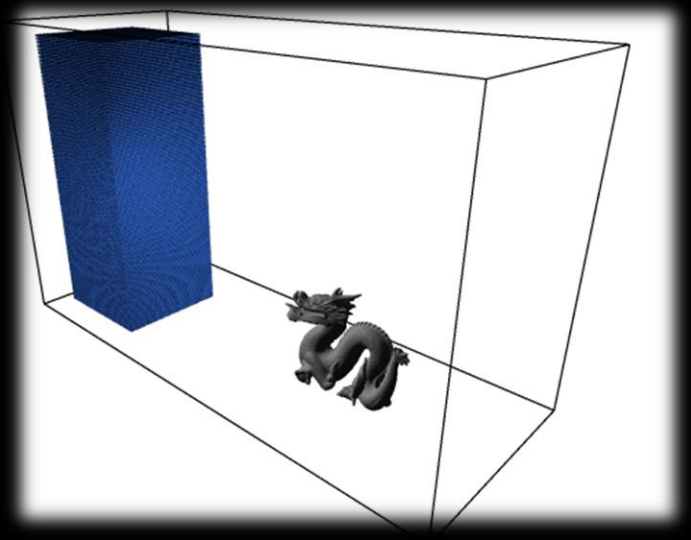
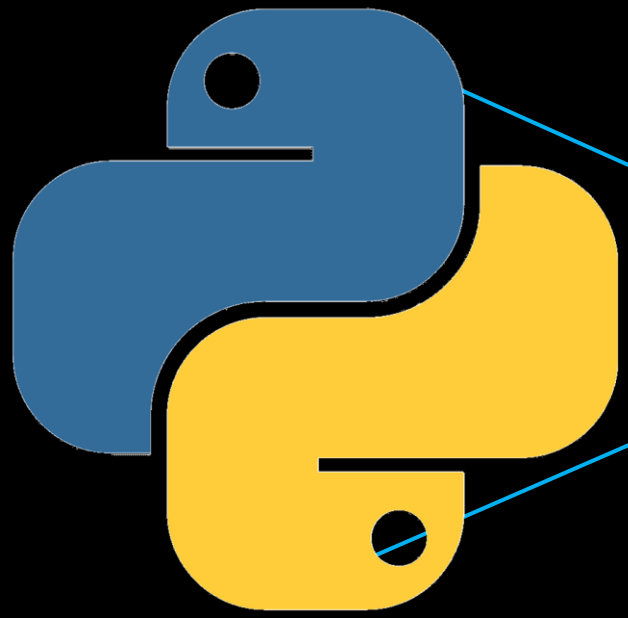
Taichi Lang



JIT Python pour calcul haute performance

B. Gailleton – 22/03/2025 Café Calcul



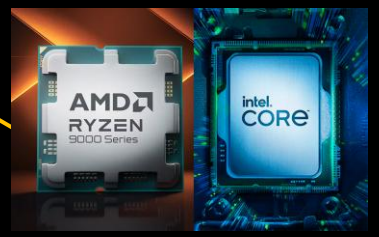
Python

Taichi Lang

HPC

NVIDIA CUDA

Vulkan



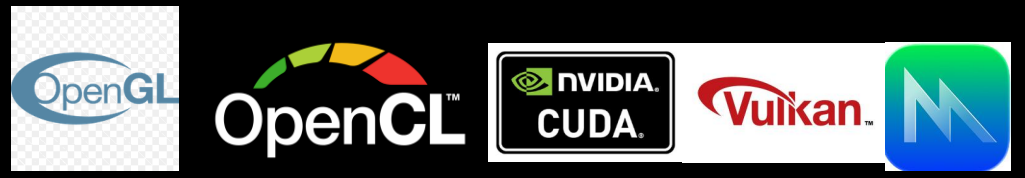
Pourquoi Taichi Lang ?

Programmer pour calcul GPU

Low-level

Pros: control fin, performances

Cons: Temps de développement, niveau d'entrée haut



Cœur en C/C++/assimilé

Taichi Lang

Pros: Bon ratio performance/temps de développement

High-level

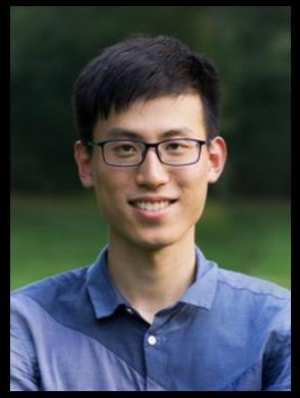
Cons: Pas de flexibilité, peu de controle mémoire, pas de noyaux

Très orienté IA
Calcul vectorisable/broadcasting
Python/julia/...

Pourquoi Taichi Lang ?

Taichi Lang

Compilateur JIT python pour calcul massivement parallèle



Yuanming Hu

Ethan 胡渊鸣 博士

Ph.D., MIT EECS

Founder & CEO, Meshy AI

Phase de développement 2018-2023
Depuis 2023 phase de maintiens
Soutenu par grosse entreprise d'IA (meshy AI)

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

Points vendeurs principaux

Custom kernels (e.g. comme CUDA)

Advanced data structure (matrices creuses, structures)

Intégré dans l'écosystème python

```
$ pip install taichi
```

Un seul code pour toutes les backends

NVIDIA CUDA

Vulkan

AMD RYZEN 9000 Series

intel CORE

Quickstart

Installation: `pip install taichi`

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

1) Importer taichi

2) Initialiser la backend

```python
# Simulation Parameters
N = 512
dx = 0.01          # Explicit spatial step
alpha = 1.0        # Thermal diffusivity
dt = (dx**2) / (4 * alpha) * 0.9  # Stability limit (CFL condition)

x = ti.field(dtype=ti.f32, shape=(N, N))
new_x = ti.field(dtype=ti.f32, shape=(N, N))
x.from_numpy(initial_T) # initial_T est une array numpy 2D
new_x.fill(0.)
```

3) Initialiser les données: paramètre/constants et arrays de donnée

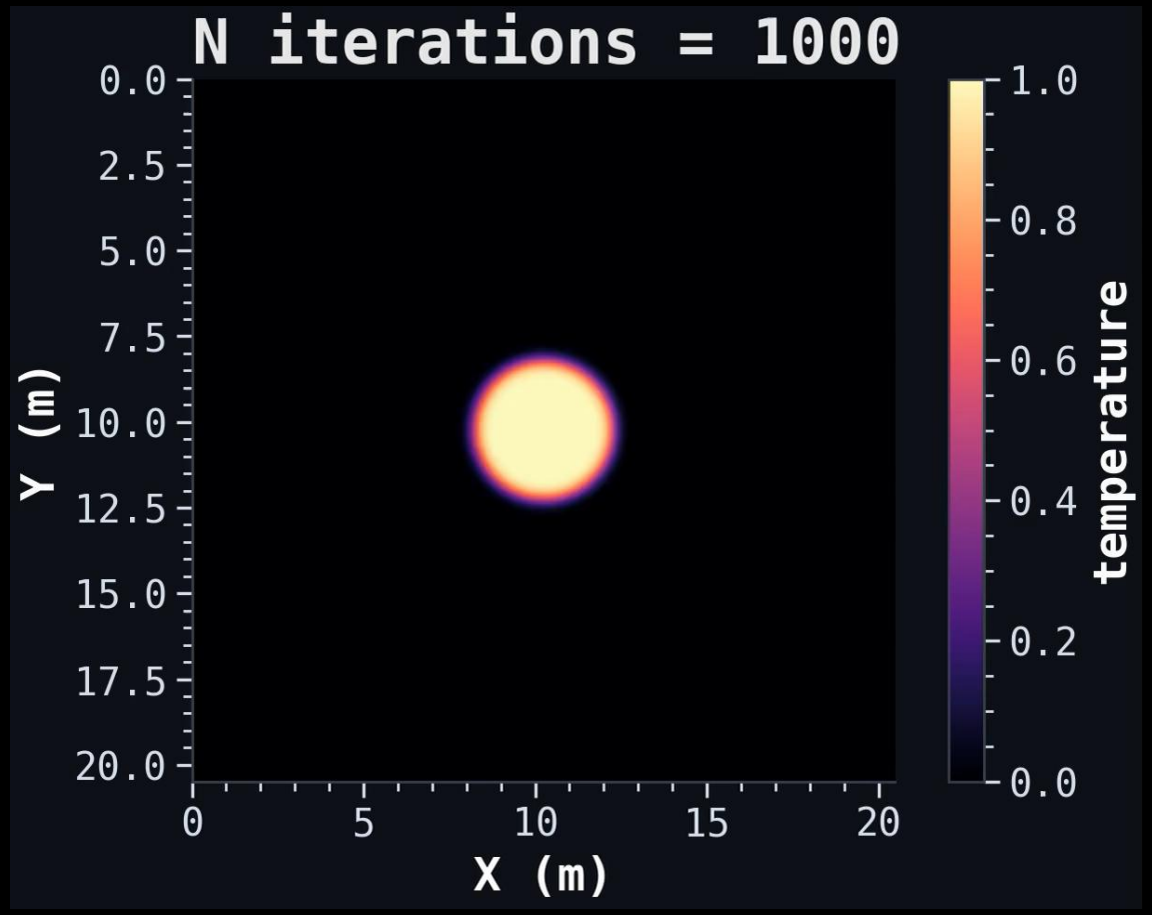
```python
@ti.kernel
def iterate():
  for i, j in x:
    # Boundary conditions
    if i == 0 or i == N-1 or j == 0 or j == N-1:
      new_x[i, j] = 0.0
    else:
      # Finite difference Laplacian
      laplacian = (x[i-1, j] + x[i+1, j] + x[i, j-1] + x[i, j+1] - 4.0 * x[i, j]) / (dx**2)
      new_x[i, j] = x[i, j] + alpha * dt * laplacian
```

