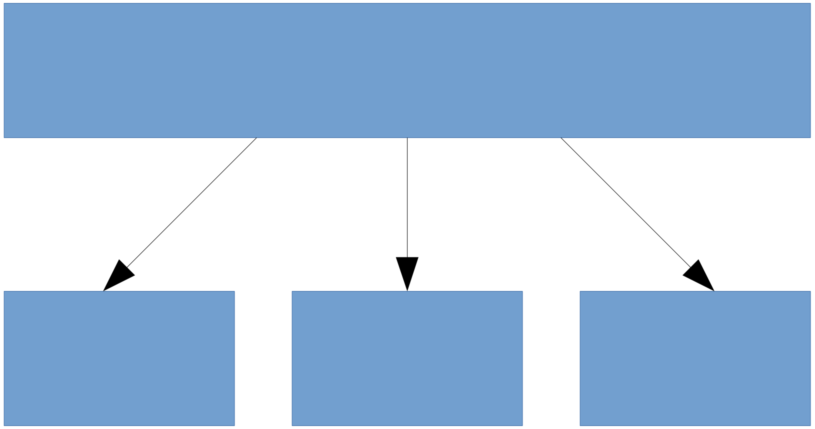
JUNGFRAU

Data Conversion with CUDA

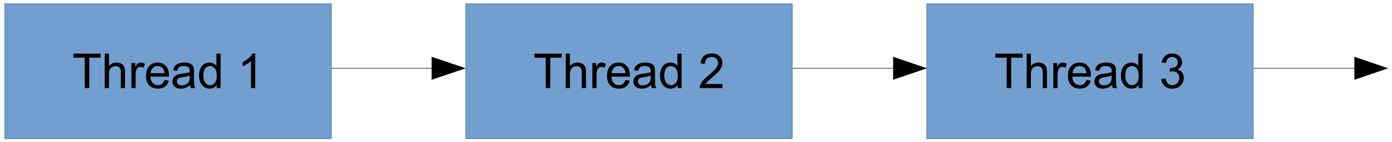
Parallel Computing

- take a large problem
- break it into smaller parts
- solve small problems concurrently



Task Parallelism

- every thread works on a different task (e.g. pipelining)
- example: Video processing
  - Thread 1: Load frames
  - Thread 2: Remove blur
  - Thread 3: Adjust colors, ...



