Stanford University

CME 253A

# Introduction to High Performance Computing and Parallel (GPU) Computing

## Stanford Summer Session 5

15 July 2019 | Y2E2 111

Ludovic Räss

Stanford University, Department of Geophysics

lraess@stanford.edu

SIGMA

## Session 5 - CUDA MPI and Q&A

Today’s agenda

- Lecture: Multi-GPU implementation with CUDA MPI
- Programming: 1/ Conceptual MATLAB example
  2/ CUDA MPI 1D wave propagation
- Q&A regarding the final project

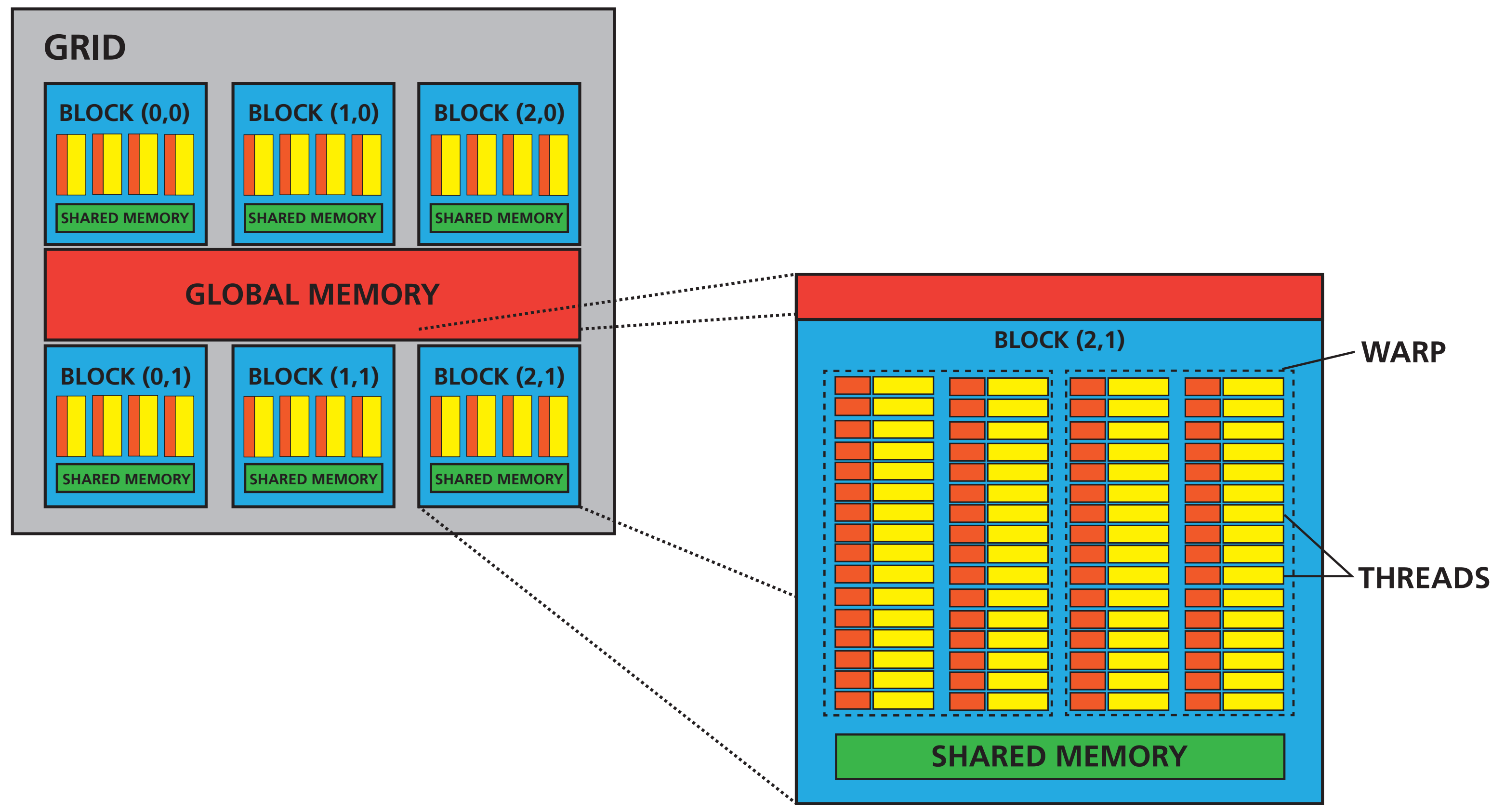
# Session 5 - CUDA MPI and Q&A

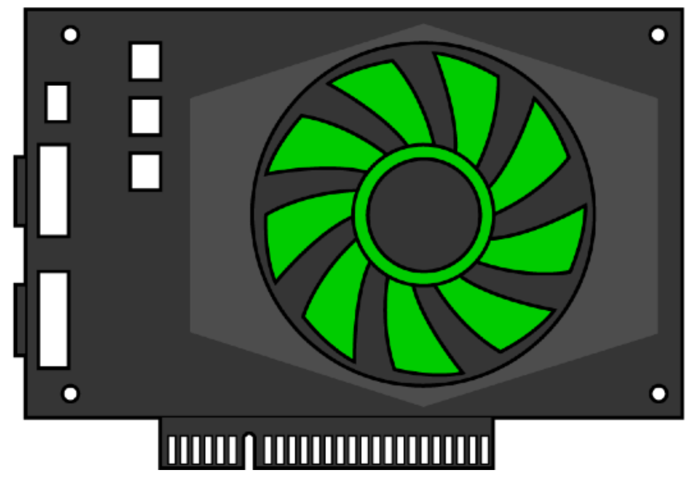
## Today's agenda

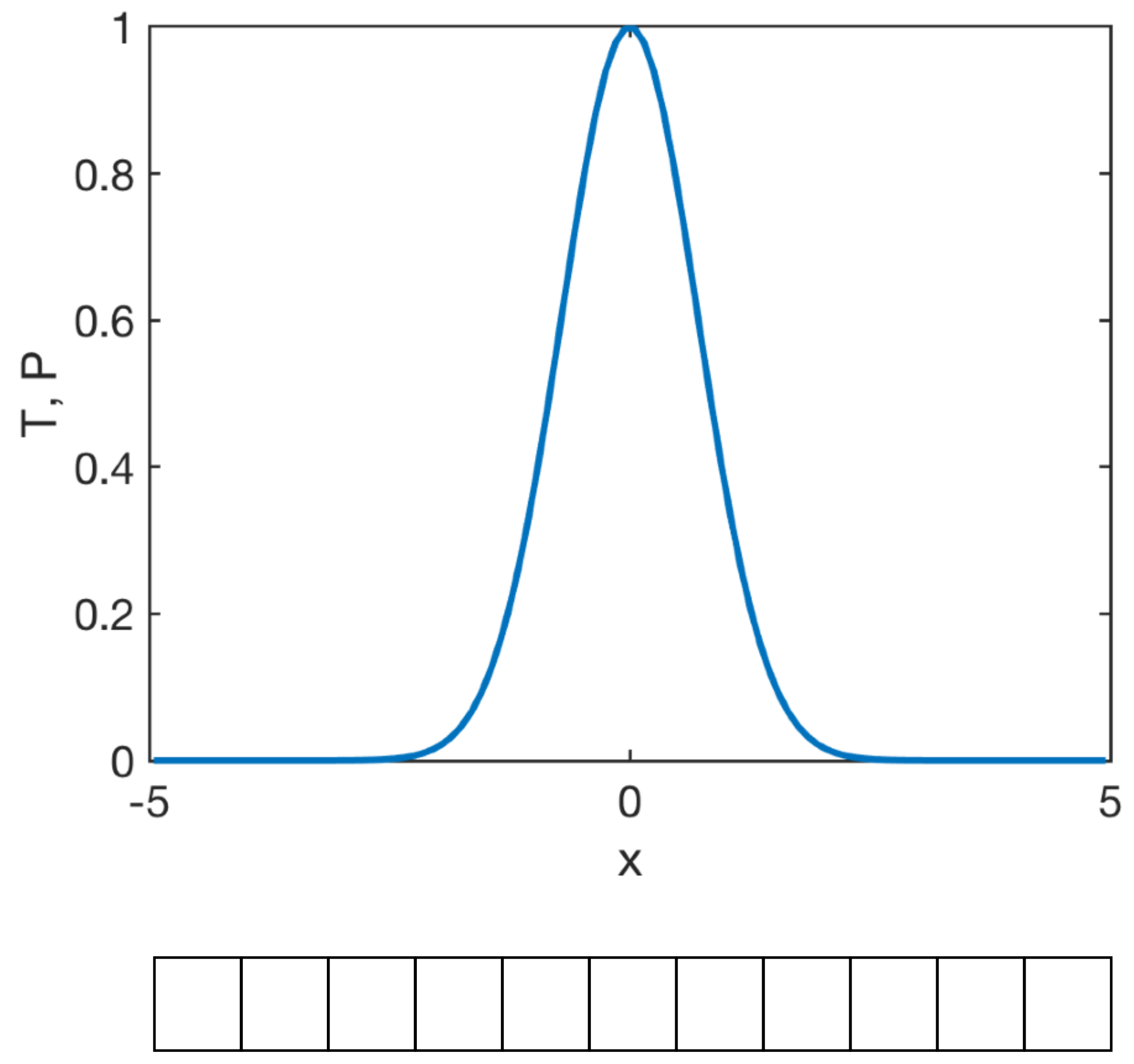
- Lecture: Multi-GPU implementation with CUDA MPI
- Programming: 1/ Conceptual MATLAB example
  2/ CUDA MPI 1D wave propagation
- Q&A regarding the final project

# Multi-GPU implementation with CUDA MPI

- Up to now we mapped and parallelised our FD domain on a single GPU
- All threads share the same global GPU memory

