Life of a Taichi Kernel

A trip through Taichi's internal design and implementation

Yuanming Hu

August 24, 2020

Before we start: interacting with this file

This PDF file has links that point you to more details:

- C++ source code, e.g., ▸`program/program.h`
- Python source code, e.g., ▸`lang/kernel.py`
- Documentation, e.g., ▸doc:Hello, world!

Overview

This talk (or design doc) briefly covers some internal design decisions of Taichi, for developers and users who want to dig deeper into Taichi. Topics include

- How does Taichi work?
- Why do we end up the current design decisions?
- What do I need to know to improve Taichi?

Setting up Taichi locally

- Installation via pip: `python3 -m pip install taichi`
- Building from source (or Docker) for development: ▸doc:Developer installation

Pick one installation only

Avoid having both `pip`-installed Taichi and the version build from source.

Table of Contents

What is Taichi?

High-performance domain-specific language (DSL) embedded in **Python**, for **computer graphics** applications

- **Productivity** and **portability**: easy to learn, to write, and to share
- **Performance:** data-oriented, parallel, megakernels
- **Spatially sparse** programming: save computation and storage on empty regions
- **Decouple** data structures from computation
- **Differentiable** programming support

Taichi v.s. deep learning frameworks

Why is Taichi different from TensorFlow, PyTorch, NumPy, JAX, ... ?

Quick answer: Taichi uniquely supports **megakernels** and **spatial sparsity**.

Longer answer: Those systems serve their own application domains (e.g., convolutional neural networks) very well, but their design decisions surrounding **immutable, dense tensors** (e.g., feature maps) with **simple, regular operators** (e.g., element-wise add and 2D convolutions) do not serve well more irregular computational patterns, including

- Computer graphics, including physical simulation and rendering
- Irregular neural network layers (e.g., gathering/scattering) that are emerging
- General differentiable programming cases

Without Taichi people tend to manually write CUDA or abuse deep learning programming interfaces. Taichi offers performance, productivity, and portability in those cases.

Hello, world! (Julia set, $z \leftarrow z^2 + c$)

```
import taichi as ti
ti.init(arch=ti.gpu)
n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

More details: ▸doc:Hello, world! Run it: `ti example fractal`

Table of Contents

Taichi's frontend

Taichi's old C++ frontend

Taichi used to be embedded in C++14. However, that solution is mostly abandoned, because

1. C++ itself is too complex for most users to learn, not to say a DSL embedded in C++. E.g., ▸`math/svd.h` ☹
2. Getting C++ AST is almost impossible. We had to heavily use templates/macros tricks, which harm readability. ☹

Taichi's new Python frontend

Now the whole Taichi system is deeply embedded in Python.

1. Python is easy to learn and widely adopted. ☺
2. Python allows flexible AST inspection and manipulation. ☺

Both of these allow us to invent a new high-performance language out of Python.

Other goodies of the Python frontend

- Easy to run. No ahead-of-time compilation is needed.
- Reuse and interact with existing python infrastructure:
  1. IDEs such as PyCharm.
  2. Package manager (pip)
  3. Existing packages such as `matplotlib` and `numpy`
- The built-in AST manipulation tools (`import ast`) in python allow us to do magical things, as long as the kernel body can be parsed by the Python parser.

Kernels marked with `@ti.kernel` will be compiled into a Taichi AST and then to parallel kernels on CPU/GPUs.

Generating a Taichi AST from Python AST

(i) Taichi kernel (Python) $\longrightarrow$ Taichi AST generator (Python)

Taichi has a series of Python AST transformers ▸`lang/transformer.py` that transforms a Taichi kernel (in Python) into another Python script, which is a Taichi AST generator.

(ii) Taichi AST generator (Python) $\longrightarrow$ Taichi AST (C++)

The Taichi AST generator is a Python script that calls AST builder functions in C++ (exported via pybind11) when executed. The result of this step is a Taichi AST.

Confused? Let's take a look at an example ☺

Generating a Taichi AST (example)

Example

```
import taichi as ti

ti.init(print_preprocessed=True)

@ti.kernel
def foo():
    for i in range(10):
        if i == 2:
            print(i)

foo()
```

Inspecting Taichi AST transforms

Set `print_preprocessed=True` in `ti.init` to make Taichi print out processed AST.

Generating a Taichi AST (example)

```
# (1) Input kernel (Python function)
@ti.kernel
def foo():
    for i in range(10):
        if i % 2 == 0:
            print(i)
```

```
# (2) Transformed AST generator (another Python function)
def foo():
    import taichi as ti
    if 1:
        i = ti.Expr(ti.core.make_id_expr(''))
        ___begin = ti.Expr(0)
        ___end = ti.Expr(10)
        ___begin = ti.cast(___begin, ti.i32)
        ___end = ti.cast(___end, ti.i32)
        ti.core.begin_frontend_range_for(i.ptr,
            ___begin.ptr, ___end.ptr)
        if 1:
            __cond = ti.chain_compare([i, 2], ['Eq'])
            ti.core.begin_frontend_if(ti.Expr(__cond).ptr)
            ti.core.begin_frontend_if_true()
            ti.ti_print(i)
            ti.core.pop_scope()
            ti.core.begin_frontend_if_false()
            ti.core.pop_scope()
        ti.core.end_frontend_range_for()
        del i # Note: Taichi has lexical scoping!
```

