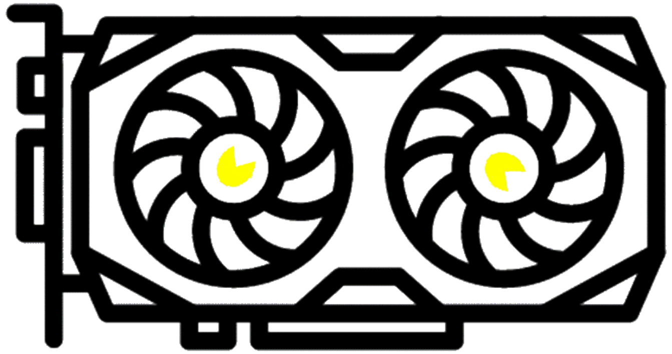
University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering



Parallel Computing
GPU Program Flow (like a boss)

CUDA Streams

- A sequence of operations executed on the device **in order** as executed on the host
- The operations (kernels and data transfers) in a stream **cannot** overlap
- The default stream:
  - Synchronizing stream with respect to operations on the device on **any other stream**
  - Operation starts when all previously issued operations in any stream are finished
  - New launched operations begin after the default stream operation is finished
- Non-default steam:
  - All operations are async (non-blocking)

CUDA Streams – How to

- Create a (or N) stream(s)

```
cudaStream_t stream;
gpuErrCheck(cudaStreamCreate(&stream));                 // gpuErrCheck is our PAC macro
```

- Execute a kernel

~~pacKernel<<< blocks, threads, SHMbytes >>>();   // default stream~~

```
pacKernel<<< blocks, threads, SHMbytes, stream >>>();   // use a non-default stream
gpuErrCheck(cudaPeekAtLastError());                     // did the kernel launch work?
```

- We are not mad and cleanup our mess

```
gpuErrCheck(cudaStreamDestroy(stream));
```

CUDA Streams

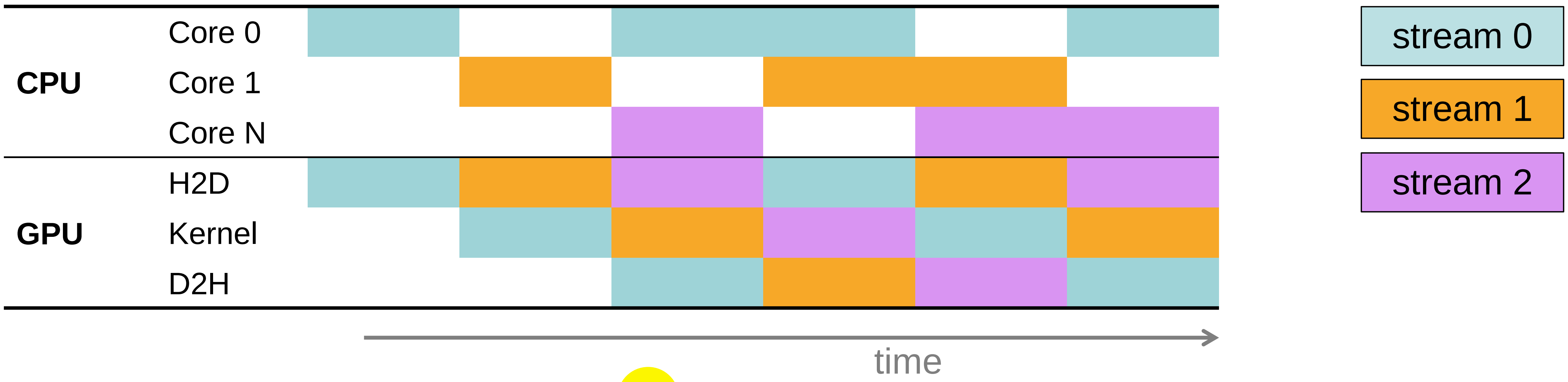
- Since everything is executed async in respect to the host, we need to synchronize
- `cudaDeviceSynchronize()` wait until all previous issued operations **in all streams** are done
- `cudaStreamSynchronize(stream)`
  wait until all previous issued operations **in this stream** are done
- `cudaStreamQuery(stream)` check if all operations in this stream are finished (empty steam)
- `cudaEventSynchronize(event)` and `cudaEventQuery(event) -` see code of last week
- `cudaStreamWaitEvent(event)`
  can sync on a specific event of any stream, even of another device

CUDA Streams – Some notes

- CUDA 7 has a major improvement: `--default-stream per-thread` compile argument
- Every thread gets its own default stream which **is a non-default-stream**
  - No global device sync
- Handy for OMP parallel directives as no tracking of streams needs to be done
- You must implement a domain decomposition of data and processing
- Don't overdo it! A few streams are most often more than enough!

