

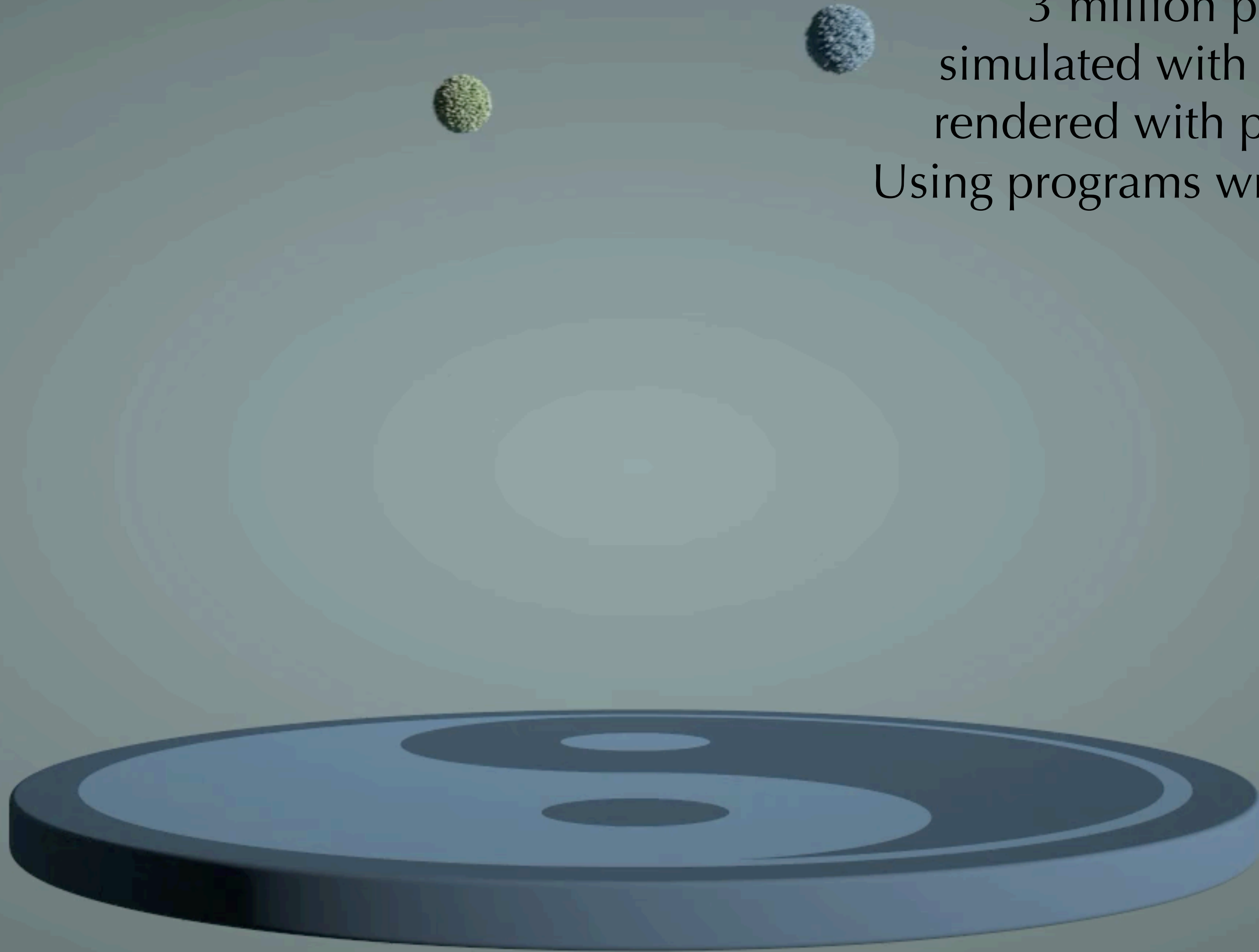


# High-Performance Computation on Spatially Sparse Data Structures

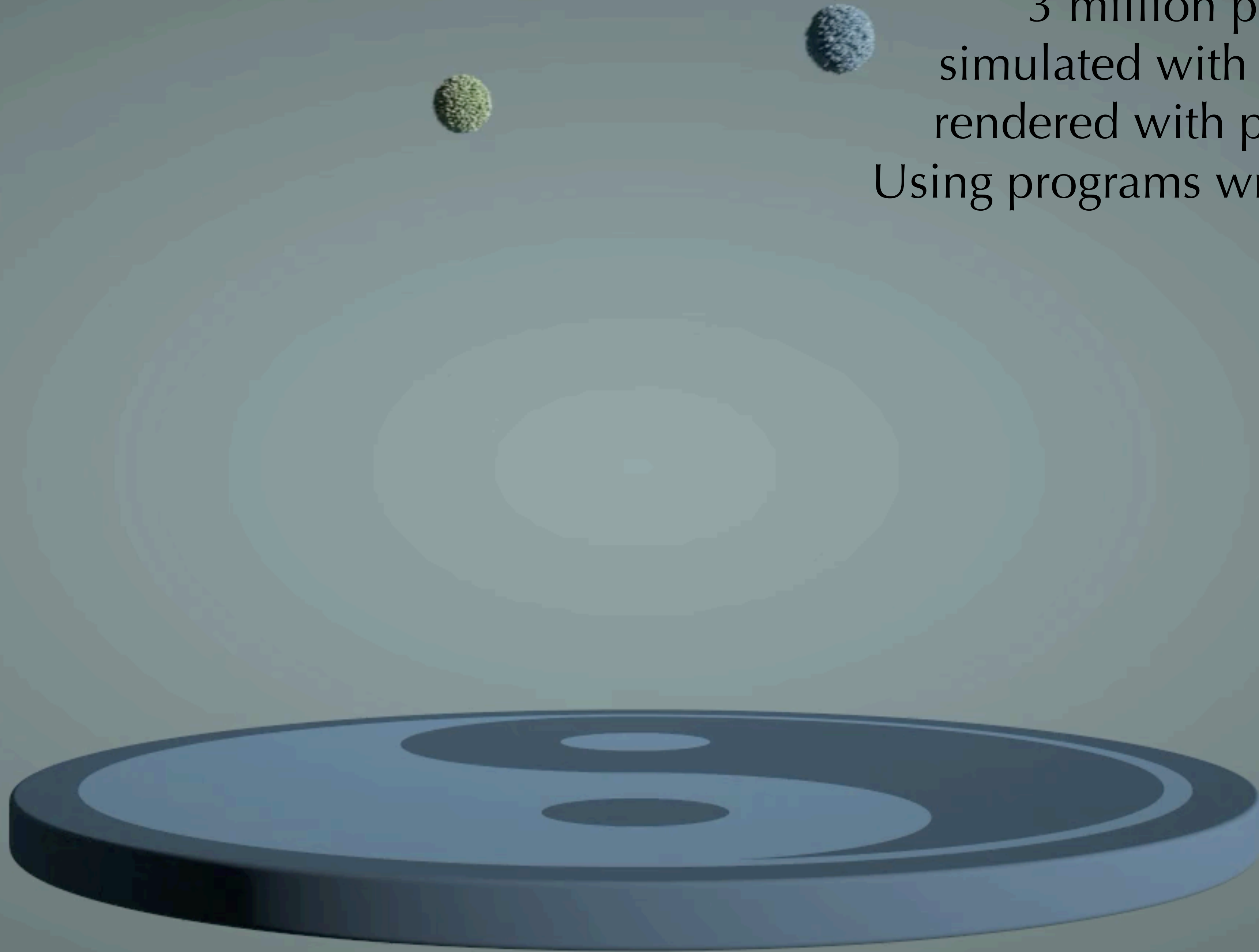
**Yuanming Hu**<sup>1</sup> Tzu-Mao Li<sup>2</sup> Luke Anderson<sup>1</sup> Jonathan Ragan-Kelley<sup>2</sup>  
Fredo Durand<sup>1</sup>

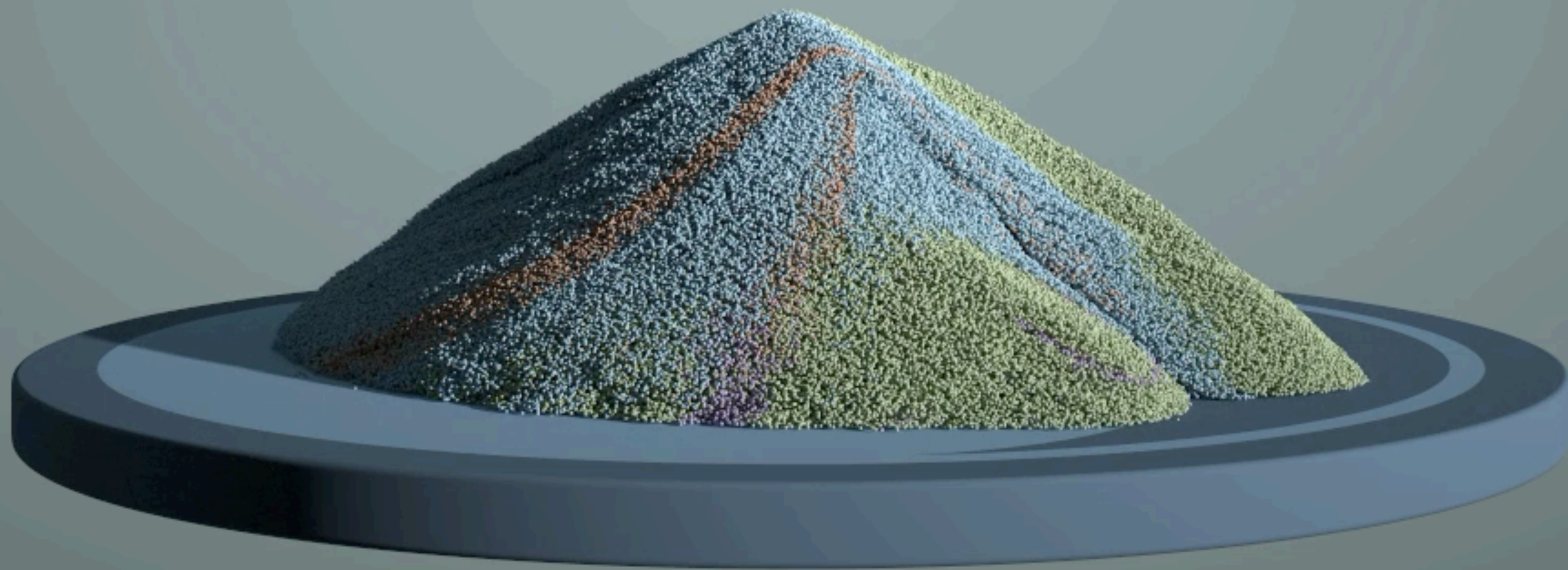
<sup>1</sup>MIT CSAIL <sup>2</sup>UC Berkeley

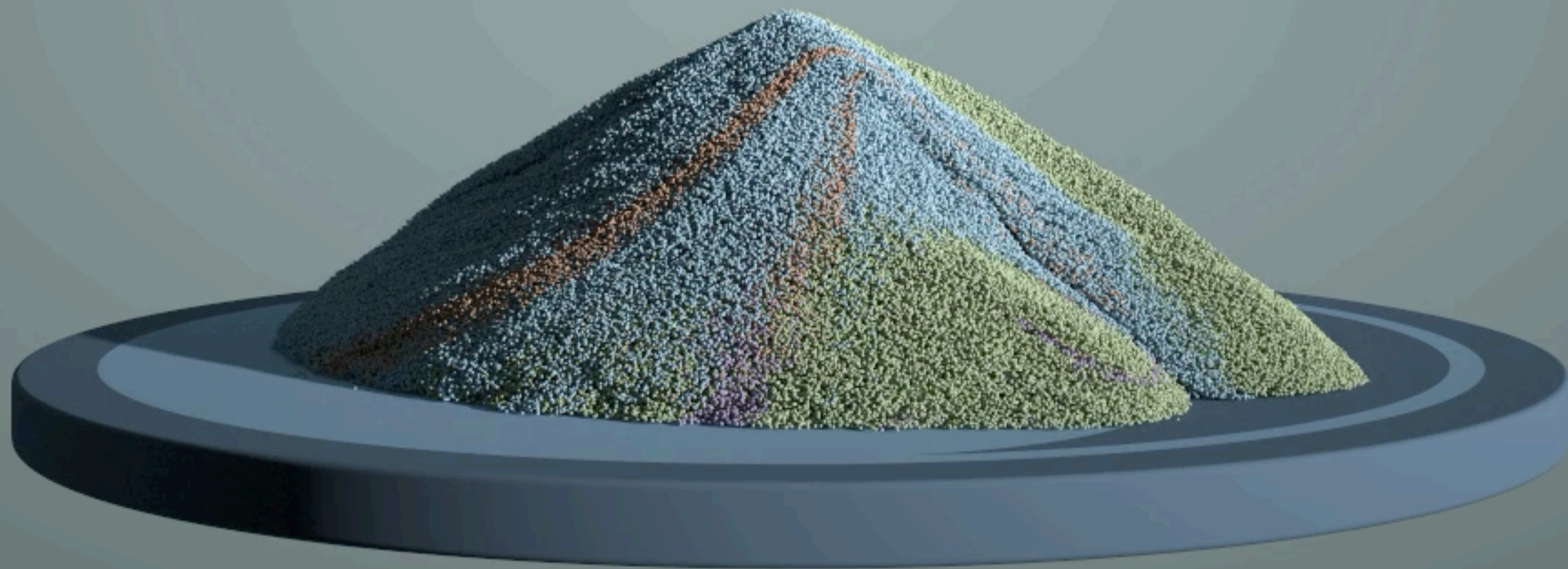
3 million particles  
simulated with MLS-MPM;  
rendered with path tracing.  
Using programs written in *Taichi*.



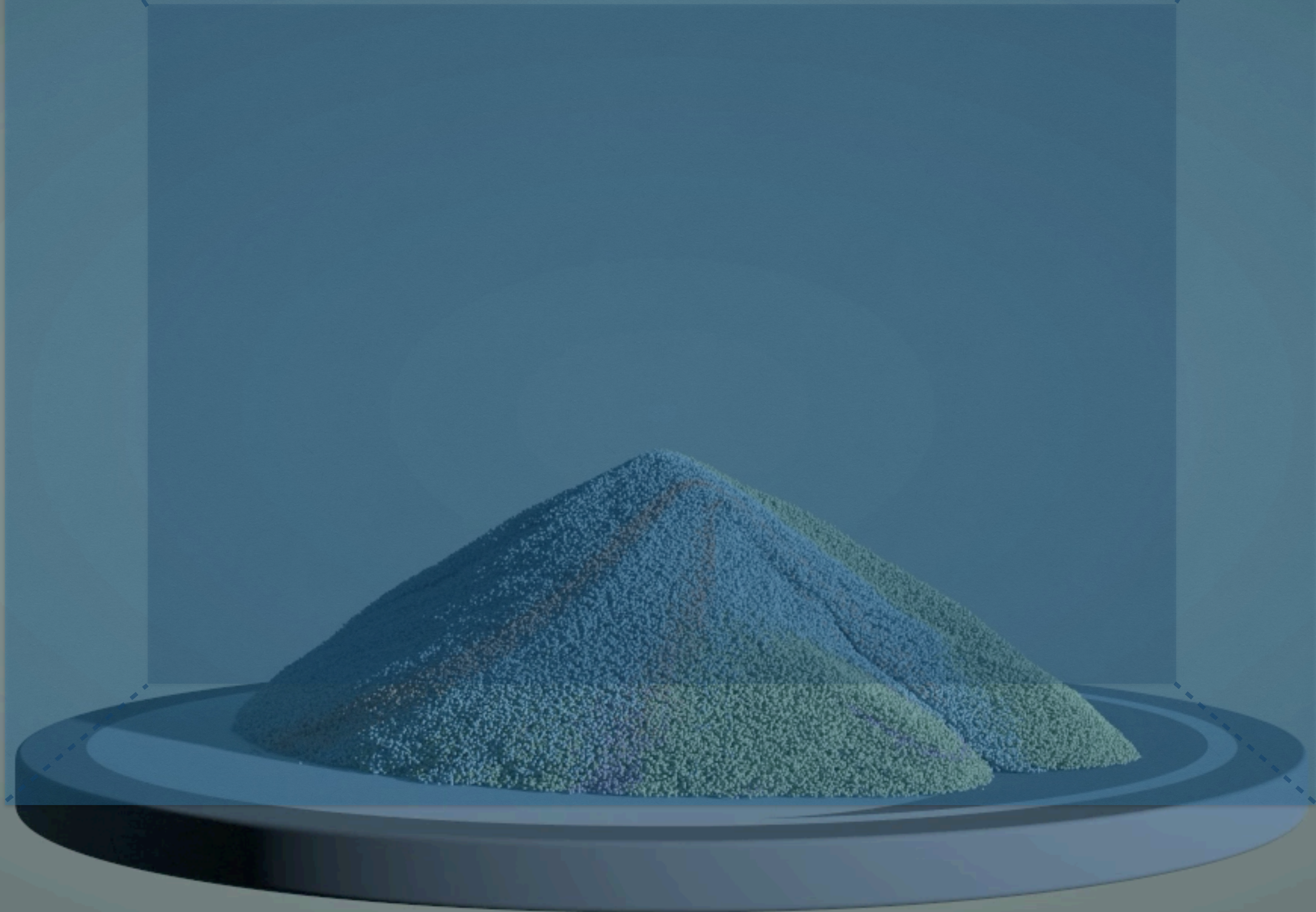
3 million particles  
simulated with MLS-MPM;  
rendered with path tracing.  
Using programs written in *Taichi*.





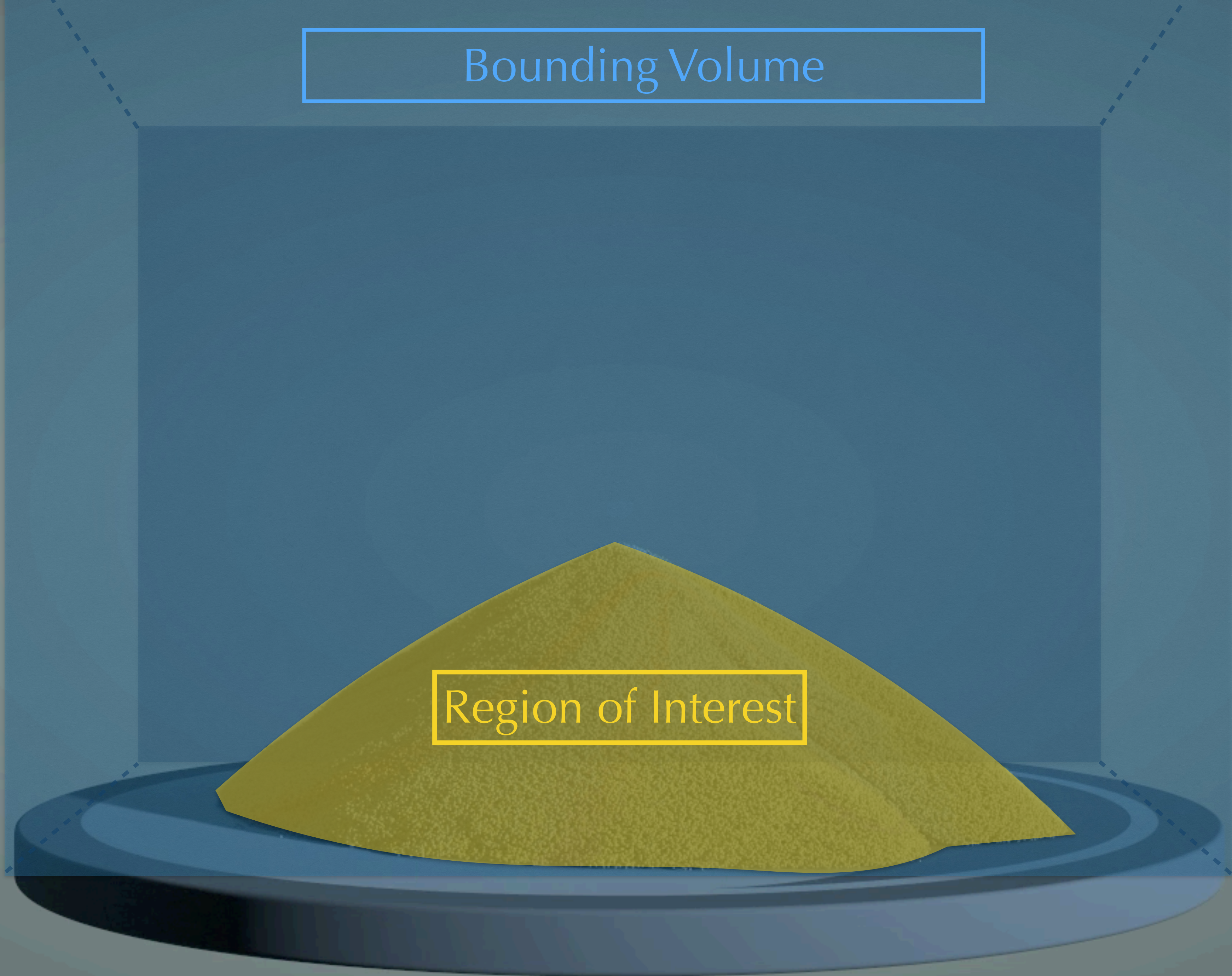


# Bounding Volume



Bounding Volume

Region of Interest



Bounding Volume

## Spatial Sparsity:

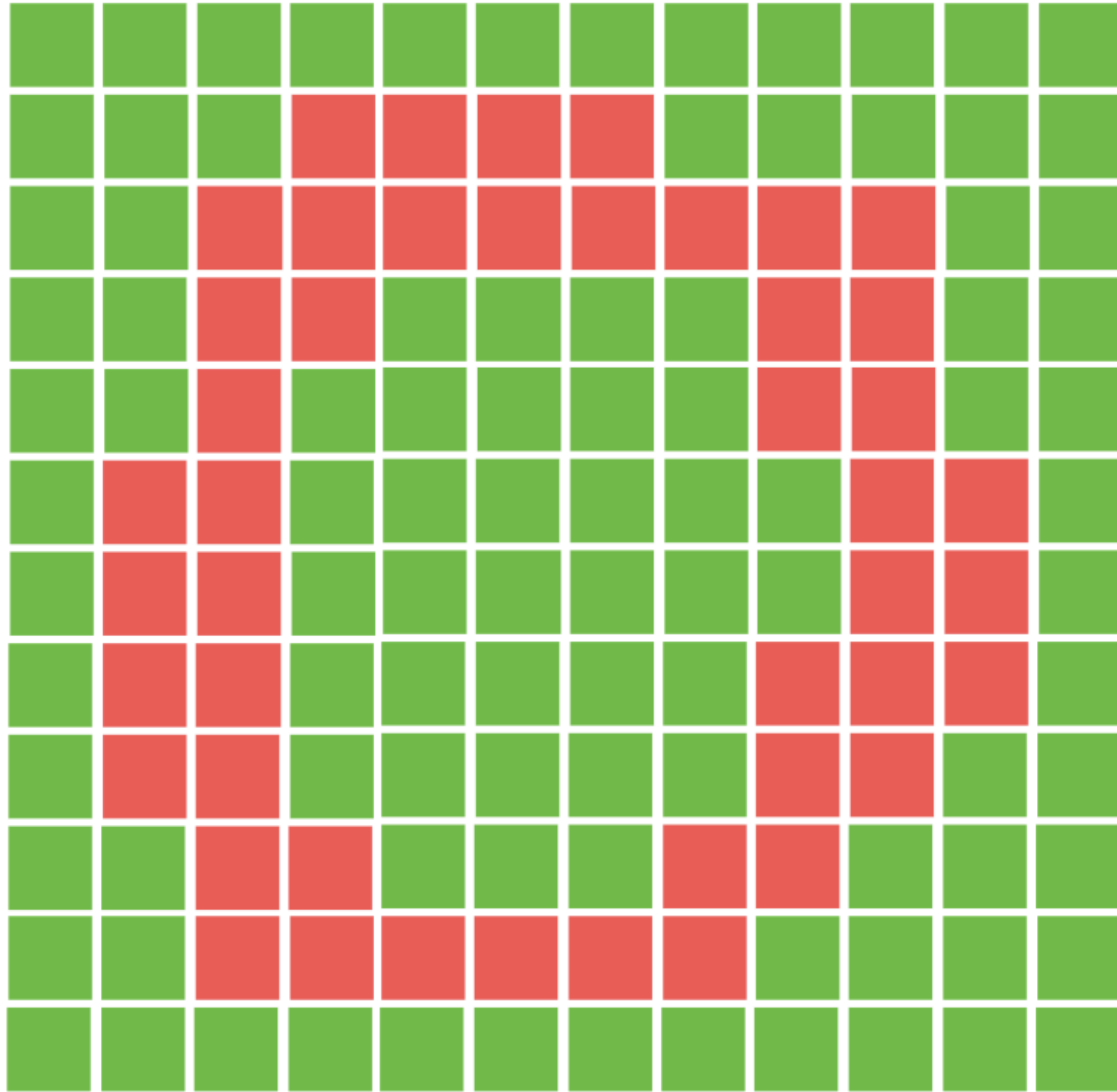
Regions of interest only occupy a small fraction of the bounding volume.