```
# (3) Generated Taichi AST
kernel {
  $0 : for @tmp0 in range(
     (cast_value<int32> 0),
     (cast_value<int32> 10))
     block_dim=adaptive {
    $1 : if (1 & (@tmp0 == 2)) {
      $2 = eval @tmp0
      print %2, "\n"
    } else {
    }
  }
}
```

Note

Taichi (frontend) AST $\neq$ Taichi (middle-end) IR

Table of Contents

Taichi intermediate representation (IR): Two components

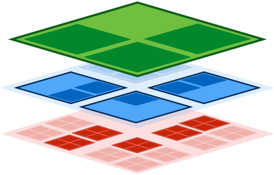
1. Computation IR

- Static-single assignment
- Hierarchical (instead of CFG+BB)
- Differentiable
- Statically and strongly typed

```
$0 = offloaded range_for(0, 10)
body {
  <i32 x1> $1 = loop $0 index 0
  <i32 x1> $2 = const [3]
  <i32 x1> $3 = cmp_eq $1 $2
  <i32 x1> $4 = const [1]
  <i32 x1> $5 = bit_and $3 $4
  $6 : if $5 {
    print $1, "\n"
  }
}
```

2. Structural Node (SNode) IR

- Describes data organization
- Tree-structured
- Spatial sparsity
- Favors powers of two



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

IR design goals: a) enable domain-specific optimizations for sparse computation. b) decouple data layout from computation (more on this later).

Progressive compilation

1. Taichi compiler progressively compiles a frontend AST into executable CPU/GPU kernels.
2. During this process, $30+$ compilation passes are applied to the Taichi AST/IR.
3. The only big step is from Taichi AST to Taichi IR (AST lowering ▸`transforms/lower_ast.cpp`), where two different sets of statements (instructions) are used.
4. Each other pass makes a small step towards hardware-friendly code only. Almost the same sets of statements are used for input/output Taichi IR.
5. There is **no** clear separation of IR statements (e.g. high-level, mid-level, low-level IR) after AST lowering.

10 key compilation passes ▸transforms

Taichi's IR transformation passes gradually converts an input Taichi AST to parallel executable kernels ▸transforms/compile_to_offloads.cpp.

1. Taichi frontend AST to Taichi IR
   1. Lower Taichi AST to SSA ▸transforms/lower_ast.cpp
   2. Type checking ▸transforms/type_check.cpp
2. Taichi IR to offloaded tasks
   1. (Optional) Automatic differentiation ▸transforms/auto_diff.cpp
   2. (Optional) Insert bound checks ▸transforms/check_out_of_bound.cpp
   3. Flag and weaken access ▸transforms/flag_access.cpp
   4. Automatic parallelization ▸transforms/offload.cpp
3. Offloaded tasks to executable
   1. Demote dense struct-fors to range-fors ▸transforms/demote_dense_struct_fors.cpp
   2. Create thread local storage ▸transforms/make_thread_local.cpp
   3. Create block local storage ▸transforms/make_block_local.cpp
   4. Lower access ▸transforms/lower_access.cpp

(Optimization passes are skipped to save space)

Statements

Taichi IR has ~ 70 statements.

Typical statements include:

- Arithmetic ▸`ir/ir.h`
  1. `UnaryOpStmt`. Operators ▸`inc/unary_op.inc.h` (`sqrt`, `sin`, ...)
  2. `BinaryOpStmt`. Operators ▸`inc/binary_op.inc.h` (`add`, `mul`, ...)
  3. ...
- Memory access: `Global[Ptr/Load/Store]Stmt`
- AutoDiff Stack operations (more on this later):
  `Stack[Alloca/LoadTop/LoadTopAdj/Pop/Push/AccAdjoint]Stmt`
- SNode Micro Ops (after the `lower_access` pass)
- ...

General-purpose optimization passes

Why optimize Taichi IR? Can't we leave the job to LLVM?

Quick answer:

- Higher optimization quality (next slide)
- Fewer instructions to LLVM: faster JIT

General-purpose optimization passes

- Control-flow based optimizations: CSE, DIE, ...
  ▸`transforms/cfg_optimization.cpp`
- Constant folding ▸`transforms/constant_fold.cpp` (more details later)
- Algebraic simplification ▸`transforms/alg_simp.cpp`
- ...

Why not fully trust backend compilers?

Long story short: backend compilers (e.g., LLVM) do not have as much information as the Taichi compiler do.
For example, Taichi IR and optimizer

1. ...can do index analysis
2. ...have a tailored instruction granularity
3. ...have strict data access semantics

Therefore traditional compilers often fail to do certain important optimizations.

Benchmarks show Taichi IR optimization passes (mostly on data accesses) make programs $3.02\times$ faster[1], even if we use `-O3` on backend compilers.

[1] Y. Hu et al. (2019). "Taichi: a language for high-performance computation on spatially sparse data structures". In: *ACM Transactions on Graphics (TOG)* 38.6, pp. 1–16.

The IR granularity spectrum