4) écriture du (des) noyau(x) - kernel de calcul en python "restreint"

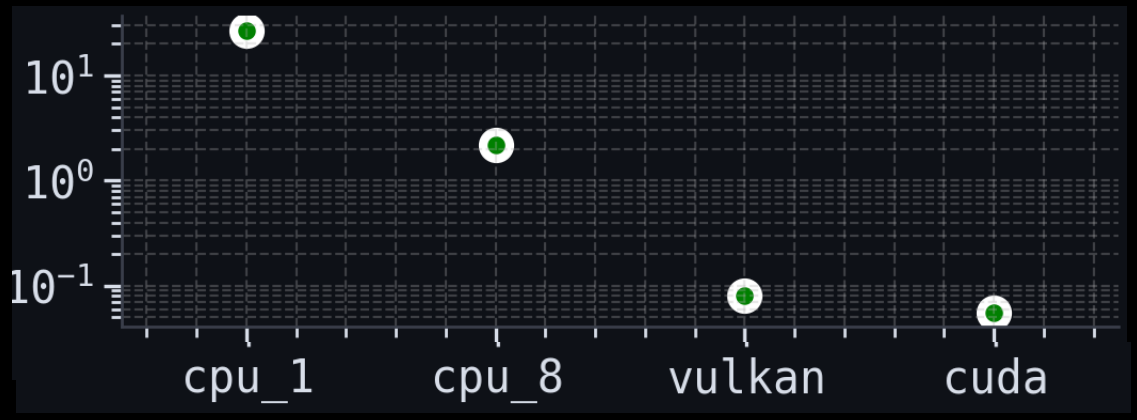
```python
while True:
  iterate()
  x.copy_from(new_x)
```

5) Execution flow en python classic

Quickstart: la Benchmark rapide



Temps pour 1000 itérations (s)



CPU Backend *GPU*

Temps d'exécution x 10 - 100

ti.init(...) : choix de la backend

Option par défault (ti.gpu, ti.cpu, None)

Option précise (ti.vulkan, ti.cuda, ti.metal, ti.opengl, ...)

Backend	i8	i16	i32	i64	u1	u8	u16	u32	u64	f16	f32	f64
CPU	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CUDA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OpenGL	✗	✗	✓	○	✓	✗	✗	✗	✗	✗	✓	✓
Metal	✓	✓	✓	✗	✓	✓	✓	✓	✗	✗	✓	✗
Vulkan	○	○	✓	○	✓	○	○	✓	○	✓	✓	○

○: Requiring extensions for the backend.

Backend "full": cpu et Cuda
Backend "OK for most": vulkan
Backend "OK for the basics": Metal, opengl

I: integer – u: unsigned integer – f: floating point – XX nombre de bits

Les fonctions et noyaux I : fonctionnement

```python
@ti.func
def check_boundary(i:ti.i32, j:ti.i32, N:ti.i32):
  check = True
  if i == 0 or i == N-1 or j == 0 or j == N-1:
    check = False
  return check

@ti.kernel
def iterate(x:ti:template(), new_x:ti.template(),
            N:ti.i32, dx:ti.f32, dt:ti.f32, alpha:ti.f32):
  for i, j in x:
    # Boundary conditions
    if check_boundary(i,j,N):
      new_x[i, j] = 0.0
    else:
      # Finite difference Laplacian
      laplacian = (x[i-1, j] + x[i+1, j] + x[i, j-1] + x[i, j+1] - 4.0 * x[i, j]) / (dx**2)
      new_x[i, j] = x[i, j] + alpha * dt * laplacian
```

@ti.kernel : noyau de calcul que l'on appelle depuis python

@ti.func : fonction automatiquement *inline* que l'on appelle dans les noyaux

Arguments: les arguments doivent être typés

- Scalaire: ti.i8/32/64/...
- Générique: ti.template()

Boucle(s) externe(s) parallèlisé:

for i, ... in field:

Boucle "manuelle"

```python
@ti.kernel
def loop_2d():
    for i, j in ti.ndrange(2, 5):
        print(i, j)
```

Boucle génériques

```python
@ti.kernel
def test(arr: ti.types.ndarray()):
    for I in ti.grouped(arr):
        arr[I] += 2
```

Interopérabilité avec ti.ndarray

PyTorch NumPy

```python
@ti.kernel
def foo(A: ti.types.ndarray(dtype=ti.f32, ndim=2)):
    do_something()
```

Les fonctions et noyaux II : librairie et fonctions disponible

ti.math :

- toutes les fonctions classiques (sin, cos, … min, max, …, norm, grad, …)
- https://docs.taichi-lang.org/api/master/taichi/math/
- ```
  mat2 = ti.math.mat2
  vec3 = ti.math.mat3
  vec4 = ti.math.vec4 ...
  ```

ti.atomic_…:

- ti.atomic_add(x, y)
- ti.atomic_sub(x, y)
- ti.atomic_and(x, y)
- ti.atomic_or(x, y)
- ti.atomic_xor(x, y)
- ti.atomic_max(x, y)
- ti.atomic_min(x, y)

- Structure: return la vielle valeur, modify x in place
- Limitation : pas de atomic_CAS
- ⚠️ +=, &=, … sont considéré atomiques

Pour les backends CPU et CUDA:

- Solver de matrice creuses/système linéaires
- *https://docs.taichi-lang.org/docs/master/linear_solver*
  ```

La donnée I : les scalaires

⚠️ Les GPUs sont (de `manière générale) *memory-bound* et non *compute-bound*

```python
# Simulation Parameters
N = 512
dx = 0.01          # Explicit spatial step
alpha = 1.0        # Thermal diffusivity
dt = (dx**2) / (4 * alpha) * 0.9  # Stability limit (CFL condition)
```

Constantes ou scalaire cotés *host*:

- Scalaire basiques (int, float, bool, string)
- ⚠️ **Si global : la valeure est fixée au temps de compilation**

Scalaires coté *device* :

- scal = ti.field(ti.f32,shape = ())
- Modification (kernel ou host): scal[None] = 42.

La donnée II : les fields à N dimensions