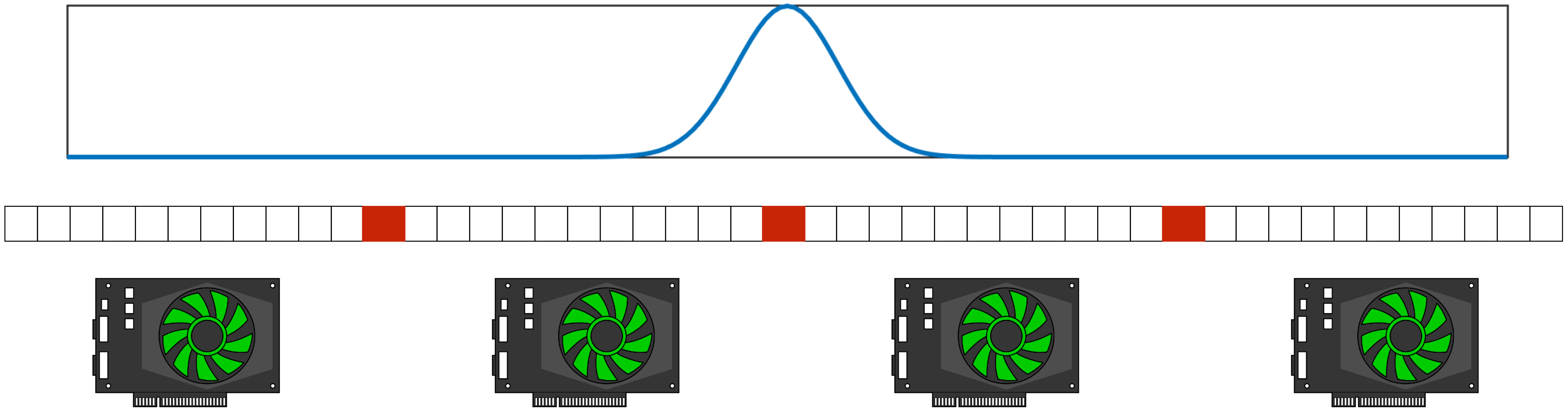






# Multi-GPU implementation with CUDA MPI

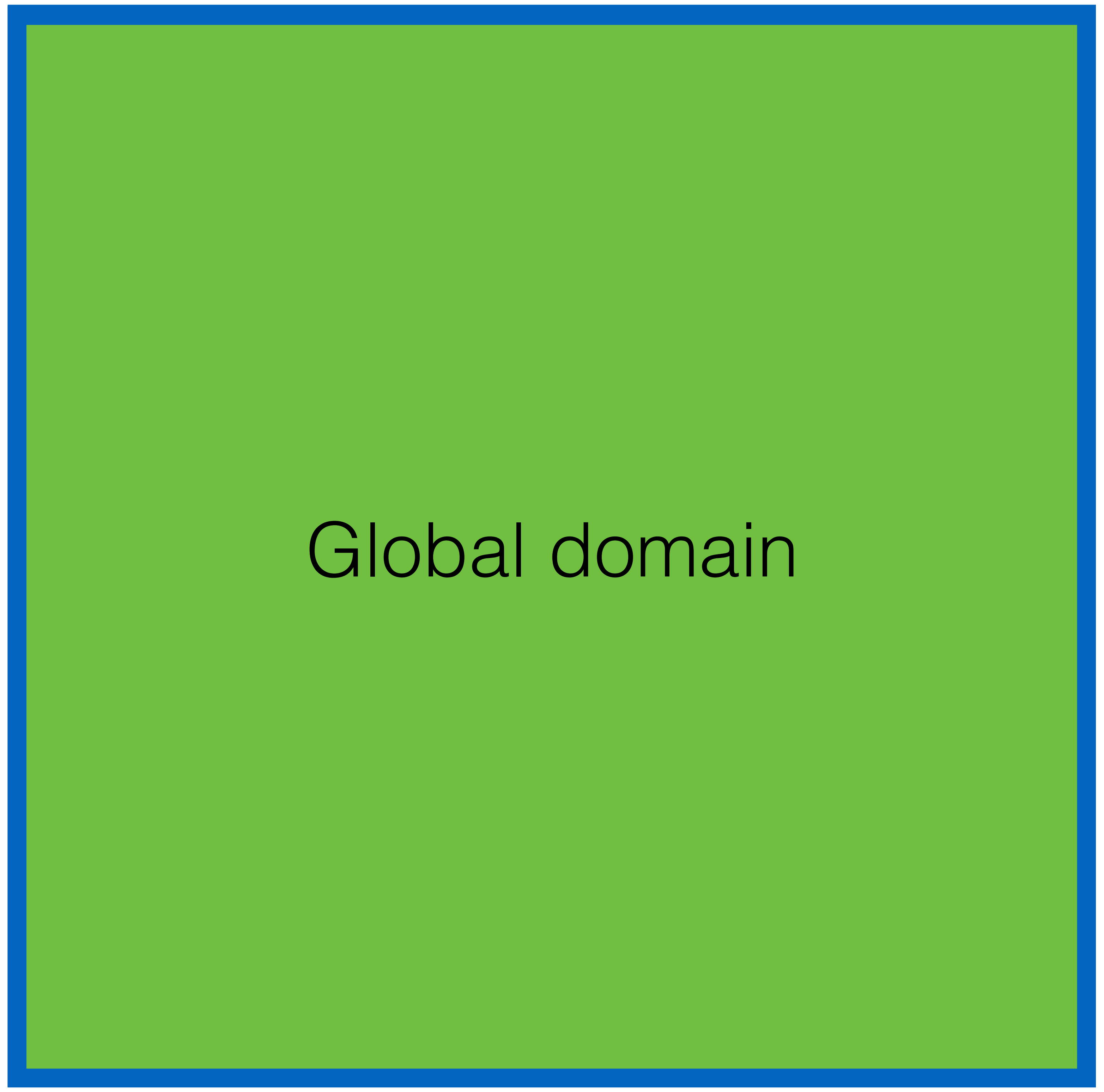
- How to use 2+ GPUs in order to compute larger domains
- Using multiple GPUs requires to handle distributed memory
- MPI (message passing interface) library is useful in this task



