intel Software

IMPLEMENTING HIGH-PERFORMANCE MPI OVER ONEAPI LEVEL ZERO

Sayantan Sur
Principal Engineer, Intel

MUG, 2020

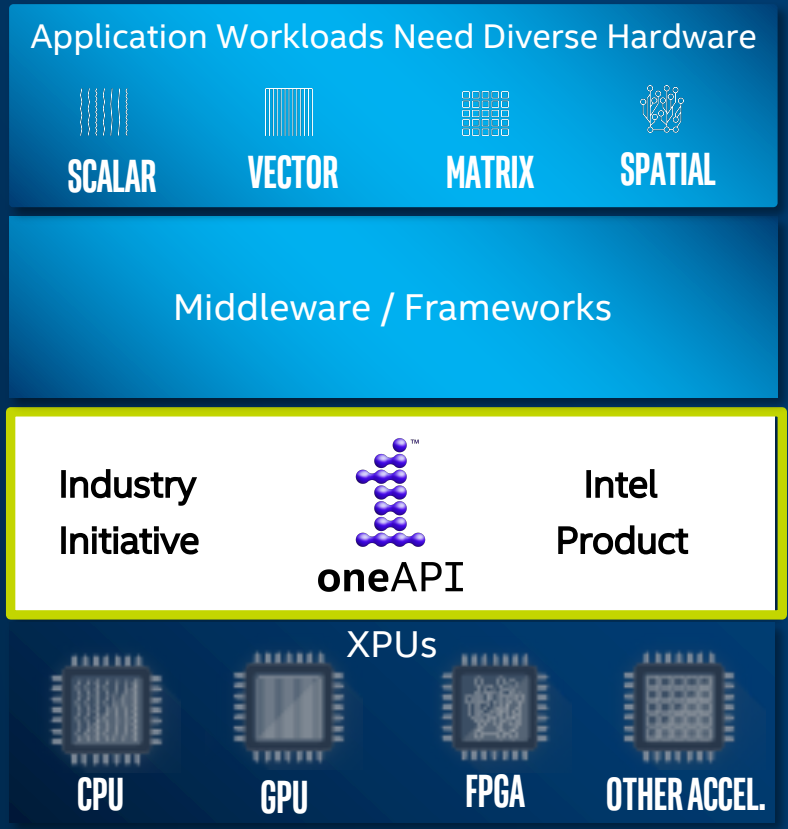
INTRODUCING ONEAPI

Challenges

- Growth in specialized workloads
- No common language or APIs
- Inconsistent tool support across platforms

Introducing oneAPI

- Unified and simplified language and libraries for expressing parallelism
- Based on industry standards and open specifications
  - Interoperable with existing HPC programming models
- Spec available at: www.oneapi.com



ONEAPI LEVEL ZERO

Motivation

- Expose underlying device capabilities to higher level languages
- More control and lower level access to rich device feature set
  - Low latency direct-to-metal interface
  - Support many core threaded applications (e.g. batching)
- Support broader language features such as
  - Function pointers, Virtual functions, Unified Memory, I/O capability

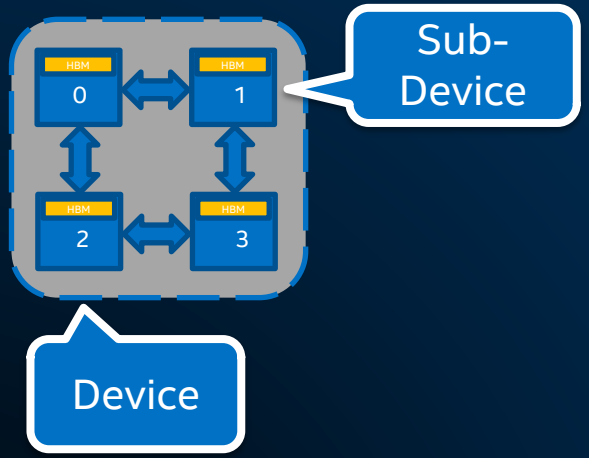
APIs

- Core: Devices, Memory, Peer-to-Peer, Inter-Process …
- Tools: Metrics, Profiling, Debug …
- Sysman: Device configuration, diagnostics, firmware updates …

