GAMES 201

Advanced Physics Engines 2020: A Hands-on Tutorial

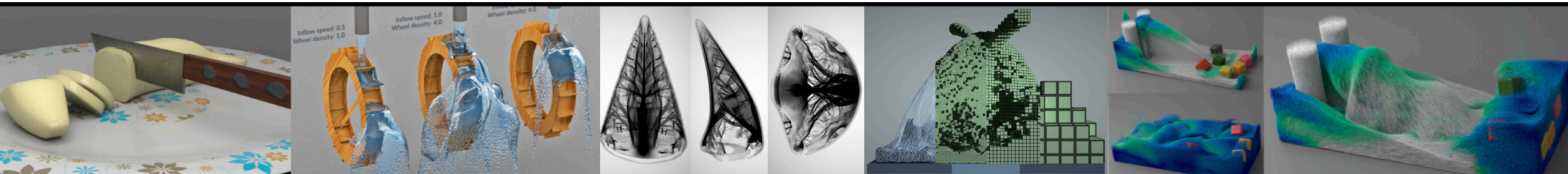
高级物理引擎实战2020 （基于太极编程语言）

第九讲：高性能物理模拟

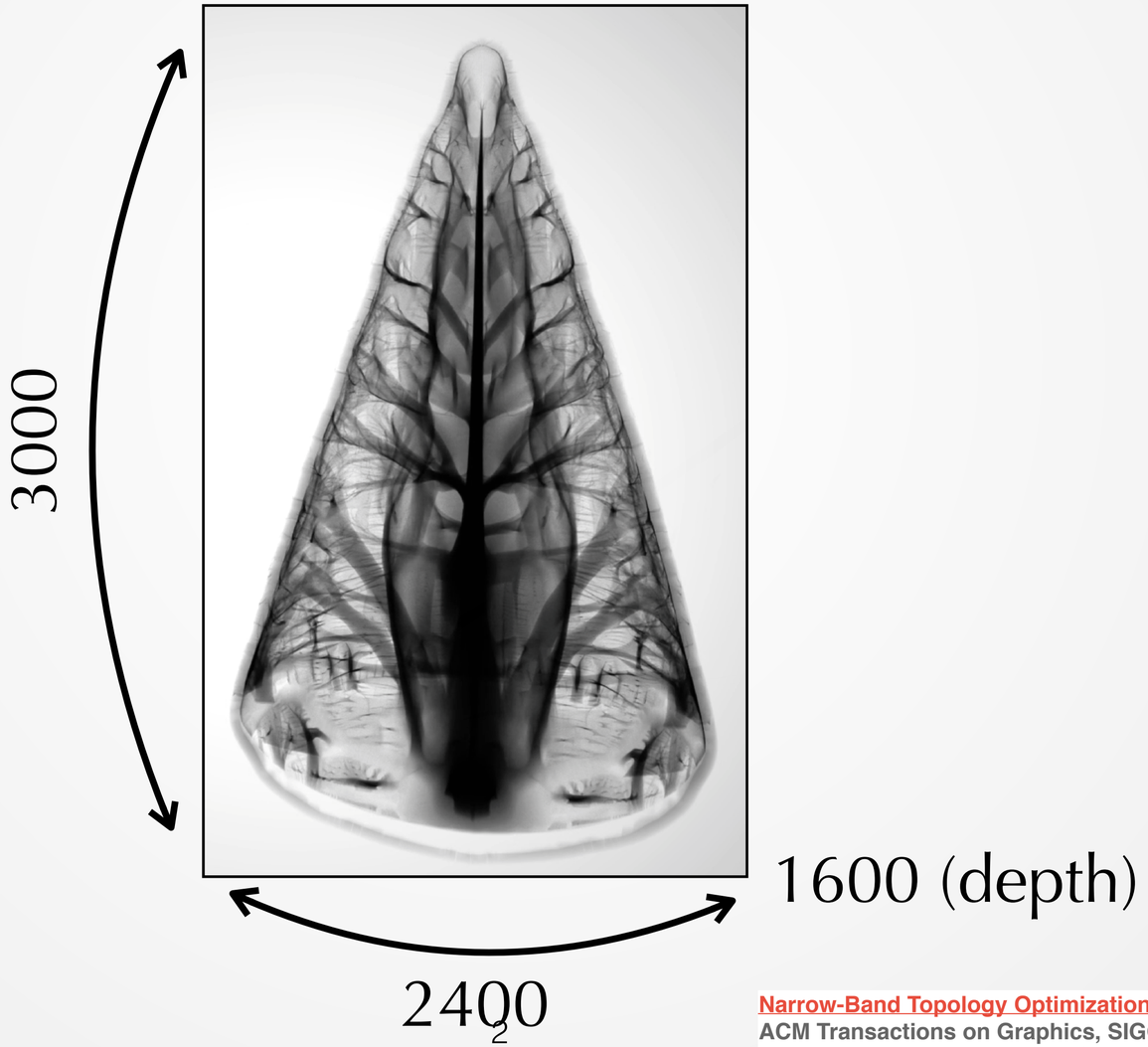
Yuanming Hu 胡渊鸣 MIT CSAIL

麻省理工学院 计算机科学与人工智能实验室

Taichi Programming Language

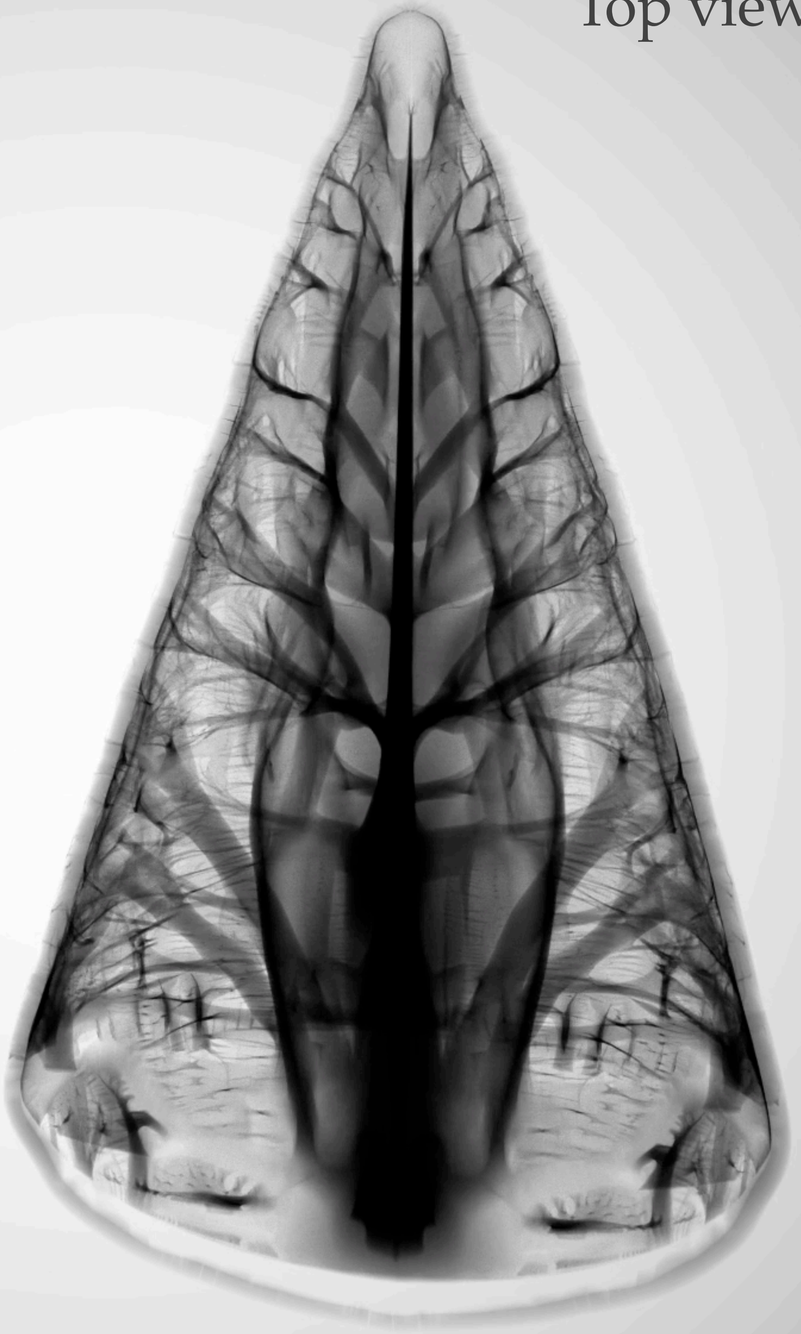


11,520,000,000 background voxels
1,040,875,347 active voxels

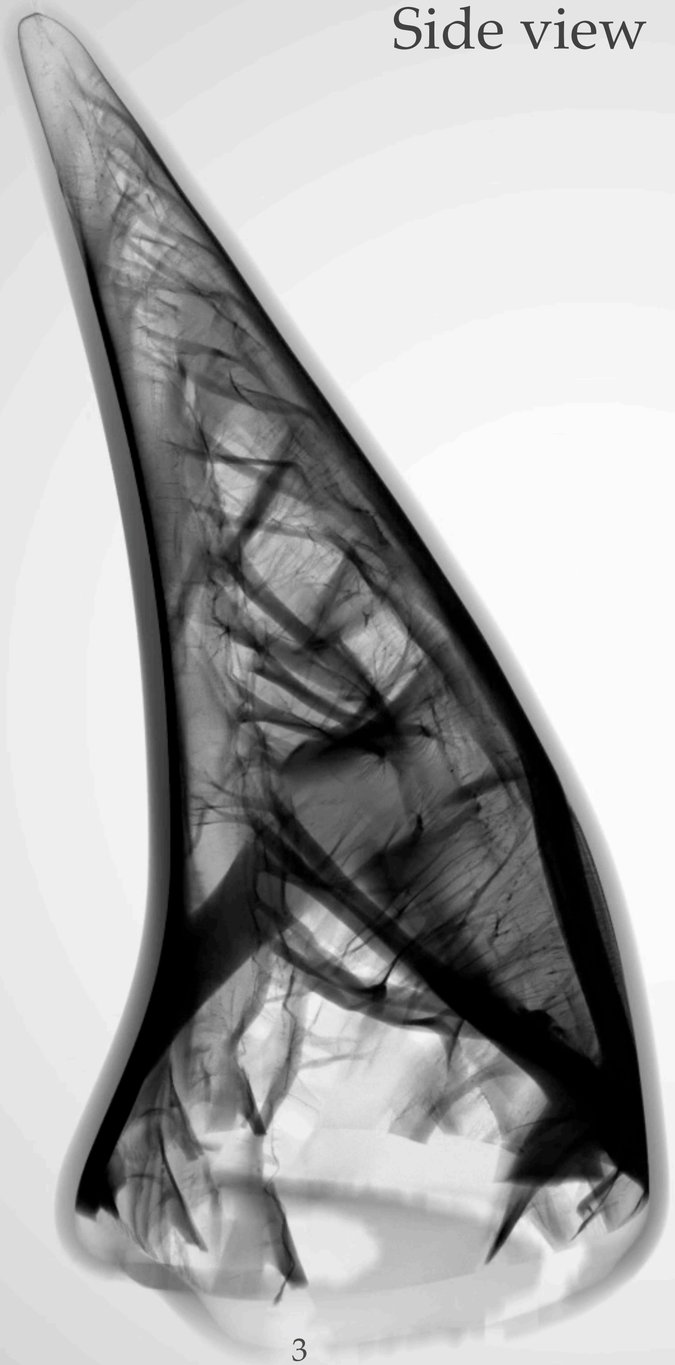


Narrow-Band Topology Optimization on a Sparsely Populated Grid
ACM Transactions on Graphics, SIGGRAPH Asia 2018
Haixiang Liu, Yuanming Hu* (joint first authors), Bo Zhu, Wojciech Matusik, Eftychios Sifakis*

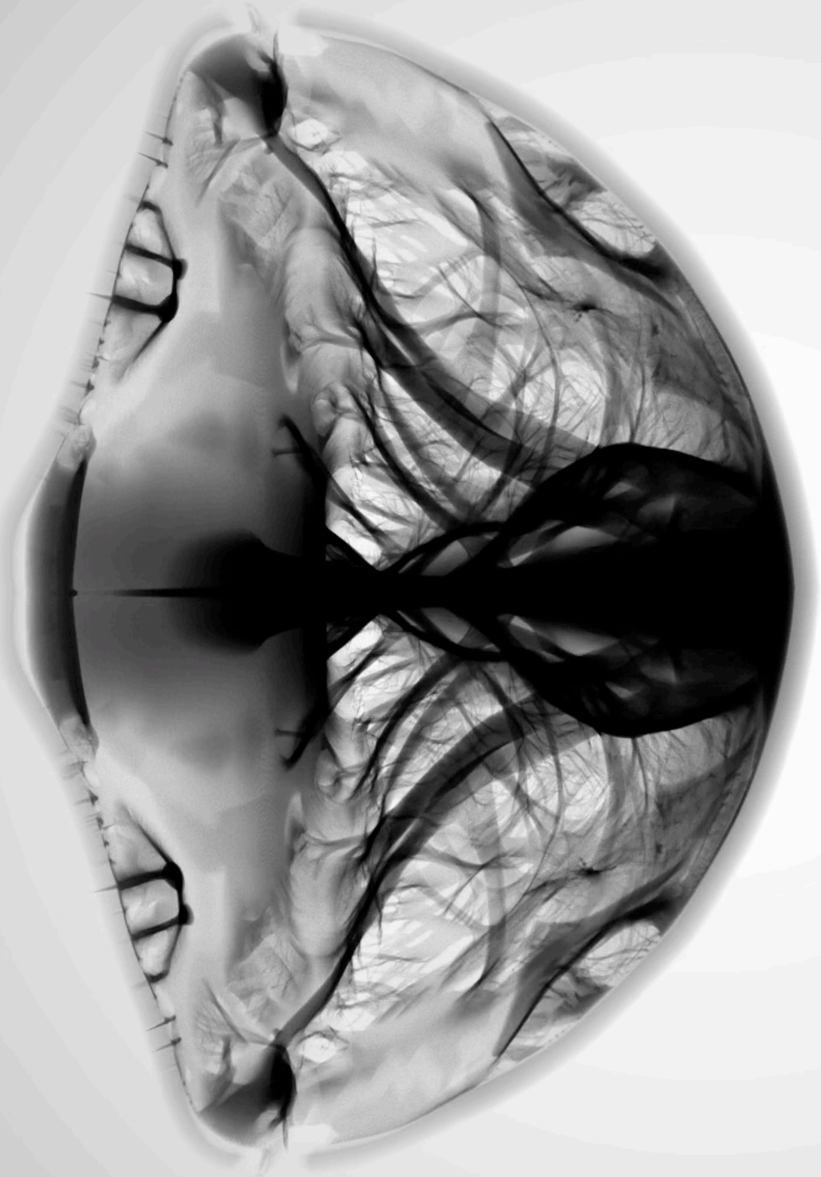
Top view



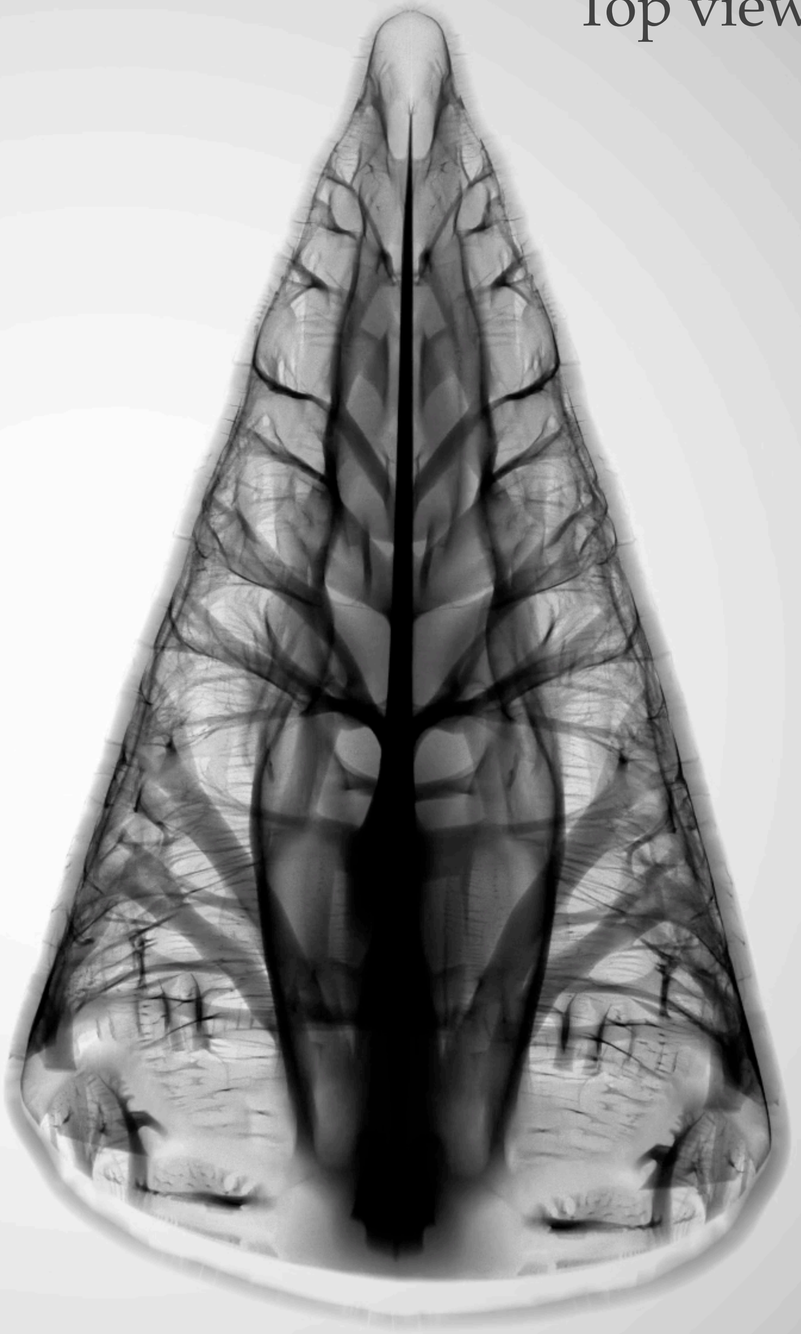
Side view



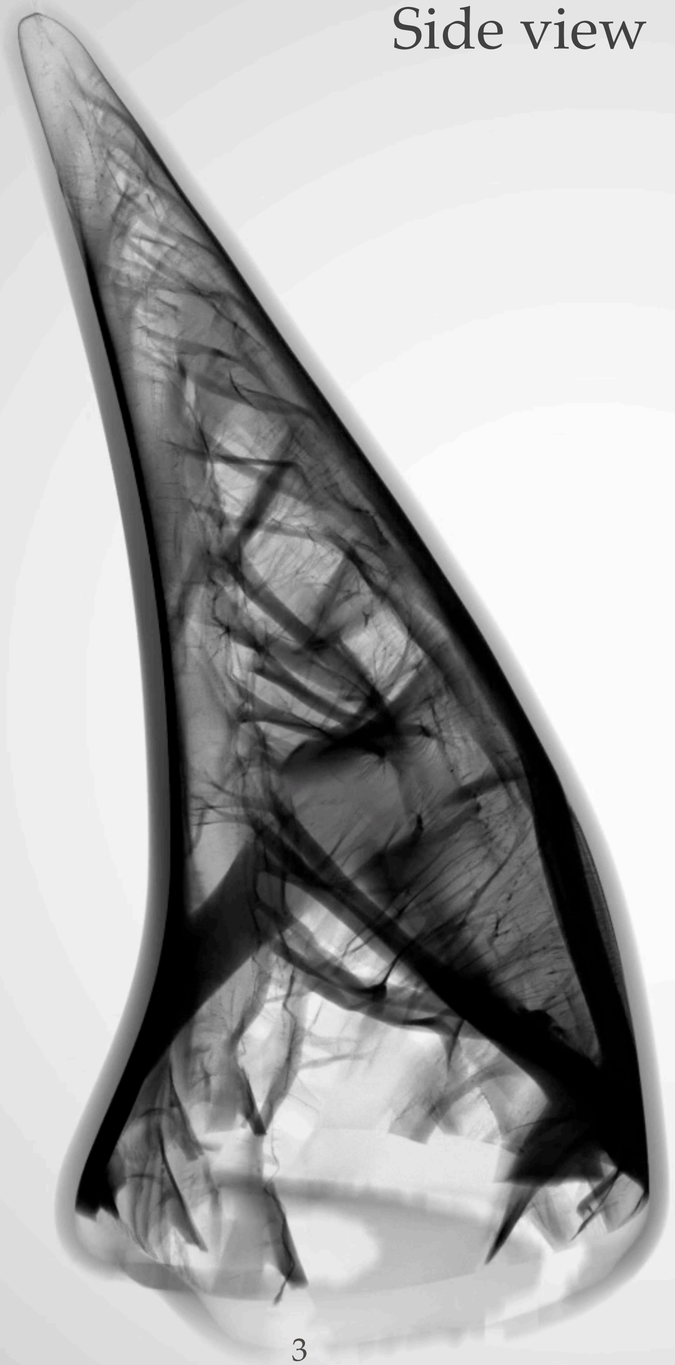
Back view



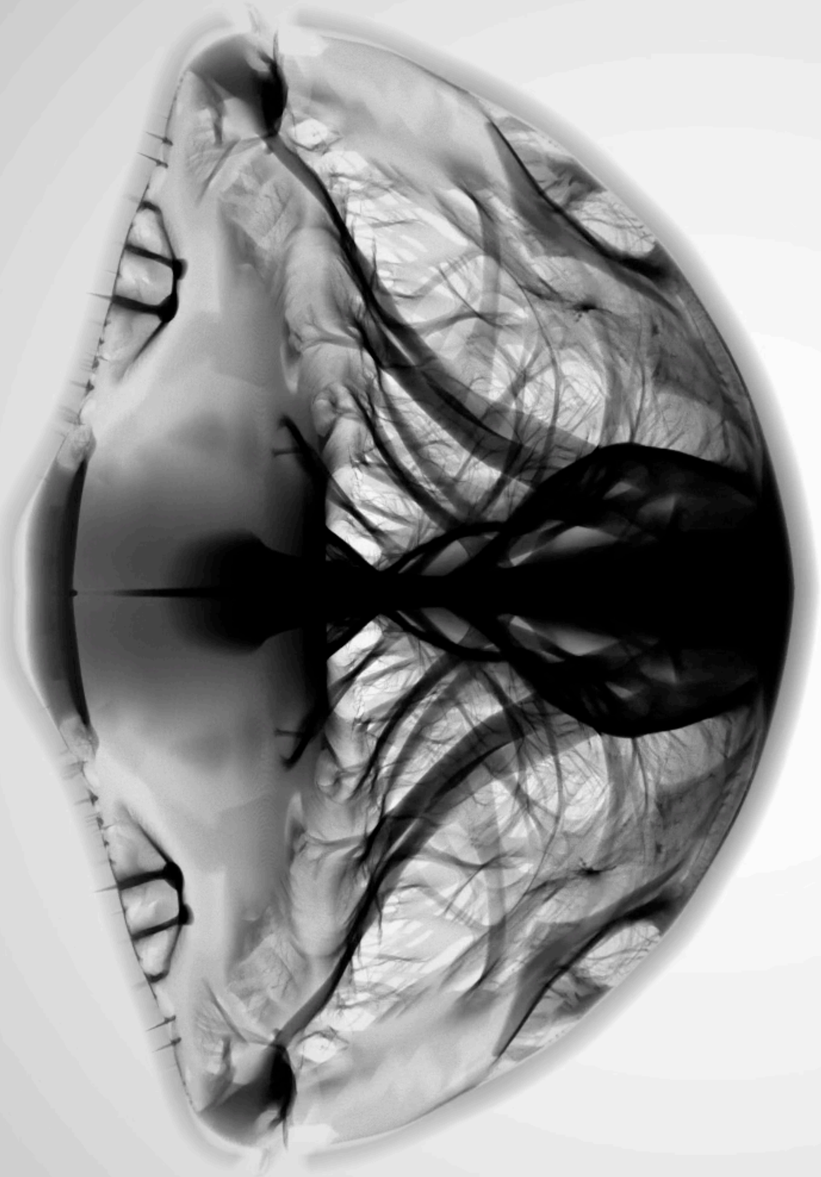
Top view



Side view



Back view



Motivation

- **Physical simulation is computationally expensive**
  - A single simulation demo for SIGGRAPH can take days to run
- **Performance is key to high quality**
  - Performance from algorithmic improvement (do **less** work)
    - quick sort v.s. insertion sort; O(n lg n) v.s. O(n^2)
    - Pre-conditioners for linear system solvers that make things converge faster
  - Performance from low-level programming (do work **faster**)

Performance Engineering Example

Timing (ms)	Reference	Ours (MPM)	Ours* (MPM)	Ours* (MLS-MPM)
P2G (1 thread)	4760 (1×)	5744 (0.83×)	2685 (1.77×)	1283 (3.71×)
P2G (4 threads)	1220 (1×)	1525 (0.80×)	688 (1.77×)	328 (3.72×)
G2P (1 thread)	8255 (1×)	7476 (1.10×)	1144 (7.21×)	589 (14.01×)
G2P (4 threads)	2070 (1×)	2011 (1.03×)	313 (6.61×)	163 (12.70×)

Reference: Tampubolon et al. 2017.
Multi-species simulation of porous sand and water mixtures
(Material Point Method, MPM)

Performance Engineering Example

Timing (ms)	Reference	Ours (MPM)	Ours* (MPM)	Ours* (MLS-MPM)
P2G (1 thread)	4760 (1×)	5744 (0.83×)	2685 (1.77×)	1283 (3.71×)
P2G (4 threads)	1220 (1×)	1525 (0.80×)	688 (1.77×)	328 (3.72×)
G2P (1 thread)	8255 (1×)	7476 (1.10×)	1144 (7.21×)	589 (14.01×)
G2P (4 threads)	2070 (1×)	2011 (1.03×)	313 (6.61×)	163 (12.70×)

Baseline: Traditional MPM

Performance Engineering Example

Timing (ms)	Reference	Ours (MPM)	Ours* (MPM)	Ours* (MLS-MPM)
P2G (1 thread)	4760 (1×)	5744 (0.83×)	2685 (1.77×)	1283 (3.71×)
P2G (4 threads)	1220 (1×)	1525 (0.80×)	688 (1.77×)	328 (3.72×)
G2P (1 thread)	8255 (1×)	7476 (1.10×)	1144 (7.21×)	589 (14.01×)
G2P (4 threads)	2070 (1×)	2011 (1.03×)	313 (6.61×)	163 (12.70×)

Optimized Traditional MPM

(Low-level performance engineering, **7x speed up**)

Performance Engineering Example

Timing (ms)	Reference	Ours (MPM)	Ours* (MPM)	Ours* (MLS-MPM)
P2G (1 thread)	4760 (1×)	5744 (0.83×)	2685 (1.77×)	1283 (3.71×)
P2G (4 threads)	1220 (1×)	1525 (0.80×)	688 (1.77×)	328 (3.72×)
G2P (1 thread)	8255 (1×)	7476 (1.10×)	1144 (7.21×)	589 (14.01×)
G2P (4 threads)	2070 (1×)	2011 (1.03×)	313 (6.61×)	163 (12.70×)

Optimized MLS-MPM

(algorithmic improvement)
2x speedup

Performance Engineering Example

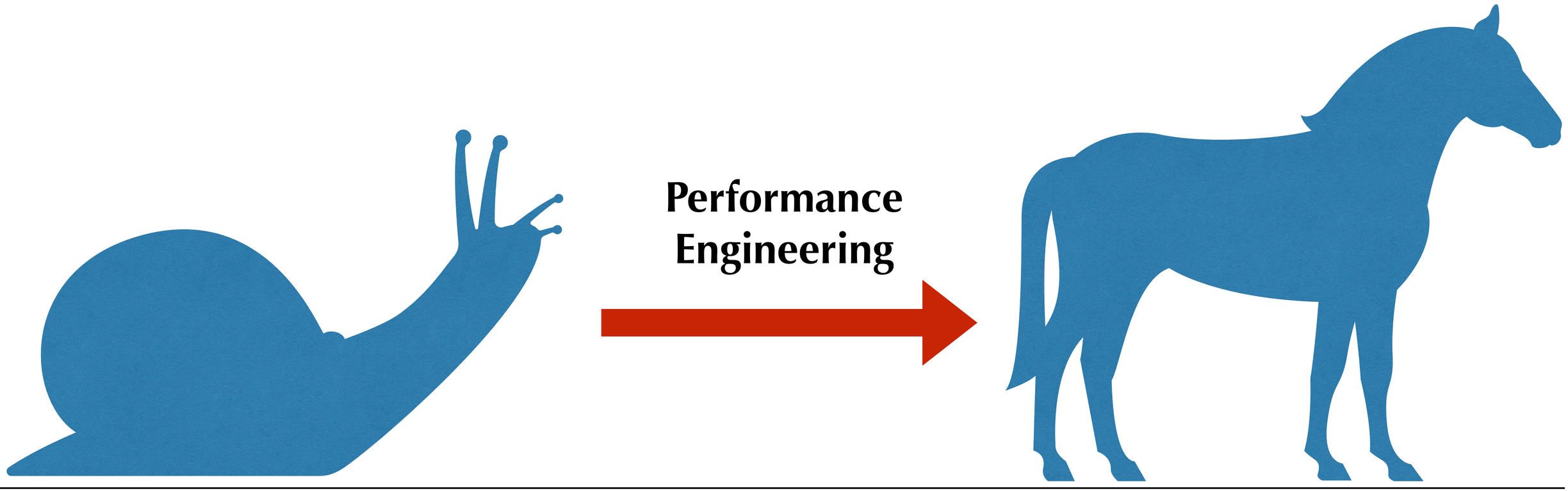
Timing (ms)	Reference	Ours (MPM)	Ours* (MPM)	Ours* (MLS-MPM)
P2G (1 thread)	4760 (1×)	5744 (0.83×)	2685 (1.77×)	1283 (3.71×)
P2G (4 threads)	1220 (1×)	1525 (0.80×)	688 (1.77×)	328 (3.72×)
G2P (1 thread)	8255 (1×)	7476 (1.10×)	1144 (7.21×)	589 (14.01×)
G2P (4 threads)	2070 (1×)	2011 (1.03×)	313 (6.61×)	163 (12.70×)

Baseline

7x faster
performance engineering
(Messy implementation details in the supplemental document. No one cares?)

2x faster
improved algorithm
(Major contribution claimed in the paper, Reviewers love it!)

