Redefining the Future of Accelerator Computing with Level Zero

Presenter: Jaime Arteaga

Contributors: Michal Mrozek, Ravindra Babu Ganapathi, Ben Ashbaugh, Brandon Fliflet, Aravind Gopalakrishnan, and Maria Garzaran

Hybrid Working Group MPI Forum 2022

intel®

Outline

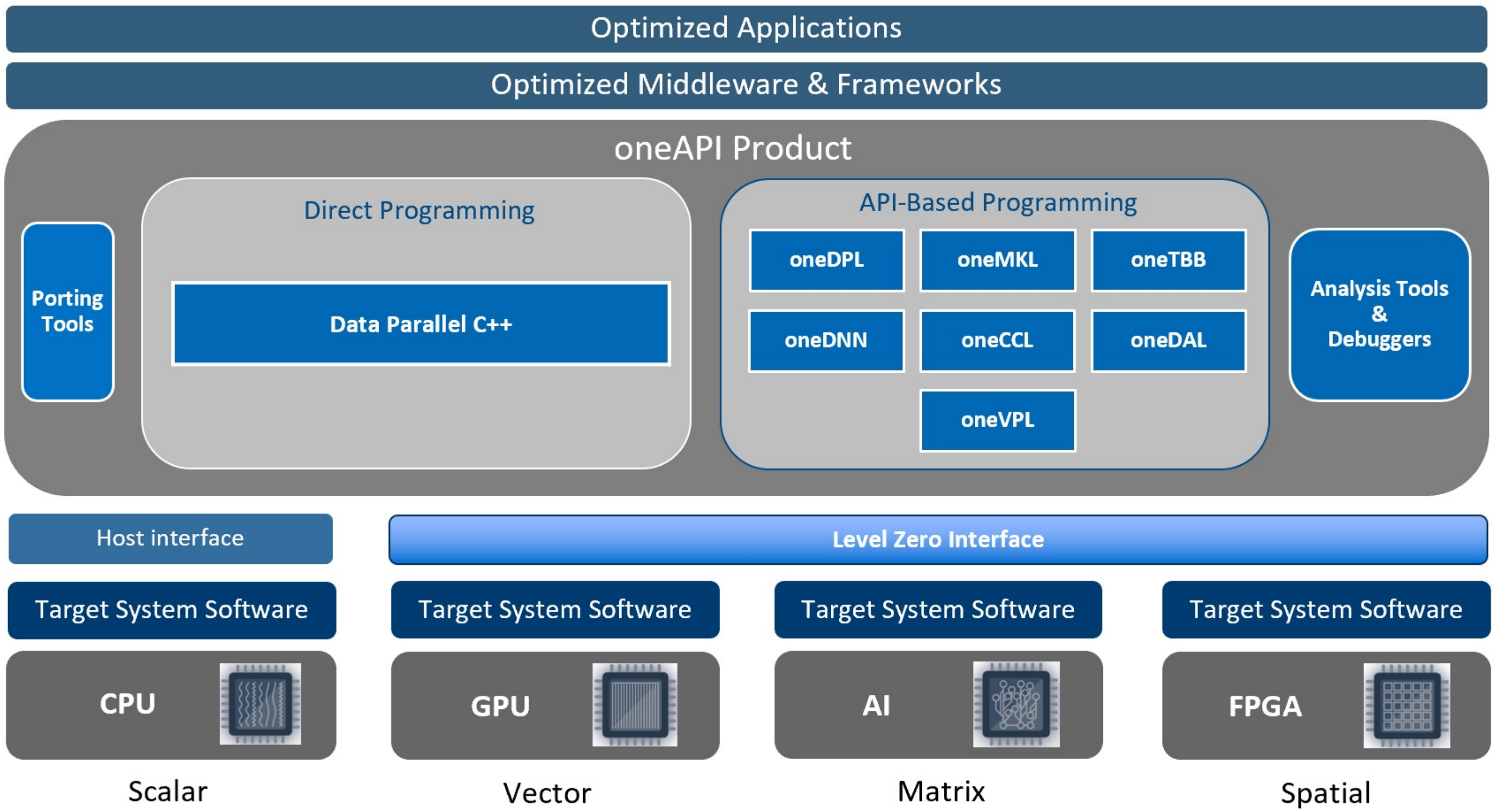
- *What* is Level Zero?
- *Level Zero Device Model*
- *Level Zero Memory Model*
- *Level Zero Scheduling Model*
- *IPC*
- *Summary*

What is Level Zero?

