center for
excellence in parallel
programming

CUDA

Georges-Emmanuel Moulard

Bull
atos technologies

Multi GPUs

31-10-2016

Bull
atos technologies

Device Management

31-10-2016

Bull
atos technologies

Get/Set Device

- On a multi GPUs node, each GPU is associated with a unique ID
- By default, a CUDA application will use the device with the ID 0
- The ID of the current device used by the application can be retrieve using:

```
cudaGetDevice ( int  *device )
```

- The device to used can be change using:

```
cudaSetDevice ( int  device )
```

- cudaSetDevice **change the device for the calling** host **thread**

How to Choose Your Device

- Available number of device with compute capability greater or equal to 2.0

```
cudaGetDeviceCount (int * nbDevices);
```

- Properties of a device can be retrieve using:

```
cudaGetDeviceProperties(&prop, device);
```

  - the code sample deviceQuery provided by Nvidia displays the properties

- User can ask for a device that matches at best the desire properties

```
cudaChooseDevice(int* device, const cudaDeviceProp* prop)
```

Devices and Threads / Processes

31-10-2016

Bull
atos technologies

CUDA Context

- A CUDA context is analogous to a CPU process
- CUDA Context initializes the first time a runtime function is called
  - no explicit initialization
  - functions from the device and version management sections of the reference manual do not initialize a context
- Runtime creates a CUDA context for **each device** at initialization
- This context is the primary context for this device and it is shared among all the host threads of the application
- cudaDeviceReset() destroys the primary context of the device the host thread currently operates on

Compute Mode

- The way processes or threads can use a device will depends of the compute mode:
  - "Default" means multiple contexts are allowed per device
  - "Exclusive Process" means only one context is allowed per device, usable from multiple threads at a time
  - "Prohibited" means no contexts are allowed per device (no compute apps)
  - "Exclusive Thread" means only one context is allowed per device, usable from one thread at a time (deprecated)
- As runtime creates a CUDA context for **each device** at initialization be careful with other mode than default

Devices and Processes

- By default, a CUDA application will use the device with the ID 0
  - at “the first” CUDA call
  - it will create a CUDA context for the current thread

- If multiple processes use GPU on a node, they will all use the device 0
  - depends of the compute mode
    - if multiple contexts are allowed per device: ok
    - else: error

- cudaSetDevice **change the device for the calling** host **thread**

Devices and Processes

- A good practice to associate processes and device with MPI can be to:
- Have a number of MPI processes on a system (node) equal to the number of devices
- Or:
  - retrieve the number of devices in the current system (node): nbDevices
  - retrieve the processes rank on the system (node): localRank
  - use cudaSetDevice with a **device number = localRank % nbDevices**
  - !!! Multiple processes on a single GPU could not operate concurrently
    - to avoid context switch and allow concurrently execute considers using MULTI-PROCESS SERVICE (MPS)

Devices and Processes

- Useful MPI function:

```
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
                        MPI_Info info, MPI_Comm *newcomm)
```

- Partitions the group associated with comm into disjoint subgroups, based on the type specified by split_type
- Each subgroup contains all processes of the same type
- The following type is predefined by MPI: **MPI_COMM_TYPE_SHARED**
  - this type splits the communicator into subcommunicators, each of which can create a shared memory region
- **Helpful to have the local rank**

Devices and Threads

- Also depends of the compute mode
- **Each thread have to call cudaSetDevice**
- Common mistake

```
cudaSetDevice(3)

#pragma omp parallel
{
   float *array;
   cudaMalloc( (void **) &array, size);

}
```

Threads will not have the device 3 set !!!

