Computing beyond single GPU

Filippo Spiga | fspiga@nvidia.com

Contents

DISCLAIMER

(and forgive my drawings)

The following illustrations are indicative of the internals mechanisms of a GPU.

Several simplifications in terminology and assumptions in execution behavior are made and there is no time to explain them all.

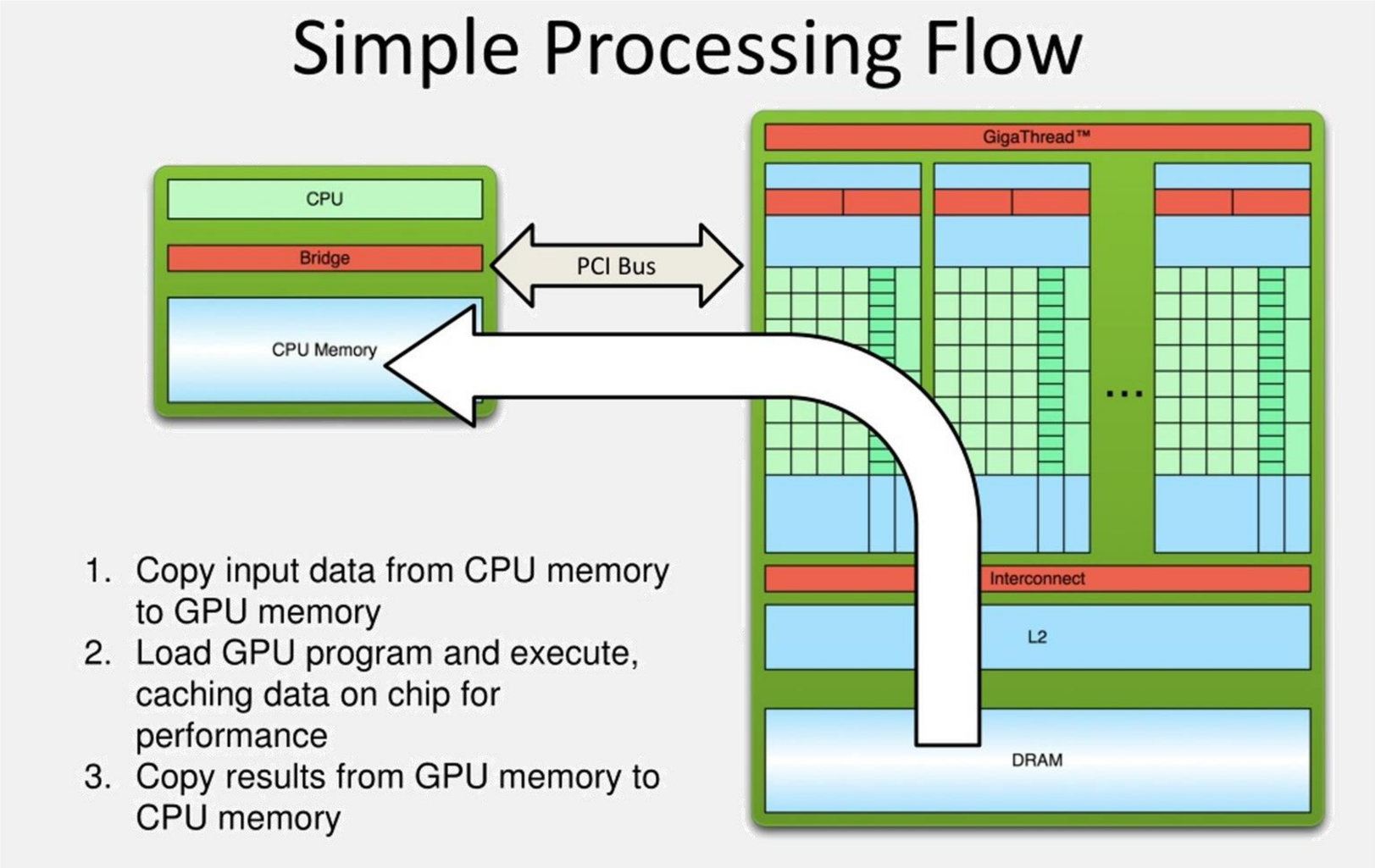
Things in practice are a bit more complicated than this (-:

How a GPU works?

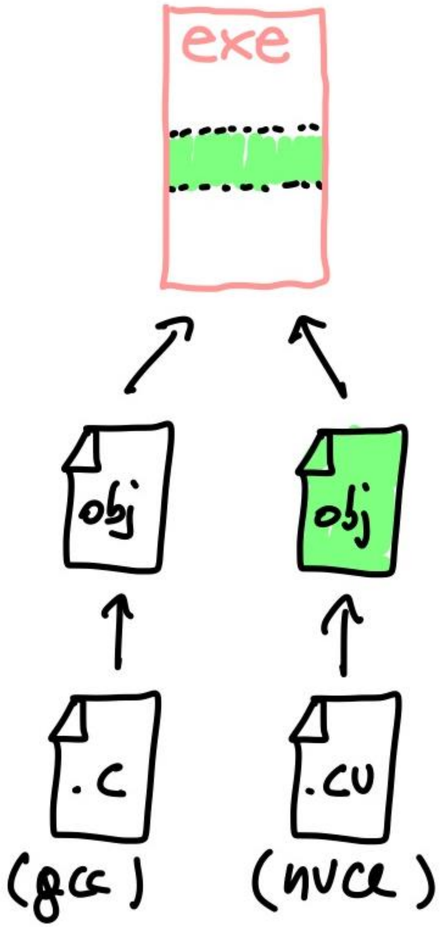
““A GPU’s scheduler is like an air traffic controller with a never-ending queue—somehow, thousands of flights land smoothly, but nobody’s quite sure how it all avoids a crash.”
(AI generated)

The most simple and fundamental flow

GPU needs a CPU as **control**



How you build CPU and GPU code…

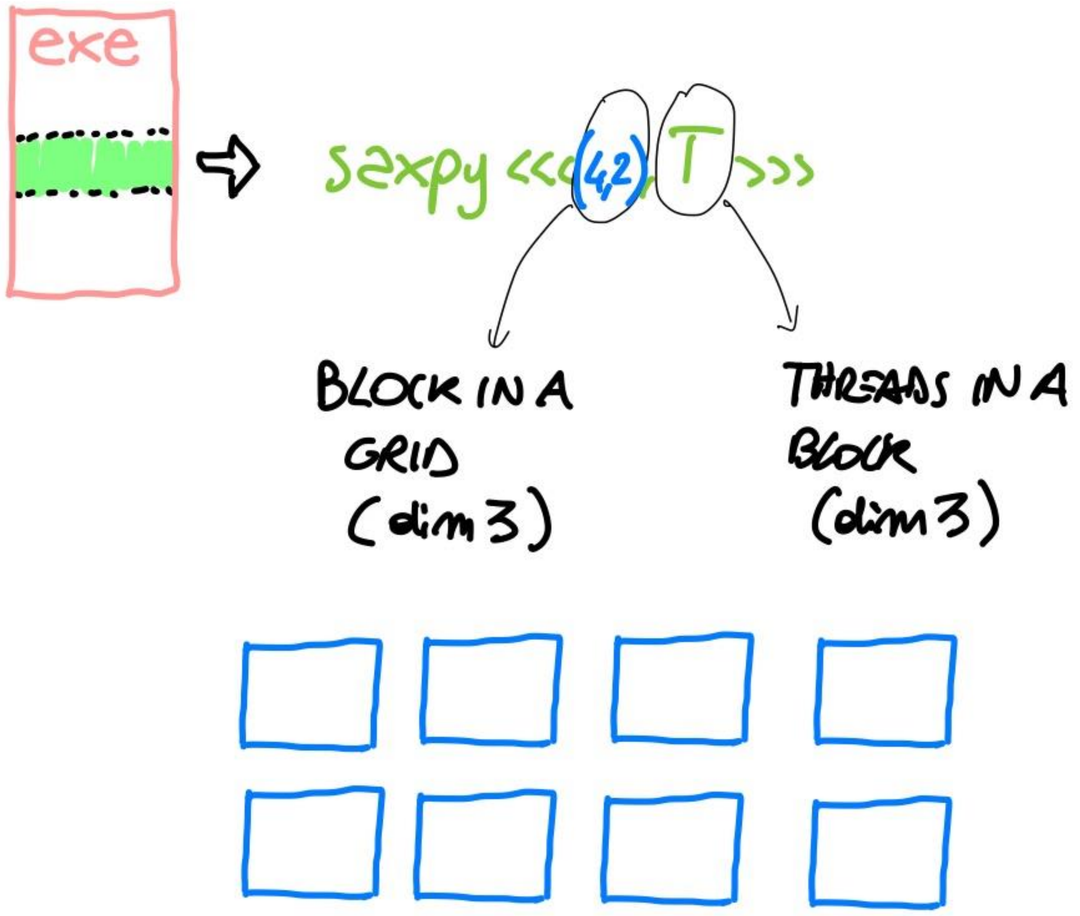


CUDA in one slide

CUDA (Compute Unified Device Architecture) is more than just a language

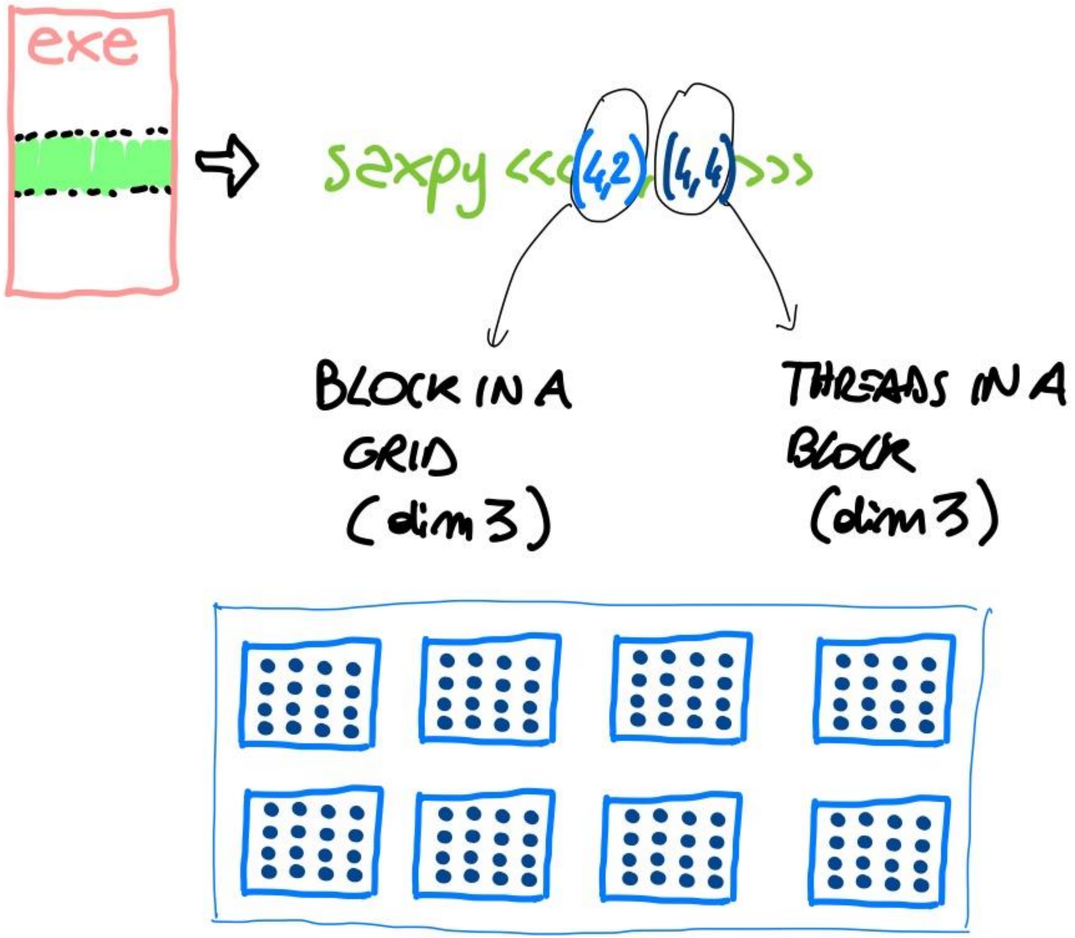
CUDA in one slide

CUDA (Compute Unified Device Architecture) is more than just a language



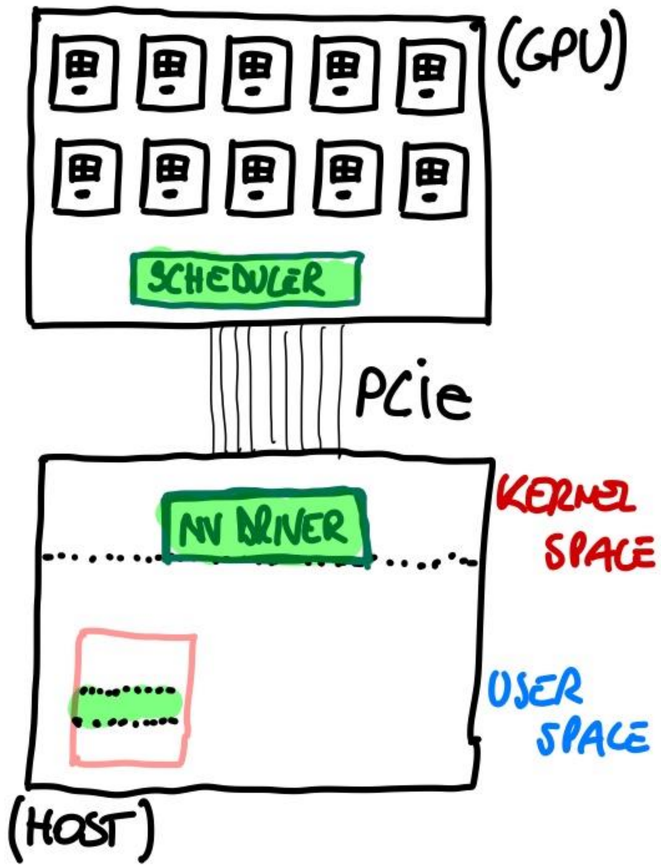
CUDA in one slide

CUDA (Compute Unified Device Architecture) is more than just a language



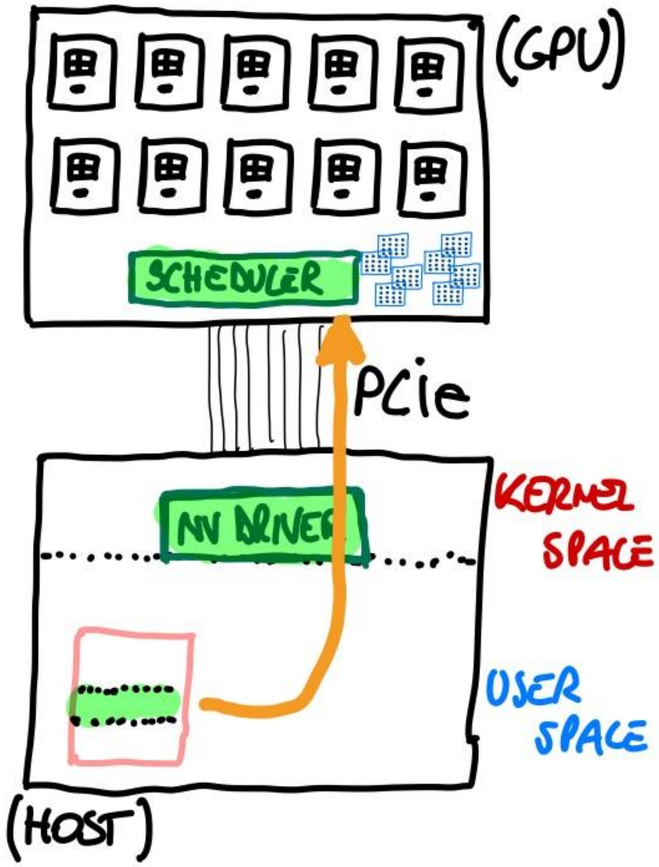
How does the GPU run it?

(let ignore the data movement)



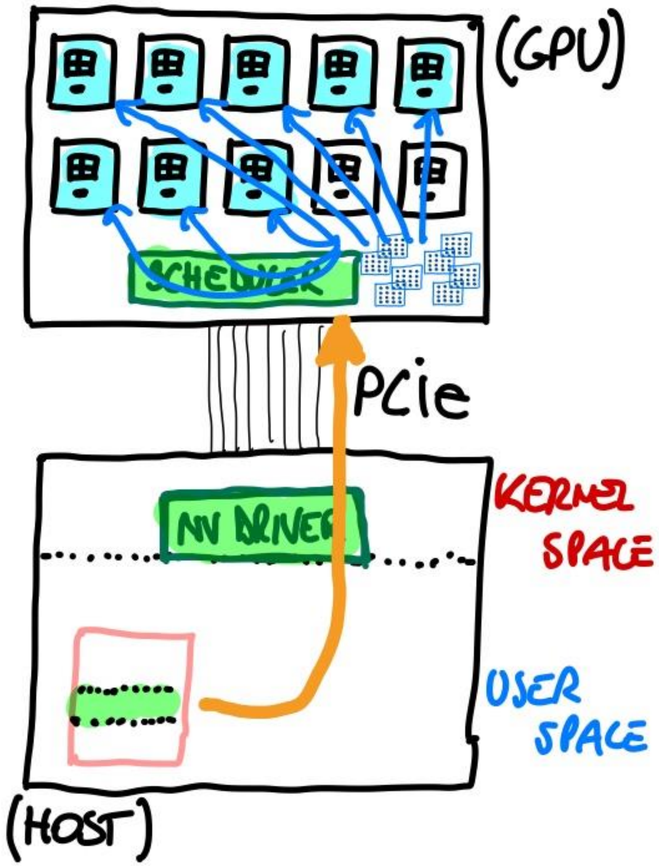
How does the GPU run it?

(let ignore the data movement)



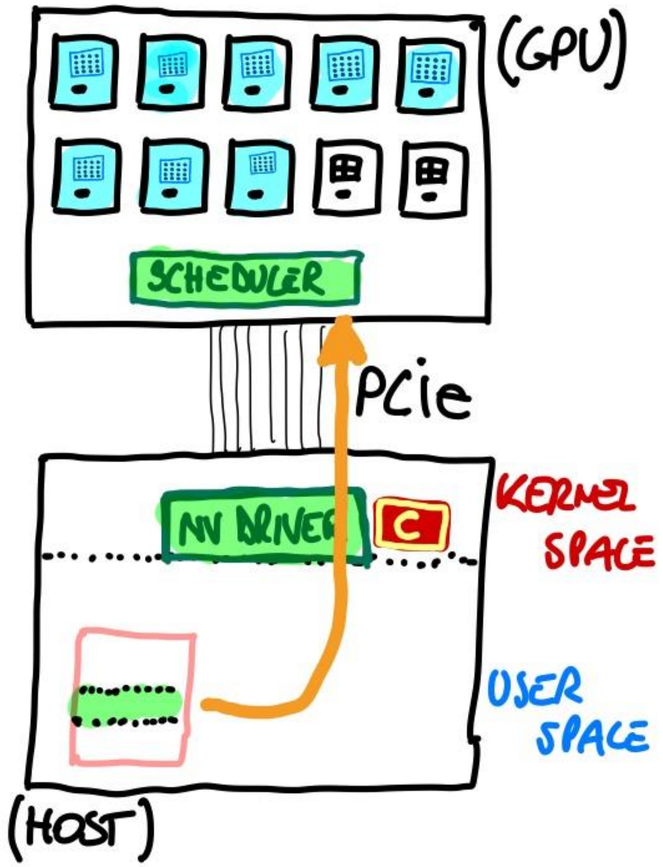
How does the GPU run it?

(let ignore the data movement)



How does the GPU run it?

