Redefining the Future of Accelerator Computing with Level Zero

Jaime Arteaga
Michal Mrozek
Ravindra Babu Ganapathi
Ben Ashbaugh
Brandon Fliflet
Aravind Gopalakrishnan

X.Org Developer Conference 2021
September 16th, 2021

Outline

- *What* is Level Zero?
- *How* do we use Level Zero?
  - How to run a basic kernel
- *Taking Level Zero to Next Level*
  - *Advanced Features*
  - *Optimization Techniques*

What is Level Zero?

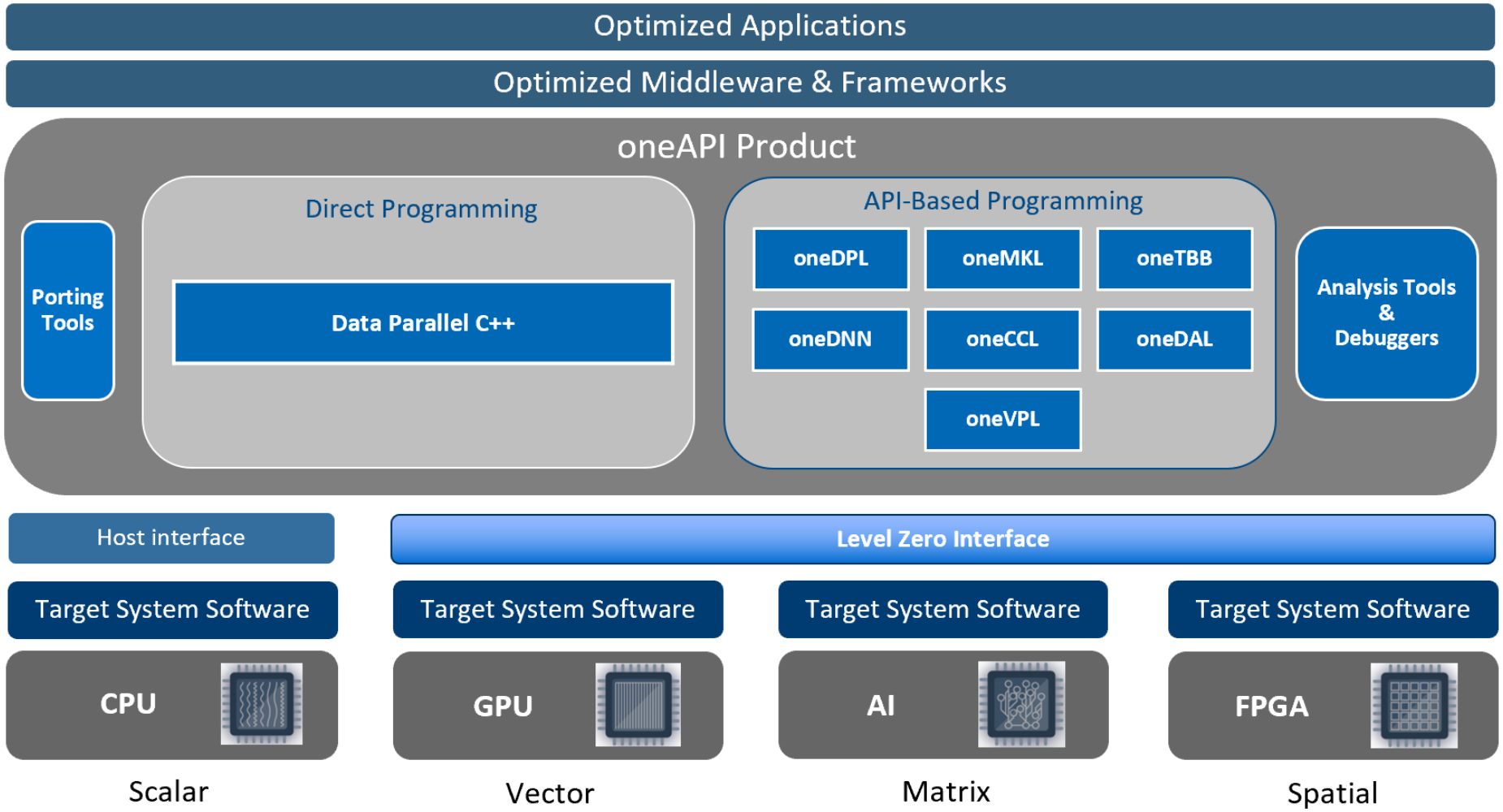
- Low-level driver interface exposing underlying device capabilities to higher level languages
- Abstraction layer for HW accelerators that can be implemented by any vendor for any type of accelerator
- More control and lower-level access to rich device feature set
  - Leads to higher performance and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

