

The dynamix software for XPCS

1 - Introduction to XPCS

1.1 - XPCS: what ?

- X-ray photon correlation spectroscopy (XPCS)
- Probe dynamics in systems that are non-ergodic/out of equilibrium state systems
- Far-field scattering in small-angle (SAXS) and wide-angle (WAXS) geometries
- Adapted to fast dynamics (20 kHz detector), and long time scales (100s of seconds)

1.2 - XPCS: who ?

At ESRF:

- ID10
- ID02
- in the future: ID18

1.3 - XPCS: how ?

- Acquire a series of images of scattering spectra
- Temporal correlation of a speckle-like pattern

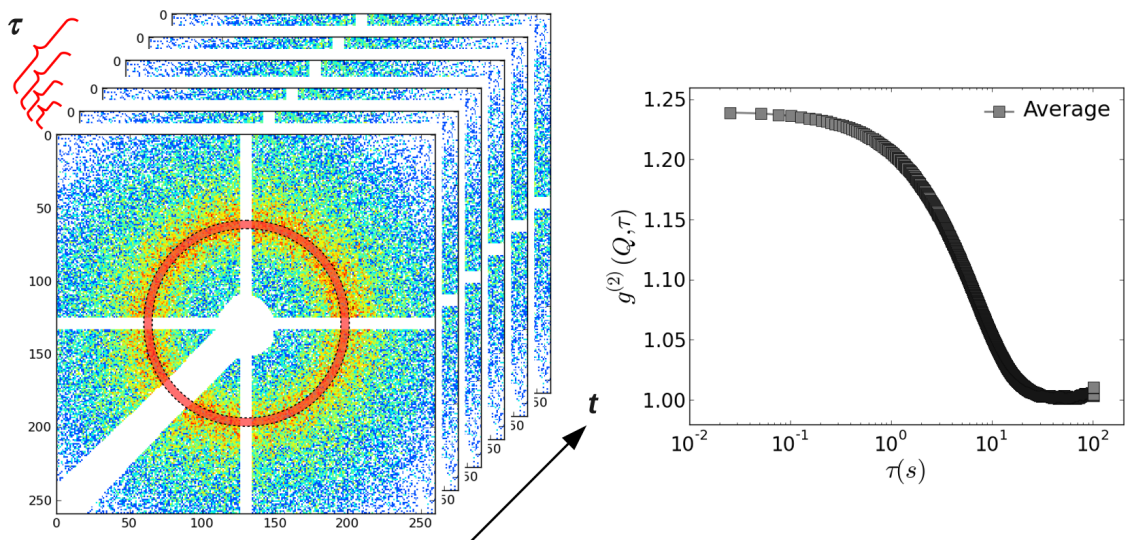


image courtesy: Y. Chushkin

In XPCS, the quantity of interest is the **correlation function** [1, 2]

$$g_2(q, \tau) = \frac{\langle \langle I(t, p) I(t + \tau, p) \rangle_p \rangle_t}{\langle \langle I(t, p) \rangle_p \rangle_t \langle \langle I(t + \tau, p) \rangle_p \rangle_t}$$

where $\langle \cdot \rangle_t$ and $\langle \cdot \rangle_p$ denote **time averaging** and **pixels averaging** respectively.

This work is about computing g_2 efficiently.

[1] P.N. Pusey, W. Van Megen, "Dynamic Light Scattering by non-ergodic media", Physica A 157 (1989) 705-741

[2] D Lumma et al. "Area detector based photon correlation in the regime of short data batches: Data reduction for dynamic x-ray scattering". In: Review of Scientific Instruments 71.9 (2000), pp. 3274–3289.

The ensemble average is defined as

$$\langle I(p, t) \rangle_p = \frac{1}{|P_{\vec{q}}|} \sum_{p \in P_{\vec{q}}} I(p, t)$$

where $|P_{\vec{q}}|$ is the cardinality of the set $P_{\vec{q}}$.

This is a radial integration, in the frame acquired at time t , of all the pixels p belonging to the Q -vector \vec{q} .

Therefore, the time correlation of the ensemble averaging is defined as

$$\langle \langle I(p, t) I(p, t + \tau) \rangle_p \rangle_t = \frac{1}{\Delta t} \sum_{t=0}^{\Delta t} \left(\frac{1}{|P_{\vec{q}}|} \sum_{p \in P_{\vec{q}}} I(p, t) I(p, t + \tau) \right)$$

$$g_2(q, \tau) = \frac{\langle \langle I(t, p) I(t + \tau, p) \rangle_p \rangle_t}{\langle \langle I(t, p) \rangle_p \rangle_t \langle \langle I(t + \tau, p) \rangle_p \rangle_t}$$

where $\langle \cdot \rangle_t$ and $\langle \cdot \rangle_p$ denote **time averaging** and **pixels averaging** respectively.