(let ignore the data movement)



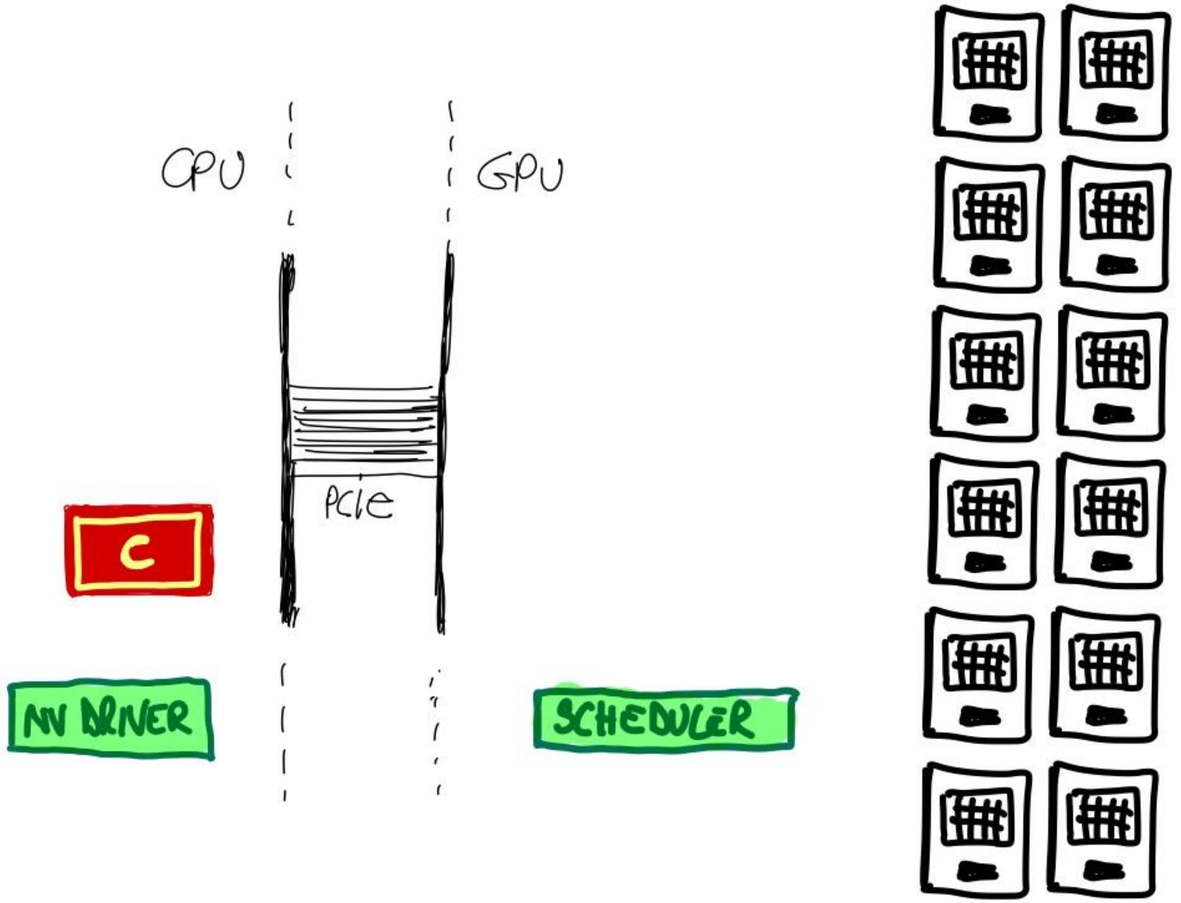
Every process creates its own **GPU CONTEXT**

The context is a stateful object required to run CUDA

Automatically created for you when using the CUDA runtime API

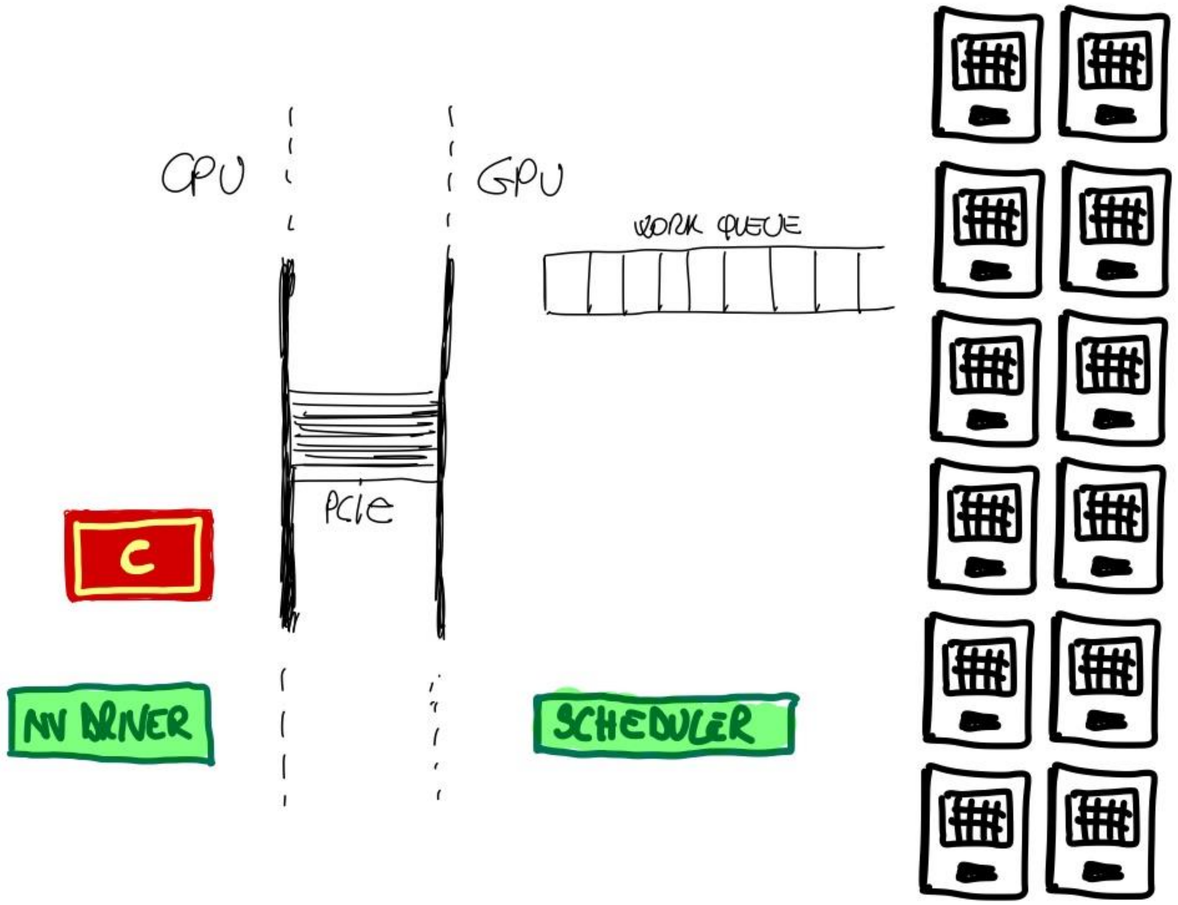
Running on GPU

GPU CONTEXT: the only gateway to GPU resources



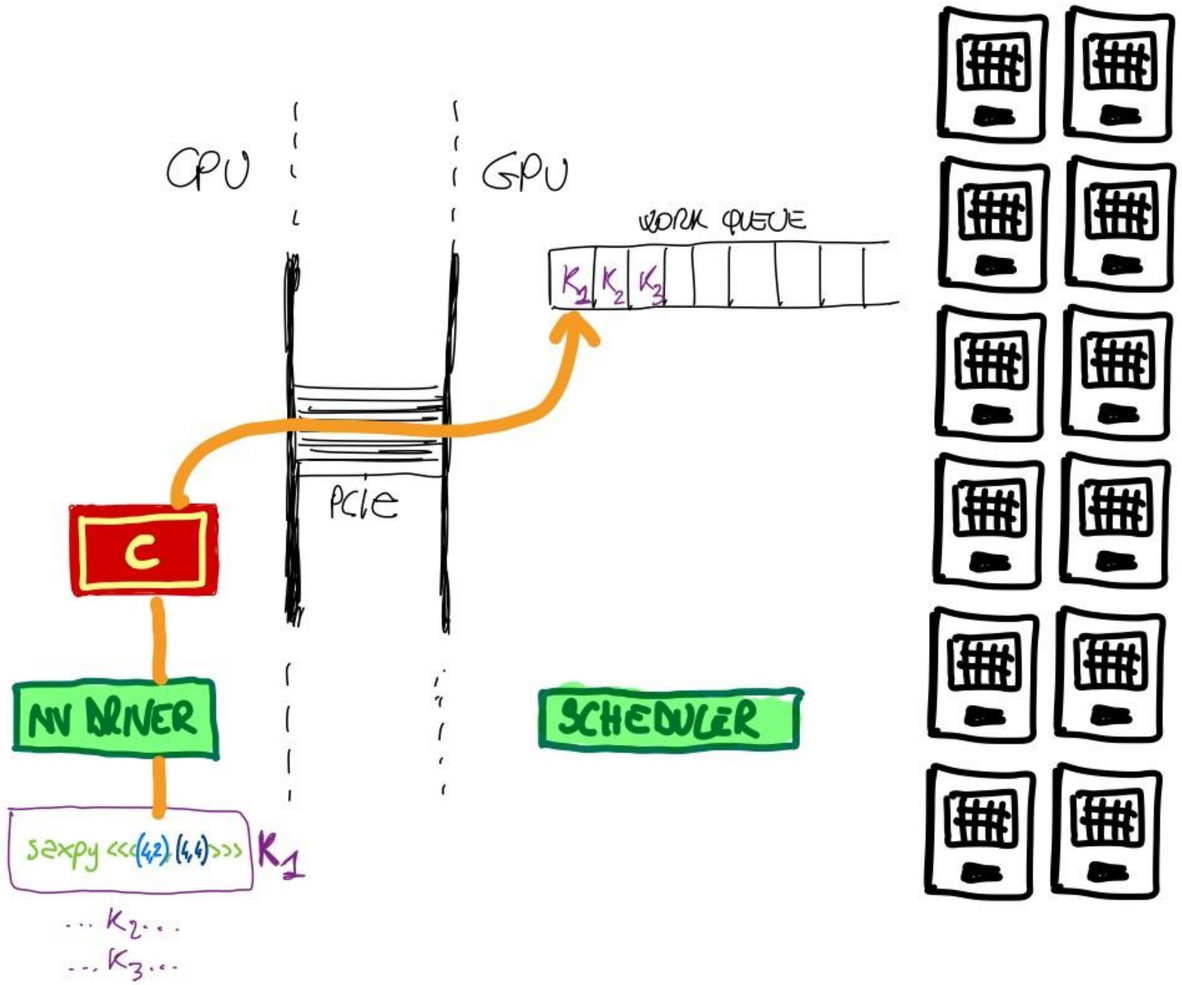
Running on GPU

How work is distributed by the GPU scheduler?



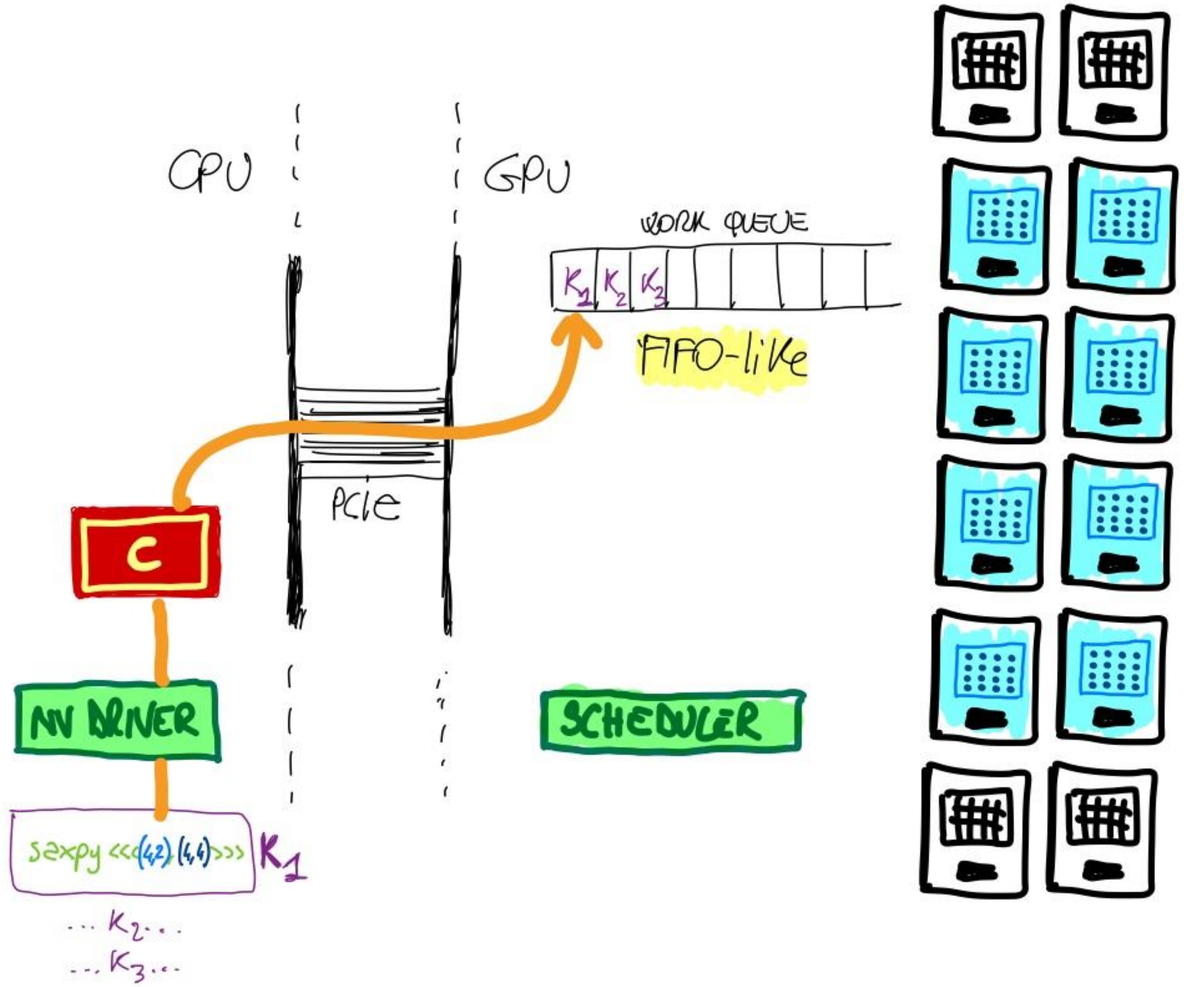
GPU kernels are “queued” for execution

1 GPU CONTEXT = 1 Work Queue



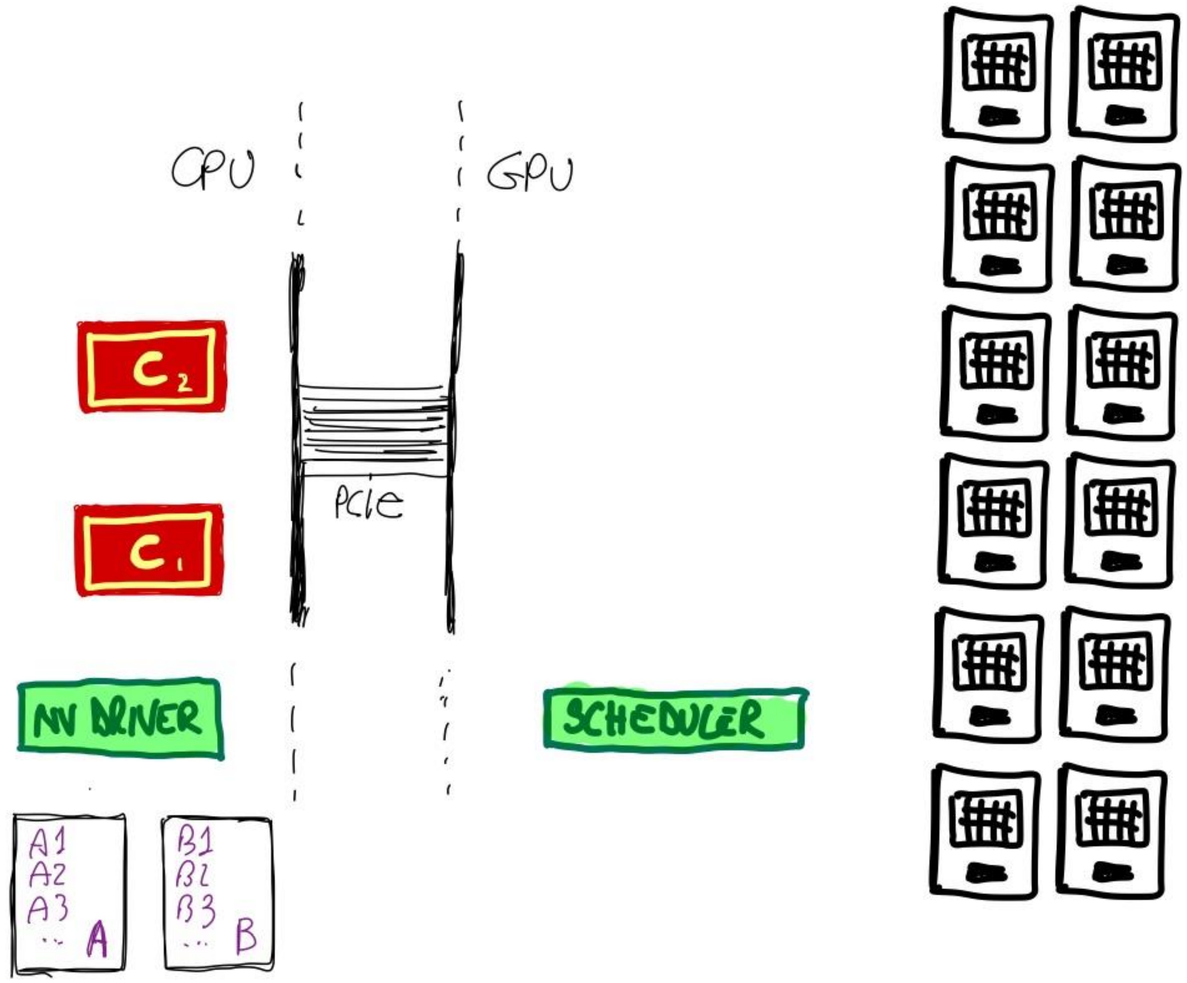
Kernels are executed one after the other

Scheduler dispatches a kernel (a group of blocks or threads) to each SM



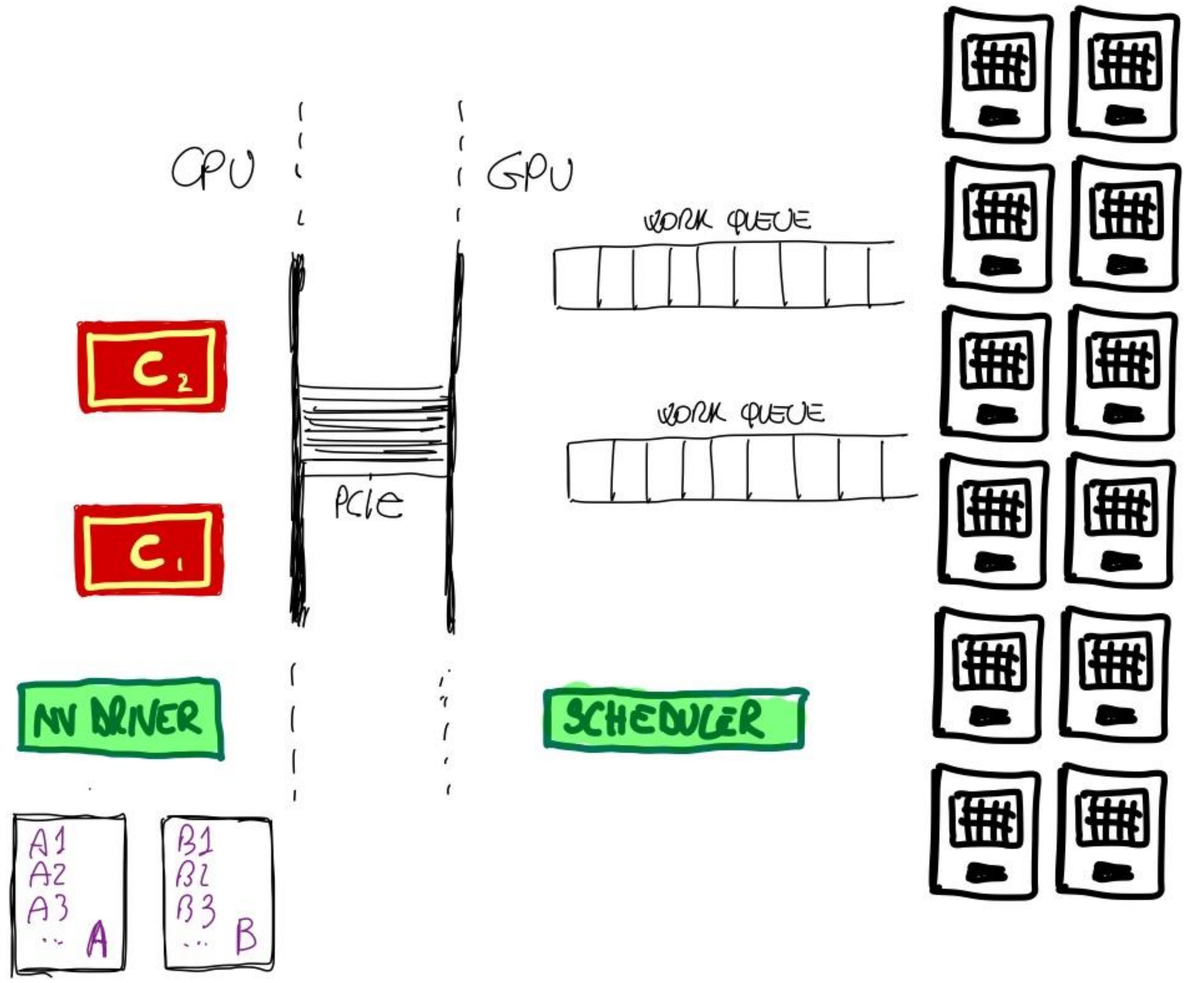
What happen if there are TWO processes?

When a process initialize a GPU, it create its own PRIVATE GPU context



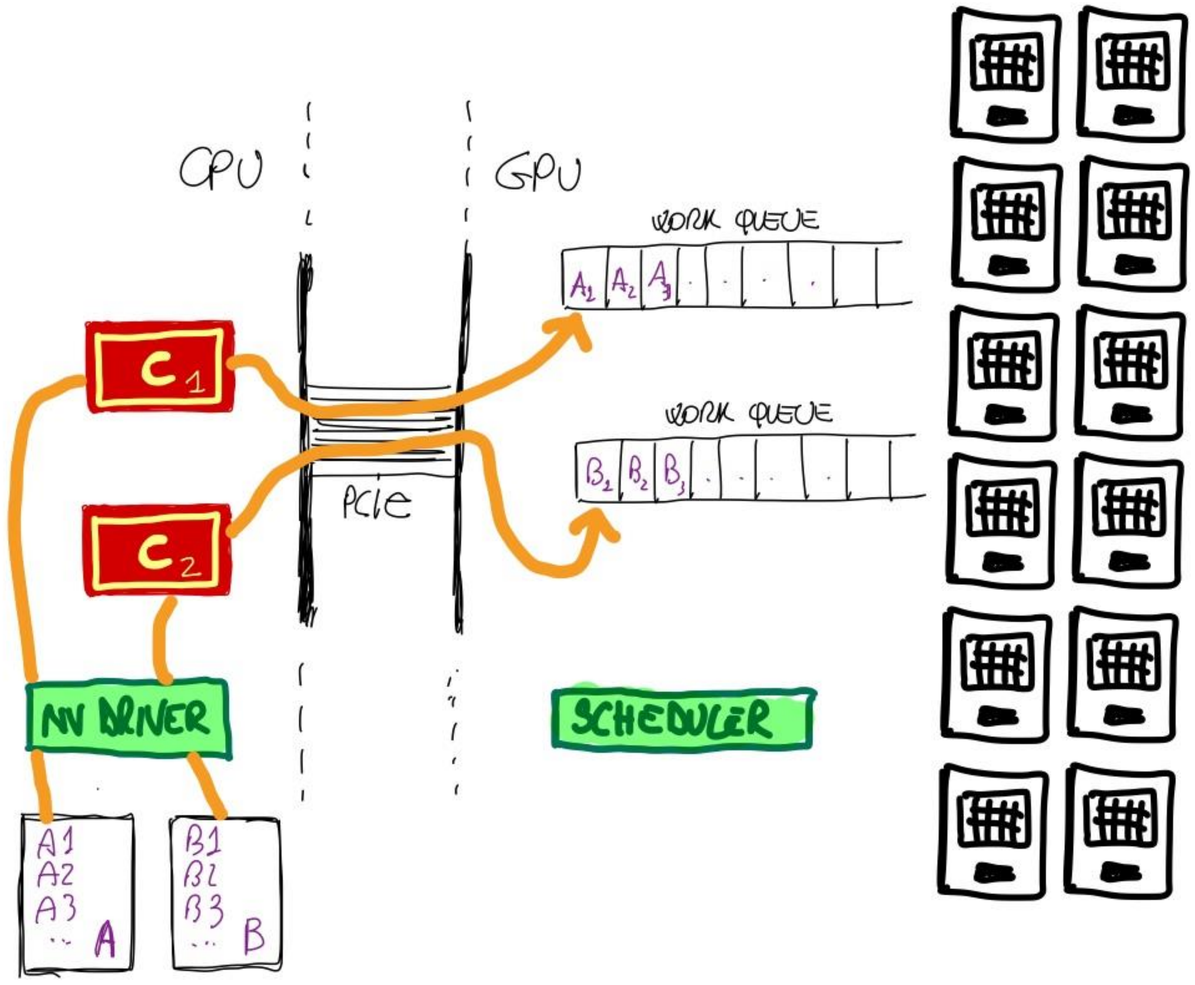
2 processes == 2 Work Queues

Kinda obvious, right?



