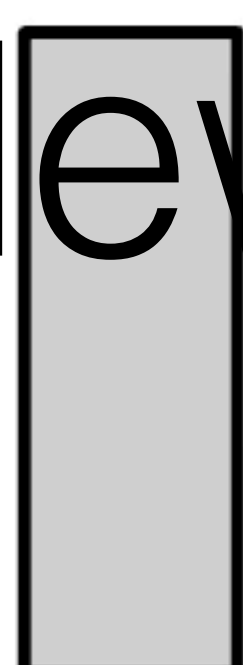
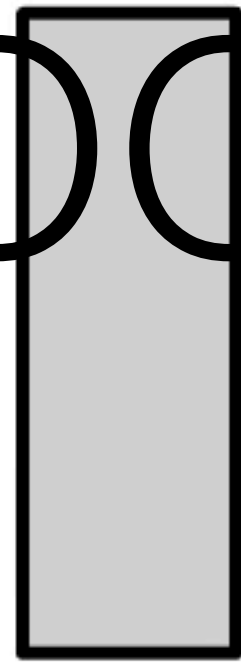
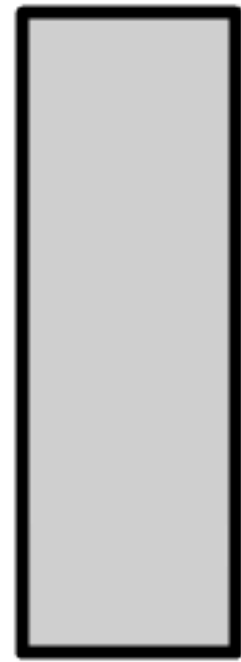


Now 12 ticks (was 27)

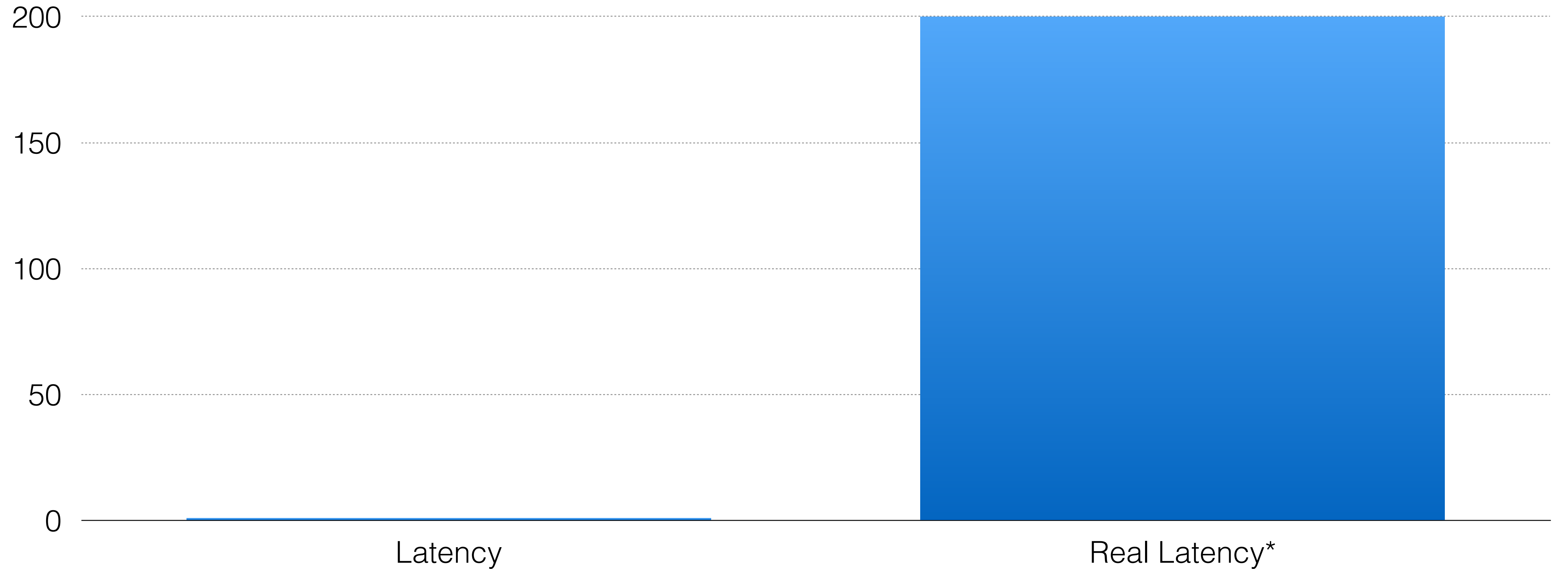




0000: The New Old



# Real CPU Latency

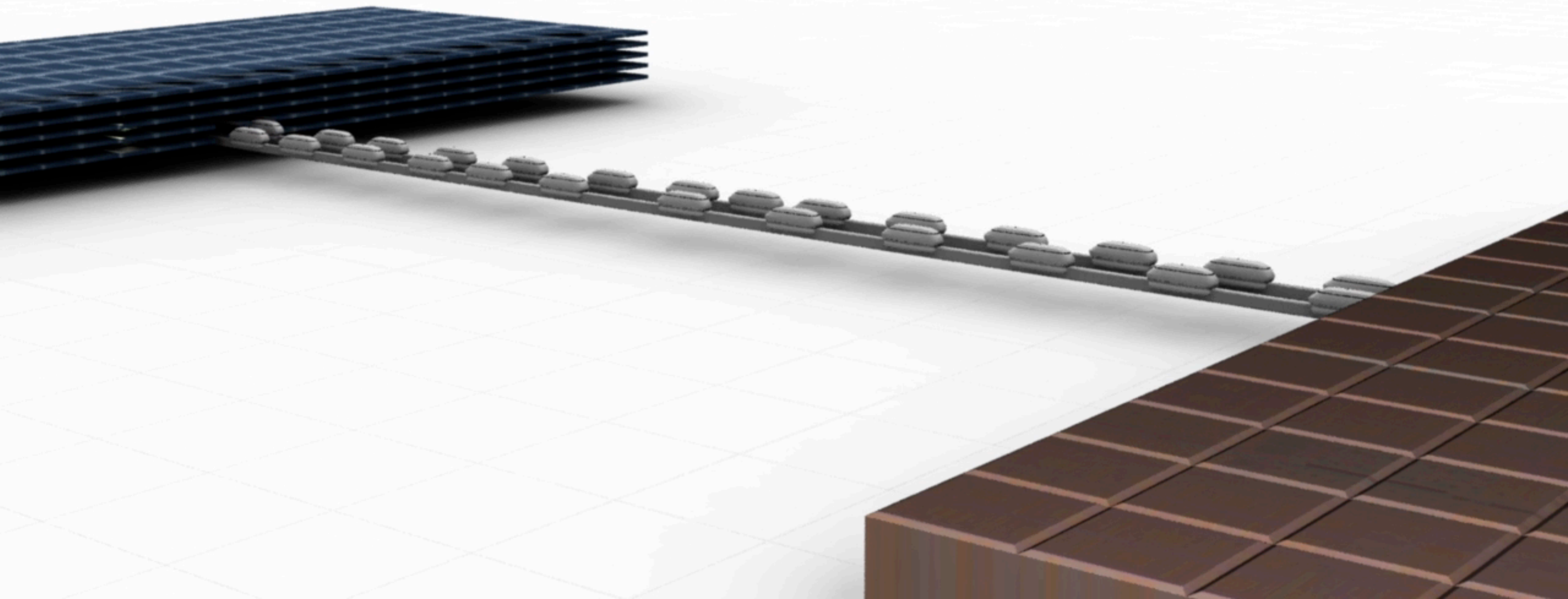


\* The Real Latency: a technical term (tm)

