POLITECNICO DI MILANO

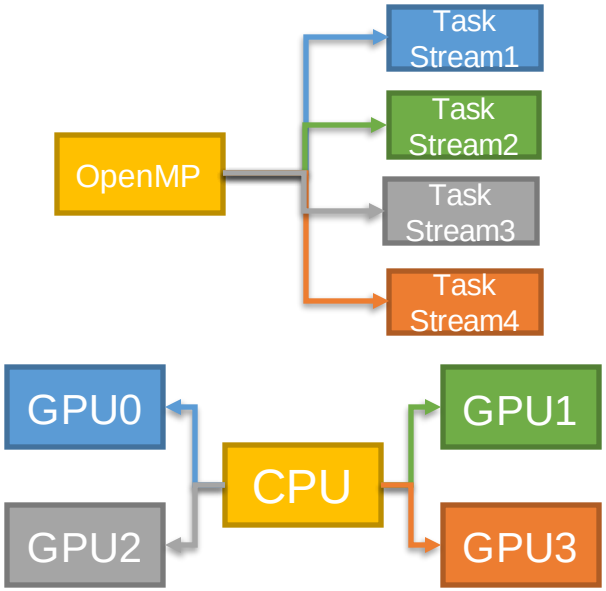
CUDA and OpenMP
CUDA and MPI

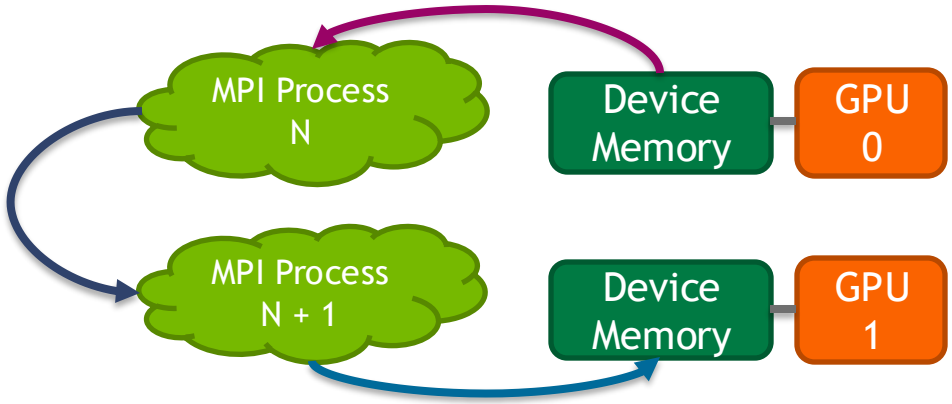
Parallel Computing

Serena Curzel
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
serena.curzel@polimi.it

Motivations

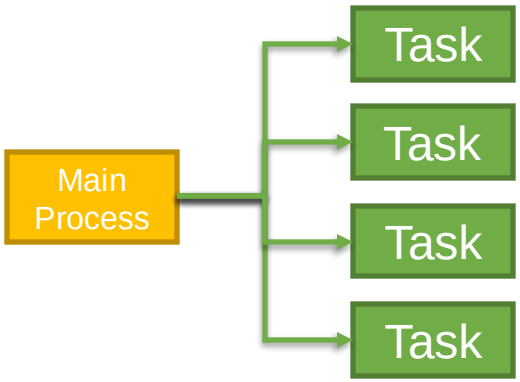
- Systems with multiple GPUs
  - Use OpenMP to dispatch parallel tasks
- Distributed clusters with multiple nodes that contain one or more GPUs
  - Use MPI for communication across nodes

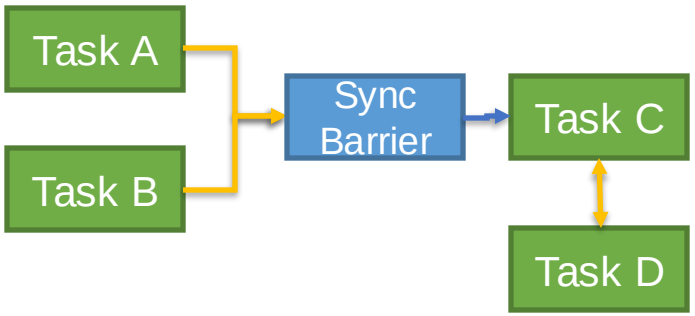




Computation patterns in a Multi-GPU environment

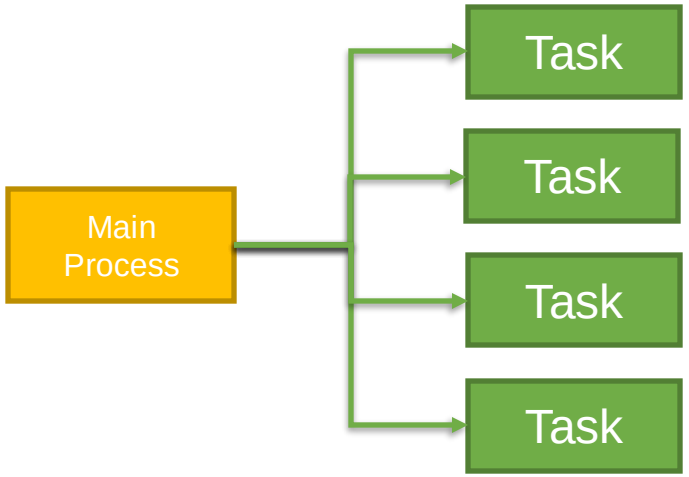
- Batch processing
  - Execute the same independent task multiple times with different data
- Cooperative patterns
  - Tasks need to cooperate between each other to collectively reach a goal





