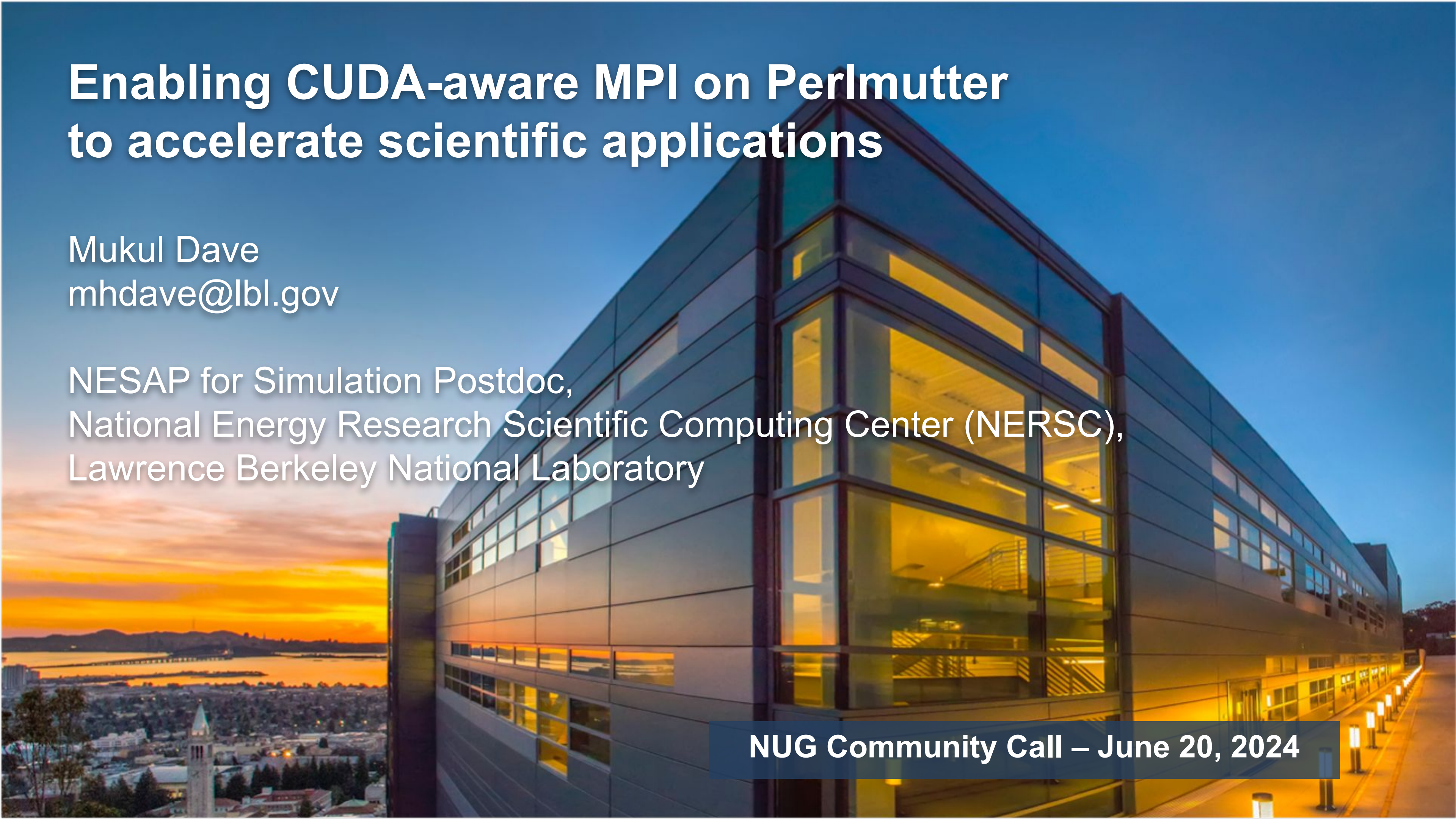


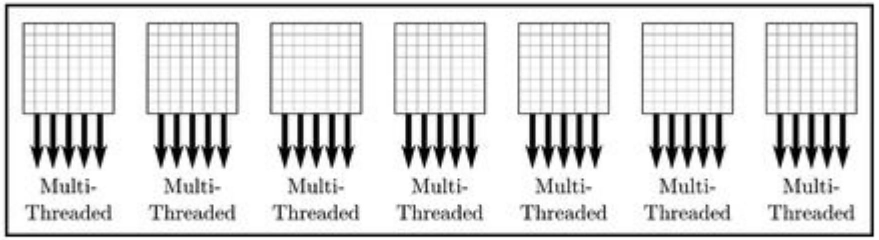
Enabling CUDA-aware MPI on Perlmutter to accelerate scientific applications

