



Sandia  
National  
Laboratories

# Introduction to MPI- Distributed Computing with GPUs



**ISC** High Performance  
The HPC Event.

Simon Garcia de Gonzalo, Sandia National Laboratories

May 21<sup>st</sup> 2023

Hamburg Germany



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Multi-GPU Computing: What you will learn

- CUDA-aware MPI
- Example: Jacobi-Solver
- Under the hood (why you should use CUDA-aware MPI)
  - GPUs in Clusters
  - CUDA Unified Virtual Addressing
  - GPUDirect P2P and GPUDirect RDMA



# Message Passing Interface - MPI

- Standard to exchange data between processes via messages
  - Defines API to exchange messages
    - Point to Point: e.g. `MPI_Send`, `MPI_Recv`
    - Collectives: e.g. `MPI_Reduce`, `MPI_Allreduce`, `MPI_Bcast`
- Multiple implementations (open source and commercial)
  - Bindings for C/C++, Fortran, Python, ...
  - e.g. MPICH, OpenMPI, MVAPICH, IBM Spectrum MPI, Cray MPT, ParaStation MPI, ...

# Example MPI Program

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Finalize();
}
```

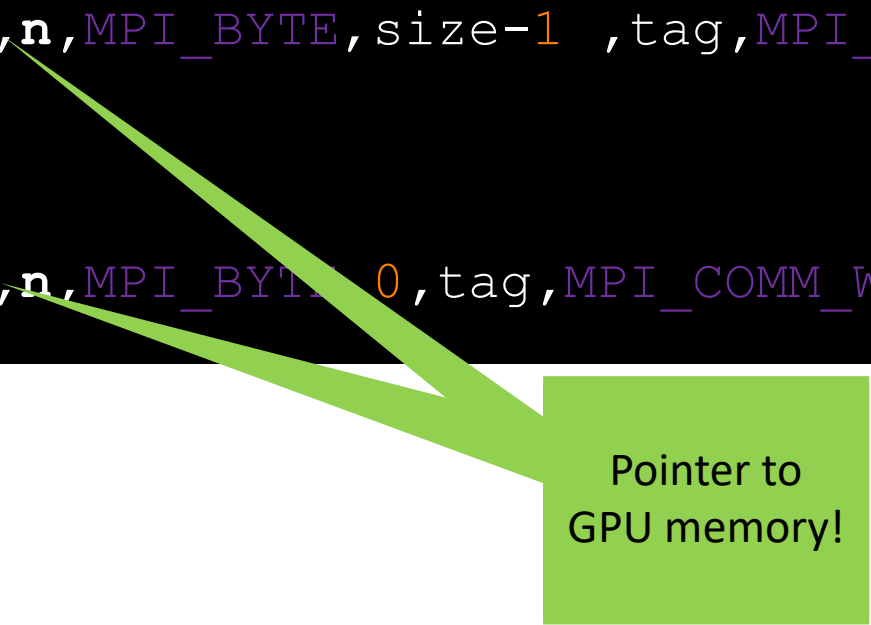
```
mpicc -o hello_mpi.out hello_mpi.c
mpirun -n 4 ./hello_mpi.out
```

# CUDA-aware MPI

CUDA-aware MPI allows you to use Pointers to GPU-Memory as source and destination

```
//MPI rank 0
MPI_Send(s_buf_d,n,MPI_BYTE,size-1 ,tag,MPI_COMM_WORLD) ;

//MPI size-1
MPI_Recv(r_buf_d,n,MPI_BYTE,0,tag,MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;
```



Pointer to  
GPU memory!

# LAUNCH MPI+CUDA

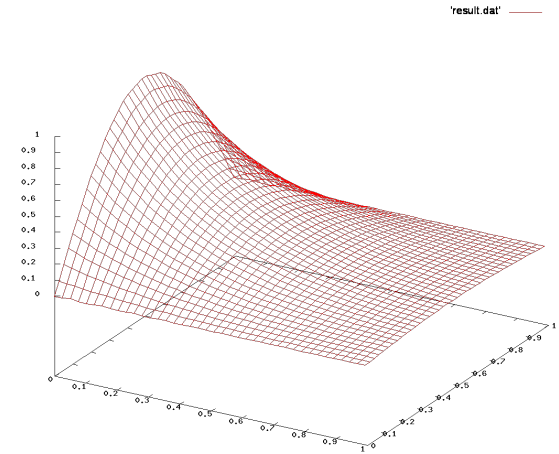
- Launch one process per GPU
- How to use CUDA-aware MPI
  - *MVAPICH*: `$MV2_USE_CUDA=1 mpirun -np ${np} ./myapp <args>`
  - *Open MPI*: *CUDA-aware features are enabled per default (using UCX)*
  - *Cray*: `MPICH_RDMA_ENABLED_CUDA`
  - *IBM Spectrum MPI*: `$mpirun -gpu -np ${np} ./myapp <args>`
  - *ParaStation MPI* (using UCX): `$PSP_CUDA=1 mpirun -np ${np} ./myapp <args>`
- On JUWELS Booster:
  - Load CUDA-aware OpenMPI or ParaStation MPI modules
  - GPU-tuning is done via module loading
  - `srun --gres=gpu:4 -n {np} ./myapp <args>`

# How to compile