Observations

- Using multiple streams can increase the occupancy of the hardware significantly
- The streams need different CPU-threads in order to run in parallel
- There are free resources on the CPU:
  Maybe one of the tasks can be done on the CPU instead of the GPU?



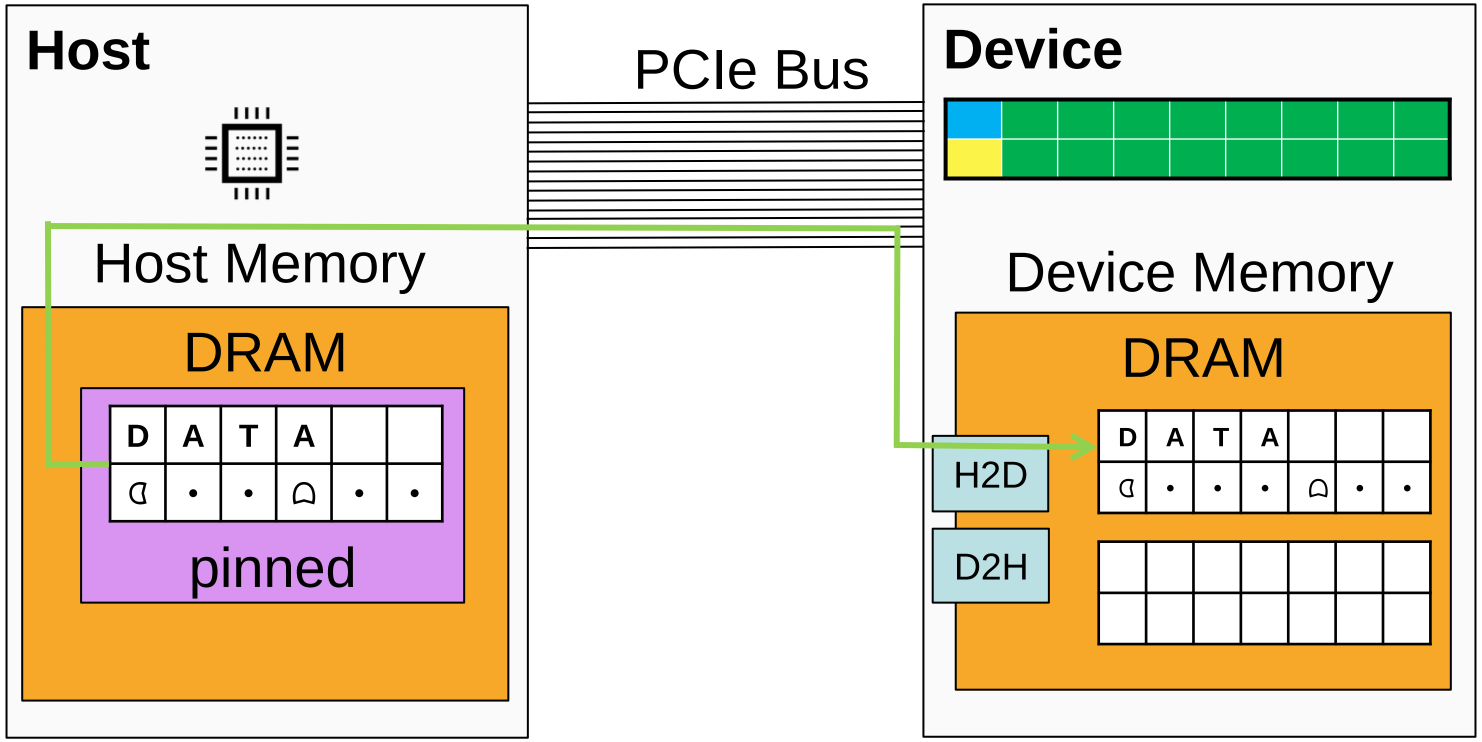
CUDA Processing Flow using async data transfer

