Towards a Streaming Algorithm for AGIPD Calibration and Photon Counting

Felix Brausse
Postdoc @ MID

Data-Analysis Satellite Meeting
Hamburg, Jan. 25th, 2022

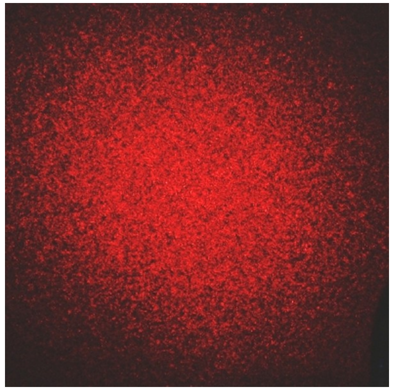
Case Study: XPCS Experiments

- EuXFEL Instruments cover a vast range of experiments with widely varying conditions

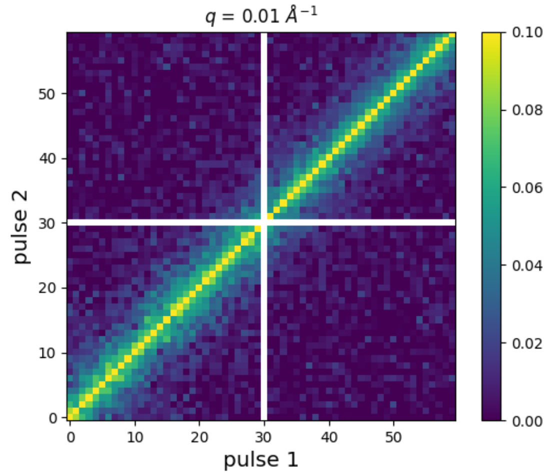
  → For certain operation conditions, we may need more tailored tools

- Take XPCS: X-Ray Photon Correlation Spectroscopy
  - Based on **Speckle**
  - Speckle patterns change **stochastically** with dynamics
  - Autocorrelation reveals the driving dynamics

- Characteristic conditions:
  - Very often *very* low count rates, ~10^{-2} ph/pix/pulse; → many runs for good statistics
  - Analysis requires **temporal autocorrelation** function



