4th COMCHA School, Zaragoza, 2026

Optimized programming

Jiahui Zhuo (IFIC)

Who am I?

- **Jiahui Zhuo**, **IFIC** (Instituto de Física Corpuscular), **Valencia**.
- **LHCb** experiment.
- Working on the **LHCb** High Level Trigger (**HLT**) system.
- **HLT** processes **30 MHz** of collisions in real time. Performance matters!
- Experienced in **CUDA** and **SIMD** optimization.

What is this course about?

Two key optimization strategies: **SIMD** vectorization and **CUDA** GPU programming

The course has two parts:

- **Part 1: Lecture**
  - What are SIMD and CUDA, and why do they matter?
  - How to apply them to optimize your code
  - A complete optimization example, step by step

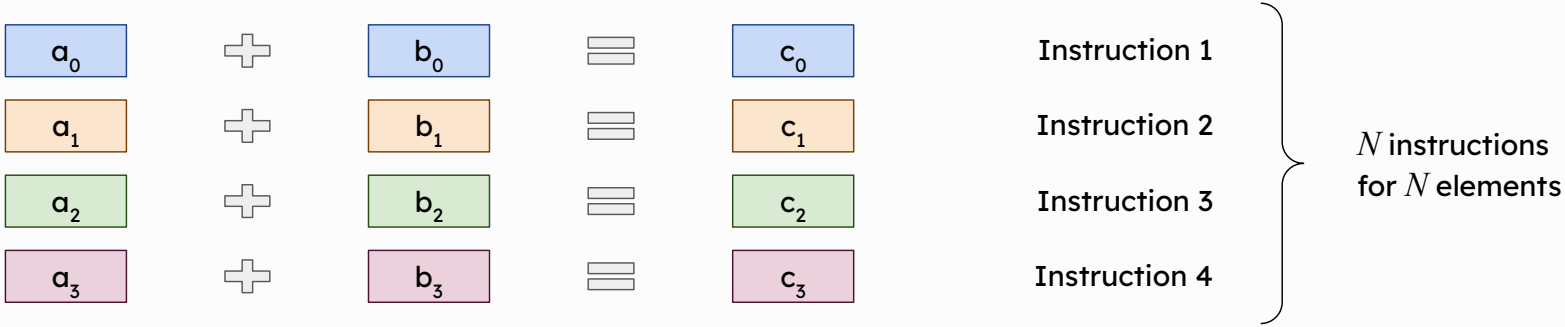
- **Part 2: Hands-on**
  - You face a real optimization problem on Google Colab
  - Apply what you learned in Part 1 to optimize the code yourself

I Introduction

SIMD

SIMD means “Single instruction, multiple data”

Scalar

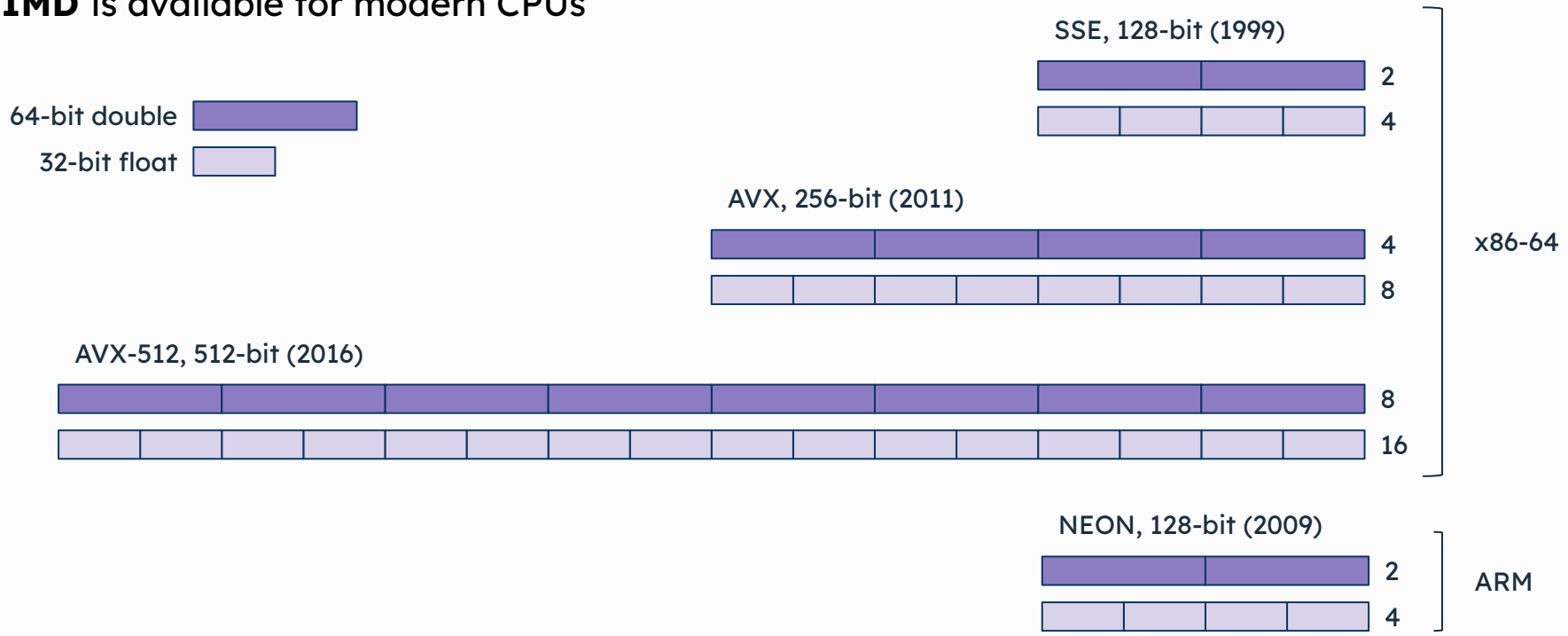


SIMD ($\omega = 4$)