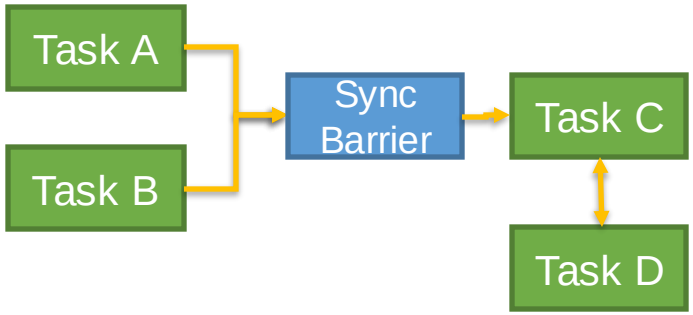
Batch processing

- Typical operations to accomplish batch processing:
  - Get number of available devices
  - Initialize and copy the memory needed by the algorithm
  - Create CUDA streams for each of the concurrent tasks
  - Launch the kernel in parallel



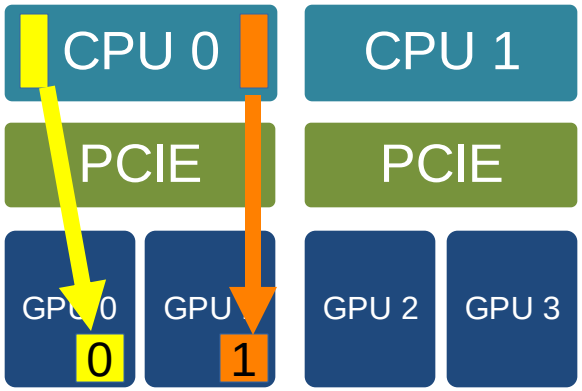
Cooperative patterns

- No single fit solution
- The process consists in a loop of:
  - Launching the code in parallel
  - Profiling it
  - Analyzing and removing bottlenecks



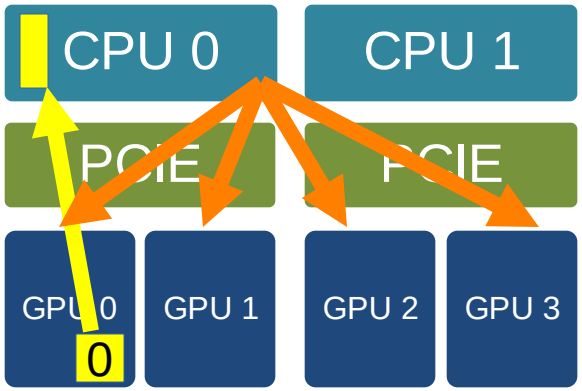
CUDA API calls for multi-GPUs

- `cudaSetDevice()`
  - Set GPU device to use for device code execution on the active host thread
  - Requires one parameter, an integer with the device ID number
  - Doesn't affect other host threads, so setting the device on one thread will not set the device in other host threads and won't affect previous async calls



CUDA API calls for multi-GPUs

- `cudaGetDevice()`
  - Get GPU device currently used by the active host thread
  - Requires one parameter, a pointer to store the device ID number
- `cudaGetDeviceCount()`
  - Get the number of CUDA-capable devices in the system
  - Requires one parameter, a pointer to store the number of devices



CUDA API calls for multi-GPUs

- CUDA runtime calls are affected by `cudaSetDevice()`
- If `cudaSetDevice()` is called before a kernel launch call, the kernel will execute on the active device
  - All memory used by the kernel needs to be available on the device!
- If it is called before `cudaStreamCreate()`, the stream will be associated with the active device
- All synchronization functions are affected, they will only synchronize tasks on the active device and active host thread

