Performance of ELT-size AO RTC on GPUs within the framework of DARC

Urban Bitenc
Durham University

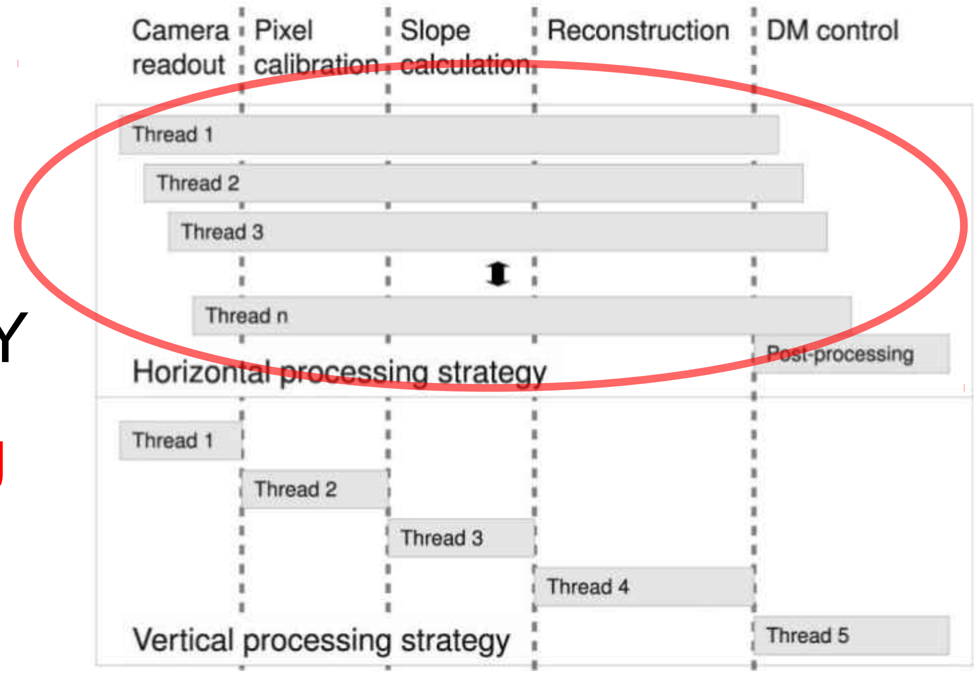
Workshop on Real-Time Control for Adaptice Optics, 4th edition
2016 Dec 20

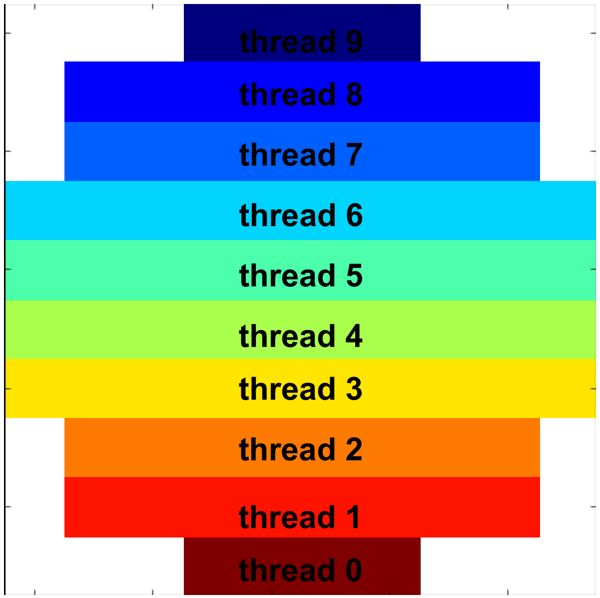
Contents

- Implement the AO RTC pipeline on GPUs within DARC
- Answer some questions:
  - How fast does it run on a GPU?
  - How much jitter?
  - How fast does it run on two, three, four GPUs?
  - Understand the limitations, the bottleneck(s)

Durham AO Real-Time Controller

- Developed by Alastair Basden
- Used on-sky with CANARY
- Key: horizontal processing strategy - make use of pipelineability
  - Process chunks of data as soon as they arrive



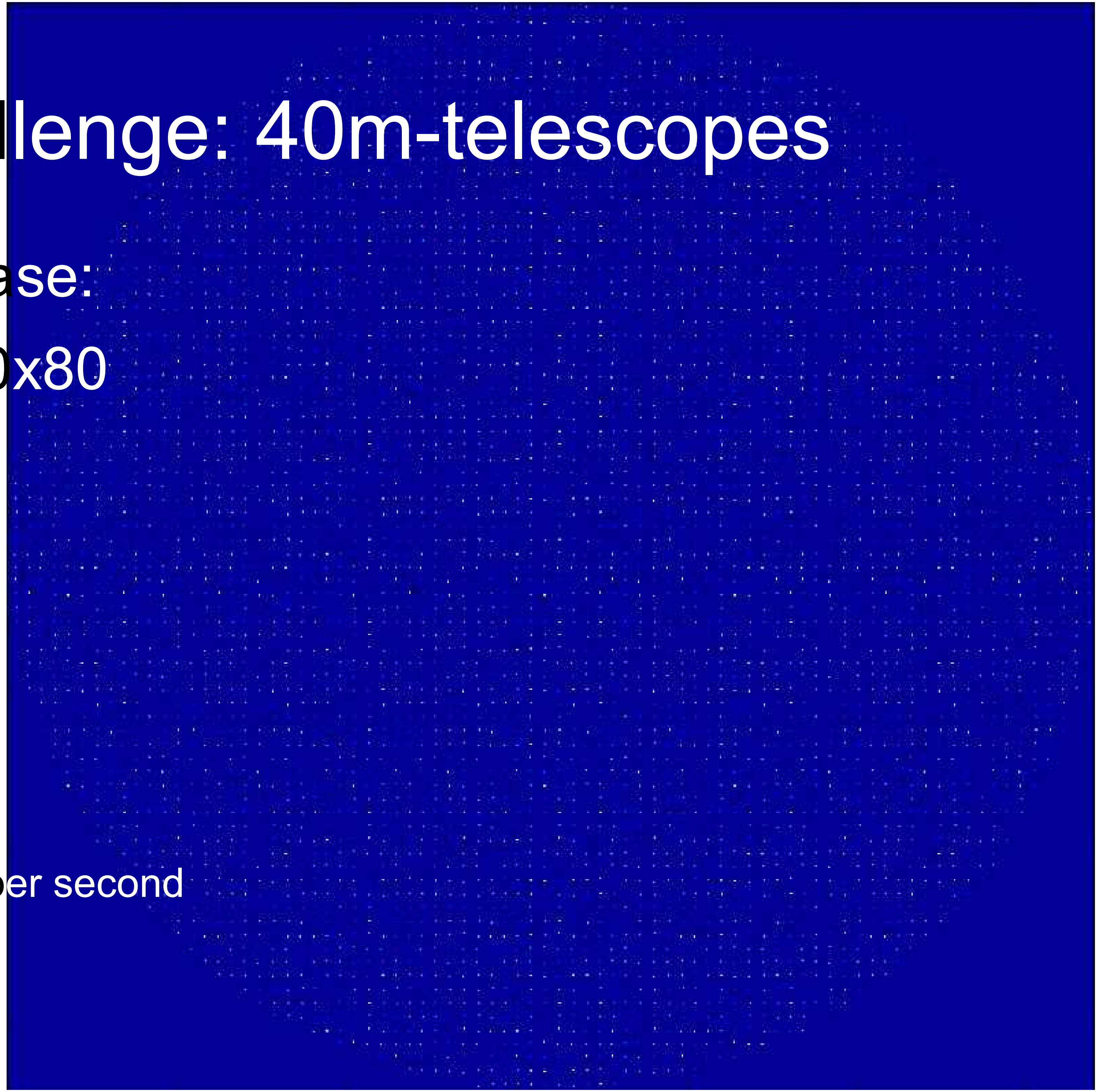


The challenge: 40m-telescopes

My test case:

- SCAO, 80x80
- 16x16 pixels
- Matrix-vector multiplication: 9248 x 4828

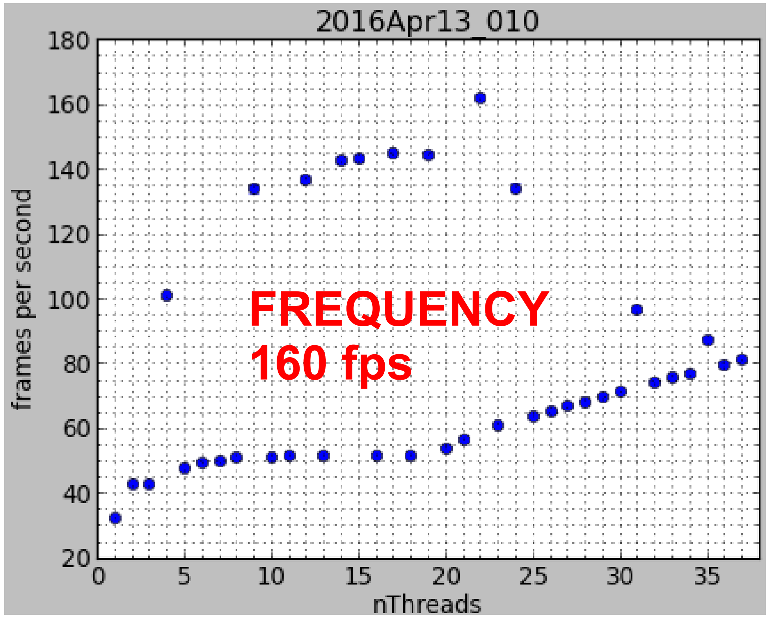
- Goal: process 1000 frames per second

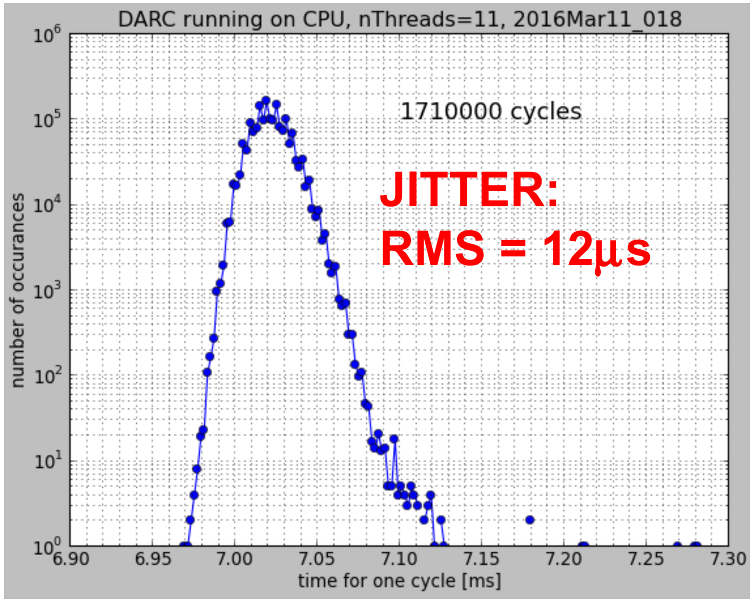


Without acceleration hardware

- Running DARC on a CPU:
  - up to 160 frames per second: too slow

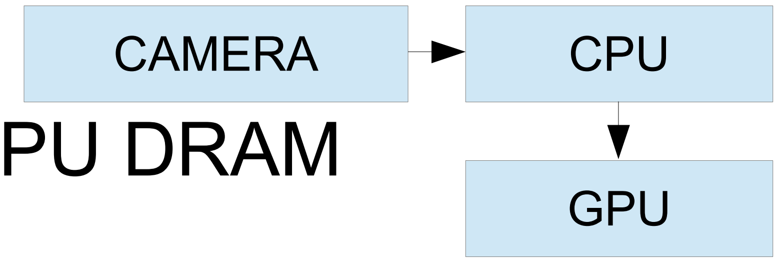
    ==> accelerate using GPUs



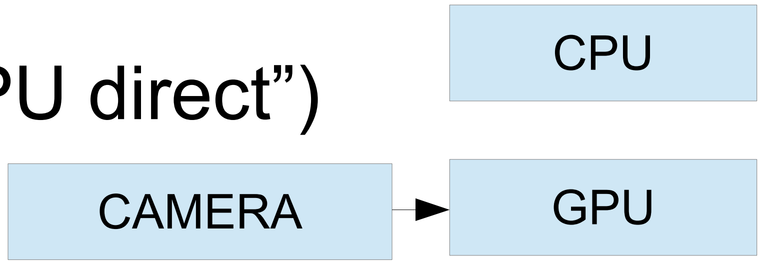


To use GPU: copy pixel data

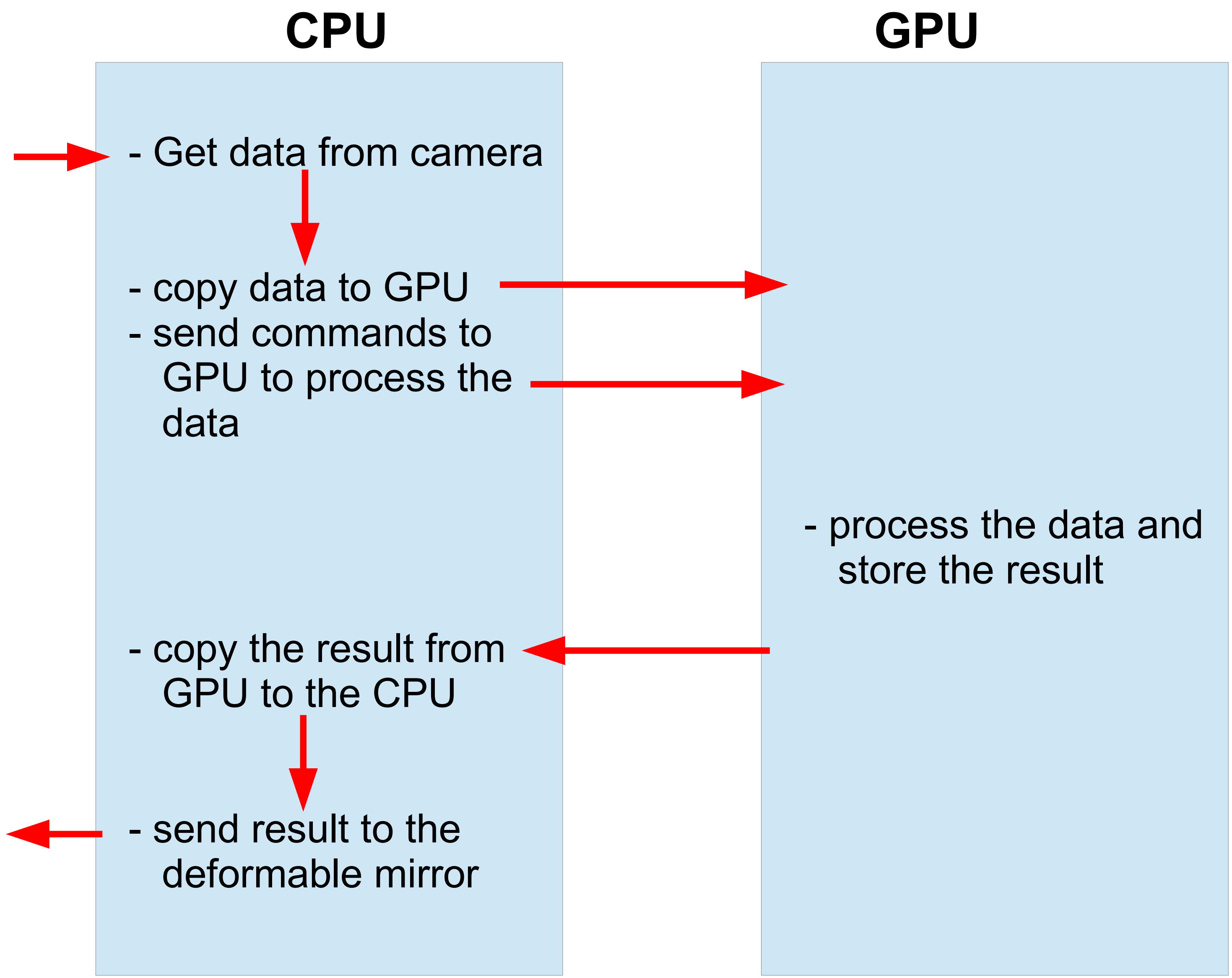
- Standard way:
  - camera --> CPU DRAM --> GPU DRAM
  - advantage: simpler
  - disadvantage: slower, more jitter



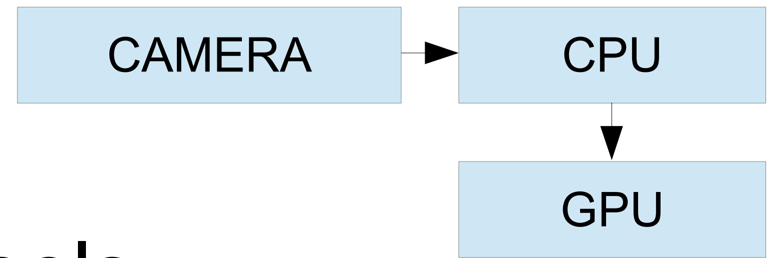
- Ideal way:
  - camera --> GPU DRAM   ("GPU direct")
  - advantage: faster, less jitter
  - disadvantage: no commercial solution available (Do It Yourself)