1. Load data into Host Memory
   - CPU load
   - Needs to be pinned memory

```
int* h_matrixA;
cudaMallocHost(&h_matrixA,
               size*sizeof(int));
```

2. Copy data to Device using H2D (async)
   - H2D engine load

```
cudaMemcpyAsync(d_matrixA,
                h_matrixA,
                size * sizeof(int),
                cudaMemcpyHostToDevice,
                stream);
```



CUDA Processing Flow using async data transfer

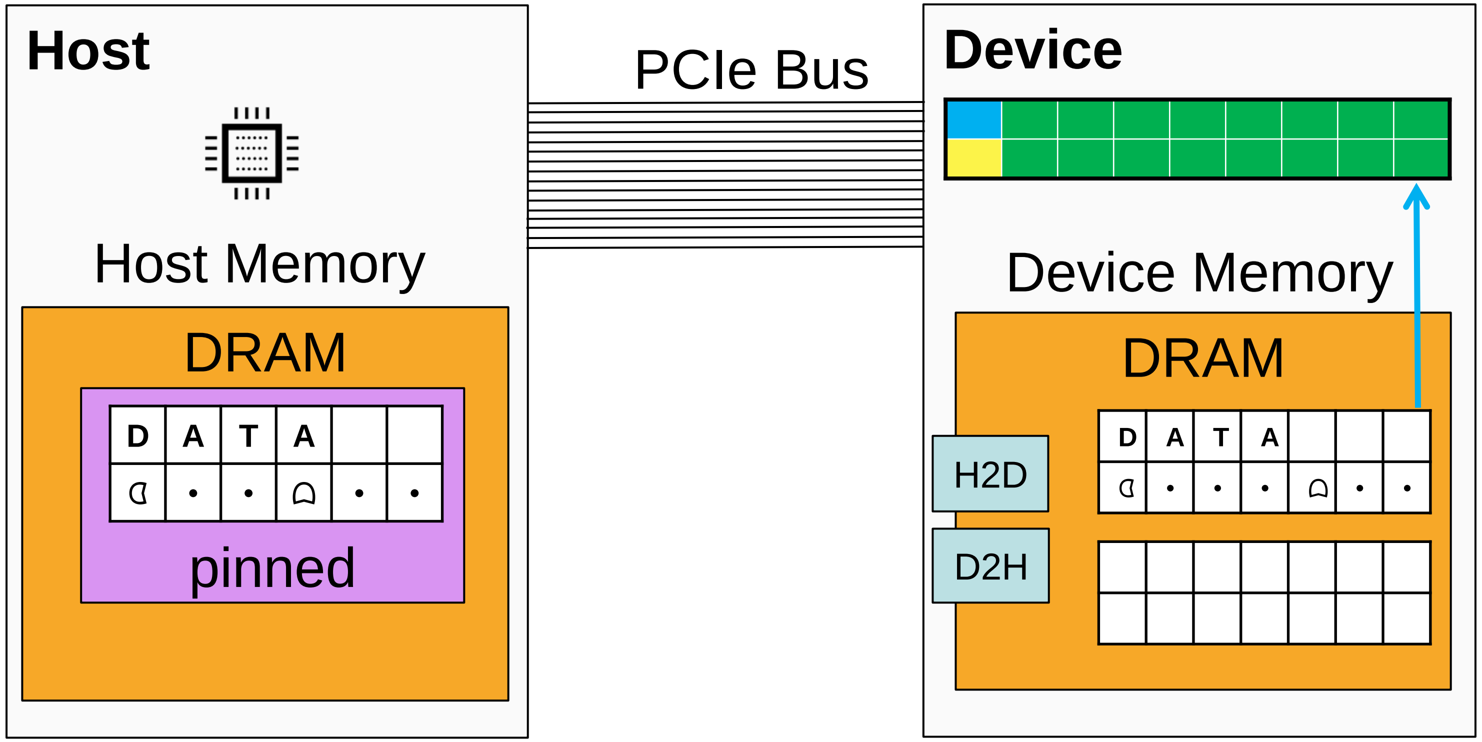
3. Execute kernel
   - GPU load (kernel engine)

```
packernel<<< blocks,
              threads,
              SHMbytes,
              stream >>>(...);
```

