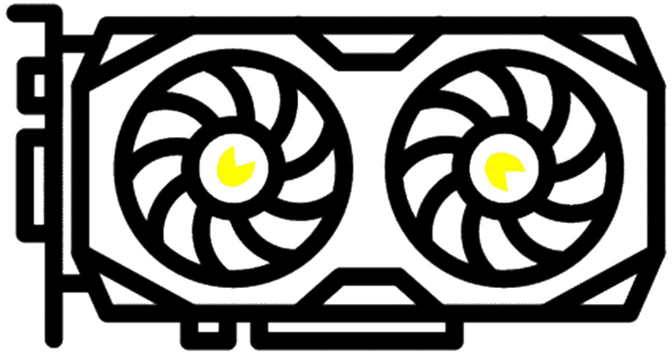
University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

member of swissuniversities

Parallel Computing

Communication Patterns

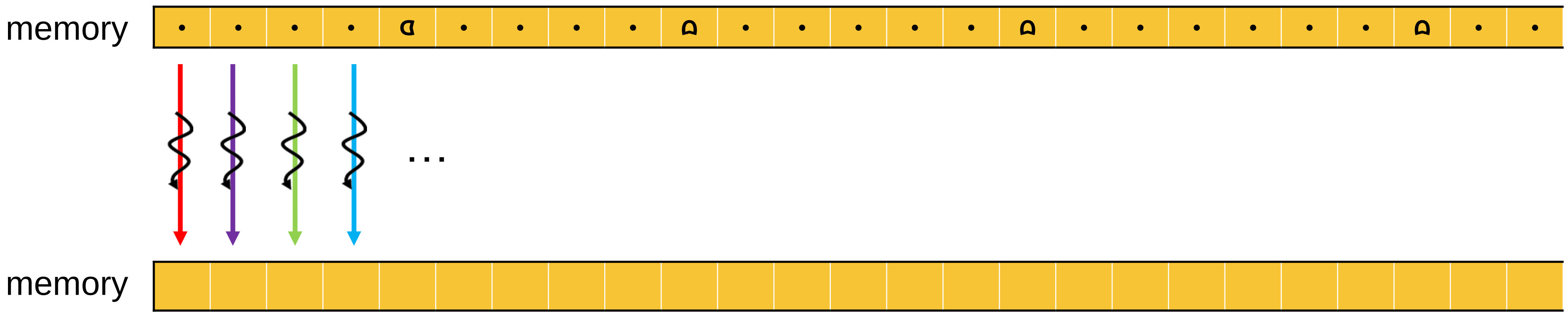


Communication Patterns

- Communication within the CUDA execution model (also on other massive parallel processors) is only supported via sharing data
- Understanding the communication pattern(s) of an algorithm is essential to build fast applications
- It is all about **how threads access memory**

Communication Patterns - Map

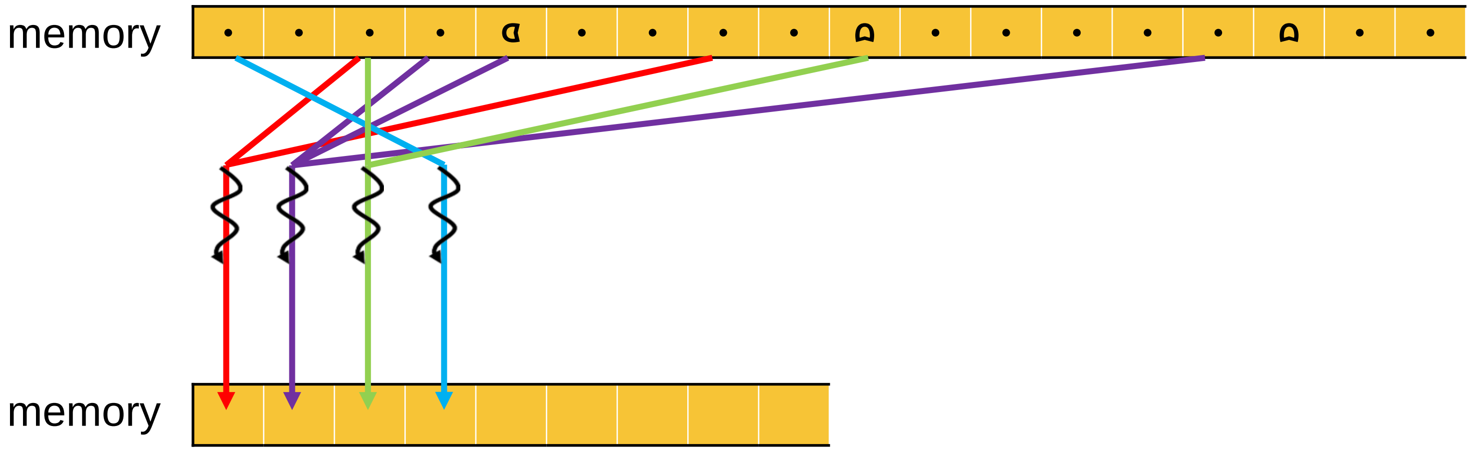
- Very easy to implement – each thread dos exactly one element - its globalID
- One-to-One