Mukul Dave
mhdave@lbl.gov

NESAP for Simulation Postdoc,
National Energy Research Scientific Computing Center (NERSC),
Lawrence Berkeley National Laboratory

NUG Community Call – June 20, 2024

GPUs speed up applications through higher computational throughput…

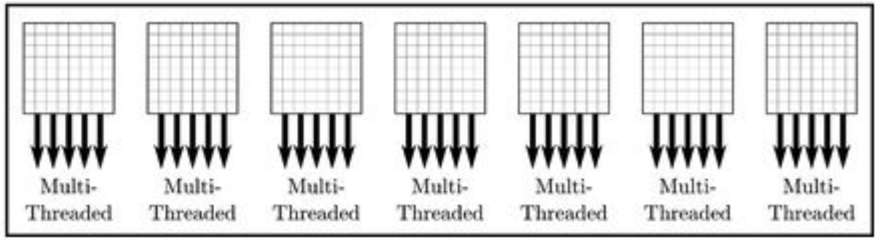


https://developer.nvidia.com/blog/multi-gpu-programming-with-standard-parallel-c-part-1/

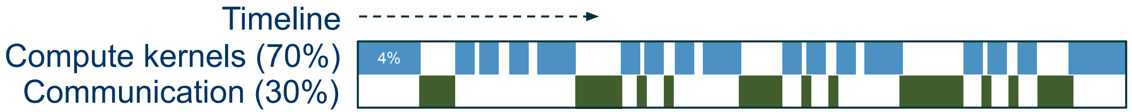
… but we’d like to use them optimally to get maximum speedups.

Communication between GPUs is a major bottleneck

Higher throughput of computation means increasingly larger amounts of data to be transferred between GPUs.



https://developer.nvidia.com/blog/multi-gpu-programming-with-standard-parallel-c-part-1/



Accelerating communication can provide a significant boost to the overall performance.

CUDA-aware MPI makes GPU-GPU communication easy to program and more efficient

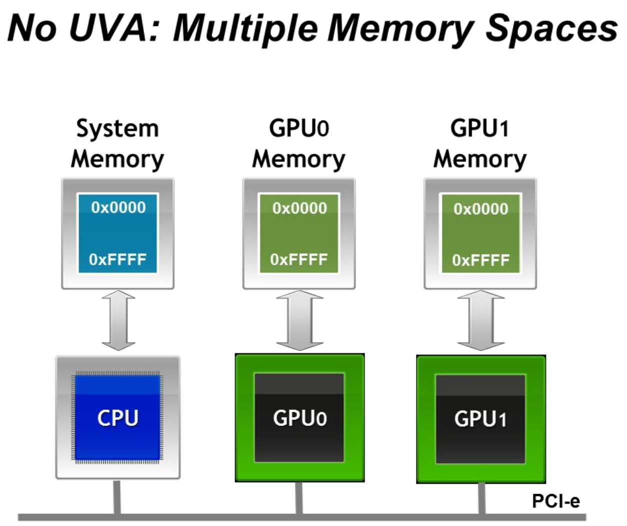
In this talk:

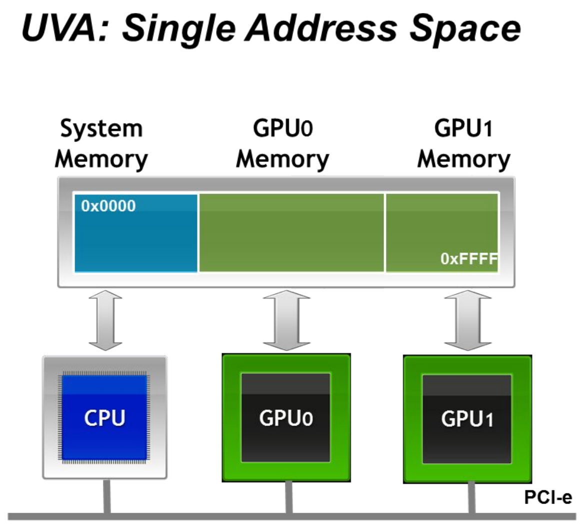
- What is CUDA-aware MPI?
- How does it accelerate communication?
- How do you enable and test it on Perlmutter?
- What is the performance benefit?