N/ω instructions for N elements

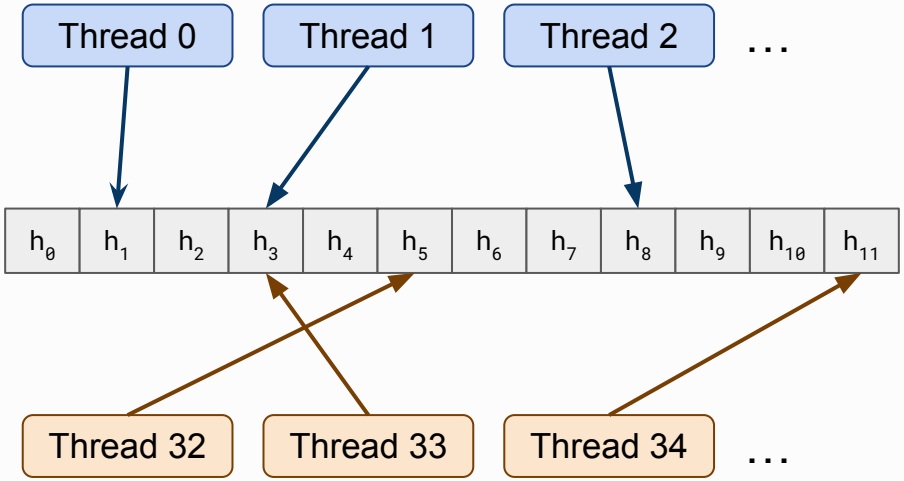
SIMD

SIMD is available for modern CPUs

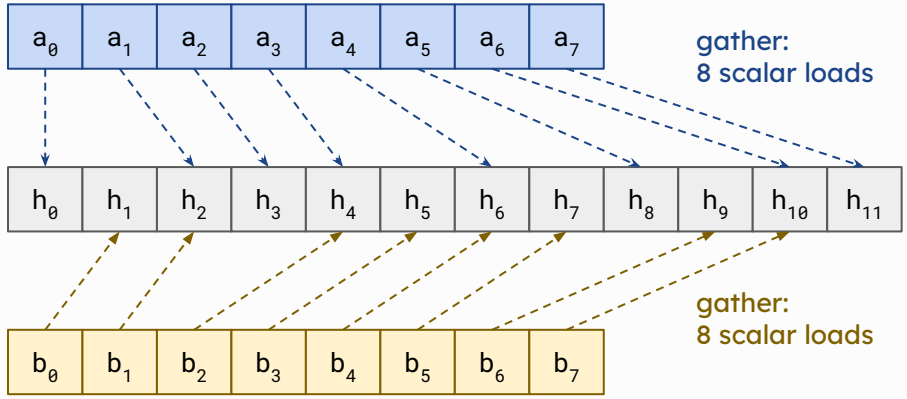


SIMD vs CUDA

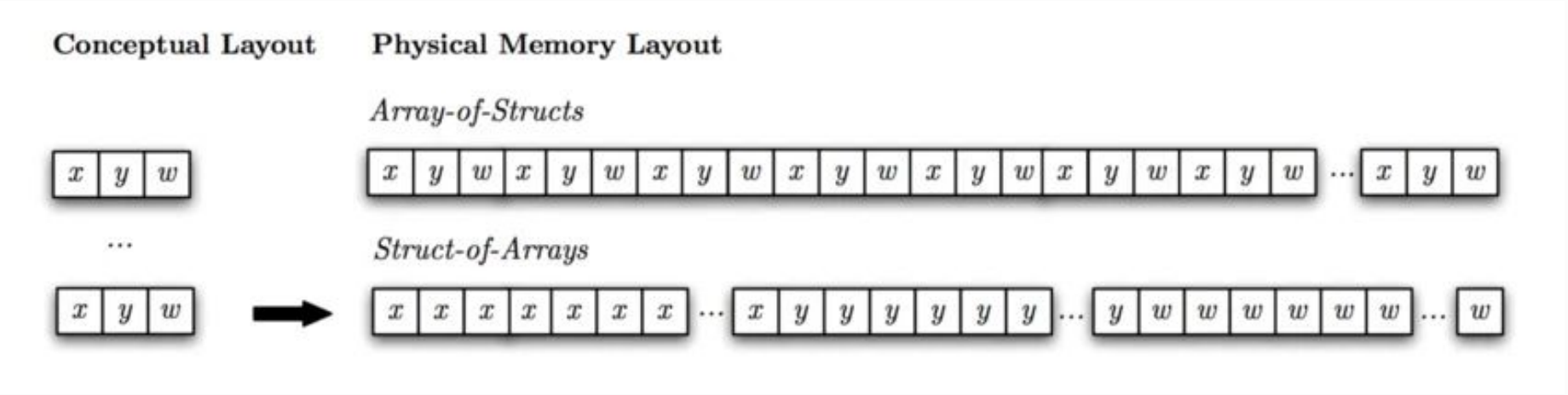
CUDA



SIMD



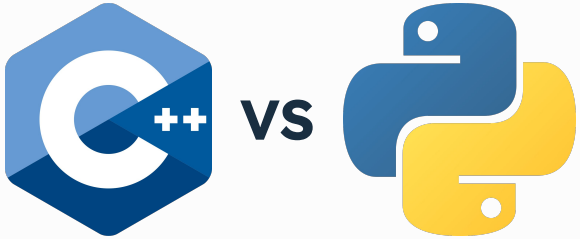
AoS vs SoA


Conceptual Layout
Physical Memory Layout
Array-of-Structs
x y w
...
x y w x y w x y w x y w x y w x y w x y w x y w ... x y w
Struct-of-Arrays
x y w
x x x x x x x ... x y y y y y y ... y w w w w w w ... w

II

C++ or Python

C++ vs Python



- **Common belief:** compiled C++ is faster than interpreted Python
- So the simplest optimization is to **rewrite everything** in **C++**... right?

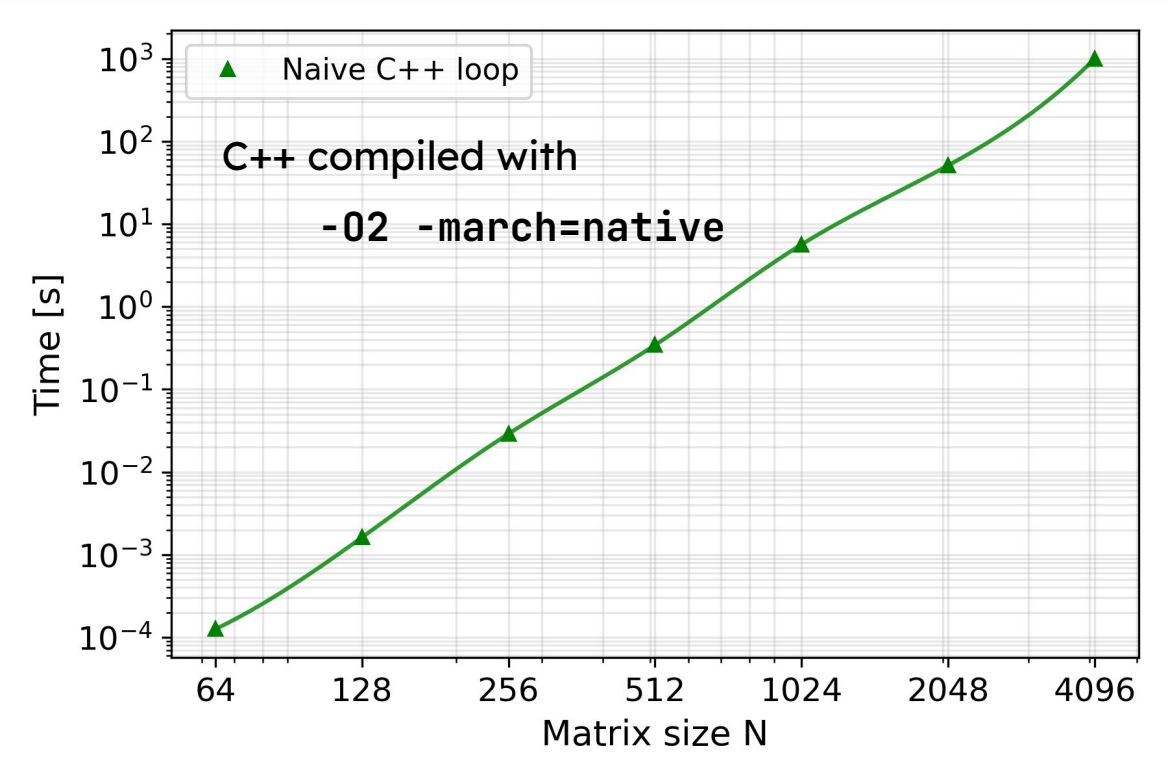
Let's find out!

Matrix multiplication

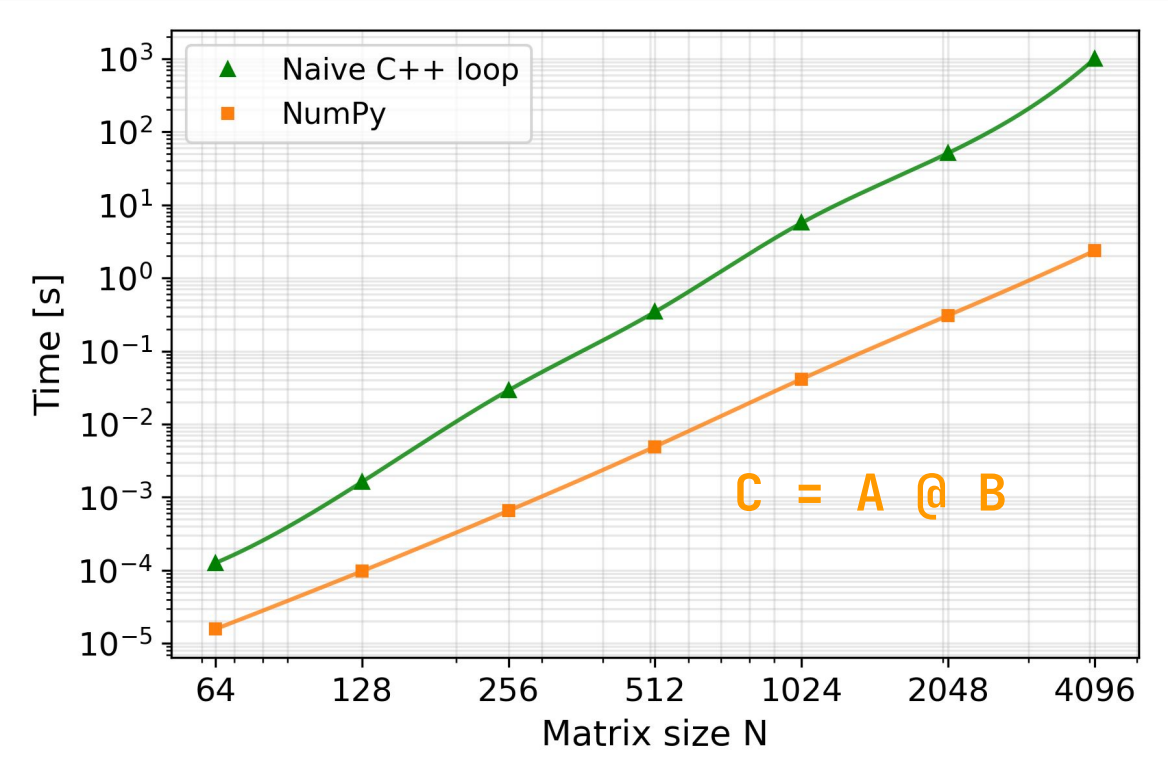
Let's **benchmark** with a simple **matrix multiplication**

```
void matmul(const double* A, const double* B, double* C, int N)
{   for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += A[i*N + k] * B[k*N + j];
            C[i*N + j] = sum;
        }
}
```

Matrix multiplication



Matrix multiplication



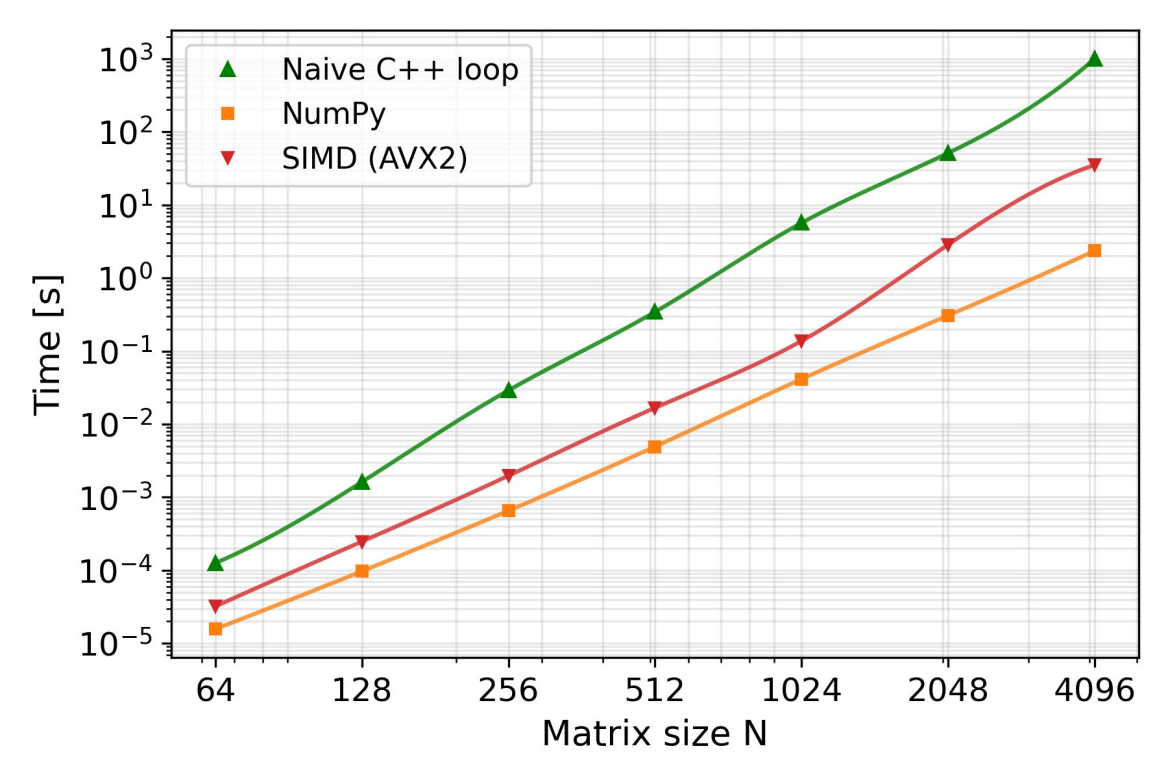
Wait, so Python is faster?

