High-Performance Computation on Spatially Sparse Data Structures

Yuanming Hu¹ Tzu-Mao Li² Luke Anderson¹ Jonathan Ragan-Kelley² Fredo Durand¹

¹MIT CSAIL ²UC 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.
Spatial Sparsity:
Regions of interest only occupy a small fraction of the bounding volume.
Spatial Sparsity
globally sparse, locally dense

“General” Sparsity

Legend
- Active
- Inactive
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...
In reality...
In reality…
Essential Computation
Data Structure Overhead

99%

1%

In reality...
In reality...

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

Essential Computation: 1%

Data Structure Overhead: 99%
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.
Data structure access:
- 50 clock cycles / element

Simple Stencil Computation:
- 0.5 clock cycle / element
Sparse data structure overhead can be \textbf{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

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

Dense Data Structure

Array addressing

Time (clock cycles)
Data Accesses Drawn Proportionally…

Dense Data Structure

Array addressing

Time (clock cycles)
Data Accesses Drawn *Proportionally*…

Dense Data Structure

Array addressing

Time (clock cycles)
Data Accesses Drawn Proportionally…

Time (clock cycles)

Sparse Data Structure

Hash table access

Array addressing
Data Accesses Drawn Proportionally…

Sparse Data Structure

Hash table access

Array addressing

Time (clock cycles)
Data Accesses Drawn Proportionally…

Sparse Data Structure

Hash table access

Array addressing
Data Accesses Drawn Proportionally...

Sparse Data Structure

Hash table access

Array addressing
Data Accesses Drawn Proportionally…

Sparse Data Structure

Hash table access

Array addressing
Data Accesses Drawn Proportionally...

Sparse Data Structure

Hash table access

Array addressing
Data Accesses Drawn Proportionally...
Data Accesses Drawn Proportionally...

Sparse Data Structure

Time (clock cycles)

Hash table access

Array addressing
Data Accesses Drawn Proportionally…

Reducing redundant data access is the key to high-performance spatially sparse computation!
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”
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*
Our Solution: 

The Taichi Programming Language

1) **Decouple** computation from data structures

<table>
<thead>
<tr>
<th>Computational Kernels</th>
<th>(Sparse) Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Kernel(laplace).def() {</code></td>
<td><code>2D Laplace operator</code></td>
</tr>
<tr>
<td><code>  For(u, [Expr i, Expr j]){</code></td>
<td></td>
</tr>
<tr>
<td><code>    auto c = 1.0f / (dx * dk);</code></td>
<td></td>
</tr>
<tr>
<td><code>    u[i, j] = c * (4 * v[i, j] - v[i+1, j]</code></td>
<td></td>
</tr>
<tr>
<td><code>       - v[i-1, j] - v[i, j+1] - v[i, j-1];</code></td>
<td></td>
</tr>
<tr>
<td><code>  });</code></td>
<td></td>
</tr>
<tr>
<td><code>}));</code></td>
<td></td>
</tr>
</tbody>
</table>

2) **Imperative** computation language
Our Solution: 

**The Taichi Programming Language**

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

<table>
<thead>
<tr>
<th>Computational Kernels</th>
<th>(Sparse) Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Kernel(laplace).def([&amp;]()) {</code></td>
<td><code>Global(u, f32); Global(v, f32);</code></td>
</tr>
<tr>
<td><code>For(u, [&amp;](Expr i, Expr j){</code></td>
<td><code>layout([&amp;](){</code></td>
</tr>
<tr>
<td><code>auto c = 1.0f / (dx * dx);</code></td>
<td><code>auto ij = Indices(0, 1);</code></td>
</tr>
<tr>
<td><code>u[i, j] = c * (4 * v[i, j] - v[i+1, j]</code></td>
<td><code>root.dense(ij, {128, 128}).pointer()</code></td>
</tr>
<tr>
<td><code> - v[i-1, j] - v[i, j+1] - v[i, j-1];</code></td>
<td><code>.dense(ij, {8, 8}).place(u, v);</code></td>
</tr>
<tr>
<td><code>});</code></td>
<td><code>});</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>2D Laplace operator</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>1024^2</code> sparse grid with $8^2$</td>
</tr>
</tbody>
</table>

2) **Imperative** computation language

3) **Hierarchical** data structure description language
Our Solution:

---

The Taichi Programming Language

1) **Decouple** computation from data structures

<table>
<thead>
<tr>
<th>Computational Kernels</th>
<th>(Sparse) Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Kernel(laplace).def([]){</code></td>
<td><code>Global(u, f32); Global(v, f32);</code></td>
</tr>
<tr>
<td><code>    For(u, [])(Expr i, Expr j){</code></td>
<td><code>    layout([]){</code></td>
</tr>
<tr>
<td><code>        auto c = 1.0f / (dx * dx);</code></td>
<td><code>            auto ij = Indices(0, 1);</code></td>
</tr>
<tr>
<td><code>    }};</code></td>
<td><code>            .dense(ij, [8, 8]).place(u, v);</code></td>
</tr>
<tr>
<td><code>2D Laplace operator</code></td>
<td><code>1024^2</code> sparse grid with $8^2$</td>
</tr>
</tbody>
</table>

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

**Computational Kernels**

```cpp
Kernel(laplace).def([&]() {
    For(u, [i](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**

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

1024^2 sparse grid with 8^2

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*

2) **Imperative** computation language

3) **Hierarchical** data structure description language

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

5) Auto *parallelization*, memory management, …

### Computational Kernels

```cpp
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

```cpp
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);
});
```

1024² sparse grid with 8²

### High-Performance CPU/GPU Kernels

<table>
<thead>
<tr>
<th>Ours v.s. State-of-the-art:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLS-MPM</td>
</tr>
<tr>
<td>FEM Kernel</td>
</tr>
<tr>
<td>MGPCG</td>
</tr>
<tr>
<td>Sparse CNN</td>
</tr>
</tbody>
</table>

### IR & Optimizing Compiler

### Runtime System

- Auto parallelization
- Memory management
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, ...

---

**Computational Kernels**

```cpp
Kernel(laplace).def([]() {
    For(u, [i](Expr i, Expr j)){
        auto c = 1.0f / (dx + dy);
        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**

```cpp
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);
});
```

1024² sparse grid with 8²

---

10x shorter code, 4.55x faster

**High-Performance CPU/GPU Kernels**

- **Ours v.s. State-of-the-art:**
  - MLS-MPM: 13x shorter code, 1.2x faster
  - FEM Kernel: 13x shorter code, 14.5x faster
  - MGPCG: 7x shorter code, 1.9x faster
  - Sparse CNN: 9x shorter code, 13x faster

---

**Runtime System**

- **Auto parallelization**, memory management, ...
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

```python
@ti.kernel
def laplace():
    for i, j in u:
        c = 1 / (dx * dx)
        u[i, j] = c * (4.0 * v[i, j] - v[i-1, j] - v[i+1, j]
                        - 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.
## Describing Data Structures

<table>
<thead>
<tr>
<th>Structural Nodes</th>
<th>Node Decorators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dense</strong>: A fixed-length contiguous array.</td>
<td><strong>morton</strong>: Reorder the data in memory using a Z-order curve (Morton coding), for potentially higher spatial locality. For dense only.</td>
</tr>
<tr>
<td><strong>hash</strong>: Use a hash table to maintain the mapping from active coordinates to data address in memory. Suitable for high sparsity.</td>
<td><strong>bitmasked</strong>: Use a mask to maintain sparsity information, one bit per child. For dense only.</td>
</tr>
<tr>
<td><strong>dynamic</strong>: 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.</td>
<td><strong>pointer</strong>: Store pointers instead of the whole structure to save memory and maintain sparsity. For dense and dynamic.</td>
</tr>
</tbody>
</table>
Describing Data Structures

**Structural Nodes**

- **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.

**Node Decorators**

- **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`. 
Describing Data Structures

**Structural Nodes**

- **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.

**Node Decorators**

- **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.

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

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

**VDB** [Museth 2013]

**SPGrid** [Setaluri et al. 2014]
Describing Data Structures

**Structural Nodes**

- **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.

**Node Decorators**

- **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`.