Inputs and guidance from: Daniel Margala, Kevin Gott, and Brandon Cook @ NERSC.

Inspired heavily from the technical blog by Jiri Kraus @ NVIDIA: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi

Unified Virtual Addressing combines host and GPU memory into a single virtual address space





By using UVA, CUDA-aware MPI accepts GPU buffers as input

no GPU-aware MPI

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank 1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

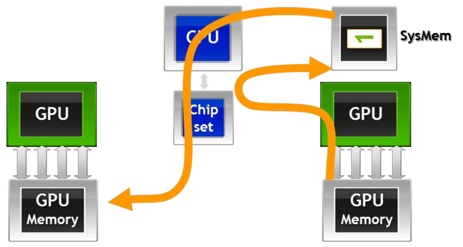
with GPU-aware MPI

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```

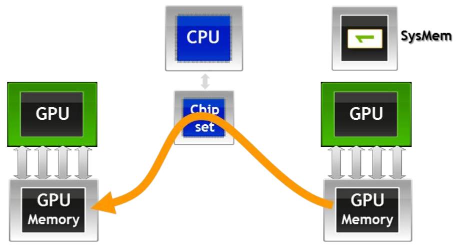
This is easier to program, what about performance?

Data buffers can be directly copied between GPUs without staging through the host

No GPUDirect P2P

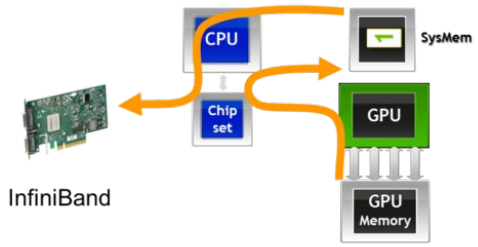


GPUDirect P2P

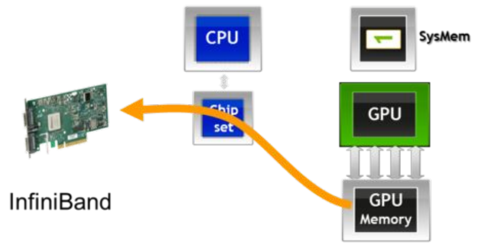


This cuts the overheads from extra buffer copies on the host.

