

Migrating From Cori to Perlmutter: GPU Codes



March 10, 2023

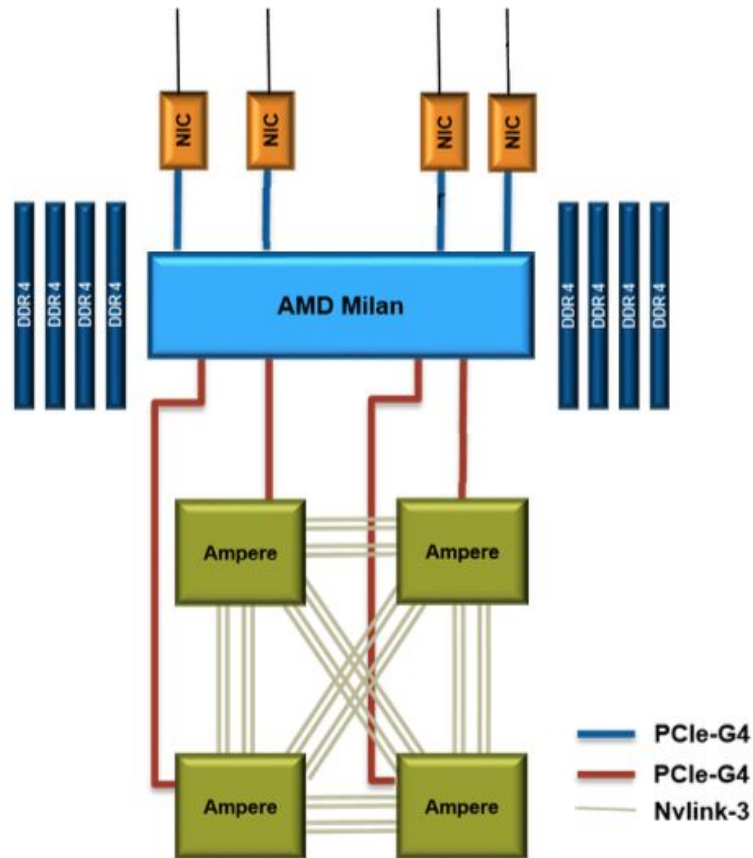
Muaaz Awan, Stephen Leak, Helen He

Outline

- Perlmutter GPU Nodes.
- Programming Environment.
- Hands on Exercises walk through:
 - Launching Jobs
 - Building for GPUs
 - GPU Affinity
 - CUDA-Aware MPI
 - Other GPU Programming Models

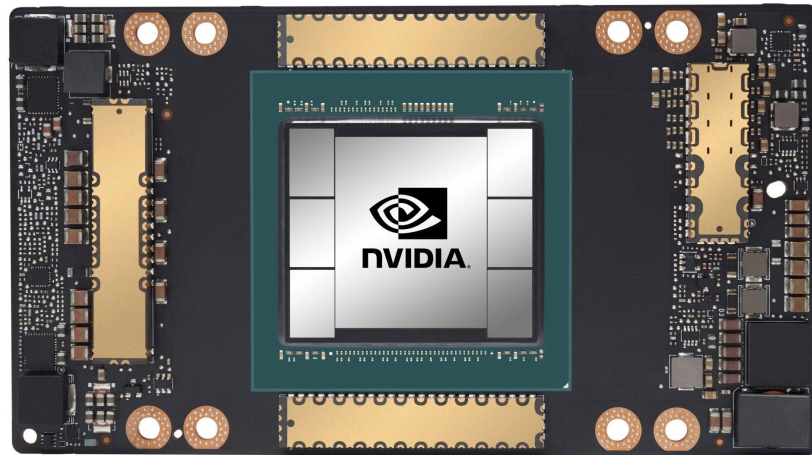
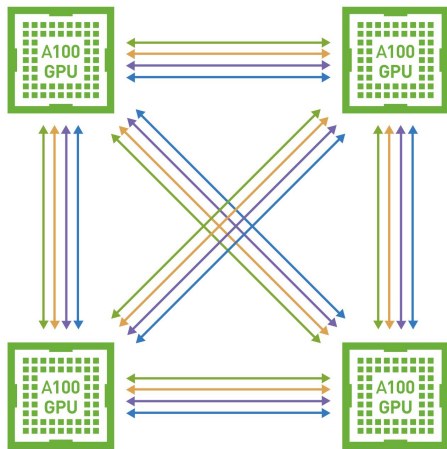
GPU Nodes

