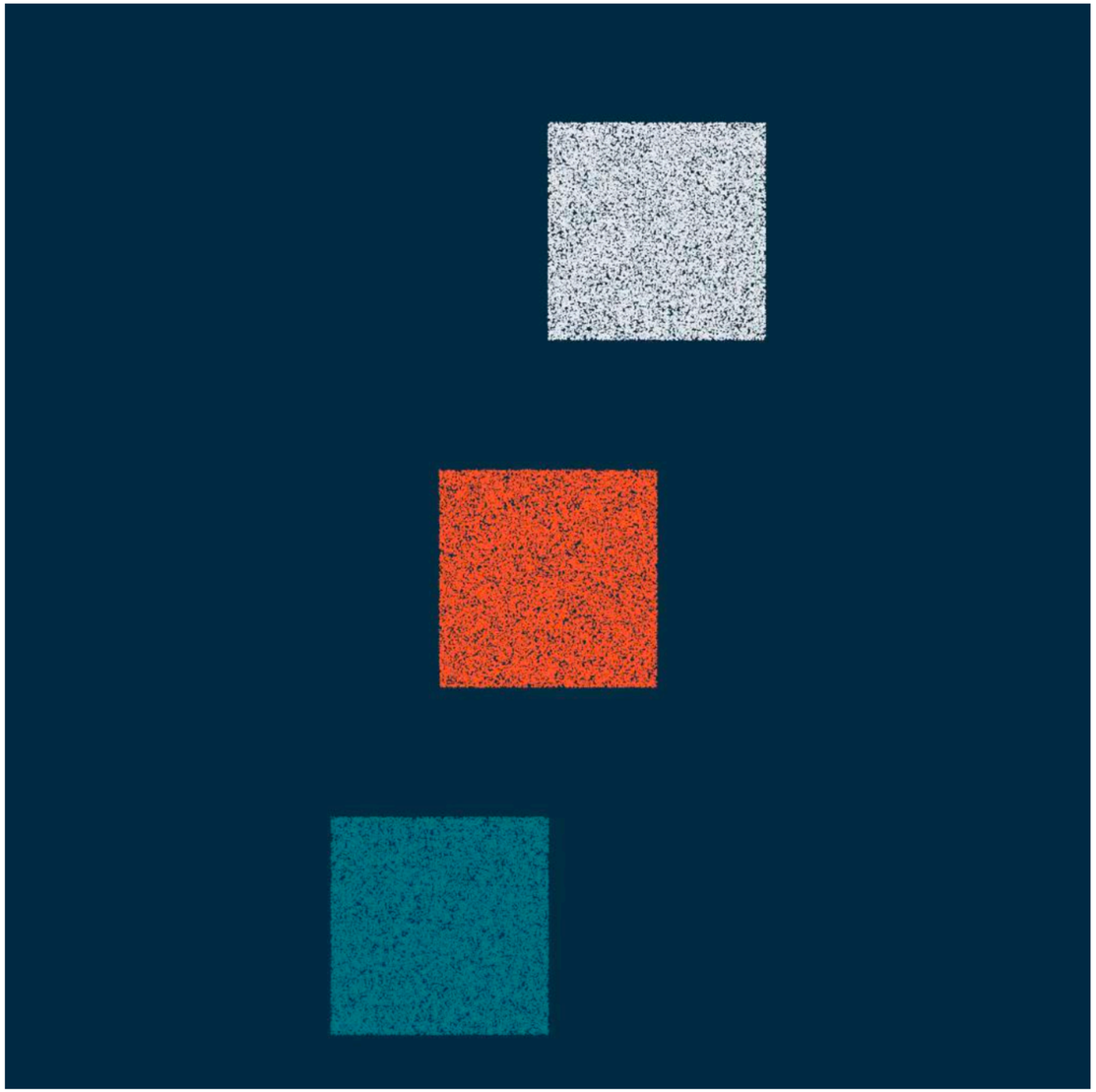
Taichi-lang 上手培训

李泽宇
Email: li_zeyu@pku.edu.cn
Github: Galeselee

Outline

- Why Taichi
- Taichi Tutorial
- Advanced topic

初识Taichi



99行代码的《冰雪奇缘》

胡渊鸣
知乎十年新知答主

已关注

编辑推荐

知乎日报、fifizoo、东半球吃货等 26,141 人赞同了该文章

——闲聊物理引擎，可微编程，SIGGRAPH生产力难题，与Taichi编程语言

2021年4月29日更新：Taichi社区的 VictoriaCity 同学关于相关话题的精美视频介绍：

99 行代码写出来的特效，《冰雪奇缘》都用过？揭秘物理引擎中的算法和生产力工...
www.bilibili.com/video/BV19K4y1P7fb?sh...

目录

1. 序
2. Material Point Method（物质点法）
3. Moving Least Squares Material Point Method（移动最小二乘物质点法）
4. Differentiable MLS-MPM (ChainQueen可微物理引擎)

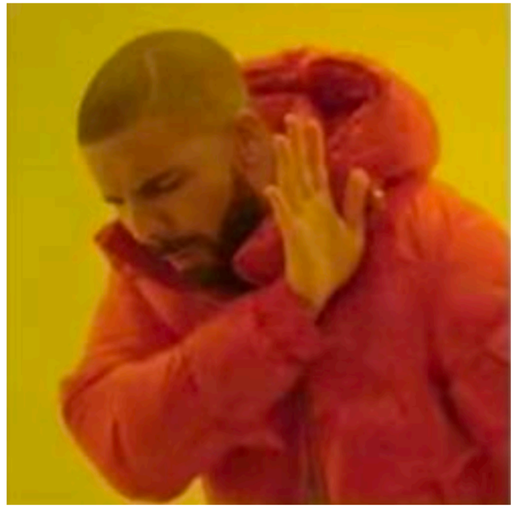
已赞同 2.6 万　992 条评论　分享　喜欢　收藏　申请转载　···

MPM

~~7行代码实现冰雪奇缘~~

~~“7行《冰雪奇缘》”~~

```
#include <your_awesome_mpm_lib>

int main(int argc, char **argv)
{
    AWESOME_LIB::run_mpm(argc, argv);
    return 0;
}
```



MPM

“99行《冰雪奇缘》”

```python
import numpy as np

import taichi as ti

ti.init(arch=ti.gpu)  # Try to run on GPU
quality = 1  # Use a larger value for higher-res simulations
n_particles, n_grid = 9000 * quality**2, 128 * quality
dx, inv_dx = 1 / n_grid, float(n_grid)
dt = 1e-4 / quality
p_vol, p_rho = (dx * 0.5)**2, 1
p_mass = p_vol * p_rho
E, nu = 0.1e4, 0.2  # Young's modulus and Poisson's ratio
mu_0, lambda_0 = E / (2 * (1 + nu)), E * nu / (
    (1 + nu) * (1 - 2 * nu))  # Lame parameters
x = ti.Vector.field(2, dtype=float, shape=n_particles)  # position
v = ti.Vector.field(2, dtype=float, shape=n_particles)  # velocity
C = ti.Matrix.field(2, 2, dtype=float,
                    shape=n_particles)  # affine velocity field
F = ti.Matrix.field(2, 2, dtype=float,
                    shape=n_particles)  # deformation gradient
material = ti.field(dtype=int, shape=n_particles)  # material id
Jp = ti.field(dtype=float, shape=n_particles)  # plastic deformation
grid_v = ti.Vector.field(2, dtype=float,
                         shape=(n_grid, n_grid))  # grid node momentum/velocity
grid_m = ti.field(dtype=float, shape=(n_grid, n_grid))  # grid node mass
```

Data

Computation

```python
@ti.kernel
def substep():
    for i, j in grid_m:
        grid_v[i, j] = [0, 0]
        grid_m[i, j] = 0
    for p in x:  # Particle state update and scatter to grid (P2G)
        base = (x[p] * inv_dx - 0.5).cast(int)
        fx = x[p] * inv_dx - base.cast(float)
        # Quadratic kernels  [http://mpm.graphics   Eqn. 123, with x=fx, fx-1,fx-2]
        w = [0.5 * (1.5 - fx)**2, 0.75 - (fx - 1)**2, 0.5 * (fx - 0.5)**2]
        F[p] = (ti.Matrix.identity(float, 2) +
                dt * C[p]) @ F[p]  # deformation gradient update
        h = ti.exp(
            10 *
            (1.0 -
             Jp[p]))  # Hardening coefficient: snow gets harder when compressed
        if material[p] == 1:  # jelly, make it softer
            h = 0.3
        mu, la = mu_0 * h, lambda_0 * h
        if material[p] == 0:  # liquid
            mu = 0.0
        U, sig, V = ti.svd(F[p])
        J = 1.0
        for d in ti.static(range(2)):
            new_sig = sig[d, d]
            if material[p] == 2:  # Snow
                new_sig = min(max(sig[d, d], 1 - 2.5e-2),
                              1 + 4.5e-3)  # Plasticity
            Jp[p] *= sig[d, d] / new_sig
            sig[d, d] = new_sig
            J *= new_sig
        if material[
                p] == 0:  # Reset deformation gradient to avoid numerical instability
            F[p] = ti.Matrix.identity(float, 2) * ti.sqrt(J)
        elif material[p] == 2:
            F[p] = U @ sig @ V.transpose(
            )  # Reconstruct elastic deformation gradient after plasticity
        stress = 2 * mu * (F[p] - U @ V.transpose()) @ F[p].transpose(
        ) + ti.Matrix.identity(float, 2) * la * J * (J - 1)
        stress = (-dt * p_vol * 4 * inv_dx * inv_dx) * stress
        affine = stress + p_mass * C[p]
        for i, j in ti.static(ti.ndrange(
                3, 3)):  # Loop over 3x3 grid node neighborhood
            offset = ti.Vector([i, j])
            dpos = (offset.cast(float) - fx) * dx
            weight = w[i][0] * w[j][1]
            grid_v[base + offset] += weight * (p_mass * v[p] + affine @ dpos)
            grid_m[base + offset] += weight * p_mass
    for i, j in grid_m:
        if grid_m[i, j] > 0:  # No need for epsilon here
            grid_v[i,
                   j] = (1 / grid_m[i, j]) * grid_v[i,
                                                    j]  # Momentum to velocity
            grid_v[i, j][1] -= dt * 50  # gravity
            if i < 3 and grid_v[i, j][0] < 0:
                grid_v[i, j][0] = 0  # Boundary conditions
            if i > n_grid - 3 and grid_v[i, j][0] > 0: grid_v[i, j][0] = 0
            if j < 3 and grid_v[i, j][1] < 0: grid_v[i, j][1] = 0
            if j > n_grid - 3 and grid_v[i, j][1] > 0: grid_v[i, j][1] = 0
    for p in x:  # grid to particle (G2P)
        base = (x[p] * inv_dx - 0.5).cast(int)
        fx = x[p] * inv_dx - base.cast(float)
        w = [0.5 * (1.5 - fx)**2, 0.75 - (fx - 1.0)**2, 0.5 * (fx - 0.5)**2]
        new_v = ti.Vector.zero(float, 2)
        new_C = ti.Matrix.zero(float, 2, 2)
        for i, j in ti.static(ti.ndrange(
                3, 3)):  # loop over 3x3 grid node neighborhood
            dpos = ti.Vector([i, j]).cast(float) - fx
            g_v = grid_v[base + ti.Vector([i, j])]
            weight = w[i][0] * w[j][1]
            new_v += weight * g_v
            new_C += 4 * inv_dx * weight * g_v.outer_product(dpos)
        v[p], C[p] = new_v, new_C
        x[p] += dt * v[p]  # advection
```

Visualization

```python
group_size = n_particles // 3

@ti.kernel
def initialize():
    for i in range(n_particles):
        x[i] = [
            ti.random() * 0.2 + 0.3 + 0.10 * (i // group_size),
            ti.random() * 0.2 + 0.05 + 0.32 * (i // group_size)
        ]
        material[i] = i // group_size  # 0: fluid 1: jelly 2: snow
        v[i] = ti.Matrix([0, 0])
        F[i] = ti.Matrix([[1, 0], [0, 1]])
        Jp[i] = 1

initialize()
gui = ti.GUI("Taichi MLS-MPM-99", res=512, background_color=0x112F41)
while not gui.get_event(ti.GUI.ESCAPE, ti.GUI.EXIT):
    for s in range(int(2e-3 // dt)):
        substep()
    gui.circles(x.to_numpy(),
                radius=1.5,
                palette=[0x068587, 0xED553B, 0xEEEEF0],
                palette_indices=material)
    gui.show()  # Change to gui.show(f'{frame:06d}.png') to write images to disk
```

Why Taichi

What’s Taichi

- Taichi - A EDSL(Embedded Domain Specific Language) for Computer Graphics
  - Host language: Python
    - Python similar syntax
    - Taichi compiler compiles taichi-scope code.
  - Independent IR(Intermediate representation), compilation optimization and codegen
  - Automatic parallelism on multi-cores(both cpu and gpu)
  - Good interaction with third-party libraries

Why Taichi

- Productivity
  - Friendly learning curve
  - Shorter code, higher perf.
- Portability
  - Multi-backend
- Performance
  - Auto parallelism and data locality

Why Taichi

Productivity reduction_sum

main / taichi_benchmark / suites / reduce_sum / src / cuda /

feisuzhu feat: More aggresive refactoring

..

benchmark.py	feat: More aggresive refactoring
reduce_sum.cuh	feat: More aggresive refactoring
reduce_sum_cub.cu	feat: More aggresive refactoring
reduce_sum_cuda.cu	feat: More aggresive refactoring
reduce_sum_thrust.cu	feat: More aggresive refactoring

```cpp
__device__ float WarpReduceSum(float val) {
    int lane = threadIdx.x & 31;
    float tmp = __shfl_up_sync(0xffffffff, val, 1);
    if (lane >= 1) val += tmp;
    tmp = __shfl_up_sync(0xffffffff, val, 2);
    if (lane >= 2) val += tmp;
    tmp = __shfl_up_sync(0xffffffff, val, 4);
    if (lane >= 4) val += tmp;
    tmp = __shfl_up_sync(0xffffffff, val, 8);
    if (lane >= 8) val += tmp;
    tmp = __shfl_up_sync(0xffffffff, val, 16);
    if (lane >= 16) val += tmp;
    __syncthreads();
    return val;
}

__device__ float BlockReduceSum(float val) {
    int warp_id = threadIdx.x >> 5;
    int lane = threadIdx.x & 31;
    __shared__ float warp_sum[32];

    val = WarpReduceSum(val);
    if(lane == 31) warp_sum[warp_id] = val;
    __syncthreads();
    if(warp_id == 0) {
        if (lane >= 1) warp_sum[lane] += warp_sum[lane-1];
        __syncwarp();
        if (lane >= 2) warp_sum[lane] += warp_sum[lane-2];
        __syncwarp();
        if (lane >= 4) warp_sum[lane] += warp_sum[lane-4];
        __syncwarp();
        if (lane >= 8) warp_sum[lane] += warp_sum[lane-8];
        __syncwarp();
        if (lane >= 16) warp_sum[lane] += warp_sum[lane-16];
    }
    __syncthreads();
    if(warp_id > 0) val += warp_sum[warp_id-1];
    return val;
}

__global__
void ReduceSumKernel(float *in, float *out,
    int num_items, int num_part) {
    float val = 0;
    int idx = 0;
    for(int ii = blockIdx.x; ii < num_part; ii += gridDim.x) {
        idx = threadIdx.x +  blockDim.x * ii;
        val += idx < num_items ? in[idx] : 0;
    }
    val = BlockReduceSum(val);
    if(threadIdx.x == blockDim.x - 1&& (idx < num_items) || (idx == num_items - 1)) {
        out[blockIdx.x] = val;
    }
}
```

repo: https://github.com/taichi-dev/taichi_benchmark

Why Taichi

Productivity reduction_sum

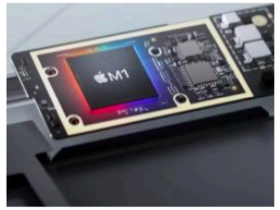
```python
import taichi as ti

ti.init(ti.cuda,
    kernel_profiler=True,
    )

def reduce_sum(n_items, repeats=5):
    f = ti.field(ti.f32, (n_items,))
    @ti.kernel
    def init_kernel():
        for i in f:
            f[i] = 1.0
    @ti.kernel
    def reduce_sum_kernel():
        sum = 0.0
        for i in ti.grouped(f):
            ti.atomic_add(sum, f[i])
    kernel_time = 0.0
    n_iter = 0
    while n_iter < repeats:
        init_kernel()
        ti.profiler.clear_kernel_profiler_info()
        reduce_sum_kernel()
        ti.sync()
        kernel_time += ti.profiler.get_kernel_profiler_total_time() * 1000
        n_iter += 1
    return kernel_time/repeats
```

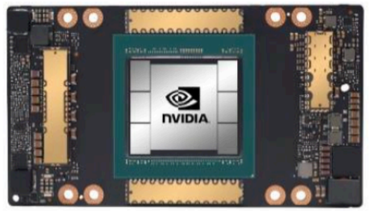
repo: https://github.com/taichi-dev/taichi_benchmark

Why Taichi

Portability

```python
import taichi as ti

ti.init(arch = ti.cpu/ti.gpu/ti.metal/ti.vulkan/ti.cuda....)
```





- ti.gpu: cuda / metal / vulkan / amdgpu(WIP)
  - If gpu isn’t avaliable, it will back to ti.cpu