4. Same way back using async D2H ☺

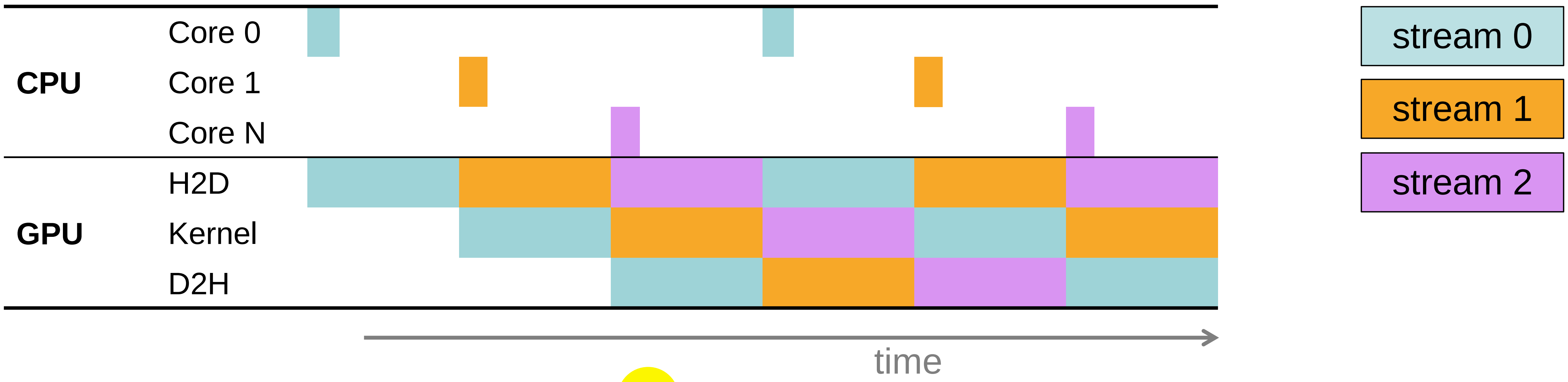
5. Clean up your mess

```
cudaFree(d_matrixA);
cudaFreeHost(h_matrixA);
...
```