Today's Topic



Normal Code

High-Performance Code:
5-20x faster

Table of Contents

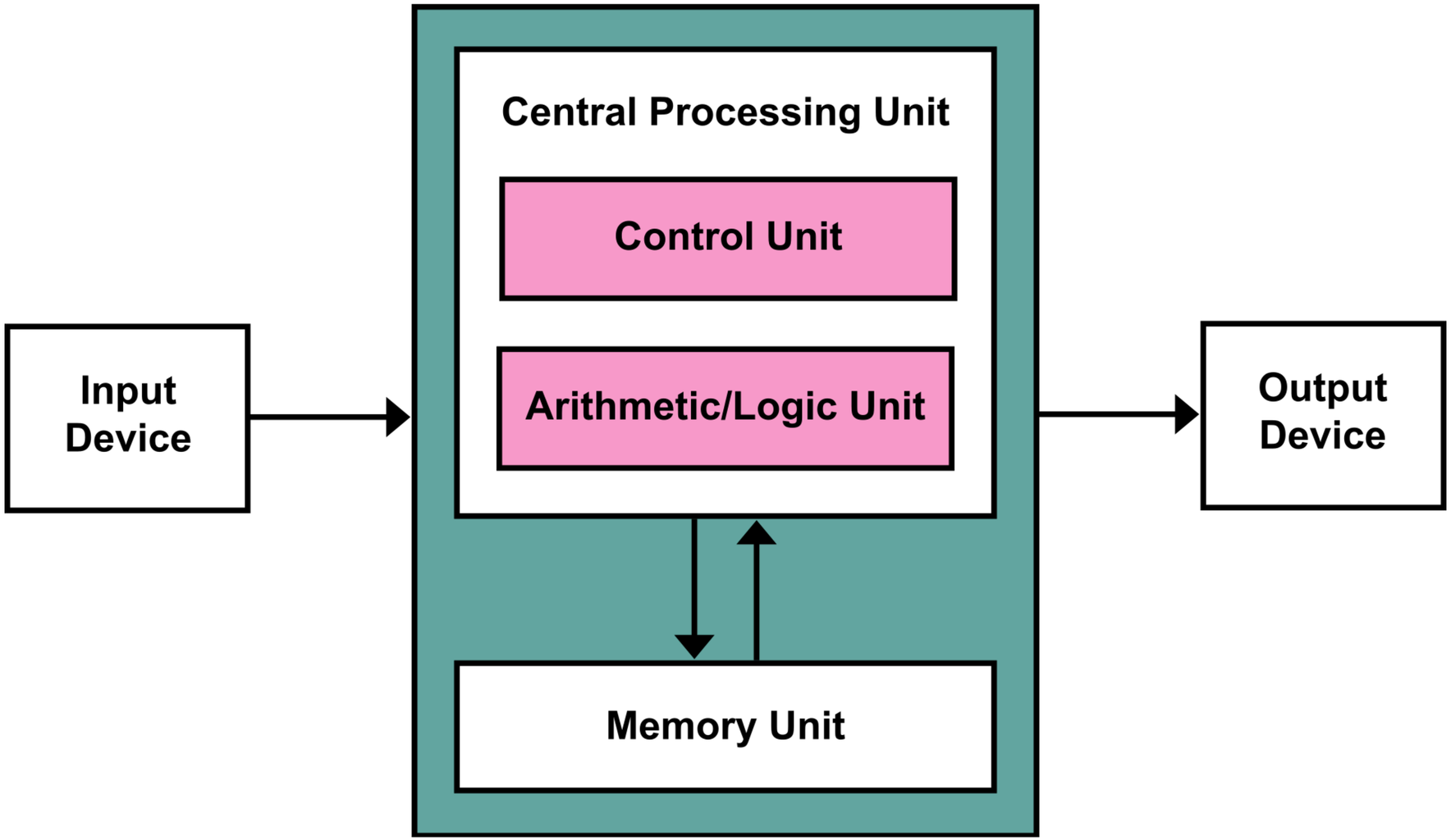
Part I.
Hardware Architecture

- ✦ Background
- ✦ Memory Hierarchy
- ✦ CPU Microarchitecture
- ✦ Quantitative Analysis

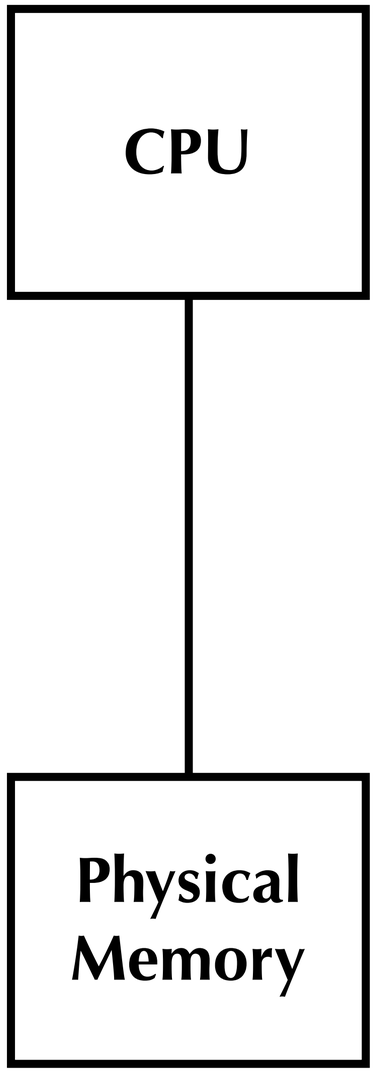
Part II.
Advanced Taichi Programming

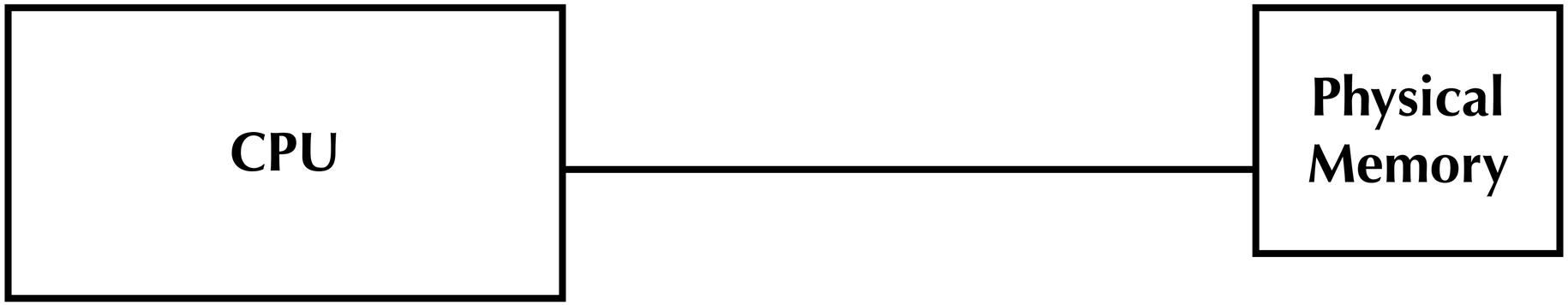
- ✦ Describing data layouts
- ✦ Sparse programming

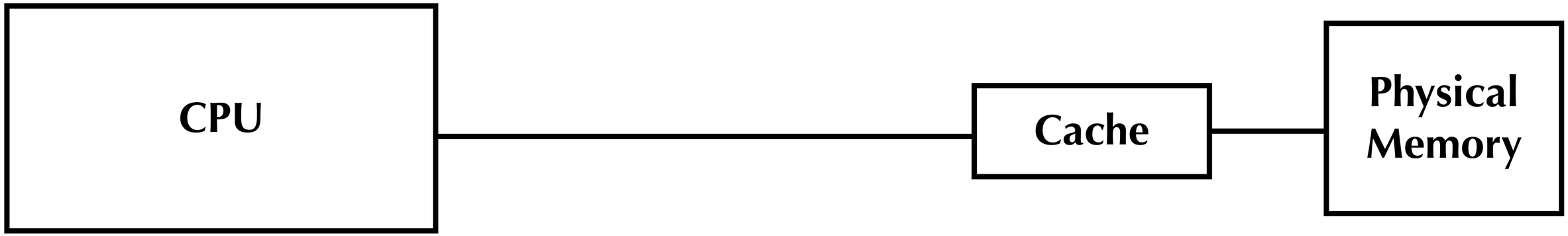
The von Neumann Architecture (1945)

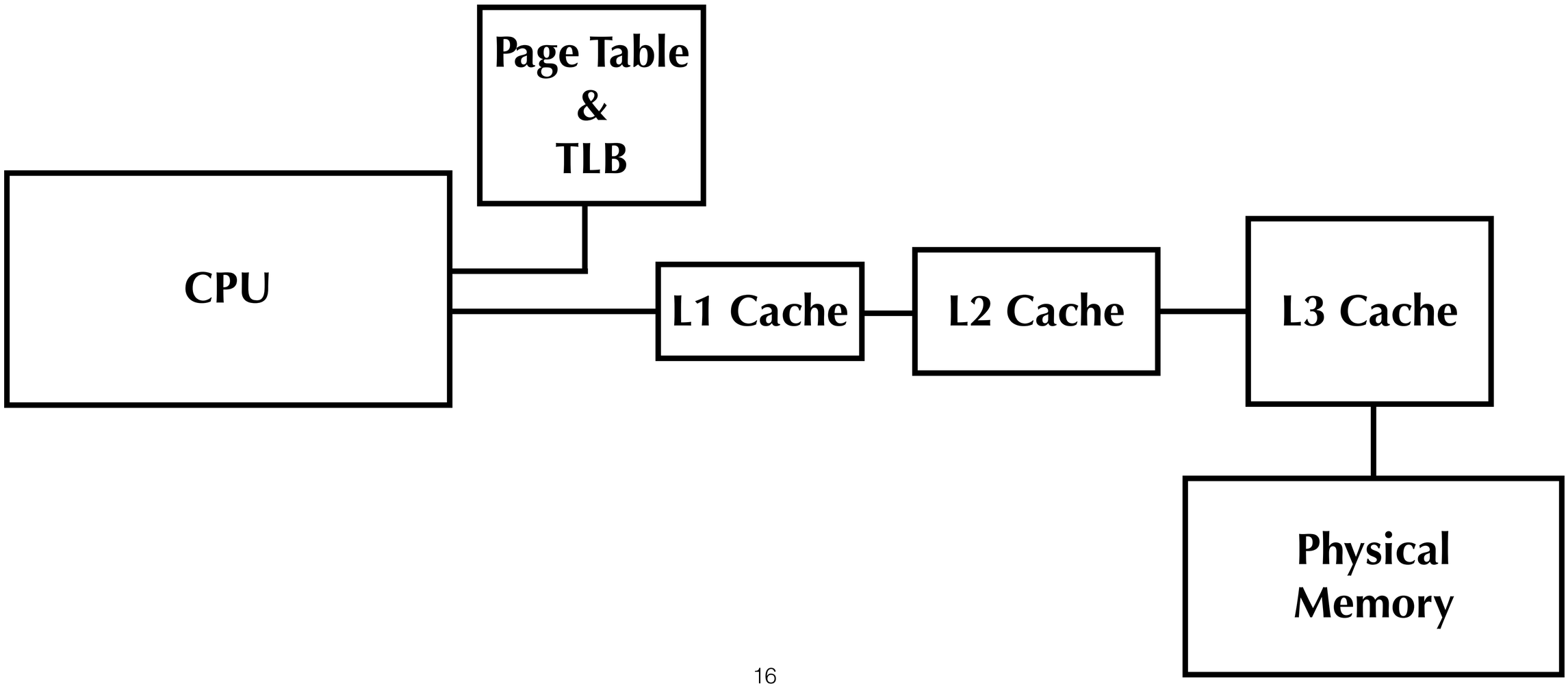


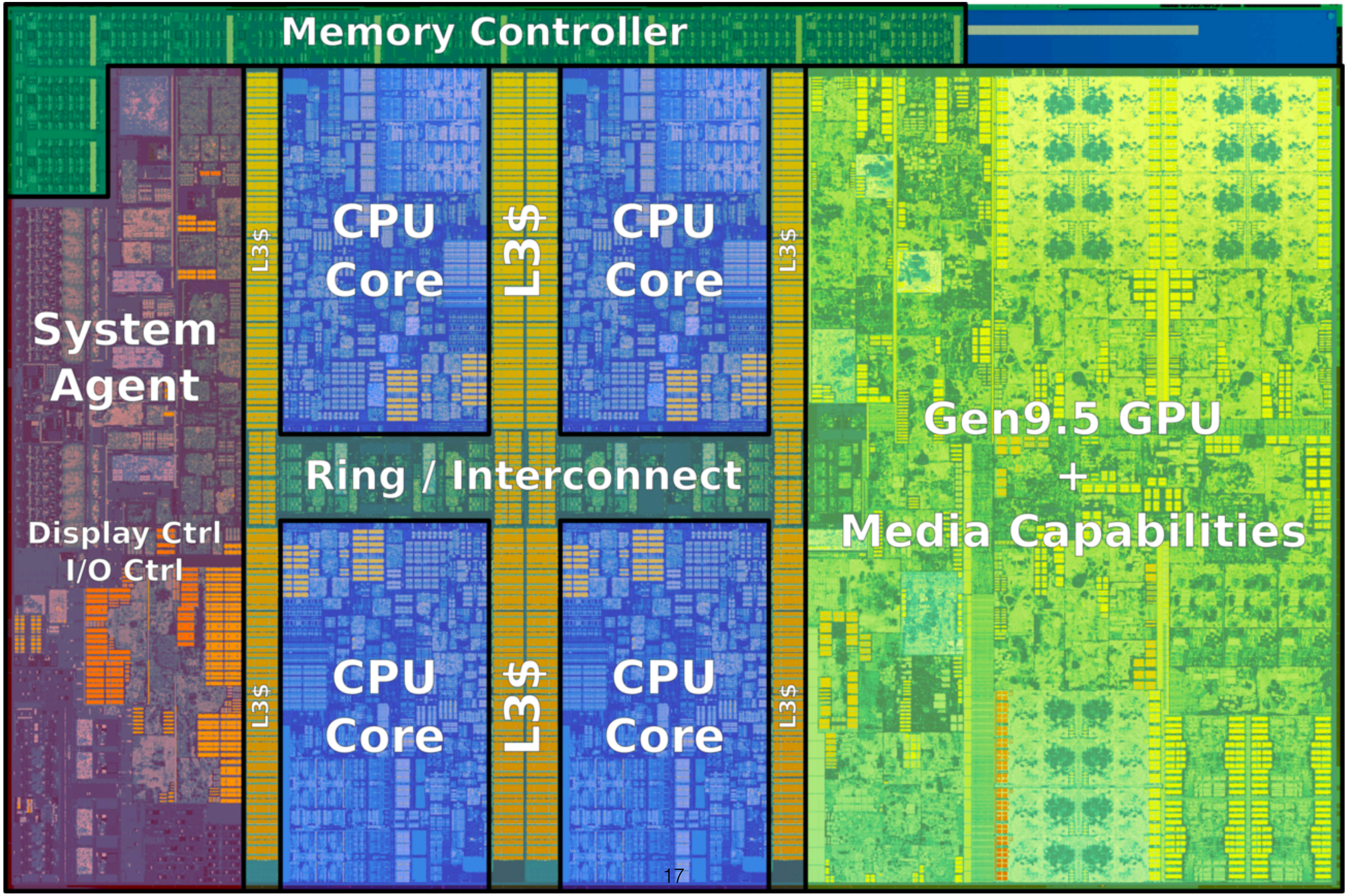
The von Neumann Architecture (1945)

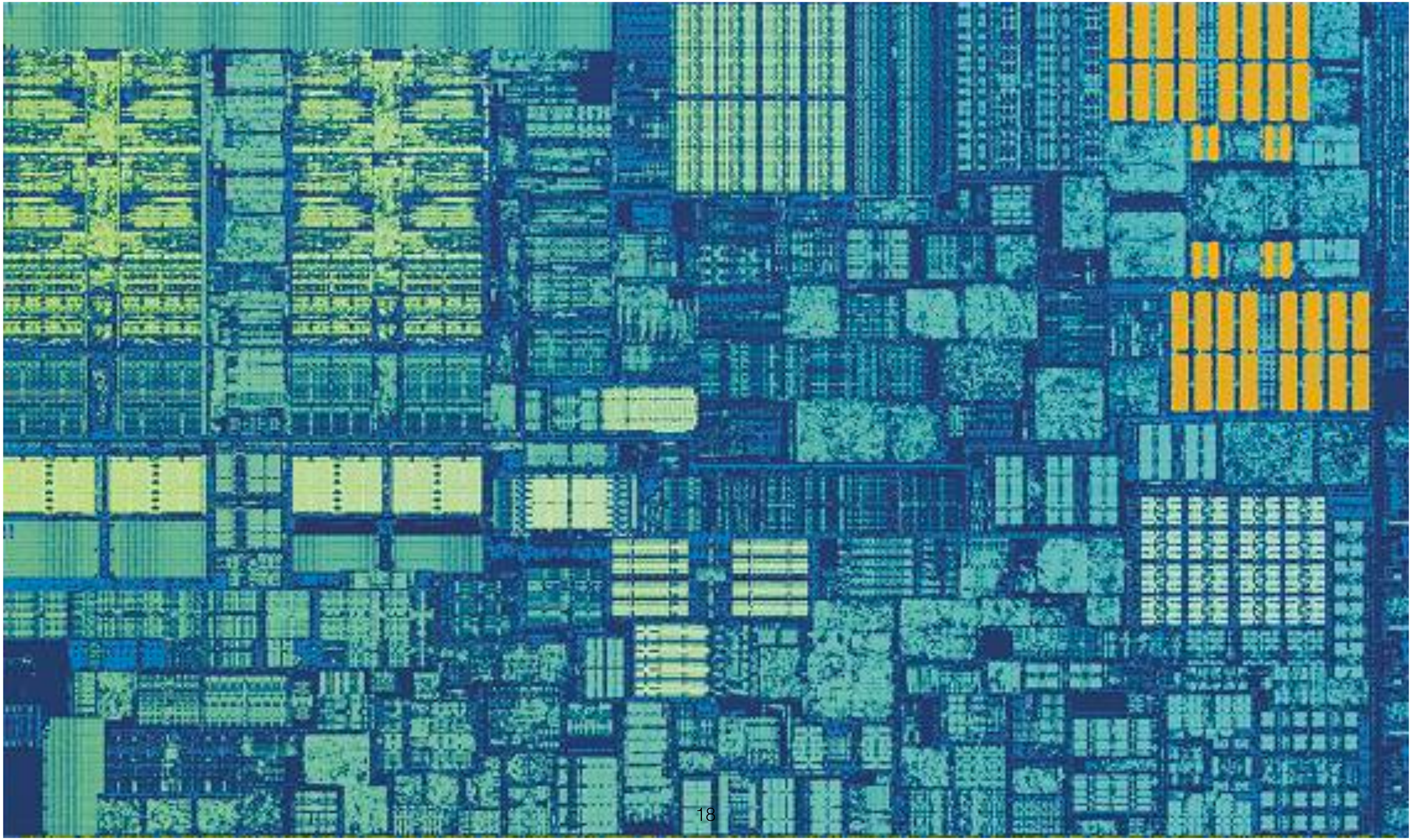



CPU
Physical
Memory

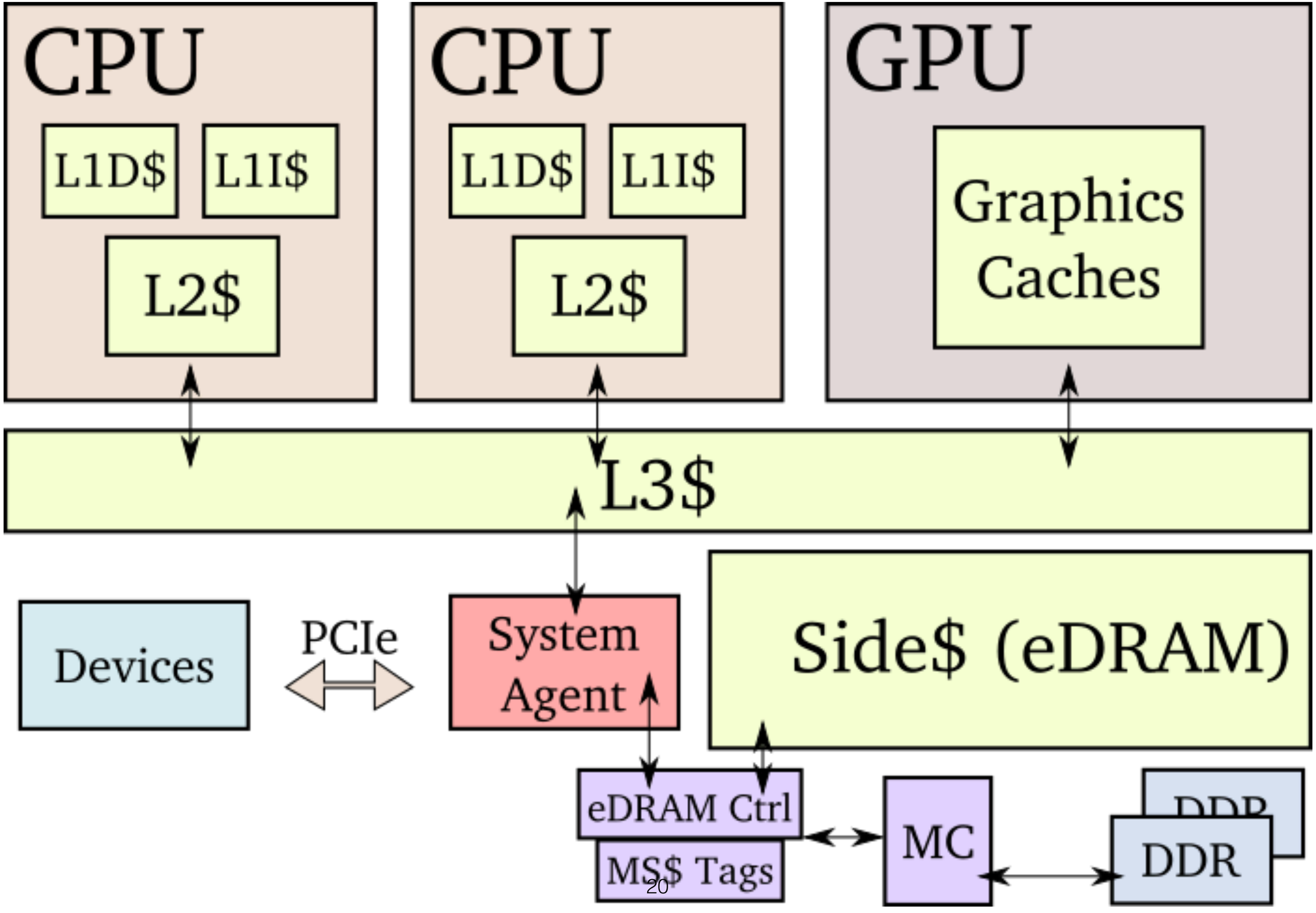

CPU
Cache
Physical
Memory


Page Table
&
TLB
CPU
L1 Cache
L2 Cache
L3 Cache
Physical
Memory

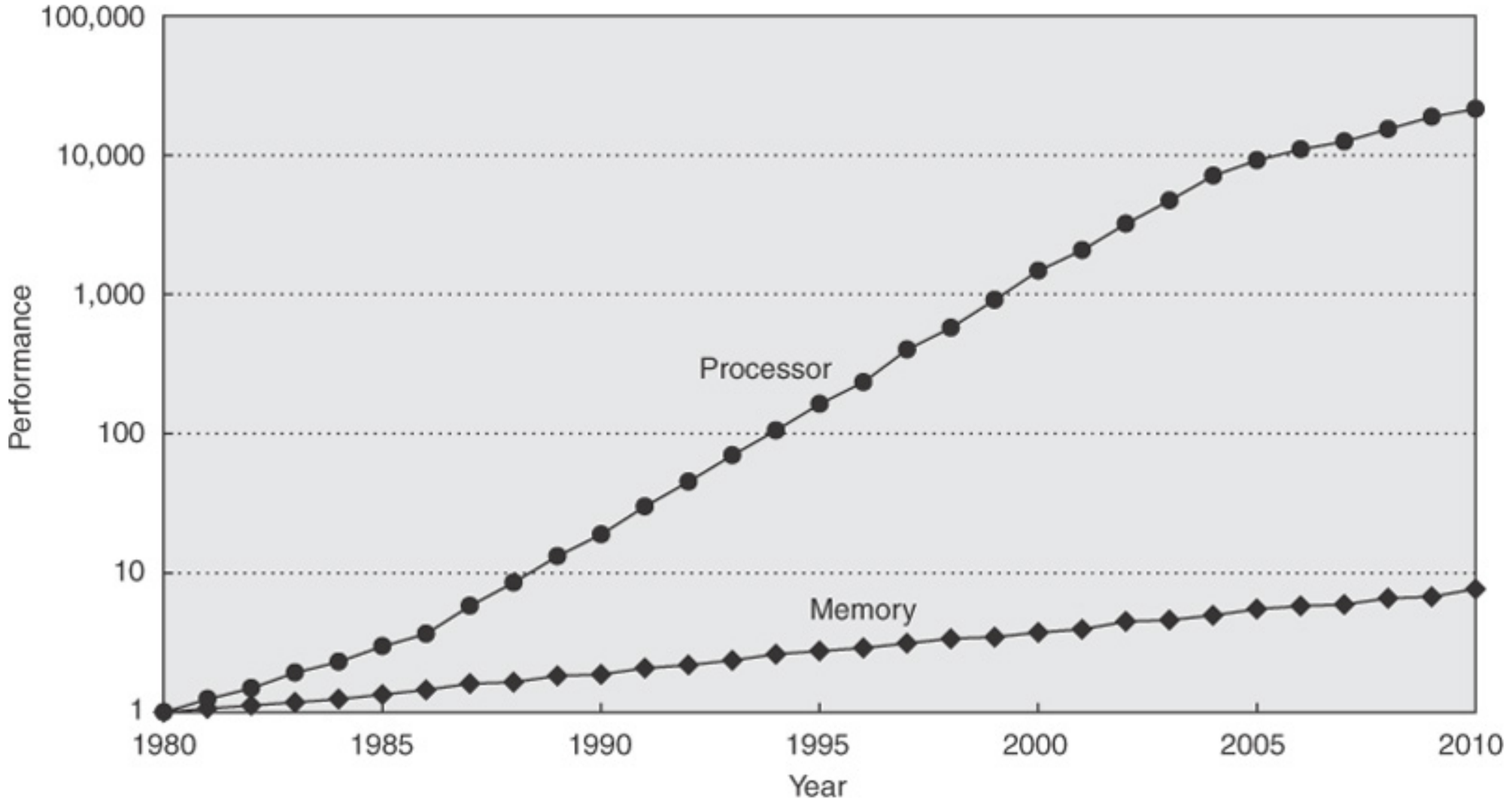

Memory Controller
System Agent
Display Ctrl
I/O Ctrl
CPU Core
CPU Core
CPU Core
CPU Core
L3$
L3$
L3$
L3$
Ring / Interconnect
Gen9.5 GPU
+
Media Capabilities

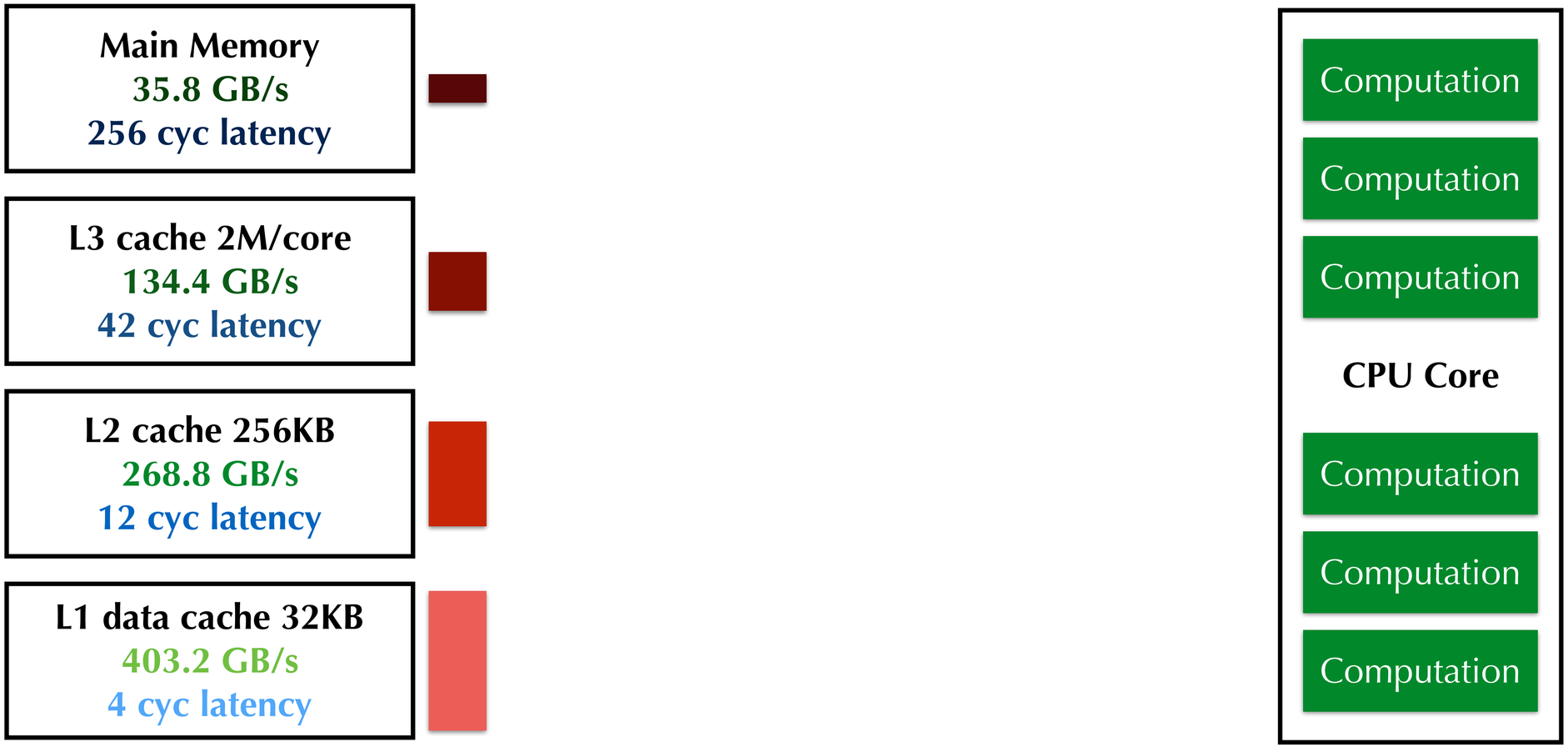



L1D$
L2$
Execution Units
Mem Management & Execution
OoO Scheduling Retirement
Decode & MSROM
L1I$


CPU
L1D$
L1I$
L2$
CPU
L1D$
L1I$
L2$
GPU
Graphics Caches
L3$
Devices
PCIe
System Agent
Side$ (eDRAM)
eDRAM Ctrl
MS$ Tags
MC
DDR
DDR

The era of slow memory…





* Caches are not drawn to scale.
* Data collected from the Intel Skylake architecture.
* There can be multiple data transfers happening simultaneously.
* Access to slower memory is invoked by faster memory cache miss.

* Caches are not drawn to scale.
* Data collected from the Intel Skylake architecture.
* There can be multiple data transfers happening simultaneously.
* Access to slower memory is invoked by faster memory cache miss.

Locality

- **Spatial locality**: try to access spatially neighboring data in main memory
  - Higher cacheline utilization
  - Fewer Cache/TLB misses
  - Better hardware prefetching on CPUs
- **Temporal locality**: reuse the data as much as you can
  - Higher cache-hit rates
  - Lower main memory bandwidth pressure
- Shrink the working set, so that data resides in lower-level (higher throughput, lower latency) memory

Understanding Cachelines (memory.cpp)

```
constexpr int n = 256 * 1024 * 1024;
int a[n];

void benchmark() {
    auto t = get_time();
    for (int i = 0; i < n; i += stride) {
      a[i] = i;
    }
    printf("%f\n", get_time() - t);
}
```

Understanding Cachelines (memory.cpp)

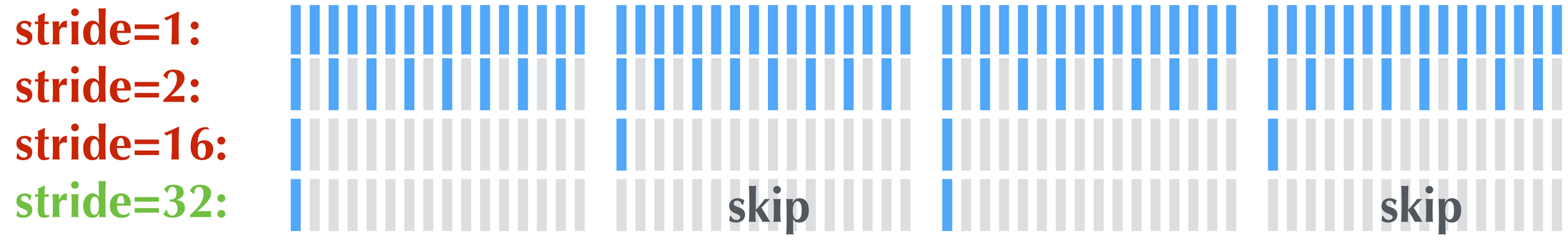
```
constexpr int n = 256 * 1024 * 1024;
int a[n];

void benchmark() {
    auto t = get_time();
    for (int i = 0; i < n; i += stride) {
      a[i] = i;
    }
    printf("%f\n", get_time() - t);
}
```

stride=1: 0.13s
stride=2: 0.13s
stride=4: 0.13s
stride=8: 0.13s
stride=16: 0.13s
stride=32: 0.096s
stride=64: 0.069s

Understanding Cachelines (memory.cpp)

```
constexpr int n = 256 * 1024 * 1024;
int a[n];

void benchmark() {
    auto t = get_time();
    for (int i = 0; i < n; i += stride) {
      a[i] = i;
    }
    printf("%f\n", get_time() - t);
}
```

stride=1: 0.13s
stride=2: 0.13s
stride=4: 0.13s
stride=8: 0.13s
stride=16: 0.13s
stride=32: 0.096s
stride=64: 0.069s



Understanding Cachelines (matmul.cpp)

A

```
constexpr int n = 512;
int A[n][n];
int B[n][n];
int C[n][n];

for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      C[i][j] += A[i][k] * B[k][j];
   }
  }
}
```

B

```
constexpr int n = 512;
int A[n][n];
int B[n][n];
int C[n][n];

for (int i = 0; i < n; i++) {
  for (int k = 0; k < n; k++) {
    for (int j = 0; j < n; j++) {
      C[i][j] += A[i][k] * B[k][j];
   }
  }
}
```

Which is faster?

Understanding Cachelines (matmul.cpp)

A

```
constexpr int n = 512;
int A[n][n];
int B[n][n];
int C[n][n];

for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      C[i][j] += A[i][k] * B[k][j];
   }
  }
}
```

0.244s

B

```
constexpr int n = 512;
int A[n][n];
int B[n][n];
int C[n][n];

for (int i = 0; i < n; i++) {
  for (int k = 0; k < n; k++) {
    for (int j = 0; j < n; j++) {
      C[i][j] += A[i][k] * B[k][j];
   }
  }
}
```

0.039s

Which is faster?

Understanding Caches (Advanced)

A

```
constexpr int n = 512;
int A[n][n];
int B[n][n];
int C[n][n];

for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      C[i][j] += A[i][k] * B[k][j];
   }
  }
}
```

B

```
constexpr int n = 513;
int A[n][n];
int B[n][n];
int C[n][n];

for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      C[i][j] += A[i][k] * B[k][j];
   }
  }
}
```

google “cache sets/cache tags”

Which is faster?

Understanding Caches (Advanced)

A

```
constexpr int n = 512;
int A[n][n];
int B[n][n];
int C[n][n];

for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      C[i][j] += A[i][k] * B[k][j];
   }
  }
}
```

0.244s

B

```
constexpr int n = 513;
int A[n][n];
int B[n][n];
int C[n][n];

for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      C[i][j] += A[i][k] * B[k][j];
   }
  }
}
```

0.135s

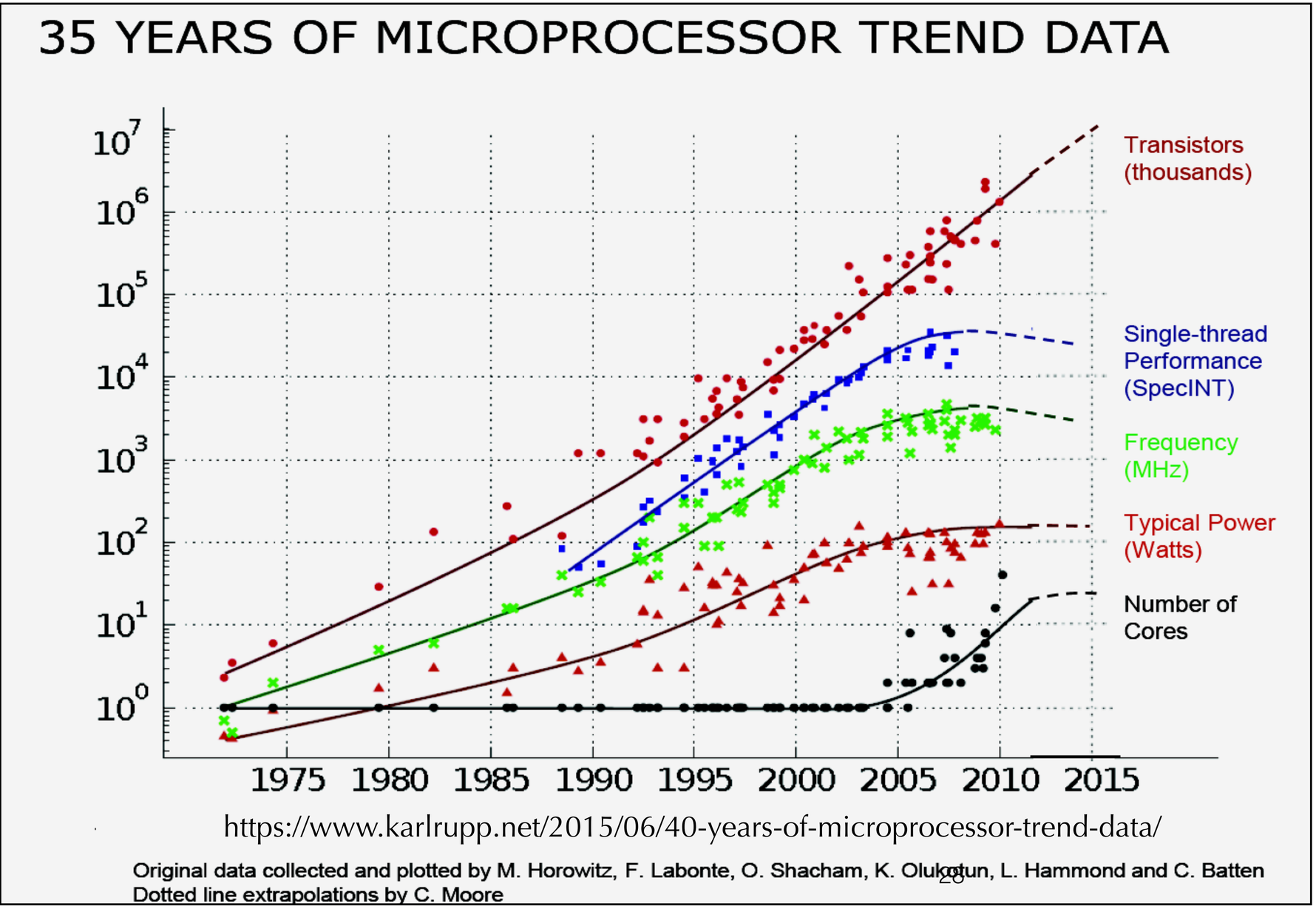
google “cache sets/cache tags”

Which is faster?