Memory management with multiple GPUs

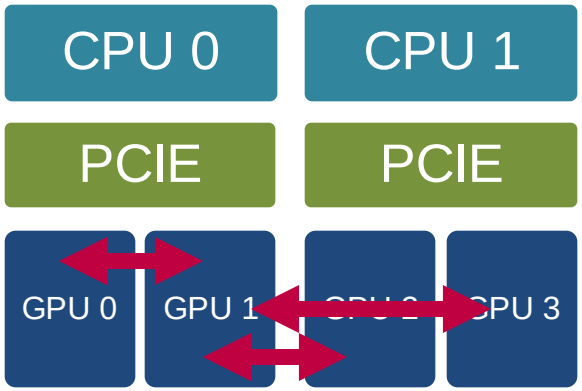
- With manual memory management
  - To allocate or associate memory with a specific device, call `cudaSetDevice()` before doing the allocation call (e.g. `cudaMalloc()`)
- With Unified Memory
  - Situation 1: the flag `cudaDevAttrConcurrentManagedAccess` is set in all devices, no need to set the active device before calling `cudaMallocManaged()`
  - Situation 2: the flag is not set but devices can access each other's memory, setting the device will establish that all other devices access data through PCIE

Vector addition with multiple GPUs


```
float *m_A0, float *m_B0, *m_A1, float *m_B1, int n;
int size = n * sizeof(float);

cudaSetDevice(0);                       // Will set the active device to 0
cudaMalloc((void**) &m_A0, size);       // Will allocate memory on device 0
cudaMalloc((void**) &m_B0, size);       // Will allocate memory on device 0

cudaSetDevice(1);                       // Will set the active device to 1
cudaMalloc((void**) &m_A1, size);       // Will allocate memory on device 1
cudaMalloc((void**) &m_B1, size);       // Will allocate memory on device 1

// Memory initialization on the Host and memory transfers

cudaSetDevice(0);                     // Set the device for kernel execution
vecAdd<<<gridDim, blockDim>>>(m_A0,m_B0);

cudaSetDevice(1);                   // Set the device for kernel execution
vecAdd<<<gridDim, blockDim>>>(m_A1,m_B1);

cudaFree(m_A0); cudaFree(m_B0);
cudaFree(m_A1); cudaFree(m_B1);
```

Memory transfers between GPUs

- Three ways to transfer memory regions from one device to another
  - Fully explicit, specifying source and destination device
  - Partially explicit through unified addressing
  - Implicit, performed by the driver
- Not all three possibilities are available in every system



Explicit memory transfer between peers

- `cudaMemcpyPeerAsync()`
  - Asynchronous transfer between two devices
  - Requires the IDs of the two devices, pointers to source and destination regions, size, stream
  - Example:

```
float *A0, *A1;
int size;

cudaSetDevice(0);
cudaMalloc((void**) &A0, size);

cudaSetDevice(1);
cudaMalloc((void**) &A1, size);

cudaMemcpyPeerAsync(A1, 1, A0, 0, size, stream);

cudaSetDevice(1);
kernel<<<gridDim, blockDim, 0, stream>>>(A1);

cudaFree(A0);
cudaFree(A1);
```

Memory transfer through unified addressing

- If the flag `cudaDevAttrUnifiedAddressing` is set, you may copy regions between devices using `cudaMemcpy()`, setting the direction to `cudaMemcpyDefault`
  - Check if the flag is set with `cudaDeviceGetAttribute()`
  - Example:

```
cudaDeviceGetAttribute(unifiedAddr_flag0, cudaDevAttrUnifiedAddressing, 0);
cudaDeviceGetAttribute(unifiedAddr_flag1, cudaDevAttrUnifiedAddressing, 1);

if( unified_addressing_flag0 == 1 && unified_addressing_flag1 == 1 )
 cudaMemcpy(A1, A0, size, cudaMemcpyDefault);
else
 // Throw error indicating the copy couldn't be performed
```

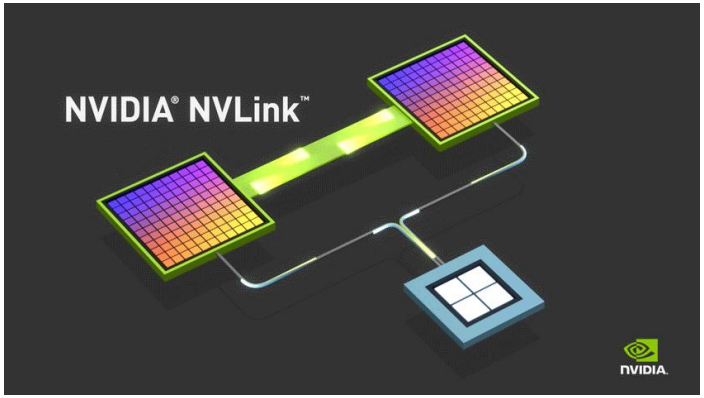
Implicit peer memory access

- With implicit peer access, kernels can directly take the data they need from a different device
  - `cudaDeviceCanAccessPeer()` to query whether it is supported (not symmetric!)
  - `cudaDeviceEnablePeerAccess()` and `cudaDeviceDisablePeerAccess()` to toggle
  - Example:

```
cudaDeviceCanAccessPeer(&BcanAccessA, devB, devA);

cudaSetDevice(devB);
if(BcanAccessA == 0)
  error = cudaDeviceEnablePeerAccess(devA, 0);

if(error == cudaSuccess) {
  kernel<<<gridDim, blockDim, 0, stream>>>(ptrA);
}

cudaDeviceDisablePeerAccess(devA);
```