- Low-level driver interface exposing underlying device capabilities to higher level languages
- Abstraction layer for HW accelerators that can be implemented by any vendor for any type of accelerator
- More control and lower-level access to rich device feature set
  - Leads to higher performance and functionality
  - Low latency direct-to-metal interface
  - Support many core threaded applications (e.g., batching)
- Support for broader language features such as
  - Function pointers
  - Unified memory
  - I/O capabilities

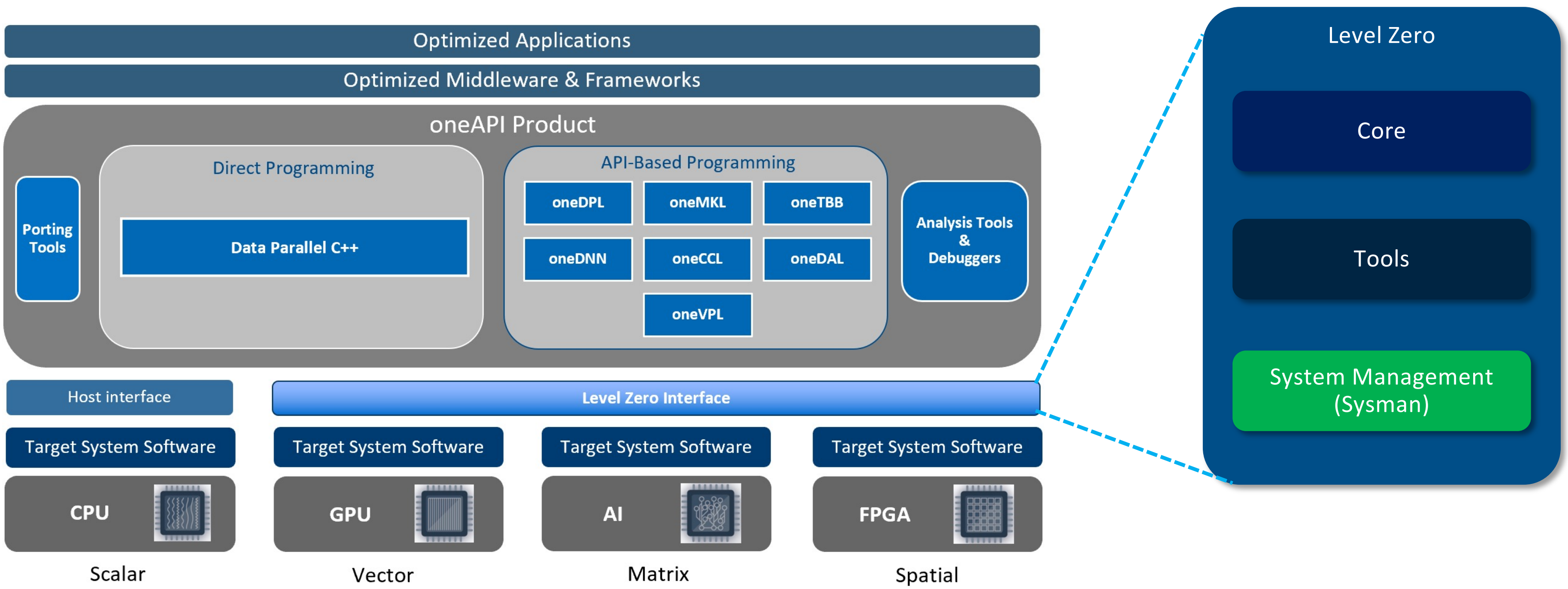
https://spec.oneapi.io/level-zero/latest/index.html