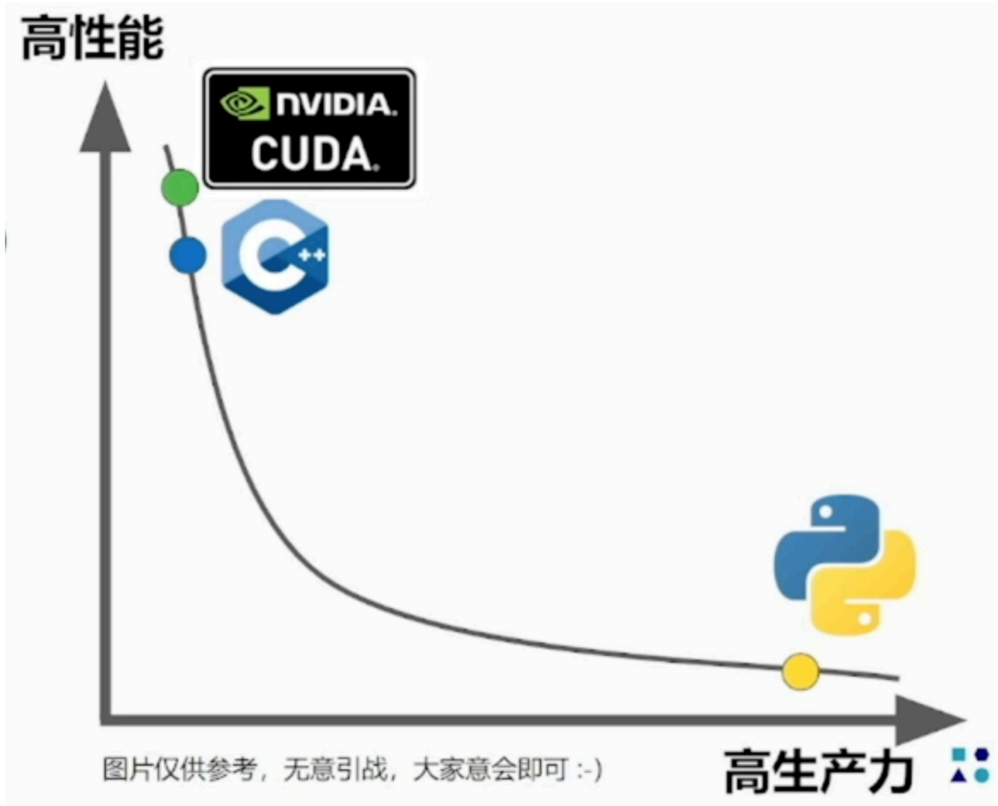
Why Taichi

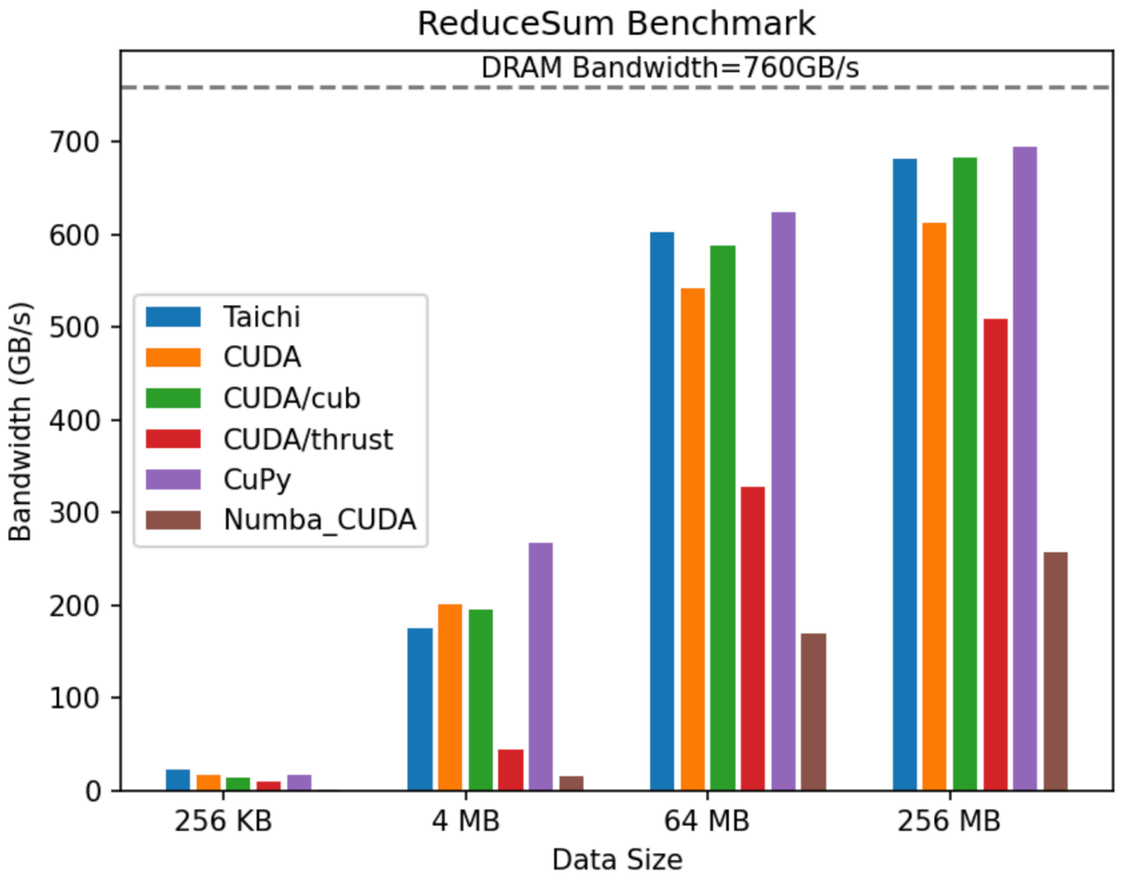
Portability

	CPU	CUDA	OpenGL	Apple Metal	Vulkan	AMDGPU
Windows	YES	YES	YES	NO	YES	NO
MacOS	YES	NO	NO	YES	YES	NO
Linux	YES	YES	YES	NO	YES	WIP

Why Taichi

Performance





测试环境：Nvidia RTX 3080

Why Taichi

Performance

- 编译优化（Just In Time）
- 提供了和CUDA等价的编译产物（LLVM、PTX）

```python
import taichi as ti
ti.init(arch=ti.cuda, print_kernel_nvptx=True)

f = ti.field(dtype=ti.f32, shape=(1280,))

@ti.kernel
def init_kernel():
    for i in f:
        f[i] = 1.0

@ti.kernel
def reduce_sum():
    sum = 0.0
    for i in ti.grouped(f):
        ti.atomic_add(sum, f[i])

reduce_sum()

```

```
//
// Generated by LLVM NVPTX Back-End
//

.version 6.3
.target sm_75
.address_size 64

    // .globl   reduce_sum_c78_0_kernel_0_serial // --
                                                // @reduce_sum
.visible .entry reduce_sum_c78_0_kernel_0_serial(
    .param .align 8 .b8 reduce_sum_c78_0_kernel_0_seri
)
.maxntid 1, 1, 1
.minnctapersm 2
{
    .reg .b32   %r<2>;
    .reg .b64   %rd<4>;

// %bb.0:                                   // %entry
    ld.param.u64    %rd1, [reduce_sum_c78_0_kernel_0_s
    cvta.to.global.u64  %rd2, %rd1;
    ld.global.u64   %rd3, [%rd2+32864];
    mov.u32     %r1, 0;
    st.u32  [%rd3], %r1;
    ret;
                                            // -- End func
}
    // .globl   reduce_sum_c78_0_kernel_1_range_for //
.visible .entry reduce_sum_c78_0_kernel_1_range_for(
    .param .align 8 .b8 reduce_sum_c78_0_kernel_1_rang
)
.maxntid 128, 1, 1
.minnctapersm 2                             // @reduce_sum
{
    .reg .pred  %p<5>;
    .reg .b32   %r<12>;
    .reg .f32   %f<22>;
    .reg .b64   %rd<15>;

// %bb.0:                                   // %entry
    mov.u32     %r1, %tid.x;
    mov.u32     %r2, %ntid.x;
```

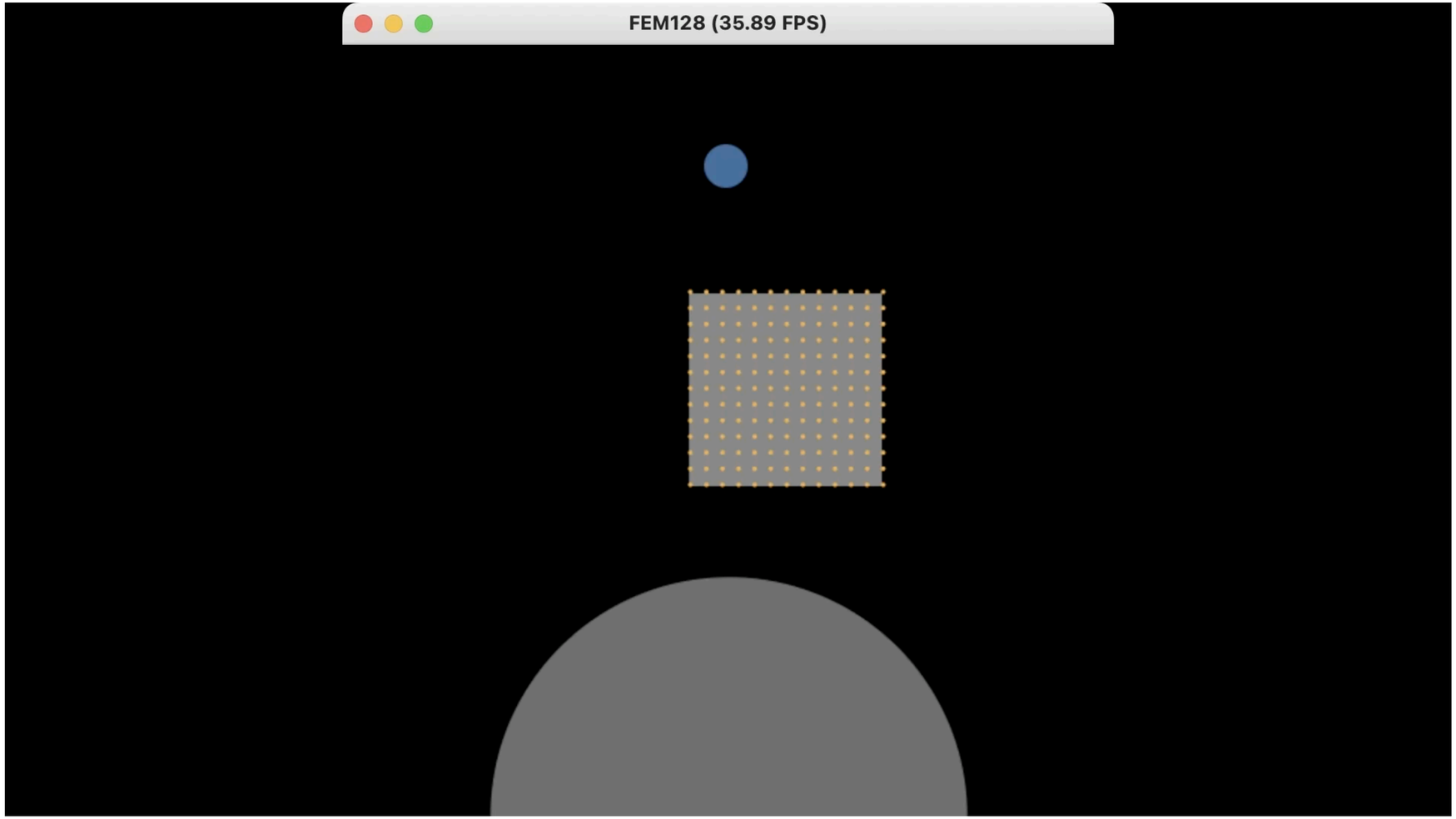
在ti.init中设置`print_kernel_llvm_ir=True`，则会输出相应的LLVM IR

Why Taichi

- Productivity
  - Friendly learning curve
  - Shorter code, higher perf.
- Portability
  - Multi-backend
- Performance
  - Optimized for bandwidth, auto parallelism and data locality
- Open Source

Why Taichi

What you can do using Taichi



物理仿真模拟
FEM
(Finite Element Methods)

Why Taichi

What you can do using Taichi

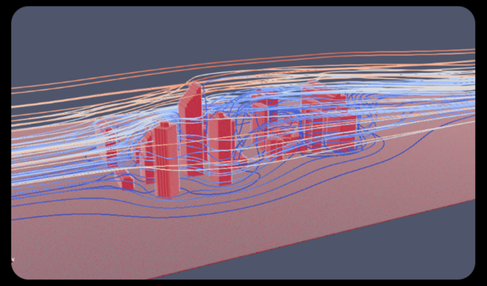
Python 加速

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

@ti.kernel
def monte_carlo_pi(n: int) -> float:
    total = 0
    for i in range(n):
        x = ti.random()
        y = ti.random()
        if (x ** 2 + y ** 2) < 1.0:
            total += 1
    return 4.0 * total / n
```

用 Taichi 将 Python 代码加速 100+ 倍

点击体验

数值仿真



用 Taichi 实现的 LBM 流体仿真，模拟风场穿越城市

查看详情

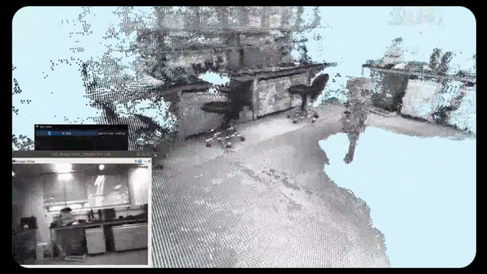
3D 渲染

用 Taichi 实现的简单精致的体素渲染器

* 图为 2022 年 Taichi 体素大赛冠军作品，仅需 96 行代码，作者 @SuperSodaSea

查看详情

机器人视觉



用 Taichi 实现的激光雷达 SLAM 算法

查看详情

Example1

素数

```
import taichi as ti
ti.init(arch=ti.x64, kernel_profiler=True)
@ti.func
def is_prime(n: int):
    result = True
    for k in range(2, int(n ** 0.5) + 1):
        if n % k == 0:
            result = False
            break
    return result
@ti.kernel
def count_primes(n: int) -> int:
    count = 0
    for k in range(2, n):
        if is_prime(k):
            count += 1
    return count
count_primes(10000000)
#ti.profiler.print_kernel_profiler_info()
```

https://github.com/Galeselee/taichi_tutorial

Taichi Tutorial

Install Taichi

```
(taichi) zeyuli@lizeyudeMacBook-Pro build % pip install taichi -i https://pypi.douban.com/simple
Looking in indexes: https://pypi.douban.com/simple
Collecting taichi
  Downloading https://pypi.doubanio.com/packages/f6/ff/094de1d31ec97bc7272276f095132ea3adac724427016095bc2510e50432/taichi-1.3.0-cp310-cp310-macosx_10_15_
x86_64.whl (37.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 37.0/37.0 MB 1.5 MB/s eta 0:00:00
Requirement already satisfied: numpy in /Users/zeyuli/opt/anaconda3/envs/taichi/lib/python3.10/site-packages (from taichi) (1.24.1)
Requirement already satisfied: rich in /Users/zeyuli/opt/anaconda3/envs/taichi/lib/python3.10/site-packages (from taichi) (13.0.0)
Requirement already satisfied: sourceinspect>=0.0.4 in /Users/zeyuli/opt/anaconda3/envs/taichi/lib/python3.10/site-packages (from taichi) (0.0.4)
Requirement already satisfied: colorama in /Users/zeyuli/opt/anaconda3/envs/taichi/lib/python3.10/site-packages (from taichi) (0.4.6)
Requirement already satisfied: dill in /Users/zeyuli/opt/anaconda3/envs/taichi/lib/python3.10/site-packages (from sourceinspect>=0.0.4->taichi) (0.3.6)
Requirement already satisfied: commonmark<0.10.0,>=0.9.0 in /Users/zeyuli/opt/anaconda3/envs/taichi/lib/python3.10/site-packages (from rich->taichi) (0.9.
1)
Requirement already satisfied: pygments<3.0.0,>=2.6.0 in /Users/zeyuli/opt/anaconda3/envs/taichi/lib/python3.10/site-packages (from rich->taichi) (2.11.2)
Installing collected packages: taichi
Successfully installed taichi-1.3.0
```

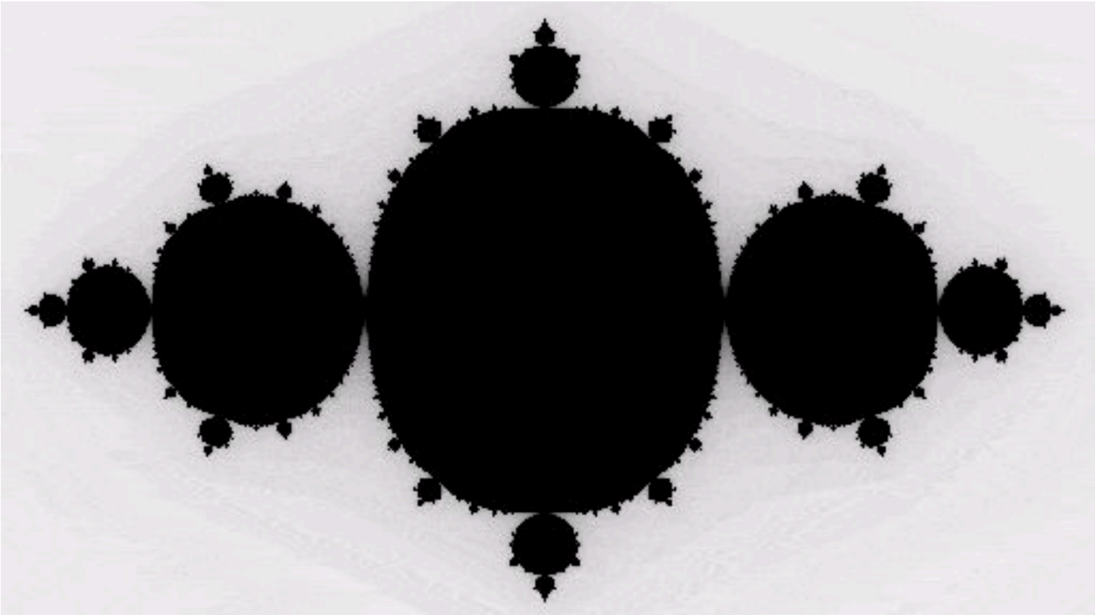
ti.kernel and ti.func

import taichi as ti

- **The entry point for ALL Taichi project**

```
• import taichi as ti
• ti.init(arch=ti.gpu)
```

```python
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

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()
```



The Python frontend of Taichi

```python
import taichi as ti

ti.init(arch=ti.cpu)

d = 1

def foo():
    d_python = d
    print("d_python =", d_python)

@ti.kernel
def bar():
    d_taichi = d
    print("d_taichi =", d_taichi)

d = d + 1 # d = 2
foo()     # d_python = 2
bar()     # d_taichi = 2
d = d + 1 # d = 3
foo()     # d_python = 3
bar()     # d_taichi = 2
```

Python-scope

Taichi-scope

Taichi-scope v.s. Python-scope

- Python-scope
  - Everything in a normal Python script is in the **Python-scope**

```
import taichi as ti

ti.init(arch=ti.cpu)

def foo():
    print("This is a normal python function")

foo()
```

- Taichi-scope
  - Everything decorated by @ti.kernel or @ti.func is in the **Taichi-scope**

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("This is now a Taichi kernel")

foo()
```

Kernels

- A Python function decorated by @ti.kernel is a Taichi kernel
  - Taichi kernels can only be called **from the Python scope**

```python
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("foo")

@ti.kernel
def bar():
    print("bar")

foo()
bar()
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

def foo():
    print("foo")
    bar()

@ti.kernel
def bar():
    print("bar")

foo()
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("foo")
    bar()

@ti.kernel
def bar():
    print("bar")

foo()
```

For-loops in a @ti.kernel

- For loops at the **outermost scope** in a Taichi kernel is **automatically parallelized**

```
@ti.kernel
def fill():
    for i in range(10): # Parallelized
        x[i] += i

        s = 0
        for j in range(5): # Serialized in each parallel thread
            s += j

        y[i] = s

    for k in rang(20): # Parallelized
        z[k] = k
```

Example

vecadd - CUDA backend

```python
@ti.kernel
def vecadd(a:ti.types.ndarray(dtype=ti.f32, ndim=1), \
           b:ti.types.ndarray(dtype=ti.f32, ndim=1), \
           c:ti.types.ndarray(dtype=ti.f32, ndim=1)):
    for ii in range(10):
        c[ii] = a[ii] + b[ii]
```

```
void gpu_parallel_range_for(...) {
  int idx = thread_idx() + block_dim() * block_idx() + begin;
  ...
  while (idx < end) {
    func(context, tls_ptr, idx);
    idx += block_dim() * grid_dim();
  }
}
```

JIT

```llvm
define void @vecadd_c76_0_kernel_0_range_forTfddfcd41518fa3ad5dcb3651c0e904d8912837e84c5553958d084f0584f338e5_6696(%struct.RuntimeContext.8* byval(%struct.RuntimeContext.8) %context) {
entry:
  br label %body

final:                                            ; preds = %body
  ret void

body:                                             ; preds = %entry
  call void @gpu_parallel_range_for(%struct.RuntimeContext.8* %context, i32 0, i32 10, void (%struct.RuntimeContext.8*, i8*)* null, void (%struct.RuntimeContext.8*, i8*, i32)* @function_body, void (%struct.RuntimeContext.8*, i8*)* null, i64 1)
  br label %final
}
```

LLVM IR

For-loops in a @ti.kernel

- Outermost scope ?