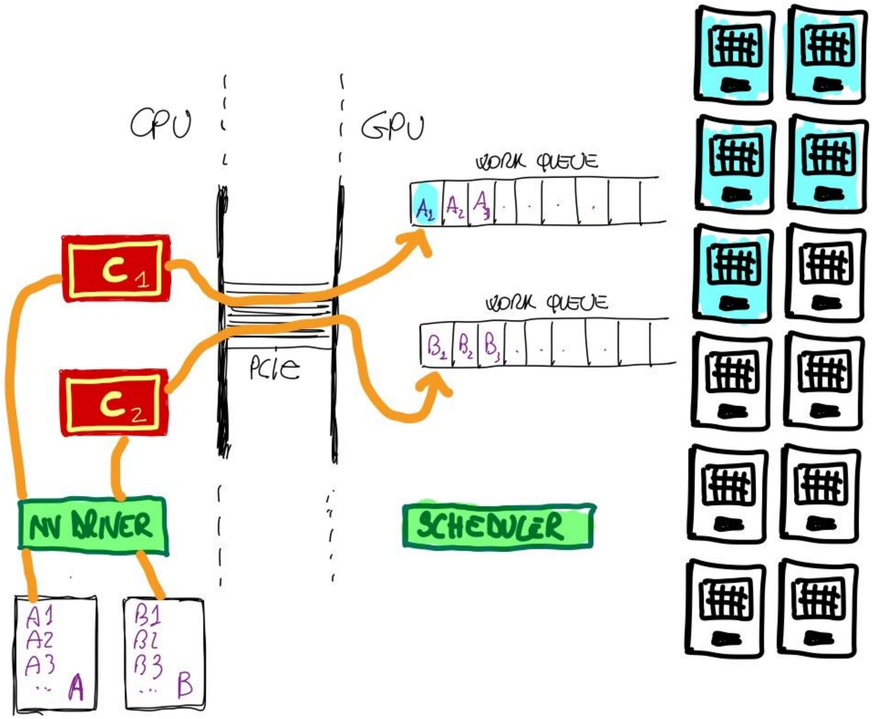
Kernels of different processes land on separated Work Queues

Who execute first?

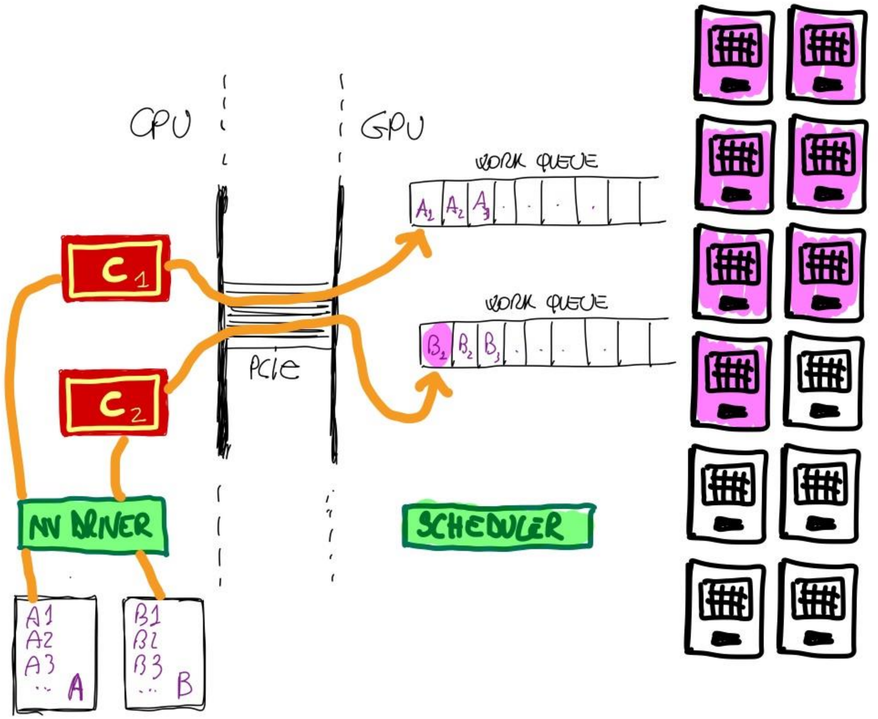


Kernels of different processes land on separated Work Queues

The scheduler decide! But they go one after the other (**time slicing**)



Time T

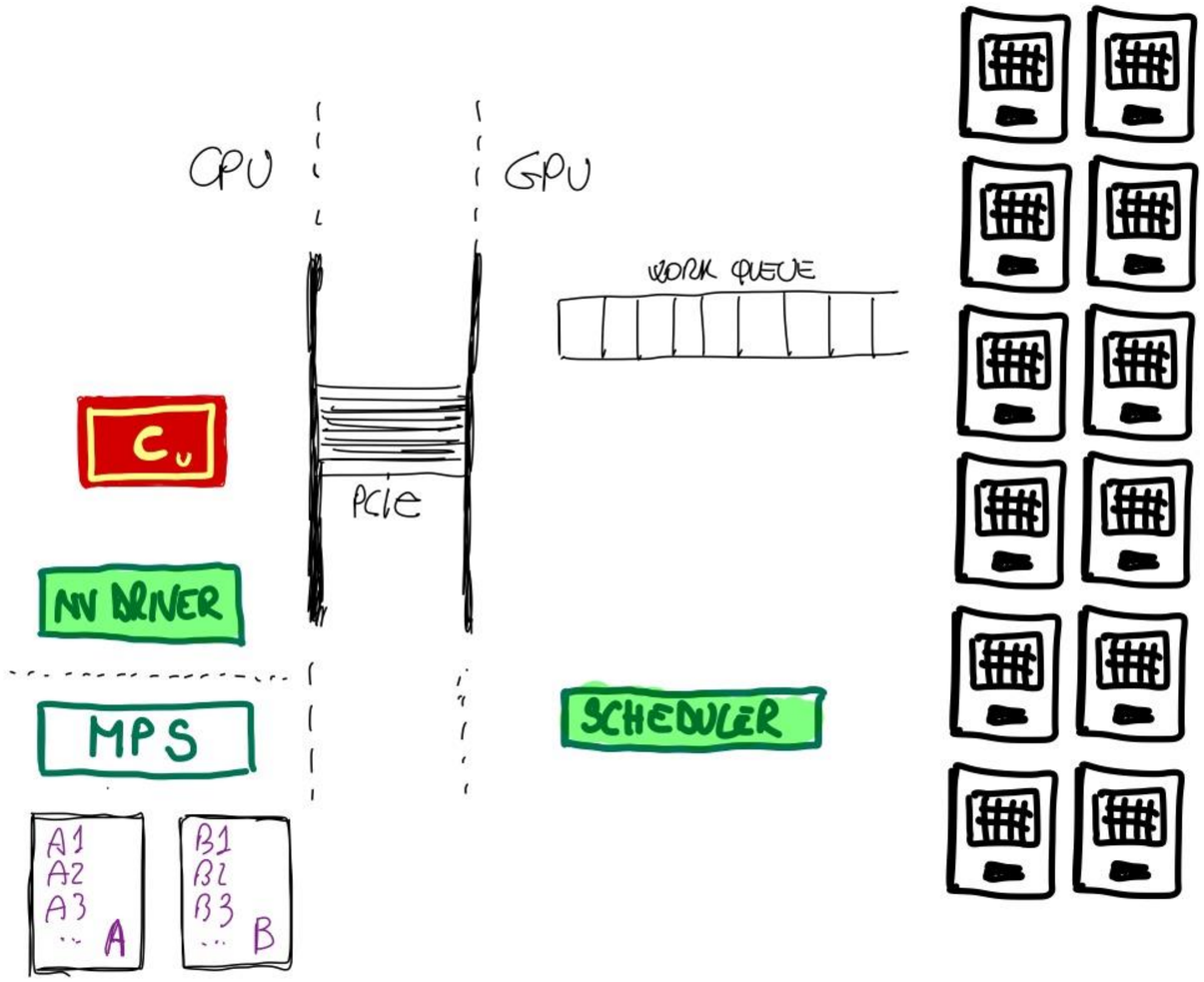


Time T+1

In both cases the GPU is under-utilized…. **BAD!**

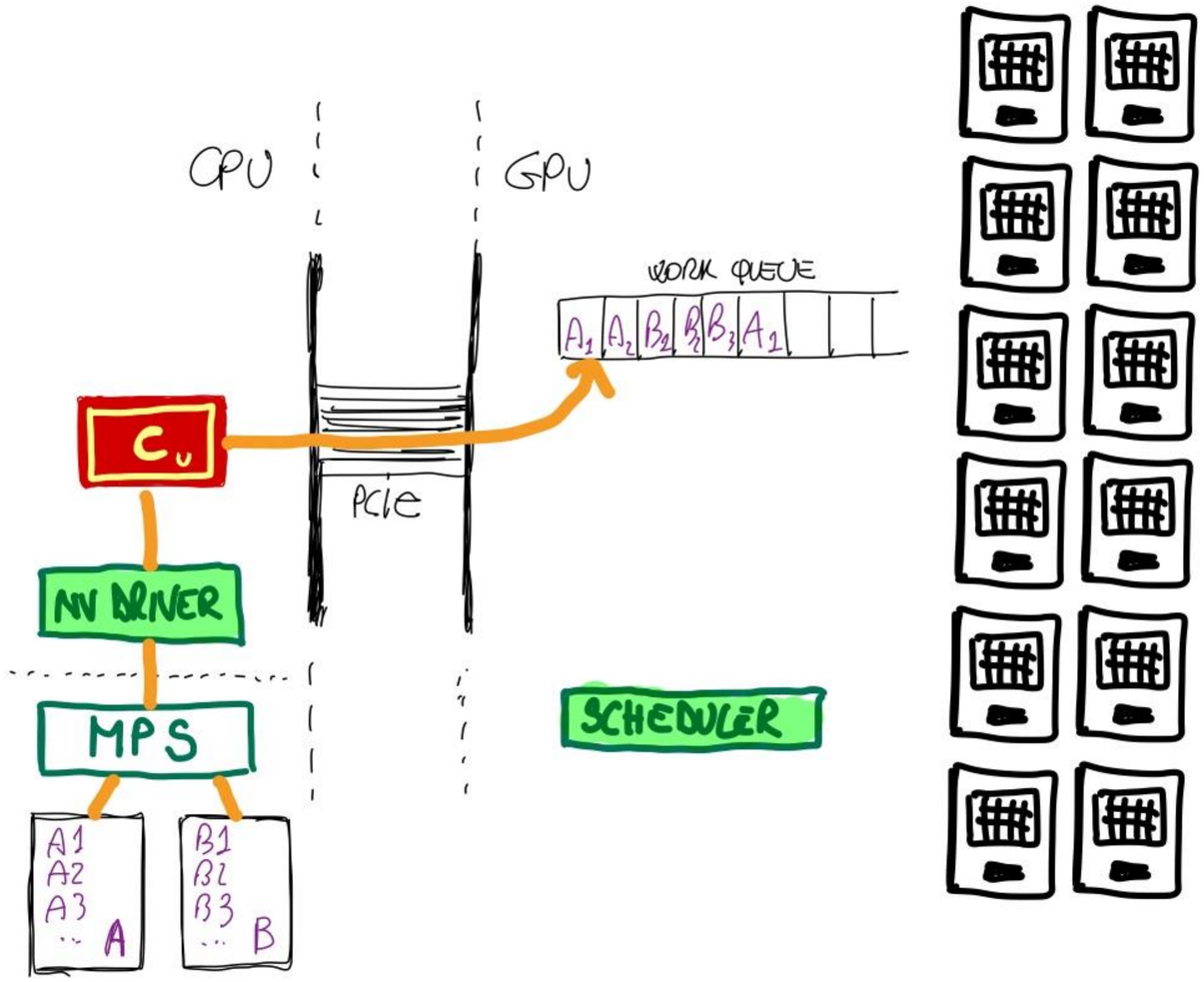
Let introduce NVIDIA Multi-Process Service

It’s “magic”: one GPU CONTEXT to rule all processes



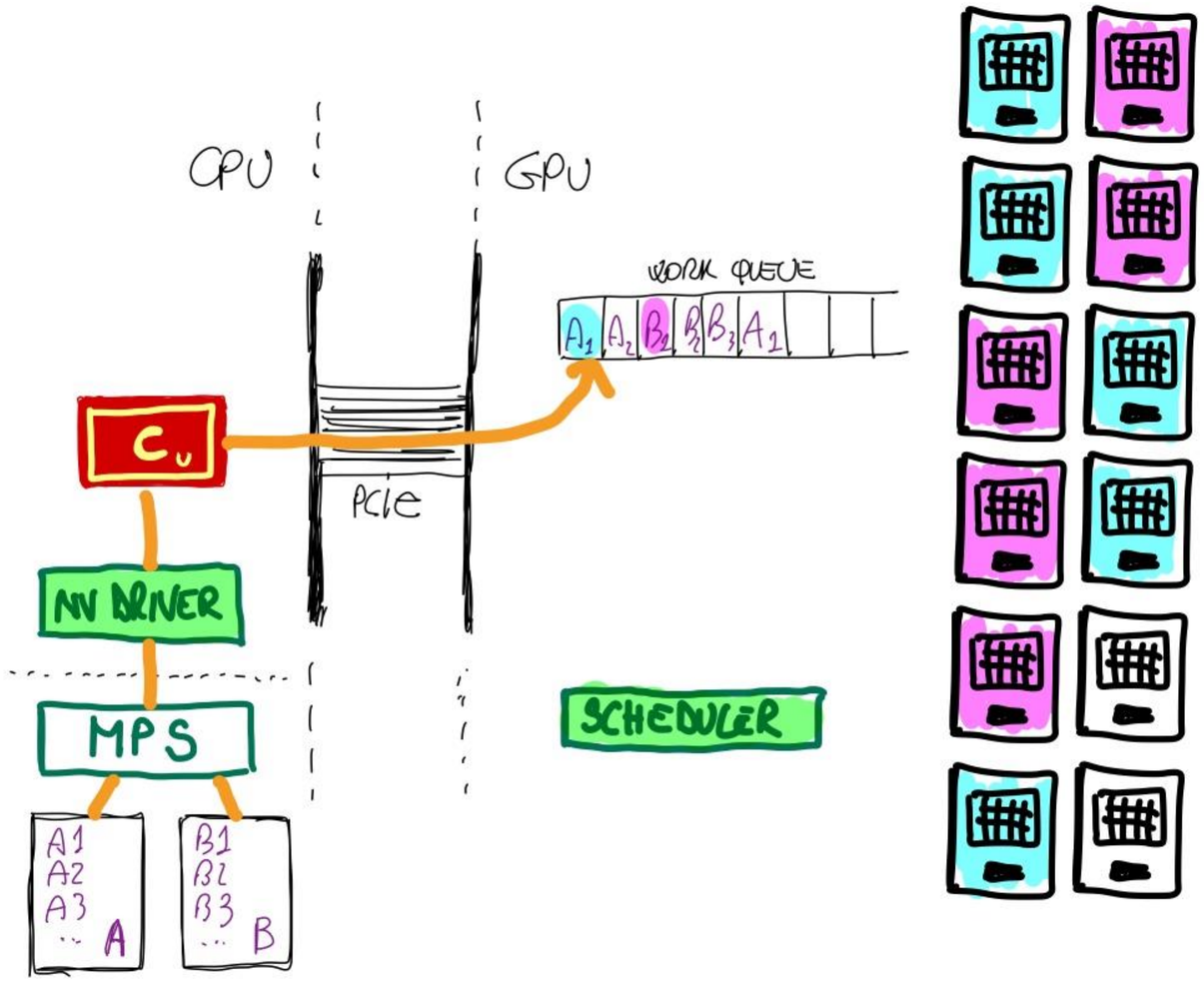
NVIDIA Multi-Process Service in action

All kernels of all processes are queued in the same Work Queue



NVIDIA Multi-Process Service in action

Kernels from **different processes** can execute **concurrently** (assuming no dependencies)



The Quest For Concurrency

“The road to concurrency is paved with threads, but beware—if you trip, you might find yourself in a race condition you never signed up for.”
(AI generated)

What if kernels could run concurrently?

Handling concurrency within the single executable

```
do i=1,N
    a(i) = i
end do

do i=1,N
     b(i) = 2.0 * i
end do

do i=1,N
     c(i) = a(i) + b(i)
end do
```

What if kernels could run concurrently?

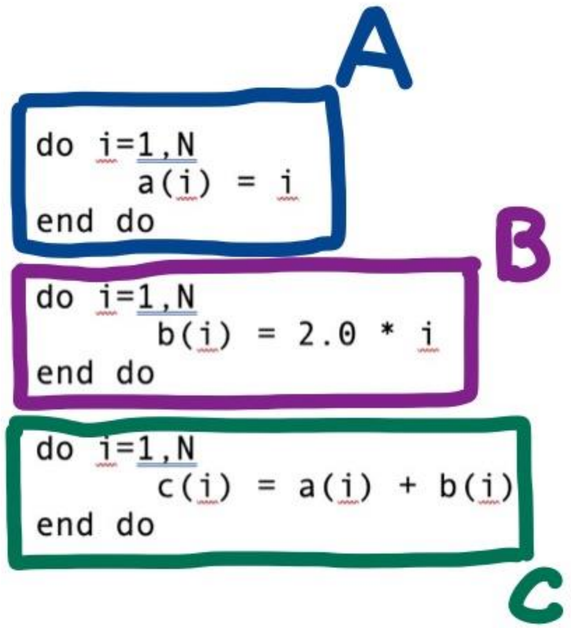
Handling concurrency within the single executable

```
do i=1,N
    a(i) = i
end do

do i=1,N
     b(i) = 2.0 * i
end do

do i=1,N
     c(i) = a(i) + b(i)
end do
```

```
!$acc parallel loop
do i=1,N
    a(i) = i
end do

!$acc parallel loop
do i=1,N
     b(i) = 2.0 * i
end do

!$acc parallel loop
do i=1,N
     c(i) = a(i) + b(i)
end do
```

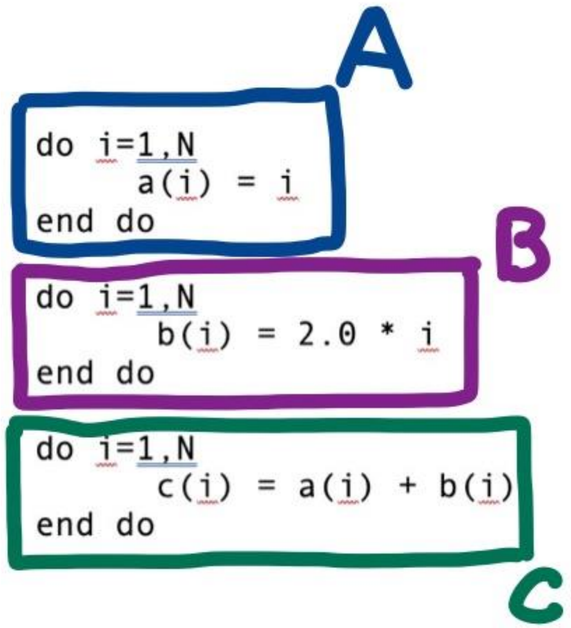
The Quest For Concurrency

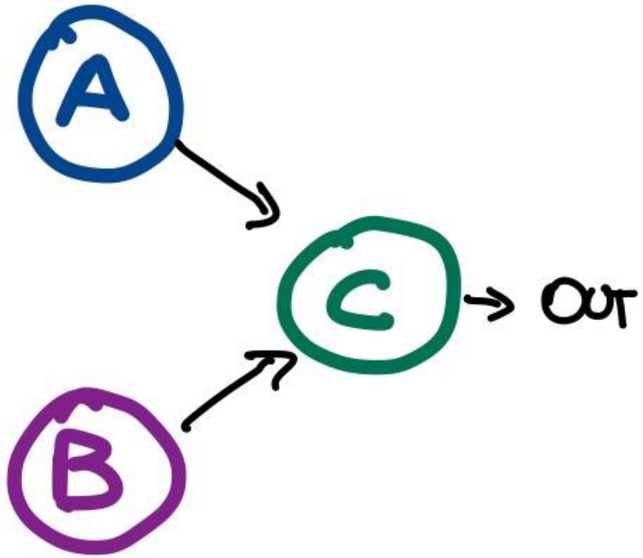
No matter how much the AI will outsmart you, handling concurrency is a hard problem



The Quest For Concurrency

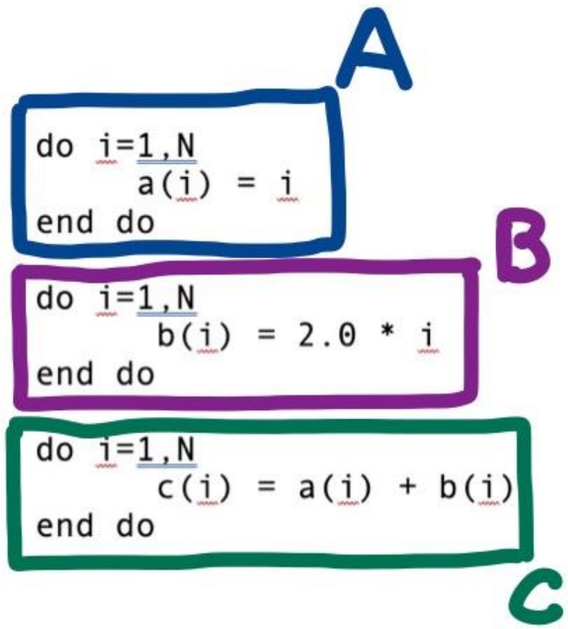
No matter how much the AI will outsmart you, handling concurrency is a hard problem

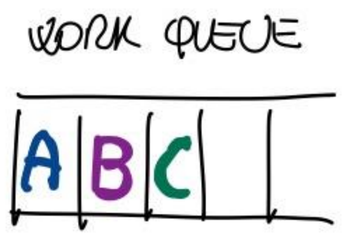


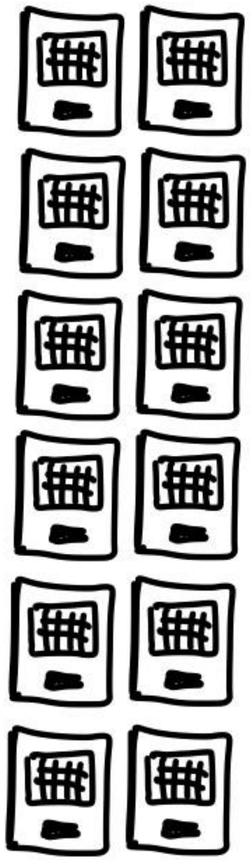


How to the scheduler avoid FIFO?