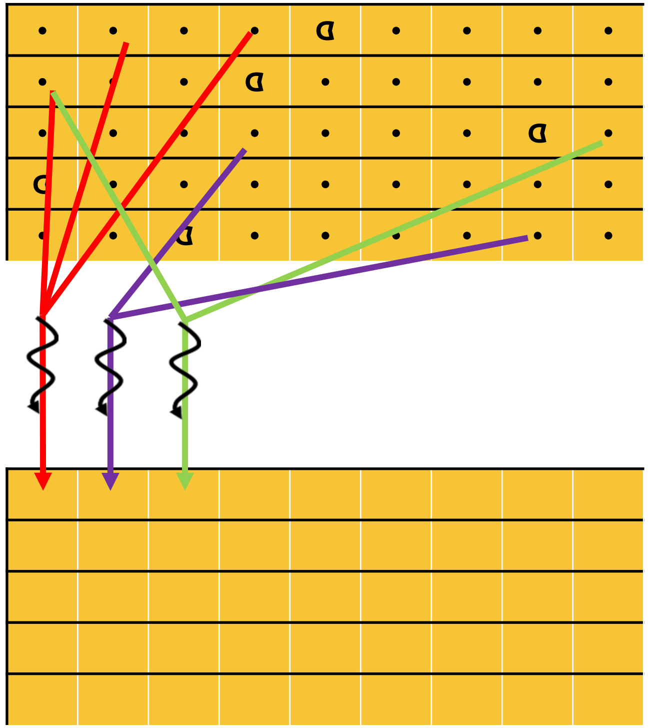
- Using a Stride-Looping Pattern such as the Grid-Stride Loop,
  there will be more than 1 item per thread. But it still is a map communication pattern.
  --> True also for all other patterns.

Communication Patterns - Gather

- Each thread gathers inputs together to create one output
- No access pattern in each neighborhood
- Many-to-One

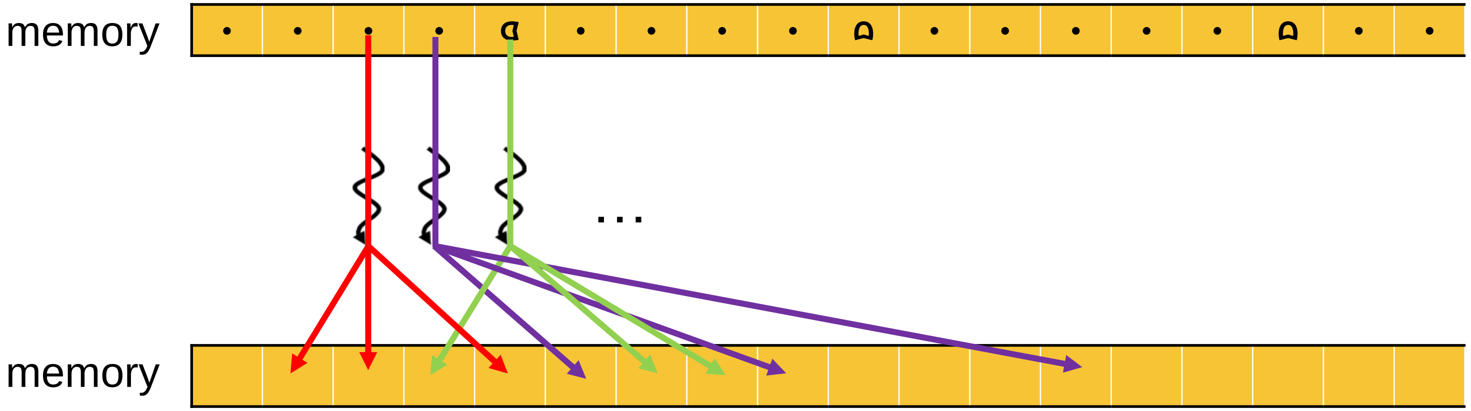


2D example:

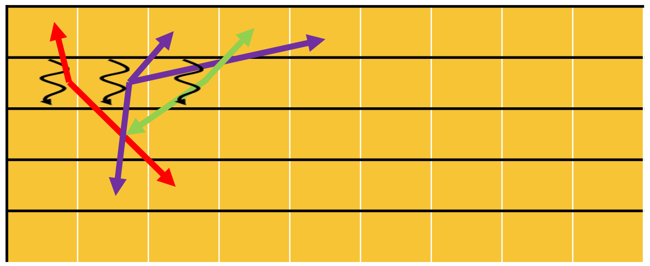


Communication Patterns - Scatter

- Each thread writes to multiple outputs – scatter the result over the memory
- E.g.: adding a fraction of the input value to different outputs
- One-to-Many --> Attention: race conditions

