Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Advanced CUDA

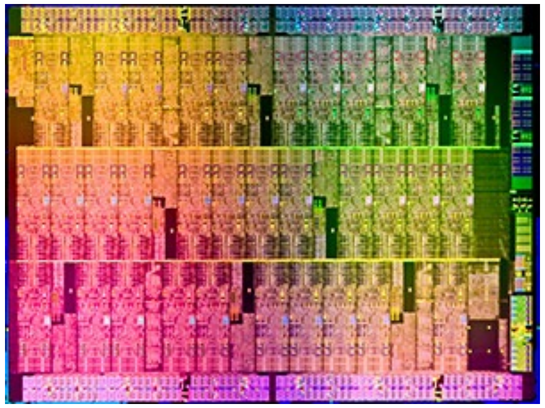
Lecture 11

István Reguly
reguly.istvan@itk.ppke.hu

Recap – modern achitectures

Multiple different components in modern computers

- Multi-core CPU
- Main memory (RAM)
- Graphic Processing Unit (GPU)
- Persistent storage (SSD, HDD)
- Network

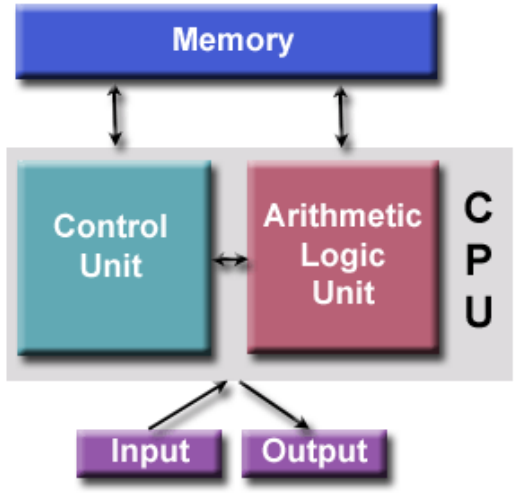


Recap - synchronicity

Neumann architecture – perfectly serial, 1 thing happening. In reality:

- Long pipelines (multiple instructions being executed)
- Multiple pipelines (different types of instructions)
- Multiple cores, etc.

You don’t think about this when programming C/C++



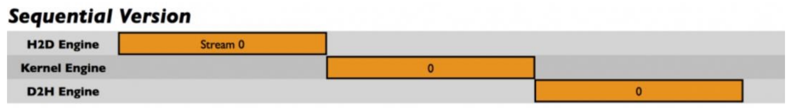
Once things become asynchronous, you really do have to!

Synchronicity in GPUs

When writing code for GPUs we have to think even more carefully, because:

- Our host code executes on the CPU(s);
- Our kernel code executes on the GPU(s)
  - . . . but when do the different bits take place?
  - . . . can we get better performance by being clever?
  - . . . might we get the wrong results?

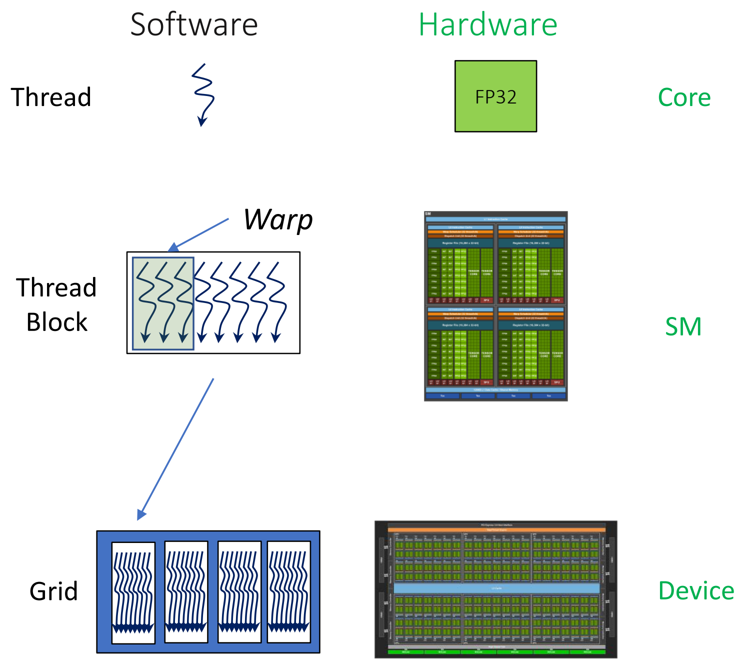
Example: copies between CPU-GPU and computations on the GPU