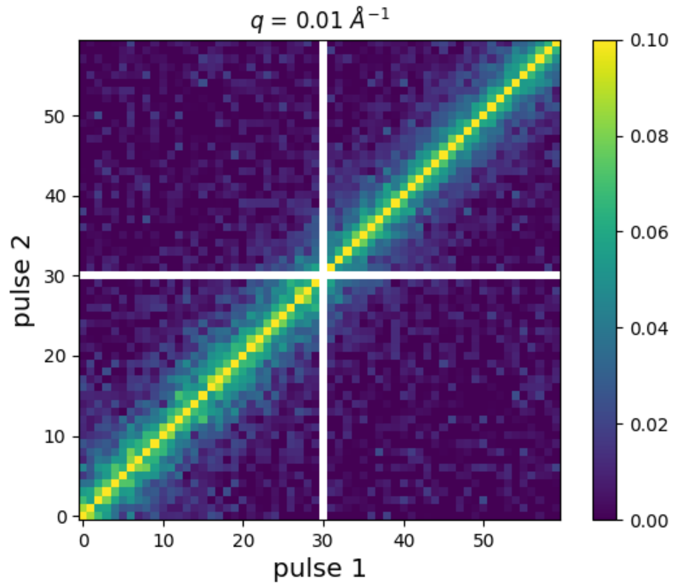
Laser-pointer speckle

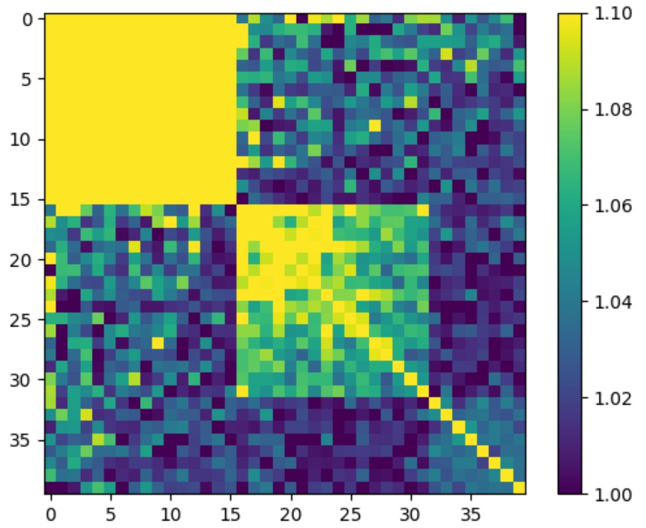


Two-time correlation function

Missed Opportunities in the Current Process

- A) The Calibration Pipeline for AGIPD is very good, but it
  - i) General Purpose, based on batch processing and calibration management
  - ii) Struggles with some quirks of the detector, like flickering baselines
  - iii) Resource intense
  - iv) Stores **dense** *proc* data (compression ratio: 1) when the number of zeros ~(1-λ)
- B) We currently cannot get temporal correlations live





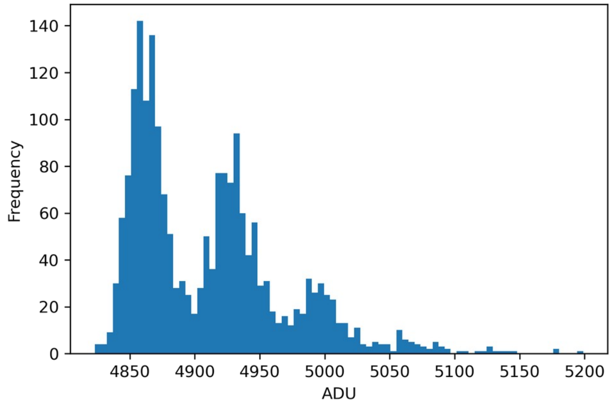
Ad 1) Wobbling Baselines

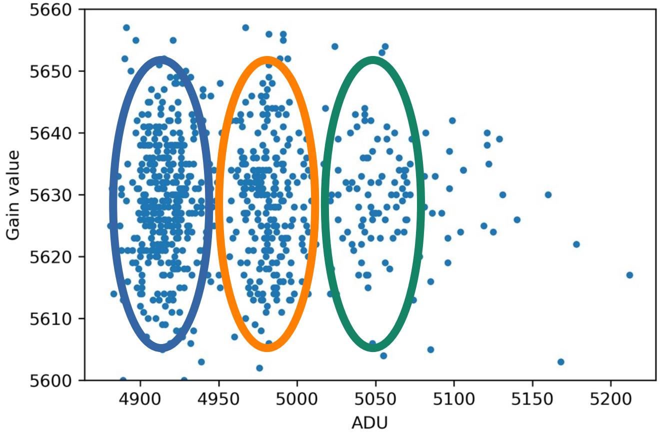
What does AGIPD Raw Data look like at Low Intensities?

- Single-photon resolution up to *k*~4 or so
- "Every statistical model is wrong". Here is how I get it wrong.
  - Convolution: Poisson * Gaussian
  - → Everything from exponential family
  - Gain *a*, offset *b*, count rate λ, (fixed) width σ

$$F(x,a,b,\lambda,\sigma)=\sum_k N*P(\lambda,k)*\exp\left\{-\left(\frac{x-(ak+b)}{a\sigma}\right)^2\right\}$$

$$P(\lambda,k)=\frac{\lambda^k}{k!}e^{-\lambda}$$





0 1 2 photons

How Do We Fit the Model (without Histograms)?

- The Maximum-Likelihood principle tells us how; Maximize the likelihood function: $L(\boldsymbol{\theta}, \mathbf{x}) = \prod_i^n F(x_i, a, b, \lambda, \sigma)$
- Requires gradients and many iterations.
- Even better: Expectation Maximization
  - Idea: There must be a latent variable; here: photon count *k*
- We can decompose $F(x_i, a, b, \lambda, \sigma) = \sum_k f_k(\boldsymbol{\theta}, x_i)$ to get the likelihood function $L(\boldsymbol{\theta}, \mathbf{x}, \mathbf{k}) = \prod_i^n \prod_k f_k(\boldsymbol{\theta}, \mathbf{x})$

How Do We Fit the Model (without Histograms)?