Matrix multiplication

Let’s do a SIMD optimization in C++

```cpp
namespace stdx = std::experimental;
using V = stdx::simd<double, stdx::simd_abi::fixed_size<4>>;
constexpr int W = V::size();  // 4 doubles per SIMD register

void matmul(const double* A, const double* B, double* C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            C[i*N + j] = 0.0;

        for (int k = 0; k < N; k++) {
            V a_ik(A[i*N + k]);              // broadcast A[i,k]
            for (int j = 0; j + W <= N; j += W) {
                V b(&B[k*N + j], stdx::element_aligned);  // load 4 B values
                V c(&C[i*N + j], stdx::element_aligned);  // load 4 C values
                c += a_ik * b;                            // 4 FMAs at once
                c.copy_to(&C[i*N + j], stdx::element_aligned);
            }
        }
    }
}
```

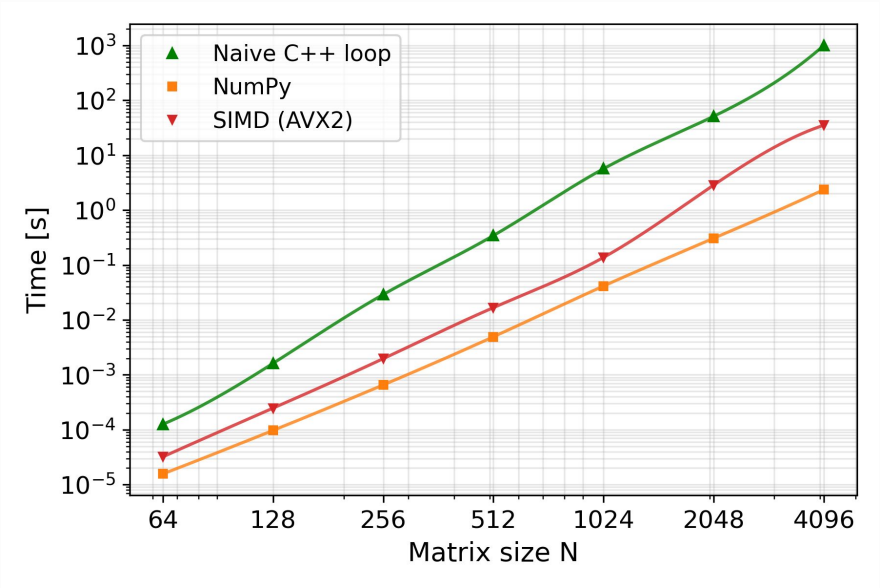
Matrix multiplication



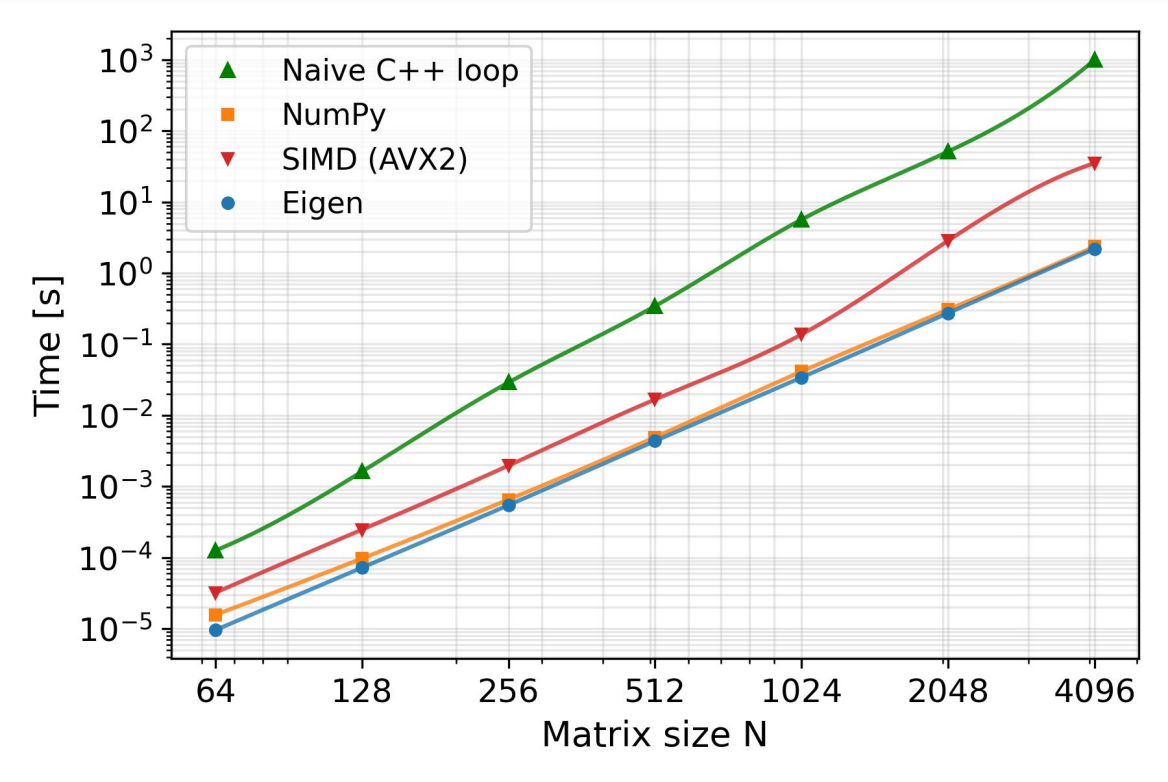
Matrix multiplication

- **SIMD** is one of the optimizations **NumPy** already uses.
- There are more optimizations available in **C++** (cache tiling, loop unrolling, ...)
- But implementing all of them manually is **unrealistic**
- Use modern libraries instead!