```
x[i, j]
```

End2end access

```
access1(i,j)
access2(i,j)
```

Level-wise Access

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

Taichi IR

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

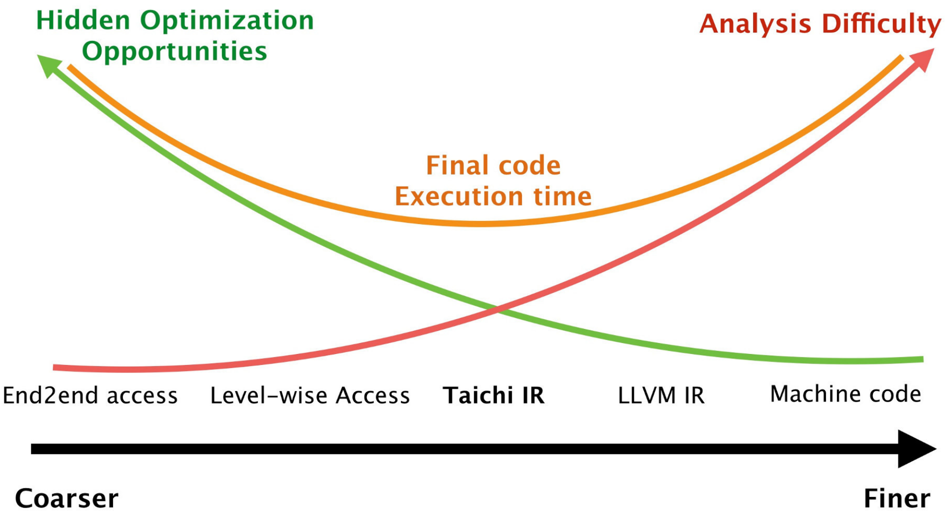
LLVM IR

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

Machine code

Coarser → **Finer**

The IR granularity trade-offs



Data access semantics in Taichi

Seemingly trivial assumptions can enable effective compiler optimization and make programmers' life easier. E.g.,

1. No pointer aliasing: `a[x, y]` and `b[i, j]` never overlaps unless `a` and `b` are the same field ▸`analysis/alias_analysis.cpp`
2. All memory accesses are done through `field[indices]` syntax
   - Pointers that can flexibly point to anything arguably makes optimization harder
3. The only way data structures get modified, is through write accesses of form `field[indices]`
4. Read access **does not** modify anything
   - No memory allocation
   - No exception if element does not exist

Example general-purpose opt pass: Constant folding

Taichi's constant folding pass uses JIT evaluation on the device ▸`transforms/constant_fold.cpp`

1. Generate a small Taichi kernel that only does the simple unary/binary operation. E.g, `f32 + f32`.
2. Evaluate the small kernel on operands
3. Replace the arithmetic statement with a constant statement with value being the result.

Evaluating on the device for correctness

Certain operations (e.g., `sin(x)`) on host evaluation (e.g., x64) may generate a different result from device (e.g., OpenGL) evaluation.
For correctness, do constant folding evaluation on devices. (... and assume no `fastmath`.)

Constant folding (example)

Madhava's series:

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-\frac{1}{3})^{k}}{2k+1} = \sqrt{12}\left(1 - \frac{1}{3\cdot 3} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

```python
import taichi as ti
ti.init(print_ir=True)

@ti.kernel
def calc_pi() -> ti.f32:
    s = 0.0
    c = 1.0
    for i in ti.static(range(10)):
        s += c / (i * 2 + 1)
        c *= -1/3
    return s * ti.sqrt(12.0)

print(calc_pi())
```

Constant folding (example)

Madhava's series:

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-\frac{1}{3})^{k}}{2k+1} = \sqrt{12}\left(1 - \frac{1}{3\cdot 3} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

Generated Taichi IR:

```
kernel {
  $0 = offloaded  {
    <f32 x1> $1 = const [3.1415904] # All computation folded
    <f32 x1> $2 : kernel return $1
  }
}
```

Offloading (automatic parallelization)

The `offload` pass ▸`transforms/offload.cpp` decomposes kernels into **offloaded statements** (`OffloadedStmt`). Each `OffloadedStmt` have one of the following types:

1. `serial`: Simple serial code
2. `range_for`: Parallel range-for
3. `clear_list`: Clear the list of nodes for `struct_for`
4. `listgen`: Generate list of nodes for `struct_for`
5. `struct_for`: Parallel struct-for
6. `gc`: Garbage collection

Offloading (automatic parallelization transform)

Challenge: outside variables are visible to loop bodies.
Solution: create global temporary variables (`global tmp var`)

```
import taichi as ti

ti.init(
    print_ir=True,
    use_thread_local=False
)

@ti.kernel
def bar():
    a = 0
    for i in range(10):
        a += i
    print(a)

bar()
```

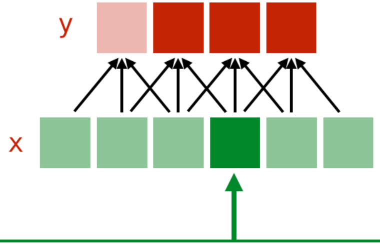
Note: `a` is accessible in the loop body (parallel), but its declaration (serial)/printing (serial) is outside.

```
kernel {
  $0 = offloaded serial
  body {
    <i32*x1> $1 = global tmp var (offset = 0 B)
    <i32 x1> $2 = const [0]
    <i32*x1> $3 : global store [$1 <- $2]
  }
  $4 = offloaded serial range_for(0, 10) grid_dim=0
       block_dim=32
  body {
    <i32 x1> $5 = loop $4 index 0
    <i32*x1> $6 = global tmp var (offset = 0 B)
    <i32 x1> $7 = atomic add($6, $5)
  }
  $8 = offloaded serial
  body {
    <i32*x1> $9 = global tmp var (offset = 0 B)
    <i32 x1> $10 = global load $9
    print $10, "\n"
  }
}
```

Block local storage (BLS)

▸transforms/make_block_local.cpp