oneAPI and Level Zero Software Stack

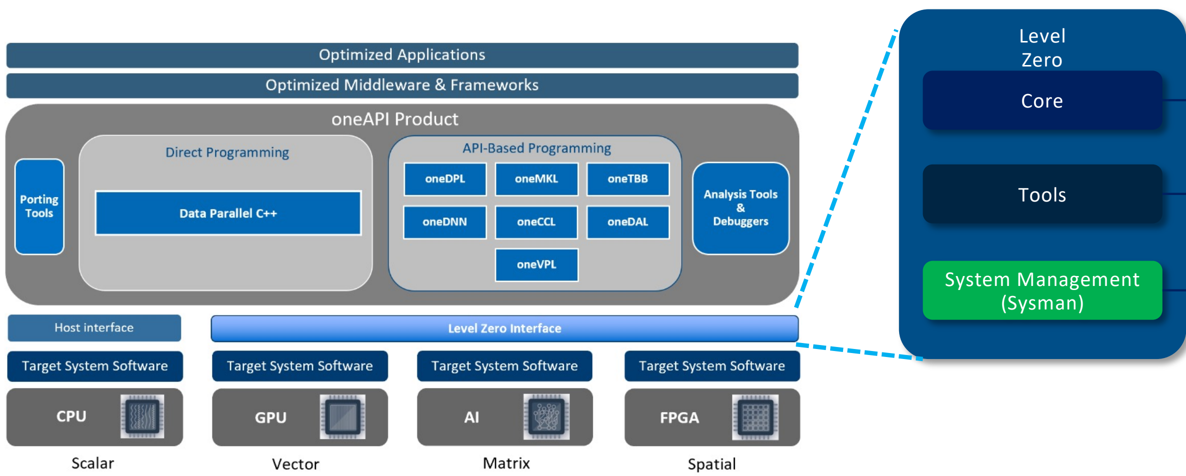


https://spec.oneapi.io/level-zero/latest/core/INTRO.html#objective

oneAPI and Level Zero Software Stack - I



oneAPI and Level Zero Software Stack - II



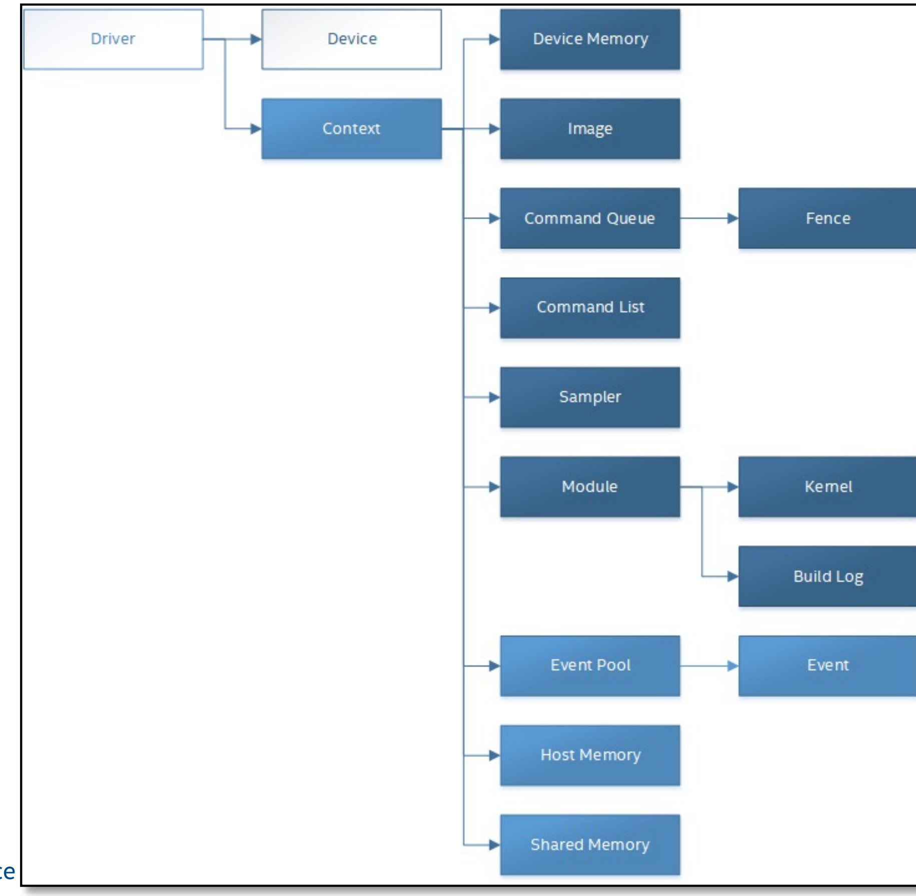
Compute
- Device discovery
- Memory management
- Command submission
- Synchronization