Region of Interest

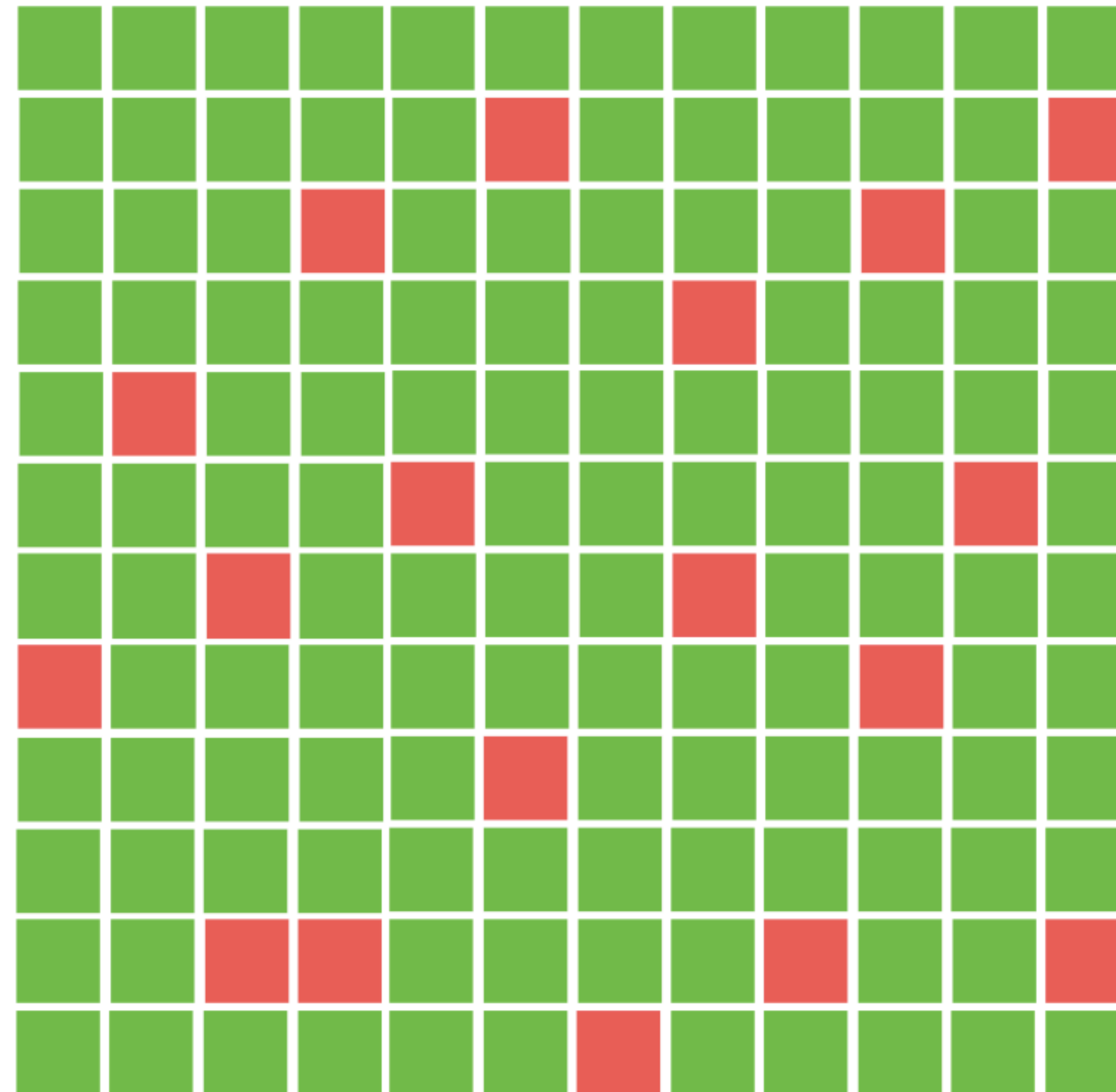


# Spatial Sparsity

globally sparse, locally dense

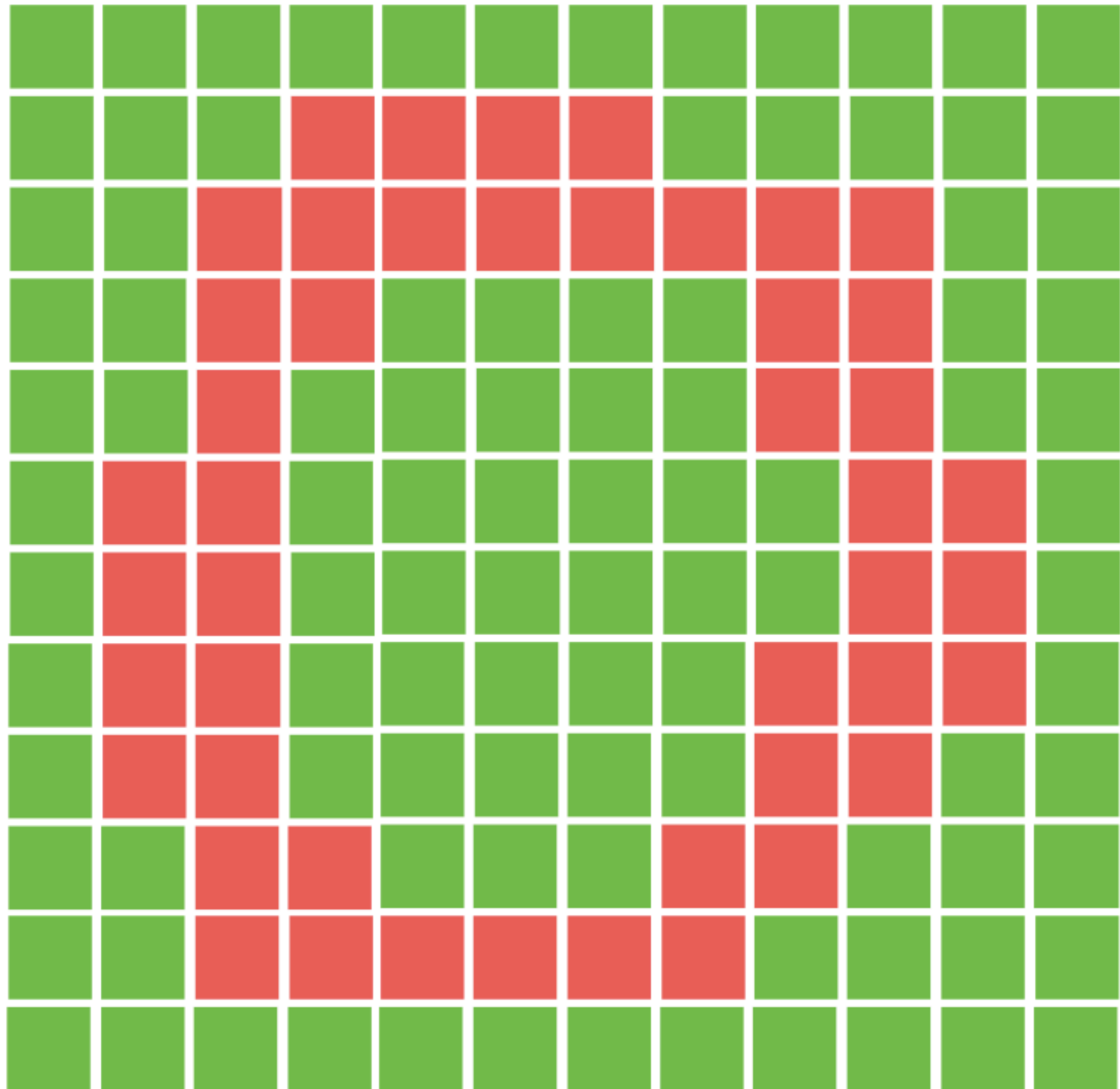


# "General" Sparsity

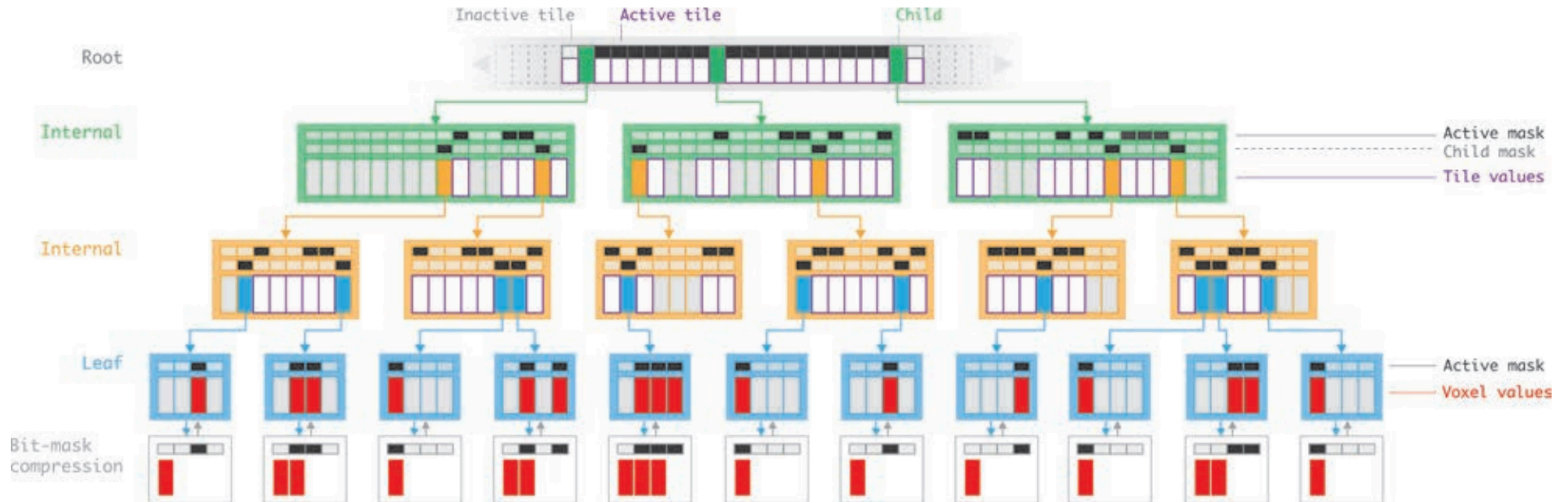


# Spatial Sparsity

globally sparse, locally dense

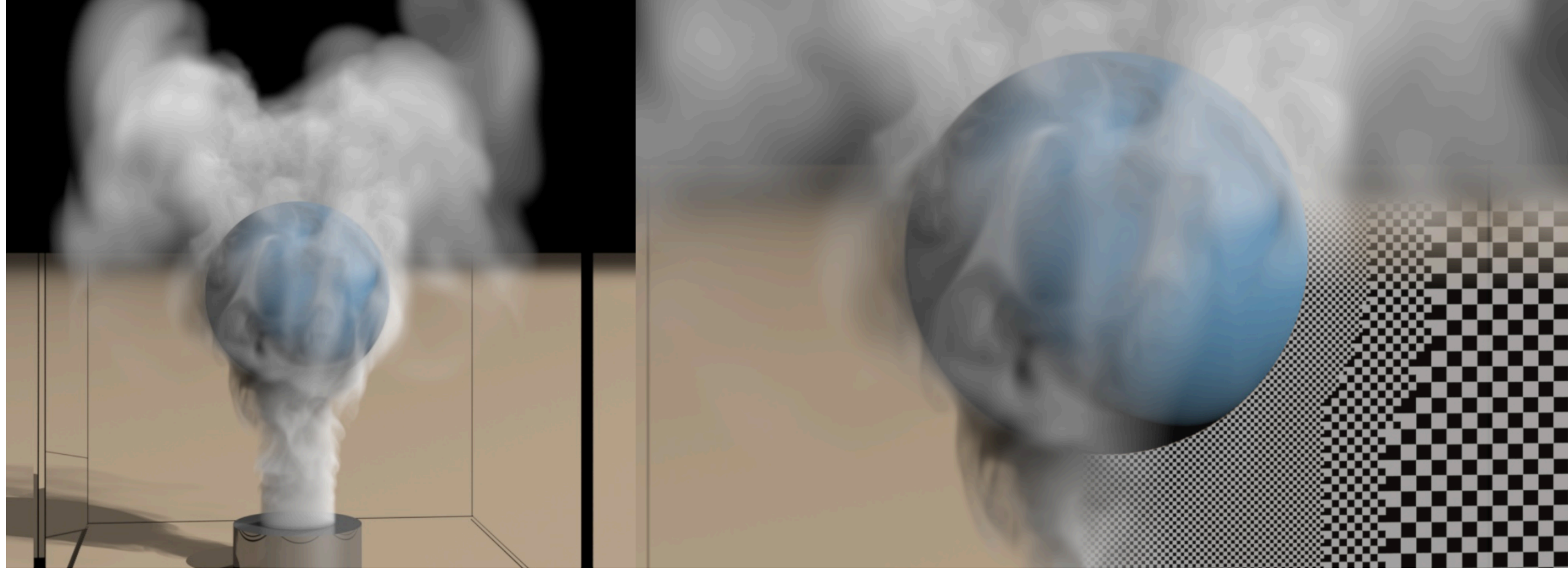


# VDB [Museth 2013]



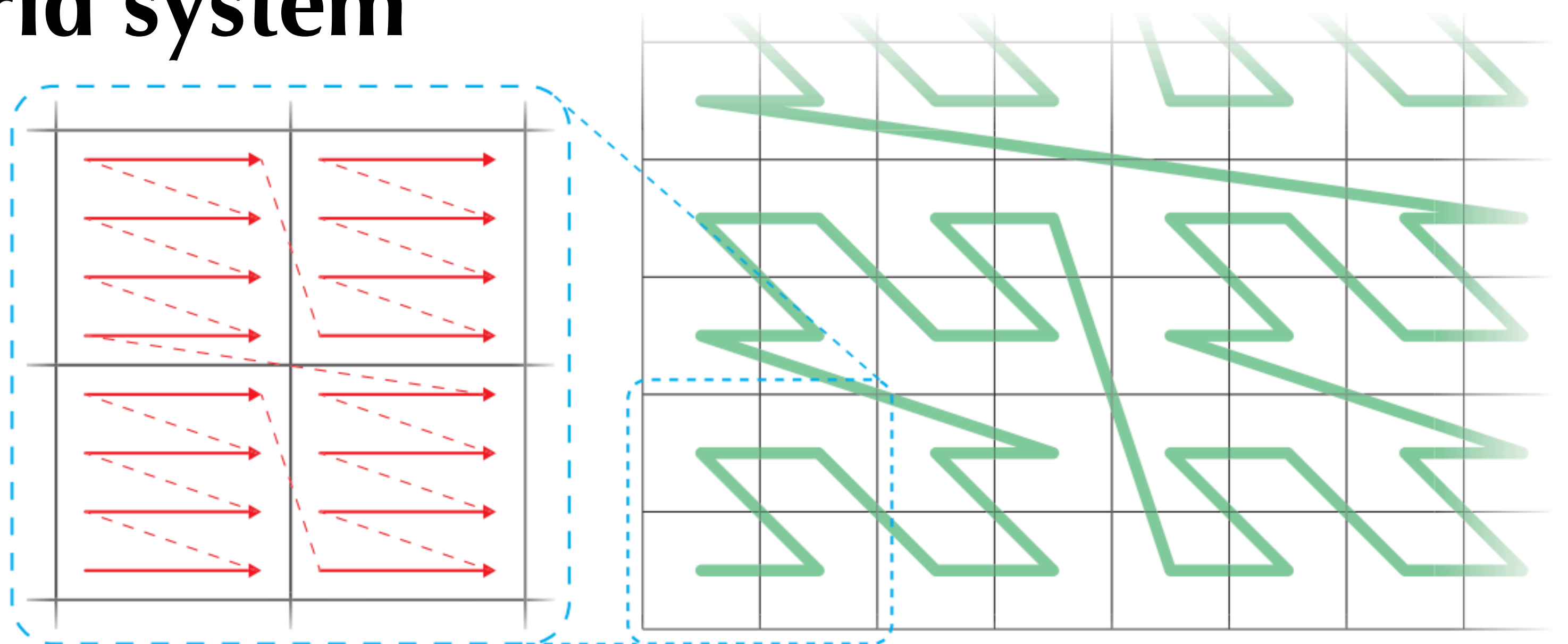
## Shallow Multi-Level Sparse Voxel Grids

# SPGrid



## Even shallower sparse grid system

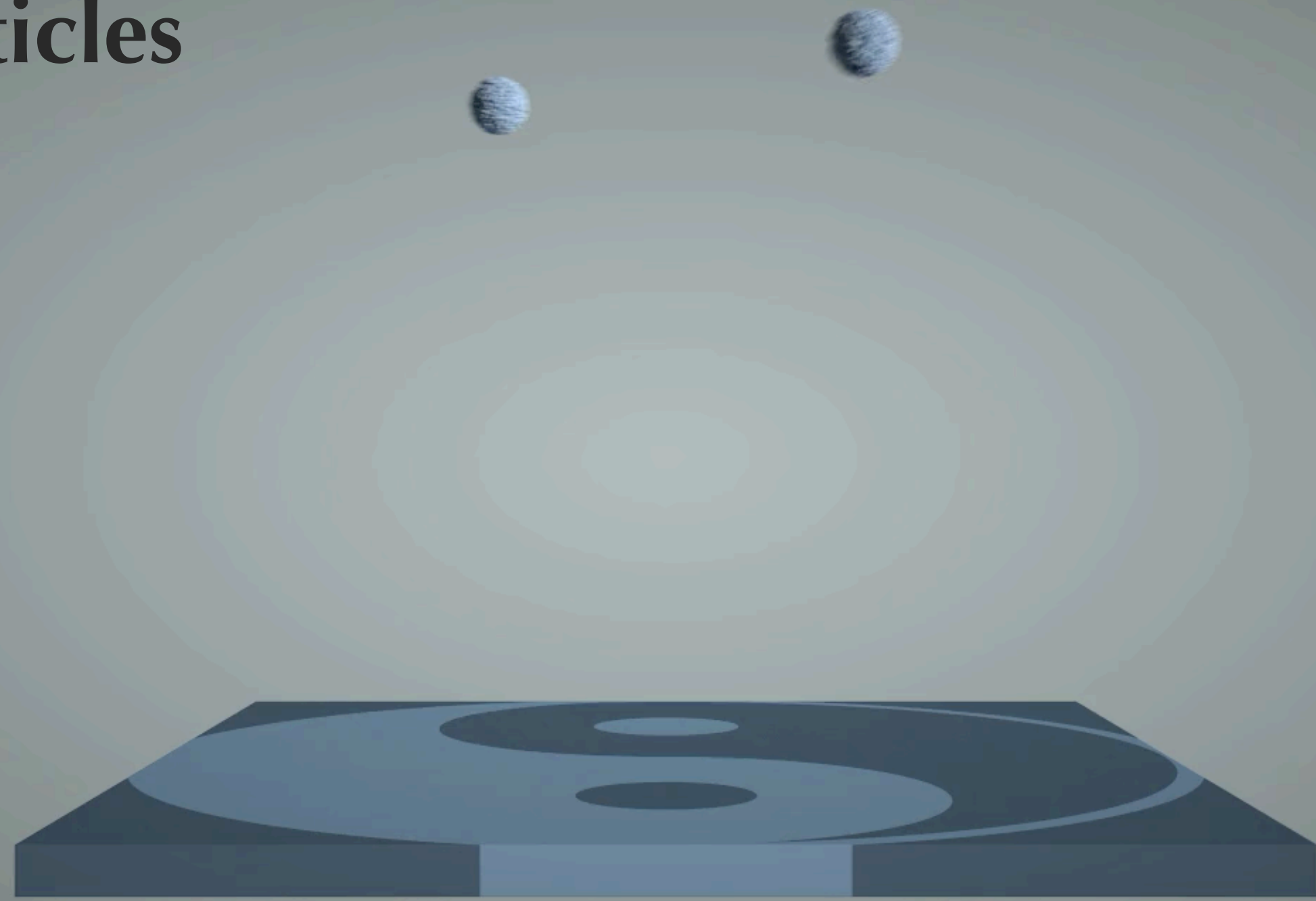
- **Virtual Memory**
- **Morton Coding**
- **Bitmasks**



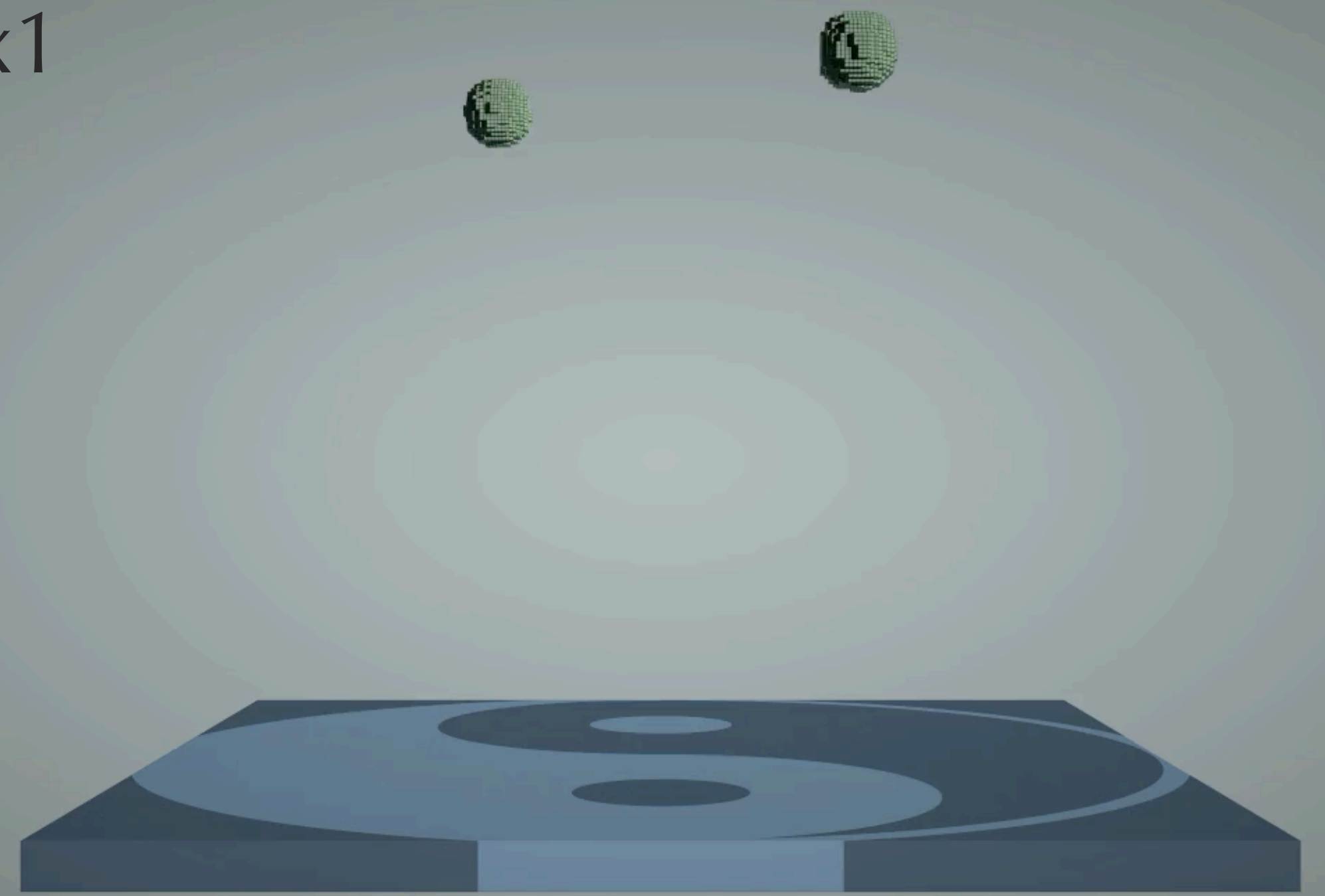
[Setaluri, Aanjaneya, Bauer, and Sifakis, SIGGRAPH Asia 2014]

SPGrid: A sparse paged grid structure applied to adaptive smoke simulation

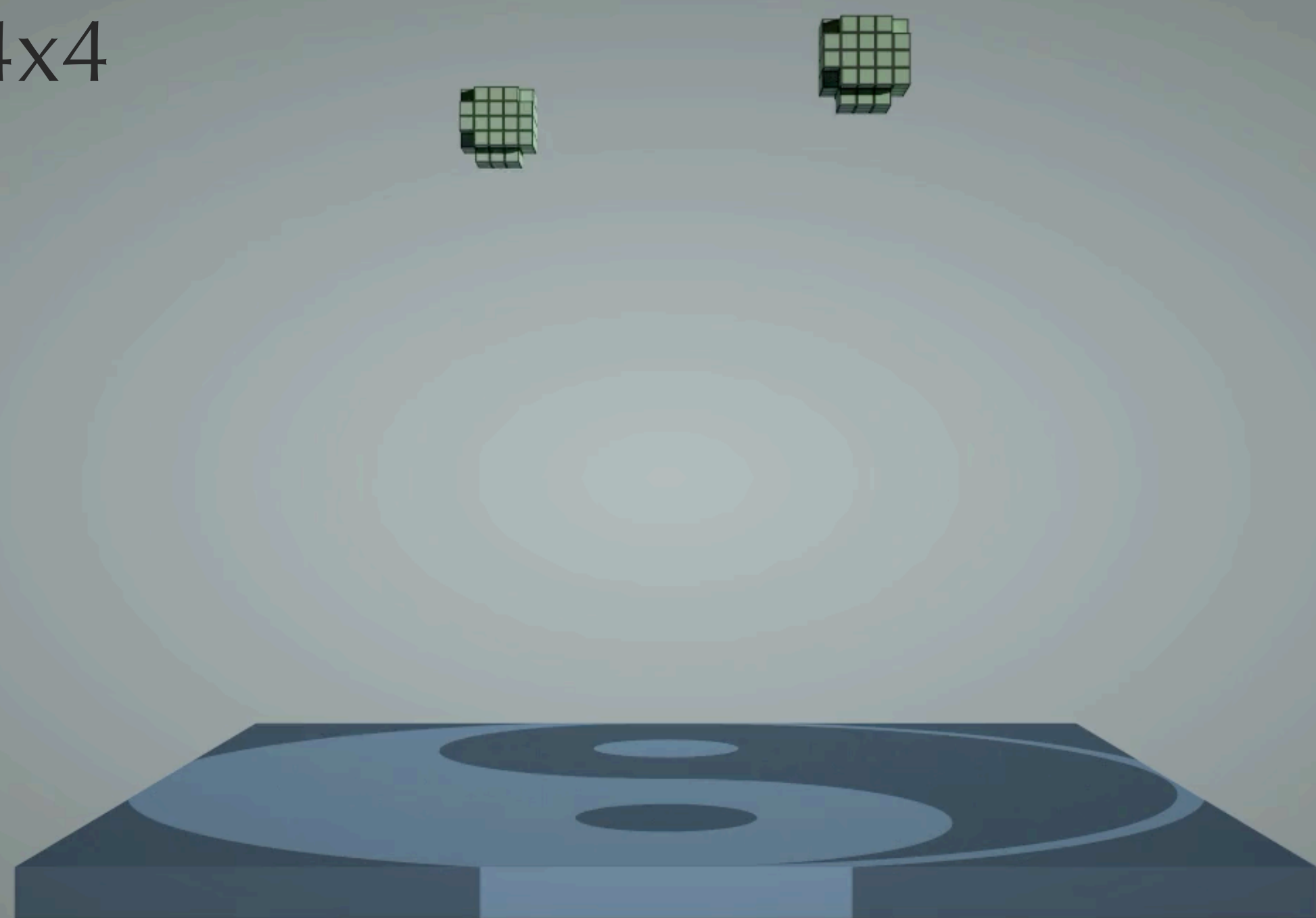
Particles



1x1x1



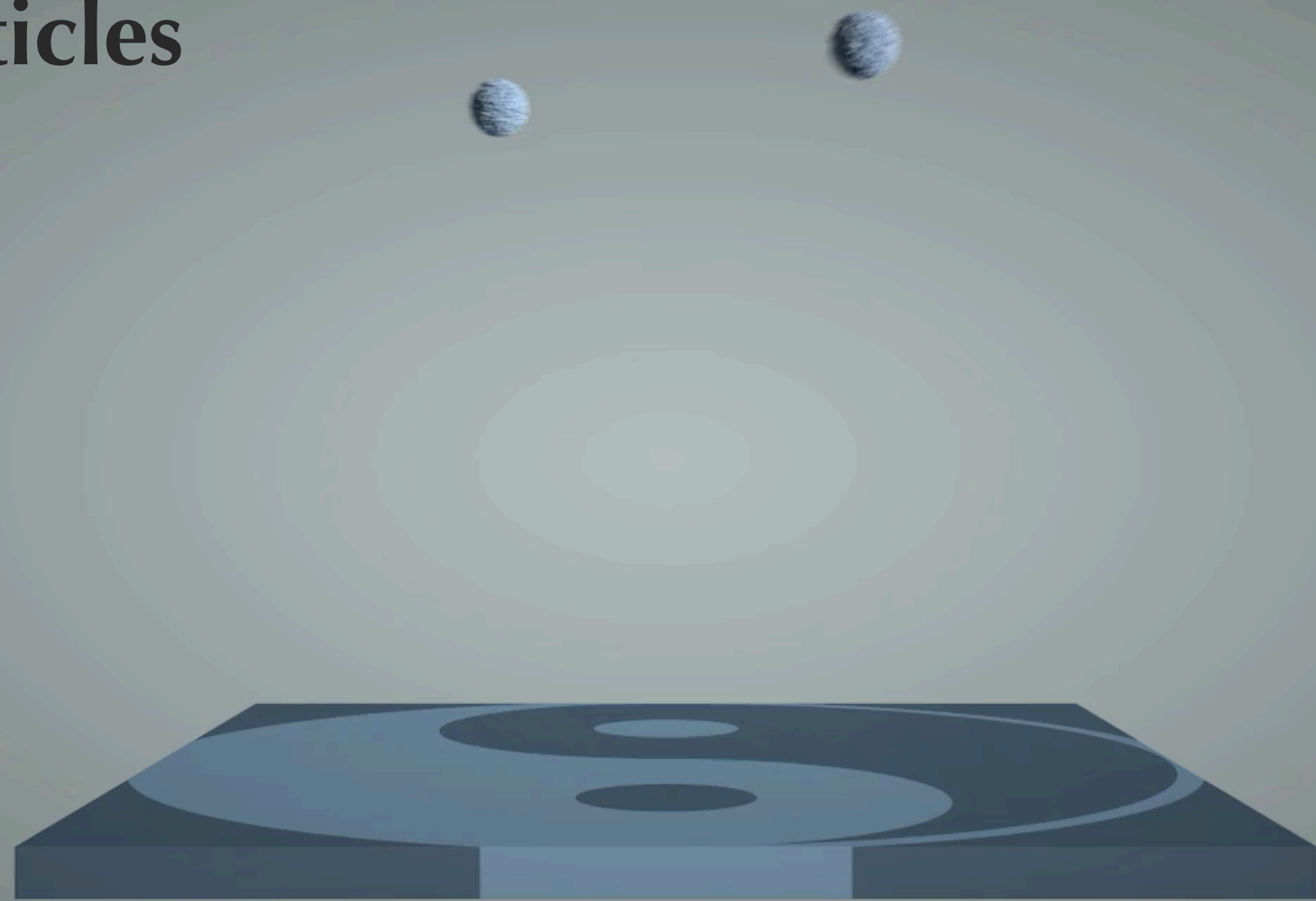
4x4x4



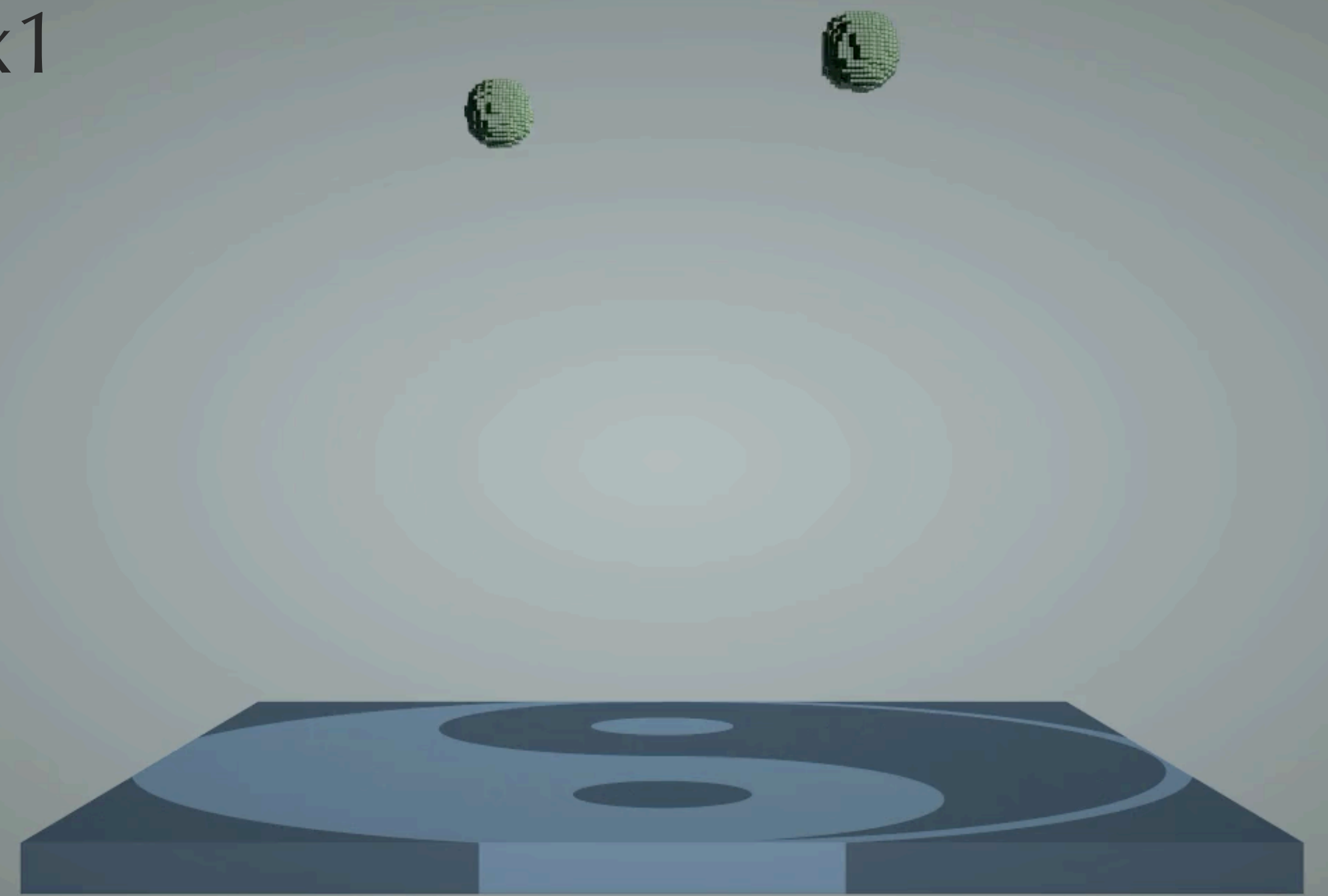
16x16x16



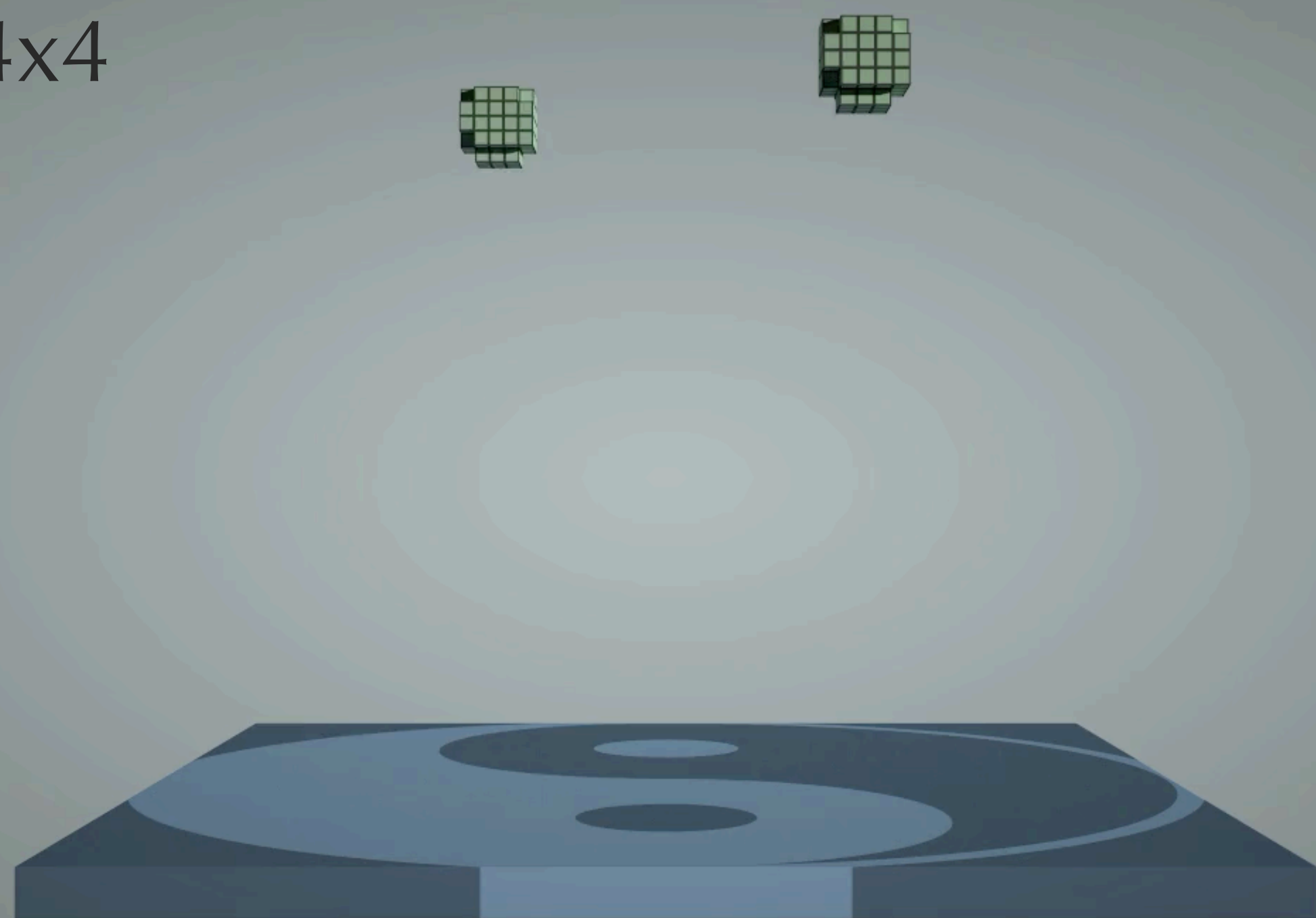
Particles



1x1x1



4x4x4



16x16x16



# 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**

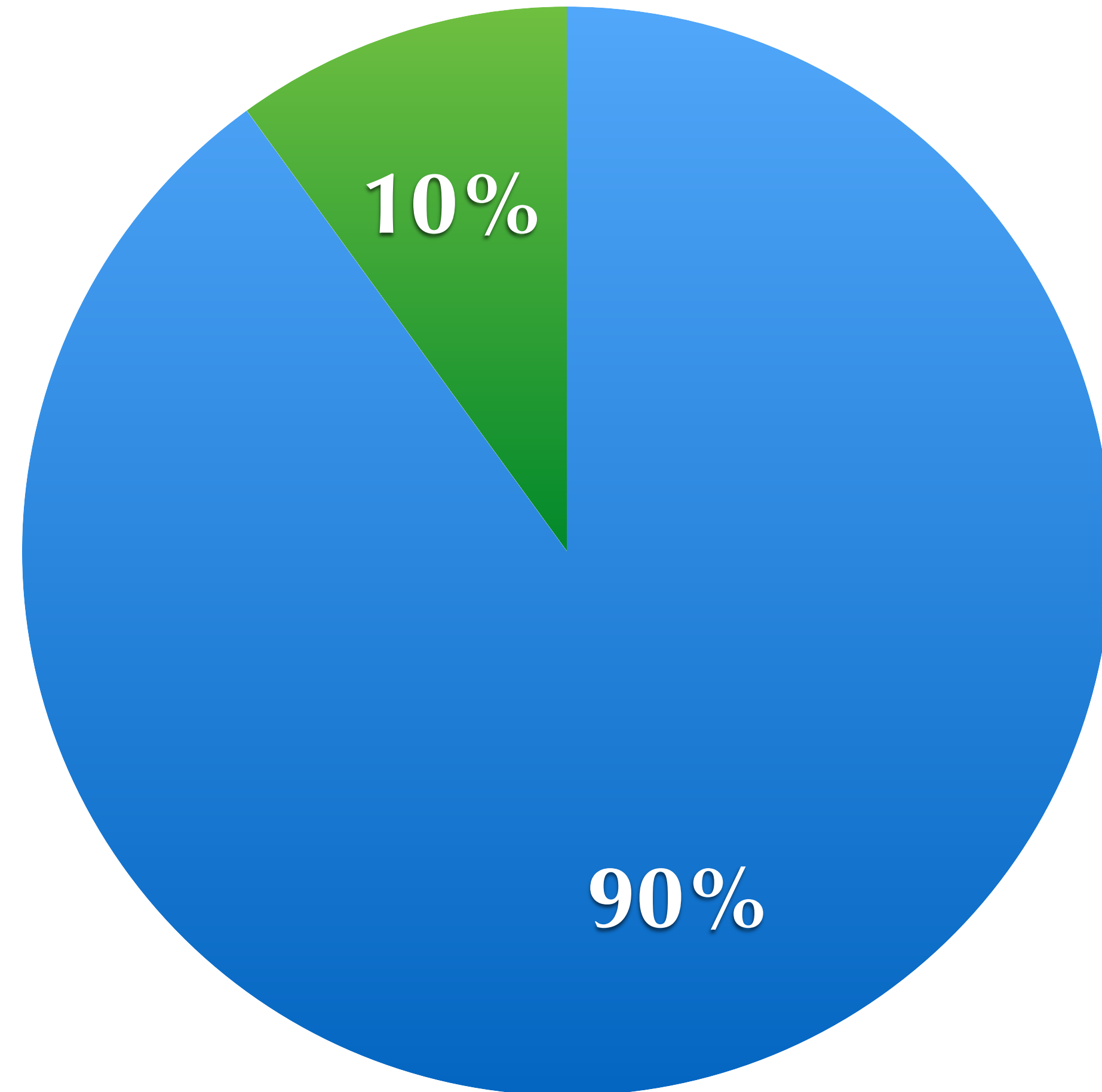
**Parallelization & Load Balancing**

...