```
x = ti.field(dtype=ti.f32, shape=(N, N))
new_x = ti.field(dtype=ti.f32, shape=(N, N))
x.from_numpy(initial_T) # initial_T est une array numpy 2D
new_x.fill(0.)
```

Field classiques (numpy-like):

- Array à N dimensions (jusqu'à 8)
- Remplissage unique fill, copy_from (à partir d'un autre field), from_numpy ou kernel

Field vecteurs ou matriciels :

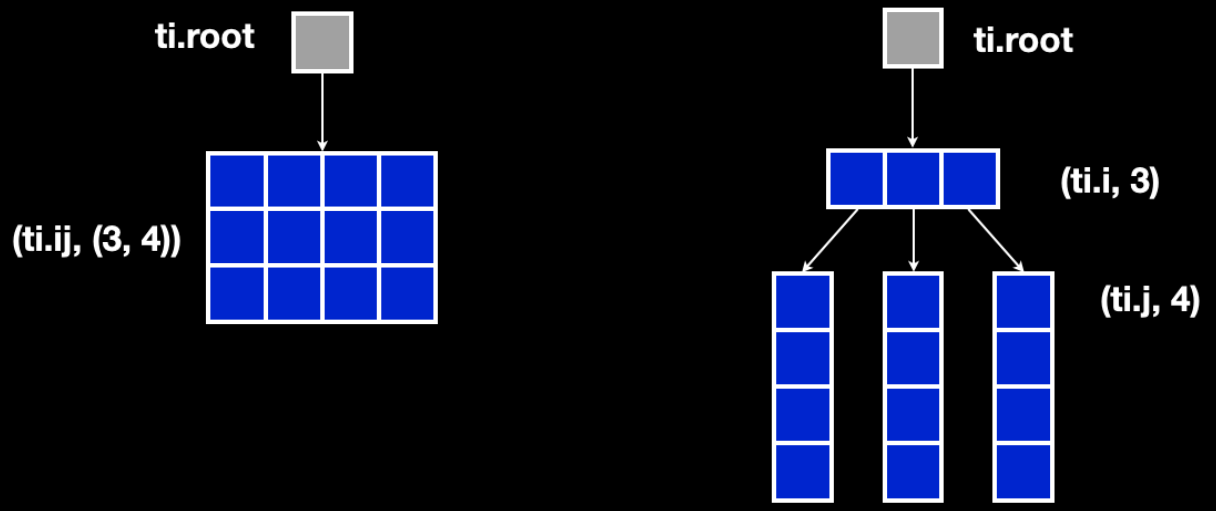
- Même principe
- Mais type de donnée lui-même multiple:
  - ti.Vector.field(n=2, dtype=float, shape=(N, N))
  - ti.Matrix.field(n=2, m=3, dtype=ti.f32, shape=(N, N))
- Dans un kernel: vec[i,j][k]

Struct field :

- particle_field = ti.Struct.field({
  "pos": ti.math.vec3,
  "vel": ti.math.vec3,
  "diffusion": ti.f32,
  "label": ti.u8,
  "active": ti.u1, # bool
  }, shape=(N,N))
- Dans un kernel, plusieurs appels :
  - particules[i,j].pos
  - particules.pos[i,j]
  - particules.pos.from_numpy(my_positions)

La donnée III : Layout avancés

```
ti.root.dense(ti.ij, (3, 4)).place(x) ti.root.dense(ti.i, 3).dense(ti.j, 4).place(x)
```



Placer les ressources manuellement :

a) Déclaration de variables par types

x = ti.field(ti.i64)

b) Placement or type d'indexage:

- ti.i, ti.j, ti.k, ...
- ti.ij, ti.ijk, ti.ikl, ...

c) Chaine de déclaration puis placement:

- ti.root.dense(...).place(x)

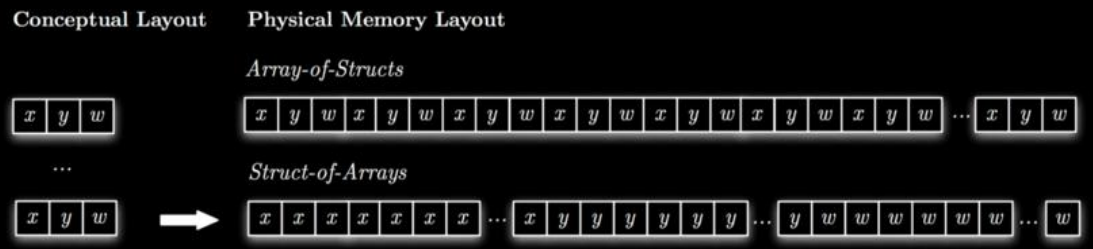
exemple 1 : row major vs column major

```python
x = ti.field(ti.f32)
y = ti.field(ti.f32)
ti.root.dense(ti.i, M).dense(ti.j, N).place(x)   # row-major
ti.root.dense(ti.j, N).dense(ti.i, M).place(y)   # column-major
```

exemple 2 : Strucure of Arrays (SoA) vs Arrays of Structures (AoS)

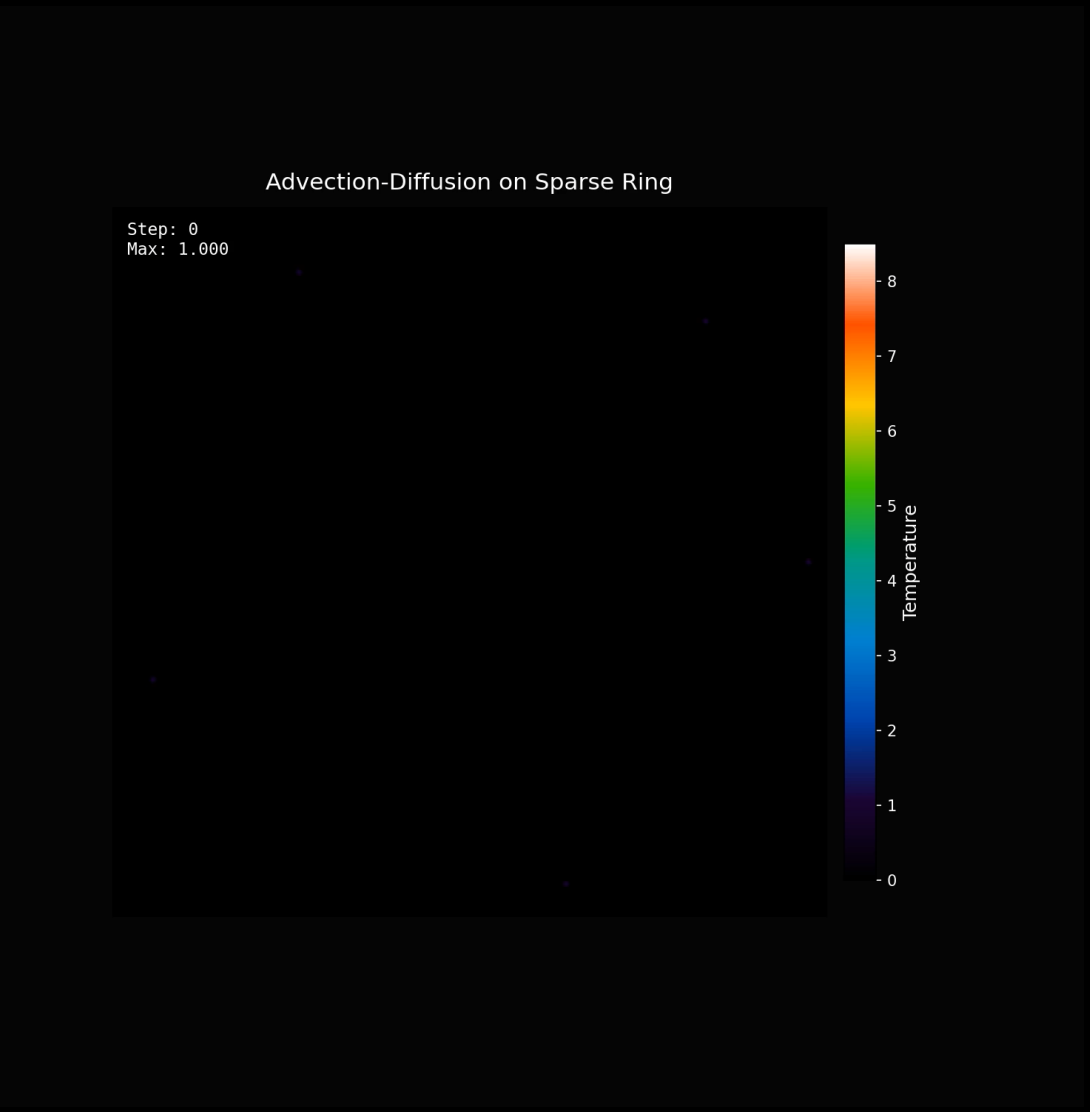
```python
# --- SETUP SOA ---
pos_soa = ti.field(ti.f32)
vel_soa = ti.field(ti.f32)
ti.root.dense(ti.i, N).place(pos_soa)
ti.root.dense(ti.i, N).place(vel_soa)