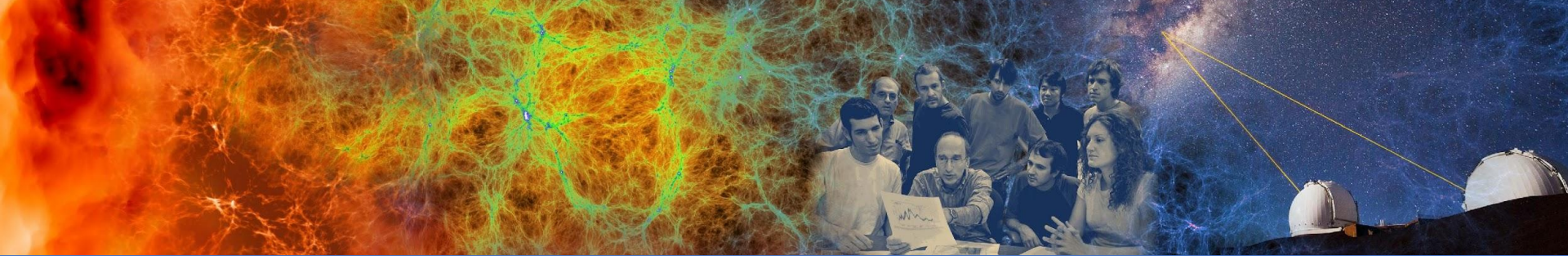
- Perlmutter has 1792 GPU Nodes
 - 1536 Nodes with 40GB-A100
 - 256 Nodes with 80GB-A100
- Each GPU node has:
 - 64 core AMD Milan CPU (7763)
 - 4 NVIDIA A100 GPUs.
- Each Milan CPU core has two hardware threads.



GPU Nodes

- Each pair of GPUs is linked via NVLink.
- CPUs and GPUs communicate via PCIe Gen 4.
- SS11 NICs and CPU communicate via PCIe Gen 4.
- Each node has 256 GB of DDR4 Memory.





Programming Environment

GPU Programming Environment

- By default module **gpu** is loaded in your environment.
- This sets up the environment for applications being built for GPUs.
- By default **cuda toolkit** and **craype-accel-nvidia80** along with other GPU accelerated math libraries are loaded.
- Do note that default programming environment has GNU compilers. If NVIDIA compilers are required, switch to **PrgEnv-nvidia**.

GPU Programming Environment

```
> ml
```

Currently Loaded Modules:

1) craype-x86-milan	6) cray-dsmml/0.2.2	11) perftools-base/23.02.0	16) cudatoolkit/11.7
2) libfabric/1.15.2.0	7) cray-libsci/23.02.1.1	12) cpe/23.02	17) craype-accel-nvidia80
3) craype-network-ofi	8) cray-mpich/8.1.24	13) xalt/2.10.2	18) gpu/1.0
4) xpmem/2.5.2-2.4_3.30__gd0f7936.shasta	9) craype/2.7.19	14) Nsight-Compute/2022.1.1	
5) PrgEnv-gnu/8.3.3	10) gcc/11.2.0	15) Nsight-Systems/2022.2.1	

GPU Programming Environment

```
mgawan@perlmutter:login37:~> ml
```

Currently Loaded Modules:

```
1) craype-x86-milan 4) perftools-base/22.06.0 7) craype/2.7.16 10) cray-libsci/21.08.1.2 13) darshan/3.4.0 16) cudatoolkit/11.7
2) libfabric/1.15.0.0 5) xpmem/2.4.4-2.3_13.8__gff0e1d9.shasta 8) cray-dsmml/0.2.2 11) PrgEnv-gnu/8.3.3 14) Nsight-Compute/2022.1.1 17)
craype-accel-nvidia80
3) craype-network-ofi 6) gcc/11.2.0 9) cray-mpich/8.1.17 12) xalt/2.10.2 15) Nsight-Systems/2022.2.1 18) gpu/1.0
```

```
mgawan@perlmutter:login37:~> CC --version
```

```
g++ (GCC) 11.2.0 20210728 (Cray Inc.)
```

Copyright (C) 2021 Free Software Foundation, Inc.