**Data Structure Overhead**

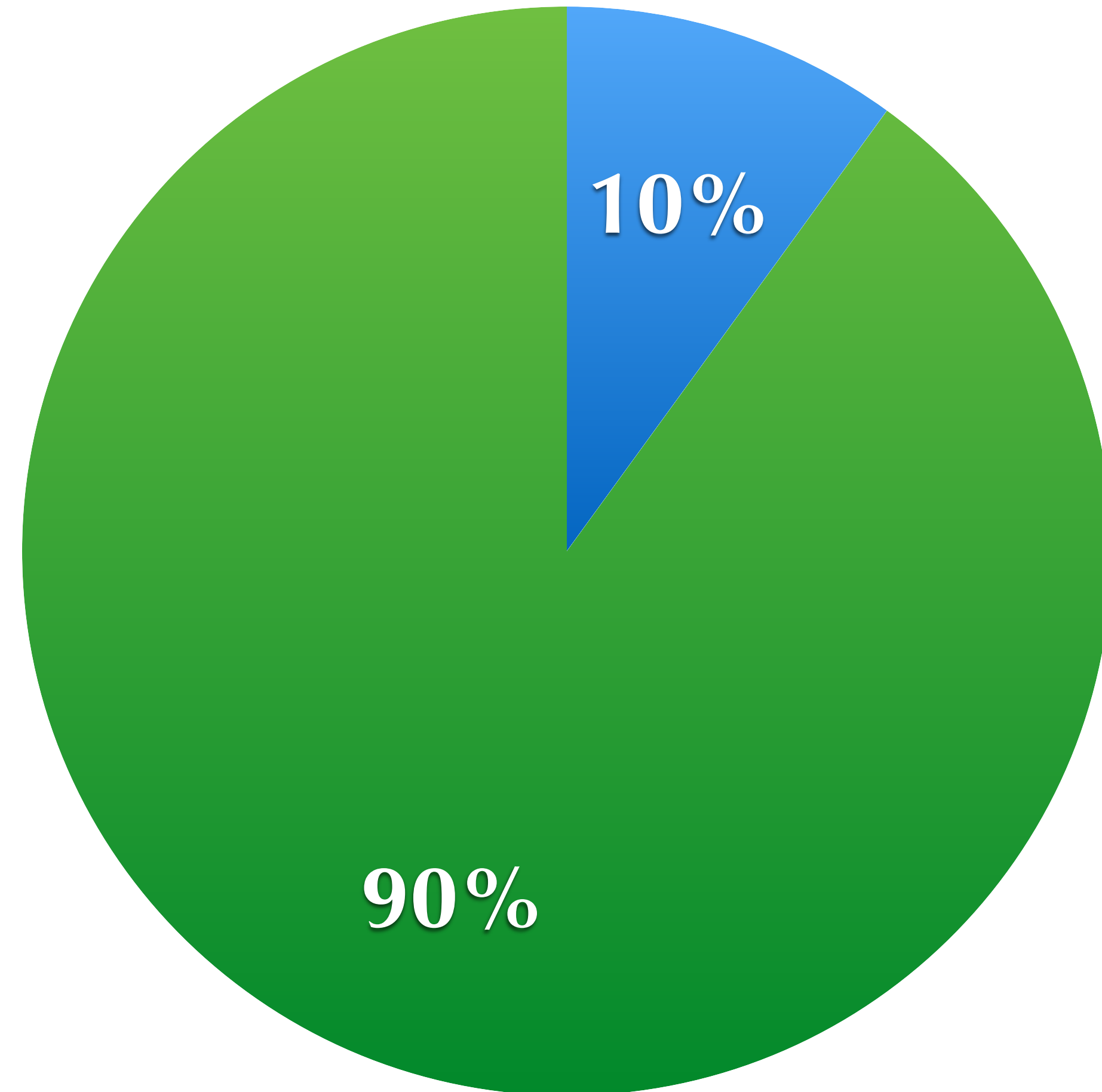


- Essential Computation
- Data Structure Overhead



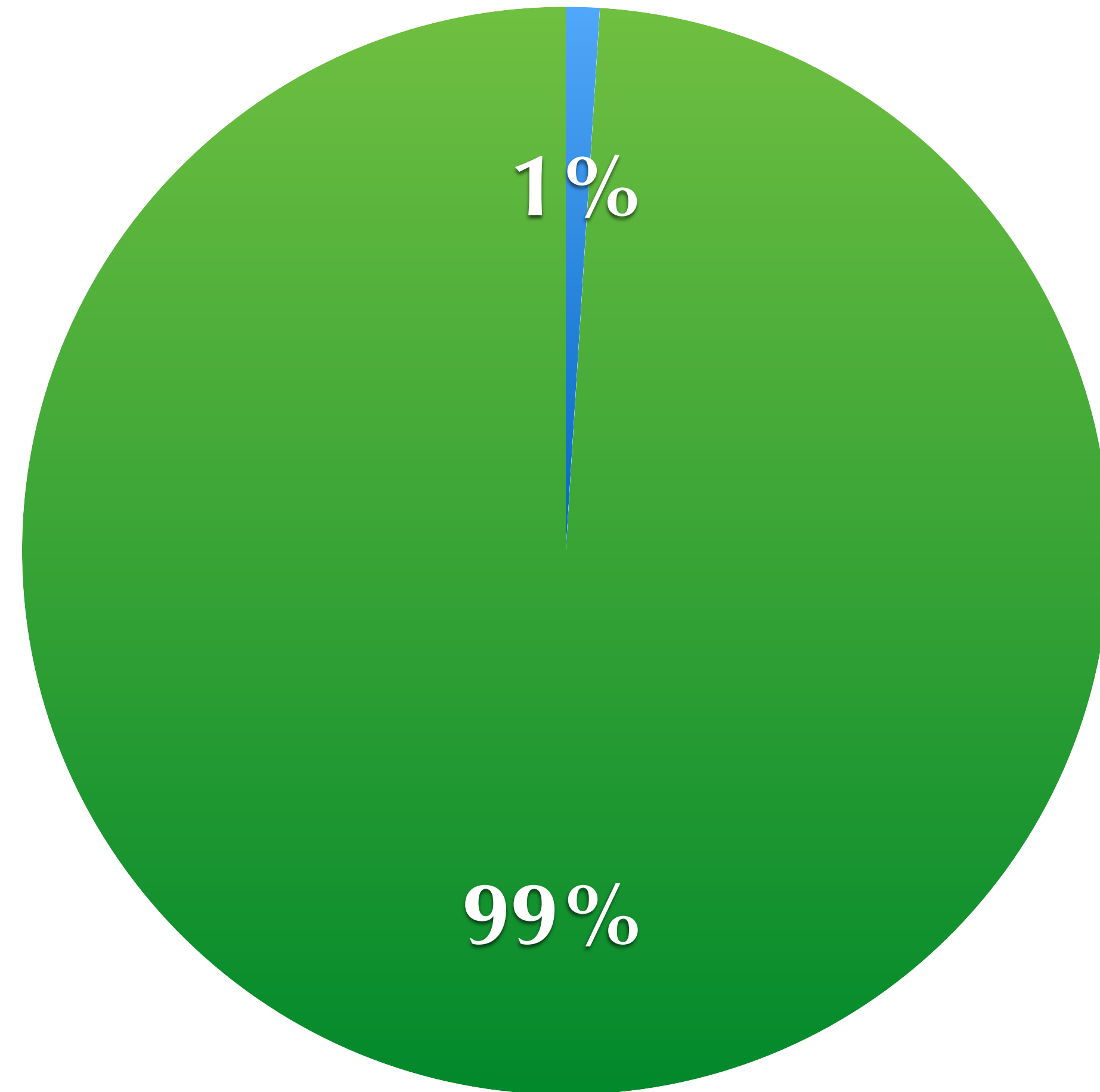
**Ideally...**

- Essential Computation
- Data Structure Overhead



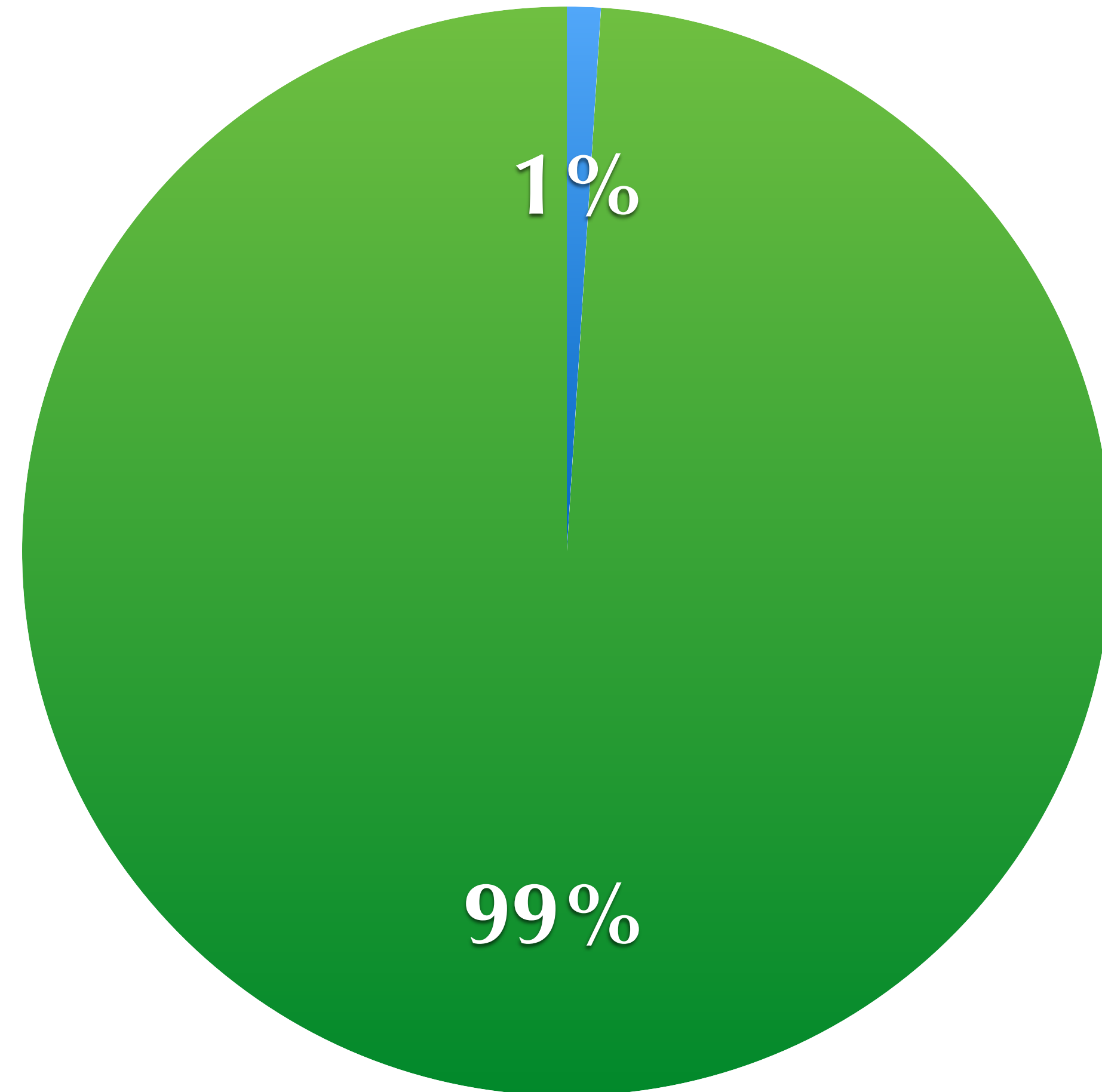
**In reality...**

- Essential Computation
- Data Structure Overhead



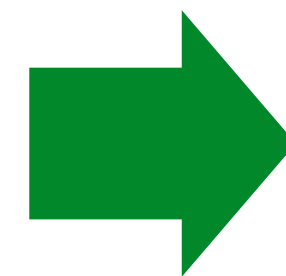
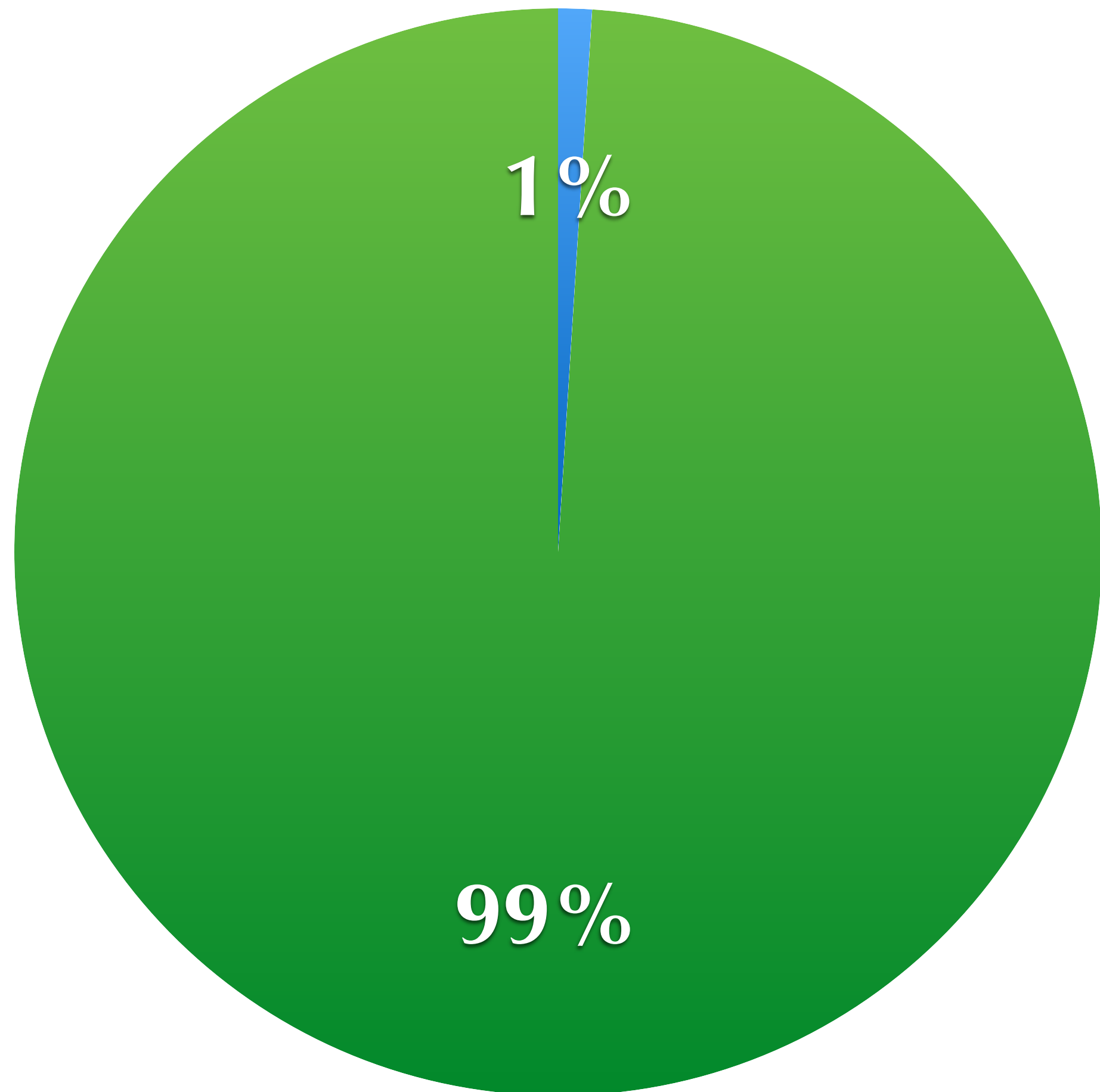
**In reality...**

- Essential Computation
- Data Structure Overhead



**In reality...**

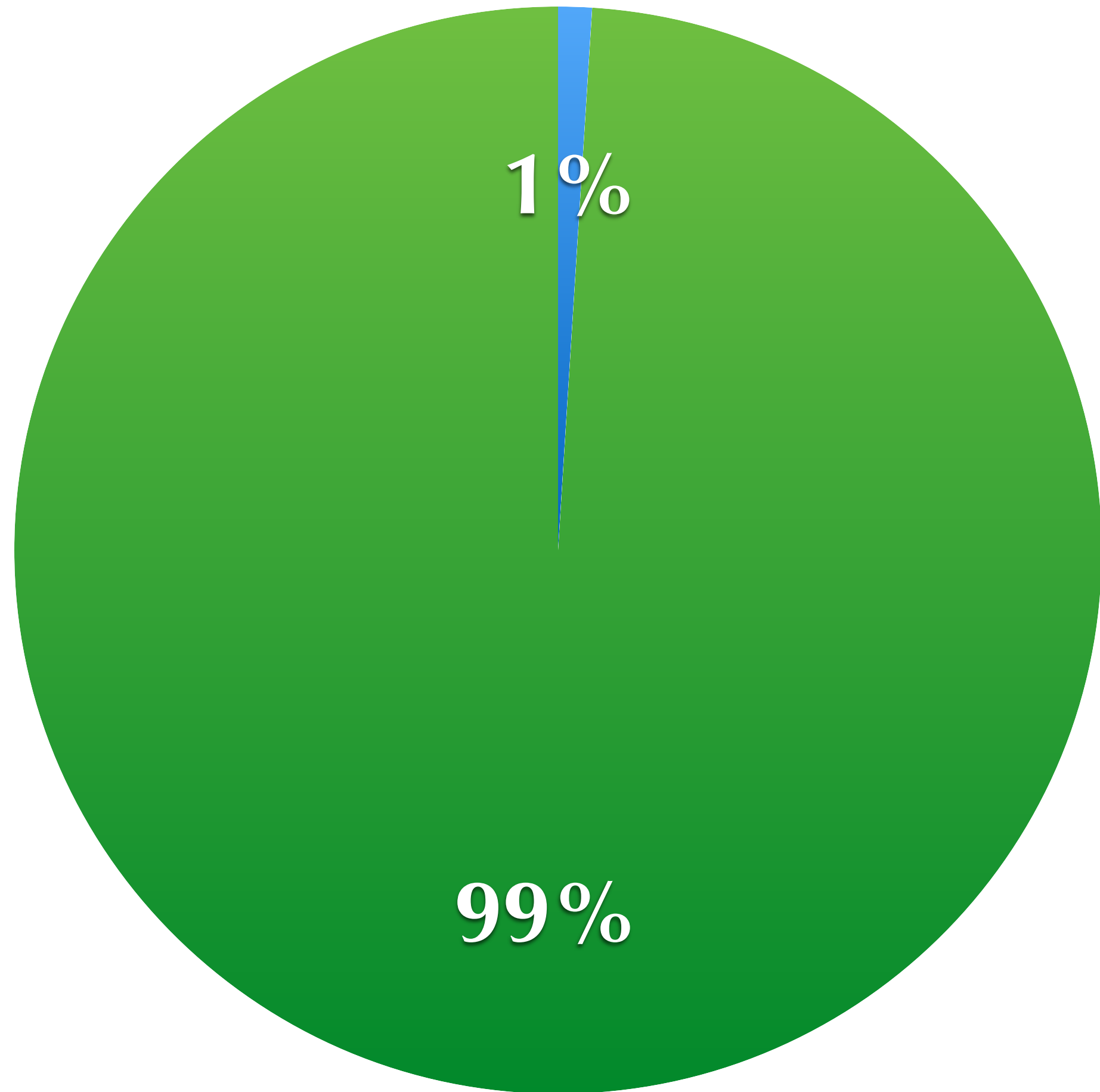
- Essential Computation
- Data Structure Overhead



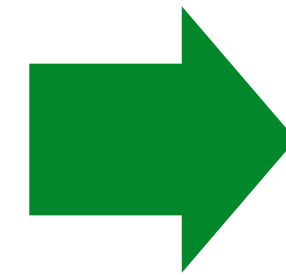
- Hash table lookup:** 10s of clock cycles
- Indirection:** cache/TLB misses
- Node allocation:** locks, atomics, barriers
- Branching:** misprediction / warp divergence
- ...

**In reality...**

- Essential Computation
- Data Structure Overhead



**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.

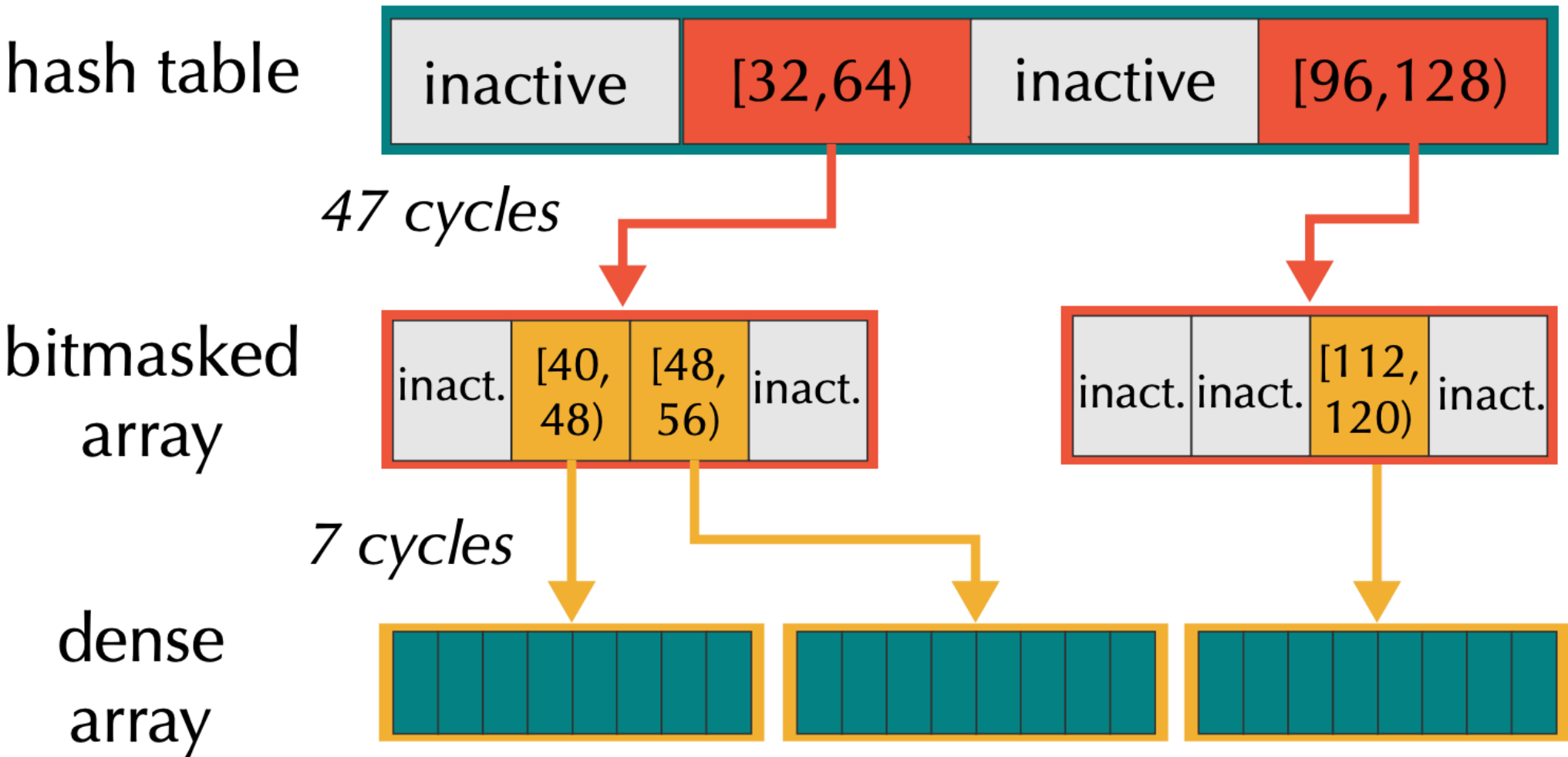
# Sparse Data Structure

## Data structure access:

- 50 clock cycles / element

## Simple Stencil Computation:

- 0.5 clock cycle / element



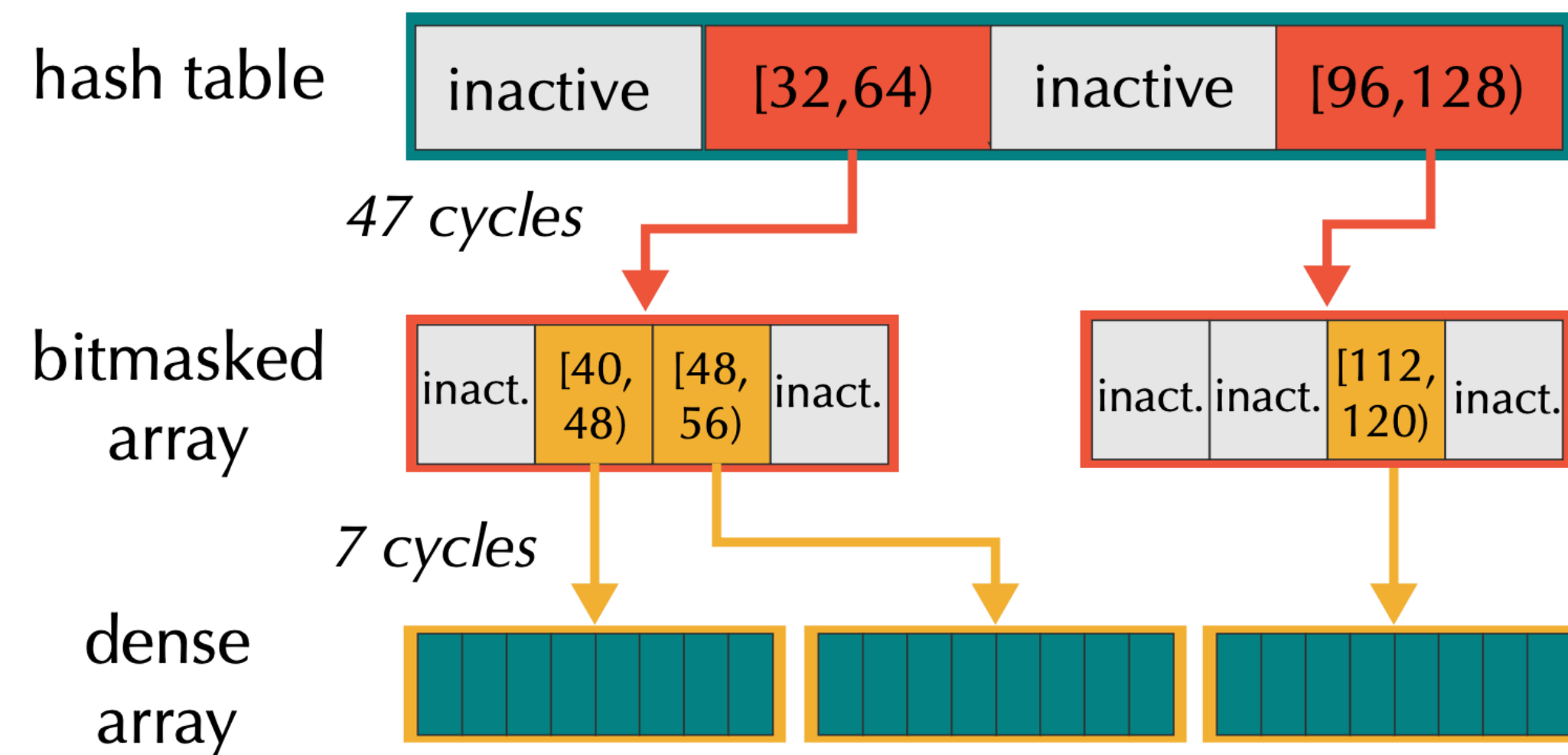
# Sparse Data Structure

## 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



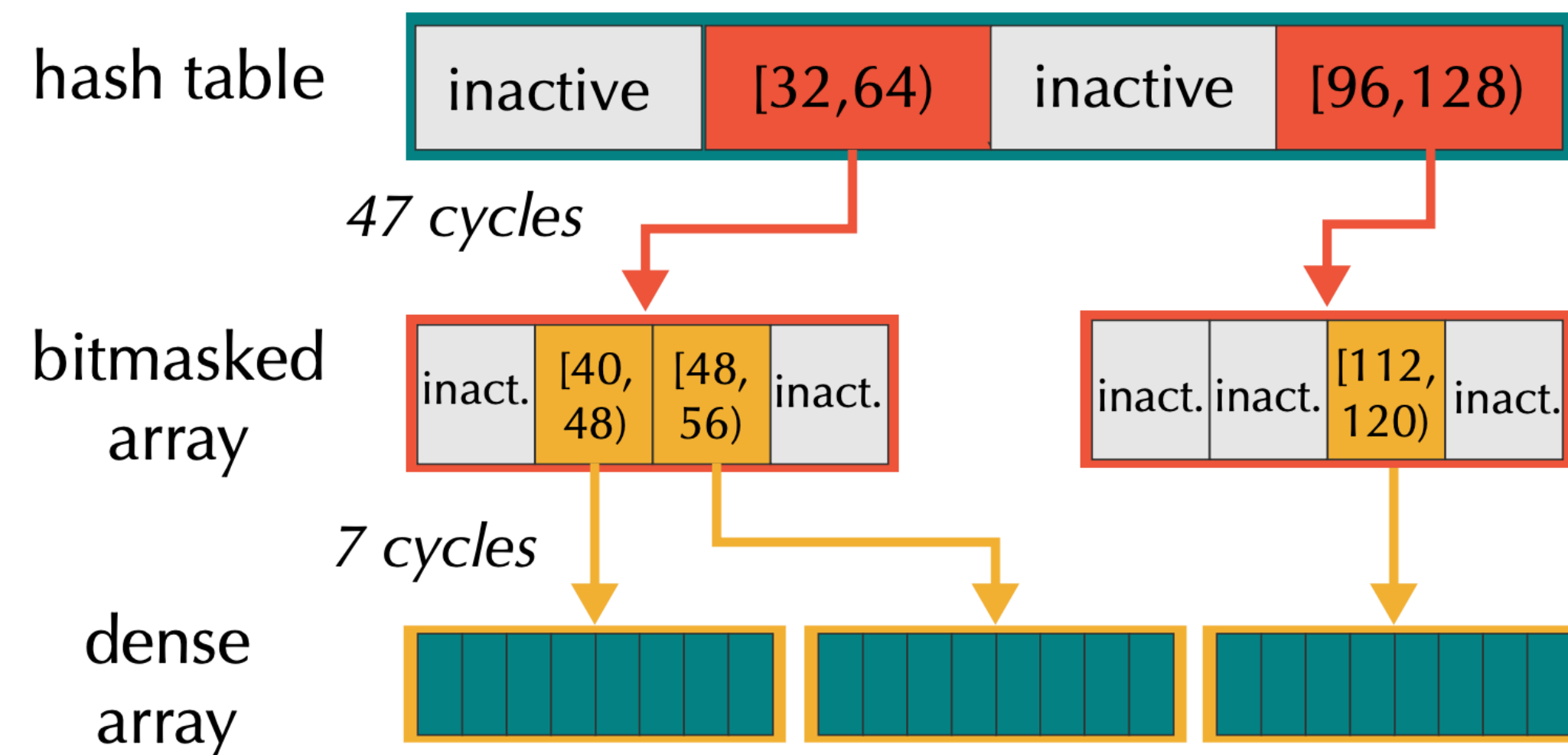
# Sparse Data Structure

## Data structure access:

- 50 clock cycles / element

## Simple Stencil Computation:

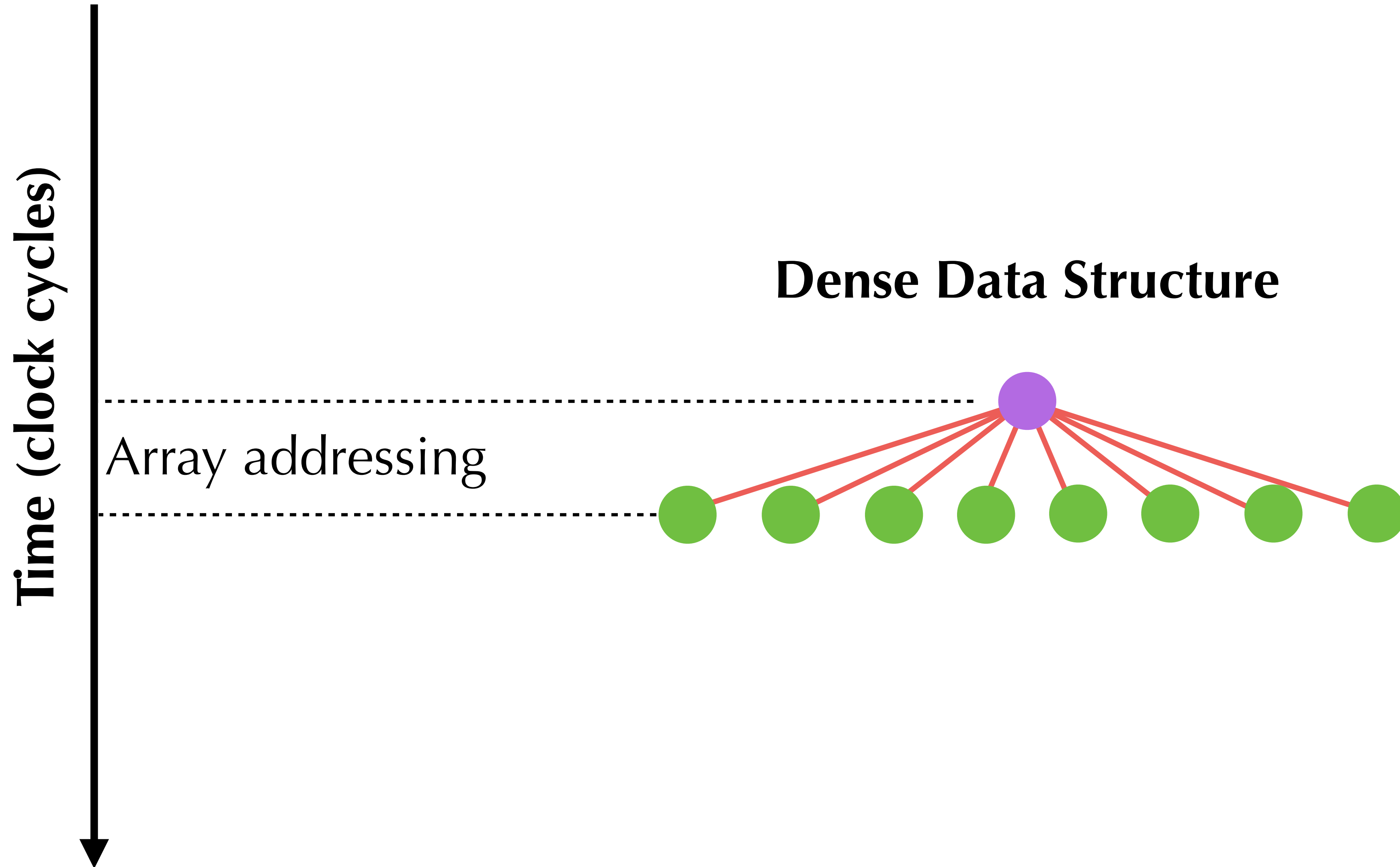
- 0.5 clock cycle / element



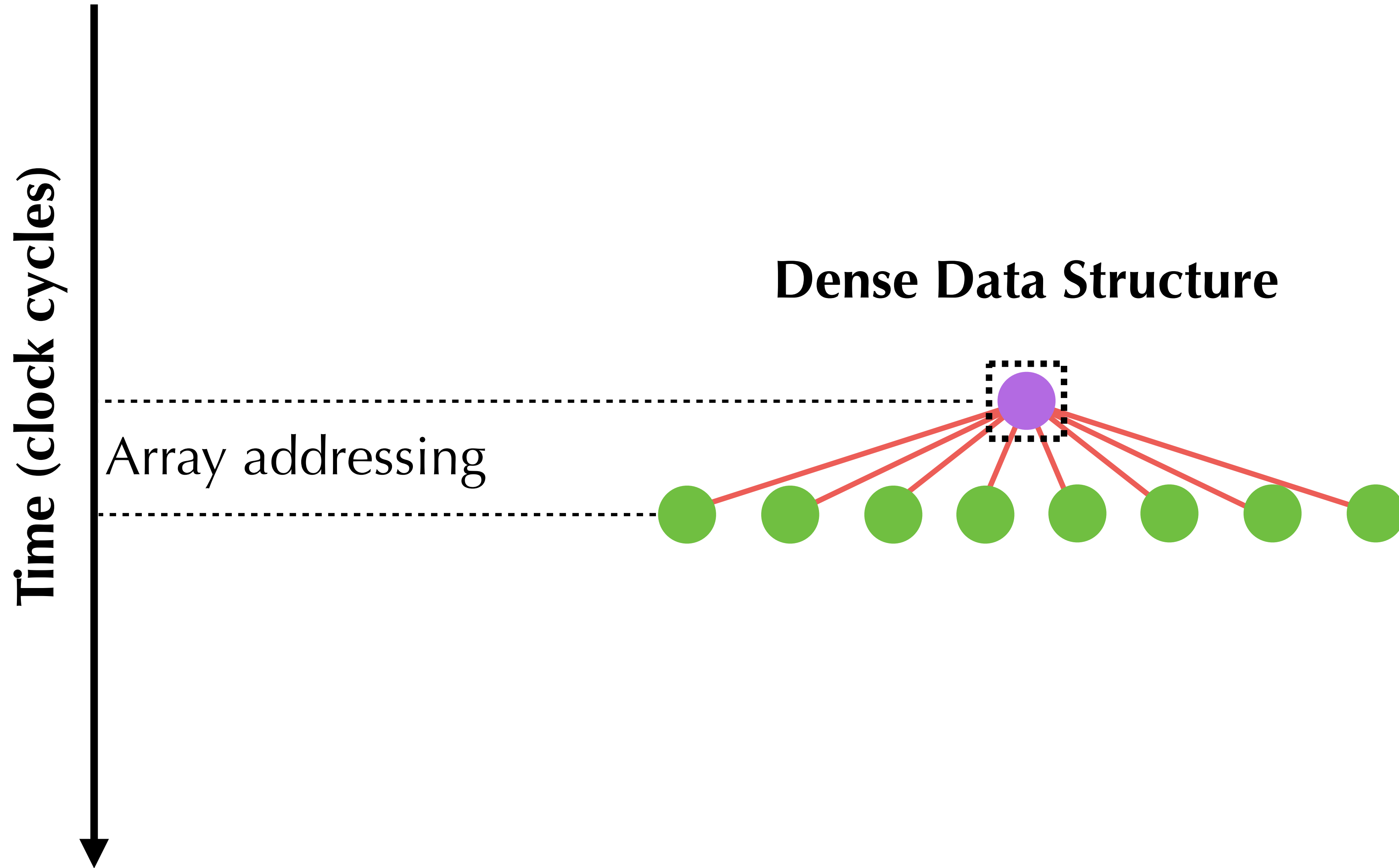
Sparse data structure overhead can be **100x** higher than essential computation

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

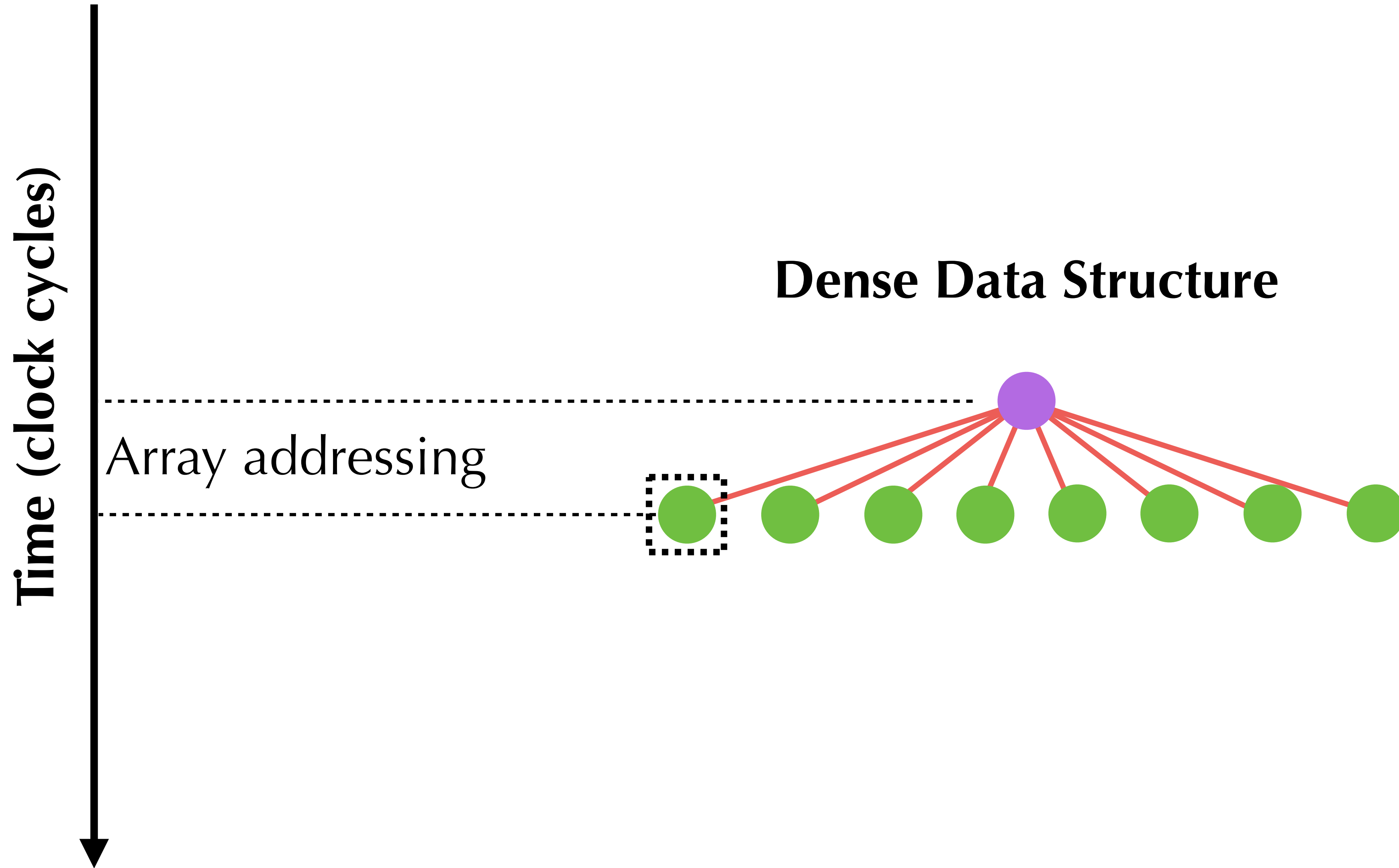
# Data Accesses Drawn **Proportionally**...



