Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures

YUANMING HU, MIT CSAIL
TZU-MAO LI, MIT CSAIL and UC Berkeley
LUKE ANDERSON, MIT CSAIL
JONATHAN RAGAN-KELLEY, UC Berkeley
FRÉDO DURAND, MIT CSAIL

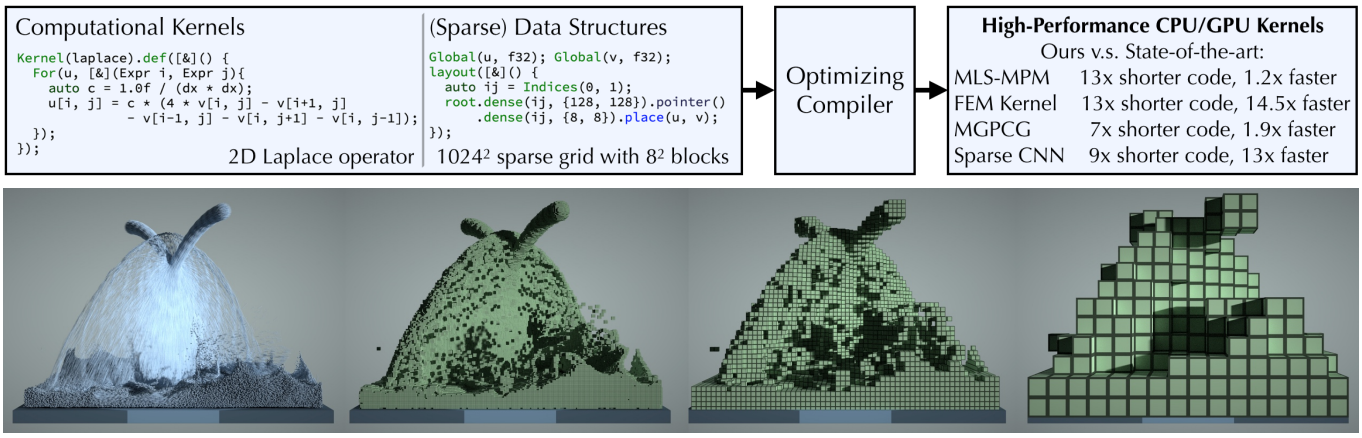


Fig. 1. (Top) We propose the Taichi programming language, which exposes a high-level interface for developing and processing spatially sparse multi-level data structures, and an optimizing compiler that automatically reduces data structure overhead. Programmers write code as if they are accessing dense voxels, while specifying the data arrangement *independently*. Our compiler automatically generates optimized, high-performance code tailored to the data structure. This results in concise code and better performance than highly-optimized reference implementations for various tasks. (Bottom) A fluid simulation using the material point method, where two liquid jets collide with each other, forming a thin sheet structure. We used a three-level sparse voxel grid with sizes 1^3, 4^3, 16^3. Involved voxels are visualized in green. Both simulation and rendering are done using programs written in Taichi.

3D visual computing data are often spatially sparse. To exploit such sparsity, people have developed hierarchical sparse data structures, such as multi-level sparse voxel grids, particles, and 3D hash tables. However, developing and using these high-performance sparse data structures is challenging, due to their intrinsic complexity and overhead. We propose *Taichi*, a new data-oriented programming language for efficiently authoring, accessing, and maintaining such data structures. The language offers a high-level, data structure-agnostic interface for writing computation code. The user independently specifies the data structure. We provide several elementary components with different sparsity properties that can be arbitrarily composed to create a wide range of multi-level sparse data structures. This *decoupling* of data structures from computation makes it easy to experiment with different data structures without changing computation code, and allows users to write computation as if they are working with a dense array. Our compiler then uses the semantics of the data structure and index analysis to automatically optimize for locality, remove redundant operations for coherent accesses, maintain sparsity and memory allocations, and generate efficient parallel and vectorized instructions for CPUs and GPUs.

Our approach yields competitive performance on common computational kernels such as stencil applications, neighbor lookups, and particle scattering. We demonstrate our language by implementing simulation, rendering, and vision tasks including a material point method simulation, finite element analysis, a multigrid Poisson solver for pressure projection, volumetric path tracing, and 3D convolution on sparse grids. Our computation-data structure decoupling allows us to quickly experiment with different data arrangements, and to develop high-performance data structures tailored for specific computational tasks. With $\frac{1}{10}$th as many lines of code, we achieve 4.55× higher performance on average, compared to hand-optimized reference implementations.

Authors' addresses: Yuanming Hu, MIT CSAIL, yuanming@mit.edu; Tzu-Mao Li, MIT CSAIL and UC Berkeley, tzumao@berkeley.edu; Luke Anderson, MIT CSAIL, lukea@mit.edu; Jonathan Ragan-Kelley, UC Berkeley, jrk@berkeley.edu; Frédo Durand, MIT CSAIL, fredo@mit.edu.

0730-0301/2019/11-ART201
https://doi.org/10.1145/3355089.3356506

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Computing methodologies → Parallel programming languages**; **Physical simulation**.

Additional Key Words and Phrases: Sparse Data Structures, GPU Computing.

ACM Reference Format:
Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* 38, 6, Article 201 (November 2019), 16 pages. https://doi.org/10.1145/3355089.3356506

1 INTRODUCTION

Large-scale 3D simulation, rendering, and vision tasks often involve volumetric data that are spatially sparse. Hierarchical and sparse data structures have been studied extensively to effectively exploit such sparsity. For example, in fluid simulation (Fig. 1), a multi-level grid is often used to represent the fluid field, where the fluid's spatial sparsity can be represented by nesting hash tables, bitmasks, or pointer arrays at different levels of the grid.

Writing high-performance code for these data structures is a daunting task due to their irregularity. Accessing their active elements in parallel imposes several engineering challenges (Fig. 2). First, naively traversing the hierarchy can take one or two orders of magnitude more clock cycles than the essential computation. This is especially troublesome for spatially coherent accesses commonly seen in, for example, stencil operations, since common access paths in the hierarchical data structure are traversed redundantly. Second, we need to ensure load-balancing for efficient parallelization. Third, we need to allocate memory and maintain sparsity when accessing inactive elements.

Data structure libraries do not guarantee high-performance code, since performance is not easily composable. Multiple calls to the library interface will result in redundant and costly traversals of the hierarchy. Unfortunately, because of the code complexity of these data structures, and potential race conditions and pointer aliasing, current general-purpose compilers often fail to optimize between library function calls. To achieve high performance, libraries usually have to expose low-level interfaces to users, leading to a leaky abstraction, making computation code highly coupled with data structures. We propose a new programming model that *decouples* data structures from computation, to achieve both high performance and easy programming (Fig. 3). Users write computation code using a high-level and data-structure-agnostic interface, as if they are operating on a dense multi-dimensional array. The internal data arrangement and the associated sparsity are specified independently from the computation code by composing elementary components such as dense arrays and hash tables to form a hierarchy.

Our compiler tailors optimizations for the specified data structure components, and generates efficient sparsity and memory maintenance code. We develop several domain-specific strategies for optimizing spatially-coherent accesses, using index analysis derived from high-level information about the data layout and the access patterns. Our compiler analyzes accesses to efficiently compute memory addresses, uses a caching strategy for better locality, and parallelizes/vectorizes loops from high-level instructions from the programmer. This is enabled by our compact intermediate representation, specially designed for optimizing hierarchical sparse data structures. Our compiler generates C++ code or CUDA code from the intermediate representation, making switching backends effortless.

On many common computations such as stencils, neighbor lookups, and particle splatting, our compiler generates code faster than highly-optimized reference implementations. We implement several popular simulation, graphics and vision algorithms in our language's embedded C++ frontend, including the material point method [Gao et al. 2018; Hu et al. 2018; Stomakhin et al. 2013], finite element kernel [Liu et al. 2018], multigrid Poisson solver [McAdams et al. 2010], sparse 3D convolution [Graham et al. 2018], and volumetric path tracing. Compared with highly-optimized reference implementations, our code requires on average only $\frac{1}{10}$th the number of lines of code, while being 4.55× faster (geometric mean).

We can quickly explore different choices of data structures, while our compiler generates high-performance code. For example, we derived more efficient data structure designs for the material point method that not only lead to performance improvements of up to 1.2× over previous, highly-optimized, state-of-the-art implementation [Gao et al. 2018], but also simplify the whole algorithm (Sec 6.1).

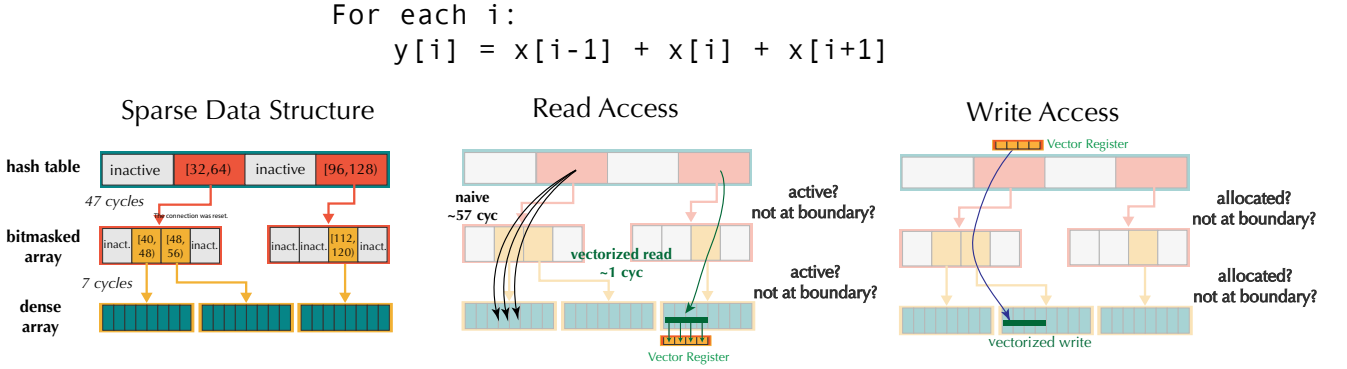


Fig. 2. Accessing a multi-level sparse array is significantly more involved than accessing a dense array. The figure illustrates an example three-level sparse array and the read and write access of a stencil `y[i]=x[i-1]+x[i]+x[i+1]`. Naive code is usually inefficient, since the hierarchy makes traversal costly, and it is especially problematic for spatially coherent accesses, where the top of the traversal is often redundant. Optimized and vectorized code needs to leverage access locality to amortize the access cost, check for sparsity, handle boundary cases, and allocate memory when necessary. Writing code for these accesses is tedious and error-prone, and it often leads to code that is highly-coupled with the data structure. Our language decouples the data structure implementation and the access, while our compiler automatically generates optimized code given the access pattern and the specific data structure. As a result, users write code as if they are accessing dense arrays, while having the freedom to change the data layout and sparsity representation without affecting the computation code.