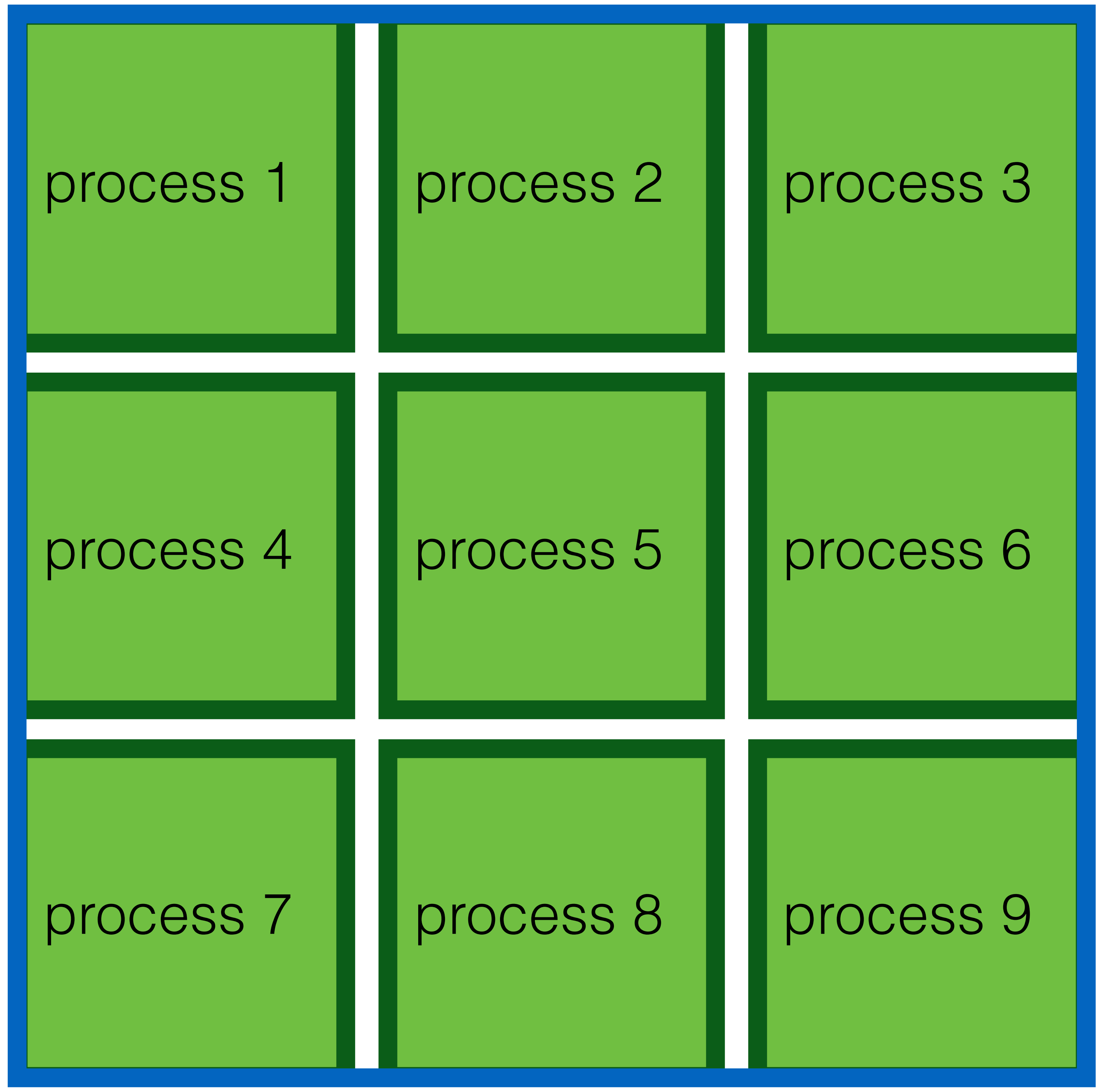
# Multi-GPU implementation with CUDA MPI

- Global domain
- Local processes
- Boundary conditions:
  - Global > physics
  - Local > MPI