# Data Accesses Drawn **Proportionally**...

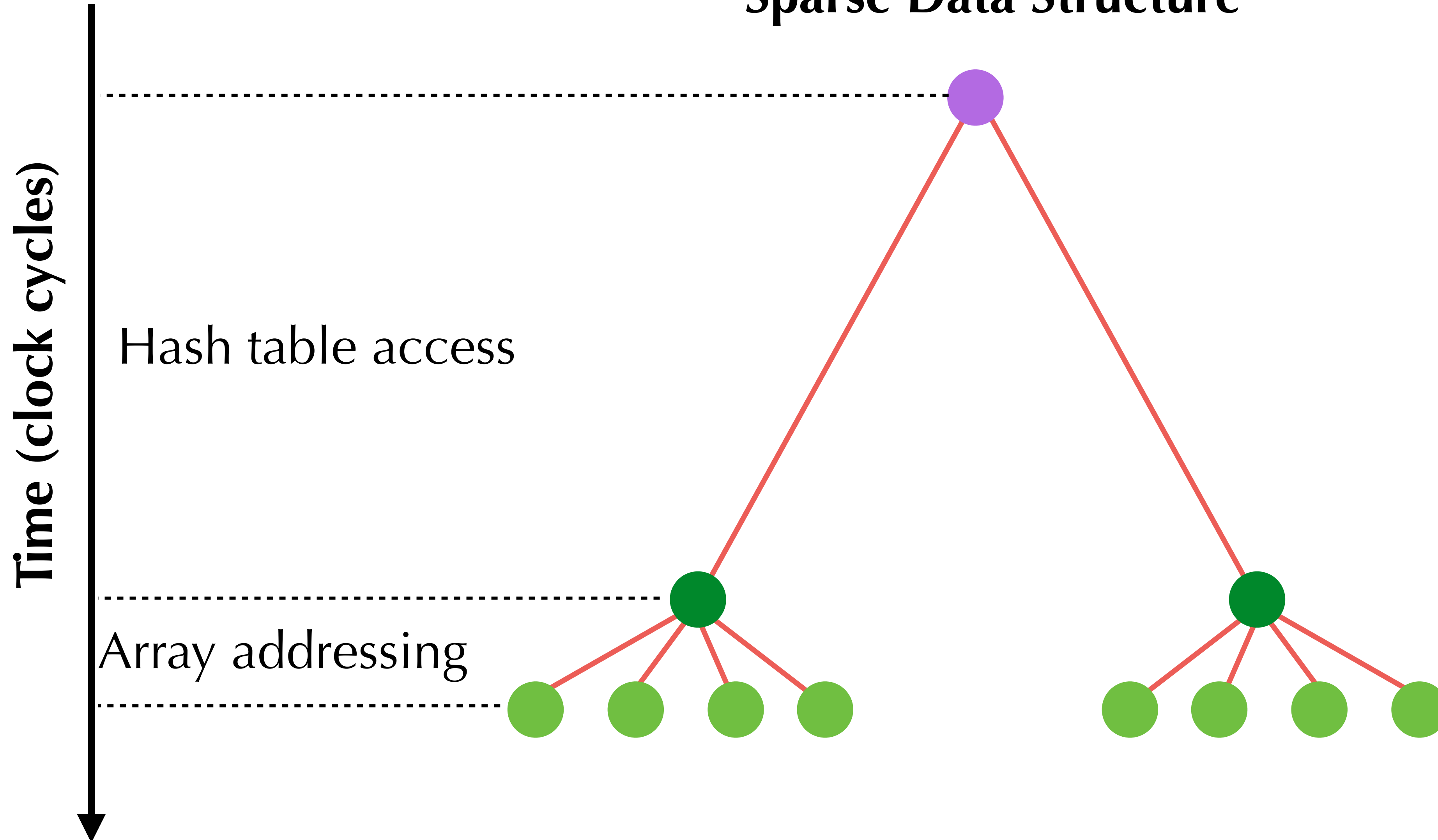


# Data Accesses Drawn **Proportionally**...

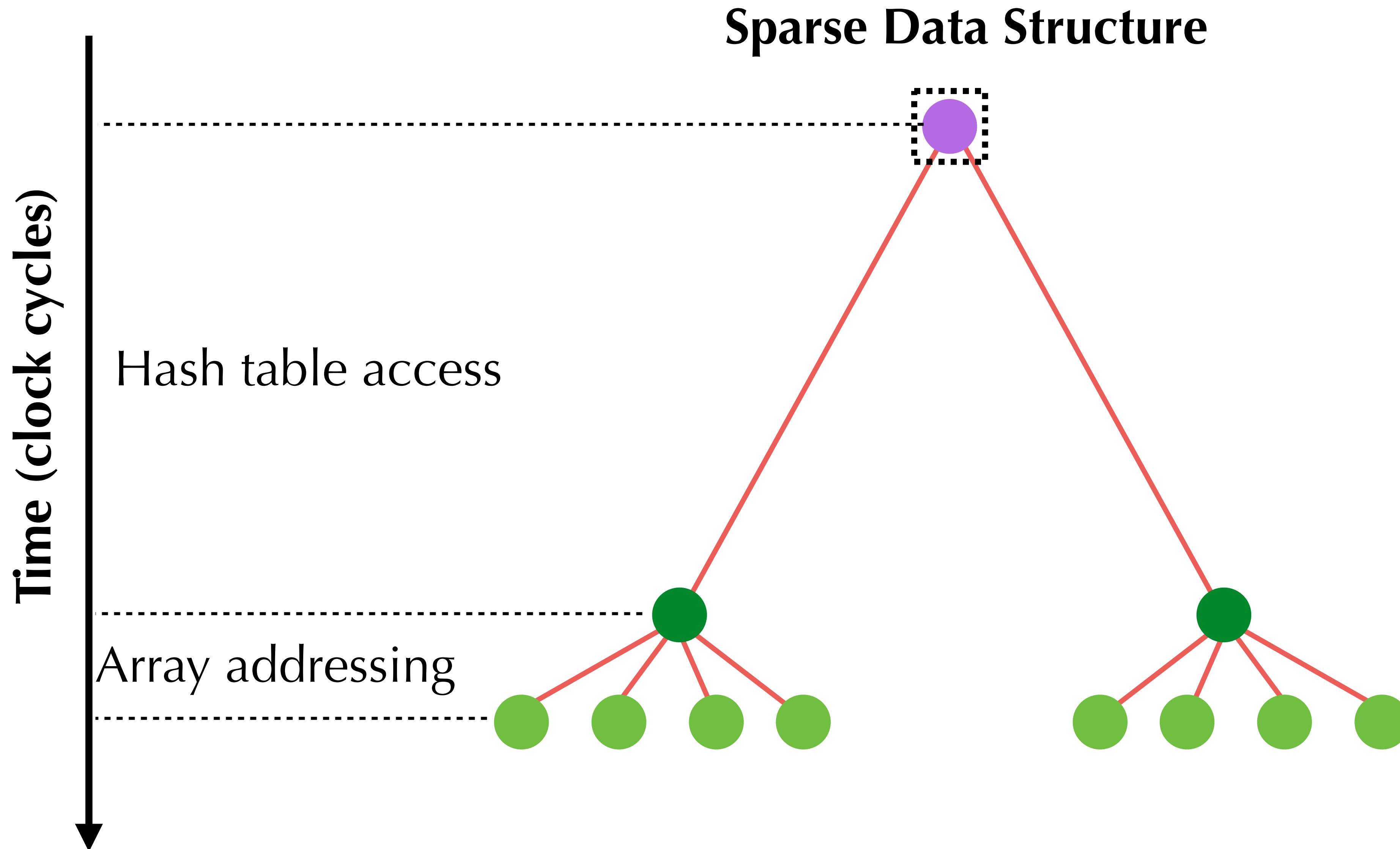


# Data Accesses Drawn Proportionally...

## Sparse Data Structure

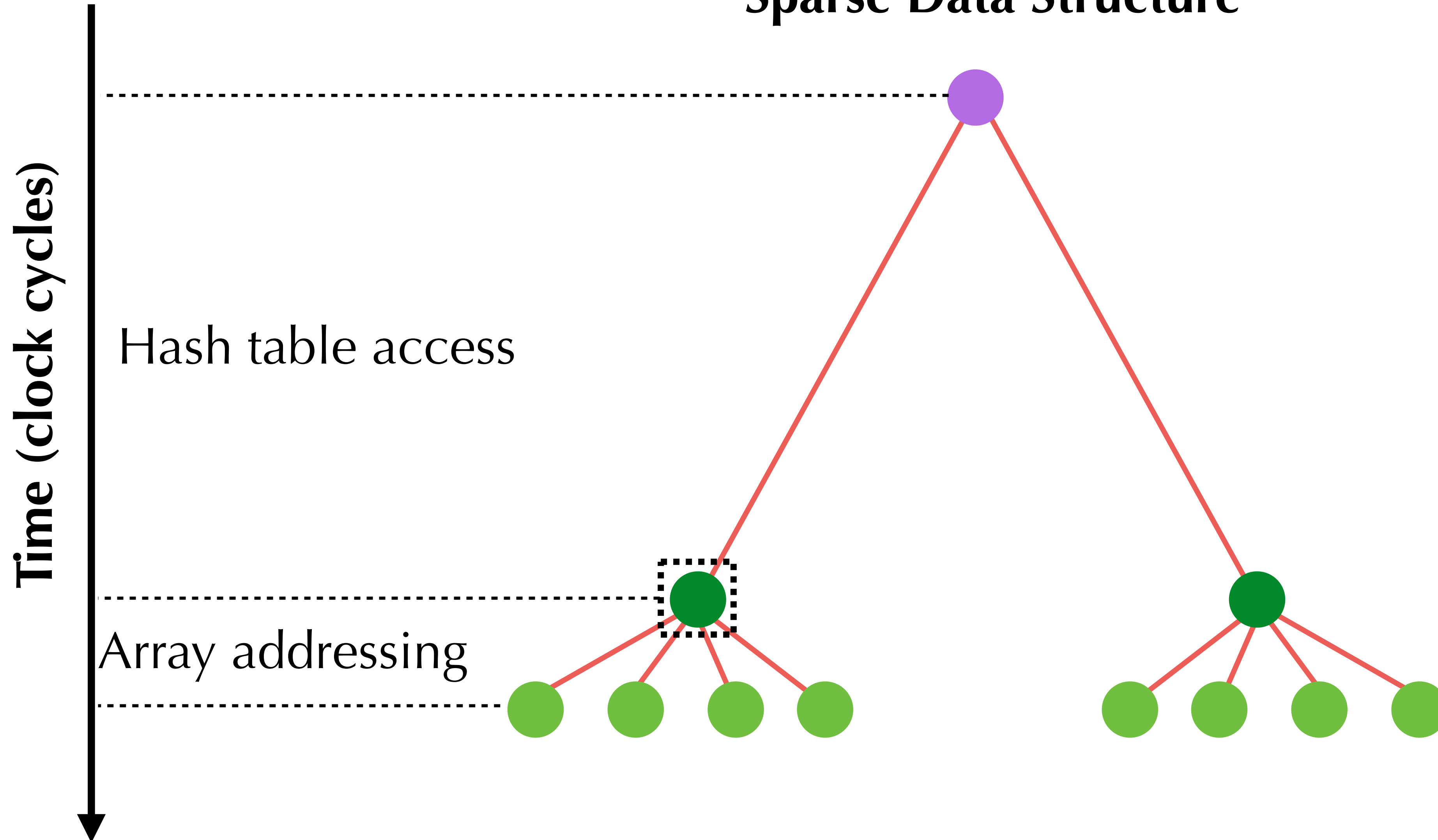


# Data Accesses Drawn Proportionally...



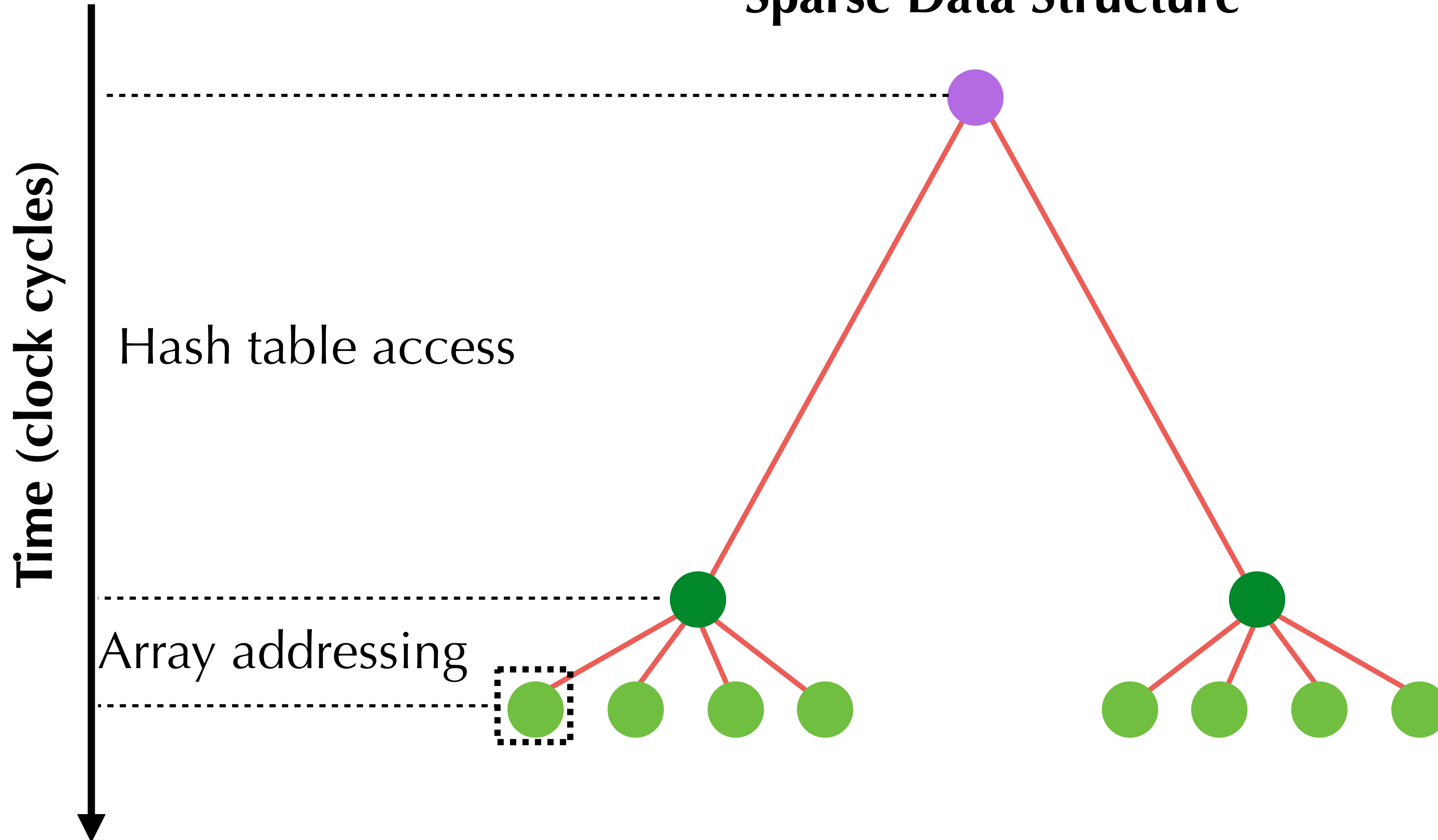
# Data Accesses Drawn Proportionally...

## Sparse Data Structure



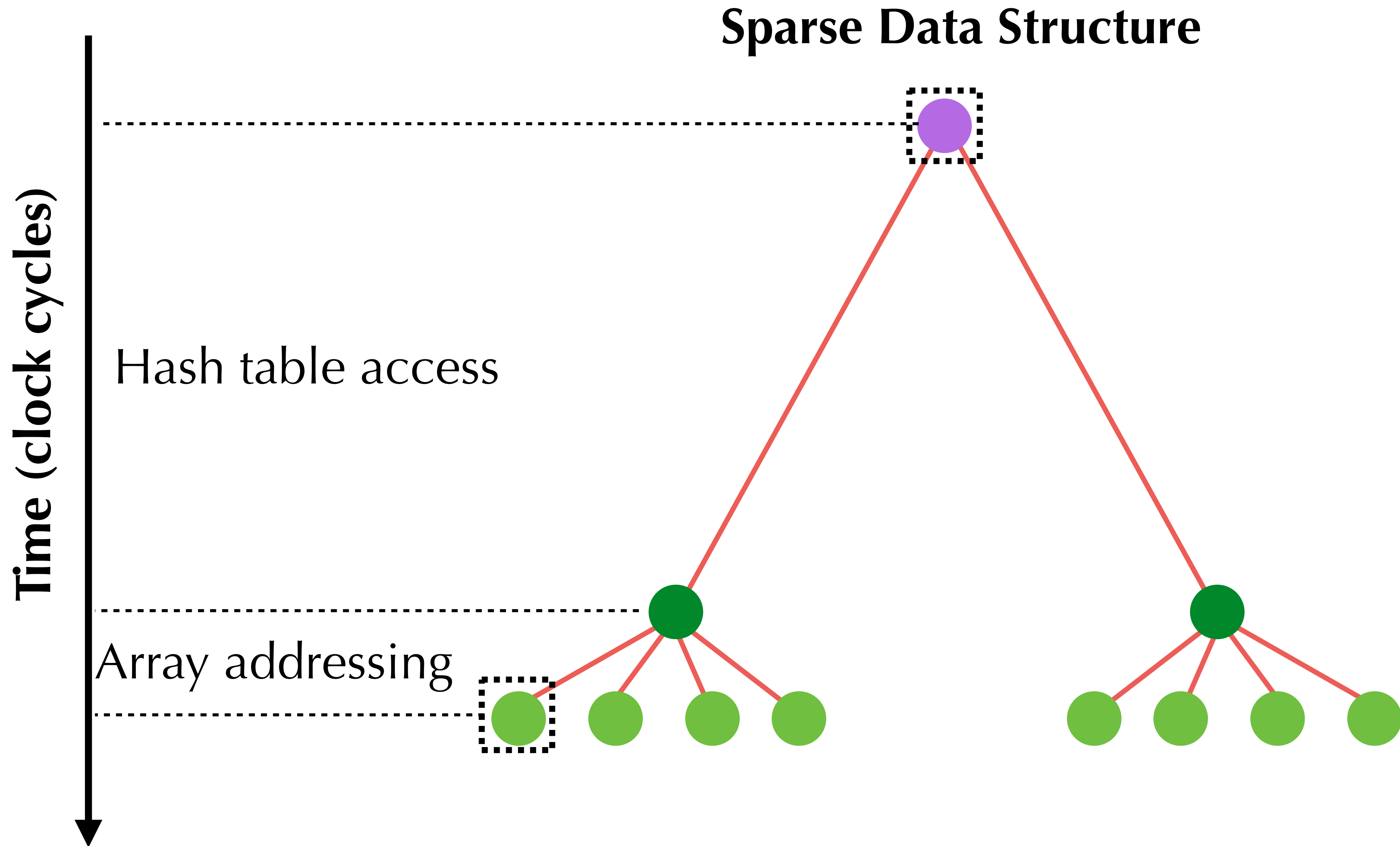
# Data Accesses Drawn Proportionally...

## Sparse Data Structure

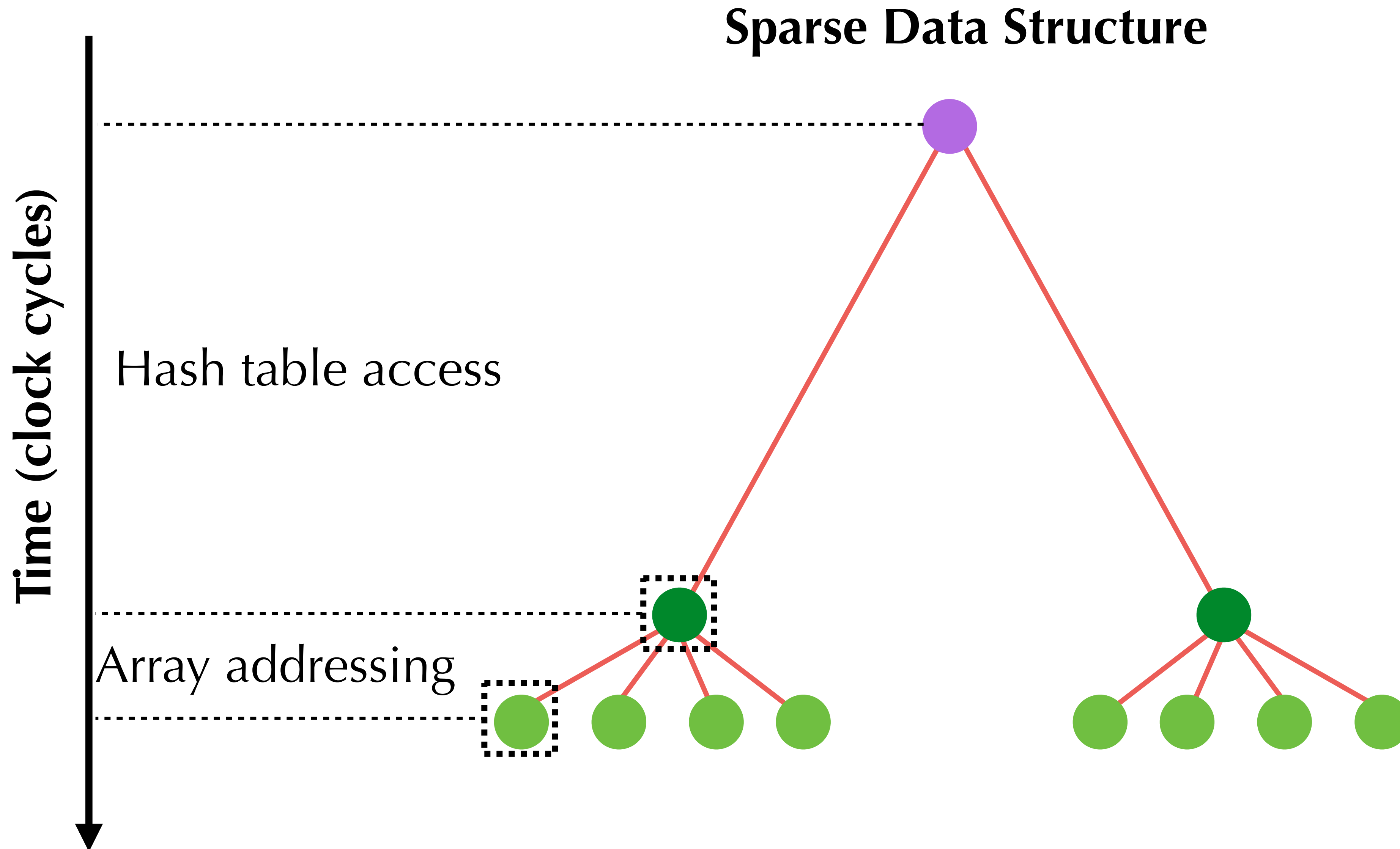




# Data Accesses Drawn Proportionally...

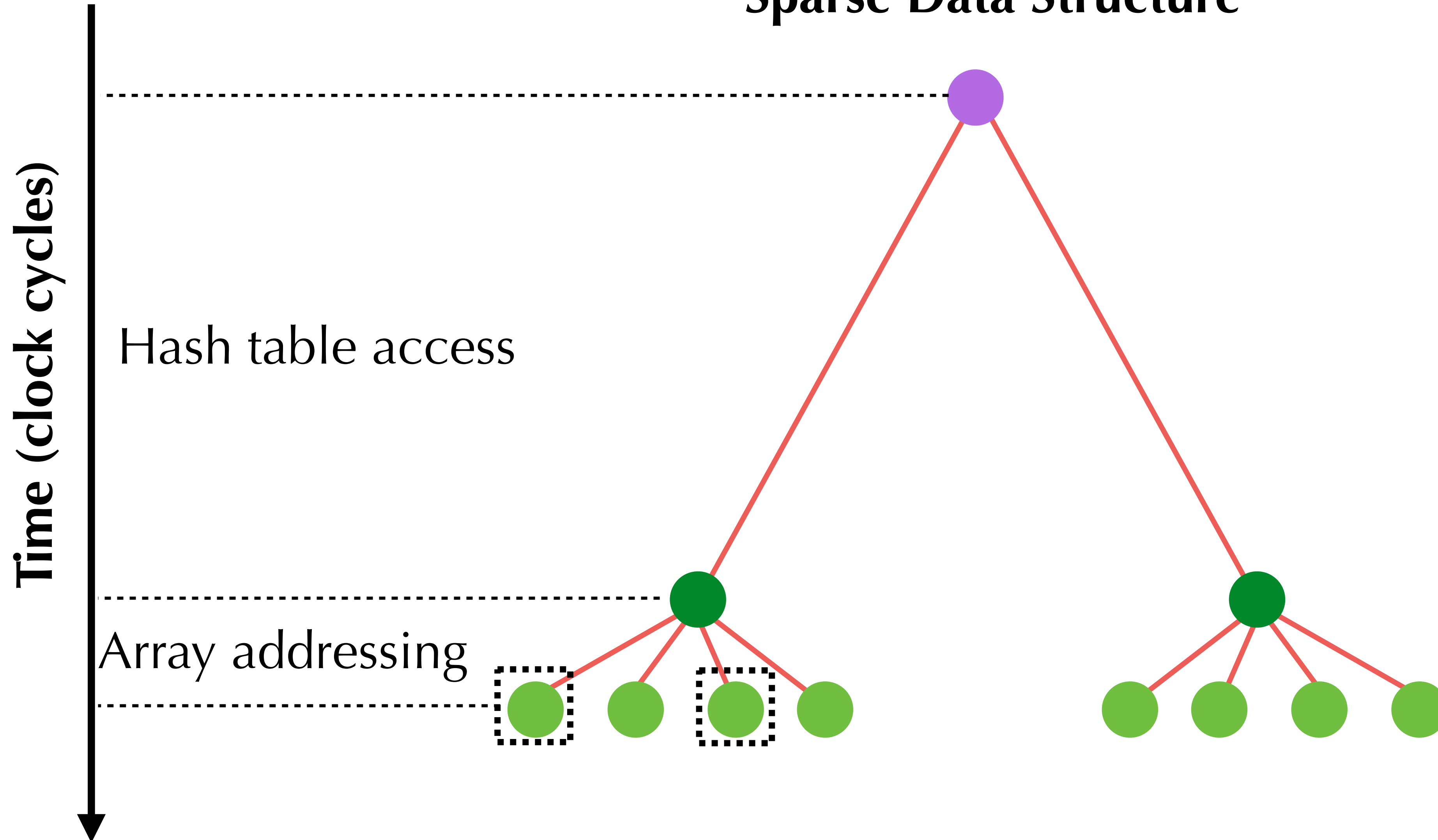


# Data Accesses Drawn Proportionally...



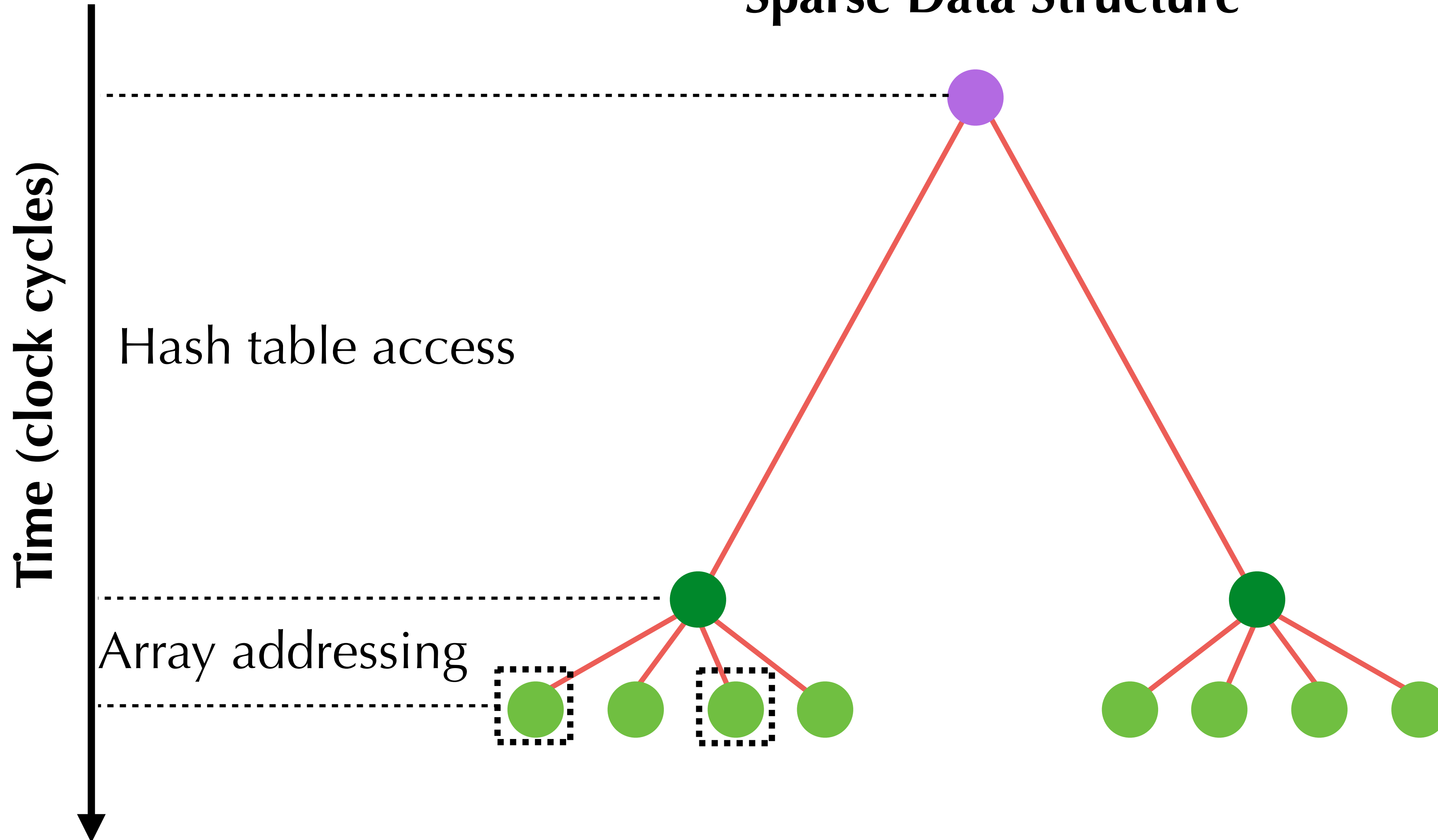
# Data Accesses Drawn Proportionally...

## Sparse Data Structure



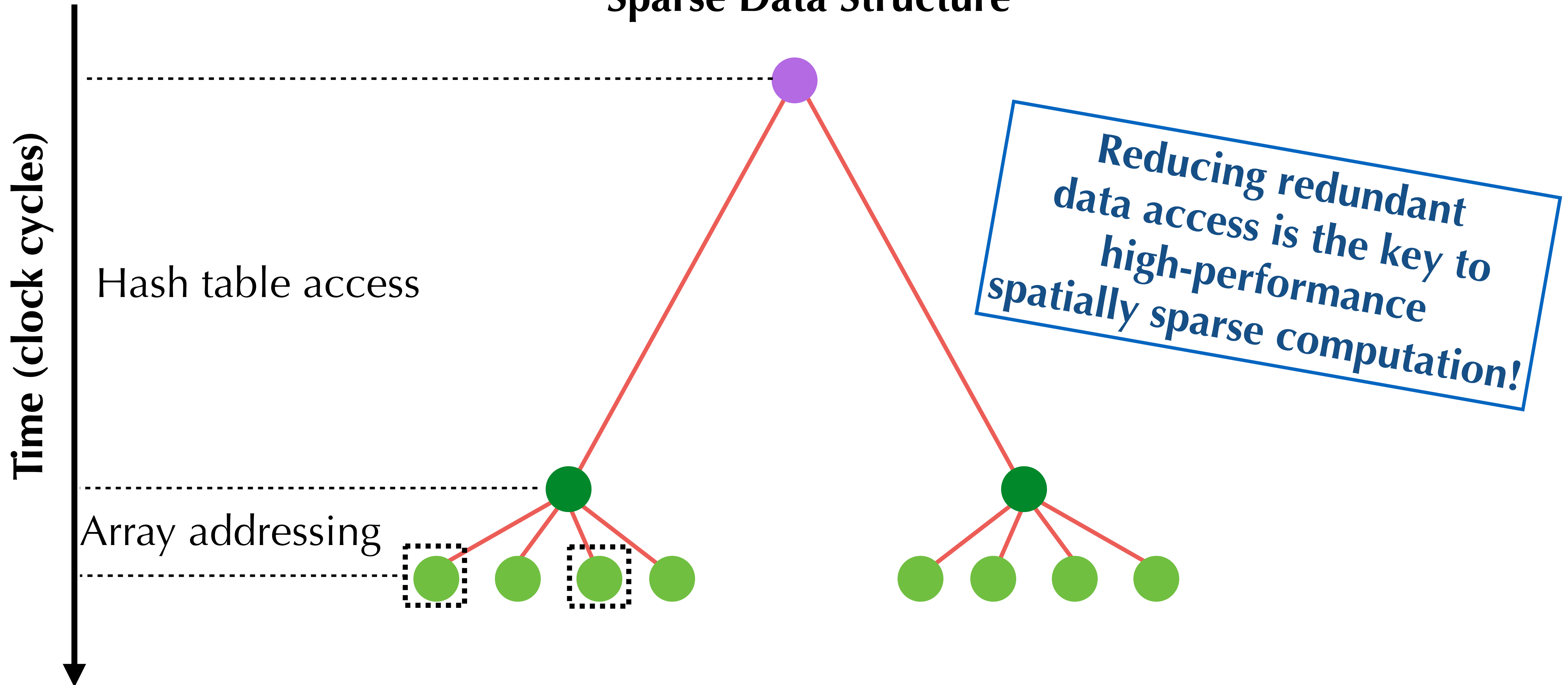
# Data Accesses Drawn Proportionally...