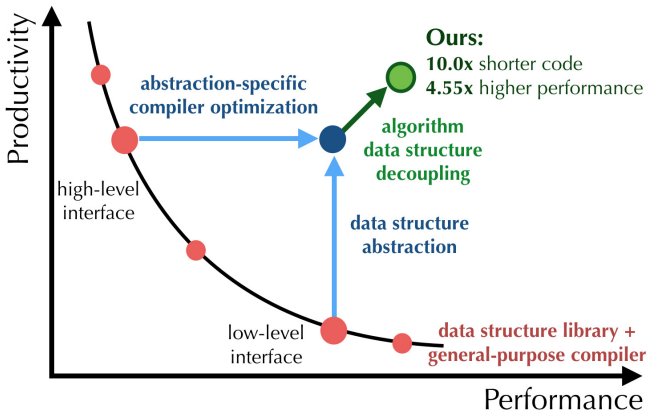


Fig. 3. Traditionally, a user who writes data structure access code faces a dilemma between easy programming and high performance. The goal of our language is to achieve both the productivity of a high-level library and the high performance of manually optimized code. Furthermore, since our language makes it easy to experiment with different data structures, users can often achieve even higher performance by exploring the data structure design space and adopting the most efficient design for a given task.

Our model can express a wide variety of data structures used in physical simulation and rendering. In particular, it can describe different multi-level sparse grids (e.g. SPGrid [Setaluri et al. 2014], OpenVDB [Museth 2013], and other novel data structures), particles, and dense and sparse matrices. We assume the hierarchy is known at compile-time to facilitate compiler optimization, therefore we do not directly model structures with variable depth such as k-d trees.

Our language and compiler are open-source[1]. All performance numbers from our system in this paper can be reproduced with the provided commands. All visual results are simulated and rendered using programs written in our language.

We summarize our contributions as follows:

- A programming language that decouples data structures from computation (Sec. 3.1). We provide a unified abstraction to map multi-dimensional indices to memory addresses. Such an abstraction allows programmers to define computation independently of the internal arrangements of the involved data structures.
- A data structure description mini-language, which provides several elementary data structure components that can be composed to form a wide range of sparse arrays with static hierarchies (Sec. 3.2).
- An optimizing compiler that uses index analysis and information from the data structures to automatically optimize for locality, minimize redundant operations for coherent accesses, manage sparsity, and to generate parallelized and vectorized backend code for x86_64 and CUDA (Sec. 4 and Sec. 5).
- A thorough evaluation of our system, and state-of-the-art implementations of several graphics and vision algorithms as by-products.

[1]https://github.com/yuanming-hu/taichi

2 GOALS AND DESIGN DECISIONS

Most sparsity patterns in 3D computing tasks exhibit spatial coherency (Figure 4). The sparsity may come from fluid simulation, clouds in volume rendering, or point clouds of surfaces from LiDAR and Kinect scans. To obtain high performance, we want to model the spatial sparsity effectively so we can utilize the spatial coherency while not wasting computational resources on empty space.

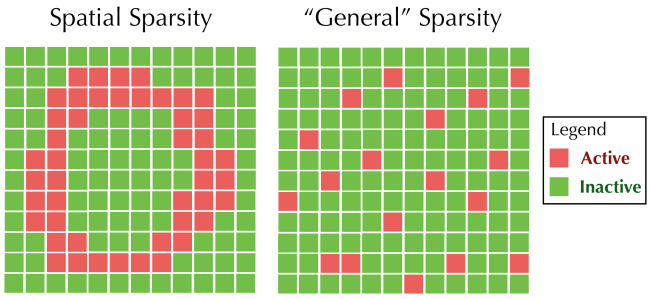


Fig. 4. **Left:** We focus on *spatial sparsity*, where the data is globally sparse yet locally dense; **Right:** *General sparse problems with random patterns* are less suited to our language.

We aim to develop a high-performance programming language to exploit spatial sparsity using dedicated data structures. The four high-level goals are as follows:

Expressiveness. Our target applications often feature complex computational kernels, such as stencils of different sizes, particle splatting, and ray-voxel intersection. Therefore, the language should be expressive enough to cover these numerical computation patterns. Taichi allows users to read/write to arbitrary elements in the sparse data structures, and provides constructs for branching and looping. This distinguishes our languages from more domain-specific ones, such as taco [Kjolstad et al. 2017] (linear algebra).

Performance. On modern computer architectures, achieving high performance means good exploitation of locality and parallelism. However, the desired memory-friendly and parallel data structure is usually task- and hardware-dependent, so existing data structure libraries that only provide a single data structure design do not completely solve the performance issue.

Productivity. Traditionally, programming on sparse data structures requires manually handling memory allocation, parallelization, exceptions and boundary conditions. Even with libraries, low-level programming is still necessary to achieve high performance. Our language allows programming on sparse data structures *as if they are dense*, while the compiler automatically generates optimized code. To our knowledge, Taichi is the first system that makes it possible to write large-scale physical simulations on complex data structures within only a few hundred lines of code.

Portability. The language should automatically generate optimized code for different hardware environments. We do **not** offer programmer access to low-level control over hardware when it would sacrifice performance portability, like the prefetch intrinsics on x86 CPUs or warp-level intrinsics on NVIDIA GPUs.

2.1 Design Decisions

Our design decisions are made based on the aforementioned goals and non-goals.

- **Decouple data structures from computation.** The user should write high-level code for computation as if they are processing a dense array, while also being able to explore different sparse data structures without affecting the computation code. We achieve this by abstracting data structure access with Cartesian indexing, while the actual data structures define the mapping from the index to the actual memory address (Sec. 3.1).
- **Regular grids as building blocks** The basic data structure entities of our system are regular grids, which can be easily flattened into 1D arrays that map closely to modern computer architecture with linear memory addressing. We do not directly model more irregular structures such as meshes or graphs[2]. Multiresolution representations such as adaptive grids [Losasso et al. 2004] need to be composed manually in our language (see Section 6.3, or Setaluri et al.'s multigrid preconditioner [2014]).
- **Describe data structures through hierarchical composition.** To model spatial sparsity, and to express a wide variety of data structures, we develop a data structure mini-language to compose data structure hierarchies (Sec. 3.2). The mini-language is made up of several elementary components, such as dense arrays and hash tables, that are arbitrarily composable.
- **Fixed data structure hierarchy.** We facilitate compiler optimizations and simplify memory allocation by assuming the hierarchy to be fixed at compile time. We do not support octrees or bounding volume hierarchies with dynamic depth. Many state-of-the-art physical simulation systems use data structures with a fixed hierarchy such as SPGrid [Setaluri et al. 2014] and VDB [Hoetzlein 2016; Museth 2013].
- **Single-Program-Multiple-Data (SPMD) with sparse iterators.** We adopt an imperative SPMD model to harness the power of modern hardware such as vectorized instructions on CPUs and massively parallel GPUs. To exploit sparsity, we design computation kernels to be parallel for loops with sparse iterators on active elements only. This provides programmers a simple yet expressive interface to sparse computation.
- **Generate optimized backend code automatically.** Our compiler should generate high-performance backend code automatically, while optimizing for locality (Sec. 4.1), minimizing redundant accesses using access coherency (Sec. 4.2), automatically parallelizing (Sec. 4.3) and allocating memory (Sec. 5.2). The user should only need to provide the backend target architecture and optionally some scheduling hints for the compiler to generate better optimized code.

3 THE TAICHI PROGRAMMING LANGUAGE

We demonstrate our language using a 2D Laplace operator $u = \nabla^2 v$, which is frequently used in physical simulation and image processing. After finite difference discretization, the operation is defined as:

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

3.1 Defining Computation

To decouple data structures from computation, we abstract data structures as *mappings from multi-dimensional indices to the actual value*. For example, access to the 2D scalar field u is always done through indexing, i.e. `u[i, j]`, no matter what the internal data structure is. This is similar to high-level interfaces of some data structure libraries, yet our compiler analyzes these accesses and produces code that minimizes redundancy across multiple accesses.

Our language's frontend is embedded in C++. Computations in our language are usually defined as kernels looping over *active* data structure elements (e.g. non-zero pixels or voxels), to efficiently exploit data sparsity. The kernel contains imperative code that operates on the data structures.

We define the aforementioned Laplace operator as a kernel, using a for loop over variable `u`, which iterates over all pairs `(i, j)` where `u[i, j]` is an *active* element:

```
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]);
  });
});
```

For loops over active elements are key to sparse computation in Taichi. The compiler automatically maintains sparsity. When reading from an inactive element of `v`, the compiler returns an *ambient value* (e.g., 0). When writing to an inactive element of `u`, the compiler automatically changes the internal data structure, allocates memory, and marks the element as active (In this specific kernel, no activation will occur, since we are only writing to active elements of `u`, and interaction with `v` is read-only).

We adopt the Single-Program-Multiple-Data paradigm. Our language is similar to other SPMD languages such as ispc and CUDA, with three additional components: 1) parallel sparse `For` loops, 2) multi-dimensional sparse array accessors, and 3) compiler hints for optimizing program scheduling.

The `For` loop is automatically parallelized and vectorized. Our language supports typical control flow statements, such as `If-Then-Else` and `While` loops. We allow users to define mutable local variables (`Var`). Our language can be used to write a full volumetric path tracer with complex control flow (Sec. 6.5). The language constructs supported inside computation kernels are listed below.

```
// Parallel loop over the sparse tensor "var"
For(Expr var, std::function)
// Loop over [begin, end)
For(Expr begin, Expr end, std::function)
// Access one element in "var" with index (i, ...)
operator[](Expr var, Expr i, ...)

While(Expr cond, std::function)
If(Expr cond)
If::Then(std::function)
If::Else(std::function)
Var(Expr) // Declare a mutable local variable
Atomic(A) += B // Atomic add to global element A
```

[2]It is possible to use 1D arrays for storing vertices and edges in meshes/graphs.