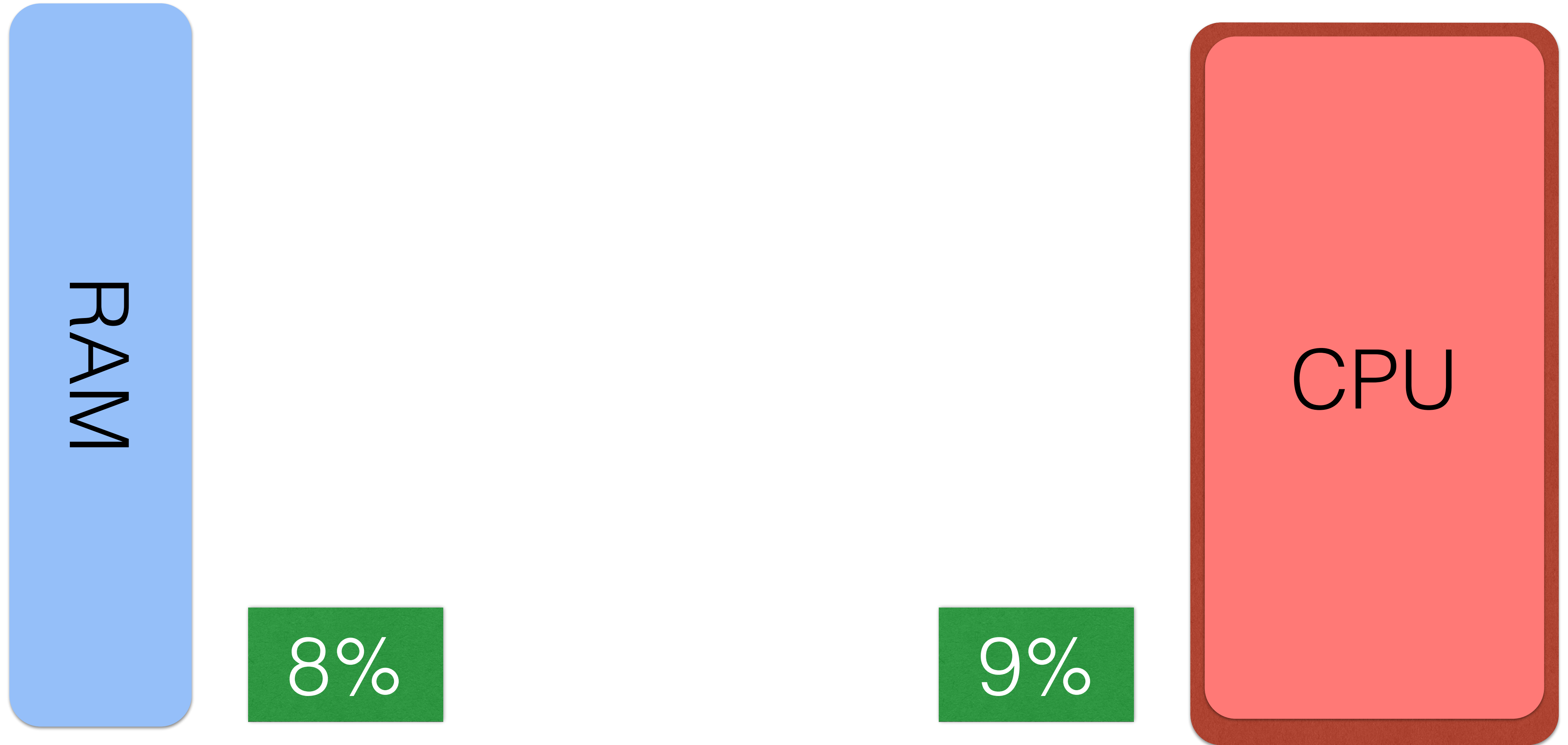
# Prefetch

Linear Streams  
Cache Miss Overlap  
Fix All or Nothing

# Maxing RAM out



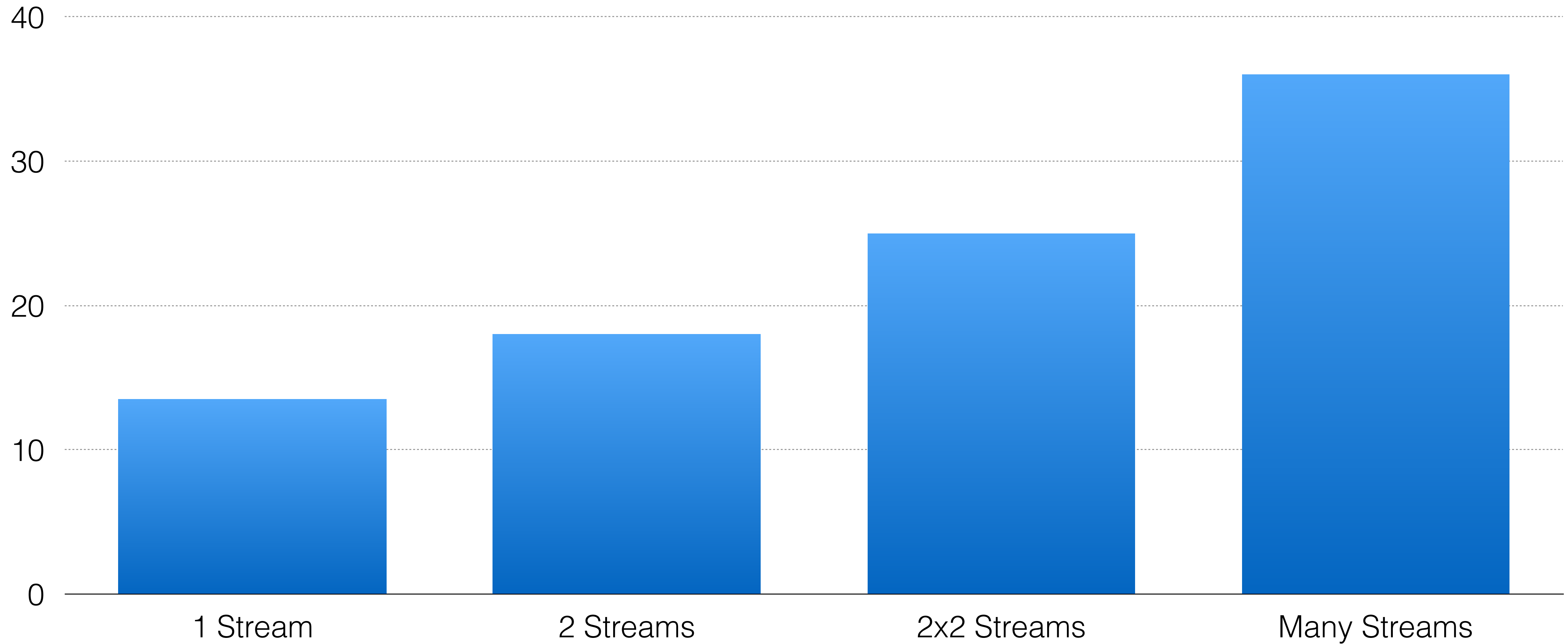
# Load Balance



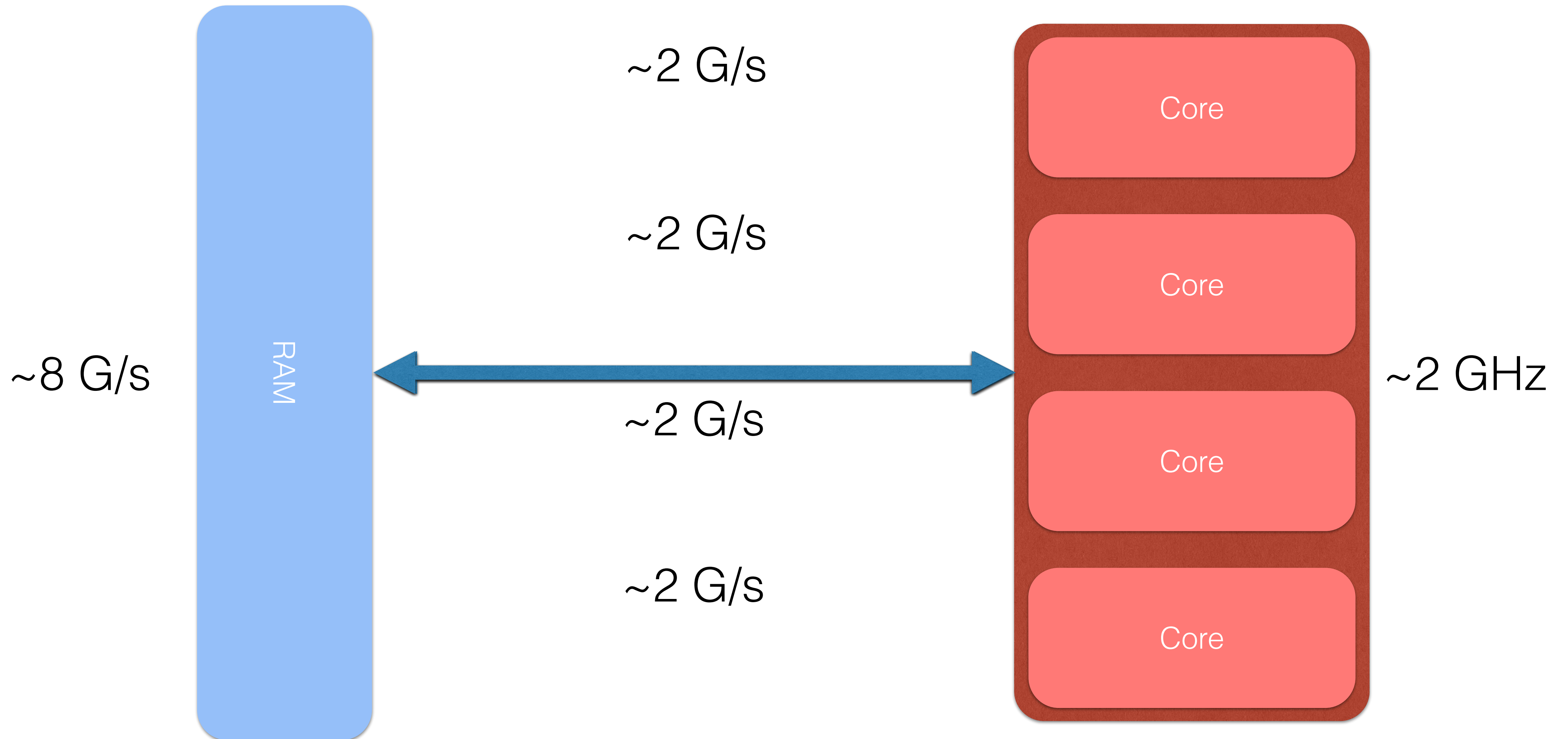
# Prefetch Techniques

Use Linear Streams  
Rely on Reordering  
Manual prefetch

# 37 GB/s? Really?



# 1 byte per op





# Cache Misses...

...beware of 'em.

# Bandwidth: take 2

1. Load Data
2. ...Other Stuff (150+ cycles)...
3. Use Data

# Bandwidth: Hi-End

Lo-End: 1 byte/cycle

Hi-End: 3-4 bytes/cycle

# Load Hit Store



Don't Mix

# Writing is Reading

Stores affect cache  
(pollution, migration)