Conceptual 2-D domain decomposition

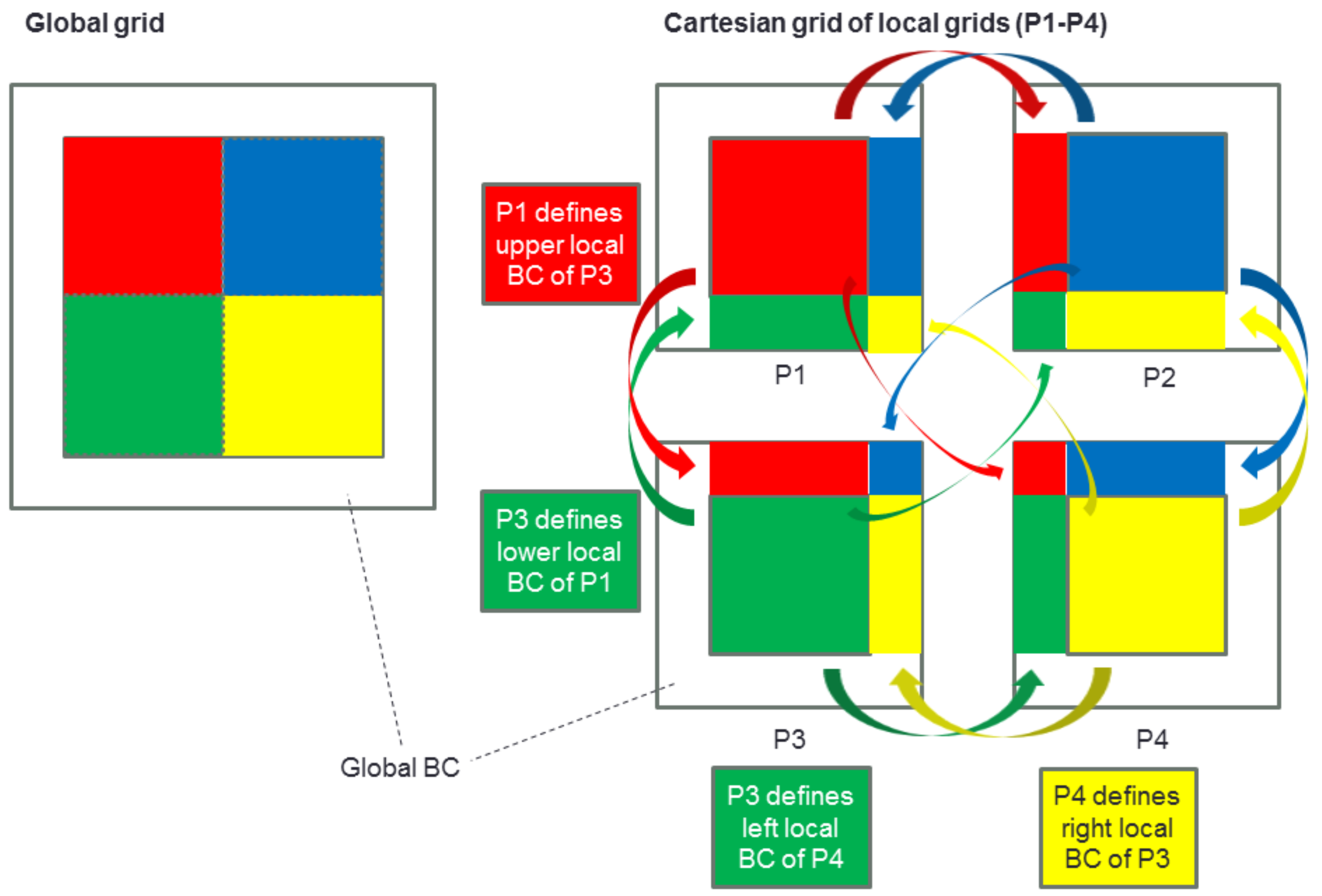


Conceptual 2-D domain decomposition



## Multi-GPU implementation with CUDA MPI

- All about exchanging the boundary values


Global grid
Cartesian grid of local grids (P1-P4)
P1 defines upper local BC of P3
P3 defines lower local BC of P1
P1
P2
P3
P4
Global BC
P3 defines left local BC of P4
P4 defines right local BC of P3

# Multi-GPU implementation with CUDA MPI

MPI - a CPU library that has (a priori) nothing to do with GPU knowledge

- Use MPI to assign one GPU to each MPI process
- “Vectorised” domain hierarchy: CUDA Threads, Blocks, Grid + MPI Dims
- Naive design: - use MPI rank to get gpu_id (get local rank per node for correct gpu_id)
  - exchange inner boundary conditions between neighbouring processes
  - send and receive host pointers via MPI, needs both CPU and GPU buffers

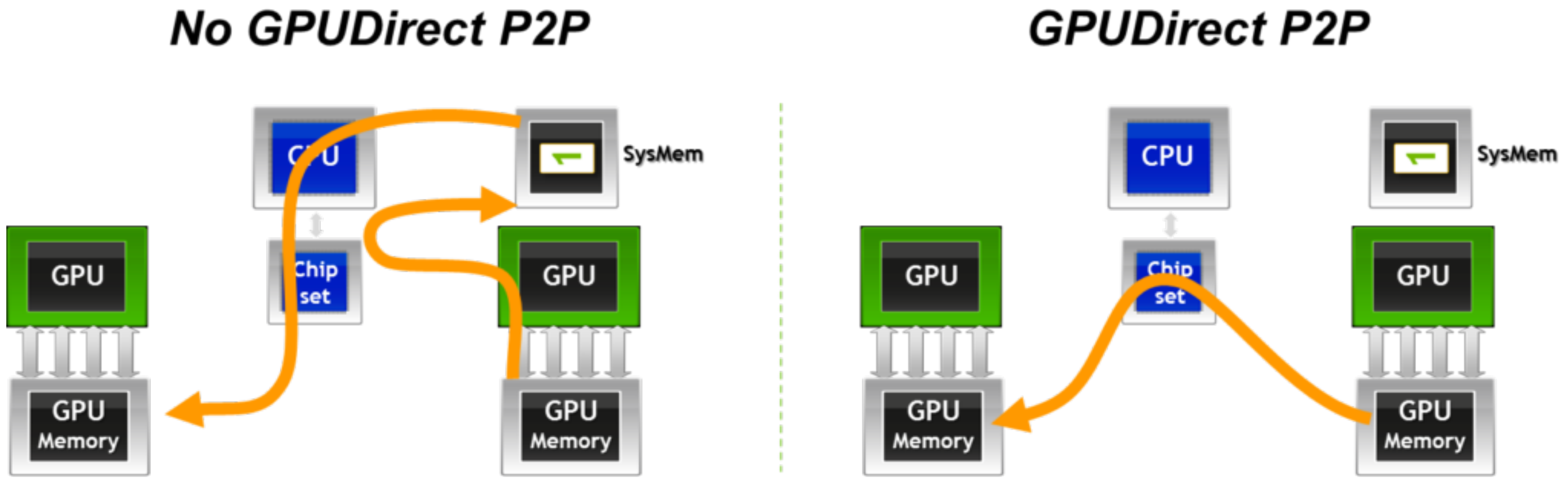
## Multi-GPU implementation with CUDA MPI

### CUDA-aware MPI

- MPI send and receive functions can accept device pointers (GPU)
- No need for explicit buffer copy on the host
- CPU and GPU MPI are nearly identical except:
  Context initialisation
  Pointers (host or device) passed to the send/recv MPI functions