Our language also offers compiler hints for scheduling:

```
// For CPU
Parallelize(int num_threads) // Multi-threading
Vectorize(int width)         // Loop vectorization
// For GPU
BlockDim(int blockDim) // Specify GPU block size
// For scratchpad optimization
AssumeInRange(Expr base, int lower, int upper)
Cache(Expr)
// Cache data into GPU L1 cache
CacheL1(Expr)
```

More discussions on hints for scratchpad optimization (`AssumeInRange` and `Cache`) and `CacheL1` are in Section 4.1.

3.2 Describing Internal Structures Hierarchically

After writing the computation code, the user needs to specify the internal data structure hierarchy. Specifying a data structure includes choices at both the macro level, dictating how the data structure components nest with each other and the way they represent sparsity, and the micro level, dictating how data are grouped together (e.g. structure of arrays vs. array of structures).

Structural nodes and their decorators. Our language provides *structural nodes* to compose the hierarchy, and *decorators* to provide structural nodes with particular properties. These constructs and their semantics are listed below:

`dense`: A fixed-length contiguous array.
`hash`: Use a hash table to maintain the mapping from active coordinates to data addresses 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 in a block.

(a) 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`.

(b) node decorators

These data structure components provide trade-offs regarding access cost and space consumption. For example, a hash table has relatively long access time (e.g. 50 CPU cycles), but it is very economical in terms of memory space, especially in extremely sparse cases (e.g. 0.1%). Therefore it is often suitable for the top layer, when only a few hundred children are active out of, say, $128 \times 128 \times 128$. On the other hand, a dense array with a bitmask can be activated and accessed quickly, but the bitmask will occupy space inefficiently in highly sparse cases.

Defining the hierarchy. Users can compose the data structure components arbitrarily to form desired hierarchies and to explore different trade-offs. The compiler will then synthesize how computational kernels are executed on the specific sparse data structure (Fig. 5).

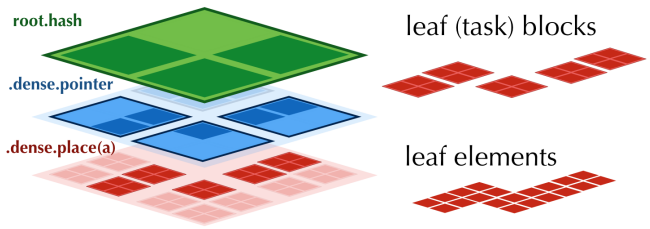


Fig. 5. In our language, programmers define data structures by nesting elementary components such as hash tables and dense arrays. Kernels are defined as iterations over **leaf elements** (i.e., voxels or pixels), independent of the internal data organization. **Leaf blocks**, immediate blocks of leaf elements, are the smallest quantum of storage and computation tasks.

For example, the following code specifies two fixed-size 2D dense arrays over `u` and `v`.

```
Global(u, f32); Global(v, f32);
layout([&]() {
  auto ij = Indices(0, 1);
  // Allocate a structure-of-arrays dense grid.
  // Equivalent to:
  // float u[256][256]; float v[256][256];
  root.dense(ij, {256, 256}).place(u);
  root.dense(ij, {256, 256}).place(v);
});
```

`Global(u, type)` declares an N-dimensional (sparse) tensor of name `u` and type `type`. These tensors are accessible by all kernels, so we call them *global variables*.

`layout` takes a C++ lambda function that describes the data structure hierarchy. `Indices` are used to specify sizes of structural nodes.

`root` denotes the root of the hierarchy. `dense`, a *structural node* of the tree, creates a child node of the root. Calling `dense` on `root` twice creates two children. Each structural node function call has two arguments, the first specifies the dimensions of its children, the second specifies the number of elements in the corresponding dimension. Here, `dense(ij, {256, 256})` means the 2D dense array has 256 cells along index `i` (x-axis) and 256 cells along `j` (y-axis).

`place(u)` and `place(v)` assign the global variables `u, v` to the corresponding data structure hierarchies. The equivalent C-style data structure definition is provided in the comments.

The code above specifies a structure-of-arrays (SOA) layout. We can easily switch to an array-of-structures (AOS) layout using the following code:

```
// struct node {float u, v;};
// node data[256][256];
auto &node = root.dense(ij, {256, 256});
node.place(u); node.place(v);
// or equivalently
root.dense(ij, {256, 256}).place(u, v);
```

In this case, a single `dense` node contains both `u` and `v`, since we called `place` twice on the same `dense` node. As syntactic sugar,

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

(a) 3D VDB-style [Museth 2013] structure with configuration [5, 4, 3]. The root-level hash table allows negative coordinates to be accessed, providing the user with an unbounded domain.

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

(b) 3D SPGrids [Setaluri et al. 2014] occupying voxels in the bounding box $[0, 4096) \times [0, 2048) \times [0, 2048)$. The data structure is relatively shallow (only two levels), so root-to-leaf accesses have relatively low cost.

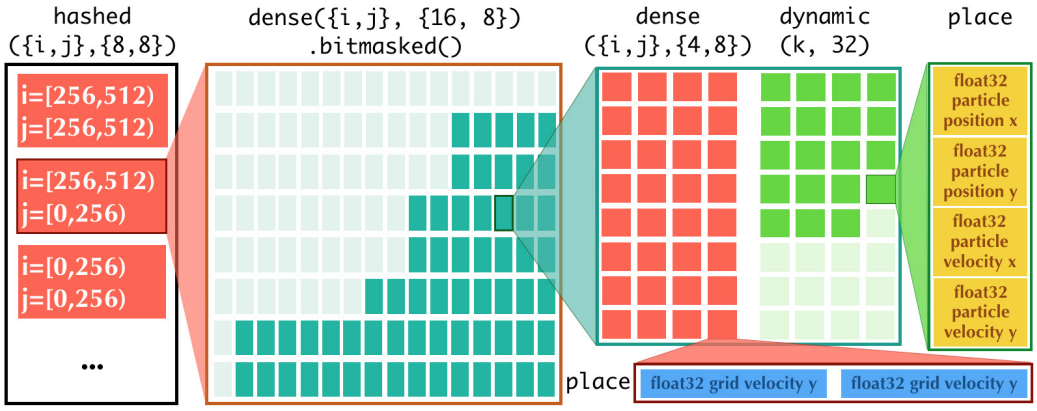
```
// "Hierarchical Particle Buckets": each leaf block contains all indices of particles within its range
root.dynamic(l, 2048).place(particle_x, particle_y, particle_z, particle_mass);
root.hash(ijk, 512).dense(ijk, 32).pointer().dense(ijk, 8).pointer().dynamic(l, 2048).place(particle_index);

// "SPVDB": Unbounded shallow data structures with bitmasks and Morton coding. (VDB and SPGrid combined.)
root.hash(ijk, 512).dense(ijk, 512).morton().bitmasked().dense(ijk, {8, 4, 4}).place(flags, u, v, w);
```

```
// "Hybrid Eulerian-
     Lagrangian Grid"
auto &block = root
 .hash(ij, 8)
 .dense(ij, {16, 8})
 .bitmasked();

// Child 1: grid nodes
block.dense(ij, {4, 8})
     .place(grid_vx)
     .place(grid_vy);

// Child 2: particles
block.dynamic(i, 32)
     .place(part_x)
     .place(part_y)
     .place(part_vx)
     .place(part_vy);
```



(c) **"HPB", "SPVDB", "HLEG":** We can easily design new data structures with customized features.

Fig. 6. The layout language allows users to define data structures using our building blocks. We can reproduce two popular multi-level sparse grid used in simulation (a) (b). Furthermore, we can use our language to design new data structures (c) by chaining and forking elementary components. Hybrid Eulerian-Lagrangian simulations (e.g. FLIP [Zhu and Bridson 2005] and MPM [Stomakhin et al. 2013]) often need to maintain both particles and grids, and the required data structures are usually complicated. Using these building blocks, we easily found a data structure with a hierarchical pointer list of particles, which we call *Hierarchical Particle Buckets*, that is especially useful for the material point method simulation (Sec.6.1).

`place` can also take more than one parameter. When materialized in memory, in this AOS layout $u_{i,j}$ and $v_{i,j}$ are next to each other, while in the previous SOA layout $u_{i,j}$ is next to $u_{i,j+1}$ and is far away from $v_{i,j}$. These two layouts have very different memory behaviors (e.g. cacheline utilization) in different applications.

We can *nest* the structural nodes to specify the hierarchical tree structure in a top-down order. For example, the following code defines a three-level sparse grid, with the top-level being a hash table, the second-level being a dense array of pointers, and the third-level being a fixed-size dense array (Fig. 5):

```
root.hash(ij, {4, 4})
    .dense(ij, {4, 4}).pointer()
    .dense(ij, {16, 16}).place(u, v);
```

Apart from multiple global variables, structural nodes can also have multiple structural nodes as children. For example, the following code defines a bitmasked sparse array, where each of its elements is composed of a dense array and a dynamic array (similar to `std::vector`):

```
Global(u, f32); Global(v, f32); Global(p, f32);
auto k = Index(2);
auto &block = root.dense(ij, {16, 16});
// Child 1: dense array
block.dense(ij, {16, 16}).place(u, v);
// Child 2: dynamic array
block.dynamic(k, 256).place(p);
```

The equivalent C++ code is:

```
struct Child1Node {
  float u;
  float v;
};
struct Block {
  Child1Node child1[16][16];
  std::vector<float> child2; // p
  // Note: in Taichi the dynamic array has a
  // pre-defined maximum size, unlike std::vector that
       grows arbitrarily.
};
struct Root {
  Block blocks[16][16];
};
```

The structural node types are concise, but they are capable of expressing a large variety of data structures. Figure 6 illustrates a few complex data structures represented with our language. A new data structure can be designed with a few lines of code. Rapidly experimenting with these data structures allows us to find the optimal one for a specific task and hardware architecture.

4 DOMAIN-SPECIFIC OPTIMIZATIONS

Hierarchical data structures provide an efficient representation for sparse fields but have high access costs due to their complexity, especially when parallelism is desired. Our compiler reduces access overhead from three typical sources:

Out-of-cache access. In modern architectures, loading data from main memory is around one hundred times slower than an in-cache access. Ensuring data locality is thus crucial for performance. This is particularly important on GPU, and it means that we need to efficiently utilize shared memory to cache data.