# --- SETUP AOS ---
pos_aos = ti.field(ti.f32)
vel_aos = ti.field(ti.f32)
ti.root.dense(ti.i, N).place(pos_aos, vel_aos)
```

Temps d'éxécution x 1.5-3



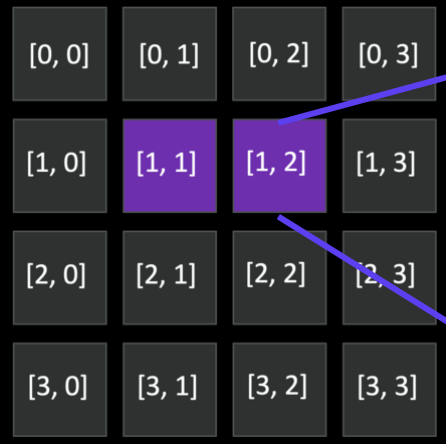
La donnée IV : données creuses et éparses

Au delà de ti.root.dense -> block coalescent




```
# ================================================
# DENSE LAYOUT
# ================================================
dense_val = ti.field(dtype=ti.f32, shape=(N, N))
dense_val_new = ti.field(dtype=ti.f32, shape=(N, N))

# ================================================
# SPARSE LAYOUT (pointer -> bitmasked)
# ================================================
sparse_val = ti.field(dtype=ti.f32)
sparse_val_new = ti.field(dtype=ti.f32)

block = ti.root.pointer(ti.ij, (N // BLOCK_SIZE, N // BLOCK_SIZE))
pixel = block.bitmasked(ti.ij, (BLOCK_SIZE, BLOCK_SIZE))
pixel.place(sparse_val, sparse_val_new)
```

.pointer (0 noeud actif = pas de données en mémoire)



.bitmask (noeud actif = iteration, else: skip)

Temps d'éxécution x 3-20

La donnée V : Extra

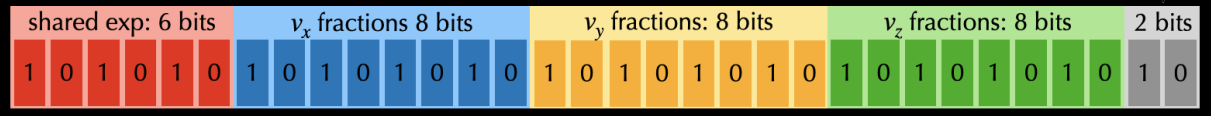
1) Possibilité d'activer et désactiver "manuellement" les noeuds des ti.root.pointer ou des ti.root.bitmask
pour un controle maximal. ⚠️ cela demande un travail rigoureux (0 automatisation) ⚠️ Seulement avec ti.cuda et ti.cpu

2) Champs dynamic (comme une liste python: append possible) avec ti.root.dynamic(ti.i, size_max, size_chunk)

3) Dataclass et orienté objet ⚠️ 5-30% overhead

4) Quantized data type ⚠️ Seulement avec ti.cuda et ti.cpu
https://docs.taichi-lang.org/docs/master/quant


```
u5 = ti.types.quant.int(bits=5, signed=False)
fixed_type_a = ti.types.quant.fixed(bits=10,
max_value=20.0)
```



Temps d'éxécution
x 1.5-8

Benchmarking and profiling

Classique via les outils python : ⚠️ à deux détails

```python
def benchmark(fn, name, warmup=WARMUP, runs=RUNS):
    for _ in range(warmup):
        fn()
    ti.sync()
    start = time.perf_counter()
    for _ in range(runs):
        fn()
    ti.sync()
    elapsed = time.perf_counter() - start
    return (elapsed / runs) * 1000
```

Warmup = 1er lancement compile la fonction

ti.sync() = force l'attente de la fin des kernel
(non automatique tant que les données ne sont pas demandé cotés python)

Via les outils taichi :

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

```python
ti.profiler.print_kernel_profiler_info()
```

```
================================================================================
Kernel Profiler(count, default) @ CUDA on NVIDIA RTX 3500 Ada Generation Laptop GPU
================================================================================
[      %     total   count |      min       avg       max   ] Kernel name
--------------------------------------------------------------------------------
[ 34.85%   0.023 s    300x |    0.046     0.076     0.239 ms] k_fma_c86_0_kernel_0_range_for
[ 34.25%   0.023 s    300x |    0.062     0.075     0.243 ms] k_add_c82_0_kernel_0_range_for
[ 14.78%   0.010 s    300x |    0.031     0.032     0.191 ms] k_saxpy_c84_0_kernel_0_range_for
[ 10.41%   0.007 s    200x |    0.034     0.034     0.041 ms] k_stencil_c88_0_kernel_0_range_for
[  3.60%   0.002 s    100x |    0.023     0.024     0.029 ms] k_reduce_sum_c90_0_kernel_1_range_for
[  1.06%   0.001 s    200x |    0.003     0.003     0.008 ms] k_stencil_c88_0_kernel_1_serial
[  0.54%   0.000 s    100x |    0.003     0.004     0.011 ms] k_reduce_sum_c90_0_kernel_2_serial
[  0.52%   0.000 s    100x |    0.003     0.003     0.013 ms] k_reduce_sum_c90_0_kernel_0_serial
--------------------------------------------------------------------------------
[100.00%] Total execution time:   0.066 s   number of results: 8
================================================================================
```

Pour info: Backend cuda compatible avec CUPTI

Control flow à partir de python : classique, mais pièges à éviter

Minimiser les copy CPUs :

```python
for i in range(N_steps):
    mon_kernel1(a,b,c)
    mon_kernel2(a,b,c)
    data = a.to_numpy()
```

Lent et couteux

```python
for i in range(N_steps):
    mon_kernel1(a,b,c)
    mon_kernel2(a,b,c)

    # if i % 100 == 0:
    #    ...

data = a.to_numpy()
# ou ti.sync()
```

OK

Redéfinition != deallocation

OK

```python
a = ti.field(ti.f32, shape = (512,256))
```

Memory leak

```python
for i in range(N_steps):
    a = ti.field(ti.f32, shape = (512,256))
```

Favoriser les constantes lorsque possible

```python
@ti.kernel
def run_1(a:ti.template(), K:ti.f32, dt:ti.f32):
    for i in a:
        a[i] = dt/K * a[i]


# ... K et dt constant
@ti.kernel
def run_1(a:ti.template()):
    for i in a:
        a[i] = dt/K * a[i]
```

Taichi pour un logiciel/libraire de recherche complet : quelques conseils/limites

https://github.com/TopoToolbox/pyfastflow

Compile-time évaluation avec ti.static():

```
@ti.kernel
def run_1(a:ti.template()):
    for i in a:
        if check_boundary(i) == False:
            continue
        a[i] = dt/K * a[i]

def check_boundary(i:ti.i32):
    res = False
    if ti.static(GLOBAL.BOUNDARY_MODE == 0):
        res = normal_boundary(i)
    elif ti.static(GLOBAL.BOUNDARY_MODE == 1):
        res = periodic_boundary(i)
    elif ti.static(GLOBAL.BOUNDARY_MODE == 2):
        res = custom_boundary(i)
    return res