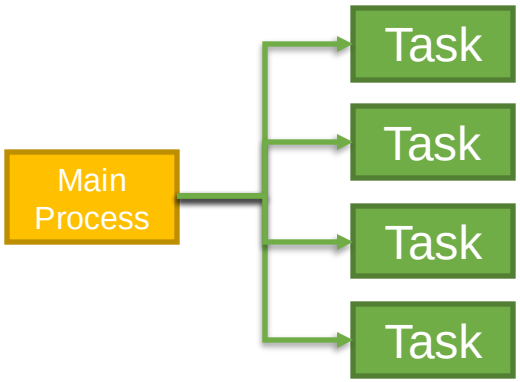
Communication channels

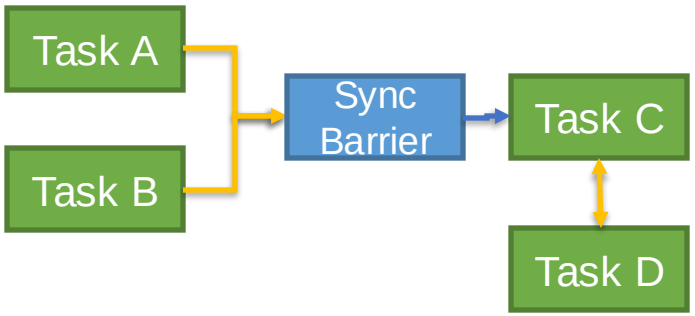
- There are two main types of communication channels connecting GPUs together:
  - Standard PCIE 3.0 link, 32 GB/s
  - NVLINK, faster proprietary interconnect technology developed by NVIDIA (300 GB/s on the Tesla V100)



Computation patterns in a Multi-GPU environment

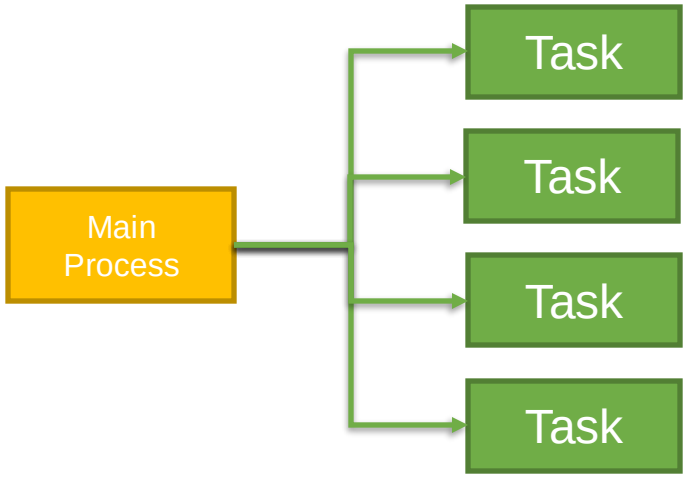
- Batch processing
  - Execute the same independent task multiple times with different data
- Cooperative patterns
  - Tasks need to cooperate between each other to collectively reach a goal





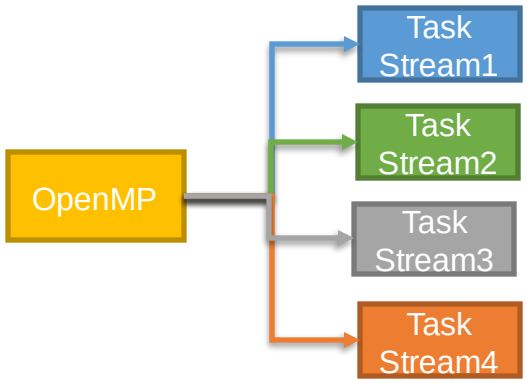
Batch processing

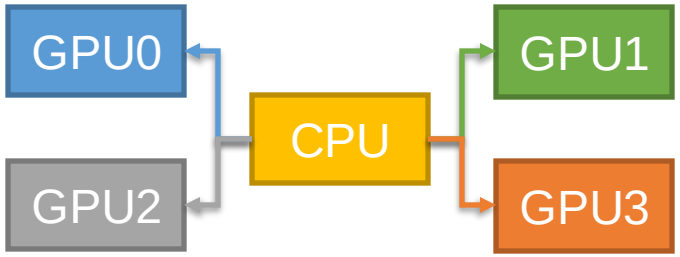
- Typical operations to accomplish batch processing:
  - Get number of available devices
  - Initialize and copy the memory needed by the algorithm
  - Create CUDA streams for each of the concurrent tasks
  - Launch the kernel in parallel