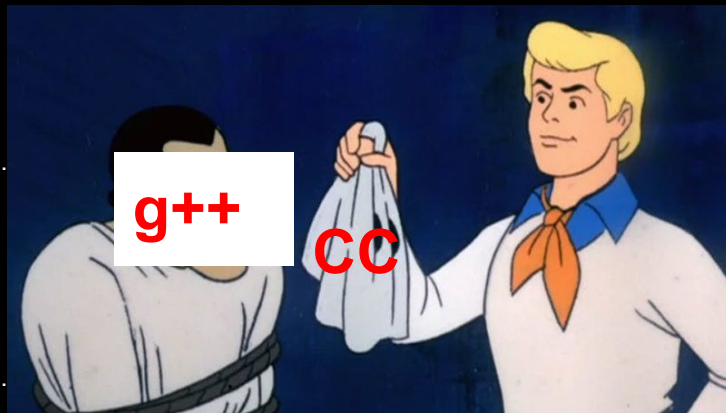
This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

```
mgawan@perlmutter:login37:~> cc --version
```

```
gcc (GCC) 11.2.0 20210728 (Cray Inc.)
```

Copyright (C) 2021 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.



GPU Programming Environment

```
mgawan@perlmutter:login37:~> module load PrgEnv-nvidia
```

Lmod is automatically replacing "gcc/11.2.0" with "nvidia/22.5".

Lmod is automatically replacing "PrgEnv-gnu/8.3.3" with "PrgEnv-nvidia/8.3.3".

Due to MODULEPATH changes, the following have been reloaded:

1) cray-mpich/8.1.17

```
mgawan@perlmutter:login37:~> CC --version
```

```
nvc++ 22.5-0 64-bit target on x86-64 Linux -tp zen3-64
```

NVIDIA Compilers and Tools

Copyright (c) 2022, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

```
mgawan@perlmutter:login37:~> cc --version
```

```
nvc 22.5-0 64-bit target on x86-64 Linux -tp zen3-64
```

NVIDIA Compilers and Tools

Copyright (c) 2022, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

Available Programming Models

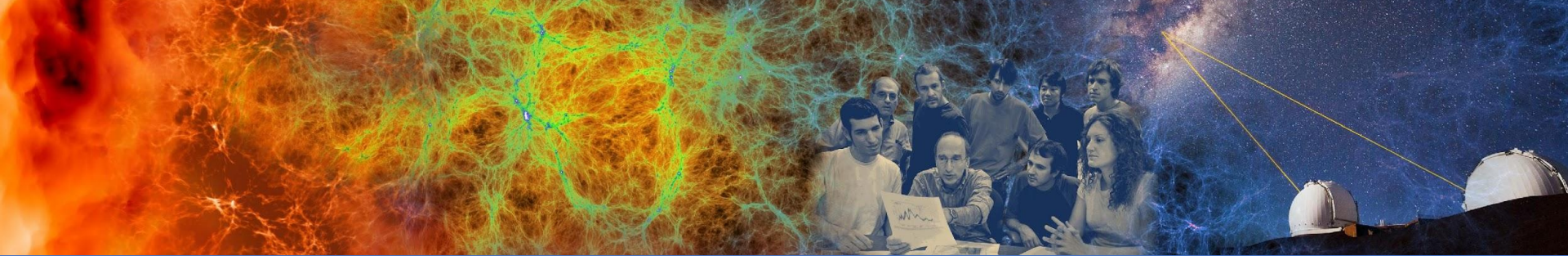
-	Fortran/C/C++	CUDA	OpenACC 2.x	OpenMP 5.x	CUDA Fortran	Kokkos/Raja	MPI	HIP	DPC++/SYCL
nvidia	x	x	x	x	x	x	x		
CCE	x		x	x		x	x		
GNU	x	x	x	x		x	x		
LLVM	x	x		x		x	x	x	x

Recommended Environment

Programming Model	Programming Environment
CUDA	PrgEnv-nvidia/PrgEnv-gnu
Kokkos	PrgEnv-nvidia/PrgEnv-gnu
OpenMP offload	PrgEnv-nvidia/PrgEnv-cray
OpenACC	PrgEnv-nvidia
stdpar	PrgEnv-nvidia