- works well on multi-core CPUs
- only good for coarse-grained work on GPUs

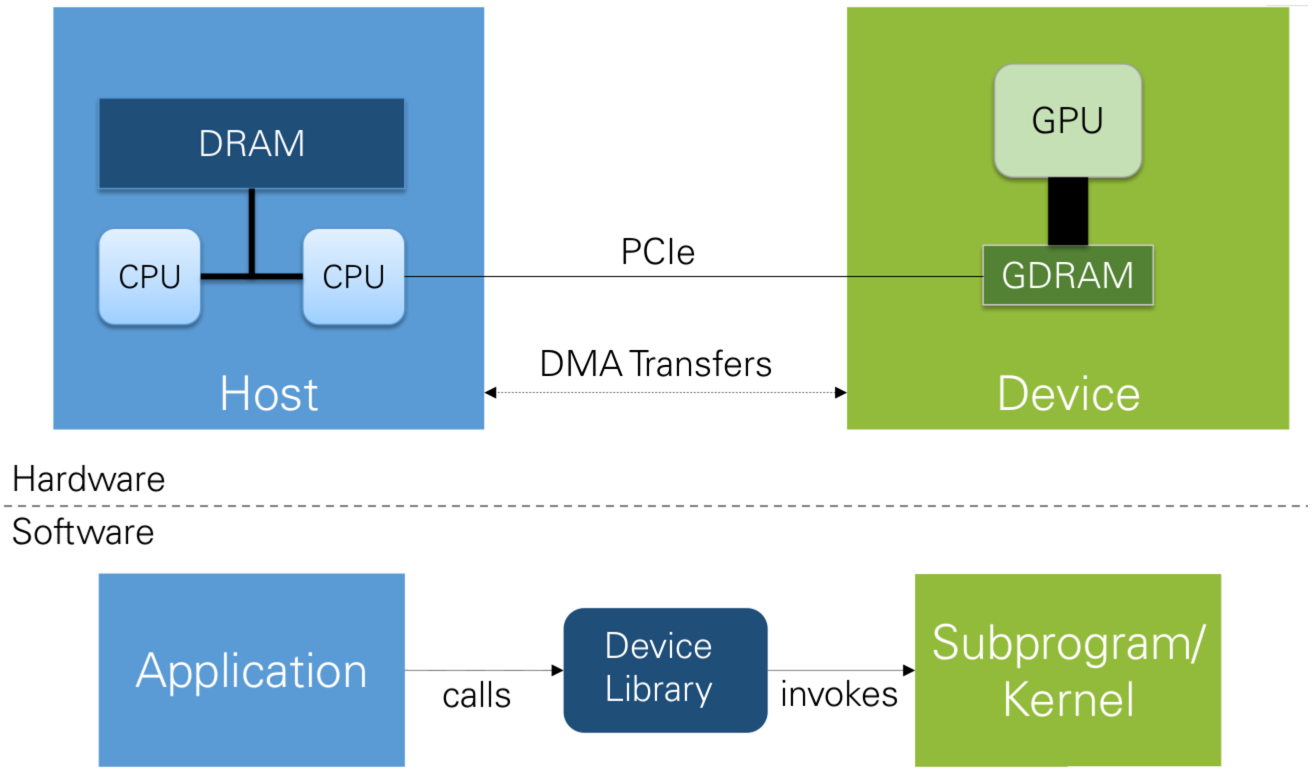
Data Parallelism

- every thread performs the same task on different data
- example: Video processing
  - Thread 1: works on top left corner
  - Thread 2: works on top right corner
  - Thread 3: works on bottom left corner
  - Thread 4: works on bottom right corner