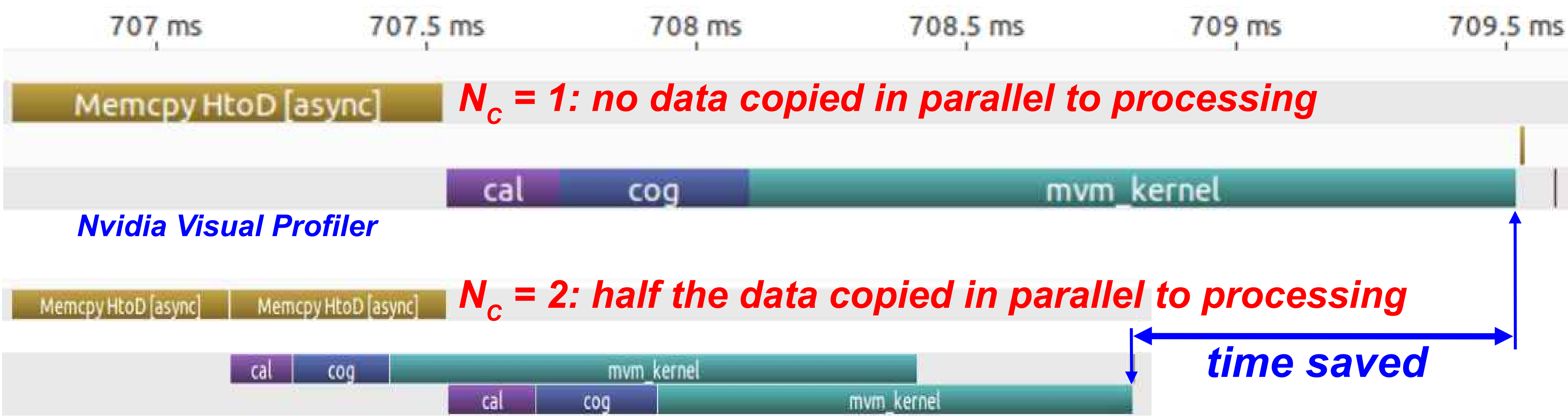
Using only standard CUDA tools



Copying pixel data



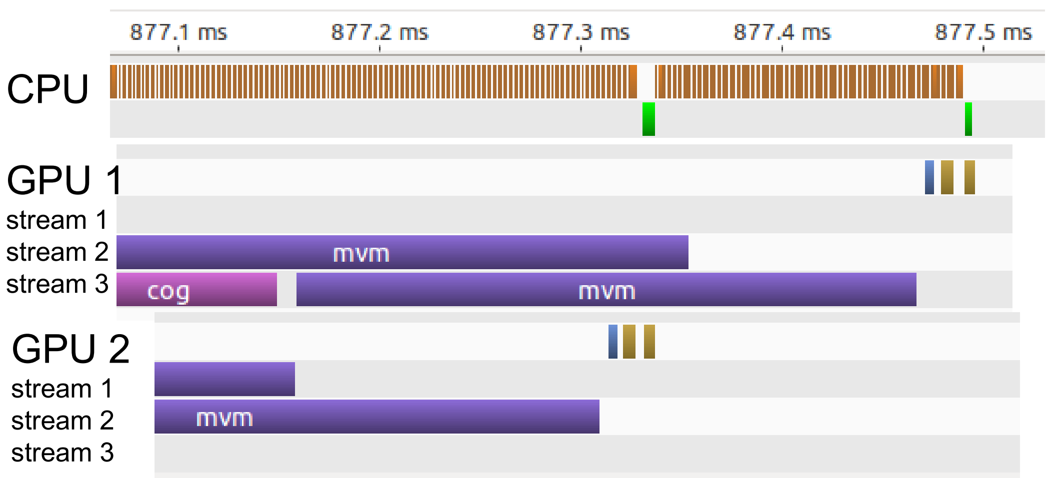
- Use only standard CUDA tools
- Process the pixel data in parallel to copying



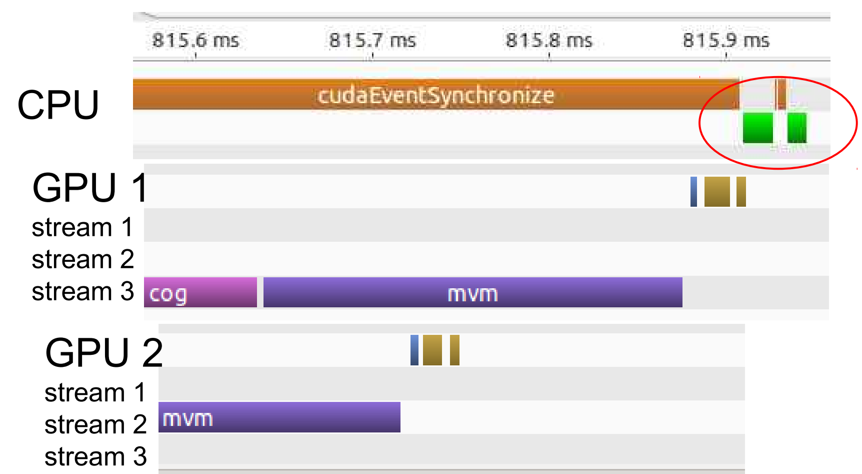
***N_c*: number of data chunks**

Running on several GPUs

- Synchronisation in the end



GPU Synchronization using **cudaEventQuery**



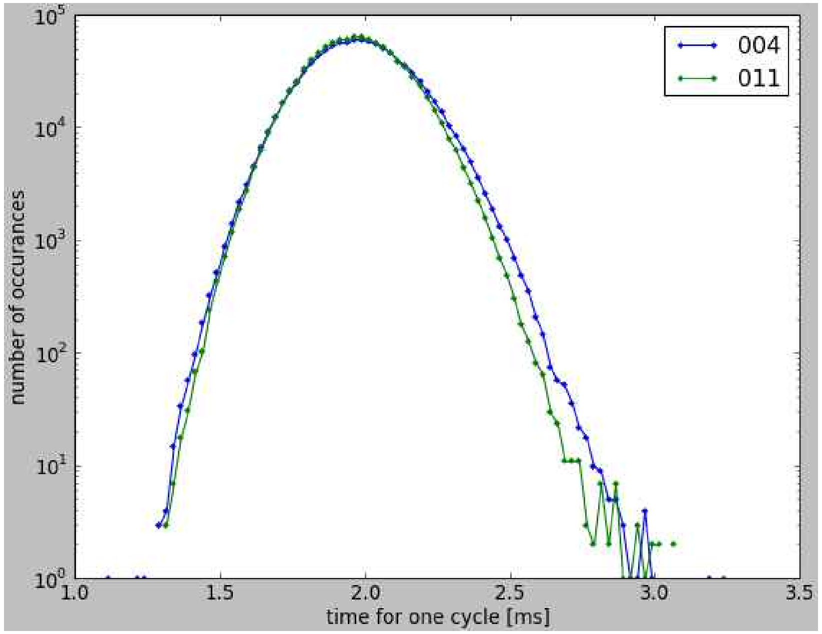
Synchronization using **cudaEventSynchronize**
SLOWER

- Nvidia Visual Profiler was very helpful

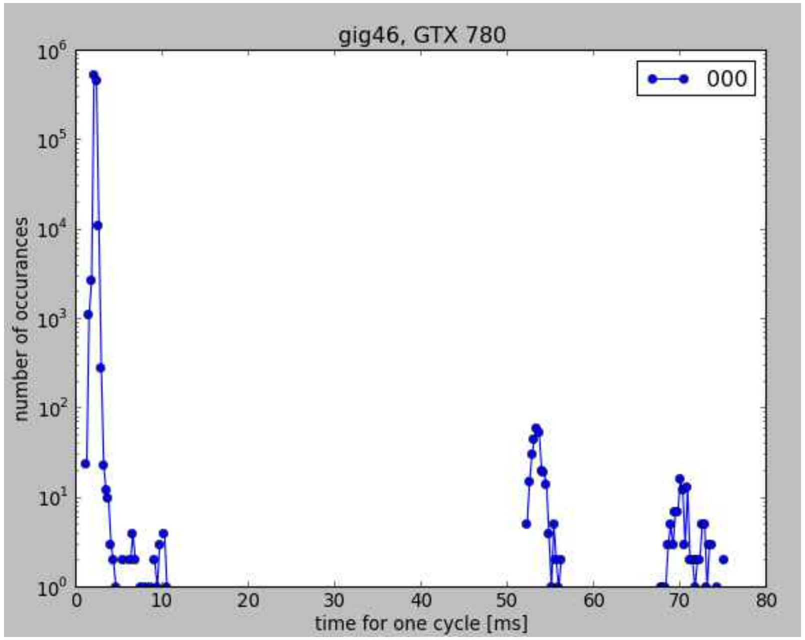
Reduce jitter (1)