Compiling

- *.c, *.cpp, *.f90 => CPU source code
 - may include MPI
 - may use directives for GPU
 - **compile with regular compilers (Cray wrappers)**
 - **CC** for C++
 - **cc** for C
 - **ftn** for Fortran
- *.cu => CUDA kernels
 - **compile with nvcc**
- (Note: With PrgEnv-nvidia, CUDA can be incorporated into same source files as CPU code, add "-cuda" or "-gpu" flag at compile time)



Hands on Exercises

Hands on Exercises

- Exercises along with instructions are available at: <https://github.com/NERSC/Migrate-to-Perlmutter>
- For GPU examples, move to the GPU folder.
- The README.md file details seven exercises and list steps to build and run them.
- It is suggested to go through these examples during the hands-on session to understand usage of different programming environments.

Hands on Exercises

What's covered:

- Building and running CUDA, OpenACC and OpenMP codes for GPUs.
- Building and running GPU + MPI codes using NVIDIA and GNU programming environment.
- Building and running a CUDA-Aware MPI example.
- Understanding GPU affinity with an example.

Hands on Exercises

- For all the examples, build steps are provided in a **Makefile** within each exercise's directory.
- Each exercise's directory also contains a **batch.sh** file which the users can use to run.

Required SBATCH options (1)

```
#!/bin/bash
#SBATCH -q regular          # "regular" QOS for most jobs
#SBATCH -N 2                # number of Nodes requested
#SBATCH -t 5                # max wallclock time (5 minutes)
#SBATCH -n 8                # number of MPI tasks
#SBATCH -c 32               # reserve 32 cpus per task
#SBATCH --ntasks-per-node=4 # 8 tasks / 4 per node = 2 nodes
#SBATCH --gpus-per-task=1   # 1 GPU per task
#SBATCH -A ntrain8           # project/repo
#SBATCH -C gpu
#SBATCH -reservation=pm_gpu_mar10 # reservation
```

Each GPU node has 64 cores x 2
hyperthreads, so 128 CPUs => 32
cpus is 1/4th of a node

Necessary SBATCH options (2)

```
#!/bin/bash
#SBATCH -q regular           # regular QOS
#SBATCH -N 2                 # number of Nodes requested
#SBATCH -t 5                 # max wallclock time (5 minutes)
#SBATCH -n 8                 # number of MPI tasks
#SBATCH -c 32                # reserve 32 cpus per task
#SUBMIT --ntasks-per-node=4   # 8 tasks / 4 per node = 2 nodes
#SUBMIT --gpus-per-task=1     # reserve 1 (of four) GPUs per task
#SUBMIT -A ntrain8           # GPU
#SUBMIT -C gpu                # Specify a constraint of "run only on gpu nodes"
#SUBMIT -reservation=pm_gpu_mar10 # reservation
```

Useful Runtime Environment Variables

- Generates runtime debug info such as kernel launch and data transfers between host and device.
- PrgEnv-gnu (upcoming gcc/12 compiler)
 - `% export GOMP_DEBUG=1`
- PrgEnv-nvidia (Nvidia compiler)
 - `% export NVCOMPILER_ACC_NOTIFY=<value>`
 - where value can be: 1: kernel launches 2: data transfers
 - 4: region entry/exit 8: wait operations or synchronizations with the device
 - 16: device memory allocates and deallocates
- PrgEnv-cray (CCE compiler)
 - `% export CRAY_ACC_DEBUG=<value>`
 - where value can be 1, 2, 3

Exercise-1: Simple CUDA Kernel

- The source file contains a simple CUDA kernel that adds two vectors and stores the sum in third.
- *nvcc* by default identifies .cu files as containing CUDA.
 - `nvcc -arch=sm_80 vecAdd.cu -o vec_add`
 - `CC -cuda vecAdd.cpp -o vec_add`
- But this practice may not work for larger projects where rest of the code relies on a different compiler.