Need an explicit call of cudaSetDevice by each thread

Streams and Multiple Devices

31-10-2016

Bull
atos technologies

Streams and Multiple Devices

- A kernel launch will fail if it is issued to a stream that is not associated to the current device:

```
cudaSetDevice(0);                  // Set device 0 as current
cudaStream_t s0;
cudaStreamCreate(&s0);             // Create stream s0 on device 0
MyKernel<<<gs, bs, 0, s0>>>();     // Launch kernel on device 0 in s0

cudaSetDevice(1);                  // Set device 1 as current
cudaStream_t s1;
cudaStreamCreate(&s1);             // Create stream s1 on device 1
MyKernel<<<gs, bs, 0, s1>>>();     // Launch kernel on device 1 in s1

//kernel launch will fail:
MyKernel<<<gs, bs, 0, s0>>>();     // Launch kernel on device 1 in s0
```

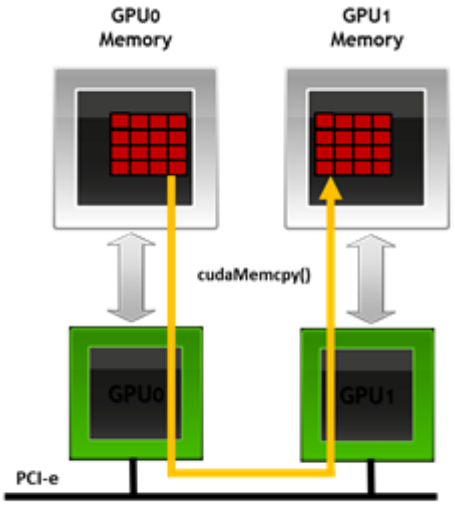
Peer to Peer Accesses

31-10-2016

Bull
atos technologies

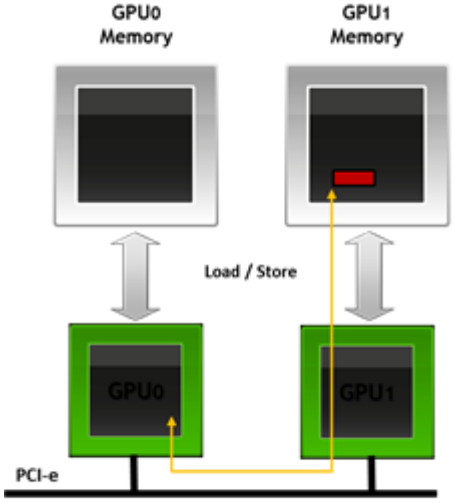
Peer to Peer Accesses

- Direct transfers
  - Use high-speed DMA transfers to copy data between the memories of two GPUs on the same system/PCIe bus.
- Direct Access
  - Optimize communication between GPUs using NUMA-style access to memory on other GPUs from within CUDA kernels.



P2P Direct Transfers

Source : nvidia



P2P Direct Access

Peer to Peer Accesses

- Requirements for Peer to Peer Accesses
  - Needs to be a 64bit application
  - Fermi-class Tesla GPU
  - Linux or Windows TCC
  - CUDA 4.0
  - Drivers v270.41.19 or later
  - GPUs need to be on same IOH (or connected with nvlink)

- Need to allow GPU to be able to make peer-to-peer accesses

```
cudaDeviceCanAccessPeer ( int* canAccessPeer, int  device, int  peerDevice )
```

```
cudaDeviceDisablePeerAccess ( int  peerDevice )
```

```
cudaDeviceEnablePeerAccess ( int  peerDevice, unsigned int  flags (must be 0) )
```

Peer to Peer Accesses

```
cudaSetDevice(0);
int *d_0;
cudaMalloc((void **) &d_0, size*sizeof(int));
cudaMemcpy(d_0, h_val, size*sizeof(int), cudaMemcpyHostToDevice);

cudaSetDevice(1);
int *d_1;
cudaMalloc((void **) &d_1, size*sizeof(int));

cudaDeviceEnablePeerAccess(0,0);
cudaMemcpy(d_1, d_0, size*sizeof(int), cudaMemcpyDeviceToDevice);
cudaMemcpyPeer(d_1, 1, d_0, 0, size*sizeof(int));

cudaMemcpy(h_res, d_1, size*sizeof(int), cudaMemcpyDeviceToHost);
```

```
==36727== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
42.09%  679.77us         2  339.89us  339.29us  340.48us  [CUDA memcpy PtoP]
32.36%  522.62us         1  522.62us  522.62us  522.62us  [CUDA memcpy HtoD]
25.55%  412.70us         1  412.70us  412.70us  412.70us  [CUDA memcpy DtoH]
```