## Sparse Data Structure



# Data Accesses Drawn Proportionally...

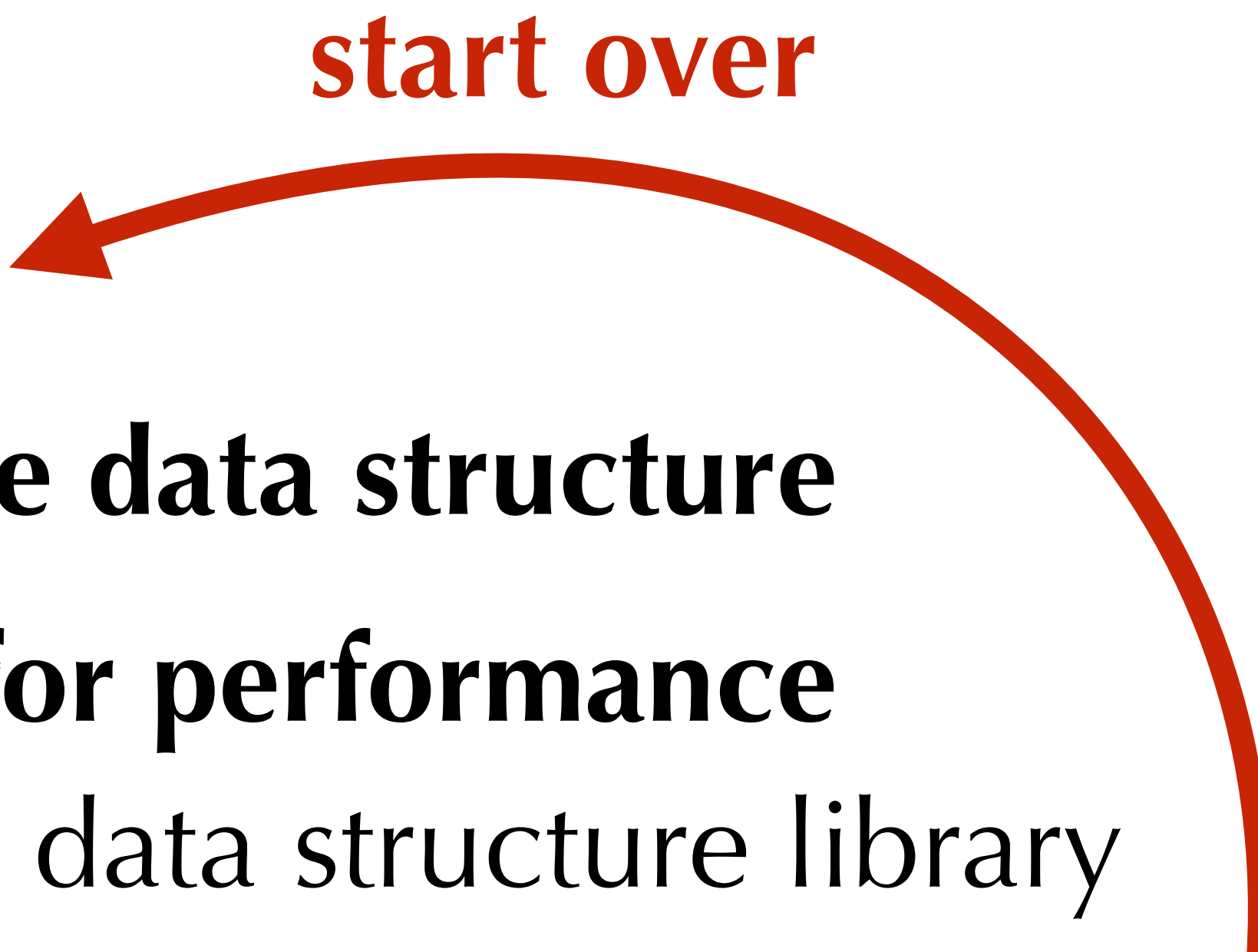
## Sparse Data Structure



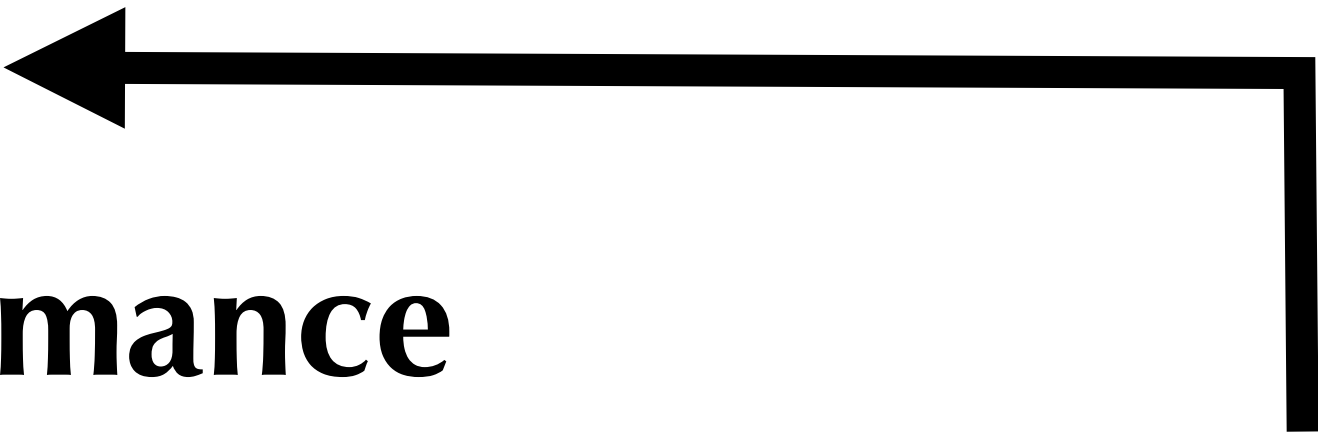
# Traditional Sparse Computation Workflow

- 1. Choose a sparse data structure library**
- 2. Implement the algorithm on that sparse data structure**
- 3. Do low-level engineering to optimize for performance**
  - Code is complex and coupled with the data structure library

# Traditional Sparse Computation Workflow

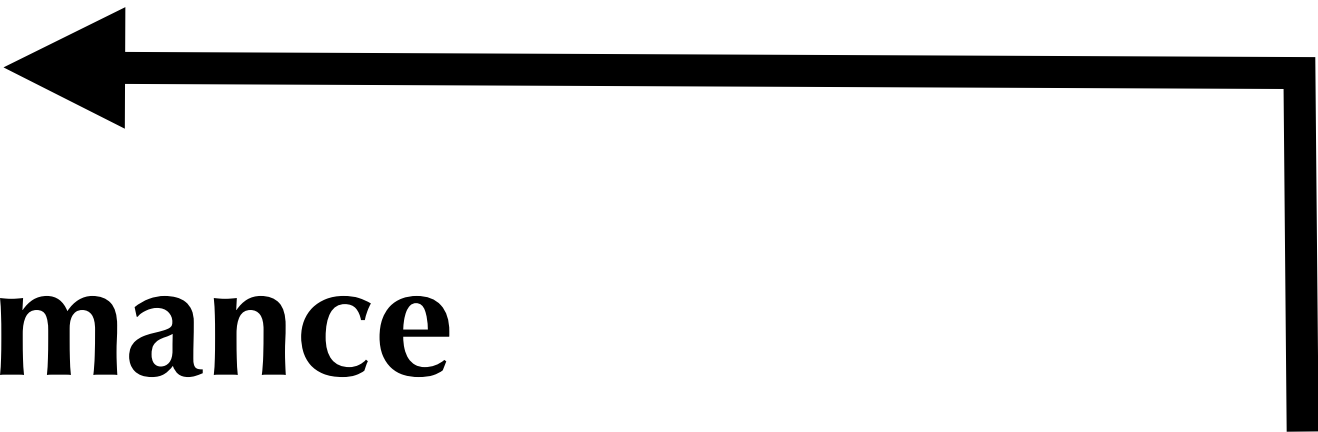
- 1. Choose a sparse data structure library**
  - 2. Implement the algorithm on that sparse data structure**
  - 3. Do low-level engineering to optimize for performance**
    - Code is complex and coupled with the data structure library
    - *“Oh no, this data structure isn’t really optimal for this algorithm”*
- 
- start over**

# Ideal Sparse Computing Workflow

1. **Implement the algorithm as if all grids are dense**
2. **Describe your data structure** 
3. **The compiler optimizes performance**
  - Benchmark performance, and **try different data structures**



# Ideal Sparse Computing Workflow

1. **Implement the algorithm as if all grids are dense**
2. **Describe your data structure** 
3. **The compiler optimizes performance**
  - Benchmark performance, and **try different data structures**

Related work: split languages,

e.g., **Halide** [Ragan-Kelley, Adams, Paris, Levoy, Amarasinghe, Durand. SIGGRAPH 2012]

Our Solution:

***The Taichi Programming Language***

Our Solution:

# *The Taichi Programming Language*

1) **Decouple** *computation* from *data structures*

Computational Kernels

(Sparse) Data Structures

Our Solution:

# *The Taichi Programming Language*

## 1) **Decouple** *computation* from *data structures*

### Computational Kernels

```
Kernel(laplace).def([&]()) {  
  For(u, [&](Expr i, Expr j){  
    auto c = 1.0f / (dx * dx);  
    u[i, j] = c * (4 * v[i, j] - v[i+1, j]  
                  - v[i-1, j] - v[i, j+1] - v[i, j-1]);  
  });  
};  
2D Laplace operator
```

### (Sparse) Data Structures

## 2) **Imperative** computation language

Our Solution:

# *The Taichi Programming Language*

## 1) **Decouple** *computation* from *data structures*

### Computational Kernels

```
Kernel(laplace).def([&]() {  
  For(u, [&](Expr i, Expr j){  
    auto c = 1.0f / (dx * dx);  
    u[i, j] = c * (4 * v[i, j] - v[i+1, j]  
      - v[i-1, j] - v[i, j+1] - v[i, j-1]);  
  });  
});
```

2D Laplace operator

### (Sparse) Data Structures

```
Global(u, f32); Global(v, f32);  
layout([&]() {  
  auto ij = Indices(0, 1);  
  root.dense(ij, {128, 128}).pointer()  
  .dense(ij, {8, 8}).place(u, v);  
});
```

1024<sup>2</sup> sparse grid with 8<sup>2</sup>

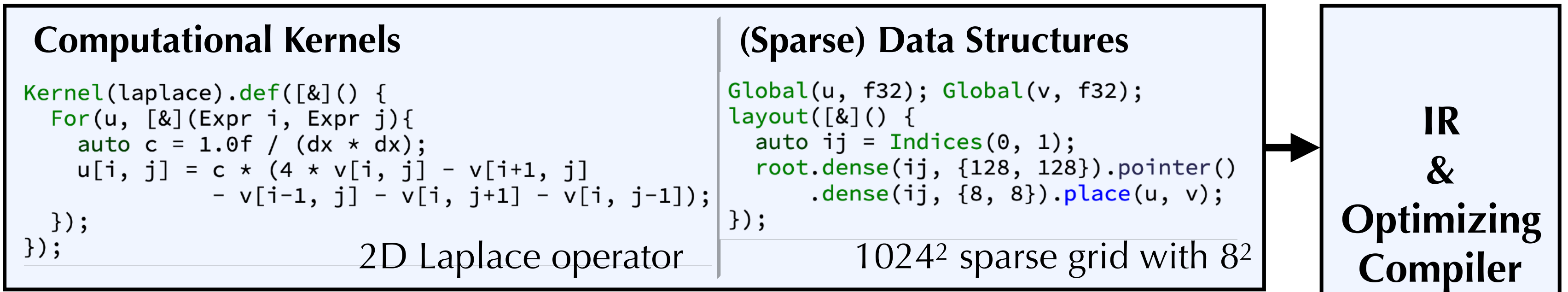
2) **Imperative** computation  
language

3) **Hierarchical** data  
structure description  
language

Our Solution:

# *The Taichi Programming Language*

1) **Decouple** *computation* from *data structures*



2) **Imperative** computation language

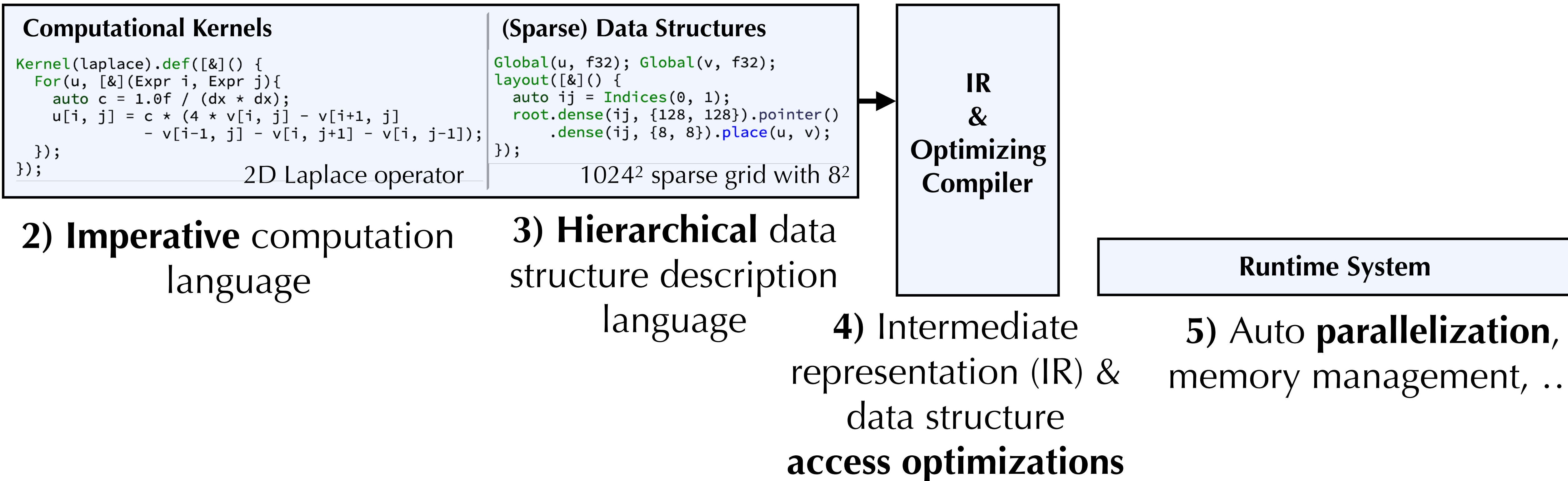
3) **Hierarchical** data structure description language

4) Intermediate representation (IR) & data structure access optimizations

Our Solution:

# *The Taichi Programming Language*

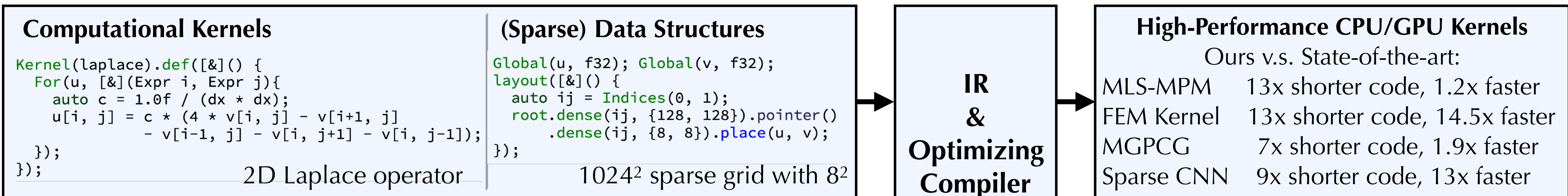
1) **Decouple** *computation* from *data structures*



Our Solution:

# *The Taichi Programming Language*

1) **Decouple** *computation* from *data structures*



2) **Imperative** computation language

3) **Hierarchical** data structure description language

4) Intermediate representation (IR) & data structure access optimizations

5) Auto **parallelization**, memory management, ...

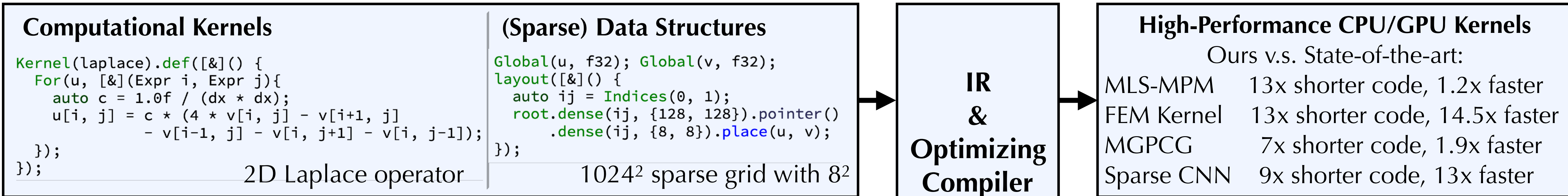


Our Solution:

# *The Taichi Programming Language*

1) **Decouple** *computation* from *data structures*

**10x shorter code, 4.55x faster**



2) **Imperative** computation language

3) **Hierarchical** data structure description language

4) Intermediate representation (IR) & data structure access optimizations

5) Auto **parallelization**, memory management, ...

**Runtime System**

# Defining Computation

## Finite Difference Stencil

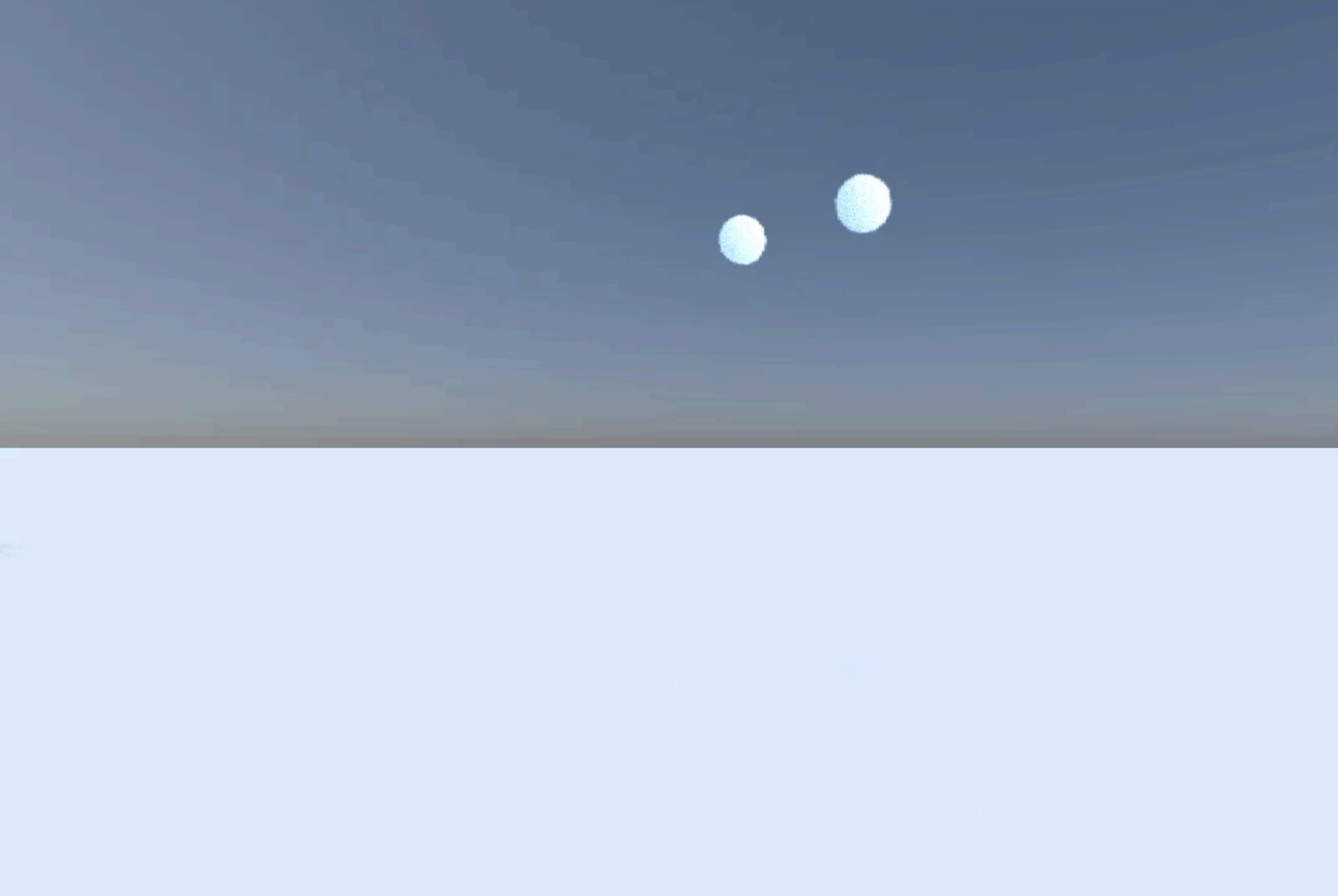
$$u_{i,j} = \frac{1}{\Delta x^2} (4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1})$$



## Taichi Kernel

```
1 @ti.kernel
2 def laplace():
3     for i, j in u:
4         c = 1 / (dx * dx)
5         u[i, j] = c * (4.0 * v[i, j] - v[i-1, j] - v[i+1, j]
6                       - v[i, j-1] - v[i, j+1])
```

- Program on **sparse** data structures as if they are **dense**;
- **Parallel** for-loops (Single-Program-Multiple-Data, like CUDA/ispc);
- Loop over only active elements in the sparse data structure;
- Complex **control flows** (e.g. **If, While**) supported.



Sample/pixel/sec: 7.126

depth\_limit: 20

density\_scale: 400.000

max\_density: 724.000

ground\_y: 0.029

light\_phi: 0.419

light\_theta: 0.218

light\_smoothness: 0.050

light\_ambient: 0.150

exposure: 0.567

gamma: 0.500

file\_id: 0

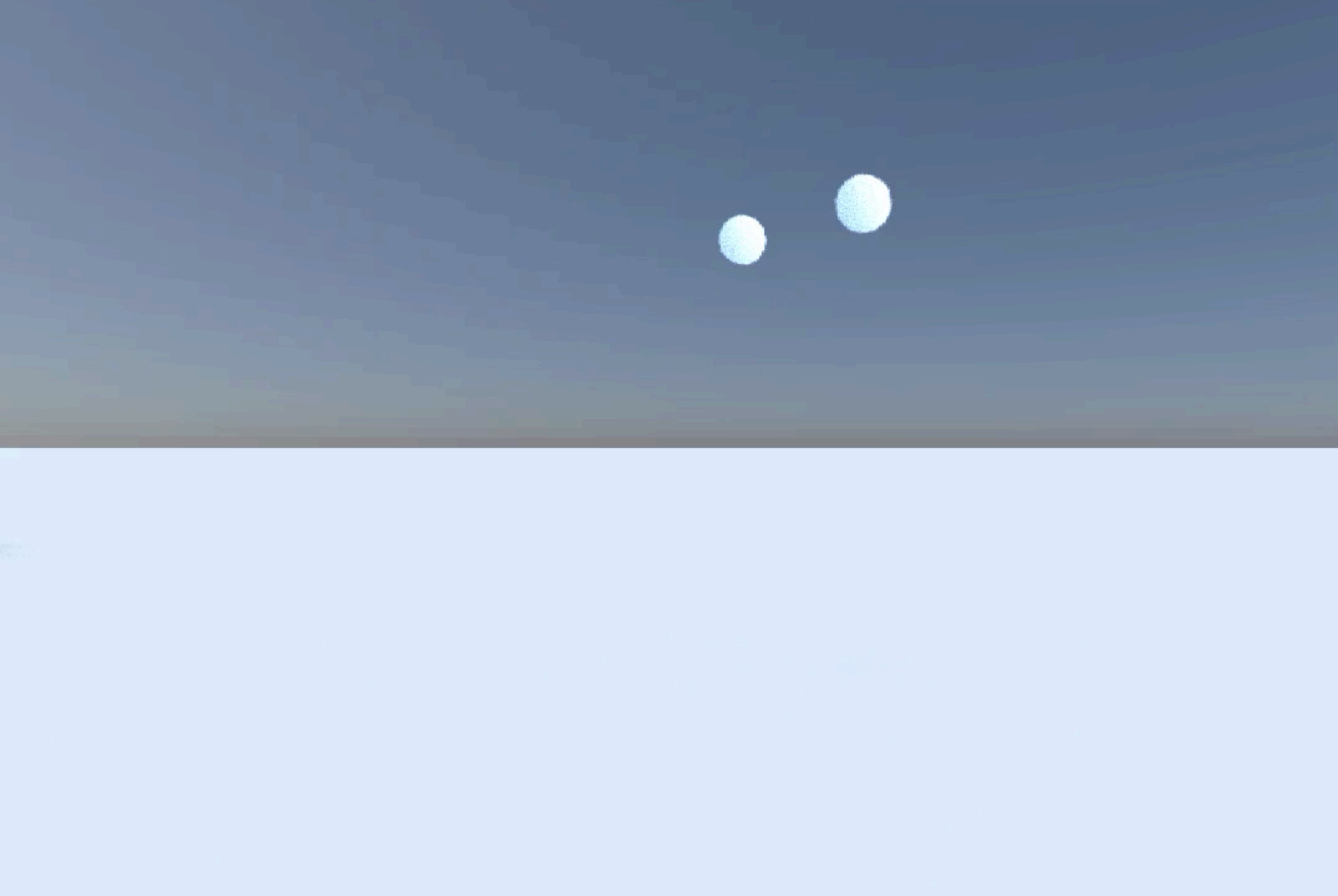
output\_samples: 10

grid\_level: 1

video\_step: 1

Save

Render All



Sample/pixel/sec: 7.126

depth\_limit: 20

density\_scale: 400.000

max\_density: 724.000

ground\_y: 0.029

light\_phi: 0.419

light\_theta: 0.218

light\_smoothness: 0.050

light\_ambient: 0.150

exposure: 0.567

gamma: 0.500

file\_id: 0

output\_samples: 10

grid\_level: 1

video\_step: 1

Save

Render All

# Describing Data Structures

**dense**: A fixed-length contiguous array.

**hash**: Use a hash table to maintain the mapping from active coordinates to data address in memory. Suitable for high sparsity.

**dynamic**: Variable-length array, with a predefined maximum length. It serves the role of `std::vector`, and can be used to maintain objects (e.g. particles) contained by a block.

## Structural Nodes

**morton**: Reorder the data in memory using a Z-order curve (Morton coding), for potentially higher spatial locality. For **dense** only.

**bitmasked**: Use a mask to maintain sparsity information, one bit per child. For **dense** only.

**pointer**: Store pointers instead of the whole structure to save memory and maintain sparsity. For **dense** and **dynamic**.

## Node Decorators

# Describing Data Structures

**dense**: A fixed-length contiguous array.

**hash**: Use a hash table to maintain the mapping from active coordinates to data address in memory. Suitable for high sparsity.

**dynamic**: Variable-length array, with a predefined maximum length. It serves the role of `std::vector`, and can be used to maintain objects (e.g. particles) contained by a block.

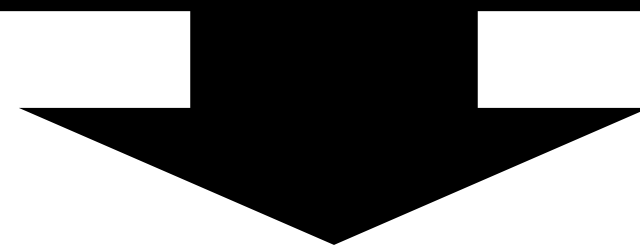
**morton**: Reorder the data in memory using a Z-order curve (Morton coding), for potentially higher spatial locality. For **dense** only.

**bitmasked**: Use a mask to maintain sparsity information, one bit per child. For **dense** only.

**pointer**: Store pointers instead of the whole structure to save memory and maintain sparsity. For **dense** and **dynamic**.

Structural Nodes

Node Decorators



# Describing Data Structures

**dense**: A fixed-length contiguous array.

**hash**: Use a hash table to maintain the mapping from active coordinates to data address in memory. Suitable for high sparsity.

**dynamic**: Variable-length array, with a predefined maximum length. It serves the role of `std::vector`, and can be used to maintain objects (e.g. particles) contained by a block.

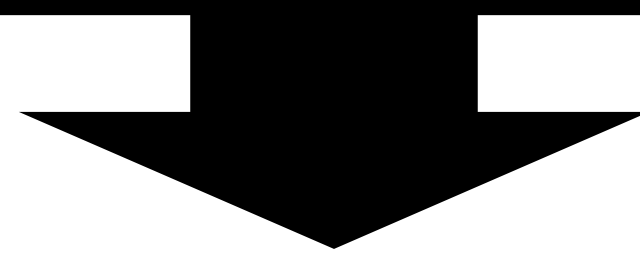
**morton**: Reorder the data in memory using a Z-order curve (Morton coding), for potentially higher spatial locality. For **dense** only.

**bitmasked**: Use a mask to maintain sparsity information, one bit per child. For **dense** only.

**pointer**: Store pointers instead of the whole structure to save memory and maintain sparsity. For **dense** and **dynamic**.

## Structural Nodes

## Node Decorators



```
root.hash(ijk, 32).dense(ijk, 16).pointer()  
  .dense(ijk, 8).place(u, v, w);
```

VDB [Museth 2013]

```
root.dense(ijk, 512).morton().bitmasked()  
  .dense(ijk, {8, 4, 4}).place(flags, u, v, w);
```

SPGrid [Setaluri et al. 2014]

# Describing Data Structures

**dense**: A fixed-length contiguous array.

**hash**: Use a hash table to maintain the mapping from active coordinates to data address in memory. Suitable for high sparsity.

**dynamic**: Variable-length array, with a predefined maximum length. It serves the role of `std::vector`, and can be used to maintain objects (e.g. particles) contained by a block.

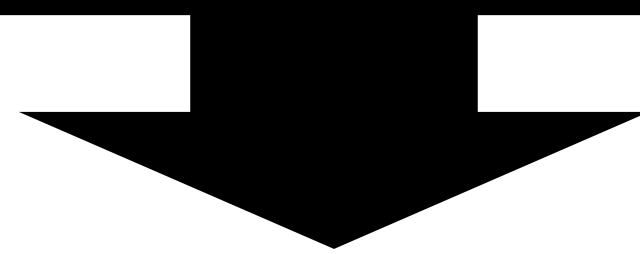
**morton**: Reorder the data in memory using a Z-order curve (Morton coding), for potentially higher spatial locality. For **dense** only.

**bitmasked**: Use a mask to maintain sparsity information, one bit per child. For **dense** only.

**pointer**: Store pointers instead of the whole structure to save memory and maintain sparsity. For **dense** and **dynamic**.