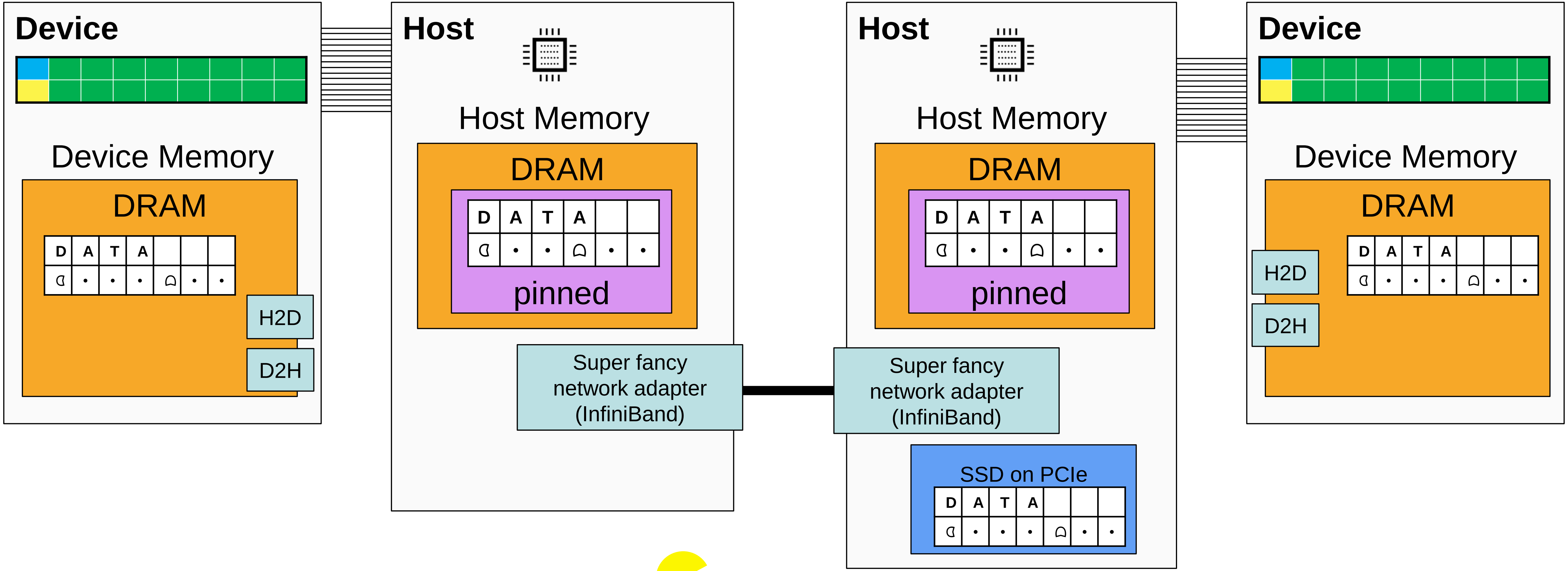


Observations

- Offload CPU work to DMA engines by using async copy (and thus pinned memory)
- Think hard how to use these free CPU cycles in parallel and efficient
  Maybe do some fancy AVX stuff ☺



CUDA Processing Flow – think outside of the box - RDMA

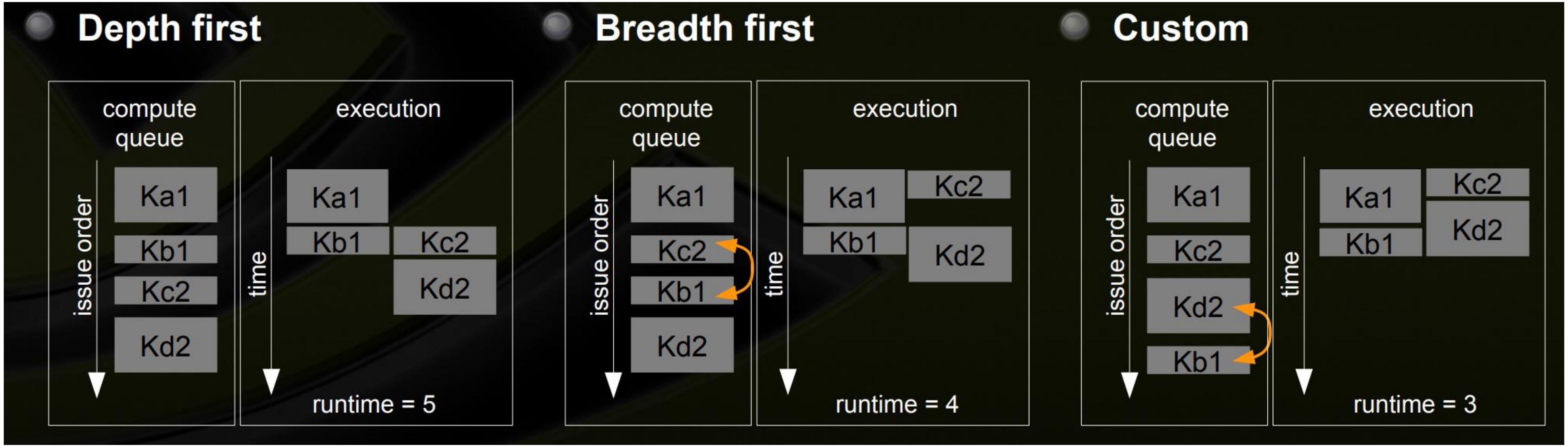


CUDA Streams – Some more notes

- In real life cases, the kernel uses more time than the copy
  → Even one CPU thread using multiple streams can overlap H2D/D2H with the kernels
  → You can prepare work in advance
  → Use CUDA events to synchronize/wait at the right spots
- Use the free CPU cores to:
  - Proceed with the GPU results (preferred)
  - Do the same thing as the GPU but on the CPU, even if significant slower
- Using multiple processes using the same GPU needs additional work
  - 1 process = 1 context on the GPU – Contexts cannot run in parallel on the GPU
  - Multi-process Service (MPS) will time multiplex all calls of N processes into one context

CUDA Streams – Some more notes ++

- Additional knowledge – not needed for exam ;-)
- Use modern hardware / CUDA versions / sm-arch or things will get more complicated …



* source: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdfvidia.com