CUDA AWARE MPI

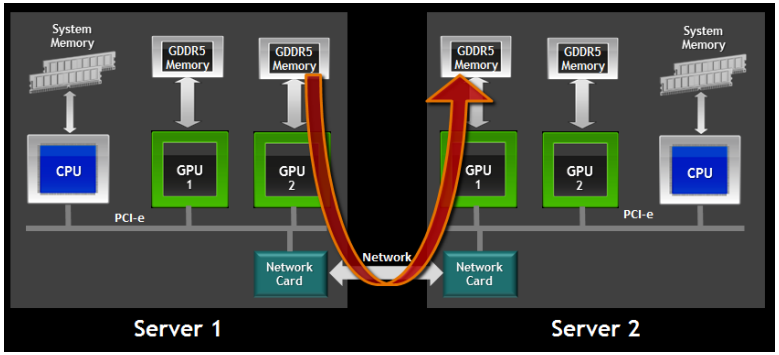
31-10-2016

Bull
atos technologies

CUDA AWARE MPI

- Supported by:
  - MVAPICH2
  - IBM™ Spectrum
  - The Open MPI Project (openmpi): version 1.7 and later

- CUDA-aware MPI implementation can:
  - switch automatically to GPUDirectv2 P2P if GPUs can used P2P accesses
  - bypasse host memory:
    - eliminates CPU bandwidth and latency bottlenecks using RDMA transfers between GPUs and other PCIe devices



MULTI-PROCESS SERVICE (MPS)

31-10-2016

Bull
atos technologies

Multi-Process Service (MPS)

- In CUDA each process has a unique context
- A device can activate only one context at a time
- => Multiple processes using the same GPU can not operate concurrently

- The multi-process service is a software layer that sits between the driver and the application:
  - all CUDA calls will share a single context
  - multiple processes can execute concurrently on the device