You need to understand what is going on, and when – then you can figure out how to do it better

Recap – GPU execution

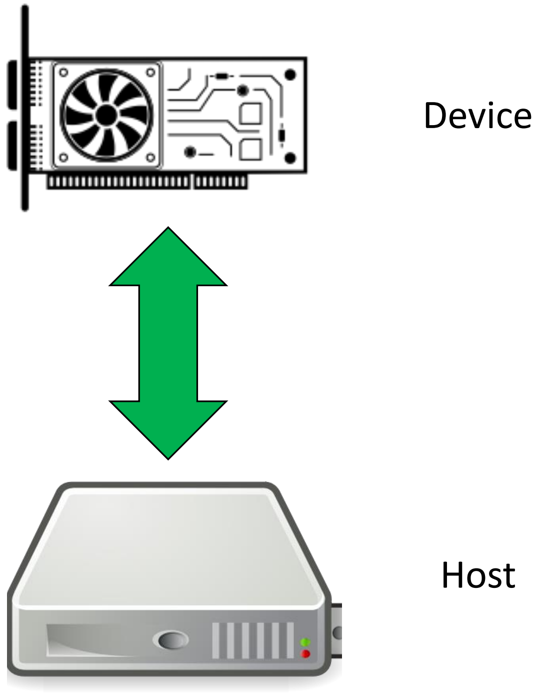
- We program from the perspective of threads
- Threads are in warps of 32, they are synchronous
- Different warps are asynchronous – can sync with __syncthreads() in the same block
- Different blocks of the grid execute independently, potentially overlapped



The simple case

The basic / simple / default behaviour in CUDA is that we have:

- 1x CPU.
- 1x GPU.
- 1x thread on CPU (i.e. scalar code).
- 1x “**stream**” on GPU (called the “**default stream**”).



Simple host code – blocking calls

Most CUDA calls are synchronous (often called “blocking”). An example of a blocking call is cudaMemcpy().

- Host call starts the copy (HostToDevice / DeviceToHost).
- Host **waits** until it the copy has finished.
- Host continues with the next instruction in the host code once the copy has completed.

Why do this???

- This mode of operation ensures correct execution from the CPU’s perspective
- For example it ensures that data is present if the next instruction needs to read from the data that has been copied...