Branches are inexpensive

# Branch (mis)prediction

Predictability=performance

1-4 deep history

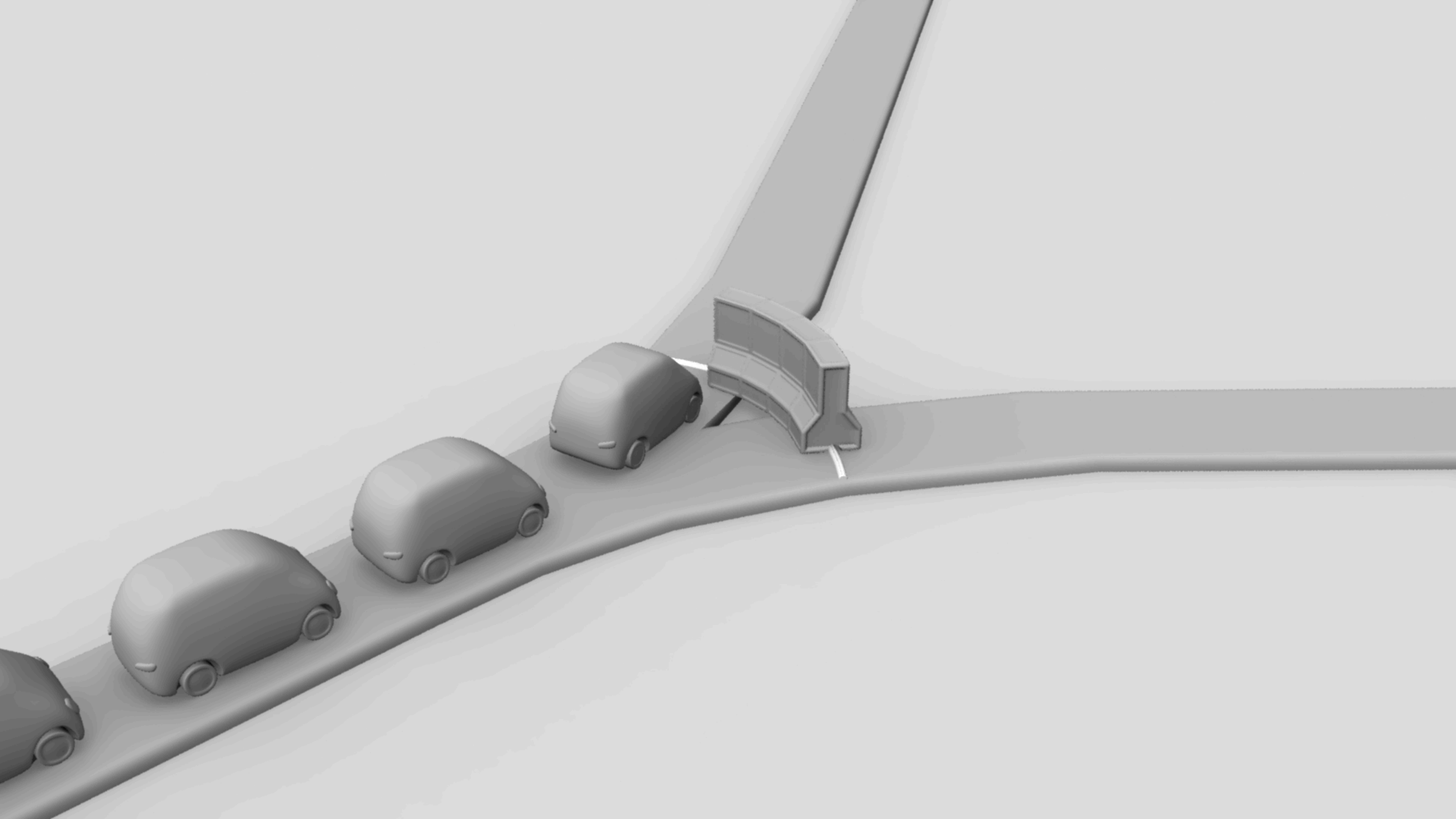
random = 50% mispredicted

misprediction is ~10-20 ticks

# Branch Prediction

predicted branch ~0-1 tick

~100% consistent = ~100% predicted





# Careful: Recursion

Return Stack Buffer  
Typically 8-24 deep

# Case study: cloth

- 16 verts
- 1.6 Instructions per Cycle
- 600 cycles
- 5 cache lines streamed
- 8 cache lines random access
- ~1.4 bytes per cycle
- ALU bound, RAM: 4.7 G/s

# Cycles / Iteration

```
nStart = __rdtsc();  
  
for ( int i = 0; i < N; ++i )  
    OneIteration();  
  
Cycles += __rdtsc() - nStart;  
Iterations += N;
```