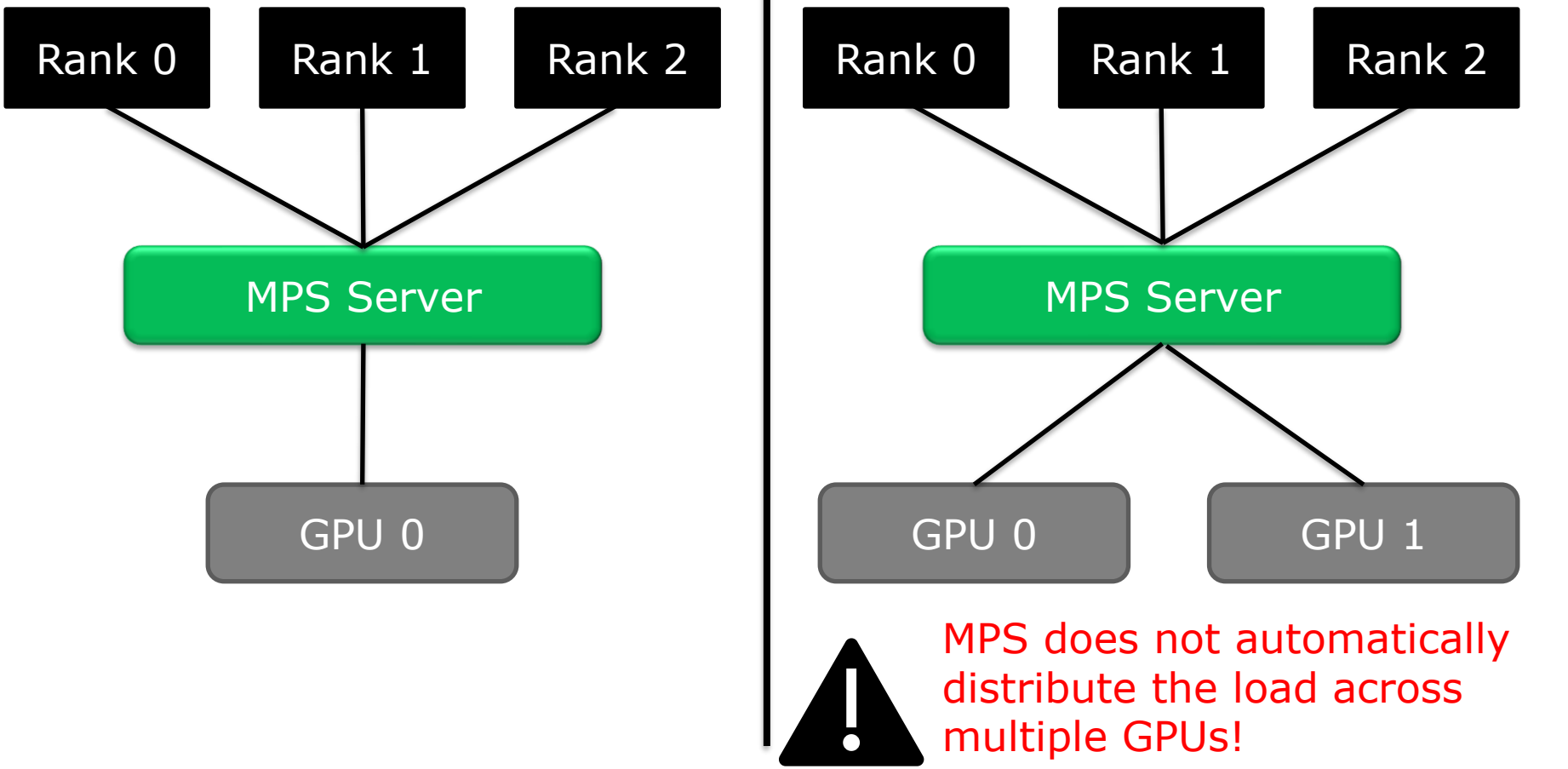
Multi-Process Service (MPS)

- Multiple processes can be run per node using MPS to enable more concurrency
- Benefits:
  - MPS allows kernel and memcopy operations from different processes to overlap on the GPU, achieving higher utilization and shorter running times
  - Reduced GPU context switching
    - Without MPS, when processes share the GPU their scheduling resources must be swapped on and off the GPU
- When to use MPS?
  - when each application process does not generate enough work to saturate the GPU:
    - ex: having a small number of blocks per-grid

Multi-Process Service (MPS)

- MPS does not require any changes to the application / code

Multi-Process Service (MPS)



MPS does not automatically distribute the load across multiple GPUs!
GPU affinity is decided by app

Multi-Process Service (MPS)

Since CUDA 7:

- step1: set the GPU in exclusive mode (need root privilege)

```
sudo nvidia-smi –c 3 –i 0,1,...
```

- step2: start the mps deamon (adjust pipe/log directory)

```
export CUDA_VISIBLE_DEVICES= ...

nvidia-cuda-mps-control –d
```

Multi-Process Service (MPS)

- step3: launch your processes :
  - if you want to use several GPUs, use CUDA_VISIBLE_DEVICES:
    - try to get the local rank of your processes
      - ex: OMPI_COMM_WORLD_LOCAL_RANK
    - as device number for CUDA_VISIBLE_DEVICES use the local rank modulo the number of devices
    - do not hesitate to pinned your processes according to the devices (PCI)
      - numactl
- step4: **Stop the mps deamon & set GPUs their previous compute mode**

```
echo "quit" | nvidia-cuda-mps-control

sudo nvidia-smi –c ??? –i 0,1,...
```

Thanks

For more information please contact:
Georges-Emmanuel Moulard
M+ 33 6 85529054
georges-emmanuel.moulard@atos.net

31-10-2016

Bull
atos technologies