```
nvcc -o my_kernel.o $(NVCC_FLAGS) my_kernels.cu -c  
mpicc -o my_multiGPUapp -lcudart my_kernel.o my_multiGPUapp.c
```

# Example: Jacobi Solver

- Solves the 2D-Laplace Equation on a rectangle
- $\Delta \mathbf{u}(\mathbf{x}, \mathbf{y}) = \mathbf{0} \ \forall \ (\mathbf{x}, \mathbf{y}) \in \Omega \setminus \delta\Omega$
- Dirichlet boundary conditions on left/right boundaries (constant values on boundaries)
- Reflecting boundaries on top and bottom
- Iterative solver:  
 $\mathbf{u}(\mathbf{t}+1) = \mathbf{f}(\mathbf{u}(\mathbf{t}))$





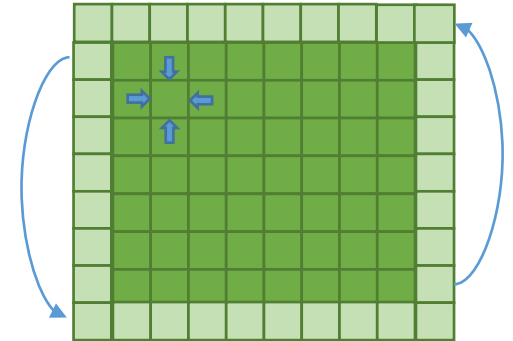
# Example: Jacobi solver

While not converged  
do Jacobi step:

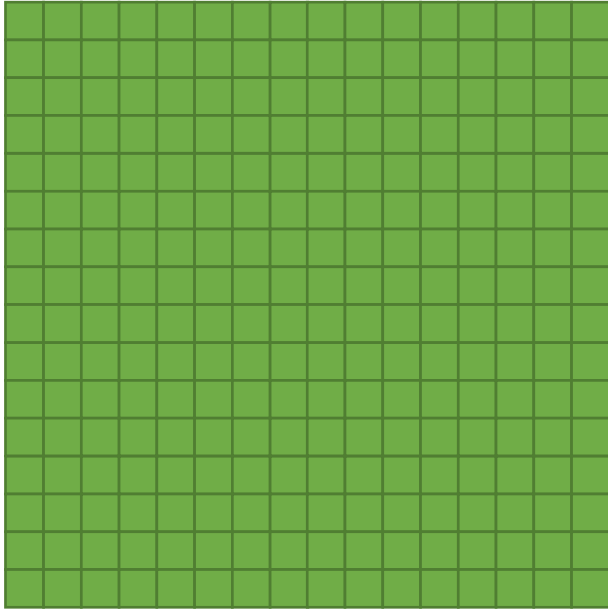
```
int iy = blockIdx.y * blockDim.y + threadIdx.y
int ix = blockIdx.x * blockDim.x + threadIdx.x
if (iy < ny-1 && ix < nx-1) {
    new_val =
        0.25 * (a[iy * nx + ix + 1] + a[iy * nx + ix - 1]
               + a[(iy + 1) * nx + ix] + a[(iy - 1) * nx + ix]);
    a_new[iy * nx + ix] = new_val;
}
```

apply boundary Condition

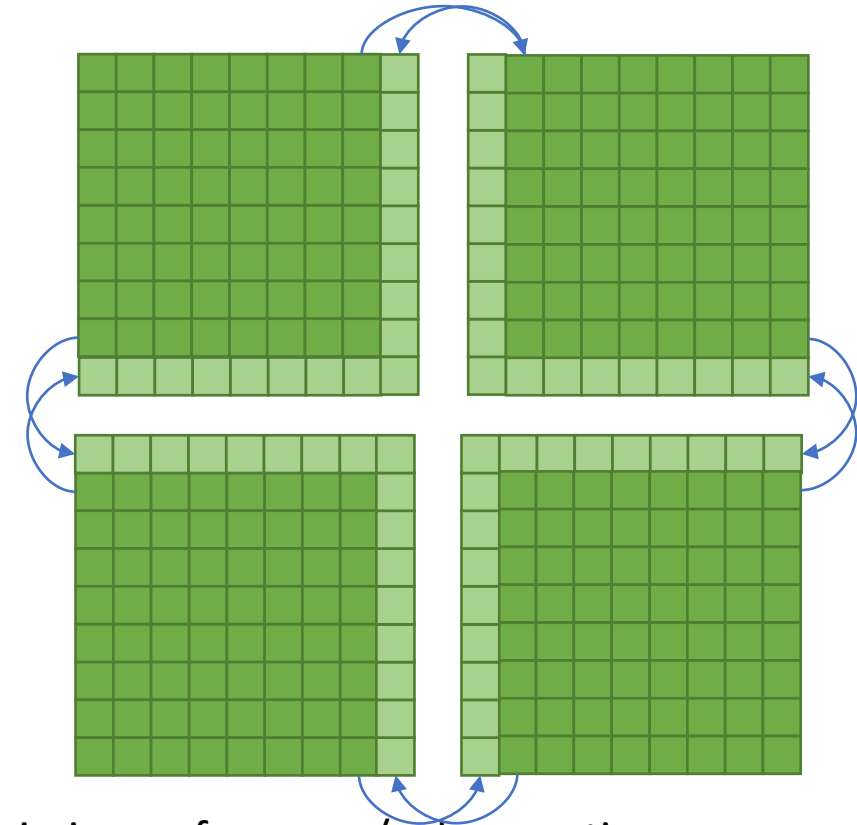
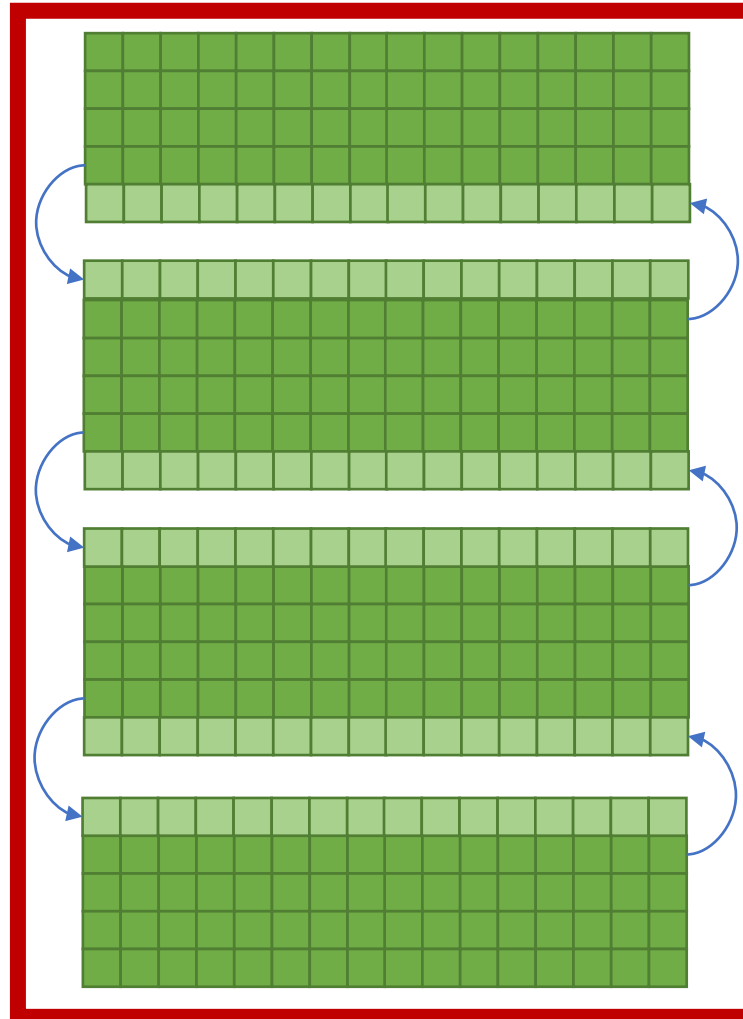
swap a\_new and a  
next iteration



# Domain Decomposition



Minimize number of neighbors:  
Communicate to less neighbors  
Optimal for latency bound communication,  
Continuous Transfers



Minimize surface area/volume ratio:  
Communicate less data  
Optimal for bandwidth bound communication  
Non-Continuous Transfers

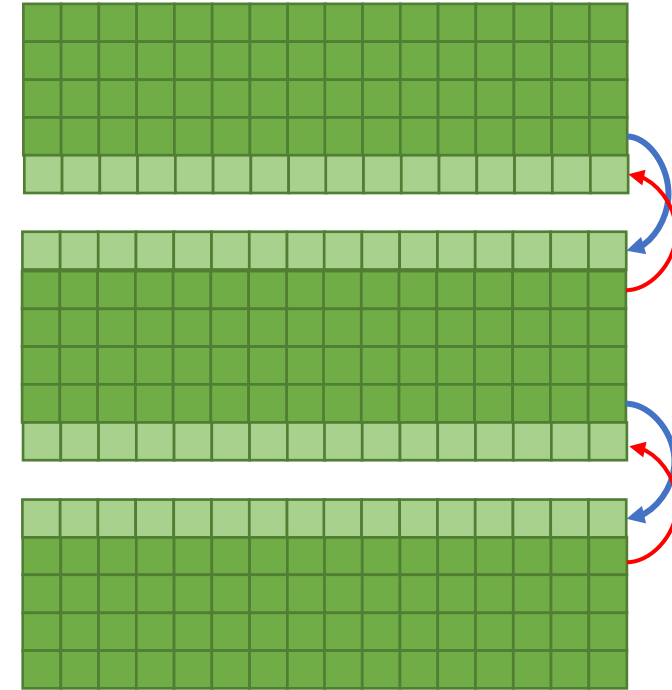
# Jacobi example: Top and Bottom Boundaries