```
for i in x:
    y[i] = x[i - 1] - 2 * x[i] + x[i + 1]
```

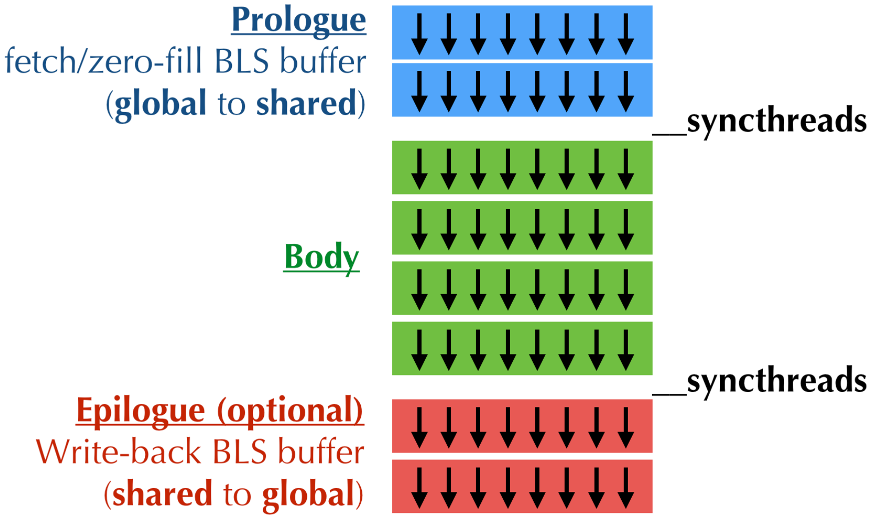


3x global memory accesses for a single element

To reduce global memory accesses (using block local storage):

Each block first fetches all the needed **x**'s to a **block local** buffer. When evaluating **y**, read from the block local buffer instead of from global memory.

Block local storage (BLS): Workflow on CUDA



Block local storage (BLS): Productivity

CUDA

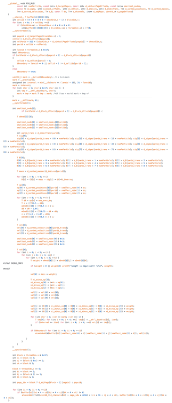
```
__global__ void laplace_shared(float *a, float *b) {
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;

  unsigned int tid = blockDim.y * threadIdx.x + threadIdx.y;

  __shared__ float pad[bs + 2][bs + 2];
  auto pad_size = (bs + 2) * (bs + 2);

  if (bs <= i && i < N - bs && bs <= j && j < N - bs) {
    while (tid < pad_size) {
      int si = tid / (bs + 2);
      int sj = tid % (bs + 2);
      int gi = si - 1 + blockIdx.x * blockDim.x;
      int gj = sj - 1 + blockIdx.y * blockDim.y;
      pad[si][sj] = a[gi * N + gj];
      tid += blockDim.x * blockDim.y;
    }
  }

  __syncthreads();

  if (bs <= i && i < N - bs && bs <= j && j < N - bs) {
    auto ret = -4 * pad[threadIdx.x + 1][threadIdx.y + 1] +
               pad[threadIdx.x + 2][threadIdx.y + 1] +
               pad[threadIdx.x][threadIdx.y + 1] +
               pad[threadIdx.x + 1][threadIdx.y + 2] +
               pad[threadIdx.x + 1][threadIdx.y];
    b[i * N + j] = ret;
  }
}
```

Taichi

```
@ti.kernel
def laplace():
    ti.cache_shared(a)
    for i, j in a:
        b[i, j] = -4 * a[i, j] - a[i - 1][j] - \
                  a[i][j - 1] - a[i + 1][j] - a[i][j + 1]
```

Block local storage (BLS): Productivity

CUDA



Taichi

```python
ti.cache_shared(grid_m, grid_v)
for I in ti.gropued(pid): # Particle state update and scatter to grid (P2G)
    p = pid[I]
    base = ((x[p] + Offset) * inv_dx - 0.5).cast(int)
    for d in ti.static(range(3)):
        base[d] = ti.assume_in_range(base[d], I[d], 0, 1)
    fx = (x[p] + Offset) * inv_dx - base.cast(float)
    w = [0.5 * ti.sqr(1.5 - fx), 0.75 - ti.sqr(fx - 1), 0.5 * ti.sqr(fx - 0.5)]
    stress = ...
    stress = (-dt * p_vol * 4 * inv_dx * inv_dx) * stress
    affine = stress + p_mass * C[p]
    for i, j, k in ti.static(ti.ndrange(3, 3, 3)): # Loop over 3x3x3 grid
        offset = ti.Vector([i, j, k])
        dpos = (offset.cast(float) - fx) * dx
        weight = w[i][0] * w[j][1] * w[k][2]
        grid_v[base + offset] += weight * (p_mass * v[p] + affine @ dpos)
        grid_m[base + offset] += weight * p_mass
```

Data types

Taichi is statically and strongly and typed. Supported types include

- Signed integers: `ti.i8/i16/i32/i64`
- Unsigned integers: `ti.u8/u16/u32/u64`
- Float-point numbers: `ti.f32/f64`

`ti.i32` and `ti.f32` are the most commonly used types in Taichi. Boolean values are represented by `ti.i32` for now.

Data type compatibility

The CPU and CUDA backends support all data types. Other backend may miss certain data type support due to backend API constraints. See the documentation for more details.

Fields

Taichi is a *data-oriented* programming language where **fields** are first-class citizens.

- Fields are essentially multi-dimensional arrays
- An element of a field can be either a scalar (`ti.field`), a vector (`ti.Vector.field`), or a matrix (`ti.Matrix.field`)
- Field elements are *always* accessed via the `a[i, j, k]` syntax. (No pointers.)
- Access out-of-bound is undefined behavior in non-debug mode
- *(Advanced) Fields can be spatially sparse*

Manipulating data layouts

- **Data layout** refers to the `dimensionality, shape, memory layout, and sparsity structure` of `fields.`
- A carefully designed data layout can significantly improve cache/TLB-hit rates and cacheline utilization.
- Taichi decouples algorithms from data layouts, and the Taichi compiler automatically optimizes data accesses on a specific data layout.
- These Taichi features allow programmers to quickly experiment with different data layouts and figure out the most efficient one on a specific task and computer architecture.
- In Taichi, the layout is defined in a recursive manner.

(From now on we focus on scalar fields `ti.field`.)

Data layout in three languages

Taichi (Python)