Tools
- Debug
- API tracing
- Metrics
- Program instrumentation

Resource management
- Power
- Frequency
- Temperature
- RAS

Level Zero Drivers and Contexts

- Driver objects represent a collection of physical devices in system
- More than one driver may be available in the system.
  - Example: One driver may support two accelerators from one vendor and another driver support an accelerator from a different vendor.
- Contexts are primarily used during creation and management of resources that may be used by multiple devices.



https://spec.oneapi.io/level-zero/latest/core/PROG.html#device

Level Zero Device Discovery

- Device object represents a physical device in the system
- Device discovery API enumerates devices in system
  - Use zeDeviceGet to
    - Query number of Level Zero devices supported by driver
    - Obtain devices objects which are read-only global constructs.
  - Device Properties queried using zeDeviceGetProperties
    - Device type, max memory allocation size, …
  - Universal Unique Identifier (UUID)
    - 16-byte globally unique and immutable identifier
    - Uniquely identify particular device in a node within a datacenter
- Device handle is primarily used during creation and management of resources that are specific to a device

https://spec.oneapi.io/level-zero/latest/core/PROG.html#drivers-and-devices

Sub-Devices

- Level Zero allows for fine-grain abstraction of HW by exposing *sub-devices*
- Implementations are free to define what a sub-device is, including:
  - Compute capabilities: Number of queues, scheduling policies available.
  - Memory: How much memory each sub-device uses.
  - Memory affinity: How memory affinitize to available queues.
- API available to query and obtain a sub-device from a parent device
- No distinction between sub-devices and parent devices in scheduling and allocation interfaces, as both use same handle (or data type).

Level Zero Device Discovery

L0 driver initialization

Driver discovery

Context creation

Device discovery

```
zeInit(ZE_INIT_FLAG_GPU_ONLY);

uint32_t driverCount = 0;
zeDriverGet(&driverCount, nullptr);
if (driverCount == 0) {
    std::terminate();
}

zeDriverGet(&driverCount, &driverHandle);

ze_context_desc_t contextDesc = {};
zeContextCreate(driverHandle, &contextDesc, &context);

uint32_t deviceCount = 0;
zeDeviceGet(driverHandle, &deviceCount, nullptr);
if (deviceCount == 0) {
    std::terminate();
}

std::vector<ze_device_handle_t> devices(deviceCount);
zeDeviceGet(driverHandle, &deviceCount, devices.data());
```

Level Zero Memory

- Visible to upper-level software stack as unified memory with single VA space.
- Two types of allocations
  - Memory – linear, unformatted allocations for direct access from host/device
  - Images – non-linear, formatted allocations for direct access from device
- Memory
  - Host – Owned by host and accessible by host and one or more devices
  - Device – Owned by device (local mem) and generally not accessible by host.
  - Shared – Share ownership between device and host (intended to migrate)
- Designed support for system allocator (e.g., malloc/new)

https://spec.oneapi.io/level-zero/latest/core/PROG.html#memory-and-images

Level Zero Memory

Host Memory

```
void *hostBuffer = nullptr;
ze_host_mem_alloc_desc_t hostDesc = {ZE_STRUCTURE_TYPE_HOST_MEM_ALLOC_DESC};
zeMemAllocHost(context, &hostDesc, srcMemorySize, 1, &hostBuffer);
```

Device Memory

```
void *deviceBuffer = nullptr;
ze_device_mem_alloc_desc_t deviceDesc = {ZE_STRUCTURE_TYPE_DEVICE_MEM_ALLOC_DESC};
zeMemAllocDevice(context, &deviceDesc, allocSize, allocSize, device, &deviceBuffer);
```