```
MPI_Sendrecv(a_new_d+offset_last_row, m-2, MPI_DOUBLE,  
b_nb, 1, a_new_d+offset_top_boundary, m-2,  
MPI_DOUBLE, t_nb, 1, MPI_COMM_WORLD,  
MPI_STATUS_IGNORE);
```

bottom  
neighbor

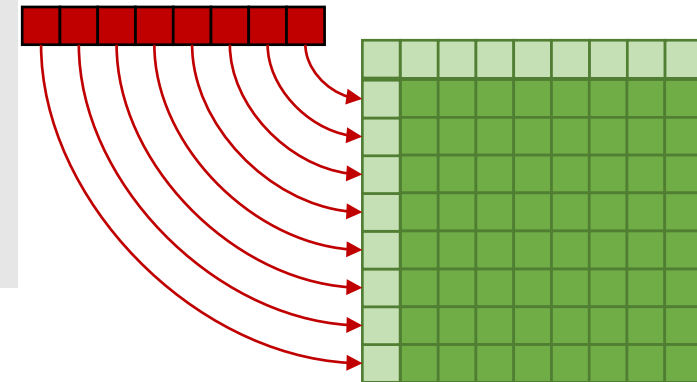
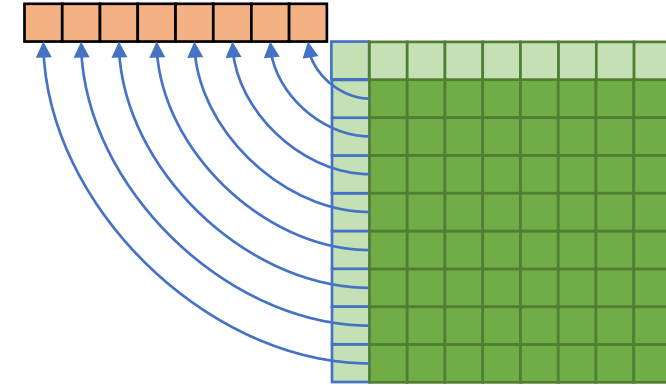
top  
neighbor