- Linux kernel: use ***lowlatency*** kernel, not *generic*

lowlatency kernel,
GTX 580
NO OUTLIERS



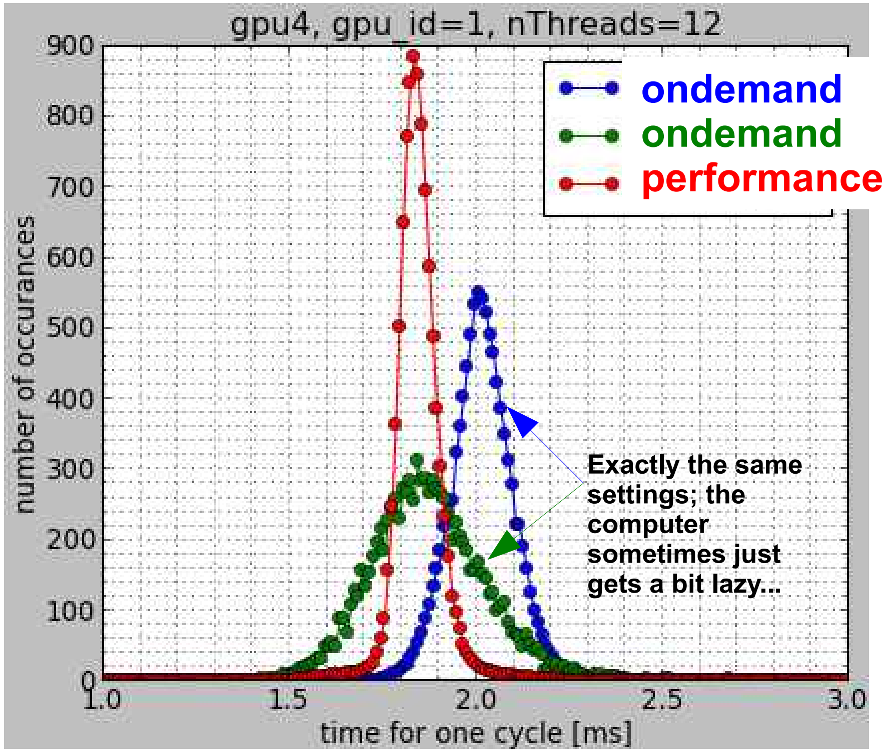
generic kernel,
GTX 780
SEVERAL LARGE OUTLIERS



Reduce Jitter (2)

- Switch off power saving of the CPU:

  cpu frequency scaling_governor = **performance**, not ondemand



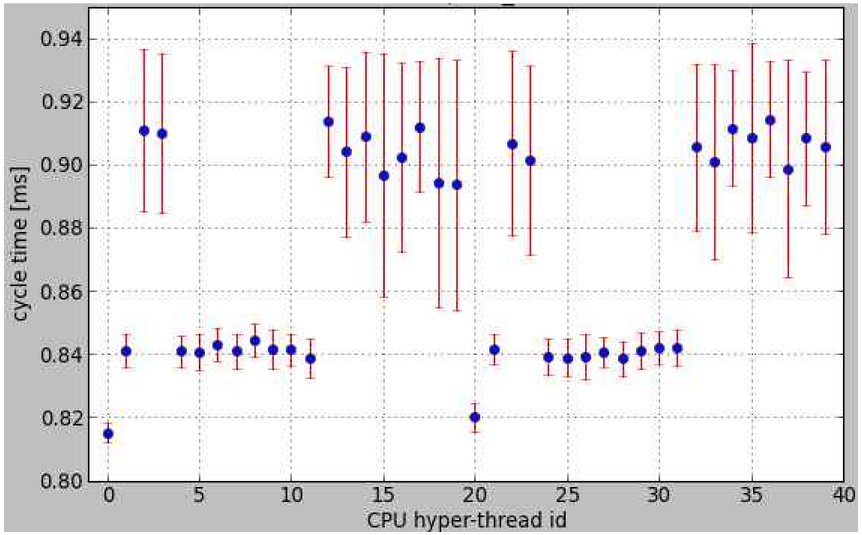
Jitter (3)

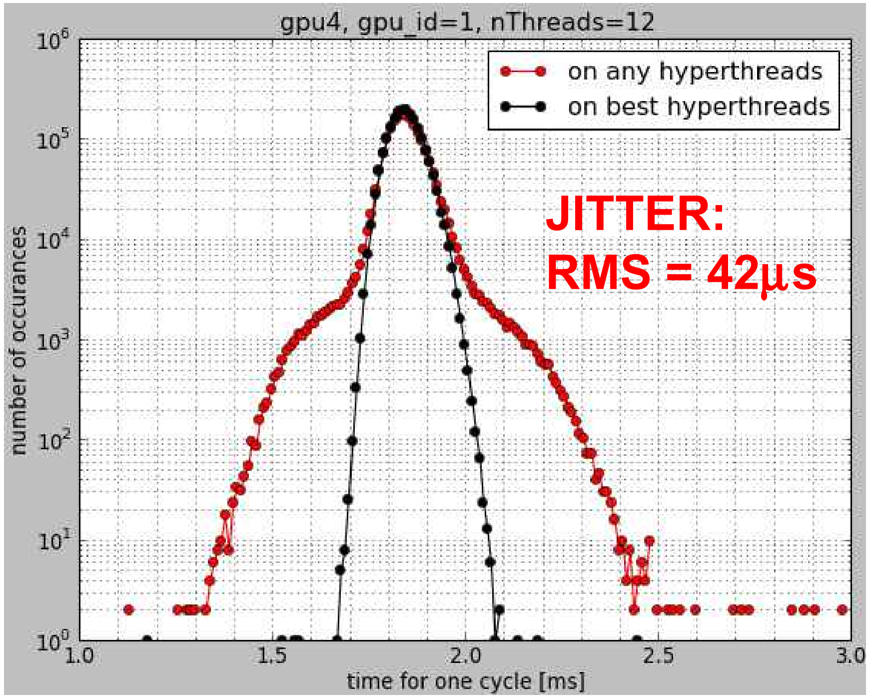
- Lock the threads onto the right hyper-threads

  The CPU has 20 cores, 40 hyper-threads; some are better connected to the GPU than others.

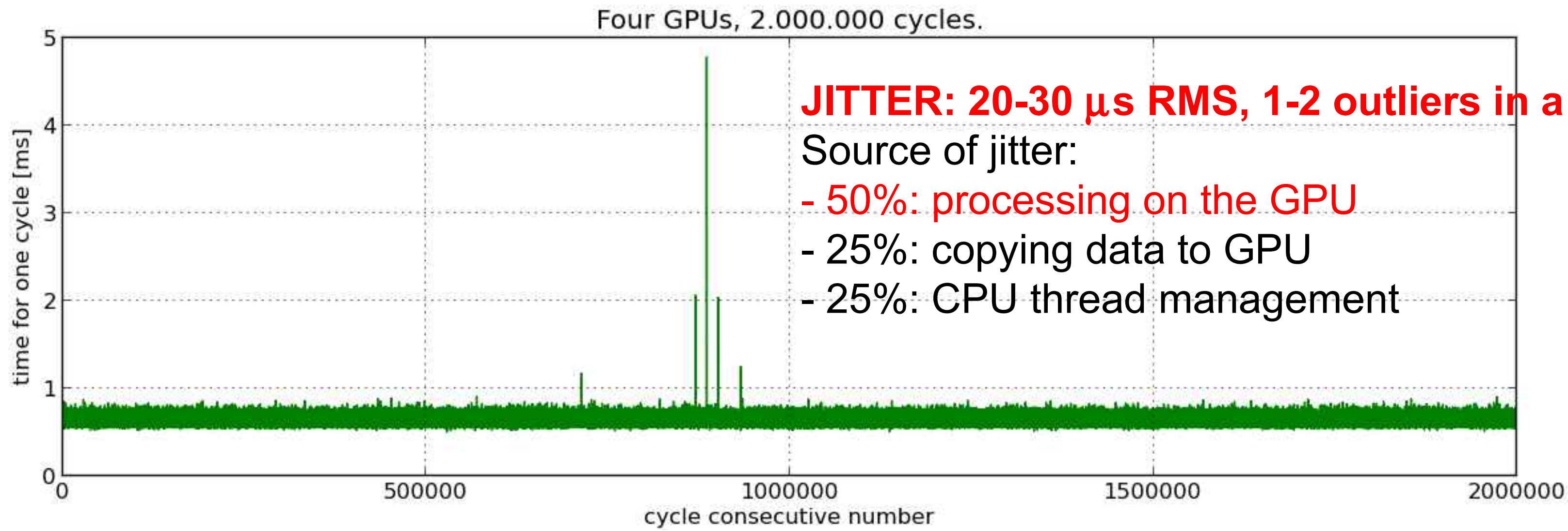
- Contributions to jitter:
  - CPU (organizing threads, launching kernels): **9 µs**
  - copy pixels to GPU: **11 µs**
  - GPU processing (kernel execution): **22 µs**
  - Note: **GPU processing is the biggest contribution**