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo(k: ti.i32):
    for i in range(10): # Parallelized :-)
        if k > 42:
            ...

@ti.kernel
def bar(k: ti.i32):
    if k > 42:
        for i in range(10): # Serial :-(
            ...
```

For-loops in a @ti.kernel

- Design your for loops for best performance

```
def my_for_loop():
    for i in range(10): # I don't want to parallelize this for
        for j in range(100): # I want to parallelize this for
            ...

my_for_loop()
```

```
def my_for_loop():
    for i in range(10):
        my_taichi_for()

@ti.kernel
def my_taichi_for():
    for j in range(100):
        ...

my_for_loop()
```

For-loops in a @ti.kernel

- break is NOT supported in the parallel for-loops

```
@ti.kernel
def foo():
  for i in range(10):
      ...
      break # Error!

@ti.kernel
def foo():
  for i in range(10):
      for j in range(10):
          ...
          break # OK!
```

For-loops in a @ti.kernel

- Race condition
  - Taichi uses += as an atomic add
  - The compiler optimizes for unnecessary atomic operations

```
@ti.kernel
def sum():
    for i in range(10):
        # 1. OK
        total[None] += x[i]

        # 2. OK
        ti.atomic_add(total[None], x[i])

        # 3. data race
        total[None] = total[None] + x[i]
```

For-loops in a @ti.kernel

- Types of for-loops in Taichi
  - range-for: loops over a **range**, identical to Python range-for
  - struct-for: loops over a **ti.field**, only lives at the outermost scope

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.field(dtype=ti.i32, shape=N)

@ti.kernel
def foo():
  for i in range(N):
      x[i] = i

foo()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.Vector.field(2,dtype=ti.i32, shape=(N,N))

@ti.kernel
def foo():
  for i,j in x:
      x[i,j] = ti.Vector([i, j])

foo()
```

For-loops in a @ti.kernel

- Types of for-loops in Taichi
  - range-for: loops over a **range**, identical to Python range-for
  - struct-for: loops over a **ti.field**, only lives at the outermost scope

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.field(dtype=ti.i32, shape=N)

@ti.kernel
def foo():
  for i in range(N):
      x[i] = i

foo()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.Vector.field(2,dtype=ti.i32, shape=(N,N))

@ti.kernel
def foo():
  for i,j in x:
      x[i,j] = ti.Vector([i, j])

foo()
```

Functions

- A Python function decorated by @ti.func is a Taichi function
  - Taichi functions can only be called **from the Taichi scope**

```python
import taichi as ti

ti.init(arch=ti.cpu)

def foo():
    print("foo")
    bar()

@ti.func
def bar():
    print("bar")

foo()
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("foo")
    bar()

@ti.func
def bar():
    print("bar")

foo()
```

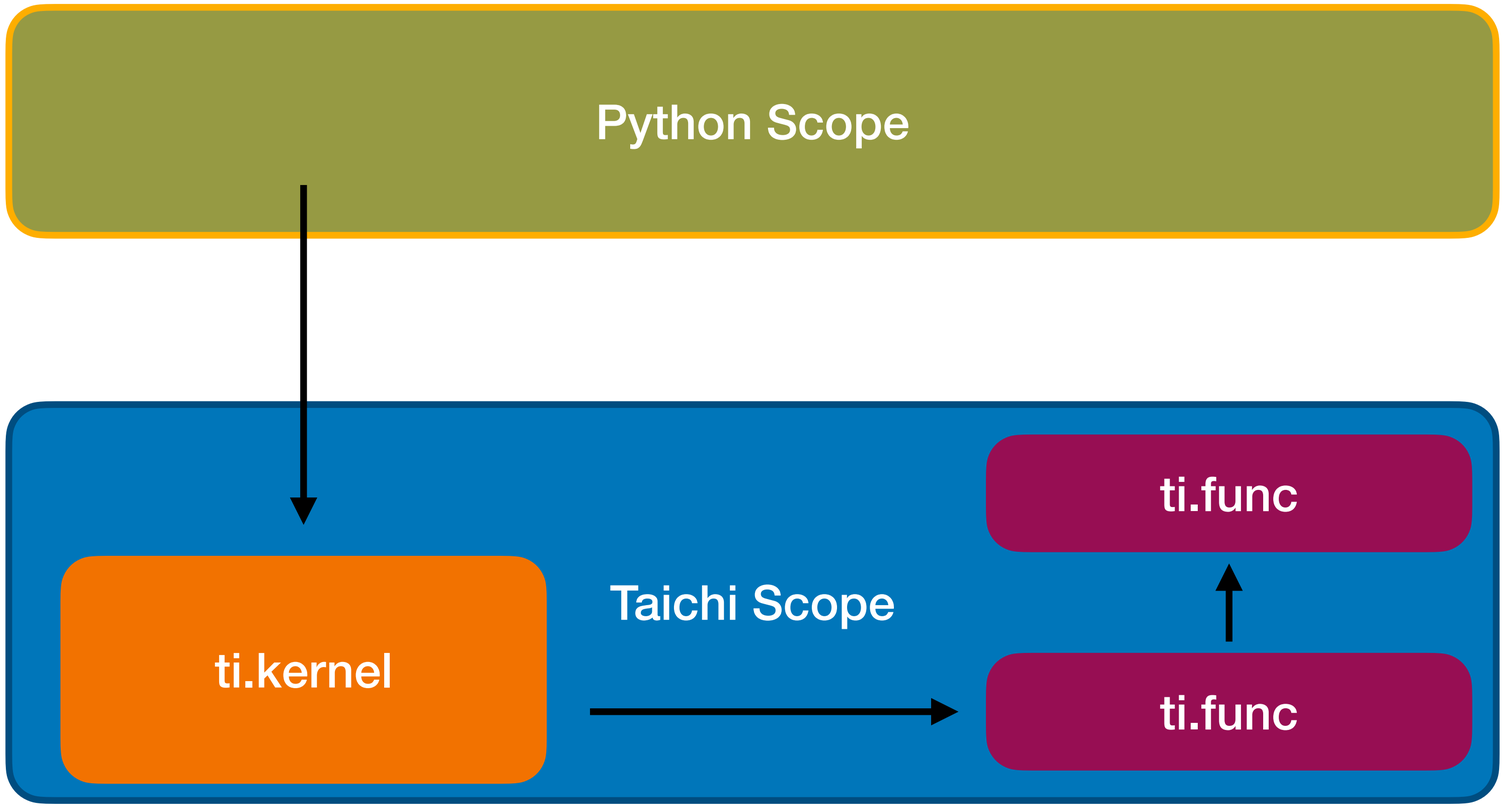
Anything in @ti.kernel and @ti.func are in the **Taichi scope**

- The @ti.kernels and @ti.func will be compiled JIT.
- The Taichi scope variables can not see the Python scope variables at run time.

```python
import taichi as ti

ti.init(arch=ti.cpu)

a = 42

@ti.kernel
def print_a():
    print('a=', a)

print_a()  # "a= 42"
a = 53
print('a=', a) # "a= 53"
print_a()  # "a= 42"
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

a = ti.field(ti.i32, shape=())

@ti.kernel
def print_a():
    print('a=', a[None])

a[None] = 42
print_a()  # "a= 42"
a[None] = 53
print_a()  # "a= 53"
```

Python ti.kernel and ti.func



A ⟶ B

表示A可以调用B

Taichi Tutorial

arguments

- Taichi是一个强类型的语言，类似于C/C++

- ti.kernel 传入参数时必须指明参数类型

```
@ti.kernel
def foo(x: ti.i32):
    x = x + 1
    print("x in foo:", x)
```

- ti.func 输入参数时不需要指明参数类型

taichi-lang-docs: https://docs.taichi-lang.org/

Math Ops

Taichi supports most math operations in Python

```
ti.sin(x)
ti.cos(x)
ti.asin(x)
ti.acos(x)
ti.atan2(y, x)
ti.sqrt(x)
ti.cast(x, data_type)
```

```
ti.floor(x)
ti.ceil(x)
ti.inv(x)
ti.tan(x)
ti.tanh(x)
ti.exp(x)
ti.log(x)
```

```
ti.random(data_type)
abs(x)
int(x)
float(x)
max(x, y, ...)
min(x, y, ...)
x ** y
```

```
A.transpose()
A.inverse()
A.trace()
A.determinant()
v.normalized()

A + B, A * B, A @ B, ...
```

```
R, S = ti.polar_decompose(A, ti.f32)
U, sigma , V = ti.svd(A, ti.f32)
lam, V = ti.eig(A, ti.f32)

u.dot(v)            # scalar
u.cross(v)          # vector
u.outer_product(v)  # matrix
```

Data type

Taichi Tutorial

Primitive types

- signed integers: ti.i8, ti.i16, ti.i32, ti.i64
- unsigned integers: ti.u8, ti.u16, ti.u32, ti.u64
- floating points: ti.f32, ti.f64

Default type

- Taichi picks the more precise type to store the algebraic results
  - i32+f32=f32
  - i32+i64=i64

Type	i8	i16	i32	i64	u8	u16	u32	u64	f32	f64
CPU/CUDA	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
Metal	YES	YES	YES	NO	YES	YES	YES	NO	YES	NO
OpenGL	NO	NO	YES	EXT	NO	NO	NO	NO	YES	YES

Type Casts

- variable = ti.cast(variable, type)

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = 1.7
    b = ti.cast(a, ti.i32)
    c = ti.cast(b, ti.f32)
    print("b =", b) # b = 1
    print("c =", c) # c = 1.0

foo()
```

Compound types

- Using ti.types to **create** compound types including:
  - vector / matrix / struct

```
import taichi as ti

ti.init(arch=ti.cpu)

vec3f = ti.types.vector(3, ti.f32)
mat2f = ti.types.matrix(2, 2, ti.f32)
ray = ti.types.struct(ro=vec3f, rd=vec3f, l=ti.f32)

@ti.kernel
def foo():
    a = vec3f(0.0)
    print(a)          # [0.0, 0.0, 0.0]
    d = vec3f(0.0, 1.0, 0.0)
    print(d)          # [0.0, 1.0, 0.0]
    B = mat2f([[1.5, 1.4], [1.3, 1.2]])
    print("B =", B)        # B = [[1.5, 1.4], [1.3, 1.2]]
    r = ray(ro=a, rd=d, l=1)
    print("r.ro =", r.ro)  # r.ro = [0.0, 0.0, 0.0]
    print("r.rd =", r.rd)  # r.rd = [0.0, 1.0, 0.0]

foo()
```

ti.field

- “a global N-d array of elements”

```
heat_field = ti.field(dtype=ti.f32, shape=(256, 256))
```

- ti.init(device_memory_GB=4GB)
  - Taichi will prealloca CUDA memory

ti.field

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct

ti.field

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct

ti.field

- "a global N-d array of elements"
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct
  - access elements in a field using [i,j,k,…] indexing
    - Special case, access a zero-d field using [None]

```
zero_d_scalar = ti.field(ti.f32, shape=())
zero_d_scalar[None] = 1.5

zero_d_vec = ti.Vector.field(2, ti.f32, shape=())
zero_d_vec[None] = ti.Vector([2.5, 2.6])
```

Note: Taichi dosen't support slice

ti.field examples

- “3D gravitational field in a 256x256x128 room”

```
gravitational_field = ti.Vector.field(n = 3,dtype=ti.f32,shape=(256,256,128))
```

- “2D strain-tensor field in a 64x64 grid”

```
strain_tensor_field = ti.Matrix.field(n = 2,m = 2, dtype=ti.f32, shape=(64,64))
```

- “a global scalar that I want to access in a Taichi kernel”

```
global_scalar = ti.field(dtype=ti.f32, shape=())
```

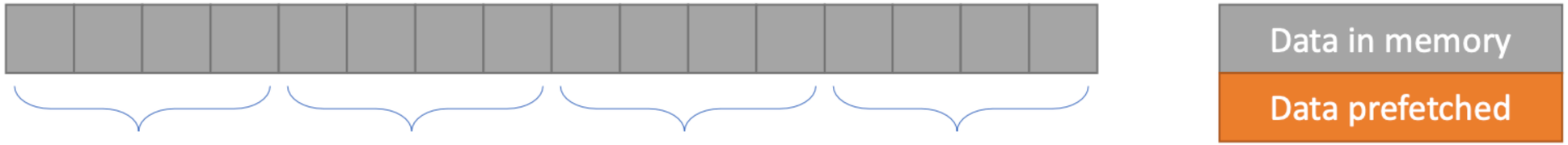
Packed Mode

- Initialized in *ti.init()*
- Decides whether to pad the data to the power of two
  - Default choice: *packed=False*, will do the padding
  - We assume *packed=True* in this class for simplicity

```
ti.init()  # default: packed=False
a = ti.field(ti.i32, shape=(18, 65))  # padded to (32, 128)
```

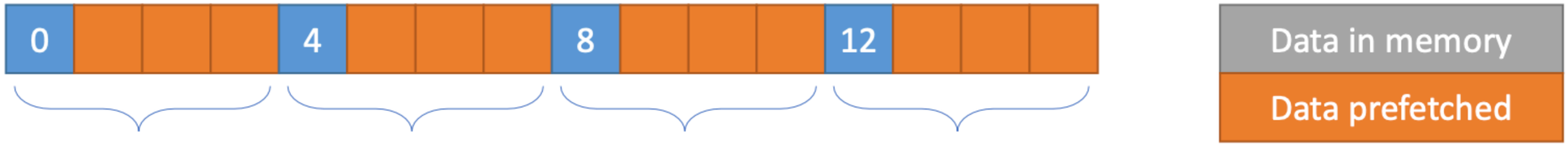
```
ti.init(packed=True)
a = ti.field(ti.i32, shape=(18, 65))  # no padding
```

Taichi: optimized for data-access



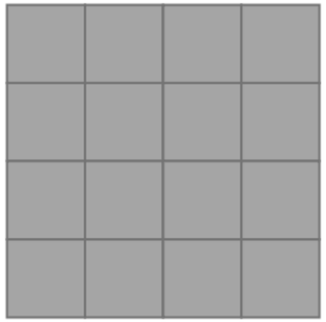

```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

Taichi: optimized for data-access



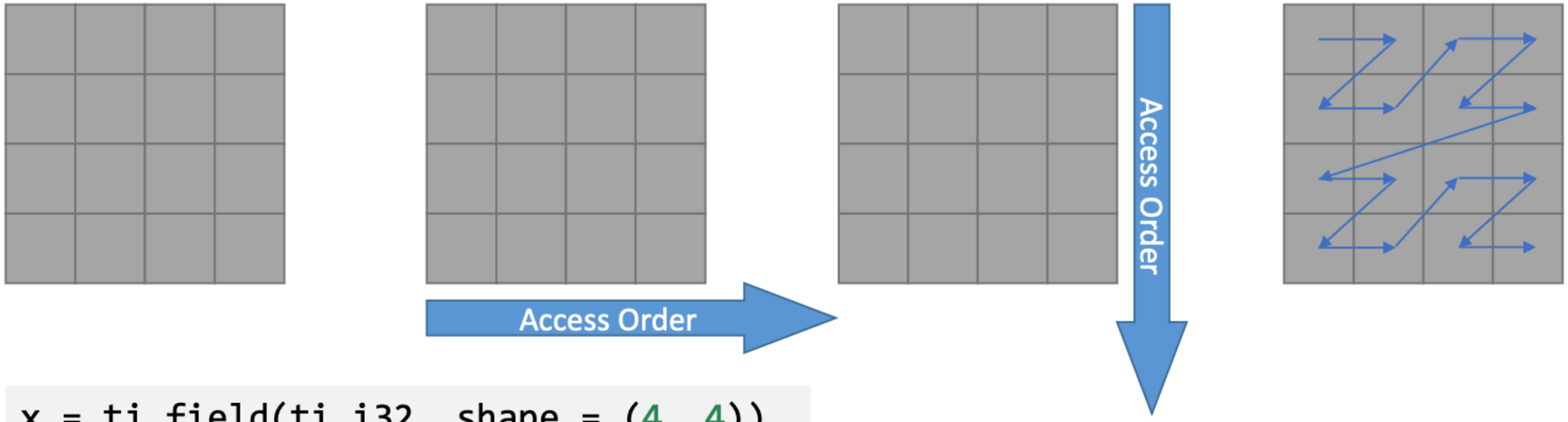

```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

How about multi-dimensional fields?



```
x = ti.field(ti.i32, shape = (4, 4))

@ti.kernel
def fill():
    for i,j in x:
        x[i,j] = 10*i + j

fill()
```