```
MPI_Sendrecv(a_new_d+offset_first_row, m-2, MPI_DOUBLE,  
t_nb, 0, a_new_d+offset_bottom_boundary, m-2,  
MPI_DOUBLE, b_nb, 0, MPI_COMM_WORLD,  
MPI_STATUS_IGNORE);
```



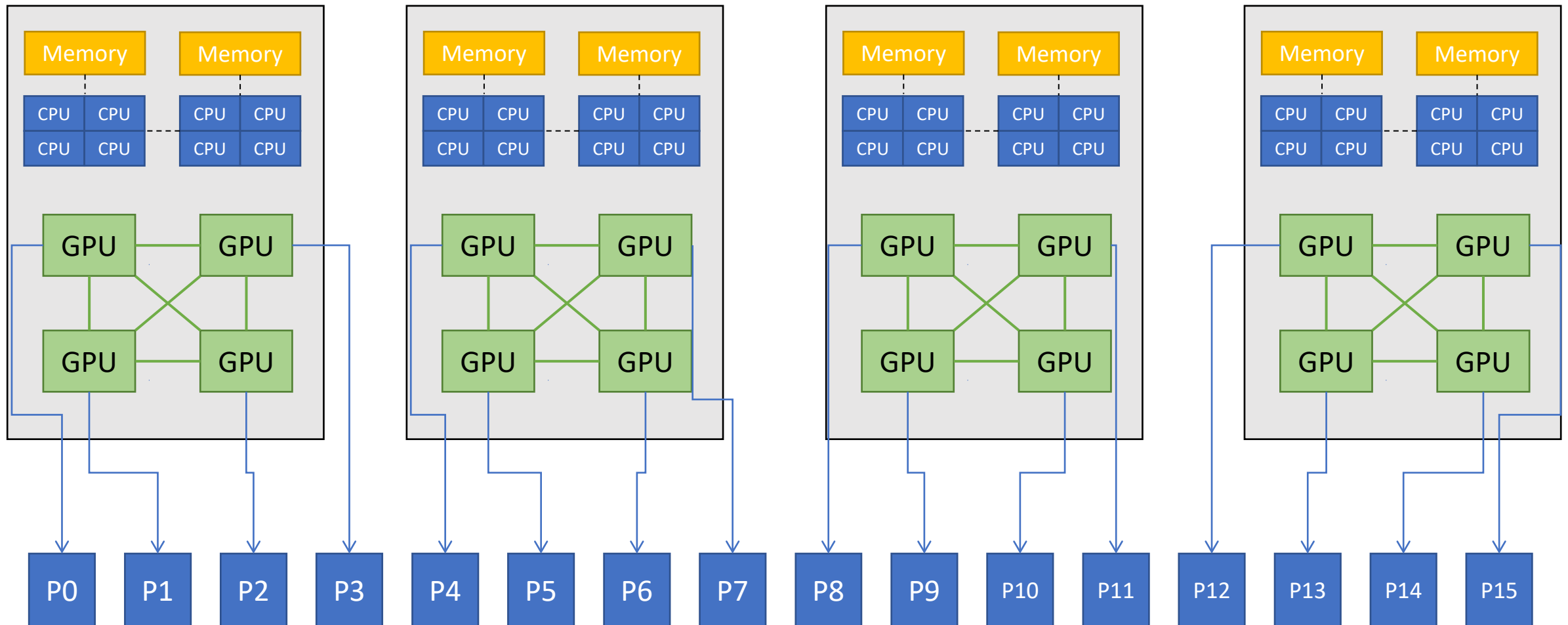
# Jacobi: Left (and right) neighbours

```
//right neighbor omitted
pack<<<gs,bs,0,s>>>(to_left_d, u_new_d, n, m);
cudaStreamSynchronize(s);
MPI_Sendrecv(to_left_d, n-2, MPI_DOUBLE, l_nb, 0,
             from_right_d, n-2, MPI_DOUBLE, r_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE );
unpack<<<gs,bs,0,s>>>(u_new_d, from_right_d, n, m);
cudaStreamSynchronize(s);
```



# Process Mapping on Multi GPU Systems

## One GPU per Process



# Distribute GPUs to local Nodes

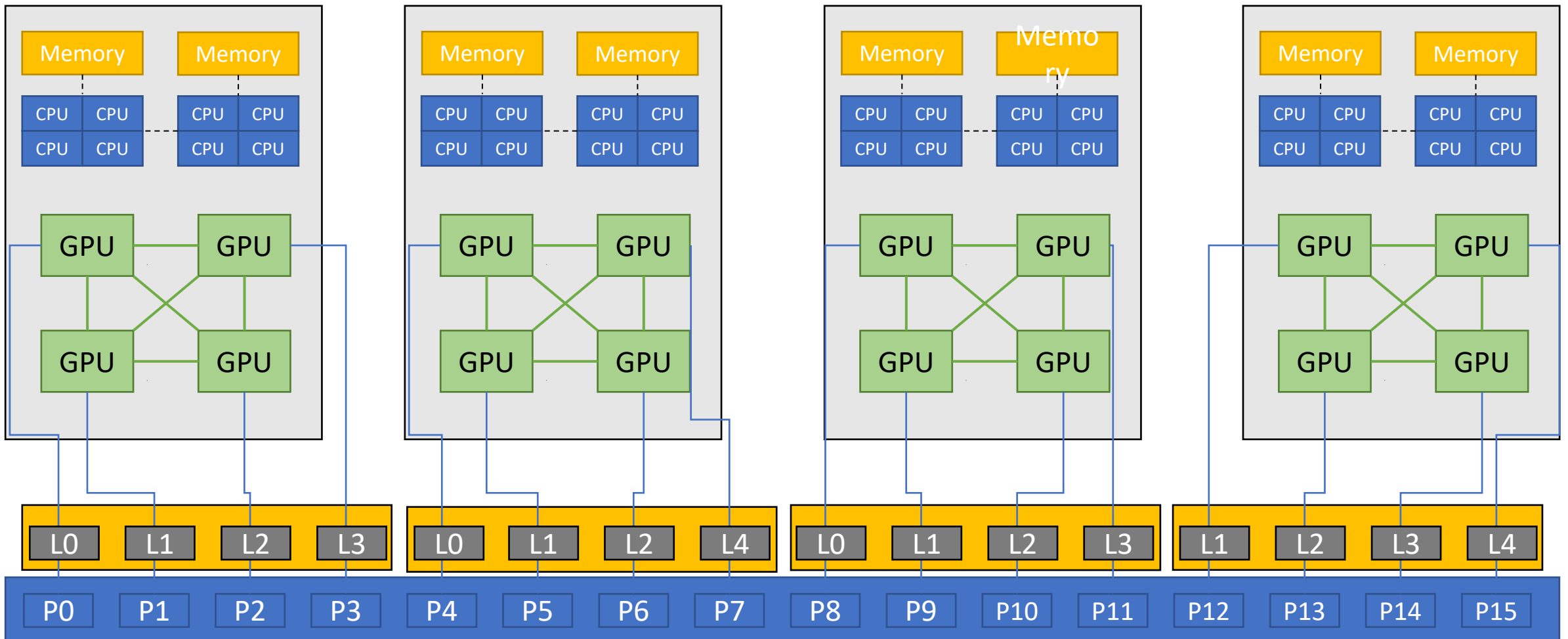
```
MPI_Comm local_comm;  
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, rank,  
                   MPI_INFO_NULL, &local_comm);  
  