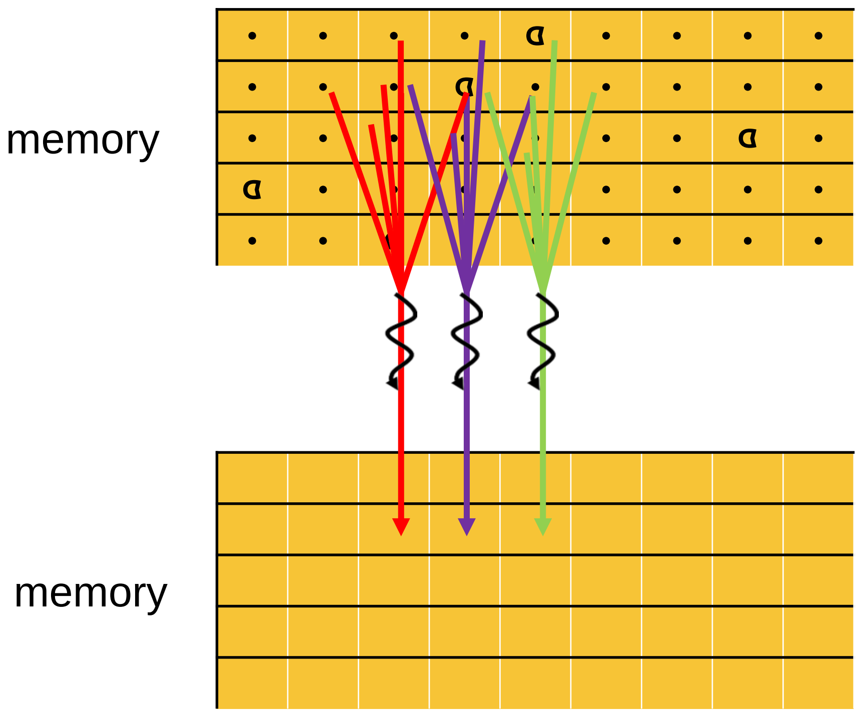


2D example (in place):

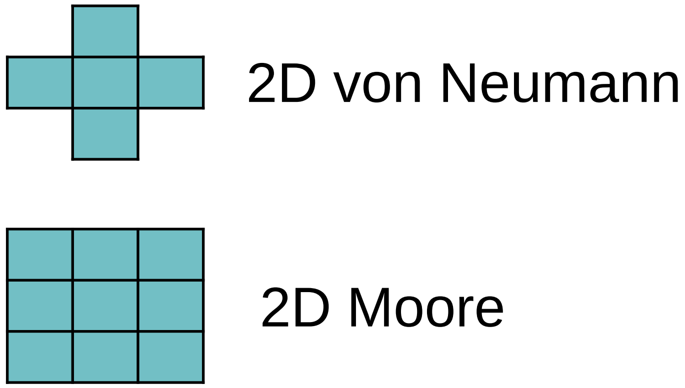


Communication Patterns - Stencil

- Each thread gathers inputs together **from a fixed neighborhood** to create one output
- Massive data reuse – due to pattern in neighborhood
- Specialized Gather – Several-to-One

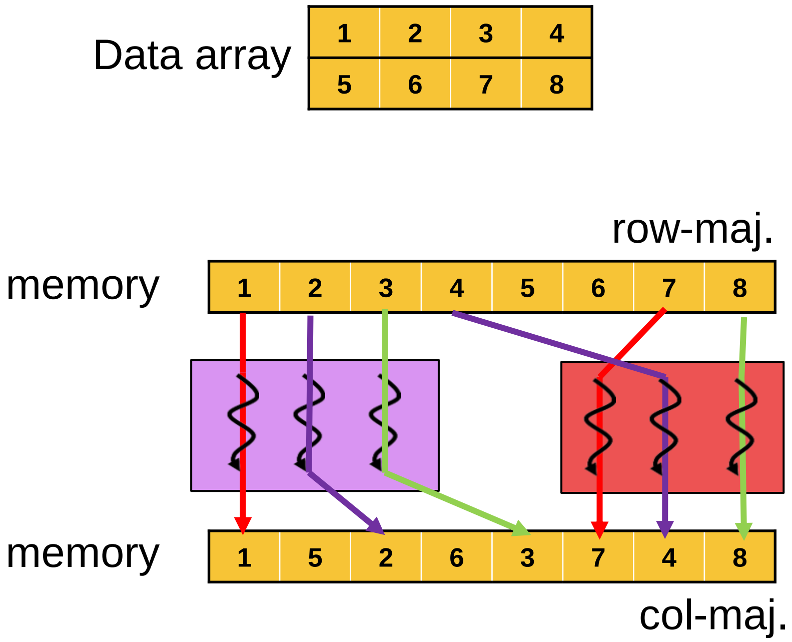


2D stencil examples:

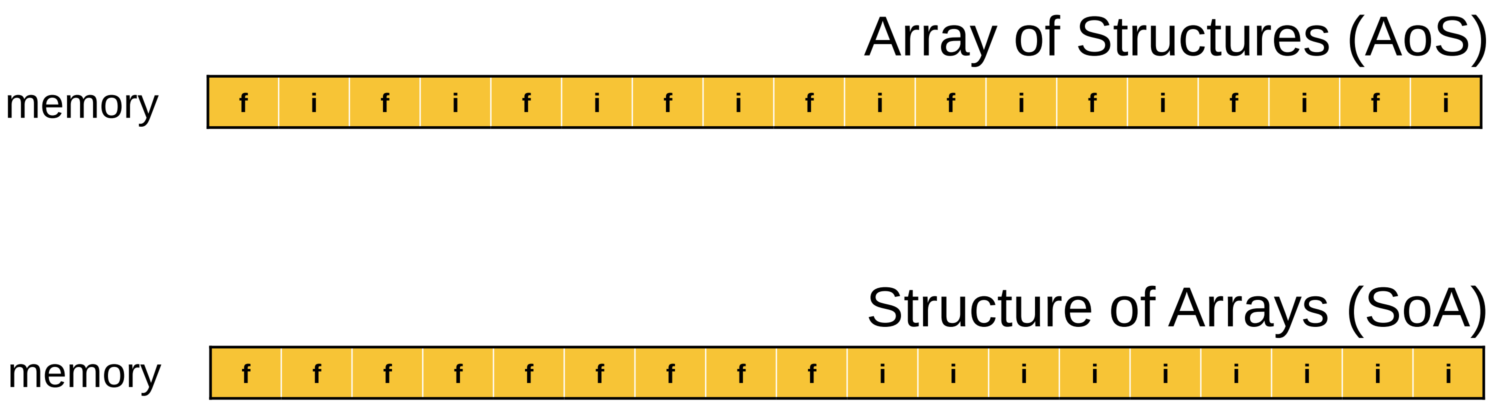


Communication Patterns - Transpose

- Threads reorder data elements in memory
- Can be done as Scatter or Gather
- One-to-One




```
struct pacStruct {
    float f;
    uint32_t i;
};
```



Communication Patterns – Quiz (yay!)

```
// Kernel code – did some init and magic – 2D thread block
x = threadIdx.x; y = threadIdx.y;

output[x] = 13.37 * input[x]
// -->


output[x + y * 64] = input[y + x * 64]
// -->
```

Communication Patterns – Quiz (yay!)

```
// Kernel code – did some init and magic – 2D thread block
x = threadIdx.x; y = threadIdx.y;

if (x % 2 == 0) {

  output[x – 1] += 13.37 * input[x];
  output[x + 1] += 13.37 * input[x];
  // -->

  output[x] = (input[x - 1] + input[x] + input[x + 1]) / 3.0;
  // -->

}
```

Grid Stride Loop [1]

- How do we specify (and optimize) the total number of threads?
  - Use 1 thread per data point (either input or output)
  - Use a more dynamic approach to play around with this number
- Creating thread blocks comes with some costs
  - Thread creation and destruction
  - Shared memory and private memory initialization
- Use a Grid-Stride loop to reuse threads
  - Flexible kernel supporting anything from 1 thread to max threads
  - Balance of load on SMs
  - Save some overhead costs

[1] Full credits to Nvidia Blog: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops

Grid Stride Loop [1]

- Classic approach for vector addition kernel

```
__global__ void vectorAdd(float *vecA, int *vecB, int *vecC, int size) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < size)
      vecC[idx] = vecA[idx] + vecB[idx];
}
```

- Iterate with the total size of your grid (all threads) over the data --> grid size loop

```
__global__ void vectorAdd(float *vecA, int *vecB, int *vecC, int size) {
   for (int idx = blockIdx.x * blockDim.x + threadIdx.x;
        idx < size;
        idx += blockDim.x * gridDim.x)
      {
        vecC[idx] = vecA[idx] + vecB[idx];
      }
}
```

[1] Full credits to Nvidia Blog: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops

Grid Stride Loop [1]

- Still have a perfect memory coalesced access pattern
- Change the code to a serial execution for debugging

```
VectorAdd<<<1,1>>>(deviceVecA, deviceVecB, deviceVecC, size);
```