Data structure hierarchy traversal. Traversing hierarchical data structures is expensive. For example, hash table queries may take tens or hundreds of clock cycles. Fortunately, we can often amortize the cost by leveraging spatial locality (Fig. 2).

Instance activation. For write access, we need to activate previously inactive nodes. This usually involves atomic operations or spinlocks, which are not only intrinsically slow, but also serialized.

We present three types of optimizations that lead to higher performance, through better cache locality, reduction of redundant accesses, and automatic parallelization and vectorization.

4.1 Scratchpad Optimization through Boundary Inference

The "scratchpad" pattern is a common optimization to reduce load-to-use latency and memory bandwidth consumption, when potential data reuse exists. Scratchpads are small software-managed local data arenas, typically stored in L1 cache (CPU) or shared memory (GPU), and are intended for fast local computation. But programming with scratchpads is error-prone, and the size of a scratchpad is coupled with the leaf block size.

We provide a construct `Cache(v)` to enable the scratchpad optimization, which can be specified in a kernel. For example, take the discrete Laplace operator from Sec 3.1. Let us also assume that the inputs are stored in dense arrays at the leaf level (`dense(ij, {4, 4}).place(v)`). Our bound inference engine will infer that each output `u[i, j]` requires values from the 3×3 neighborhood of input `v`, and then allocate a local scratchpad array with the necessary size for this leaf block (Fig. 7, left and middle). We use interval analysis for bounds inference as in Halide [Ragan-Kelley et al. 2012] to determine a rectangular bound.

Our bounds inference requires the access offsets to be known at compile time. However, in many cases this is too restrictive. Fortunately, oftentimes it is possible to determine bounds using domain knowledge from the data. Therefore we provide an `AssumeInRange` construct for specifying the bounds of individual variables. The compiler then propagates these bounds to generate a scratchpad. For example, in the semi-Lagrangian advection kernel below, the backtrace distance is bounded by the Courant-Friedrichs-Lewy number and supplied to the compiler:

```
Kernel(advect).def([&]() {
  For(m, [&](Expr i, Expr j){
    auto u = velocity(0)[i, j];
    auto v = velocity(1)[i, j];

    auto backtrace_i = Var(i - cast<int32>(u * dt/dx));
    auto backtrace_j = Var(j - cast<int32>(v * dt/dx));

    backtrace_i = AssumeInRange(i, {-2,3});
    // i.e., i - 2 <= backtrace_i < i + 3;
    backtrace_j = AssumeInRange(j, {-2,3});
    // i.e., j - 2 <= backtrace_j < j + 3;

    m = m_input[backtrace_i, backtrace_j];
}});
```

In our current implementation, the scratchpad optimization is only applied for the GPU backend. The latency and bandwidth difference between software managed shared memory and hardware managed L2 cache makes such an optimization especially profitable on GPU. We anticipate this optimization would also help for the CPU backend, but since the CPU L1 cache already plays a similar role, the improvement might be less significant.

Apart from the `Cache` construct that provides shared memory usage hints, the `CacheL1(V)` construct for the GPU backend instructs the compiler to issue `__ldg` intrinsics to force data loads from the global variable `V` into GPU L1 cache. Unlike x86 CPUs, NVIDIA GPUs cache data in L2 cache by default. Since L1 caches are maintained by GPU hardware on the fly, no compile-time bound inference is needed.

4.2 Removing Redundant Accesses

As illustrated in Fig. 2, the cost of an expensive hierarchical data structure's access can often be amortized. By considering multiple accesses simultaneously, the compiler can leverage common traversal paths. This is a form of constant propagation and common subexpression elimination. We develop a minimal intermediate representation to represent data structure operations. The intermediate representation is specially designed for vectorized accesses and contains explicit information about accesses and data structure boundaries. This allows us to perform optimizations that a typical compiler cannot conduct automatically. We detail the intermediate representation and the expression simplification algorithms in Section 5 and Appendix A.

As an example, consider again the Laplace operator from Sec. 3.1. This time we assume we are accessing a three-level data structure like the one in Fig. 2. If index `j` is 4-wide loop vectorized, we know that `j` must be a multiple of four and `x[i, j+1]` must share the

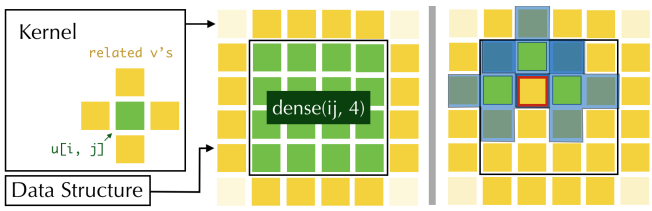


Fig. 7. **Left and middle:** Combining kernel and data structure information, the compiler will infer the elements required by this compute block. In this specific case, a 6×6 scratchpad will be generated, covering region $[-1, 5) \times [-1, 5)$. **Right:** The v (yellow square with red border) element is loaded into shared memory, and then reused by u five times (three shown in green blocks with their stencil shown in semitransparent blue). By doing this we reduce data load-to-use latency and main memory bandwidth consumption significantly.

same ancestor with `x[i, j]`. Therefore, it will be possible to traverse the data structure just once for both `i, j` and `i, j+1`. Our compiler detects this and handles boundary cases using the specific offset information stored in the IR (Fig. 8), while traditional compilers' heuristics usually fail to optimize due to code complexity and potential race conditions and pointer aliasing.

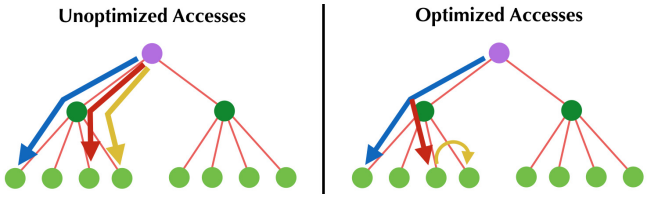


Fig. 8. Access optimization assuming the three accesses occur from left to right. The common paths of the accesses are eliminated. The yellow access is simplified to a compile-time known offset relative to the red access.

A similar optimization can also be applied for write operations. If two write accesses happen in the same memory address in the same kernel, the second write does not need to perform the expensive sparsity check and allocation.

4.3 Automatic Parallelization and Task Management

Parallelization and Load Balancing. Evenly distributing work onto processor cores is challenging on sparse data structures. Naively splitting an irregular tree into pieces can lead to partitions with drastically different numbers of leaf blocks (Fig.9).

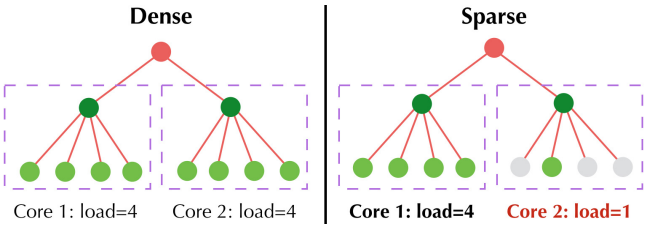


Fig. 9. Unlike the dense case (left), in sparse data structures, partitioning leaf nodes at a certain level may lead to an unsatisfactory load imbalance and therefore inefficient parallelism (right).

Our strategy is to generate a task list of leaf blocks, which flattens the data structure into a 1D array, circumventing the irregularity of incomplete trees. Importantly, we generate a task per block instead of a task per element (Fig. 5), to amortize the generation cost.

On CPU, generating the task list can be done via a light-weight traversal of the tree in serial. The task list is then queued into a thread pool. We then process the task queue in parallel via OpenMP.

On GPU, generating the task list in serial is infeasible. Instead, we maintain multiple task lists, one for each structural node on the root-to-leaf path. The lists are generated in a layer-by-layer manner: starting from the root node, the queue of active parent nodes is used to generate the queue of active child nodes. A global atomic counter is used to keep track of the current queue head.

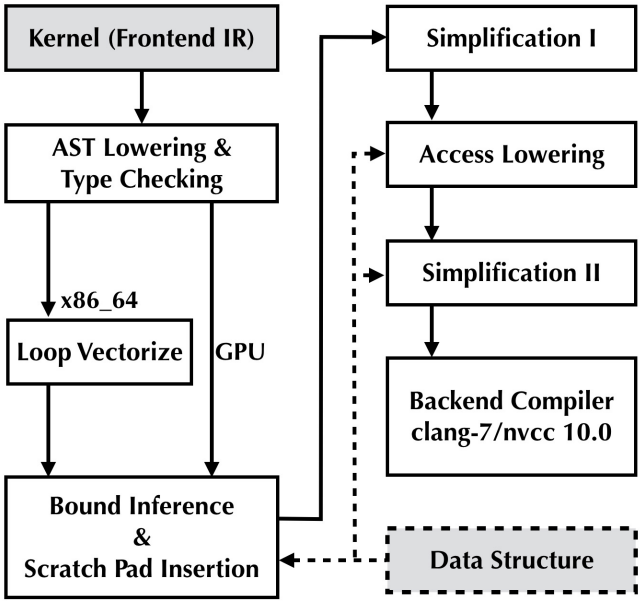


Fig. 10. The compilation pipeline. The solid lines represent our computation IR pipeline, while dotted lines indicate the use of data structure information.

Kernel launch management on GPU. Synchronizing GPU kernels with the CPU host can be quite costly. In our system, CPU-GPU synchronization (i.e., `cudaDeviceSynchronize()`) will only happen when the user explicitly calls the synchronization function or tries to read/write data from/to the data structure on GPU memory. This design makes asynchronized execution on GPUs transparent to the user.

5 COMPILER AND RUNTIME IMPLEMENTATION

The Taichi programming language is embedded in C++14, providing easy interoperability with the host language. We plan to release a Python 3 embedding to further lower the language learning barrier and development cost. The compiler is implemented in C++17, borrowing infrastructure from the Taichi library [Hu 2018]. The frontend kernel code is lowered to an intermediate representation before being compiled into standard C++ or CUDA code. Key components of our compiler and runtime are a two-phase simplifier for reducing instructions and removing redundant accesses, an access lowering transform, a customized memory management system for memory allocation and garbage collection, and a CPU loop vectorizer. The compilation workflow is summarized in Fig. 10.

Our intermediate representation follows the static single assignment design and is similar to LLVM [Lattner and Adve 2004]. Our intermediate representation is more high-level, containing explicit information about data structure accesses, such as the access index bounds and the size of the data structure element. This, combined with data structure composition information, makes it possible for our compiler to perform automatic access optimizations. The full list of intermediate representation nodes is described in Appendix A. We also include a snippet of compiled code in the supplementary material.