MPI REQUIREMENTS

- Device enumeration, selection
  - Distributing GPUs amongst ranks, selecting assigned accelerator
- Inter node Peer-to-peer communication
  - RDMA over NICs
- Intra-node Peer-to-peer communication
  - Copies across address spaces

LEVEL ZERO DEVICE MODEL



Device → a physical device in the system

zeDeviceGet() to enumerate devices

Devices can be partitioned into sub-devices

zeDeviceGetSubDevices() to get a list

Subdevices used just like devices (same APIs)

NUMA / locality optimizations with subdevices

Memory allocations with subdevices APIs can be used by the parent Device

DEVICE ASSIGNMENT REQUIREMENTS

Problem:

1. How do all libraries within a process see the same device/subdevice?
   - E.g. User obtains memory using OpenMP/DPC++, and uses MPI to transfer the data
2. How do you distribute devices/subdevices between ranks?
   - E.g. Assign Device 0 to Rank 0, Device 1 to Rank 1 …
   - E.g. For 4 MPI ranks on one device:
     - Assign Subdevice 0 to Rank 0, Subdevice 1 to Rank 1 ..

DEVICE AFFINITY MASK

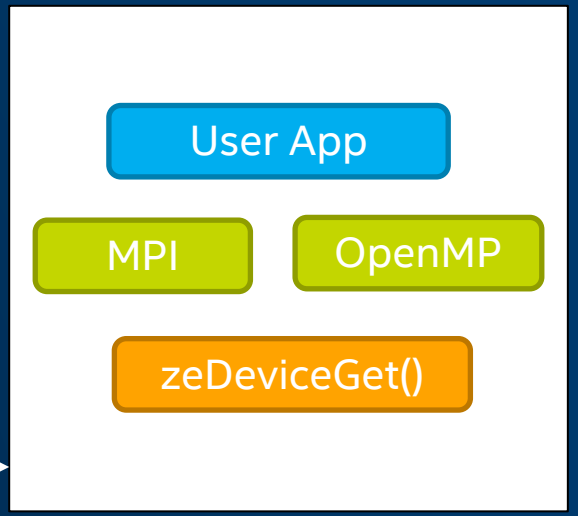
ZE_AFFINITY_MASK environment variable controls which devices are shown and their order

zeInit() reads ZE_AFFINITY_MASK env var

zeDeviceGet() returns devices specified by mask

Spawn Process with ZE_AFFINITY_MASK=1.1,1.2

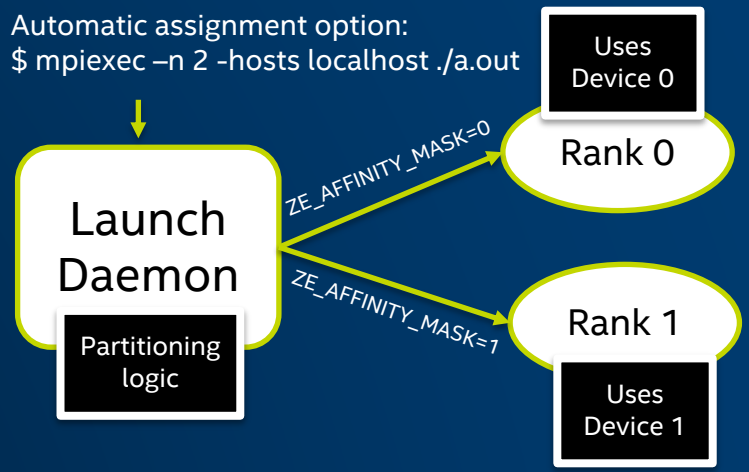
Report Device 1 as “Device 0” and 1.1 as “Subdevice 0” and 1.2 as “Subdevice 1”



ASSIGNING DEVICES TO RANKS

Approaches:

1. Automatic NUMA node awareness in Rank/Device assignment
   - Use OS specific techniques such as PCI hierarchy [abstracted by HWLOC]
   - Level Zero Sysman exposes PCI device IDs
2. Distribute Devices per user specified configuration
   - MPI implementers choice