  e.g. Eigen: `C.noalias() = A * B`



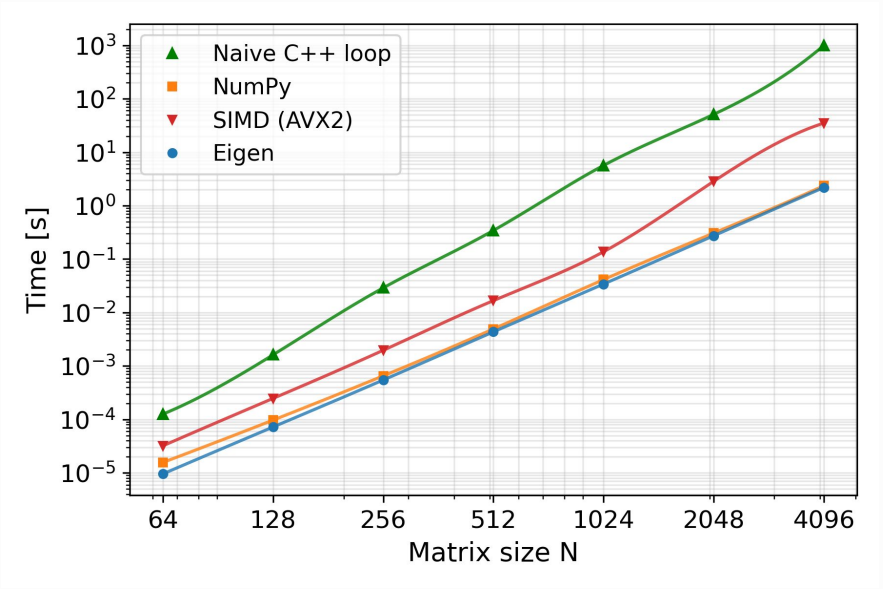
Matrix multiplication



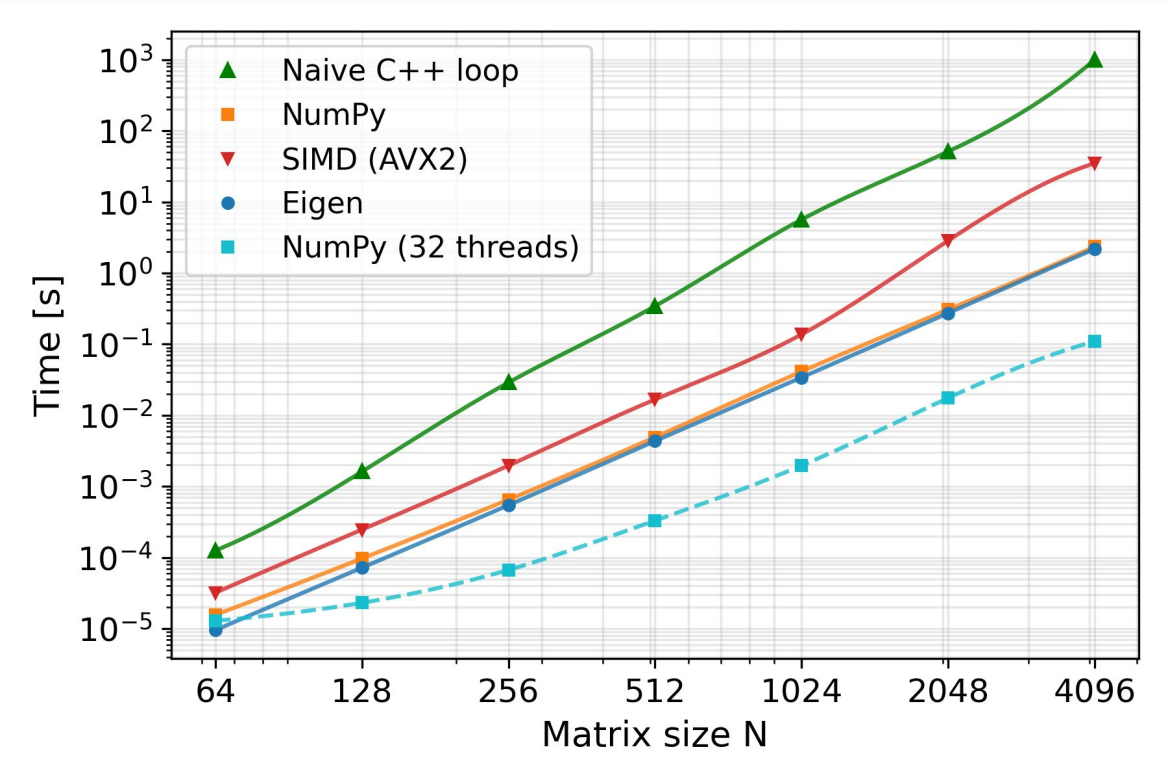
Matrix multiplication

- **NumPy** and **Eigen** show comparable performance.
- What matters is not the programming language, but the optimizations behind it.

Note: NumPy multi-threading is disabled here for a fair comparison (it is enabled by default).

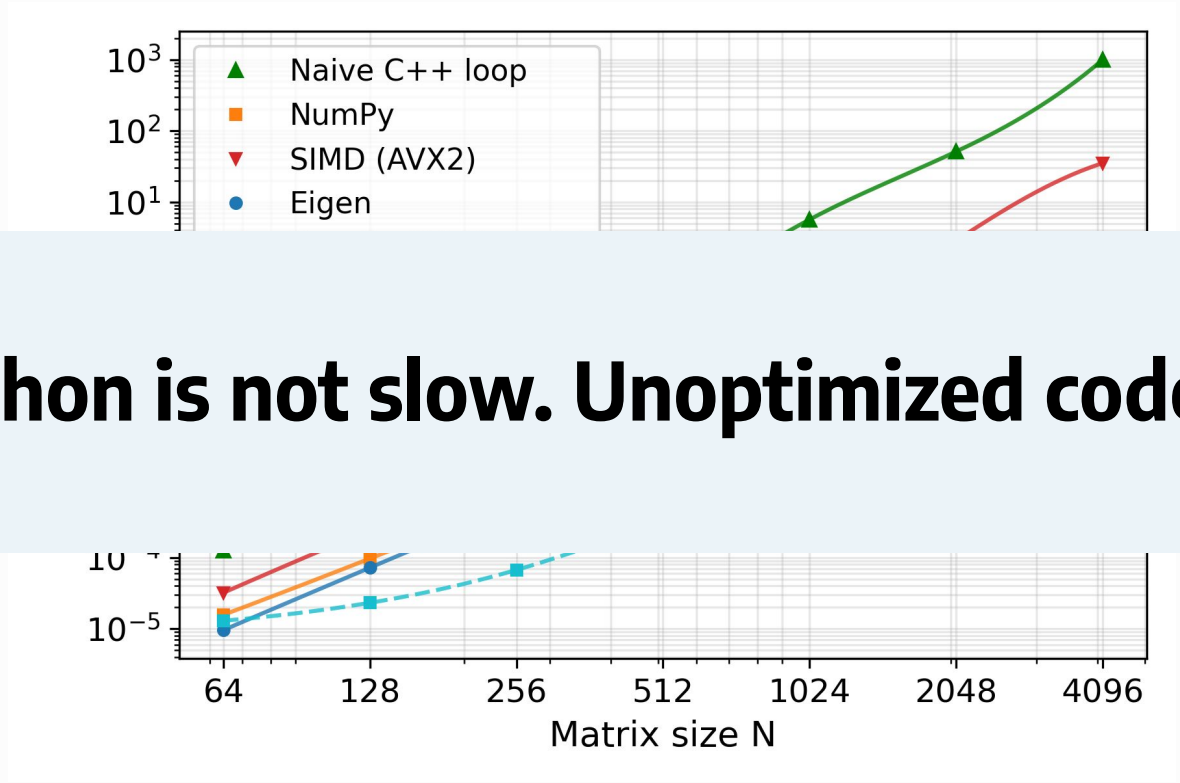


Matrix multiplication



The default NumPy is even faster than Eigen.

Matrix multiplication



Python is not slow. Unoptimized code is.

III

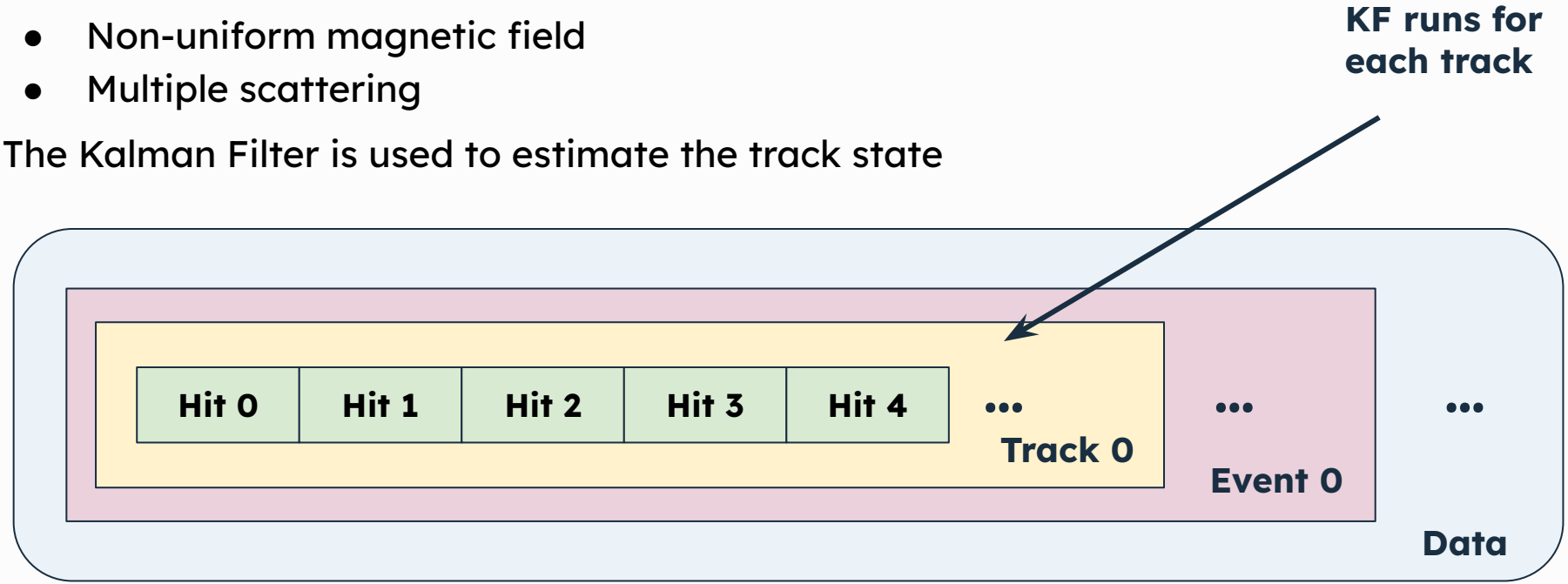
Kalman Filter

Kalman Filter

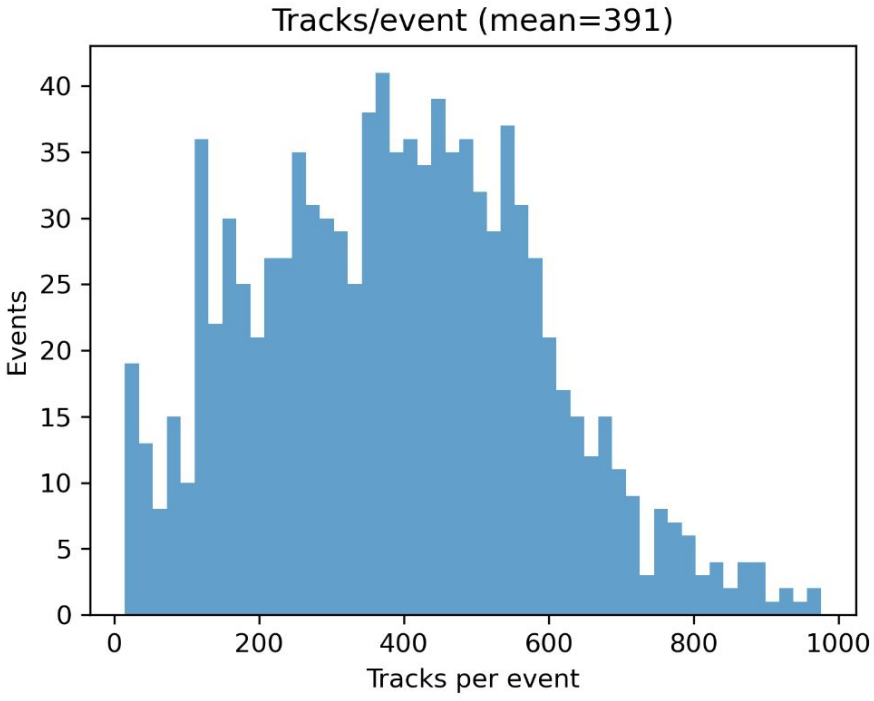
In HEP experiments, charged particle trajectories are complex:

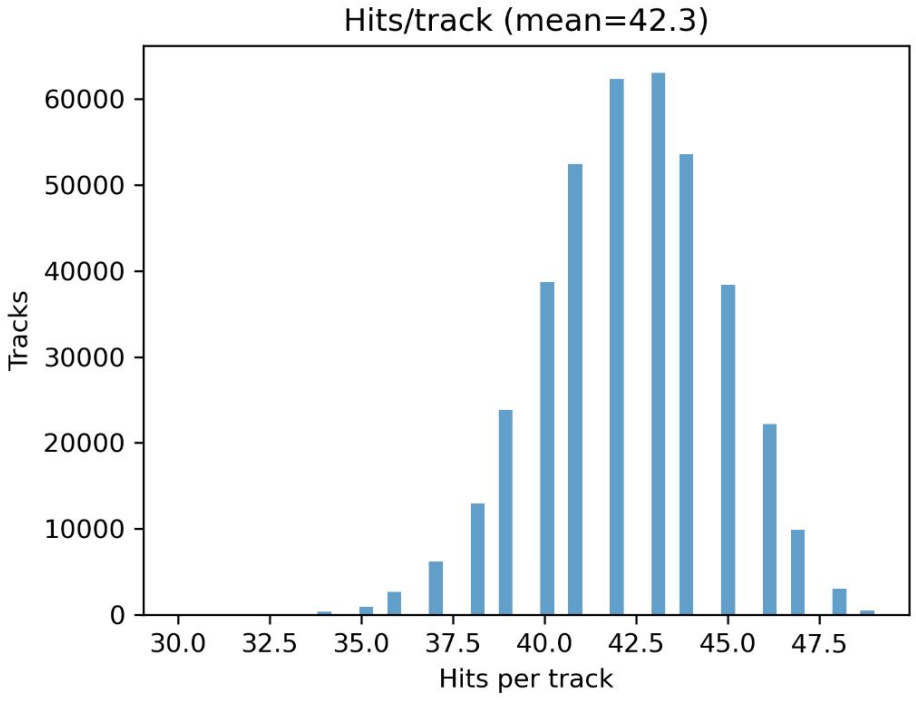
- Non-uniform magnetic field
- Multiple scattering

The Kalman Filter is used to estimate the track state

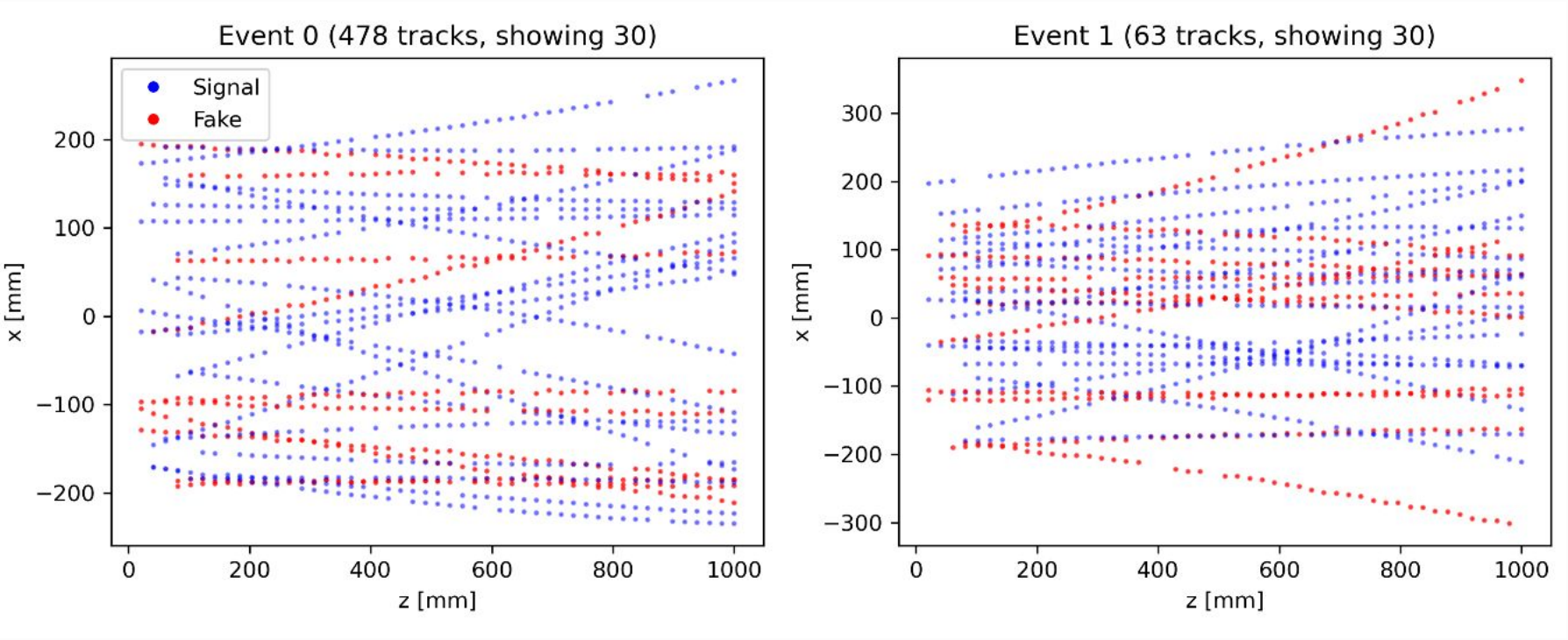


Kalman Filter





Kalman Filter



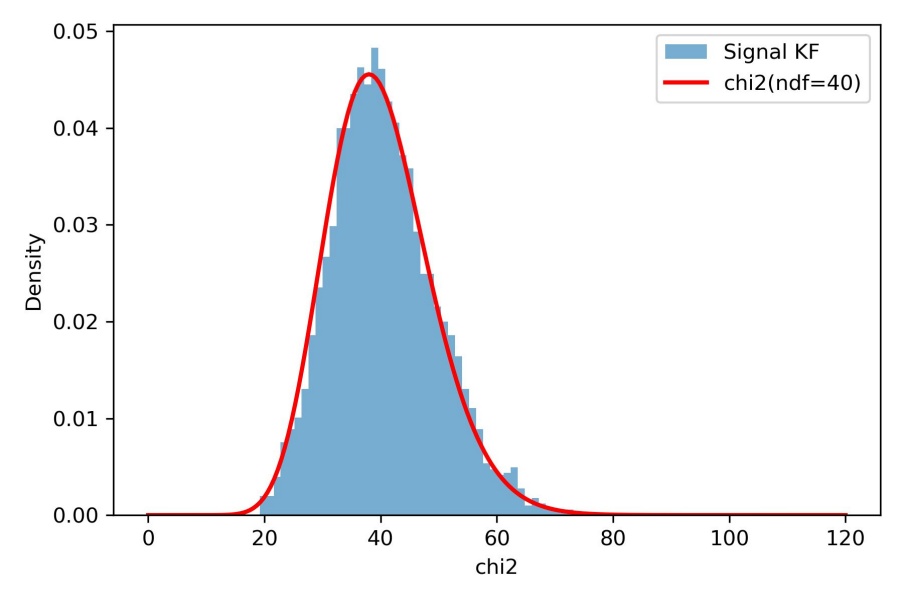
Kalman Filter

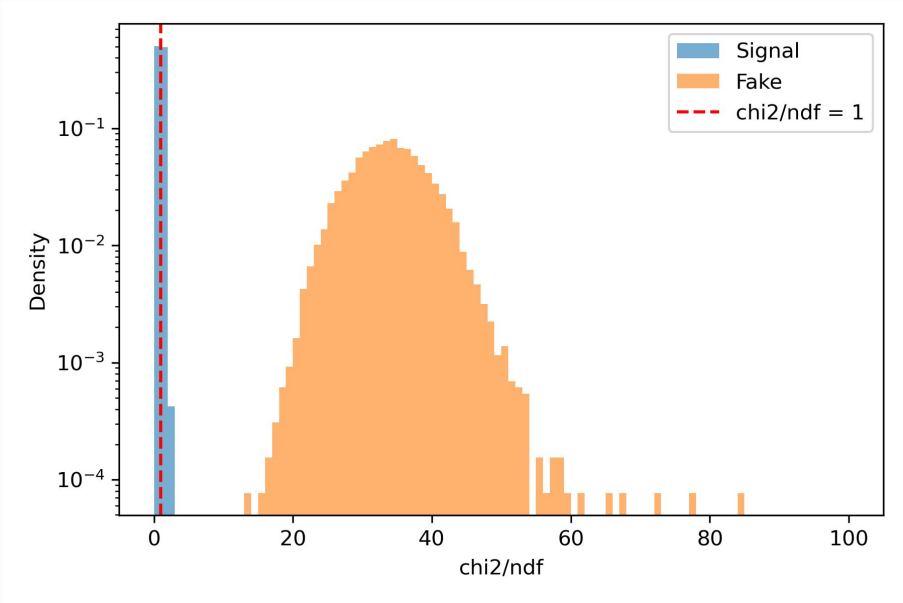
```python
def kf_fit_track(hit_x, hit_z, qop_seed, sigma_x, B, c_light):
    n = len(hit_x)
    H = np.array([[1.0, 0.0, 0.0]])
    R = np.array([[sigma_x**2]])

    state = np.array([hit_x[0], tx_seed, qop_seed])
    C = np.diag([sigma_x**2, 1.0, 1.0])
    chi2 = 0.0

    for i in range(1, n):
        F = kf_predict_jacobian(state[1], state[2], dz, B,
c_light)state = np.array([x_pred, tx_pred, state[2]])
        C = F @ C @ F.T

        residual = hit_x[i] - H @ state
        S = H @ C @ H.T + R
        K = C @ H.T @ np.linalg.inv(S)
        chi2 += (residual @ np.linalg.inv(S) @ residual.T).item()
        state = state + (K @ residual).ravel()
        C = (np.eye(3) - K @ H) @ C

    return chi2, n - 3
```

```python
class Hit:
    __slots__ = ("x", "z")

class Track:
    __slots__ = ("hits", "true_qop", "is_signal")

class Event:
    __slots__ = ("tracks",)
```