In 2019: work to compute $g_2(\tau)$ without computing the full matrix

In 2023: work to compute the **two-times correlation matrix** $g_2(t_1, t_2)$:

$$g_2(q, t_1, t_2) = \frac{\langle I(t_1, p)I(t_2, p) \rangle_p}{\langle I(t_1, p) \rangle_p \langle I(t_2, p) \rangle_p}$$

2. Calculation of $g_2(\tau)$, introduction

2.1 - XPCS, the ~naive~ simple way

Calculating the g_2 function can be done through **matrix-matrix multiplication**.

Let p_0 be pixel index 0 in the current q-bin

(spatial coordinate converted to 1D eg. (0, 0) --> 0),

p_1 the spatial index 1, and so on.

Stack the (flattened) frames into a matrix of shape (N_t, N_p) :

$$\vec{A} = \begin{bmatrix} I_{0,p_0} & I_{0,p_1} & \dots & I_{0,p_{N_p}-1} \\ I_{1,p_0} & I_{1,p_1} & \dots & I_{1,p_{N_p}-1} \\ \vdots & \vdots & \dots & \vdots \\ I_{N_t-1,p_0} & I_{N_t-1,p_1} & \dots & I_{N_t-1,p_{N_p}-1} \end{bmatrix}$$

Then:

$$\vec{A}\vec{A}^T = \begin{bmatrix} \sum_p I_{0,p}^2 & \sum_p I_{0,p}I_{1,p} & \dots & \sum_p I_{0,p}I_{N_t-1,p} \\ \sum_p I_{1,p}I_{0,p} & \sum_p I_{1,p}^2 & \dots & \sum_p I_{1,p}I_{N_t-1,p} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_p I_{N_t-1,p}I_{0,p} & \sum_p I_{N_t-1,p}I_{1,p} & \dots & \sum_p I_{N_t-1,p}^2 \end{bmatrix}$$

- This matrix is the numerator of TTCF $g_2(t_1, t_2)$
- Summing over diagonal k gives the numerator of $g_2(t, t + k)$

```

def dense_correlator(xpcs_data, mask):
    ind = np.where(mask > 0) # unused pixels are 0
    xpcs_data = np.array(xpcs_data[:, ind[0], ind[1]], np.float32)
    # (n_tau, n_pix)
    meanmatr = np.mean(xpcs_data, axis=1) # i.e
    xpcs_data.sum(axis=-1).sum(axis=-1)/n_pix
    ltimes, lenmatr = np.shape(xpcs_data) # n_tau, n_pix
    meanmatr.shape = 1, ltimes

    num = np.dot(xpcs_data, xpcs_data.T) # big matrix
    multiplication
    denom = np.dot(meanmatr.T, meanmatr) # small matrix
    multiplication

    g2 = np.zeros(ltimes)
    g2_std = np.zeros_like(g2)

    for i in range(ltimes):
        dia_n = np.diag(num, k=i) / lenmatr
        dia_d = np.diag(denom, k=i)
        g2[i] = np.sum(dia_n) / np.sum(dia_d)
        g2_std[i] = np.std(dia_n / dia_d) / sqrt(len(dia_d))
    return (g2, g2_std, num, denom)

```

Works well... until a certain point.

2.2 - The need for doing the computations on a compact data format

Problem 1: memory

A typical XPCS acquisition is 10k - 1.2M frames (usually ~50k frames).

Eiger 4M with 1 byte/pixel:

- 180 GB for 40k frames
- 450 GB for 100k frames

Even 10k frames would not fit in a big GPU

- data type is 1-Byte wide, but computations are done on 4 Bytes.

Problem 2: computational efficiency

XPCS data is inherently very sparse: most of the data consists in zeros.

So matrix multiplications entails 99% of "multiplying zeros with zeros".