Node topology specific optimizations possible using apriori knowledge of NUMA nodes and PCI device topology

Level Zero sysman APIs can be used to obtain PCI device IDs

MPI REQUIREMENTS

- Device enumeration, selection
  - Distributing GPUs amongst ranks, selecting assigned accelerator
- Inter node Peer-to-peer communication
  - RDMA over NICs
- Intra-node Peer-to-peer communication
  - Copies across address spaces

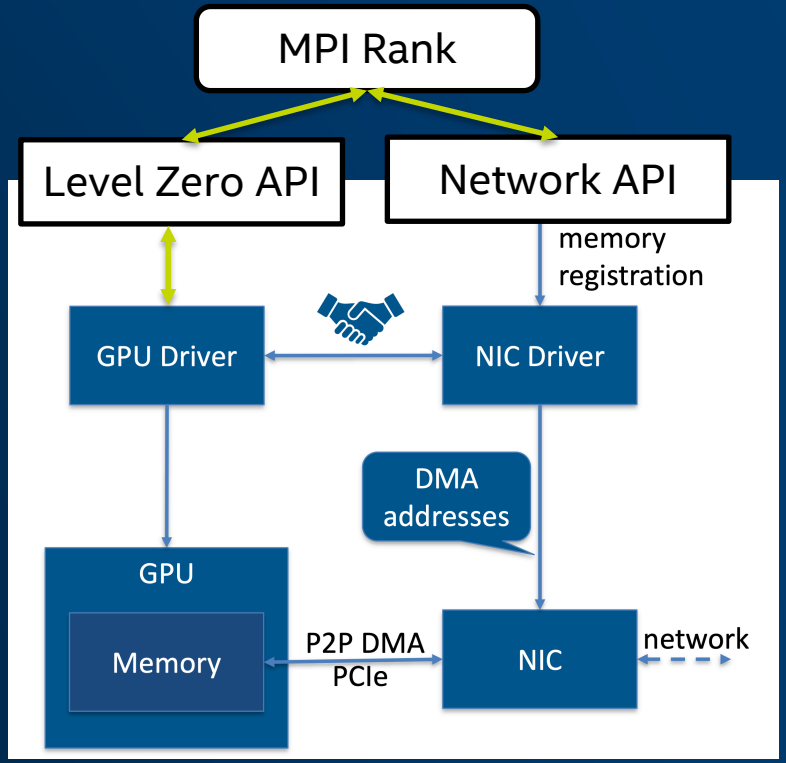
RDMA USING DMA-BUF

Dma-buf is a standard mechanism in kernel for sharing buffers between device drivers

File-descriptor that represents a memory region

fd created by “exporting” driver, passed to the “importing” driver, obtain DMA addresses

zeMemGetIpcHandle() is the Level Zero API used to obtain a Dma-buf handle by the MPI library



Courtesy: Jianxin Xiong (OFA Workshop 2020)

WHERE IS THE BUFFER?

MPI_Send() does not specify if user is trying to send a Device or CPU buffer

MPI library must find out dynamically

- zeMemGetAllocProperties(context, send_buffer, &prop, &device)
  - prop: [output] properties of buffer
    - prop.id will change if VA has been re-assigned to different PA
      - Used for MPI registration caching
  - device: [output] device on which buffer is resident
  - Returns ZE_MEMORY_TYPE_UNKNOWN if buffer wasn't allocated by Level Zero for specified context

```
struct ze_memory_allocation_properties_t
{
       […]
       ze_memory_type_t type;
       uint64_t id;
       uint64_t pageSize;
};

enum ze_memory_type_t
{
       ZE_MEMORY_TYPE_UNKNOWN,
       ZE_MEMORY_TYPE_HOST,
       ZE_MEMORY_TYPE_DEVICE,
       ZE_MEMORY_TYPE_SHARED,
       …
}
```