```
x = ti.field(dtype=ti.i32);
ti.root.dense(ti.i, 16).place(x)
```

Equivalent C++

```
int x[16];
```

Natural language

- `ti.root` is the root data structure.
- Dot (`A.B`) means "Each cell of `A` has `B`"
- `dense(ti.i, 16)` means "a `dense` container with 16 cells along the `ti.i` axis"
- `place` means "... `field` ..."

"Each cell of `root` has a `dense` container with 16 cells along the `ti.i` axis. Each cell of `dense` has `field` `x`."

More data layouts

Structure of arrays (SOA):

Taichi (Python)

```
ti.root.dense(ti.i, 16).place(x)
ti.root.dense(ti.i, 16).place(y)
```

Equivalent C++

```
int x[16];
int y[16];
```

Array of structures (AOS):

Taichi (Python)

```
ti.root.dense(ti.i, 16).place(x, y)
# or equivalently
point = ti.root.dense(ti.i, 16)
point.place(x)
point.place(y)
```

Equivalent C++

```
struct Point {int x, y}
Point points[16];
```

Nesting: Array of structures of arrays (AOSOA) example

Taichi (Python)

```
chunk = ti.root.dense(ti.i, 2)

chunk.dense(ti.i, 8).place(x)
chunk.dense(ti.i, 8).place(y)
```

Equivalent C++

```
struct PointChunk {
    int x[8]; int y[8];
};
PointChunk points[2];
```

Discussions

In Taichi, you can always access `x` and `y` using syntax like `x[5]` or `y[7]`.
In C++ the access is data-layout dependent.

This is just for dense data. For sparse data it is even trickier in C++.

Higher dimensionality

Taichi (Python)

```
ti.root.dense(ti.ij, (4, 8)).place(x)
```

Equivalent C++

```
int x[4][8];
```

Taichi (Python)

```
block = ti.root.dense(ti.ij, 2)
block.dense(ti.ij, 4).place(x)
# or simply
ti.root.dense(ti.ij, 2)
        .dense(ti.ij, 4).place(x)
```

Equivalent C++