Shared Memory

```
void *sharedBuffer = nullptr;
ze_device_mem_alloc_desc_t deviceDesc = {ZE_STRUCTURE_TYPE_DEVICE_MEM_ALLOC_DESC};
ze_host_mem_alloc_desc_t hostDesc = {ZE_STRUCTURE_TYPE_HOST_MEM_ALLOC_DESC};
zeMemAllocShared(context, &deviceDesc, &hostDesc, allocSize, 1, device, &sharedBuffer);
```

Level Zero Scheduling Model

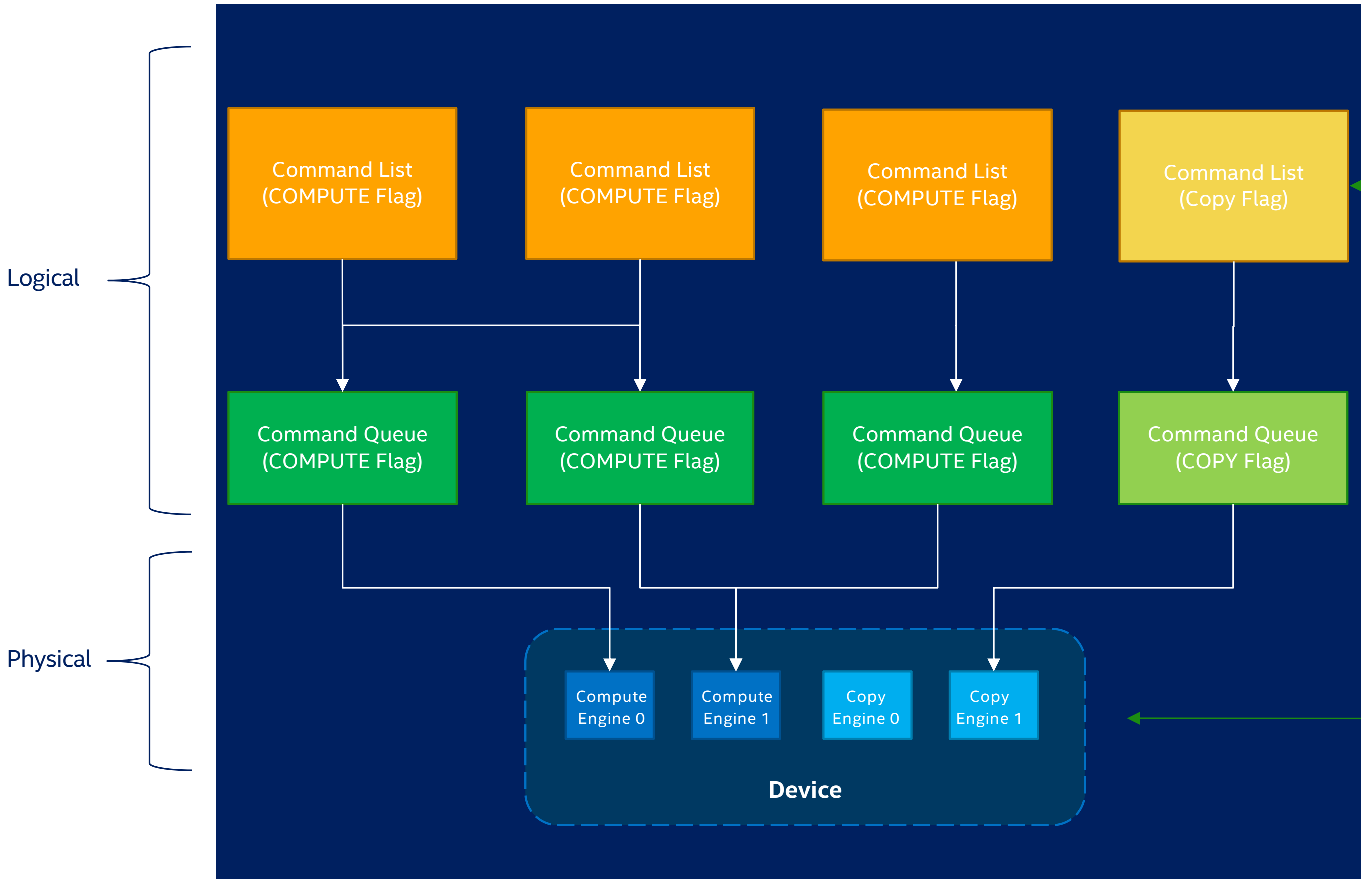
- Commands (user kernels, copies, synchronization actions) are appended to **command lists**.
  - A command list represents a sequence of commands to be executed in the accelerator
    - Can be recycled by resetting the list, without needing to create it again.
    - Can be reused, by submitting the same sequence the commands several times, without needing to re-append commands.
- Command lists are then submitted for execution to a **command queue.**
  - Logical object associated to a physical input stream in the device
  - Can be configured as synchronous or asynchronous
  - Types of command queues exposed as **Queue Groups**
    - **Compute** and **Copy** groups in a GPU for instance