Exercise-2: CUDA separate compilation

- For complex projects where the host compiler does not recognize CUDA.
- Compile CUDA code separately (in separate files) and link to it later.

```
NVCCFLAGS = -arch=sm_80  
vec_add: kernels.o kernels.h vecAdd.cpp  
    CC -o $@ vecAdd.cpp kernels.o  
kernels.o: kernels.cu kernels.h  
    nvcc $(NVCCFLAGS) -c kernels.cu -o $@
```

Exercise-3: simple MPI + CUDA

- A simple example of MPI + CUDA in same source file is best built with “PrgEnv-nvidia”.
- The CC wrappers link with the MPI specifically built for “PrgEnv-nvidia”.
- When using “PrgEnv-nvidia” wrappers, *-gpu* flags need to be mentioned:

```
NVCCFLAGS = -arch=sm_80
NVCFLAGS = -cuda -gpu=cc80
vec_add: vecAdd.cu
CC $(NVCFLAGS) vecAdd.cu -o $@
```

Exercise-4: separate compilation (MPI + CUDA)

- When using a programming environment other than “PrgEnv-nvidia”, device code needs to be built separately.
- Load the programming environment of your choice, build the CUDA code separately and then link it with MPI wrappers.

```
NVCCFLAGS = -arch=sm_80
NVCCFLAGS = -gpu=cc80
vec_add: kernels.o kernels.h vecAdd.cpp
    CC -o $@ vecAdd.cpp kernels.o
kernels.o: kernels.cu kernels.h
    nvcc $(NVCCFLAGS) -c kernels.cu -o $@
```

Compute Nodes Comparison for CPU Affinity

	Cori Haswell	Cori KNL	Perlmutter CPU	CPU on Perlmutter GPU
Physical cores	32	68	128	64
Logical CPUs per physical core	2	4	2	2
Logical CPUs per node	64	272	256	128
NUMA domains	2	1	8	4
-c value for srun	$2 * \text{floor}(32/\text{tpn})$	$4 * \text{floor}(68/\text{tpn})$, usually do $4 * \text{floor}(64/\text{tpn})$	$2 * \text{floor}(128/\text{tpn})$	$2 * \text{floor}(64/\text{tpn})$

tpn = Number of MPI tasks per node

Launch options and affinity (GPUs)

```
#!/bin/bash
#SBATCH --account=mxxx
#SBATCH --qos=regular
#SBATCH --nodes=2
#SBATCH --time=60
#SBATCH --constraint=gpu
#SBATCH --jobname=myjob
#SBATCH --ntasks-per-node=64
#SBATCH --cpus-per-task=2
#SBATCH --gpus-per-node=4
```

$$c = 2 * \text{floor}(64 / \text{tpn})$$

Where:

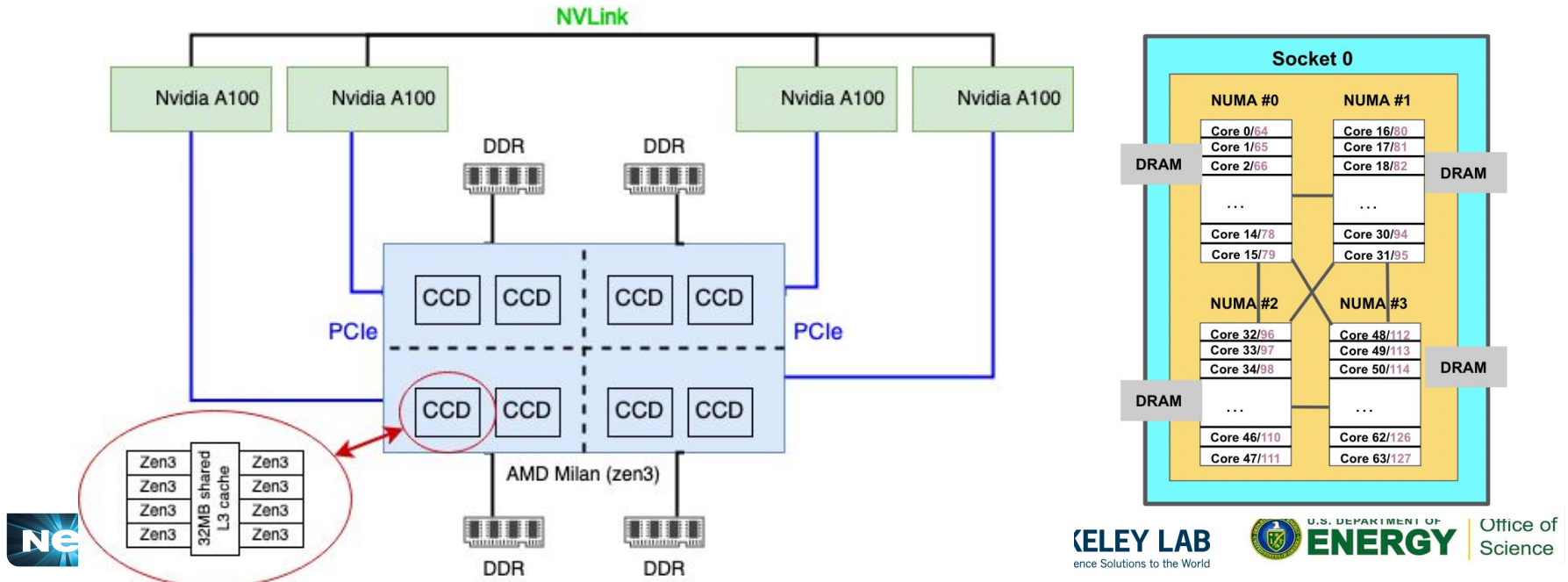
$$\text{tpn} = \text{ntasks-per-node}$$

```
export OMP_NUM_THREADS=1
srun -n 128 -cpu-bind=cores -gpus-bind=closest <executable>
```