- Now we can optimize each component individually; using the weights

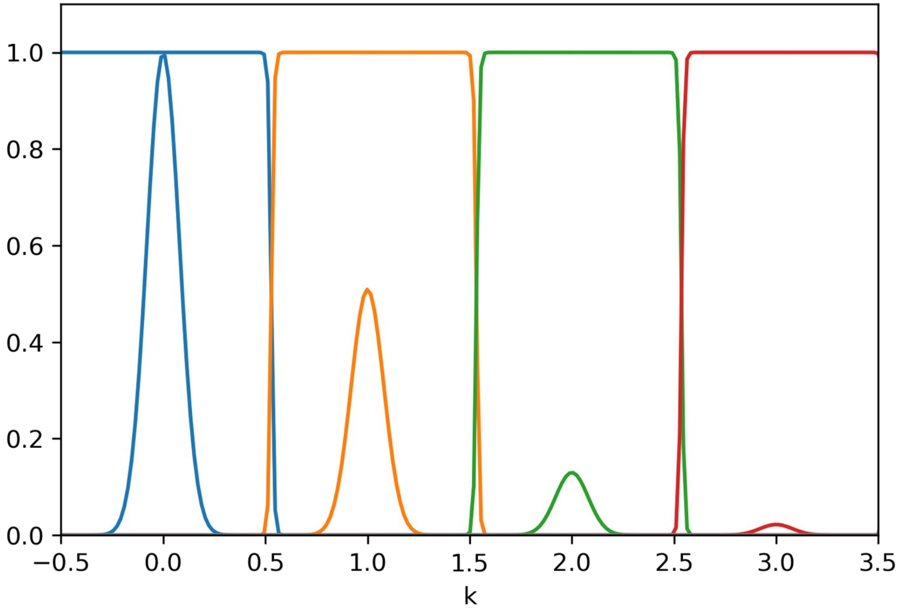
  $$T_{i,k} = \frac{f_k(\boldsymbol{\theta}, x_i)}{\sum_k f_k(\boldsymbol{\theta}, x_i)}$$

  we can use the Maximum-Likelihood update for a Gaussian: **the mean** $\mu_k^{(t+1)} = \frac{\sum_i T_{i,k} x_i}{\sum_i T_{i,k}}$

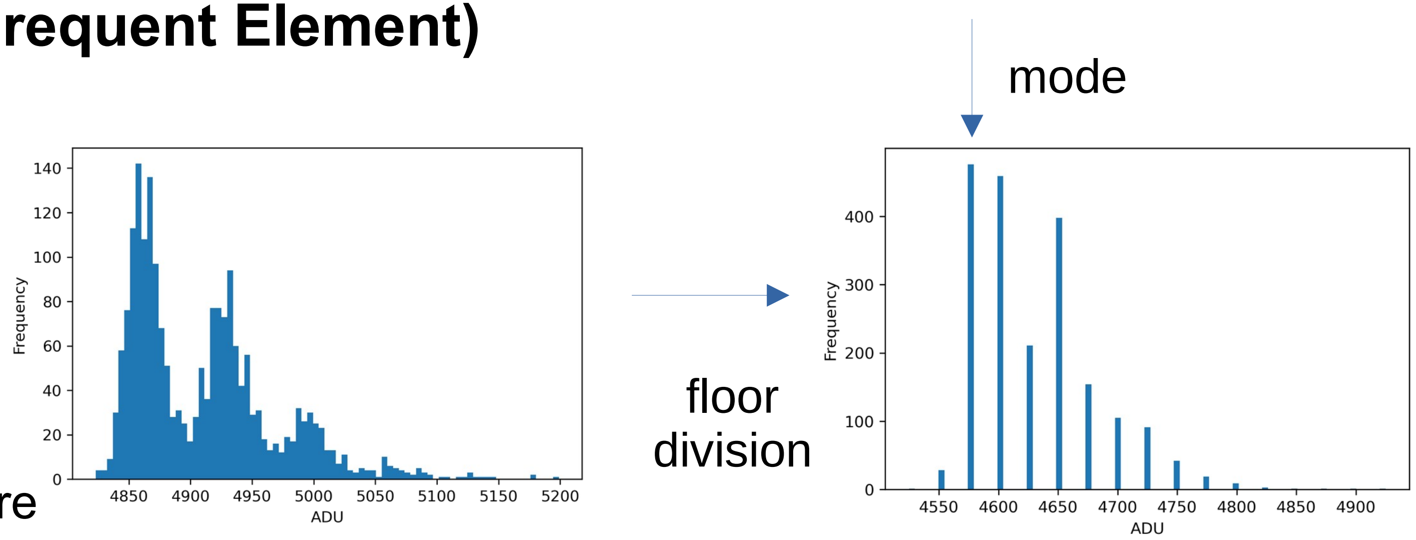
  (this looks like a *softmax*!)