==> Need to perform the computations natively on a compact data format

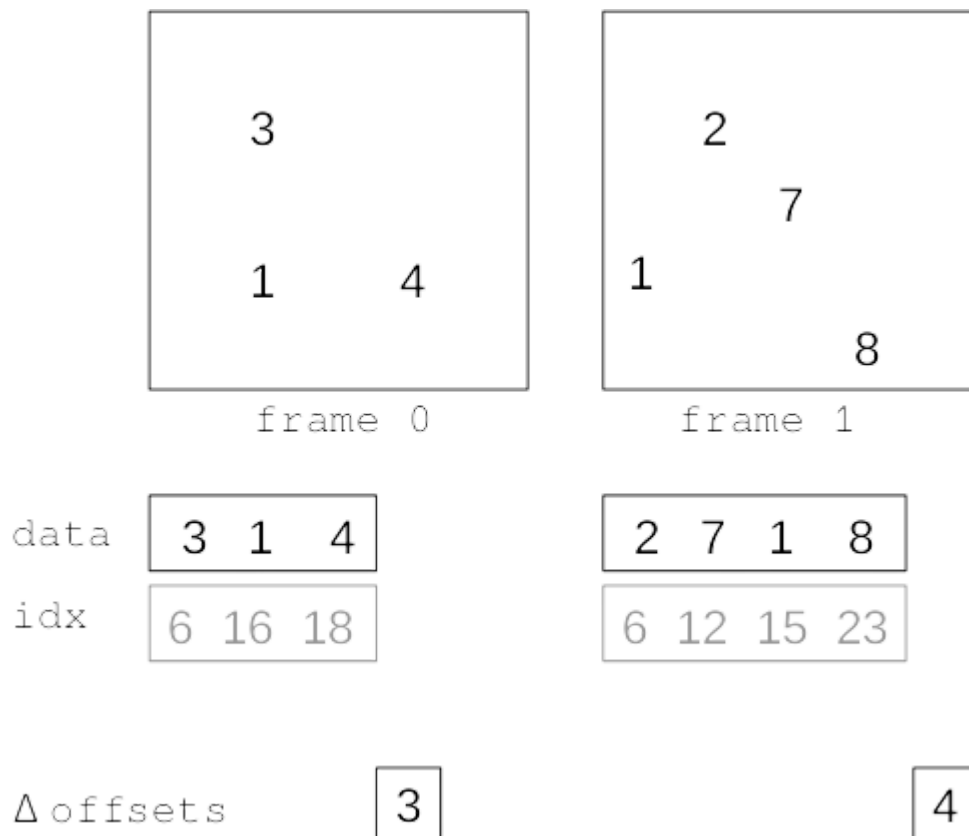
3. Compacts formats

3.1 - Space-compact format

- Very simple to understand and implement.
 - Implemented by default in Lima2
- Keep only the non-zero elements of each frame
- Store the spatial indices, and an "offset" array to switch between frames

```
indices = [numpy.where(xpcs.ravel())[0] for xpcs in xpcs_frames]  
offsets = numpy.cumsum([len(s) for s in _indices])
```

3.1 - Space-compact format



3.2 - Time-compact format

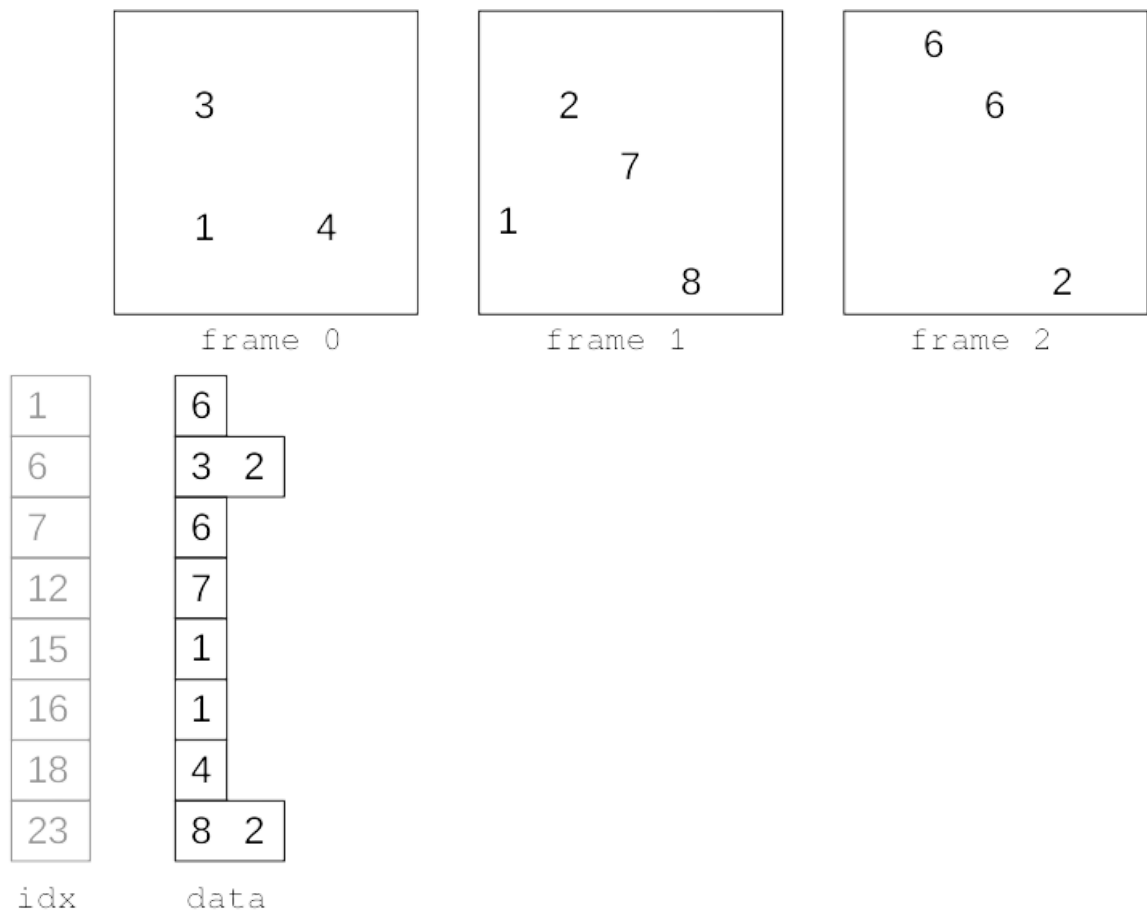
Alternate: compact the data along the "time" dimension.