Structural Nodes

Node Decorators



```
root.hash(ijk, 32).dense(ijk, 16).pointer()  
  .dense(ijk, 8).place(u, v, w);
```

VDB [Museth 2013]

```
root.dense(ijk, 512).morton().bitmasked()  
  .dense(ijk, {8, 4, 4}).place(flags, u, v, w);
```

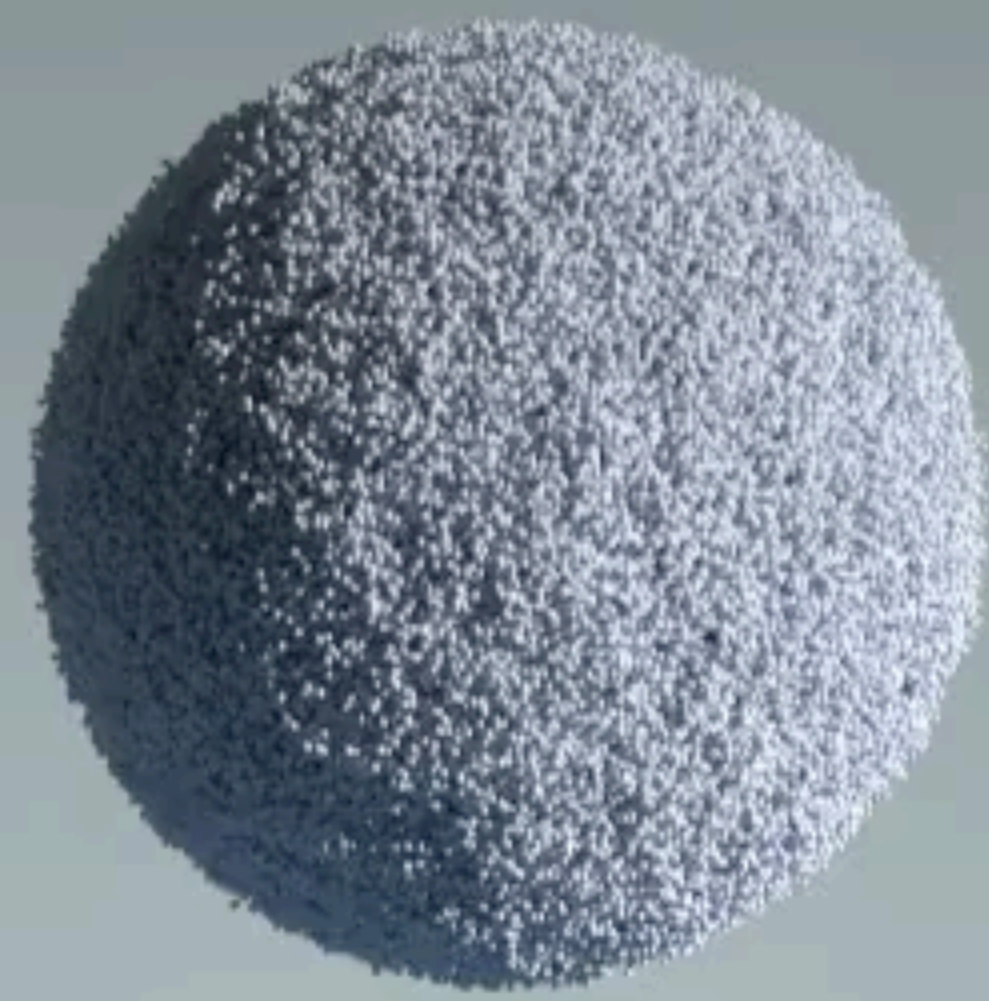
SPGrid [Setaluri et al. 2014]

“SPVDB”

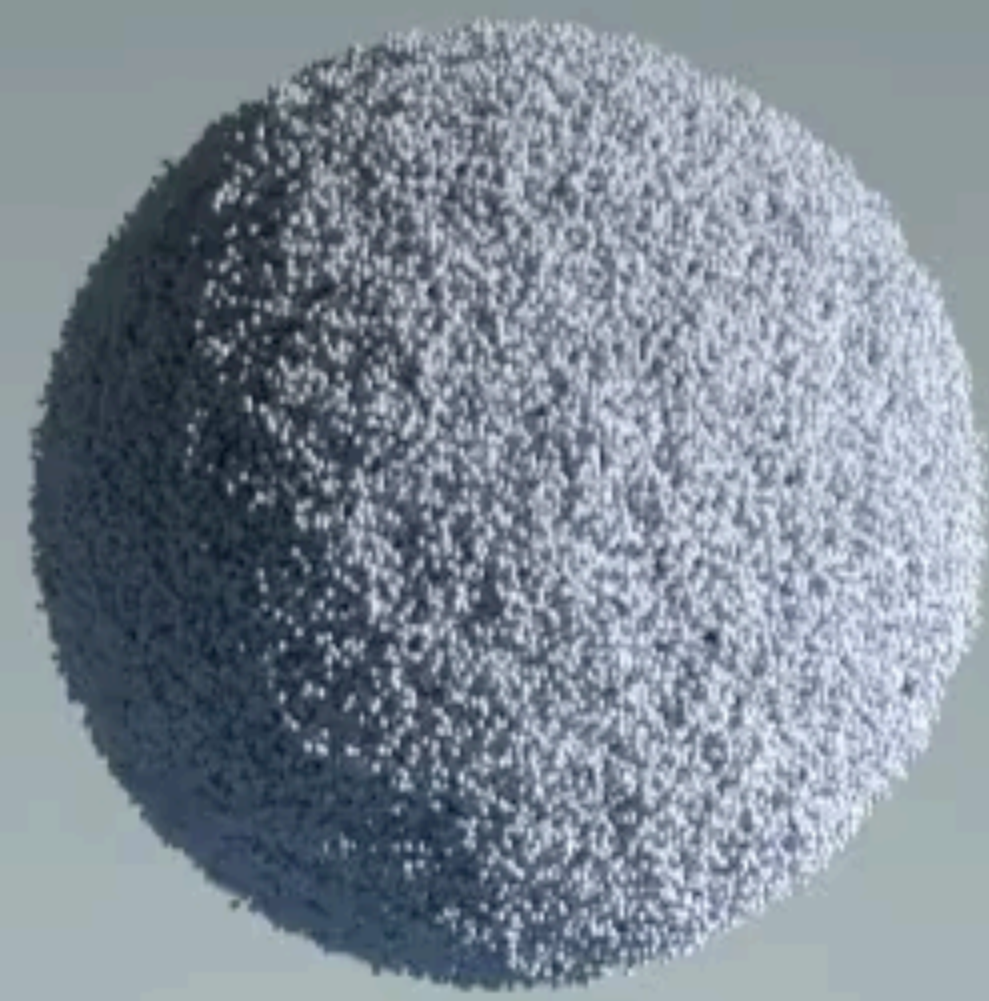
```
root.hash(ijk, 512).dense(ijk, 512).morton().bitmasked().dense(ijk, {8, 4, 4}).place(flags, u, v, w);
```



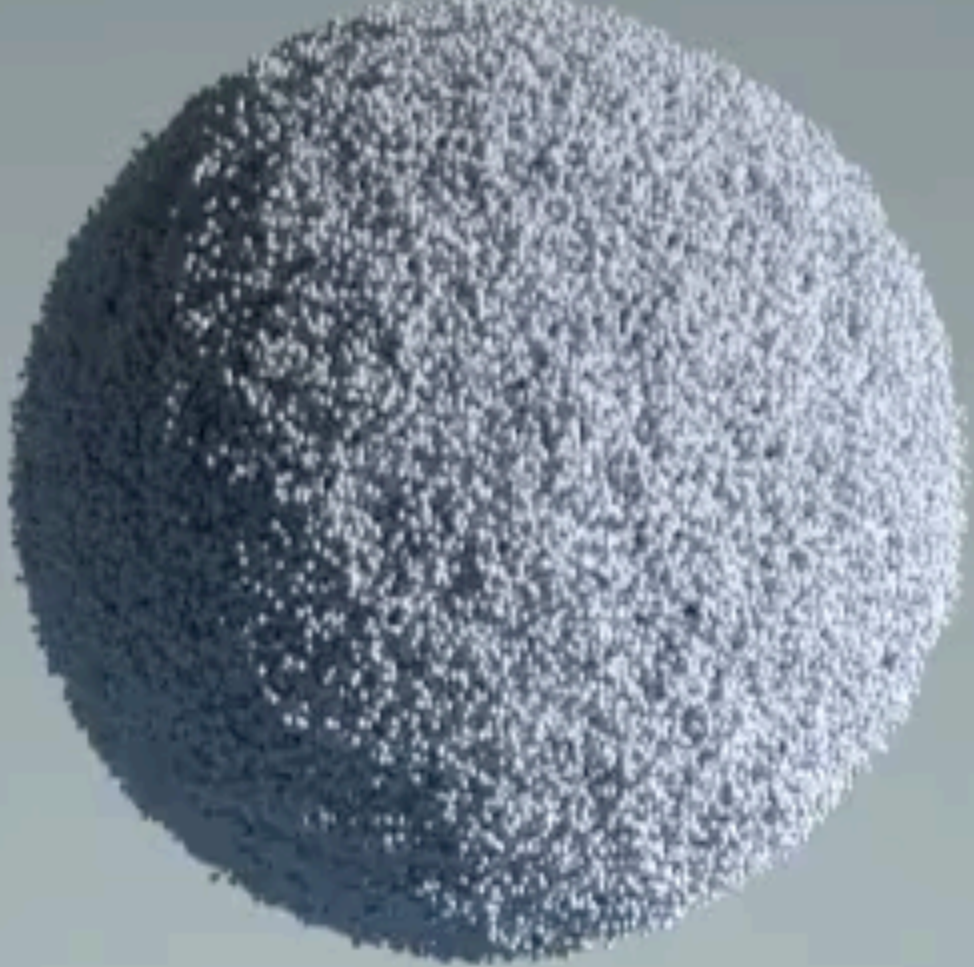
Bounded sparse grid structure



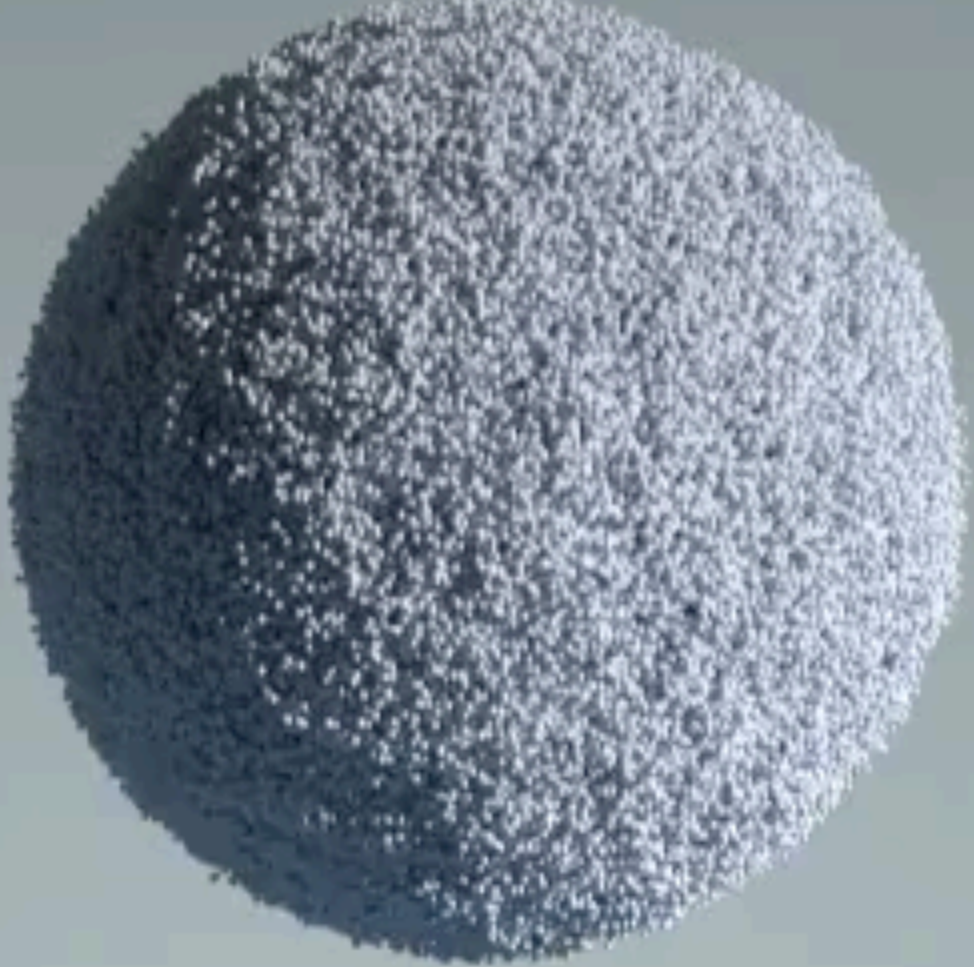
Bounded sparse grid structure



Unbounded sparse grid structure



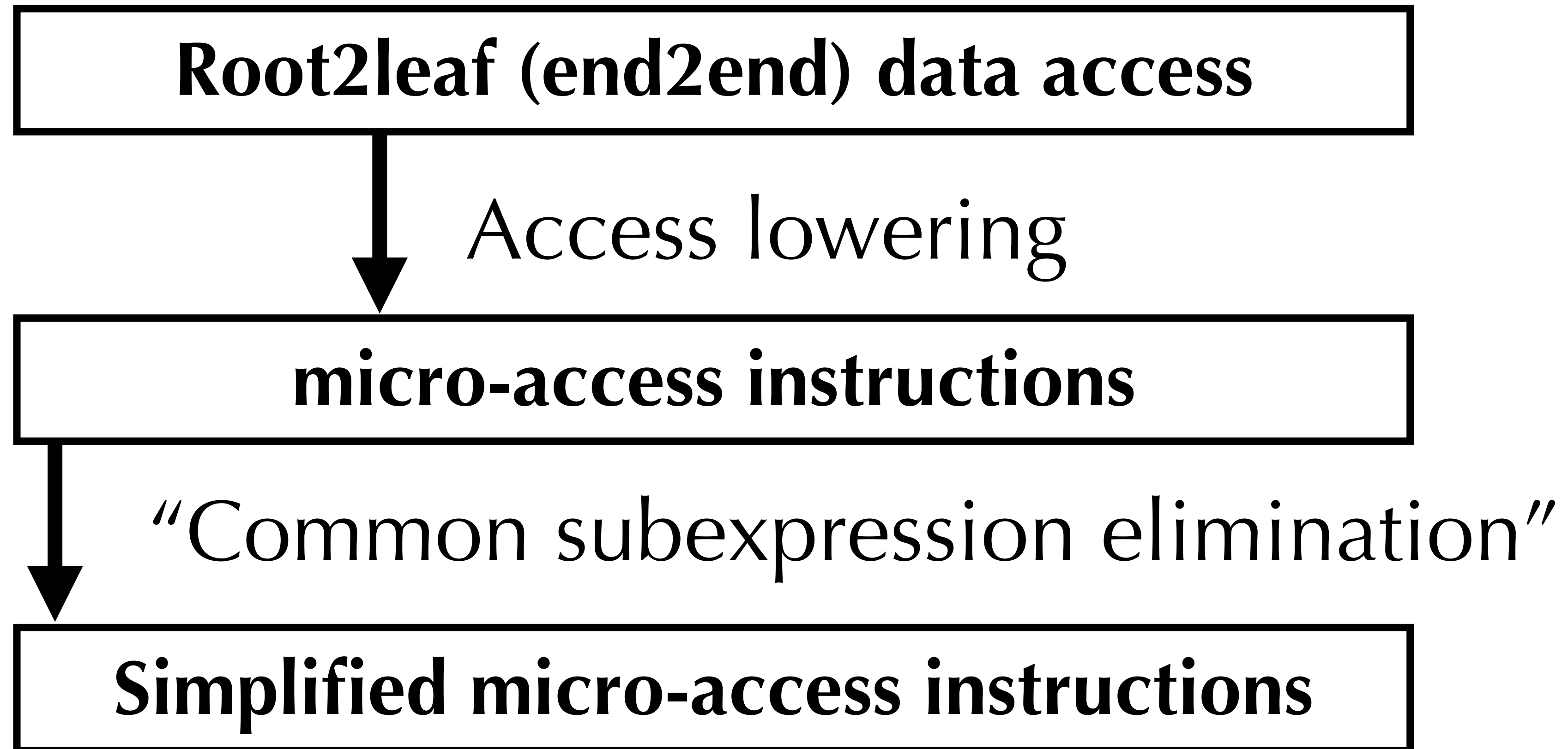
Unbounded sparse grid structure



# **Access Simplification**

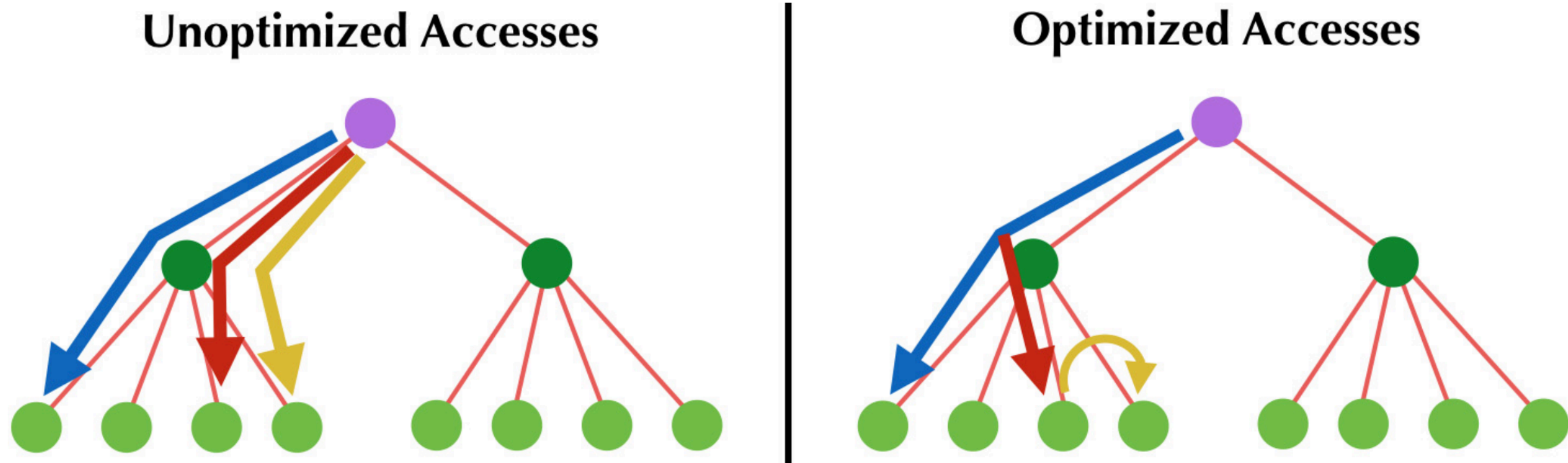
based on computation and data structure info

# Access Simplification



# Access Simplification

Removing redundant data structure traversals



## More optimizations:

- shared memory utilization on GPUs;
- avoid unnecessary activation checks;
- better vectorized loads on CPUs;
- ...

3.02x faster

# Results

10.0x shorter code

4.55x higher performance

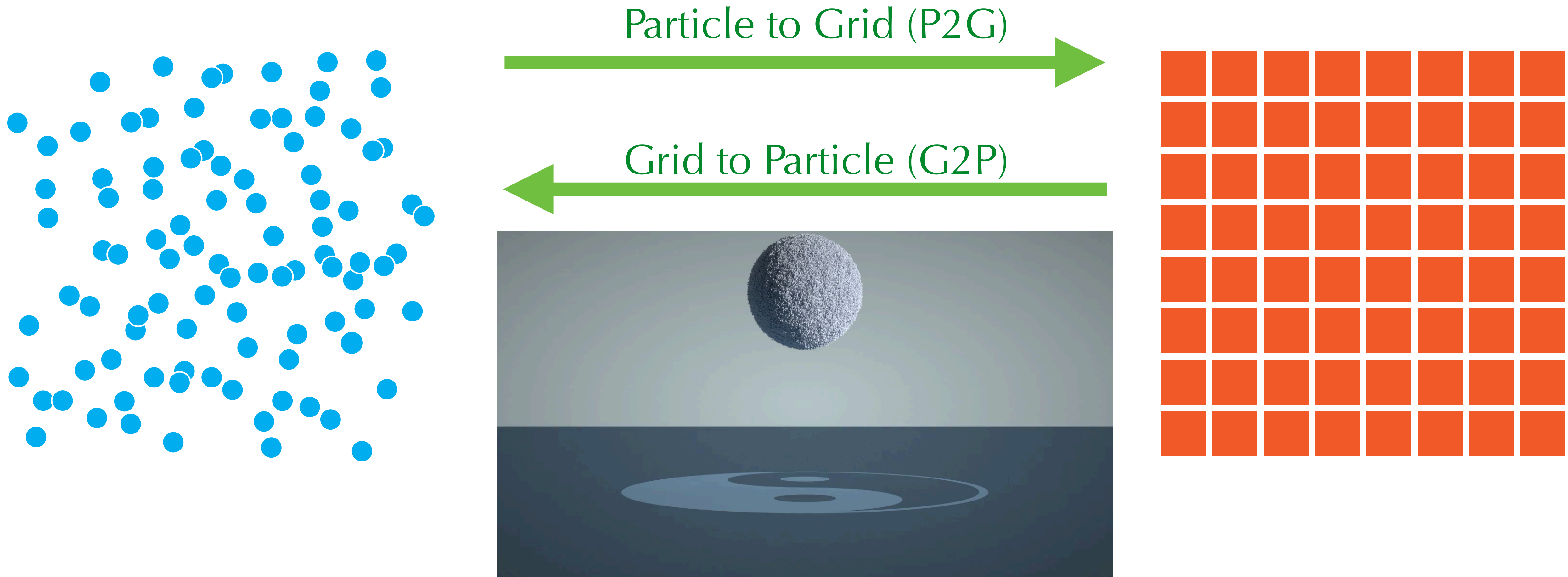
## High-Performance CPU/GPU Kernels

Ours v.s. State-of-the-art:

<b>MLS-MPM</b>	13x shorter code, 1.2x faster
<b>FEM Kernel</b>	13x shorter code, 14.5x faster
<b>MGPCG</b>	7x shorter code, 1.9x faster
<b>Sparse CNN</b>	9x shorter code, 13x faster

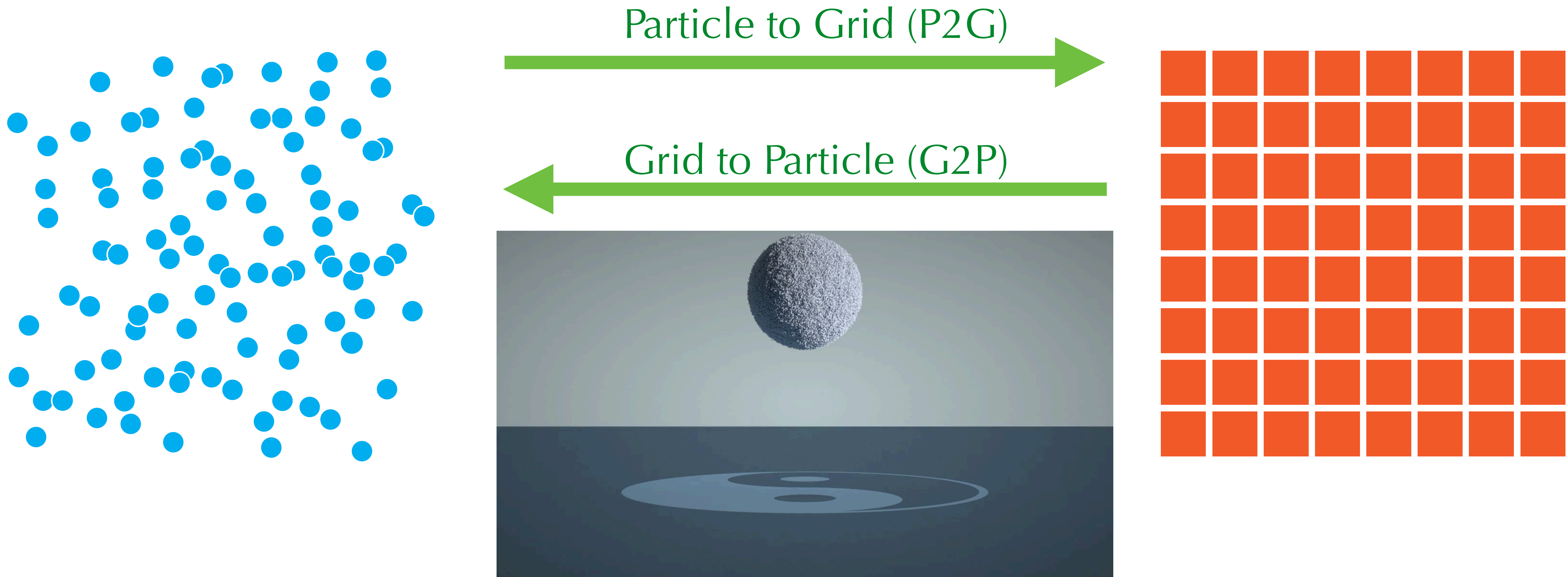


# Benchmarks: *MLS-MPM*



**Patterns: Particle scatter/gather**

# Benchmarks: *MLS-MPM*



**Patterns: Particle scatter/gather**

# Benchmarks: *MLS-MPM*

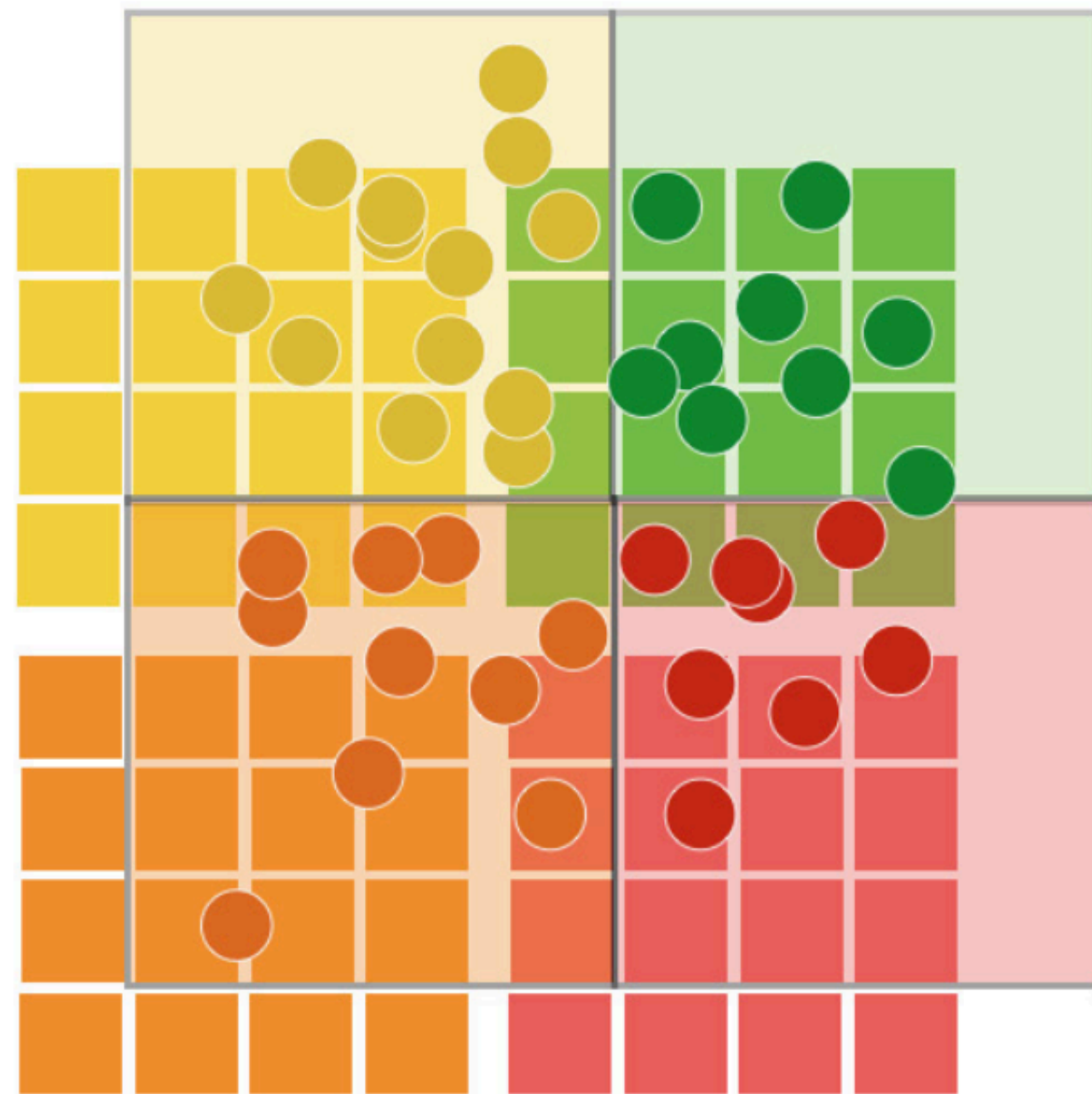
Particle Layout	Ordered	Randomly Shuffled
SOA	3.52ms	21.23 ms
AOS	3.15ms	4.28 ms

**AOS much faster than SOA for random access!  
No sorting needed.**

**Reproduce:** `ti mpm_benchmark particle_soa=[true/false] initial_shuffle=[true/false]`

# Benchmarks: MLS-MPM

The use of scratch pad memory [NVIDIA shared memory]



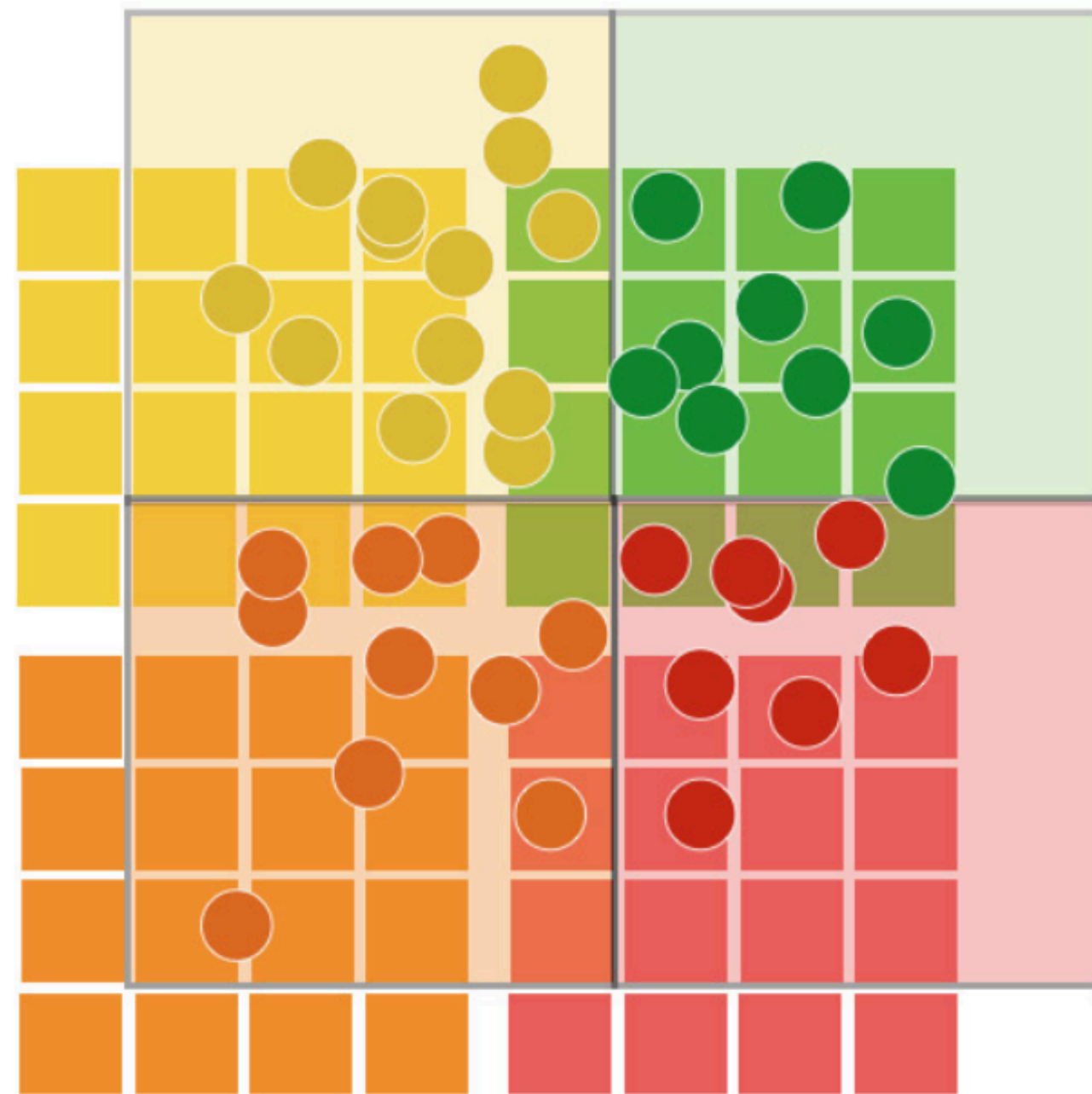
Group particles into blocks  
& scatter/gather

	GPU-SPM	GPU+SPM
P2G	5.102ms	2.011ms
G2P	1.975ms	0.722ms

**Reproduce:** `ti mpm_benchmark use_cache=[true/false]`

# Benchmarks: MLS-MPM

The use of scratch pad memory [NVIDIA shared memory]



Group particles into blocks  
& scatter/gather

	GPU-SPM	GPU+SPM
P2G	5.102ms	2.011ms
G2P	1.975ms	0.722ms

**2-3x faster using  
shared memory**

Index analysis for  
scratchpad memory  
size inference

**Reproduce:** `ti mpm_benchmark use_cache=[true/false]`

# Benchmarks: *MLS-MPM*

Compared with baseline [Gao et al.]:

**[GPU] 1.2x faster**

**13x shorter code**

**Reproduce:** `ti mpm_benchmark particle_soa=[true/false] initial_shuffle=[true/false]`

# Benchmarks: FEM Kernel

$$\underline{\mathbf{f}} = \sum_{c \in \mathcal{C}(i)} \sum_{j \in \mathcal{V}(c)} (\underline{\mu}^{(c)} \cdot \mathbf{K}^{\mu} + \underline{\lambda}^{(c)} \cdot \mathbf{K}^{\lambda})_{i^{(c)}j^{(c)}} \cdot \underline{\mathbf{u}}_j.$$

# Benchmarks: FEM Kernel

3 channels

$$\underline{\mathbf{f}} = \sum_{c \in \mathcal{C}(i)} \sum_{j \in \mathcal{V}(c)} (\underline{\mu}^{(c)} \cdot \mathbf{K}^{\mu} + \underline{\lambda}^{(c)} \cdot \mathbf{K}^{\lambda})_{i(c)j(c)} \cdot \underline{\mathbf{u}}_j.$$