# Multi-GPU implementation with CUDA MPI

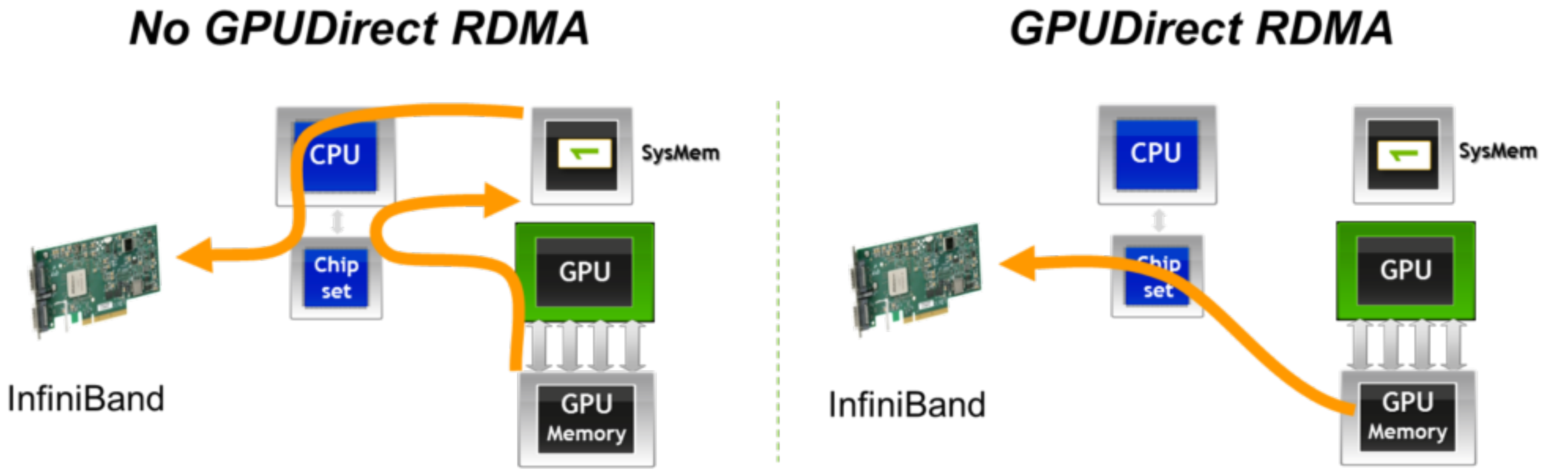
- GPU + MPI - what is going on within a multi-GPU node (Peer-to-peer P2P)



*https://devblogs.nvidia.com/introduction-cuda-aware-mpi/*

# Multi-GPU implementation with CUDA MPI

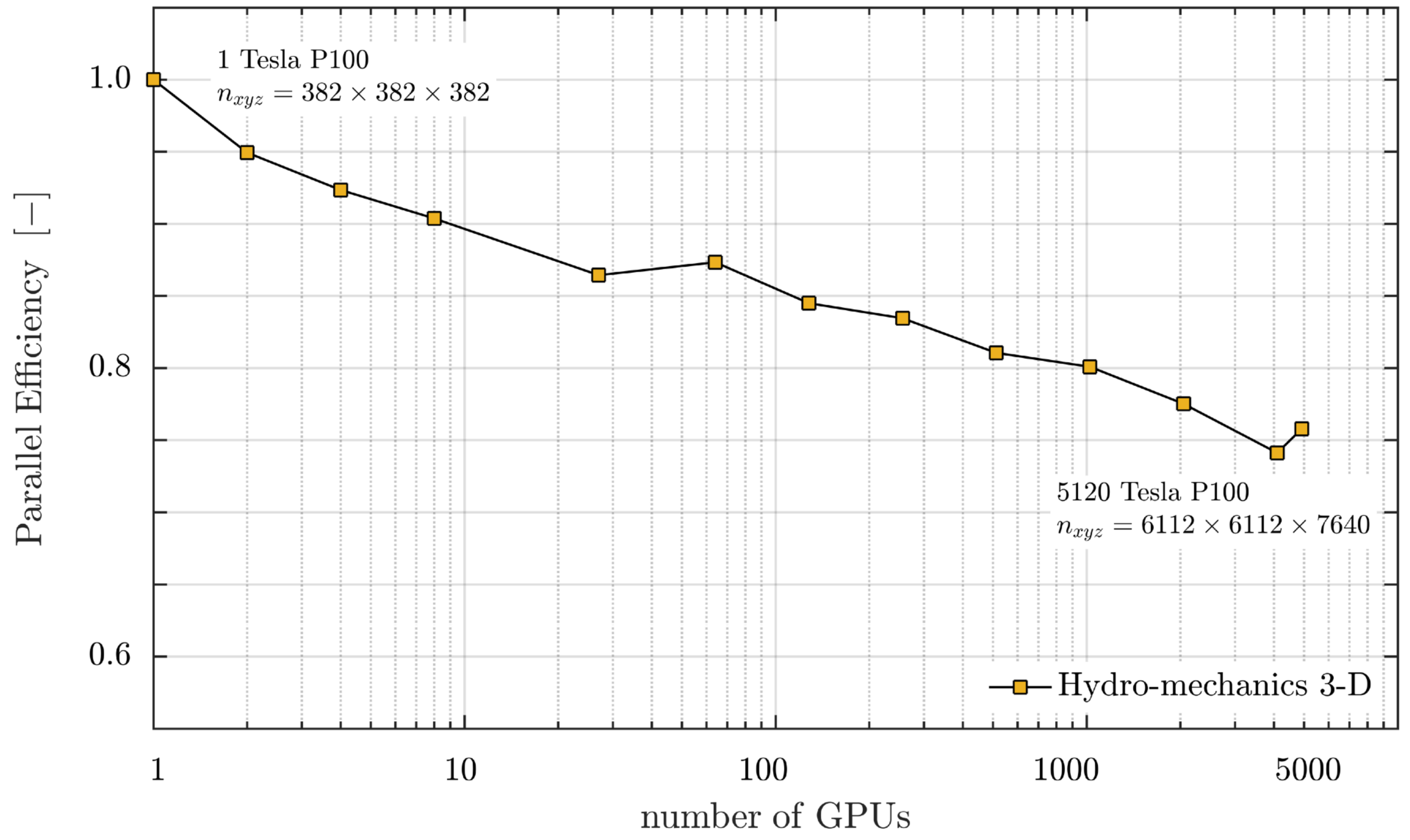
- GPU + MPI - what is going on between different GPU nodes (RDMA + IB)