---

**VDB [Museth 2013]**

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

**SPGrid [Setaluri et al. 2014]**

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

---

“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

Root2leaf (end2end) data access

Access lowering

micro-access instructions

“Common subexpression elimination”

Simplified micro-access instructions
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

<table>
<thead>
<tr>
<th>High-Performance CPU/GPU Kernels</th>
<th>Ours v.s. State-of-the-art:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLS-MPM</td>
<td>13x shorter code, 1.2x faster</td>
</tr>
<tr>
<td>FEM Kernel</td>
<td>13x shorter code, 14.5x faster</td>
</tr>
<tr>
<td>MGPCG</td>
<td>7x shorter code, 1.9x faster</td>
</tr>
<tr>
<td>Sparse CNN</td>
<td>9x shorter code, 13x faster</td>
</tr>
</tbody>
</table>
Benchmarks: MLS-MPM

Patterns: Particle scatter/gather

Particle to Grid (P2G)

Grid to Particle (G2P)
Benchmarks: MLS-MPM

Patterns: Particle scatter/gather
Benchmarks: MLS-MPM

<table>
<thead>
<tr>
<th>Particle Layout</th>
<th>Ordered</th>
<th>Randomly Shuffled</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>3.52ms</td>
<td>21.23 ms</td>
</tr>
<tr>
<td>AOS</td>
<td>3.15ms</td>
<td>4.28 ms</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th></th>
<th>GPU-SPM</th>
<th>GPU+SPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2G</td>
<td>5.102ms</td>
<td>2.011ms</td>
</tr>
<tr>
<td>G2P</td>
<td>1.975ms</td>
<td>0.722ms</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th></th>
<th>GPU-SPM</th>
<th>GPU+SPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2G</td>
<td>5.102ms</td>
<td>2.011ms</td>
</tr>
<tr>
<td>G2P</td>
<td>1.975ms</td>
<td>0.722ms</td>
</tr>
</tbody>
</table>

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

\[ f = \sum_{c \in C(i)} \sum_{j \in V(c)} (\mu^{(c)} \cdot K^\mu + \lambda^{(c)} \cdot K^\lambda)_{i(c)j(c)} \cdot u_j. \]
Benchmarks: FEM Kernel

3 channels

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

8 elements 8 elements

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

3x8x8x3x2 = 1152 FLOPs/vertex
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

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

**Benchmarks: FEM Kernel**

<table>
<thead>
<tr>
<th>Ablation</th>
<th>CPU Time</th>
<th>GPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No multithreading</td>
<td>73.43ms</td>
<td>-</td>
</tr>
<tr>
<td>No vectorization</td>
<td>83.54ms</td>
<td>-</td>
</tr>
<tr>
<td>No vectorized load instructions</td>
<td>22.69ms</td>
<td>-</td>
</tr>
<tr>
<td>No simplification I</td>
<td>17.01ms</td>
<td>2.13 ms</td>
</tr>
<tr>
<td>No access lowering</td>
<td>182.19ms</td>
<td>6.046 ms</td>
</tr>
<tr>
<td>No simplification II</td>
<td>85.51ms</td>
<td>11.784 ms</td>
</tr>
<tr>
<td>AOS instead of SOA</td>
<td>136.03ms</td>
<td>20.992 ms</td>
</tr>
</tbody>
</table>

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

## Benchmarks: FEM Kernel

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

AOS is really bad in this case since
1) no vectorized ld/st
2) low cacheline util.

**Reproduce:**
Benchmarks: FEM Kernel

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

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

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

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)

After access lowering:
Vectorized FEM Access Optimization

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

```
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) &
and 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)
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)
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

- **Finer**
  - LLVM IR
  - Machine code

- **Coarser**
  - End2end access
  - Level-wise Access
  - Taichi IR
  - LLVM IR
  - Machine code
Data Access Semantics

✧ (Seemingly trivial) assumptions that enables compiler optimization:

- No pointer aliasing: \( a[x, y] \) and \( b[i, j] \) never overlaps if \( a \neq b \)
- All memory accesses are done through \texttt{sparse_grid}[indices]
- The only way data structures get modified, is through write accesses of form \texttt{sparse_grid}[indices]
- Read access \textbf{does not} modify anything
  - No memory allocation
  - No exception if out of ranges (element does not exist)
Performance vs. Productivity

- High-level interface
- Low-level interface
- Data structure library + general-purpose compiler
Performance vs Productivity

- High-level interface
- Low-level interface
- Data structure library + general-purpose compiler
1) data structure abstraction

2) abstraction-specific compiler optimization

Data structure library + general-purpose compiler
1) data structure abstraction

2) abstraction-specific compiler optimization

3) algorithm data structure decoupling

**Taichi:**
- 10.0x shorter code
- 4.55x higher performance
DiffTaichi:
Differentiable Programming for Physical Simulation

End2end optimization of neural network controllers with gradient descent
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!