However the access pattern is not determined…

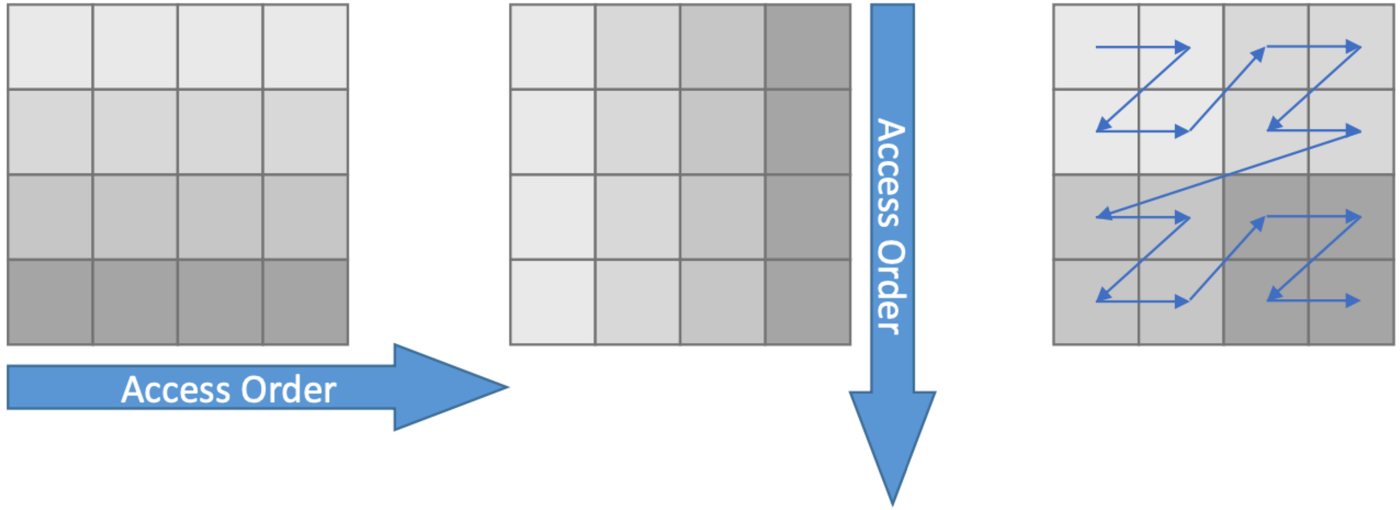


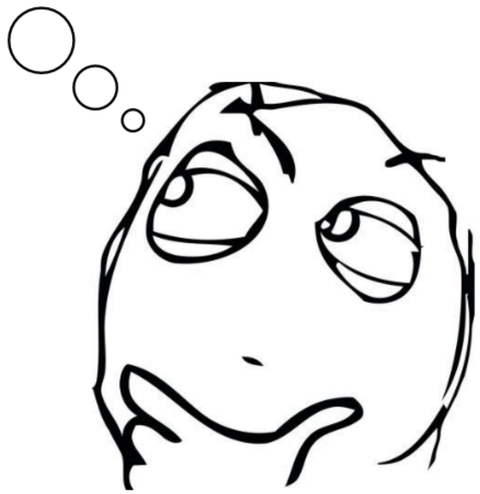

```
x = ti.field(ti.i32, shape = (4, 4))

@ti.kernel
def fill():
    for i,j in x:
        x[i,j] = 10*i + j

fill()
```

What we want:

- memory-access-friendly?
- using ti.field()?

```
x = ti.field(ti.i32, shape = (4, 4))
```





Upgrade your ti.field()

Layout 101: from *shape* to *ti.root*

```
x = ti.Vector.field(3, ti.f32, shape = 16)
```

```
x = ti.Vector.field(3, ti.f32)
ti.root.dense(ti.i, 16).place(x)
```

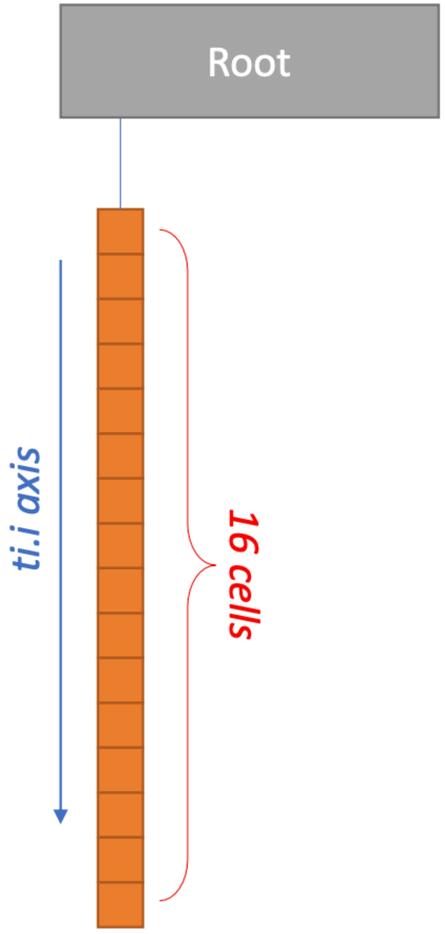
Root

ti.root In English:
Each cell of *root* has a *dense* container with *16 cells* along the *ti.i axis*. Each cell of a *dense* container has *field x*

Layout 101: from *shape* to *ti.root*

```
x = ti.Vector.field(3, ti.f32, shape = 16)
```

```
x = ti.Vector.field(3, ti.f32)
ti.root.dense(ti.i, 16).place(x)
```



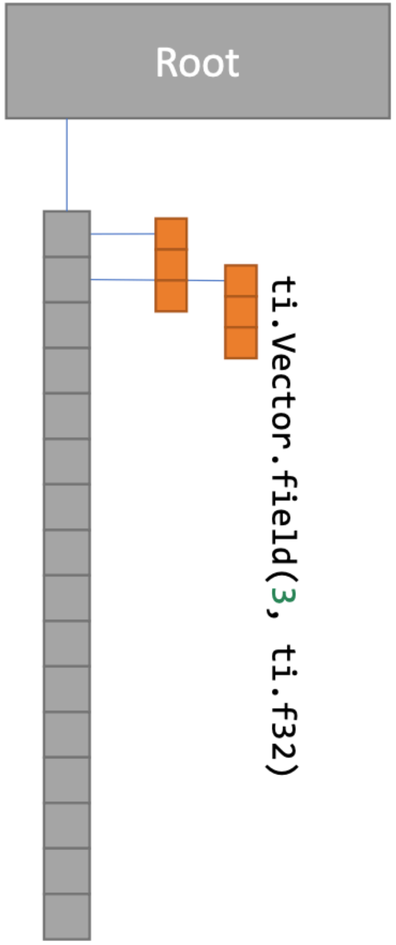
ti.root In English:
Each cell of *root* has a *dense* container with *16 cells* along the *ti.i axis*. Each cell of a *dense* container has *field x*

Layout 101: from *shape* to *ti.root*

```
x = ti.Vector.field(3, ti.f32, shape = 16)
```

```
x = ti.Vector.field(3, ti.f32)
ti.root.dense(ti.i, 16).place(x)
```

ti.root In English:
Each cell of *root* has a *dense* container with *16 cells* along the *ti.i axis*. Each cell of a *dense* container has *field x*



Access row/col-majored fields

```python
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

x = ti.field(ti.i32)
ti.root.dense(ti.i, 3).dense(ti.j, 2).place(x)
# row-major

@ti.kernel
def fill():
    for i,j in x:
        x[i, j] = i*10 + j

@ti.kernel
def print_field():
    for i,j in x:
        print("x[",i,",",j,"]=",x[i,j],sep='', en
d=' ')

fill()
print_field()
```

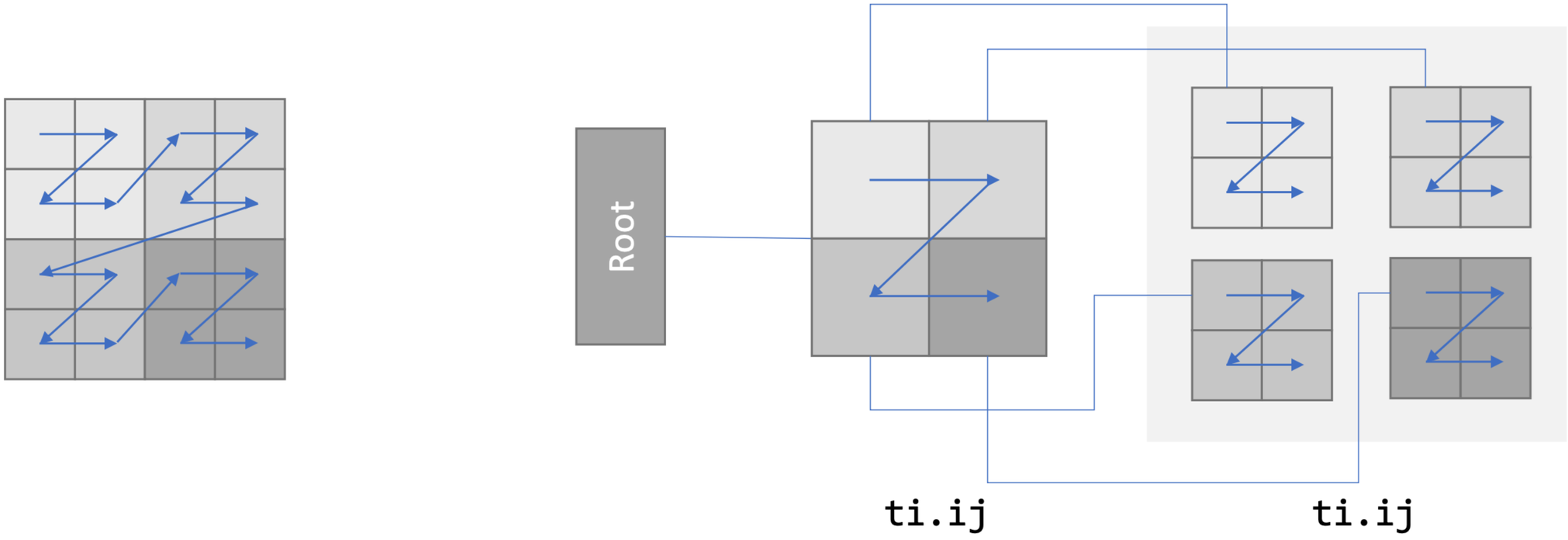
Loop over ***ti.j*** first

```python
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

x = ti.field(ti.i32)
ti.root.dense(ti.j, 2).dense(ti.i, 3).place(x)
# column-major

@ti.kernel
def fill():
    for i,j in x:
        x[i, j] = i*10 + j

@ti.kernel
def print_field():
    for i,j in x:
        print("x[",i,",",j,"]=",x[i,j],sep='', en
d=' ')

fill()
print_field()
```

Loop over ***ti.i*** first

Block-majored access using hierarchical fields

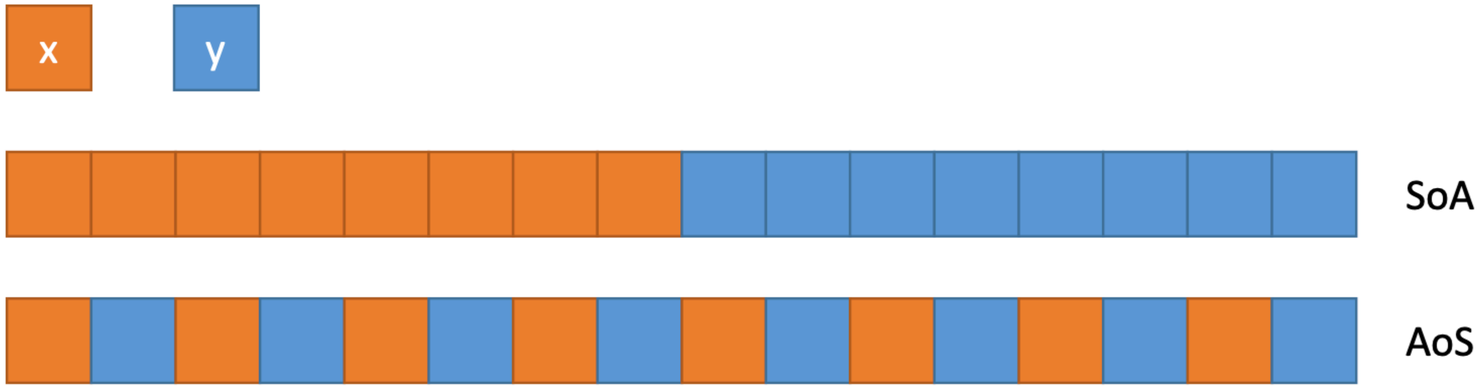
```
x = ti.field(ti.i32)
ti.root.dense(ti.ij, (2,2)).dense(ti.ij, (2,2)).place(x)   # block-major
```



Array of structures (AoS) v.s. structure of arrays (SoA) in C/C++

```
struct S1
{
    int x[8];
    int y[8];
}
S1 soa;
```

```
struct S2
{
    int x;
    int y;
}
S2 aos[8];
```



SoA in Taichi

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 8).place(x)
ti.root.dense(ti.i, 8).place(y)
#  address: low ............................ high
#           x[0]  x[1] … x[7]  y[0]  y[1] … y[7]
```

AoS in Taichi

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 8).place(x, y)
#  address: low ........................... high
#           x[0]  y[0]  x[1]  y[1] … x[7]  y[7]
```

Example2

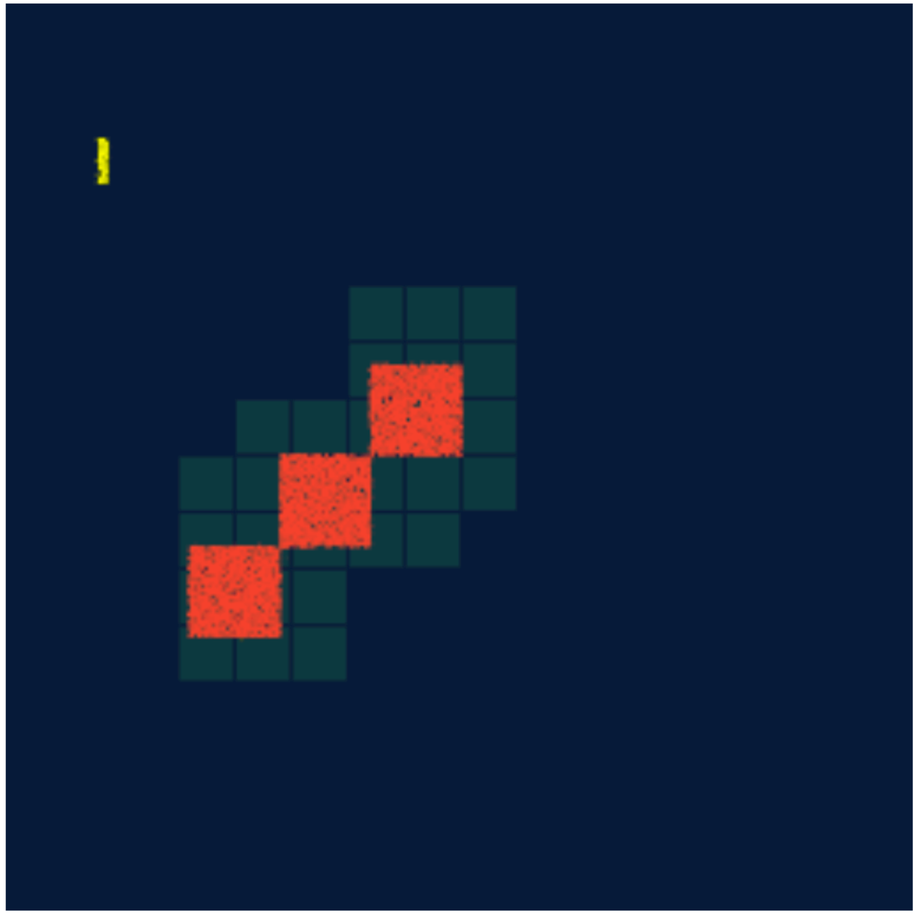
AOS vs SOA

```python
import taichi as ti

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

n = 64 * 1024 * 1024
dim = 4
unroll = 8

a = ti.Vector.field(dim, dtype=ti.i32, shape=n,
                    layout=ti.Layout.SOA)

@ti.kernel
def assign_all():
    for i in range(n):
        for k in ti.static(range(dim)):
            a[i][k] = i + k

@ti.kernel
def assign_single():
    for i in range(n):
        a[i][0] = i

@ti.kernel
def assign_all_random():
    for i_ in range(n):
        for k in ti.static(range(dim)):
            i = (i_ * 10007) & (n - 1) # Random index
            a[i][k] = i + k

for repeat in range(30):
    assign_all()


for repeat in range(30):
    assign_single()


for repeat in range(30):
    assign_all_random()


ti.profiler.print_kernel_profiler_info()
```

https://github.com/Galeselee/taichi_tutorial

Sparse data layout

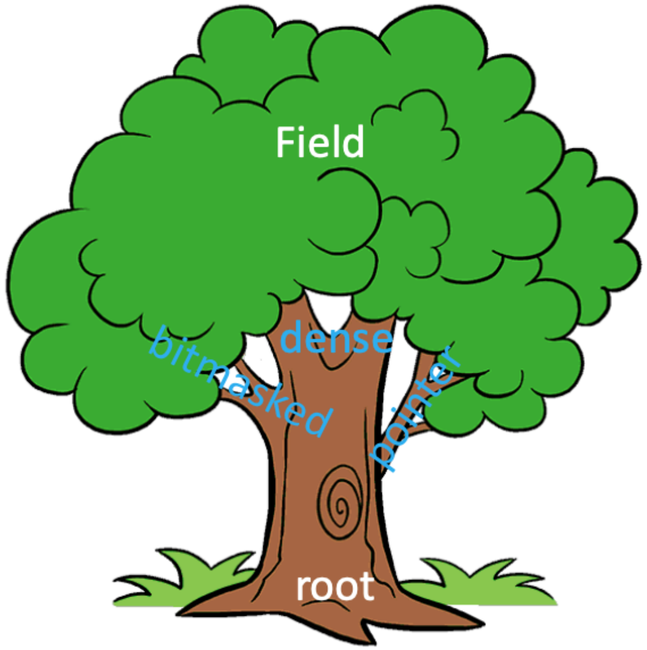
- MPM simulation ➔➔
  - 256x256 grid cells in total
  - Subdivided to 16x16 blocks
    - Each block has 16x16 grid cells
  - Allocating memory for the total 256x256 grid cells is a waste.
    - The dark blocks are filled with zeros anyway



More details: https://docs.taichi-lang.org/docs/sparse

The SNode-tree

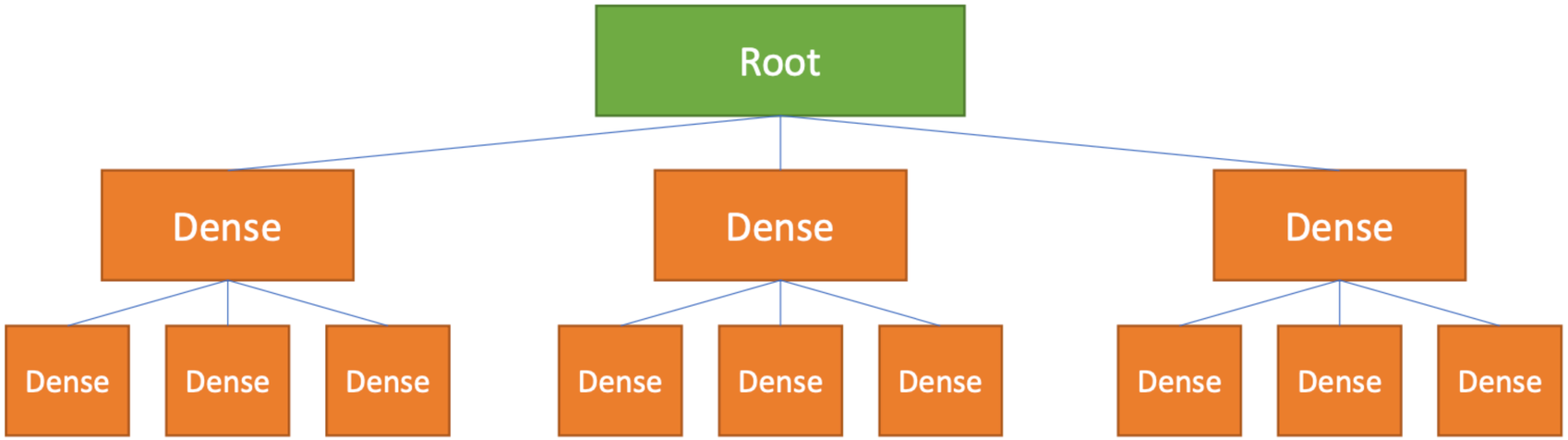
- root: the root of the data structure
- dense: a fixed-length contiguous array.
- bitmasked: similar to dense, but it also uses a mask to maintain ***sparsity*** information, one bit per child.
- pointer: stores pointers instead of the whole structure to save memory and maintain ***sparsity***



Sparse computation! Then how?