# Derived Metrics

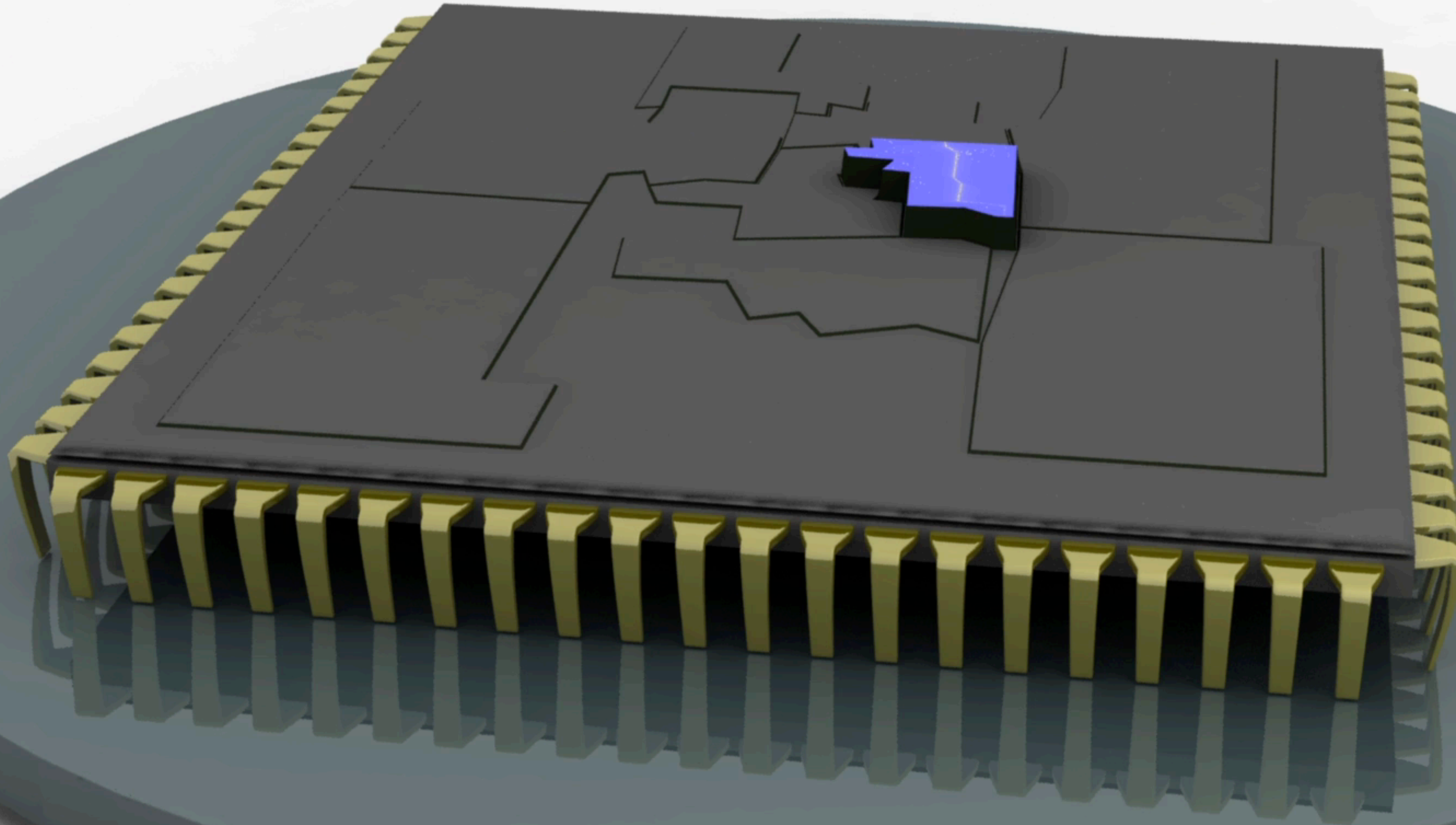
Instructions per cycle

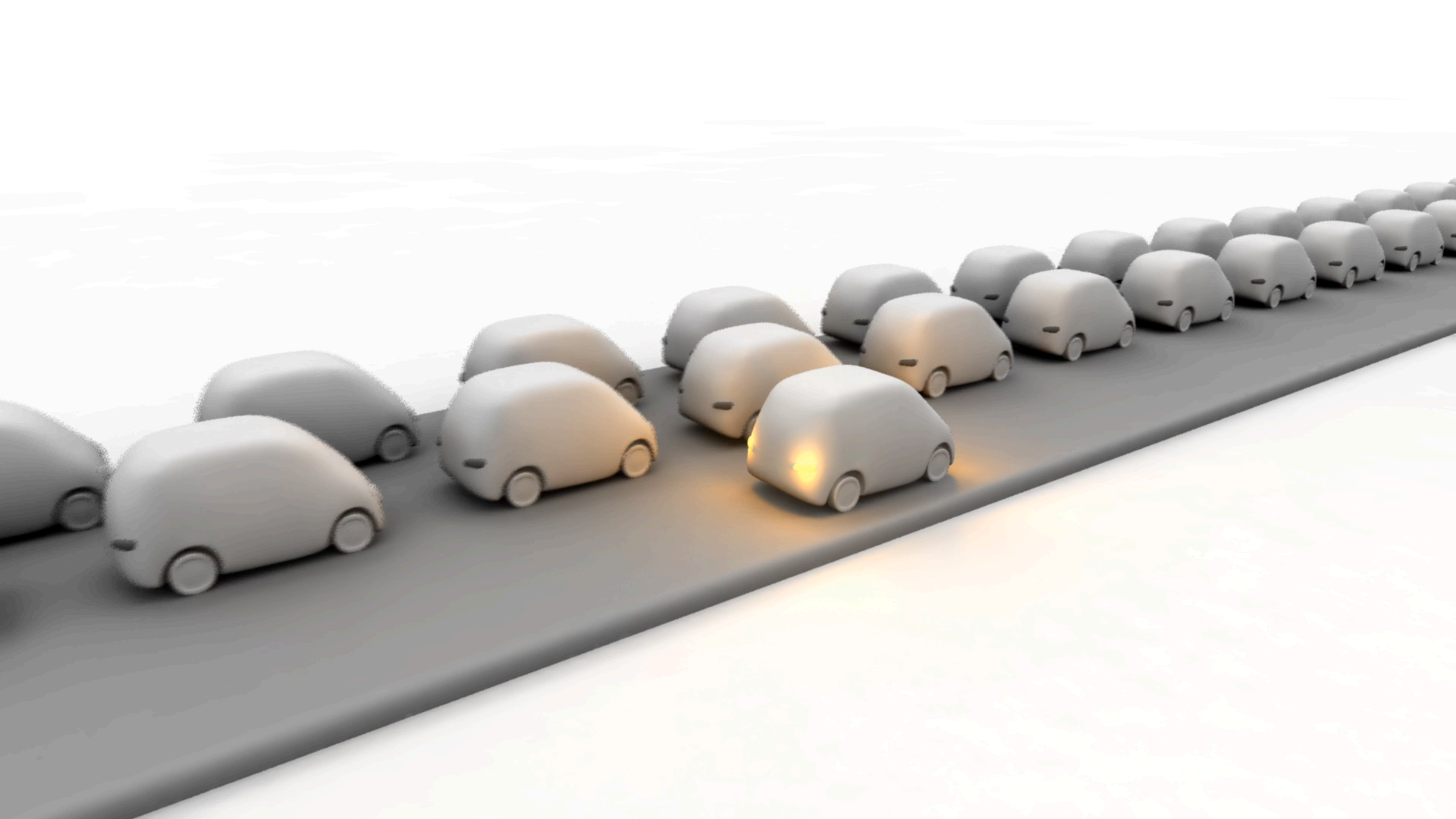
Cycles per byte

Instructions per byte

Bytes per second

# Reorder Buffer





# Summary

- Cache
- OOO
- RAM
- 1 byte per op

# 8 bit Rulez!

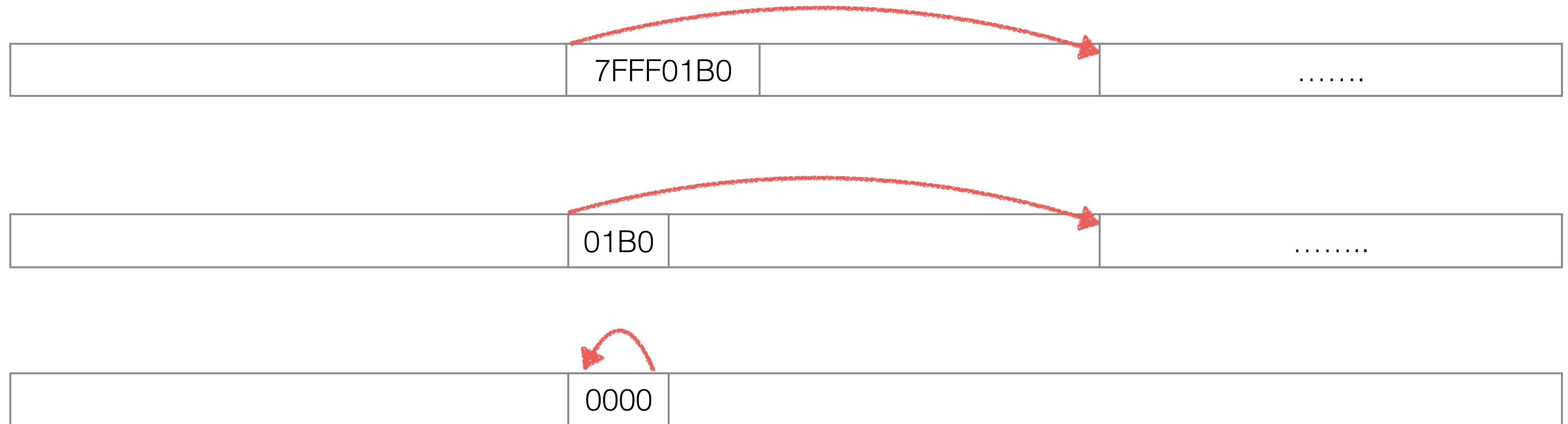
	8-bit	64-bit
Small Pointers	<b>Yes</b>	<b>No</b>
Fixed Addresses	<b>Yes</b>	<b>No</b>



# Relative Pointer

	RPointer< Foo >	Foo *
Same in 32- and 64-bit	<b>Yes</b>	<b>No</b>
Moveable	<b>Yes</b>	<b>No</b>

# *Relative* Pointer



# Limitations



16-bit  $\rightarrow$   $\pm 32\text{KB}$

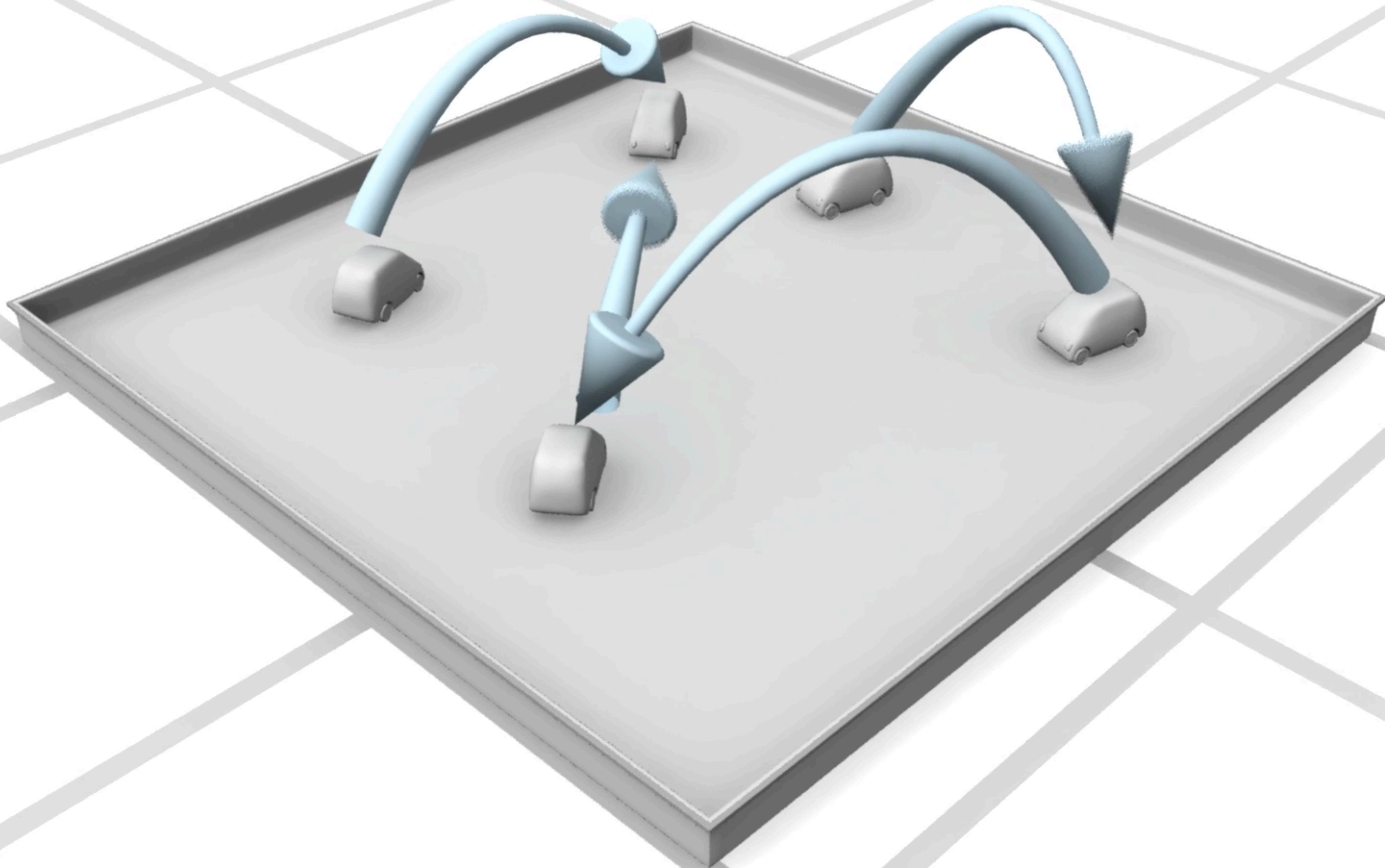
32-bit  $\rightarrow$   $\pm 2\text{GB}$

# RPointer<T>

```
template <typename T>
class RPointer
{
    unsigned short m_nOffset;
public:
    T* operator -> ()
    {
        Assert( m_nOffset != 0 ); // must not be NULL
        return ( ( byte* )&m_nOffset ) + m_nOffset;
    }

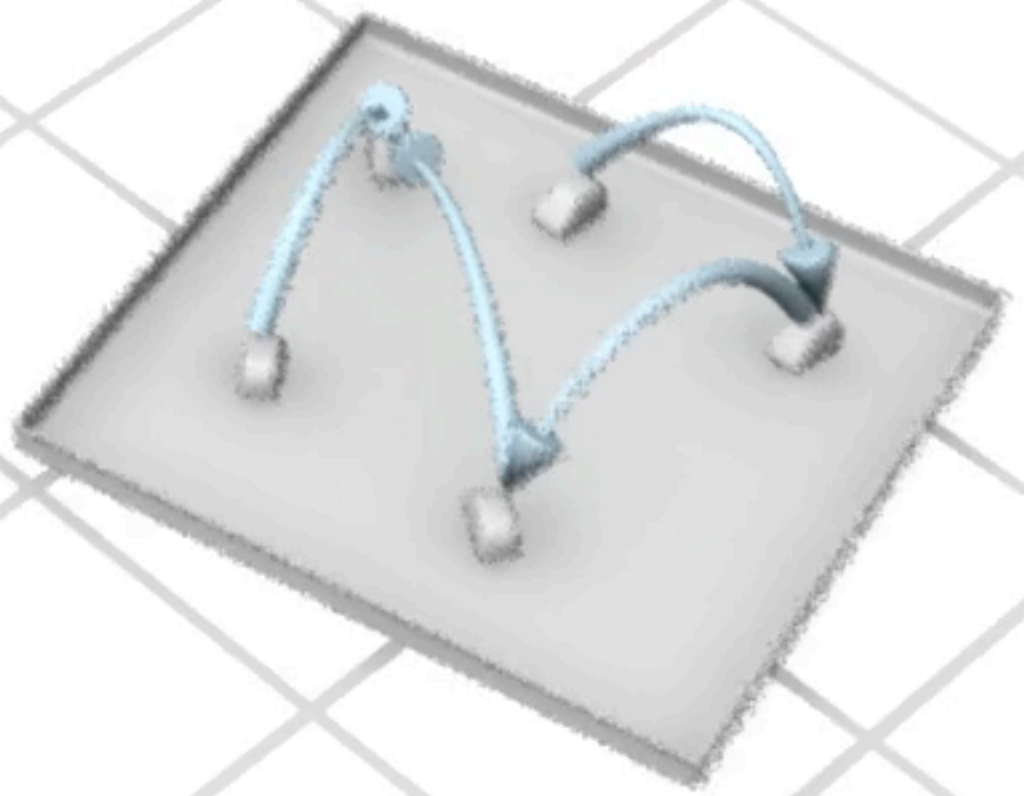
    void operator = (T*p){m_nOffset = p?((byte*)p)-(byte*)this:0;}
};
```





# Cache Misses...

...did I mention 'em?