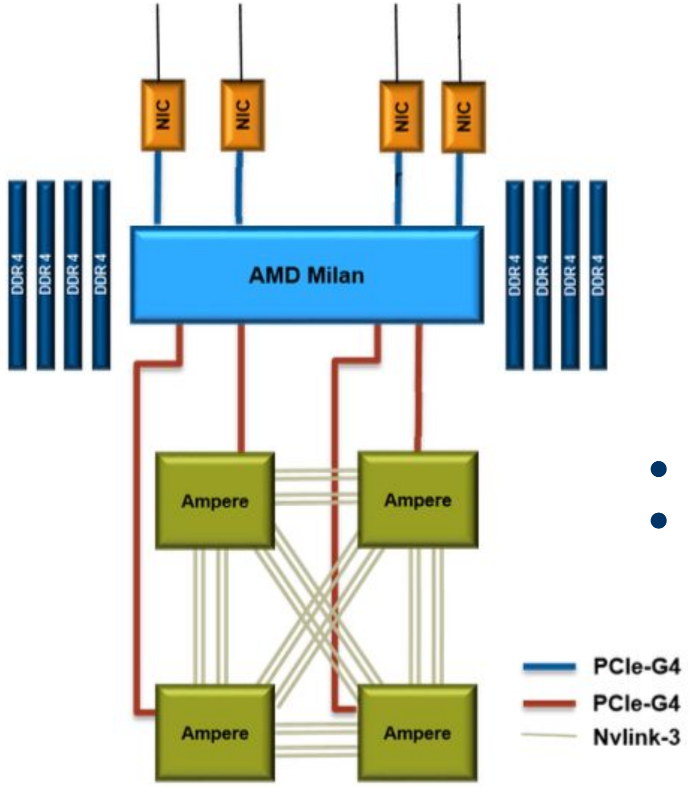
No GPUDirect RDMA



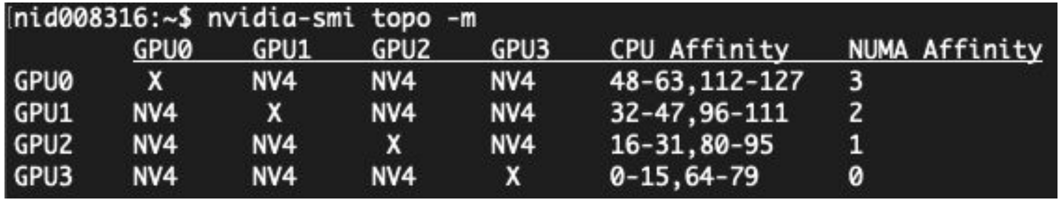
GPUDirect RDMA



Perlmutter GPU node has four GPUs and four NICs



- NVLink directly connects the four GPUs on a node with each other.
- The GPUs and NUMA domains have an “inverse” order of affinity.



```
[nid008316:~$ nvidia-smi topo -m
        GPU0    GPU1    GPU2    GPU3    CPU Affinity    NUMA Affinity
GPU0     X      NV4     NV4     NV4     48-63,112-127   3
GPU1    NV4      X      NV4     NV4     32-47,96-111    2
GPU2    NV4     NV4      X      NV4     16-31,80-95     1
GPU3    NV4     NV4     NV4      X      0-15,64-79      0
```

Enabling CUDA-aware MPI with Cray MPICH and compiler wrappers

At compile time:

```
$ export CRAY_ACCEL_TARGET=nvidia80
```

At run time:

```
$ export MPICH_GPU_SUPPORT_ENABLED=1
```

These are set by default on Perlmutter.

https://docs.nersc.gov/development/programming-models/mpi/cray-mpich/#cuda-aware-mpi

But the process-GPU affinities need to be set manually as SLURM cgroups doesn’t work well with CUDA IPC

For *optimal* affinity, reverse the order of GPUs assigned to the MPI processes and pin processes to the NICs closest to the assigned GPU.

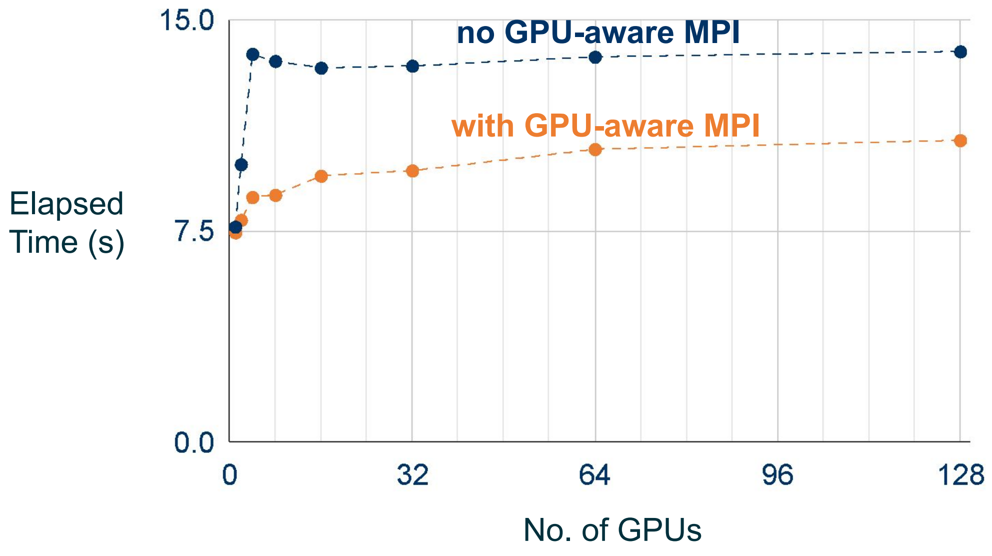
```
#SBATCH --ntasks-per-node=4
#SBATCH --gpus-per-node=4
#SBATCH --gpu-bind=none

# pin to closest NIC to GPU
export MPICH_OFI_NIC_POLICY=GPU

# set ordering of CUDA visible devices inverse to
# local task IDs for optimal affinity
srun -N 2 -n 8 --cpus-per-task=32 --cpu-bind=cores bash -c "
  export CUDA_VISIBLE_DEVICES=\$((3-SLURM_LOCALID));
  ./exe inputs"
```

See docs for more info:
https://cpe.ext.hpe.com/docs/mpt/mpich/intro_mpi.html
https://slurm.schedmd.com/sbatch.html