Spoiler: the schedule is smart but not a genius, it needs some help…

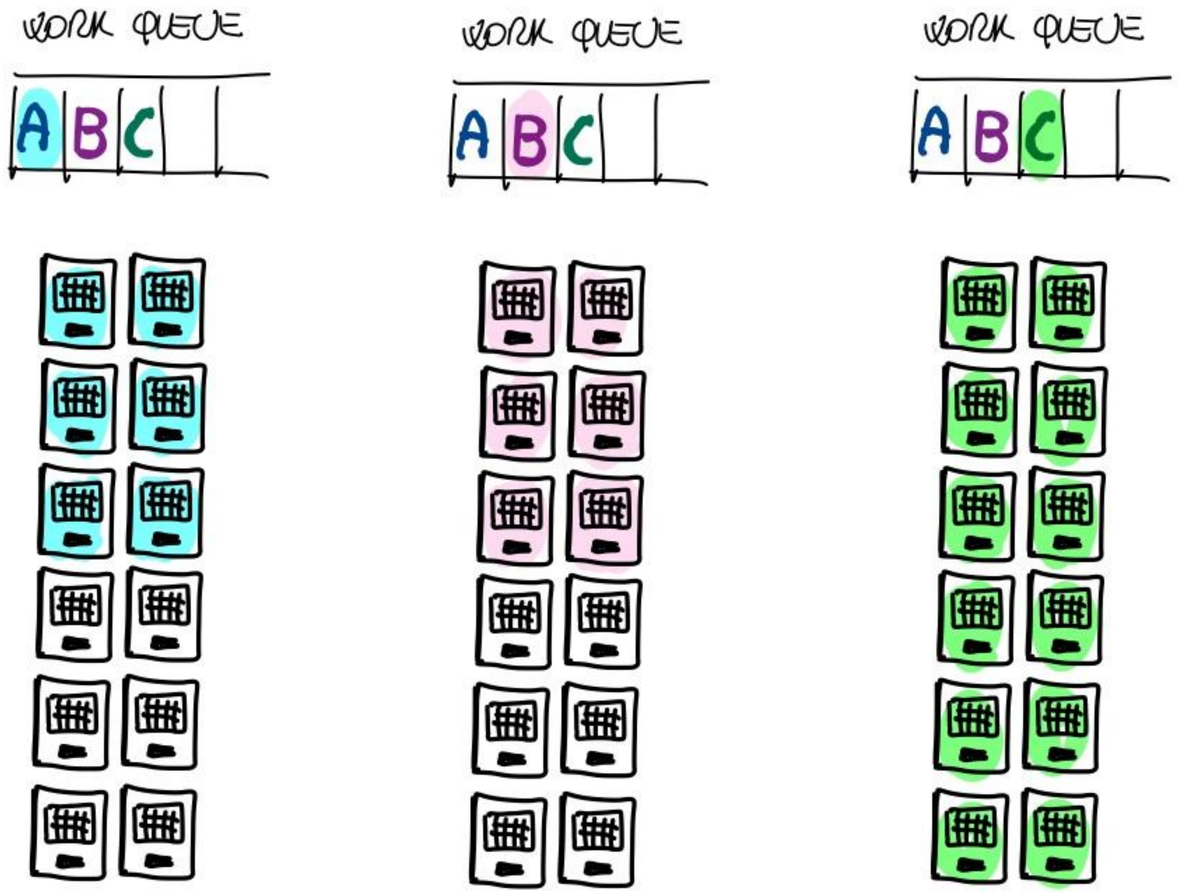






This is not what a “fast execution” should look like. How to fix?

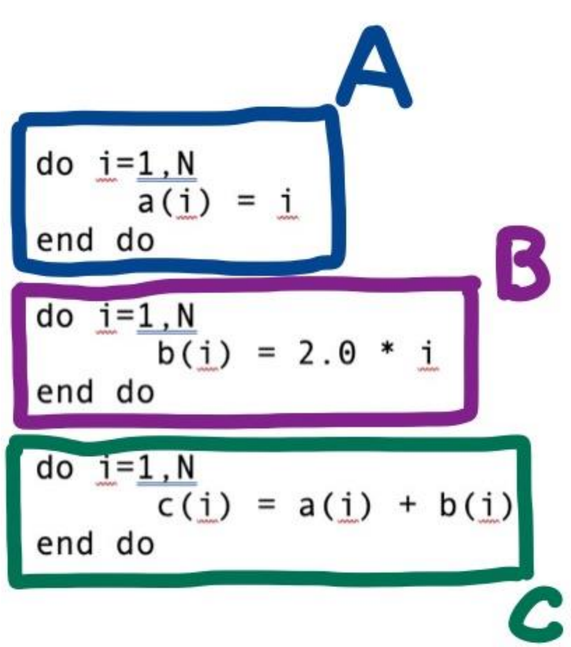
Independent kernels (A and B) run one after the other while GPU resources are poorly utilized

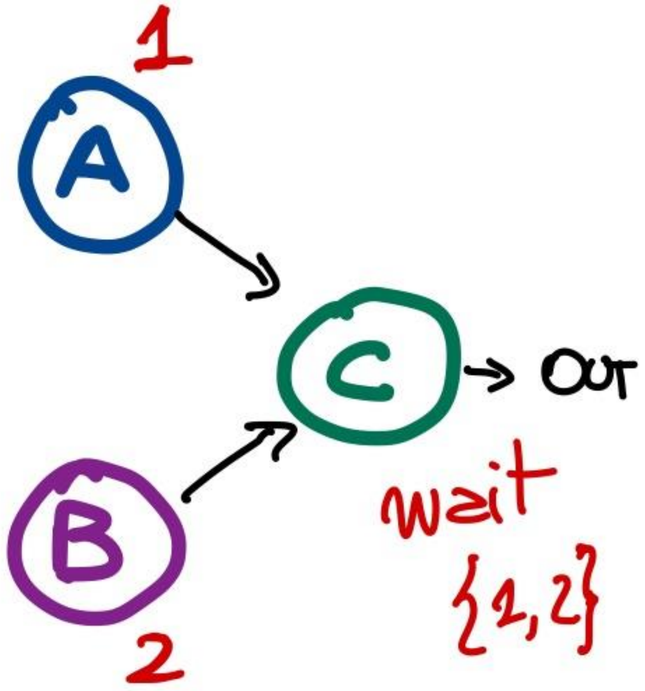


Time T | Time T+1 | Time T+2

How to avoid FIFO scheduler?

Spoiler: the schedule is smart but not a genius, it needs some help…

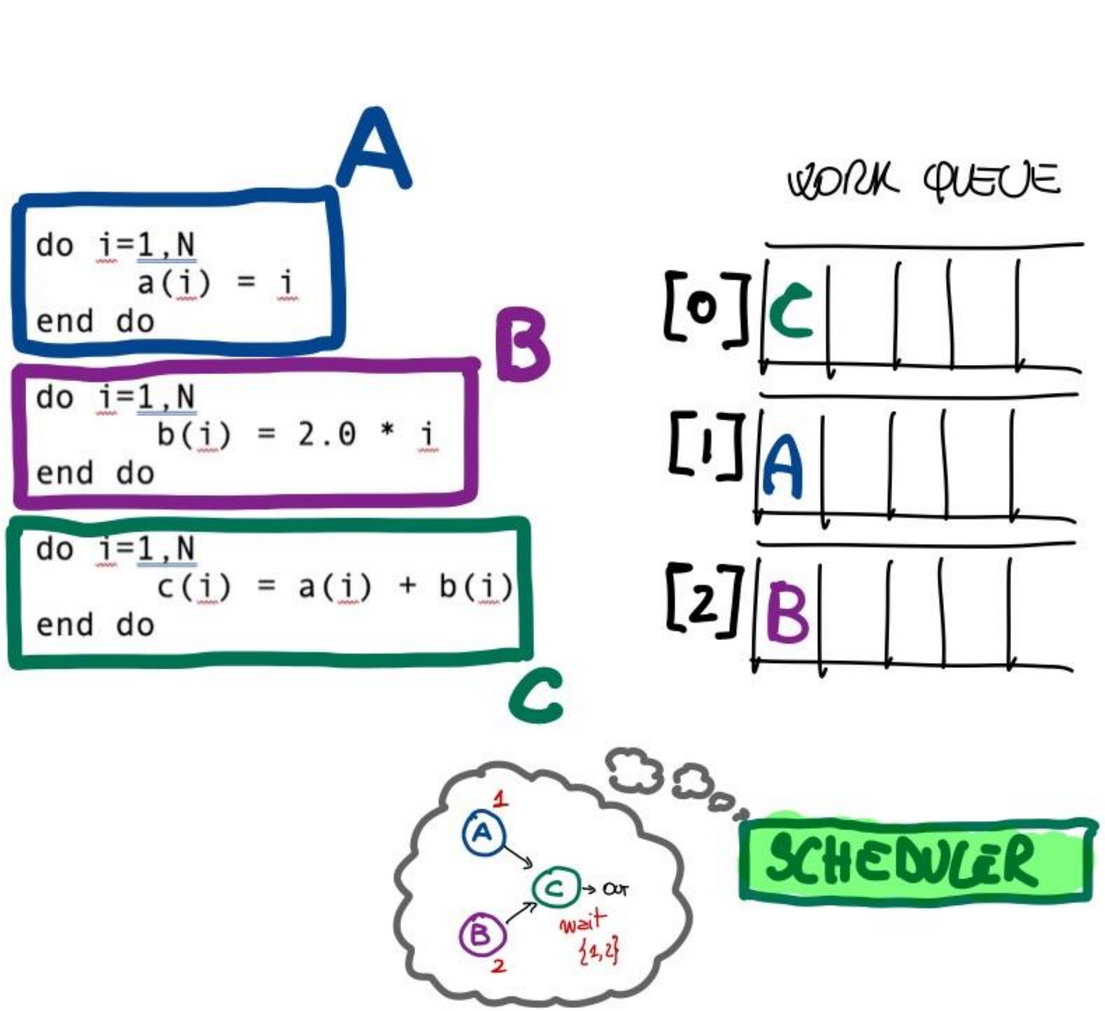




It is the **DEVELOPER'S RESPONSABILITY** to identify concurrency and ensure correctness

Introducing GPU STREAMS

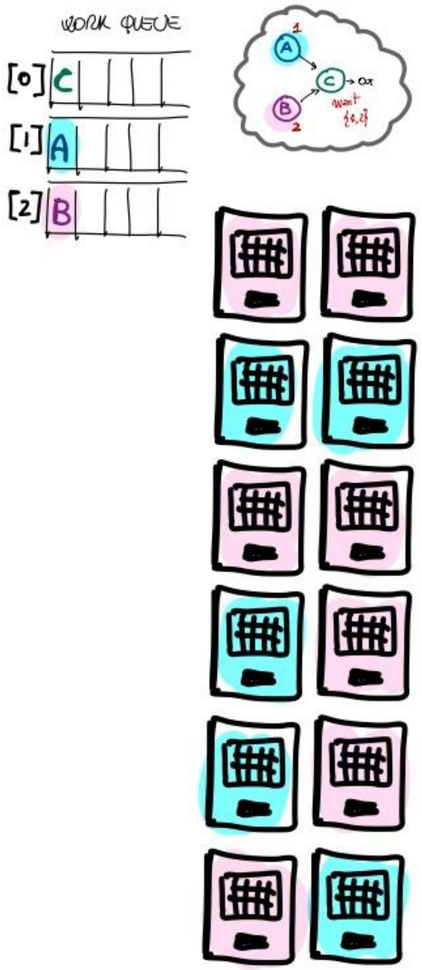
Each Work Queue can have multiple streams, within each stream kernels are executed in FIFO order



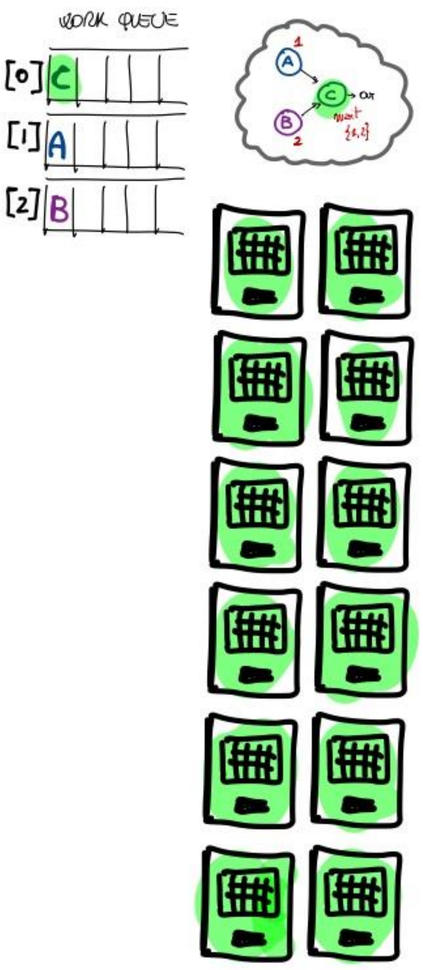
Concurrent executions of eligible kernels unlocked!

The scheduler can track some data dependencies

Also CPU<->GPU data movement can be overlapped and manages with streams



Time T

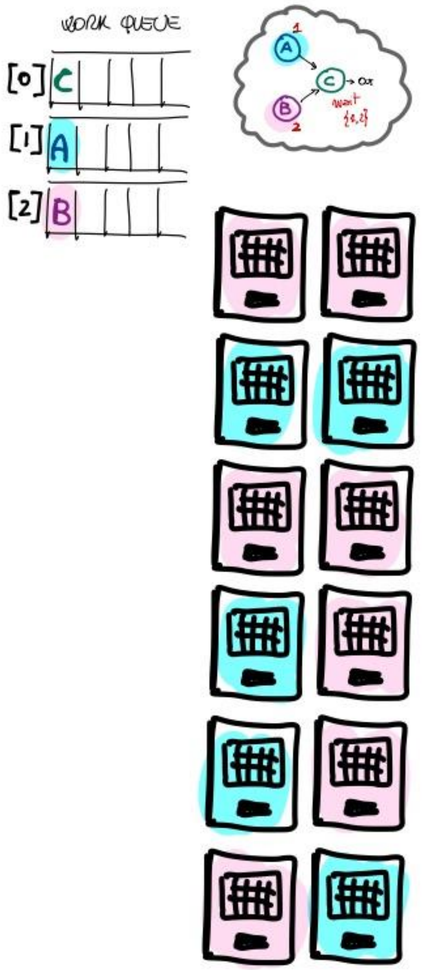


Time T+1

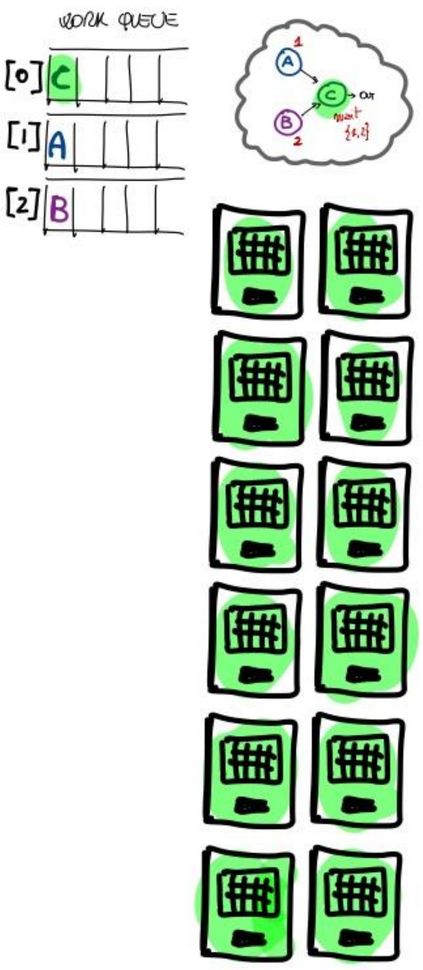
Managing Streams with OpenACC

What we want to achieve

Concurrency within a SINGLE PROCESS issuing multiple kernels



Time T



Time T+1

GPU STREAM recap

Semantic & Limitations to be aware of

- **Two operations issued into the same stream will execute in issue order**. Operation B issued after Operation A will not begin to execute until Operation A has completed.

- Two operations issued into separate streams have no ordering prescribed by the programming model.
  - Operation A issued into stream 1 may execute before, during, or after Operation B issued into stream 2.

- Stream 0 is the Default Stream
  - All device activity issued prior to the item in the default stream must complete before default stream item
  - begins
  - All device activity issued after the item in the default stream will wait for the default stream item to finish
  - All host threads share the same default stream for legacy behavior
  - Consider avoiding use of default stream during complex concurrency scenarios

- Kernels and (async) data movement operations can leverage streams
  - Pinning memory for async D2H and H2D is a must

The “async” and ”wait” clauses

Logically map to STREAM concept

- It enables asynchronous execution, allowing computation on the accelerator to proceed concurrently with the host CPU, improving performance by overlapping operations.
- Useful when dealing with independent code sections or data transfers that can run concurrently with other computations.

How it works:

- The **async** and **wait** clause can be added to parallel, kernels, and data directives.
- When an async clause is present, the associated operation (compute or data transfer) is launched on the accelerator, and the host thread continues execution without waiting for the operation to complete.
- This allows the accelerator and the host to work in parallel, potentially speeding up the overall execution time.
- To synchronize these asynchronous operations, the async clause can be used with the **wait** directive.

From Theory to Practice

```
do i=1,N
    a(i) = i
end do

do i=1,N
     b(i) = 2.0 * i
end do

do i=1,N
     c(i) = a(i) + b(i)
end do
```

```
!$acc parallel loop
do i=1,N
    a(i) = i
end do

!$acc parallel loop
do i=1,N
     b(i) = 2.0 * i
end do

!$acc parallel loop
do i=1,N
     c(i) = a(i) + b(i)
end do
```

ASSUMPTION: Data has been moved in the right place already (not straightforward task)

From Theory to Practice

Which version is the correct one?

```
!$acc parallel loop async(1)
do i=1,N
    a(i) = i
end do

!$acc parallel loop async(2)
do i=1,N
     b(i) = 2.0 * i
end do

!$acc parallel loop wait(1,2)
do i=1,N
     c(i) = a(i) + b(i)
end do
```

Option A

```
!$acc parallel loop async(1)
do i=1,N
    a(i) = i
end do

!$acc parallel loop async(2)
do i=1,N
     b(i) = 2.0 * i
end do
!$acc wait(1,2)
!$acc parallel loop
do i=1,N
     c(i) = a(i) + b(i)
end do
```

Option B

```
!$acc parallel loop async(1)
do i=1,N
    a(i) = i
end do

!$acc parallel loop async(2)
do i=1,N
     b(i) = 2.0 * i
end do
!$acc wait(1)
!$acc parallel loop async(2)
do i=1,N
     c(i) = a(i) + b(i)
end do
!$acc wait(2)
```

Option C

Which one is correct?

ASSUMPTION: Data has been moved in the right place already (not straightforward task)

From Theory to Practice

Which version is the correct one?

```
!$acc parallel loop async(1)
do i=1,N
    a(i) = i
end do

!$acc parallel loop async(2)
do i=1,N
     b(i) = 2.0 * i
end do

!$acc parallel loop wait(1,2)
do i=1,N
     c(i) = a(i) + b(i)
end do
```

Option A

```
!$acc parallel loop async(1)
do i=1,N
    a(i) = i
end do

!$acc parallel loop async(2)
do i=1,N
     b(i) = 2.0 * i
end do
!$acc wait(1,2)
!$acc parallel loop
do i=1,N
     c(i) = a(i) + b(i)
end do
```

Option B

```
!$acc parallel loop async(1)
do i=1,N
    a(i) = i
end do

!$acc parallel loop async(2)
do i=1,N
     b(i) = 2.0 * i
end do
!$acc wait(1)
!$acc parallel loop async(2)
do i=1,N
     c(i) = a(i) + b(i)
end do
!$acc wait(2)
```

Option C

All three!