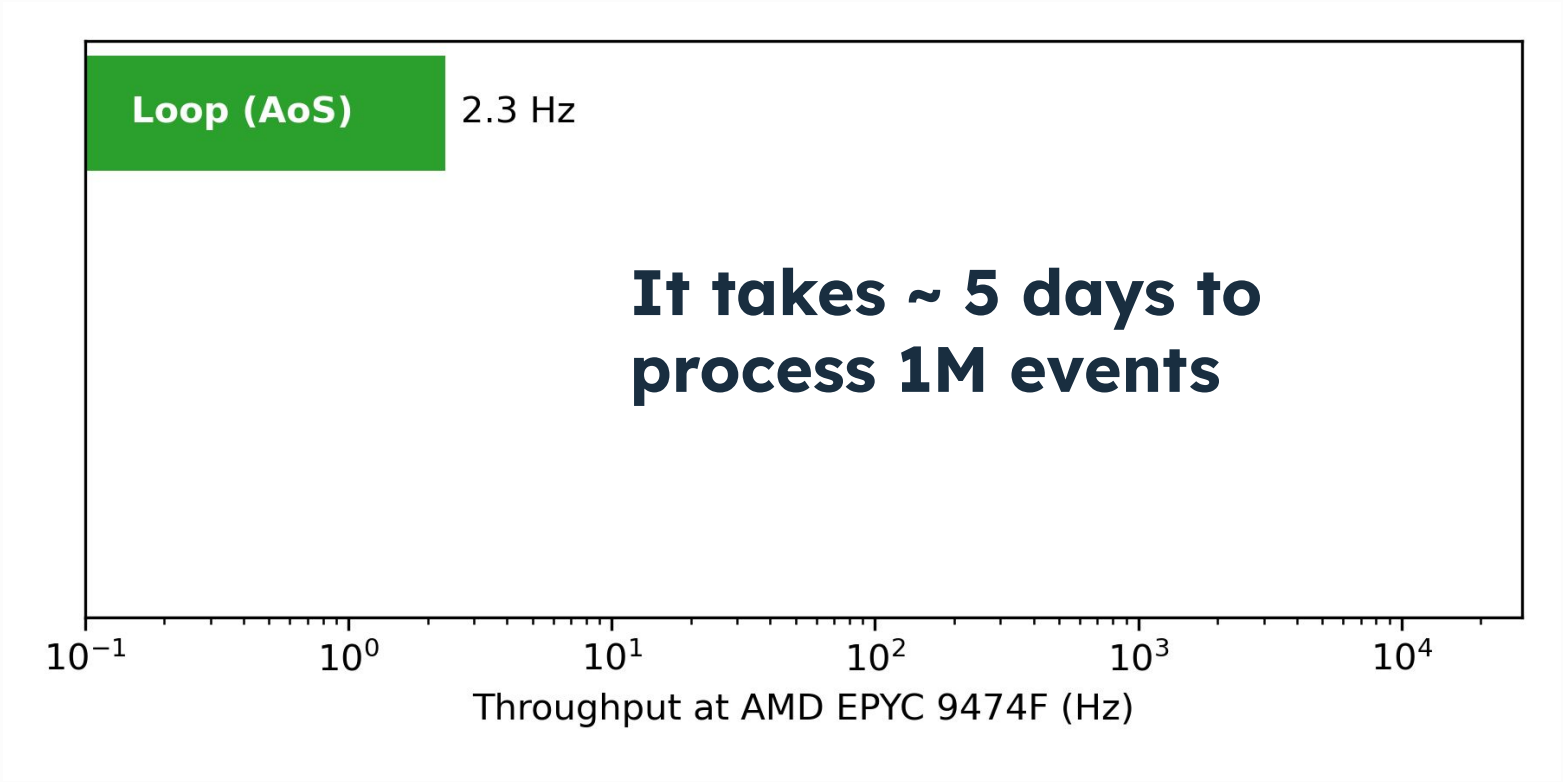
```python
for event in events:
    for track in event.tracks:
        chi2, ndf = kf_fit_track(
          track.hit_x, track.hit_z, ...)
```

Kalman Filter





Kalman Filter



Kalman Filter

```
Track 0: [(x,z), (x,z), (x,z)]
Track 1: [(x,z), (x,z)]
Track 2: [(x,z), (x,z), (x,z), (x,z)]
```

Convert from AoS to SoA

(padded)

```
padded_x:
          h0   h1   h2   h3
T0:    [ x    x    x    __ ]
T1:    [ x    x    __   __ ]
T2:    [ x    x    x    x  ]

padded_z:
          h0   h1   h2   h3
T0:    [ z    z    z    __ ]
T1:    [ z    z    __   __ ]
T2:    [ z    z    z    z  ]
```

Kalman Filter

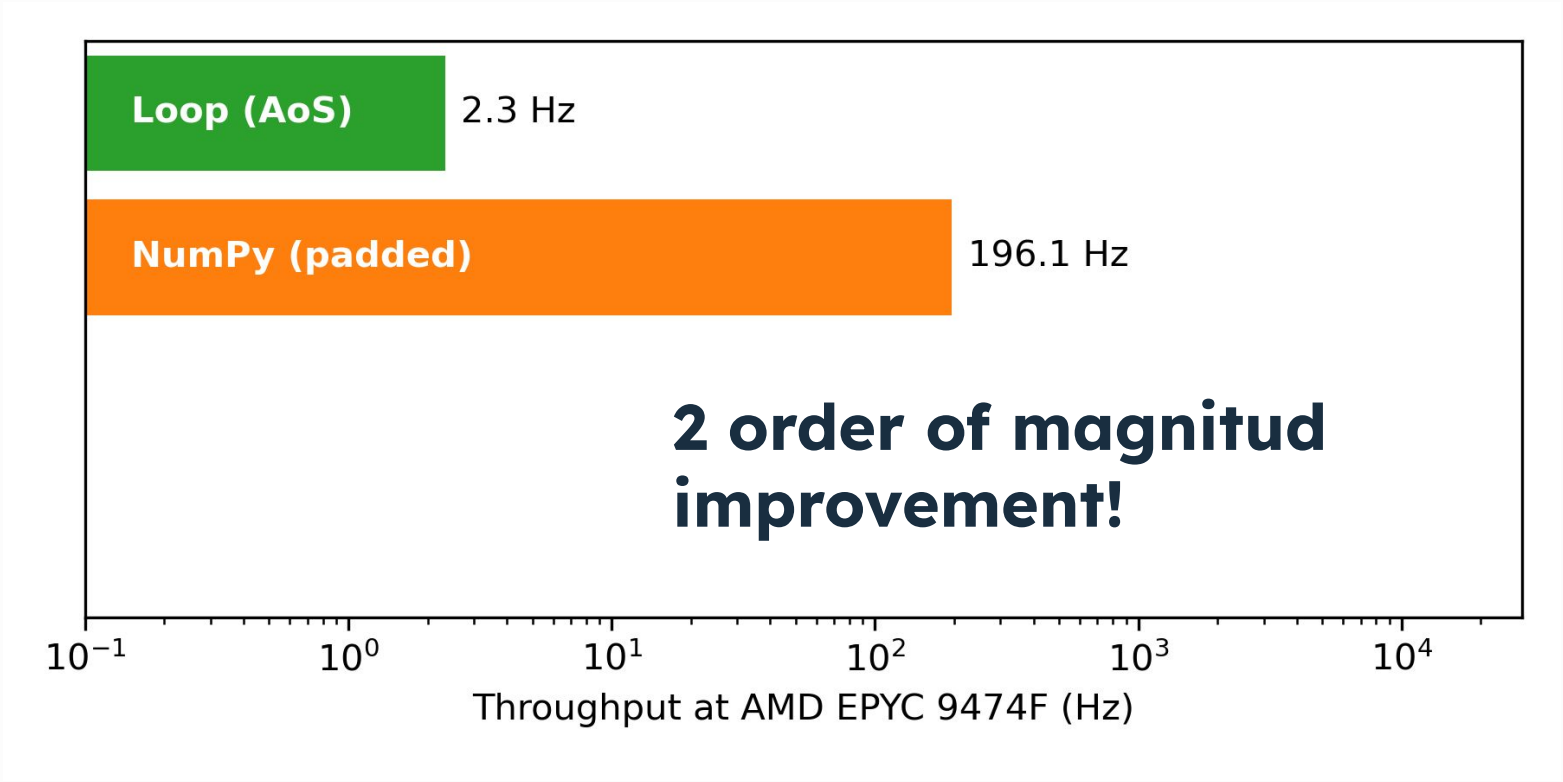
```python
def kf_padded(padded_x, padded_z, hit_mask, true_qop, sigma_x, B, c_light):
    N, M = padded_x.shape     # n_tracks, max_nhits
    state = np.zeros((N, 3))  # all tracks in parallel
    C = np.zeros((N, 3, 3))
    chi2 = np.zeros(N)

    for i in range(1, M):                          # loop over hit index
        active = hit_mask[:, i]

        F = np.zeros((N, 3, 3))                     # batched Jacobian
        ...
        C = F @ C @ F.transpose(0, 2, 1)            # batched matmul

        res = padded_x[:, i] - state[:, 0]          # vectorized residual
        S_val = C[:, 0, 0] + R
        chi2 += np.where(active, res**2 / S_val, 0)  # masked accumulation

        K = C[:, :, 0] / S_val[:, None]
        state += K * res[:, None]
        C -= K[:, :, None] * C[:, None, 0, :]

    return chi2
```

```python
# Pad all tracks to same length: (n_tracks, max_nhits)
padded_x = np.zeros((n_tracks, max_nhits))
padded_z = np.zeros((n_tracks, max_nhits))
hit_mask = np.zeros((n_tracks, max_nhits), dtype=bool)
```

NumPy supports batched matrix multiplication, so we can use it to optimize our code.