Batch processing with OpenMP

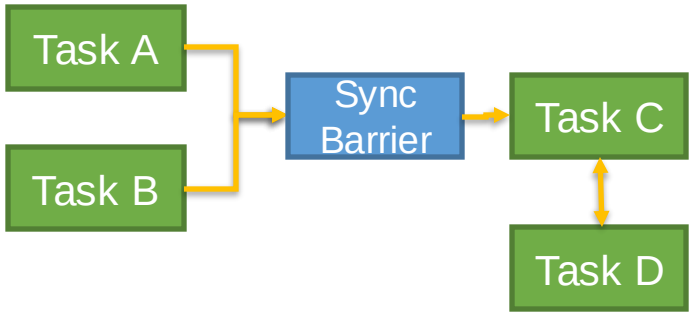

```
int deviceCount;
cudaGetDeviceCount(& deviceCount);
std::vector<cudaStream_t> streams(deviceCount);

#pragma omp parallel for num_threads(deviceCount)
for(int dev = 0; dev < deviceCount; ++dev) {
    cudaSetDevice(dev);
    cudaStreamCreate(&streams[i]);
    // Allocate, initialize and transfer memory
}

#pragma omp parallel for num_threads(deviceCount)
for(int dev = 0; dev < deviceCount; ++dev) {
    cudaSetDevice(dev);
    kernel<<<gridDim, blockDim, streams[i]>>>(…);
}
```





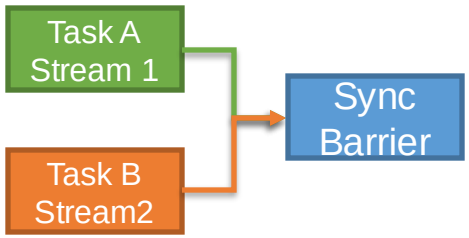
Cooperative patterns

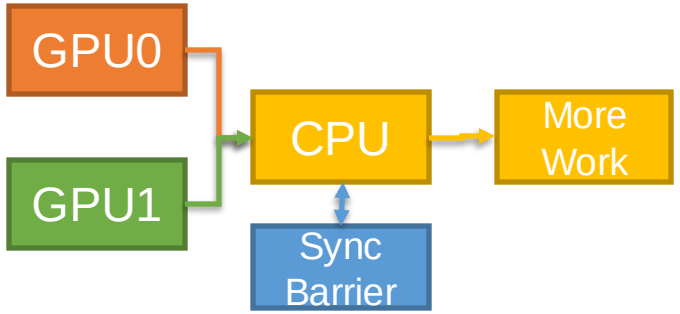
- No single fit solution
- The process consists in a loop of:
  - Launching the code in parallel
  - Profiling it
  - Analyzing and removing bottlenecks



Cooperative patterns with OpenMP

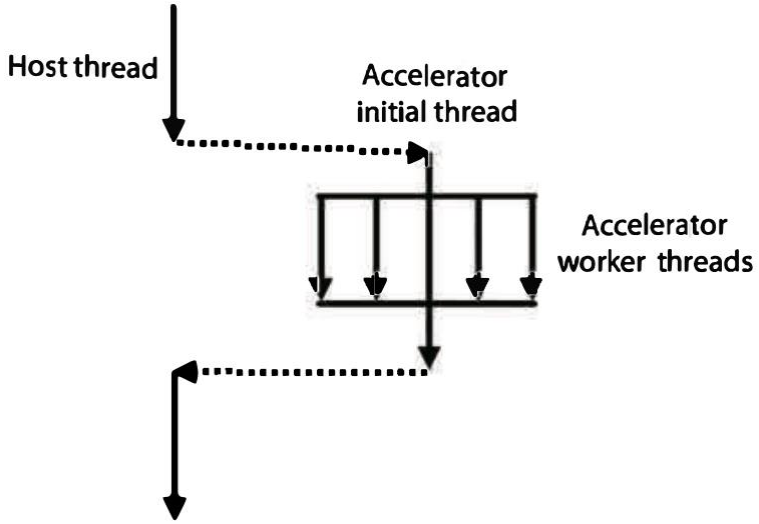

```
std::vector<cudaStream_t> streams;
// Initialization of the streams on each device.

#pragma omp parallel
{
  // Launch the different kernels on the streams.

#pragma omp for num_threads(streams.size())
  for(auto& stream : streams)
      cudaStreamSynchronize(stream);

#pragma omp barrier
}
```





OpenMP for heterogeneous architectures

- Recent versions of the OpenMP standard support parallel execution on heterogeneous architectures
  - host CPU
  - one or more attached accelerators (GPUs, FPGAs, DSPs, …)
- The same code can be run on the host and on the target device



OpenMP for heterogeneous architectures


```
#pragma omp target
        /* code region */
```

- Offloads the execution of code to an accelerator device (if present)
  - If not present, continues execution on the host
  - Same when an `if` clause evaluates to false
- A thread on the target device executes the code
- Synchronous execution: the host thread blocks until the device thread has finished
  - Use the `nowait` clause to avoid

OpenMP for heterogeneous architectures


```
#pragma omp target map(a,b,c,d)
        {
          #pragma parallel for
          for (i=0; i<N; i++) {
            a[i] =  b[i] * c + d;
          }
        } // End of target
```