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

What is Level Zero?

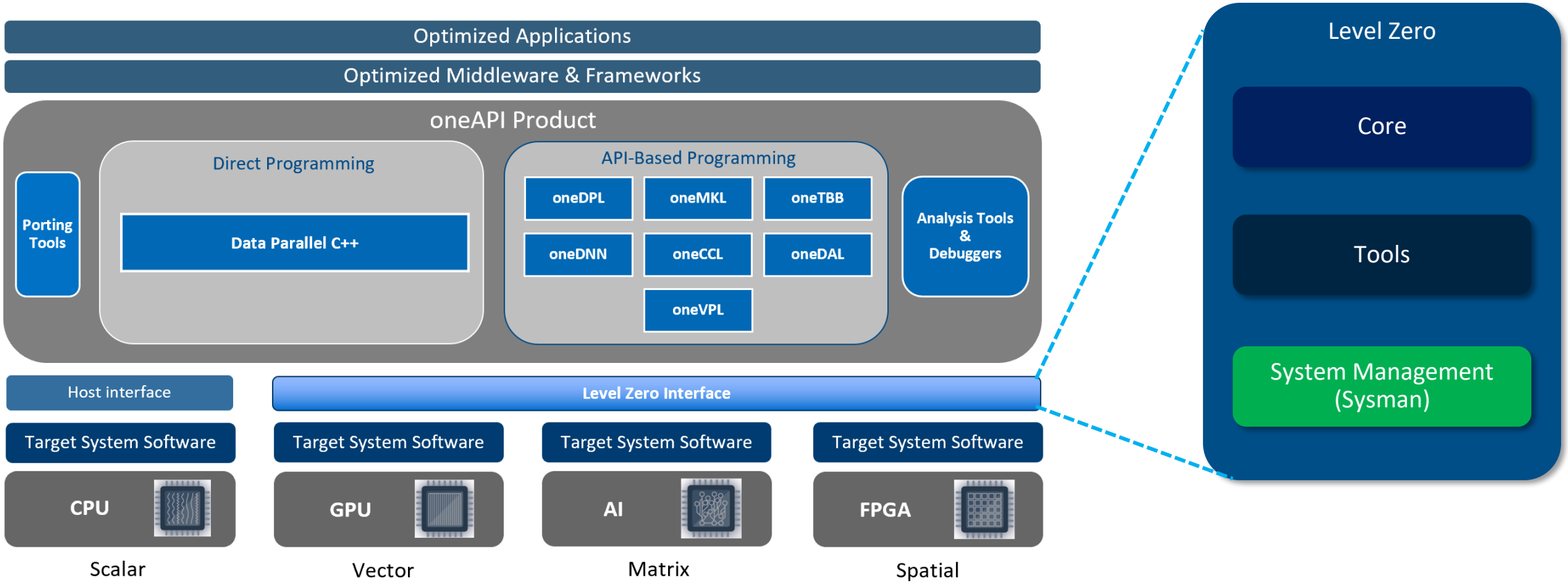
- Full open-source!
  - Specification:
    - https://spec.oneapi.io/level-zero/latest/index.html
  - Level Zero loader:
    - https://github.com/oneapi-src/level-zero
  - Level Zero Conformance Tests:
    - https://github.com/oneapi-src/level-zero-tests
  - Level Zero Implementation for Intel GPUs:
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