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

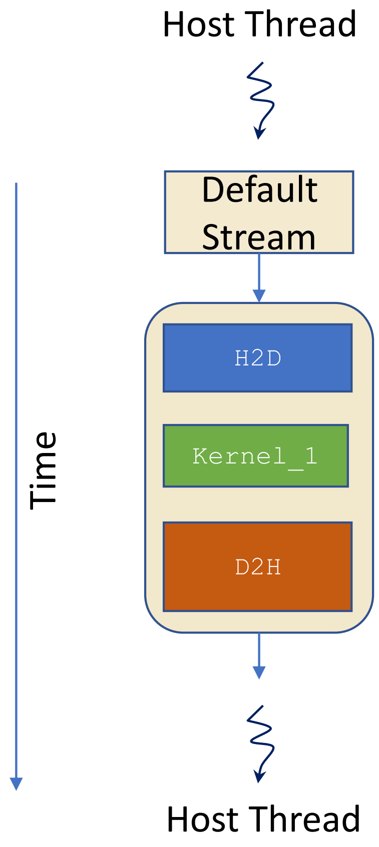
Simple host code – blocking calls

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

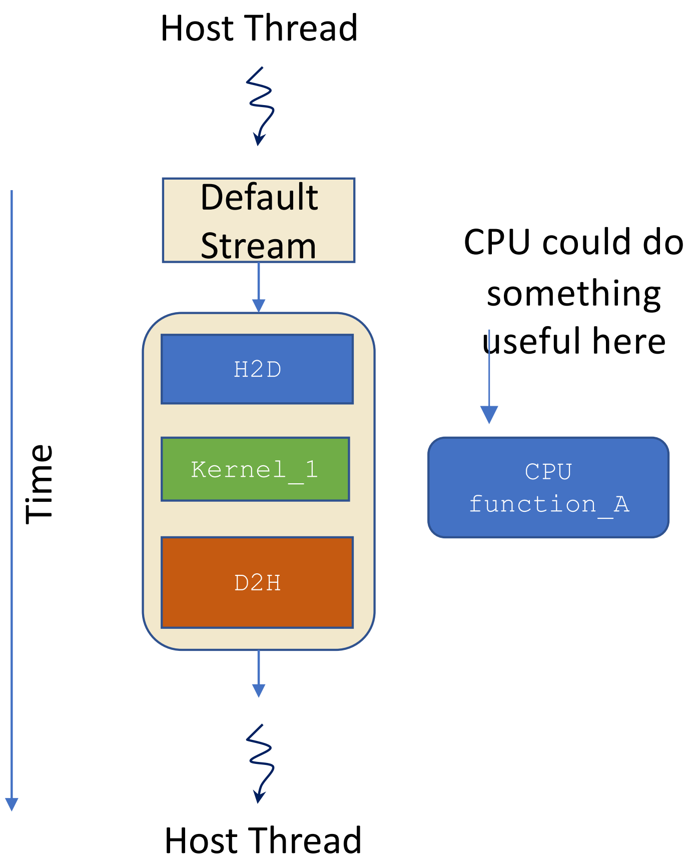


Non-blocking calls

In CUDA, kernel launches are asynchronous (often called “non-blocking”).

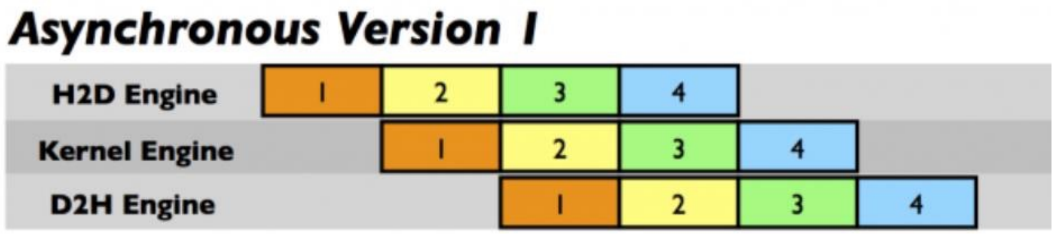
An example of kernel execution from host perspective:

- Host call starts the kernel execution.
- Host does not wait for kernel execution to finish.
- Host moves onto the next instruction



Non-blocking calls

- Another example of a non-blocking call is `cudaMemcpyAsync()`.
- This function starts the copy but CPU doesn't wait for completion.
- Synchronisation is performed explicitly from CPU, or with streams in GPU
- You must use page-locked memory (also known as pinned memory) – see Documentation.
- In both of our examples, the host eventually waits when at (for example) a `cudaDeviceSynchronize()` call.