- The (spatial) indices becomes time indices
- Same for offsets
- Less intuitive and convenient

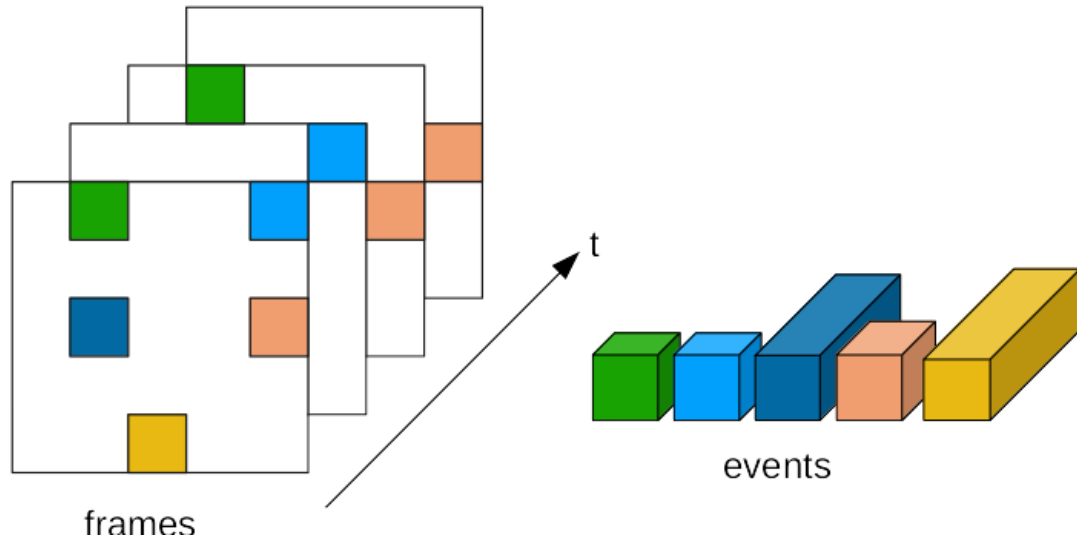
Based on the "event correlator" data structure from ID10 [3]

[3] Y Chushkin, C Caronna, and A Madsen. “A novel event correlation scheme for X-ray photon correlation spectroscopy”. In: Journal of Applied Crystallography 45.4 (2012), pp. 807–813.

3.2 - Time-compact format



3.2 - Time-compact format



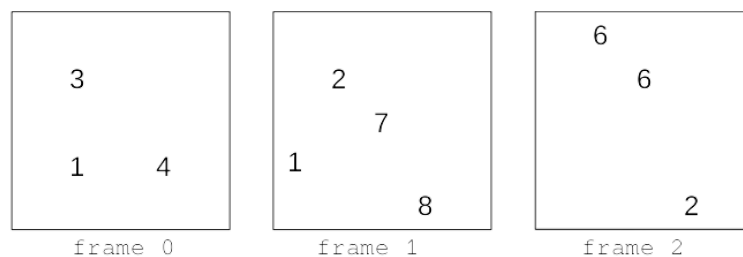
3.3 - (Bonus) conversion from space-compact to time-compact

Data can be converted from space-compact format to time-compact, without going into the "dense domain"

1. Count how many non-zero elements there are at each fixed pixel location.
 - This is done simply by a "bincount(pix_idx)".
2. Cumulate-sum this "counter" structure to get the "times offset"
 - `t_offset[i+1] - t_offset[i]` gives how many non-zero pixels there are at pixel location "i".