# ... after setting globals :
check_boundary = ti.func(check_boundary)
```

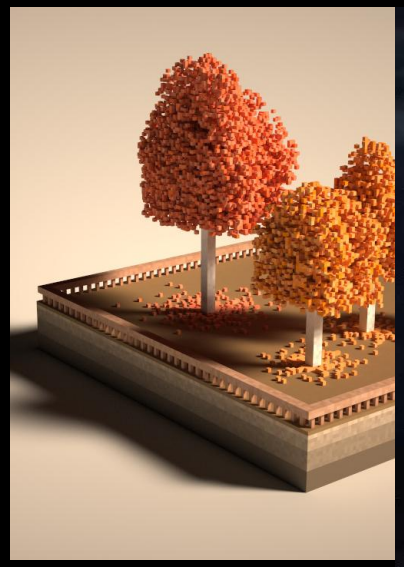
Gestion de la mémoire manuelle possible:

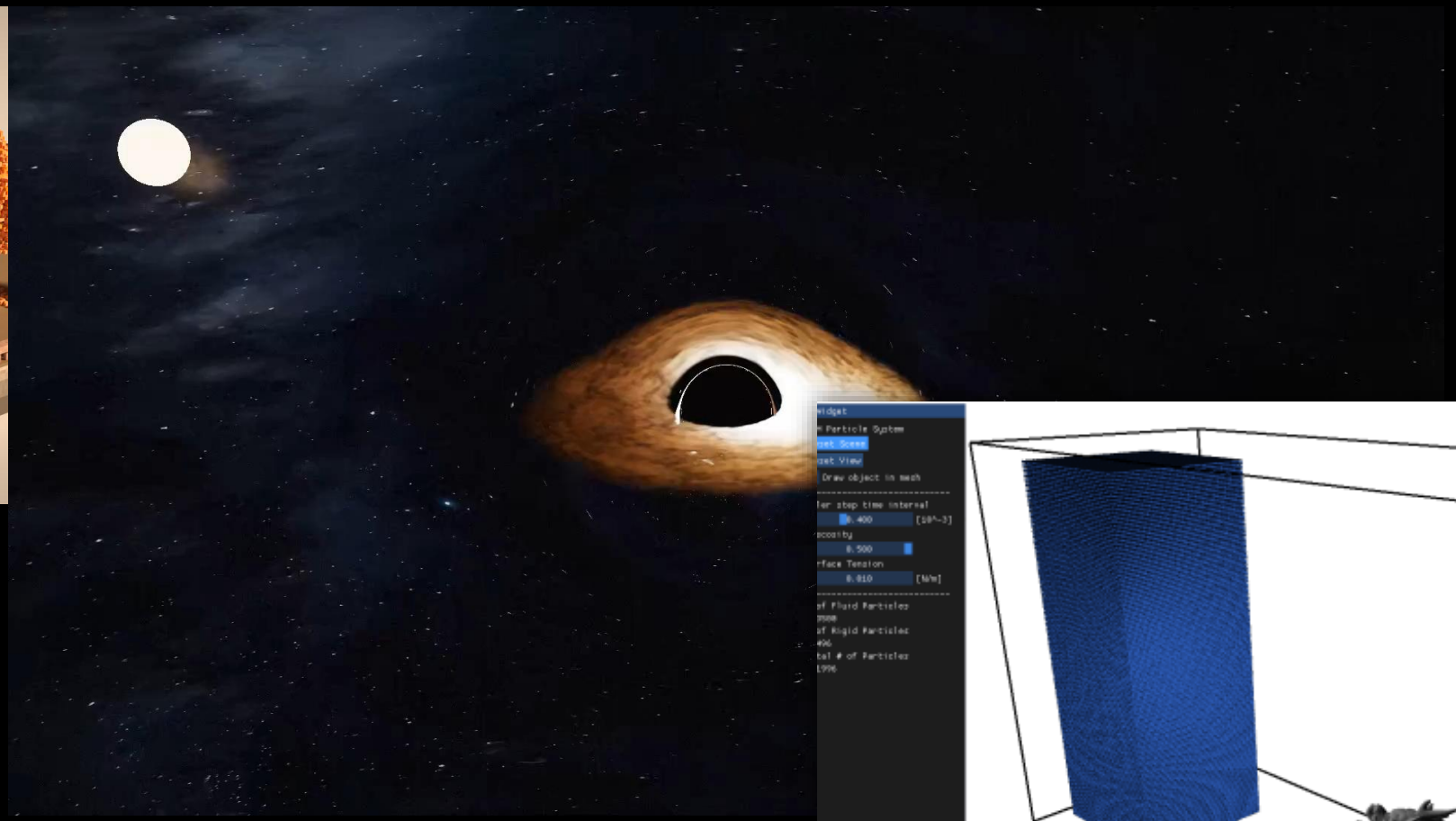
```
fb1 = ti.FieldsBuilder()
x = ti.field(dtype=ti.f32)
fb1.dense(ti.ij, (5, 5)).place(x)
fb1_snode_tree = fb1.finalize()
# ... simulation ...
fb1_snode_tree.destroy()  # Destruction
```

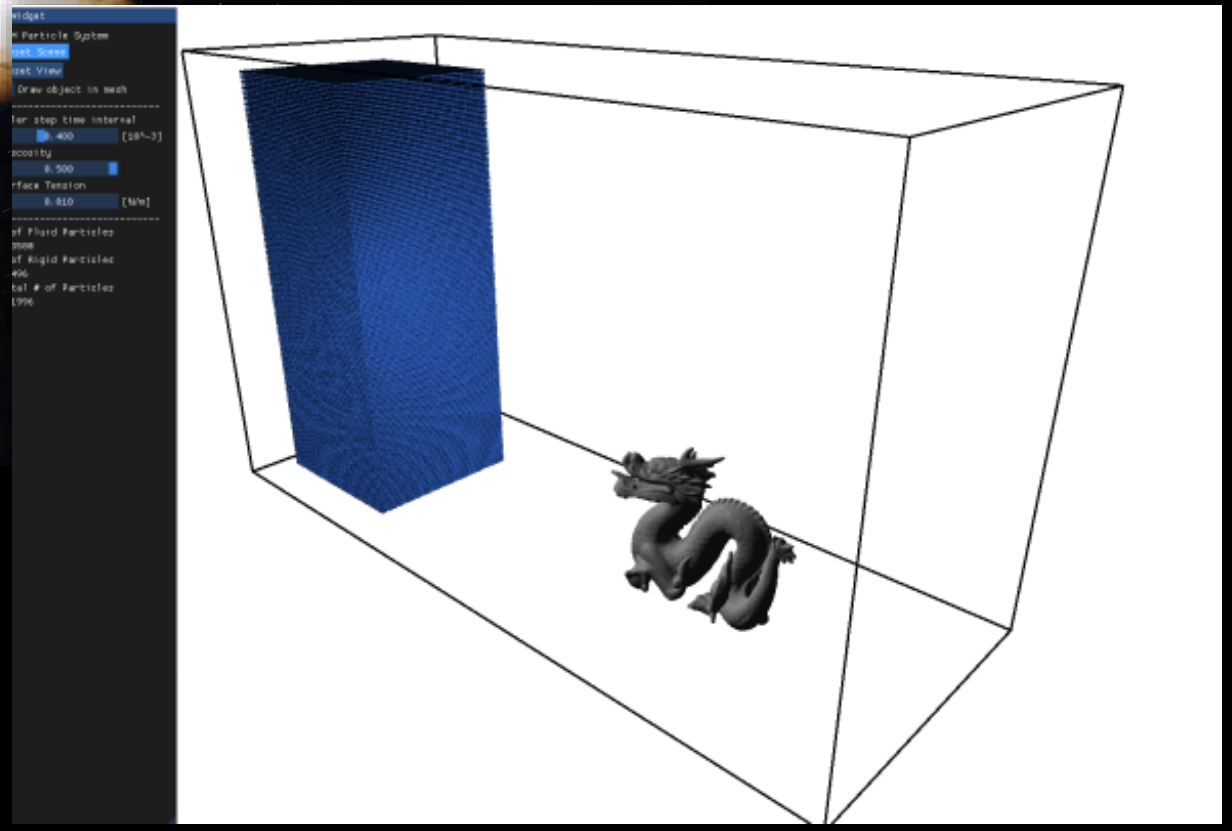
Attention à l'overhead du JIT sur des gros projets

- Vite amorti pour des taches > 5s
- Couteux pour les taches "uniques" et courtes
- Reset total buggy (e.g. en cas de constantes)

Taichi pour la visualisation graphique/scientifique







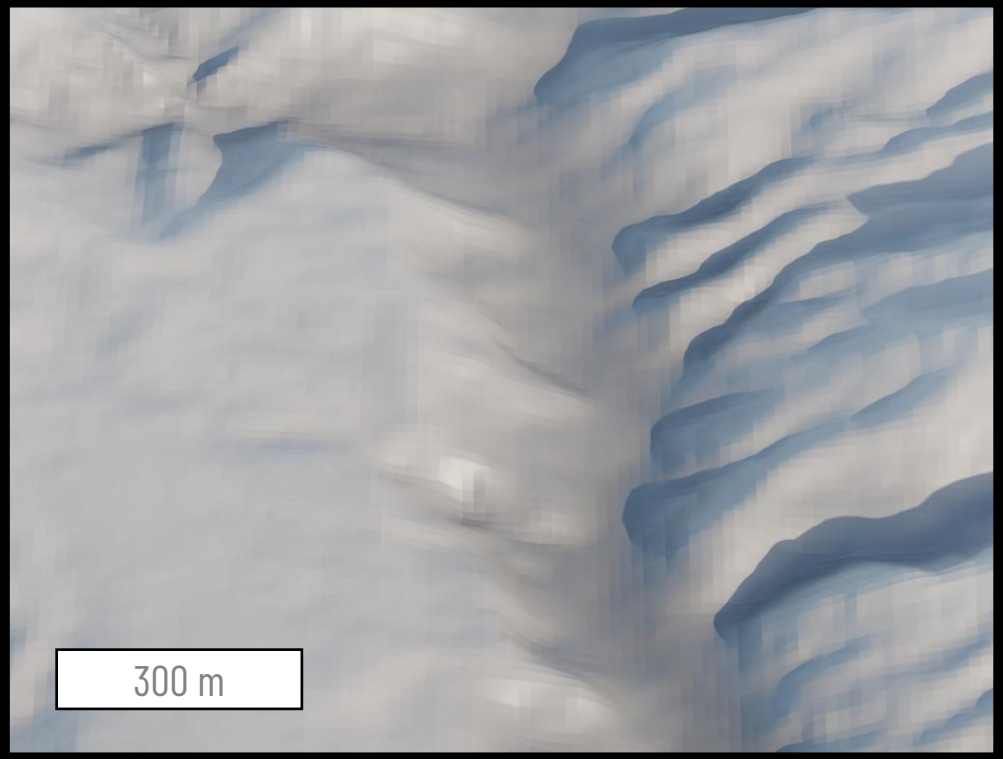
Visualisation très performante

- Rendu full GPU