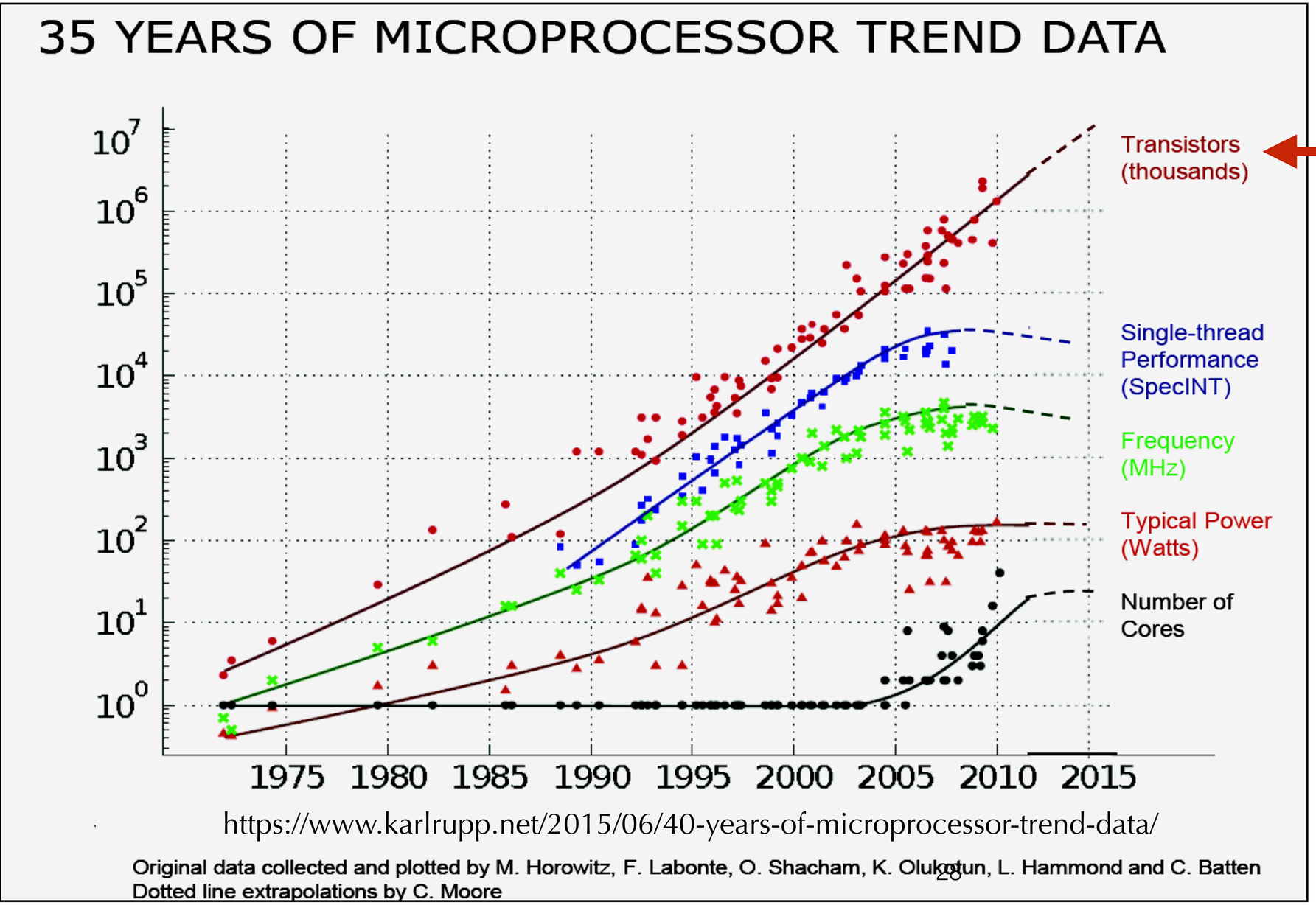
Memory: References

- ***Computer Systems: A Programmer's Perspective*** (highly recommended)
- ***What Every Programmer Should Know About Memory***

CPU μArch: Trends

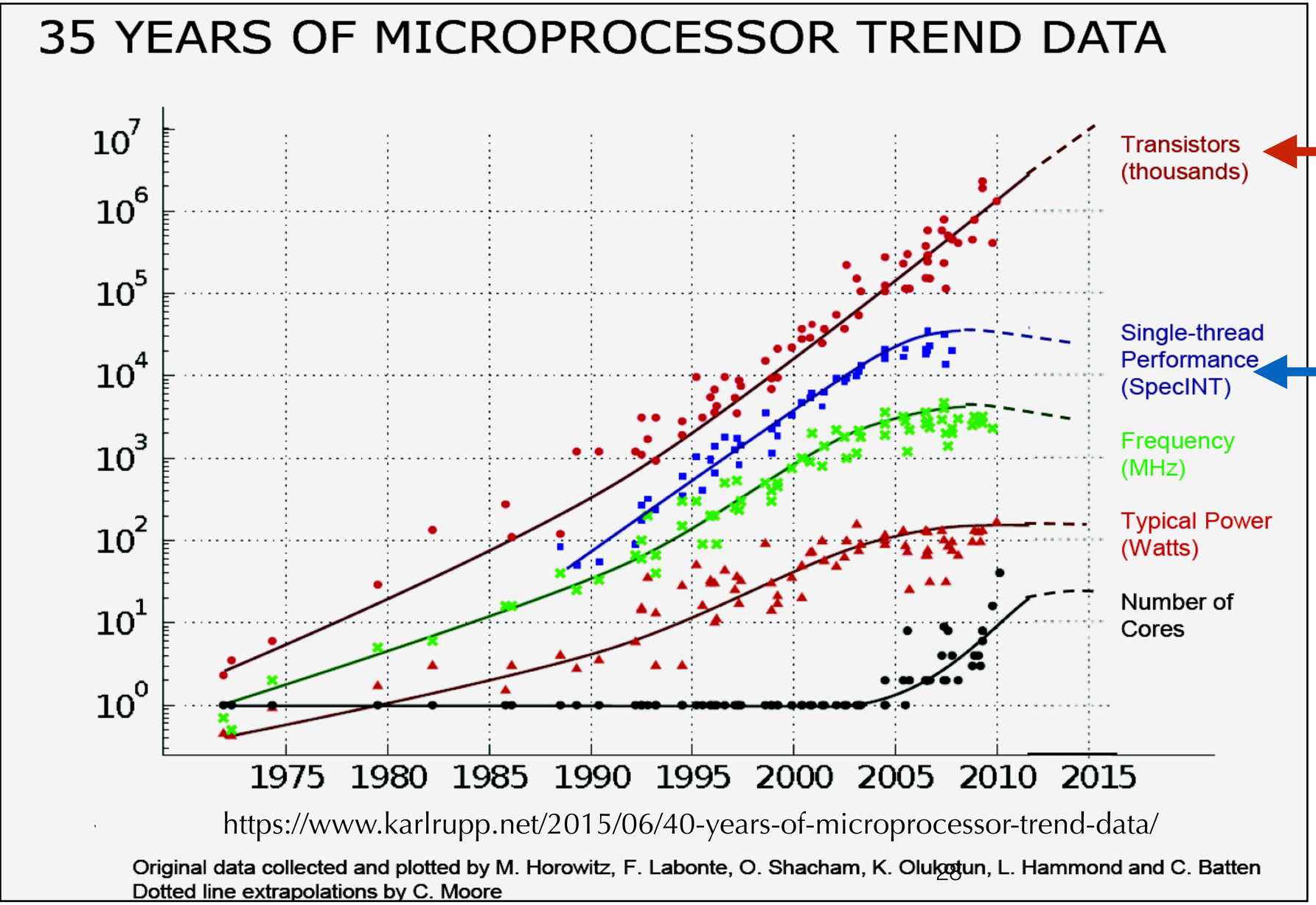


CPU μArch: Trends



Moore's law continues

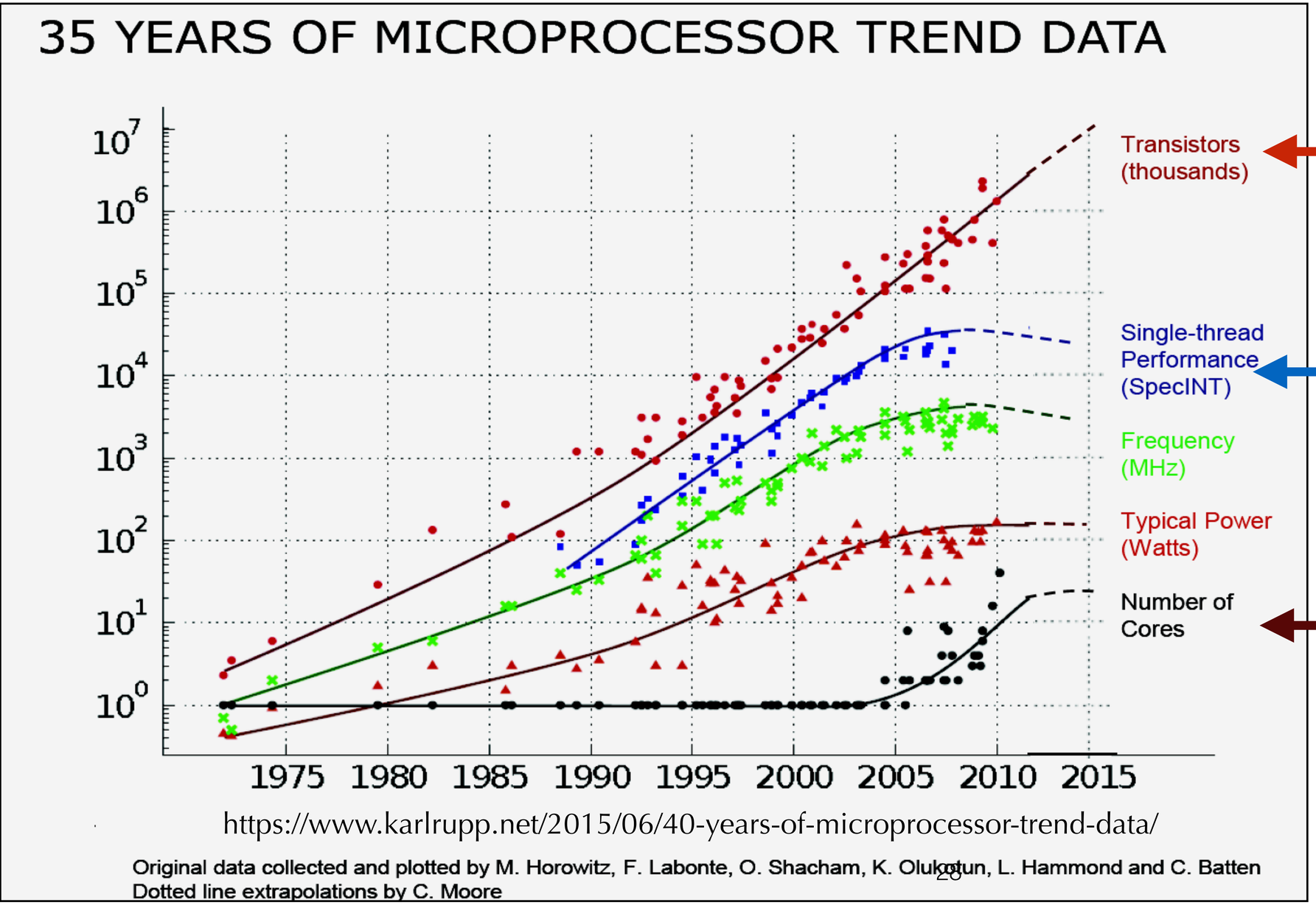
CPU μArch: Trends



Moore's law continues

Single-core performance stops growing

CPU μArch: Trends

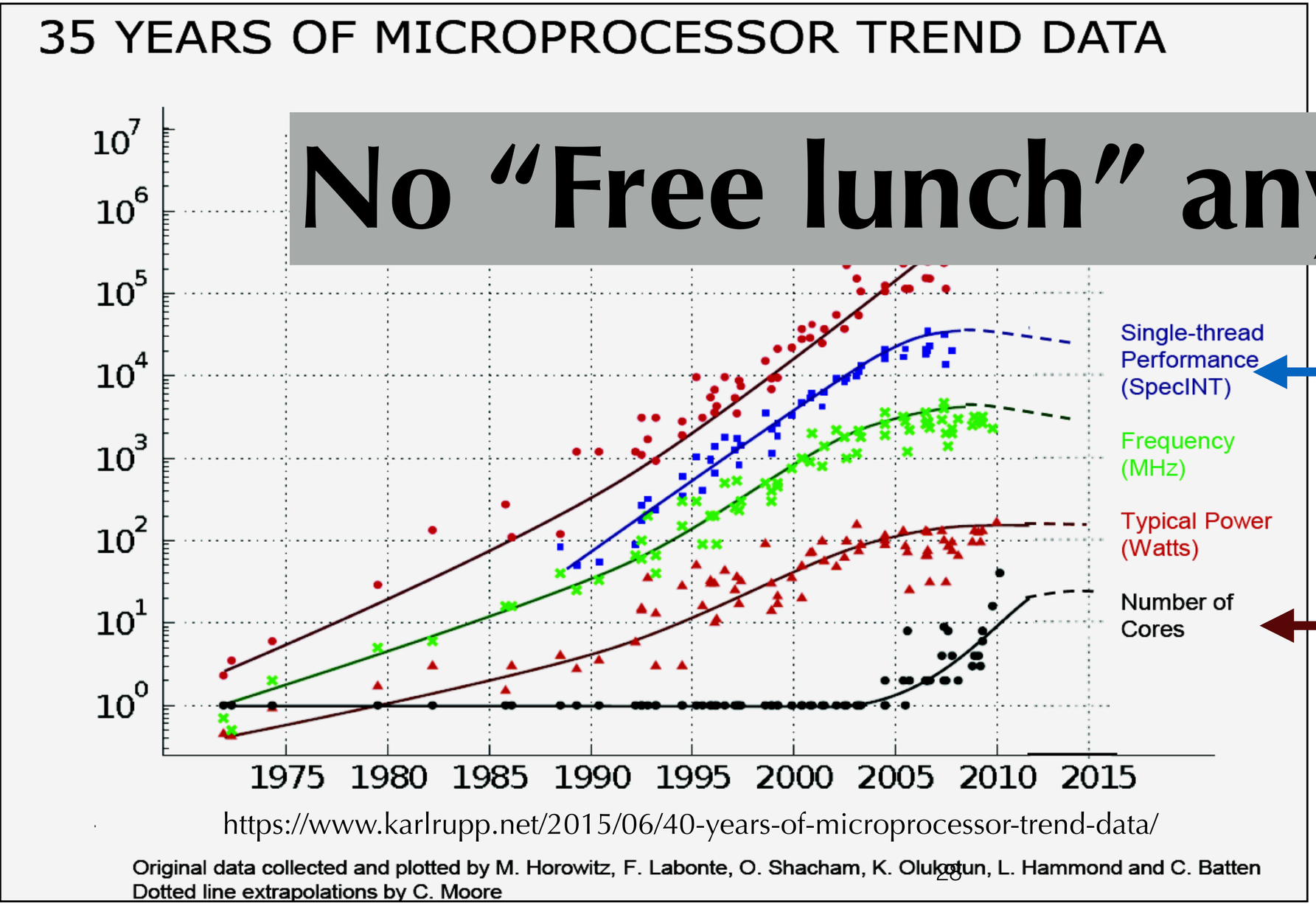


Moore's law continues

Single-core performance stops growing

Instead of "better" cores, we have "more" cores (with wider SIMD)

CPU µArch: Trends



No "Free lunch" anymore!

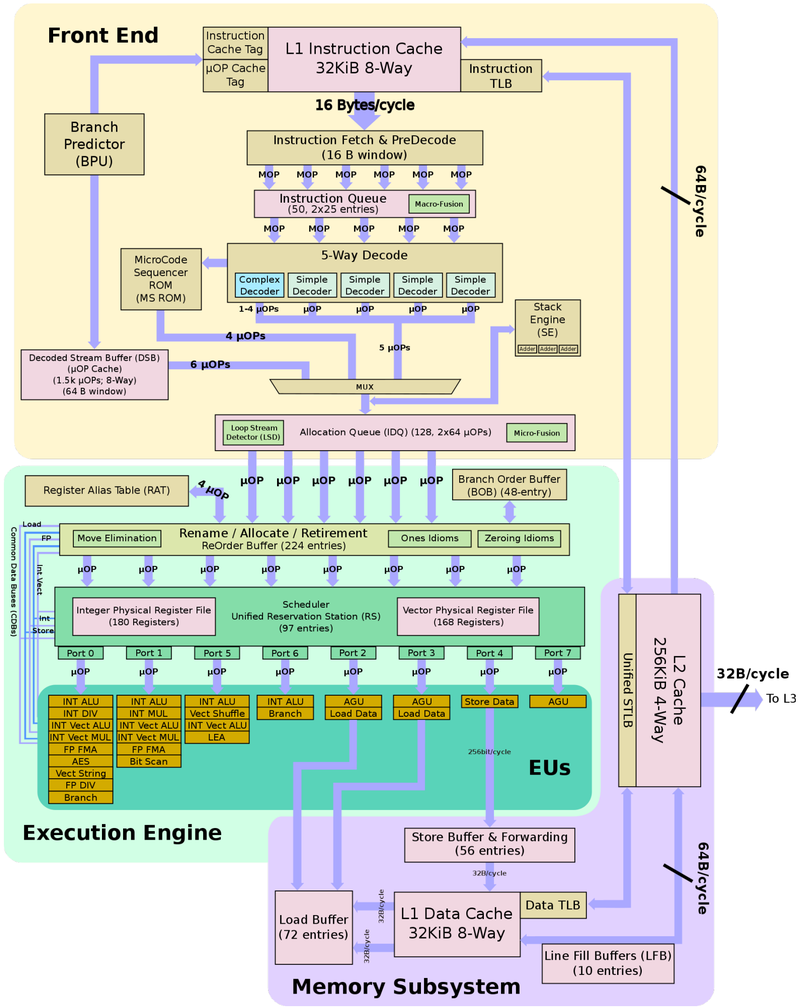
continues

Single-core performance stops growing

Instead of "better" cores, we have "more" cores (with wider SIMD)

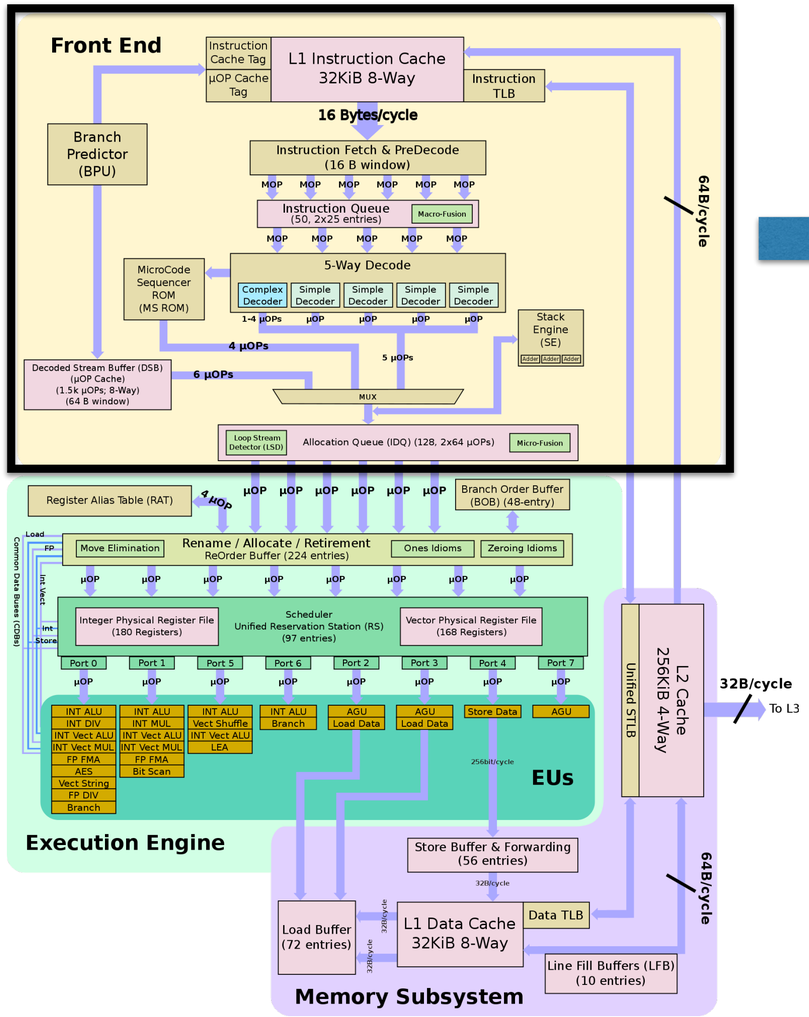
CPU µArch: Overview

https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

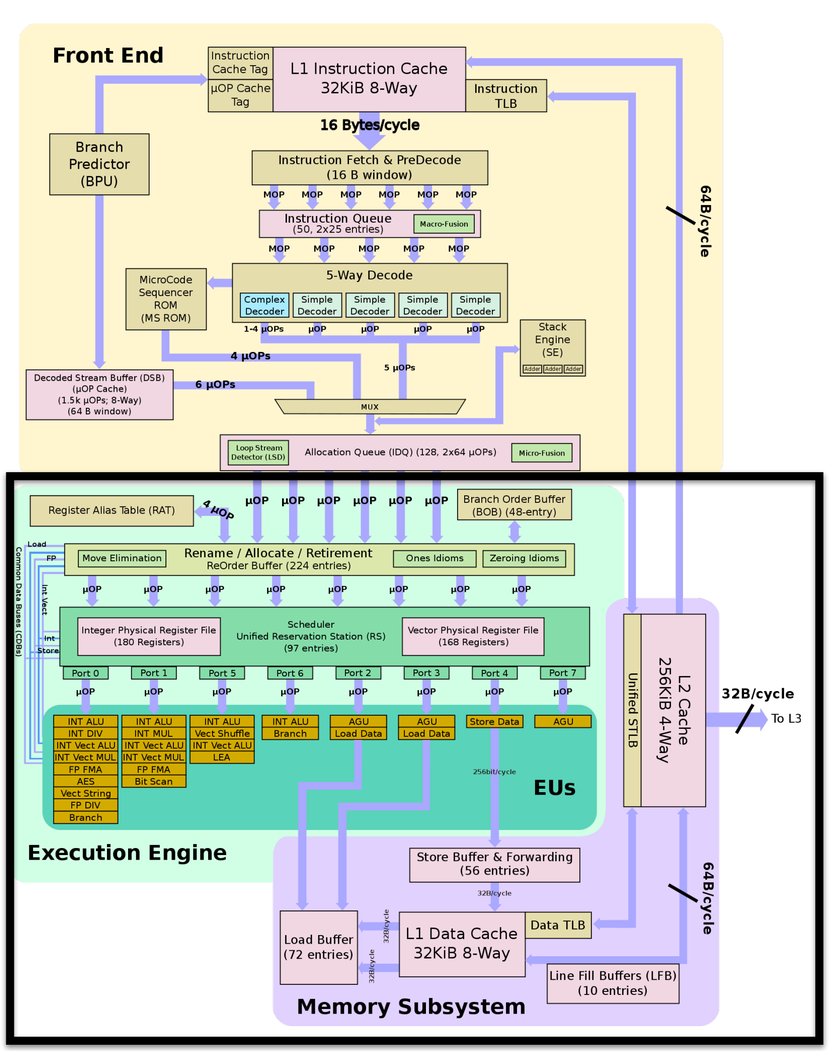


CPU µArch: Overview

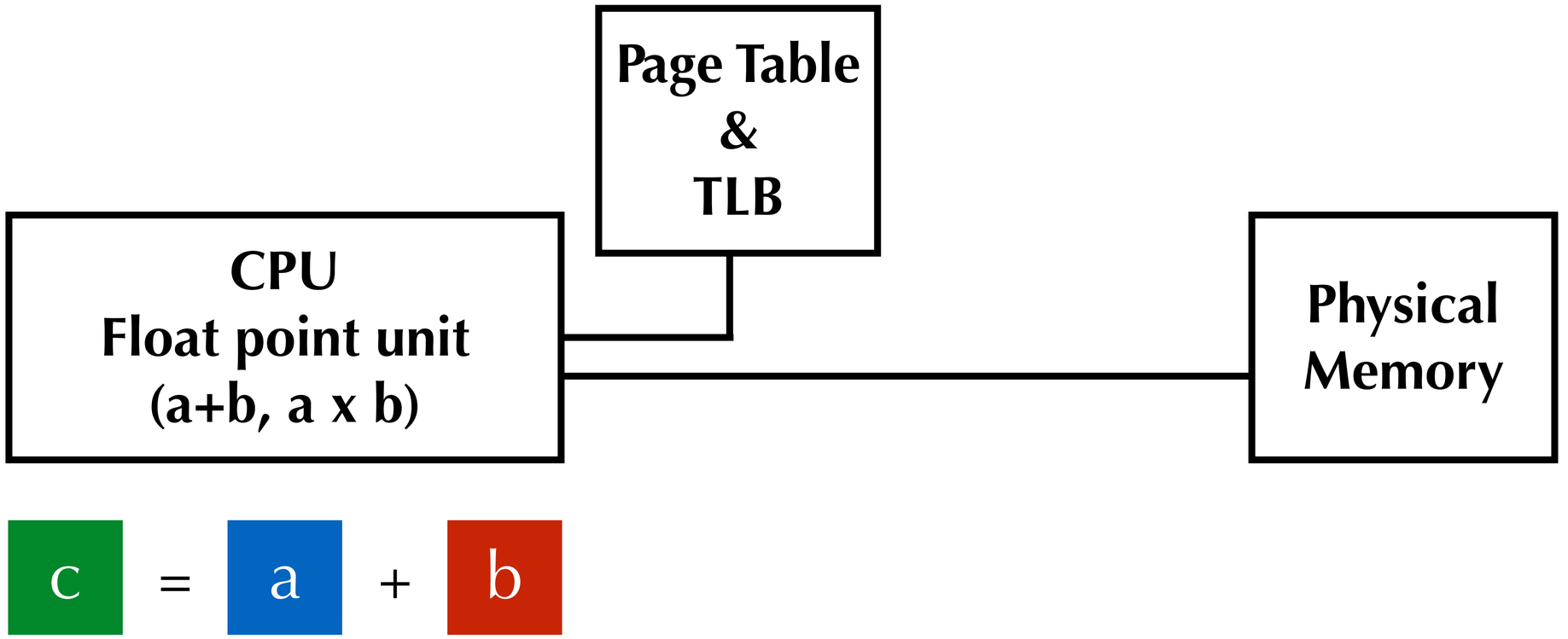
https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

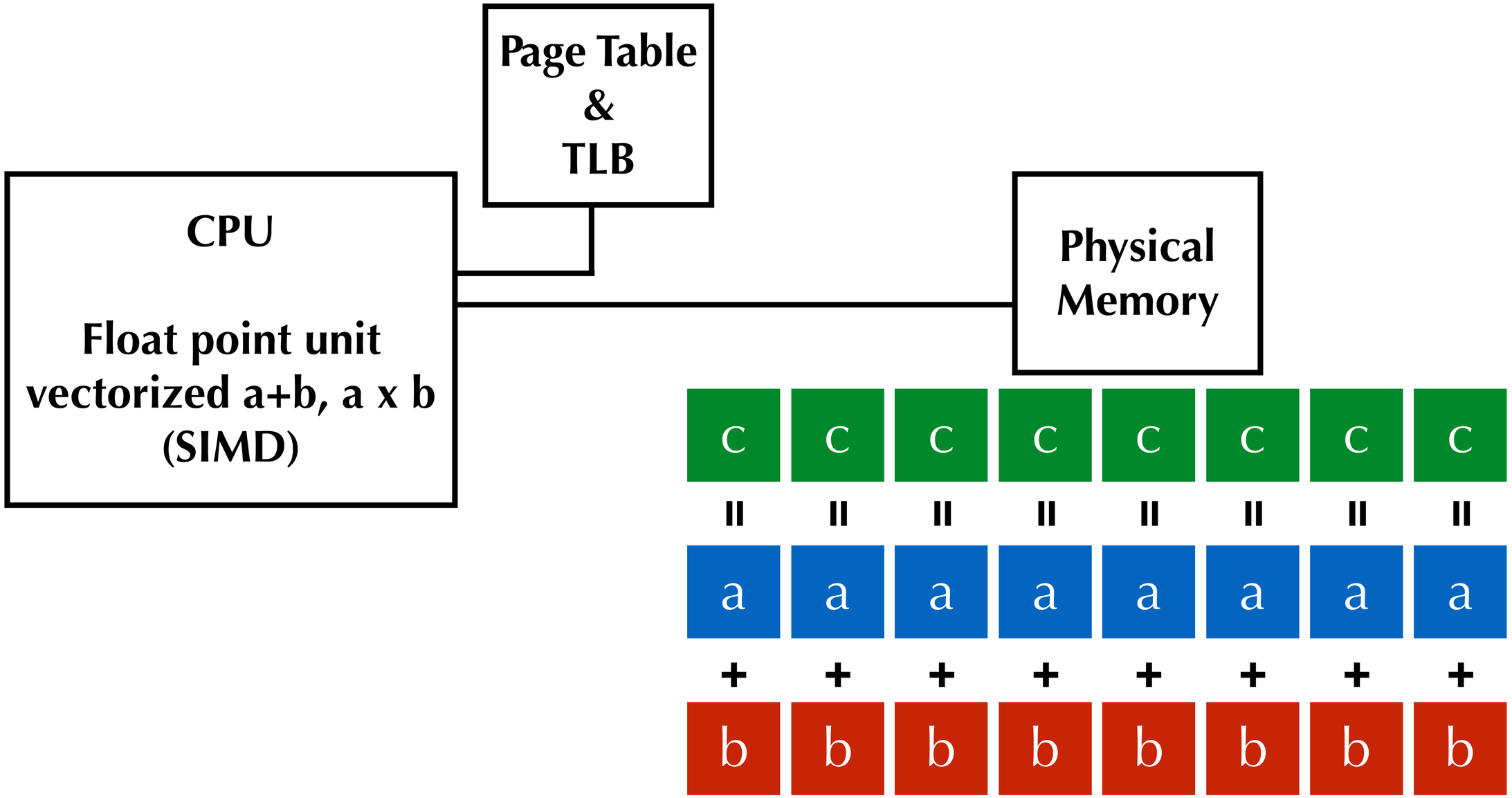


CPU µArch: Overview

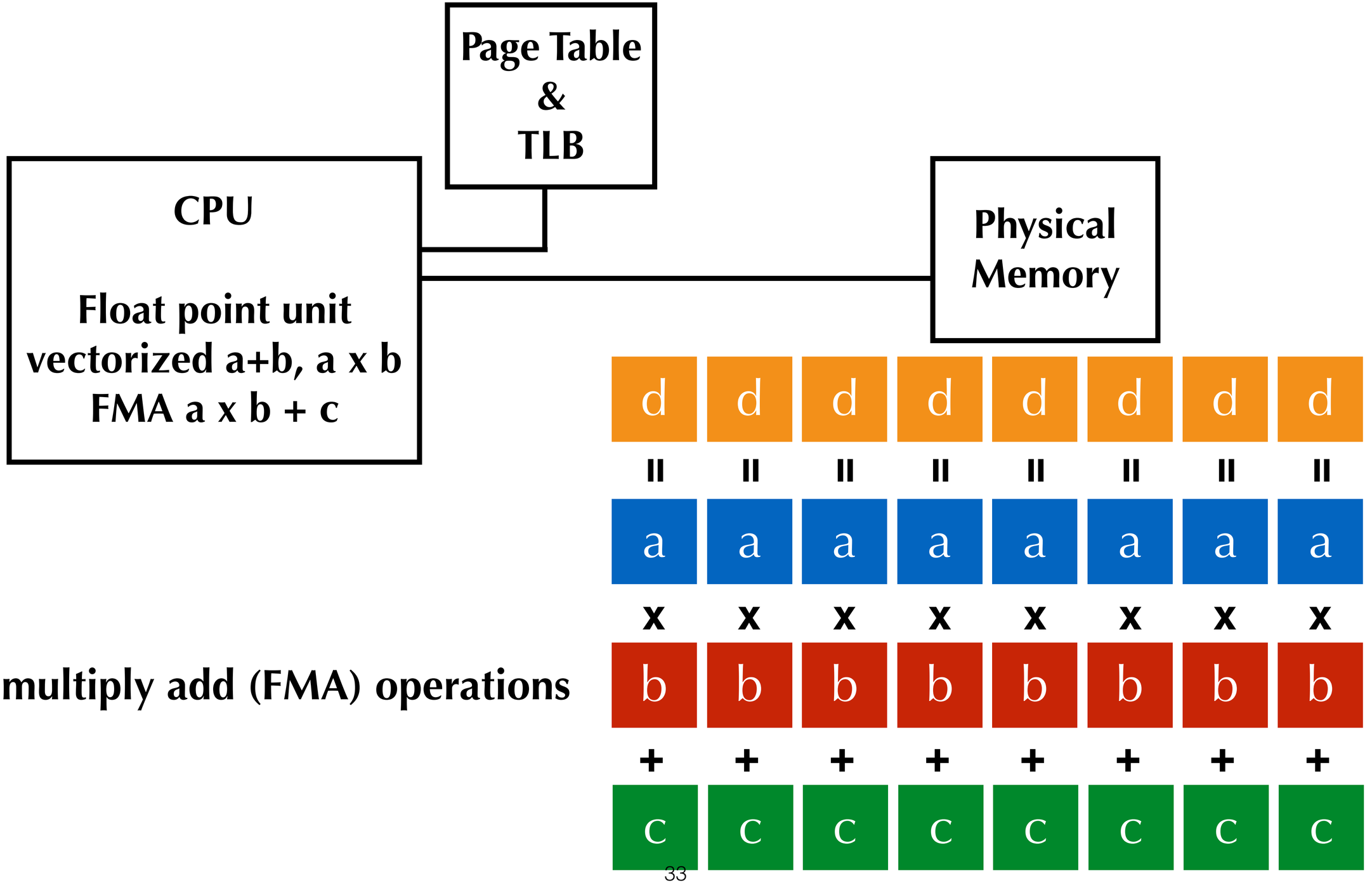


CPU μArch: Float-Point Units

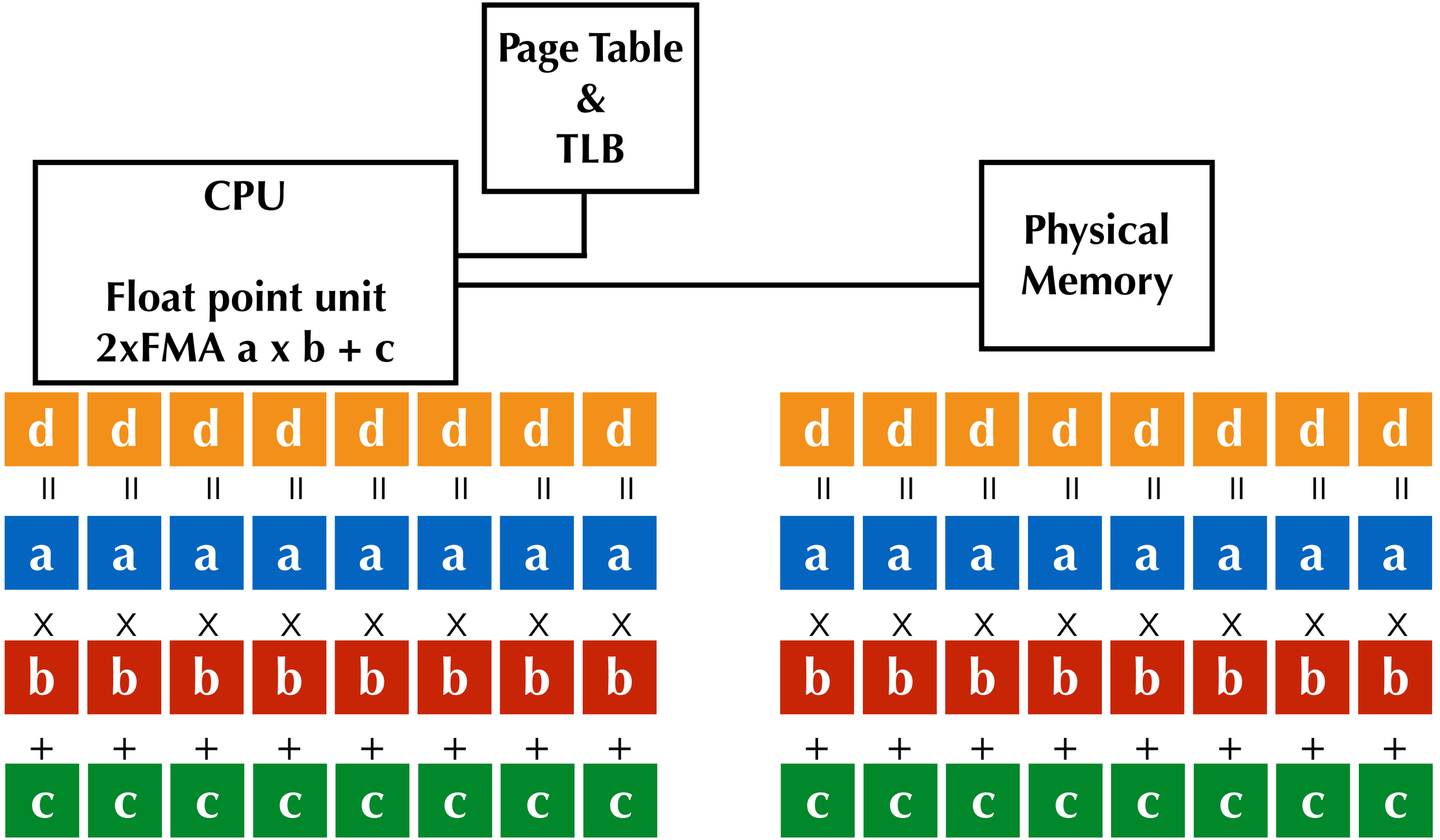




AVX2: 8 single precision/4 double precision float point number operations in a row
AVX512: 16 single precision/8 double precision float point number operations in a row