The idea is to process all tracks at once.

Kalman Filter



Kalman Filter

```
Track 0: [(x,z), (x,z), (x,z)]
Track 1: [(x,z), (x,z)]
Track 2: [(x,z), (x,z), (x,z), (x,z)]
```

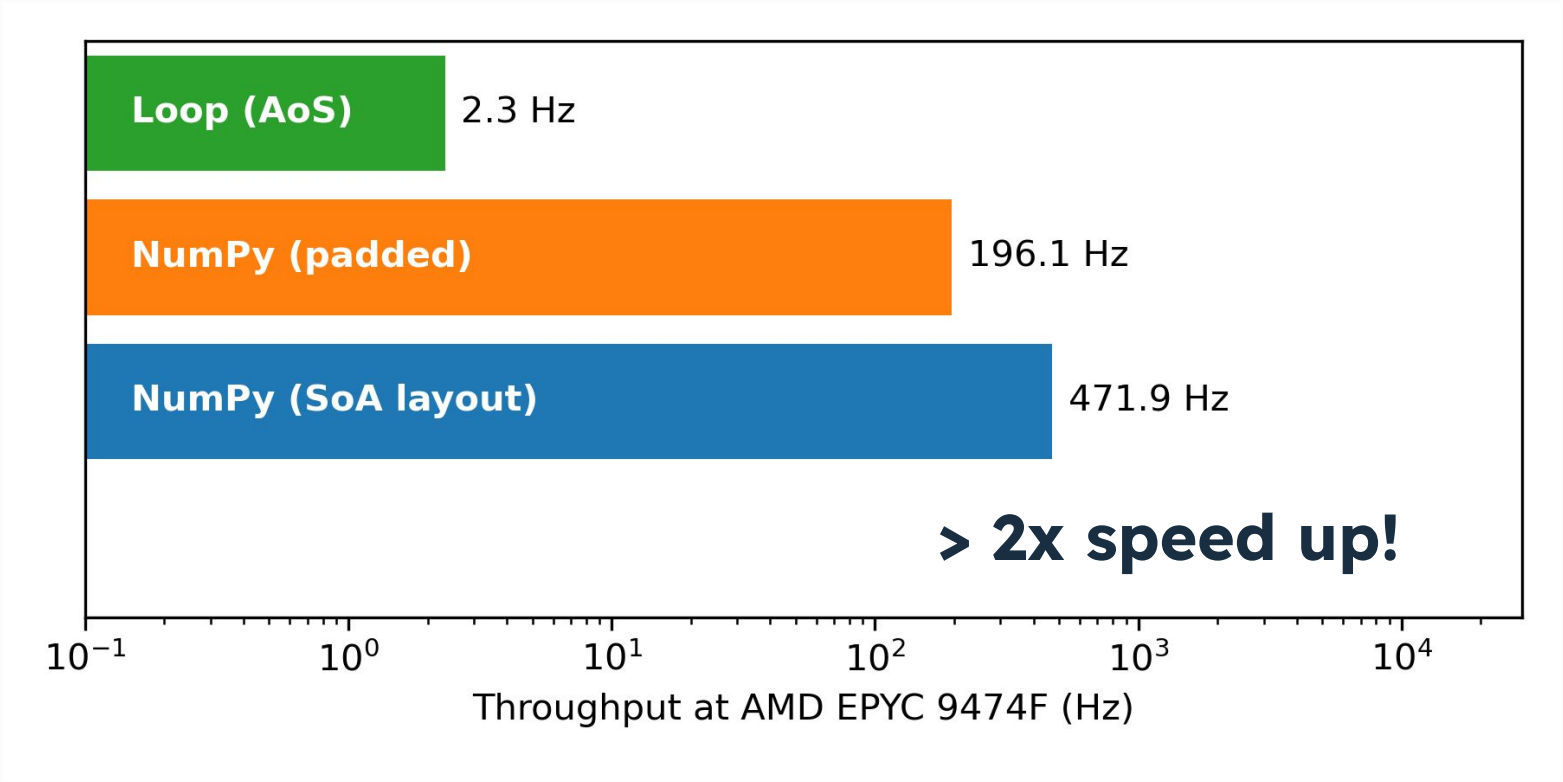
Convert from AoS to SoA

```
layer_x:                          z_planes:
          T0    T1    T2          Plane 0: z₀
Plane 0: [ x     x     x ]        Plane 1: z₁
Plane 1: [ x     x     x ]        Plane 2: z₂
Plane 2: [ x     __    x ]        Plane 3: z₃
Plane 3: [ __    __    x ]
```

Kalman Filter

```
def kf_soa(layer_x, layer_has_hit, true_qop, z_planes, sigma_x, B, c_light, xp=np):
    for p in range(n_planes):                    # loop over planes (50 iterations)
        active = layer_has_hit[p]                # which tracks have a hit here
        m = active & seeded

        # predict + update using scalar arrays — no matrix allocation
        dz = z_planes[p] - last_z[m]
        ...
        C00[m] = fc00 + fc01*F01 + fc02*F02   # expanded F @ C @ F^T
        ...
        chi2[m] += res**2 / S                  # scalar update

    return chi2
```

Kalman Filter



Kalman Filter

```
def kf_soa(layer_x, layer_has_hit, true_qop, z_planes, sigma_x, B, c_light, xp=np):
    for p in range(n_planes):                    # loop over planes (50 iterations)
        active = layer_has_hit[p]                # which tracks have a hit here
        m = active & seeded

        # predict + update using scalar arrays — no matrix allocation
        dz = z_planes[p] - last_z[m]
        ...
        C00[m] = fc00 + fc01*F01 + fc02*F02   # expanded F @ C @ F^T
        ...
        chi2[m] += res**2 / S                  # scalar update

    return chi2
```

We use mask to replace if-else operations

Kalman Filter

NumPy

```
import numpy as np
x_cpu = np.zeros((10,))
W_cpu = np.zeros((10, 5))
y_cpu = np.dot(x_cpu, W_cpu)
```

```
y_cpu = cp.asnumpy(y_gpu)
```

CuPy

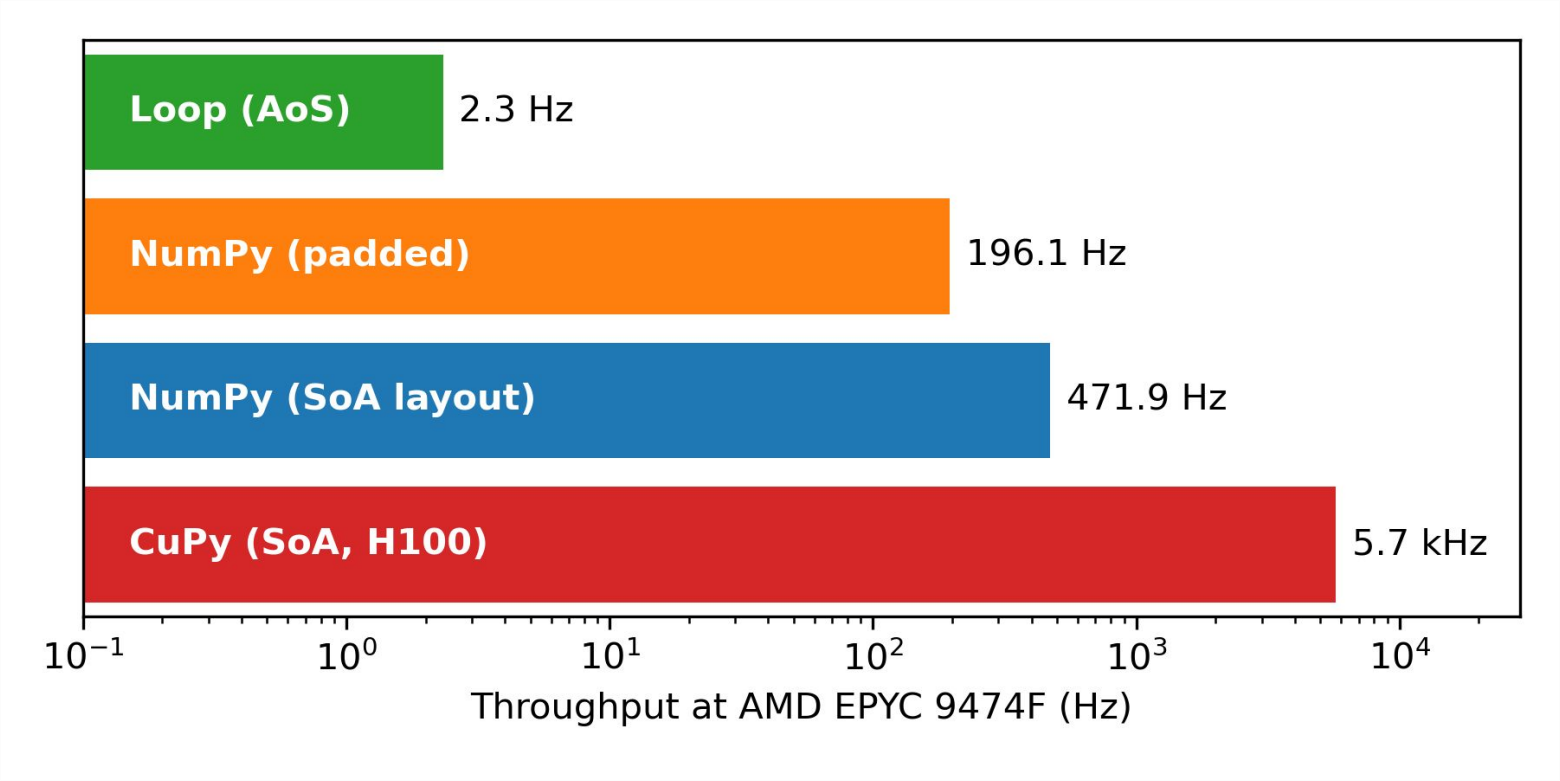
```
import cupy as cp
x_gpu = cp.zeros((10,))
W_gpu = cp.zeros((10, 5))
y_gpu = cp.dot(x_gpu, W_gpu)
```

```
y_gpu = cp.asarray(y_cpu)
```