- Interface via imgui (légér)
- Steep learning curve

Contexte en géomorphologie

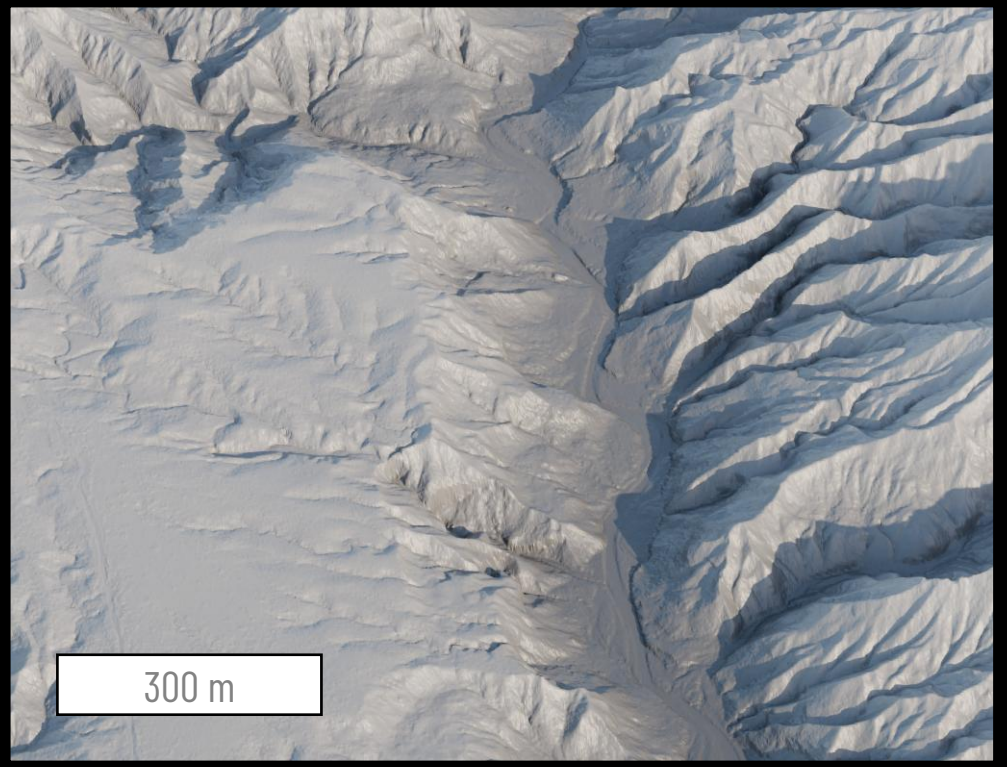
B. Gailleton
Géomorphologue
Postdoc à Géosciences Rennes
bgailleton.github.io

SRTM (30 m)



Volume de données x1800

LiDAR-DTM (1m)

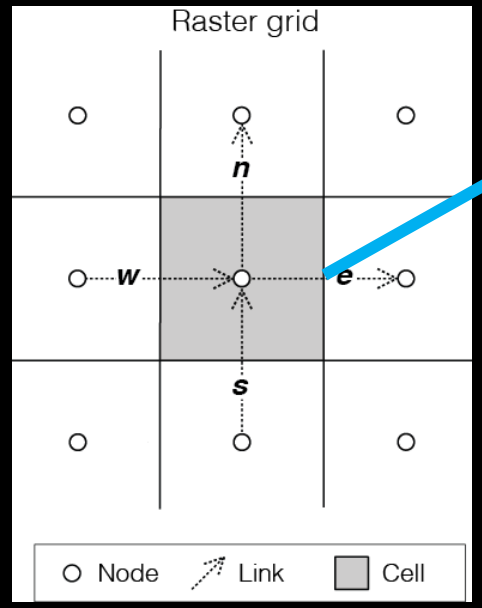


MNT = matrice 2D d'altitude à partir desquels nous quantifions des processus climatiques/tectoniques

Application en géosciences I : simuler les écoulements d'eau

Défis numérique en géomorphologie quantitative :

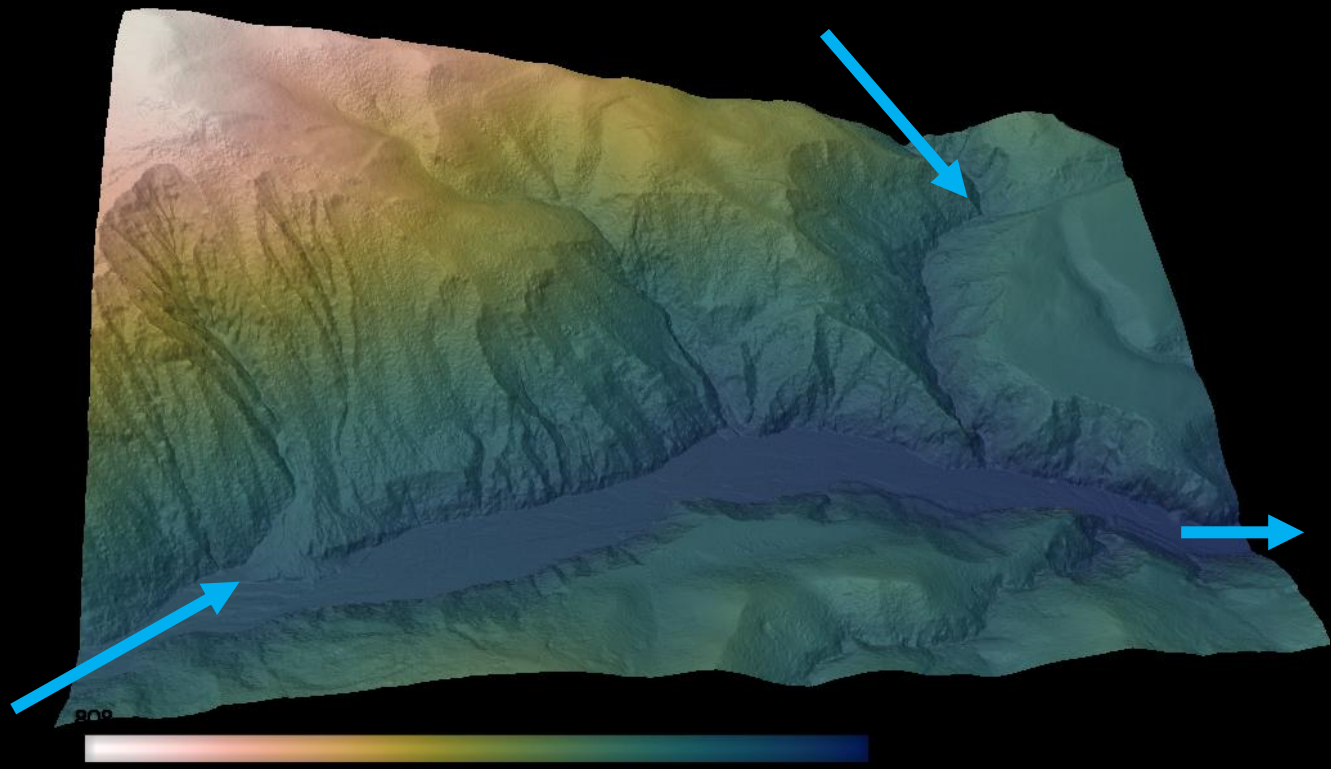
- Simuler des écoulements d'eau sur des grandes échelles spatio-temporelle



$$q_{t+\Delta t} = \frac{q_t - g h_t \Delta t \frac{\partial (h_t + z)}{\partial x}}{\left(1 + g h_t \Delta t n^2 q_t / h_t^{10/3}\right)}$$