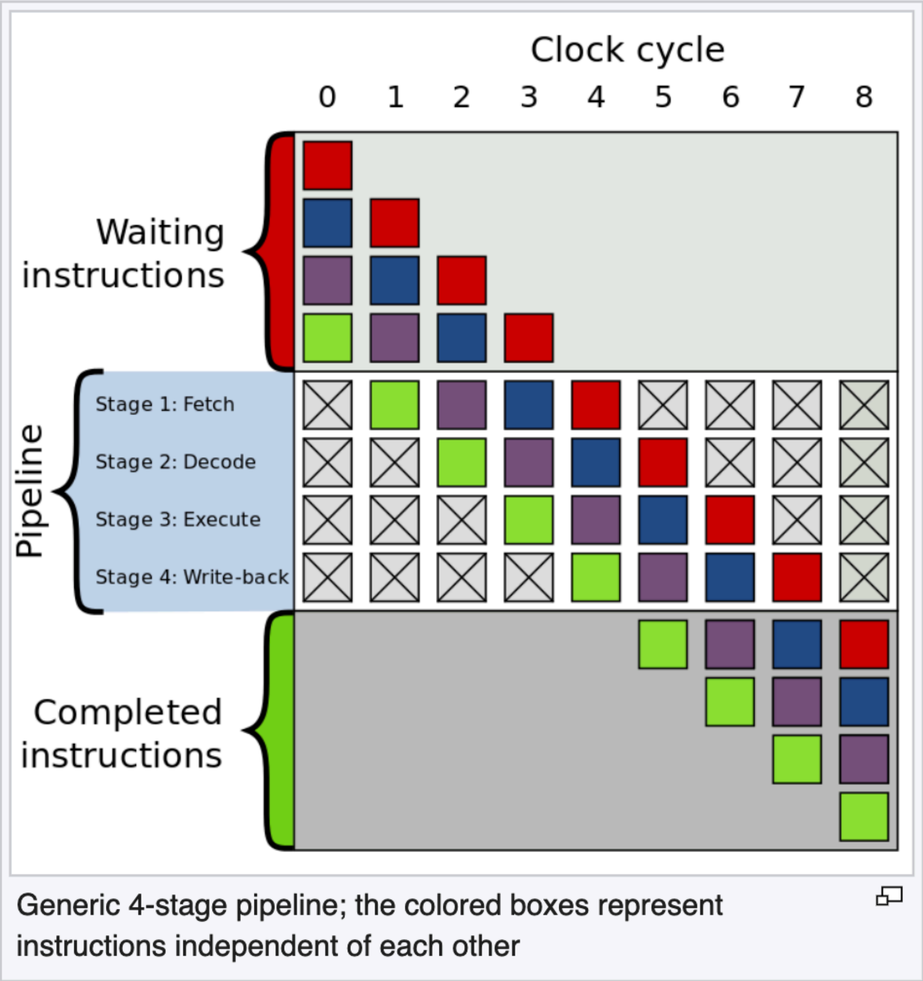


Fused multiply add (FMA) operations



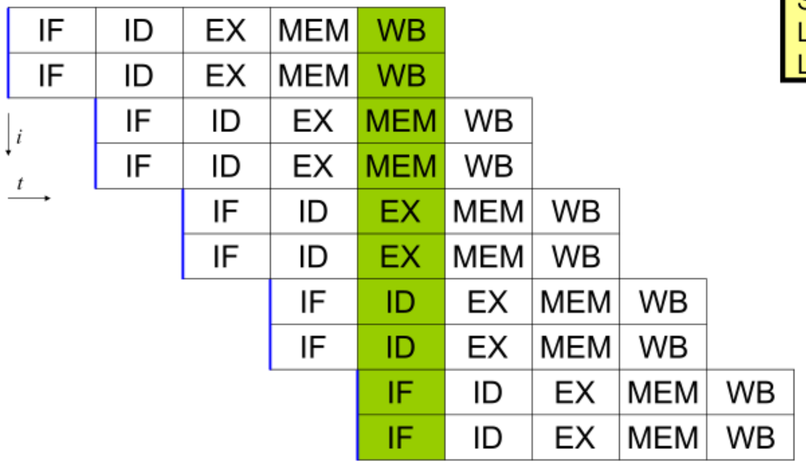
4.2G Hz x 2 FMA/cycle x 16 FLOPS/FMA x 4 cores = 538G FLOPs

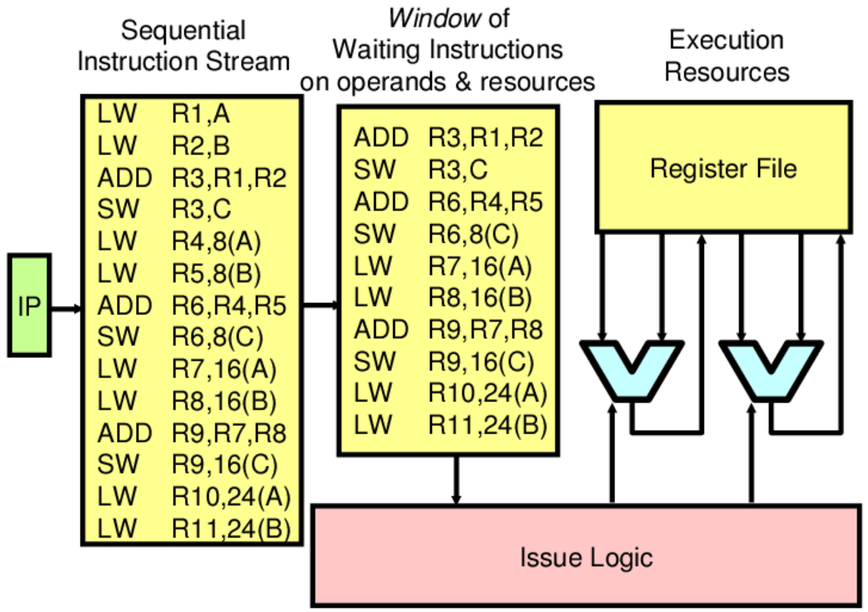
CPU μArch: Instruction-Level Parallelism Pipelining, Superscalar, Out-of-order



Generic 4-stage pipeline; the colored boxes represent instructions independent of each other

Superscalar





OoO

http://hpca23.cse.tamu.edu/taco/utsa-www/cs5513-fall07/lecture5.html

CPU μArch: References

- "Official" optimization manual:
  - Intel® 64 and IA-32 Architectures Optimization Reference Manual
- Chip specifications:
  - WikiChip

Quantitative Analysis: Main Memory Bandwidth

>> **sudo lshw**

…

*-memory
description: System Memory
physical id: 3c
slot: System board or motherboard
size: 32GiB
*-bank:0
description: DIMM DDR4 Synchronous Unbuffered (Unregistered) **2400 MHz** (0.4 ns)
vendor: Fujitsu
physical id: 0
serial: B8520000
slot: ChannelA-DIMM0
size: 8GiB
width: **64 bits**
clock: 2400MHz (0.4ns)

2 (Channels)

x 8 (64 bits=8bytes)

x 2.4GHz (Double Data Rate)

$=38.4 \times 10^9$ B/s = 35.76 GiB/s

Quantitative Analysis: Main Memory Bandwidth

>> **sudo lshw**

…

```
   *-memory
        description: System Memory
        physical id: 3c
        slot: System board or motherboard
        size: 32GiB
      *-bank:0
           description: DIMM DDR4 Synchronous
Unbuffered (Unregistered) 2400 MHz (0.4 ns)
           vendor: Fujitsu
           physical id: 0
           serial: B8520000
           slot: ChannelA-DIMM0
           size: 8GiB
           width: 64 bits
           clock: 2400MHz (0.4ns)
```

2 (Channels)

x 8 (64 bits=8bytes)

x 2.4GHz (Double Data Rate)

$=38.4 \times 10^9$ B/s = 35.76 GiB/s

Where is cacheline size?

Quantitative Analysis: Main Memory Bandwidth

- **Streaming (memory-bound) tasks with 1024^3 float elements**
  - sasum 130ms sum = Σ x[i]
    - **(4B/element)** Reads 4 bytes
  - memset 126ms x[i] = 0
    - **(4B/element)** Writes 4 bytes
  - sscal 262ms y[i] = a x[i]
    - **(8B/element)** Reads 4 bytes, writes 4 bytes
  - saxpy 428ms y[i] = a x[i] + y[i]
    - **(12B/element)** Reads 8 bytes, writes 4 bytes

4 GB/0.126s =31.75 GB/s
89% of peak bandwidth (35.8GB/s)

Experiment (recommended):
Implement your own version and
see how much bandwidth you achieve.

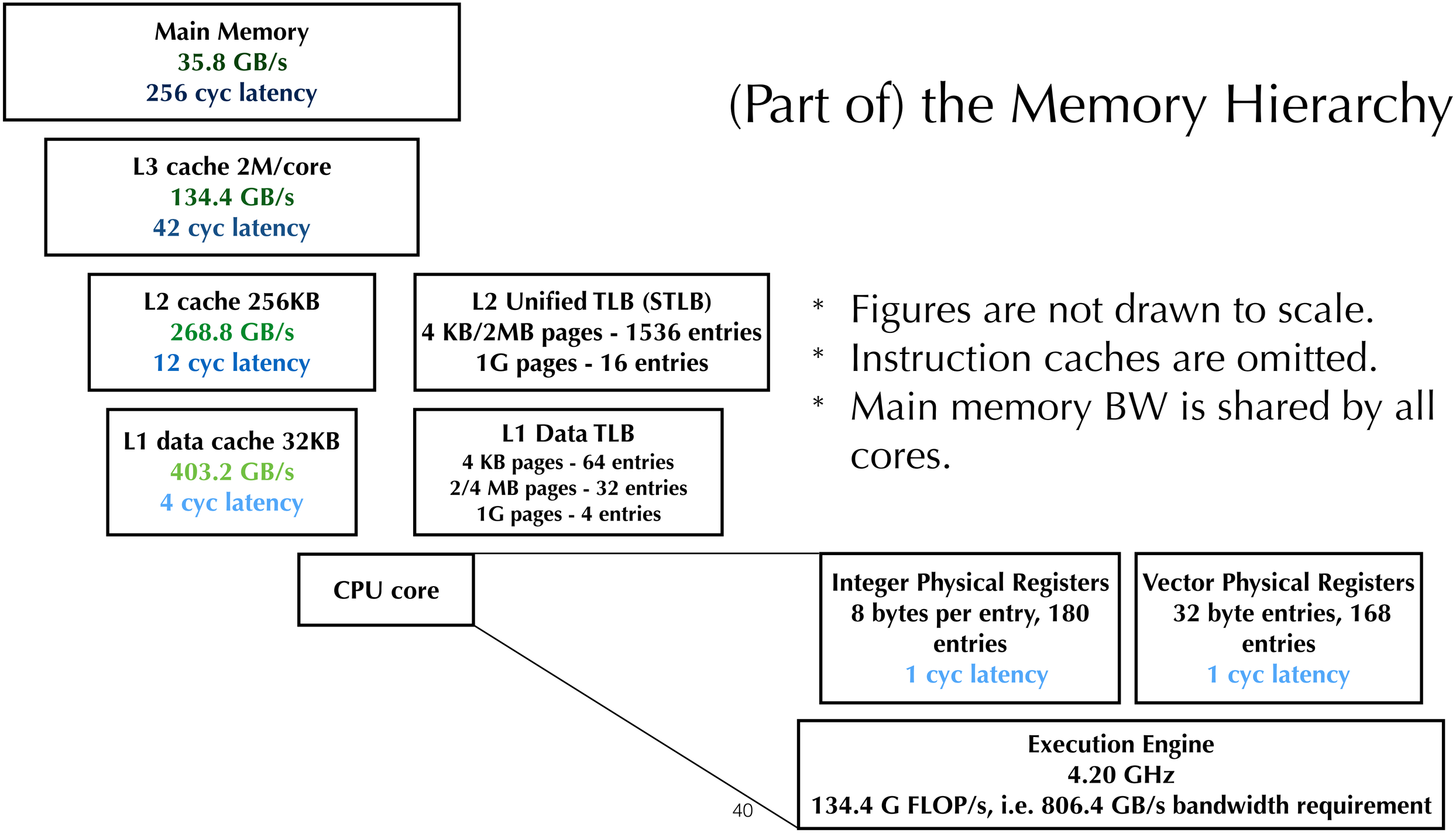
Quantitative Analysis: Peak Float-Point Operation per Second (FLOPS)

- 4.2GHz x 32=134.4G GLOPS per core
  - ~0.5T FLOPS on a quad-core i7
- In code:
  - The “easy” way: issue a bunch of independent vectorized FMA instructions ~100% peak FLOPS
  - The hard way: write a highly optimized matrix multiplication
- Practical programs
  - Unoptimized ones usually achieves 0.1% peak FLOPS
  - Achieving 20% (constantly) is already challenging

```
uint64 fma(uint64 n) {
  n = n / 10;
#define FMA(x) "vfmadd213ps %%ymm" #x ", %%ymm" #x ", %%ymm" #x ";"
  __asm__(
      "mov %0, %%rax;"
      "start:"
          FMA(0)
          FMA(1)
          FMA(2)
          FMA(3)
          FMA(4)
          FMA(5)
          FMA(6)
          FMA(7)
          FMA(8)
          FMA(9)
      "sub $0x1, %%rax;"
      "jne start;"
      : "=r"(n)
      : "r"(n)
      : "%rax",
```

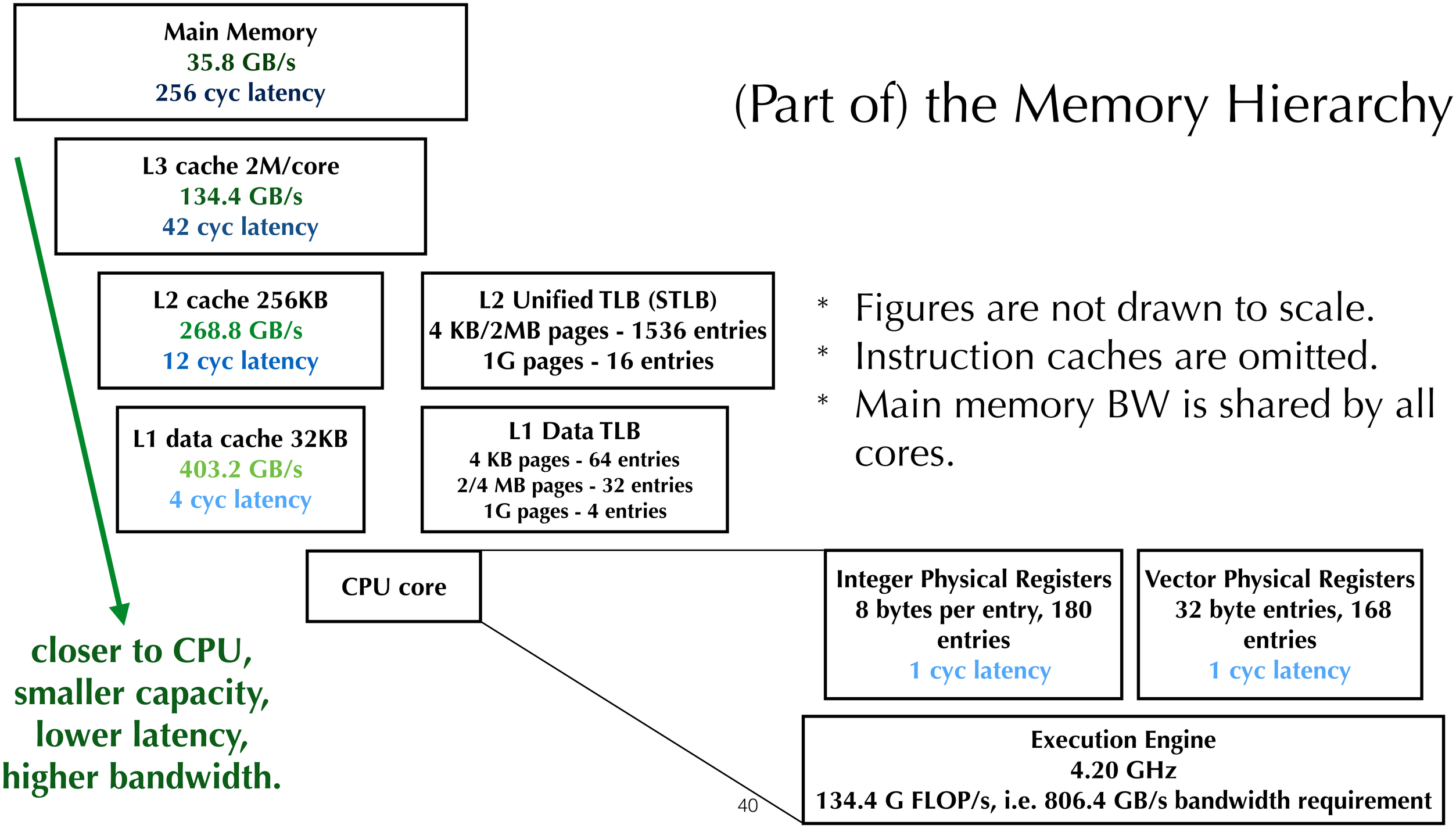
https://github.com/yuanming-hu/performance/blob/master/src/flops.cpp

(Part of) the Memory Hierarchy



* Figures are not drawn to scale.
* Instruction caches are omitted.
* Main memory BW is shared by all cores.

(Part of) the Memory Hierarchy



* Figures are not drawn to scale.
* Instruction caches are omitted.
* Main memory BW is shared by all cores.

Main Memory

35.8 GB/s in total, **9 GB/s per core**

256 cyc latency

Execution Engine

4.20 GHz

134.4 G FLOP/s, **i.e. 806.4 GB/s bandwidth requirement**

Main Memory

35.8 GB/s in total, **9 GB/s per core**

256 cyc latency

Without data reuse (temporal locality),
Processors need 100x higher bandwidth than
what they actually have!

Execution Engine

4.20 GHz

134.4 G FLOP/s, **i.e. 806.4 GB/s bandwidth requirement**

Takeaways:

1. **[Architecture-aware programming]** will become increasingly important in the future since processors and memory are stopping getting faster.

2. Processors are more capable than you thought.
   [Parallelism: Computation is cheap]

3. Memory bandwidth is a more scarce resource nowadays.
   [Locality: Communication is expensive]

Takeaways:

1. **[Architecture-aware programming]** will become increasingly important in the future since processors and memory are stopping getting faster.

2. Processors are more capable than you thought.
   [Parallelism: Computation is cheap]

3. Memory bandwidth is a more scarce resource nowadays.
   [Locality: Communication is expensive]

Same rules apply to GPUs

Advanced Taichi Programming

Apply what we learned to Taichi programs…

- Advanced Data Layouts
- Sparse programming

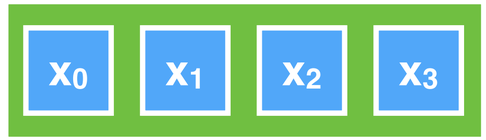
Advanced Data Layouts with Structural Nodes

Structural Nodes (SNodes)

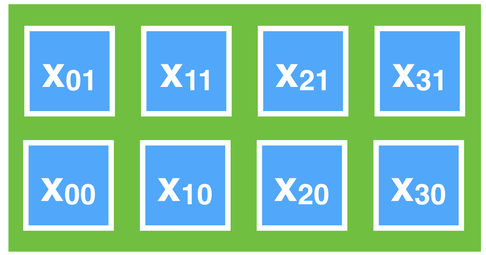
- **In Taichi, data layout is defined in a recursive tree structure. *Structural Nodes (SNodes) are the language we use to succinctly describe the tree.***
  - *List of SNodes:* ***root****,* ***dense****,* ***pointer****, bitmasked, dynamic,* ***place***
- **No matter what the data layout is, users always access fields using `a[i, j, k]` syntax**
  - which means you can change the data layout without modifying kernels...

The dense SNode

- `x = ti.field(dtype=ti.i32) # Note: no shape=(...) here`
- `ti.root.dense(ti.i, 4).place(x)`



- `ti.root.dense(ti.ij, (4, 2)).place(x)`



The dense SNode

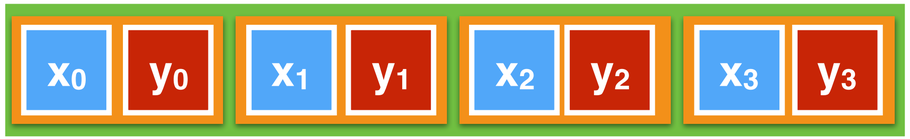
- x = ti.field(dtype=ti.i32)
- y = ti.field(dtype=ti.i32)

- ti.root.dense(ti.i, 4).place(x)
- ti.root.dense(ti.i, 4).place(y)

Structure of arrays

- ti.root.dense(ti.i, 4).place(x, y)

Array of structures



- Or cell=ti.root.dense(ti.i, 4); cell.place(x); cell.place(x)

The dense SNode

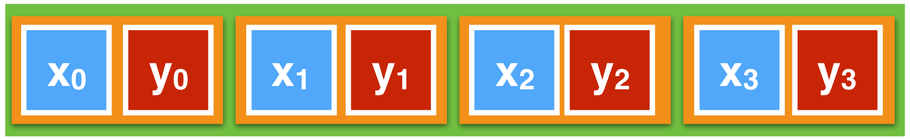
- x = ti.field(dtype=ti.i32)
- y = ti.field(dtype=ti.i32)

- ti.root.dense(ti.i, 4).place(x)
- ti.root.dense(ti.i, 4).place(y)

Structure of arrays

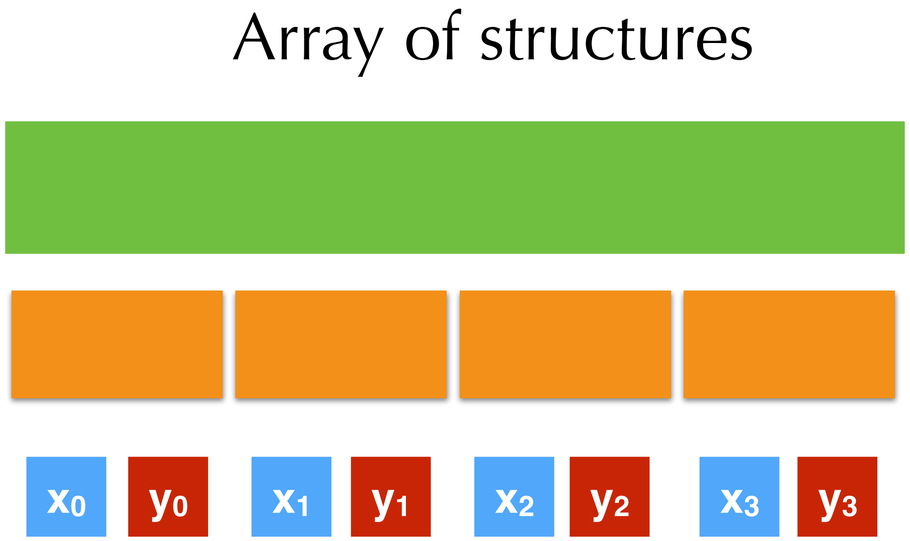
- ti.root.dense(ti.i, 4).place(x, y)

Array of structures



- Or cell=ti.root.dense(ti.i, 4); cell.place(x); cell.place(x)

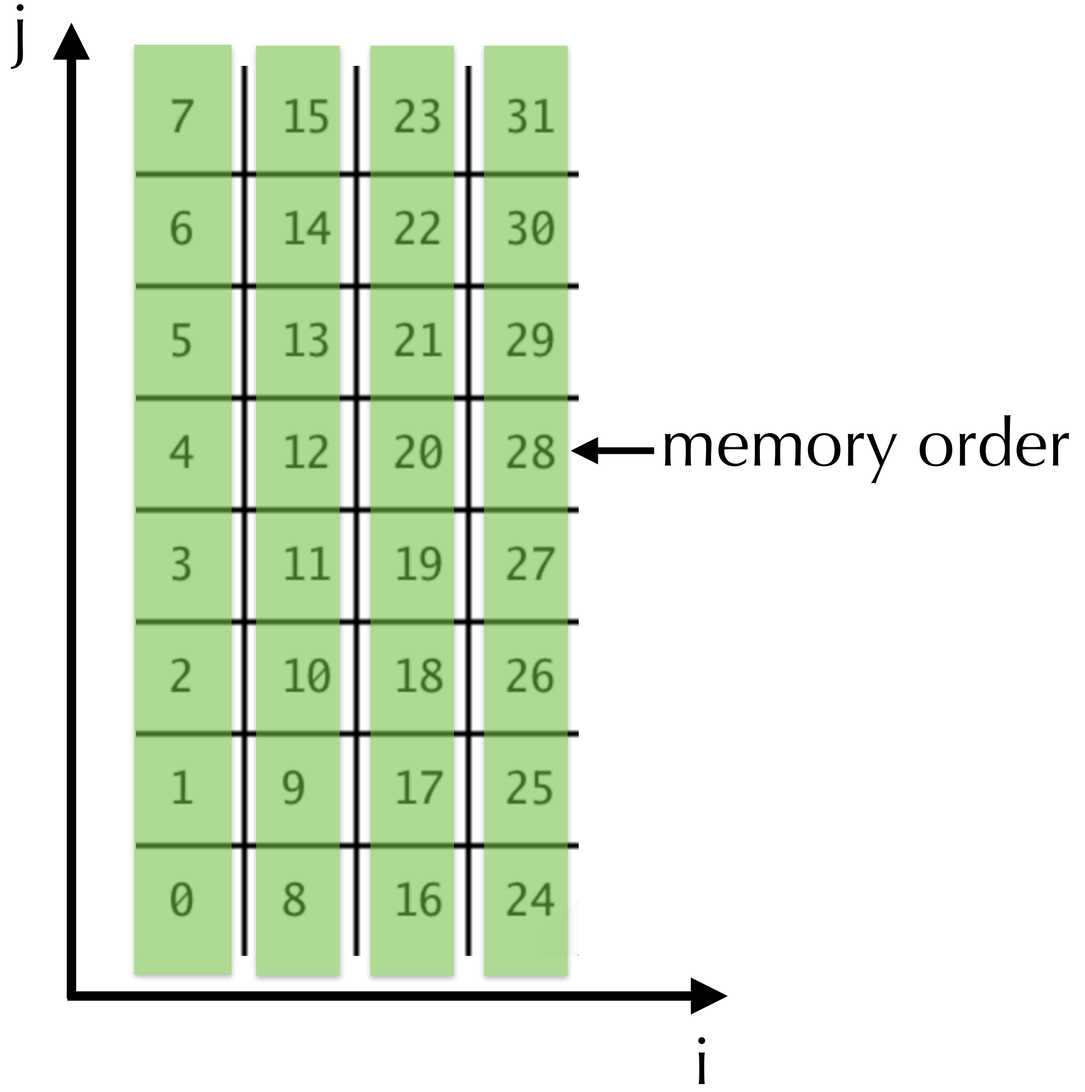
The dense SNode




```
ti.root
.dense(ti.i, 4)
.place
(x, y)
```

Nesting the dense SNodes

ti.root.dense(ti.ij, (4,8)).place(a)
or
ti.root.dense(ti.i, 4).dense(ti.j, 8).place(a)

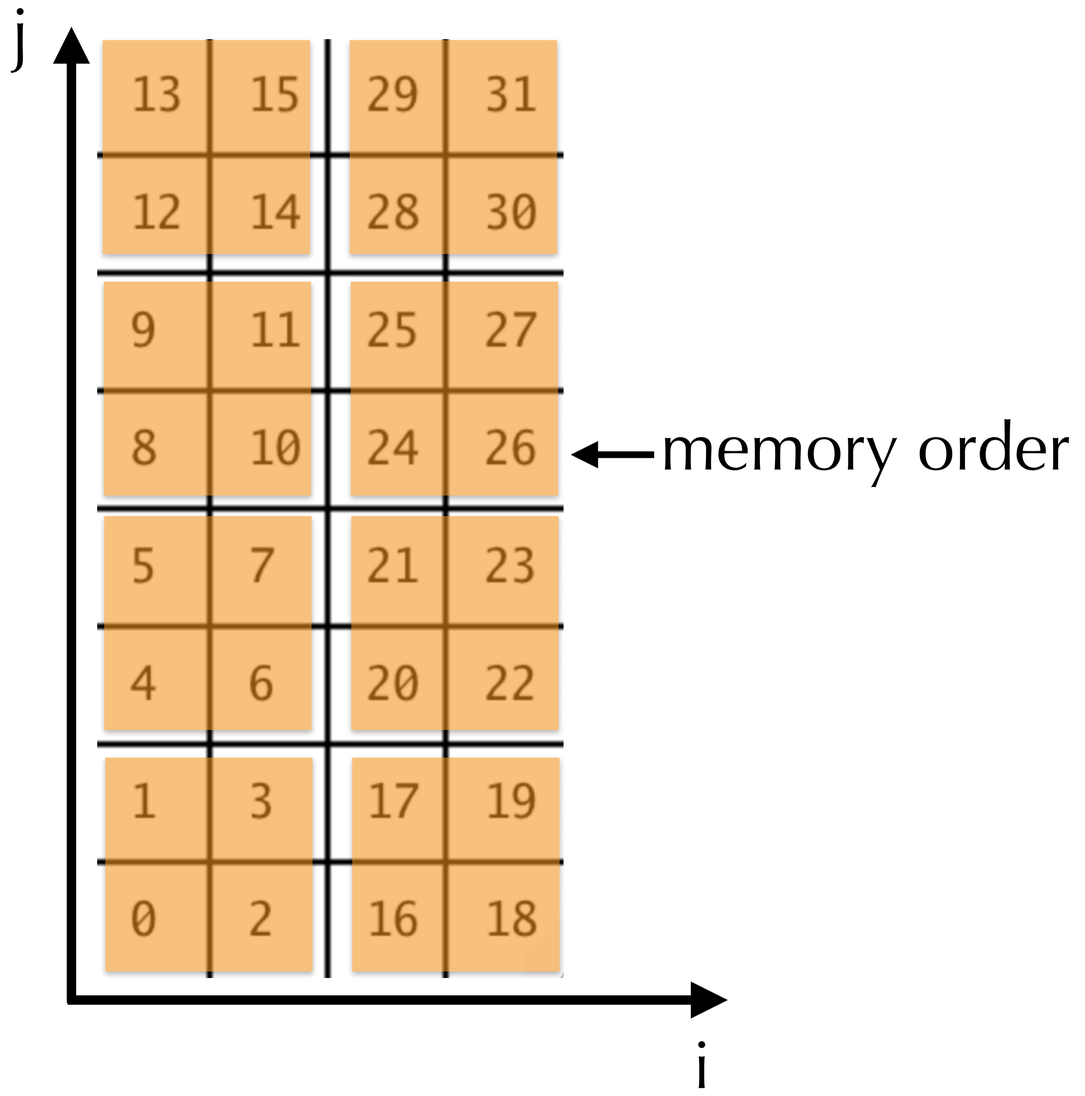


Nesting the dense SNodes

ti.root.dense(ti.ij, (2, 4)).dense(ti.ij, (2, 2)).place(a)

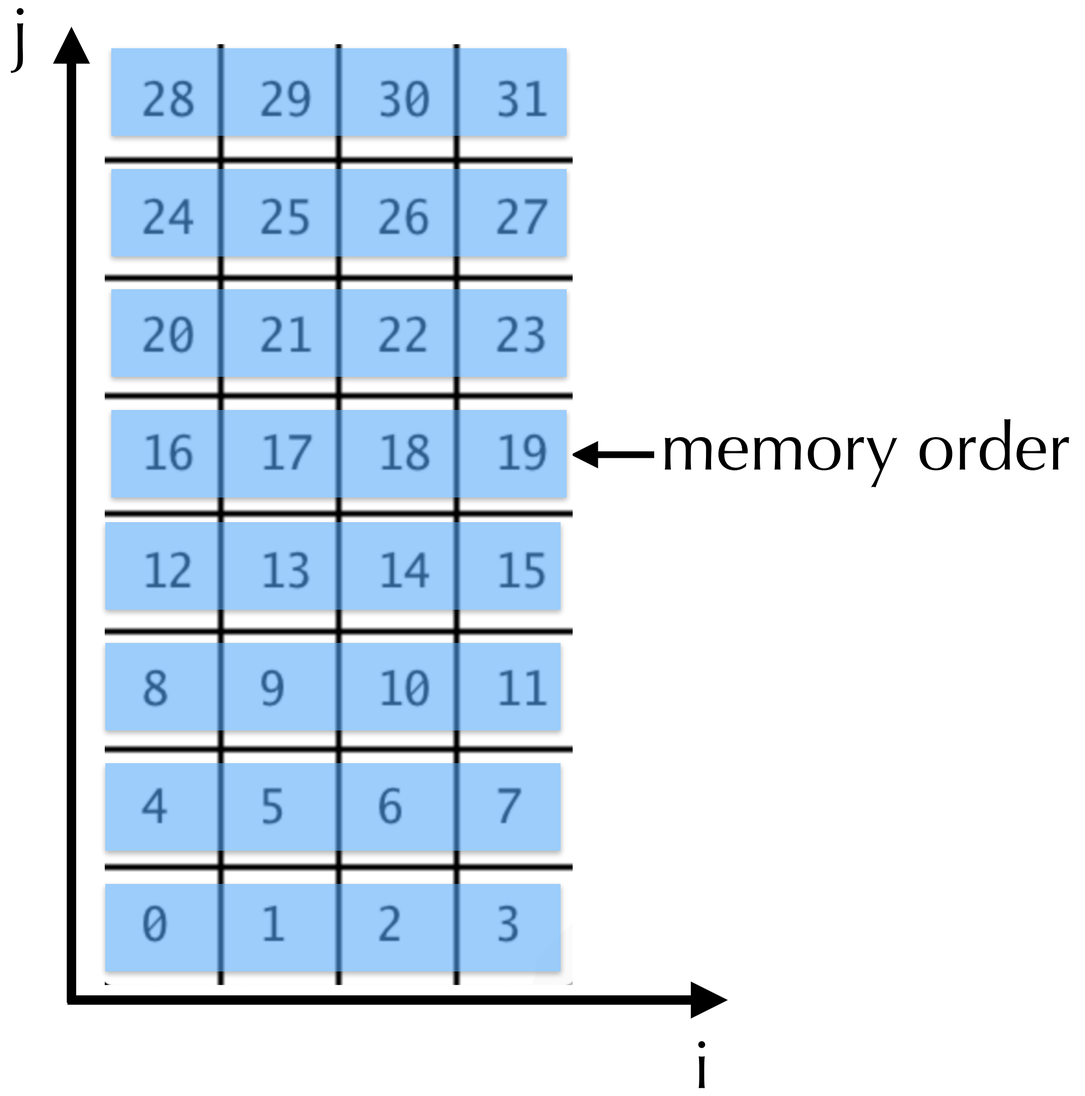
or simply

ti.root.dense(ti.ij, (2, 4)).dense(ti.ij, 2).place(a)