https://spec.oneapi.io/level-zero/latest/core/PROG.html#command-queues-and-command-lists

Level Zero Scheduling Model

- Commands and submissions can be synchronized using:
  - **Events:** Fine-grain synchronization between commands, host, and devices
    - Can be associated with any command list
    - Can be shared between processes and devices
    - Can be signaled/wait upon on either host or device, synchronously or asynchronously
    - Can provide timestamps for kernel execution
  - **Fences:** Coarse-grain synchronization of a command queue submission, signaled by device and waited upon on the host

Level Zero Scheduling Model



- Container for commands
- Simultaneous construction across host threads.
- Submission to multiple command queues supported.

- Associated with device input streams
- Provide (near) zero-latency access to device

- Device, sub-device, and memory are only physical abstraction
- Command queues are bound to device input streamer at create time using an ordinal

https://spec.oneapi.io/level-zero/latest/core/PROG.html#command-queues-and-command-lists

Level Zero Scheduling Model

Query queue groups

Look for a queue group with compute capabilities

Create command queue

Create command list

```
zeCall(zeDeviceGetCommandQueueGroupProperties(device, &numQueueGroups, nullptr));

std::vector<ze_command_queue_group_properties_t> queueProperties(numQueueGroups);
zeCall(zeDeviceGetCommandQueueGroupProperties(device, &numQueueGroups,
                                              queueProperties.data()));


uint32_t computeOrdinal = std::numberic_limits<uint32_t>::max();
for (uint32_t i = 0; i < numQueueGroups; i++) {
    if (queueProperties[i].flags & ZE_COMMAND_QUEUE_GROUP_PROPERTY_FLAG_COMPUTE) {
        computeOrdinal = i;
        break;
    }
}

ze_command_queue_handle_t queue;
ze_command_queue_desc_t cmdQueueDesc = {};
cmdQueueDesc.ordinal = computeOrdinal;
cmdQueueDesc.mode = ZE_COMMAND_QUEUE_MODE_ASYNCHRONOUS;
zeCommandQueueCreate(context, device, &cmdQueueDesc, &queue);


ze_command_list_handle_t list;
ze_command_list_desc_t listDesc = {};
listDesc.commandQueueGroupOrdinal = ordinal;
zeCommandListCreate(context, this->device, &listDesc, &list);
```