oneAPI and Level Zero Software Stack

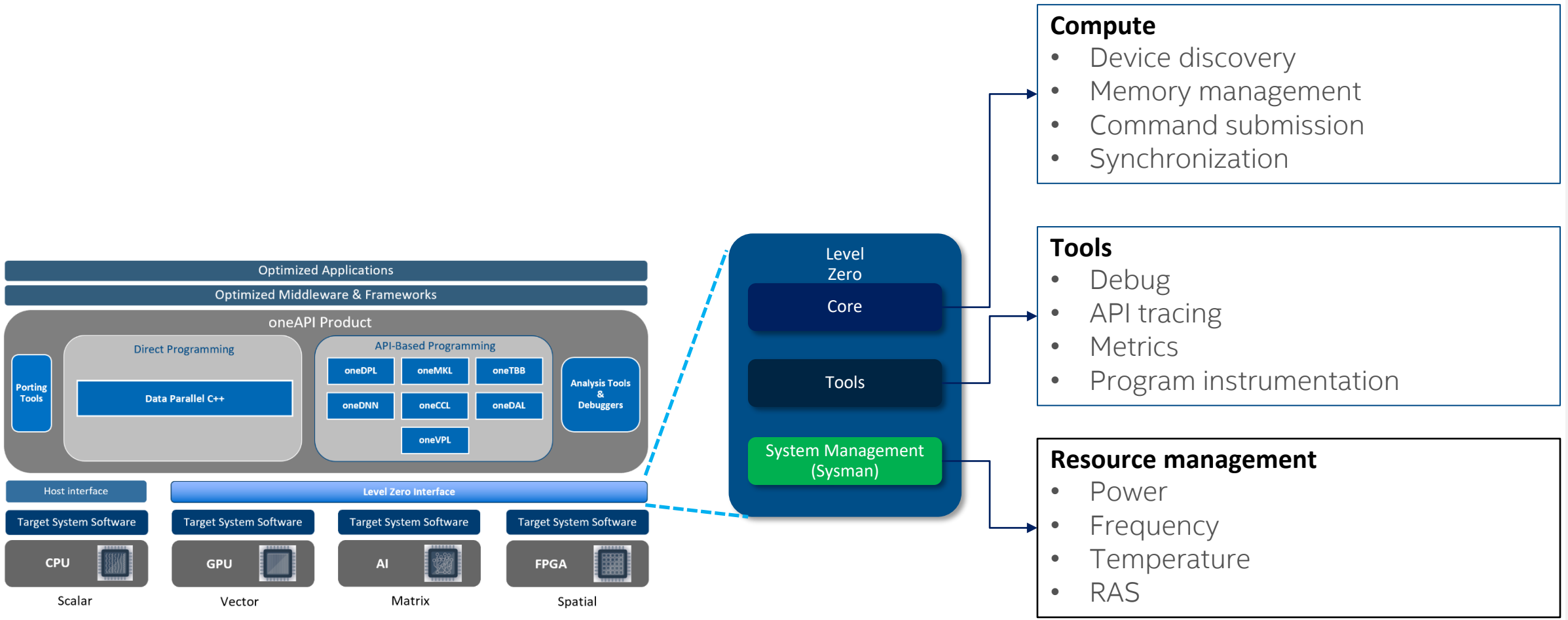


https://spec.oneapi.io/level-zero/latest/core/INTRO.html#objective

oneAPI and Level Zero Software Stack

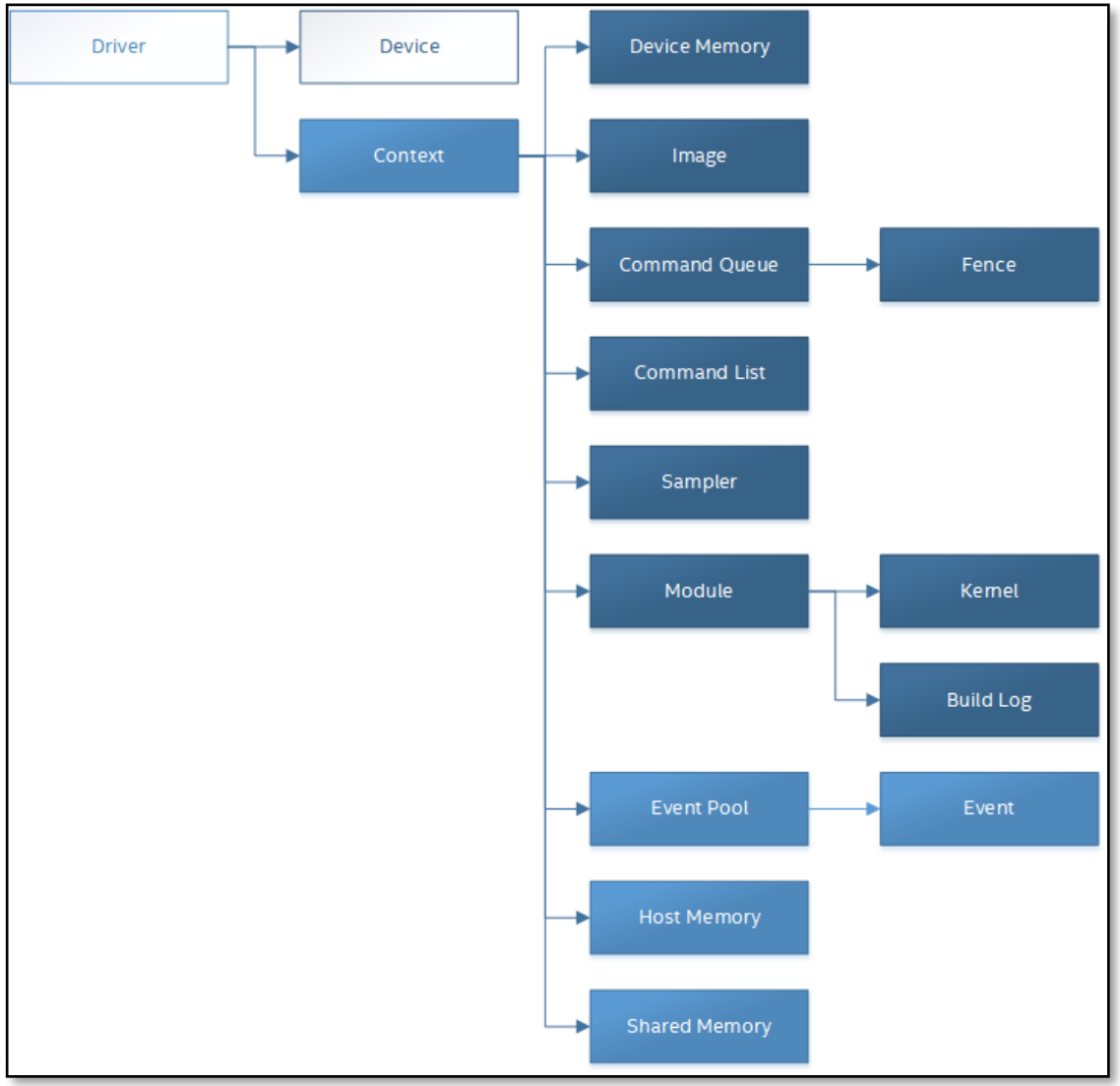


oneAPI and Level Zero Software Stack



How do we use Level Zero

- 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 to enumerate devices in system
  - Use zeDeviceGet to
    - Query number of Level Zero devices supported by driver
    - Obtain devices objects which are read-only global constructs.
  - Properties – Device, Compute, Kernel, Memory, Cache, and P2P
  - 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 is 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