- By default all processes will have access to all GPUs.
- A round robin assignment does not guarantee affinity.
- To guarantee that closest GPU is assigned: **-gpus-bind=closest**
- To bind ranks to individual cores: **-cpu-bind=cores**

Affinity and binding

Perlmutter GPU nodes are configured as "NPS4" => 4 NUMA nodes per socket. Each GPU is "closest" to certain cores



Exercise-5: Rank to GPU binding

- This example prints out the cores each MPI rank is residing on along with the GPUs that are visible to each rank.
- By default all the MPI ranks will be able to view all the GPUs.
- Build and test the example by first running with *script_reg.sh* sbatch script.
- Then test with *script_close.sh* sbatch script.
- Notice that using the latter sbatch script each MPI rank can view only the GPU located closest to the corresponding NUMA node.
- The only difference was usage of `--gpu-bind=closest` flag. You can explore other ways this binding can be done, refer to: <https://slurm.schedmd.com/srun.html>

Launch options and affinity (GPUs)

```
srun -n8 --cpu-bind=cores ./vec_add
```

```
Rank 1/8 (PID:73658 on Core: 16) from nid003497 sees 4 GPUs, GPU assigned to me is: = 0000:41:00.0
```

```
Other 3 GPUs are:
```

```
**rank = 0: 0000:03:00.0 **
```

```
**rank = 2: 0000:81:00.0 **
```

```
**rank = 3: 0000:C1:00.0 **
```

```
Rank 5/8 (PID:73662 on Core: 17) from nid003497 sees 4 GPUs, GPU assigned to me is: = 0000:41:00.0
```

```
Other 3 GPUs are:
```

```
**rank = 0: 0000:03:00.0 **
```

```
**rank = 2: 0000:81:00.0 **
```

```
**rank = 3: 0000:C1:00.0 **
```

```
Rank 0/8 (PID:73657 on Core: 0) from nid003497 sees 4 GPUs, GPU assigned to me is: = 0000:03:00.0
```

```
Other 3 GPUs are:
```

```
**rank = 1: 0000:41:00.0 **
```

```
**rank = 2: 0000:81:00.0 **
```

```
**rank = 3: 0000:C1:00.0 **
```

```
Rank 2/8 (PID:73659 on Core: 32) from nid003497 sees 4 GPUs, GPU assigned to me is: = 0000:81:00.0
```

```
Other 3 GPUs are:
```

```
**rank = 0: 0000:03:00.0 **
```

```
**rank = 1: 0000:41:00.0 **
```

```
**rank = 3: 0000:C1:00.0 **
```