- A dense SNode-tree:

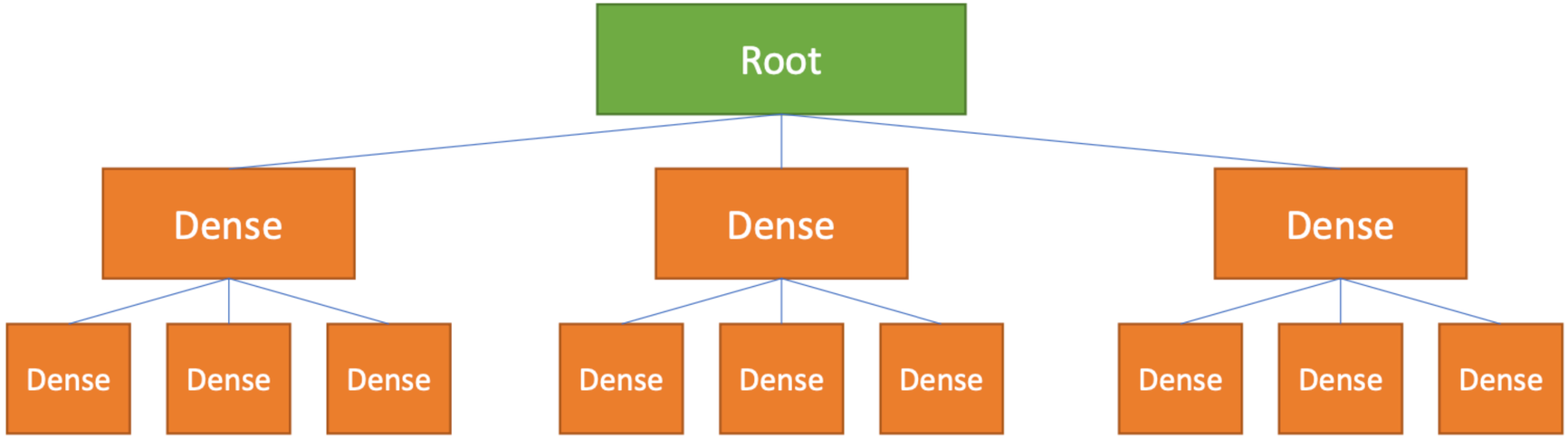
```
x = ti.field(ti.i32)
block1 = ti.root.dense(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.dense(ti.i,3)
.dense(ti.j,3).place(x)
```



Sparse computation! Then how?

- A dense SNode-tree:

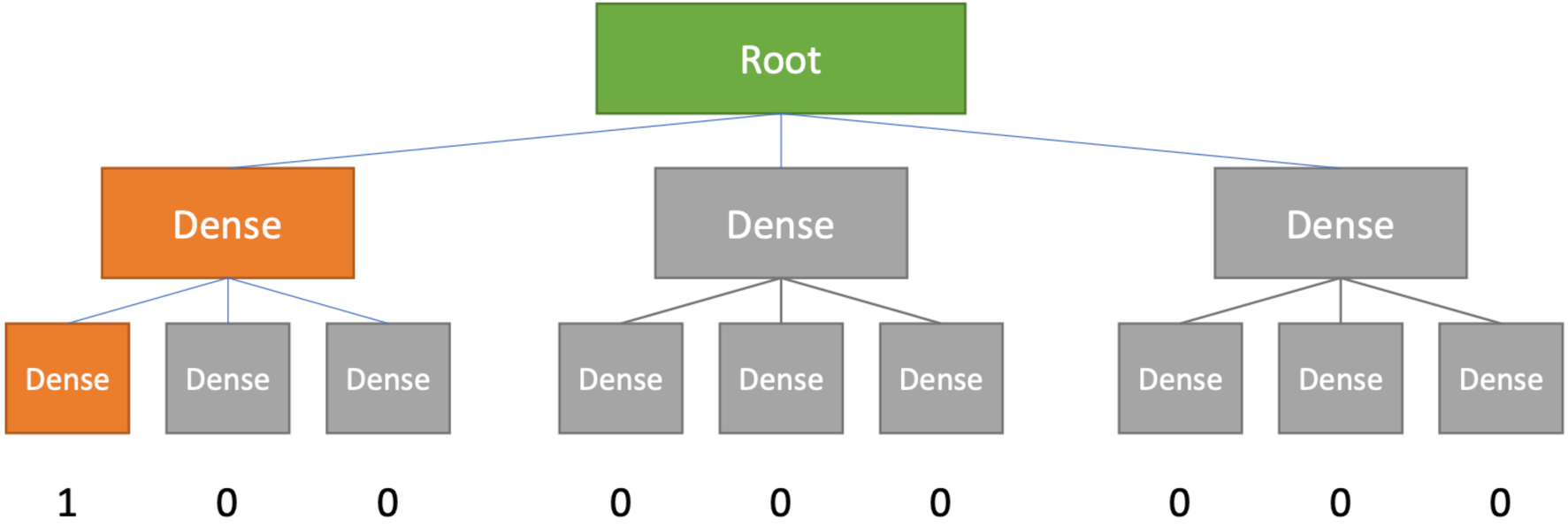
1	0	0
0	0	0
0	0	0



Sparse computation! Then how?

- A dense SNode-tree:

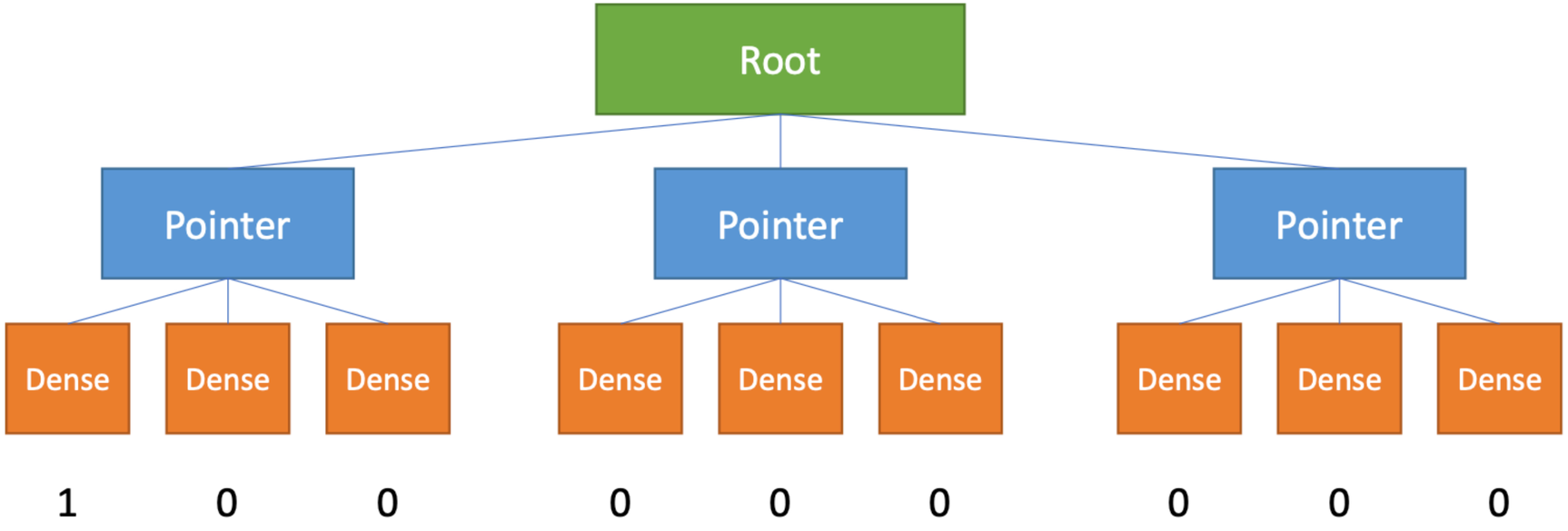
1	0	0
0	0	0
0	0	0



From .dense() to .pointer()

- A sparse SNode-tree:

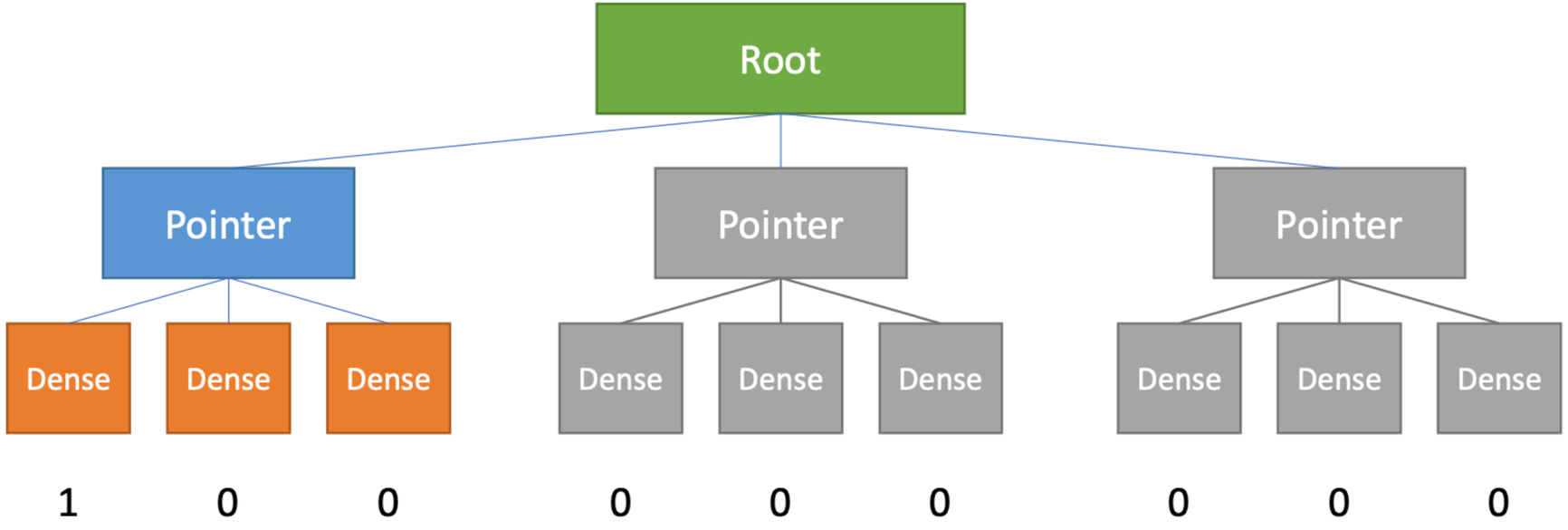
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```



From .dense() to .pointer()

- A sparse SNode-tree:

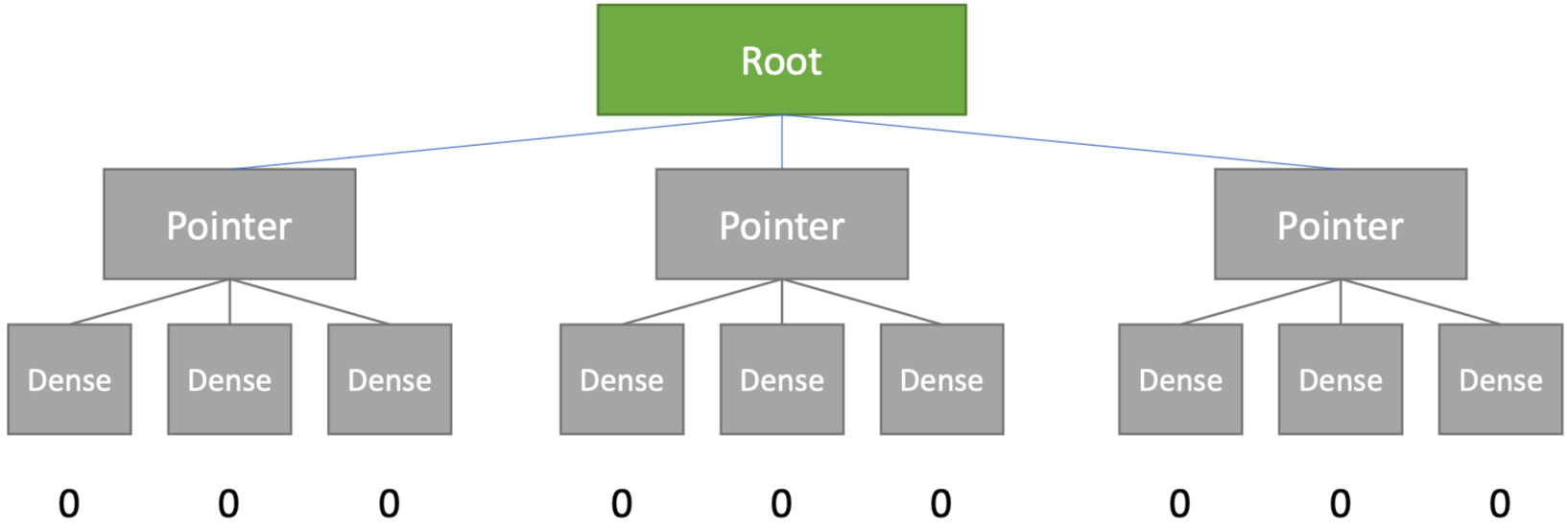
1	0	0
0	0	0
0	0	0



Activation

- A sparse SNode-tree born empty:

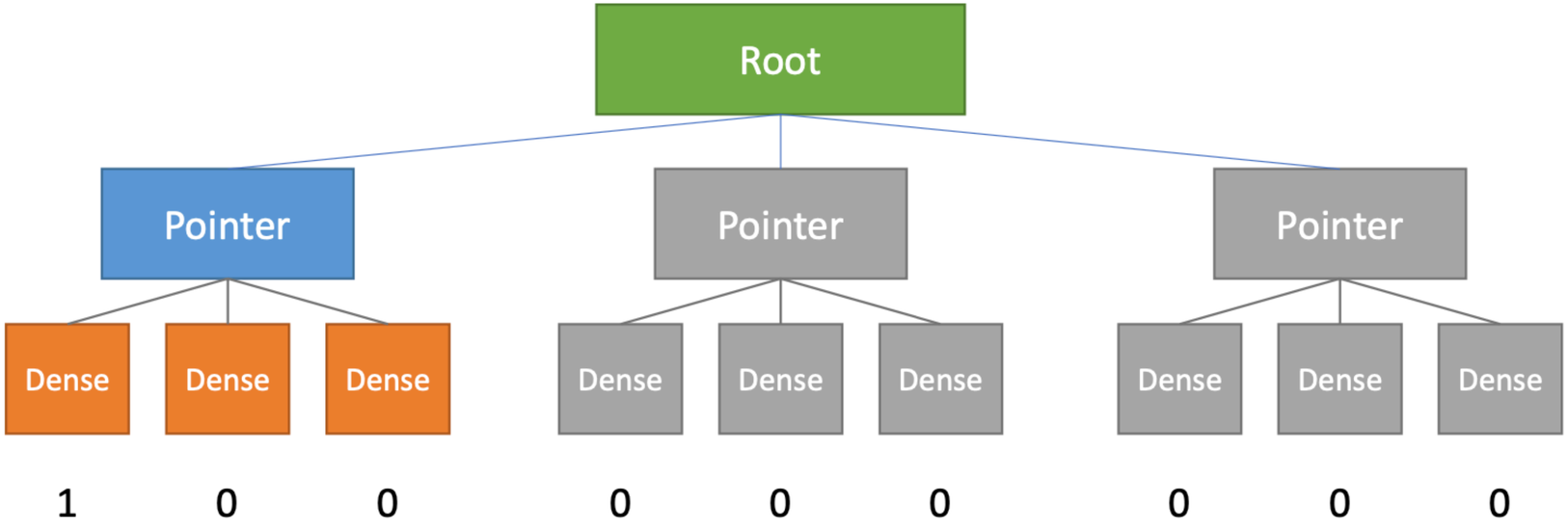
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```



Activation

- Once ***writing*** an inactive cell:

```
x[0,0] = 1
# activates block1[0] and thereby block2[0],
 block2[1] and block2[2]
```

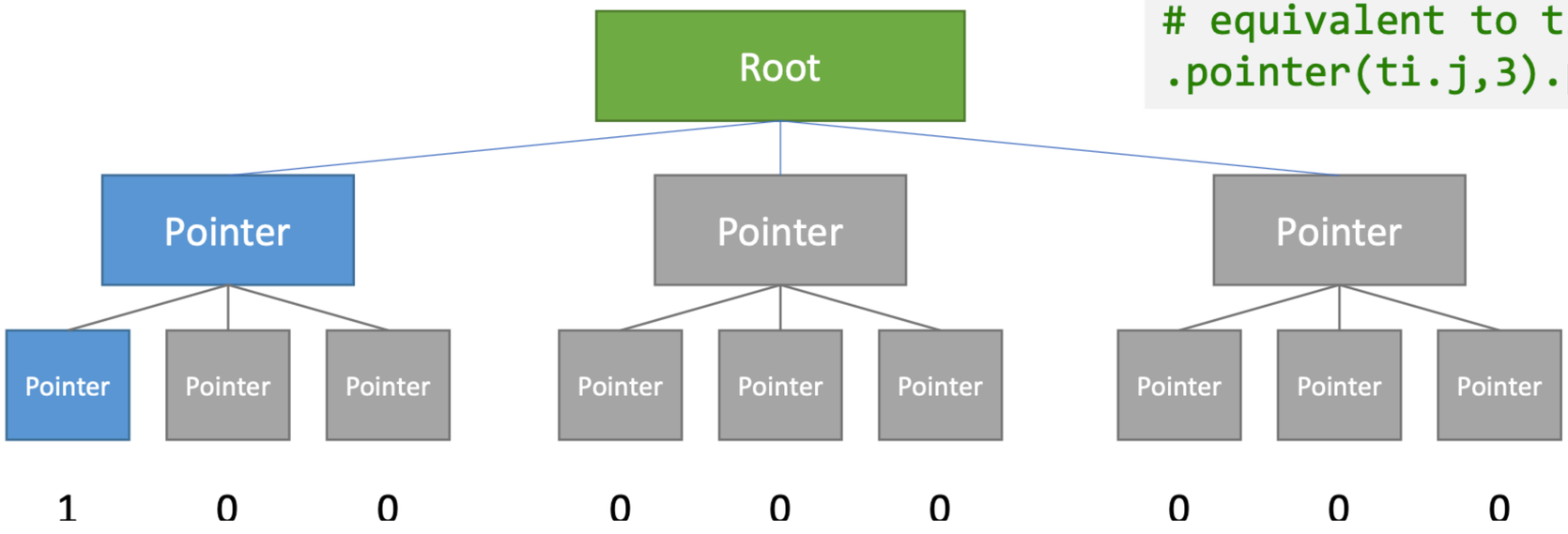


https://docs.taichi-lang.org/docs/sparse#explicitly-manipulating-and-querying-sparsity

Why not using pointer everywhere?