int local_rank = -1;  
MPI_Comm_rank(local_comm, &local_rank);  
MPI_Comm_free(&local_comm);
```

```
int num_devs = 0;  
cudaGetDeviceCount(&num_devs);  
cudaSetDevice(local_rank % num_devs);
```

Needed, if  
resource manager  
handles GPU-  
affinity

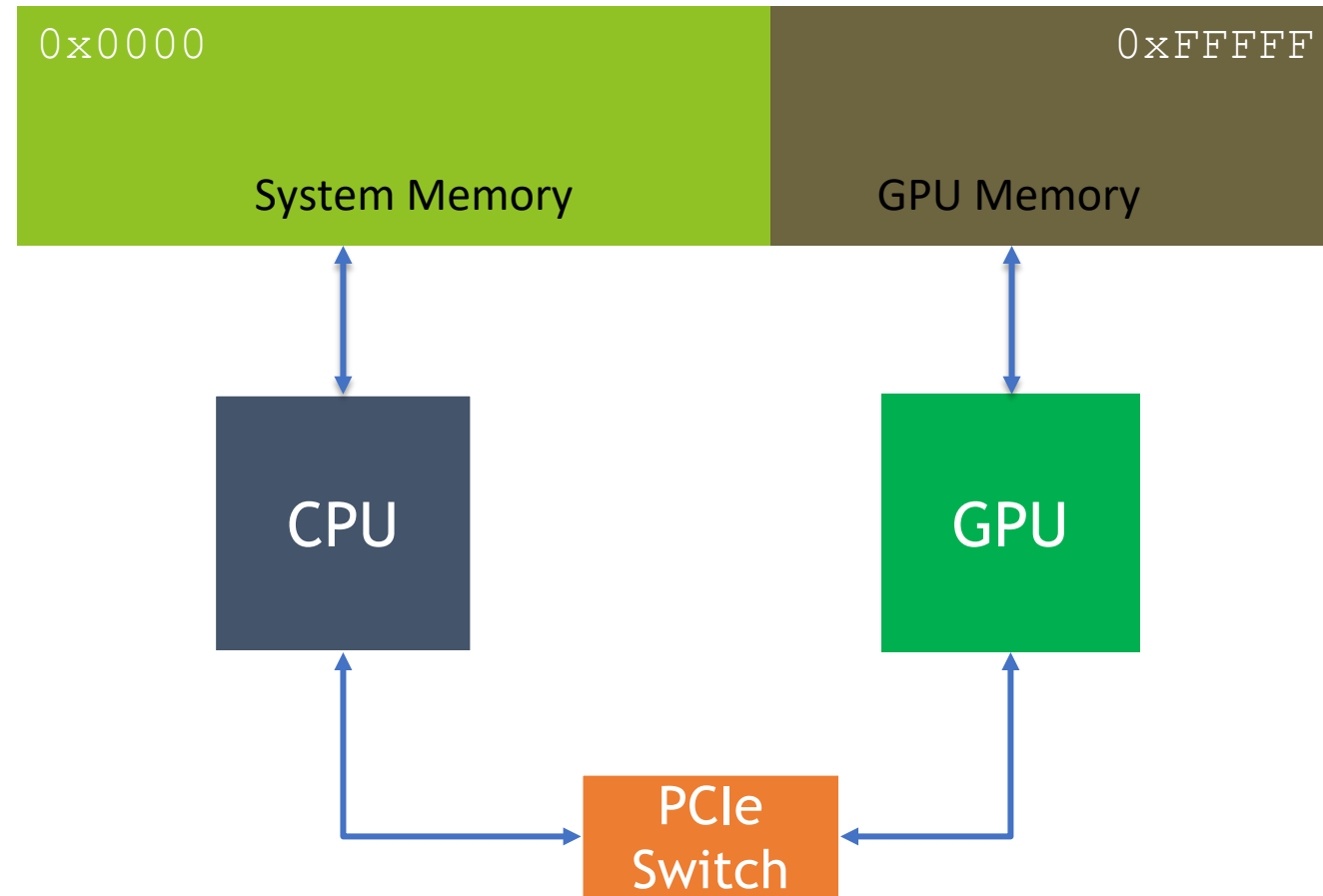
# Process Mapping on Multi GPU Systems

## One GPU per Process



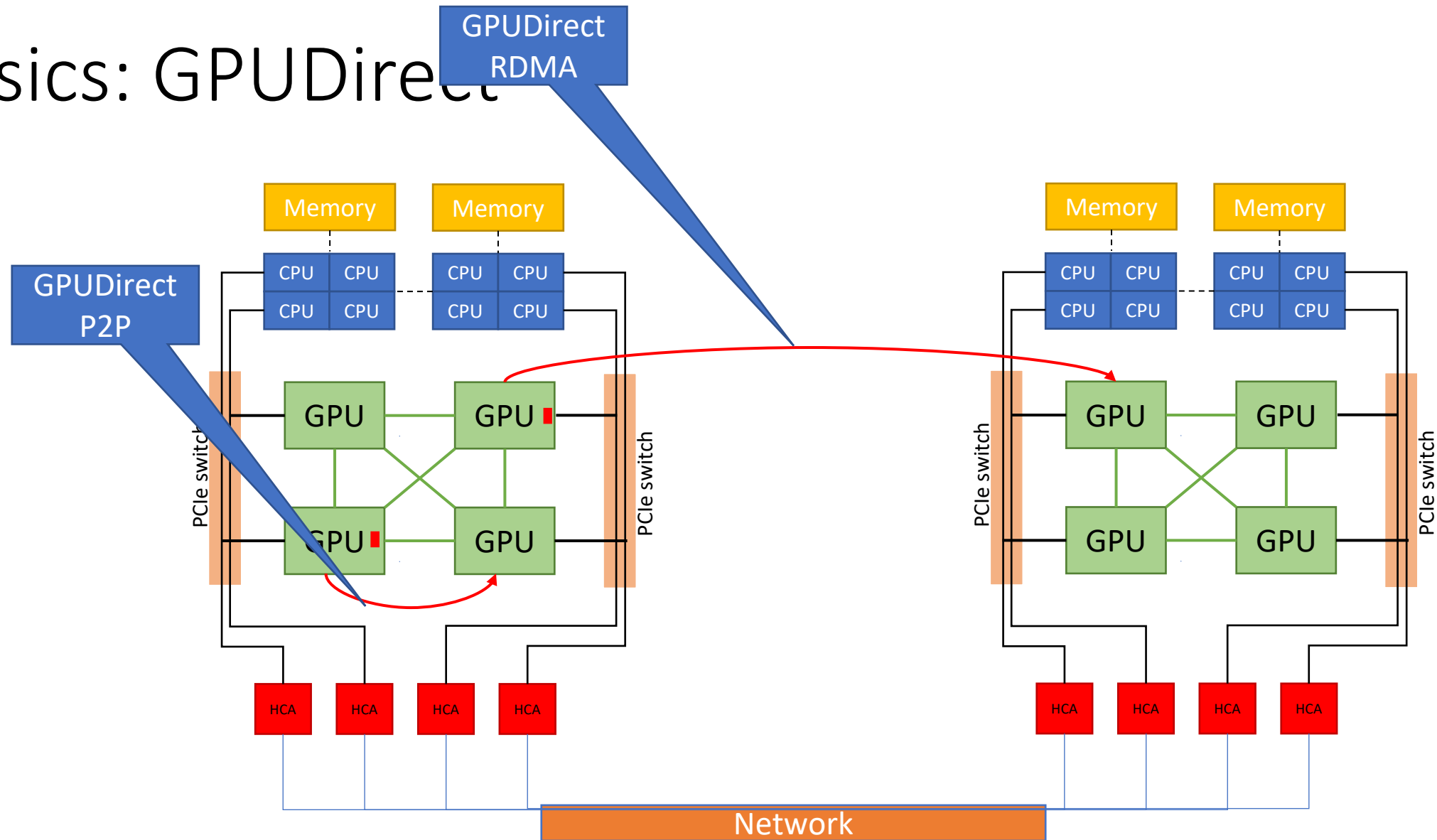
# CUDA Unified Virtual Addressing

- One address space for all CPU and GPU memory
  - Determine physical memory location from a pointer value