5.1 Simplification

Apart from the dedicated optimization for the data structure access, our simplification phase applies most common general-purpose compiler optimizations, such as common subexpression elimination, local variable store forwarding, dead instruction elimination, and lowering "if"-statements into conditional moves.

We split the simplification into two phases. The first phase greatly reduces and simplifies the number of instructions and makes it easier for the second simplification phase. In practice we have observed cases where disabling the first phase increases compilation time from a few seconds to tens of minutes. Removing "if"-statements yields bigger straightline code regions, enabling more potentially helpful optimizations.

Central to data structure access simplification are what we call *micro access* instructions: `OffsetAndExtractBit`, `SNodeLookup`, `GetCh`, and `IntegerOffset`. They are produced during the *access lowering* phase, where a root-to-leaf access (e.g. `x[i]`) is broken down into several stages for each level in the hierarchy. Since many different accesses share a similar path from root to leaf, similar micro access operations can be merged. As shown in Table 4, disabling the access lowering phase has a significant impact on performance.

The stages of moving down a single hierarchy in the data structure are as follows. First, offsets at each dimension are computed, along with the starting and ending position of each index represented as bit masks (`OffsetAndExtractBit`). This instruction is data-structure-aware. For example, if the kernel is 4-wide loop vectorized over index `j` and the child of the current block has a size larger than 4, we are guaranteed that `OffsetAndExtractBit` will return the same value for `j, j + 1, j + 2, j + 3`. Inference like this allows us to aggressively simplify accesses. Next, the extracted multi-dimensional indices are flattened into a linear offset (`Linearize`). Then a pointer to the item in the data structure is fetched from the current level of the data structure using the linear offset, along with a check of whether the node is active or not (`SNodeLookUp`). We need to pay special attention to `SNodeLookUp` when the node is not active: for read accesses `SNodeLookUp` returns an "ambient node" with all fields being ambient values such as 0; for write access `SNodeLookUp` first allocates the node and then return the new node. Finally the corresponding field in the item is fetched (`GetCh`).

In cases where two micro access instructions of the same type lead to a compile-time-known non-zero offset, we replace the second micro access instruction with an `IntegerOffset` instruction, representing the relationship between the two accesses in bytes, avoiding data structure traversals.

5.2 Memory Management

Our system relies heavily on the allocation-on-demand mechanism and supports data structures with dynamic topology. Therefore, efficient management of memory is a key to performance, especially on massively parallel GPUs.

Memory allocators for variable size requests usually need complex data structures to maintain available segments, leading to an unacceptable runtime cost. Therefore, we designed a memory management system that needs only very simple data structures, specialized for our abstraction.

The memory manager has a memory allocator tailored for each node that requires on-demand allocation, e.g. the pointer and hash nodes. The benefit of having multiple allocators is that each allocator only needs to allocate memory segments of a fixed size, which greatly simplifies and accelerates the process.

To minimize the internal data structure used by each memory allocator, we conservatively reserve a memory pool from our virtual address space, whose size is equal to the amount of physical memory. Only the actual used space will become a resident page in physical memory. This design allows us to implement memory allocation with a single integer atomic operation.

We make heavy use of the virtual memory system in modern operating systems, inspired by the SPGrid virtual memory design [Setaluri et al. 2014]. The runtime system will first reserve a virtual address space of size 2^{40}B = 1TB. The memory pages will not be allocated immediately, but in an on-demand manner, with pages zero-initialized by the hardware. We use the unified memory access feature on NVIDIA GPUs, thus this address space is shared by the CPU and GPU.

We additionally maintain a list of metadata for each block, including its memory location and coordinates.

5.3 Loop Vectorization on CPUs

We designed a loop vectorizer to utilize vector instruction sets such as SSE and AVX2/512 on modern CPUs. The design is similar to ISPC [Pharr and Mark 2012] where masking is used to avoid side effects of diverging control flow. We ensure that access to data structures is done through vectorized loads and writes whenever possible.

Vectorized memory access on CPUs. To achieve good memory behavior, it is necessary to issue vectorized memory operations instead of scalar loads.[3] We emit SIMD loads and then blend instructions to make maximum usage of the vector units, based on the compile-time-known offset information after the access simplifications (Fig. 11).

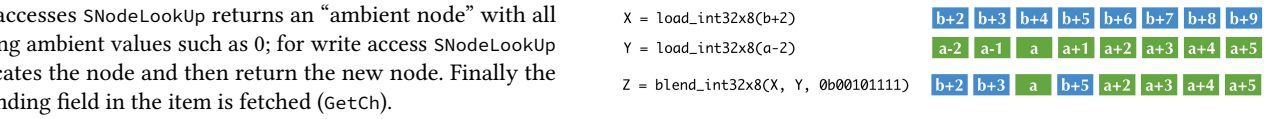


Fig. 11. Loading an 8-wide vector with elements $[x_{b+2}, x_{b+3}, x_a, x_{b+5}, x_{a+2}, x_{a+3}, x_{a+4}, x_{a+5}]$. The Taichi compiler, to utilize AVX instructions on x86 for high performance, only issues two vectorized loads that fetch contiguous data from memory, and then a SIMD blend to generate the desired vector, with binary mask "00101111". Note that a naive data loading code generator would issue one scalar load, one scalar to vector promotion, one vector shuffle, and finally one vector blend instruction, for *each* element in the vector.

5.4 Interaction with the Host Language

Our language can interact with the C++ host language easily. C++ can be used to initialize the data, invoke the compiled kernels, and possibly store the outputs. After a `Kernel`, `laplace`, has been defined, it can be used as follows:

[3] On GPUs this optimization is done via the memory coalescing hardware on the fly, relieving the compiler of the burden of this optimization.

Table 1. Benchmarks. Commands to reproduce our performance numbers are provided in detailed tables in each subsection. Geometric means of the four benchmarks where access has strong coherence are calculated for the summary. "-Opt" means with our domain-specific optimizations off, leaving the code generation and optimization to the backend general-purpose compiler; "+Opt" means with our optimizations on. If we include comparisons of our GPU backend with reference CPU implementations, we are on average 4.55× faster, otherwise 2.82× faster on the same hardware. Machine specifications for each benchmark are detailed in Appendix B. `clang-format-6.0` was used to reformat the code into the same style to get fair lines of code (LoC) numbers, with a right margin at 80 characters and all empty lines removed.

Benchmark	Reference Timing	CPU-Opt	CPU+Opt	GPU-Opt	GPU+Opt	Ref. LoC	Ours LoC
MLS-MPM	3.85ms (GPU, Pascal)	-	-	7.24ms	3.15ms	3091	237
FEM Linear Elasticity	30.71ms (CPU, AVX2)	182.19ms	17.16ms	11.78ms	2.11ms	267	21
MGPCG Solver	2.20s (CPU)	5.68s	2.98s	1.78s	1.13s	~ 2000	~ 300
Sparse CNN	37.44ms (GPU, Turing)	-	-	5.56 ms	3.02 ms	183	20
Summary (coherent cases) :	**Ours:Ref**=2.82×	**Opt On:Off**=3.02×		**GPU:CPU**=4.63×		~ 10.0× **shorter code**	
Volumetric Path Tracing	554.14s (CPU)	243.69s	232.52s	2.34s	2.35s	-	-

```
// Initialize
for (int i = 0; i < n; i++)
  for (int j = 0; j < 32; j++)
    x.val<float32>(i, i + j) = sin(j);

// Run the kernel on the active region
laplace();

// Output
printf("%f\n", y.val<float32>(n/2, n/2));
```

6 EVALUATION AND APPLICATIONS

In this section, we evaluate our language on end-to-end applications for large-scale visual computing tasks covering physical simulation, rendering, and 3D deep learning. The results are summarized in Table 1. In computation with coherent accesses, our domain-specific optimizations boost performance by a geometric mean of 3.02× on the same device. Our implementations require $\frac{1}{10}$ as many lines of code and run 2.82× faster than the reference implementations. The code for our implementations can be found in the supplementary material.

6.1 Moving Least Squares Material Point Method

The Material Point Method [Stomakhin et al. 2013; Sulsky et al. 1995] is a hybrid Eulerian-Lagrangian method, and is one of the state-of-the-art approaches for elastoplastic continuum simulation. The method is challenging to implement efficiently due to the interaction between particles and grids. Gao et al. [2018] implemented a high-performance Moving Least Squares Material Point Method [Hu et al. 2018] solver on GPU with intensive manual optimization[4], including

(1) A tailored SPGrid variant on GPUs;
(2) Staggered particle-block ownership (Fig. 15, left and middle) for parallel scattering, with shared memory utilization;
(3) Warp-level reductions to reduce atomic operations during scattering;
(4) Dedicated sorting and delayed reordering to reduce memory bandwidth consumption.

It took us a few attempts, but thanks to the easy data structure exploration supported by our language, we eventually surpassed their performance by 18%. We initially followed the structure of arrays (SOA) particle layout in their reference implementation. Although we are easily able to implement optimization (1) and (2) within ten lines of code (instead of hundreds in the reference implementation), the warp-level optimization (3) is below our level of abstraction[5], and we did not implement the complex sorting and reordering scheme (4) for simplicity. When particles are perfectly sorted, we were able to achieve comparable performance with the reference implementation. However, when the simulation progresses and the spatial distribution of particles changes, our performance drops drastically (Table 2, row "SOA"), especially when simulating liquids.

Table 2. [Gao et al. 2018] used an SOA particle layout that makes sequential access efficient, yet complex sorting and reordering schemes are needed. When particles' attributes are randomly shuffled in memory, the simulation runs 6.03× slower due to insufficient GPU cacheline utilization under random memory access. Our AOS particle layout is easy to implement and, more importantly, less sensitive to particle order, because even under random particle access order, different attributes of the particle stay in the same or nearby cacheline. Unlike CPUs, NVIDIA GPUs have no prefetching, so cacheline usage is key to performance, and access predictability is of less importance. This makes sorting unnecessary, leading to a much simpler and more efficient algorithm. [Reproduce: **ti mpm_benchmark particle_soa=[true/false] initial_shuffle=[true/false]**]

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

[4]We obtained their open-source CUDA solver and did further performance optimizations which made this reference implementation 1.98× faster, and carefully confirmed that we have achieved the best-human-effort performance following their design decisions.