3 channels

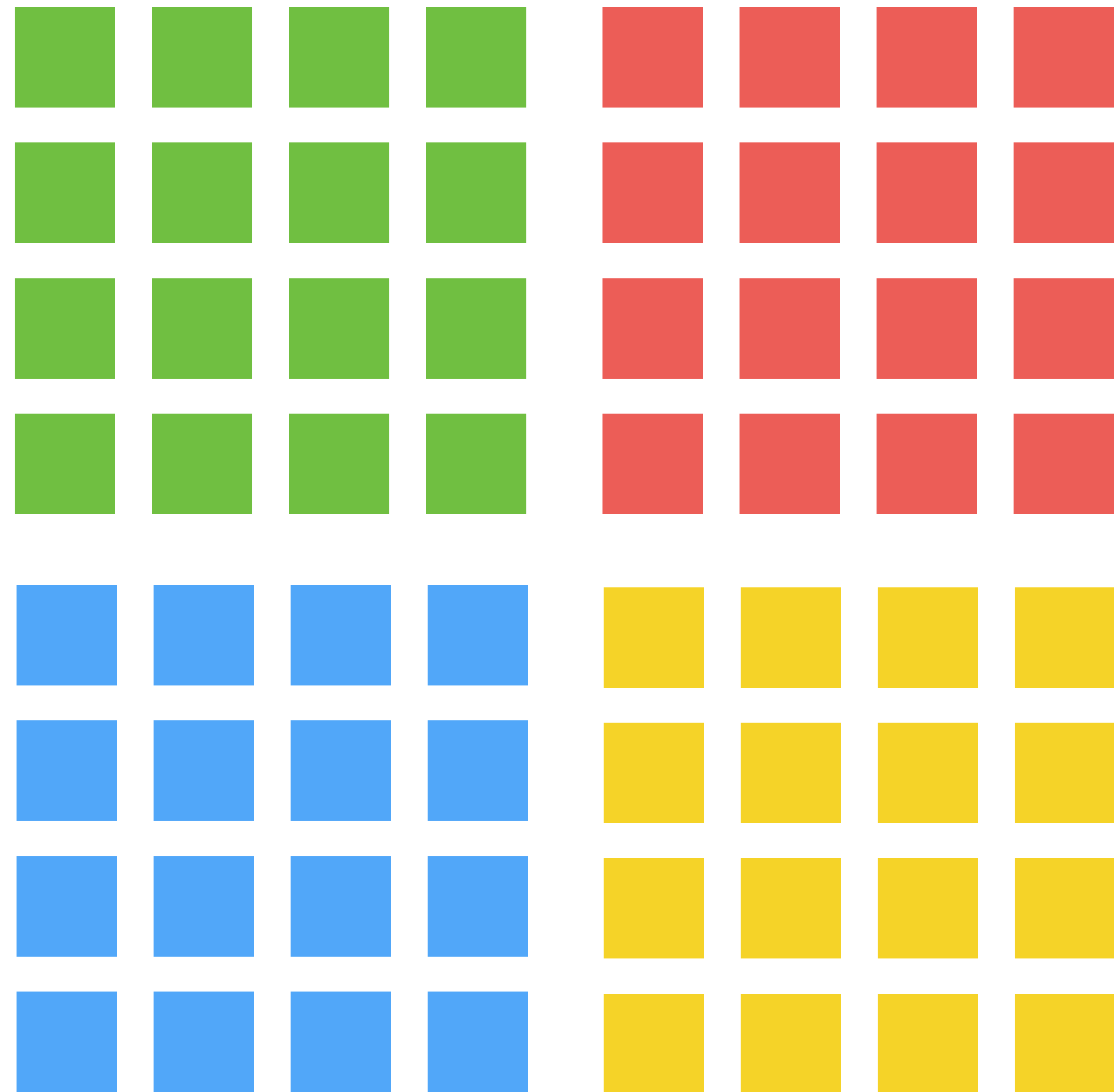
8 elements 8 elements

**3x8x8x3x2=1152 FLOPs/vertex**

**Patterns: Stencils with very high arithmetic intensity (compute bound)**

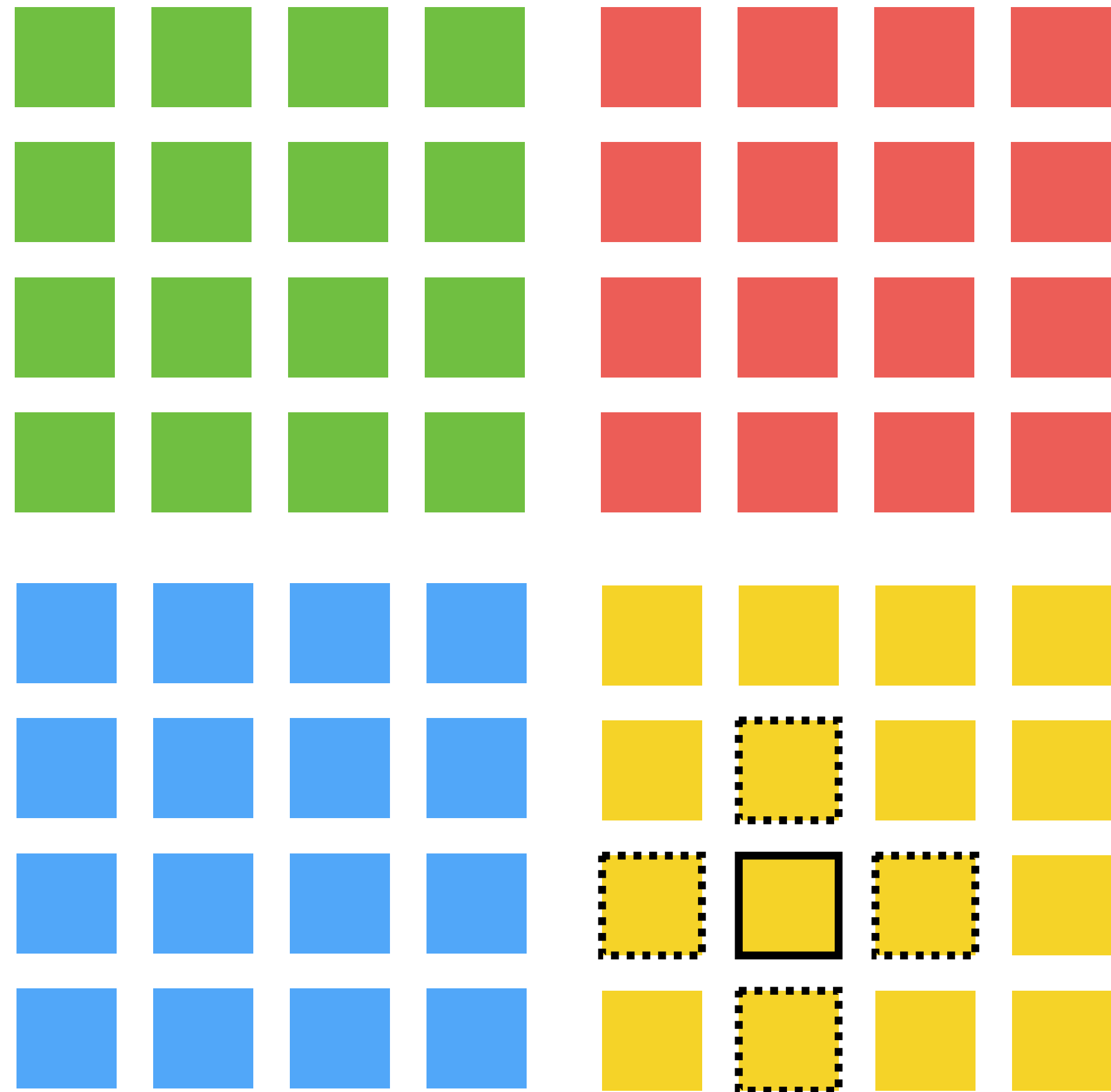


# Benchmarks: FEM Kernel



4x4-Blocked  
Sparse Grid

# Benchmarks: FEM Kernel



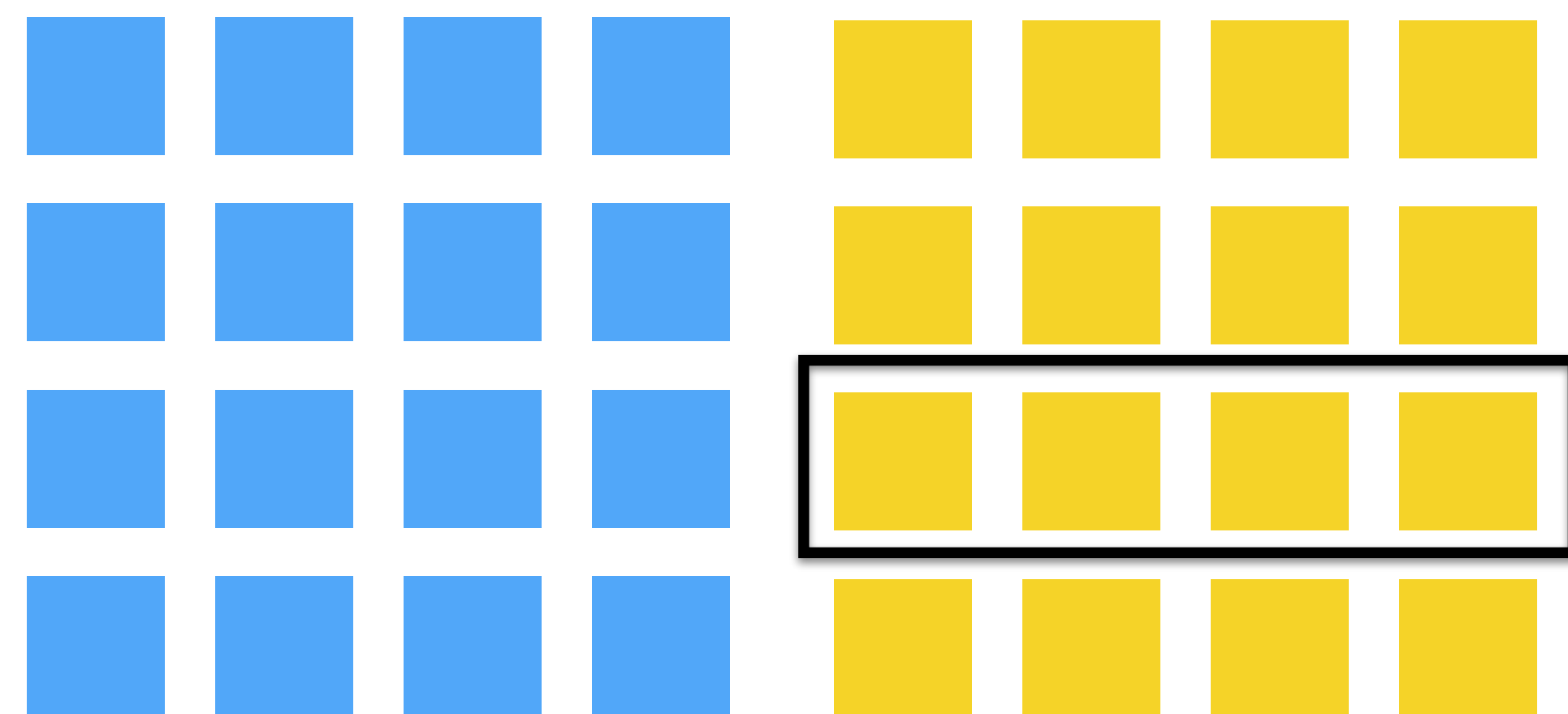
**5-Point Stencil  
(Scalar)**

(Simplified: actual  
stencil is much larger)

# Benchmarks: FEM Kernel

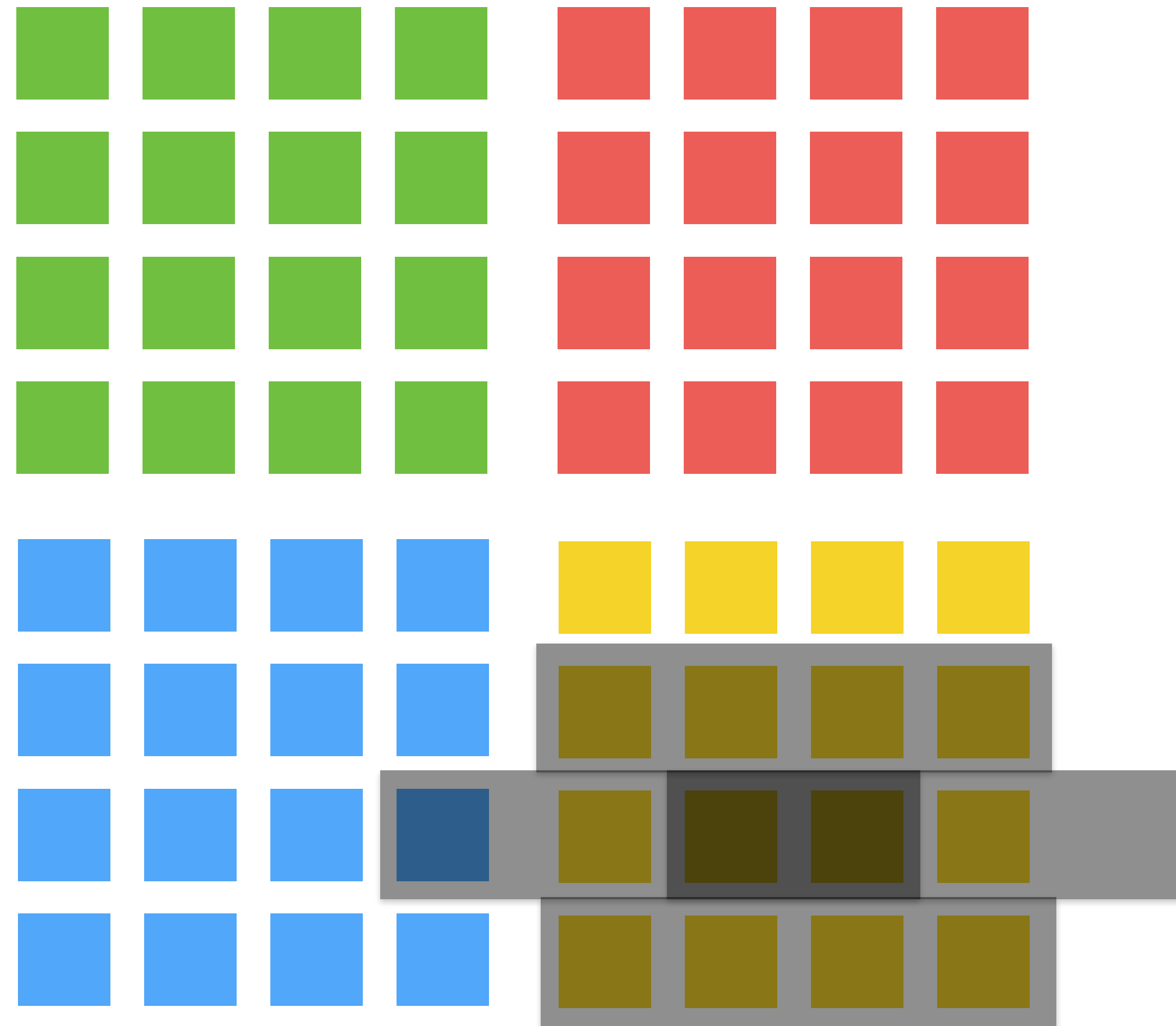


**5-Point Stencil  
(4-wide Vectorized)**



**Taichi compiler merges  
4 addressing into  
1, and then do a vectorized load  
(more details later)**

# Benchmarks: FEM Kernel



**5-Point Stencil  
(4-wide Vectorized)**

# Benchmarks: FEM Kernel

Ablation	CPU Time	GPU Time
No multithreading	73.43ms	-
No vectorization	83.54ms	-
No vectorized load instructions	22.69ms	-
No simplification I	17.01ms	2.13 ms
No access lowering	182.19ms	6.046 ms
No simplification II	85.51ms	11.784 ms
AOS instead of SOA	136.03ms	20.992 ms
All optimizations on	17.16ms	2.11 ms

**Reproduce:** ti fem gpu=[t/f] simp1=[t/f] vec=[t/f] threads=[1-8] lower\_access=[t/f] simp2=[t/f] vec\_load\_cpu=[t/f] block\_soa=[t/f]

# Benchmarks: FEM Kernel

Ablation	CPU Time	GPU Time
No multithreading	73.43ms	-
No vectorization	83.54ms	-
No vectorized load instructions	22.69ms	-
No simplification I	17.01ms	2.13 ms
<b>No access lowering</b>	<b>182.19ms</b>	<b>6.046 ms</b>
No simplification II	85.51ms	11.784 ms
AOS instead of SOA	136.03ms	20.992 ms
<b>All optimizations on</b>	<b>17.16ms</b>	<b>2.11 ms</b>

**Without access lowering,  
the backend compiler  
(gcc/clang/nvcc)  
fails to discover  
potential vectorized loads  
and reduce data access**

**Reproduce:** ti fem gpu=[t/f] simp1=[t/f] vec=[t/f] threads=[1-8] lower\_access=[t/f] simp2=[t/f] vec\_load\_cpu=[t/f] block\_soa=[t/f]

# Benchmarks: FEM Kernel

Ablation	CPU Time	GPU Time
No multithreading	73.43ms	-
No vectorization	83.54ms	-
No vectorized load instructions	22.69ms	-
No simplification I	17.01ms	2.13 ms
No access lowering	182.19ms	6.046 ms
No simplification II	85.51ms	11.784 ms
<b>AOS instead of SOA</b>	<b>136.03ms</b>	<b>20.992 ms</b>
<b>All optimizations on</b>	<b>17.16ms</b>	<b>2.11 ms</b>

**AOS is really bad in this case since**

- 1) no vectorized ld/st
- 2) low cacheline util.

**Reproduce:** ti fem gpu=[t/f] simp1=[t/f] vec=[t/f] threads=[1-8] lower\_access=[t/f] simp2=[t/f] vec\_load\_cpu=[t/f] block\_soa=[t/f]

# Benchmarks: FEM Kernel

Ablation	CPU Time	GPU Time
No multithreading	73.43ms	-
No vectorization	83.54ms	-
No vectorized load instructions	22.69ms	-
No simplification I	17.01ms	2.13 ms
No access lowering	182.19ms	6.046 ms
No simplification II	85.51ms	11.784 ms
AOS instead of SOA	136.03ms	20.992 ms
All optimizations on	17.16ms	2.11 ms

**Compared with baseline:  
[Vectorized CPU] 2x faster  
[GPU] 14.5x faster  
13x shorter code**

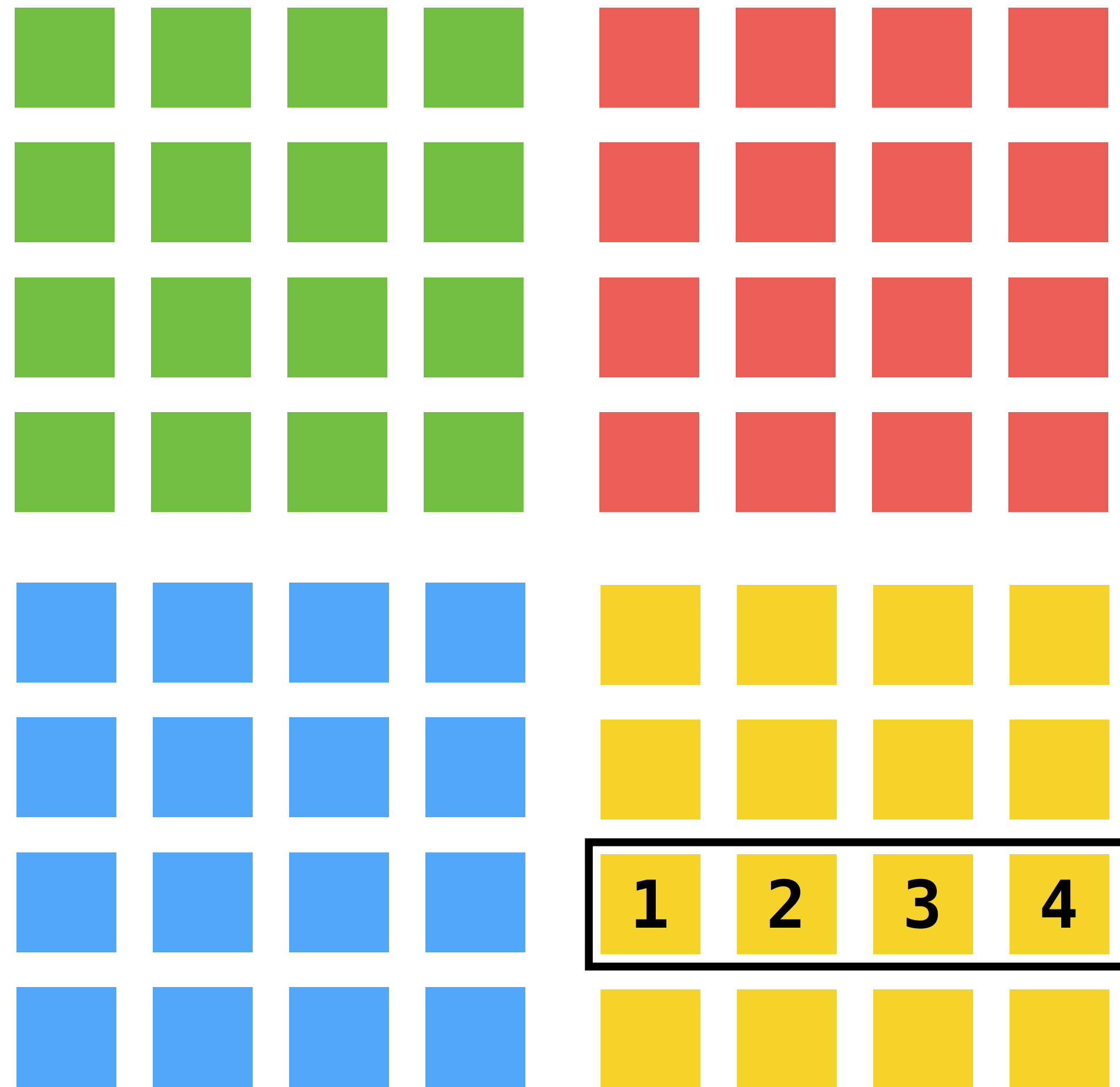
Baseline: (handwritten AVX2)

**Liu, Hu, Zhu, Matusik, and Sifakis:  
Narrow-band Topology Optimization  
on a Sparsely Populated Grid**

**Reproduce:** ti fem gpu=[t/f] simp1=[t/f] vec=[t/f] threads=[1-8] lower\_access=[t/f] simp2=[t/f] vec\_load\_cpu=[t/f] block\_soa=[t/f]



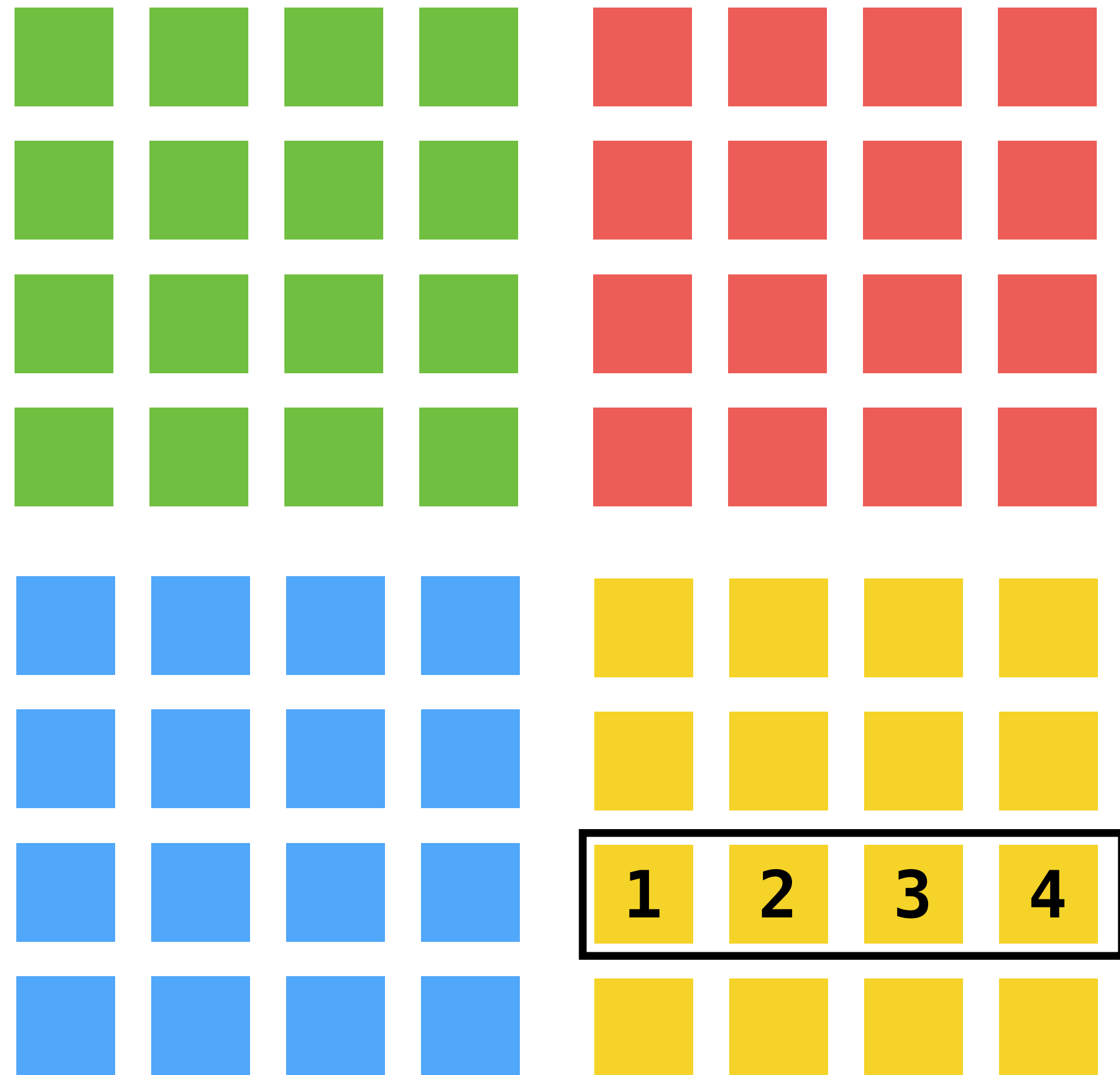
# Vectorized FEM Access Optimization



**Initial IR**

```
for i in range(0, n, step 4):  
    %1 = load voxel 1 from root  
    %2 = load voxel 2 from root  
    %3 = load voxel 3 from root  
    %4 = load voxel 4 from root  
    %9 = make_vector(%1,%2,%3,%4)
```

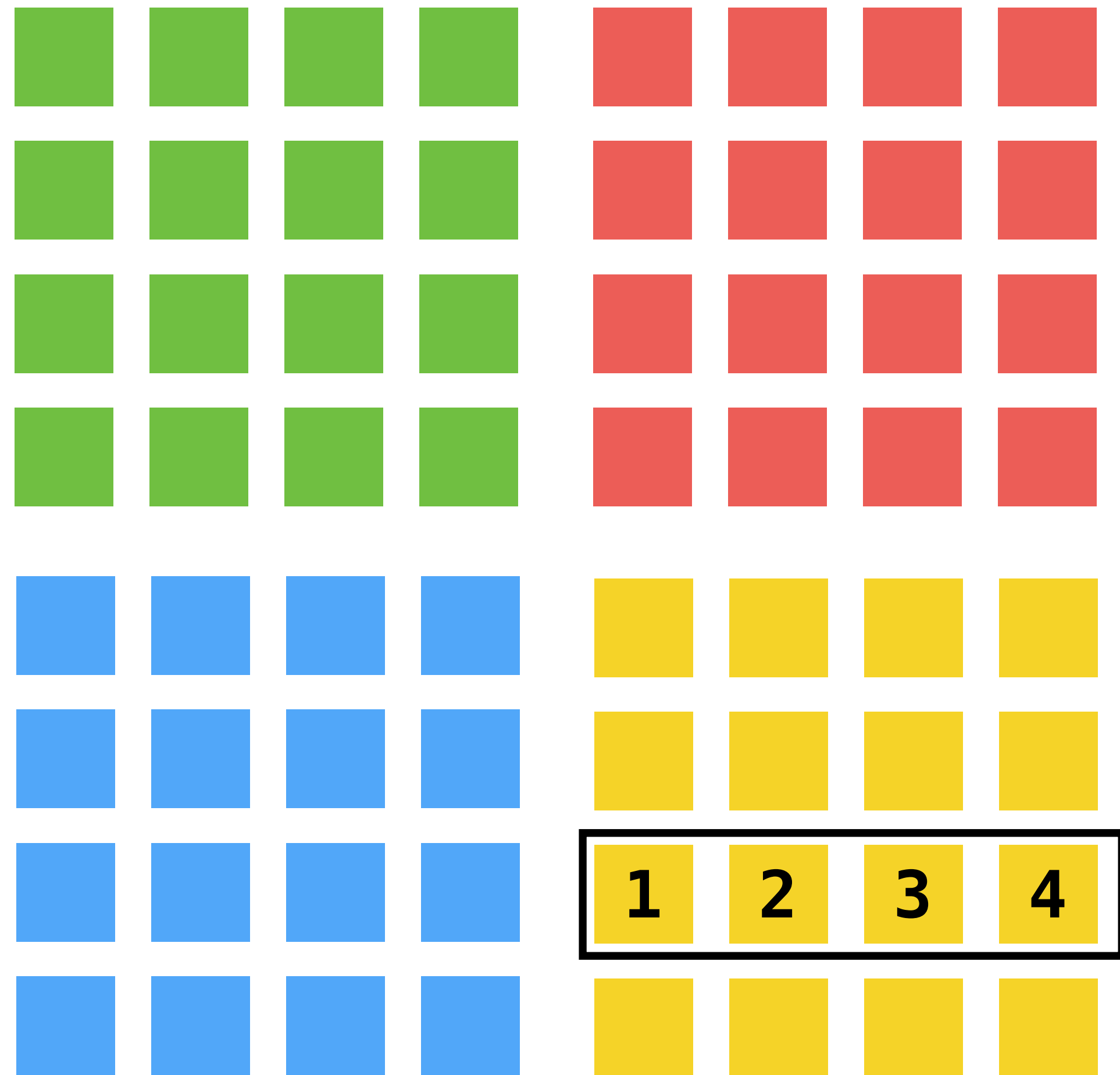
# Vectorized FEM Access Optimization



**After access lowering:**

```
for i in range(0, n, step 4):  
    %1 = get_block for voxel 1  
    %2 = get_voxel 1 from %1  
    %3 = get_block for voxel 2  
    %4 = get_voxel 2 from %3  
    %5 = get_block for voxel 3  
    %6 = get_voxel 3 from %5  
    %7 = get_block for voxel 4  
    %8 = get_voxel 4 from %7  
    %9 = make_vector(%2,%4,%6,%8)
```

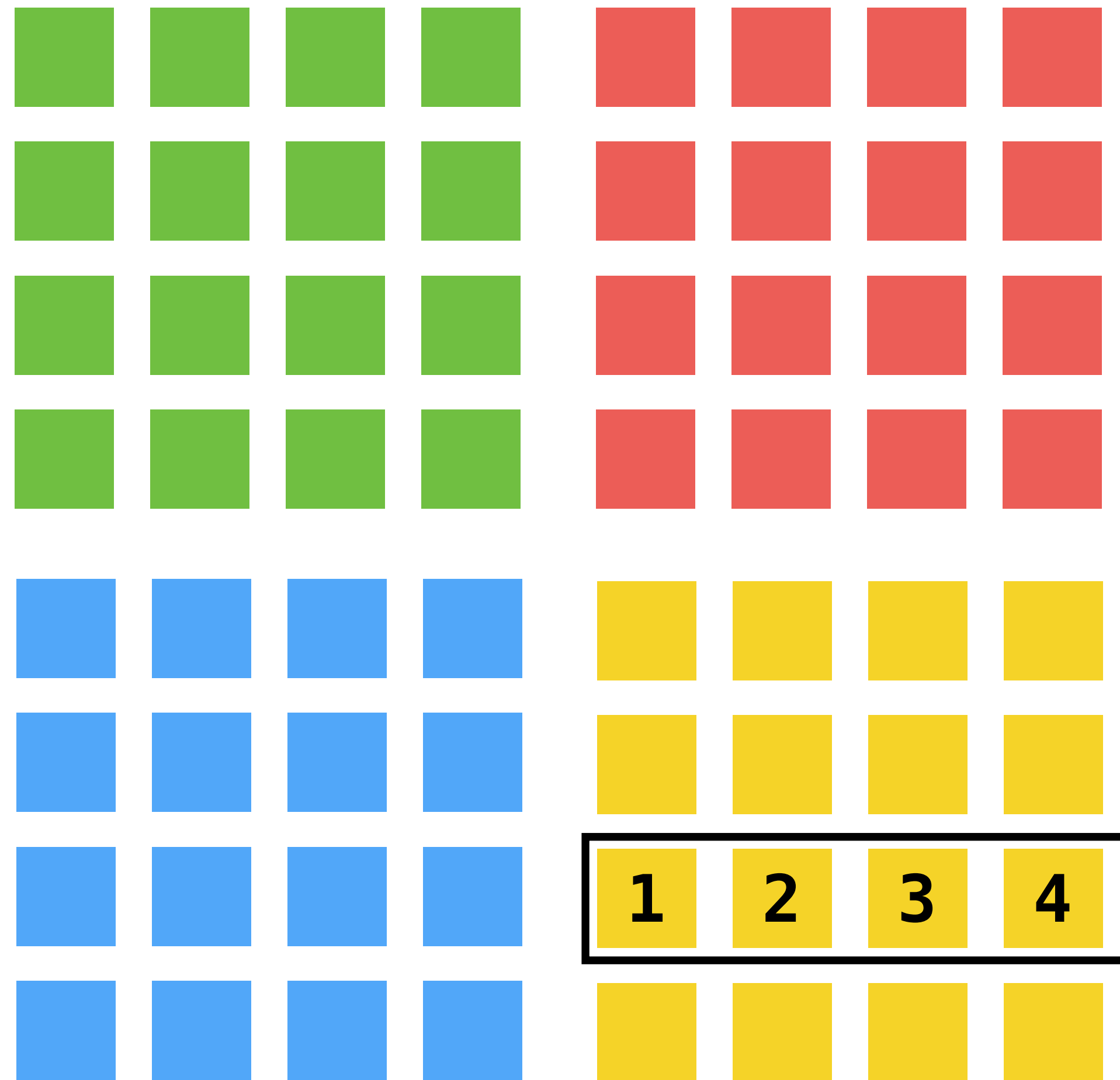
# Vectorized FEM Access Optimization



## Index analysis

```
for i in range(0, n, step 4):  
    %1 = get_block for voxel  $i+0$   
    %2 = get voxel  $i+0$  from %1  
    %3 = get_block for voxel  $i+1$   
    %4 = get voxel  $i+1$  from %3  
    %5 = get_block for voxel  $i+2$   
    %6 = get voxel  $i+2$  from %5  
    %7 = get_block for voxel  $i+3$   
    %8 = get voxel  $i+3$  from %7  
    %9 = make_vector(%2,%4,%6,%8)
```

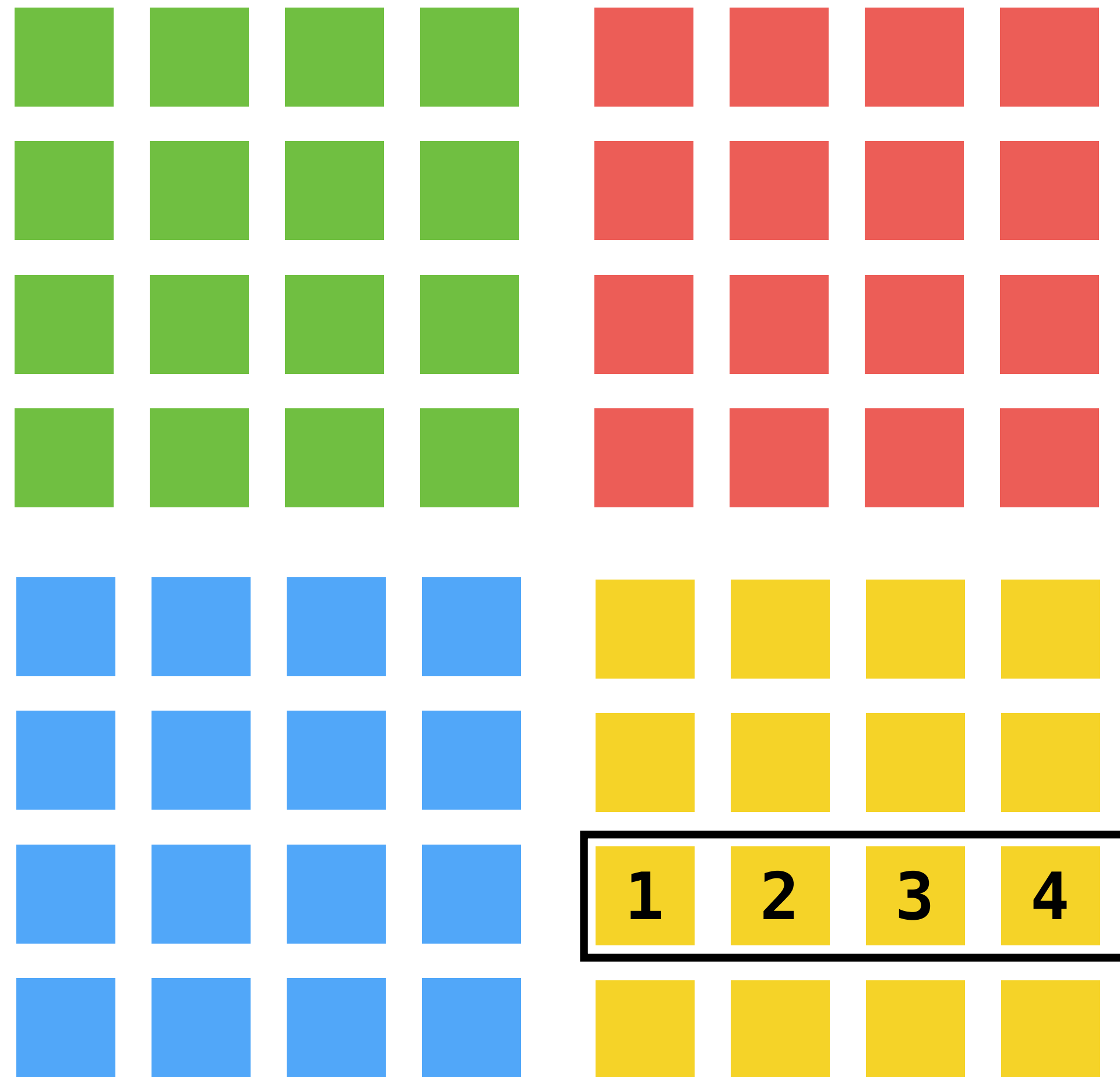
# Vectorized FEM Access Optimization



With data structure info (block size=16)

```
for i in range(0, n, step 4):  
    %1 = get block (i+0)/16  
    %2 = get voxel i+0 from %1  
    %3 = get block (i+1)/16  
    %4 = get voxel i+1 from %3  
    %5 = get block (i+2)/16  
    %6 = get voxel i+2 from %5  
    %7 = get block (i+3)/16  
    %8 = get voxel i+3 from %7  
    %9 = make_vector(%2,%4,%6,%8)
```