- Balance SM utilization by launch a multiple number of blocks of the SMs

```
int numSMs;
cudaDeviceGetAttribute(&numSMs, cudaDevAttrMultiProcessorCount, deviceId);
VectorAdd<<<8*numSMs,1024>>>(deviceVecA, deviceVecB, deviceVecC, size);
```

- If `size==totalThreads` the loop has more or less the same cost as the if statement

[1] Full credits to Nvidia Blog: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops

08_GridStride

Task:

- Implement the GPU kernel `cudaAdd2DGridStride`

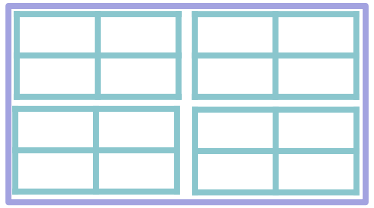
Link:

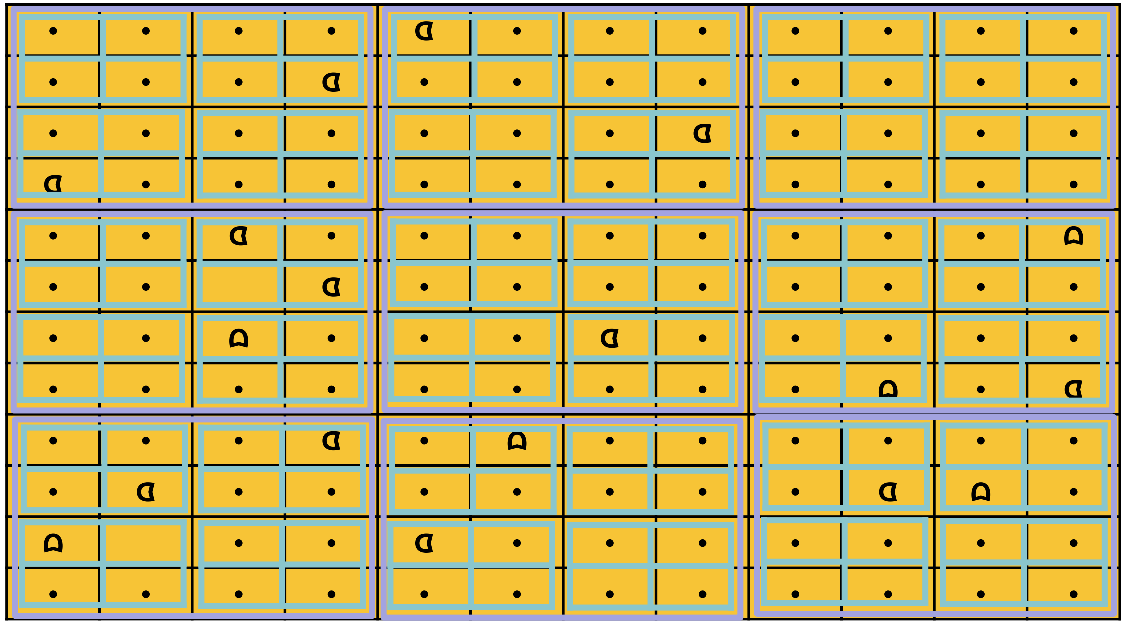
- https://classroom.github.com/a/tfsHe-Ok

Goal:

- Learn how to use the GridStride loop concept
- Learn how to write flexible kernels
- Extend the 1D concept to 2D data structures

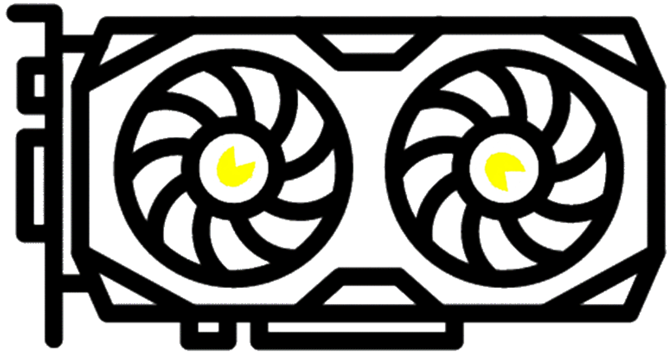
`<<dim3(2,2),dim3(2,2)>>>`
--> needs 9 iterations of grid size