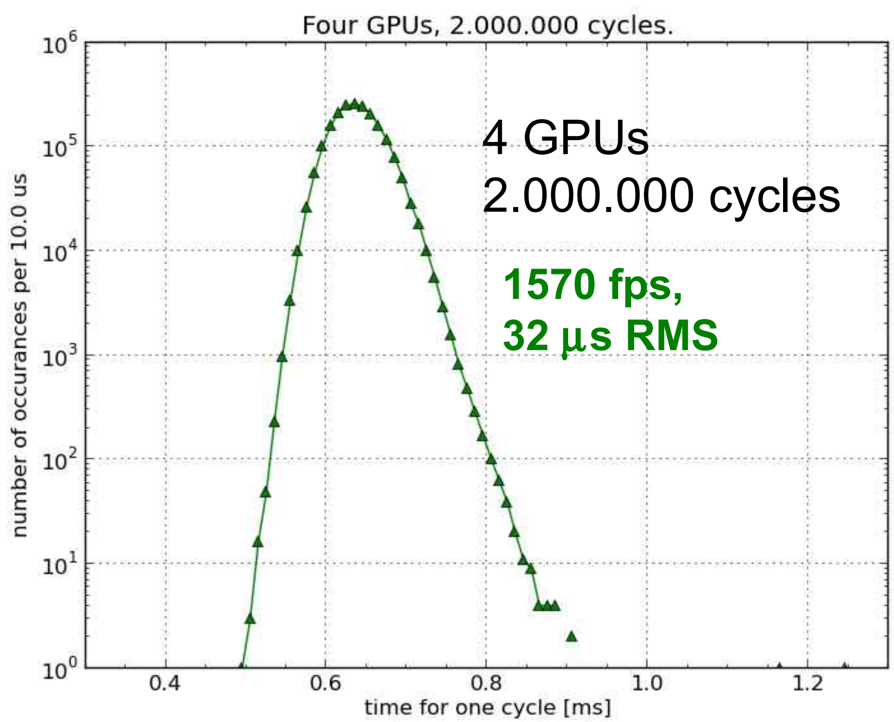
Time needed for copying data to the GPU:





Jitter result





Optimize the parameters

- **MAXIMIZE MEAN FRAME TIME, MINIMIZE JITTER**
- **CPU threads: - N_C - how many** (6,7,...,21)

  - on which hyper-thread they should run
- **CPU: mutex_lock** when launching kernels (yes or no)
- Calibration kernel: number of threads per block (32,64,...,512)
- **MVM kernel:** - number of threads per block (32,64,96,...,512)

  - **size of loop unroll** (15,16,...,125)

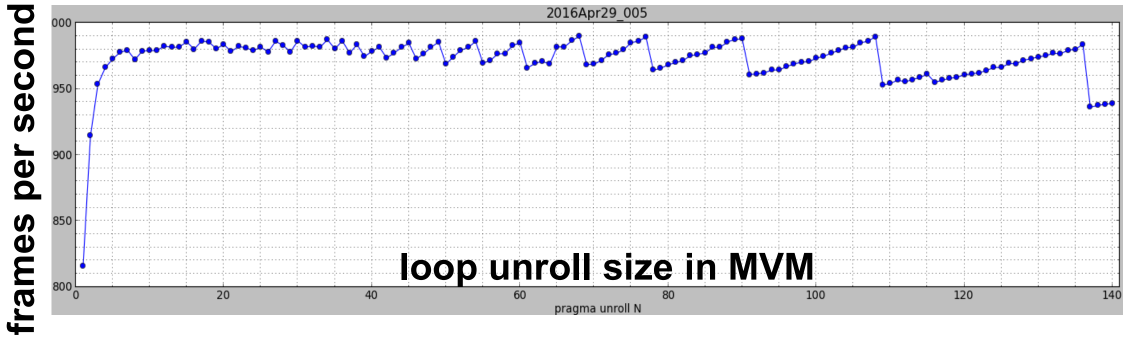
  - copy slopes to shared memory or leave in global?

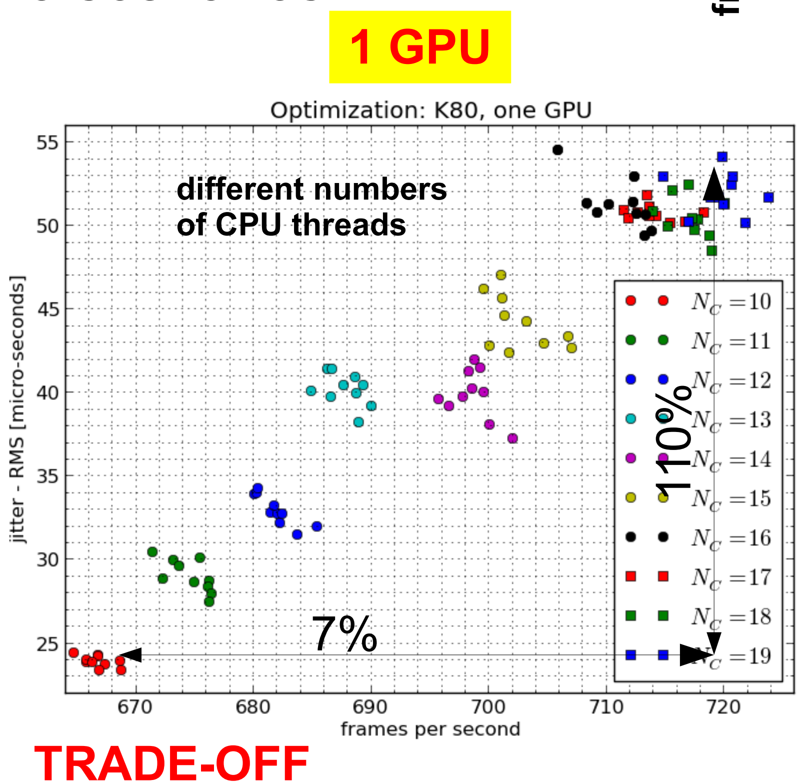
  - use an "if" clause to stop from processing invalid data (yes or no)
- GPU: - cudaStreamSynchronize (improves speed for fermi GPUs)

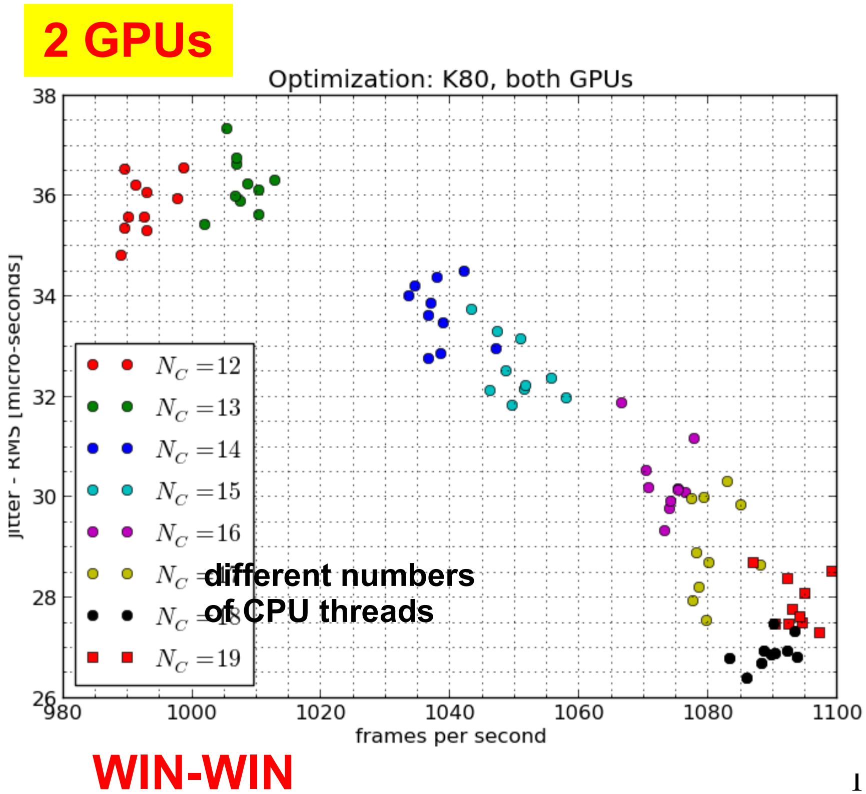
  - end synchronization
- **Run on several GPUs**
- Additional options:
  - operating system (generic or lowlatency)
  - log on as root or as a normal user
  - copy pixels to GPU or not copy pixels to GPU
  - balance between speed and jitter?

