CS**5643**

06 Intro to Taichi

Steve Marschner
Cornell University
Spring 2023

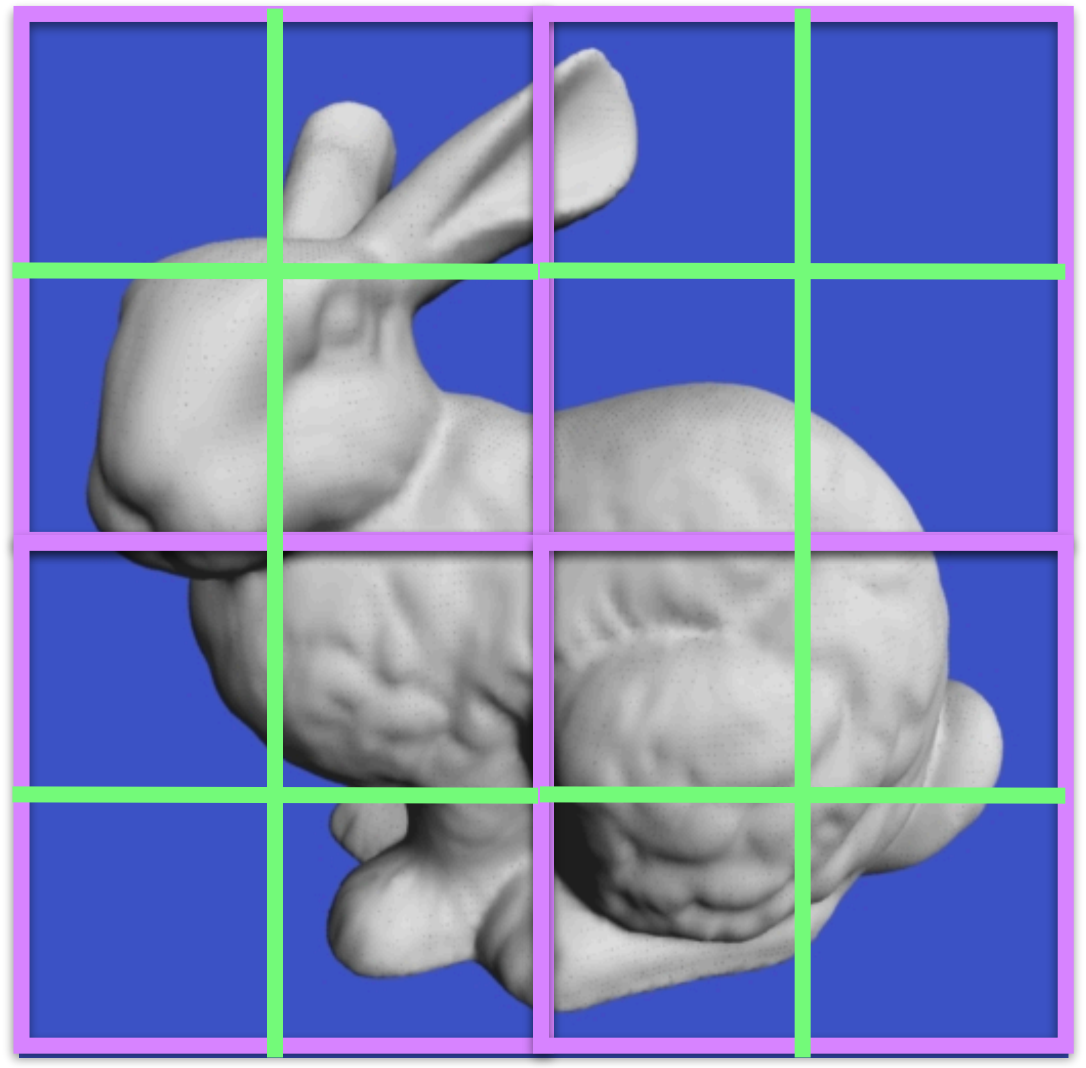
Taichi

A domain-specific language for parallel computation on sparse spatial data

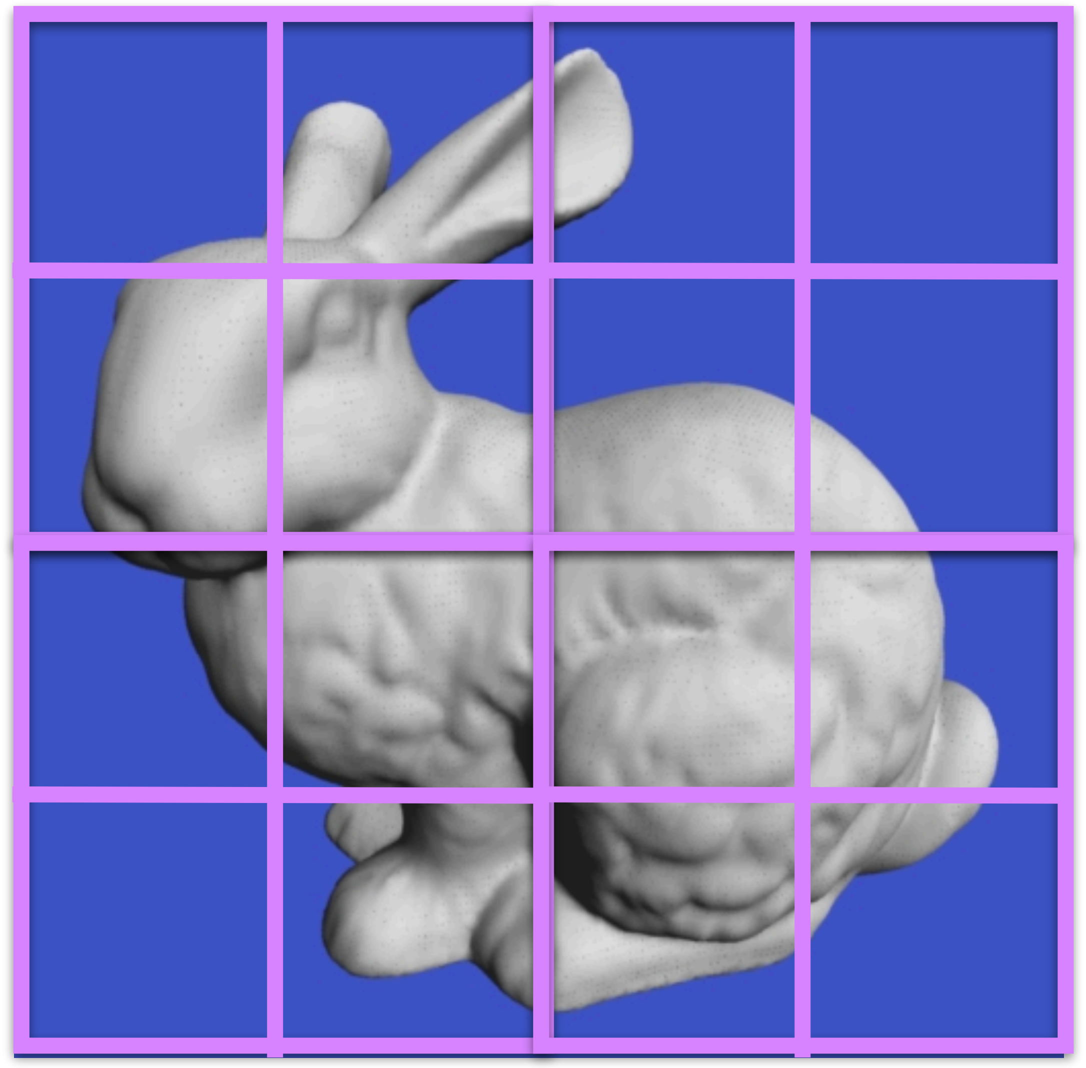
- motivation: decouple the data structures from the computation

Data Structure ⟷ Decoupled ⟷ Computation Code

Hierarchical Tree of Sparse Grids



DS1



DS2

First-level Grid Divider

2nd-level Grid Divider

```
@ti.func
def gnoise(p : vec2):
    # the four corners of the integer square where p falls
    i00 = tm.floor(p)
    i10 = i00 + vec2(1,0)
    i01 = i00 + vec2(0,1)
    i11 = i00 + vec2(1,1)
    # the values of the four pseudorandom gradients, evaluated at p
    v00 = (p - i00).dot(randunit(i00))
    v01 = (p - i01).dot(randunit(i01))
    v10 = (p - i10).dot(randunit(i10))
    v11 = (p - i11).dot(randunit(i11))
    # the two blending factors (f.x and f.y) we will use to interpolate
    a = p - i00
    f = 3*a*a - 2*a*a*a
    # bilinear interpolation between the four gradient values
    return (
        (v00 * (1-f[0]) + v10 * f[0]) * (1 - f[1]) +
        (v01 * (1-f[0]) + v11 * f[0]) * f[1]
        )
```