ASSUMPTION: Data has been moved in the right place already (not straightforward task)

Not only concurrent compute

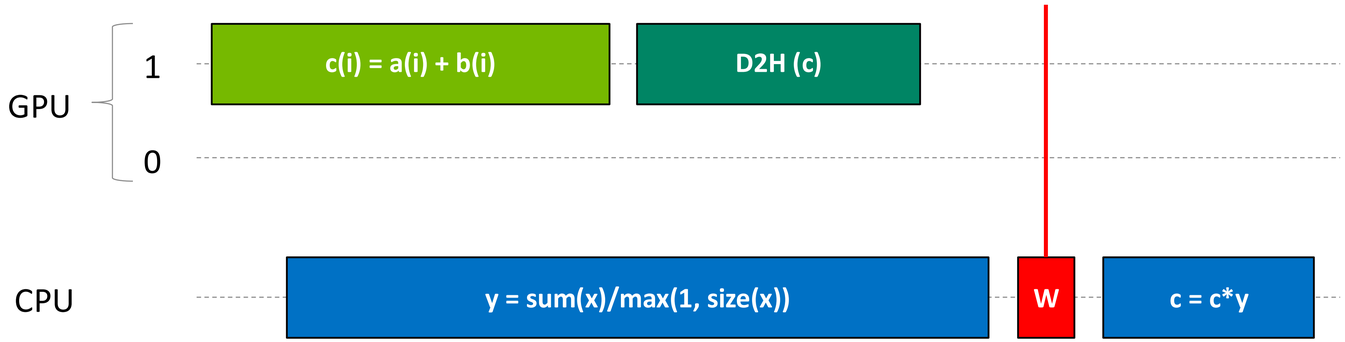
Data movement can be associated to streams

```
!$acc parallel loop async(1)
do i=1,N
    c(i) = a(i) + b(i)
end do

!$acc update self(c) async (1)

y = sum(x)/max(1, size(x))

!$acc wait
c = c*y
```

Not only concurrent compute

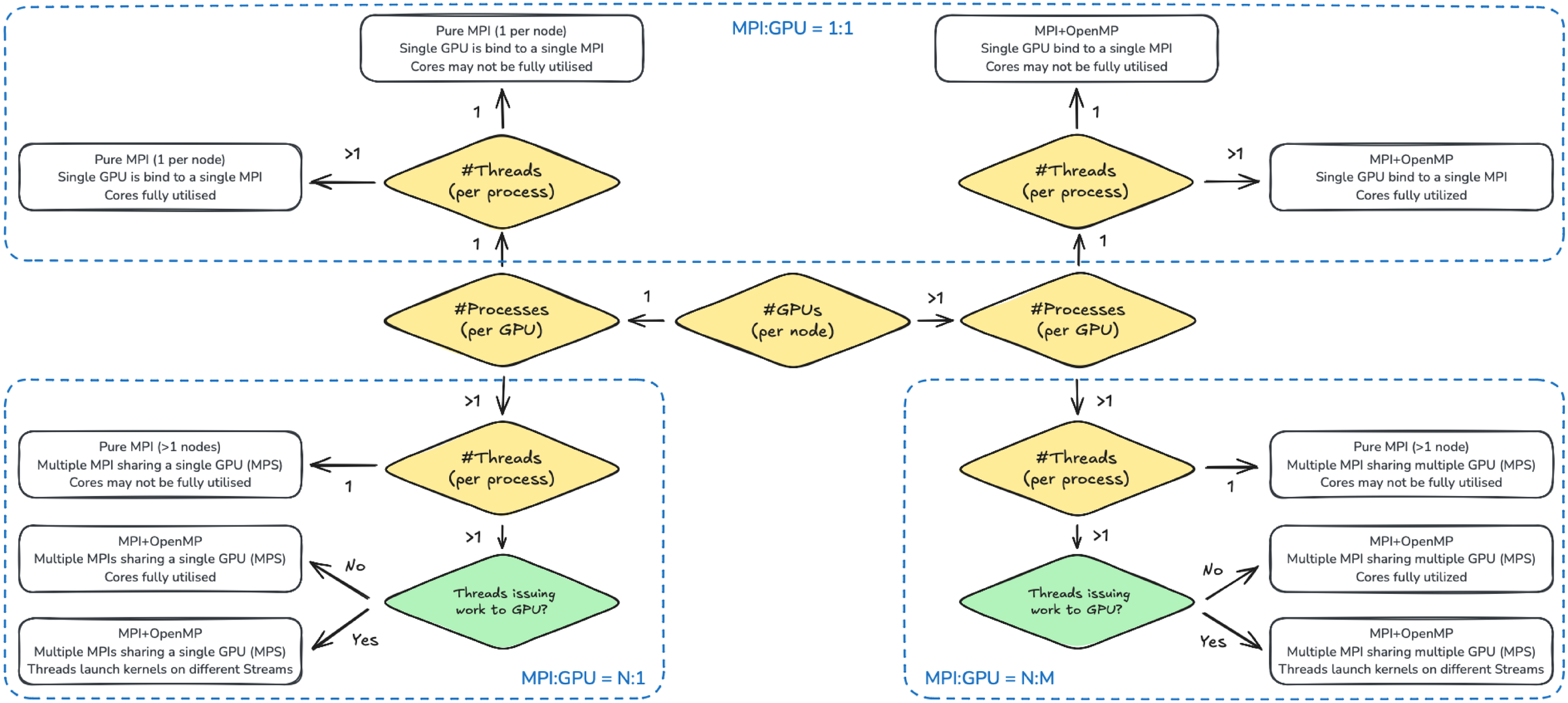
Data movement can be associated to streams

```
!$acc parallel loop async(1)
do i=1,N
    c(i) = a(i) + b(i)
end do

!$acc update self(c) async (1)

y = sum(x)/max(1, size(x))

!$acc wait
c = c*y
```

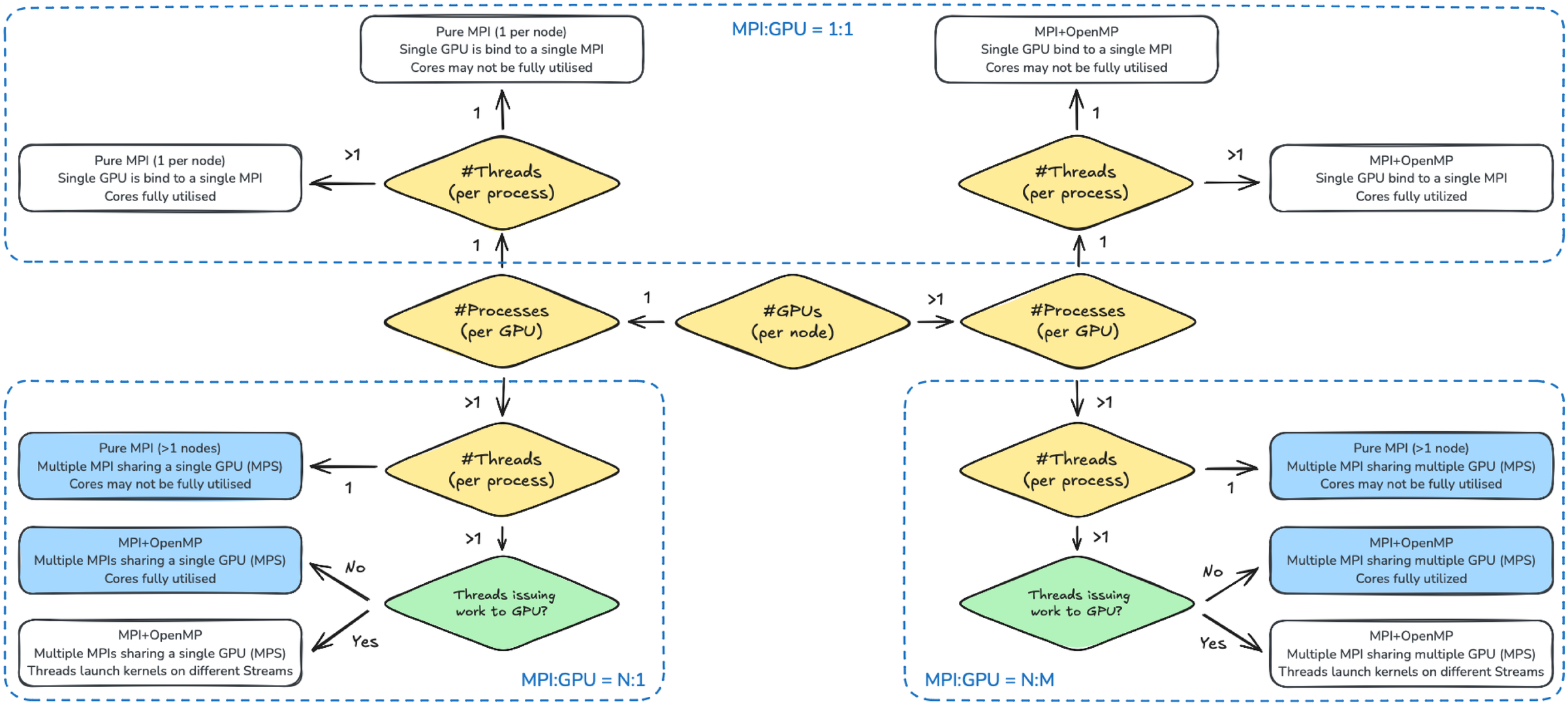


Before dive into single-process multi-GPU …

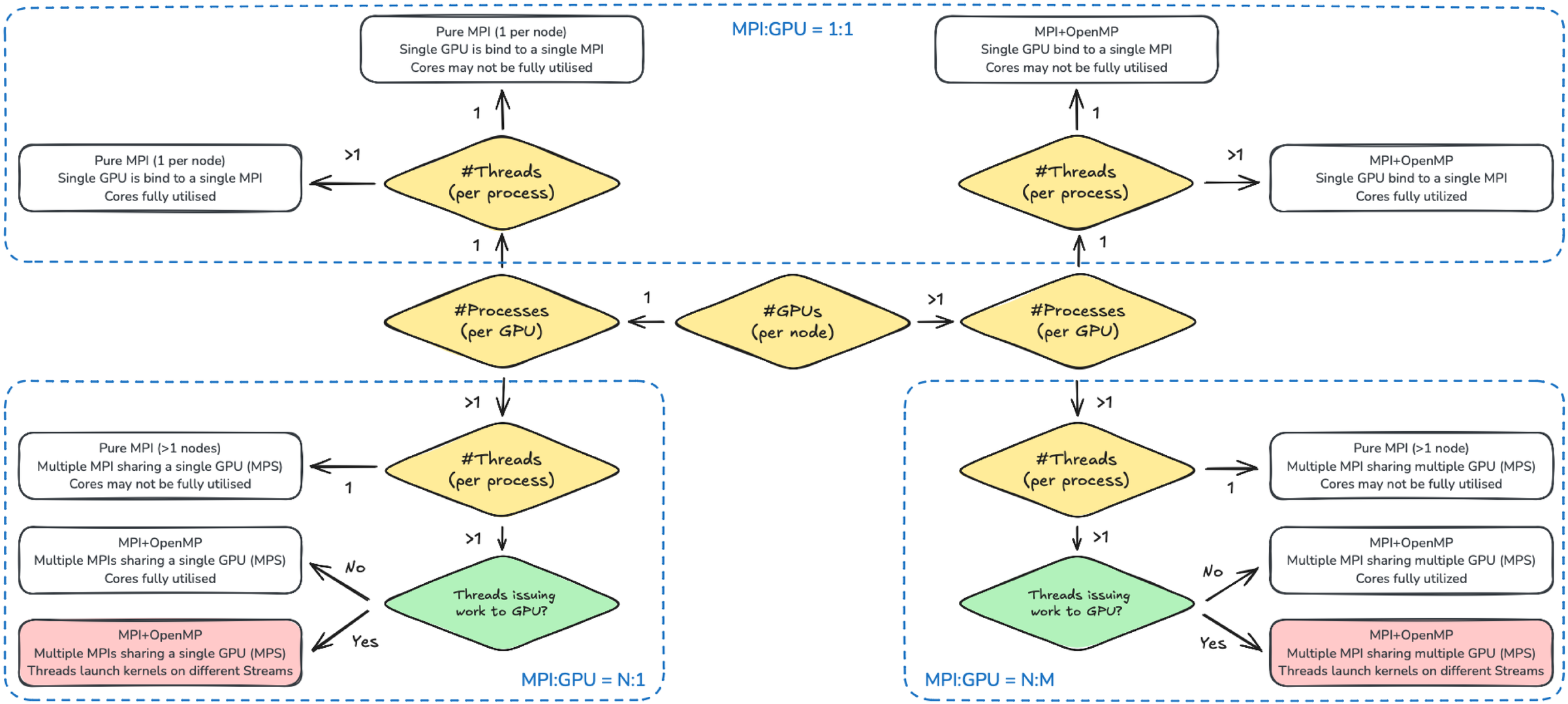
What if I have an MPI application?



What if I have an MPI application?



What if I have an MPI application?



Managing multiple GPUs from a single process

How do we select a GPU in OpenACC?

Spoiler: there are no directive, only runtime API

- OpenACC presents devices numbered 0 – (N-1) for each device type available.
- The order of the devices comes from the runtime, almost certainly the same as CUDA
- By default all data and work go to the current device
- Developers must change the current device and maybe the current device type using an API (requires explicit header of module)
- Special keyword “acc_device_nvidia” identify NVIDIA GPU type of accelerator

ROUTINES:

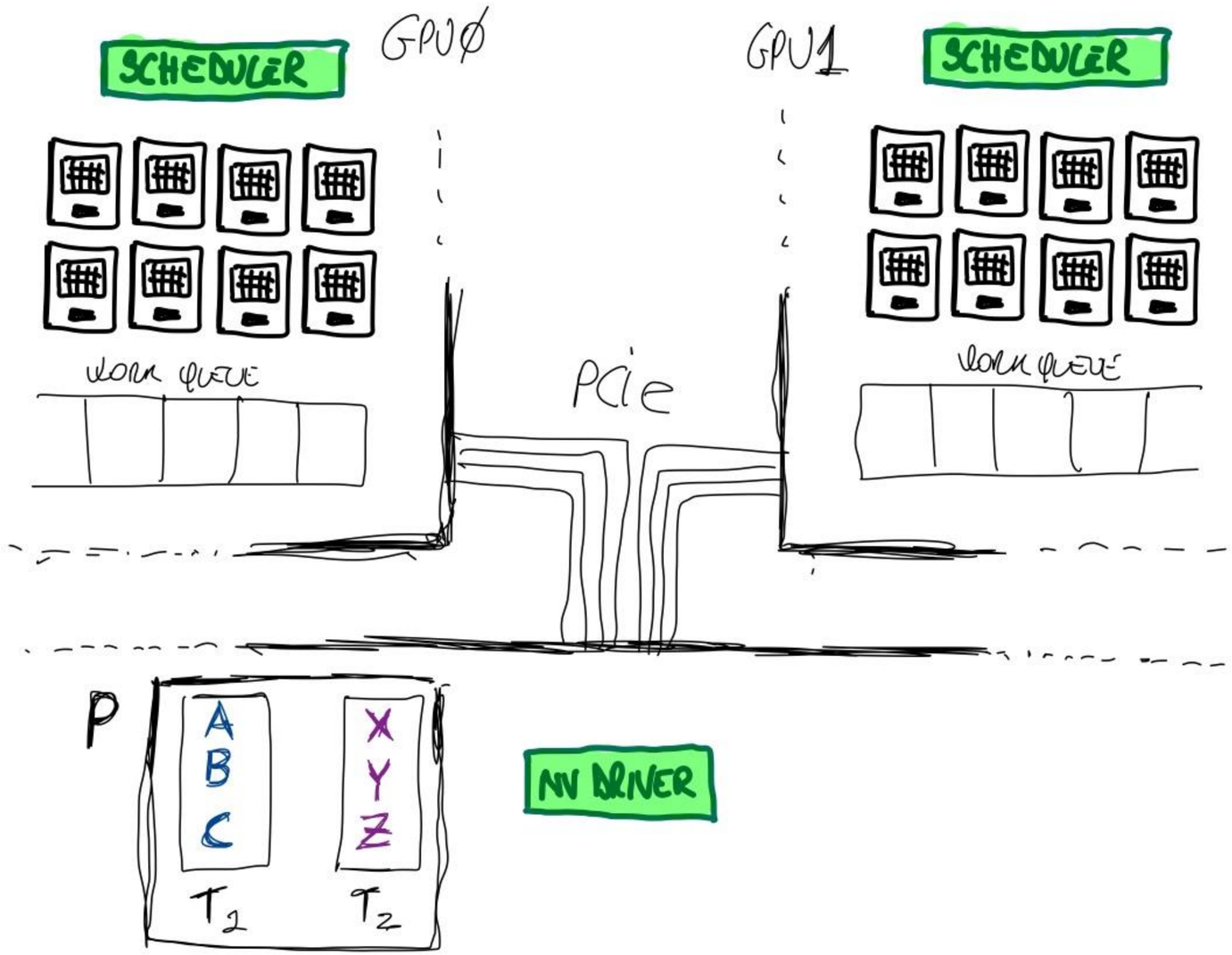
acc_get_num_devices(X)	Return number of devices of type
acc_set_device_num(i, X)	Set device number **i** of type **X** active

INCLUDES:

C/C++	#include <openacc.h>
Fortran	use openacc

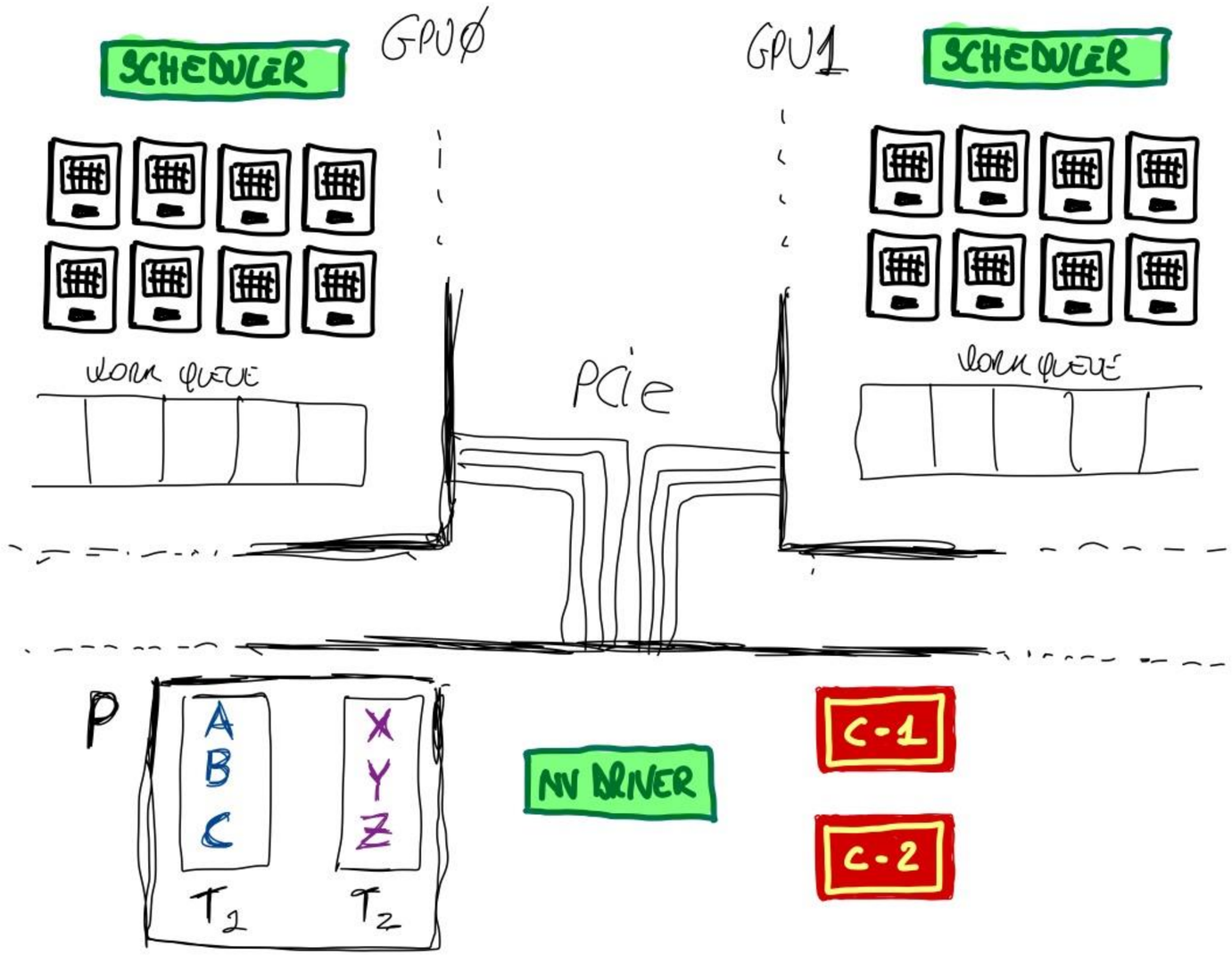
Multiple OpenMP threads issuing kernels to different GPUS

#OpenMP threads == #GPU → How many GPU CONTEXT?



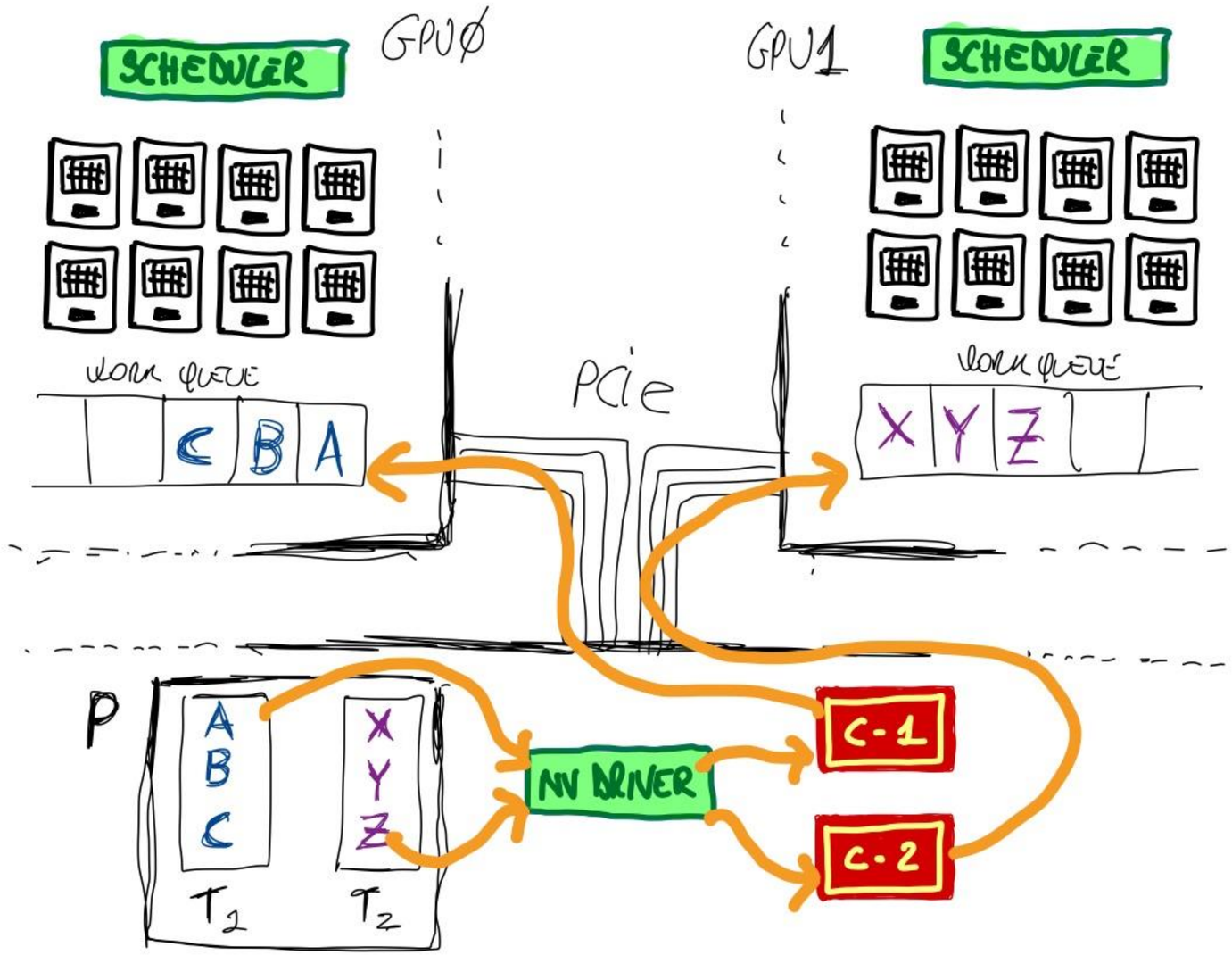
Multiple OpenMP threads issuing kernels to different GPUS

One GPU CONTEXT per GPU (and per process)