- Bad design idea:
  - a ti.f32 → 32 bits
  - a taichi pointer → 64 bits

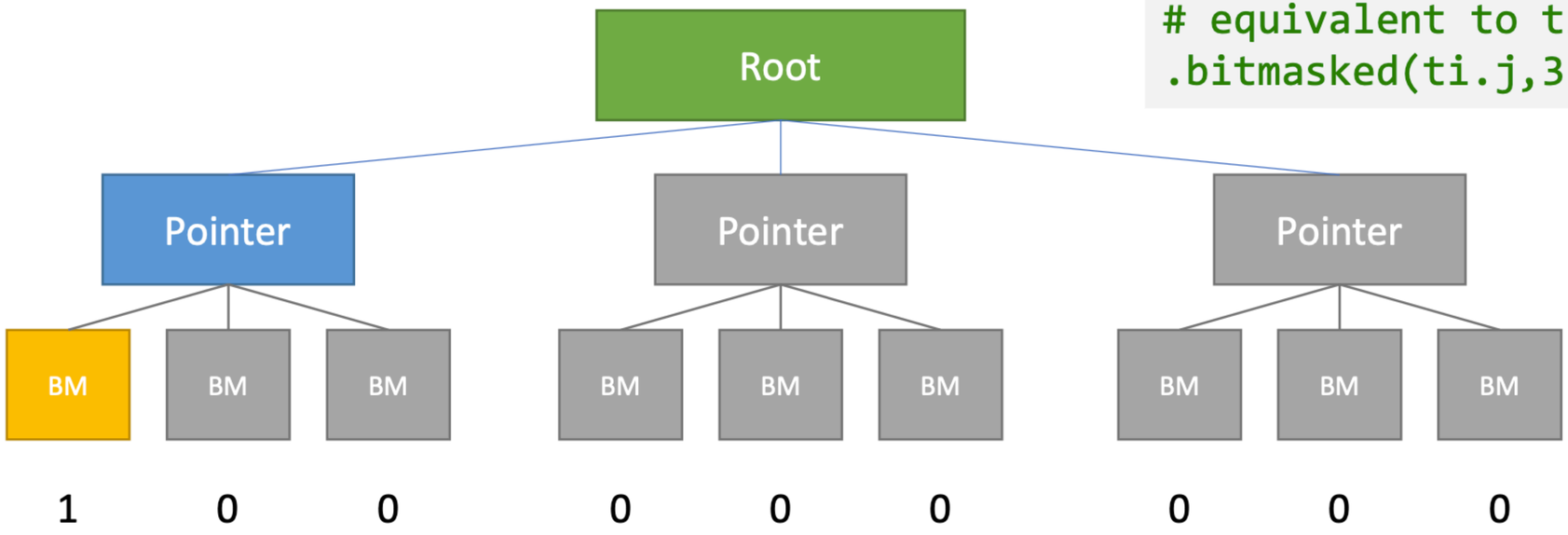
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.pointer(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.pointer(ti.j,3).place(x)
```



BitMasked

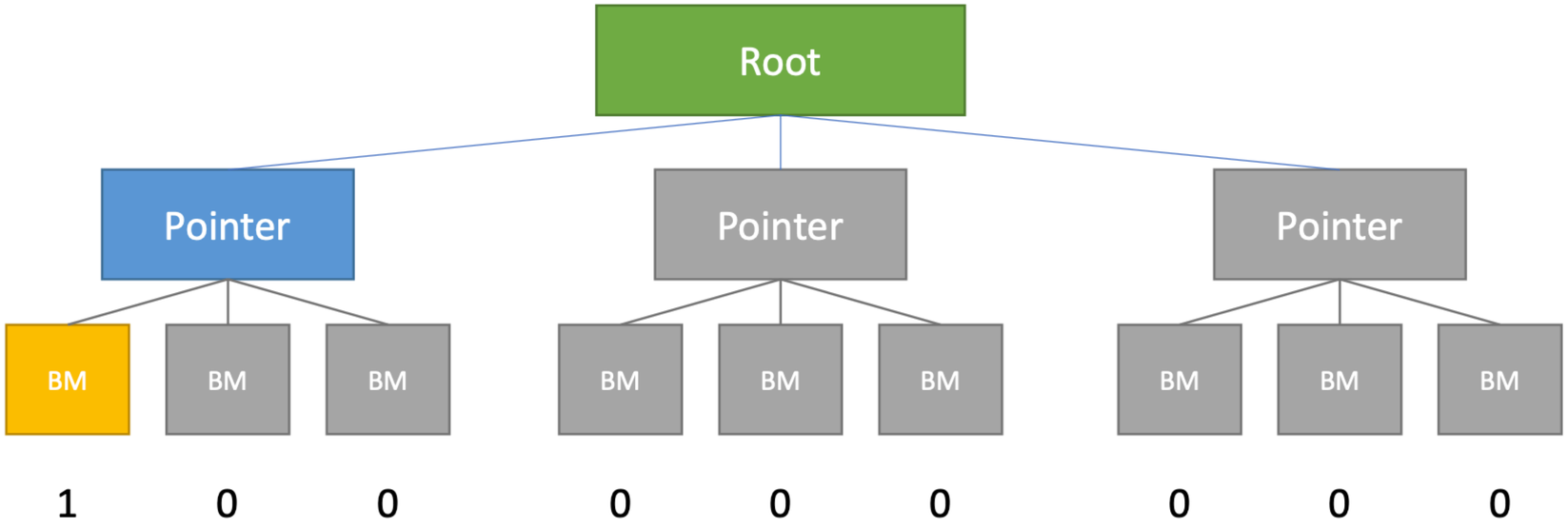
- Works for leaf cells only
- Each leaf cell has its own activation flag

```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.bitmasked(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.bitmasked(ti.j,3).place(x)
```



BitMasked

- Cost 1-bit-per-cell extra
- Skip struct-for(s) when bitmasked inactive



Example3

Taichi-sparse

```
from taichi.examples.patterns import taichi_logo

import taichi as ti

ti.init(arch=ti.cpu, offline_cache=False)

n = 512
x = ti.field(dtype=ti.i32)
res = n + n // 4 + n // 16 + n // 64
img = ti.field(dtype=ti.f32, shape=(res, res))

block1 = ti.root.pointer(ti.ij, n // 64) # 8 times 8 pointers
block2 = block1.pointer(ti.ij, 4)
block3 = block2.pointer(ti.ij, 4)
block3.dense(ti.ij, 4).place(x)
```

https://github.com/Galeselee/taichi_tutorial

Debugger

Runtime Print

```python
import taichi as ti
ti.init(arch=ti.cuda)

@ti.kernel
def kern():
    print('inside kernel')

print('before kernel')
kern()
print('after kernel')
ti.sync()
print('after sync')

# print ==>
# before kernel
# after kernel
# inside kernel
# after sync
```

- Hurts your performance
  - Print requires a system call
- Produces randomly ordered results in your parallel for loops
- Support comma-separated parameters in the Taichi Scope

```python
@ti.kernel
def foo():
    print('a[0] = ', a[0]) # right
    print(f'a[0] = {a[0]}') # wrong
    print("a[0] = %f" % a[0]) # wrong
```

Debugger

Compiler-time Print

- Print Python-scope objects and constants in Taichi scope
  - Will print only once at compile-time

```
x = ti.field(ti.f32, (2, 3))
y = 1
A = ti.Matrix([[1, 2], [3, 4], [5, 6]])

@ti.kernel
def inside_taichi_scope():
    ti.static_print(y)
    # => 1
    ti.static_print(x.shape)
    # => (2, 3)
    ti.static_print(A.n)
    # => 3
    for i in range(4):
        ti.static_print(A.m)
        # => 2
        # will only print once
```

Debugger

Debug mode

- Turn on debug mode when you debug
  - debug mode is off by default
  - debug mode will do more synchronize operations which will hurt performance

```
ti.init(arch=ti.cpu, debug=False)

x = ti.field(ti.f32, shape = 4)

print("x[4] =", x[4]) # x[4] = 0.0
```

```
ti.init(arch=ti.cpu, debug=True)

x = ti.field(ti.f32, shape = 4)

print("x[4] =", x[4]) # RuntimeError
```

Mostly seen problems

case0: data race

```
x = ti.field(ti.f32, shape = 16)

@ti.kernel
def foo():
    for i in range(1,15):
        x[i-1] = x[i-1] + 2
        x[i+1] = x[i+1] + 4
```

```
x = ti.field(ti.f32, shape = 16)

@ti.kernel
def foo():
    for i in range(1,15):
        x[i-1] += 2
        x[i+1] += 4
```

Mostly seen problems

case1: deepcopy and shallow copy

```
a = ti.field(ti.f32, shape=())
b = ti.field(ti.f32, shape=())

a[None] = 0
b[None] = 1
print(a[None]) # 0.0
b = a
b[None] = 128
print(a[None]) # 128.0
```

```
a = ti.field(ti.f32, shape=())
b = ti.field(ti.f32, shape=())

a[None] = 0
b[None] = 1
print(a[None]) # 0.0
b.copy_from(a)
b[None] = 128
print(a[None]) # 0.0
```

Mostly seen problems

Case2: data types in Taichi are static

```
x = ti.field(ti.f32, shape=128)

@ti.kernel
def reduction():
    sum = 0
    for i in x:
        sum += x[i]

    print(sum) # integer
```

```
[Taichi] version 0.8.3, llvm 10.0.0, commit 021af5d2, win, python 3.8.10
[Taichi] Starting on arch=x64
[W 10/19/21 11:01:47.465 2260] [type_check.cpp:taichi::lang::TypeCheck::visit@70] [$15]
Atomic add (f32 to i32) may lose precision, at
[W 10/19/21 11:01:47.467 2260] [type_check.cpp:taichi::lang::TypeCheck::visit@71]

0
```

Profiler

simple profiler

- Enable Taichi's kernel profiler

```
ti.init(kernel_profiler=True, arch=ti.gpu)
```

- Output your profiling info

```
ti.print_kernel_profile_info("count")
```

- Clear profiling info

```
ti.clear_kernel_profile_info()
```

Performance tuning

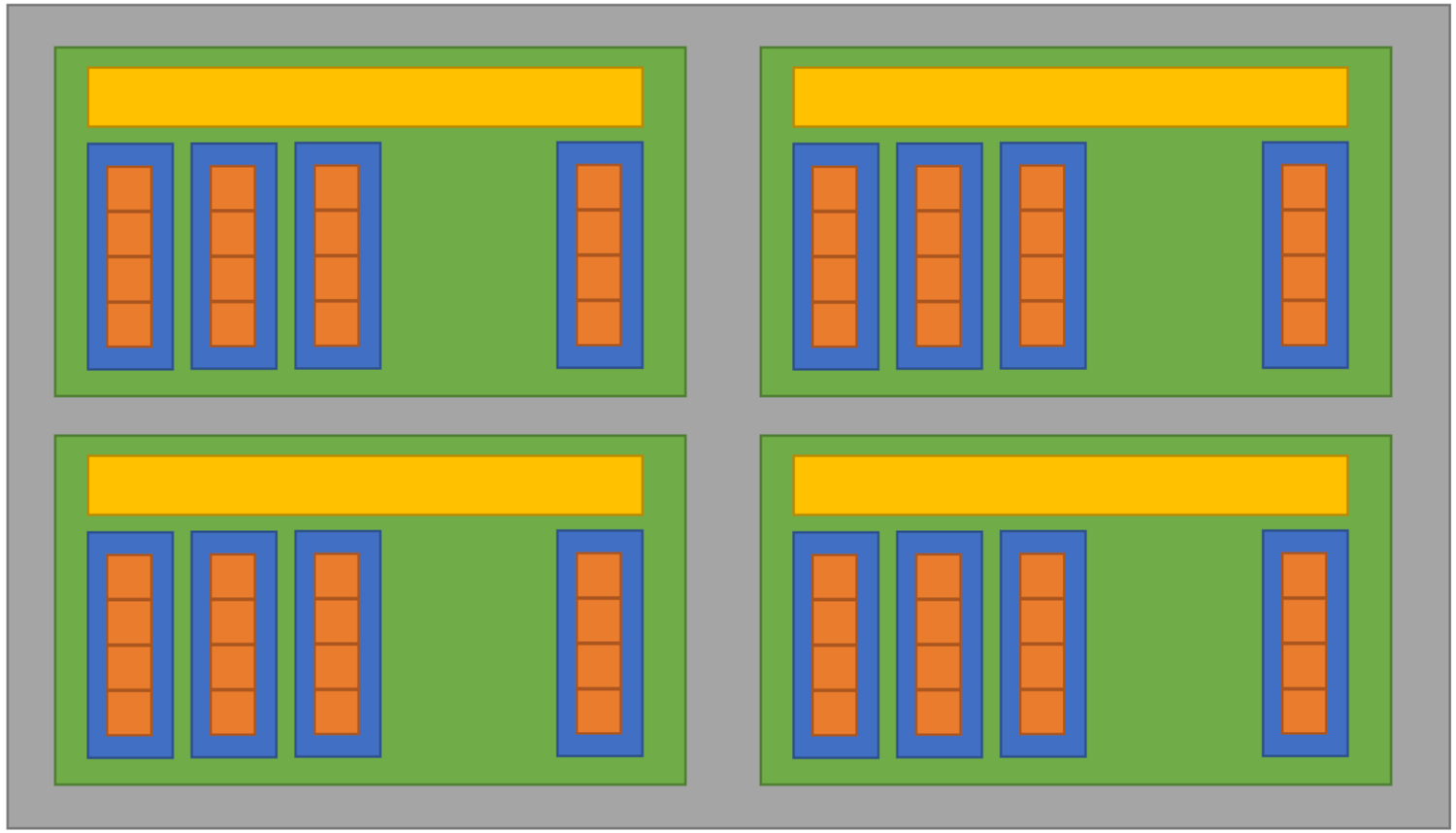
- Find the slowest component in program (bottleneck)

- Optimize that slowest component

```
----------------------------------------------------------------------------
Kernel Profiler(count, default) @ X64
============================================================================
[      %     total   count |      min       avg       max   ] Kernel name
----------------------------------------------------------------------------
[ 92.28%  14.119 s     30x |  466.911   470.618   496.956 ms] assign_all_random_c110_0_kernel_0_range_for
[  4.54%   0.695 s     30x |   22.311    23.173    34.396 ms] assign_all_c106_0_kernel_0_range_for
[  3.18%   0.486 s     30x |   15.961    16.193    17.635 ms] assign_single_c108_0_kernel_0_range_for
----------------------------------------------------------------------------
[100.00%] Total execution time:  15.300 s   number of results: 3
============================================================================
```

Optimization tip

BLS - background



Iteration: each iteration in a for-loop

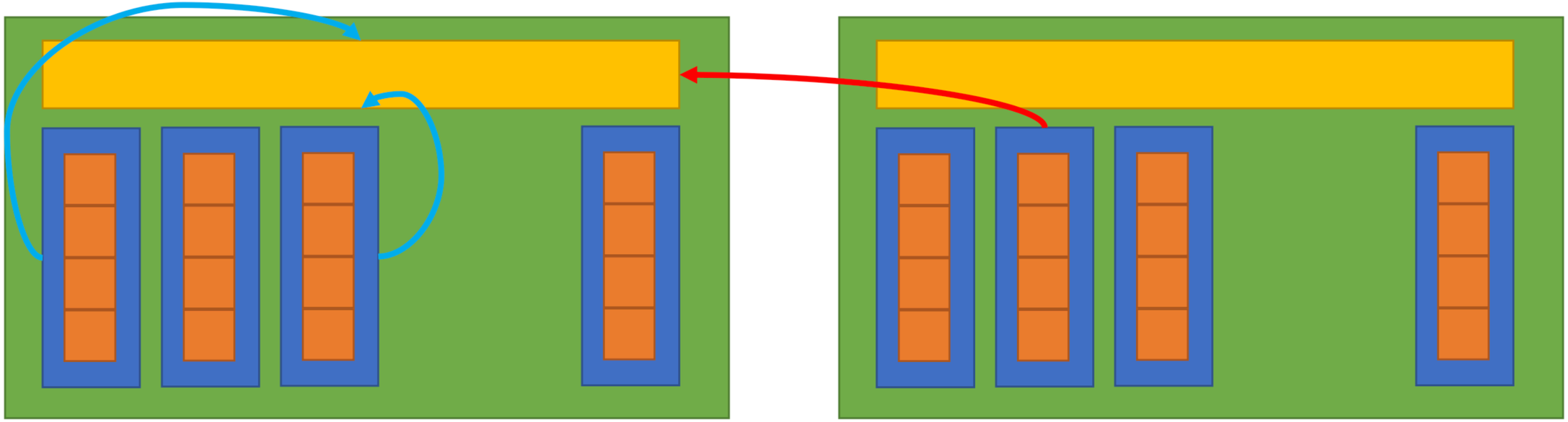
Thread: the minimal parallelizable unit

Block: threads are grouped in blocks with shared block local storage.

Grid: the minimal unit that being launched from the host

BLS (block local storage)

- Implemented using the shared memory in GPU
- Fast to read/write but small in size



BLS (block local storage)

- Decide the size of the blocks by prepending a *ti.block_dim()* before a parallel for-loop: (default_block_dim = 256 Bytes)