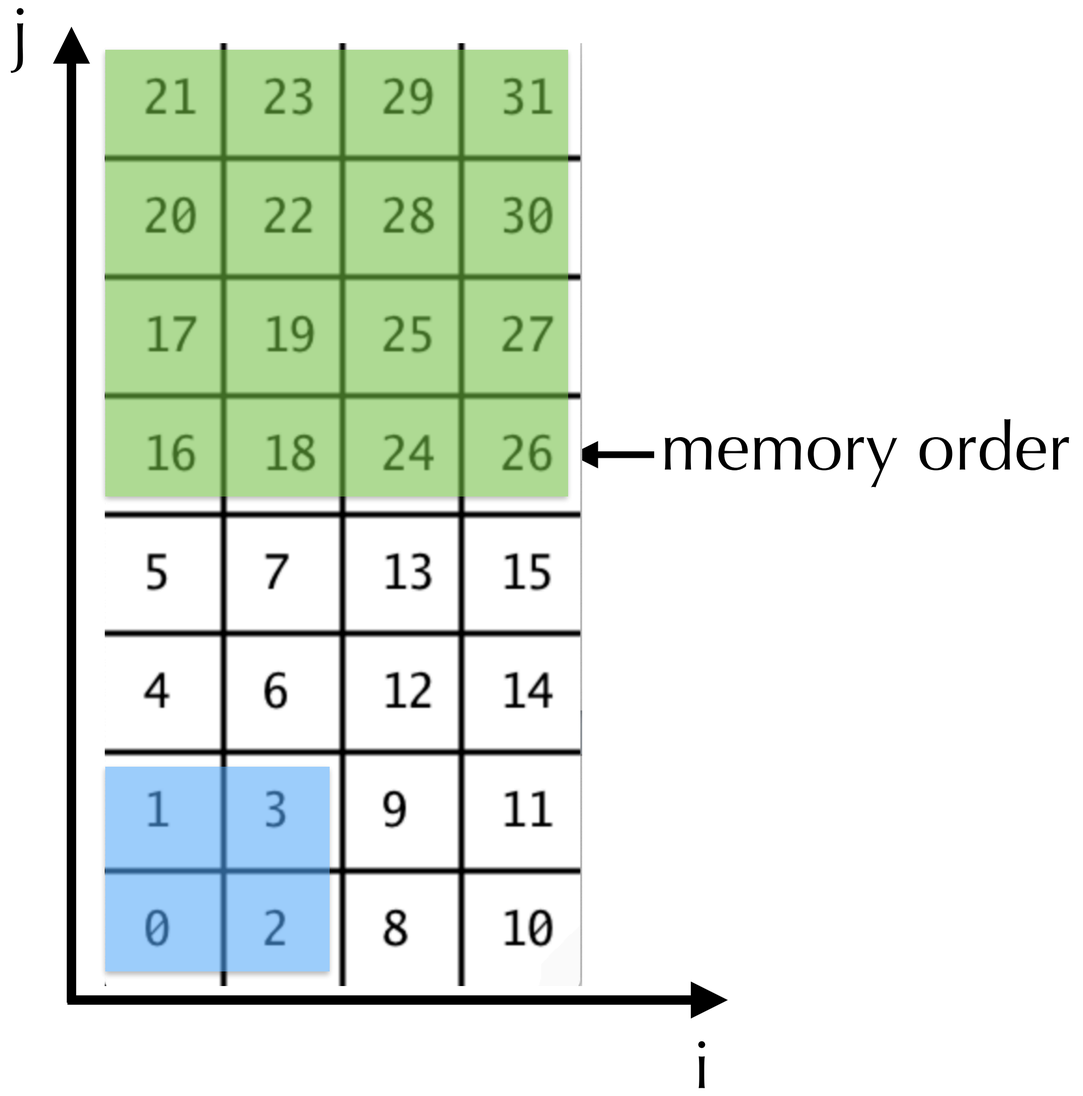
Nesting the `dense` SNodes

```
ti.root.dense(ti.i, 1).dense(ti.j, 8).dense(ti.i, 4).place(a)
```



Nesting the dense SNodes

```
ti.root.dense(ti.ij, (1, 2))
        .dense(ti.ij, 2)
        .dense(ti.ij, 2)
        .place(a)
```

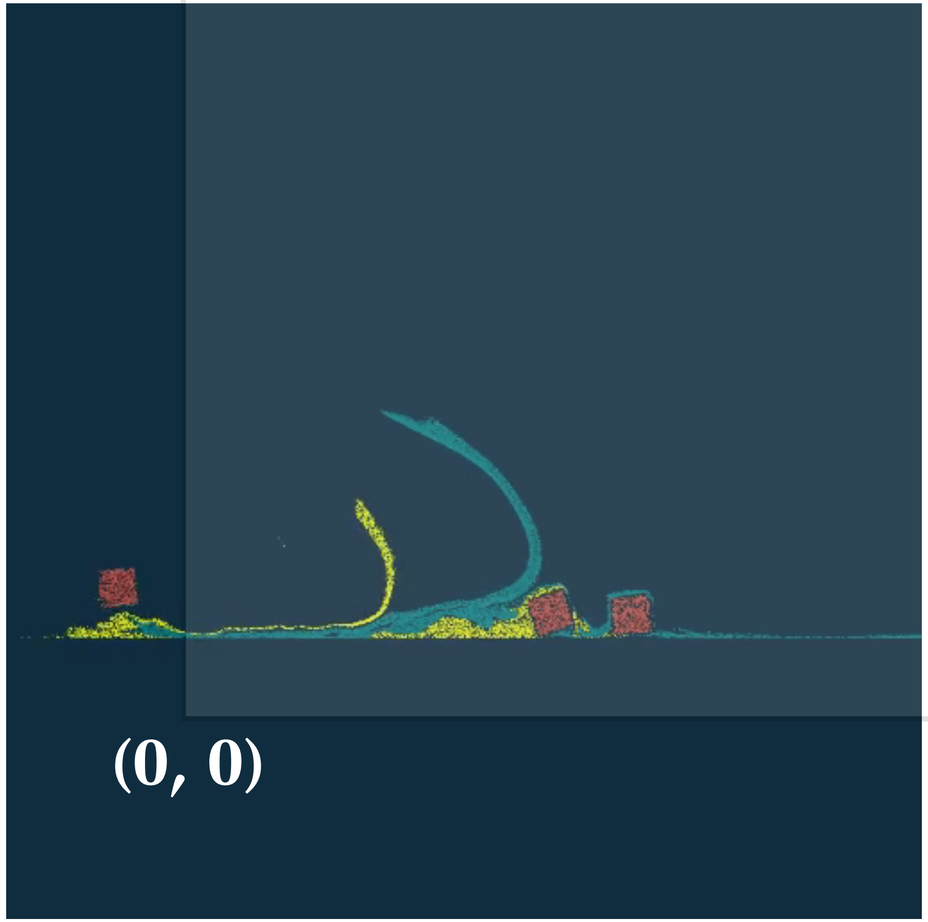


Using coordinate space `offsets`

- `a = ti.field(dtype=ti.i32, shape=(1024, 1024))`
  - a[i, j] with 0 <= i, j < 1024

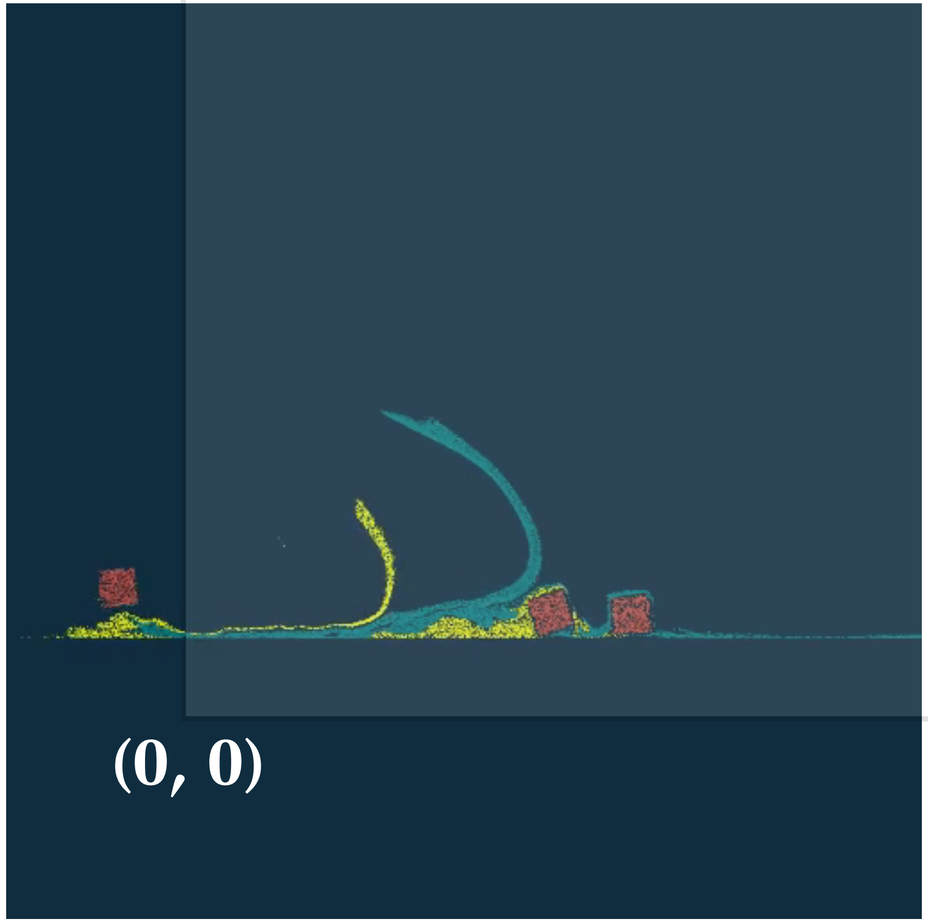
- **What if we want -512 <= i < 512 and -256 <= j < 768?**
- `a = ti.field(dtype=ti.i32, shape=(1024, 1024), offset=(-512, -256))`

Example: unbounded simulation




```
grid_m = ti.field(dtype=ti.f32, shape=(1024, 1024), offset=(-512, -512))
```

Example: unbounded simulation




```
grid_m = ti.field(dtype=ti.f32, shape=(1024, 1024), offset=(-512, -512))
```

What exactly are ti.Matrix.field(shape=(…))?

- **Are they arrays-of-structures (AOS) or structures-of-arrays (SOA)?**
  - `v = ti.Vector.field(2, dtype=ti.i32, shape=(128, 512))` is equivalent to
    - `v = ti.Vector.field(2, dtype=ti.i32)`
    - `ti.root.dense(ti.ij, (128, 512)).place(v(0), v(1))# Array of structures`
  - If you want SOA:
    - `v = ti.Vector.field(2, dtype=ti.i32)`
    - `ti.root.dense(ti.ij, (128, 512)).place(v(0))`
    - `ti.root.dense(ti.ij, (128, 512)).place(v(1))`

AOS v.s. SOA Layout

✦ Array of Structures (AOS)

```
struct Particle {float x, y, z};

Particle particles[8192];
```

✦ Structure of Arrays (SOA)

```
struct particles {
   float particle_x[8192];
   float particle_y[8192];
   float particle_z[8192];
};
```