Launch options and affinity (GPUs)

```
NUMA node(s): 4
NUMA node0 CPU(s): 0-15,64-79
NUMA node1 CPU(s): 16-31,80-95
NUMA node2 CPU(s): 32-47,96-111
NUMA node3 CPU(s): 48-63,112-127
```

```
NUMANode L#0 (P#0 62GB)
```

```
PCI c1:00.0 (3D)
```

```
NUMANode L#1 (P#1 63GB)
```

```
PCI 82:00.0 (3D)
```

```
NUMANode L#2 (P#2 63GB)
```

```
PCI 41:00.0 (3D)
```

```
NUMANode L#3 (P#3 63GB)
```

```
PCI 03:00.0 (3D)
```

```
Rank 1/8 (PID:102481 on Core: 1) from nid001364 sees 1 GPUs, GPU assigned to me is: = 0000:C1:00.0
Other 0 GPUs are:
```

```
Rank 0/8 (PID:102480 on Core: 0) from nid001364 sees 1 GPUs, GPU assigned to me is: = 0000:C1:00.0
Other 0 GPUs are:
```

```
Rank 5/8 (PID:102486 on Core: 33) from nid001364 sees 1 GPUs, GPU assigned to me is: = 0000:41:00.0
Other 0 GPUs are:
```

```
Rank 2/8 (PID:102482 on Core: 16) from nid001364 sees 1 GPUs, GPU assigned to me is: = 0000:82:00.0
Other 0 GPUs are:
```

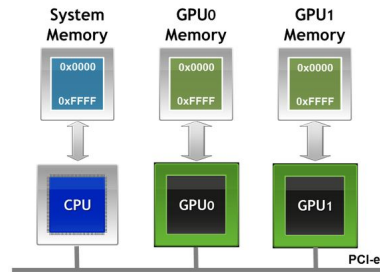
```
Rank 4/8 (PID:102485 on Core: 32) from nid001364 sees 1 GPUs, GPU assigned to me is: = 0000:41:00.0
Other 0 GPUs are:
```

CUDA-aware MPI

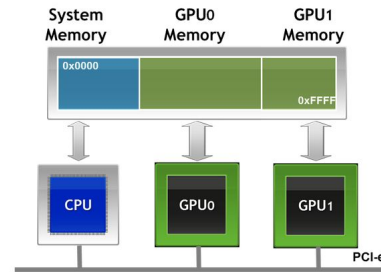
Nvidia UVA (Unified Virtual Addressing) presents GPU device memory as part of the same address space as CPU main memory

- Allows a CUDA-aware MPI implementation (eg Cray-MPICH) to send and receive messages directly from/to GPU memory - no copy-to-main-memory needed

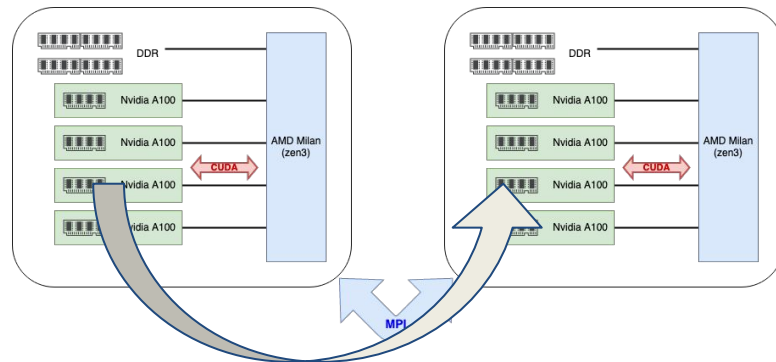
No UVA: Multiple Memory Spaces



UVA: Single Address Space



(from <https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/>)



Example-6: CUDA-aware MPI

Make sure **gpu** module is loaded before trying this example as CUDA-aware MPI requires certain environment setup.

If your executable uses CUDA-aware MPI, `ldd` should show `libmpi_gtl_cuda.so.0`, eg:

```
libmpi_gtl_cuda.so.0 =>  
/opt/cray/pe/lib64/libmpi_gtl_cuda.so.0
```

Exercise-7: OpenACC and OpenMP offload

- This example demonstrates building OpenACC and OpenMP offload codes.
- The example implements the same kernel from previous examples but this time using different programming models.
- Make sure that you have **PrgEnv-nvidia** loaded before trying out the example.

```
ifeq ($(OPENMP),y)
CXXFLAGS += -mp=gpu -gpu=cc80 -Minfo
EXE = vec_add.openmp
else ifeq ($(OPENACC),y)
CXXFLAGS += -acc -Minfo=accel
EXE = vec_add.openacc
```




Thank you!

More questions? Need help? ...
<http://help.nersc.gov/>