https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

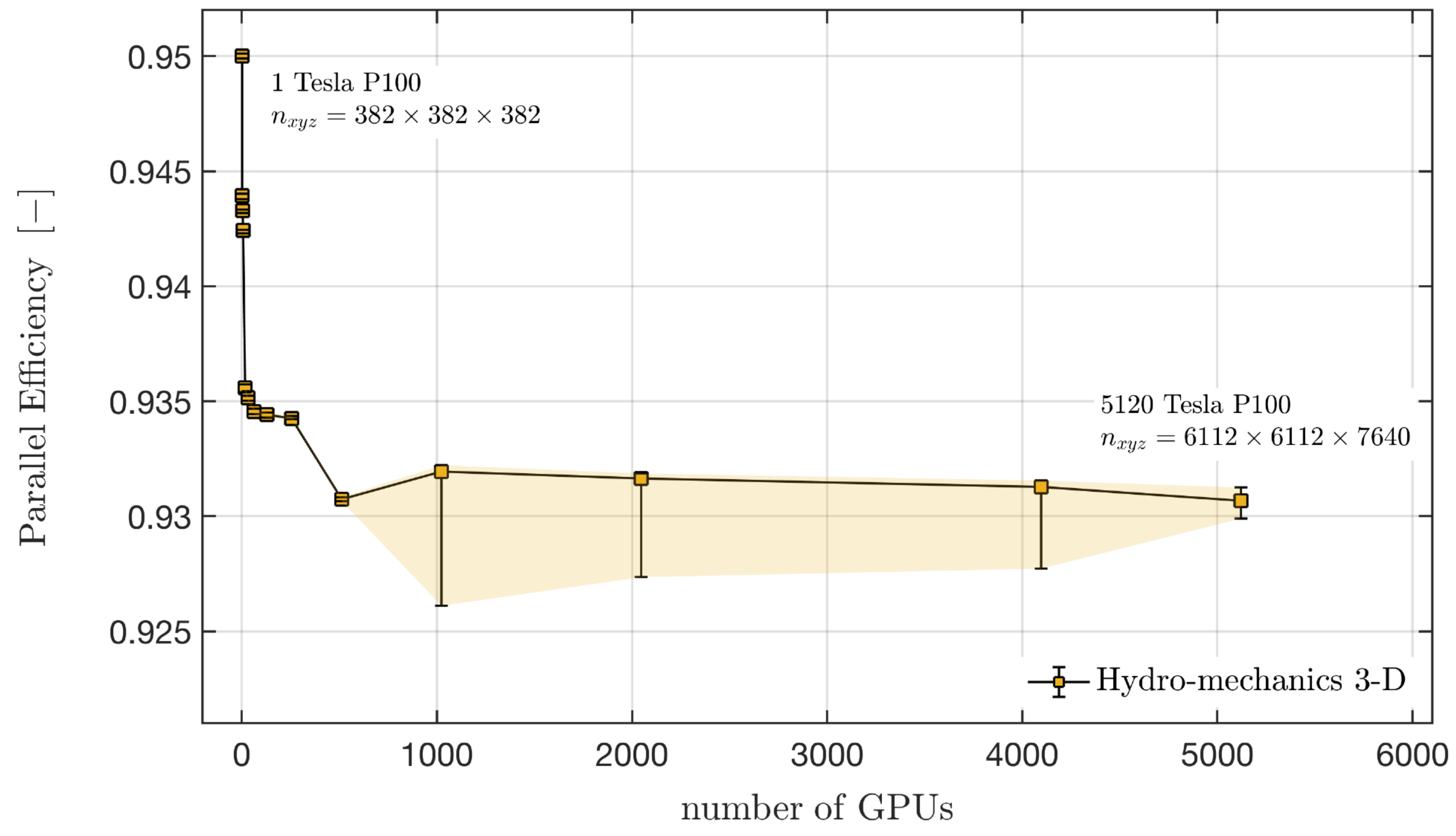
# Multi-GPU implementation with CUDA MPI

- Parallel efficiency
- How much time is lost in communication
- Ideal: close to 1
- MPI point-to-point shows ok perfs
- But perfs gets worse with growing number of MPI processes
- Good news - there is a fix !