[5]For portability, we do not provide warp-level intrinsics such as `__ballot`.

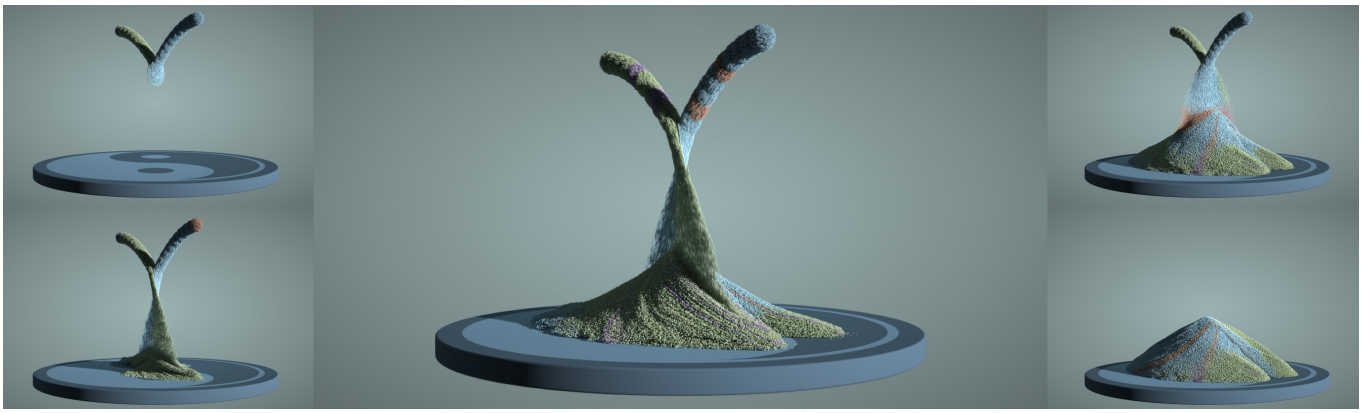


Fig. 12. A sand jet animation using MLS-MPM with up to 3 million particles, simulated using on average 2.2 sec/frame (100 substeps/frame). [Reproduce: **ti mpm_full scene=4 material=sand output=sand**]

Fortunately, using our language we were able to quickly explore different particle/grid layout schemes and found that switching particle layouts from structure of arrays to array of structures resolved this issue (Table 2, row "AOS"). In contrast, in the reference implementation the data layout is tightly coupled with the computational kernels, making it difficult to experiment with different data structures.

The data structure code for the high-performance data structure we found for MPM is illustrated in Figure 13. For particles we use array of structures, for grids we use structure of arrays, and each block maintains a list of indices of its contained particles. This greatly simplifies the data structure and algorithms used by Gao et al., for example we avoid the complex radix sort of the particles. The original grid hierarchy used by [Gao et al. 2018] is sparse yet bounded. This leads to simplicity and lower access cost, yet often leads to unnatural behavior when the simulation bound cannot be predetermined. In our language, adding a `hash` or `dense().pointer()` node at the top level of the grid conveniently makes the simulation domain virtually unbounded, which will suit corresponding boundary conditions (Figure 14).

Our implementation has only four kernels: sort particle indices to their containing blocks, particle to grid (P2G), grid normalization, and grid to particle (G2P). In contrast, the reference implementation has over 20 kernels, with the majority of them dealing with data structure maintenance. Our compiler automatically generates code to maintain the topology of the data structure. For example, it automatically activates a block and its parents when a particle touches it.

In the P2G and G2P kernels, we use the `AssumeInRange` construct to hint to the compiler the spatial relationship between blocks and their containing particles. We also apply Gao et al.'s stagger particle-grid ownership optimization by offsetting the particle position by Δx (Fig. 15), leading to a tighter access bound at the parent level. The compiler will automatically allocate scratchpads for each particle's $3\times3\times3$ span on each $4\times4\times4$ block, which is a $6\times6\times6$ scratchpad in shared memory. We did an ablation study on the scratchpad optimization, and it indeed leads to a significant speedup (Table 3).

```
auto i = Index(0), j = Index(1), k = Index(2);
auto p = Index(3);
auto &fork = root.dynamic(p, max_n_particles);
// Particle array of structures
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    fork.place(particle_F(i, j)); // 3x3 force matrix
// ... do the same for other particle attributes
// Grid structures of arrays
auto &block = root.dense({i, j, k}, n / grid_block_size)
    .pointer();
block.dense({i,j,k}, grid_block_size).place(grid_v(0));
block.dense({i,j,k}, grid_block_size).place(grid_v(1));
block.dense({i,j,k}, grid_block_size).place(grid_v(2));
block.dense({i,j,k}, grid_block_size).place(grid_m);
// Each block stores a list of particle indices
block.dynamic(p, pow(grid_block_size, 3) * 64).place(l);
```

Fig. 13. The data structure code for our material point method simulation. The interaction between particle and grid in this hybrid-Eulerian-Lagrangian approach leads to a huge space of potential data structure designs. We use array of structures for the particles and structure of arrays for the grids. We also store a dynamic list of particles in each block for speeding up particle lookup (the "Hierarchical Particle Buckets" in Fig. 6). We can easily modify the code to change the layout, or switch to a hash table for the top level of the grid to achieve an unbounded domain (Fig. 14).

Table 3. Using scratchpad memory (SPM, a.k.a. "shared memory" on NVIDIA GPUs) makes the P2G kernel 2.54× faster and G2P kernel 2.73× faster. In our language this optimization can be easily achieved with the "Cache" hint. [Reproduce: **ti mpm_benchmark use_cache=[true/false]**]

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

Examples of MLS-MPM sand and liquid animation simulated and rendered with Taichi programs are shown in Fig. 12 and Fig. 16.

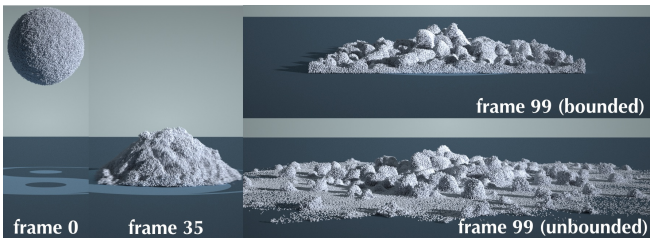


Fig. 14. Smashing a snow ball onto the ground: bounded vs. unbounded simulation. By changing data structures (and boundary conditions), we can easily switch to a virtually unbounded domain. [Reproduce: **ti mpm_full scene=1 scene=1 material=snow output=snow ground_friction=0.2 frame_dt=0.001 dt_mul=0.5 E=4e4 group_size=100 total_frames=200 bbox=[t/f]**]

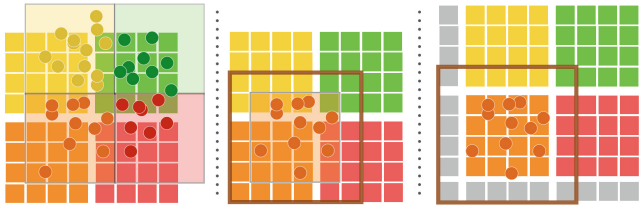


Fig. 15. Optimizing particle sorting. **Left:** We first sort the indices of particles to their respective grid blocks. P2G and G2P can then be done in a block-wise manner with high locality. **Middle:** We sort particles to a block staggered by Δx. Gao et al. [2018] describe a similar optimization: by doing so, particles sorted to each block will touch $2 \times 2 \times 2$ blocks only, instead of $3 \times 3 \times 3$ blocks in the case without staggering. **Right**: Note the extra grey cells without staggering. Since our compiler can automatically apply bounds inference, we quickly experimented with this approach and observed a 1.29× speed up. [Reproduce: **ti mpm_benchmark stagger=[t/f]**]

6.2 Linear Elasticity Finite Element Kernel

A large scale sparse grid-based finite element solver was presented by Liu et al. [2018] for high-resolution topology optimization. They proposed a matrix-free elasticity operator for the conjugate gradient iterations on x86_64 with vectorization. Their hand-optimized kernel is tailored for SPGrid [Setaluri et al. 2014], with carefully implemented vectorized load instructions (e.g. via the _mm256_loadu_ps intrinsic). This is a highly compute-bound task. For each voxel, over one thousand multiply and add instructions are issued, while fetching material parameters from only $2 \times 2 \times 2$ cells and 3D displacements from $3 \times 3 \times 3$ nodes. The whole algorithm is gather-only so it parallelizes naturally. We consider Liu et al.'s code a highly-optimized reference implementation for evaluating our language and compiler in a compute-bound situation.

We reproduced their algorithm in our language. Our compiler is especially good at compute-bound tasks, as our access optimization and auto-vectorization significantly reduce the number of instructions (Table 4). With all optimizations on, our implementation is 1.77× faster on an x86 CPU. Without modifying the code, our program runs on a GPU 8.2× faster than the generated CPU code, and 14.6× faster than the reference CPU implementation. We conduct a comprehensive ablation study of our compiler optimizations in Table 4, and found our compiler optimizations lead to 10.6× and 5.58× higher performance on CPU and GPU.

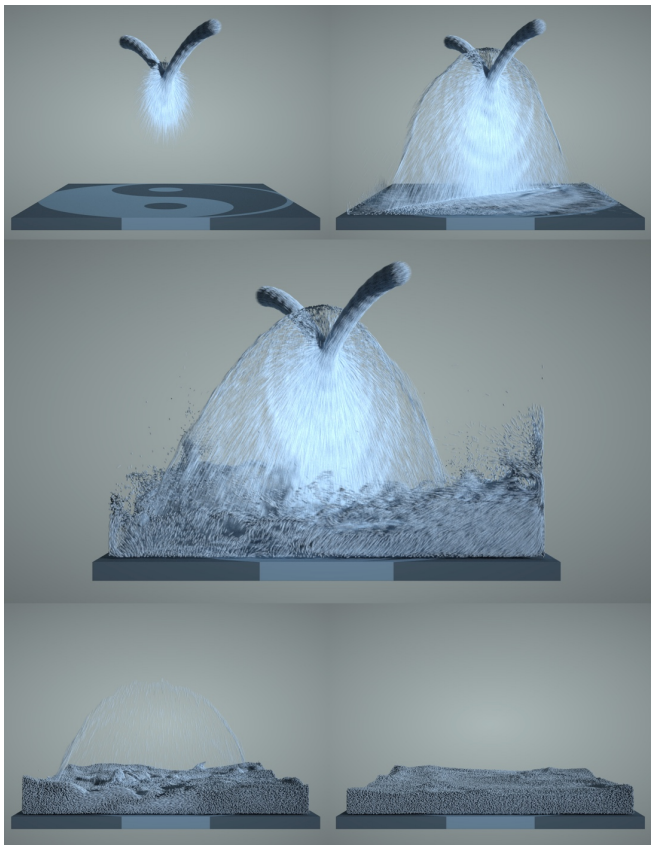


Fig. 16. A fluid animation using MLS-MPM with up to 3 million particles. [Reproduce: **ti mpm_full scene=3 material=fluid output=fluid dt_mul=0.7 bbox=true**]

6.3 Multigrid Poisson Solver

Large-scale Poisson equation solving has extensive use in graphics, including fluid simulation [Losasso et al. 2004], image processing [Agarwala 2007] and mesh reconstruction [Kazhdan et al. 2006]. We implement a Multigrid-Preconditioned Conjugate Gradients (MGPCG) solver [McAdams et al. 2010], which has become popular for pressure projection in physically based animation.

