CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Advanced Features Overview

CSCS Summer School 2025

Andreas Jocksch, Prashanth Kanduri, Radim Janalik & Ben Cumming

Concurrency

Concurrency

Concurrency is the ability to perform multiple CUDA operations simultaneously, including:

- CUDA kernels;
- Copying from host to device;
- Copying from device to host;
- Operations on the host CPU

Concurrency

Concurrency is the ability to perform multiple CUDA operations simultaneously, including:

- CUDA kernels;
- Copying from host to device;
- Copying from device to host;
- Operations on the host CPU

What concurrency enables
Both CPU and GPU can work at the same time
Multiple tasks can run simultaneously on the GPU
Communication and computation can be overlapped.

Launch-Execute Sequence

Host Code

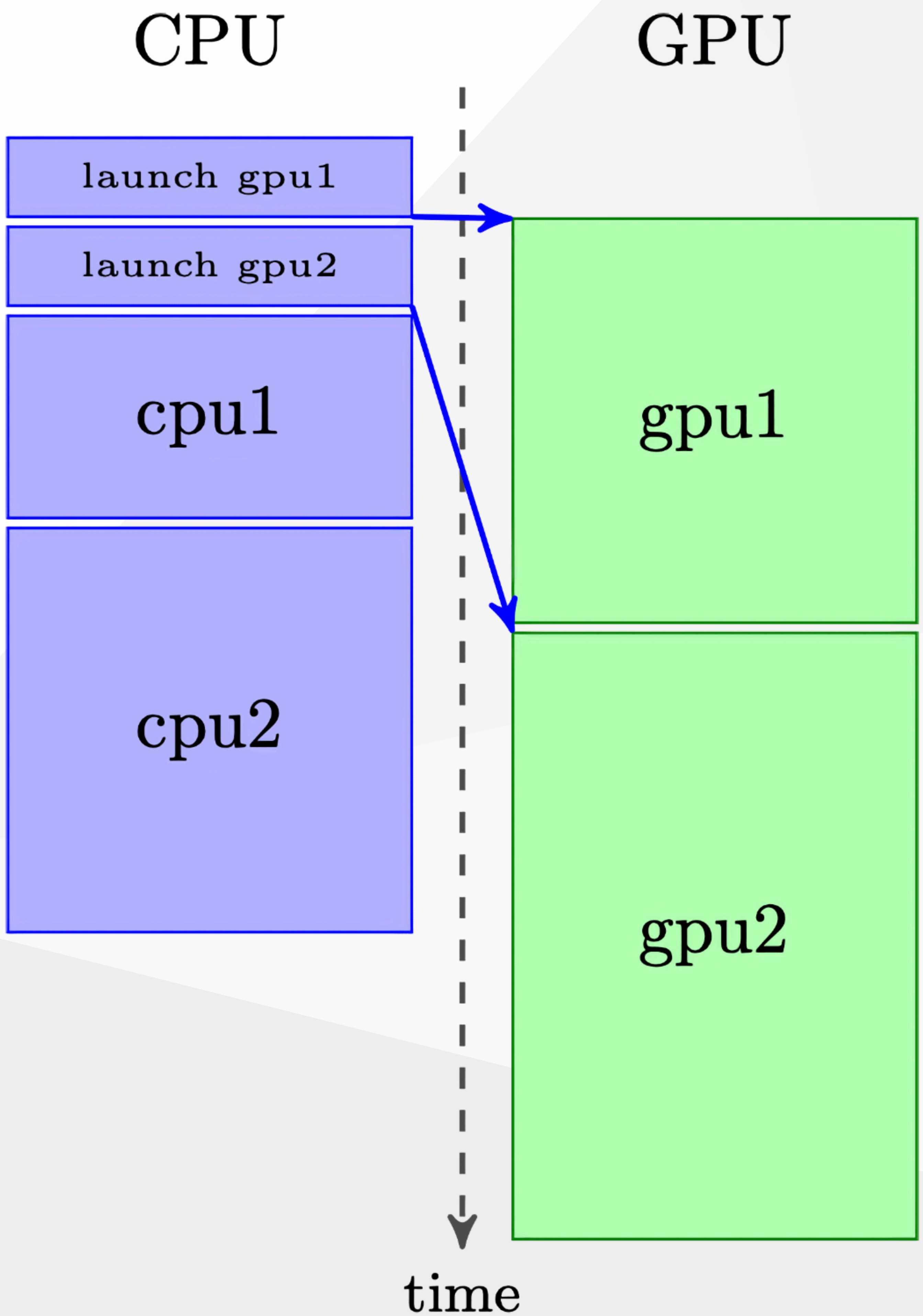
```
kernel_1<<<...>>>(...);
kernel_2<<<...>>>(...);
host_1(...);
host_2(...);
```