Thread 1	Thread 2
Thread 3	Thread 4

- works well on CPUs and GPUs
- requires dividable data structures

GPU System Setup

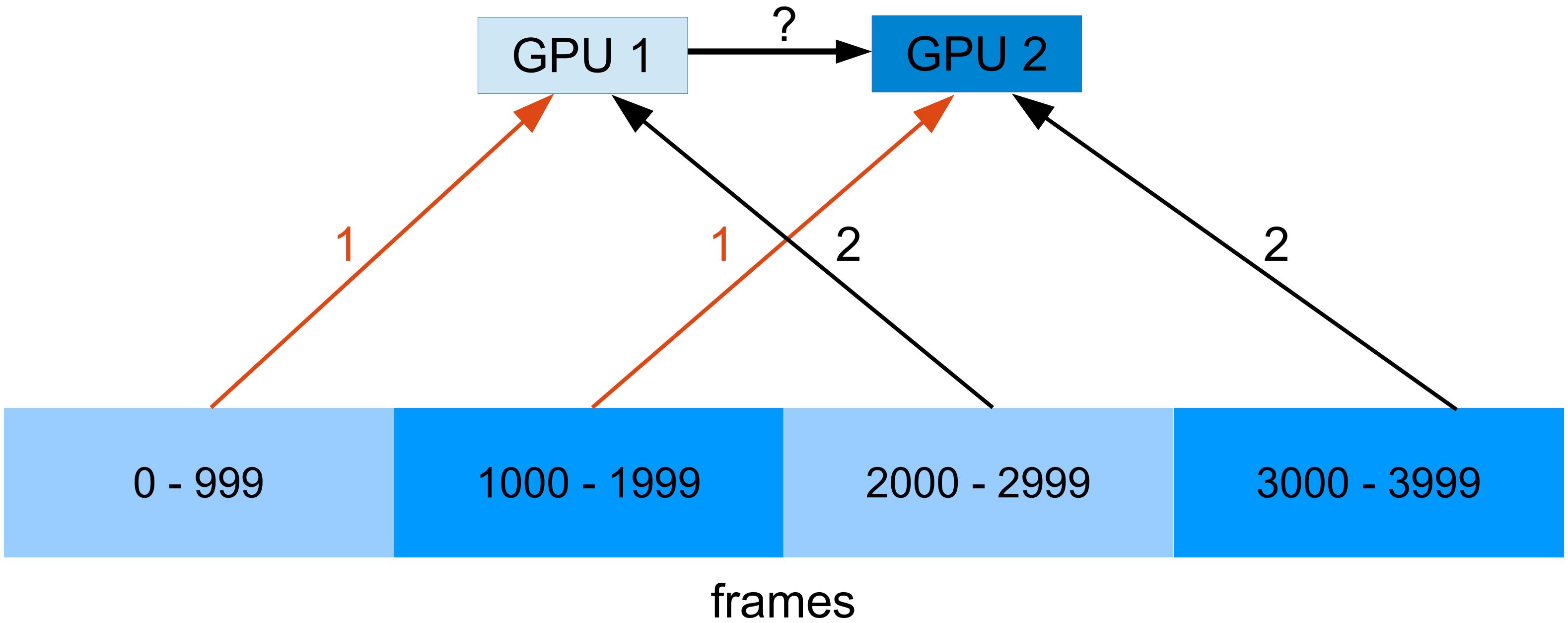


Applying GPU Processing to JUNGFRAU

- JF Module sends contiguous array of 2D matrices
- each (m x n) Matrix can be interpreted as an image
- GPUs are optimized for image processing
- run a thread for each pixel
- process each pixel concurrently with SIMD operations
- calculation complexity reduced from (m x n) to 1

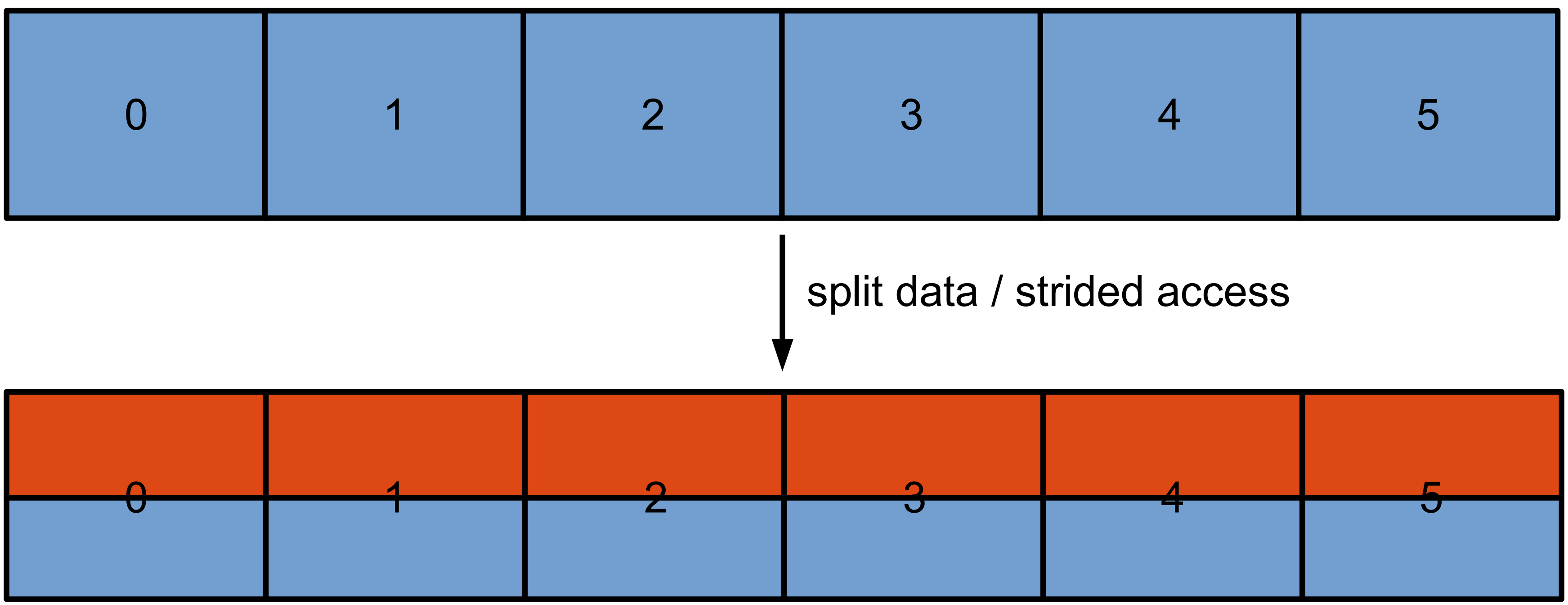
Pedestal Correction and Parallelism

- algorithm for pedestal correction introduces dependency on previous frames
- problematic if more than 1 GPU is needed