Why do it? You can overlap computations and copy to/from device

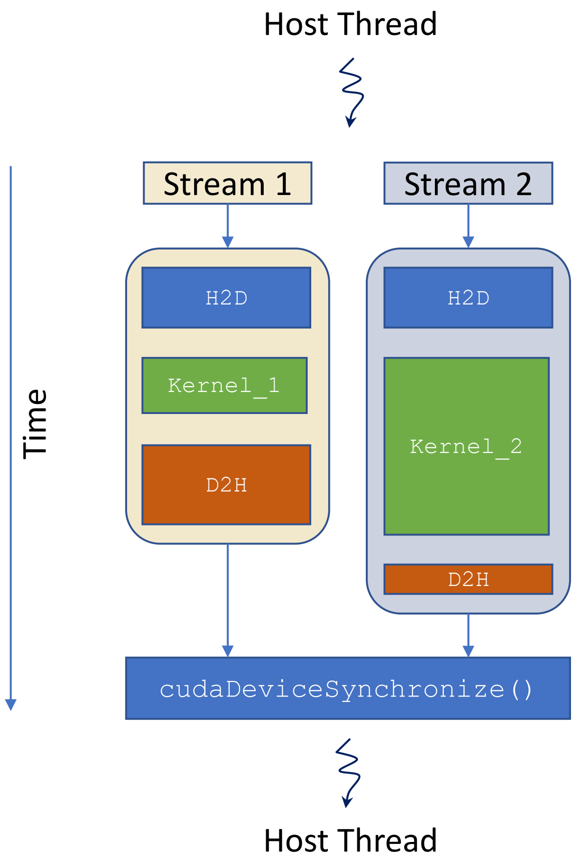
Common pitfalls

When using asynchronous calls, things to watch out for, and things that can go wrong are:

- Kernel/operation timing – need to make sure it's finished.
  - Even this can be done asynchronously (see CUDA Events)
- Could be a problem if the host uses data which is read/written directly by kernel, or transferred by `cudaMemcpyAsync().` (Think MPI non-blocking!)
- `cudaDeviceSynchronize()` can be used to ensure correctness (similar to syncthreads() for kernel code).
  - But it is a very coarse-grained tool – streams and events allow fine-grained synchronization

CUDA Streams

- A stream is a sequence of commands (*possibly issued by different host threads*) that execute in order.
- **Different streams**, on the other hand, may execute their commands **out of order** with respect to one another or concurrently.



Multiple CUDA Streams

When using streams in CUDA, you must supply a "stream" variable as an argument to:

- kernel launch
- cudaMemcpyAsync()

Which is created using `cudaStreamCreate();`

As shown over the last couple of slides:

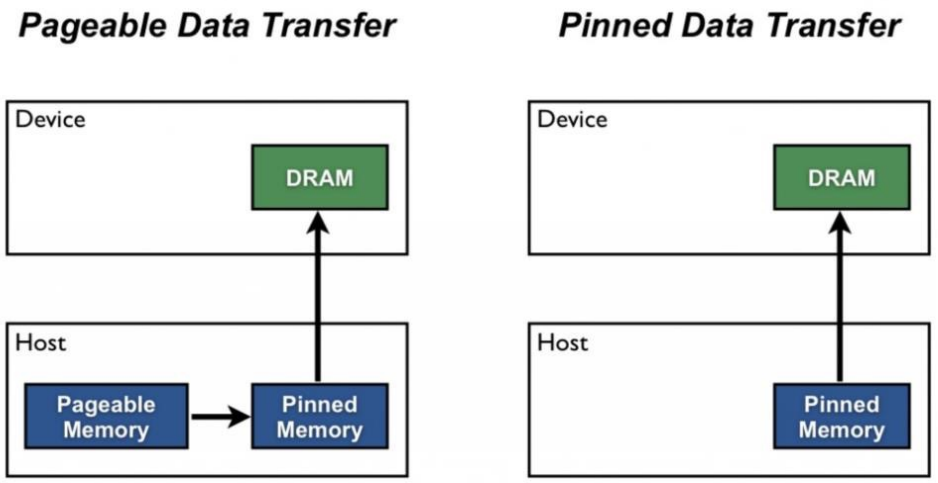
- Operations within the same stream are ordered - (i.e. FIFO – first in, first out) – they cant overlap.
- Operations in different streams are unordered wrt each other and can overlap.