The host (in order):

- launches the kernels
- execute host calls sequentially

The GPU:

- executes asynchronously to host;
- executes kernels sequentially



Overlapping Independent Operations

The CUDA language and runtime libraries provide mechanisms for coordinating asynchronous GPU execution:

- Independent kernels and memory transfers can execute concurrently on different **streams**;
- **CUDA events** can be used to synchronize streams and query the status of kernels and transfers

Streams

A CUDA stream is a sequence of operations that execute in issue order on the GPU.

Streams and concurrency
Operations in **different** streams may run **concurrently**
Operations in the **same** stream are executed **sequentially**
If no stream is specified, all kernels are launched in the default stream

Managing Streams

- Streams can be created and destroyed:
  - `cudaStreamCreate(cudaStream_t* s)`
  - `cudaStreamDestroy(cudaStream_t s)`
- Launch a kernel on a given stream:
  - `kernel<<<grid_dim, block_dim, shared_size, stream>>>(...)`
- The default CUDA stream is the `NULL` stream, or stream `0`

Managing Streams

- Streams can be created and destroyed:
  - `cudaStreamCreate(cudaStream_t* s)`
  - `cudaStreamDestroy(cudaStream_t s)`
- Launch a kernel on a given stream:
  - `kernel<<<grid_dim, block_dim, shared_size, stream>>>(...)`
- The default CUDA stream is the `NULL` stream, or stream `0`