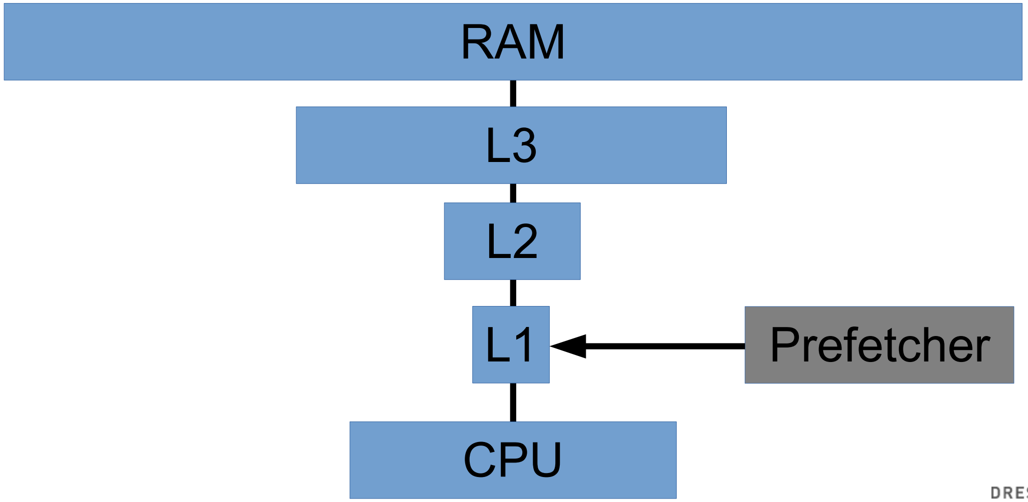
Solving the Dependency Problem

- data can be rearranged so GPUs do not share pixels
- GPUs process parts of frames instead of whole frames
- pedestal correction can be done independently
- requires a different data layout



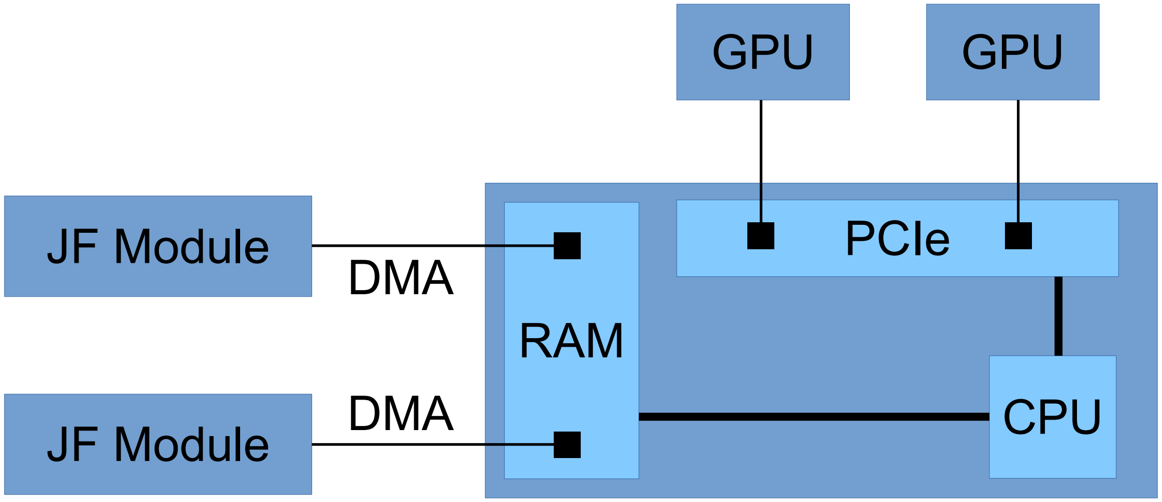
Hardware Limitations

- **working on non-contiguous data is very slow**
- causes cache misses
- many small copy operations
- might need additional buffers for each GPU
- r/w to random addresses is slow
- test results showed **~2 GiB/s** for rearranging data
- speed depends on type of RAM and CPU
- does not scale with amount of GPUs
- probably slower than serial processing of data