SAMPLE INTER-NODE P2P PSEUDOCODE

```
MPI_Send(sbuf, size…)
{
      zeMemGetAllocProperties(..sbuf, &prop);
      if (prop.type == ZE_MEMORY_TYPE_DEVICE) {
            zeMemGetAddressRange(sbuf, &base_ptr, &base_size);
            present = registration_cache_check(base_ptr, base_size,
                  &reg_cache_entry);
            if (present && reg_cache_entry.id == prop.id)
                  fi_send(sbuf..);
            } else {
                  zeMemGetIpcHandle(base_ptr, &fd);
                  ..
                  fi_mr_regattr(..&mr_attr, &mr);
                  fi_send(sbuf..);
                  reg_cache_insert(base_ptr, base_size, prop.id);
            }
      } else {
            fi_send(sbuf..);
            ..
```

MPI REQUIREMENTS

- Device enumeration, selection
  - Distributing GPUs amongst ranks, selecting assigned accelerator
- Inter node Peer-to-peer communication
  - RDMA over NICs
- Intra-node Peer-to-peer communication
  - Copies across address spaces

PEER DEVICES

Compute API

- Use this in the main MPI Library
- zeCanAccessPeer(dev, peer_dev, &access)
- Access == TRUE when:
  - Dev == Peer Dev
  - Connected via Scale Up fabric or under same PCIe root complex

Sysman API

- Use this in tools; diagnostics
- Fabric Ports, Port Properties …

```
struct zes_fabric_port_properties_t
{
       […]
       char model[..];
       ze_bool_t onSubDevice;
       uint32_t  subdeviceId;
       zes_fabric_port_id_t portId;
       zes_fabric_port_speed_t max{Rx/Tx}Speed;
};

struct zes_fabric_port_t
{
       […]
       zes_fabric_port_status_t status;
       zes_fabric_port_id_t remotePortId;
       …
}
```

INTER PROCESS COMMUNICATION (IPC)

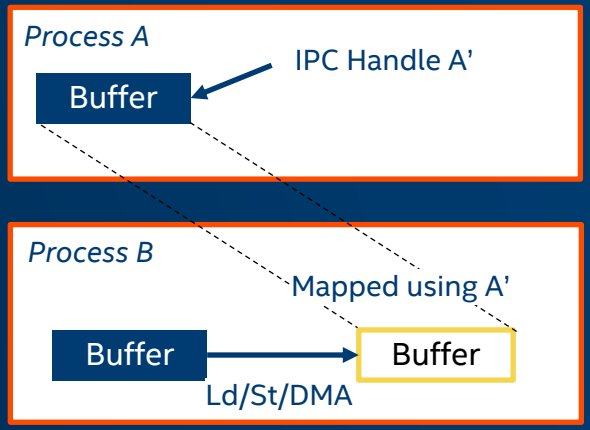
IPC enables a process to map a buffer belonging to another process into its own address space

Ld/St and DMA operations can be used

zeGetMemIpcHandle() used to create IPC Handle on exporting process

zeOpenMemIpcHandle() used to create the mapping in the importing process

IPC Handle is a ‘fd’ of type Dma-buf (for Linux, and specified by Level Zero implementation)



EVENTS, FENCES, CONCURRENCY

```
ze_fence_handle_t hFence;
zeFenceCreate(hCommandQueue,
              &fenceDesc, &hFence);

zeCommandListAppendMemoryCopy(hCommandList,
       dst, src, size,
       hSignalEvent,
       numWaitEvents, *phWaitEvents)

// Execute command list + signal of the fence

zeCommandQueueExecuteCommandLists(hCommandQueue, 1,
       &hCommandList, hFence);

// Wait for fence to be signaled
zeFenceHostSynchronize(hFence, UINT32_MAX);
zeFenceReset(hFence);
```

Start Copy operation when Wait Events have triggered and signal the Event when copy is complete