University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

member of swissuniversities



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

```
packKernel<<< blocks, threads, SHMbytes >>>();            // default stream
packKernel<<< 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

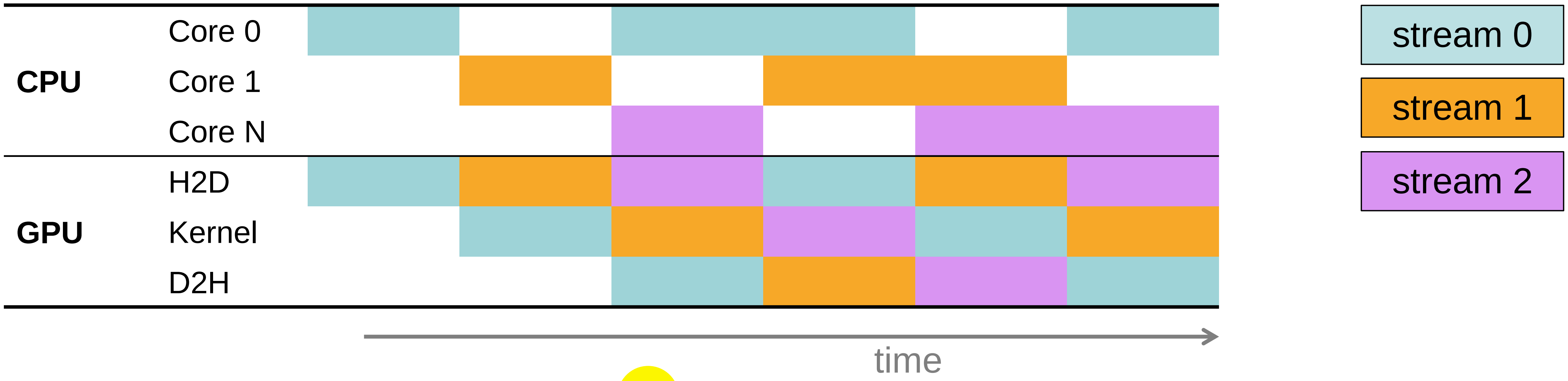
- Since everything is executed async in respect to the host, we need to synchronize