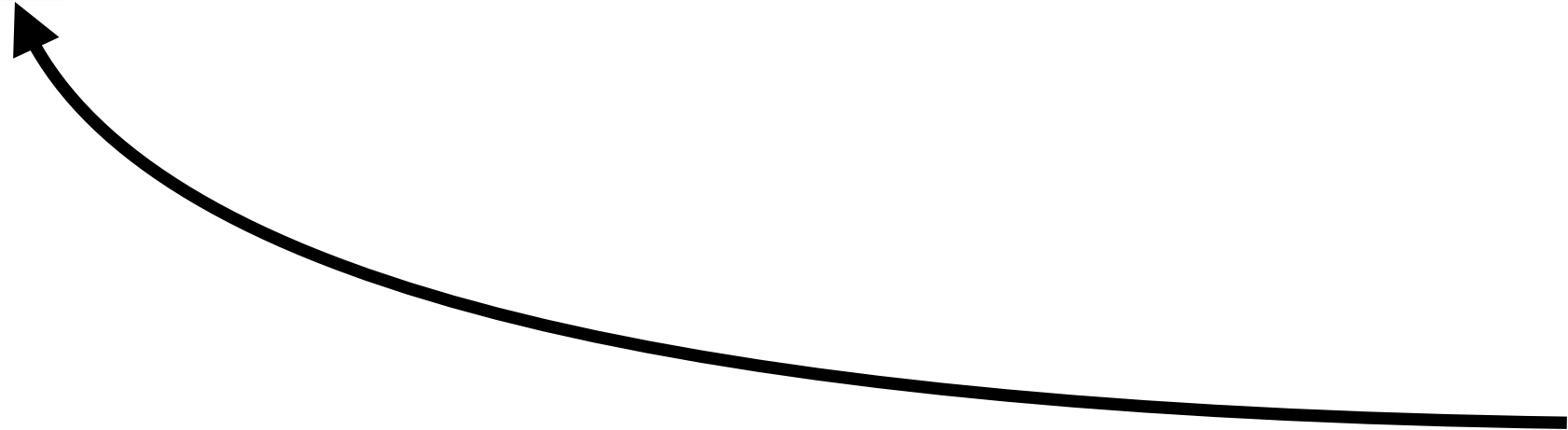


# Filling In

VirtualAlloc



Commit



# Pointer vs Offset

```
struct Elem
{
    Elem *m_pNext;
    int m_nData;
};

...
Elem *p = new(pAllocator) Elem;
p->m_pNext = NULL;
p->m_nData = 123;
```

```
struct Elem
{
    RPointer<Elem> m_pNext;
    int m_nData;
};

...
Elem *p = new(pAllocator) Elem;
p->m_pNext = NULL;
p->m_nData = 123;
```

# Strengths

- Very simple
- No runtime overhead
- Data is relocatable
- Data is more compact

# Limitations

- No Automatic Versioning
- Continuous Data Block
- More Stuff to Learn

# Summary

- Cache
- OOO
- RAM
- 1 byte per op
- Resource Pointers

A 3D rendered scene depicting a traffic jam. A long line of grey, rounded cars is stuck in a queue on a road. In the foreground, a red and white striped barrier is down, blocking the cars. A green sign with the white text 'CPU' is suspended over the road. The background is a plain white horizon under a white sky.

CPU

Big O



# Tricks





$O^*(?)$

How useful is  $O^*$ ?

“0”

Best optimization ever

Do nothing

$O(1)$

- Cache it
- Do it once per frame
- Approximate it
- Be creative...

$$O(N) \equiv O(200 * N)$$



Array



List

$$N \neq 200 * N$$



Array



List

# $O^*$ nuances

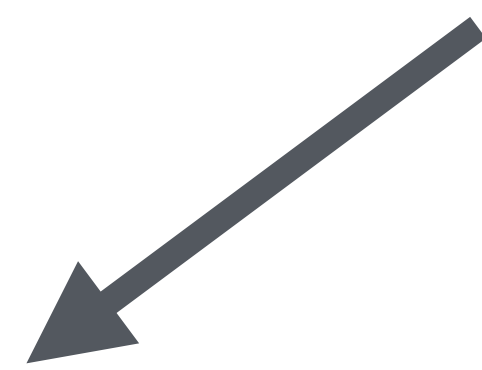
	Traversal	Cache Misses in Traversal
Array	$O(N)$	<b><math>O(0)</math></b>
Linked List	$O(N)$	$O(N)$

# Map vs Hashtable

	Classic RB-Tree	Classic Hash Table
Worst-Case	<b><math>O(\log N)</math></b>	<b><math>O(N)</math></b>
Amortized	<b><math>O(\log N)</math></b>	<b><math>O(1)</math></b>

# Priorities

Cache-Friendly Algorithms



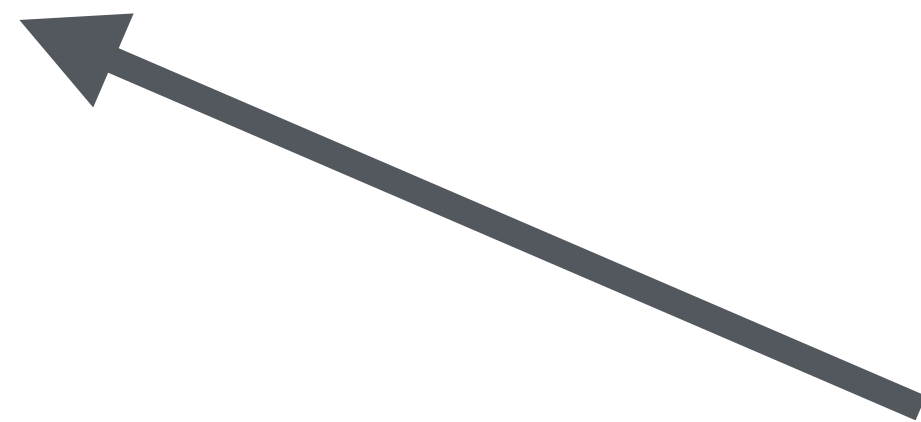
- Algorithm vs Cache

- $C1 * N$  vs  $C2 * \log N$



Depends on  $N$ ,  $C1$  and  $C2$

- CPU vs Memory



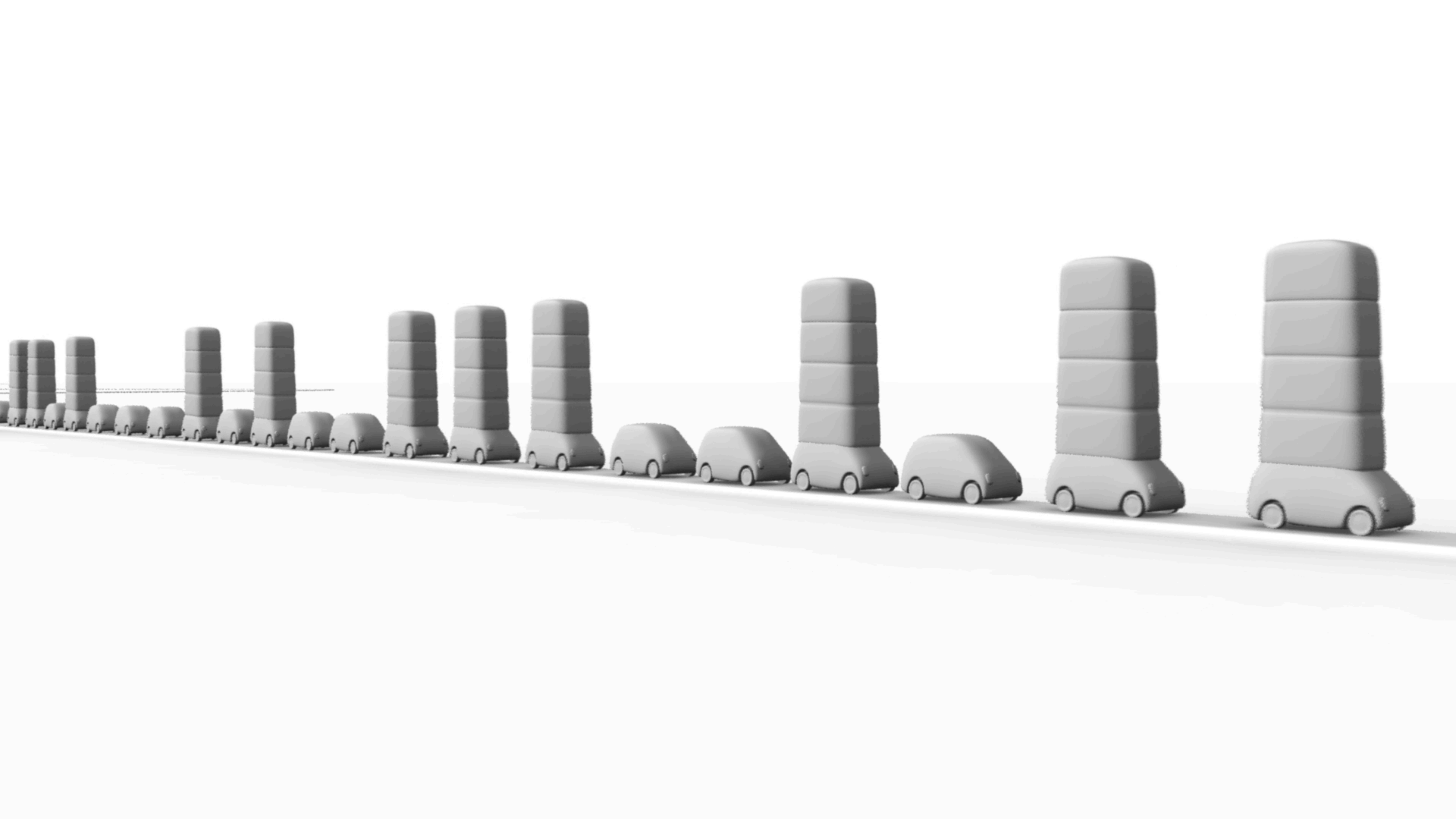
Where's the Bottleneck?