Optimize: go fast, low jitter

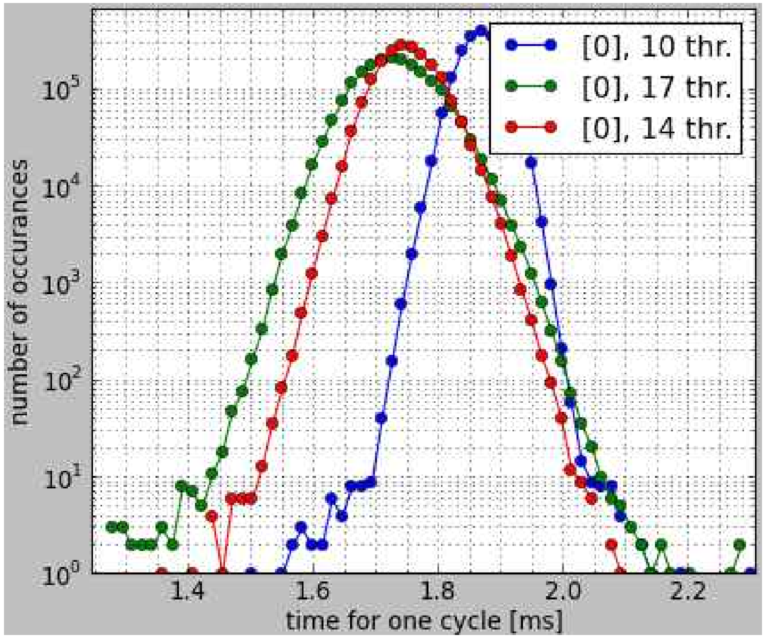
- Parameters:
  - number of CPU threads,
  - loop unroll size in MVM
- Two scenarios:

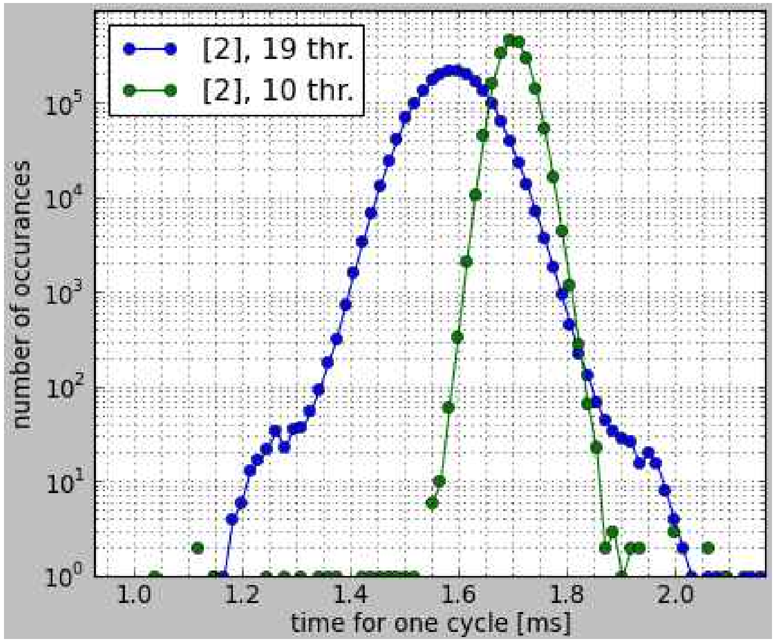




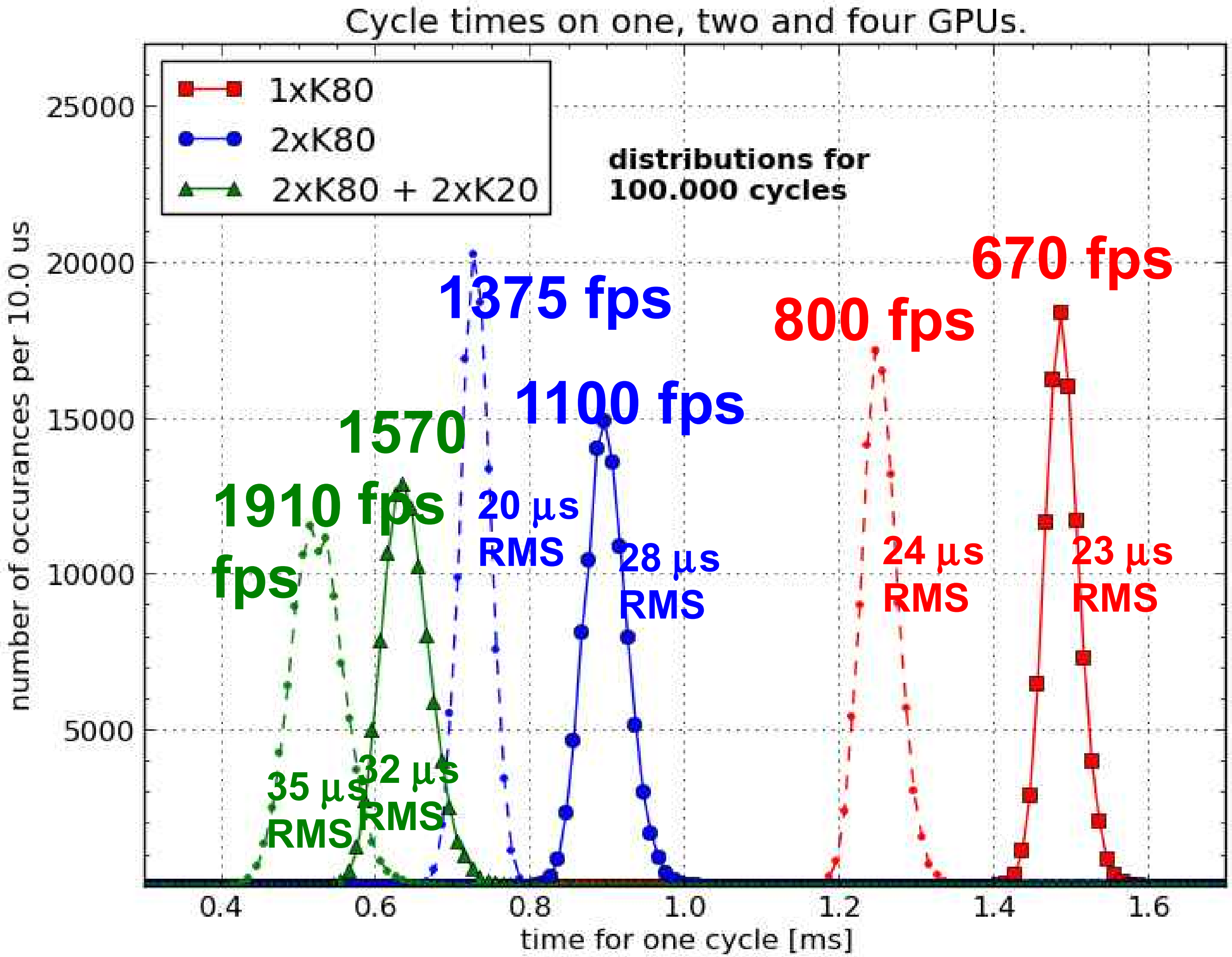


Fastest is not always best





Results



full lines: complete process
dashed lines: without copying pixel data from CPU to GPU

Key findings

- using 1 K80 card (i.e. 2 GPUs) it runs at **1.1 kHz**
- **copying pixels slows you down by 10-20%** and adds about 20% to jitter
- Using several GPUs:
  - 2 GPUs: speed of 1.6 instead of 2.0
    - (speed up of 1.8 if not copying pixels)
  - 4 GPUs: speed up 2.3 instead of 4.0
  - fundamental limitation: kernel launching time
- when running on 2 GPUs, jitter does not increase
- Also when not copying pixel data, splitting into chunks makes calculation faster.

Results

- **Correlation wavefront sensing** (for laser guide-stars)

Number of GPUs	frames per second
one GPU	282 fps
two GPUs	456 fps
four GPUs	541 fps

Conclusions

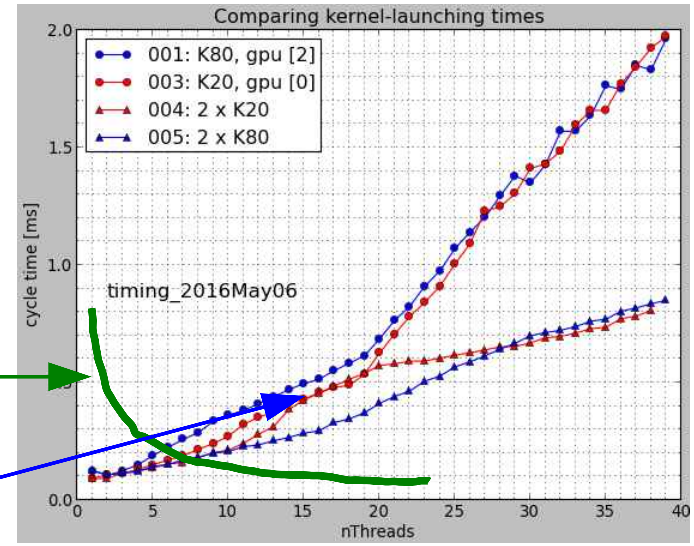
- DARC (Durham RT Controller) using GPUs and standard CUDA tools
- **Data copied to GPU** in parallel to processing
- 80x80 SCAO on a single K80 card (2 GPUs):

  1100 frames per second
- 4 GPUs: 1570 frames per second
- Jitter: RMS = 30 μs
  - one or two outliers (5ms) in a million
- **Good candidate for ELT RTC**
- SPIE 9909, 99094S (2016); submitted to Journal for Real-Time Image Processing