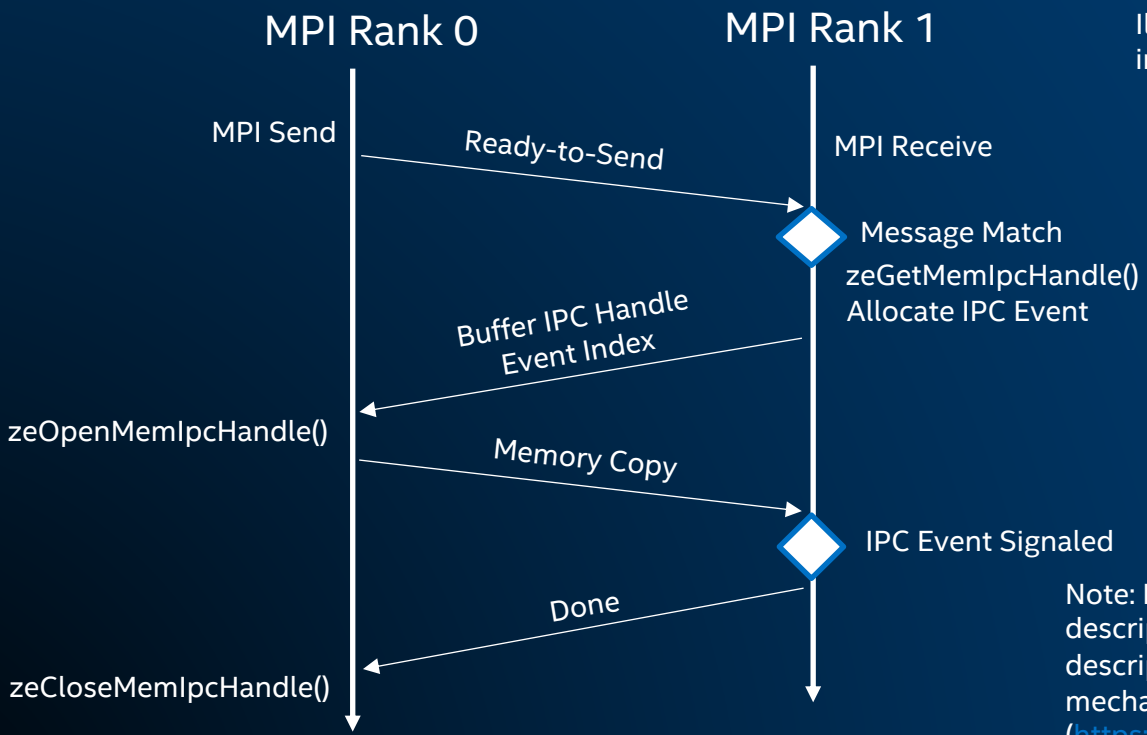
For Inter-process signaling, use exchange Event Pool IPC Handles (zeEventPoolGetIpcHandle, zeEventPoolOpenIpcHandle)

Command List executed FIFO manner, completion can be out of order

“COPY only” hint during command queue creation lets device use DMA engines

Host waits until Fence signaled (all submitted operations complete)

SAMPLE INTRA-NODE PROTOCOL



Illustrative protocol, MPI implementations can optimize

Note: IPC Handle is a process specific file descriptor and inter-process exchange of file descriptors follows standard fd-passing mechanism using UNIX domain sockets (https://www.man7.org/linux/man-pages/man7/unix.7.html → SCM_RIGHTS)

SUMMARY

oneAPI Level Zero Spec: https://spec.oneapi.com/level-zero/latest/index.html

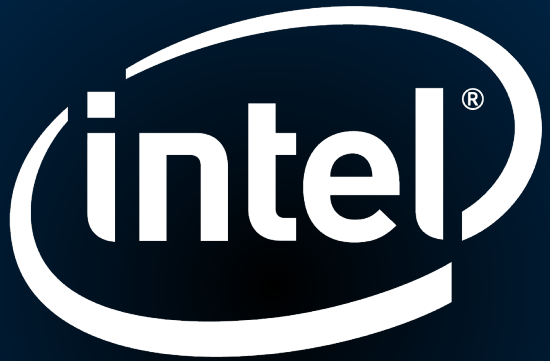
Incorporates state-of-the-art features required for MPI

RDMA operation using Level Zero, Libfabric, Verbs under progress

- Level Zero exports Dma-buf

Open-source MPI implementations in progress

- MPICH: MPL layer ported to Level Zero code, rest coming soon
- Looking forward to additional MPI stacks adopting Level Zero



Software