```
@ti.kernel
def func():
    for i in range(8192):  # no decorator, use default settings
        ...

    ti.block_dim(128)      # change the property of next for-loop:
    for i in range(8192):  # will be parallelized with block_dim=256
        ...

    for i in range(8192):  # no decorator, use default settings
        ...
```

BLS (block local storage)

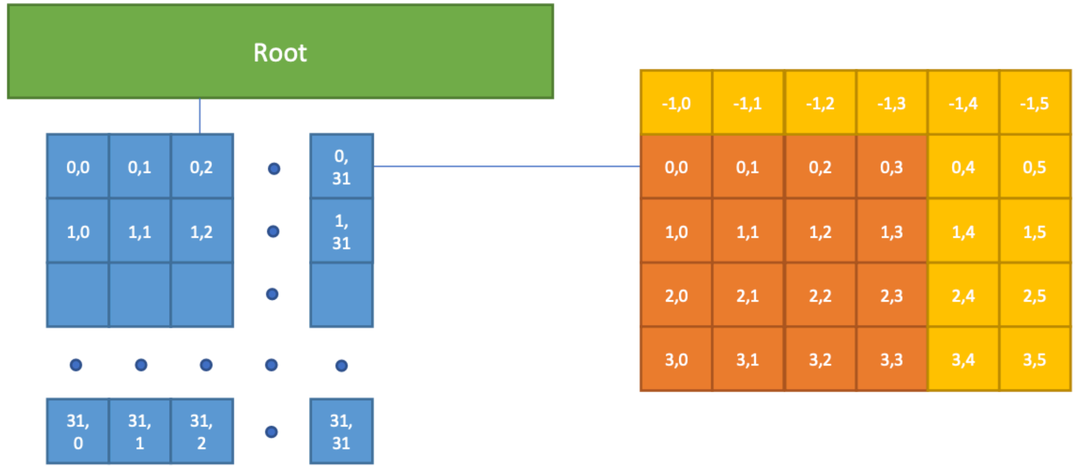
- Cache the most frequently-used data into BLS manually using *ti.block_local()*:

```
a = ti.field(ti.f32)
# `a` has a block size of 4x4
ti.root.pointer(ti.ij, 32).dense(ti.ij, 4).place(a)

@ti.kernel
def foo():
  # Taichi will cache `a` into the CUDA shared memory
  ti.block_local(a)
  for i, j in a:
    print(a[i - 1, j], a[i, j + 2])
```

BLS (block local storage)

- Cache the most frequently-used data into BLS manually using *ti.block_local()*:

```
ti.block_local(a)
for i, j in a:
  print(a[i - 1, j], a[i, j + 2])
```



bls_size = 5x6(x4 Bytes)
= 120 Bytes

Advanced Topic

Taichi x Pytorch

- Pytorch: 提供以Tensor为操作对象的算子，提供的算子一般都经过了特殊优化
- Taichi: Taichi适合更精细的控制和并行每个元素的操作，更为灵活
- 结合的场景
  - Taichi做数据预处理
  - 定制高性能ML算子

定制ML算子

RWKV：一种类似于transformer的网络结构

- 类似于一维卷积的定制算子

```
def run_formula_very_slow(w, k, B, C, T, eps):
    out = torch.empty((B, C, T), device='cpu')
    for b in range(B):
        for c in range(C):
            for t in range(T):
                s = eps
                for u in range(t-T+1, t+1):
                    s += w[c][0][(T-1)-(t-u)] * k[b][c][u+T-1]
                out[b][c][t] = s
    return out
```

- Pytorch实现

```
out = eps + F.conv1d(nn.ZeroPad2d((T-1, 0, 0, 0))(k), w.unsqueeze(1), groups=C)
```

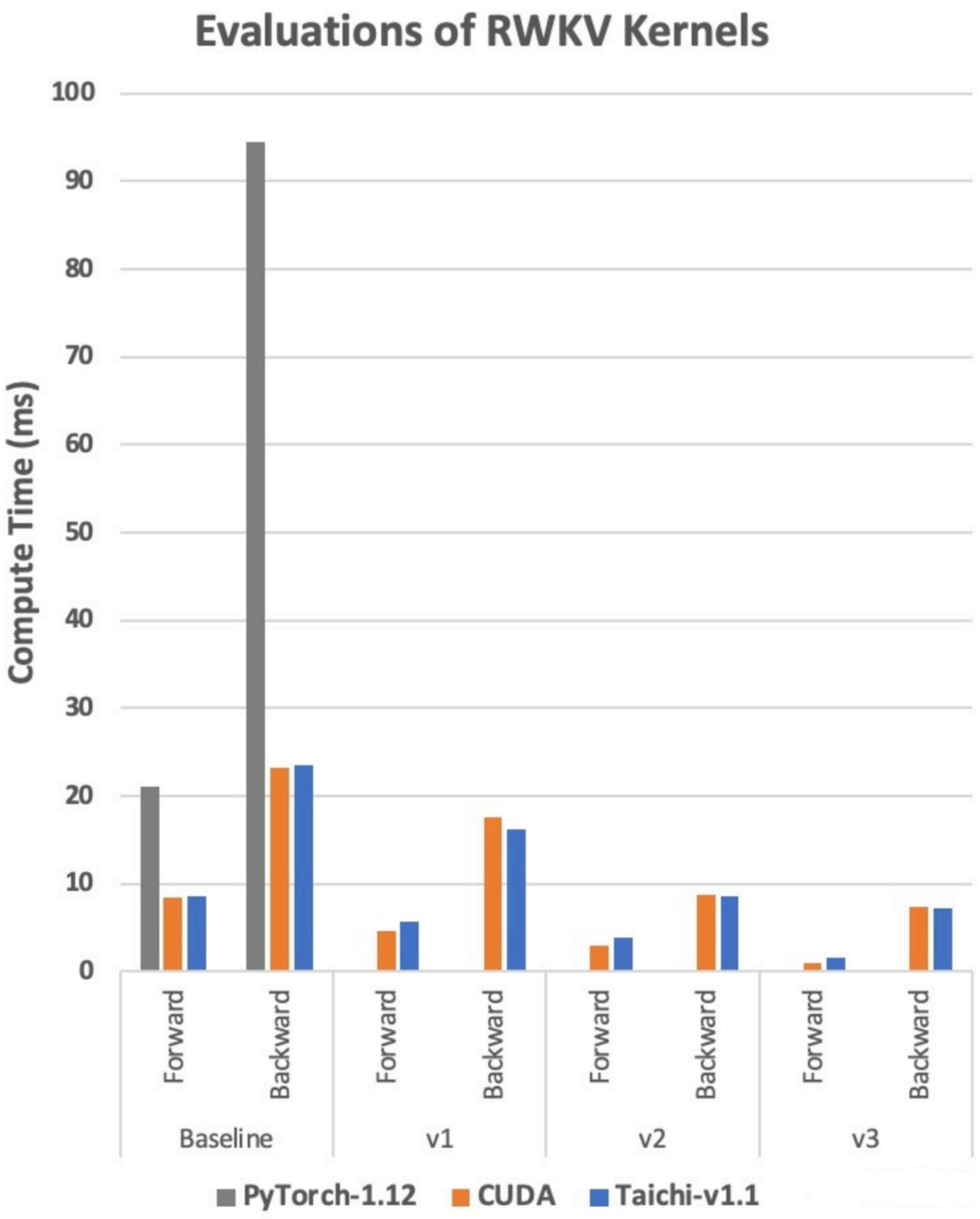
定制ML算子

Taichi 实现

```
@ti.kernel
def taichi_forward_v0(
        out: ti.types.ndarray(field_dim=3),
        w: ti.types.ndarray(field_dim=3),
        k: ti.types.ndarray(field_dim=3),
        eps: ti.f32):

    for b, c, t in out:
        s = eps
        for u in range(t-T+1, t+1):
            s += w[c, 0, (T-1)-(t-u)] * k[b, c, u+T-1]
        out[b, c, t] = s
```

定制ML算子

性能比较



https://github.com/BlinkDL/RWKV-CUDA
https://github.com/ailzhang/blog_code

Taichi x MPI

MPI4Py

- MPI4Py

```python
# mpi_hello.py
from mpi4py import MPI


comm = MPI.COMM_WORLD
print(f"Hello from MPI worker {comm.rank}, the MPI world has {comm.size} peers.")
```

- Launch

```
mpirun -np 8 --hosts node1,node2 python mpi_hello.py
```

- Result

```
Hello from MPI worker 0, the MPI world has 8 peers.
Hello from MPI worker 6, the MPI world has 8 peers.
Hello from MPI worker 1, the MPI world has 8 peers.
Hello from MPI worker 4, the MPI world has 8 peers.
Hello from MPI worker 2, the MPI world has 8 peers.
Hello from MPI worker 3, the MPI world has 8 peers.
Hello from MPI worker 5, the MPI world has 8 peers.
Hello from MPI worker 7, the MPI world has 8 peers.
```

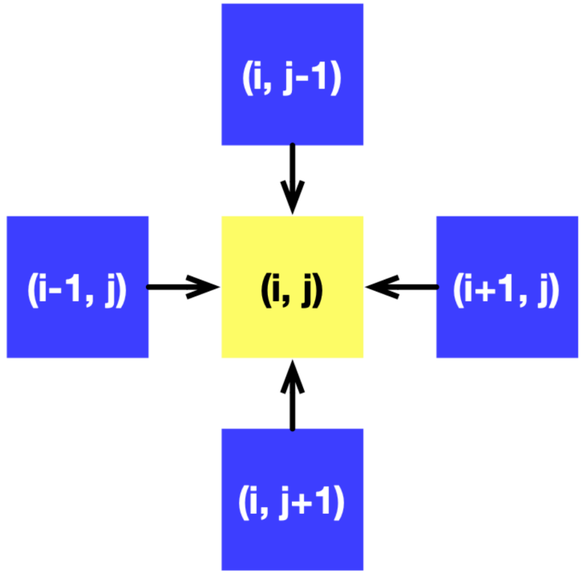
Example

Poisson: problem description

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = \sin(\pi x)\sin(\pi y), \quad (x, y) \in [0, 1] \times [0, 1].$$

- 边界条件：f(x,y) = 0

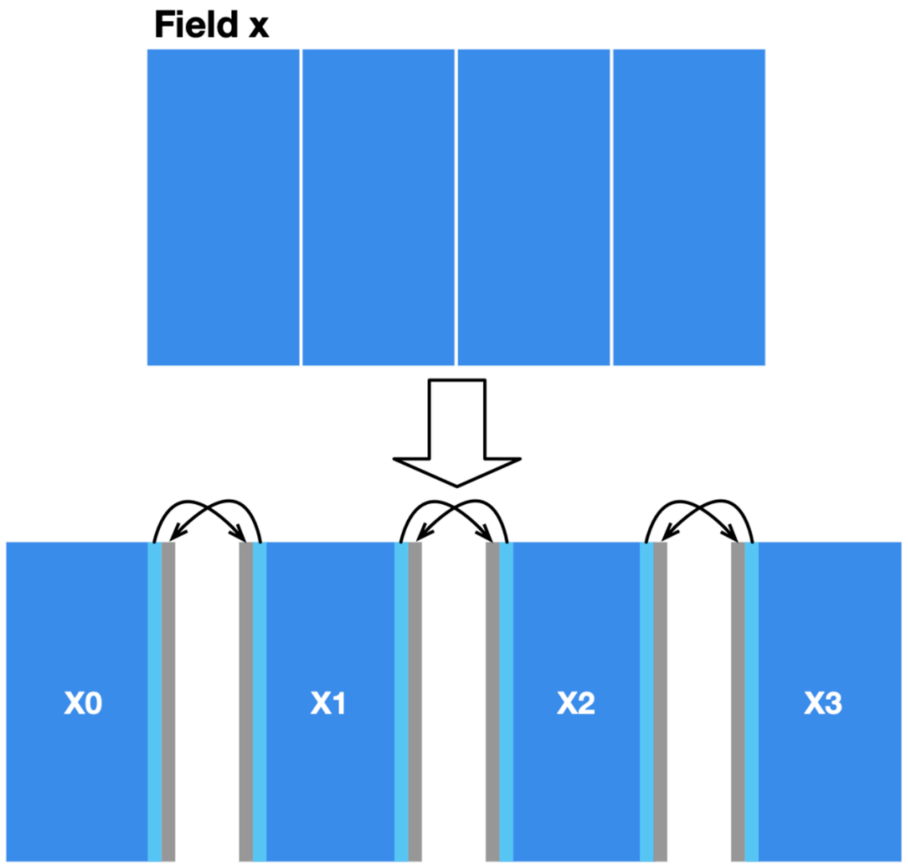
```
x = ti.field(dtype=float, shape=(N + 2, N + 2))
xt = ti.field(dtype=float, shape=(N + 2, N + 2))


@ti.kernel
def substep():
    for i,j in ti.ndrange((1, N+1), (1, N+1)):
        xt[i,j] = (-b[i,j]*dx**2+x[i+1,j]+x[i-1,j]+x[i,j+1]+x[i,j-1]) / 4.0
    for I in ti.grouped(x):
        x[I] = xt[I]
```



https://github.com/turbo0628/Taichi-MPI.git

Example

Poisson: data partition




```
n = N // 4
x = ti.field(dtype=float, shape=(N + 2, n + 2))
```

- x 不再表示完整的 field，而是每个进程自己可见的子场。在不同的进程中 x 对应不同的子场。

https://github.com/turbo0628/Taichi-MPI.git

Example

Poisson: data transfer

- MPI4Py 支持使用Numpy的ndarray来作为数据的传输缓冲区，只要保证传递的ndarray在内存中是连续的即可
- Field数据结构提供了`to_numpy`和`from_numpy`与numpy进行数据交换（有额外的copy）

```
comm = MPI.COMM_WORLD

x_arr = x.to_numpy()
x_right = np.ascontiguousarray(x_arr[:, -2])

if comm.rank < num_workers - 1:
    comm.Send(x_right, dest=comm.rank+1)
```

```
shadow_left = np.zeros(N + 2)
if comm.rank > 0:
    comm.Recv(shadow_left, source=comm.rank-1)

# Fill in the left edge
x_arr[:, 0] = shadow_left
x.from_numpy(x_arr)
```

https://github.com/turbo0628/Taichi-MPI.git

Example

Poisson: data copy

There are two ways to import a NumPy array `arr` to the Taichi scope:

- Create a Taichi field `f`, whose shape and dtype match the shape and dtype of `arr`, and call `f.from_numpy(arr)` to copy the data in `arr` into `f`. This approach is preferred when the original array is visited frequently from elsewhere in the Taichi scope (for example, in the case of texture sampling).

- Pass `arr` as an argument to a kernel or a Taichi function using `ti.types.ndarray()` as type hint. The argument is passed by reference without creating a copy of `arr`. Thus, any modification to this argument from inside a kernel or Taichi function also changes the original array `arr`. This approach is preferred when the kernel or Taichi function that takes in the argument needs to process the original array (for storage or filtering, for example).

https://docs.taichi-lang.org/docs/external

Example

Poisson: optimization

- 调用`to_numpy`会把整个field转换为numpy数组，而实际发送的数据只包含场的边界部分，有一部分数据拷贝可以省掉

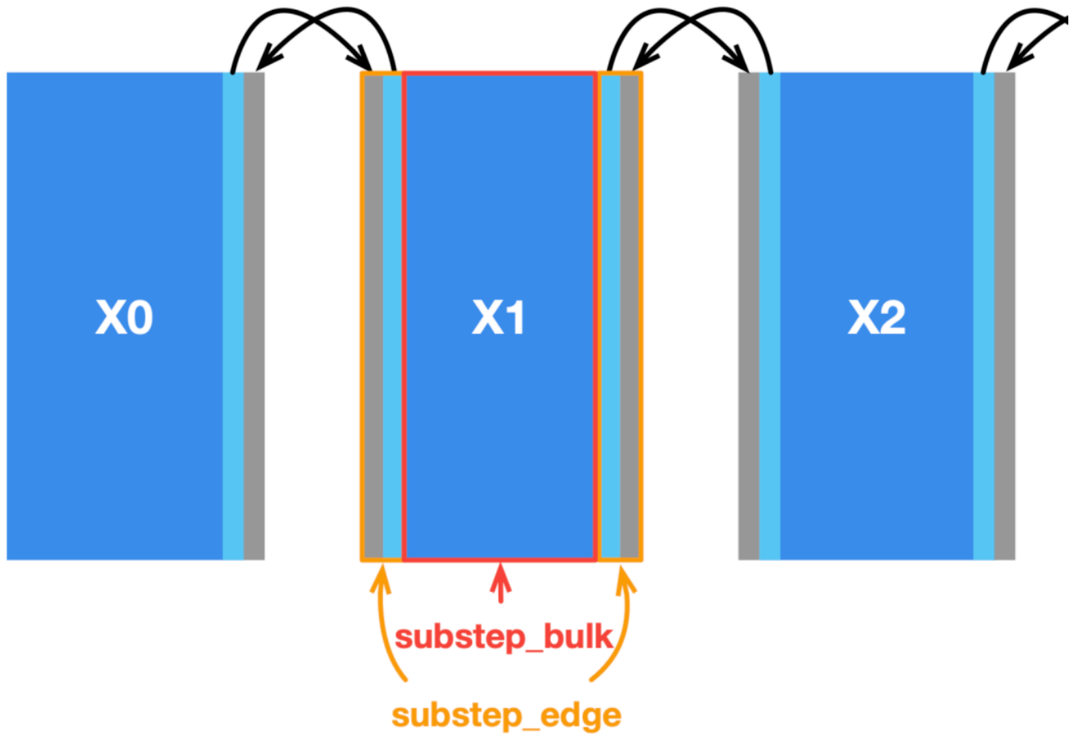
```
@ti.kernel
def fill_shadow(shadow_left : ti.types.ndarray(),
                shadow_right : ti.types.ndarray()):
    for i in ti.ndrange((0, N + 2)):
        x[i, 0] = shadow_left[i]
        x[i, n+1] = shadow_right[i]
```

https://github.com/turbo0628/Taichi-MPI.git

Example

Poisson: optimization

- 每个子场在收到数据之后才会进行下一步的迭代计算，但实际上大部分计算发生在子场内部，他们没有必要等待数据交换



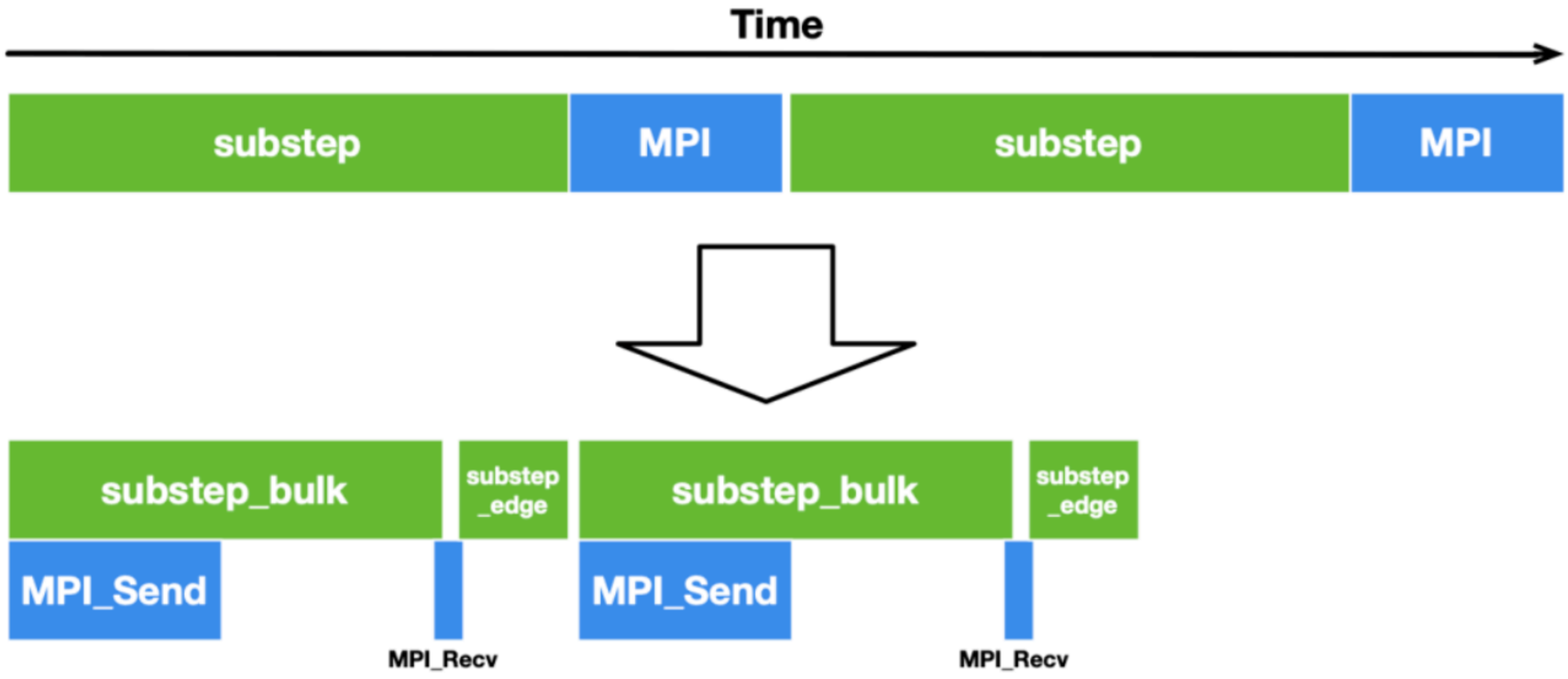