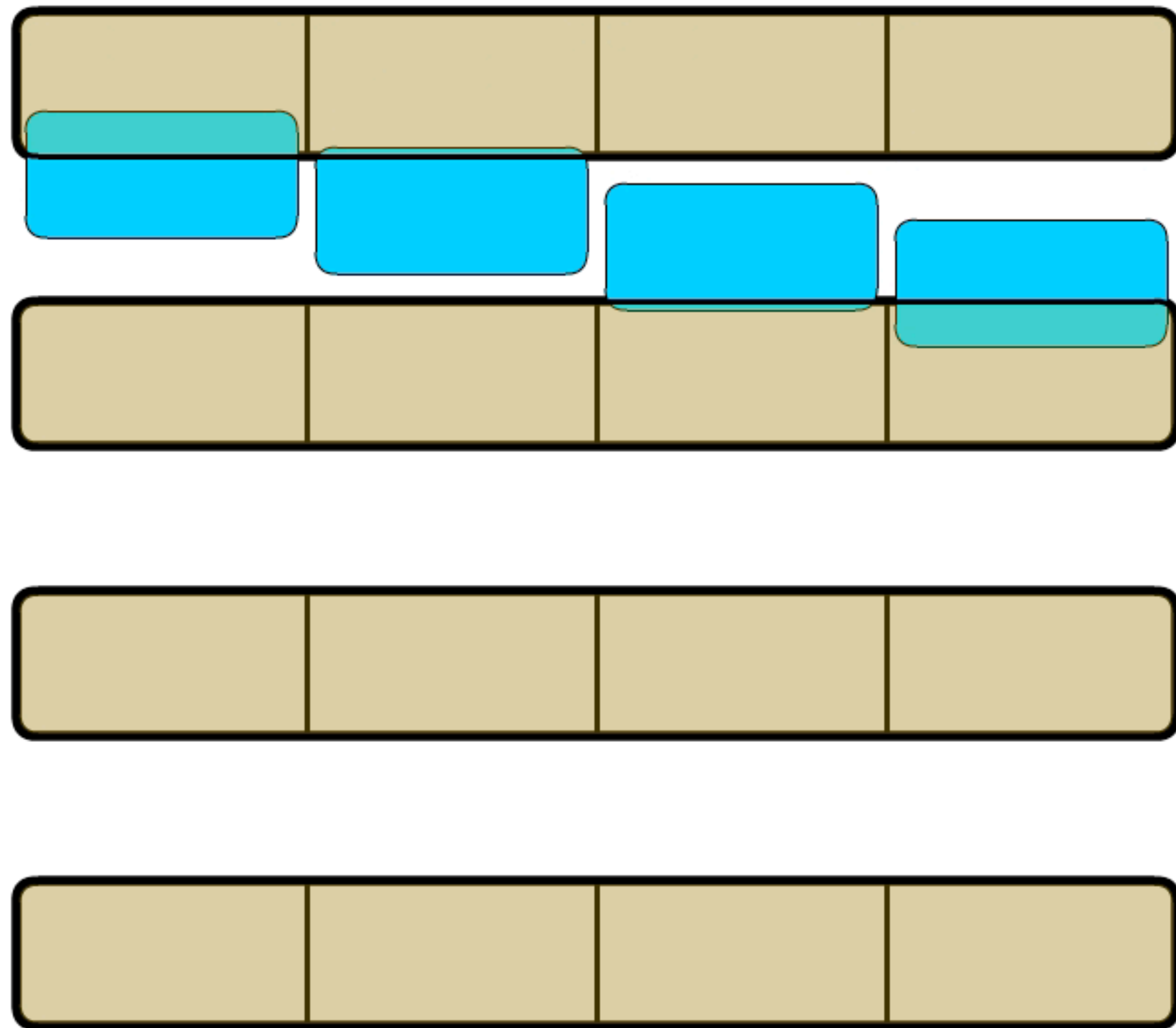


# Cache misses...

...the wildcard of perf



# What to SIMDize



Repetitive  
Independent  
Computations

# How to SIMDize

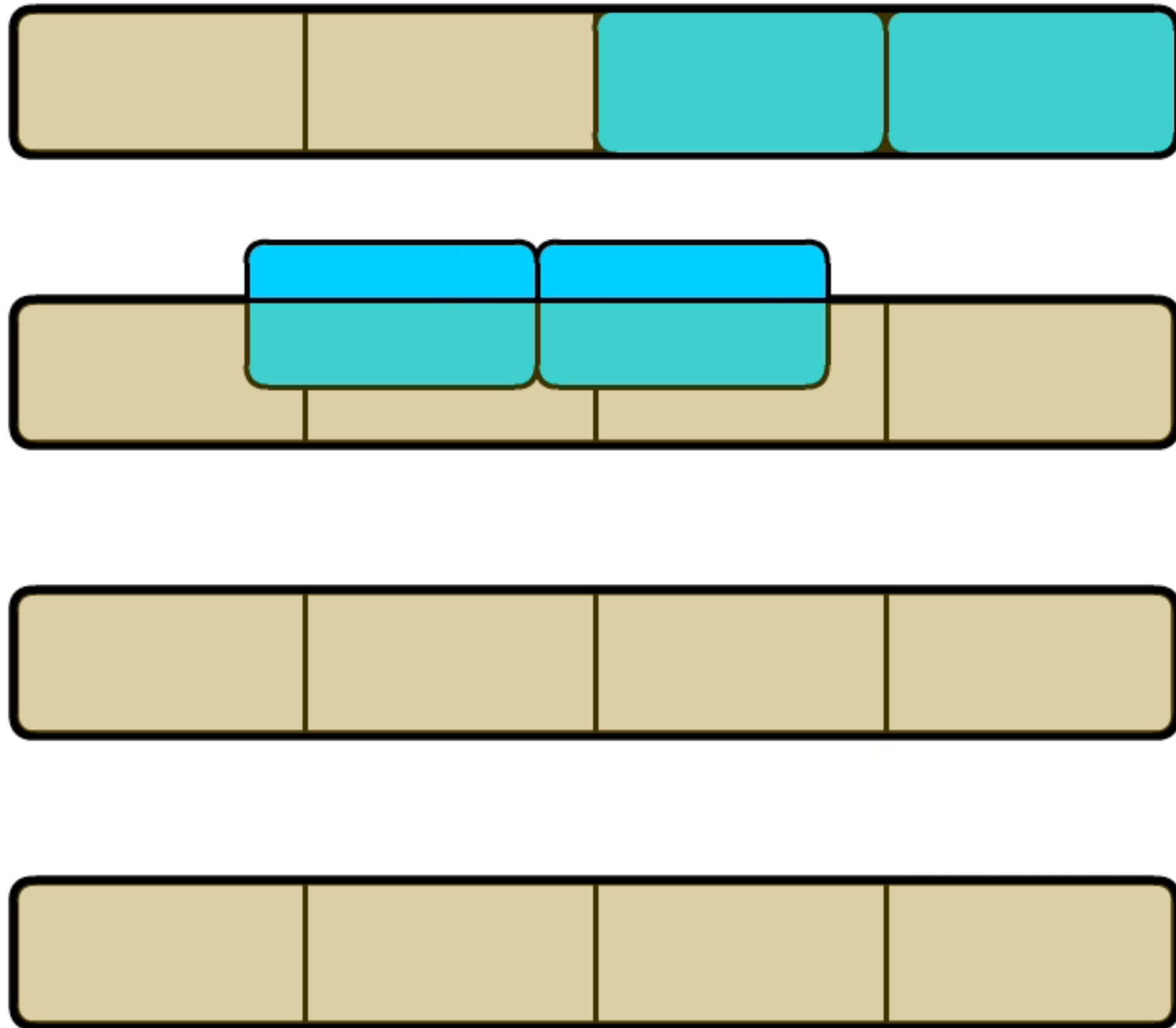
Debug Scalar code

Have No Branches

Interleave data

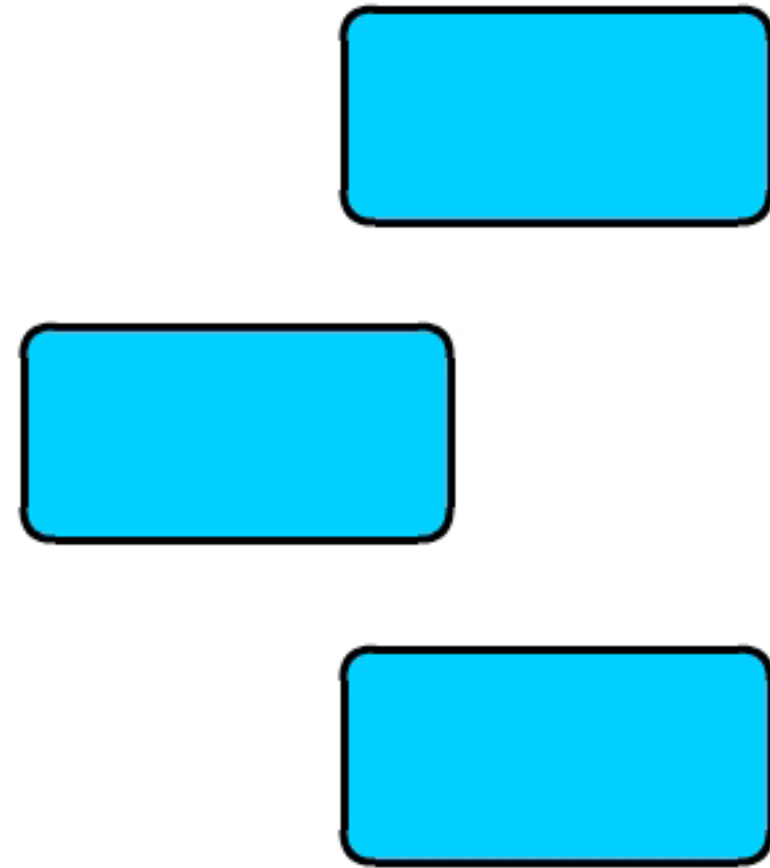
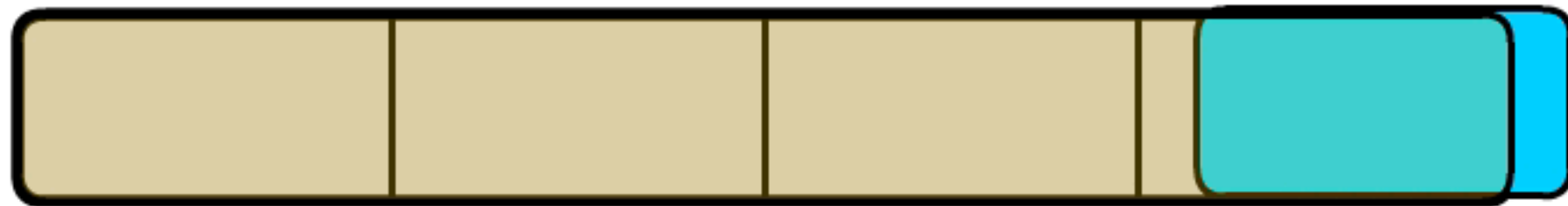
SIMDize code