Array of Structures

Linear Memory

Each particle's fields are continuous in memory

Array of Structures: Sequential Access

Linear Memory

Cacheline

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Cacheline Utilization: 100%

Array of Structures: Random Access

Linear Memory

Cacheline

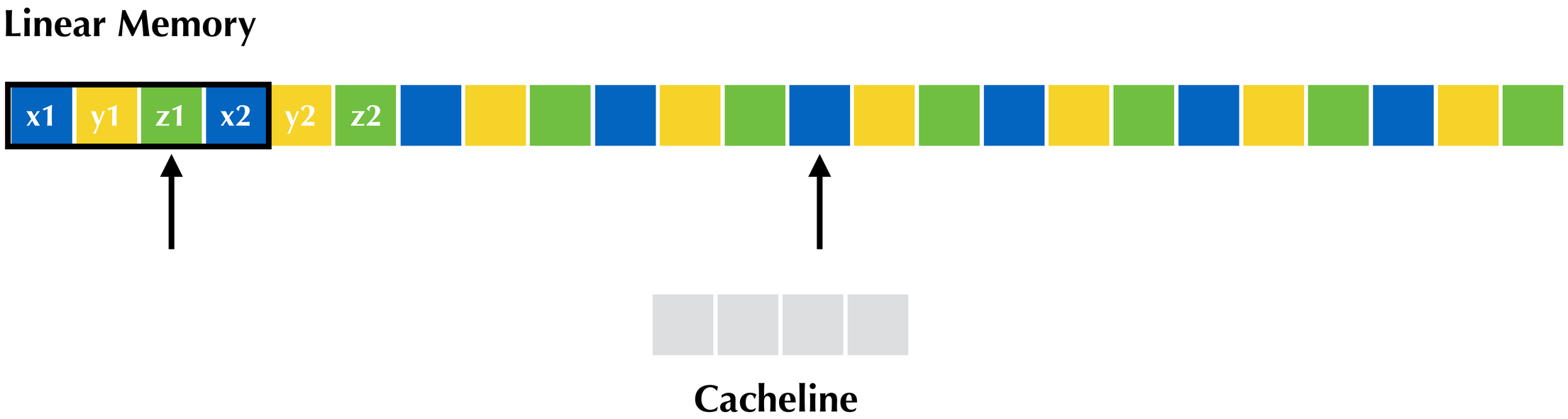
Array of Structures: Random Access

Array of Structures: Random Access

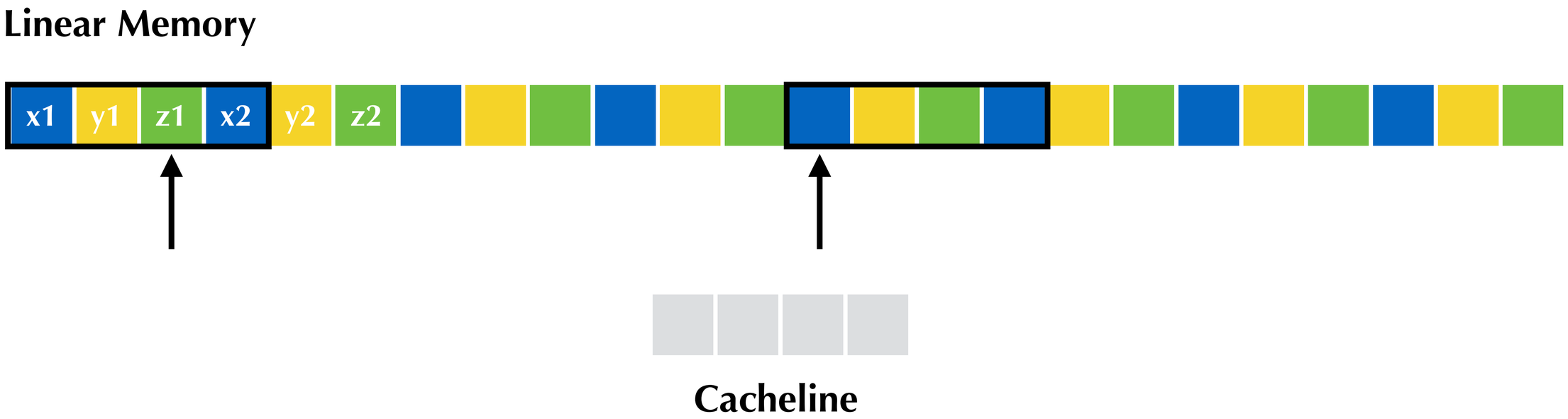
Array of Structures: Random Access

Array of Structures: Random Access

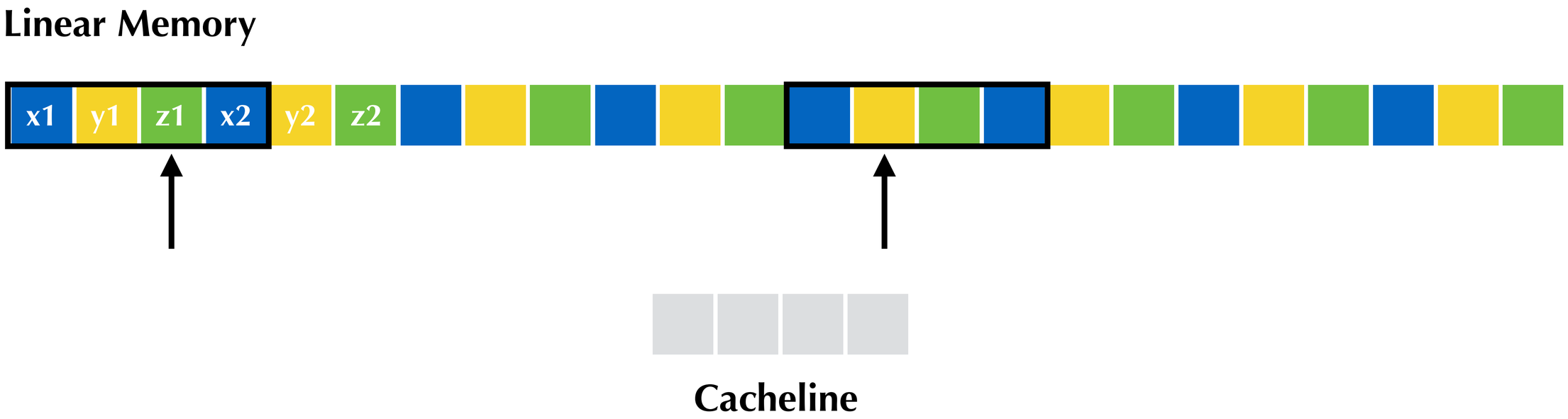
Array of Structures: Random Access



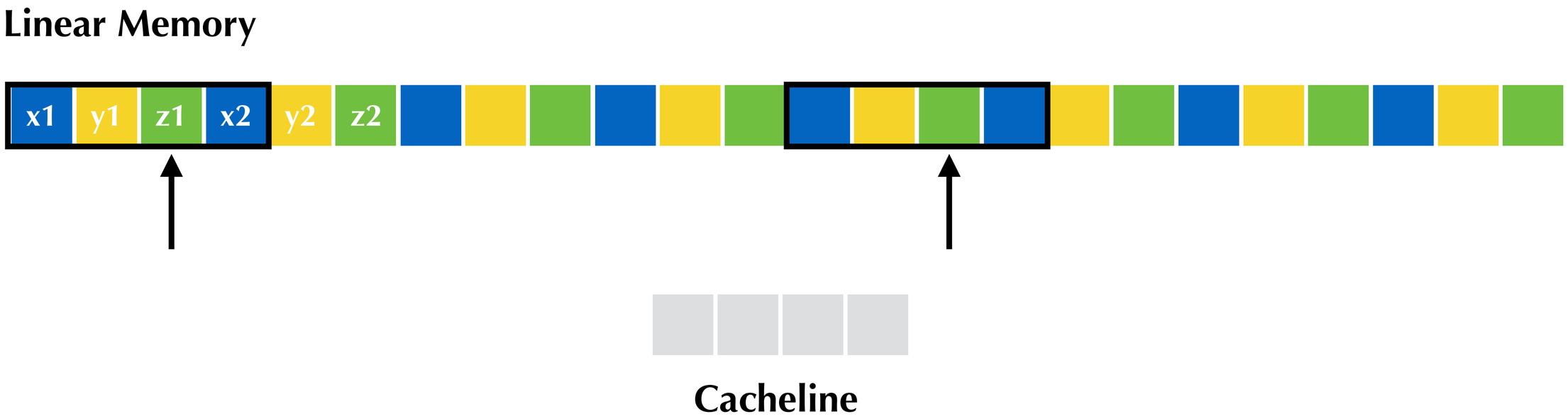
Array of Structures: Random Access



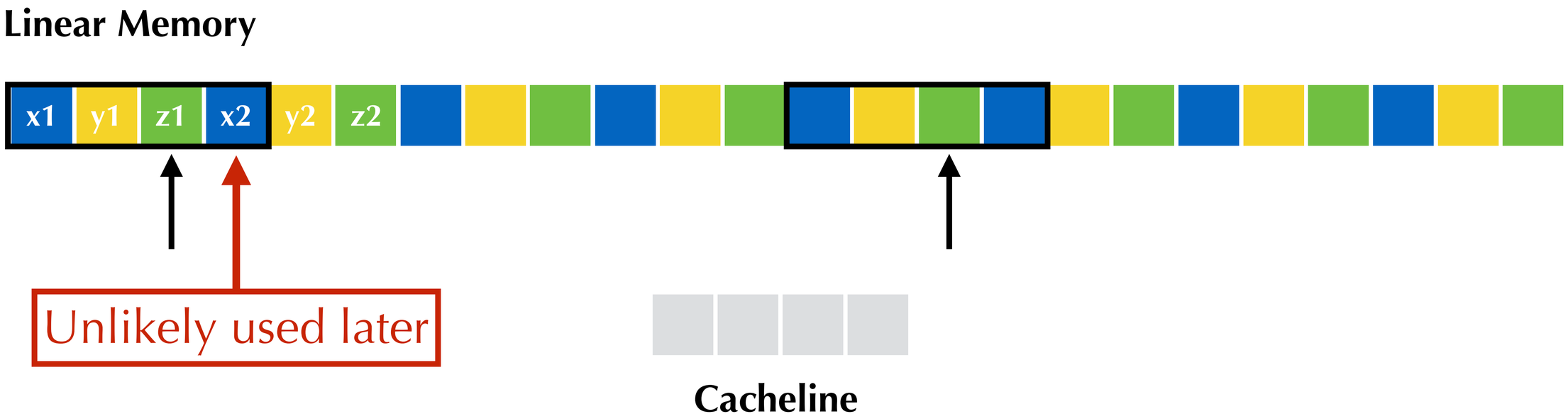
Array of Structures: Random Access



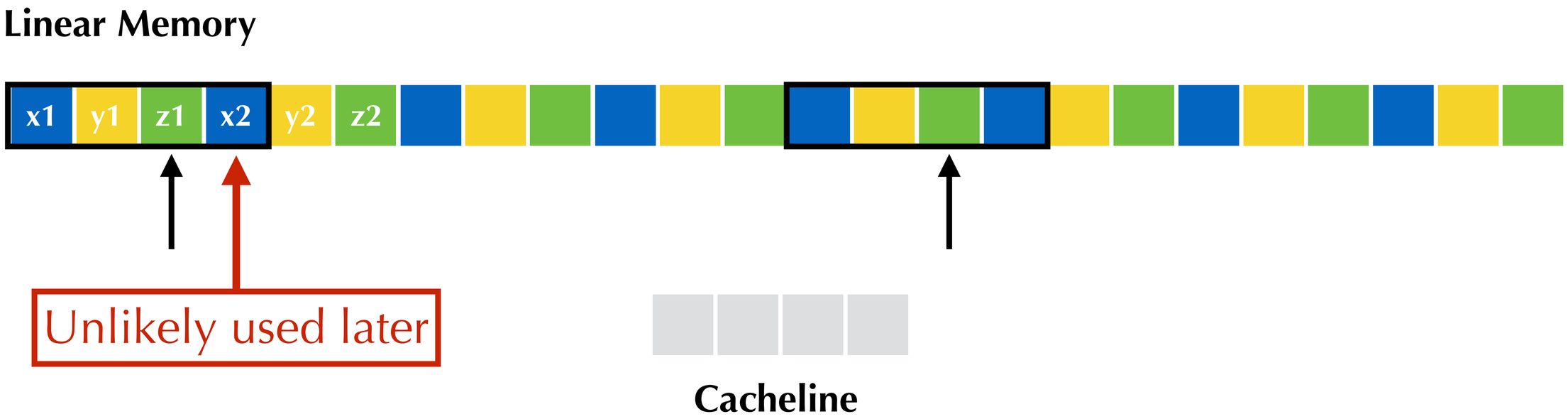
Array of Structures: Random Access



Array of Structures: Random Access



Array of Structures: Random Access



Cacheline Utilization: 75%

Array of Structures

Linear Memory

Each particle's fields are continuous in memory

Structure of Arrays

Linear Memory

Each field's particle instances are continuous in memory

Structure of Arrays: Sequential Access

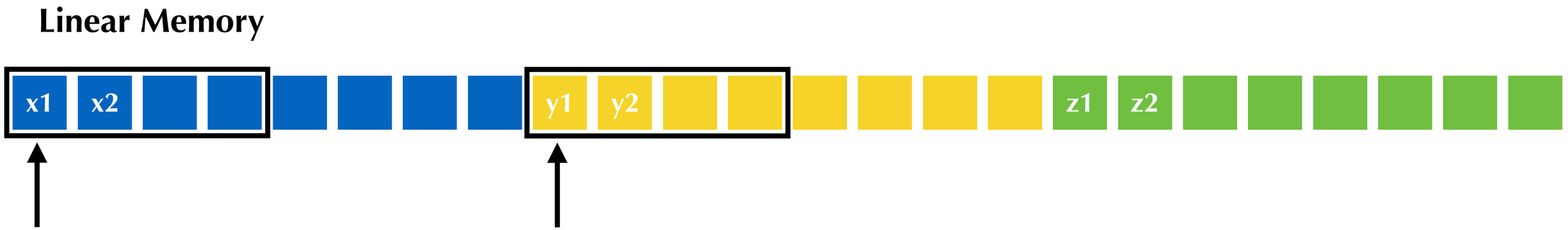
Linear Memory

Structure of Arrays: Sequential Access

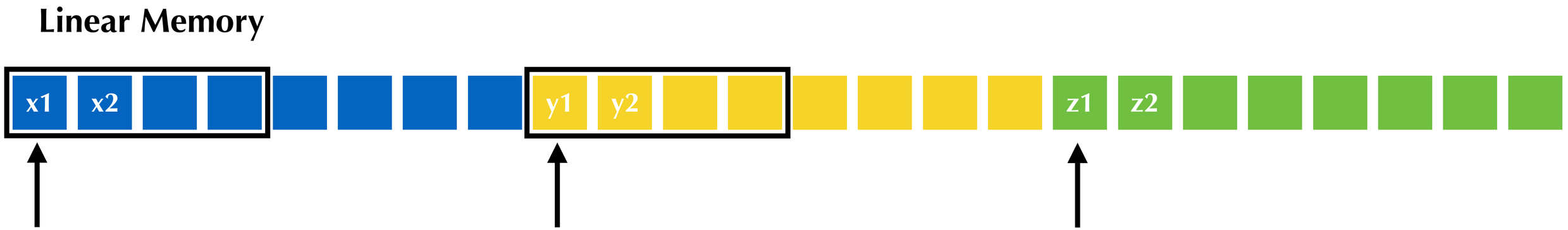
Structure of Arrays: Sequential Access

Structure of Arrays: Sequential Access

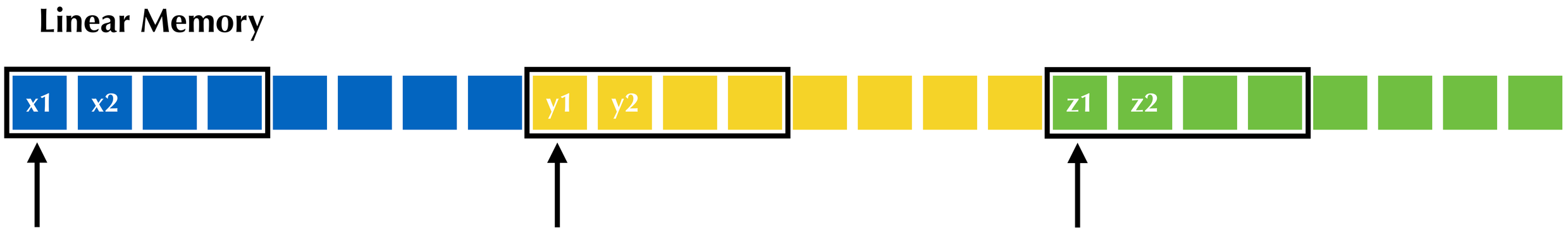
Structure of Arrays: Sequential Access



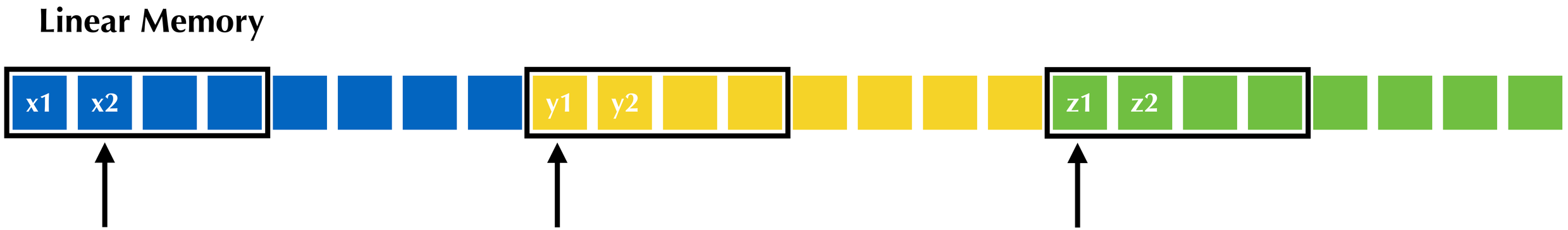
Structure of Arrays: Sequential Access



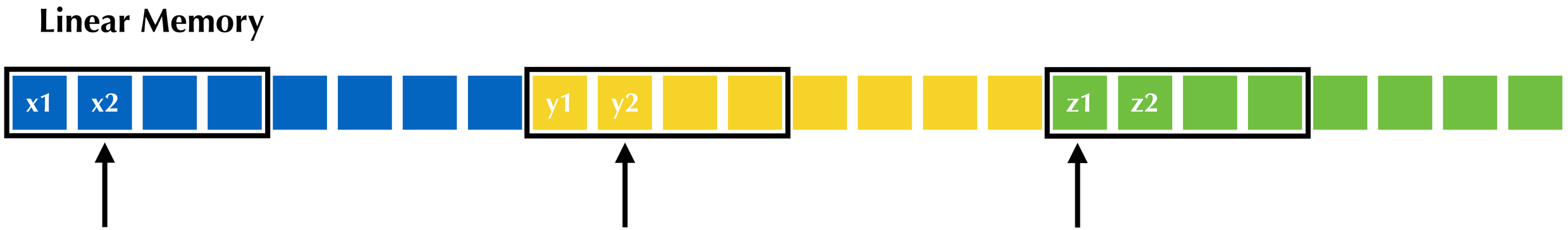
Structure of Arrays: Sequential Access



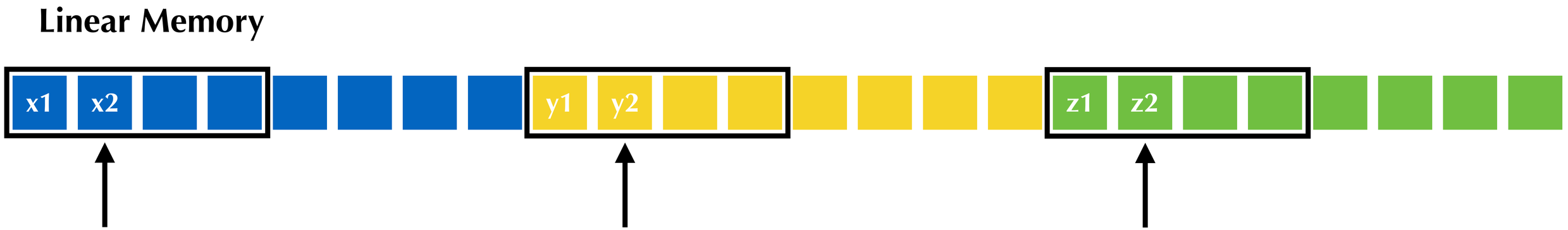
Structure of Arrays: Sequential Access



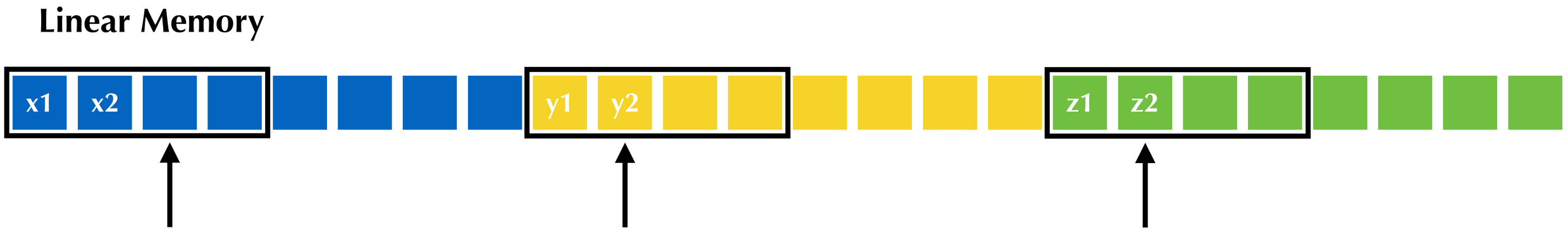
Structure of Arrays: Sequential Access



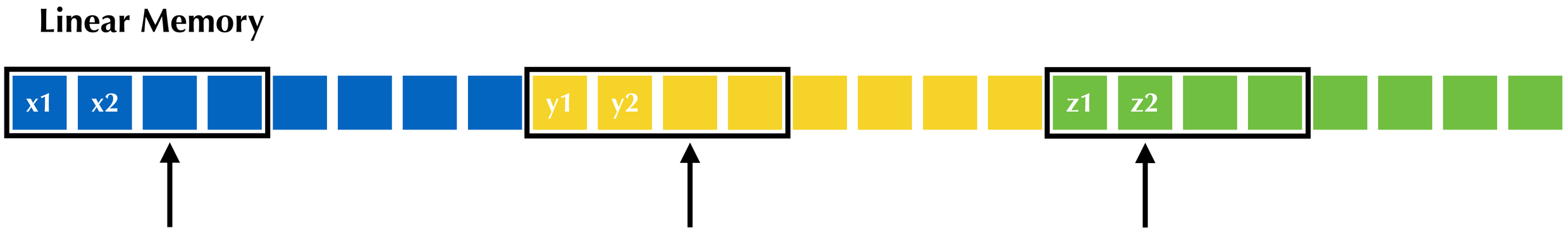
Structure of Arrays: Sequential Access



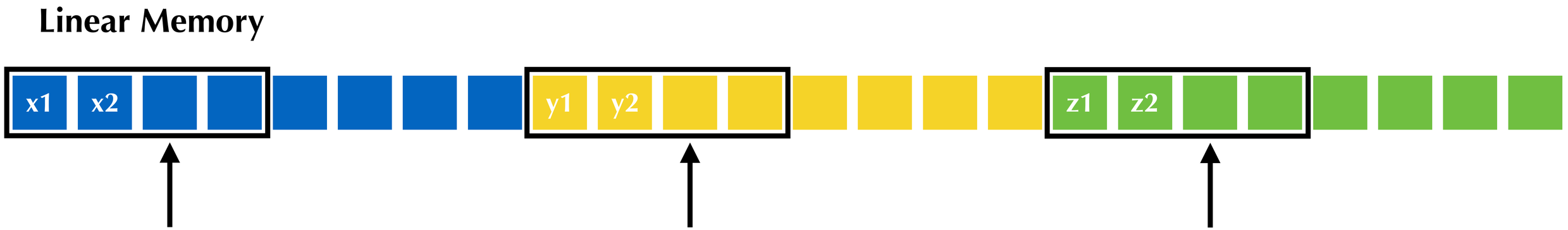
Structure of Arrays: Sequential Access



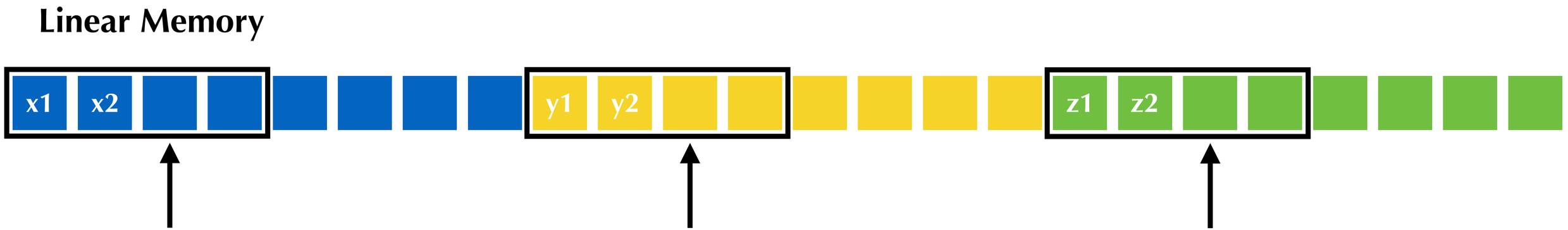
Structure of Arrays: Sequential Access



Structure of Arrays: Sequential Access



Structure of Arrays: Sequential Access



Cacheline Utilization: 100%

Structure of Arrays: Random Access

Linear Memory

Assuming cache size=3

Structure of Arrays: Random Access

Linear Memory

Assuming cache size=3

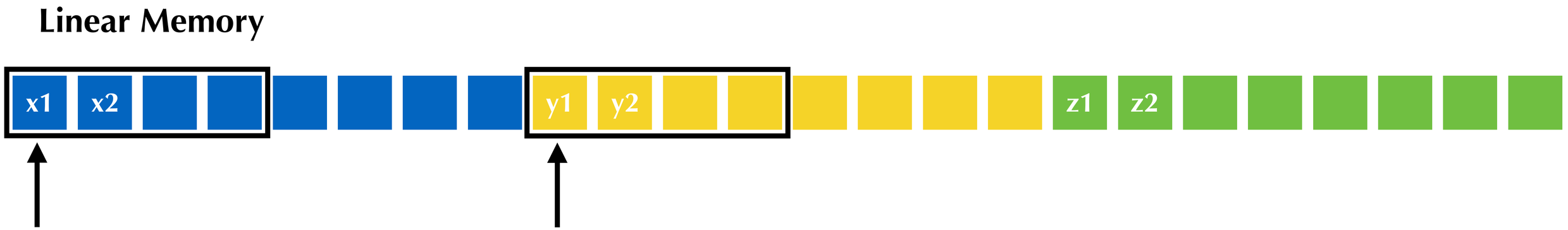
Structure of Arrays: Random Access

Assuming cache size=3

Structure of Arrays: Random Access

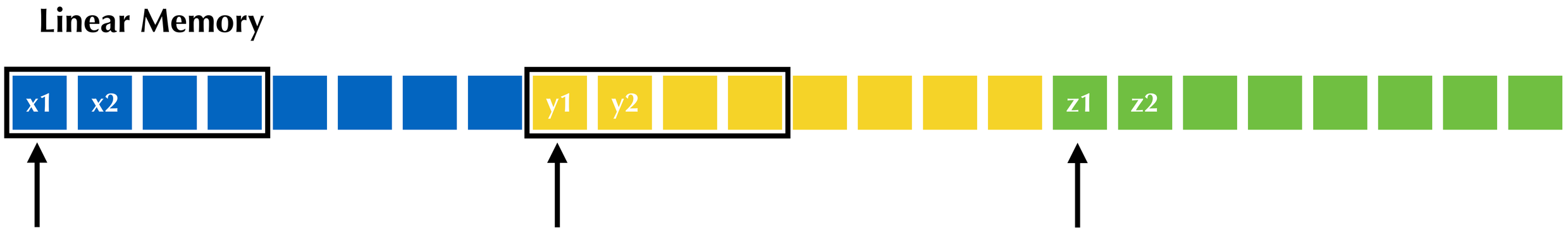
Assuming cache size=3

Structure of Arrays: Random Access



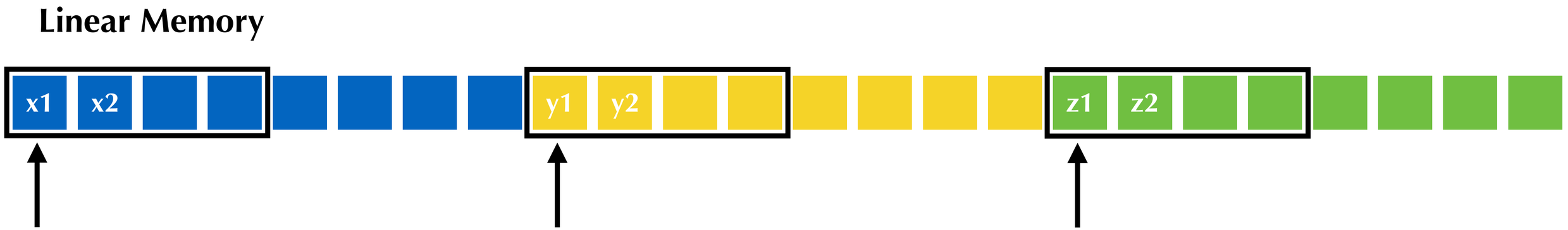
Assuming cache size=3

Structure of Arrays: Random Access



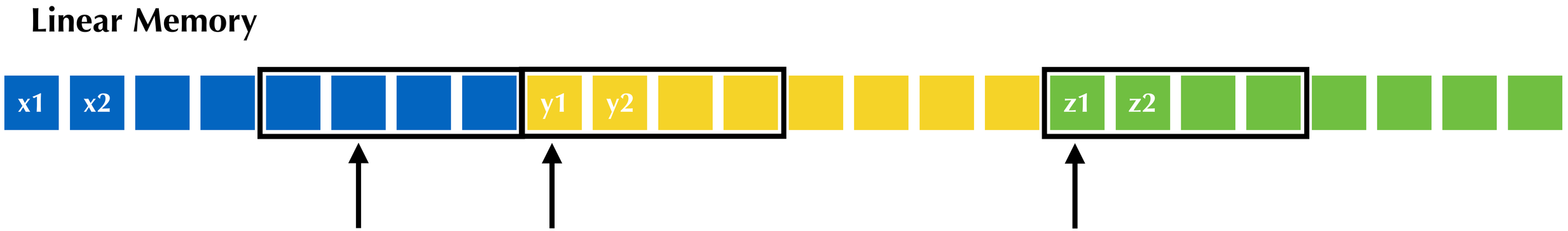
Assuming cache size=3

Structure of Arrays: Random Access



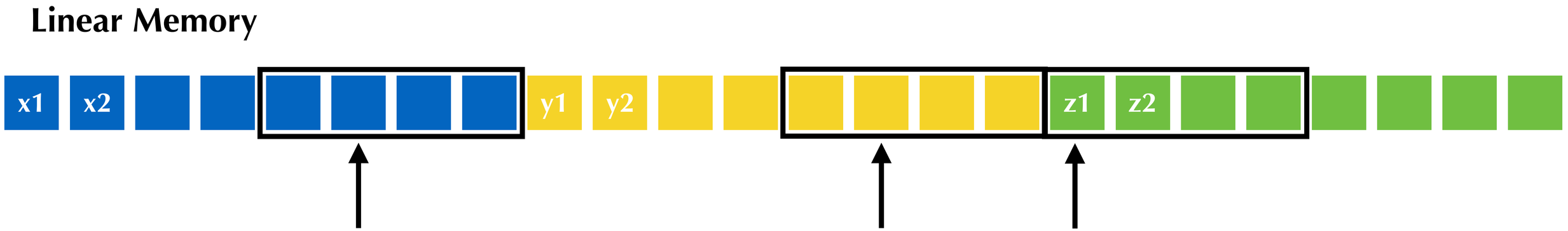
Assuming cache size=3

Structure of Arrays: Random Access



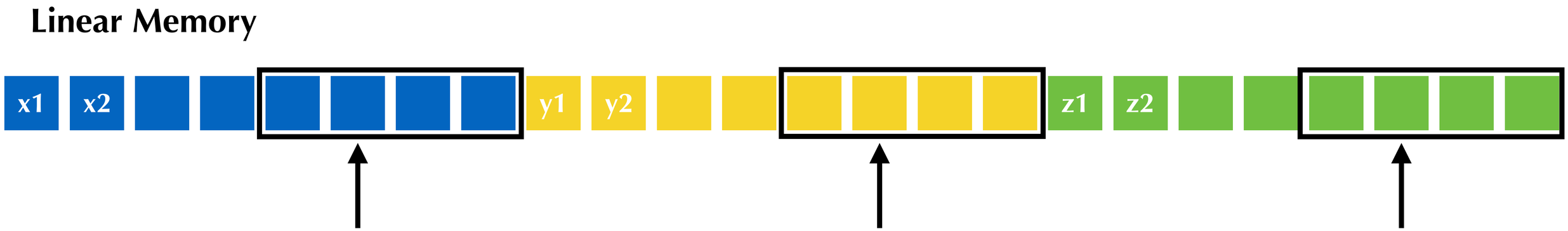
Assuming cache size=3

Structure of Arrays: Random Access



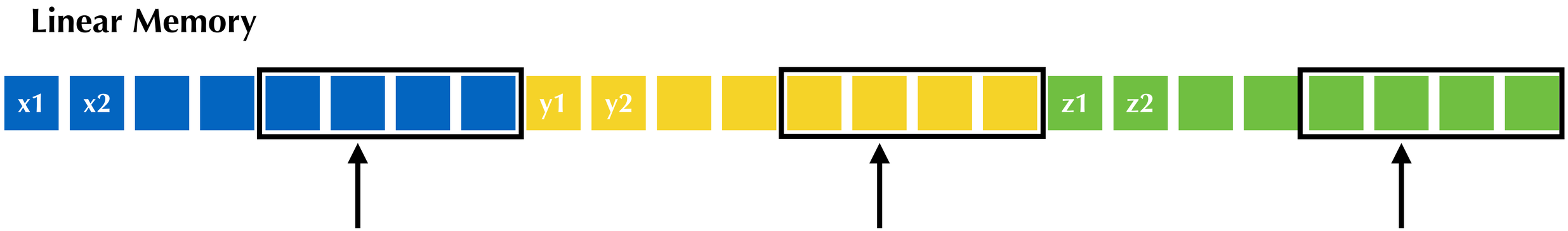
Assuming cache size=3

Structure of Arrays: Random Access



Assuming cache size=3

Structure of Arrays: Random Access



Assuming cache size=3

Cacheline Utilization: 25%

Data Structures for MPM Particles

- **SOA: very efficient when sorted**
  - Coalesced access on GPUs
  - Large number of streams (e.g. 20): may invalidate prefetchers on CPU
    - not a problem for GPU - GPUs are designed for streaming and have no prefetching
  - Random access: low cacheline utilization
- **AOS: efficient even unsorted**
  - Random access: much better than SOA but should still be avoided if possible (cache/TLB misses)
  - Sequential access: Slightly inferior than AOS
    - Vector lane shuffling on CPUs
    - Non-coalseced access on GPUs
  - No sorting needed

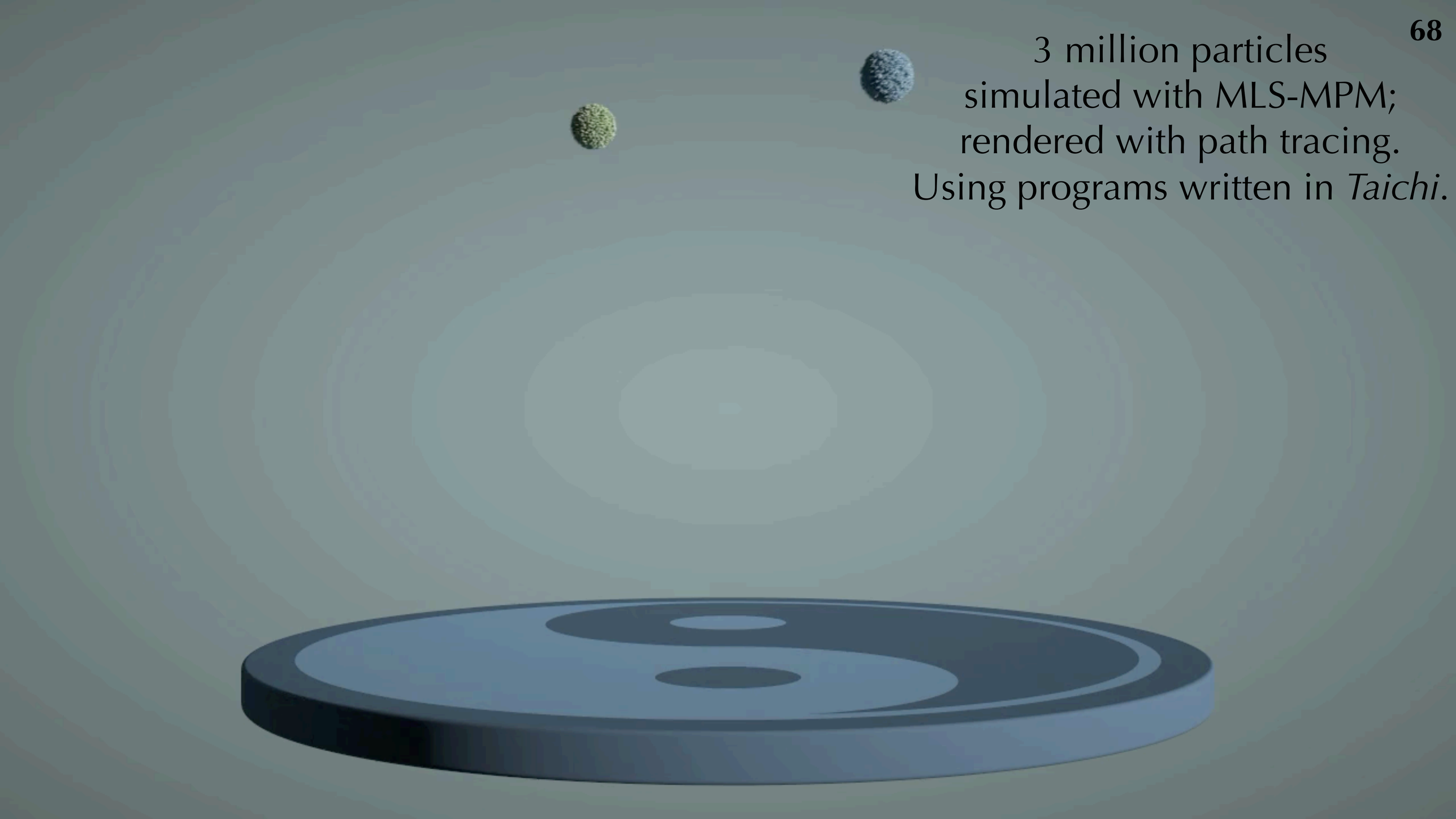
Looping over non-leaf SNodes

```python
import taichi as ti
ti.init()

a = ti.field(dtype=ti.i32)

blk = ti.root.dense(ti.ij, 2)
blk.dense(ti.ij, (2, 4)).place(a)

@ti.kernel
def task():
    for i, j in blk:
        print(i, j)

task()
```

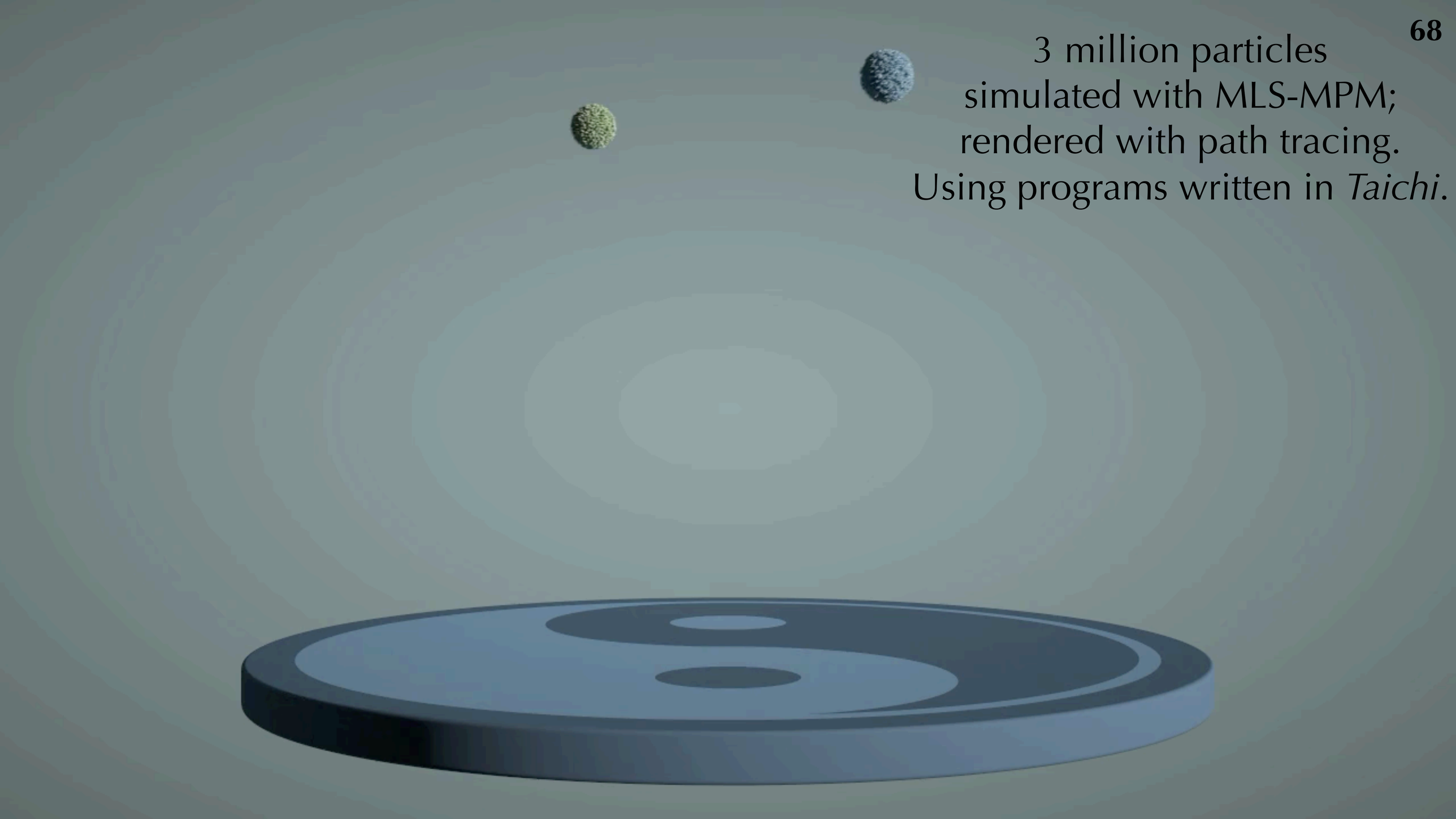
```
output:

0 0
0 4
2 0
2 4
```

Using Sparse Data Structures in Taichi

3 million particles
simulated with MLS-MPM;
rendered with path tracing.
Using programs written in *Taichi*.



3 million particles
simulated with MLS-MPM;
rendered with path tracing.
Using programs written in *Taichi*.

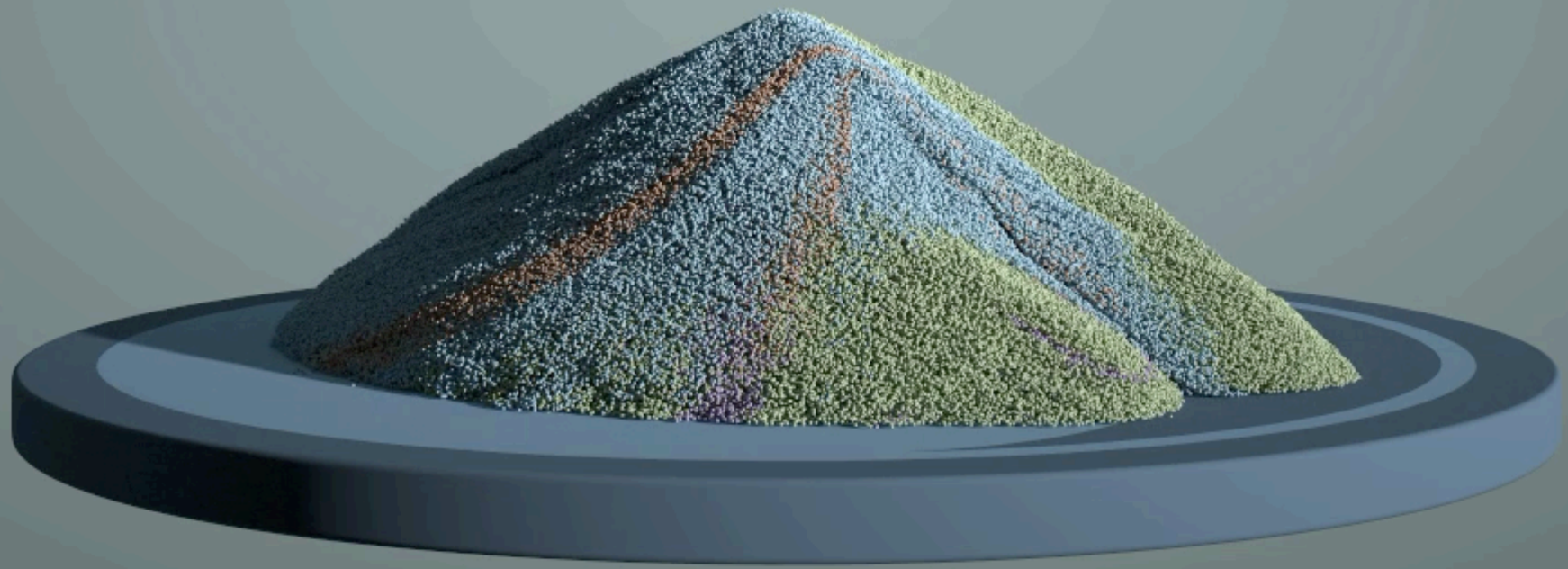


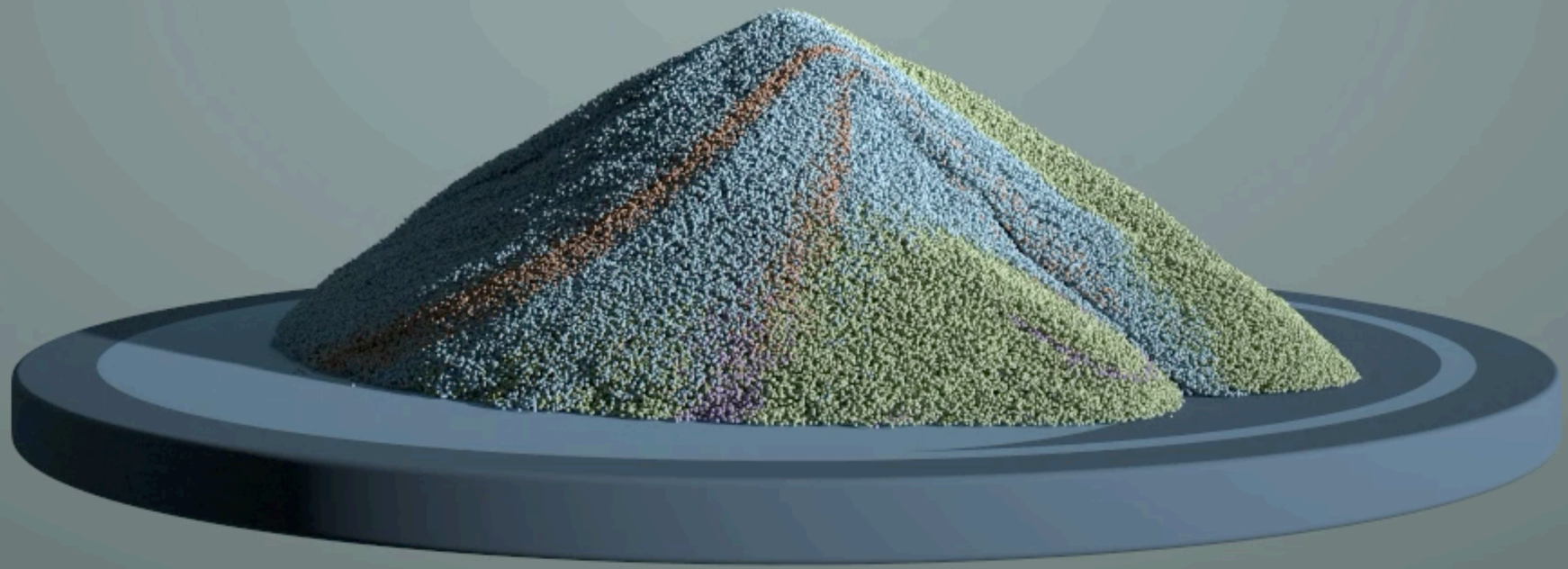
3 million particles
simulated with MLS-MPM;
rendered with path tracing.
Using programs written in *Taichi*.

3 million particles
simulated with MLS-MPM;
rendered with path tracing.
Using programs written in *Taichi*.

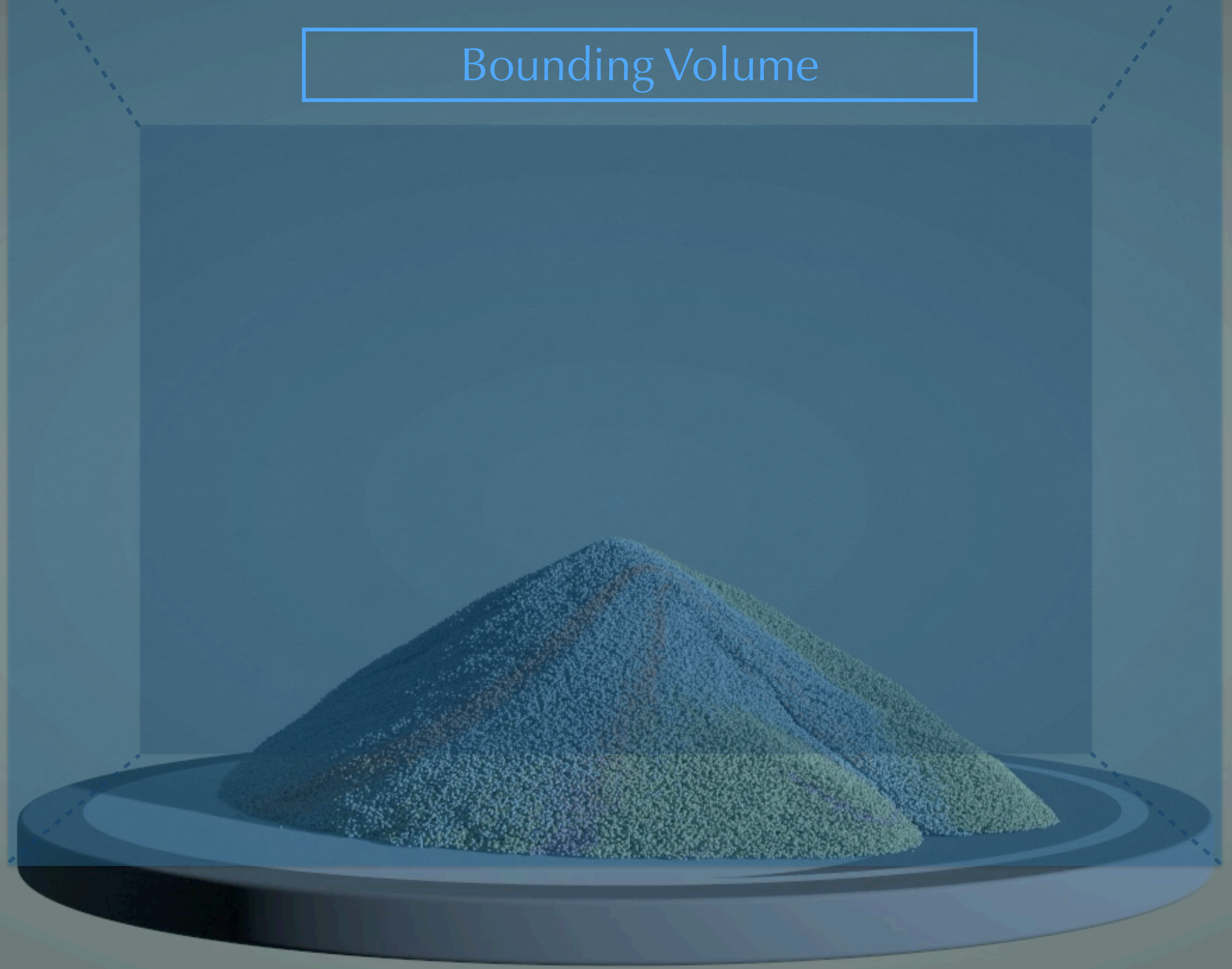
3 million particles
simulated with MLS-MPM;
rendered with path tracing.
Using programs written in *Taichi*.

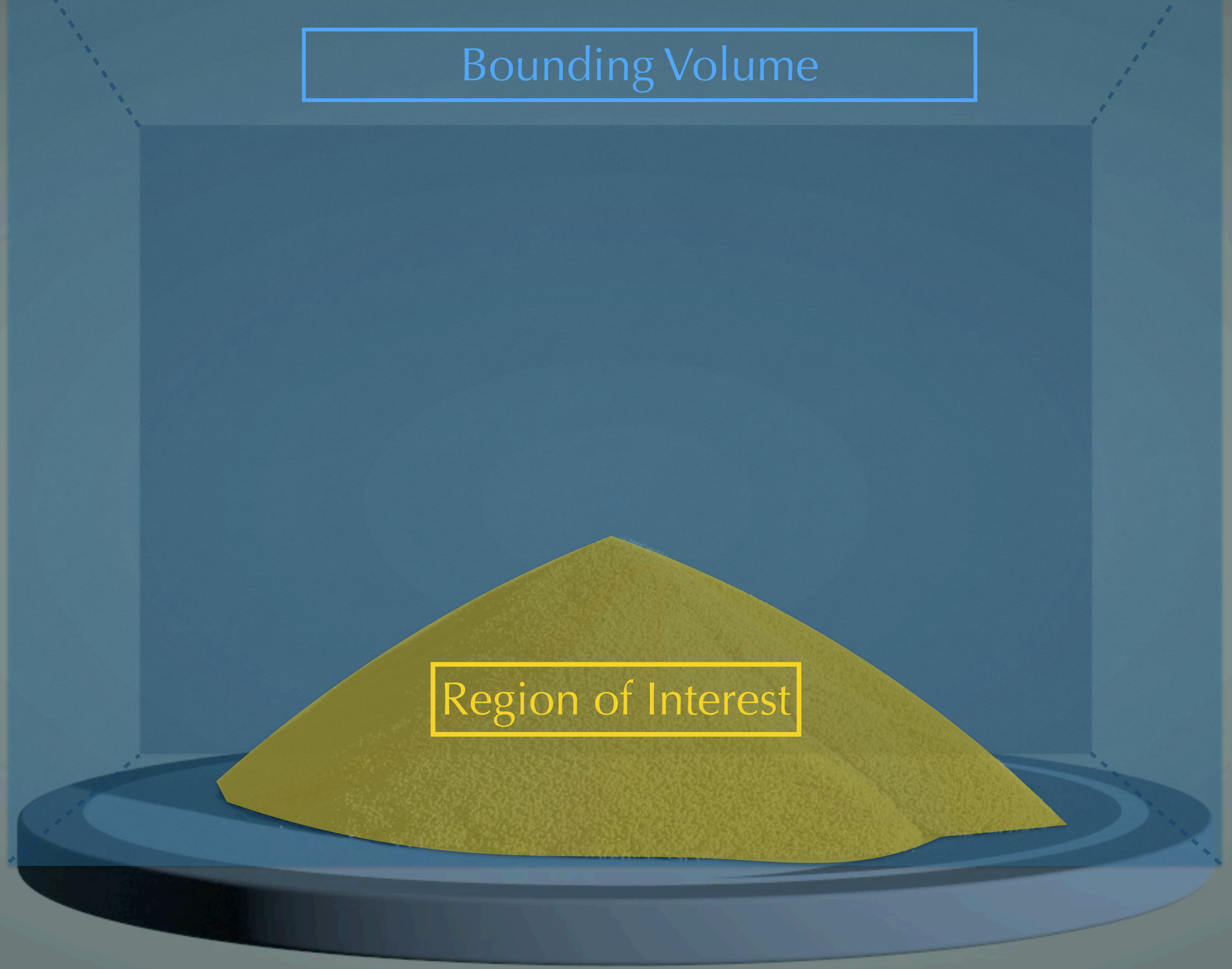
3 million particles
simulated with MLS-MPM;
rendered with path tracing.
Using programs written in *Taichi*.





Bounding Volume



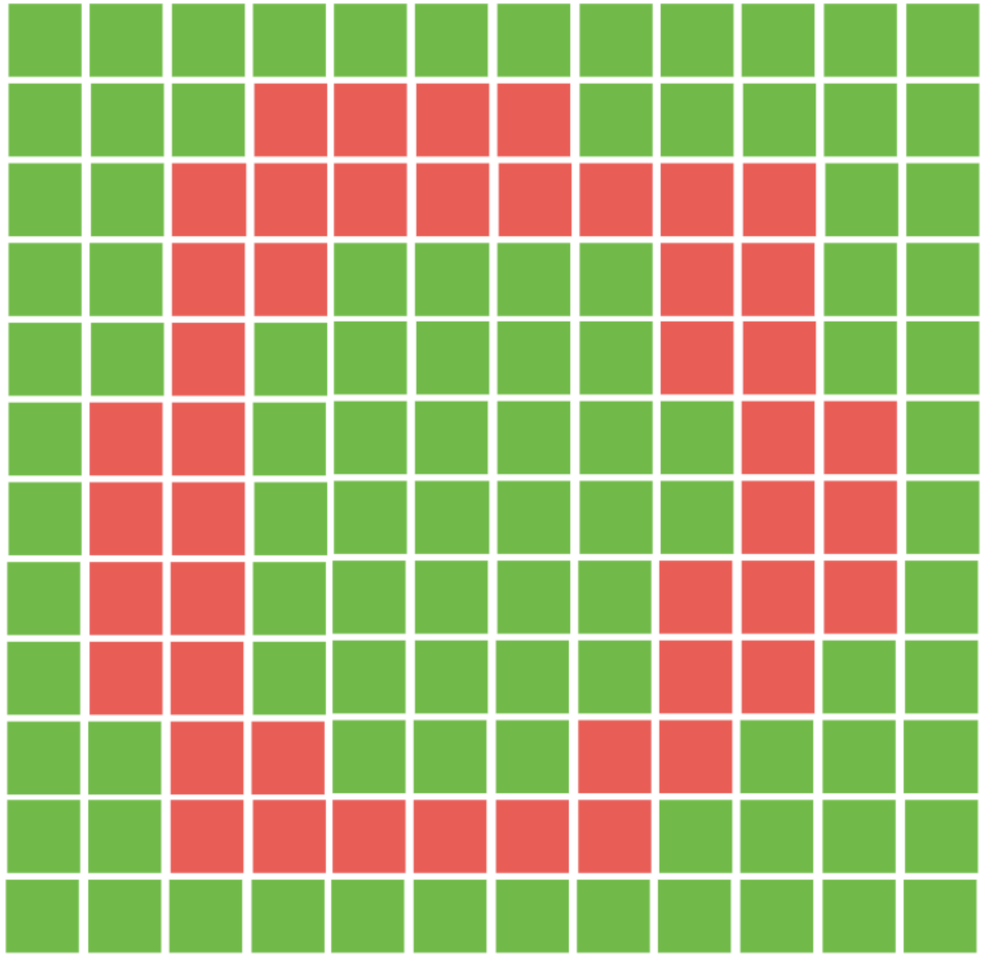

Bounding Volume
Region of Interest

Bounding Volume

Spatial Sparsity:
Regions of interest only occupy a small fraction of the bounding volume.