  - Enable libraries to simplify their interfaces (e.g. MPI )
- Supported on devices with compute capability 2.0+ for
  - 64-bit applications on Linux and Windows (+TCC)

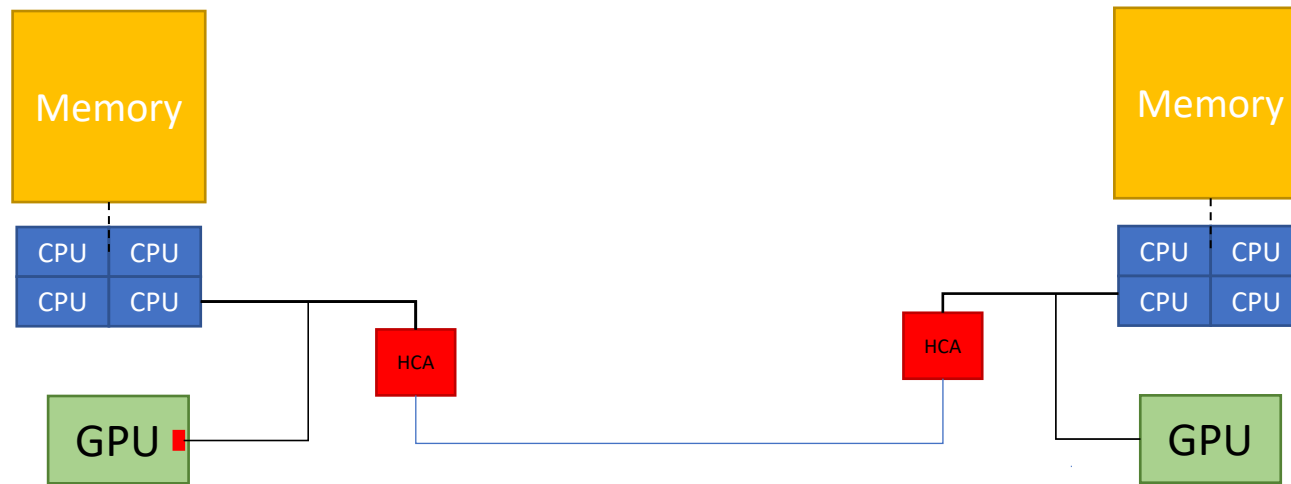




# Basics: GPUDirect

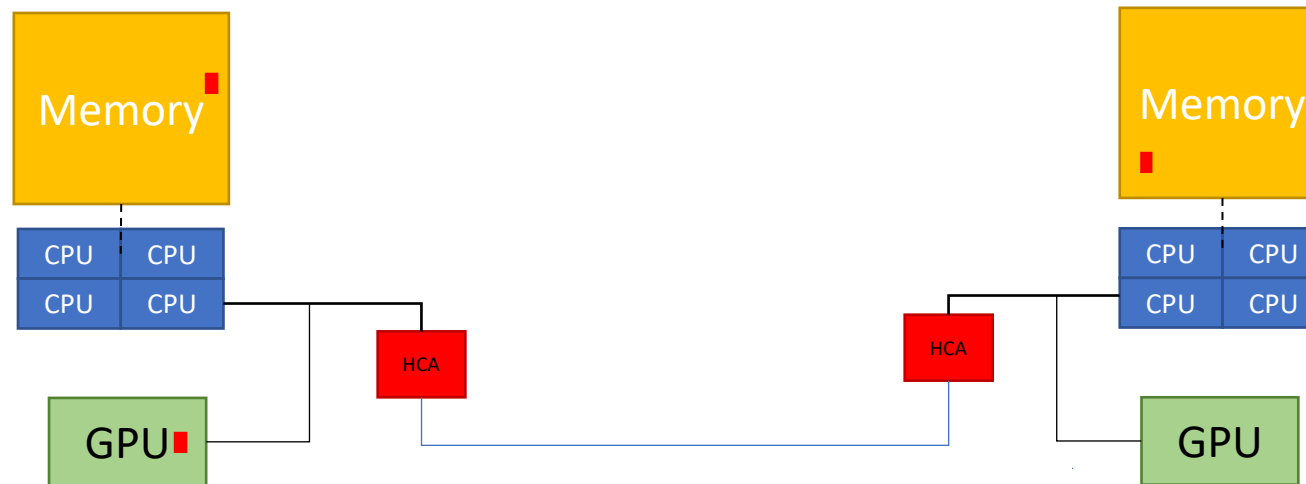


# CUDA-aware MPI with GPUDirect RDMA



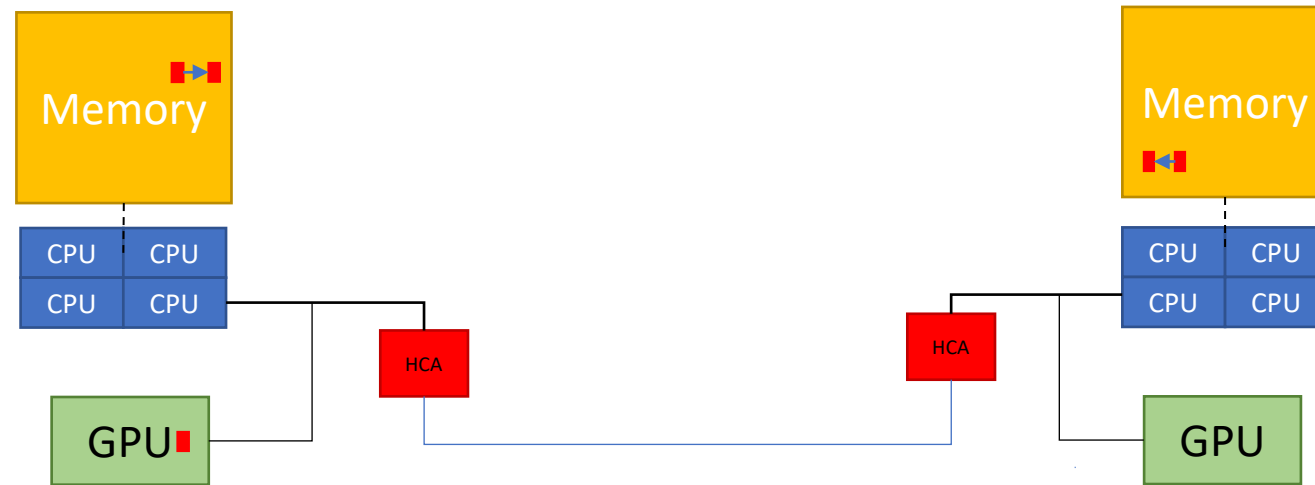
```
MPI_Send(s_buf_d,size,MPI_BYTE,1,tag,MPI_COMM_WORLD);  
MPI_Recv(r_buf_d,size,MPI_BYTE,0,tag,MPI_COMM_WORLD,&stat);
```

# CUDA-aware MPI without GPUDirect RDMA