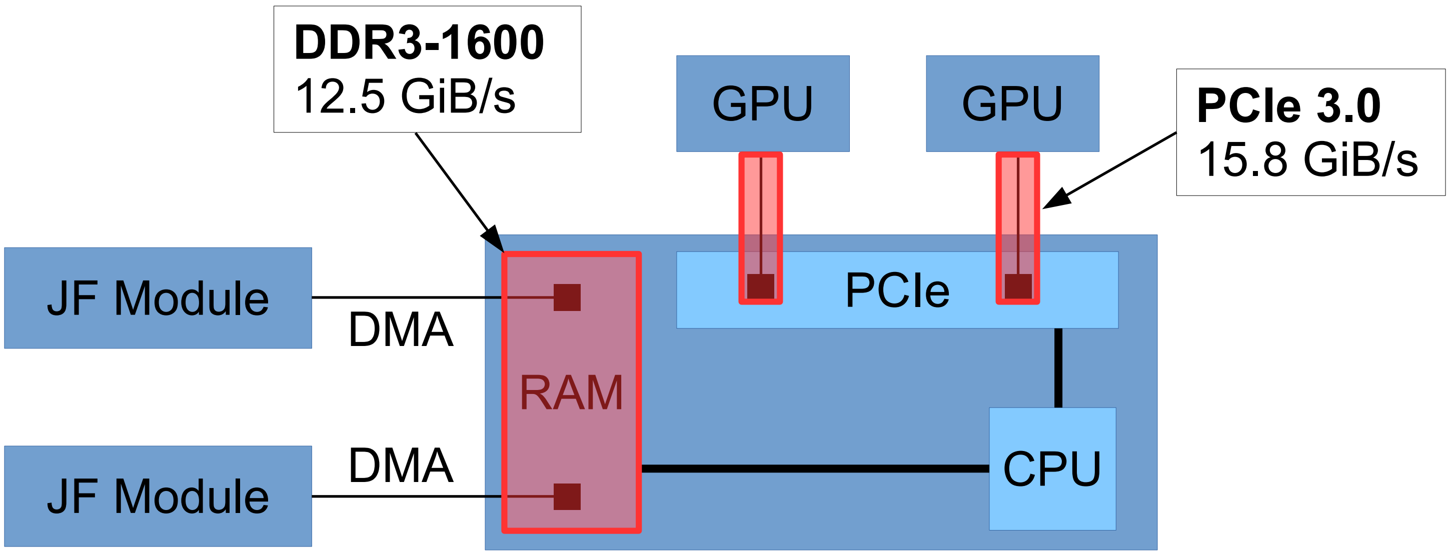
Achieving High Data Rates

- use host system mainly to transfer data to GPUs
- do not rearrange data on host or GPUs
- run small CUDA kernels
- use multi-channel RAM
- DMA system to transfer data from JF module to host