  (this is the *k*-means algorithm!)

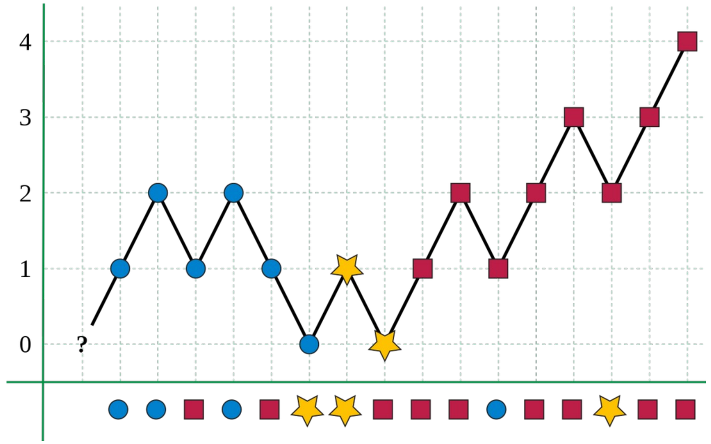
- Simplify even further: set $\max_k \{T_{i,k}\}$ to 1, the rest to 0
  - → "The-Winner-Takes-It-All" Flavor



Problem: We Need a Starting Guess
Solution: Look at the Mode (Most Frequent Element)



- For λ < 1 the mode is always 0.
- For *n* data points we have to reserve *n* memories + *n* counters *worst case*
- Special case of 1 memory: the Boyer-Moore Majority-Vote Algorithm



- → Extend majority vote to *k* memories
- SpaceSaving(k) works really well for *k* = 2

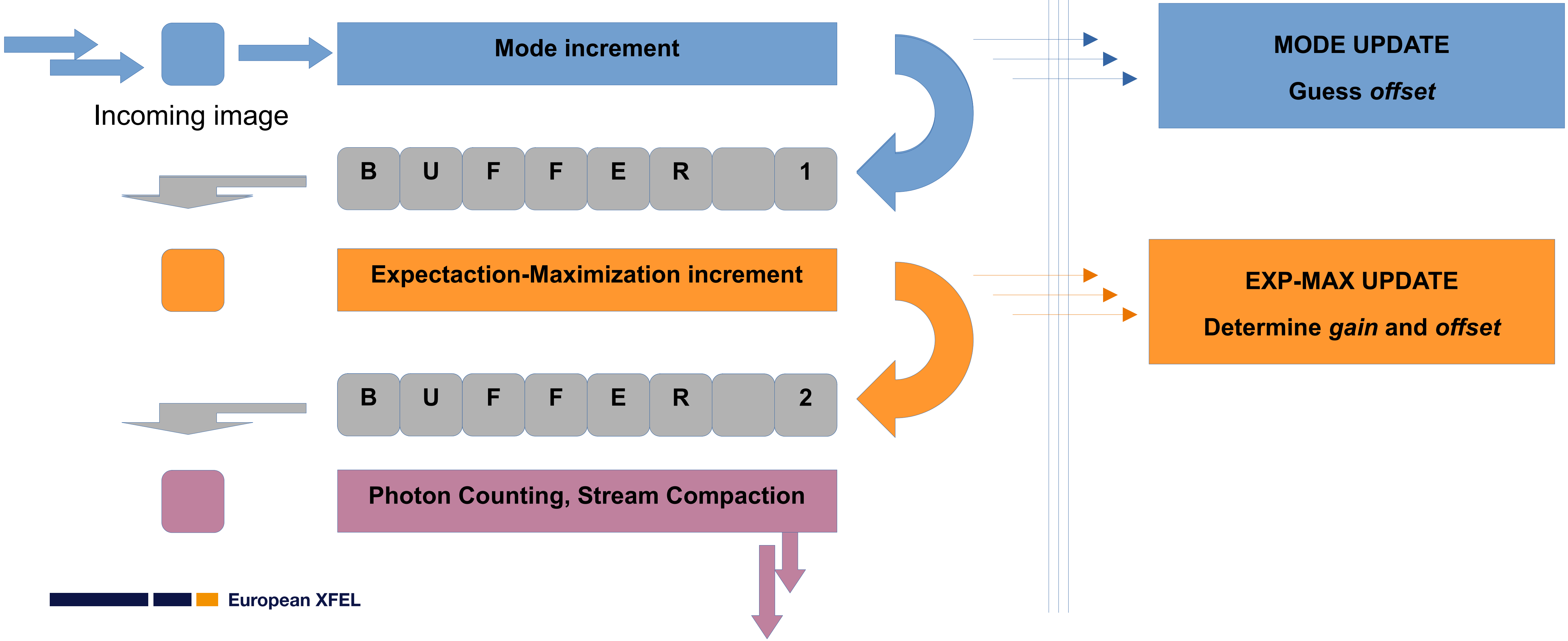
Algorithm 3: SpaceSaving(k)