We implemented a simplified version of the reference implementation, with the following differences:

- Smoothers: the reference implementation uses Gauss-Seidel for boundary smoothing and damped Jacobi for interior smoothing, while we used red-black Gauss-Seidel smoothing for both boundary and interior regions.
- Restriction and prolongation: instead of using the $4 \times 4 \times 4$ trilinear interpolation operator, we use $2 \times 2 \times 2$ averaging.
- Boundary conditions: we support zero Dirichlet boundary conditions only, while the reference implementation also supports Neumann boundaries and their coarsening.

Table 4. An ablation performance study on the linear elasticity FEM kernel. Disabling the simplification I pass before lowering access does no harm to run-time performance, yet it increases compilation time from several seconds to 40 minutes when generating CPU code. Disabling the simplification II pass after lowering access leads to a binary size of 8.1MB instead of 377KB, since clang failed to remove redundant accesses. Using a bad data layout makes performance drop by nearly an order of magnitude. [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]**]

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

- Operator fusing: the reference implementation aggressively fuses operations, such as smoothing and dot products, to save memory bandwidth. We use temporary buffers to store some of these results to simplify the code.

Fully implementing McAdams et al.'s algorithm is possible in our language. For this specific benchmark we only need a simplified version.

A user experienced with both our language and physical simulation was able to implement our multigrid preconditioner within only 80 minutes and 300 lines of code.

We run both our implementation and reference on an x86 CPU until convergence. Our performance is 1.35× lower than the reference, likely because our implementation has a slightly inferior convergence rate and uses temporary buffers to simplify the code. On the other hand, changing our backend to GPU requires no effort, and it runs 2.64× faster than our CPU version and 1.9× faster than the reference implementation. An ablation study on our compiler optimizations and parallelization is shown in Table 5.

Table 5. An ablation performance study on the MGPCG Poisson solver. [Reproduce: **ti mgpcg_poisson gpu=[t/f] vec=[t/f] threads=[1-8] lower_access=[t/f] vec_load_cpu=[t/f]**]

Ablation	CPU Time	GPU Time
No multithreading	7.30s	-
No vectorization	4.01s	-
No access lowering	5.68s	1.78 s
All optimizations on	2.98s	1.13 s

Our solver automatically generalizes to an irregular and sparse case, while the reference implementation deals with only dense grids. To implement this multi-resolution approach, we generated a structural tree of 36 nodes and 31 kernels for a five-level multigrid hierarchy.

The performance data are obtained from a 256^3 dense grid, with zero Dirichlet surrounding voxels. The initial right-hand side is an analytical field $\sin(2\pi x)\cos(2\pi y)\sin(2\pi z)$, with $x, y, z \in [0, 1)$ being the spatial coordinates. Initial guesses for the conjugate gradient are set to zero. We stop iterating when the l_2 norm of residual is reduced by a factor of 10^6.

The same solver can potentially be used for other graphics applications such as panorama image stitching [Agarwala 2007] or mesh reconstruction [Kazhdan et al. 2006].

6.4 3D Convolutional Neural Networks

3D deep learning requires convolutional neural networks to operate on voxels instead of images. Unlike images, voxels require an efficient sparse representation for high-resolution 3D objects. Several sparse voxel approaches have been proposed for 3D deep learning [Graham et al. 2018; Riegler et al. 2017; Wang et al. 2017, 2018]. We implemented a 3D convolution layer, operating on multi-channel sparse voxels. The kernel is as simple as the mathematical definition of convolution, while the compiler automatically generates the code for efficiently accessing the sparse voxels. We compare to the Sparse Convolutional Network [Graham et al. 2018] implemented in CUDA, which uses a hash table with pointers to a dense matrix to store sparse 3D feature maps. We take the Stanford bunny, voxelize it into a $256 \times 256 \times 256$ grid, and copy over 16 channels. We then apply a $3 \times 3 \times 3 \times 16 \times 16$ convolution layer. By using a two-level hierarchy with pointer arrays, under 1% sparsity, we are roughly 12 times faster than the reference code. Under 10% sparsity, we are 23 times faster. We use the `CacheL1` schedule to cache the convolution weights in GPU L1 cache. This schedule hint boosts performance by 1.8×. [Reproduce: **ti cnn opt=[t/f] cache_l1=[t/f].**]

6.5 Volumetric Path Tracing

We implemented a volumetric path tracer inspired by Mitsuba [Jakob 2010]'s implementation, with Woodcock tracking [1965] and an isotropic phase function. We compare against the Tungsten renderer[6], which uses VDB [Museth 2013] to represent volumes.

The benchmark scene includes a $584 \times 576 \times 440$ density field containing bunny-shaped smoke and a single point light source. We rendered 512×512 images with 128 samples per pixel and a path length limit of 128. On CPU, our implementation is 2.38× faster than the reference implementation. Our GPU version is 98.86× faster than our CPU version and 235.6× faster than the reference implementation. [Reproduce: **ti smoke_renderer gpu=[t/f] opt=[t/f]**] Our domain-specific optimizations only lead to a 5% performance boost on CPU and no performance improvement on GPU, since the access pattern is largely incoherent in volume rendering. Still, we obtain a fast GPU renderer with no additional implementation, and are able to explore different sparse data structures.

[6] https://github.com/tunabrain/tungsten

We made our best effort to match our implementation to Tungsten's. With slight modifications to both renderers[7], we get qualitatively similar results (see the supplemental material).

7 LIMITATIONS

Although we in general get satisfactory results on the five benchmark cases, there are limitations and potential for future work:

Low arithmetic intensity tasks. In the material point method (Sec. 6.1) and finite element method (Sec. 6.2) cases, when the performance is compute-bound, our access optimizer can greatly improve performance by reducing access instructions. However, in the multigrid Poisson solver case (Sec. 6.3), although the optimizer improves performance by a factor of 1.9×, we are soon bounded by memory bandwidth. In these cases reducing instructions no longer helps. As a result, the unvectorized reference implementation is still faster than our vectorized implementation by 1.3×. This is because the reference is more bandwidth-efficient, due to operator fusing optimizations. This suggests investigating approaches that can automatically fuse operators, which might require further decoupling of computation and scheduling [Ragan-Kelley et al. 2012].

Less coherent accesses. For the volume rendering example (Sec. 6.5), while the rays exhibit some coherent behavior, our compiler is not able to infer this at compile time. Approaches that extract locality information at run-time such as ray reordering [Pharr et al. 1997] could potentially be used to boost performance.

8 RELATED WORK

8.1 Array Compilers

Many programming models for efficiently compiling array operations have been proposed.

Halide [Ragan-Kelley et al. 2012, 2013] decouples image processing operations and lower-level scheduling such as loop transformations and vectorization. Several polyhedral compilers adopt a similar idea [Baghdadi et al. 2015, 2019; Mullapudi et al. 2015; Vasilache et al. 2018]. All these compilers focus on dense data structures and do not model sparsity. Our language decouples algorithms from the internal organization of sparse data structures, allowing programmers to quickly switch between data organizations to achieve high performance.

Several sparse tensor compilers target linear algebra operations (e.g. taco [Chou et al. 2018; Kjolstad et al. 2017], ParSy [Cheshmi et al. 2017, 2018]). They focus on constructing efficient iteration spaces between different sparse matrices under linear algebra operations. Several compilers target graph operations such as breath-first-search or shortest path (e.g. [Wang et al. 2016; Zhang et al. 2018]). In contrast, we focus on generating high-performance traversal code for spatially coherent access to hierarchical and sparse data structures.

To efficiently vectorize access to data structures, we adopt the Single-Program-Multiple-Data model [Darema et al. 1988] in our computational kernels, which is the foundation of modern parallel languages such as CUDA, OpenCL [Stone et al. 2010], ispc [Pharr and Mark 2012], and IVL [Leißa et al. 2012].

Physical Simulation Languages. Several domain-specific languages exist for physical simulation. They usually abstract the domain as a graph structure for representing meshes. Liszt [DeVito et al. 2011] focuses on solving partial differential equations on meshes. Simit [Kjolstad et al. 2016] models the domain as sparse matrices while Ebb [Bernstein et al. 2016] employs a relational data model. We provide a different abstraction for lower-level optimizations, focusing on hierarchical sparse data structures.

8.2 Data-Oriented Design

Inspired by the increasing relative expense of memory operations, the video game and visual effects industries have recently started to adopt the data-oriented design philosophy [Acton 2014; Lee et al. 2017]. It is a software engineering approach focused on data access, as opposed to the more traditional object-oriented design where the storage is fragmented. Adopting a similar philosophy, ispc [Pharr and Mark 2012] and IVL [Leißa et al. 2012] both provide constructs for transformations between array of structures and structure of arrays. Our language facilitates data-oriented design and shares the same philosophy through decoupling of data structures and computation.