Use multiple streams to increase performance by overlapping memory communication with compute

```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
my_kernel_one<<<blocks,threads,0,stream1>>>(…);
cudaStreamDestroy(stream1);
```

Page-locked or pinned memory

- Host memory is usually paged, so run-time system keeps track of where each page is located.
- For higher performance, pages can be fixed (fixed address space, always in RAM), but means less memory available for everything else.
- CUDA uses this for better host <–> GPU bandwidth, and also to hold arrays accessible by the device in host memory.
- Can provide up to 100% improvement in bandwidth
- You must use page-locked memory with `cudaMemcpyAsync();`
- Page-locked memory is allocated using `cudaHostAlloc()`, or registered by `cudaHostRegister();`



Default stream

- For legacy reasons (codes not using streams don’t break) the “default” stream (when you don’t specify a stream, or use 0) is synchronous to everything else
  - Operations in default stream do not overlap with operations in any other stream
  - no flag, or --default-stream legacy

- Or --default-stream per-thread
- This forces new (good) behaviour in which the default stream doesn’t affect the others.

Stream commands

Some useful stream commands are:

- `cudaStreamCreate(&stream)`
  - Creates a stream and returns an opaque “handle” – the “stream variable”.
- `cudaStreamSynchronize(stream)`
  - Waits until all preceding commands have completed.
- `cudaStreamQuery(stream)`
  - Checks whether all preceding commands have completed.
- `cudaStreamAddCallback()`
  - Adds a callback function to be executed on the host once all preceding commands have completed.