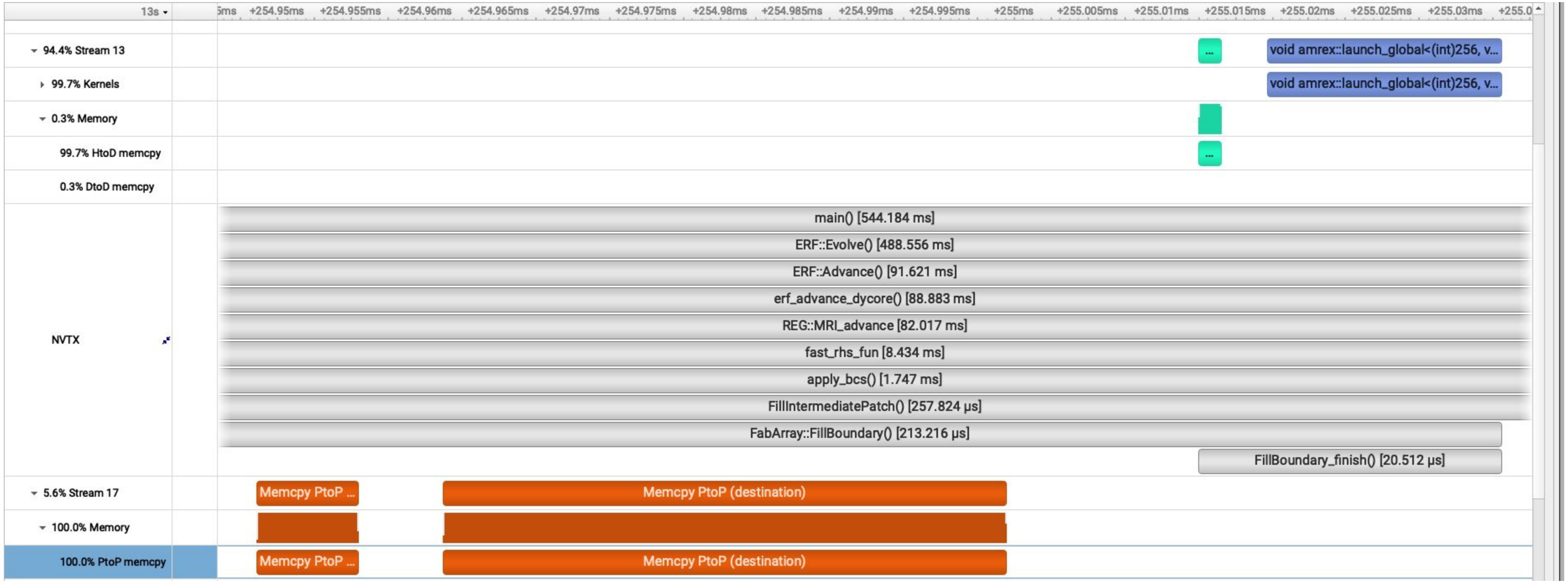
Reduces wall times by 20% for the ERF atmosphere modeling code

Weak scaling of an atmospheric boundary layer simulation using ERF on Perlmutter

The domain size is 128x128x512 for a single GPU;
this is progressively scaled up to 2048x1024x512 for 128 GPUs (over 32 nodes).



P2P transfers are identified by profiling with Nsight Systems



main() [544.184 ms]
ERF::Evolve() [488.556 ms]
ERF::Advance() [91.621 ms]
erf_advance_dycore() [88.883 ms]
REG::MRI_advance [82.017 ms]
fast_rhs_fun [8.434 ms]
apply_bcs() [1.747 ms]
FillIntermediatePatch() [257.824 μs]
FabArray::FillBoundary() [213.216 μs]
FillBoundary_finish() [20.512 μs]

Summary and future direction

- CUDA-aware MPI allows transferring GPU buffers through MPI.
- It can accelerate multi-GPU communication on Perlmutter by directly transferring data buffers between GPU devices, bypassing the hosts.
- This requires manually setting the CPU-GPU-NIC affinities in SLURM.
- Using the NVSHMEM / NCCL GPU communication libraries would also require these settings.
- New updates in SLURM *may* allow cgroups to work well with the CUDA IPC (inter-process communication) layer, preventing the need for users to manually implement the binding.
- Contact NERSC at https://help.nersc.gov/ with questions and feedback on application performance.

Have you used CUDA-aware MPI on Perlmutter?

- Have you used it to code your application? What has been your experience?

- If you are a user/scientist, does your application enable GPU-aware MPI?

- Did this presentation include new information or were you already aware about it?