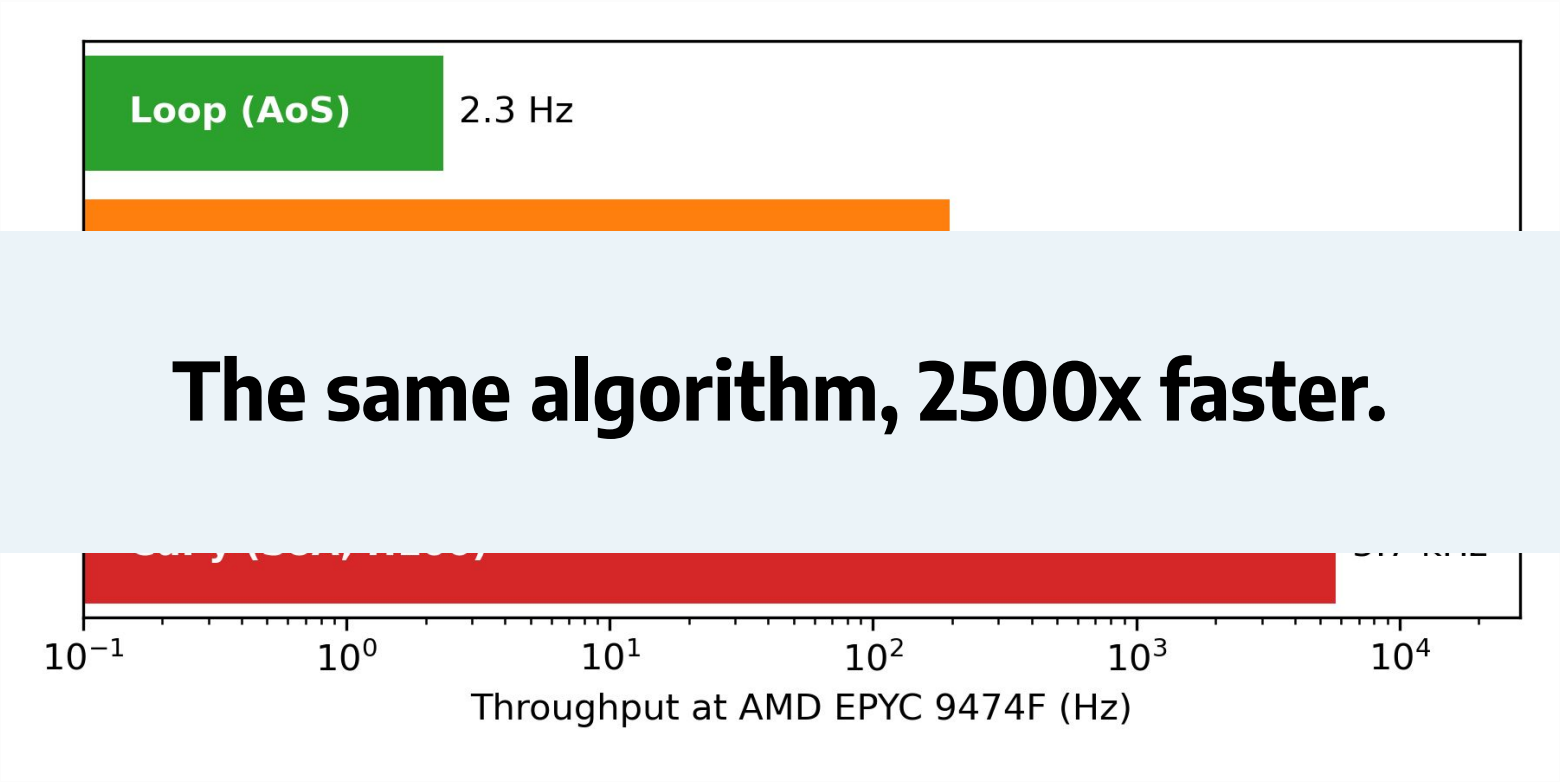
Kalman Filter

```python
import cupy as cp
from kf_core import kf_soa

# upload to GPU
layer_x_gpu     = cp.asarray(layer_x)
layer_has_hit_gpu = cp.asarray(layer_has_hit)
true_qop_gpu    = cp.asarray(true_qop)
z_planes_gpu    = cp.asarray(z_planes)

# same function, just xp=cp instead of xp=np
chi2 = kf_soa(layer_x_gpu, layer_has_hit_gpu, true_qop_gpu, z_planes_gpu,
              sigma_x, B, c_light, xp=cp)
```

Kalman Filter



Kalman Filter



The same algorithm, 2500x faster.

IV Hands on

Cholesky inverse

matrix A^{-1} could be computed in the form of $A^{-1} = (L^{-1})^* L^{-1}$.

A is a Hermitian matrix

$$AA^{-1} = I$$

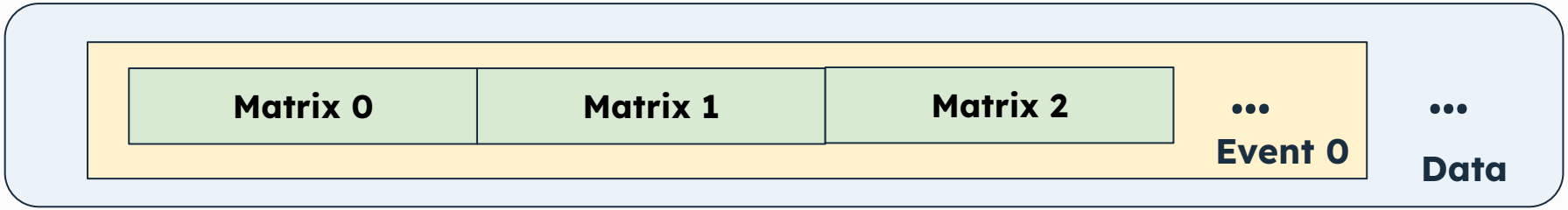
Cholesky decomposition → $$A = L * L^*$$

$$A^{-1} = (L^{-1})^* L^{-1}$$

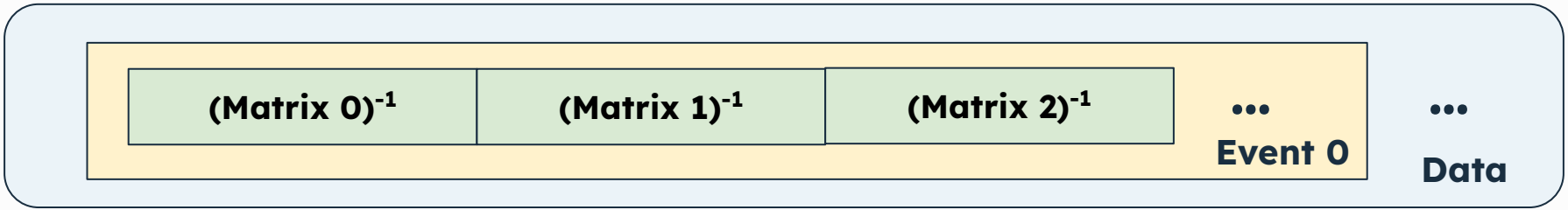
As matrix L is a triangular matrix, L^{-1} is easy to compute.

Cholesky inverse

Input data



Output data



SIMD

Input format

```
data = np.load("matrices.npz")
flat_matrices = data["flat_matrices"]
sizes = data["sizes"]
offsets = data["offsets"]
event_offsets = data["event_offsets"]
n_matrices = len(sizes)
n_events = len(event_offsets) - 1

ref = np.load("cholesky_ref.npz")
flat_inv_ref = ref["flat_inv"]
```

flat_matrices

```
[ a b c d | e f g h i j k l m | n o p q ]
  ←2×2→     ←———3×3———→       ←2×2→
```

sizes

```
[ 2, 3, 2 ]
```

offsets

```
[ 0, 4, 13, 17 ]
```

event_offsets

```
[ 0, 2, 3 ]
```

Cholesky inverse

Baseline (loop)

```
flat_inv_np = np.zeros_like(flat_matrices)

start = time.time()
for ev in range(n_events):
    m_start, m_end = event_offsets[ev], event_offsets[ev + 1]
    for i in range(m_start, m_end):
        n = sizes[i]
        M = flat_matrices[offsets[i]:offsets[i+1]].reshape(n, n)
        L = np.linalg.cholesky(M)
        L_inv = np.linalg.inv(L)
        M_inv = L_inv.T @ L_inv
        flat_inv_np[offsets[i]:offsets[i+1]] = M_inv.ravel()
elapsed = time.time() - start

print(f"np.linalg loop: {elapsed:.1f}s, {n_events / elapsed:.1f} events/s, {n_matrices / elapsed:.0f} matrices/s")
validate(flat_inv_np)
```

Cholesky inverse

Ideas

```
for each unique size n:
    gather all n×n matrices into a batch of shape (batch_size, n, n)

    # NumPy handles the entire batch at once
    L     = cholesky(batch)          # (batch_size, n, n)
    L_inv = inv(L)                   # (batch_size, n, n)
    M_inv = L_inv^T @ L_inv          # (batch_size, n, n)

    scatter results back to flat array
```

Notebook

https://links.uv.es/2cI8Tls

https://ir.uv.es/2cI8Tls

https://go.uv.es/2cI8Tls