Basic CUDA Streams Usage

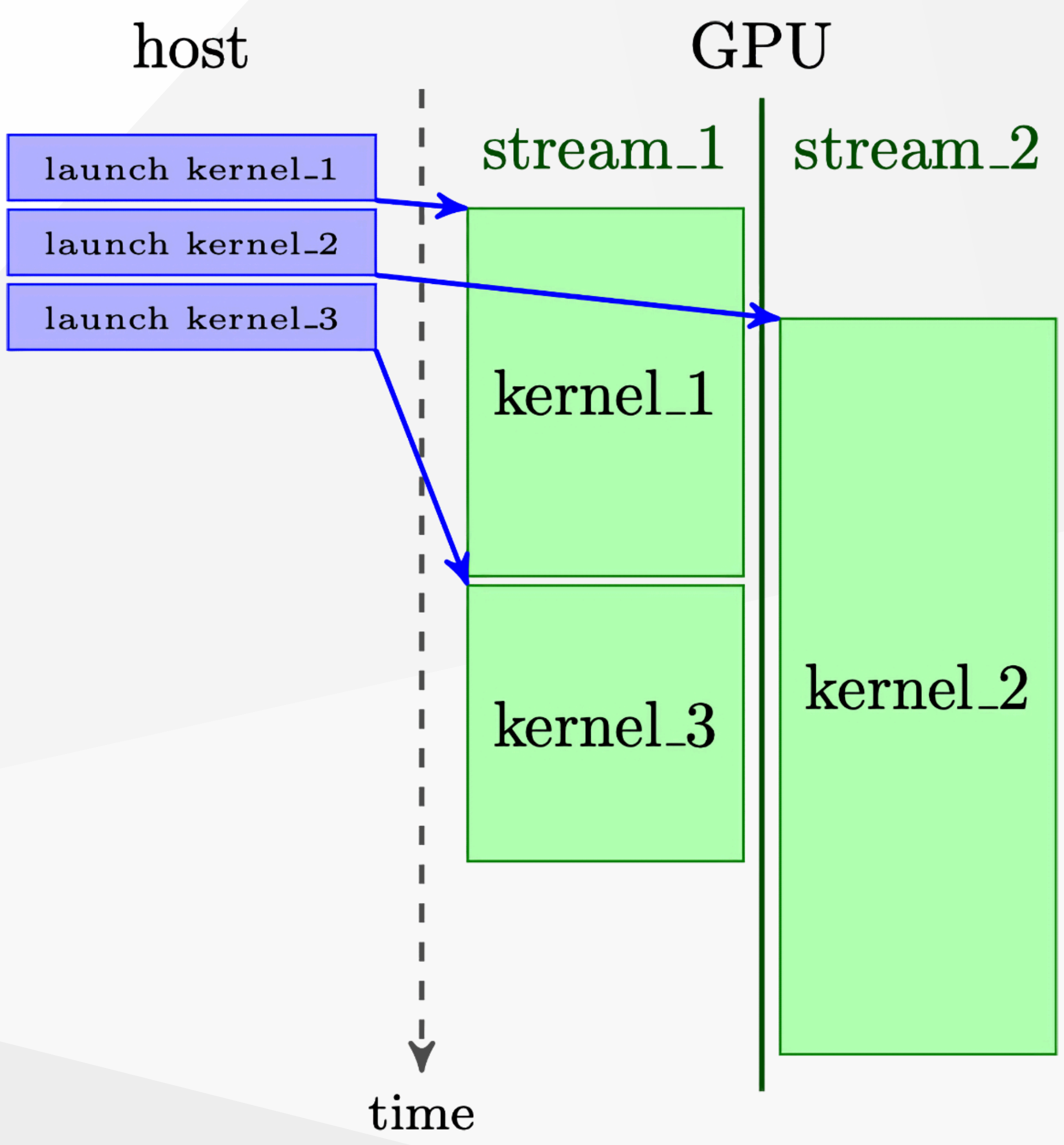
```
// create stream
cudaStream_t stream;
cudaStreamCreate(&stream);
// launch kernel in stream
my_kernel<<<grid_dim, block_dim, shared_size, stream>>>(..)
...
// release stream when finished
cudaStreamDestroy(stream);
```

Concurrent Kernel Execution

Host Code

```
kernel_1<<<_,_,_,stream_1>>>();
kernel_2<<<_,_,_,stream_2>>>();
kernel_3<<<_,_,_,stream_1>>>();
```

- `kernel_1` and `kernel_2` are serialized in `stream_1`
- `kernel_2` can run asynchronously in `stream_2`
- **Note** `kernel_2` will only run concurrently if there are sufficient resources available on the GPU, i.e. if `kernel_1` is not using all of the SMs



Asynchronous copy

```
cudaMemcpyAsync(*dst, *src, size, kind, cudaStream_t stream = 0);
```

- Takes an additional parameter stream, which is 0 by default
- Returns immediately after initiating copy:
  - Host can do work while copy is performed;
  - Only if **pinned memory** is used
- Copies in the same direction (i.e . H2D or D2H) are serialized
- Copies from host→device and device→host are concurrent if in different streams

Pinned memory

Pinned (or page-locked) memory will not be paged out to disk:

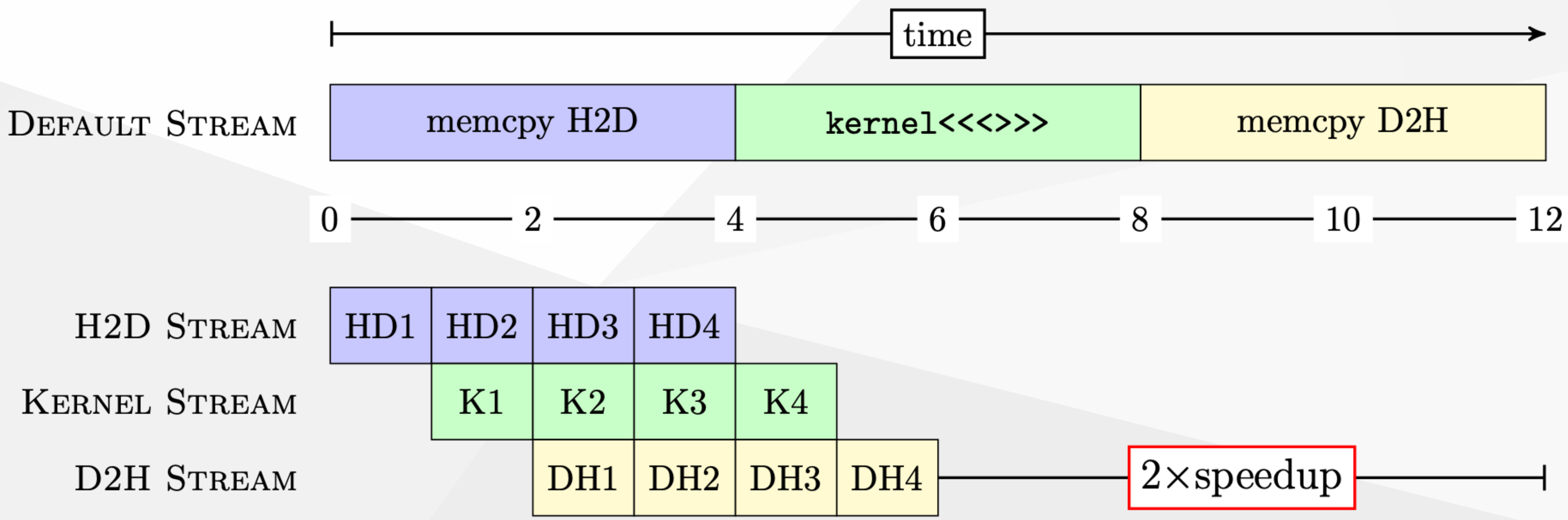
- The GPU can safely remotely read/write the memory directly without host involvement;
- Only use for transfers, because it easy to run out of memory

Managing pinned memory
`cudaMallocHost(**ptr, size);` and `cudaFreeHost(*ptr);`
Allocate and free pinned memory (`size` is in bytes)

Asynchronous Copy Example: Streaming Workloads

Computations that can be performed independently, e.g. our `axpy` example:

- Data in host memory has to be copied to the device, and the result copied back after the kernel is computed
- Overlap copies with kernel calls by breaking the data into chunks



CUDA Events

CUDA events can be used to coordinate operations on different GPU streams:

- Synchronize tasks in different streams, e.g.:
  - Don't start work in stream a until stream b has finished;
  - Wait until required data has finished copy from host before launching kernel
- Query status of concurrent tasks:
  - Has kernel finished/started yet?
  - How long did a kernel take to compute?

Managing Events

- Create and free `cudaEvent_t`
  `cudaEventCreate(cudaEvent_t*);` & `cudaEventDestroy(cudaEvent_t);`
- Enqueue an event in a stream

```
cudaEventRecord(cudaEvent_t, cudaStream_t);
```

- Make host execution wait for event to occur

```
cudaEventSynchronize(cudaEvent_t);
```

- Test if the work before an event in a queue has been completed

```
cudaEventQuery(cudaEvent_t);
```

- Get time between two events.

```
cudaEventElapsedTime(float*, cudaEvent_t, cudaEvent_t);
```

Using Events to Time Kernel Execution

```
cudaEvent_t start, end;
cudaStream_t stream;
float time_taken;

// initialize the events and streams
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaStreamCreate(&stream);

cudaEventRecord(start, stream); // enqueue start in stream
my_kernel<<<grid_dim, block_dim, 0, stream>>>();
cudaEventRecord(end, stream);   // enqueue end in stream
cudaEventSynchronize(end);      // wait for end to be reached
cudaEventElapsedTime(&time_taken, start, end);

std::cout << "kernel took " << 1000*time_taken << " s\n";

// free resources for events and streams
cudaEventDestroy(start);
cudaEventDestroy(end);
cudaStreamDestroy(stream);
```