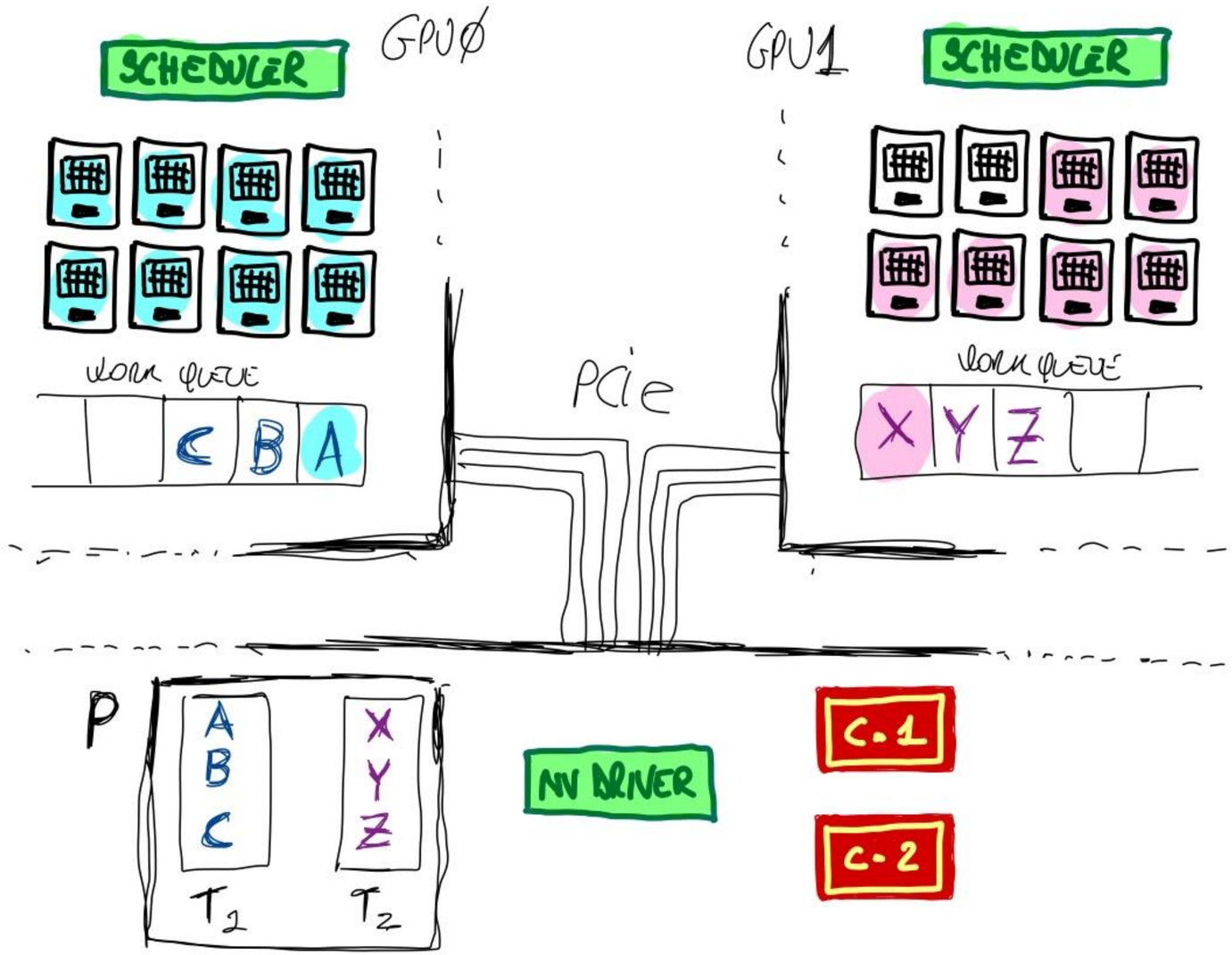
Multiple OpenMP threads issuing kernels to different GPUS

Each thread is assigned to a different GPU (different CONTEXT) and issue to a different Work Queue



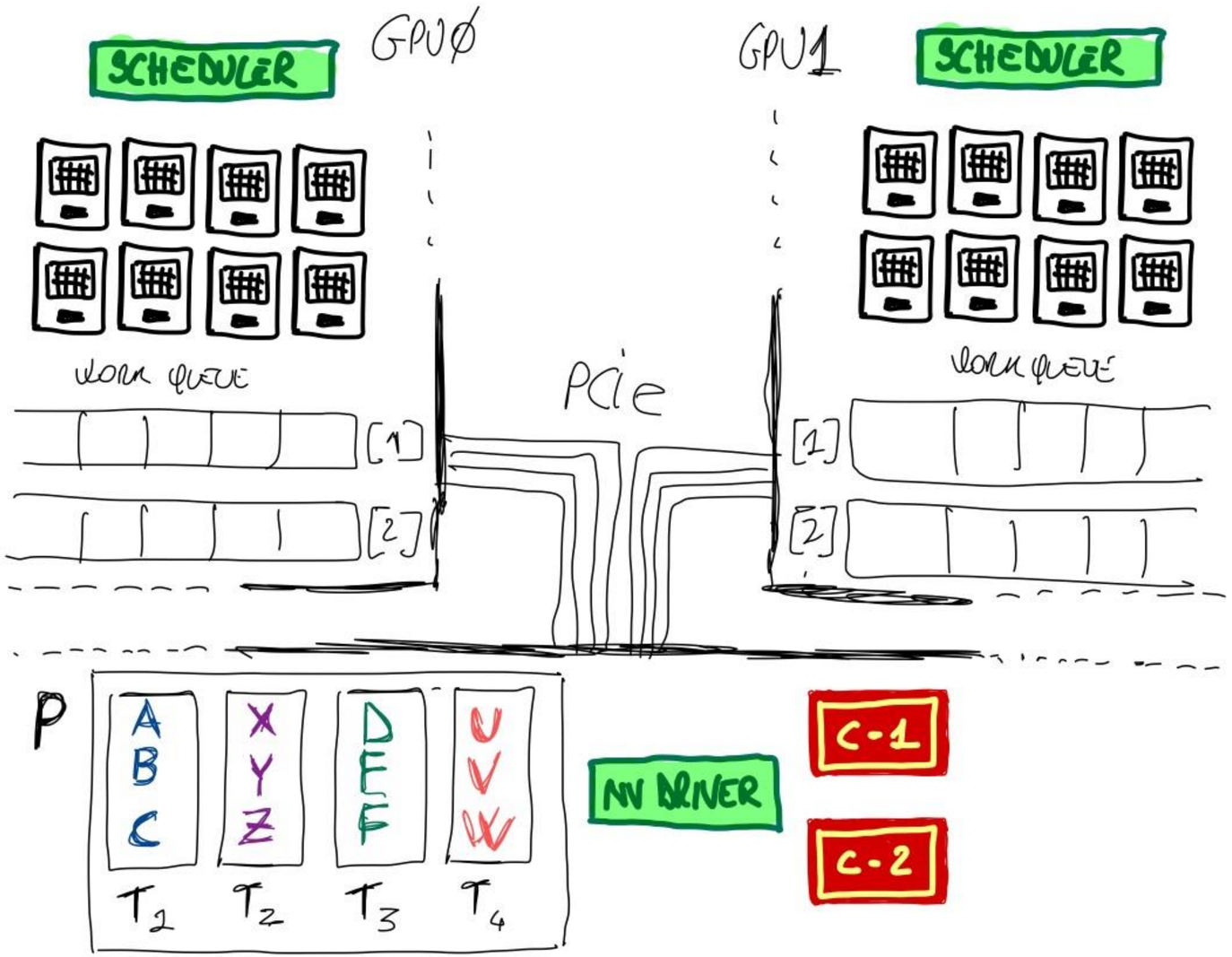
Multiple OpenMP threads issuing kernels to different GPUS

Execution on each GPU happens independently and asynchronously



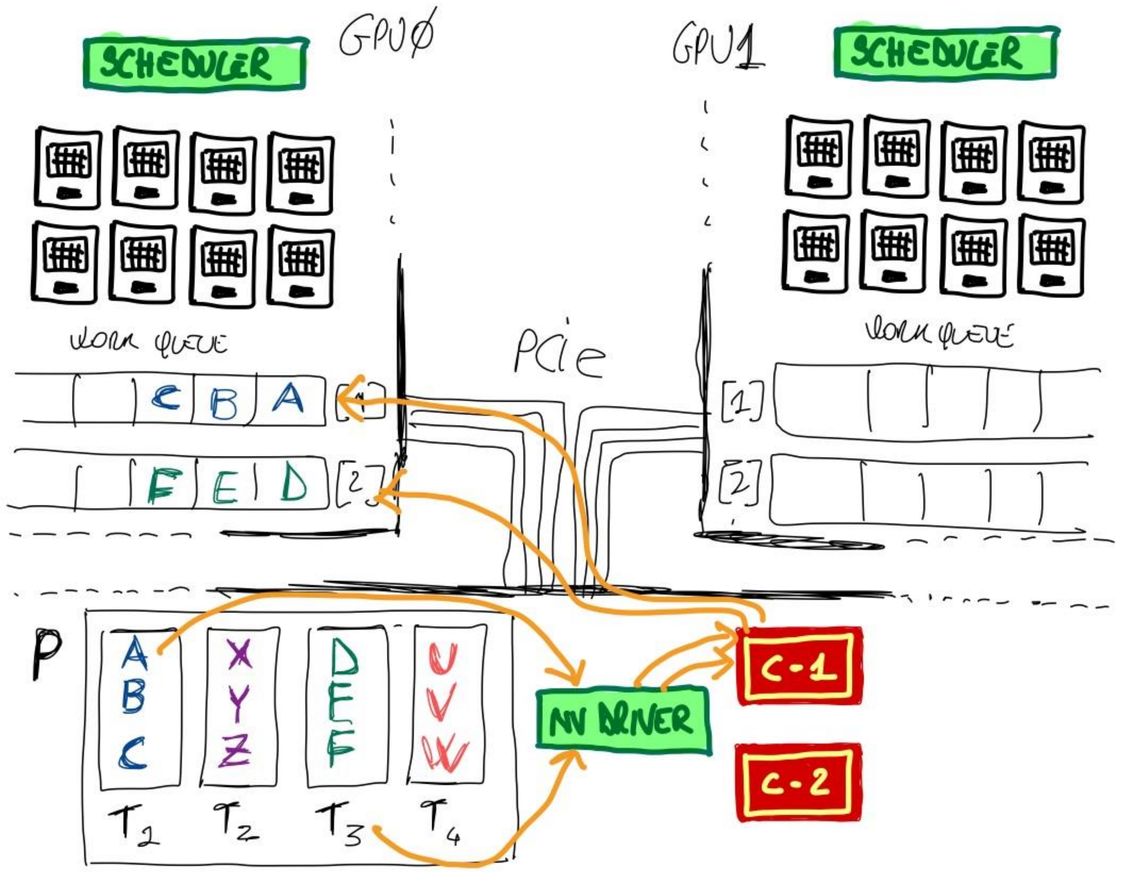
Multiple OpenMP threads issuing kernels to different GPUS

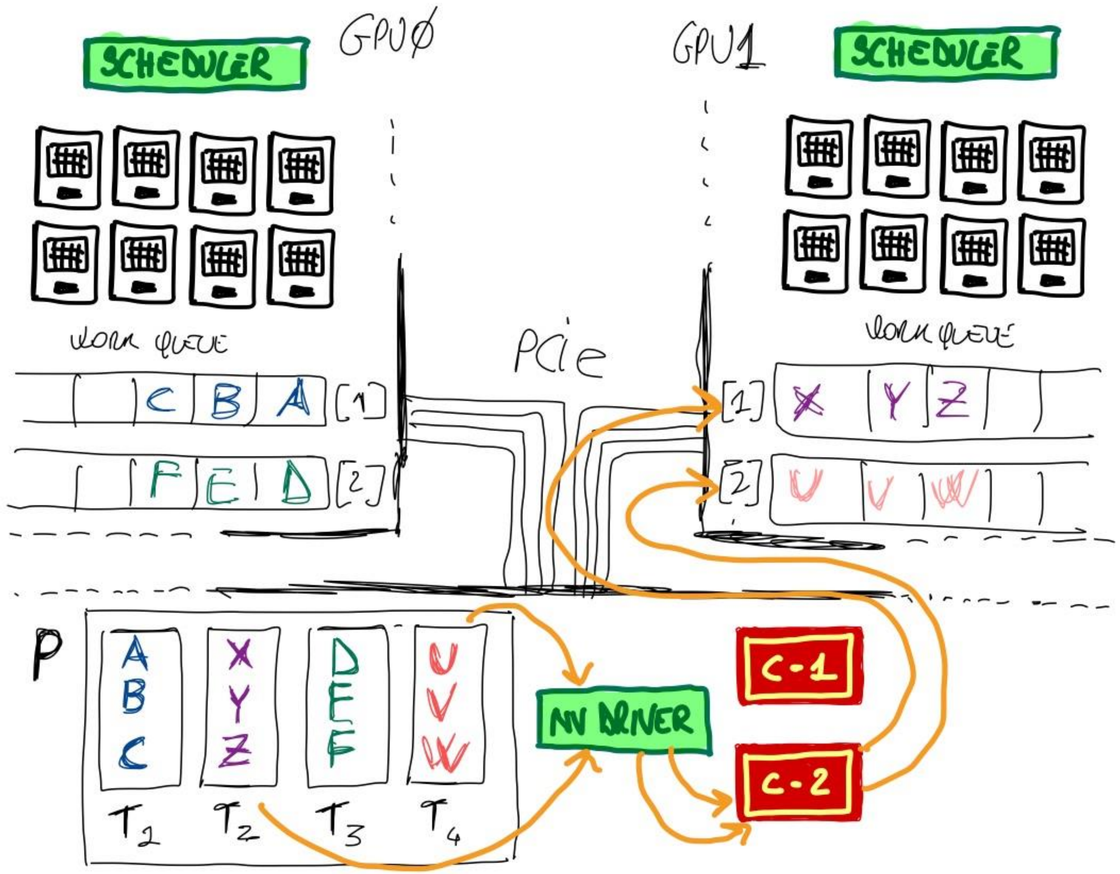
#OpenMP threads > #GPU → How many GPU CONTEXT?



Multiple OpenMP threads issuing kernels to different GPUS

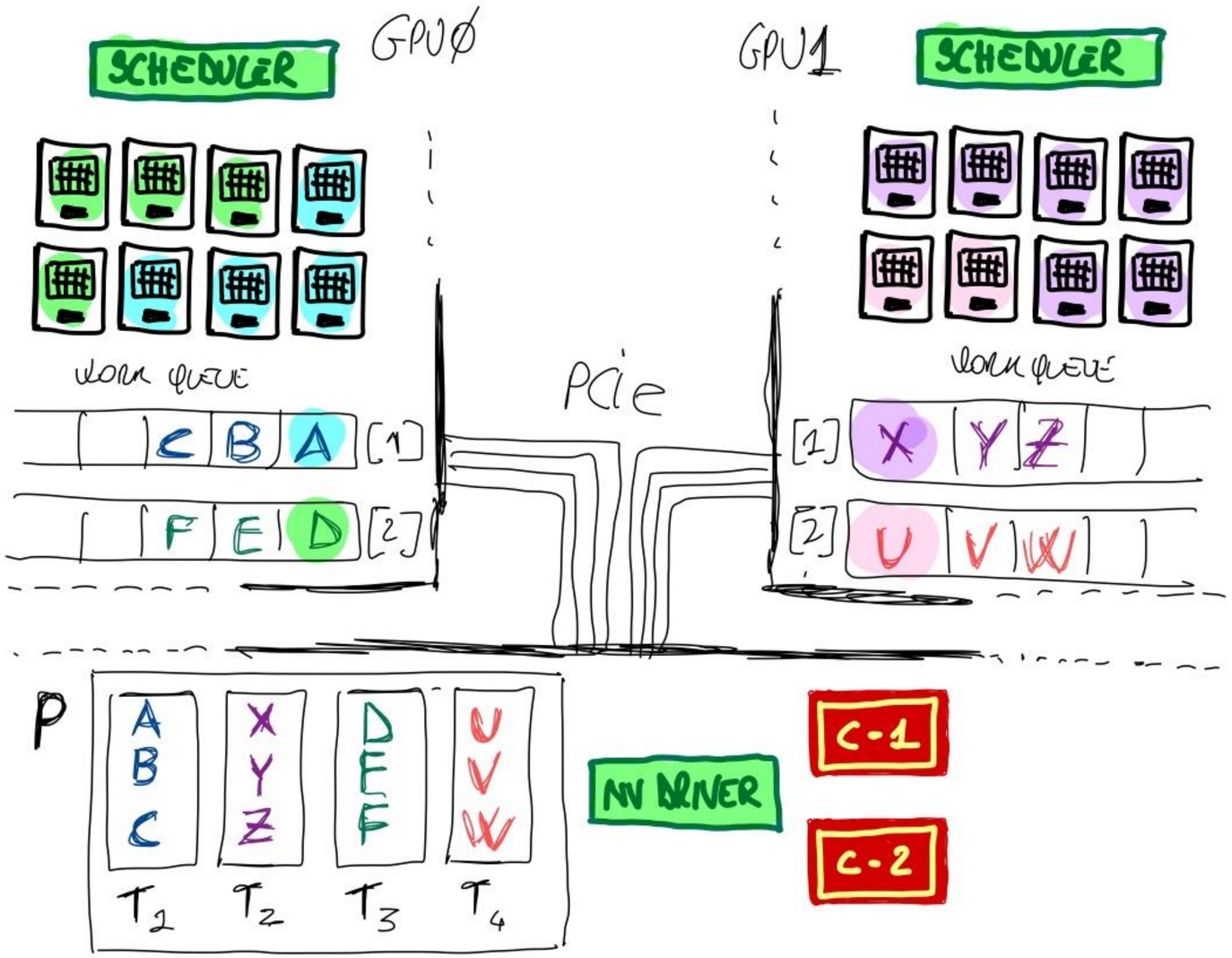
Again, one GPU CONTEXT per GPU (and per process). Dispatch strategy change!





Multiple OpenMP threads issuing kernels to different GPUS

Execution on each GPU happens independently and asynchronously. Concurrency is handled by each GPU.



What the code looks like

#OpenMP threads == #GPU

```
do j=0,num_gpus-1
  call acc_set_device_num(j,acc_device_nvidia)
  !$acc parallel loop
  do i=1,N
       pos = i+(j*CHUNCK_SIZE))
       a(pos) = pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        b(pos) = 2.0 * pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        c(pos) = a(pos) + b(pos)
   end do
end do
```

without OpenMP

```
!$omp parallel do num_threads(num_gpus)
do j=0,num_gpus-1
  call acc_set_device_num(j,acc_device_nvidia)
  !$acc parallel loop
  do i=1,N
       pos = i+(j*CHUNCK_SIZE))
       a(pos) = pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        b(pos) = 2.0 * pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        c(pos) = a(pos) + b(pos
   end do
end do
```

with OpenMP

ASSUMPTION: Data has been moved in the right place already (not straightforward task)

What the code looks like

#OpenMP threads > #GPU

```
do j=0,num_ chunks-1
  call acc_set_device_num(j%num_gpus,…)
  !$acc parallel loop
  do i=1,N
       pos = i+(j*CHUNCK_SIZE))
       a(pos) = pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        b(pos) = 2.0 * pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        c(pos) = a(pos) + b(pos)
   end do
end do
```

without OpenMP

```
!$omp parallel do
do j=0,num_chunks-1
  call acc_set_device_num(j%num_gpus,…)
  !$acc parallel loop
  do i=1,N
       pos = i+(j*CHUNCK_SIZE))
       a(pos) = pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        b(pos) = 2.0 * pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        c(pos) = a(pos) + b(pos
   end do
end do
```

with OpenMP

ASSUMPTION: Data has been moved in the right place already (not straightforward task)

What the code looks like

#OpenMP threads > #GPU + Streams

```
!$omp parallel do
do j=0,num_ chunks-1
  call acc_set_device_num(j%num_gpus,…)
  !$acc parallel loop
  do i=1,N
       pos = i+(j*CHUNCK_SIZE))
       a(pos) = pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        b(pos) = 2.0 * pos
   end do
   !$acc parallel loop
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        c(pos) = a(pos) + b(pos)
   end do
end do
```

without async

```
!$omp parallel do
do j=0,num_chunks-1
  call acc_set_device_num(j%num_gpus,…)
  !$acc parallel loop async(1)
  do i=1,N
       pos = i+(j*CHUNCK_SIZE))
       a(pos) = pos
   end do
   !$acc parallel loop async(2)
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        b(pos) = 2.0 * pos
   end do
   !$acc parallel loop wait(1,2)
   do i=1,N
        pos = i+(j*CHUNCK_SIZE))
        c(pos) = a(pos) + b(pos
   end do
end do
```

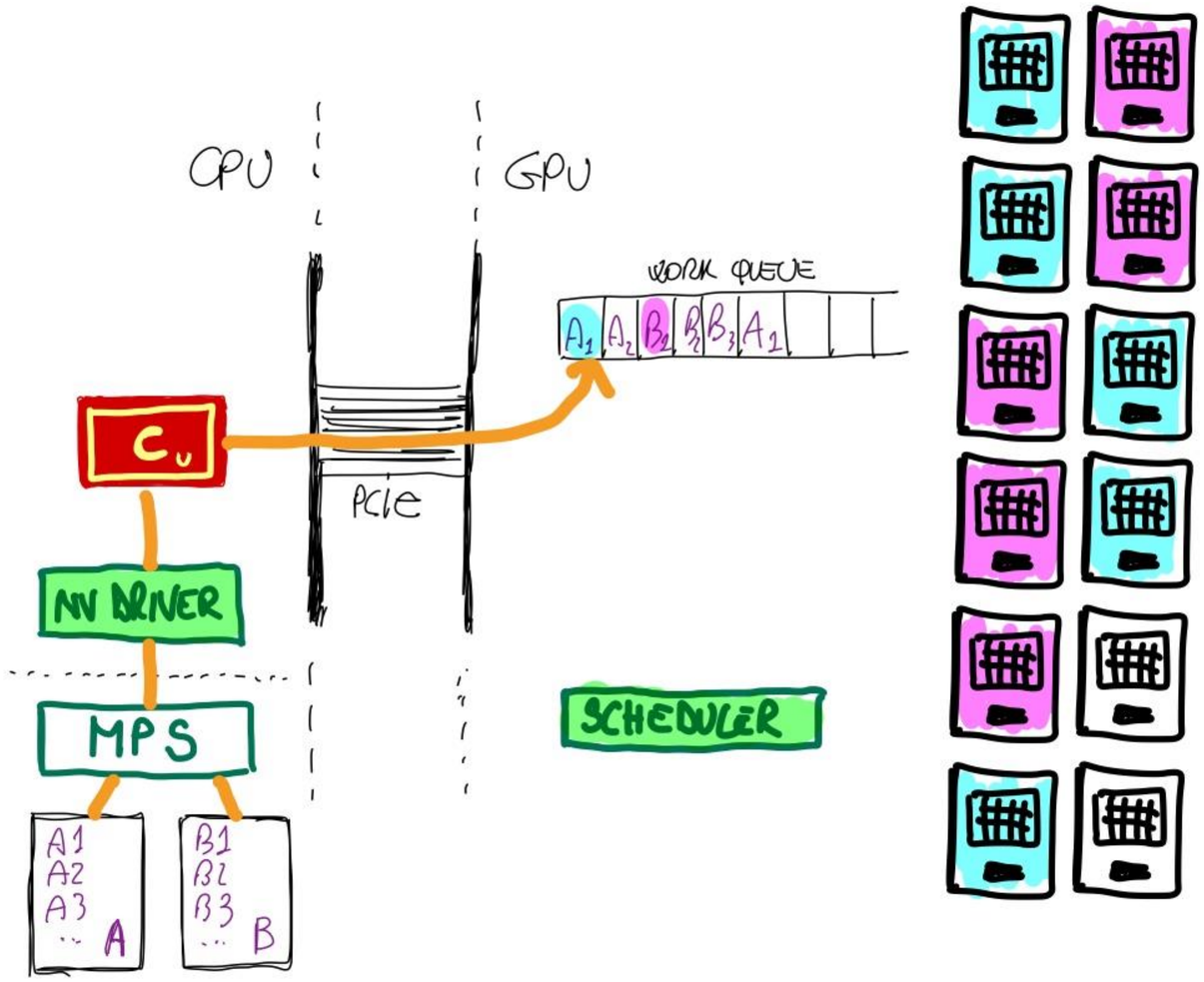
with async

ASSUMPTION: Data has been moved in the right place already (not straightforward task)

How to use NVIDIA Multi-Process Service (MPS)

What we want to achieve

MULTIPLE INDIPENDENT PROCESSES sharing GPU resources by running concurrent kernels simultaneously



NVIDIA Multi-Process Service (MPS)

- Designed to concurrently map multiple MPI ranks onto a single GPU
- Used when each rank is too small to fill the GPU on its own (efficient oversubscription)
  - No free lunch theorem still applies: if GPU is fully utilized, cannot get faster answers
- Strive to write your application so that you don’t need MPS
  - Profile your code to understand why MPS did or did not help

Things to watch out for:

- Memory Footprint
  - To provide a per-thread stack, GPU runtime reserves >=1kB of GPU memory per thread.
- Each MPS process also uploads a new copy of the executable code, which adds to the memory footprint
- Work Queue Sharing
  - MPS allows 96 hardware queues to be shared among up to 48 clients
  - MPS automatically reduces connections-per-client unless environment variable is set

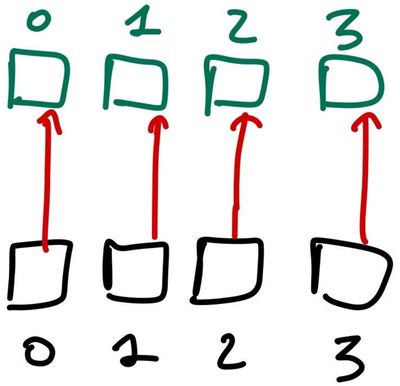
How to use NVIDIA MPS?