```
if comm.rank < num_workers - 1:
    comm.Isend(edge_right, dest=comm.rank+1, tag=11)
```

```
if comm.rank > 0:
    comm.Irecv(shadow_left, source=comm.rank-1, tag=11).Wait()
```

```
def step():
    mpi_send_edges() # Trigger communication
    substep_bulk() # Compute the bulk
    mpi_recv_edges() # Wait for communications
    substep_edge(shadow_left, shadow_right) # Compute the edges
```

https://github.com/turbo0628/Taichi-MPI.git

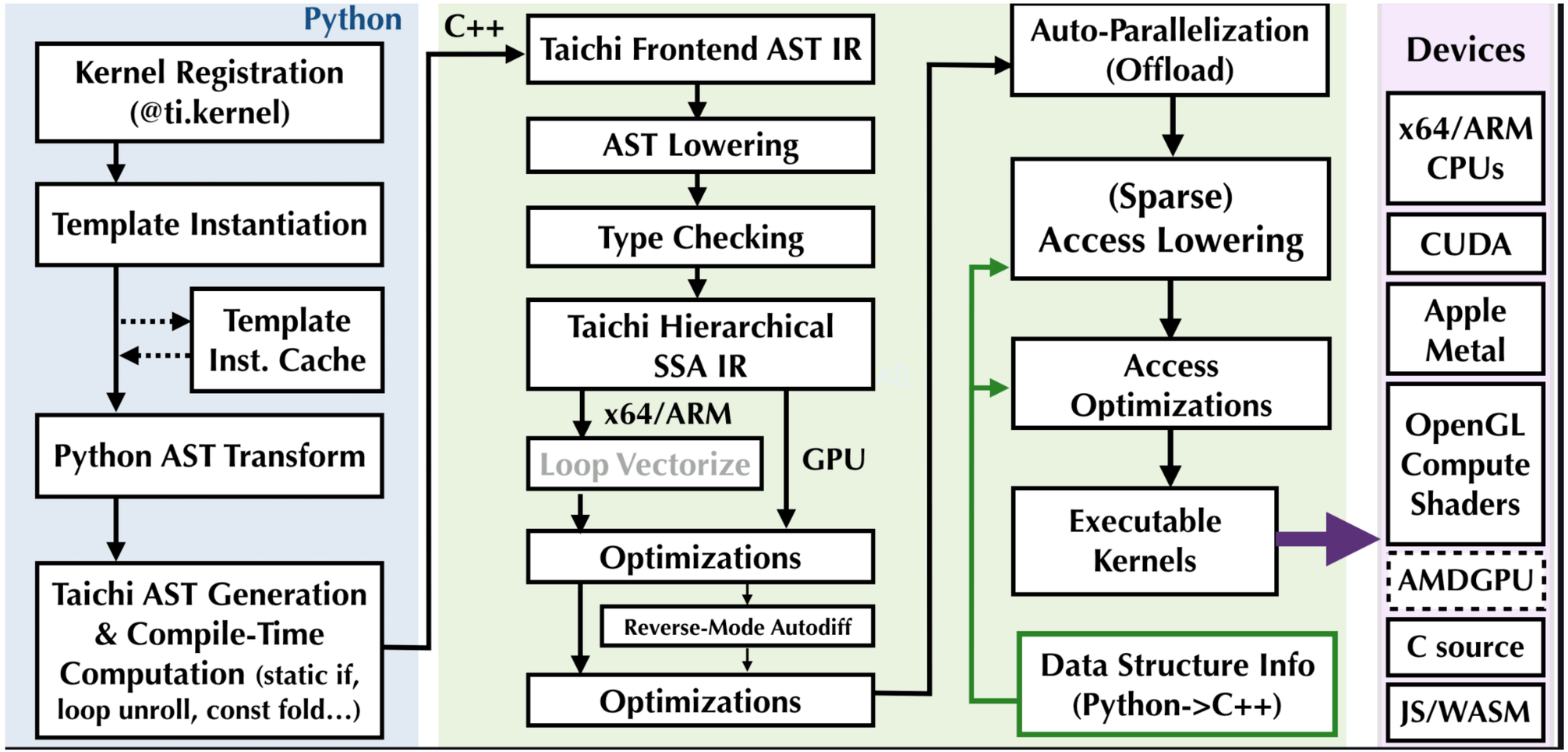
Taichi x MPI



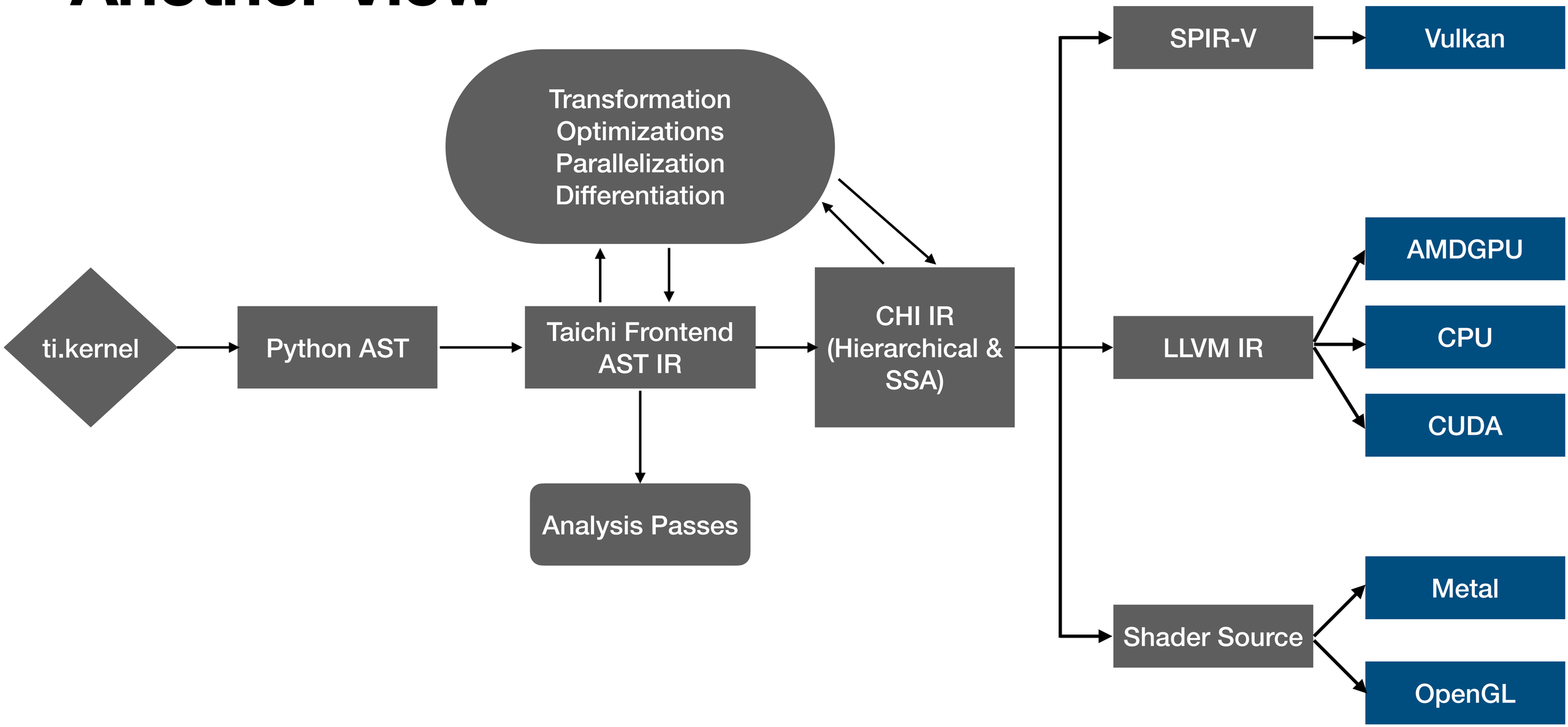
https://github.com/turbo0628/Taichi-MPI.git

Advanced topic

Whole life of a Taichi kernel



Another View



Example CHI IR

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

input_a = ti.ndarray(dtype=ti.f32, shape=(10000))
input_b = ti.ndarray(dtype=ti.f32, shape=(10000))
output_c = ti.ndarray(dtype=ti.f32, shape=(10000))

@ti.kernel
def vecadd(a:ti.types.ndarray(dtype=ti.f32, ndim=1), \
           b:ti.types.ndarray(dtype=ti.f32, ndim=1), \
           c:ti.types.ndarray(dtype=ti.f32, ndim=1)):
    for ii in range(10000):
        a[ii] = ii
        b[ii] = ii
        c[ii] = a[ii] + b[ii] + c[ii]


vecadd(input_a, input_b, output_c)
```

Vecadd

```
[I 01/09/23 02:18:23.755 243926] [compile_to_offloads.cpp:operator()@22] [vecadd
_c76_0] Initial IR:
kernel {
  $0 : for @tmp0 in range((cast_value<i32> 0), (cast_value<i32> 10000)) block_di
m=adaptive {
    $1 = alloca @tmp1
    @tmp1 = @tmp0
    1d_ext_arr (element_dim=0, dt=f32)[@tmp0] = @tmp1
    $4 = alloca @tmp2
    @tmp2 = @tmp0
    1d_ext_arr (element_dim=0, dt=f32)[@tmp0] = @tmp2
    $7 = alloca @tmp3
    @tmp3 = ((1d_ext_arr (element_dim=0, dt=f32)[@tmp0] + 1d_ext_arr (element_di
m=0, dt=f32)[@tmp0]) + 1d_ext_arr (element_dim=0, dt=f32)[@tmp0])
    1d_ext_arr (element_dim=0, dt=f32)[@tmp0] = @tmp3
  }
}
```

Initial IR

Example CHI IR

```
[I 01/09/23 02:18:23.810 243931] [compile_to_offloads.cpp:operator()@22]
_c76_0] DIE:
kernel {
$0 = offloaded range_for(0, 10000) grid_dim=2176 block_dim=128
body {
  <i32> $1 = loop $0 index 0
  <*f32> $2 = arg[0]
  <*f32> $3 = external_ptr $2, [$1] element_dim=0 layout=AOS
  <f32> $4 = cast_value<f32> $1
  $5 : global store [$3 <- $4]
  <*f32> $6 = arg[1]
  <*f32> $7 = external_ptr $6, [$1] element_dim=0 layout=AOS
  $8 : global store [$7 <- $4]
  <f32> $9 = add $4 $4
  <*f32> $10 = arg[2]
  <*f32> $11 = external_ptr $10, [$1] element_dim=0 layout=AOS
  <f32> $12 = global load $11
  <f32> $13 = add $9 $12
  $14 : global store [$11 <- $13]
}
}
```

After DIE Pass CHI IR

```
[I 01/09/23 02:18:23.810 243931] [compile_to_offloads.cpp:operat
or()@22] [vecadd_c76_0] Bit struct stores optimized:
kernel {
$0 = offloaded range_for(0, 10000) grid_dim=2176 block_dim=128
body {
  <i32> $1 = loop $0 index 0
  <*f32> $2 = arg[0]
  <*f32> $3 = external_ptr $2, [$1] element_dim=0 layout=AOS
  <f32> $4 = cast_value<f32> $1
  $5 : global store [$3 <- $4]
  <*f32> $6 = arg[1]
  <*f32> $7 = external_ptr $6, [$1] element_dim=0 layout=AOS
  $8 : global store [$7 <- $4]
  <f32> $9 = add $4 $4
  <*f32> $10 = arg[2]
  <*f32> $11 = external_ptr $10, [$1] element_dim=0 layout=AOS
  <f32> $12 = global load $11
  <f32> $13 = add $9 $12
  $14 : global store [$11 <- $13]
}
}
```

Final CHI IR

Example llvm and nvptx

```python
import taichi as ti
ti.init(arch=ti.cuda, print_kernel_llvm_ir=True,
print_kernel_nvptx=True)
```

```
.maxntid 128, 1, 1
.minnctapersm 2
{
    .reg .pred  %p<3>;
    .reg .b32   %r<10>;
    .reg .f32   %f<5>;
    .reg .b64   %rd<18>;

// %bb.0:                                // %entry
    mov.u32     %r1, %tid.x;
    mov.u32     %r2, %ntid.x;
    mov.u32     %r3, %ctaid.x;
    mad.lo.s32  %r9, %r2, %r3, %r1;
    setp.gt.u32     %p1, %r9, 9999;
    @%p1 bra    $L__BB0_3;
// %bb.1:                                // %while.body.lr.ph.i
    mov.b64     %rd9, vecadd_c76_0_kernel_0_range_for_param_0;
    mov.u64     %rd1, %rd9;
    mov.u32     %r8, %nctaid.x;
    mul.lo.s32  %r5, %r2, %r8;
    ld.param.u64    %rd2, [%rd1+8];
    ld.param.u64    %rd3, [%rd1+16];
    ld.param.u64    %rd4, [%rd1+24];
    mul.wide.u32    %rd10, %r2, %r3;
    cvt.u64.u32     %rd11, %r1;
    add.s64     %rd12, %rd10, %rd11;
    shl.b64     %rd17, %rd12, 2;
    mul.wide.u32    %rd13, %r2, %r8;
    shl.b64     %rd6, %rd13, 2;
$L__BB0_2:                                // %while.body.i
                                          // =>This Inner Loop Header: Depth=1
    add.s64     %rd14, %rd2, %rd17;
```

```
define internal void @function_body(%struct.RuntimeContext* %0, i8* %1, i32 %2) {
allocs:
  %3 = alloca i32, align 4
  br label %entry

final:                                            ; preds = %function_body
  ret void

entry:                                            ; preds = %allocs
  br label %function_body

function_body:                                    ; preds = %entry
  store i32 %2, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = call i64 @RuntimeContext_get_args(%struct.RuntimeContext* %0, i32 0)
  %6 = inttoptr i64 %5 to float*
  %7 = call i32 @RuntimeContext_get_extra_args(%struct.RuntimeContext* %0, i32 0, i32 0)
  %8 = mul i32 0, %7
  %9 = add i32 %8, %4
  %10 = getelementptr float, float* %6, i32 %9
  store float 1.000000e+00, float* %10, align 4
  %11 = call i64 @RuntimeContext_get_args(%struct.RuntimeContext* %0, i32 1)
  %12 = inttoptr i64 %11 to float*
  %13 = call i32 @RuntimeContext_get_extra_args(%struct.RuntimeContext* %0, i32 1, i32 0)
  %14 = mul i32 0, %13
  %15 = add i32 %14, %4
  %16 = getelementptr float, float* %12, i32 %15
  store float 1.000000e+00, float* %16, align 4
  %17 = call i64 @RuntimeContext_get_args(%struct.RuntimeContext* %0, i32 2)
  %18 = inttoptr i64 %17 to float*
  %19 = call i32 @RuntimeContext_get_extra_args(%struct.RuntimeContext* %0, i32 2, i32 0)
  %20 = mul i32 0, %19
  %21 = add i32 %20, %4
  %22 = getelementptr float, float* %18, i32 %21
  %23 = load float, float* %22, align 4
  %24 = fadd float %23, 3.000000e+00
  store float %24, float* %22, align 4
  br label %final
}
```

教练，我还想学……

- https://docs.taichi-lang.org/
- https://forum.taichi-lang.cn/
- https://zhuanlan.zhihu.com/p/552884861
- https://zhuanlan.zhihu.com/p/556658798
- https://github.com/taichi-dev/taichi
- P.S. 如果大家在使用过程中发现(性能)问题，欢迎大家去forum或者github直接提问～