# Gather



```
1  mov
2  mov
3  mov
4  mov
5  shuffle
6  shuffle
7  shuffle
```

# Scatter



```
8  shuffle
9  shuffle
10 shuffle
11 mov
12 mov
13 mov
14 mov
```

# Free Lunch for SIMD

Ports #0,#1,#5

**Full**

Port #2

**Partially Idle**

Port #3

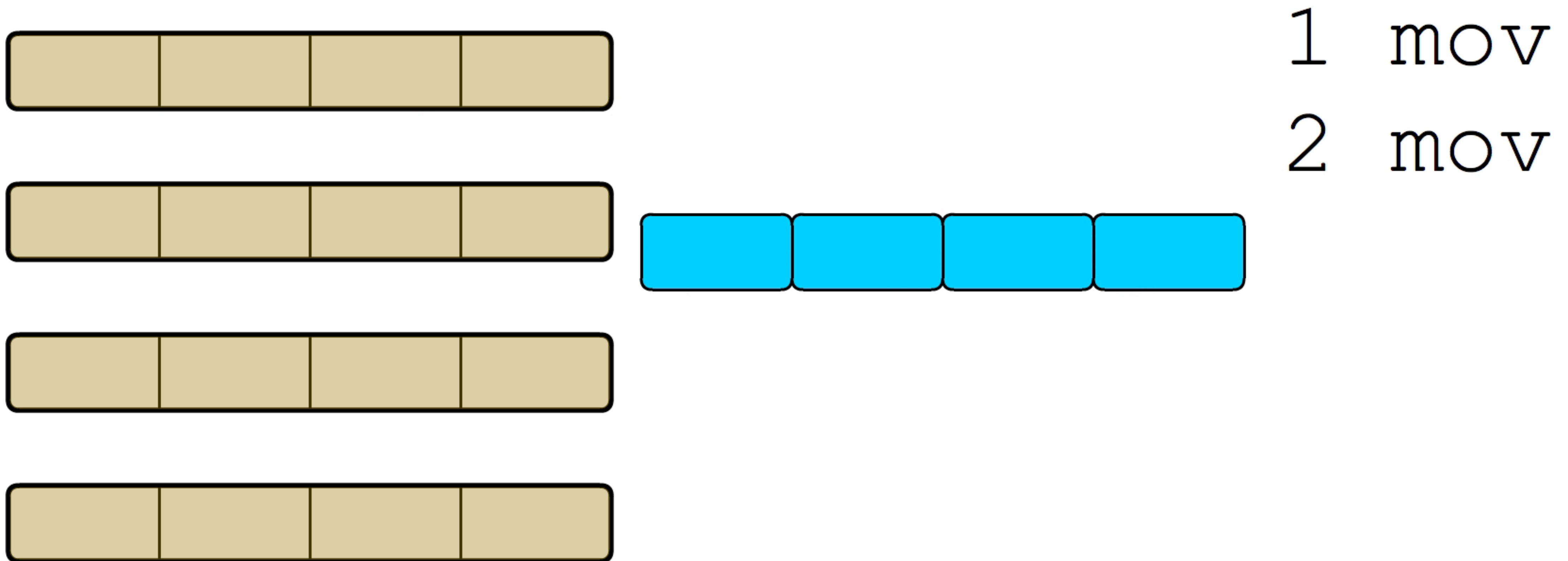
**Partially Idle**

Port #4

**Partially Idle**



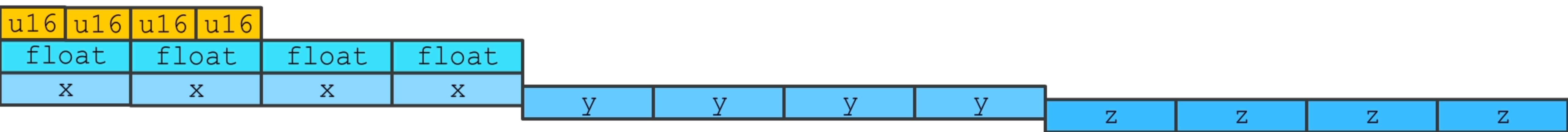
# Swizzle-Pack Data



# Gather-Scatter

Pack your data  
Gather-Scatter pipeline well  
1 op per byte

# Interleave



# template <typename Float>

```
template <typename Float >
inline SinCos< Float> ApproximateGivensRotation( const Float & a11, const Float & a12,
const Float & a22 )
{
    const Float two = Replicate<Float>( 2.0f );
    Float ch = two * ( a11 - a22 );
    Float sh = a12;
    typename FloatTraits<Float>::Bool b = CmpLt( Replicate<Float>( 5.82842712474619f ) *
sh*sh, ch*ch );
    Float omega = RsqrtEst( ch*ch + sh *sh );
    SinCos<Float>res;
    res.s = Select( Replicate<Float>( 0.3826834323650897717284599840304f ), omega * sh, b );
    res.c = Select( Replicate<Float>( 0.92387953251128675612818318939679f ), omega * ch, b );

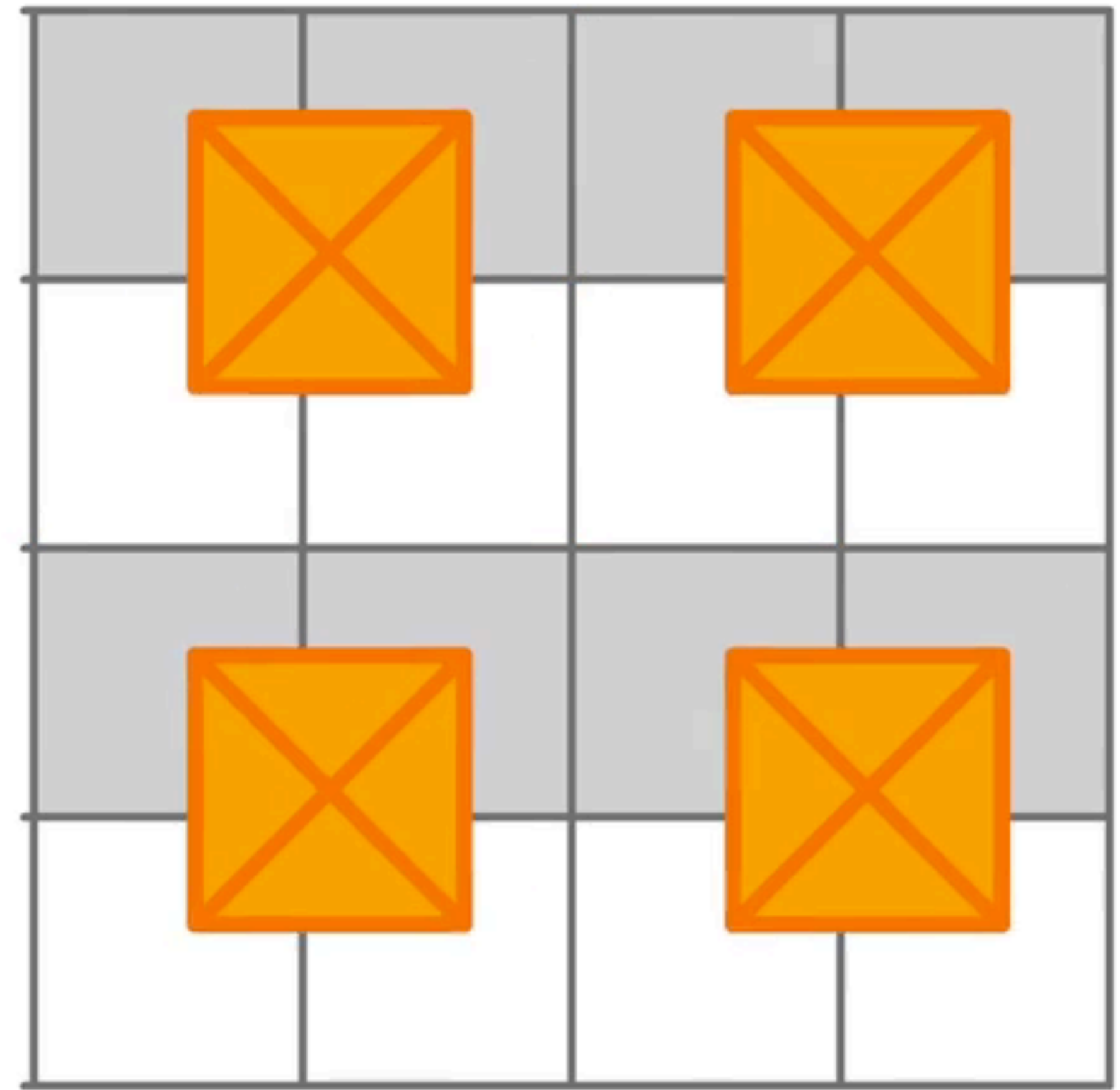
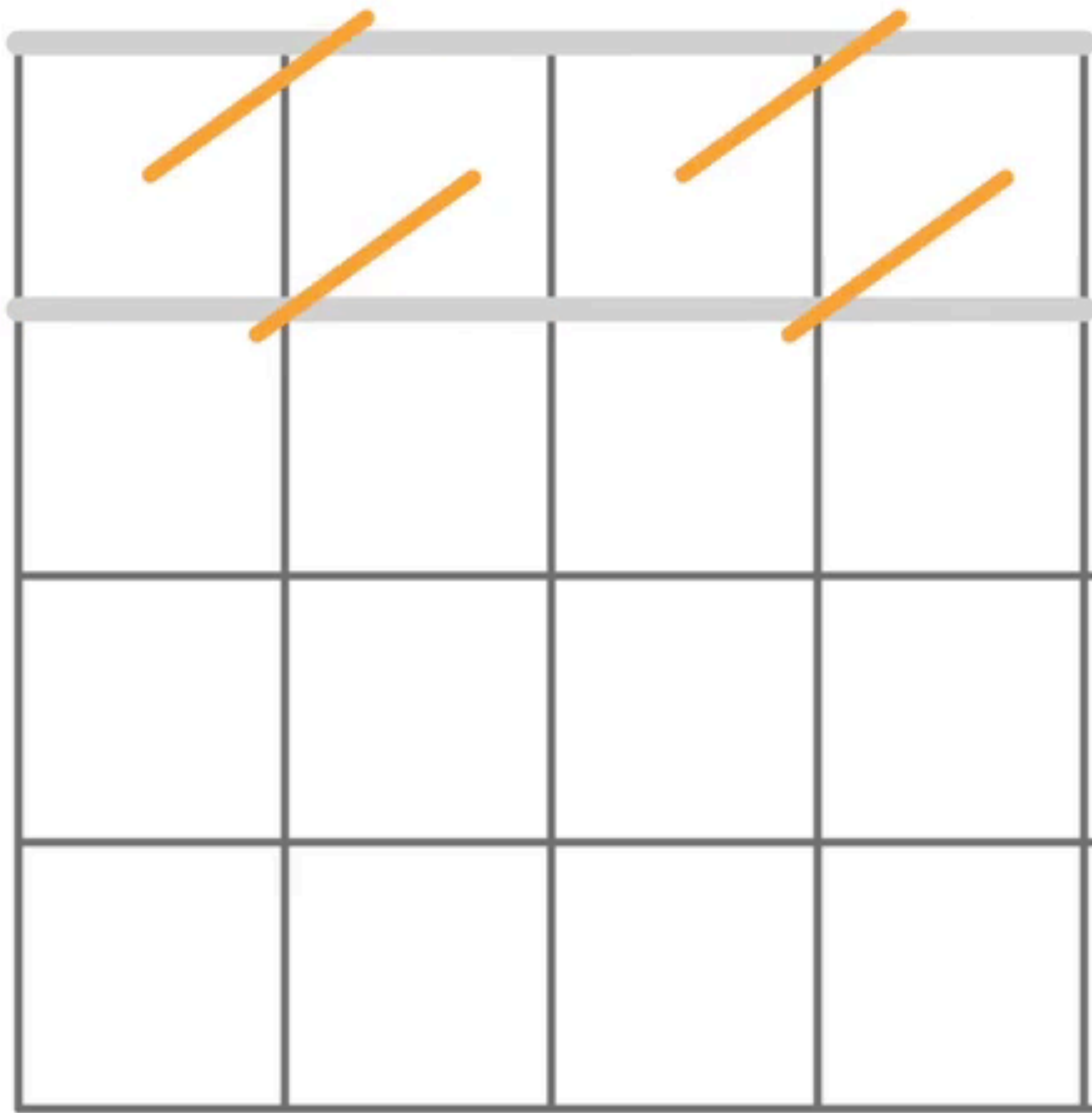
    return res;
}
```