```
struct Block {
    int x[4][4];
};
Block blocks[2][2];
```

Structural Nodes (SNodes) ▸ir/snode.h

Currently supported SNode types ▸inc/snodes.inc.h:

1. `root`: The root of the data structure.
2. `dense`: A fixed-length contiguous array.
3. `bitmasked`: similar to `dense`, but it also uses a mask to maintain sparsity information, one bit per child.
4. `pointer`: Store pointers instead of the whole structure to save memory and maintain sparsity.
5. `dynamic`: Variable-length array, with a predefined maximum length. It serves the role of `std::vector` in C++ or `list` in Python.

Access lowering ▸transforms/lower_access.cpp

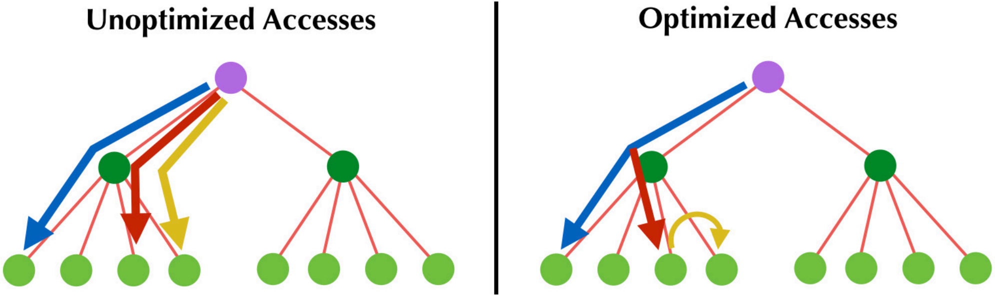


Figure: Access lowering breaks down end-to-end accesses into level-by-level micro-access operations. **Left:** An example unoptimized access; **Right:** After access lowering optimization passes can merge redundant data structure accesses.

Table of Contents

OpenVDB[2]

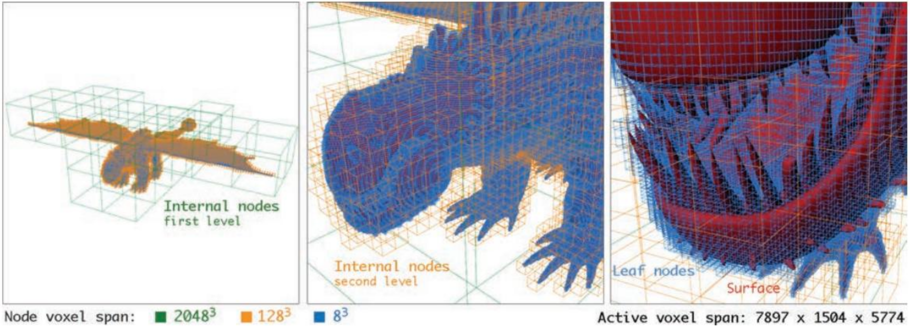
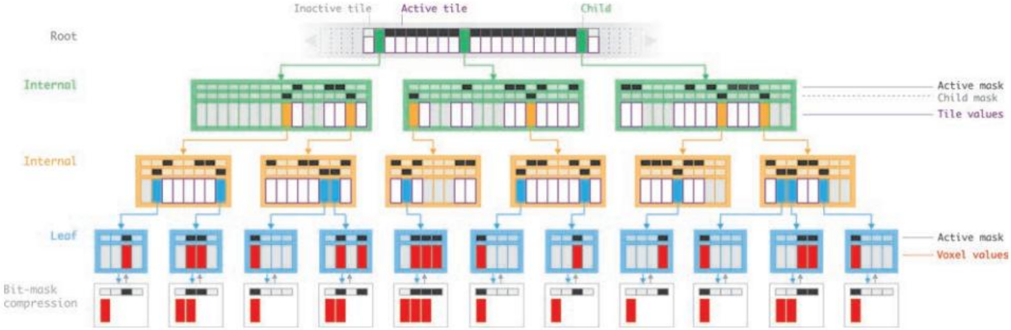


Fig. 4. High-resolution VDB created by converting polygonal model from *How To Train Your Dragon* to a narrow-band level set. The bounding resolution of the 228 million active voxels is $7897 \times 1504 \times 5774$ and the memory footprint of the VDB is 1GB, versus the $\frac{1}{4}$ TB for a corresponding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

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

OpenVDB[3]



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

Spatially sparse programming

Ideally...

Save memory and computation on inactive spaces.

Reality...

Not easy to achieve desired performance and productivity

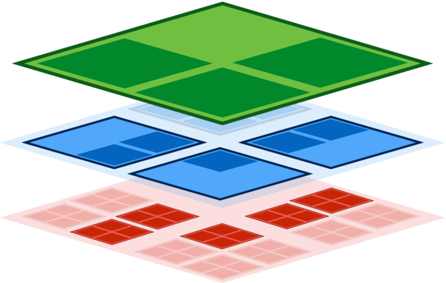
Taichi's solutions

1. Allow programmers to flexibly define sparse data structures using SNodes
2. Let programmers access sparse data structures as if they are dense
3. Automatically manage memory
4. Taichi's compiler automatically optimizes sparse data structure access

People's favorite sparse SNode

The "pointer" SNode

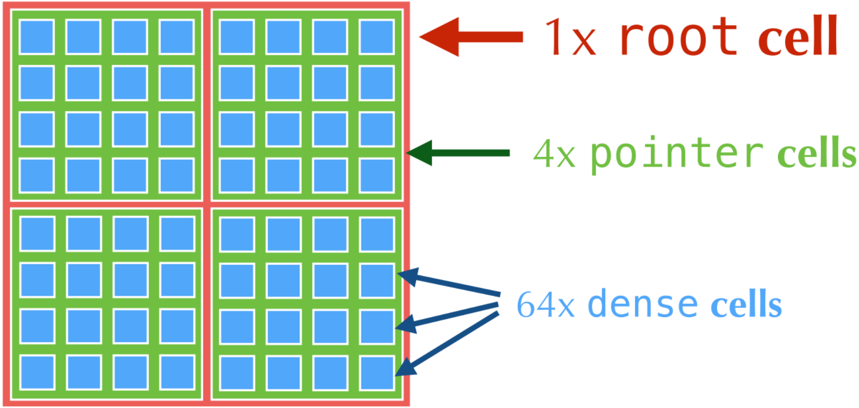
- Essentially an array with each element being a pointer
- Different from the `dense` SNode, since pointers can be `nullptr`
- The most frequently used SNode to achieve sparsity



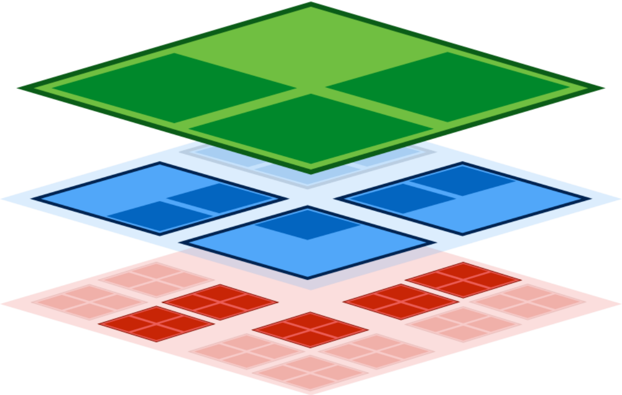
```
ti.root
.pointer(ti.ij, 2)
.pointer(ti.ij, 2)
.dense(ti.ij 2)
```

“Pointer” and “Dense” are best buddies ☺

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



Sparse for-loops: no waste computation on inactive cells



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

Table of Contents

Differentiable Programming

Forward programs evaluate $f(\mathbf{x})$; backward (gradient) programs evaluate $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$.

Taichi supports **reverse-mode automatic differentiation (AutoDiff)** that back-propagates gradients w.r.t. a scalar (loss) function $f(\mathbf{x})$.

Two ways to compute gradients:

1. Use Taichi's tape (`ti.Tape(loss)`) for both forward and gradient evaluation.
2. Explicitly use **gradient kernels** for gradient evaluation with more controls.

Gradient-based optimization

$$\min_{\mathbf{x}} \quad L(\mathbf{x}) = \frac{1}{2}\sum_{i=0}^{n-1}(\mathbf{x}_i - \mathbf{y}_i)^2.$$

1. Allocating fields with gradients:

```
x = ti.field(dtype=ti.f32, shape=n, needs_grad=True)
```

2. Defining loss function kernel(s):