- `cudaDeviceSynchronize()` wait until all previous issued operations **in all streams** are done
- `cudaStreamSynchronize(stream)`
  wait until all previous issued operations **in this stream** are done
- `cudaStreamQuery(stream)` check if all operations in this stream are finished
- `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?



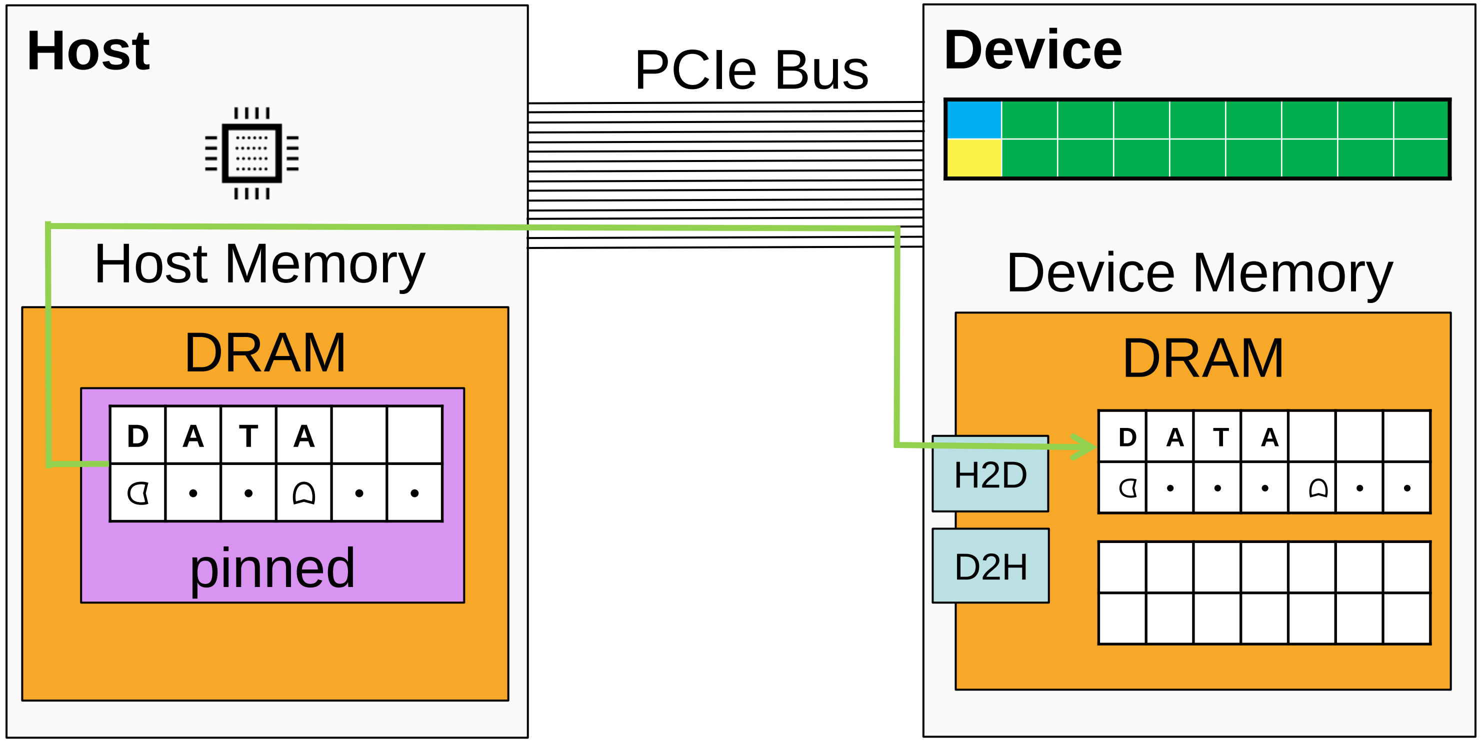
CUDA Processing Flow using async data transfer