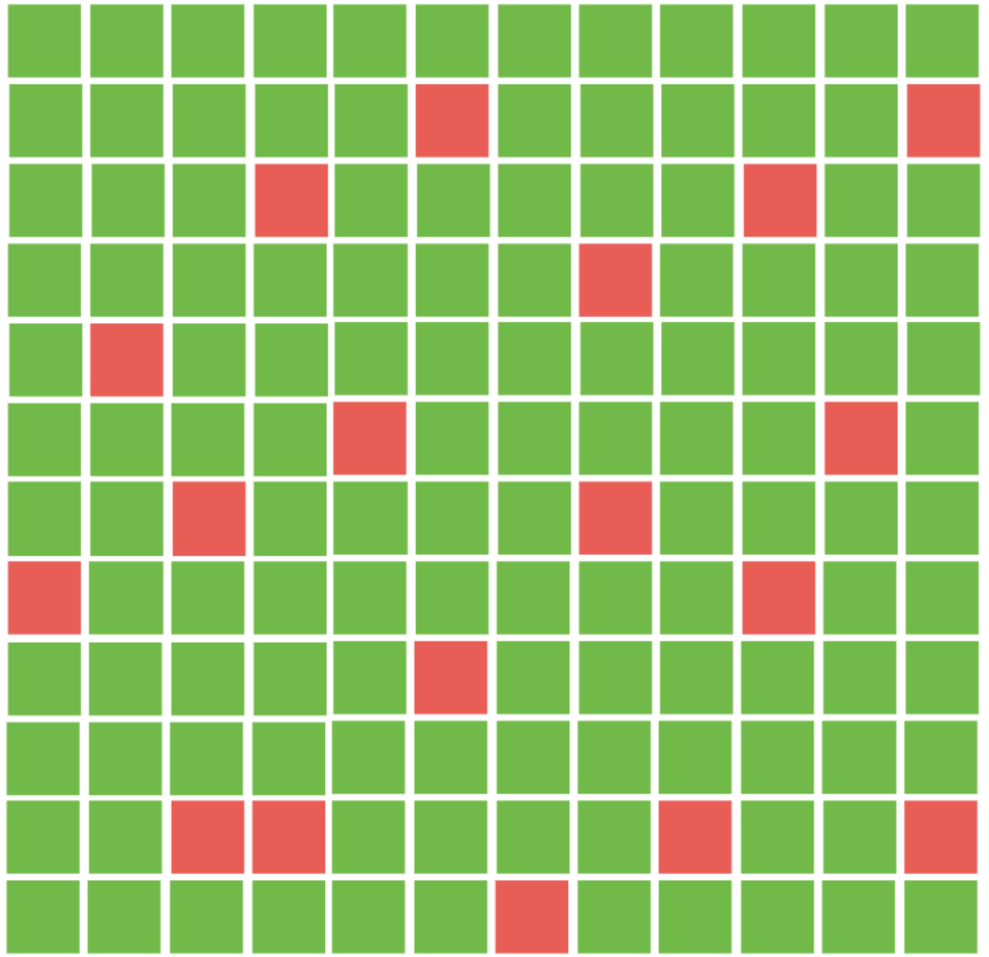
Spatial Sparsity

globally sparse, locally dense



I.e., when blocking at the leaf level makes sense.

"General" Sparsity



OpenVDB[1]

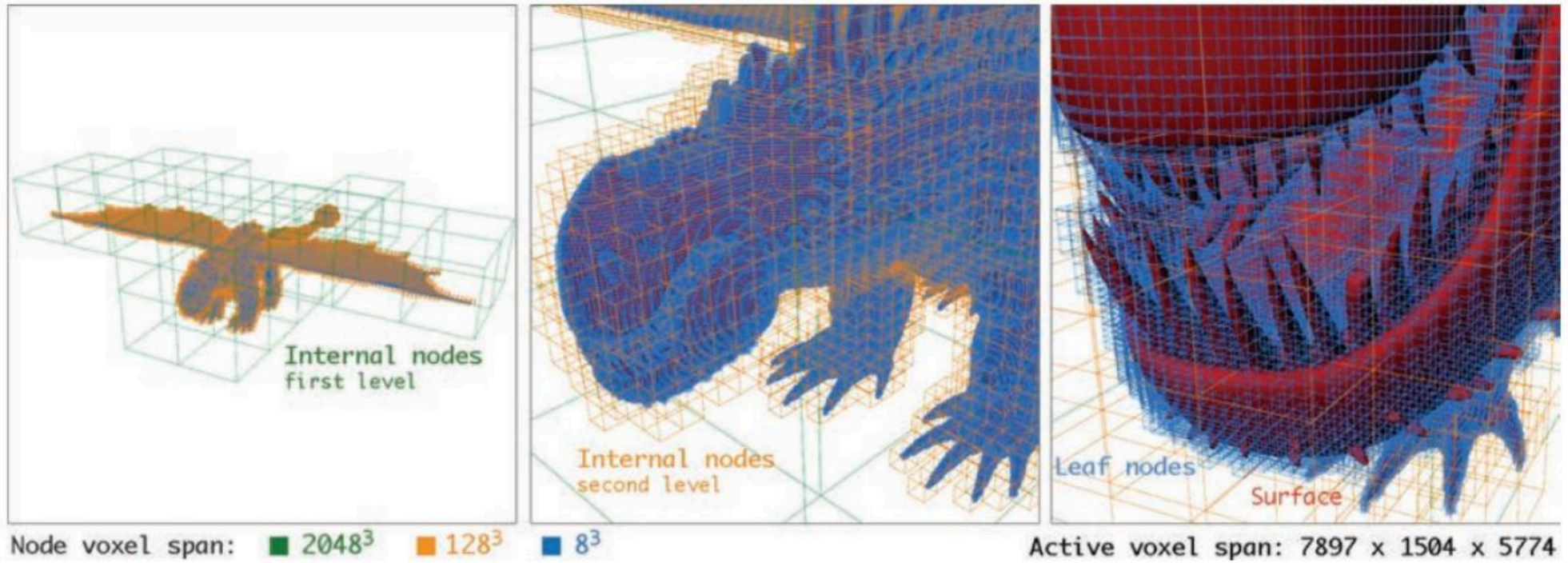
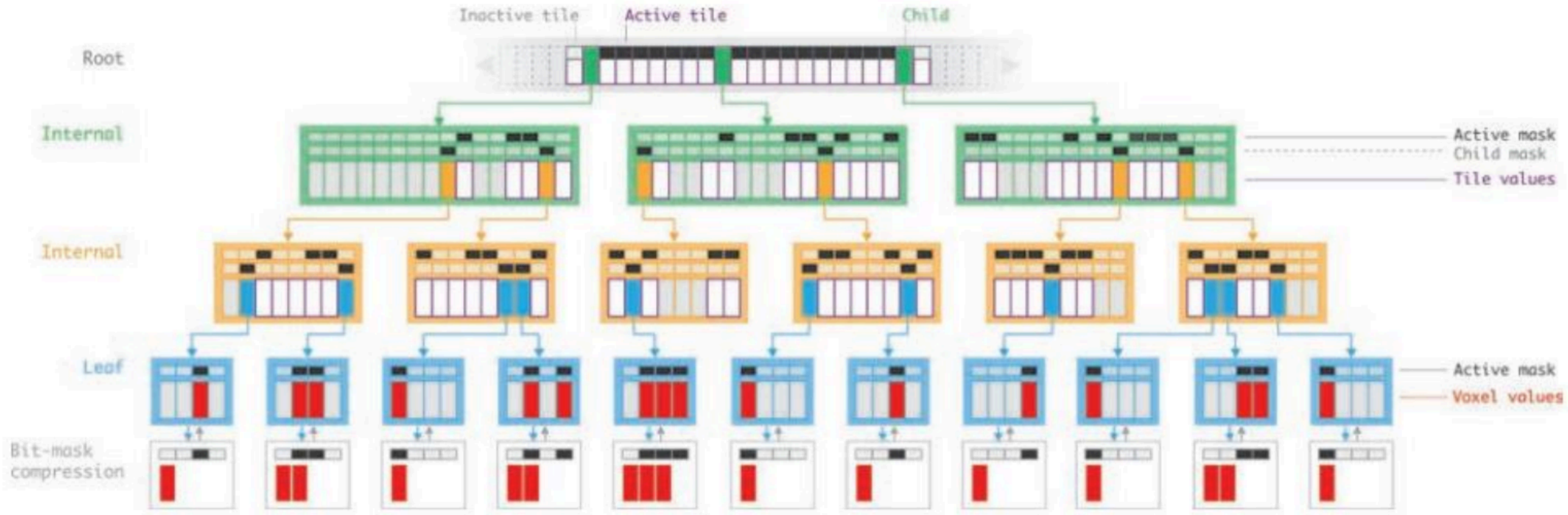


Fig. 4. High-resolution VDB created by converting polygonal model from *How To Train Your Dragon* to a narrow-band level set. The bounding resolution of the 228 million active voxels is 7897 × 1504 × 5774 and the memory footprint of the VDB is 1GB, versus the $\frac{1}{4}$ TB for a corresponding dense volume. This VDB is configured with `LeafNodes` (blue) of size 8^3 and two levels of `InternalNodes` (green/orange) of size 16^3. The index extents of the various nodes are shown as colored wireframes, and a polygonal mesh representation of the zero level set is shaded red. Images are courtesy of *DreamWorks Animation*.

[1] K. Museth (2013). "VDB: High-resolution sparse volumes with dynamic topology". In: *ACM transactions on graphics (TOG)* 32.3, pp. 1–22.

OpenVDB[2]



[2]K. Museth (2013). "VDB: High-resolution sparse volumes with dynamic topology". In: *ACM transactions on graphics (TOG)* 32.3, pp. 1–22.

Using Sparse Data Structures is Hard

Boundary Conditions

Maintaining Topology

Memory Management

Parallelization & Load Balancing

…

Data Structure Overhead

Using Sparse Data Structures is Hard

Boundary Conditions

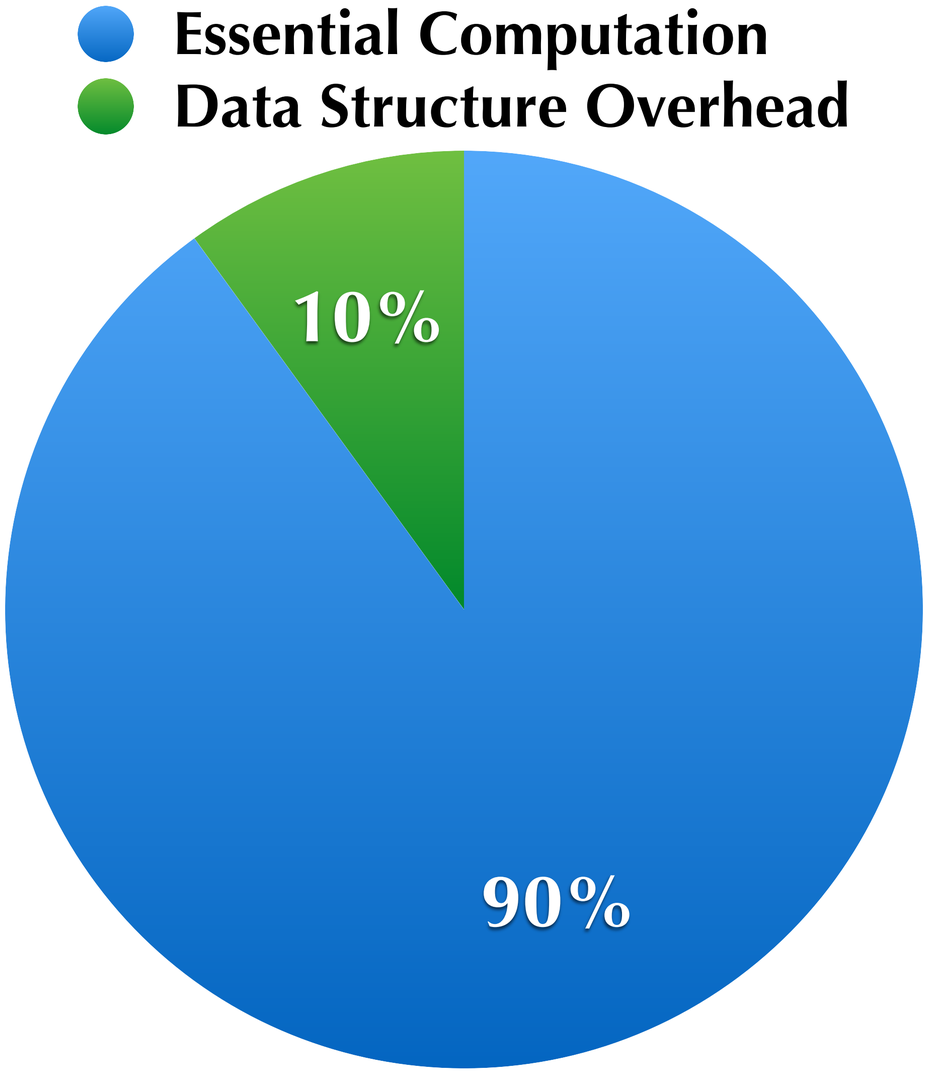
Maintaining Topology

Memory Management

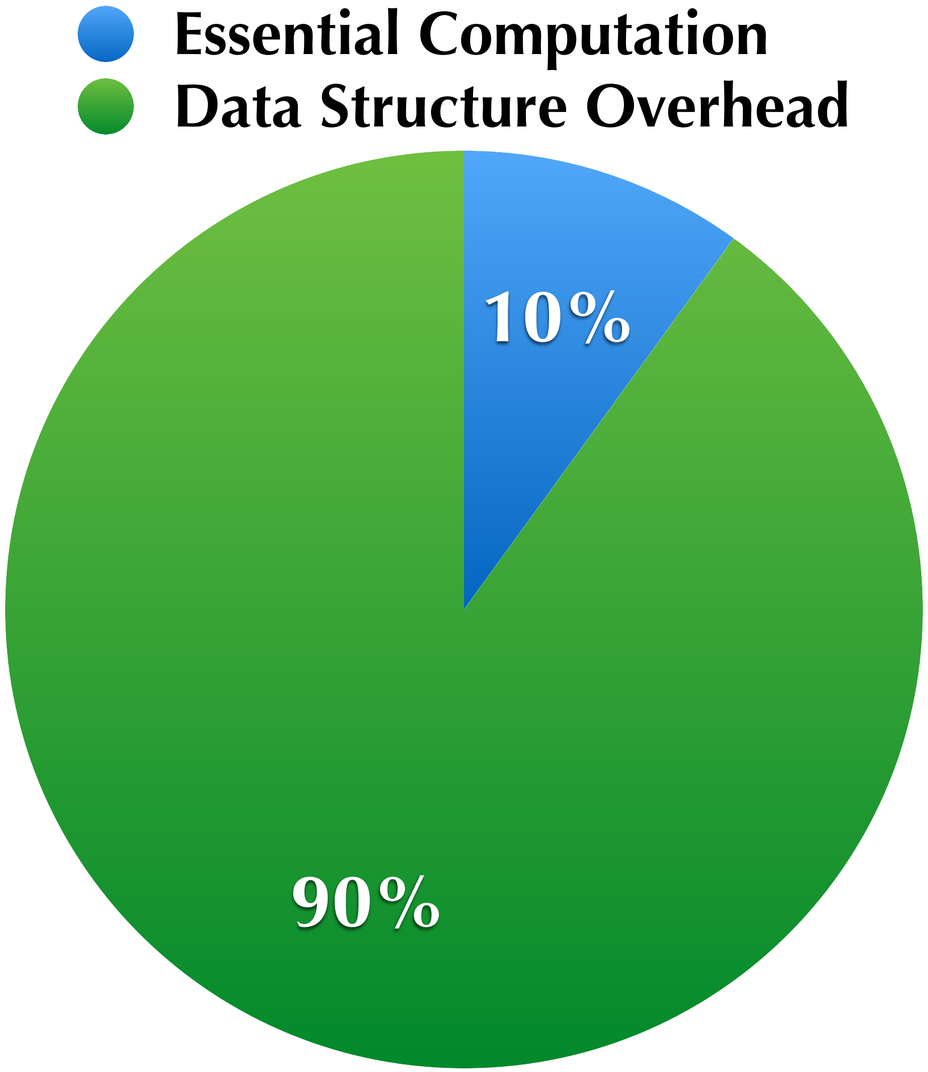
Parallelization & Load Balancing

…

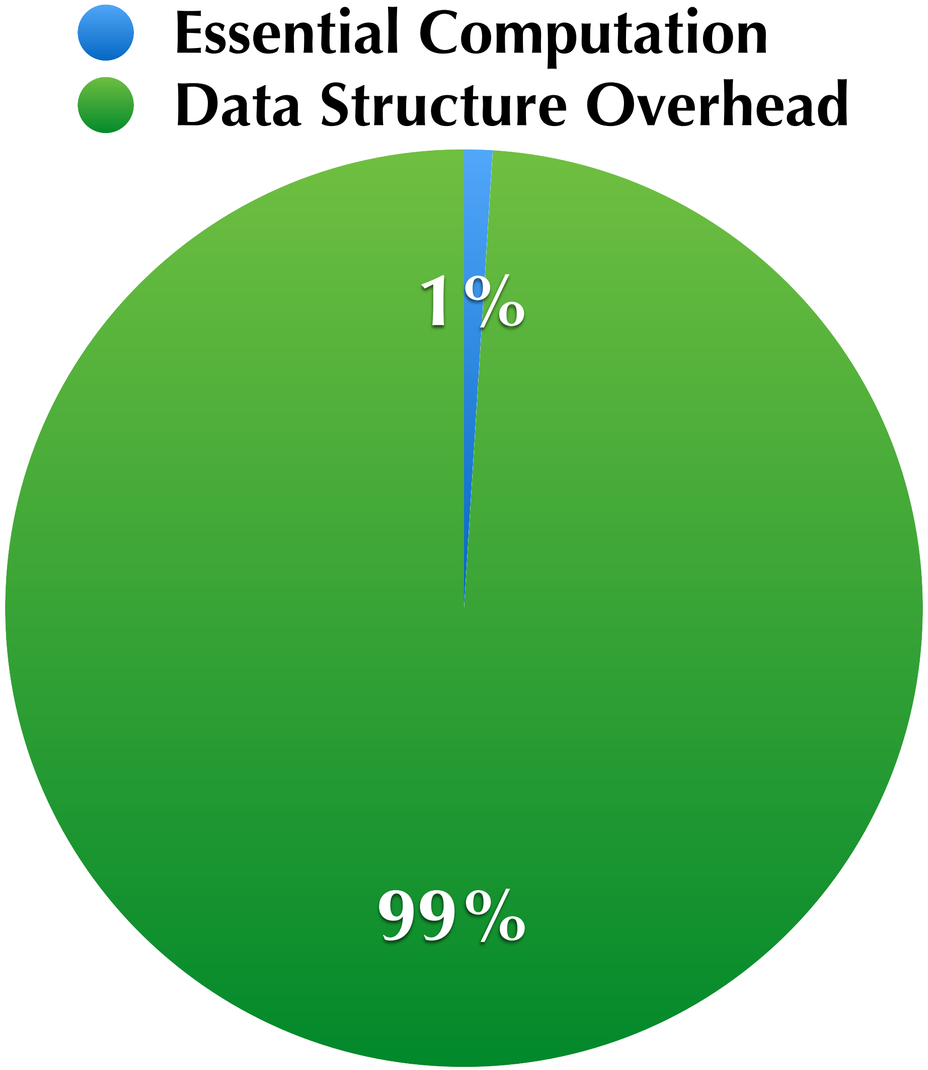
Data Structure Overhead



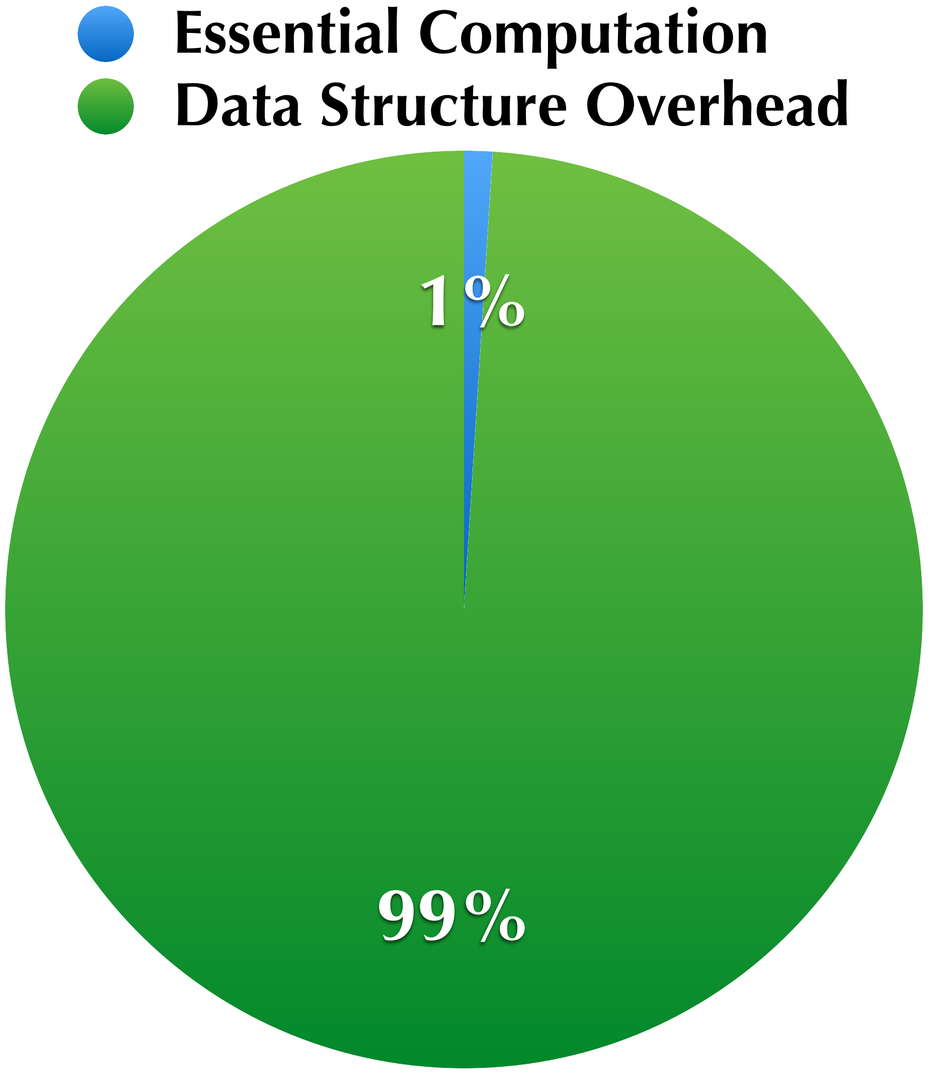
Ideally…



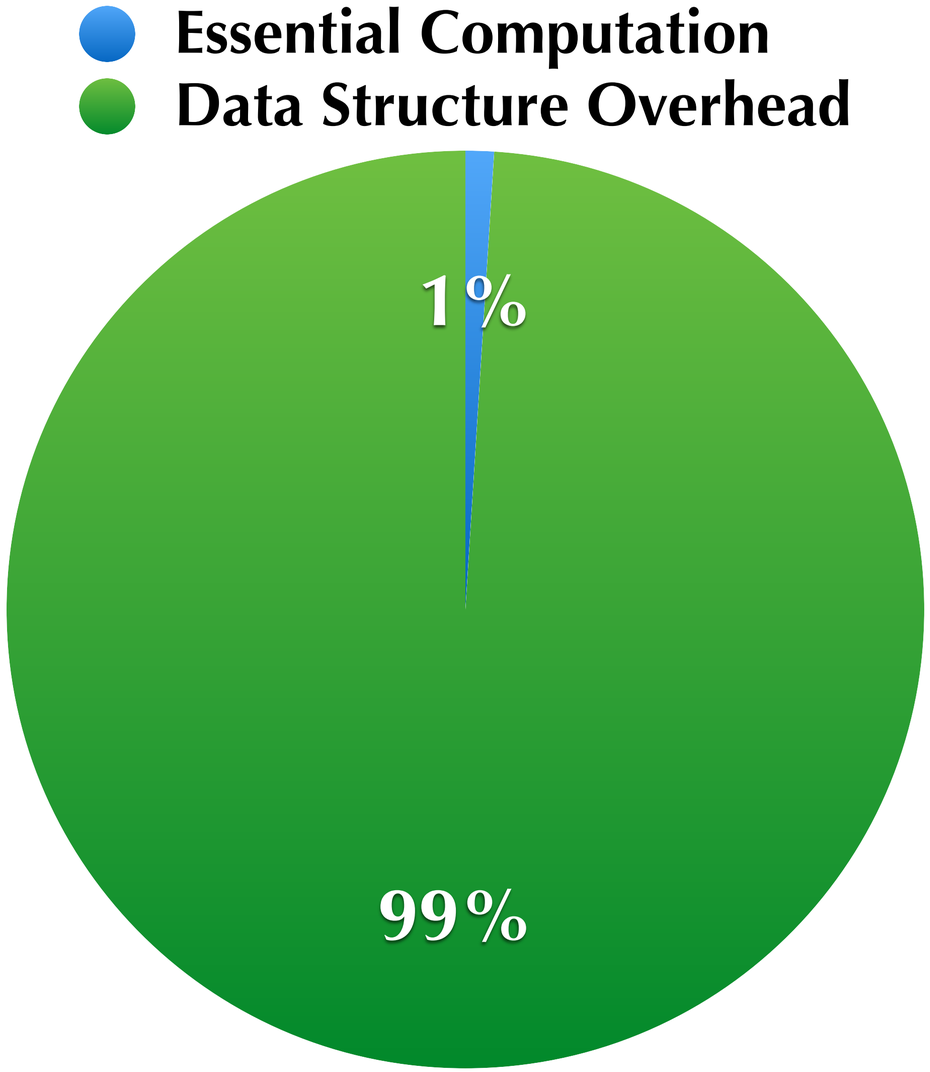
In reality…



In reality…

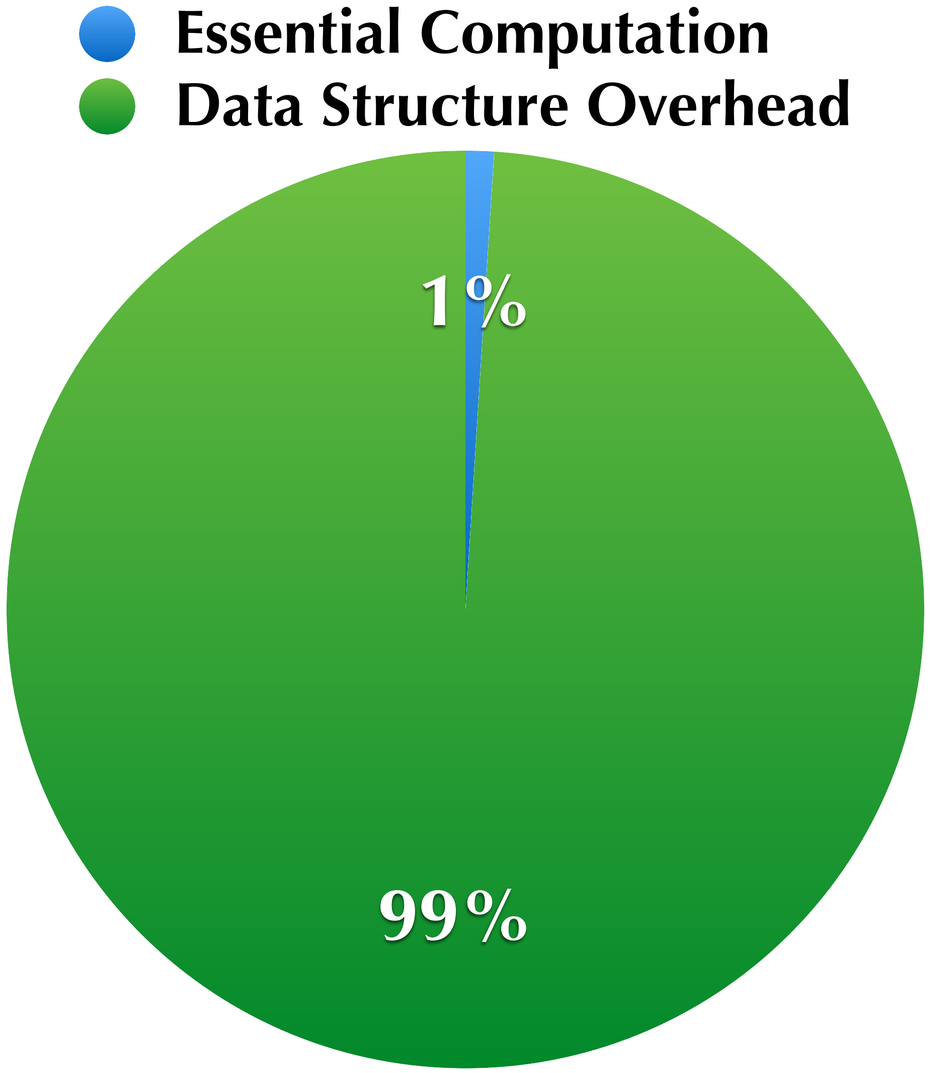


In reality…



Hash table lookup: 10s of clock cycles
Indirection: cache/TLB misses
Node allocation: locks, atomics, barriers
Branching: misprediction / warp divergence
…

In reality…



In reality…

Hash table lookup: 10s of clock cycles
Indirection: cache/TLB misses
Node allocation: locks, atomics, barriers
Branching: misprediction / warp divergence
…

Low-level engineering reduces data structure overhead, but harms productivity and couples algorithms and data structures, making it difficult to explore different data structure designs and find the optimal one.

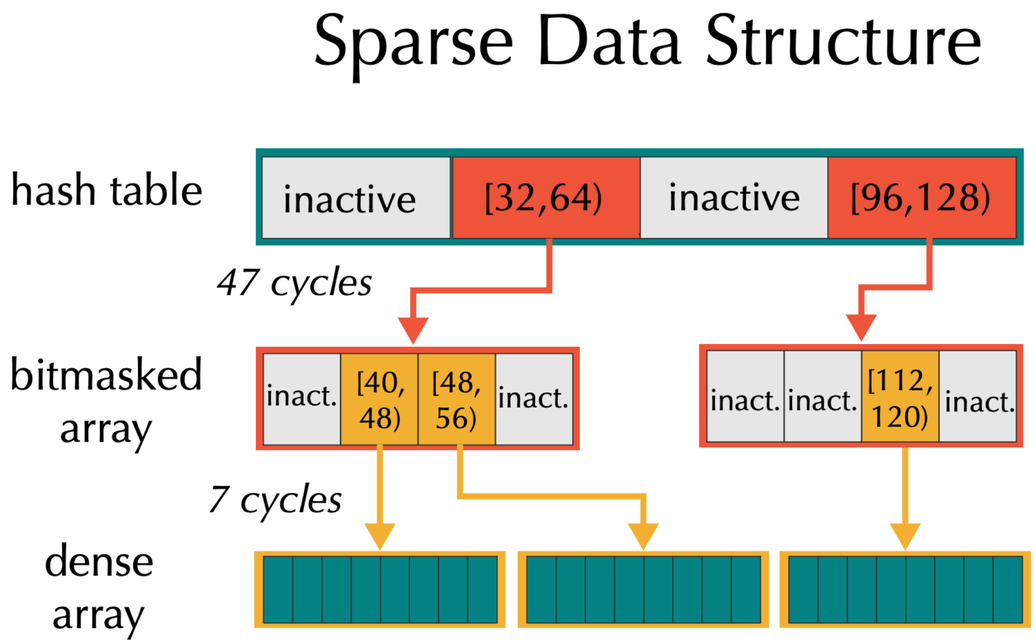
Sparse data structure overhead can be **100x** higher than essential computation

Data structure access:

- 50 clock cycles / element

Simple Stencil Computation:

- 0.5 clock cycle / element



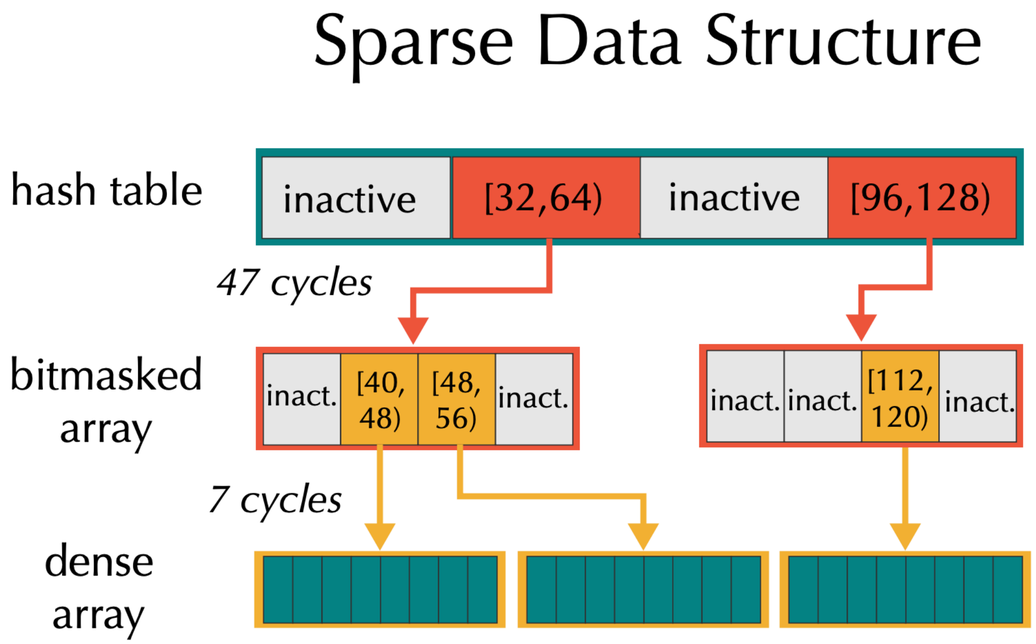
Sparse data structure overhead can be **100x** higher than essential computation

Data structure access:

- 50 clock cycles / element

Simple Stencil Computation:

- 0.5 clock cycle / element



Fun fact: without low-level engineering, dense data structures are often faster for problems with >10% sparsity