## Multi-GPU implementation with CUDA MPI

- Parallel efficiency close to 1
- Hiding MPI communication with computations
- Straight forward implementation in using CUDA streams
- Works well with stencil codes and domain decomposition



# Session 5 - CUDA MPI and Q&A

## Today's agenda

- Lecture: Multi-GPU implementation with CUDA MPI
- Programming: 1/ Conceptual MATLAB example
  2/ CUDA MPI 1D wave propagation
- Q&A regarding the final project

## 1/ Conceptual MATLAB example

- Diffusion of a step function between to conceptually distinct domains
- Communicate de appropriate inner boundary condition

- Repeat the exercise with the 1D wave equation
- Generalise it to n-processes
- Verify it against a single process implementation

# 2/ CUDA MPI 1D wave propagation

## CUDA-aware MPI - the important steps

- Initialise the context and get GPU ID from MPI

```
cudaSetDeviceFlags(cudaDeviceMapHost);
const char* me_str     = getenv("OMPI_COMM_WORLD_RANK");
const char* me_loc_str = getenv("OMPI_COMM_WORLD_LOCAL_RANK");
me     = atoi(me_str);
me_loc = atoi(me_loc_str);
gpu_id = me_loc;
```

# 2/ CUDA MPI 1D wave propagation

## CUDA-aware MPI - the important steps

- MPI init functions; create a process world in cartesian coordinates, get neighbours

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
MPI_Dims_create(nprocs, NDIMS, dims);
MPI_Cart_create(MPI_COMM_WORLD, NDIMS, dims, periods, reorder, &topo_comm);
MPI_Comm_rank(topo_comm, &me);
MPI_Cart_coords(topo_comm, me, NDIMS, coords);
for (int i=0; i<NDIMS; i++){
  MPI_Cart_shift(topo_comm,i,1,&(neighbours_l[i]),&(neighbours_r[i]));
}
```

# 2/ CUDA MPI 1D wave propagation

## CUDA-aware MPI - the important steps

- MPI update sides
  - needs temporary buffers to store data to send (boundary values)
  - needs temporary buffers to store received data (boundary values)
  - needs to actually send and receive the data to be exchanged **`update_sides()`**

```
#define update_sides() \
if (neighbours_l[0] != MPI_PROC_NULL)   write_to_mpi_sendbuffer_l0<<<grid,block>>>(V_send_l0_d,V_d);     cudaDeviceSynchronize(); \
if (neighbours_r[0] != MPI_PROC_NULL)   write_to_mpi_sendbuffer_r0<<<grid,block>>>(V_send_r0_d,V_d,nx);  cudaDeviceSynchronize(); \
if (neighbours_l[0] != MPI_PROC_NULL){ MPI_Irecv((DAT*)V_recv_l0_d, 1, MPI_DAT, neighbours_l[0], tag, topo_comm, &(req[reqnr]));  reqnr++;  } \
if (neighbours_r[0] != MPI_PROC_NULL){ MPI_Irecv((DAT*)V_recv_r0_d, 1, MPI_DAT, neighbours_r[0], tag, topo_comm, &(req[reqnr]));  reqnr++;  } \
if (neighbours_l[0] != MPI_PROC_NULL){ MPI_Isend((DAT*)V_send_l0_d, 1, MPI_DAT, neighbours_l[0], tag, topo_comm, &(req[reqnr]));  reqnr++;  } \
if (neighbours_r[0] != MPI_PROC_NULL){ MPI_Isend((DAT*)V_send_r0_d, 1, MPI_DAT, neighbours_r[0], tag, topo_comm, &(req[reqnr]));  reqnr++;  } \
MPI_Waitall(reqnr,req,MPI_STATUSES_IGNORE);  reqnr=0;  for (int j=0; j<NREQS; j++){ req[j]=MPI_REQUEST_NULL; }; \
if (neighbours_l[0] != MPI_PROC_NULL)   read_from_mpi_recvbuffer_l0<<<grid,block>>>(V_d, V_recv_l0_d);     cudaDeviceSynchronize(); \
if (neighbours_r[0] != MPI_PROC_NULL)   read_from_mpi_recvbuffer_r0<<<grid,block>>>(V_d, V_recv_r0_d, nx); cudaDeviceSynchronize();
```

## 2/ CUDA MPI 1D wave propagation

### CUDA-aware MPI - further considerations

- Buffers can live in device memory only in case of CUDA-aware MPI
- Using Isend and Irecv for nonblocking message transmission > better performance
- Always start receiving first, then sending to avoid message losses and time loss

# Session 5 - CUDA MPI and Q&A

## Today’s agenda

- Lecture: Multi-GPU implementation with CUDA MPI
- Programming: 1/ Conceptual MATLAB example
  2/ CUDA MPI 1D wave propagation
- Q&A regarding the final project

# Projects

Projects due date: August 1, 2019 - (midnight Pacific time)

- Objective: a/ 3D viscous Stokes flow: Buoyancy driven rising sphere setup
  - or -
  b/ 3D elastic wave propagation: initial gaussian pressure anomaly

- Results: 1/ Report performance (MTP), convergence with resolution increase
  2a/ Viscous stokes: add convergence check to your code
  2b/ Elastic (acoustic) wave: add P and S wave recording in a place of the domain

- Nice plots and fancy 3D graphics is a plus

- Hand in a report (max 5 pages) including: 1/ intro, 2/ motivation, 3/ mathematical model, 4/ numerical approach, 5/ results, 6/ discussion (personal thoughts on the topic) and conclusion.

- Hand in a zip file including the codes used to generate the results.

## Overall course conclusion

- Elastic waves and viscous Stokes flow
- Shared memory parallelisation: C CUDA and GPUs
- Stencil codes
- Convergence acceleration using second order scheme (damping)
- Distributed memory parallelisation: MPI + CUDA for multi-GPU configurations

## Suggested references

- Performance of stencil codes + MPI

  https://on-demand.gputechconf.com/gtc/2019/video/_/S9368/

- Iterative method for solving large 3D problems on GPUs

  http://www.nature.com/articles/s41598-018-29485-5

  https://doi.org/10.1093/gji/ggz239

  https://doi.org/10.1093/gji/ggy434

● ● ● wp.unil.ch/geocomputing/

● ● ● lraess@stanford.edu

SIGMA