```
@ti.kernel
def reduce():
    for i in range(n):
        L[None] += 0.5 * (x[i] - y[i])**2
```

3. Compute loss `with ti.Tape(loss=L): reduce()`
4. Gradient descent: `for i in x: x[i] -= x.grad[i] * 0.1`

Demo: `ti example autodiff_minimization`

Another demo: `ti example autodiff_regression`

Application 1: Forces from potential energy gradients

From the definition of potential energy:

$$\mathbf{f}_i = -\frac{\partial U(\mathbf{x})}{\partial \mathbf{x}_i}$$

Manually deriving gradients is hard. Let's use AutoDiff:

1. Allocate a 0D field to store the potential energy:
   `potential = ti.field(ti.f32, shape=())`.
2. Define forward kernels that computes potential energy from `x[i]`.
3. In a `ti.Tape(loss=potential)`, call the forward kernels.
4. Force on each particle is `-x.grad[i]`.

Application 2: Differentiating a whole physical process

10 Demos: DiffTaichi $(\mathbf{x}_{t+1}, \mathbf{v}_{t+1}, \dots) = \mathbf{F}(\mathbf{x}_t, \mathbf{v}_t, \dots)$
Pattern:

```
with ti.Tape(loss=loss):
    for i in range(steps - 1):
        simulate(i)
```

Computational history

Always keep the whole computational history of time steps for end-to-end differentiation. I.e., instead of only allocating `ti.Vector.field(3, dtype=ti.f32, shape=(num_particles))` that stores the latest particles, allocate for the whole simulation process `ti.Vector.field(3, dtype=ti.f32, shape=(num_timesteps, num_particles))`. Do not overwrite! (Use **checkpointing** to reduce memory consumption.)

Flattening-based reverse-mode AD[4]


```
int a = 0;
if (b > 0) { a = b;}
  else     { a = 2b;}
a = a + 1;
return a;
```

```
// flatten branching
int a = 0;
a = select(b > 0, b, 2b);
a = a + 1
return a;
```

```
// eliminate mutable var

ssa1 = select(b > 0, b, 2b);
ssa2 = ssa1 + 1
return ssa2;
```

[4] Y. Hu et al. (2020). "DiffTaichi: Differentiable Programming for Physical Simulation". In: *ICLR*.

Flattening-based reverse-mode AD with "good" loops

forward

$$y_i = \sin x_i^2$$

```
for i ∈ range(0, 16, step 1) do
  %1 = load x[i]
  %2 = mul %1, %1
  %3 = sin(%2)
  store y[i] = %3
end for
```

Backward

```
for i in range(0, 16, step 1) do
  // adjoint variables
  %1adj = alloca 0.0
  %2adj = alloca 0.0
  %3adj = alloca 0.0
  // original forward computation
  %1 = load x[i]
  %2 = mul %1, %1
  %3 = sin(%2)
  // reverse accumulation
  %4 = load y_adj[i]
  %3adj += %4
  %5 = cos(%2)
  %2adj += %3adj * %5
  %1adj += 2 * %1 * %2adj
  atomic add x_adj[i], %1adj
end for
```

Note: Taichi uses global tensors as checkpoints, so we need some recomputation. (Megakernel has a cost in AutoDiff but still helps.)

Stack-based reverse-mode AD

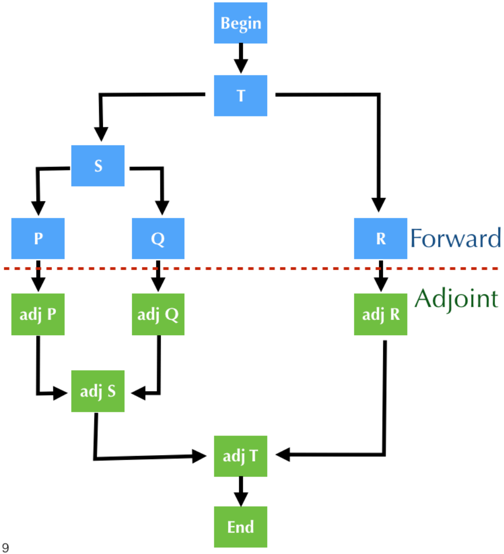
Slightly harder cases where we need stacks to trace local computational history: ▸transforms/auto_diff.cpp

```
@ti.kernel
def fib():
    for i in range(N):
        p = a[i]
        q = b[i]
        for j in range(c[i]):
            new_p = q
            new_q = p + q
            p, q = new_p, new_q
        f[i] = q
```

```
@ti.kernel
def power():
    for i in range(N):
        ret = 1.0
        for j in range(b[i]):
            ret = ret * a[i]
        p[i] = ret
```

Stack-based reverse-mode AD (Control flow)