About the bottlenecks

- The optimal nThreads is a balance between three trends:
  - data copying time

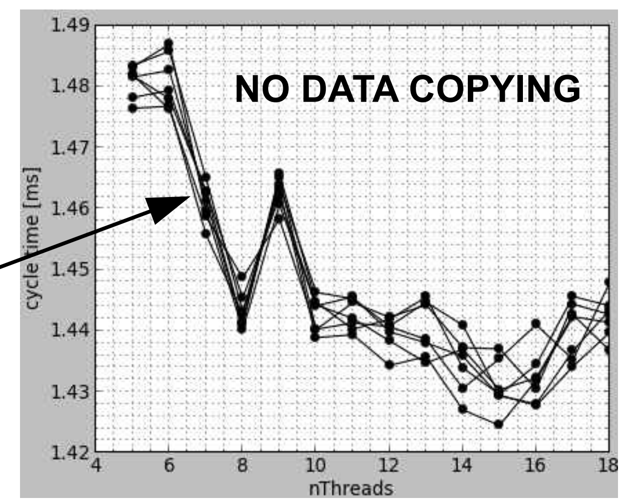
    more threads --> faster



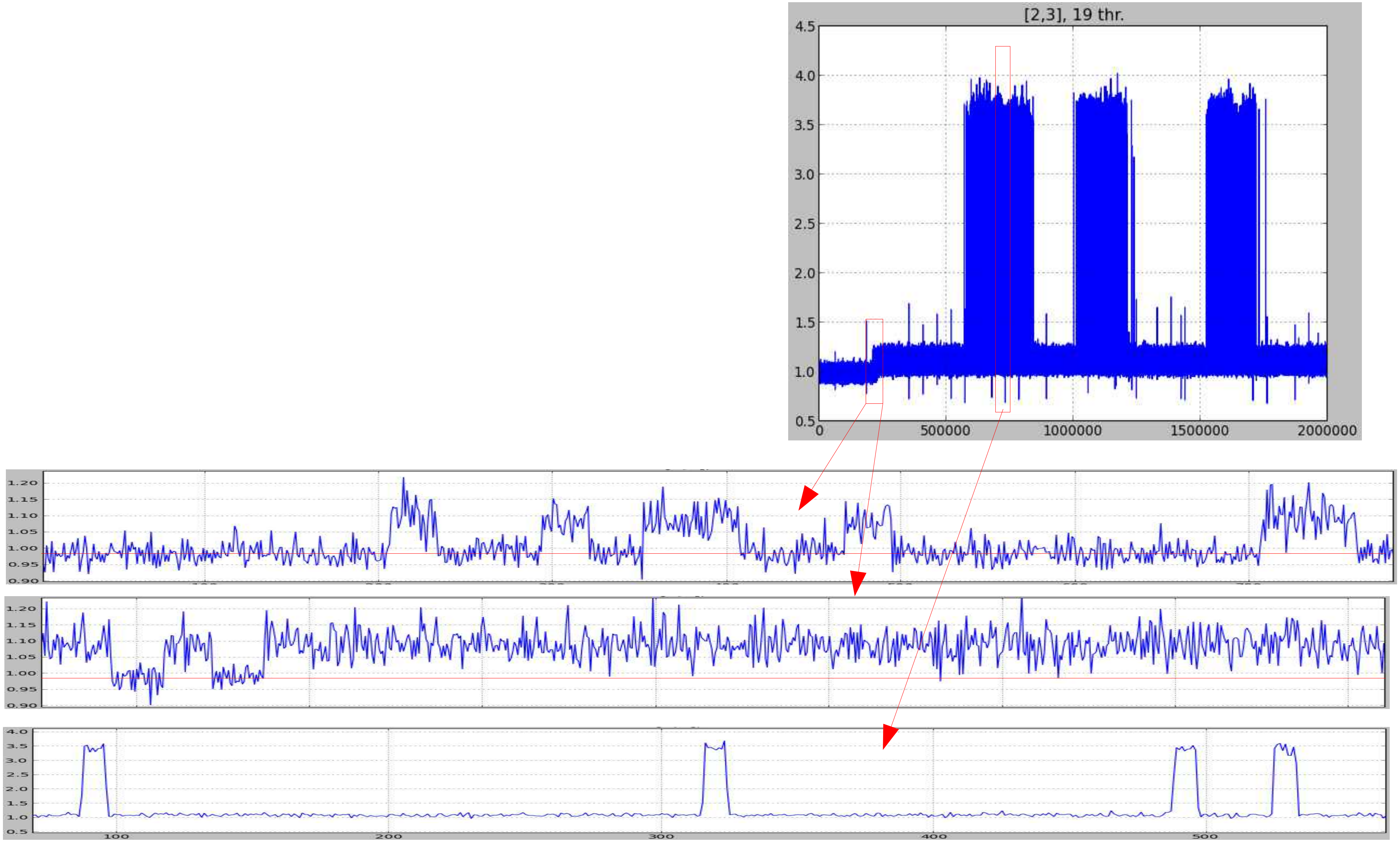
  - pipeline base time:
    - **launching GPU kernels**
    - managing of CPU threads