General guidelines

- No application modifications necessary, appropriate process binding required
- NVIDIA MPS runs in user-mode
- NVIDIA MPS primarily operates on a per-GPU basis
  - It manages resources and concurrency within a single GPU
  - On a node with multiple GPUs, it doesn't provide a unified memory space across them
  - Each GPU will have its own MPS server instance
  - During MPS initialization, mask out unrelated GPUs via CUDA_VISIBLE_DEVICE
- NVIDIA MPS can manage multiple GPU within a node
  - No automatic distribution system to route different jobs to different GPUs.
- Profiling tools are MPS-aware

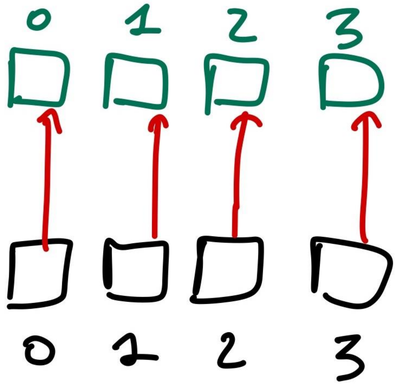
How to use NVIDIA MPS?

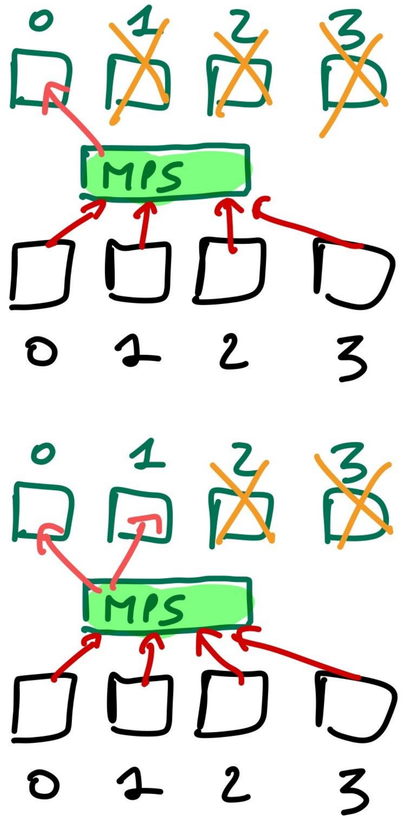
One MPS driving multiple GPUs



How to use NVIDIA MPS?

One MPS driving multiple GPUs

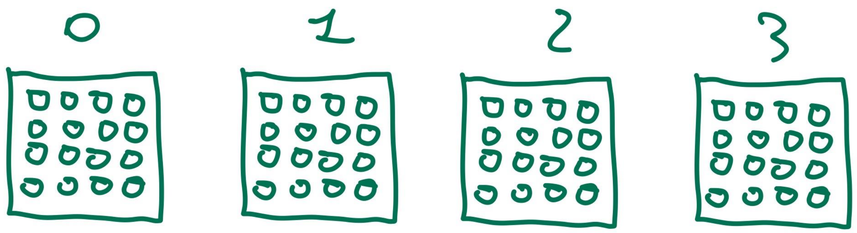


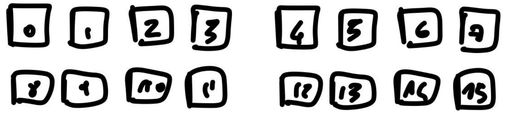


??? HOW MPS USES MULTIPLE GPUS ???

How to use NVIDIA MPS?

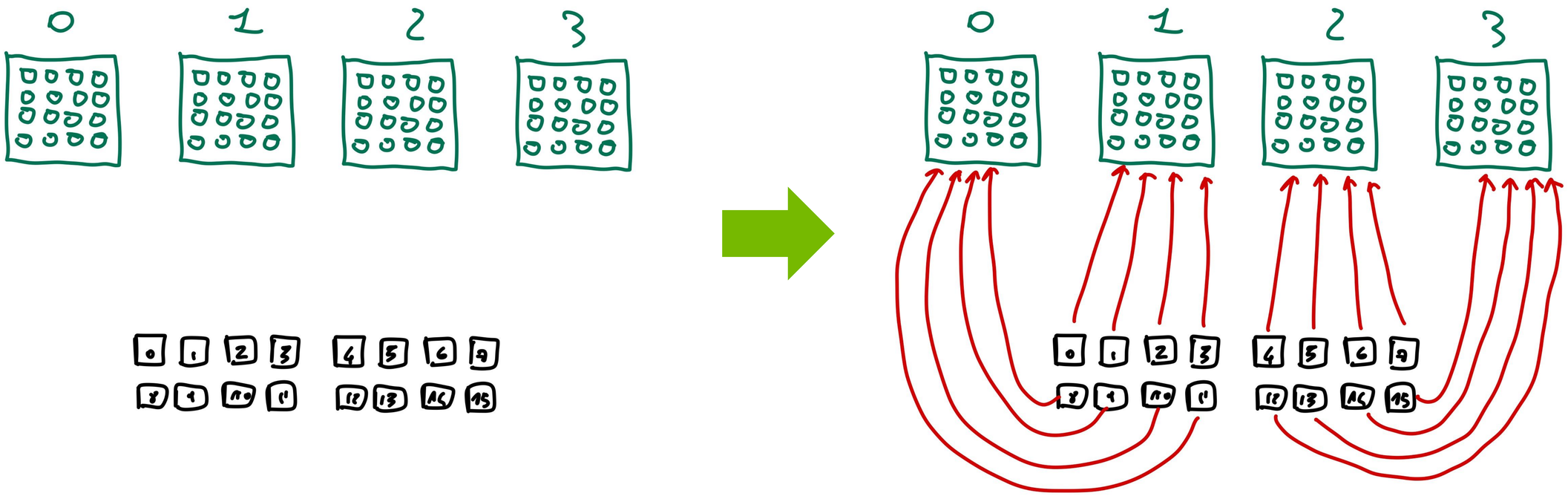
Multiple MPS driving multiple GPUs





How to use NVIDIA MPS?

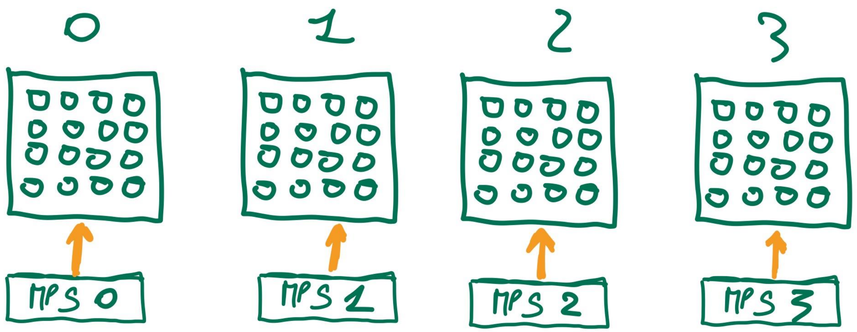
Multiple MPS driving multiple GPUs

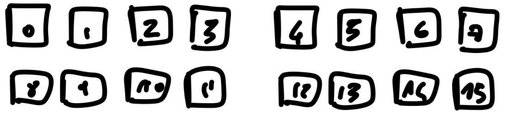


INEFFICIENT!

How to use NVIDIA MPS?

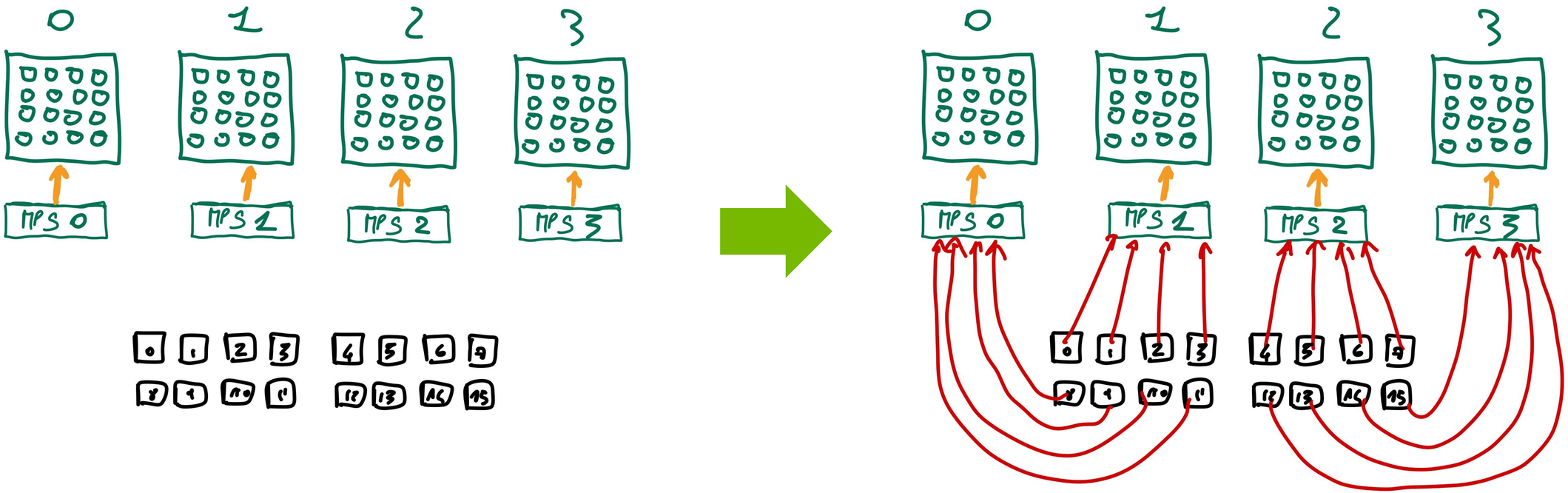
Multiple MPS driving multiple GPUs





How to use NVIDIA MPS?

Multiple MPS driving multiple GPUs



BEST MAPPING!

Demo / Hands-On

1. Profile script (DAY6/nsys-wrapper)
2. Mandelbrot (DAY6/mandelbrot-acc)
3. Laplace (DAY6/laplace-acc)

GROMACS with MPS (1/N)

Running single and multi GPU simulations

Input: 16K molecules

Modules:

```
module purge
ml load profile/chem-phys
ml load gromacs/2024.2--openmpi--4.1.6--gcc--12.2.0-cuda-12.2
```

How to run a simulation:

```
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=4
#SBATCH --gres=gpu:4

gmx_mpi grompp -f pme.mdp

srun gmx_mpi mdrun -ntomp ${OMP_NUM_THREADS} -noconfout -nsteps 16000 -nstlist 300 -nb gpu -update gpu -pme gpu -npme 1 -dlb
no -v [-gpu_id 0123]
```

How to check performance (“md.log”):

```
 Core t (s)   Wall t (s)       (%)
     Time:     XXX.XXX          XXX.XXX       XXX.XXX
               (ns/day)    (hour/ns)
Performance:     XXX.XXX            XXX.XXX
```

GROMACS with MPS (1/N)

Strating one MPS server per node, controlling multiple GPUs

```
#SBATCH --gres=gpu:4

srun ./mps-wrapper.sh -- gmx_mpi mdrun -ntomp ${OMP_NUM_THREADS} -noconfout -nsteps 16000 -nstlist 300 -nb gpu -update gpu -
pme gpu -npme 1 -dlb no –v
```

mps-wrapper.sh

```
export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps
export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log

# Launch MPS from a single rank per node
if [ $SLURM_LOCALID -eq 0 ]; then
    CUDA_VISIBLE_DEVICES=0,1,2,3 nvidia-cuda-mps-control -d
fi

# Wait for MPS to start sleep 5
sleep 5

# No longer need to see other GPUs, MPS is active
export CUDA_VISIBLE_DEVICES=0
exec "$@"
```

GROMACS with MPS (1/N)

Strating one MPS server per physical GPU

```
srun ./mps-wrapper-multi.sh -- gmx_mpi mdrun -ntomp ${OMP_NUM_THREADS} -noconfout -nsteps 16000 -nstlist 300 -nb gpu -update gpu -pme gpu -npme 1 -dlb no –v
```

mps-wrapper-multi.sh

```
# For each GPU in a node, start as many MPS service as visible GPUs
if [ $SLURM_LOCALID -eq 0 ]; then
   nGPUs=`nvidia-smi -L | wc -l`
   for ((i=0; i< ${nGPUs}; i++))
   do
      mkdir /tmp/mps_$i
      mkdir /tmp/mps_log_$i
      export CUDA_MPS_PIPE_DIRECTORY=/tmp/mps_$i
      export CUDA_MPS_LOG_DIRECTORY=/tmp/mps_log_$i
      CUDA_VISIBLE_DEVICES=$i nvidia-cuda-mps-control -d
   done
fi

# Wait for all MPS services to start
sleep 5

# Enforce appropriate MPS based on local rank
export CUDA_VISIBLE_DEVICES=0
GPU_PIPE=$(( ${SLURM_LOCALID} % nGPUs))
export CUDA_MPS_PIPE_DIRECTORY=/tmp/mps_${GPU_PIPE}
exec "$@"
```

GROMACS with MPS (1/N)

[TASK] Performance Evaluation

	Strategy	Performance [ns/day]	Speedup
(A)	4 MPI, 4 GPU, 4 OMP	???	???
(B)	8 MPI, 4 GPU, 4 OMP	???	???
(C)	16 MPI, 4 GPU, 2 OMP	???	???
(D)	16 MPI, 4 GPU, 2 OMP with MPS (per node)	???	???
(E)	16 MPI, 4 GPU, 2 OMP with MPS (per GPU)	???	???

GROMACS with MPS (1/N)

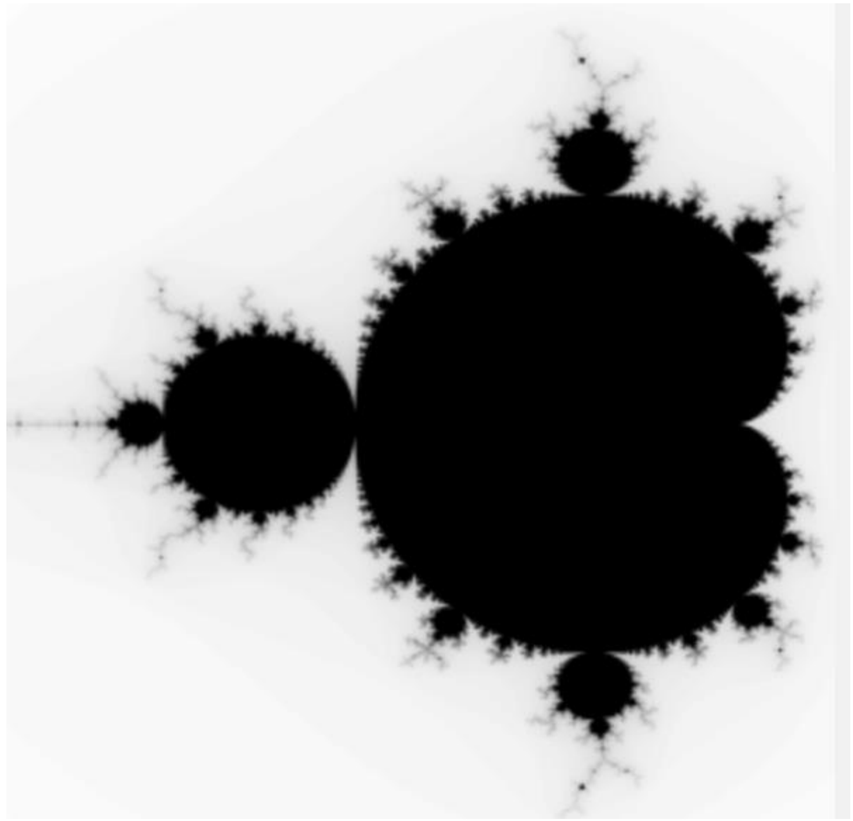
[SOLUTION] Performance Evaluation

	Strategy	Performance [ns/day]	Speedup
(A)	4 MPI, 4 GPU, 4 OMP	784	1.0x
(B)	8 MPI, 4 GPU, 4 OMP	81	-9.7x
(C)	16 MPI, 4 GPU, 2 OMP	36	-22x
(D)	16 MPI, 4 GPU, 2 OMP with MPS (per node)	205	-3.8x
(E)	16 MPI, 4 GPU, 2 OMP with MPS (per GPU)	570	0.7x

15.8x

Parallelization of Mandelbrot

Fully independent update of each cell



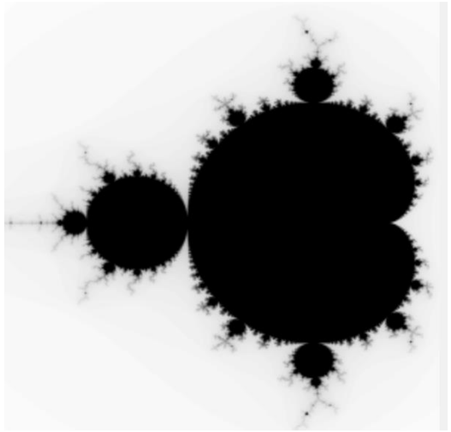
```
#pragma acc parallel loop
for(int y=0;y<HEIGHT;y++) {
  for(int x=0;x<WIDTH;x++) {
    image[y*WIDTH+x]=mandelbrot(x,y);
  }
}
```

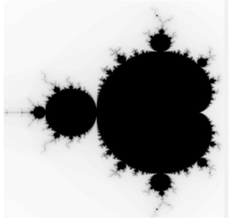
```
#pragma acc routine seq
unsigned char mandelbrot(int Px, int Py);
```

```
unsigned char mandelbrot(int Px, int Py) {
  …
  for(i=0;x*x+y*y<4.0 && i<MAX_ITERS;i++) {
    double xtemp=x*x-y*y+x0;
    y=2*x*y+y0;
    x=xtemp;
  }
  return (double)MAX_COLOR*i/MAX_ITERS;
}
```