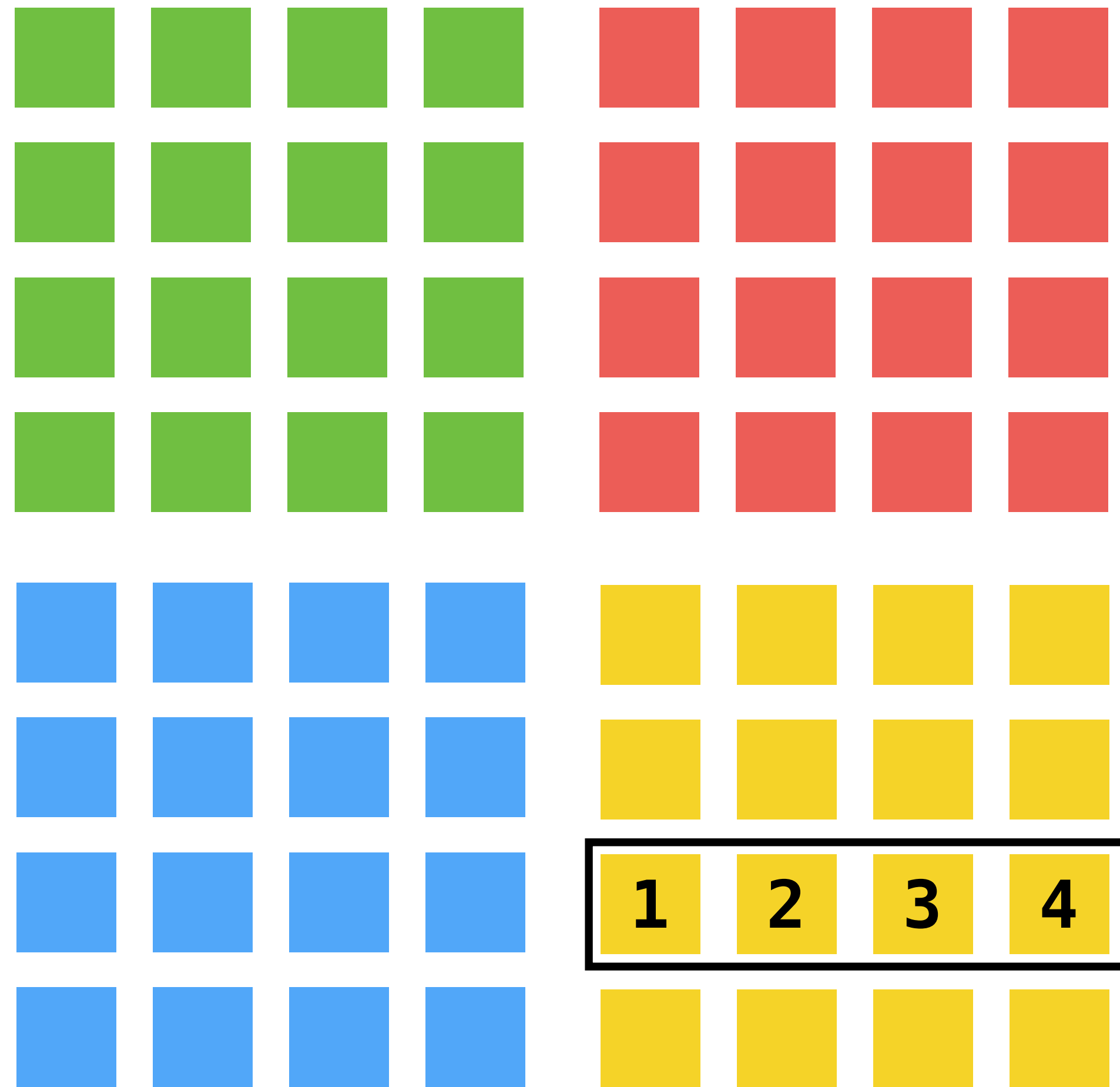
# Vectorized FEM Access Optimization



Index analysis ( $i \% 4 == 0$ ) & integer division property:

```
for i in range(0, n, step 4):  
    %1 = get block  $(i+0)/16$   
    %2 = get voxel  $i+0$  from %1  
    %3 = get block  $(i+0)/16$   
    %4 = get voxel  $i+1$  from %3  
    %5 = get block  $(i+0)/16$   
    %6 = get voxel  $i+2$  from %5  
    %7 = get block  $(i+0)/16$   
    %8 = get voxel  $i+3$  from %7  
    %9 = make_vector(%2,%4,%6,%8)
```

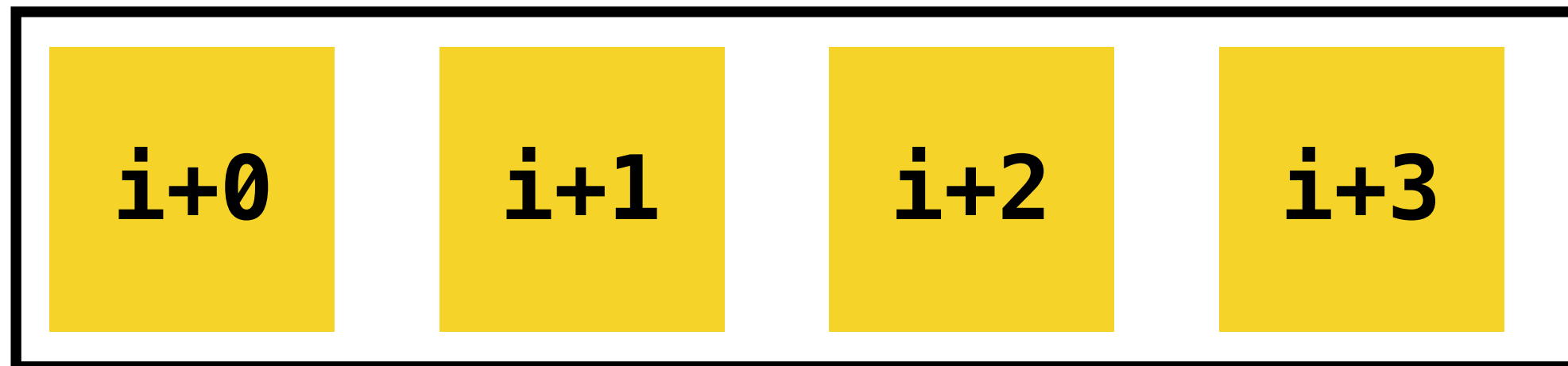
# Vectorized FEM Access Optimization



**Index analysis ( $i \% 4 == 0$ ) & simplification**

```
for i in range(0, n, step 4):  
    %1 = get_block (i+0)/16  
    %2 = get_voxel i+0 from %1  
    %4 = get_voxel i+1 from %1  
    %6 = get_voxel i+2 from %1  
    %8 = get_voxel i+3 from %1  
    %9 = make_vector(%2,%4,%6,%8)
```

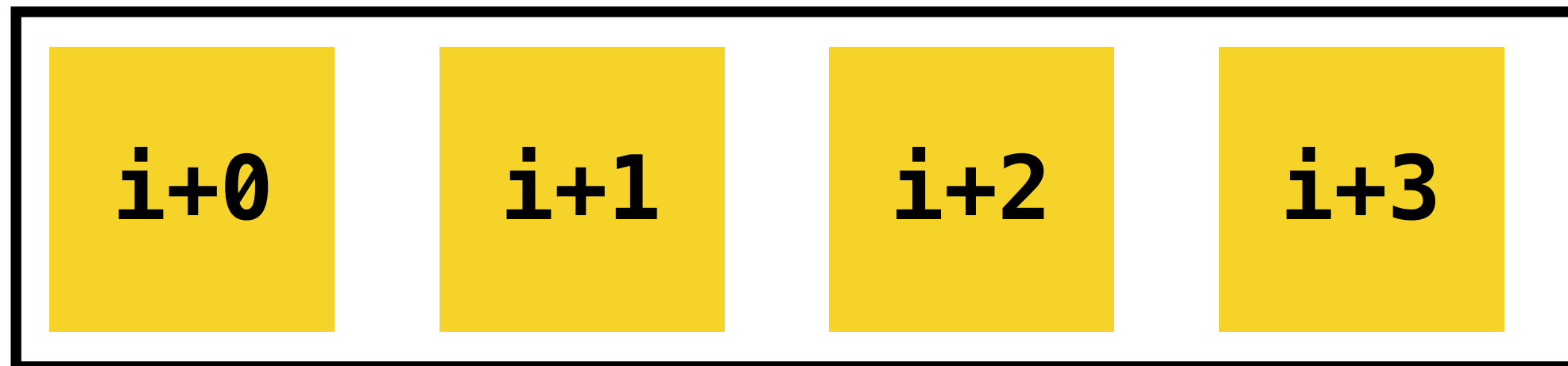
# Vectorized FEM Access Optimization



## Index analysis + data structure info

```
for i in range(0, n, step 4):  
    %1 = get_block i/16  
    %2 = get_voxel i+0 within %1  
    %3 = get_1st_voxel_right_to %2  
    %4 = get_2nd_voxel_right_to %2  
    %5 = get_3rd_voxel_right_to %2  
    %9 = make_vector(%2,%3,%4,%5)
```

# Vectorized FEM Access Optimization



**Index analysis + data structure info**

```
for i in range(0, n, step 4):  
    %1 = get block i/16  
    %2 = get voxel i within %1  
    %3 = vector_load(%2, width=4)
```



# Reasons for Performance

Why can't traditional compilers do the optimizations?

- 1) Index analysis**
- 2) Instruction granularity**
- 3) Data access semantics**

# The Granularity Spectrum

**x[i, j]**

**access1(i, j)**  
**access2(i, j)**

```

$4 = [S4][root]::lookup(root, $3) coord = {$2}
    activate = false
$5 = get child [S4->S3] $4
$6 = bit_extract($2 + 0, 7~14)
$7 = linearized(ind {$6}, stride {128})
$8 = [S3][dense]::lookup($5, $7) coord = {$2} a
    = false
$9 = get child [S3->S2] $8
$10 = bit_extract($2 + 0, 0~7)
$11 = linearized(ind {$10}, stride {128})
$12 = [S2][dense]::lookup($9, $11) coord = {$2}
    activate = false
    
```

```

%63 = lshr i32 %62, 0
%64 = and i32 %63, 255
%65 = add i32 %37, 0
%66 = lshr i32 %65, 0
%67 = and i32 %66, 255
%68 = add i32 0, %64
%69 = mul i32 %68, 256
%70 = add i32 %69, %67
%71 = bitcast %struct.DenseMeta* %5 to
call void @StructMeta_set_snode_id(%s
call void @StructMeta_set_element_size
call void @StructMeta_set_max_num_eler
call void @StructMeta_set_lookup_eleme
    _element)
call void @StructMeta_set_is_active(%s
call void @StructMeta_set_get_num_eler
    elements)
call void @StructMeta_set_from_parent_
    
```

```

movl $0, %eax
addl %eax, %ebx
popl %eax
looptop:
imul %edx
andl $0xFF, %eax
cmpl $100, %eax
jb looptop
leal 4(%esp), %ebp
movl %esi, %edi
subl $8, %edi
shrl %cl, %ebx
movw %bx, -2(%ebp)
movl $0, %eax
addl %eax, %ebx
popl %eax
looptop:
imul %edx
andl $0xFF, %eax
cmpl $100, %eax
jb looptop
leal 4(%esp), %ebp
movl %esi, %edi
subl $8, %edi
shrl %cl, %ebx
movw %bx, -2(%ebp)
    
```

**End2end access**

**Level-wise Access**

**Taichi IR**

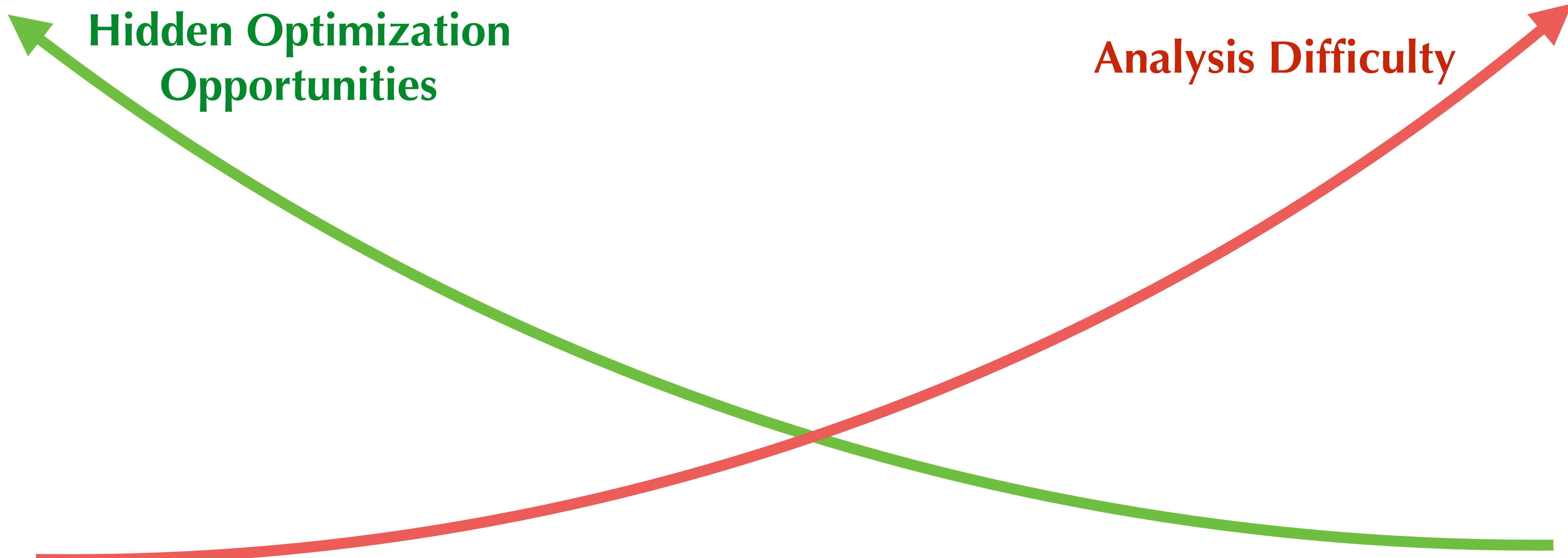
**LLVM IR**

**Machine code**



**Coarser**

**Finer**



**Hidden Optimization Opportunities**

**Analysis Difficulty**

**End2end access**

**Level-wise Access**

**Taichi IR**

**LLVM IR**

**Machine code**

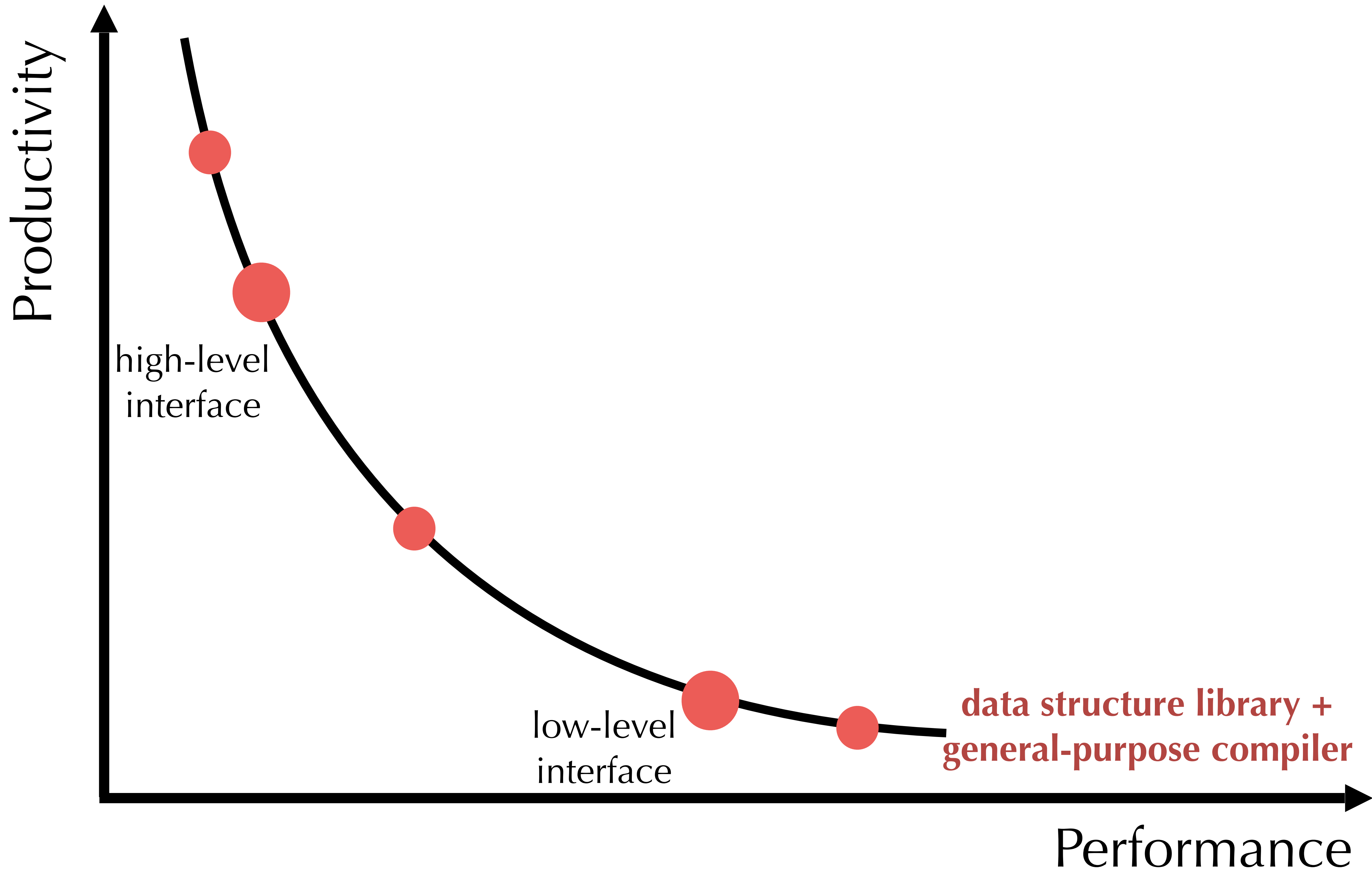
**Coarser**

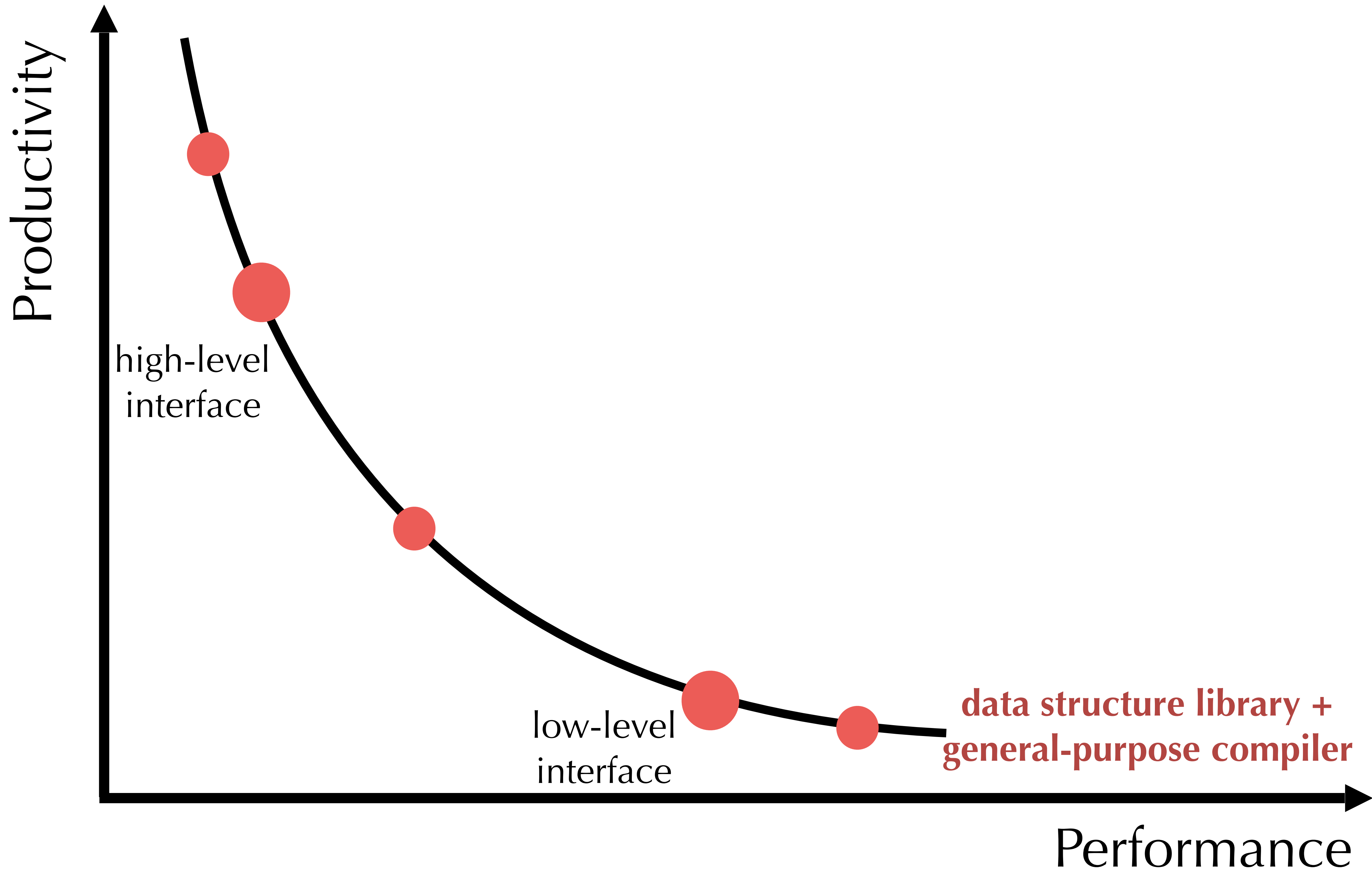
**Finer**

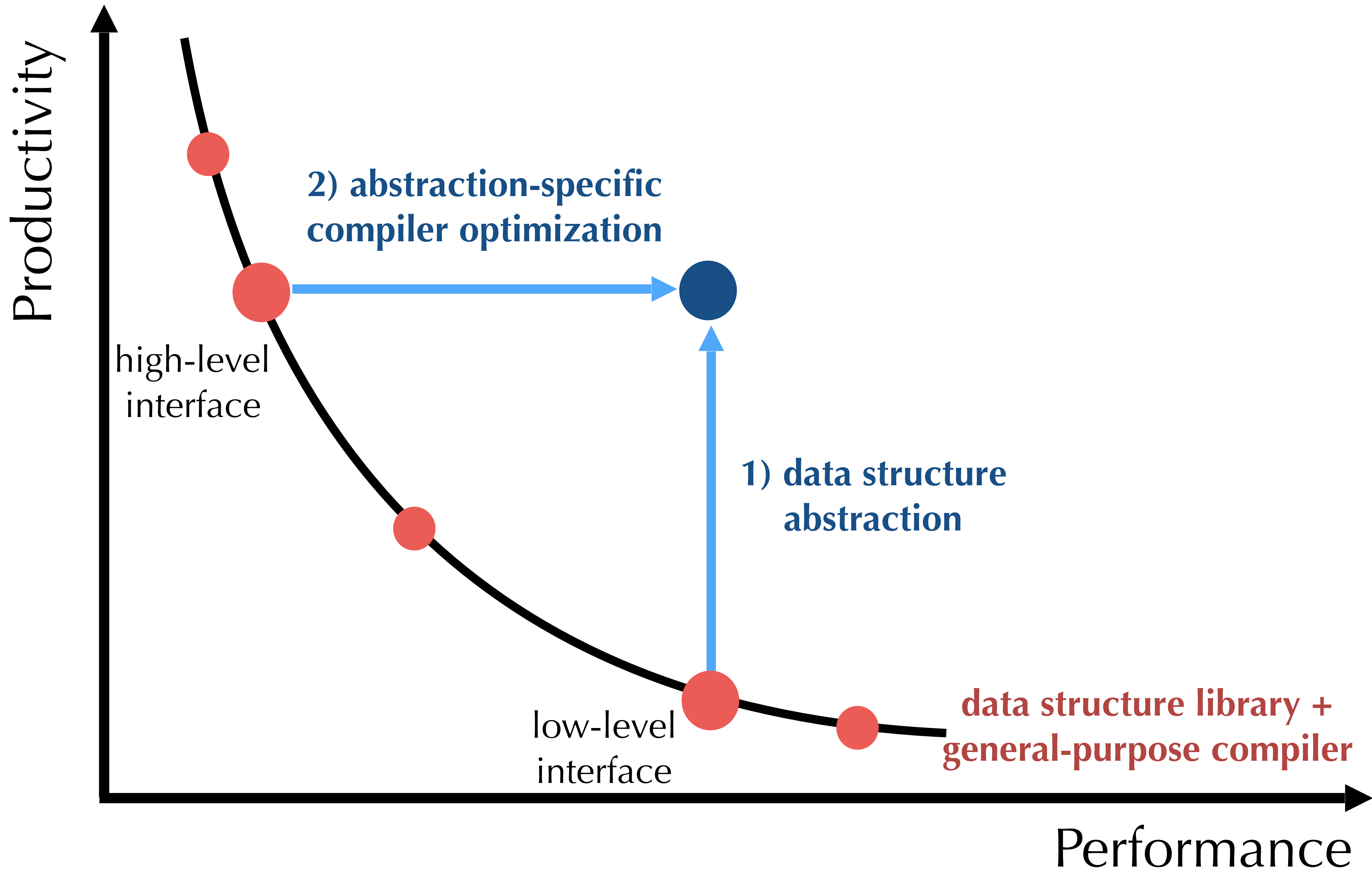
# Data Access Semantics

## ◆ (Seemingly trivial) assumptions that enables compiler optimization:

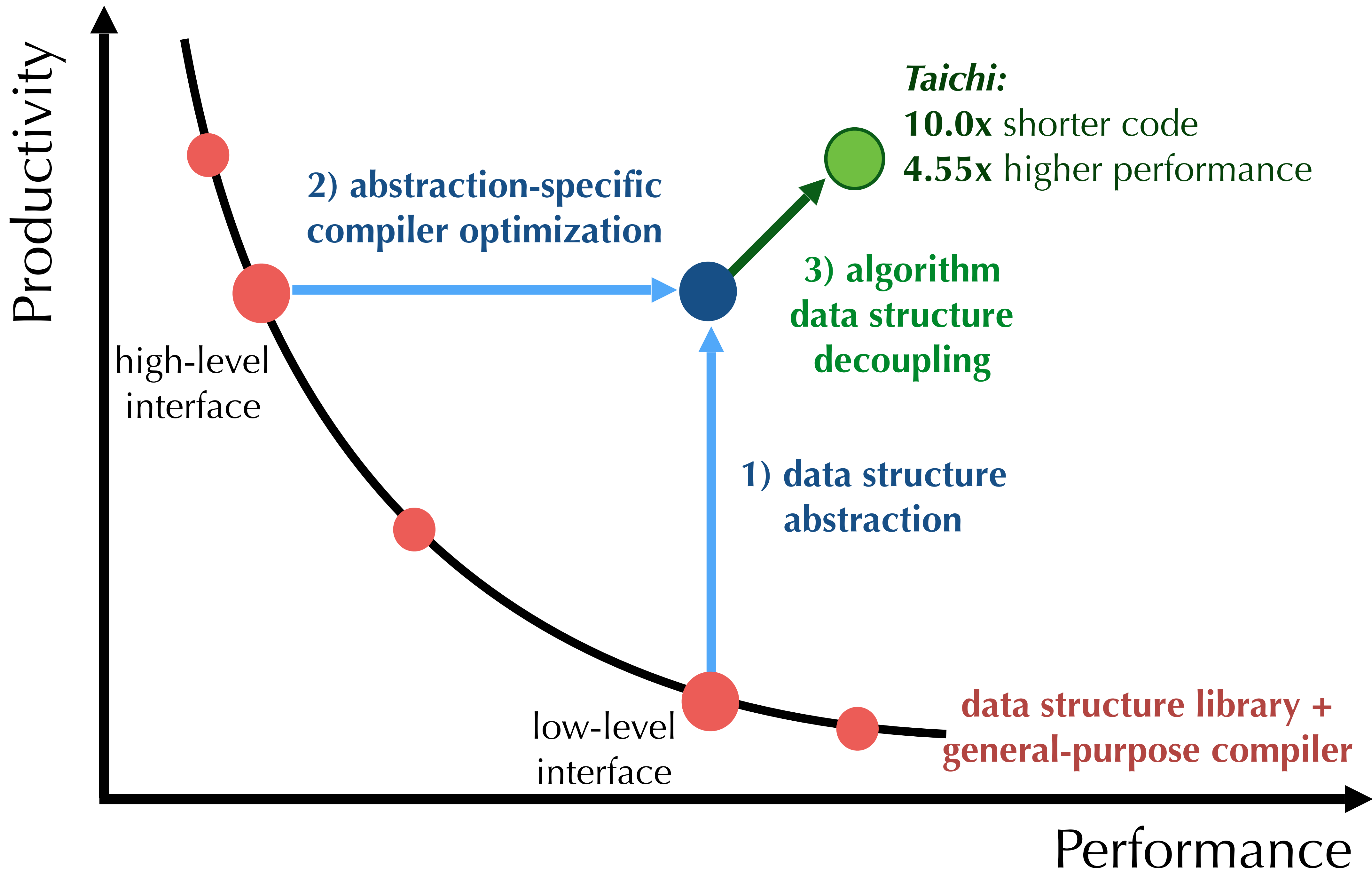
- ◉ No pointer aliasing:  $\mathbf{a}[x, y]$  and  $\mathbf{b}[i, j]$  never overlaps if  $\mathbf{a} \neq \mathbf{b}$
- ◉ All memory accesses are done through **sparse\_grid**[indices]
- ◉ The only way data structures get modified, is through write accesses of form **sparse\_grid**[indices]
- ◉ Read access **does not** modify anything
  - ▶ No memory allocation
  - ▶ No exception if out of ranges (element does not exist)







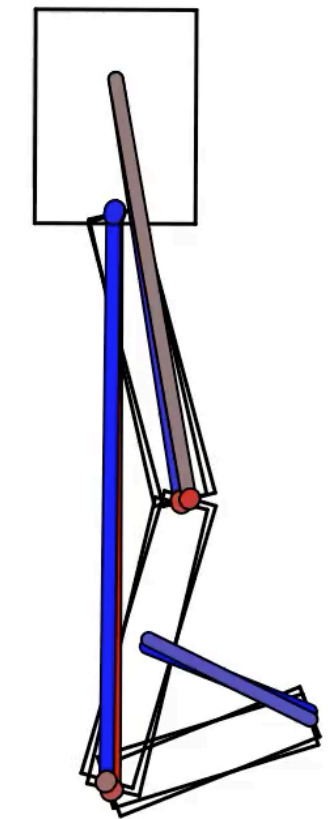
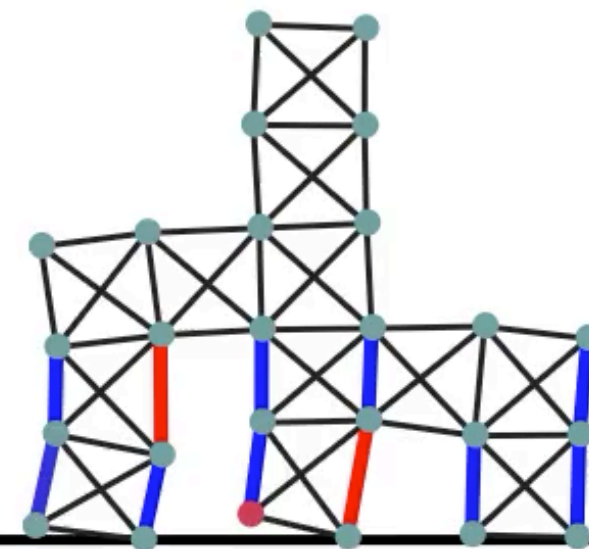
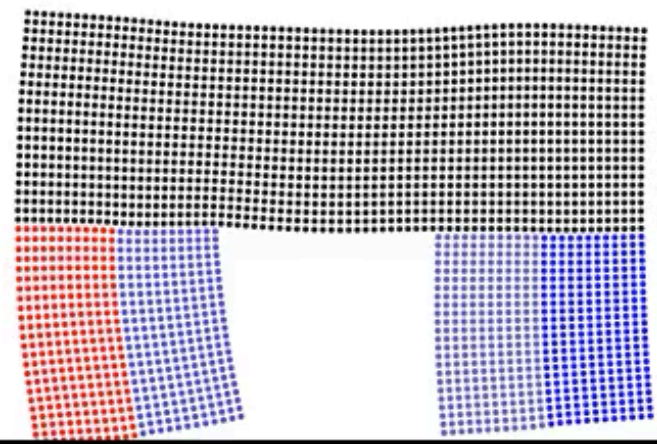




(Advertisement)

# DiffTaichi:

## Differentiable Programming for Physical Simulation

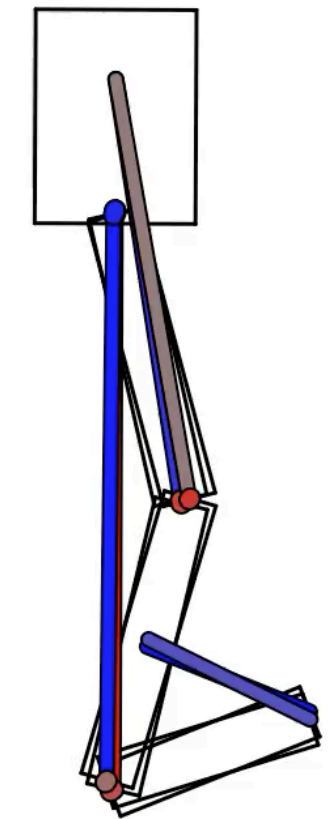
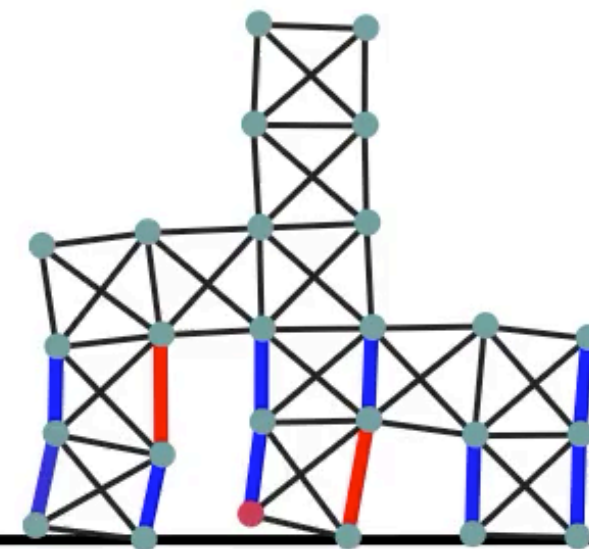
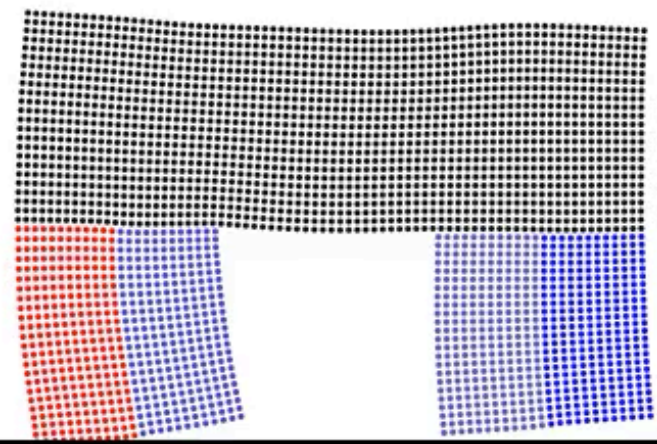


End2end optimization of neural network controllers with gradient descent

(Advertisement)

# DiffTaichi:

## Differentiable Programming for Physical Simulation



End2end optimization of neural network controllers with gradient descent

# The End

Source code: <https://github.com/yuanming-hu/taichi>

```
pip3 install taichi-nightly
```

All performance numbers from our system are reproducible (commit `dc162e11`) with a single command.

# Thank you!