8.3 Hierarchical Sparse Grids in Graphics

Computer graphics, especially in the field of physical simulation, has a long history of using multi-level sparse regular grids for finite element methods, level set methods [Osher and Sethian 1988], or Eulerian fluid simulation. Sparse grids are used for representing large-scale simulation data. Bridson [2003] uses a two-level grid, Houston et al. [2006] use run-length-encoding to compress data. DT-Grid [Nielsen and Museth 2006] employs compressed-row-storage. VDB [Museth 2013] uses a static B+tree-like structure to represent an unbounded domain. SPGrid [Setaluri et al. 2014] uses a shallow hierarchy while utilizing the modern virtual memory system. GPU variants of VDB [Hoetzlein 2016; Wu et al. 2018] and SPGrid [Gao et al. 2018] have recently been designed. Nielsen and Bridson [2016] propose a wide-branching tile tree of voxels for fluid simulation. Bailey et al. [2013] sort particles to corresponding voxel blocks, similar to our "Hierarchical Particle Buckets" described in Section 6.1. Outside of simulation, Kazhdan et al. [2006] and Agarwala [2007] used octrees for solving Poisson's equation for image stitching and mesh reconstruction, respectively. Chen et al. [2013] use a hierarchical grid for storing signed distance fields.

9 CONCLUSION

We have presented a new programming language and its optimizing compiler for developing high-performance implementations of sparse visual computing tasks. Our novel design allows the language to provide both productivity and performance.

The computation-data structure decoupling allows the programmer to quickly explore different data structure hierarchies. As an example, we used this successfully to find a new efficient layout for the material point method, demonstrating the potential of the language for developing novel, high-performance data structures.

[7] We implemented Woodcock tracking in Tungsten, and used a two-level grid in our implementation to approximate the OpenVDB hierarchical DDA traversal [Museth 2014] in Tungsten.

Our compiler's automatic parallelization and access optimizations are especially useful in reducing the number of instructions for compute-bound tasks, while the scratchpad optimization improves memory locality. Our compiler enables programmers, for the first time, to implement optimized large-scale simulations within a few hundred lines of code.

ACKNOWLEDGMENTS

We thank Sylvain Paris for his insightful comments in the early stages of this work, and Eftychios Sifakis, Yunming Zhang, Andrew Adams and the anonymous reviewers for reading our manuscript and providing valuable feedback. This work was partly supported by the NSF/Intel Partnership on Computer Assisted Programming for Heterogeneous Architectures (CCF-1723445). This work was also partially supported by the Toyota Research Institute (TRI). However, this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity. Yuanming Hu is partly supported by a Snap Research fellowship.

REFERENCES

Mike Acton. 2014. Data-oriented design and C++. (2014). https://www.youtube.com/watch?v=rX0ItVEVjHc

Aseem Agarwala. 2007. Efficient Gradient-domain Compositing Using Quadtrees. *ACM Trans. Graph. (Proc. SIGGRAPH)* 26, 3 (2007), 94.

Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. 2015. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *Parallel Architecture and Compilation*. IEEE, 138–149.

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. *Code Generation and Optimization* (2019), 193–205.

Dan Bailey, Ian Masters, Matt Warner, and Harry Biddle. 2013. Simulating fluids using a coupled voxel-particle data model. In *ACM SIGGRAPH 2013 Talks*. ACM, 15.

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2 (2016), 21:1–21:12.

Robert Edward Bridson. 2003. *Computational Aspects of Dynamic Surfaces*. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Fedkiw, Ronald.

Jiawen Chen, Dennis Bautembach, and Shahram Izadi. 2013. Scalable Real-time Volumetric Surface Reconstruction. *ACM Trans. Graph. (Proc. SIGGRAPH)* 32, 4 (2013), 113:1–113:16.

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 13:1–13:13.

Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: Inspection and transformation of sparse matrix computations for parallelism. In *International Conference for High Performance Computing, Networking, Storage, and Analysis*. 62.

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 123.

Frederica Darema, David A George, V Alan Norton, and Gregory F Pfister. 1988. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Comput.* 7, 1 (1988), 11–24.

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. 2011. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 9.

Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana-Tampubolon, Eftychios Sifakis, Yuksel Cem, and Chenfanfu Jiang. 2018. GPU Optimization of Material Point Methods. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 32, 4 (2018), 102.

Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 2018. 3D semantic segmentation with submanifold sparse convolutional networks. In *Computer Vision and Pattern Recognition*. 9224–9232.

Rama Karl Hoetzlein. 2016. GVDB: Raytracing sparse voxel database structures on the GPU. In *Proceedings of High Performance Graphics*. Eurographics Association, 109–117.

Ben Houston, Michael B. Nielsen, Christopher Batty, Ola Nilsson, and Ken Museth. 2006. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Trans. Graph.* 25, 1 (2006), 151–175.

Yuanming Hu. 2018. Taichi: An open-source computer graphics library. *arXiv preprint arXiv:1804.09293* (2018).

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 4 (2018), 150.

Wenzel Jakob. 2010. Mitsuba renderer. (2010). http://www.mitsuba-renderer.org.

Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. 2006. Poisson surface reconstruction. In *Eurographics Symposium on Geometry Processing*, Vol. 7.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 77.

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A language for physical simulation. *ACM Trans. Graph.* 35, 2 (2016), 20:1–20:21.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization*.

Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. 2017. Vectorized Production Path Tracing. In *High Performance Graphics*.

Roland Leißa, Sebastian Hack, and Ingo Wald. 2012. Extending a C-like Language for Portable SIMD Programming. *SIGPLAN Not.* 47, 8 (2012), 65–74.

Haixiang Liu, Yuanming Hu, Bo Zhu, Wojciech Matusik, and Eftychios Sifakis. 2018. Narrow-band Topology Optimization on a Sparsely Populated Grid. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 251:1–251:14.

Frank Losasso, Frédéric Gibou, and Ron Fedkiw. 2004. Simulating water and smoke with an octree data structure. In *ACM Trans. Graph. (Proc. SIGGRAPH)*, Vol. 23. ACM, 457–462.

Aleka McAdams, Eftychios Sifakis, and Joseph Teran. 2010. A parallel multigrid Poisson solver for fluids simulation on large grids. In *Symposium on Computer Animation*. ACM/Eurographics Association, 65–74.

Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News* 43, 1 (2015), 429–443.

Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.* 32, 3 (2013), 27.

Ken Museth. 2014. Hierarchical digital differential analyzer for efficient ray-marching in OpenVDB. (2014).

Michael B Nielsen and Robert Bridson. 2016. Spatially adaptive FLIP fluid simulations in bifrost. In *ACM SIGGRAPH 2016 Talks*. ACM, 41.

Michael B. Nielsen and Ken Museth. 2006. Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets. *J. Sci. Comput.* 26, 3 (2006), 261–299.

Stanley Osher and James A. Sethian. 1988. Fronts Propagating with Curvature-dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *J. Comput. Phys.* 79, 1 (1988), 12–49.

Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering Complex Scenes with Memory-coherent Ray Tracing. In *SIGGRAPH*. ACM, 101–108.

Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing*. 1–13.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph. (Proc. SIGGRAPH)* 31, 4 (2012), 32:1–32:12.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530.

Gernot Riegler, Ali Osman Ulusoy, and Andreas Geiger. 2017. Octnet: Learning deep 3d representations at high resolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3577–3586.

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 33, 6 (2014), 205.

Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 102.

John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

Deborah Sulsky, Shi-Jian Zhou, and Howard L Schreyer. 1995. Application of a particle-in-cell method to solid mechanics. *Computer physics communications* 87, 1-2 (1995),

236–252.
Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv:1802.04730* (2018).
Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, and Xin Tong. 2017. O-CNN: Octree-based convolutional neural networks for 3D shape analysis. *ACM Transactions on Graphics (SIGGRAPH)* 36, 4 (2017).
Peng-Shuai Wang, Chun-Yu Sun, Yang Liu, and Xin Tong. 2018. Adaptive O-CNN: A patch-based deep representation of 3D shapes. *ACM Transactions on Graphics (SIGGRAPH Asia)* 37, 6 (2018).
Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. *SIGPLAN Not.* 51, 8 (2016), 11:1–11:12.
E Woodcock, T Murphy, P Hemmings, and S Longworth. 1965. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Applications of Computing Methods to Reactor Problems*, Vol. 557.
Kui Wu, Nghia Truong, Cem Yuksel, and Rama Hoetzlein. 2018. Fast fluid simulations with sparse volumes on the GPU. In *Computer Graphics Forum (Proc. Eurographics)*, Vol. 37. Wiley Online Library, 157–167.
Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 121.
Yongning Zhu and Robert Bridson. 2005. Animating sand as a fluid. *ACM Transactions on Graphics (TOG)* 24, 3 (2005), 965–972.

A INTERMEDIATE REPRESENTATION INSTRUCTIONS

Our intermediate representation follows the typical static single assignment form. It contains the following control flow nodes: `StructFor` (for looping data structures), `RangeFor` (for looping data over ranges), `If`, `While`, `WhileControl` (while combined with break).

The expression tree contains the following nodes: `Const`, `Alloca` (for local mutable variables), `UnaryOp`, `BinaryOp`, `TrinaryOp`, `Rand`, `ElementShuffle` (for loop vectorization), `RangeAssumption` (for bound inference).

When a data structure is accessed, `GlobalLoad` and `GlobalStore` are issued with addresses pointed to by `GlobalPtr`. `SNodeOp` is used to activate grids and check for sparsity. When a local mutable variable is accessed, `LocalStore` and `LocalLoad` are issued. Atomic instructions are represented by `AtomicOp`. `ClearAll` cleans up the data structures.

As mentioned in Sec. 5.1, `GlobalPtr` is lowered to the following micro access nodes, to facilitate expression simplification:

- `OffsetAndExtract`
- `Linearize`
- `SNodeLookup`
- `GetCh`
- `IntegerOffset`

B BENCHMARK MACHINE SPECIFICATIONS

Here we list the machine specifications for our benchmarks for reproducing the performance numbers.

The MLS-MPM, FEM and MGPCG benchmarks were done on an Intel Core i7-7700K CPU with four cores at 4.2GHz, 32 GB main memory, and an NVIDIA GTX 1080Ti graphics card.

The CNN benchmark was done on an Intel Core-i7 9800X CPU with eight cores at 3.8GHz, 32 GB main memory, and an NVIDIA RTX 2080 GPU.

The rendering benchmark was done on an Intel Xeon E5-2690 v4 with 28 cores at 2.60GHz, 64 GB main memory, and an NVIDIA Tesla V100 GPU.

`clang-7` and `nvcc 10.0` were used as backend compilers.

Although finding a machine with these exact specifications may be difficult, the relative performance numbers are roughly machine-independent. We encourage the reader to run the example programs with the provided commands to reproduce our results, and to explore different combinations of data structures and compiler optimizations.