Achieving High Data Rates

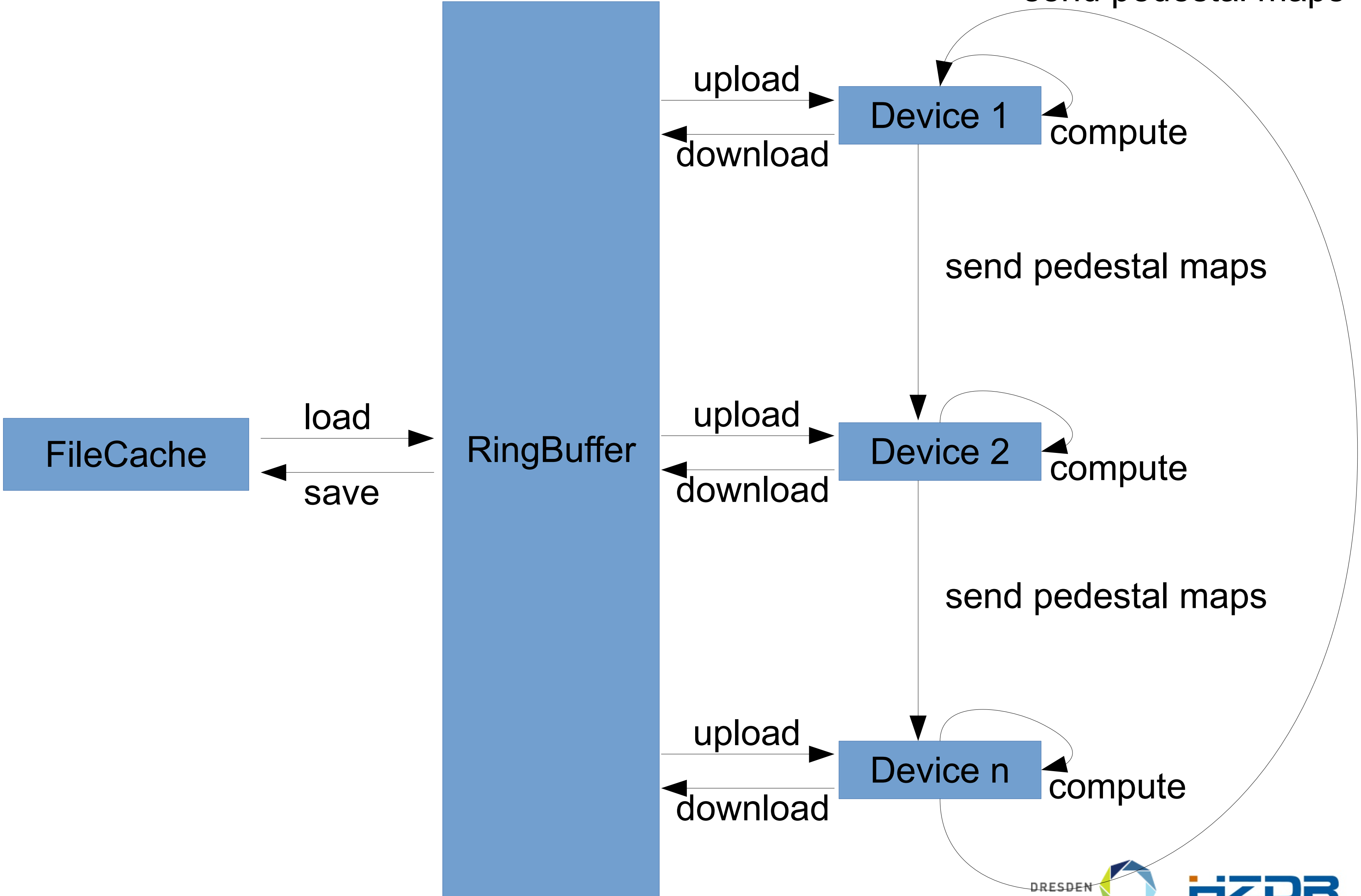
- use host system mainly to transfer data to GPUs
- do not rearrange data on host or GPUs
- run small CUDA kernels
- use multi-channel RAM
- DMA system to transfer data from JF module to host