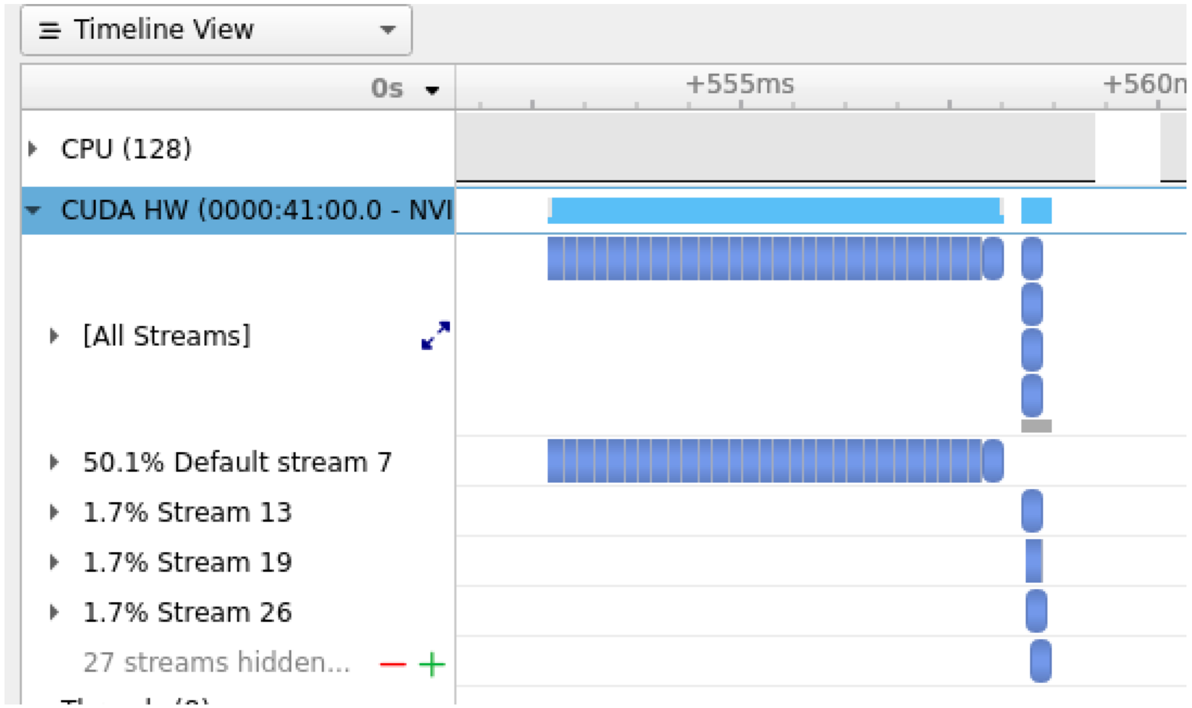
Exercise

- See kernel_overlap.cu
  - Launches kernels in different streams that can overlap for much faster execution
  - Run through visual profiler: `nvvp ./kernel_overlap &`
- See work_streaming.cu
- All the data is processed in one kernel
  - One large copy to GPU
  - One large kernel
  - Once large copy from GPU
- Split the data up into 8 blocks, do the copy-kernel-copy in a different stream for each block. Confirm with `nsys`

Visual profiler

```
srun … nsys profile ./overlap
then launch nsys-ui and open the report
```

Zoom selecting region, then right click and “zoom in on selection”



GUI needs:

- connection with MobaXTerm OR
- Download profile & launch nsys locally

Stream commands

As previously shown, each stream executes a sequence of CUDA calls. However to get the most out of your heterogeneous computer you might also want to do something on the host.

There are at least two ways of coordinating this:

Use a separate thread for each stream

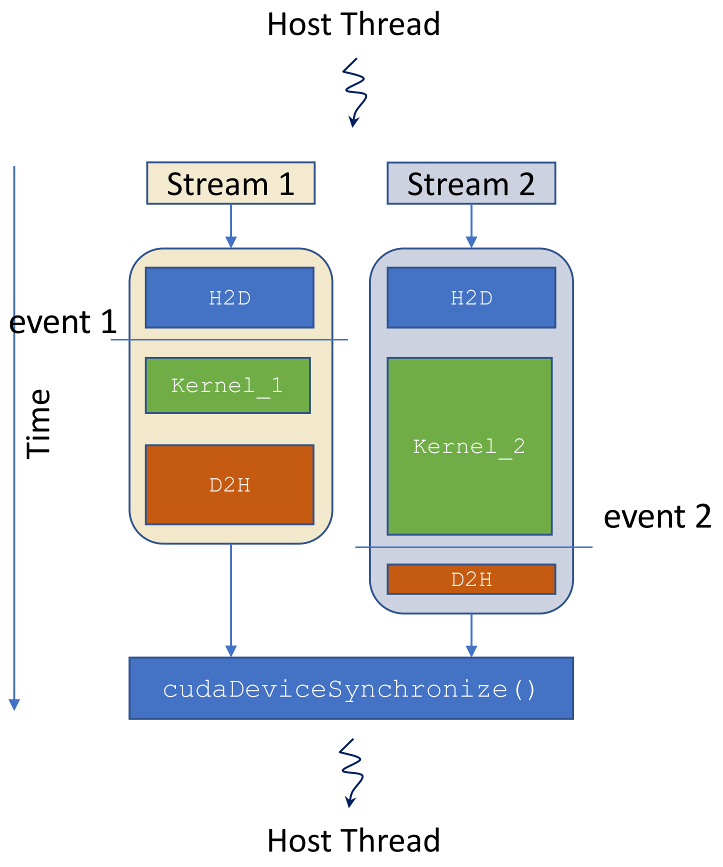
- It can wait for the completion of all pending tasks, then do what’s needed on the host.