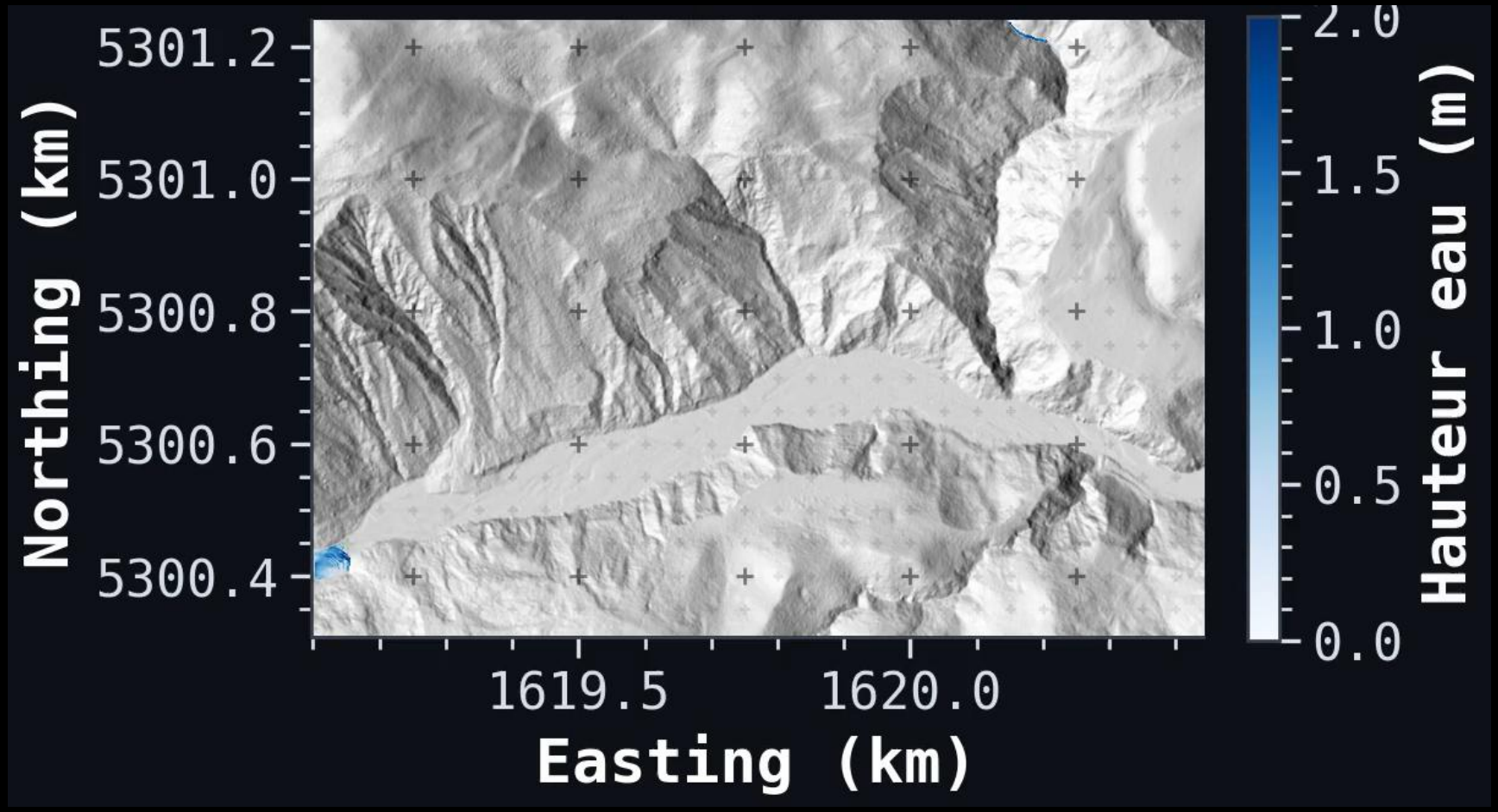
f(pente, hauteur d'eau)

Incrément d'eau dans le temps
Temps CPU : ~30-50 minutes

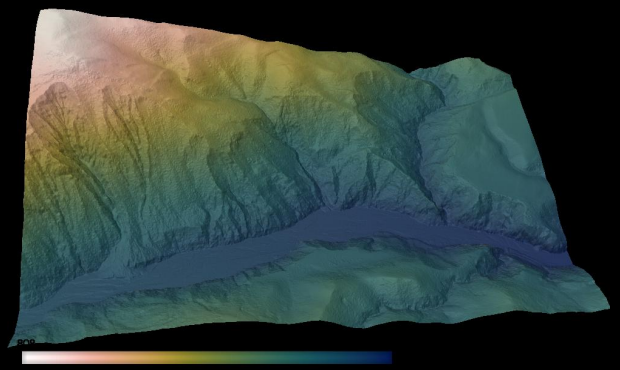


Problème parfaitement parallélisable

Application en géosciences : portage naïf

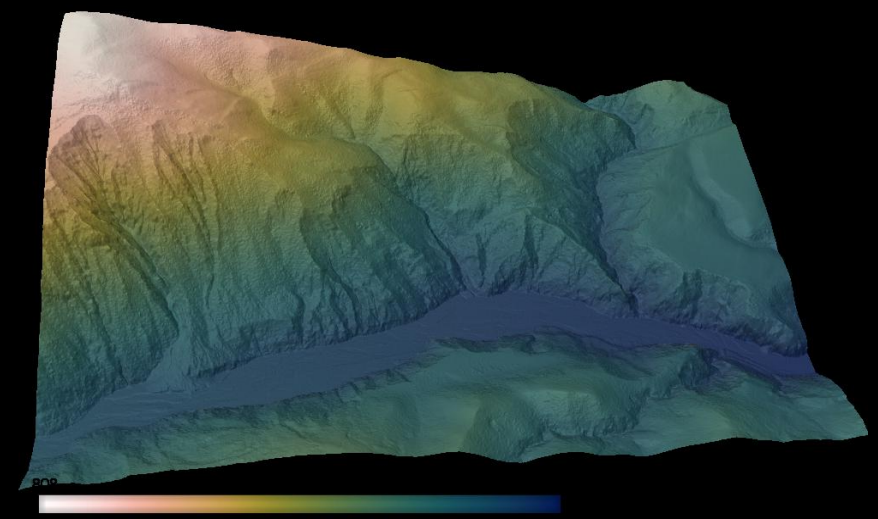


1 frame = 2000 itérations



Accélération vs CPU x 40

Application en géosciences : résolution stationnaire

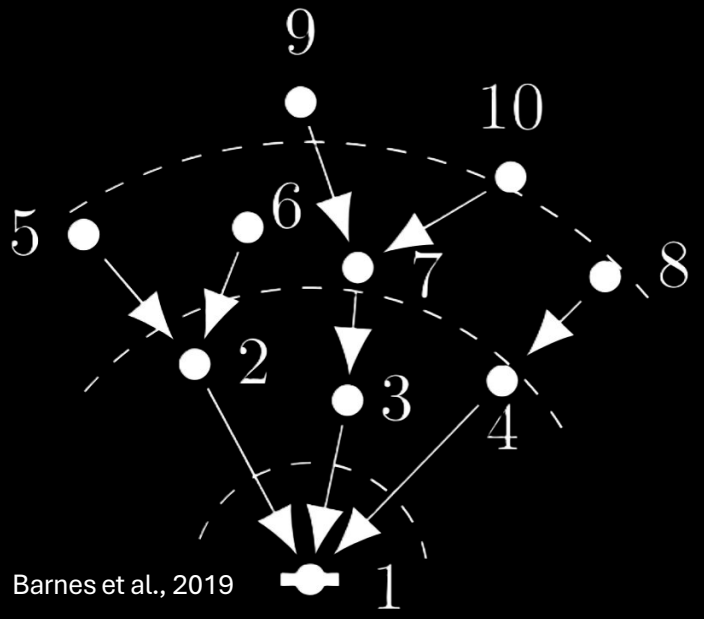


Accélération vs stationnaire
x4
Et surtout linéarisation

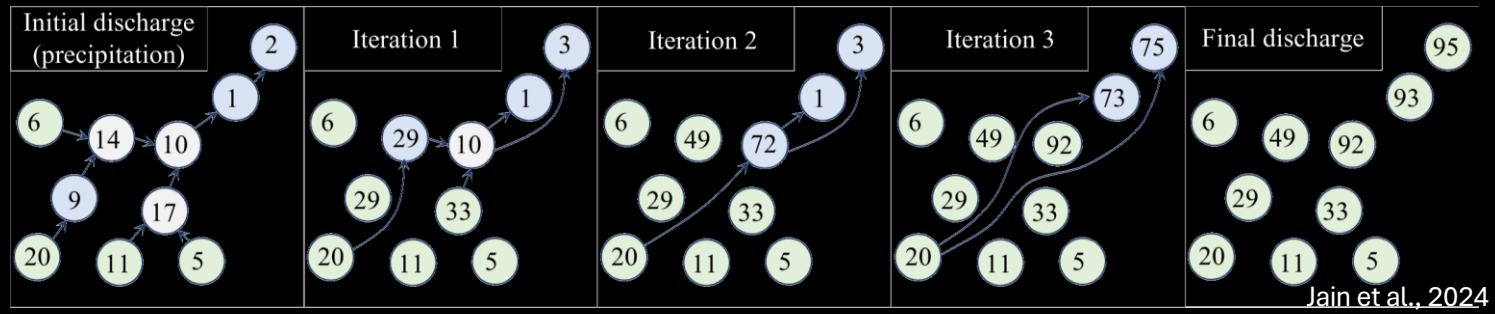
1 frame = 10 itérations

Application en géosciences : accumulation non local

Somme de préfix sur un graphe acyclique = beaucoup de *race conditions*



Barnes et al., 2019



Jain et al., 2024

Accumulation par niveaux