$n \leftarrow 0$;
$T \leftarrow \emptyset$;
foreach i **do**
  $n \leftarrow n + 1$;
  if $i \in T$ **then** $c_i \leftarrow c_i + 1$;
  else if $|T| < k$ **then**
    $T \leftarrow T \cup \{i\}$;
    $c_i \leftarrow 1$;
  else
    $j \leftarrow \arg\min_{j \in T} c_j$;
    $c_i \leftarrow c_j + 1$;
    $T \leftarrow T \cup \{i\} \setminus \{j\}$;

Putting it all together

For each train:

Every 20 trains:



Let Us Look At Some Code

- Iteration of the SpaceSaving(2)

```python
import cupy as cp

def iterMajority(im, args):

    val1, val2, cnt1, cnt2 = args

    im = im // 25

    tst1 = im == val1
    tst2 = im == val2

    cnt1 = cp.where(tst1, cnt1+1, cnt1)
    cnt2 = cp.where(tst2, cnt2+1, cnt2)

    argmin =   cnt1 < cnt2
    updMsk = ~(tst1 | tst2)

    val1 = cp.where(updMsk &  argmin,     im, val1)
    cnt1 = cp.where(updMsk &  argmin, cnt1+1, cnt1)

    val2 = cp.where(updMsk & ~argmin,     im, val2)
    cnt2 = cp.where(updMsk & ~argmin, cnt2+1, cnt2)

    return val1, val2, cnt1, cnt2

```

```python
itMajBits_kernel( (np.prod(img.shape) >> 10,),   # no. of blocks
                  (2**10,),                      # no. of threads
                  (img, val, cnt) )              # arguments
```

```c
extern "C" __global__
void iterMajorBits(unsigned short* im, unsigned int* val, unsigned int* cnt) {

    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    char cs  = 0;

    short imCast = im[tid] >> 4;
    unsigned int tmpCnt = cnt[tid], tmpVal = val[tid];

    short c1 = tmpCnt >>     16;
    short c2 = tmpCnt  & 0xFFFF;
    short v1 = tmpVal >>     16;
    short v2 = tmpVal  & 0xFFFF;

    cs |= (v1 == imCast) << 0;
```

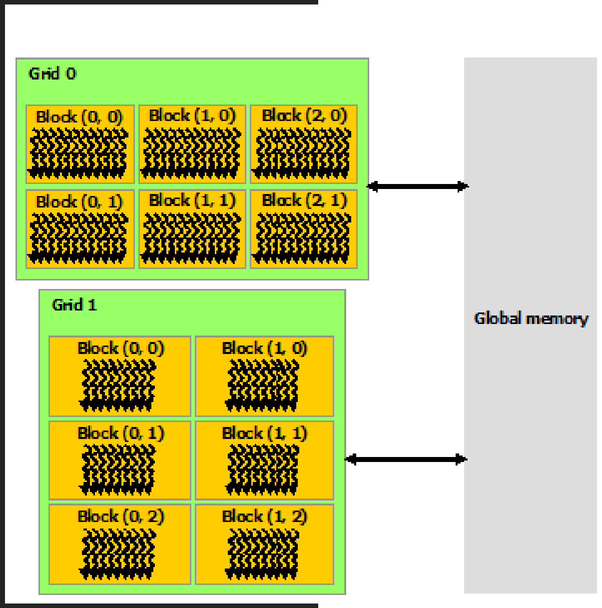
~20x faster

```c
            c1 += 1;
            cnt[tid] = (tmpCnt & 0x0000FFFF) | (c1 << 16);
            break;

        case 2:
            c2 += 1;
            cnt[tid] = (tmpCnt & 0xFFFF0000) | c2;
            break;

        case 4:
            c1 += 1;
            cnt[tid] = (tmpCnt & 0x0000FFFF) | (c1 << 16);
            val[tid] = (tmpVal & 0x0000FFFF) | (v1 << 16);
            break;

        case 0:
            c2 += 1;
            cnt[tid] = (tmpCnt & 0xFFFF0000) | c2;
            val[tid] = (tmpVal & 0xFFFF0000) | v2;
            break;
    }
}
```