OpenCL implementation takes less than 1s for 100k frames

3.3 - (Bonus) conversion from space-compact to time-compact



idx 6 16 18 | 6 12 15 23 | 1 7 23

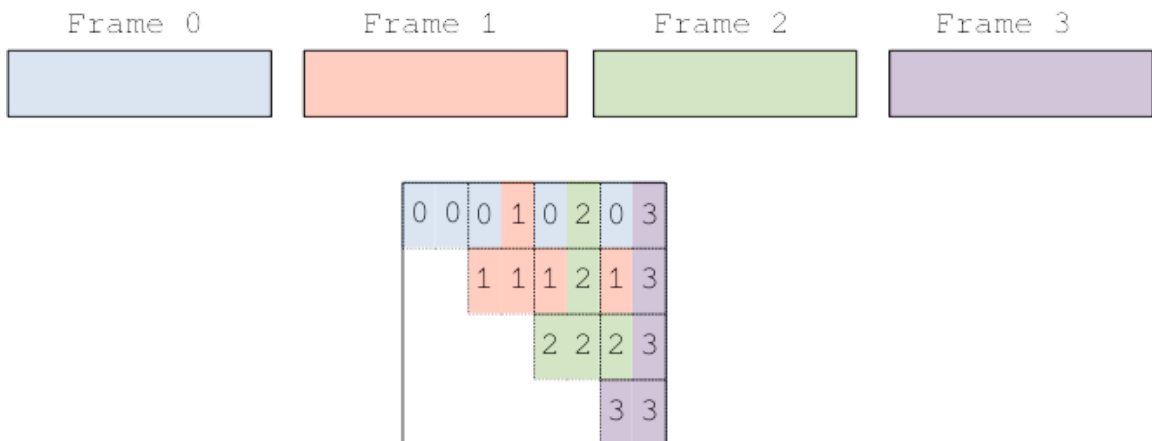
0	1	0	0	0	0	2	1	0	0	0	0	1	0	0	1	1	0	1	0	0	0	0	0	2
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

4. Computing TTCF: parallel implementation

The most difficult problem is to compute the numerator of $g_2(t_1, t_2)$:

$$\text{numerator}(g_2)(t_1, t_2) = \sum_p I(t_1, p) I(t_2, p)$$

which is a matrix.



4.1 - What is the minimum number of operations needed ?

- To get an idea of the difficulty of the problem (and some efficiency bounds on implementations), it's good to know the number of operations
- operation = memory read / arithmetic operation (add, mul) / memory write
- This is not a perfect metric for real efficiency
 - GPU: 1 global read "should" call tens of arithmetic operations, i.e try to be not 100% I/O-bound

For the dense (matrix-multiplication) approach, the total number of operations is

$$\frac{N_t(N_t + 1)}{2} \cdot N_p$$

Is there an "absolute" (data format independent) minimum number of operations ?

-> Count operations on non-zero elements.

$$\text{numerator}(t_1, t_2) = \sum_p I(t_1, p) I(t_2, p)$$

for a given spatial index p :

$\text{numerator}(t_1, t_2) = I(t_1, p) \cdot I(t_2, p) \neq 0$ iff $I(t_1, p) \neq 0$ and $I(t_2, p) \neq 0$

minimum number of operations for a given tuple (t_1, t_2) :

$$\text{min. nb. ops}(t_1, t_2) = \text{card} \{p, I(t_1, p) \neq 0 \text{ and } I(t_2, p) \neq 0\}$$

Total (minimum) number of operations:

$$\text{min. nb. ops} = \sum_{t_1, t_2} \text{nb. ops}(t_1, t_2)$$

Exchanging the summations, it's easily shown that

$$\boxed{\text{min. nb. ops} = \sum_p \frac{C(p) \cdot (C(p) + 1)}{2}}$$

where

$$C(p) = \sum_t \tilde{I}(t, p)$$

Where $\tilde{I}(t, p) = 1$ iff $I(t, p) \neq 0$, and 0 otherwise

i.e `C = (xpcs_data > 0).sum(axis=0)`.

Which is simply equivalent to computing the TTCF matrix on the *boolean* array `xpcs_data > 0`.

Example: "`dataset01_scan0012`" with 10k frames:

- $2.72 \cdot 10^{14}$ operations for matmul approach: $\frac{N_t^2}{2} \cdot N_p$ (mostly "0 × 0")
- $2.88 \cdot 10^{10}$ operations is the lower bound

4.2 - Space-based approaches

It's desirable to process the data directly in the space-compacted format (the one Lima2 saves to).

Remember that each entry $\text{numerator}(t_1, t_2)$ in the numerator matrix is the dot product $\vec{I}_1 \cdot \vec{I}_2$

where

$$\vec{I}_1 = [I(t_1, p = 0), I(t_1, p = 1), I(t_1, p = 2), \dots]$$

is simply the frame $I(t_1)$.

In other words, if `xpcs_data` is in dense format, the entry (t_1, t_2) is the spatial sum

```
(xpcs_data[t1, :] * xpcs_data[t2, :]).sum()
```

In space-compact format, we have to perform a **sparse dot product** for each entry (t_1, t_2) .

This sparse dot product can be done in complexity $O(K \times n)$ where

- n is the number of non-zero elements in \vec{I}_1
- $K < 10$ in practice

Each entry therefore takes $K \times \text{mean_nnz_space}$ operations (incl. memory read) to be done.

There are $\frac{N_t^2}{2}$ entries to compute ($N_t = \text{n_frames}$)

So in total, a minimum of

$$\frac{N_t^2}{2} \cdot K \cdot \text{mean_nnz_space} \quad \text{operations}$$

same formula as matmul, but N_p becomes $K \cdot \text{mean_nnz_space}$. "The sparser, the better".