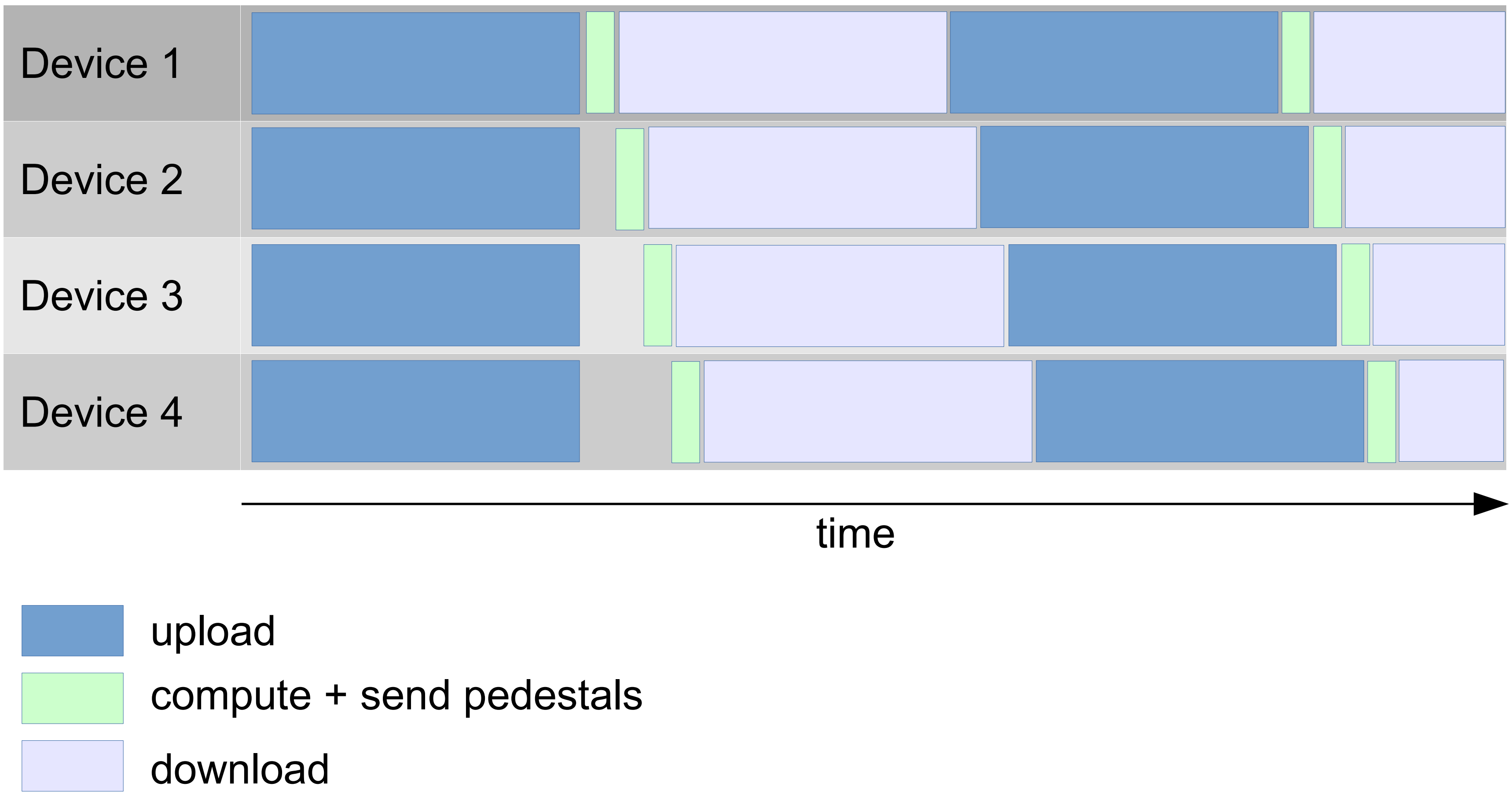
DRESDEN concept
HZDR

Member of the Helmholtz Association
Institute of Radiation Physics I www.hzdr.de

Sotfware Model

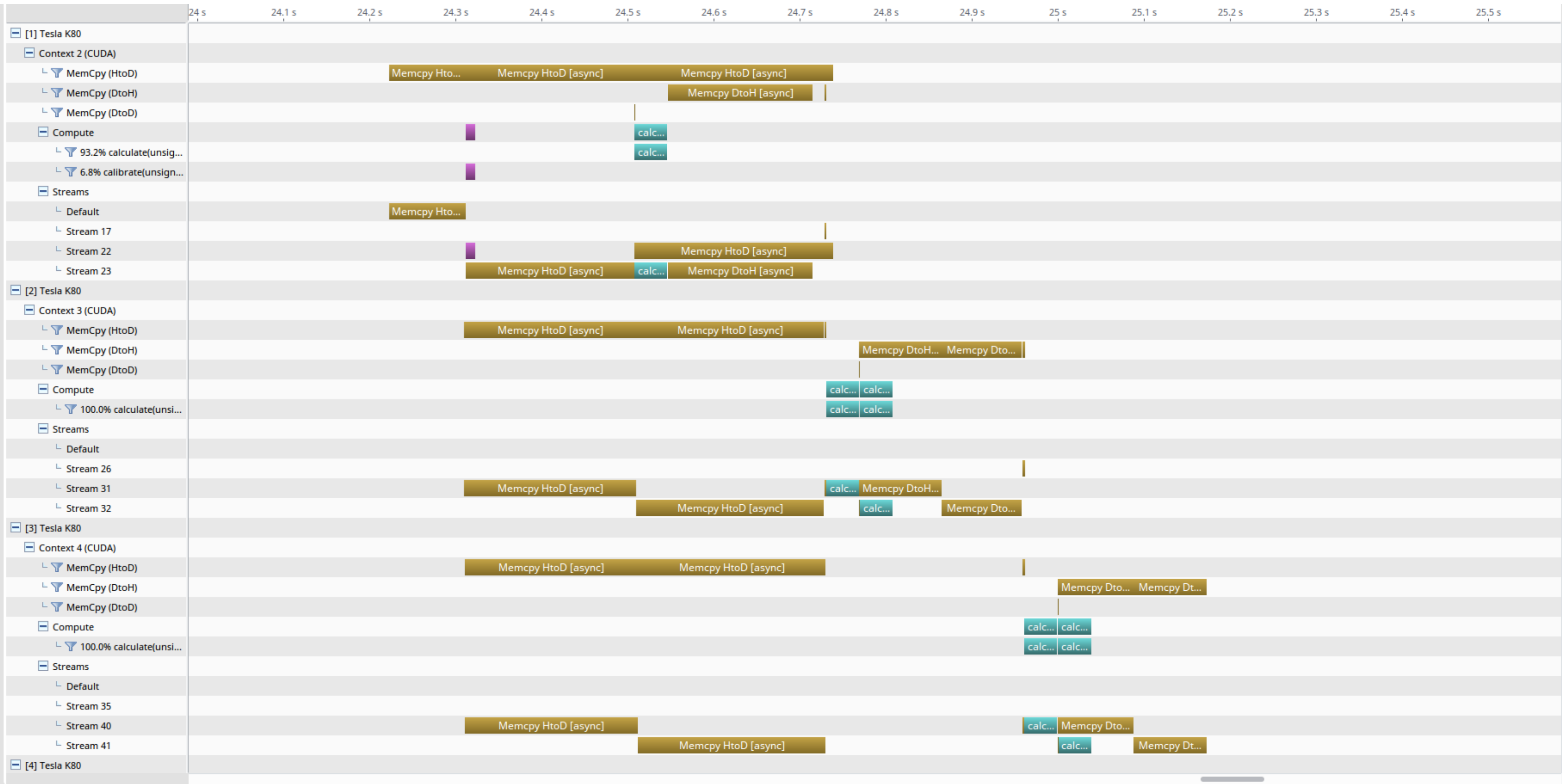


Timeline – Best Case



Performance Measurements

- NVIDIA Tesla K80 (branch: calibration 07-05-17)



Performance Measurements

- NVIDIA Tesla K80 (branch: calibration 07-05-17)
- PCIe throughput (single GPU): **12.5 GiB/s**
- PCIe throughput (8 GPUs)
  - GPU 0 to 3: **5.3 GiB/s**
  - GPU 4 to 7: **1.4 GiB/s**
  - (probably limited by PCIe)
- kernel execution time (1000 frames): **38.5 ms**
- kernel can process ~**25900** frames (1024 x 512 px)

Member of the Helmholtz Association
Institute of Radiation Physics I www.hzdr.de

Scalability

- model scales linearly (to some extent)
- calculation process is much faster than data transfer
- data transfer is bottleneck
- 5 – 12 times bandwidth improvement with nvlink
- additional GPUs improve bandwidth, too
- immense power of GPUs barely used

Future plans

- test code for correctness
- Cracen / alpaka integration (HZDR library)
- use CUDA for automatic configuration
- refactor code
- find hardware bottlenecks in multi-GPU systems