- The `map` clause copies variables that are needed by the target region in the target device memory
  - A mapped variable is a shared variable
  - Synchronization/consistency issues apply
- The initial device thread encounters a parallel region and spawns a team of device threads

OpenMP for heterogeneous architectures

- If mapping requires copy operations across distributed memories, which are expensive!

```
#pragma omp target map(a,b,c,d)
        {
          #pragma parallel for
          for (i=0; i<N; i++) {
            a[i] =  b[i] * c + d;
          }
        } // End of target
```

map-enter: copy a,b,c,d to the device memory

compute: use device memory variables

map-exit: copy a,b,c,d to the host memory

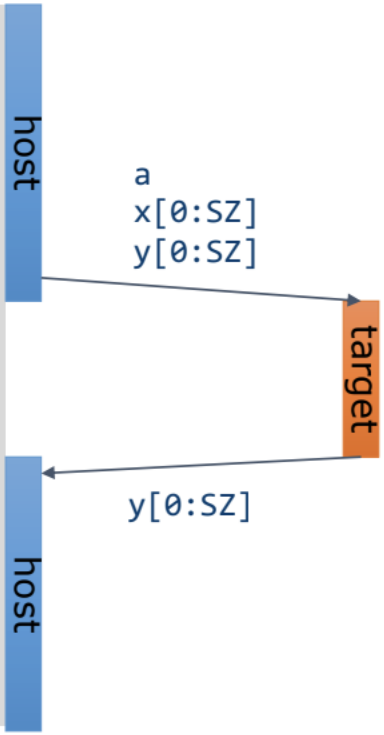
- `map-type` gives information to the compiler to avoid unnecessary data transfers

map-type	Perform map-enter copies	Perform map-exit copies
alloc	No	No
to	Yes	No
from	No	Yes
tofrom	Yes	Yes
release	–	No
delete	–	No