Example: "dataset01_scan0012" with 10k frames:

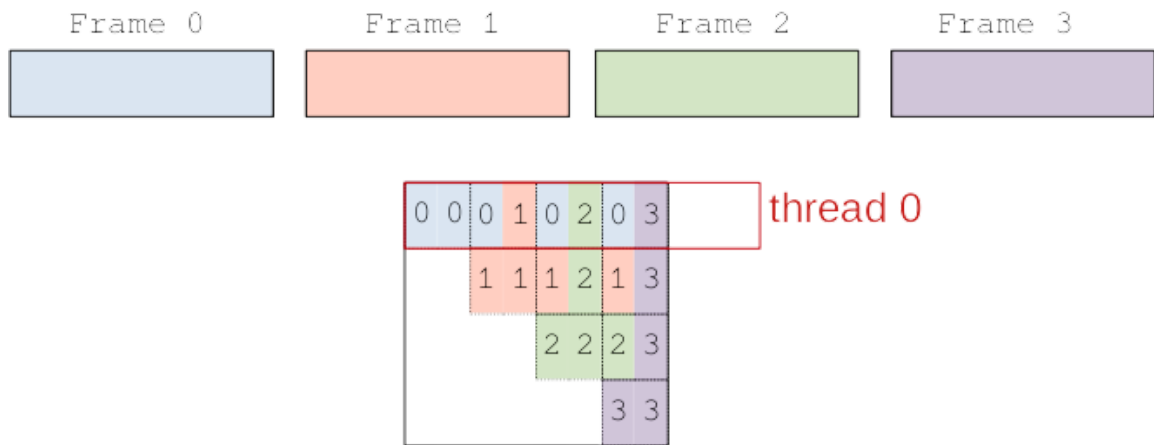
- $2.72 \cdot 10^{14}$ operations for matmul approach: $\frac{N_t^2}{2} \cdot N_p$ (mostly "0 × 0")
- $2.88 \cdot 10^{10}$ operations is the lower bound for all formats
- 6.0×10^{12} operations is the lower bound for space-compact-based approaches (for $K = 4$, probably optimistic)

Parallelization of space-based approaches

- In dynamix, three space-based approaches were tested.
- They differ mainly on how to compute the sparse dot product (parallelization scheme)

Approach 1

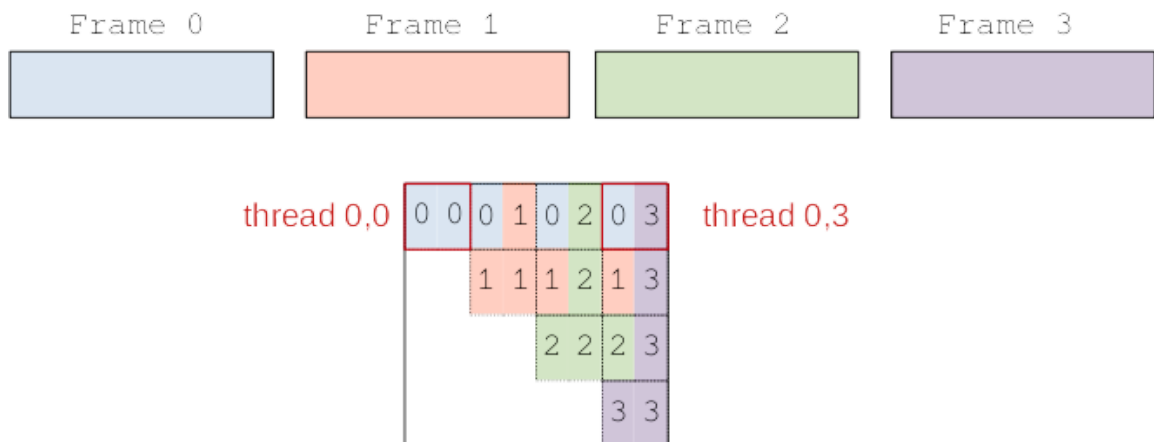
- Launch `n_frames` threads.
- Each thread computes `numerator(t0, t0), numerator(t0, t1), ..., numerator(t0, t_(n_frames-1))`.



Each thread reads (half of) the entire dataset!

Approach 2

- Launch `(n_frames, n_frames)` threads.
- Each thread computes `numerator(t1, t2)`

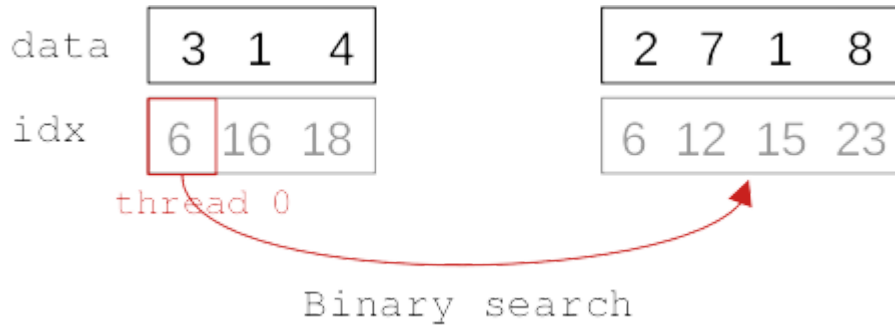


Each frame is read by all threads!