Perlin Noise Code

Taichi

Taichi Lang

Origins

- dissertation work of Yuanming Hu at MIT, introduced at SIGGRAPH in 2019–2021
- now maintained as an open source project by Yuanming at his spinoff company Taichi Graphics

What it provides

- a domain-specific language (DSL) suitable for simulation on the GPU
- a flexible set of data structures for dense and sparse grids
- an automatic differentiation system

What we will use

- we rely on the Taichi language as our way to express fast computations
- we will mainly use dense-grid data structures and will likely not use autodiff
- for your final projects you might like to explore the fancier features!

Some important issues for performance

To go fast:

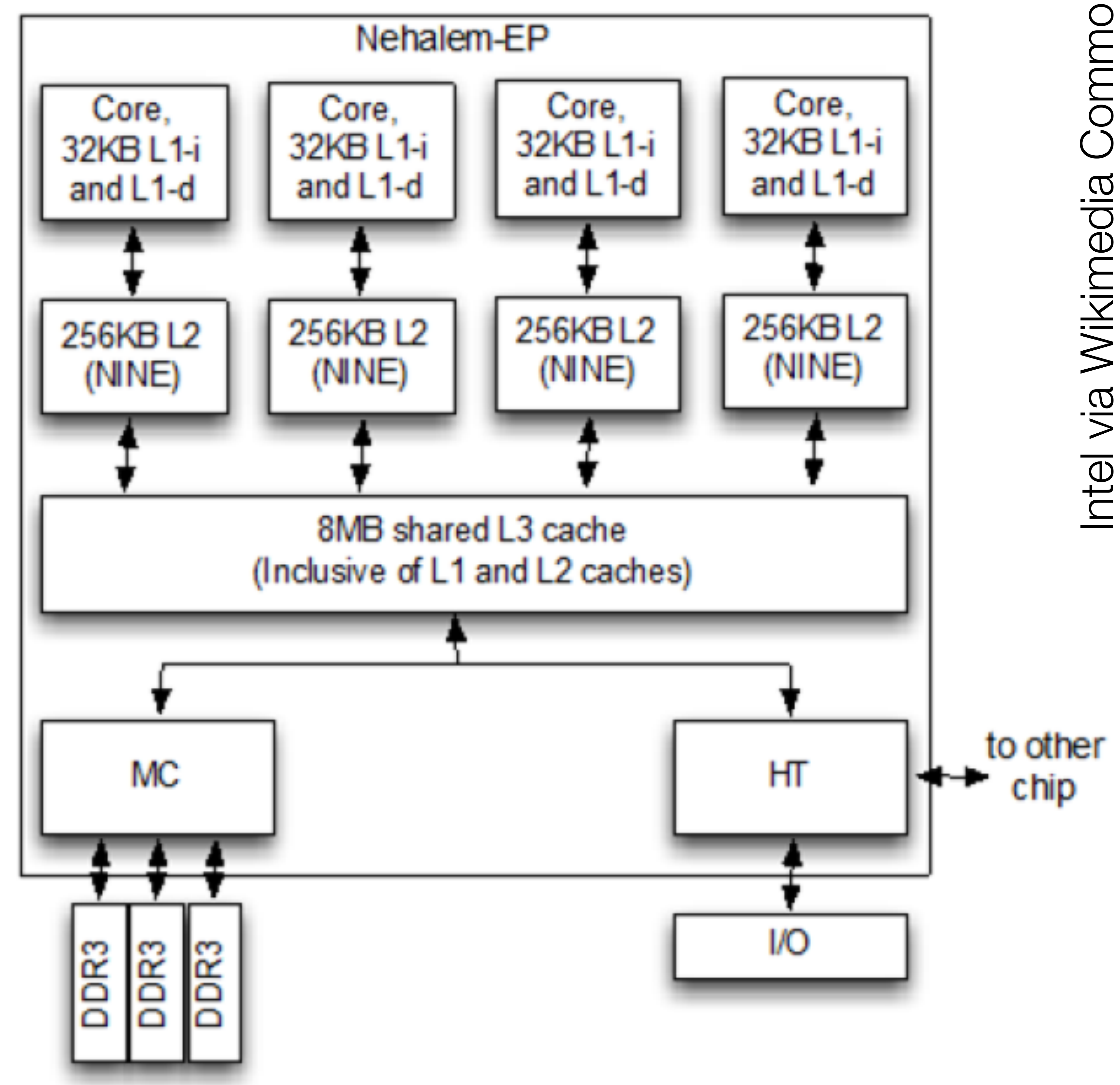
- focus performance effort on a few bottlenecks
- do work in parallel
- minimize time spent waiting for data