- The default value is `tofrom`

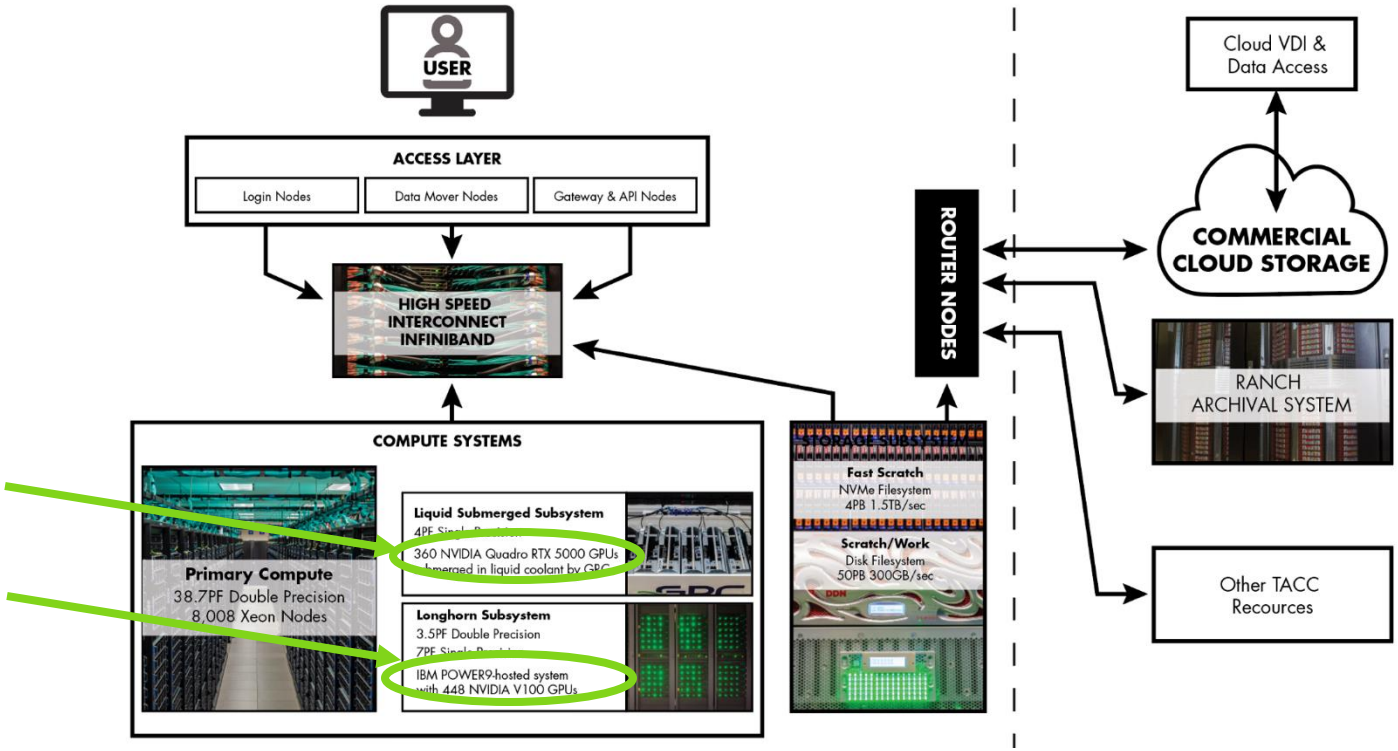
OpenMP for heterogeneous architectures


```
void saxpy() {
    double a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:SZ]) \
                   map(tofrom:y[0:SZ])
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```



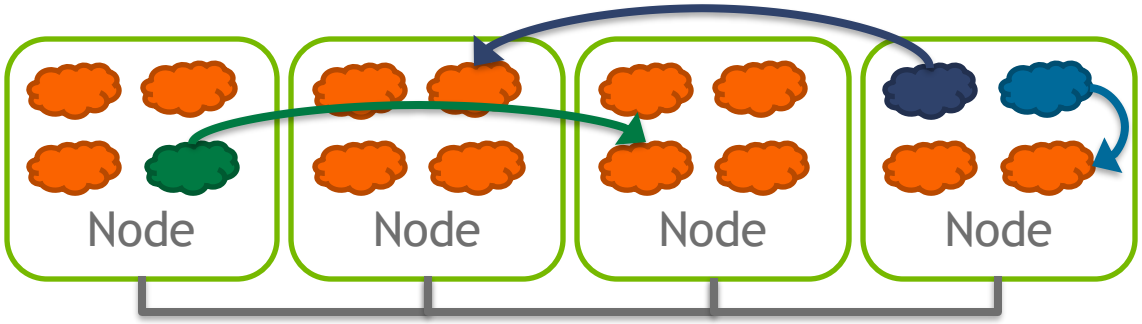
Supercomputers with hetereogeneous nodes

- Distributed clusters with multiple nodes that contain one or more GPUs
  - Example: Frontera (2019)
  - Use MPI for communication across nodes



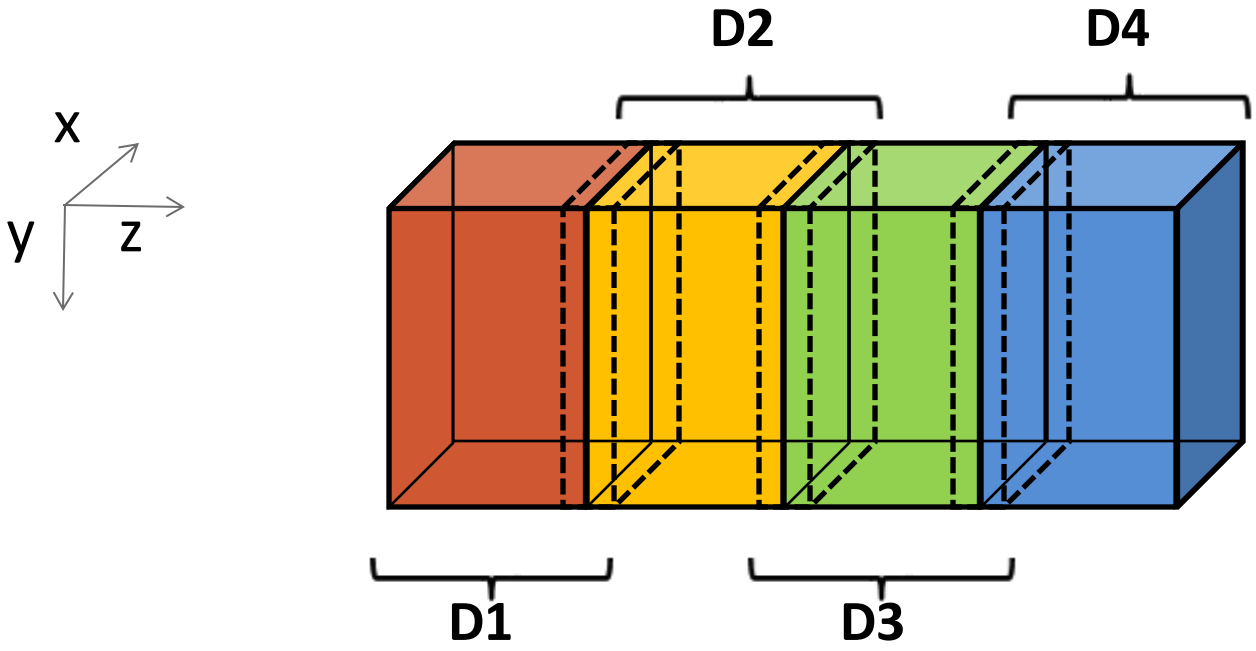
MPI programming model

- Many processes distributed in a cluster
  - Each process computes part of the output
  - Processes communicate with each other
  - Processes can synchronize



MPI-CUDA applications

- Example: wave propagation stencil
  - Calculate heat transfer in a volume
  - At each time step, calculate a weighted sum
- Domain decomposition



MPI-CUDA applications

- Main program: initialize communication, execute process, finalize communication

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &np);

if(np < 3) {
  if(0 == pid) printf("Nedded 3 or more processes.\n");
    MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
}
if(pid < np - 1)
  compute_node(dimx, dimy, dimz / (np - 1), nreps);
else
  data_server(dimx, dimy, dimz, nreps);

MPI_Finalize();
```