To reduce the I/O burden, we can use "threads groups" and shared memory.

Approach 3

- Launch `(max_time_nnz, n_frames)` threads.
- Each sparse-dot-product is computed *in parallel* by many threads
- Each of these threads does `n_frames / 2 * log2(mean_nnz_space/2)` accesses to the other vectors



Number of operations:

$$\text{mean_nnz_space} \cdot \left(1 + \frac{N_t}{2} \log_2(\text{mean_nnz_space}/2) \right) \cdot \frac{N_t}{2}$$

Example: " dataset01_scan0012 " with 10k frames:

- $2.72 \cdot 10^{14}$ operations for matmul approach: $\frac{N_t^2}{2} \cdot N_p$ (mostly "0 × 0")
- $2.88 \cdot 10^{10}$ operations is the lower bound for all formats
- 6.0×10^{12} operations is the lower bound for space-based approaches
- 27.0×10^{12} operations in total for approach 3

but in practice **v3 is the most efficient** of space-based approach, by a large margin.

Tentative explanation

	Unfriendly reads	Friendly reads	Arithmetic	Unfriendly writes	Friendly writes	Threads launched
v1	$\mu_s \cdot \frac{N_t}{2}$	0	$\mu_s \cdot N_t/2$	0	$N_t/2$	$N_t \simeq 10^4$
v2	μ_s	0	μ_s	0	1	$N_t \times N_t \simeq 10^8$
v3	$\frac{N_t}{2} \cdot \log_2\left(\frac{\mu_s}{2}\right)$	$N_t/2$	$\mu_s \cdot N_t/2$	$N_t/2$	0	$\frac{\text{max_nnz_space}}{\simeq 4 \cdot 10^4}$

$\mu_s = \text{mean_nnz_space} = \text{np.mean(np.diff(offset_space))}$ ($39 \cdot 10^4$ in the previous example)

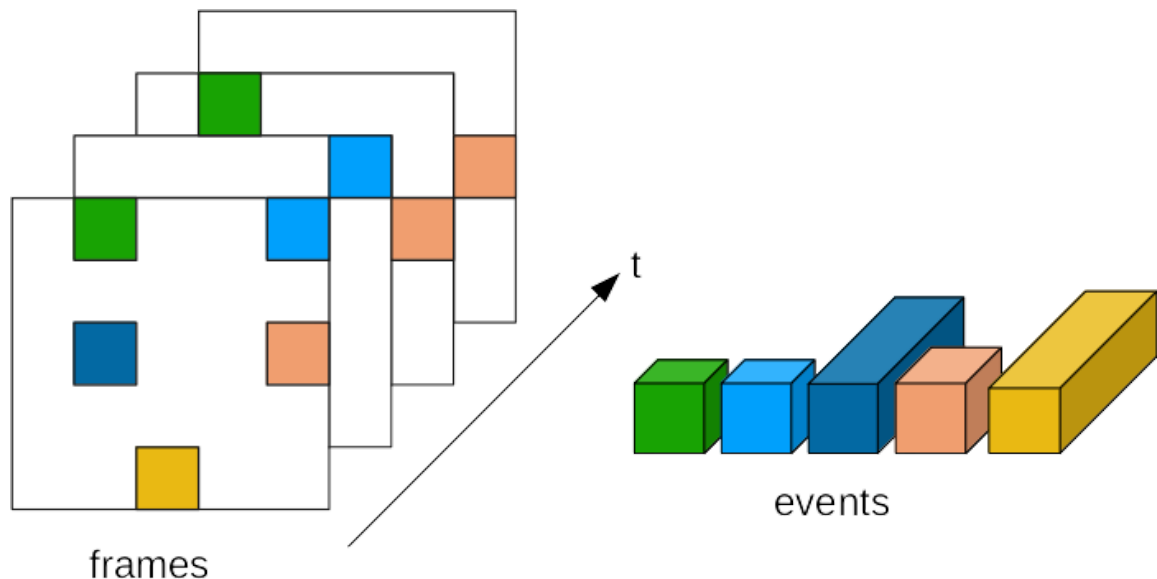
- In (v1), there are probably too many unfriendly reads (each threads reads half the total dataset in a non-coalesced pattern)
- In (v2), too many threads are doing unfriendly reads
- In (v3), a kind of balance is found. What saves the day is probably the `atomic_add` performances

Tentative explanation (2)

Sparse-dot-product is too slow if done serially (v1 and v2).