```
T
if (x) {
    S
    if (y) {
        P
    } else {
        Q
    }
} else {
    R
}
```



9

Two-scale AutoDiff ▸ lang/tape.py

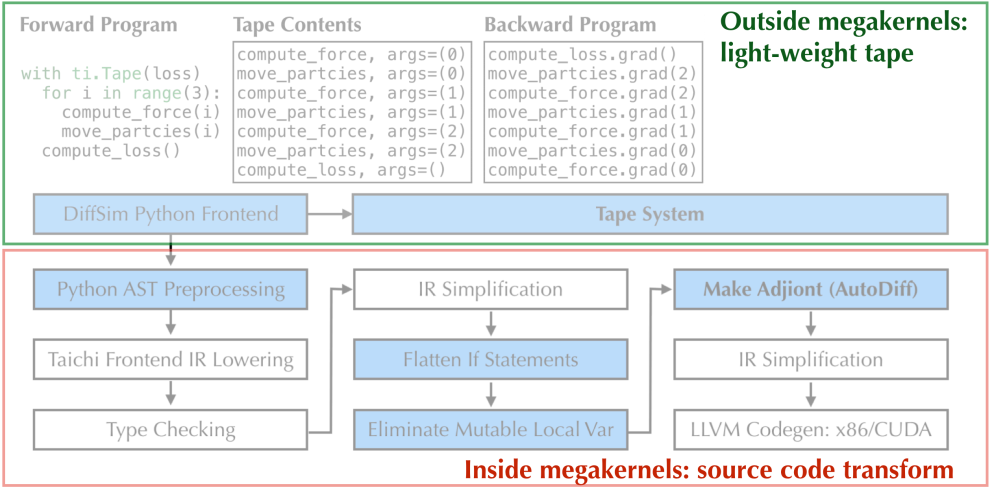


Table of Contents

7 existing backends of Taichi

- Taichi backends ▸backends
  - LLVM ▸codegen/codegen_llvm.cpp
    - CPU: x64, AArch64 (aka ARM64) ▸backends/cpu
    - CUDA (NVPTX) ▸backends/cuda
  - C99 ▸backends/cc
    - JavaScript/WebAssembly: Demos
  - OpenGL ▸backends/opengl
  - Metal ▸backends/metal

Describing backend capabilities

Not all backends are fully featured

Due to backend API limitations it may not be possible to implement certain features on all backends. For example, Apple Metal doesn't support 64-bit data types, so `ti.f64` is not available on Taichi's Metal backend.

Backend extension system

Backend extensions are used to describe the capability of each backend (beyond core functionalities). For example,

1. `Extension::sparse`: sparse computation
2. `Extension::data64`: 64-bit data types
3. `Extension::adstack`: stack-based AutoDiff
4. All backend extensions: ▸`inc/extensions.inc.h`

Which backend supports what? ▸`program/extension.cpp`

The LLVM backends

The CPU and CUDA code generators are based on LLVM (v10.0.0) ORCv2 JIT (CPU) and NVPTX backend (CUDA).

- Extremely portable: the LLVM CPU backend has no dependency
- Used as the fallback solution for all other backends
- Fully featured (i.e., support all extensions)
- Serve as references for other backends
- Has a (ideally) zero-cost runtime system (more on this later) ▸`runtime/llvm/runtime.cpp`

Source-to-source backends

Most backends do not take LLVM IR as input. In those cases we directly emit source code (i.e., source-to-source compilation).

1. C99 ▸`backends/cc`
   - + Emscripten= JavaScript/WebAssembly
2. OpenGL ▸`backends/opengl`
3. Metal ▸`backends/metal`

Legacy CPU and CUDA backends

Taichi used to emit C++ and CUDA source code for CPU and CUDA backends as well. However, this solution is abandoned due to portability issues and long compilation time.

LLVM runtime system ▸runtime/llvm

Not really a runtime system...

Design tradeoffs

- One implementation should ideally serve for all LLVM-related backends
- The runtime code need to be compiled into LLVM IR and then linked against generated code
- We want to use something like C++ templates for each SNode
- ... but we cannot do "link-time template specialization" on LLVM IR $\Longrightarrow$ Inline as much as possible to "simulate" templates.

Runtime system written in C++ ▸`runtime/llvm/runtime.cpp` will be compiled to LLVM IR using `clang` by developers and ship together with the Taichi package.

Memory allocator: design constraints

1. Portability matters
   - Need to support devices with/without (unified) virtual memory
2. Need high-performance, in parallel
   - Make it as lock-free as possible
   - Trade virtual address space and external fragmentation for performance
3. Need to support many backends
   - Make it as simple as possible
   - Use (mostly) the same code path for CPU/GPUs

Memory allocator: the “big” picture

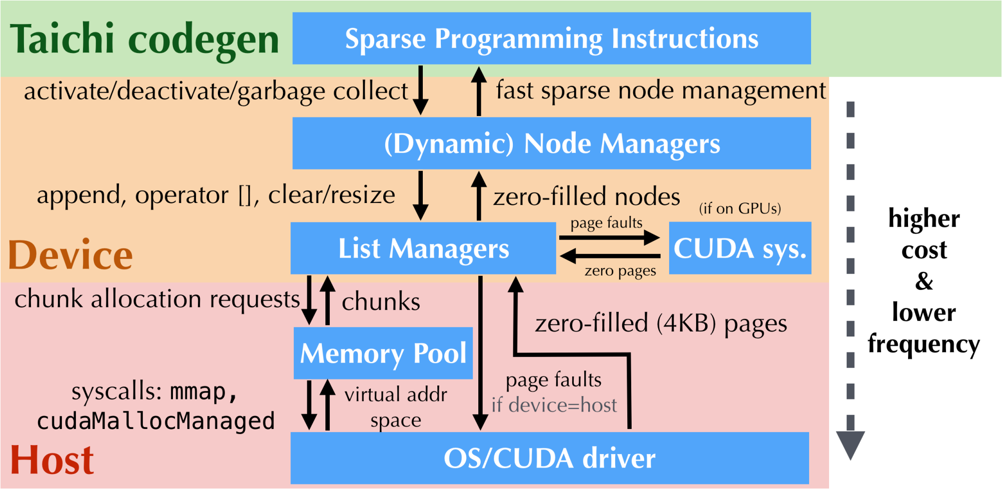


Table of Contents

Summary

- Language design and compiler engineering is a lot of fun!
- Keep it simple and practical.
- **Portability** is not an easy task: took me 4 months to get a prototype (and a SIGGRAPH Asia 2019 paper), but more than 1 year to make it portable.
- $\sim 400,000$ downloads according to PePy, since Jan 2020.

Note

Many features of Taichi are developed by **the Taichi community**.
Clearly, I am not the only developer ☺

More details...

- GitHub
- Docs
- TaichiCon
- SIGGRAPH 2020 Taichi course