```
MPI_Send(s_buf_d, size, MPI_BYTE, 1, tag, MPI_COMM_WORLD) ;  
MPI_Recv(r_buf_d, size, MPI_BYTE, 0, tag, MPI_COMM_WORLD, &stat) ;
```

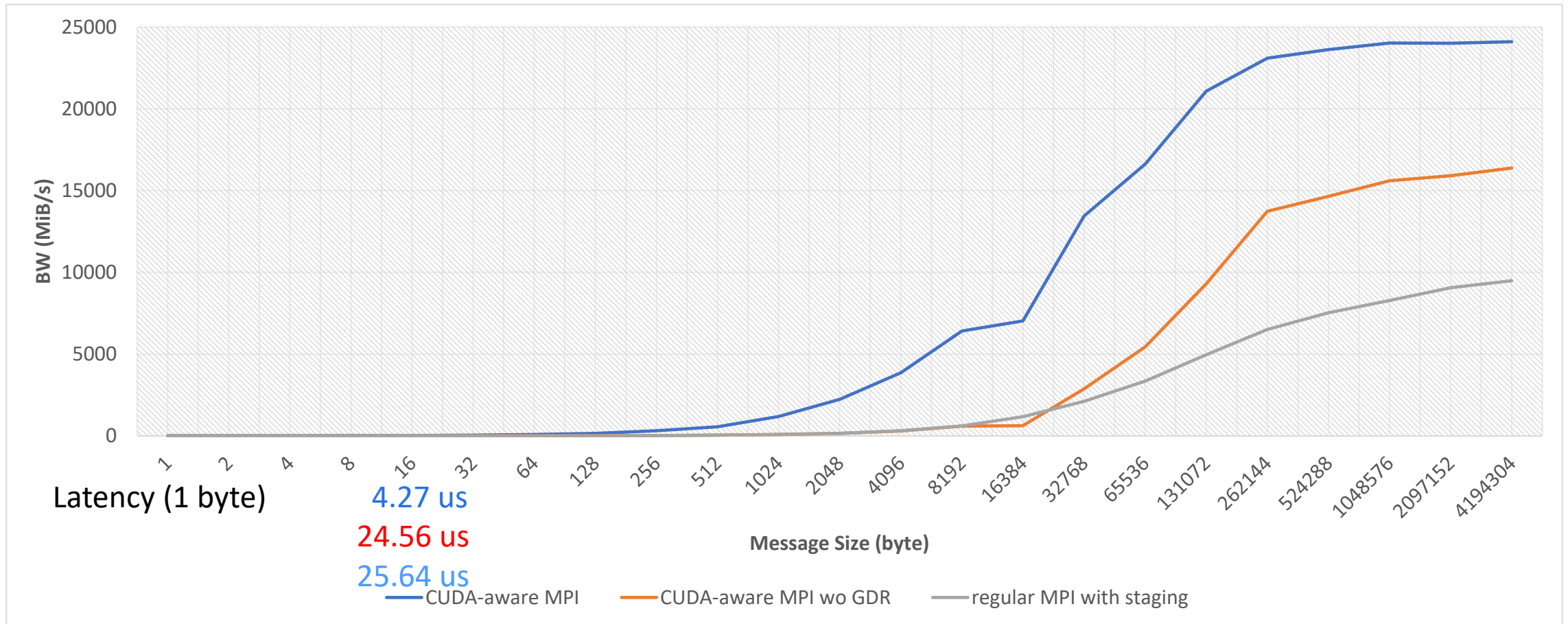
# Regular MPI