- Even sparse datasets still have 10^2 to 10^4 non-zero items in each frame
- C implementation takes 600 us on Epyc 7543 for 40k items
- Even if sparse-dot-product takes 1 us in average, there are $N_t^2/2$ such products => 5000 seconds on a 100k-frames dataset.
- Space-compact-based approach bring some speed-up wrt basic matmul implementation
- ... still, not fast enough
- For computing $g_2(\tau)$, a time-compact-based approach was successfully used in 2019
- Let's see how it performs here

4.3 - Time-based approach



If the data is natively in time-compact format:

- Parallelize over spatial indices, i.e launch N_p threads
- Each thread p browses $I(t_1, t_2)$ and accumulates into `corr_matrix[t1, t2]`

	Unfriendly reads	Friendly reads	Arithmetic	Unfriendly writes	Friendly writes	Threads launched
v1	$\mu_s \cdot \frac{N_t}{2}$	0	$\mu_s \cdot N_t/2$	0	$N_t/2$	$N_t \simeq 10^4$

	Unfriendly reads	Friendly reads	Arithmetic	Unfriendly writes	Friendly writes	Threads launched
v2	μ_s	0	μ_s	0	1	$N_t \times N_t \simeq 10^8$
v3	$\frac{N_t}{2} \cdot \log_2\left(\frac{\mu_s}{2}\right)$	$N_t/2$	$\mu_s \cdot N_t/2$	$N_t/2$	0	$\frac{\text{max_nnz_space}}{\simeq 4 \cdot 10^4}$
v4	0	μ_t	μ_t^2	μ_t^2	0	$N_x \times N_y \simeq 10^6$

$\mu_t = \text{mean_nnz_time} = \text{np.mean(np.diff(offsets_time))}$ (90 for previous example)

The total number of operations is... almost the lowest possible.

```
dt = np.diff(offsets_time)
total_nb_ops_v4 = (dt * (dt+1)/2).sum() # 2.88e10
```

By design, the Time-based approach uses (close to) the minimum number of operations.

Example: " dataset01_scan0012 " with 10k frames:

- $2.72 \cdot 10^{14}$ operations for matmul approach: $\frac{N_t^2}{2} \cdot N_p$ (mostly "0 × 0")
- $2.88 \cdot 10^{10}$ operations is the lower bound for all formats
- 6.0×10^{12} operations is the lower bound for space-based approaches
- 2.70×10^{13} operations in total for approach 3
- 2.88×10^{10} operations for approach 4 (time-compact-based)

4.4 - Performance measurements

dataset	n_frames	time(sp-compact)	time(t-compact)	Sparsity factor
01_07	20k	10.3 min	3.6 min	f=52
01_08	20k	4.8 min	57 s	f=103
01_09	20k	2.3 min	15 s	f=200
01_10	20k	88 s	7 s	f=300
01_11	20k	4.3 mins	47 s	f=113
01_12	20k	4.2 mins	45 s	f=116
01_13	20k	7 s	82 ms	f=2900
02	10k	2.5 s	21 s	f=290, SAXS
03	40k	3.5 mins	12 s	f=450
04	100k	88 s	200 ms	f=8000
05	100k	10.5 mins	13.3 s	f=920

All dense (matmul-based) correlators takes several minutes (10k-20k) to more than 1h (100k).

From 100k frames, dense correlator is too demanding in terms of memory:

$$100e3 \cdot 4e6 \cdot 4 = 1.6 \text{ TB just for the data}$$

Example: " dataset04 " with 100k frames:

- $\frac{N_t^2}{2} \cdot N_p = 1.9 \cdot 10^{16}$ operations for matmul approach
- $\sum_p \frac{C(p)(C(p)+1)}{2} = 4.9 \cdot 10^8$ operations is the lower bound for all formats ... almost reached for approach (4)
- $\frac{N_t^2}{2} K\mu_s = 11.3 \cdot 10^{12}$ operations is the lower bound for space-based approaches
- $\mu_s \cdot \left(1 + \frac{N_t}{2} \log_2(\mu_s/2)\right) \cdot \frac{N_t}{2} = 11.5 \cdot 10^{12}$ operations in total for approach (3)

Profiling on dataset01_scan0012 with **20k** frames:

Time-compact based approach ("v4"):

- Read space-compacted data: 8.7 s
- Space-compact to time-compact conversion: 3.0 s
 - 2.5 s offsets calculation (CPU/bincount)
 - 0.3 s GPU conversion
- GPU calculation of $g_2(t_1, t_2)$: 25.6 s
 - 22.0 s numerator
 - 3.6 s denominator
- Final normalization + $g_2(\tau)$ + STD: 39 ms

Matmul approach (numpy/BLAS):

- $276 \cdot 3 = 828 \text{ s}$ (276 s for each q-vector)

Conclusion, outlooks

- dynamix : python package to compute $g_2(\tau)$ and $\text{TTCF}(t_1, t_2)$ efficiently directly on compacted data
- Calculation is faster on time-compacted data
- Future works: re-binning of time indices (for n_frames > 100k), on-line computation