    more threads --> slower
  - GPU utilisation

    more threads --> faster

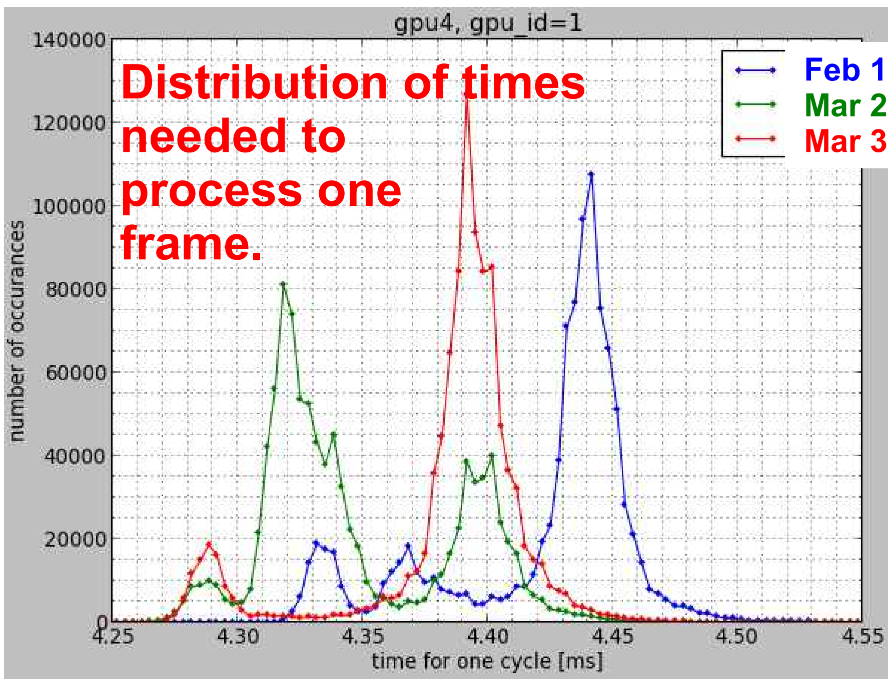


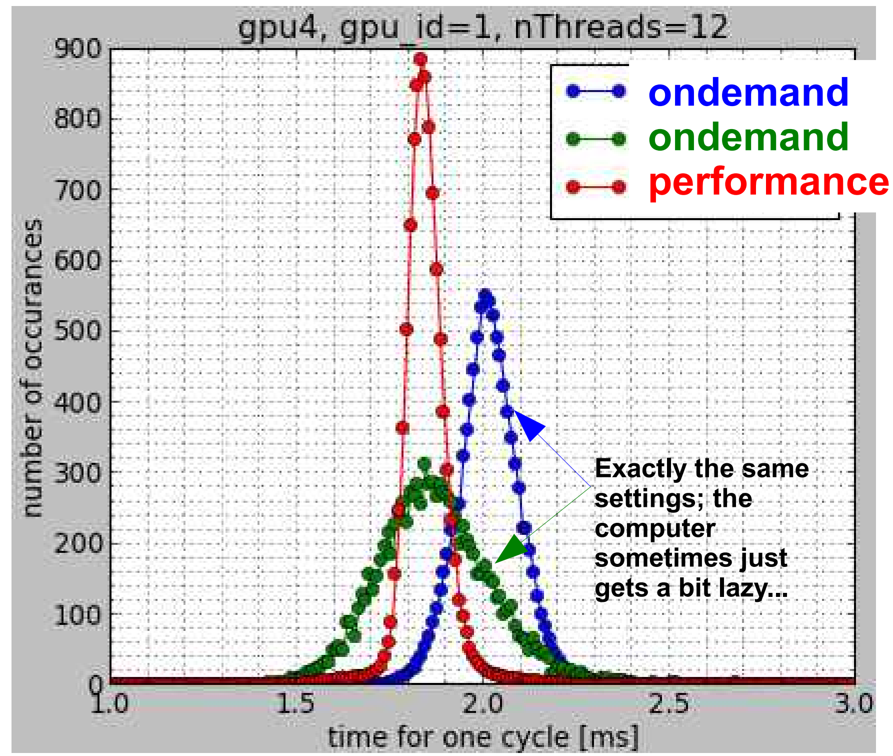
If K80 gets too hot, it slows down



Life is hard

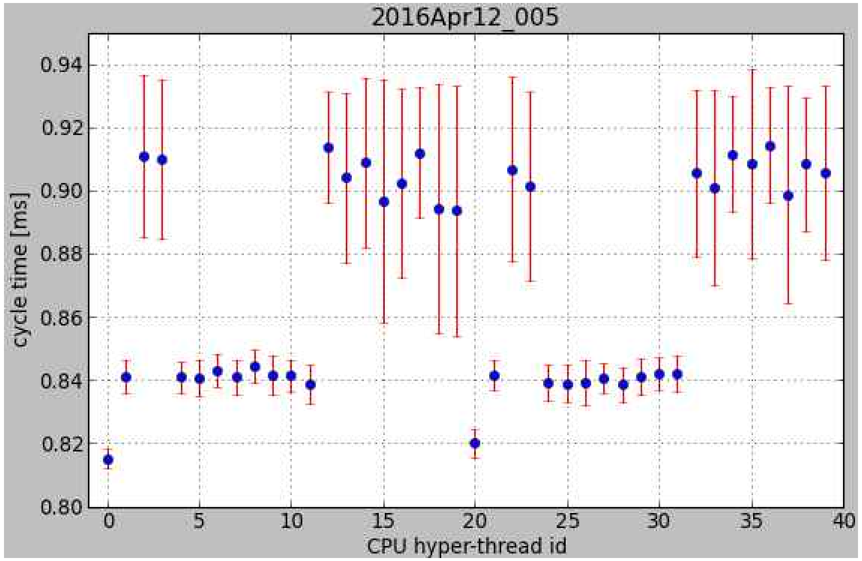
- Stuff is not repeatable



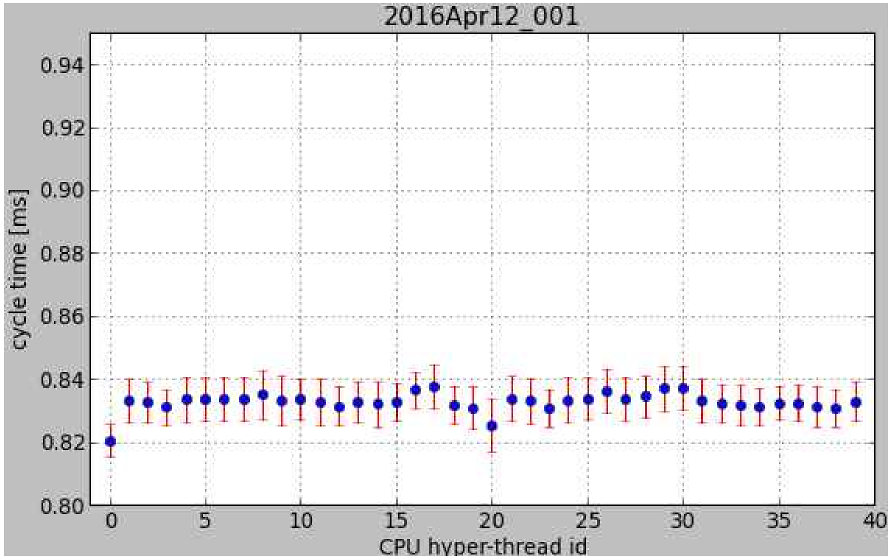


Life is hard

- There is stuff you can *not* understand:



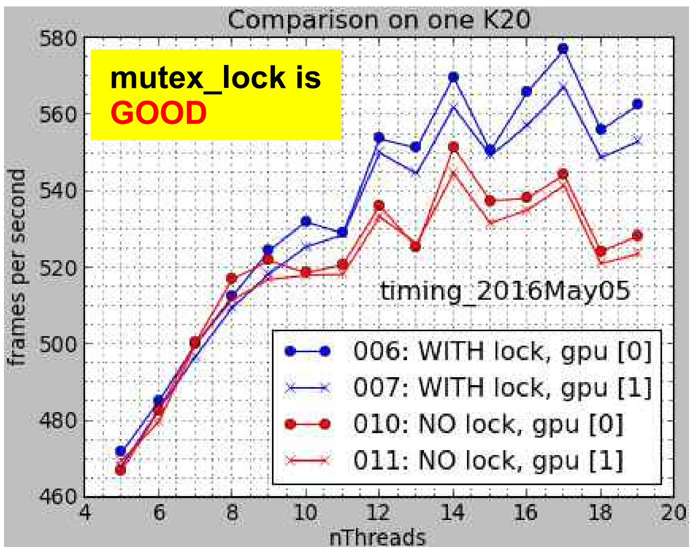
The usual behaviour

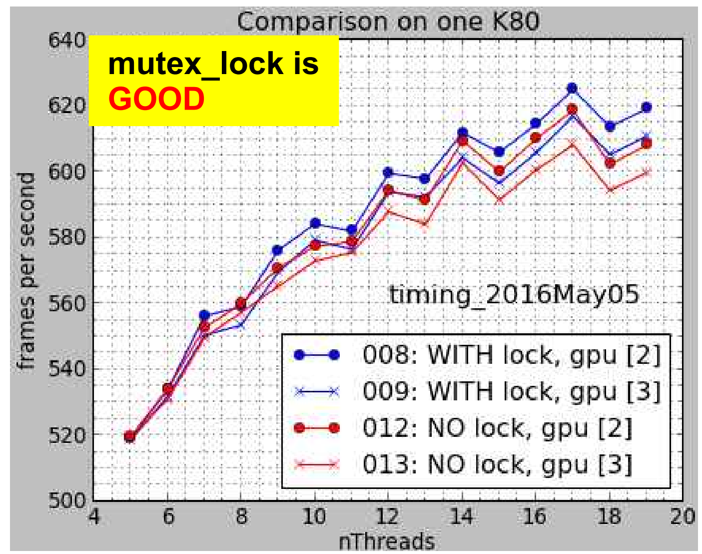


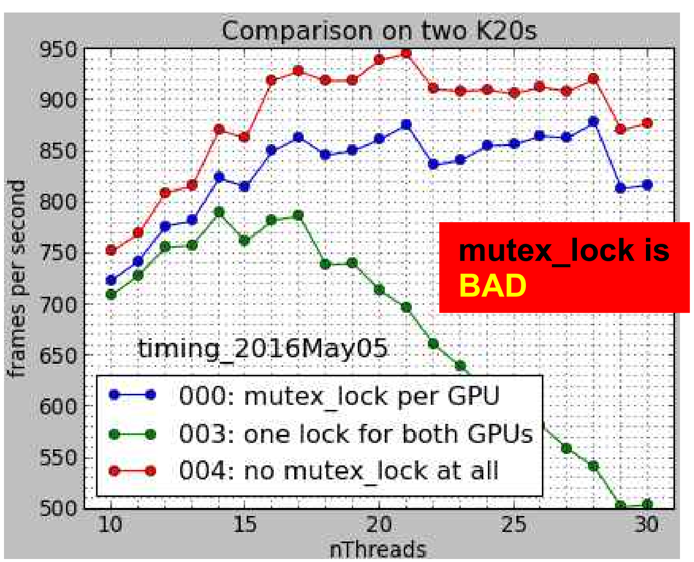
Behaviour when Saavi is running her simulation on the computer

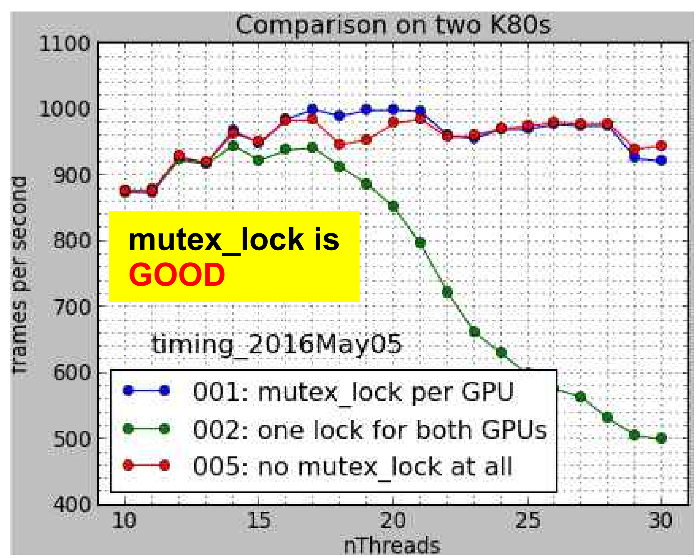
Life is hard

- The rule you've found does not apply to all the stuff:









Just like car insurance...