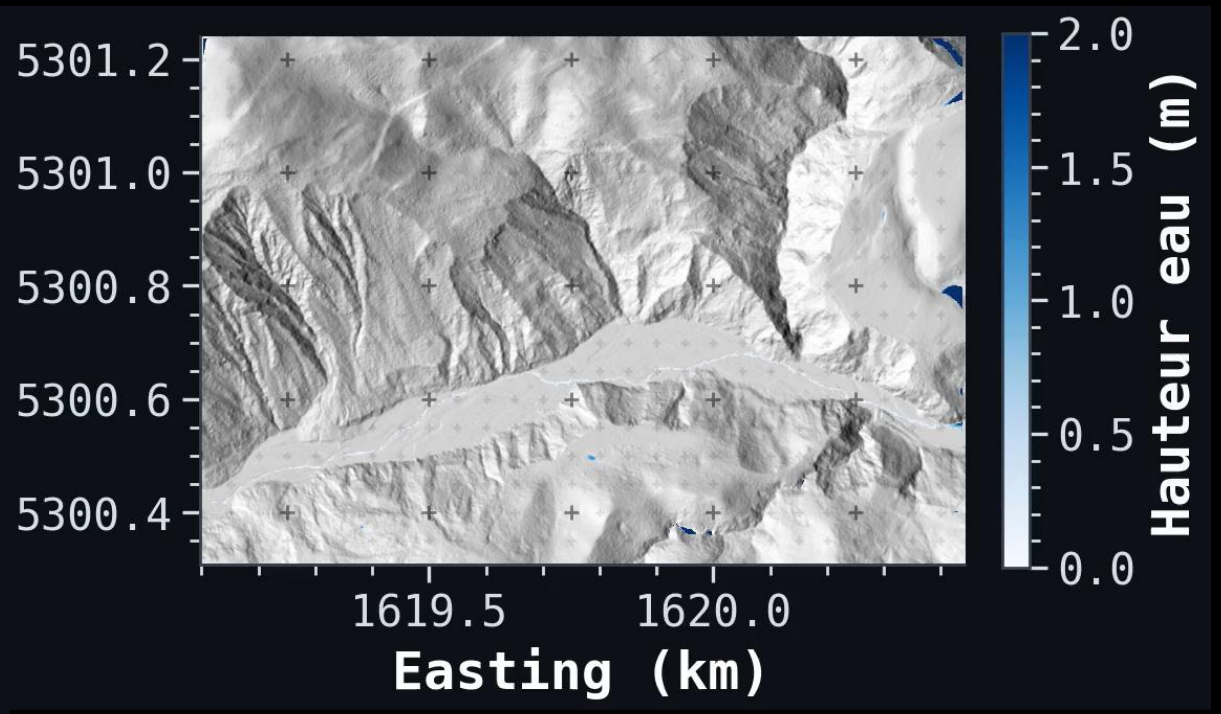
Taichi plus lent que CPU
Cuda C++ 8-15 fois plus rapide que CPU (kernel persistent)

Accumulation par saut de pointeur

Taichi 5-12 fois plus rapide que CPU

Take home message

B. Gailleton - *bgailleton.github.io - boris.gailleton@univ-rennes.fr - Café Calcul 22/01/2026*



Site principal
https://www.taichi-lang.org/

Documentation (très complète)
https://docs.taichi-lang.org/docs/hello_world

Un exemple de logiciel (un peu de pub)
https://github.com/TopoToolbox/pyfastflow

Taichi Lang

JIT pour python massivement parallele

Custom Kernels

- flexibilité de Cuda
- écrit en python

Data structure

- automatique ou avancé
- Données creuses/sparse gérées nativement

Full stack

- visualisation
- simulation
- portable