Level Zero Scheduling Model

- Module creation
- Kernel creation
- Kernel setup
- Append kernel
- Submit kernel for execution
- Wait for completion

```
ze_module_handle_t module = nullptr;
ze_module_desc_t moduleDesc = {ZE_STRUCTURE_TYPE_MODULE_DESC};
ze_module_build_log_handle_t buildlog;
moduleDesc.format = ZE_MODULE_FORMAT_IL_SPIRV;
moduleDesc.pInputModule = reinterpret_cast<const uint8_t *>(spirvInput.get());
moduleDesc.inputSize = length;
moduleDesc.pBuildFlags = "";
zeModuleCreate(context, device, &moduleDesc, &module, nullptr);

ze_kernel_handle_t kernel = nullptr;
ze_kernel_desc_t kernelDesc = {ZE_STRUCTURE_TYPE_KERNEL_DESC};
kernelDesc.pKernelName = "CopyBufferToBufferBytes";
zeKernelCreate(module, &kernelDesc, &kernel);

uint32_t groupSizeX = 32u;
uint32_t groupSizeY = 1u;
uint32_t groupSizeZ = 1u;
zeKernelSuggestGroupSize(kernel, allocSize, 1U, 1U, &groupSizeX, &groupSizeY, &groupSizeZ);
zeKernelSetGroupSize(kernel, groupSizeX, groupSizeY, groupSizeZ);

zeKernelSetArgumentValue(kernel, 1, sizeof(dstBuffer), &dstBuffer);
zeKernelSetArgumentValue(kernel, 0, sizeof(srcBuffer), &srcBuffer);

ze_group_count_t dispatchTraits;
dispatchTraits.groupCountX = allocSize / groupSizeX;
dispatchTraits.groupCountY = 1u;
dispatchTraits.groupCountZ = 1u;

zeCommandListAppendLaunchKernel(cmdList, kernel, &dispatchTraits,
                                nullptr, 0, nullptr));


zeCommandListClose(cmdList);
zeCommandQueueExecuteCommandLists(cmdQueue, 1, &cmdList, nullptr);

zeCommandQueueSynchronize(cmdQueue, std::numeric_limits<uint64_t>::max());
```

Inter-Process Communication (IPC)

- IPC calls allow sharing of memory objects across different device processes.
- There are two types of IPC APIs across processes:
  - Memory
  - Events

Inter-Process Communication (Memory)

- The sender process:
  - Gets a handle for the allocation on the sending process through zeMemGetIpcHandle
  - Sends the handle to the receiving process
- The receiver process
  - Receive the handle from receiving process
  - Opens the handle on the virtual space of the receiving process through zeMemOpenIpcHandle

Inter Process Communication (Memory)

Memory allocation →

```
void* dptr = nullptr;
zeMemAllocDevice(hContext, &desc, size, alignment, hDevice, &dptr);
```

GetIpcHandle →

```
ze_ipc_mem_handle_t hIPC;
zeMemGetIpcHandle(hContext, dptr, &hIPC);
```

Send Ipc Handle →

```
send_to_receiving_process(hIPC);
```

sender rank

Receive Ipc Handle →

```
ze_ipc_mem_handle_t hIPC;
hIPC = receive_from_sending_process();
```

Open Ipc Handle →

```
void* dptr = nullptr;
zeMemOpenIpcHandle(hContext, hDevice, hIPC, 0, &dptr);
```

receiver rank

A method to send the Ipc handle is shown in
https://github.com/TApplencourt/GPU_IPC_Handle/blob/main/syclo_ipc_copy_dma_buf.cpp#L22

Or you can use the pidfd_getfd system call: https://github.com/intel/compute-runtime/issues/448

Level Zero Websites

- Full open-source!
  - Specification:
    - https://spec.oneapi.io/level-zero/latest/index.html
  - Level Zero loader:
    - https://github.com/oneapi-src/level-zero
  - Level Zero Conformance Tests:
    - https://github.com/oneapi-src/level-zero-tests
  - Level Zero Implementation for Intel GPUs:
    - https://github.com/intel/compute-runtime

Programming models already using Level Zero

- DPC++ Level Zero Plugin:
  - https://github.com/intel/llvm/tree/sycl/sycl/plugins/level_zero
- oneAPI Julia
  - https://github.com/JuliaGPU/oneAPI.jl
- Intel oneAPI Level Zero Introduction
  - https://www.intel.com/content/www/us/en/develop/documentation/oneapi-dpcpp-cpp-compiler-dev-guide-and-reference/top/optimization-and-programming-guide/intel-oneapi-level-zero-introduction.html

Summary

- Level Zero provides a rich set of interfaces to schedule work and manage memory on different accelerators.
- Objects defined in Level Zero, such as Command Queues and Command Lists, allow for a low-level control of the underlying hardware.
- With these and available optimization techniques, high-level programming languages and applications may execute workloads with close-to-metal latencies for higher performance.