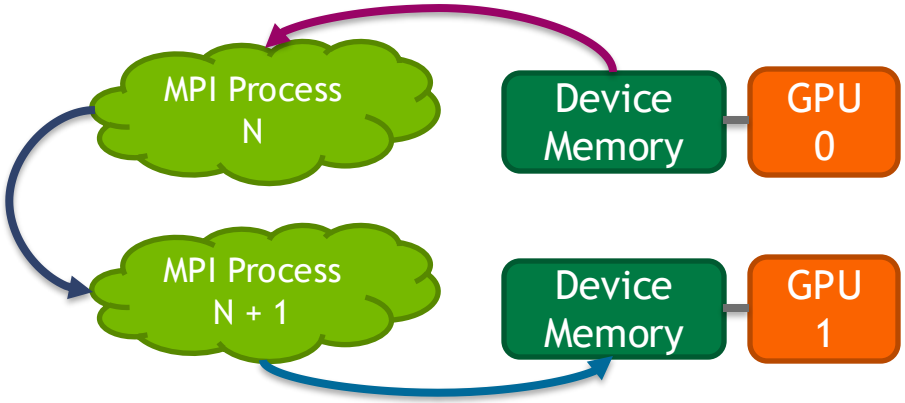
MPI-CUDA applications

- Server process: partition and distribute data

```
/* Find number of MPI processes */
/* Allocate input data */
/* Initialize input data */
/* Calculate number of points per compute process */

/* Send data to compute nodes */
for(int process = 1; process < last_node; process++) {
  MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
    DATA_DISTRIBUTE, MPI_COMM_WORLD);
  send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for(int process = 0; process < num_comp_nodes; process++)
  MPI_Recv(output + process * num_points / num_comp_nodes,
    num_points / num_comp_nodes, MPI_FLOAT, process,
    DATA_COLLECT, MPI_COMM_WORLD, &status );
}
```

MPI-CUDA applications

- Compute process: receive data and offload computation to the GPU (part 1)




```
/* Alloc host memory */
float *h_input  = (float *)malloc(num_bytes);
/* Alloc device memory for input and output data */
float *d_input = NULL;
cudaMalloc((void **)&d_input,  num_bytes );
/* Receive data and move it to device memory */
float *rcv_address = h_input + num_halo_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
                  MPI_ANY_TAG, MPI_COMM_WORLD, &status );
cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);
```

MPI-CUDA applications

- Compute process: receive data and offload computation to the GPU (part 2)

```
void launch_kernel(float *next, float *in, float *prev, float
*velocity, int dimx, int dimy, int dimz)
{
        dim3 Gd, Bd, Vd;

        Vd.x = dimx; Vd.y = dimy; Vd.z = dimz;

        Bd.x = BLOCK_DIM_X; Bd.y = BLOCK_DIM_Y; Bd.z = BLOCK_DIM_Z;

        Gd.x = (dimx + Bd.x - 1) / Bd.x;
        Gd.y = (dimy + Bd.y - 1) / Bd.y;
        Gd.z = (dimz + Bd.z - 1) / Bd.z;

        wave_propagation<<<Gd, Bd>>>(next, in, prev, velocity, Vd);
}
```

MPI-CUDA applications

- Compute process: receive data and offload computation to the GPU (part 3)

```
MPI_Status status;
int left_neighbor  = (pid > 0)     ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

/* Upload stencil cofficients */
/* Calculate offsets */

MPI_Barrier( MPI_COMM_WORLD );
/* Compute boundary values needed by other nodes first */
launch_kernel(d_output + left_stage1_offset, d_input +
    left_stage1_offset, dimx, dimy, 12, stream0);
launch_kernel(d_output + right_stage1_offset,
    d_input + right_stage1_offset, dimx, dimy, 12, stream0);
/* Compute the remaining points */
launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
    dimx, dimy, dimz, stream1);
/* Copy the data needed by other nodes to the host */
cudaMemcpyAsync(h_left_boundary, d_output + num_halo_points,
    num_halo_bytes, cudaMemcpyDeviceToHost, stream0);
cudaMemcpyAsync(h_right_boundary, d_output + right_stage1_offset +
    num_halo_points, num_halo_bytes, cudaMemcpyDeviceToHost, stream0);
cudaStreamSynchronize(stream0);
```

Credits & References

- NVIDIA Deep Learning Institute, “Accelerated Computing” Teaching Kit
- D.B. Kirk and W.W. Hwu, “Programming Massively Parallel Processors. A hands-on approach”, 3rd edition, Morgan Kaufmann 2017