Compute in many independent tasks

- need lots of tasks to make good use of GPUs
- one task's behavior should not depend on another's result
- *streaming* computation: many tasks each consuming a separate input and writing a separate output

Organize data in memory to maximize *locality*

- data accessed close together in time should be located close together in space
- increases effectiveness of memory hierarchy
- bottom line: store data close together and access it in order



a typical memory hierarchy
(newer examples have bigger numbers)

Intel via Wikimedia Commons

Oraganization of a Taichi program

Python code

- runs serially on CPU via Python interpreter
- keep things that don’t need to be fast in here because it is easier!

Taichi kernels

- compiled to optimized parallel code for CPU or GPU
- can be broken into Taichi functions for modularity/reuse
- cannot access data in regular Python variables directly

Taichi data containers

- are stored in memory that is fast for kernels to access
- provide control over how data is organized in memory
- data often must be copied between CPU and GPU memory to interoperate with Python

Initialization: ti.init()

Call it before you create your first field or call your first kernel

- OK to *define* functions and kernels before initialization

At initialization time you select a *backend*

- ti.init(arch=**ti.cpu**) and Taichi kernels run on your CPU
- ti.init(arch=**ti.gpu**) and Taichi chooses a default GPU backend
- can specify GPU API specifically with architectures **cuda**, metal, opengl, **vulkan**
- **note:** on Mac, metal is the default GPU option but vulkan is often the better/newer choice
- fancier features are only

You can also set some other useful parameters

- ti.init(arch=ti.cpu, cpu_max_num_threads=1) ensures serial execution for nicer debug output
- ti.init(arch=ti.cpu, debug=True) will enable bounds checking on all array accesses

Taichi datatypes

Taichi has the usual data types and GLSL-like vector/matrix types

- to define by example: `ti.i32` (signed 32-bit int), `ti.f64` (double-precision floating point), `ti.u16` (unsigned 16-bit int)
  - can use python types `int` and `float` as aliases for *default* integer and floating-point types (defaults set at initialization)
- vector types generated like this
  - `ti.types.vector(4, ti.f64)` — a 64-bit floating-point 4D vector type
  - `ti.types.matrix(4, 3, int)` — a 4x3 integer matrix type
- swizzling for 2,3,4 dimension vectors works like in GLSL (v.x or v.r is v[0], etc.)
- there are also structure types (we have not used them yet)

Types are Python objects so you can store them in variables to make aliases

- `vec2 = ti.types.vector(2, ti.f32)`

Taichi data containers

To store data where you can access it from Taichi code, put it in containers

Most common: fields

- a field is an ND array of scalars, vectors, or matrices
- `g = ti.field(ti.u8, (480,640))` — an 8-bit grayscale image
- `c = ti.Vector.field(3, ti.u8, (480,640))` — an RGB color image
- `f = ti.field(ti.f32, ())` — a 0D floating point field, aka. a single scalar
- dimensions are fixed at creation time

You can access data in fields from Python code

- `g[20,30] = 4`
- `c[30,20] = [3,4,5]; c[30,20][1] = 4;` *not* `c[30,20,1] = 4`
- `c.fill(4), c.to_numpy(), c.from_numpy(ar)` — where `ar.shape` is `(30,20,3)`

Taichi kernels

A kernel is a piece of Taichi code that can be called from Python

- syntax is Python, code is parsed by Python interpreter
- semantics are a bit different; code is compiled by Taichi compiler
- various restrictions exist that don't exist in Python