Parallelization of Mandelbrot

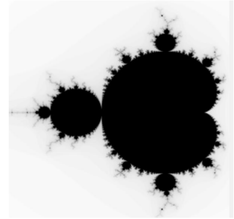
Pipelining compute and communication

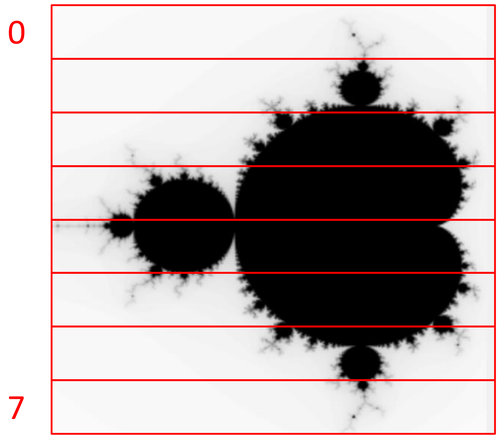




Parallelization of Mandelbrot

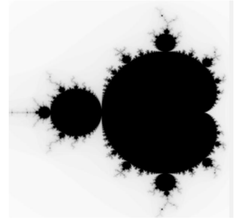
Pipelining compute and communication

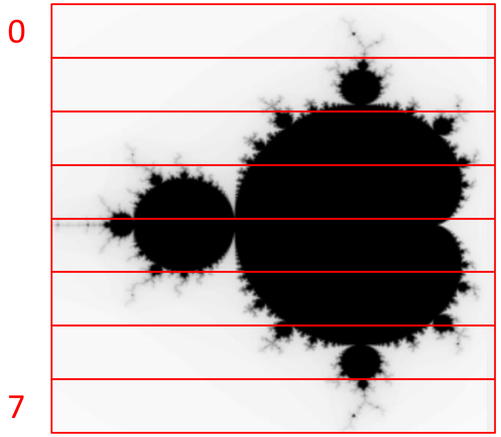


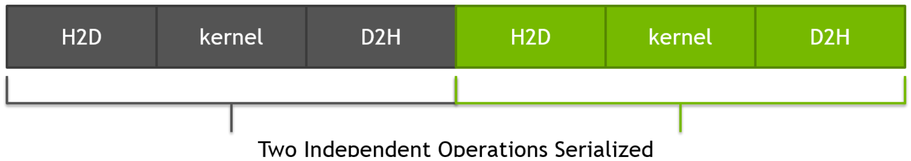


Parallelization of Mandelbrot

Pipelining compute and communication

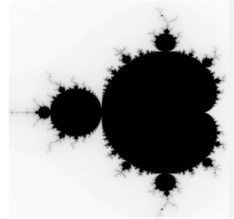


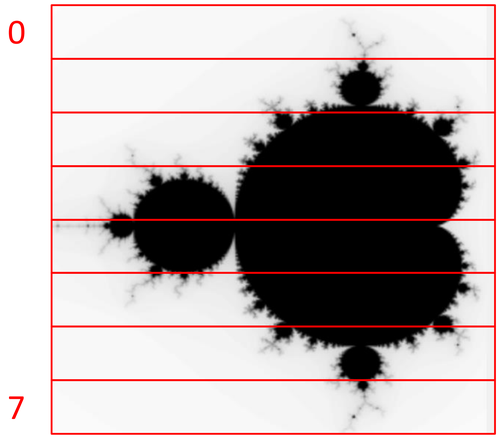


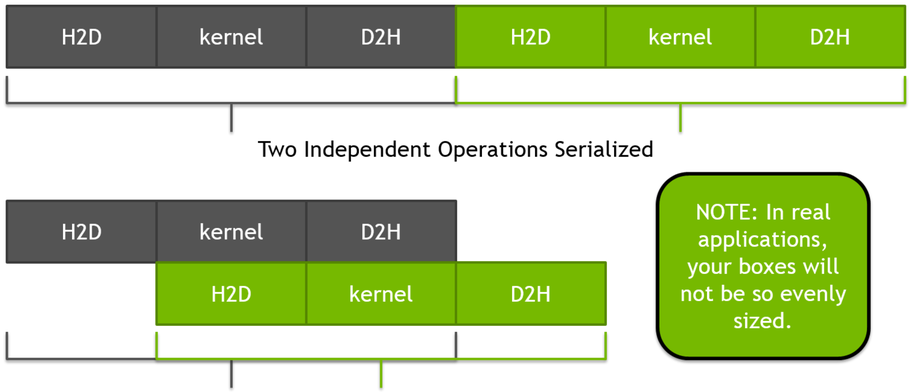


Parallelization of Mandelbrot

Pipelining compute and communication



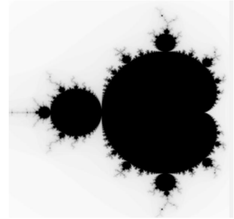


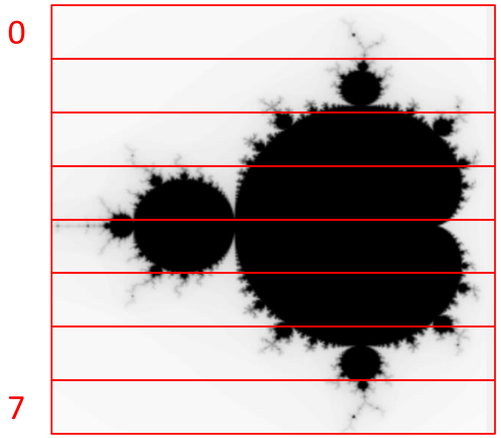


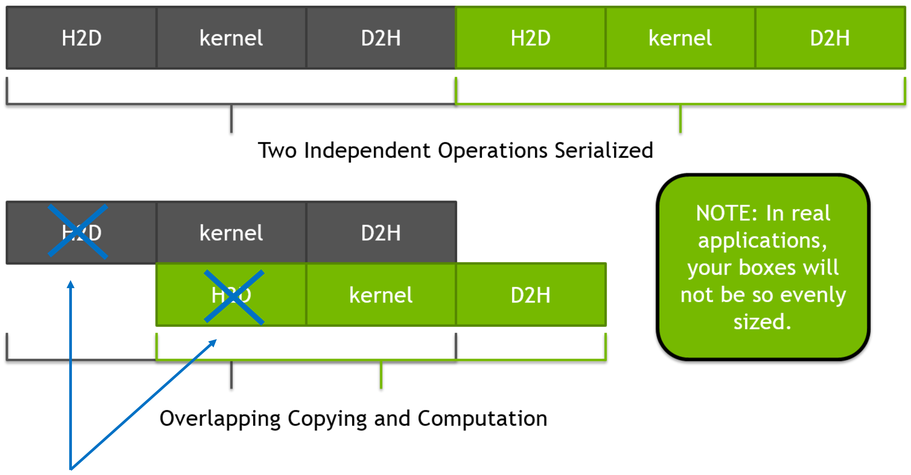
NOTE: In real applications, your boxes will not be so evenly sized.

Parallelization of Mandelbrot

Pipelining compute and communication





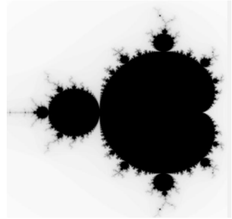


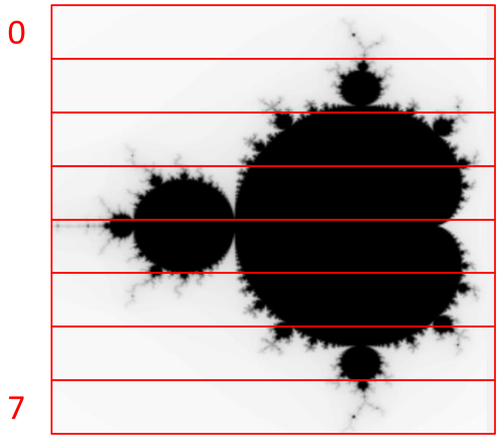
NOTE: In real applications, your boxes will not be so evenly sized.

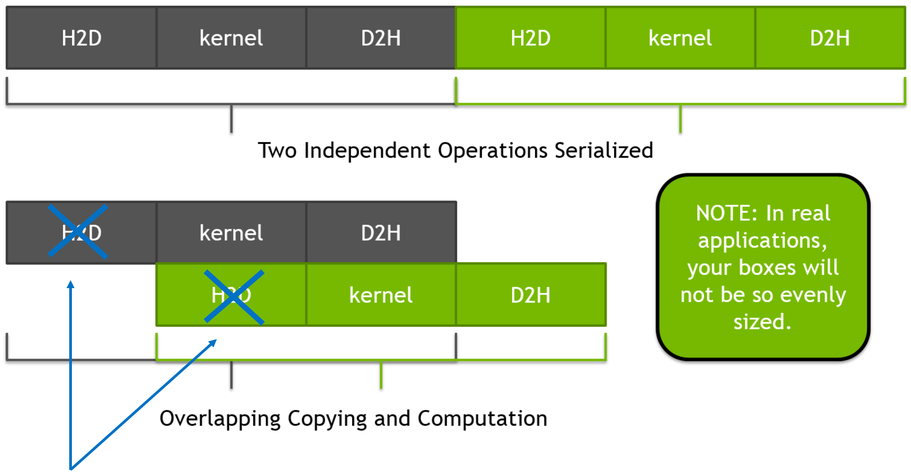
No need to copy on device,
let just create instead and copy back

Parallelization of Mandelbrot

Pipelining compute and communication





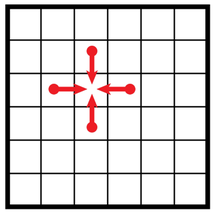


No need to copy on device,
let just create instead and copy back

```
num_blocks = 8
block_size = (HEIGHT/num_blocks)*WIDTH;

#pragma acc data create(image[WIDTH*HEIGHT])
for(int block = 0; block < num_blocks; block++ ) {

  int ystart = block * (HEIGHT/num_blocks),
  yend   = ystart + (HEIGHT/num_blocks);

  #pragma acc parallel loop
   for(int y=ystart;y<yend;y++) {
    for(int x=0;x<WIDTH;x++) {
      image[y*WIDTH+x]=mandelbrot(x,y);

   }

  #pragma acc update self(image[block*block_size:block_size])
}
```

Parallelization of Laplace Equation

Stencil-like operations, neighbors' dependency



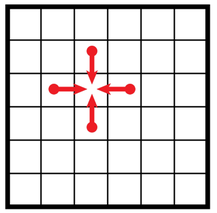
$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

$$\frac{(u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j} - 4u_{i,j})}{h^2} = 0$$

$$u_{i,j} = 0.25(u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j})$$

Parallelization of Laplace Equation

Stencil-like operations, neighbors' dependency



$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

$$\frac{(u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j} - 4u_{i,j})}{h^2} = 0$$

$$u_{i,j} = 0.25(u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j})$$

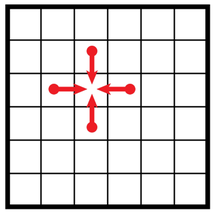
```
// grid size
#define GRIDY    2048
#define GRIDX    2048

double T_new[GRIDX+2][GRIDY+2;
double T[GRIDX+2][GRIDY+2];
```

```
// main computational kernel, average over neighbors
for(i = 1; i <= GRIDX; i++)
  for(j = 1; j <= GRIDY; j++)
    T_new[i][j] = 0.25 * (T[i+1][j] + T[i-1][j] +
                          T[i][j+1] + T[i][j-1]);
```

```
// compute the largest change and copy T_new to T
for(i = 1; i <= GRIDX; i++){
   for(j = 1; j <= GRIDY; j++){
      dt = MAX( fabs(T_new[i][j]-T[i][j]), dt);
      T[i][j] = T_new[i][j];
   }
}
```

Parallelization of Laplace Equation

Stencil-like operations, neighbors' dependency



$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

$$\frac{(u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j} - 4u_{i,j})}{h^2} = 0$$

$$u_{i,j} = 0.25(u_{i,j-1} + u_{i,j+1} + u_{i-1,j} + u_{i+1,j})$$

IMPORTANT: when chunking across multiple GPU need to consider HALO EXCHANGE

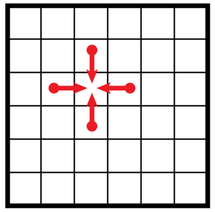
```
// grid size
#define GRIDY    2048
#define GRIDX    2048

double T_new[GRIDX+2][GRIDY+2;
double T[GRIDX+2][GRIDY+2];
```

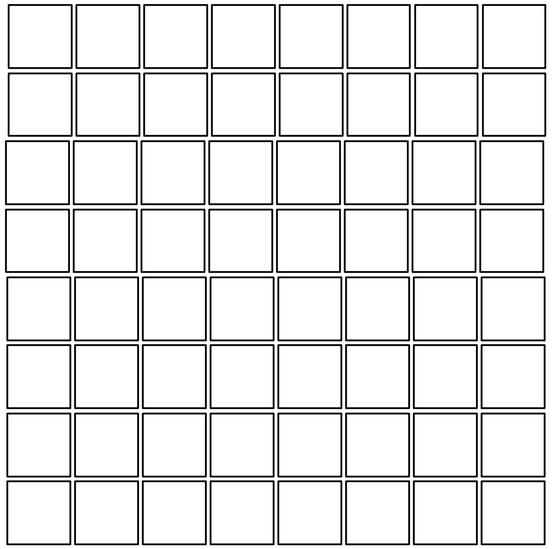
```
// main computational kernel, average over neighbors
for(i = 1; i <= GRIDX; i++)
  for(j = 1; j <= GRIDY; j++)
    T_new[i][j] = 0.25 * (T[i+1][j] + T[i-1][j] +
                          T[i][j+1] + T[i][j-1]);
```

```
// compute the largest change and copy T_new to T
for(i = 1; i <= GRIDX; i++){
   for(j = 1; j <= GRIDY; j++){
      dt = MAX( fabs(T_new[i][j]-T[i][j]), dt);
      T[i][j] = T_new[i][j];
   }
}
```

Parallelization of Laplace Equation

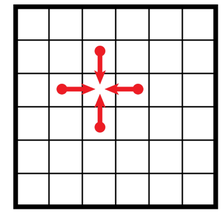
Halo Exchange

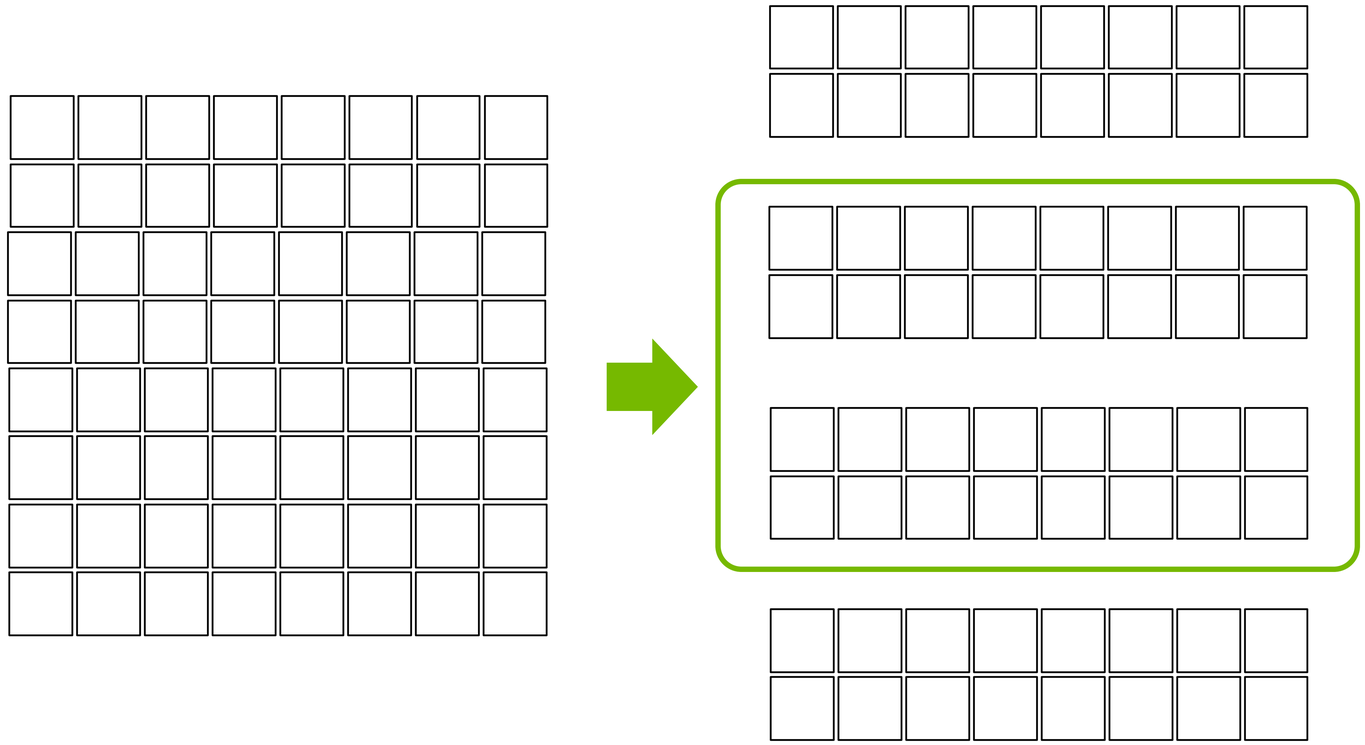




Parallelization of Laplace Equation

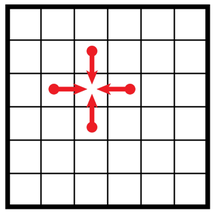
Halo Exchange

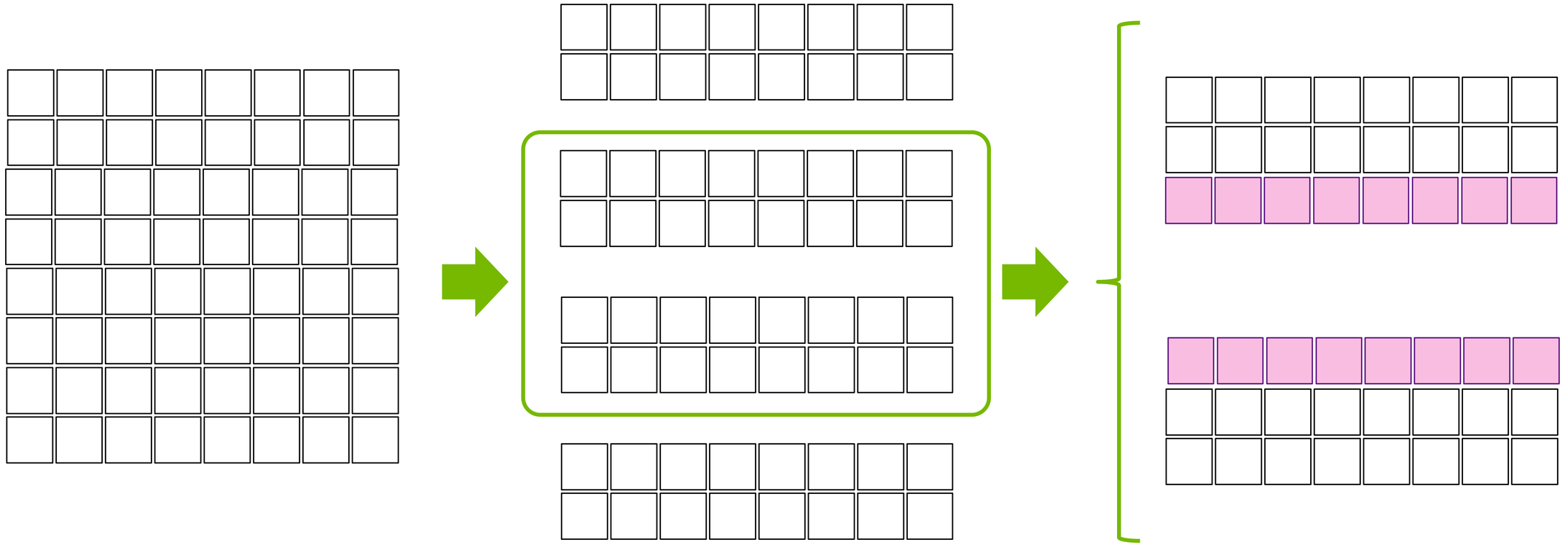




Parallelization of Laplace Equation

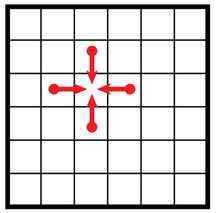
Halo Exchange

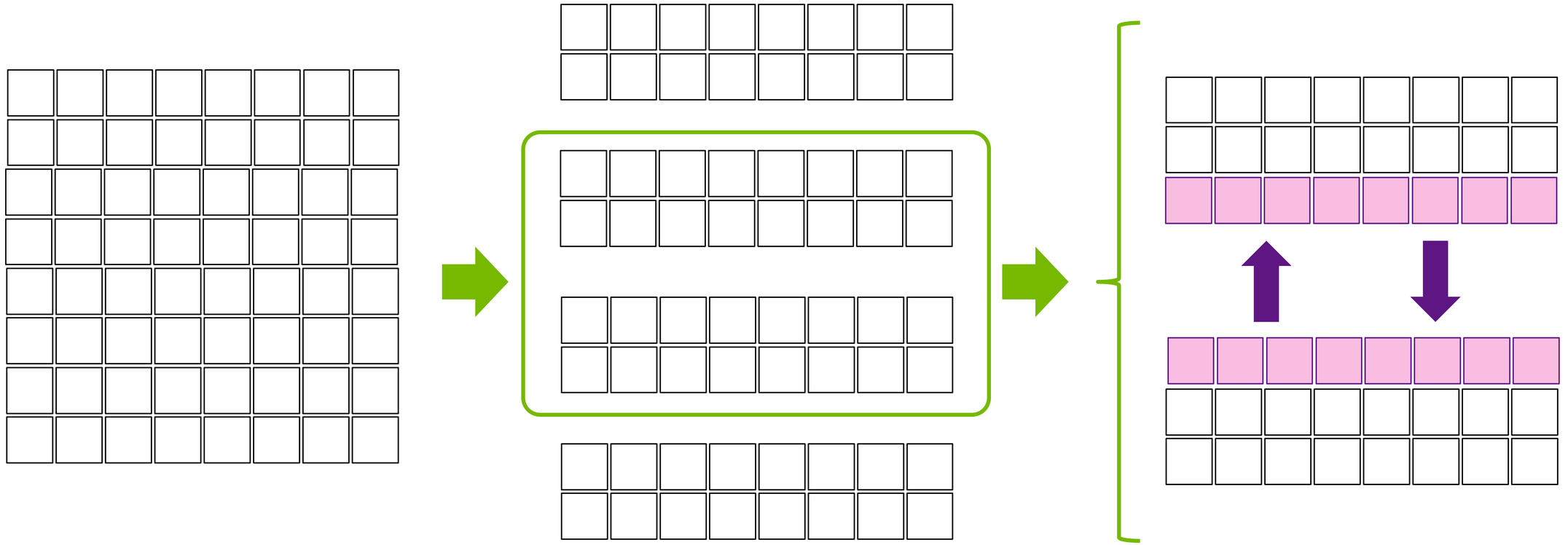




Parallelization of Laplace Equation

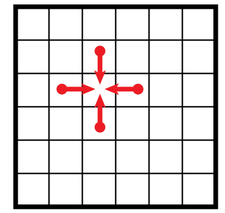
Halo Exchange

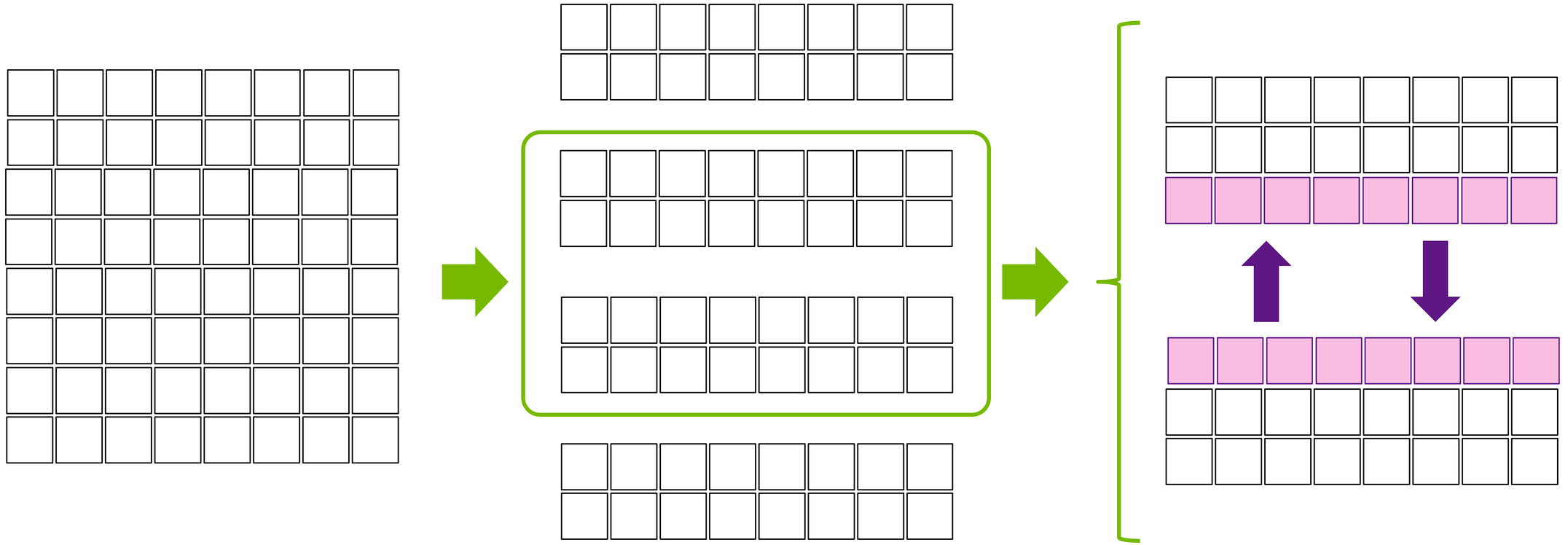




Parallelization of Laplace Equation

Halo Exchange





Why along X? In C, is simply convenience (hao exchange data is contiguous, no need of packing/unpacking routines)