# SIMD data structures

```
struct Rod
{
    uint16 nNode[ 2 ];
    float flMaxDist;
    float flMinDist;
    float flMassRatio;
    float flRelaxationFactor;
};
```

```
struct SimdRod
{
    uint16 nNode[ 2 ][ 4 ];
    float4 f4MaxDist;
    float4 f4MinDist;
    float4 f4MassRatio;
    float4 f4RelaxationFactor;
    void Init( const Rod pScalar[4] );
};
```

# Cloth SIMD order



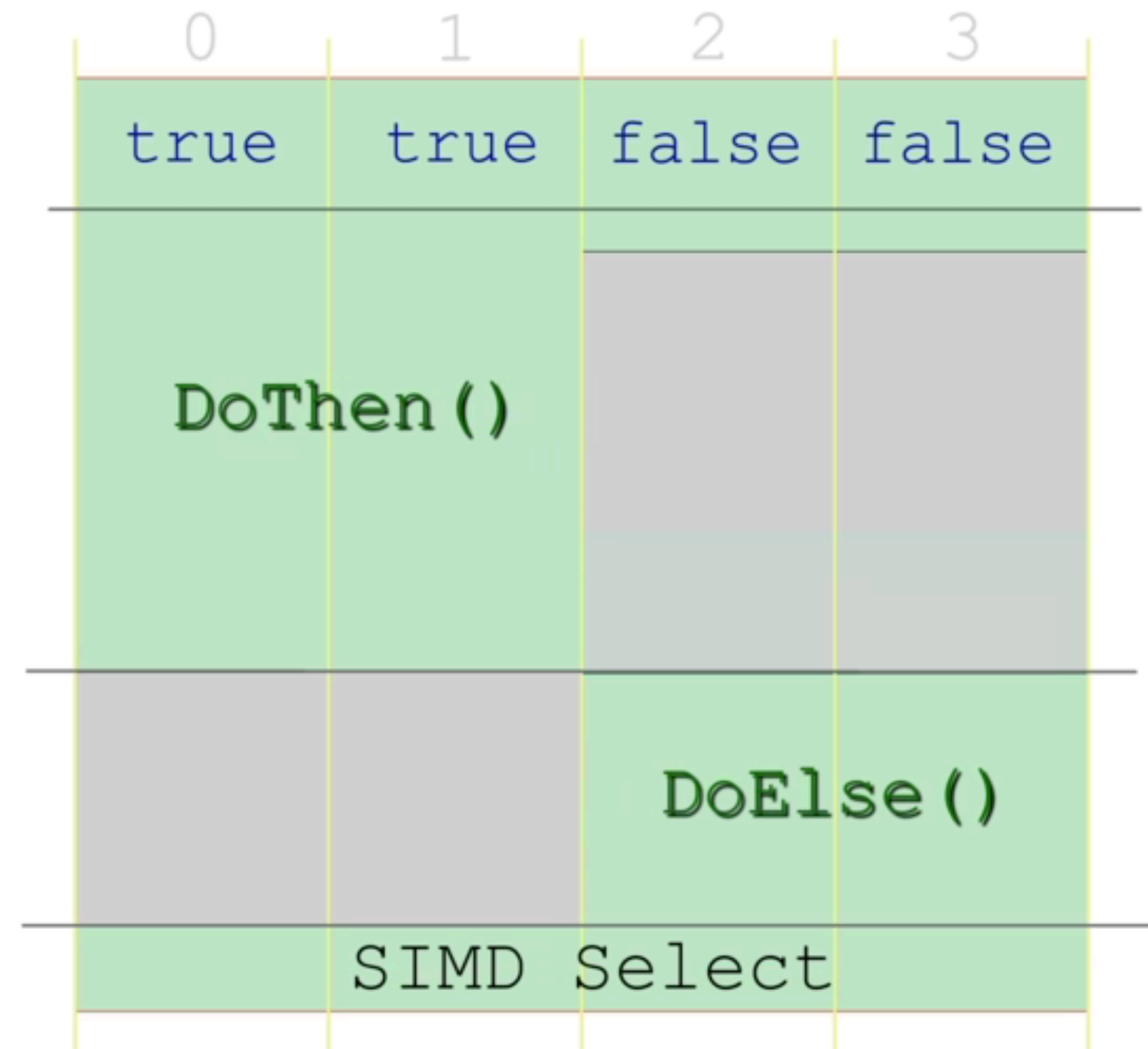
# SIMD lanes

- Homogenous
- Independent

Semi-Myth: SIMD is premature optimization

# SIMD branches (ala GPU)

```
if ( condition )  
    DoThen ();  
else  
    DoElse ();
```





# SIMD branches (multipass)

1. Sort into batches
2. Process batches
3. Merge results



# SIMD

## The Good:

- Almost 4x perf
- Separable=Easy
- AVX: 66% users
- Easy to Write

## The Bad:

- Hard to Retrofit
- Hard to Automate
- Requires Planning
- Hard to Read

# Performance Re-Cap

- Cache
- OOO
- RAM
- 1 byte per op
- Resource Pointers
- $O^*(\dots)$
- SIMD
- Multi-Thread

# Special Thanks

3D Art Support - Anna Bibikova

2D Art Support - Heather Campbell

# Links

Agner Fog Software Optimization Resources  
<http://agner.org>

Software optimization guide for AMD processors  
<http://support.amd.com/TechDocs/25112.PDF>

Intel® 64 and IA-32 Architectures Optimization Reference Manual  
<http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>

Intel® Architecture Code Analyzer  
<https://software.intel.com/en-us/articles/intel-architecture-code-analyzer>

The Art of Multiprocessor Programming

Intel VTune

VerySleepy

GlowCode

Luke Stackwalker

# Attribution

- Images of Z80: "Z80A-HD" by ZeptoBars - Licensed under CC BY 3.0 via Wikimedia Commons - <http://commons.wikimedia.org/wiki/File:Z80A-HD.jpg#mediaviewer/File:Z80A-HD.jpg>
- Image of Haswell wafer - licensed under the Creative Commons Attribution 2.0 Generic license. [http://en.wikipedia.org/wiki/Haswell\\_\(microarchitecture\)#mediaviewer/File:Haswell\\_Chip.jpg](http://en.wikipedia.org/wiki/Haswell_(microarchitecture)#mediaviewer/File:Haswell_Chip.jpg)
- Image of Radio-86RK: created by Audriusa, available under the Creative Commons CC0 1.0 Universal Public Domain Dedication. <https://commons.wikimedia.org/wiki/File:Radio86RK.png>
- Listing of 24-bit float multiplication for Z80: <https://drive.google.com/folderview?id=0B4HNIXQZLWM8Z3NQMGJTbHVhRm8&usp=sharing>, licensed for free use and distribution
- ZX Spectrum picture - [http://commons.wikimedia.org/wiki/File:ZX\\_Spectrum48k.jpg](http://commons.wikimedia.org/wiki/File:ZX_Spectrum48k.jpg) - CC2.5

# Performance Recipe

- Better Algorithm!
- 1 byte per op
- Stream
- Prefetch
- SIMDize
- Multithread
- Profit!

# Videos, Links and Errata

<http://sergiy.space>