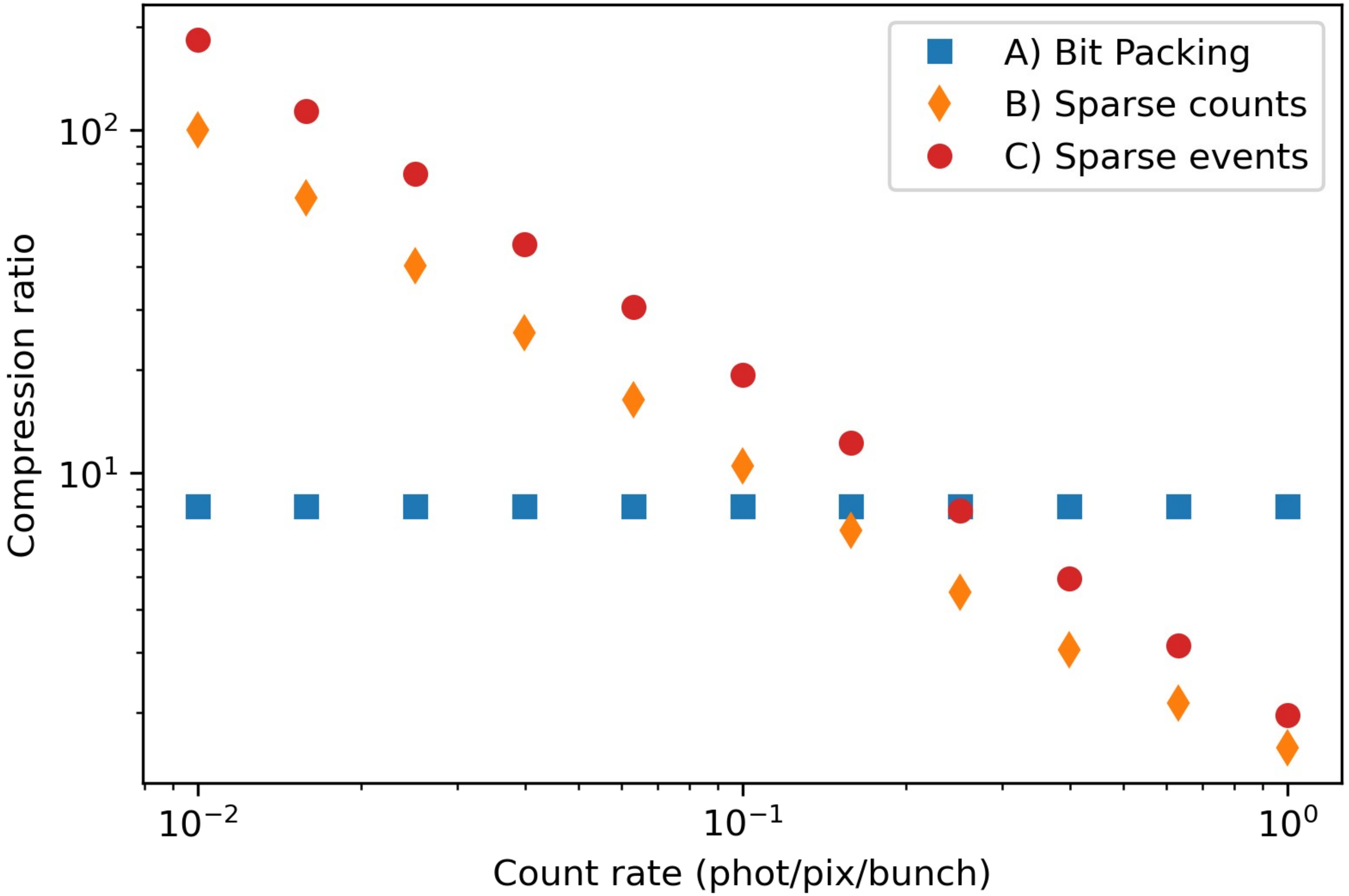


Let's Look at It in Action

- (video)
- Overall performance: Mode and Exp-Max ~300Hz
- Memory bandwidth → ~100 Hz
- Reading from storage → 30 Hz max