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

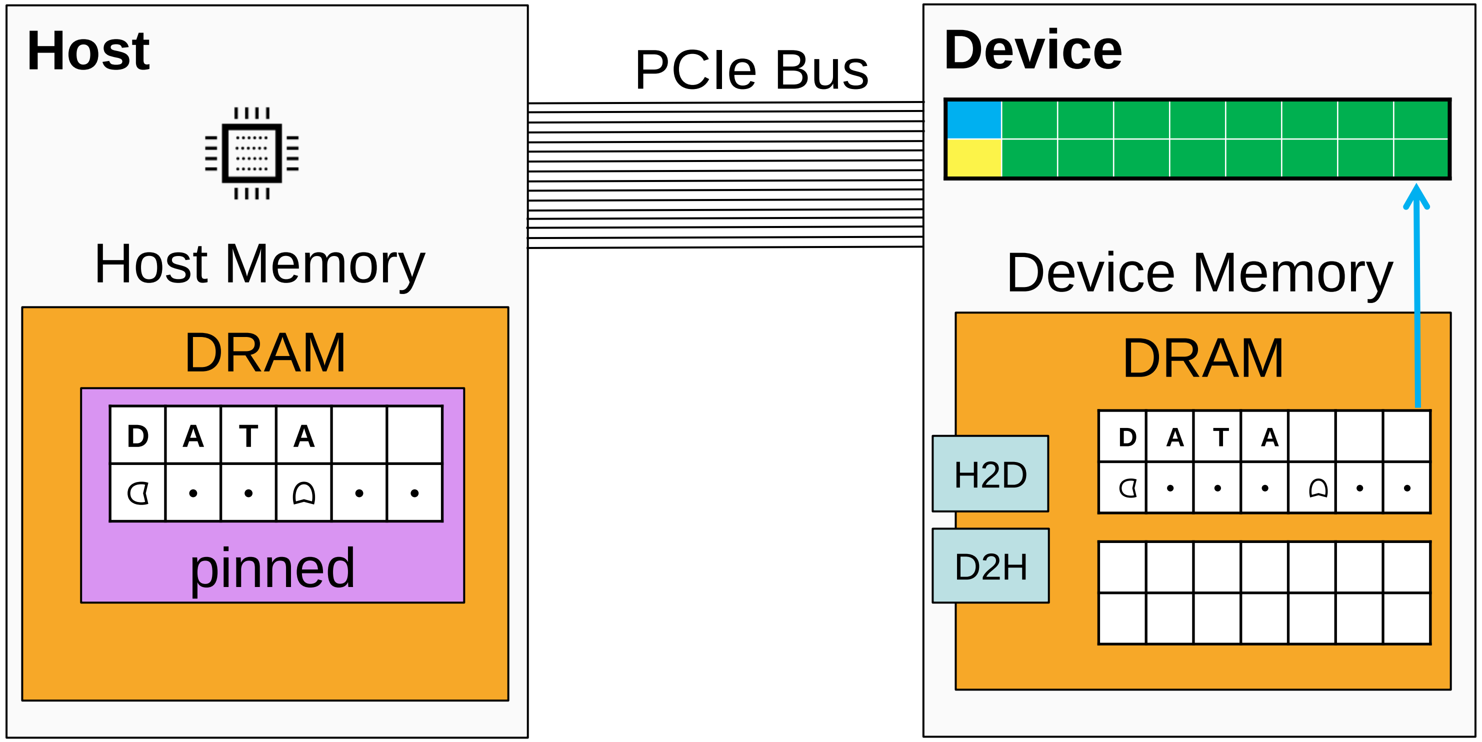
3. Execute kernel
   - GPU load (kernel engine)

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

4. Same way back using async D2H :)

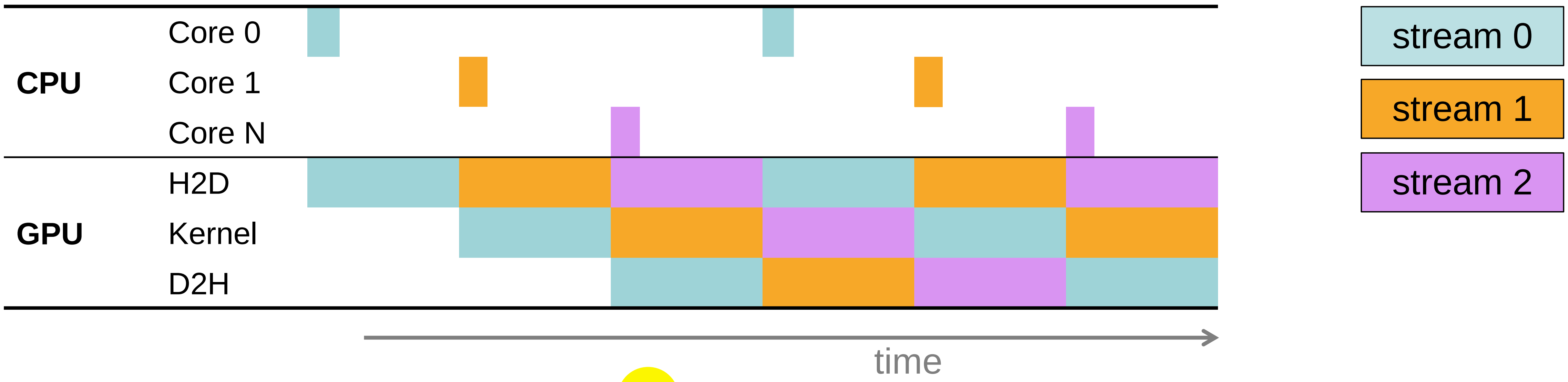
5. Clean up your mess

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

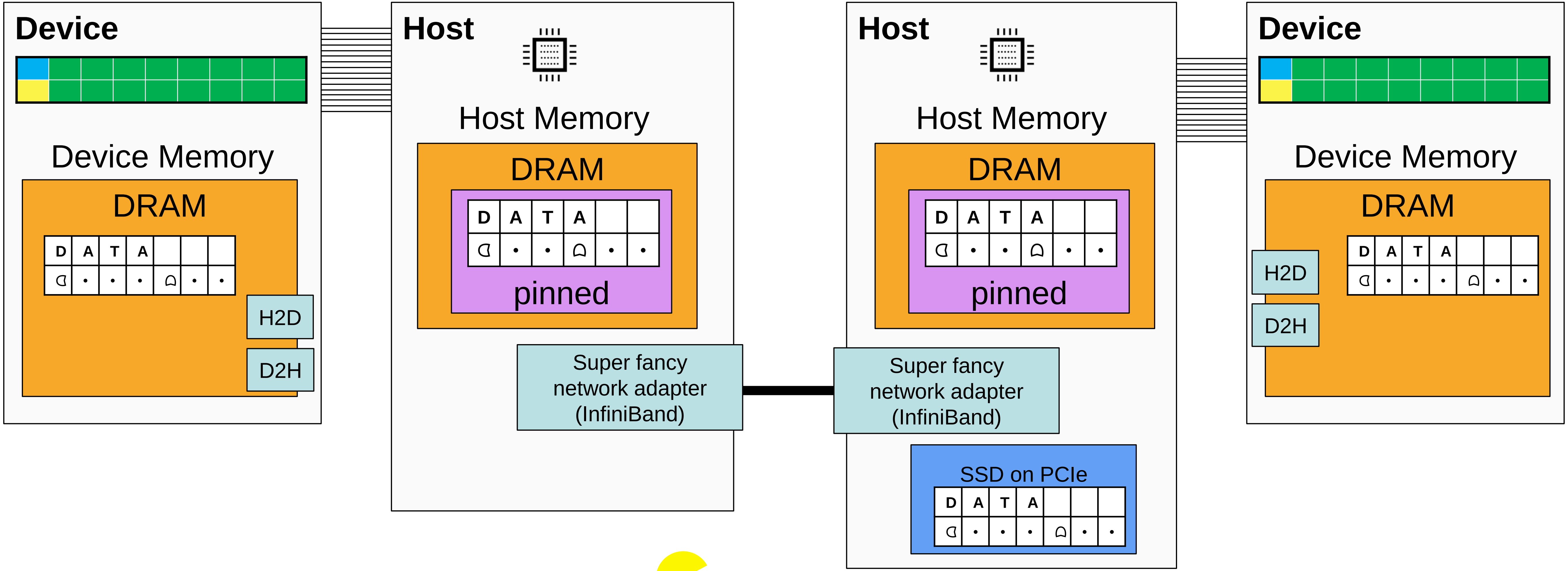


Observations

- 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