```
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);  
MPI_Send(s_buf_h,size,MPI_BYTE,1,tag,MPI_COMM_WORLD);  
  
MPI_Recv(r_buf_h,size,MPI_BYTE,0,tag,MPI_COMM_WORLD,&stat);  
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

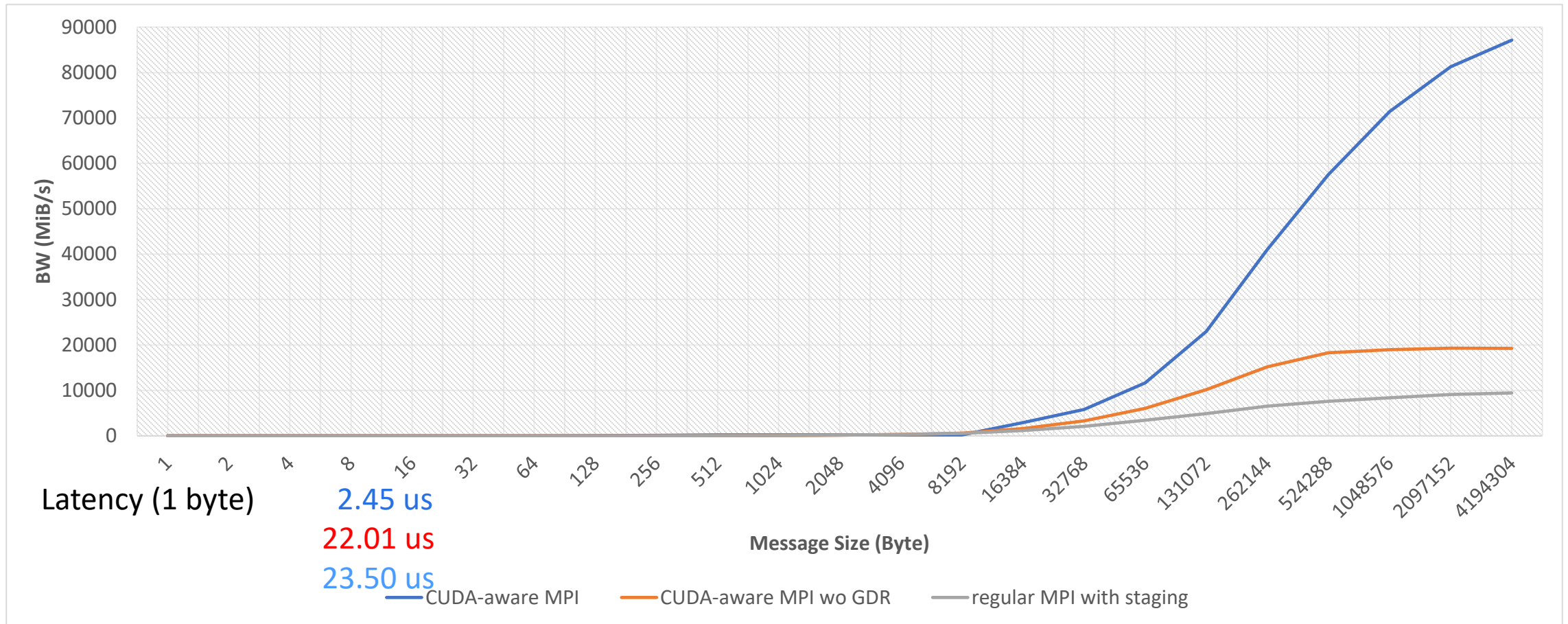
# Performance Results GPUDirect RDMA

OpenMPI 4.1.0RC1 + UCX 1.9.0 on JUWELS Booster



# Performance Results GPUDirect P2P

OpenMPI 4.1.0RC1 + UCX 1.9.0 on JUWELS-Booster



# Summary

- CUDA-aware MPI allows efficient communication for multi-GPU applications
  - Allows MPI-communication operations from GPU memory buffers
  - Simplified programming
  - Use GPUDirect technologies for performance
  - Minimizes data copies
- Most MPI versions have support for CUDA-aware MPI
- Best practice: One process per GPU
  - Use local communicator in MPI