Ad 2) Storage Requirements

AGIPD Data Compression



- Let's skip algorithmic compression
- Raw data: 2x 16 bit / pix (intensity + gain)
- Proc data: 32 bit float
- Strategy A: Bitpack into

  8 x 4 bit (ratio 8)
- Strategy B: Sparse pixels (index + count)
- **Strategy C: Sparse events (16bit index only)**
- B and C can be implemented efficiently through **prefix sums**
- B and C offer savings in sparse algorithms

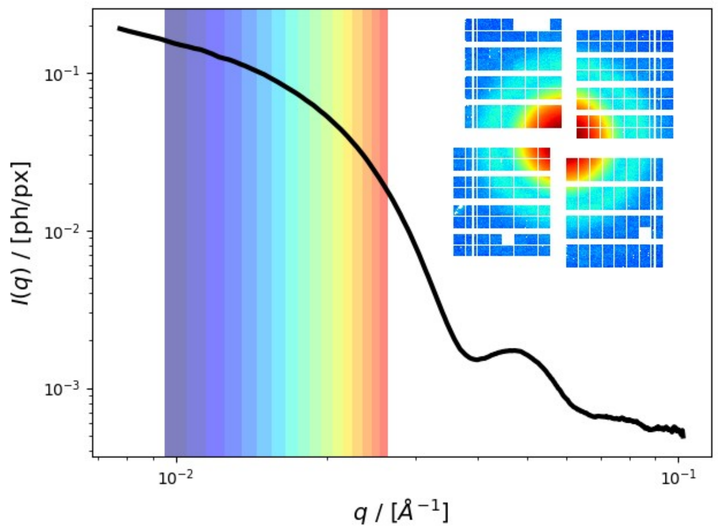
Ad 3) Online Analysis

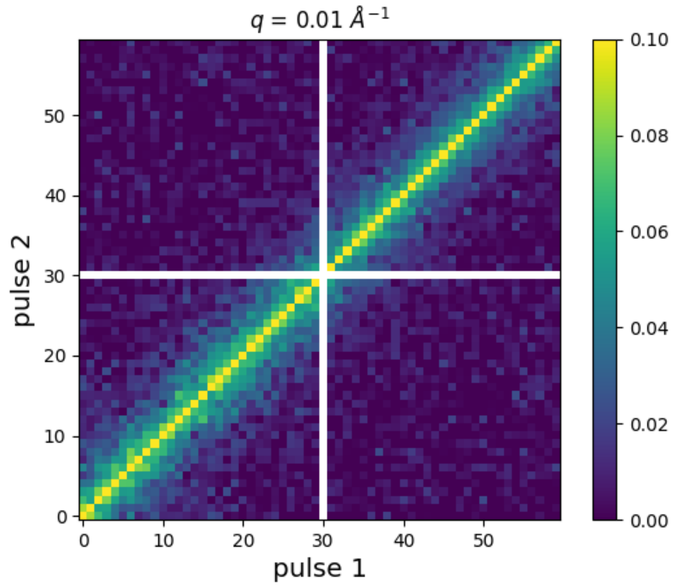
Results for Silica Particles

- Proper analysis: Whole AGIPD geometry, define circles of like *q*
- Calculate Two-Time Correlation Function from Outer Product

$$C_{i,j} = A_i * A_j$$

- Super efficient with `einsum`, even better with cuTensor
- BUT also extremely fast on multi-core CPU (OpenMP)
- (video)





All analysis by Johannes Möller

Summary

With the protoype of a streaming data-processing pipeline, three issues were addressed

- Shifting and flickering baselines in the AGIPD calibration
- Suppression of zeros
- On-line calculation of correlation
- 10 Hz (live) performance is easily within reach
- Depending on features >100Hz operation possible

GPUs can do a wonderful job for us!