Use just one thread for everything

- Use events to monitor the progress of kernels and streams
- Use callbacks…

Streams and Events

- Events are placeholders that can be inserted into streams
- No cost on GPU
- Can be used to synchronize between CPU+GPU or GPU+GPU
- E.g. can have stream 1 wait for event 2 in stream 2: will block stream 1 until event 2 happens



Stream commands

- Functions useful for synchronisation and timing between streams:
- `cudaEventCreate(event)`
  - Creates an “event”.
- `cudaEventRecord(event,stream)`
  - Puts an event into a stream (by default, stream 0).
- `cudaEventSynchronize(event)`
  - CPU waits until event occurs.
- `cudaStreamWaitEvent(stream,event)`
  - Stream waits until event occurs (doesn’t block the host).
- `cudaEventQuery(event)`
  - Check whether event has occurred.
- `cudaEventElapsedTime(time,event1,event2)`
  - Times between event1 and event2.

Multiple devices

What happens if there are multiple GPUs?

CUDA devices within the system are numbered, not always in order of decreasing performance!

- By default a CUDA application uses the lowest number device which is “visible” and available (this might not be what you want).
- Visibility controlled by environment variable `CUDA_VISIBLE_DEVICES`.

```
CUDA_VISIBLE_DEVICES=1,2 ./kernel_overlap
```
<- use devices 1 and 2

- The current device can be chosen/set by using `cudaSetDevice()`
- `cudaGetDeviceProperties()` does what it says, and is very useful.
- Each stream is associated with a particular device, which is the “current” device for a kernel launch or a memory copy.
- see simpleMultiGPU example in SDK or section 3.2.6 for more information

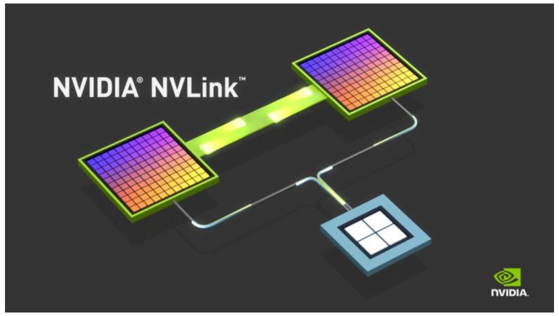
Exercise

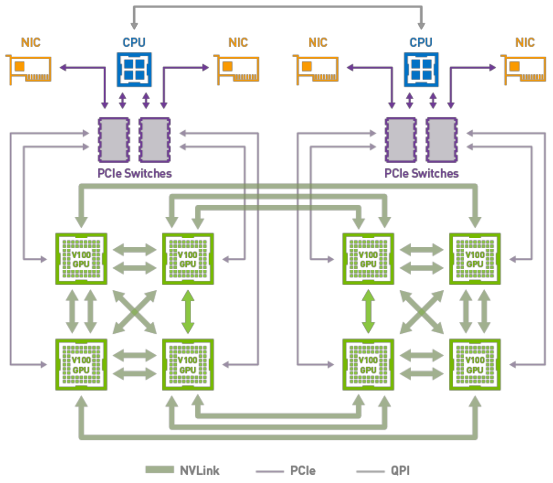
- Modify kernel_overlap and work_streaming to work on different GPUs (each stream on its own GPU – Komondor nodes have 4 GPUs per node)
  - In the loop over streams, always select the GPU
- See what happens using the nsys profiler

```
srun -p gpu --gres=gpu:4 --ntasks=1 --time=00:05:00 --mem=40G ./a.out
```

Stream commands

- If a user is running on multiple GPUs, data can go directly between GPUs (peer – peer), it doesn't have to go via CPU.
- This is the premise of the NVlink interconnect, which is much faster than PCIe (~300GB/s P2P).
- `cudaMemcpy()` can do direct copy from one GPU's memory to another.
- A kernel on one GPU can also read directly from an array in another GPU's memory, or write to it this even includes the ability to do atomic operations with remote GPU memory.
- For more information see Section 4.11, "Peer Device Memory Access" in CUDA Runtime API documentation: https://docs.nvidia.com/cuda/cuda-runtime-api/





- Previous homework due next week!