Kernels are written by decorating Python functions

- Taichi code is statically typed
- argument and return types must be provided
- max of one return statement allowed
- global Python variables are accessible but are read at compile time and become constants

```python
@ti.kernel
def square(x : ti.f32) -> ti.f32:
    return x*x
```

```python
square(42)
```

```
1764.0
```

Taichi functions

A Taichi function is a piece of Taichi code that can be called from Taichi

- kernels can call functions; functions can call functions
- functions cannot call kernels; functions cannot be called from Python
- functions are always inlined (therefore no recursion)
- functions don’t require type hints when types can be inferred

```
@ti.func
def sqr(x):
    return x*x
@ti.kernel
def fourth(x : ti.f32) -> ti.f32:
    return sqr(sqr(x))
```

```
fourth(4)
```

```
256.0
```

Getting data into Taichi

Constants

- you can just read them from Python globals
- their values are fixed at the time that compilation happens

```
@ti.kernel
def power(x : ti.f32) -> ti.f32:
    return tm.pow(x, p)
```

```
p = 4
print(power(3))
p = 2
print(power(3))
```

```
81.0
81.0
```

Kernel parameters and return values

- pass them to and from python kernels when you call them
- their values differ across invocations

Fields

- fields are global data that can be read or written by Taichi code or Python code
- be aware that accessing individual elements from Python is slow
- fields are compile time constants in Taichi but their values are not

```
@ti.kernel
def power(x : ti.f32) -> ti.f32:
    return tm.pow(x, p[None])
```

```
p = ti.field(ti.f32, ())
p[None] = 3
print(power(3))
p[None] = 2
print(power(3))
```

```
27.0
9.0
```

Loops in Taichi

Typical uses: range for or structure for

- looping over a field gives you multiple indices
- looping with ti.grouped() gives you a multi-index

Loops over constant lists

- the function ti.static() asks for an unrolled loop over a list of constant data

Loops in kernels at outermost scope are automatically parallelized

- this is where much of the performance comes from
- can be defeated for range loops with

  ti.loop_config(serialize=True)

```
@ti.kernel
def loopy():
    for i in range(3):
        print("a", i)
    for i in field1:
        print("b", i)
    for i,j in field2:
        print("c", i, j)
    for k in ti.grouped(field2):
        print("d", k)
```

```
field1 = ti.field(int, 3)
field2 = ti.field(int, (3,2))
```

```
loopy()
```

```
a 0
a 1
a 2
```

```
@ti.kernel
def loopme():
    for v in ti.static(ar):
        f[None] = f[None] + v[0] * v[1]
```

```
ar = [[1,2],[2,1],[3,2]]
f = ti.field(ti.i32, ())
loopme()
print(f[None])
```

```
10
```

```
c 2 1        d [2, 1]
c 2 2        d [2, 2]
c 2 3        d [2, 3]
```

Beware data races

If you forget your code is parallel you can get wrong answers

- on GPU architectures, for speed, concurrent accesses to the same memory location do not happen in any reliable order
- concurrent read-modify-write operations are unsafe by default
- architecture provides *atomic add* and other atomic operations that ensure concurrent accesses behave as if serialized in some order
- Taichi uses atomic operations for += and friends

```
ti.init(arch=ti.gpu)
@ti.kernel
def prefix_sum():

    sum1 = 0
    sum2 = 0
    for i in f:
        sum1 = sum1 + f[i]
        sum2 += f[i]
    print(sum1, sum2)

    sum1 = 0
    sum2 = 0
    ti.loop_config(serialize=True)
    for i in range(f.shape[0]):
        sum1 = sum1 + f[i]
        sum2 += f[i]
    print(sum1, sum2)
```

```
[Taichi] Starting on arch=metal
```

```
f = ti.field(ti.i32, 128)
f.from_numpy(np.arange(128, dtype=np.int32))
prefix_sum()
```

```
32 8128
8128 8128
```

Reference

A Hands-on Tutorial of The Taichi Programming Language @ Siggraph 2020

- https://yuanming.taichi.graphics/publication/2020-taichi-tutorial/taichi-tutorial.pdf

Taichi Paper:

- https://dl.acm.org/doi/pdf/10.1145/3355089.3356506

Taichi intro documentation:

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

Taichi detailed API docs:

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