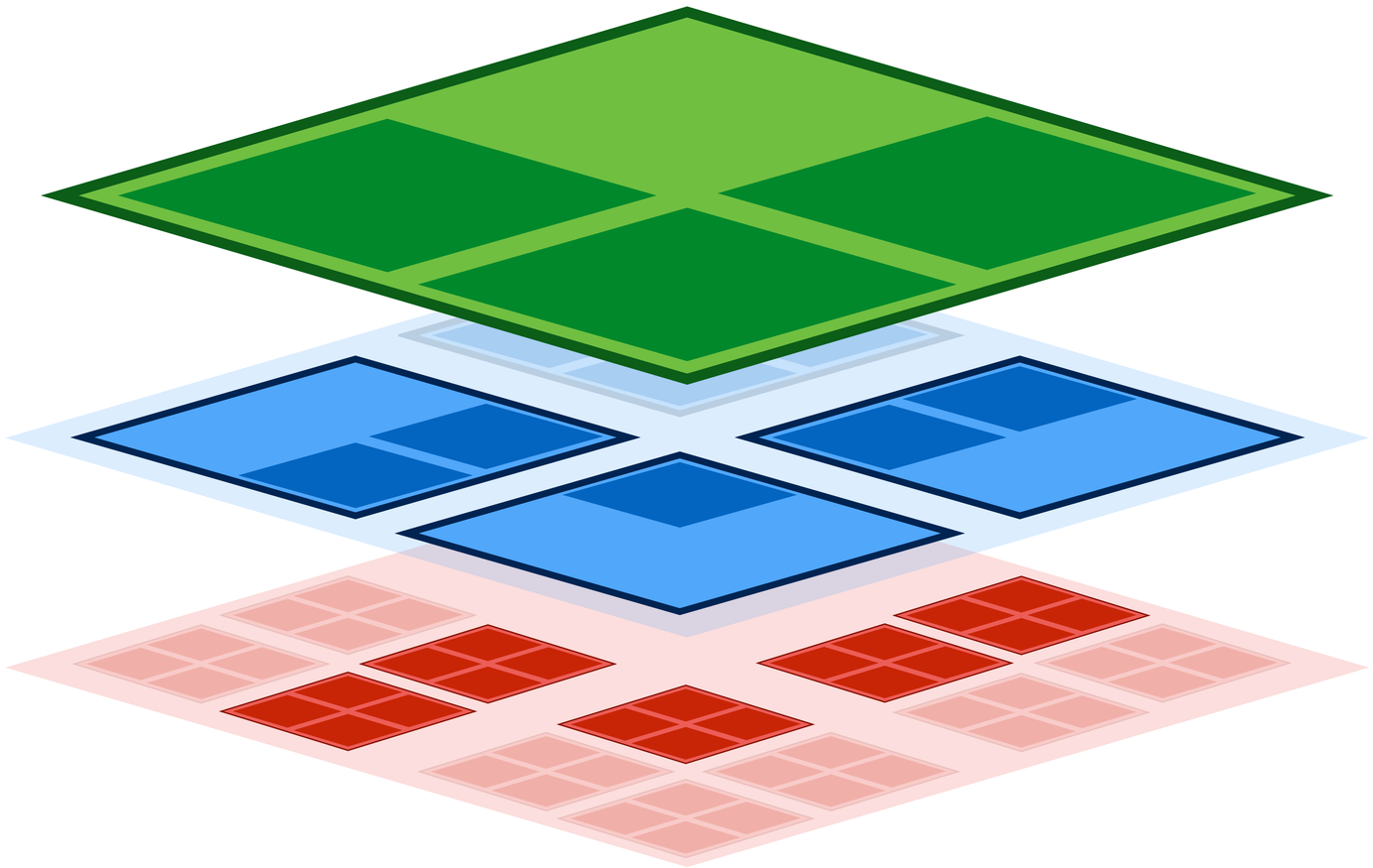
Spatially sparse data structures in computer graphics

- **Idea:** save memory and computation on inactive spaces
- **Reality:** really hard to achieve desired performance and productivity
- **Taichi's solutions:**
  - Allow programmers to flexibly define sparse data structures using SNodes
  - Let programmers access sparse data structures as if they are dense
  - Sparse struct-fors
  - Automatically manages memory
  - Taichi's compiler automatically optimizes sparse data structure access

The `pointer` SNode

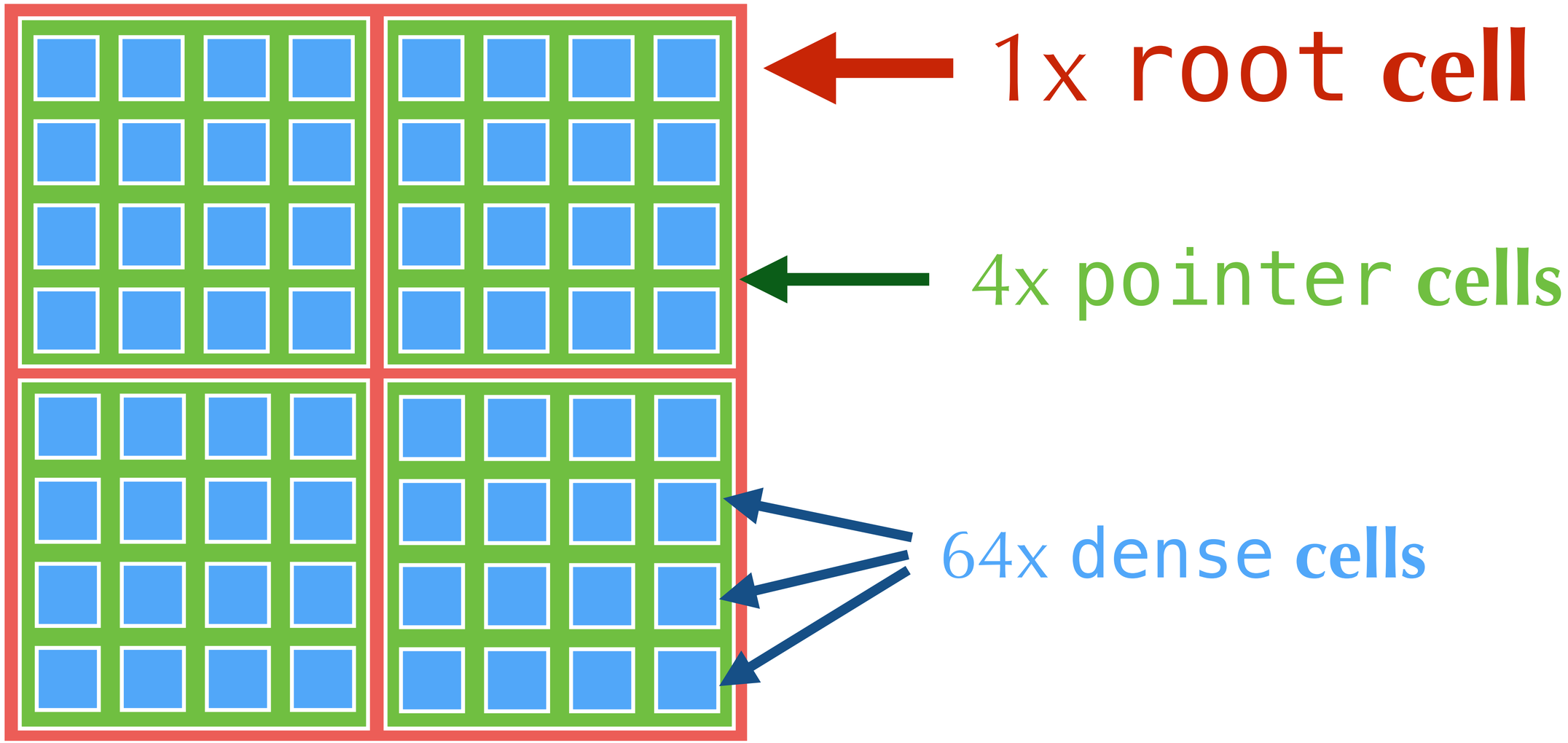
- **Essentially an array with each element being a pointer**
- **Different from the `dense` SNode since pointers can be *null***
- **The most frequently used SNode to achieve sparsity**

The `pointer` SNode



```
ti.root
.pointer(ti.ij, 2)
.pointer(ti.ij, 2)
.dense(ti.ij 2)
```

```
block = ti.root.pointer(ti.ij, 2).dense(ti.ij, 4)
```

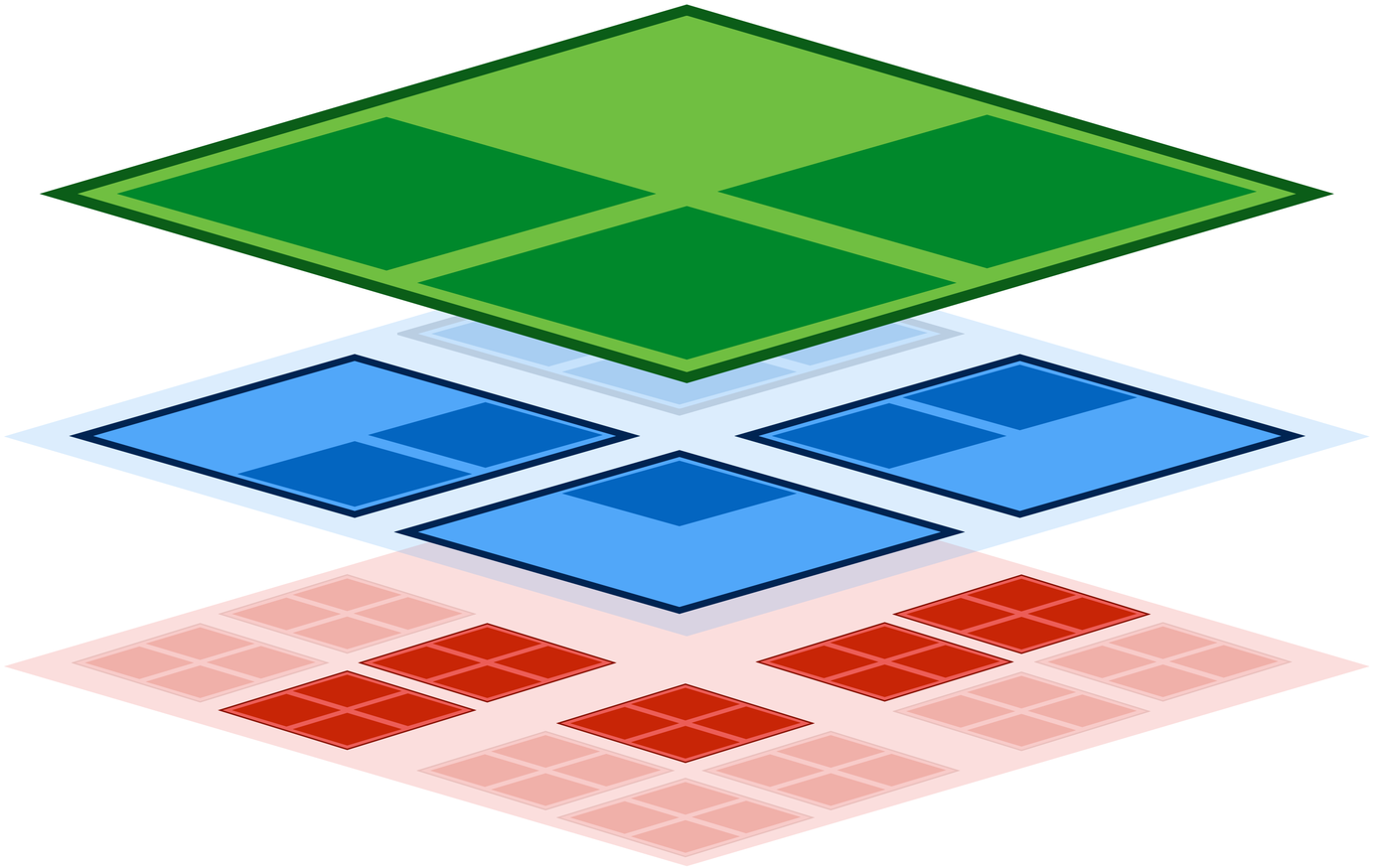


Read and write accesses to sparse fields

- **Reading an active cell**
  - Simply return the value
- **Writing to an active cell**
  - Simply write the value
- **Reading an inactive cell**
  - Return 0
- **Writing to an inactive cell**
  - Active the cell and write the value

Sparse for-loops

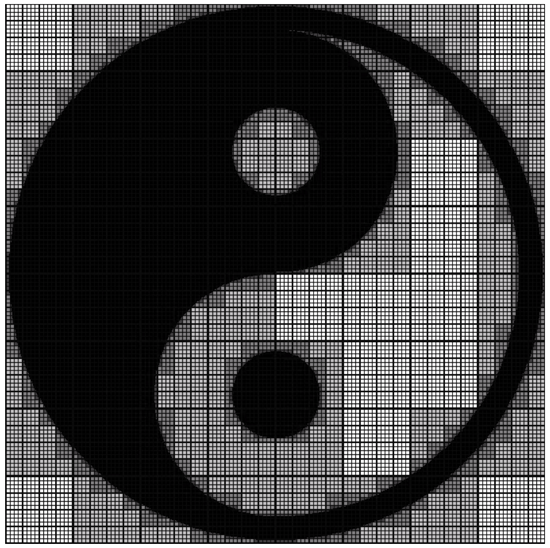
- **Avoid wasting computation on empty nodes**



```
for i, j in a:
    a[i, j] += 1
```

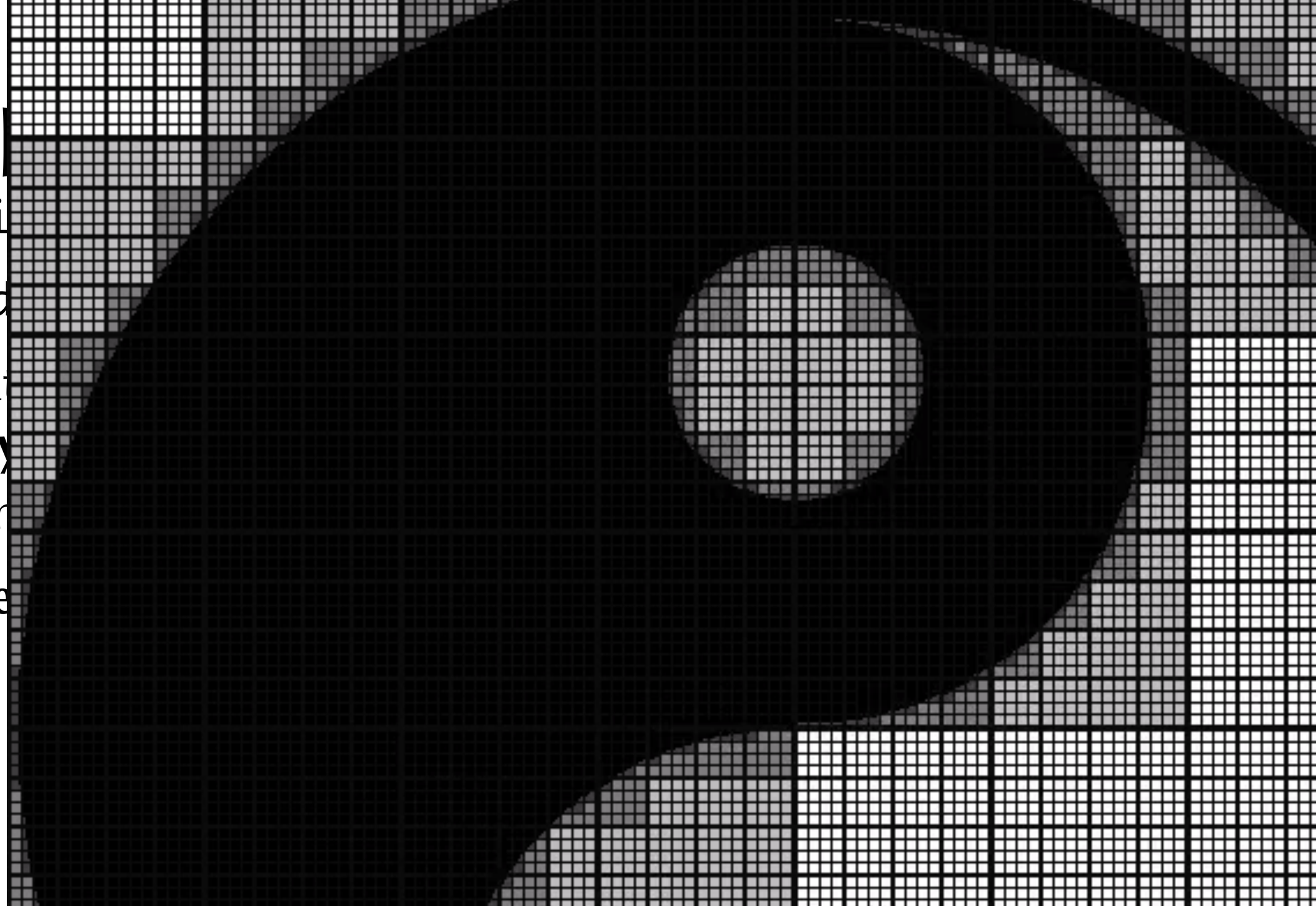
Manipulating activation

- `ti.activate/deactivate(snode, indices)`
- `snode.deactivate_all()`
- **Automatic (parallel) garbage collection happens after a kernel that may deactivate anything**
  - When a cell is reused later, its initial values are **zero.**
- **Example: ti example taichi_sparse**



Mani

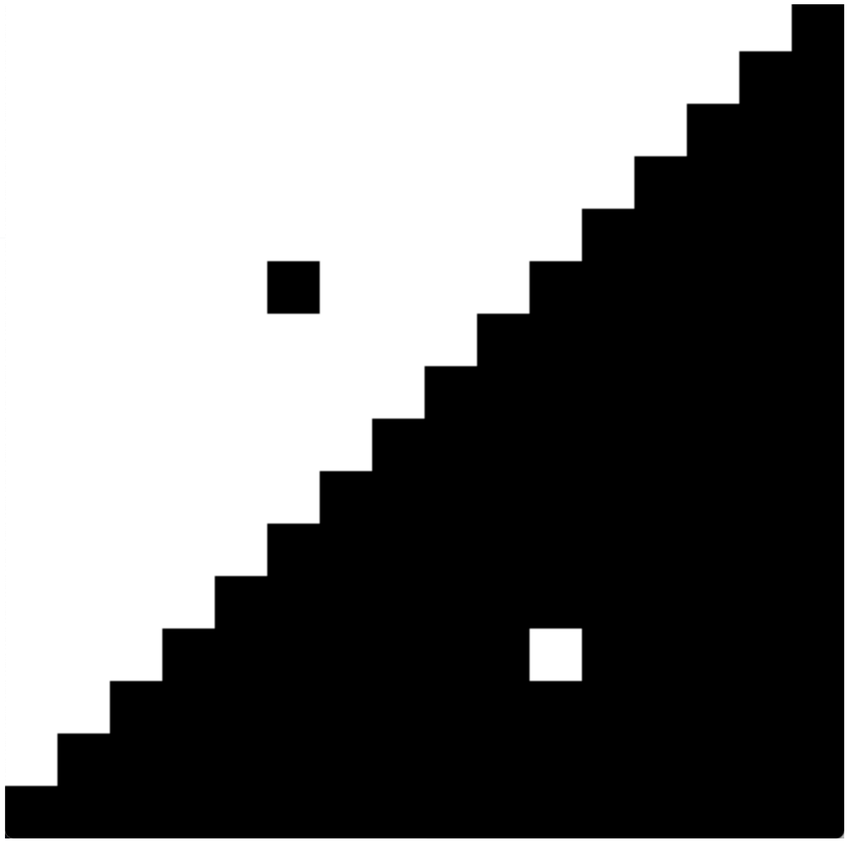
- `ti.acti`
- `snode.d`
- **Automa**
  that may
  - Wher
- **Example**



Another example

```python
import taichi as ti

ti.init()

n = 16
x = ti.var(ti.f32)
gui_res = 512
img = ti.field(ti.f32, shape=(gui_res, gui_res))

cell = ti.root.pointer(ti.ij, n)
cell.place(x)

scale = gui_res // n

@ti.kernel
def activate():
    for i, j in ti.ndrange(n, n):
        if i < j:
            x[i, j] = 1

    ti.activate(cell, [10, 3])


@ti.kernel
def deact():
    ti.deactivate(cell, [5, 10])
```

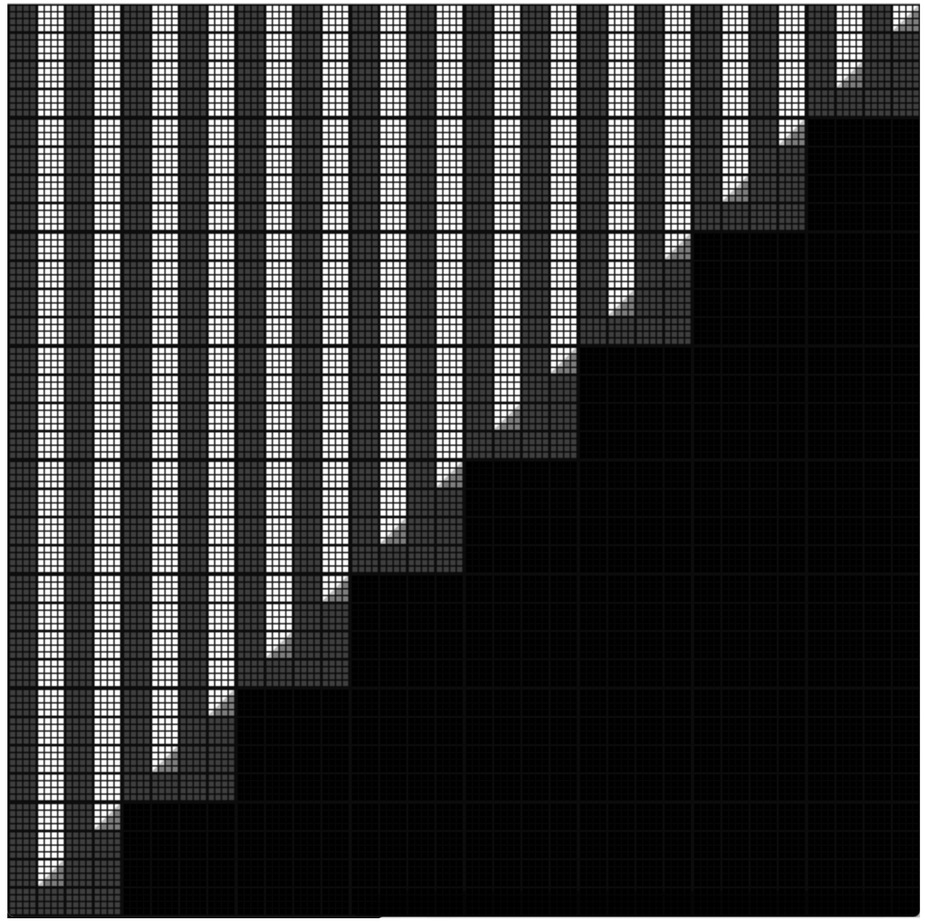
```python
@ti.kernel
def paint():
    for i, j in img:
        img[i, j] = ti.is_active(cell, [i // scale, j // scale])

gui = ti.GUI('pointer', (gui_res, gui_res))

activate()
deact()
paint()

while True:
    gui.set_image(img.to_numpy())
    gui.show()
```



Another example

```python
import taichi as ti

ti.init(arch=ti.cpu, print_ir=True)

n = 512
x = ti.var(ti.f32)
res = n + n // 4 + n // 16 + n // 64
img = ti.var(ti.f32, shape=(res, res))

block1 = ti.root.pointer(ti.ij, n // 64)
block2 = block1.pointer(ti.ij, 4)
block3 = block2.pointer(ti.ij, 4)
block3.dense(ti.ij, 4).place(x)

@ti.kernel
def activate():
    for i, j in ti.ndrange(n, n):
        if i < j:
            x[i, j] = 1


@ti.func
def scatter(i):
    return i + i // 4 + i // 16 + i // 64 + 2


@ti.kernel
def paint():
    for i, j in ti.ndrange(n, n):
        t = x[i, j]
        t += ti.is_active(block1, [i, j])
        t += ti.is_active(block2, [i, j])
        t += ti.is_active(block3, [i, j])
        img[scatter(i), scatter(j)] = t / 4
```

```python
@ti.kernel
def deact():
    for i, j in block2:
        print(i, j)
        if i % 32 == 0:
            ti.deactivate(block2, [i, j])
    # ti.deactivate(block2, [100, 100])
    # ti.deactivate(block1, [100, 100])


img.fill(0.05)

gui = ti.GUI('Sparse Grids', (res, res))

for i in range(100000):
    block1.deactivate_all()
    activate()
    deact()
    paint()
    gui.set_image(img.to_numpy())
    gui.show()
```



MLS-MPM with sparse grids

https://github.com/yuanming-hu/taichi/blob/sparseexamples/examples/mpm88_sparse.py

```python
n_particles = 8192 * 3
n_grid = 512
dx = 1 / n_grid
dt = 5e-5

p_rho = 1
p_vol = (dx * 0.5)**2
p_mass = p_vol * p_rho
gravity = 9.8
bound = 3
E = 400

x = ti.Vector.field(2, ti.f32, n_particles)
v = ti.Vector.field(2, ti.f32, n_particles)
C = ti.Matrix.field(2, 2, ti.f32, n_particles)
J = ti.field(ti.f32, n_particles)

block_size = 16
assert n_grid % block_size == 0

block0 = ti.root.pointer(ti.ij, n_grid // block_size)
block1 = block0.dense(ti.ij, block_size)

grid_v = ti.Vector.field(2, ti.f32)
grid_m = ti.field(ti.f32)

block1.place(grid_v, grid_m)
```

P2G

```python
def substep():
    for p in x:
        Xp = x[p] / dx
        base = int(Xp - 0.5)
        fx = Xp - base
        w = [0.5 * (1.5 - fx)**2, 0.75 - (fx - 1)**2, 0.5 * (fx - 0.5)**2]
        stress = -dt * 4 * E * p_vol * (J[p] - 1) / dx**2
        affine = ti.Matrix([[stress, 0], [0, stress]]) + p_mass * C[p]
        for i, j in ti.static(ti.ndrange(3, 3)):
            offset = ti.Vector([i, j])
            dpos = (offset - fx) * dx
            weight = w[i].x * w[j].y
            grid_v[base + offset] += weight * (p_mass * v[p] + affine @ dpos)
            grid_m[base + offset] += weight * p_mass
```

Grid

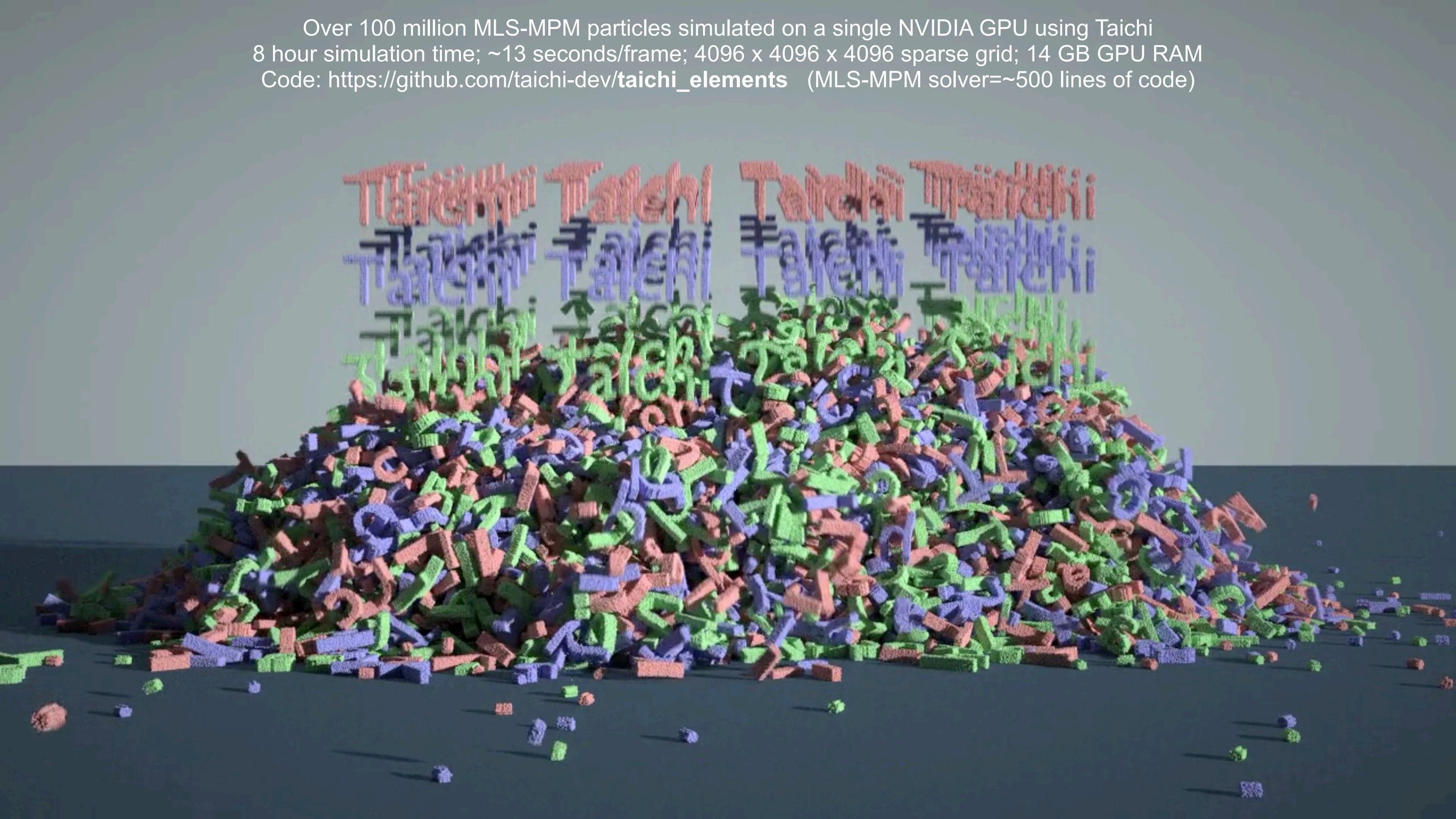
```python
    for i, j in grid_m:
        if grid_m[i, j] > 0:
            grid_v[i, j] /= grid_m[i, j]
        grid_v[i, j][1] -= dt * gravity
        if i < bound and grid_v[i, j].x < 0:
            grid_v[i, j].x = 0
        if i > n_grid - bound and grid_v[i, j].x > 0:
            grid_v[i, j].x = 0
        if j < bound and grid_v[i, j].y < 0:
            grid_v[i, j].y = 0
        if j > n_grid - bound and grid_v[i, j].y > 0:
            grid_v[i, j].y = 0
```

G2P

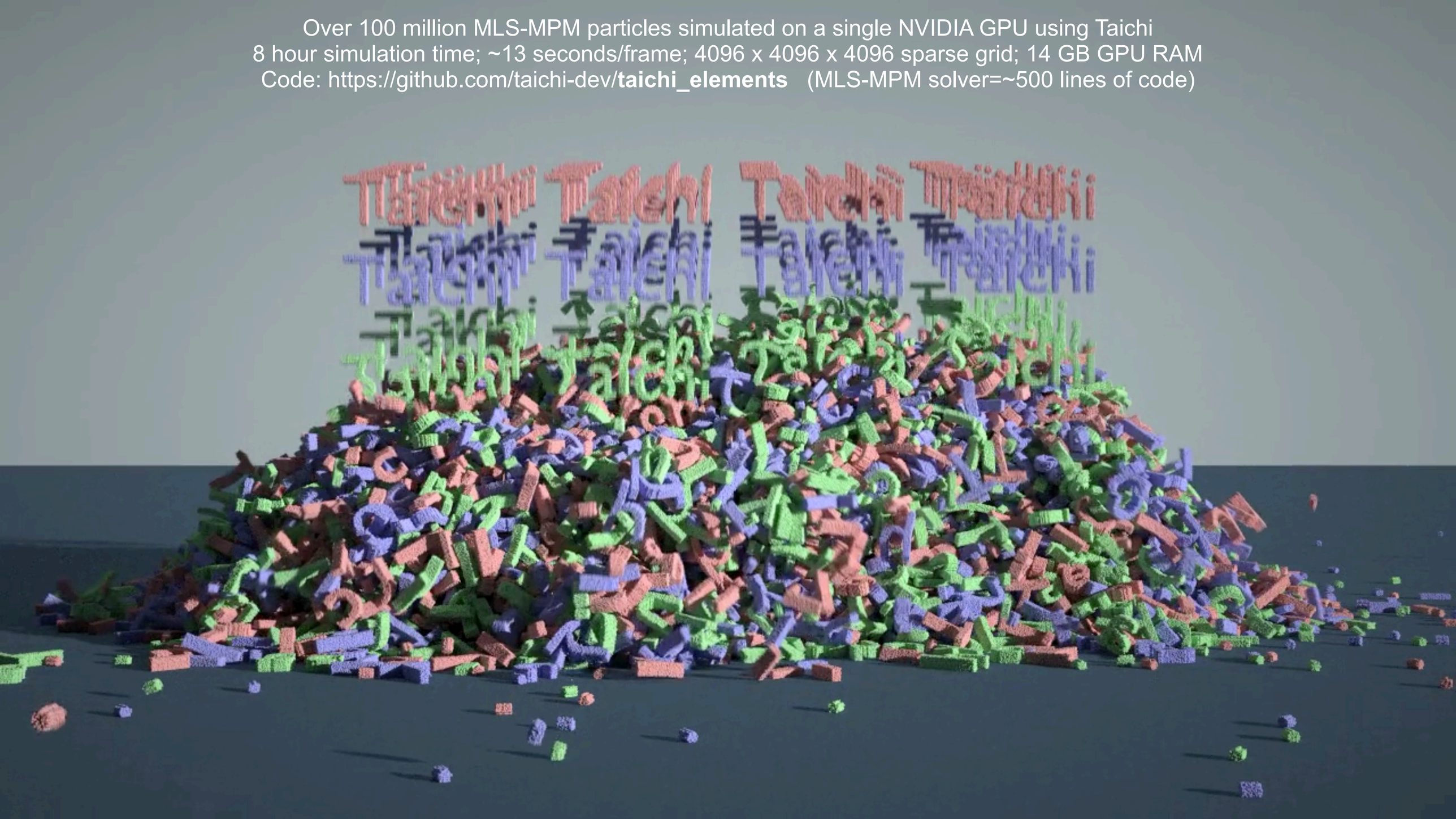
```python
    for p in x:
        Xp = x[p] / dx
        base = int(Xp - 0.5)
        fx = Xp - base
        w = [0.5 * (1.5 - fx)**2, 0.75 -
        new_v = ti.Vector.zero(ti.f32, 2)
        new_C = ti.Matrix.zero(ti.f32, 2,
        for i, j in ti.static(ti.ndrange(3
            offset = ti.Vector([i, j])
            dpos = (offset - fx) * dx
            weight = w[i].x * w[j].y
            g_v = grid_v[base + offset]
            new_v += weight * g_v
            new_C += 4 * weight * g_v.oute
        v[p] = new_v
        x[p] += dt * v[p]
        J[p] *= 1 + dt * new_C.trace()
        C[p] = new_C
```

```python
    block0.deactivate_all()
    substep()
```

Over 100 million MLS-MPM particles simulated on a single NVIDIA GPU using Taichi
8 hour simulation time; ~13 seconds/frame; 4096 x 4096 x 4096 sparse grid; 14 GB GPU RAM
Code: https://github.com/taichi-dev/**taichi_elements** (MLS-MPM solver=~500 lines of code)



Over 100 million MLS-MPM particles simulated on a single NVIDIA GPU using Taichi
8 hour simulation time; ~13 seconds/frame; 4096 x 4096 x 4096 sparse grid; 14 GB GPU RAM
Code: https://github.com/taichi-dev/**taichi_elements** (MLS-MPM solver=~500 lines of code)



Pitfalls

- **Deactivating a pointer node with active children can lead to memory leakage.**
- **Do not activate and deactivate the same SNode in a single kernel**
- **Usually the leaf-level container is `dense` or `bitmasked`**
  - Use pointer as the intermediate SNodes so that its memory consumption is amortized by this children

Recap: Taichi's advanced data layouts

- **Use SNodes to customize data layouts**
  - `ti.field/ti.Vector.field/ti.Matrix.field` **are essentially** `ti.dense(…).place(…)`
  - **Negative coordinates and index space offsets**
- **Achieve sparsity using the** `pointer` **SNode**
- **Other SNodes that are useful: bitmasked and dynamic.**