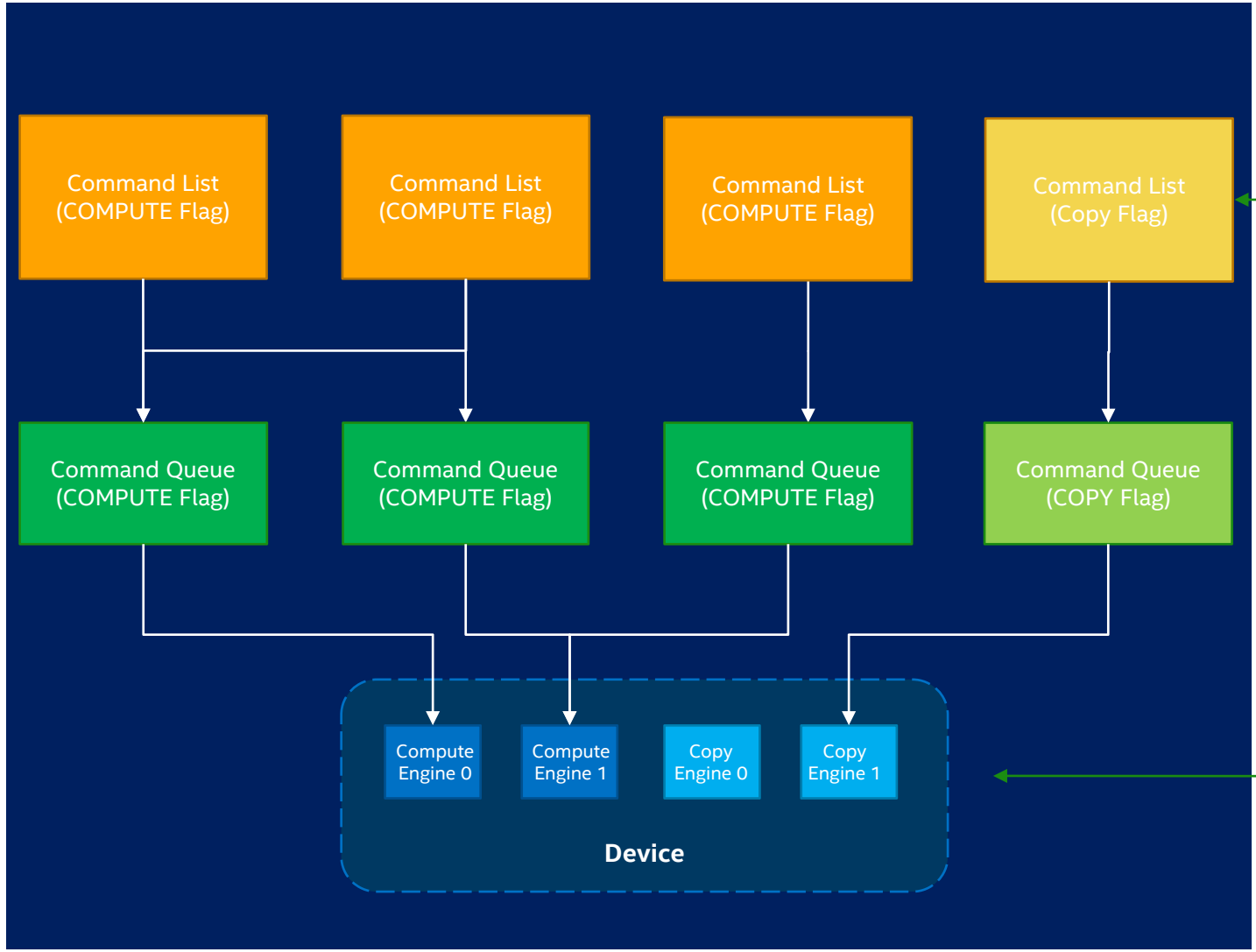
- Commands (user kernels, copies, synchronization actions) are appended to **command lists**.
  - A command list represents a sequence of commands to be executed in the accelerator
    - Can be recycled by resetting the list, without needing to create it again.
    - Can be reused, by submitting the 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());
```

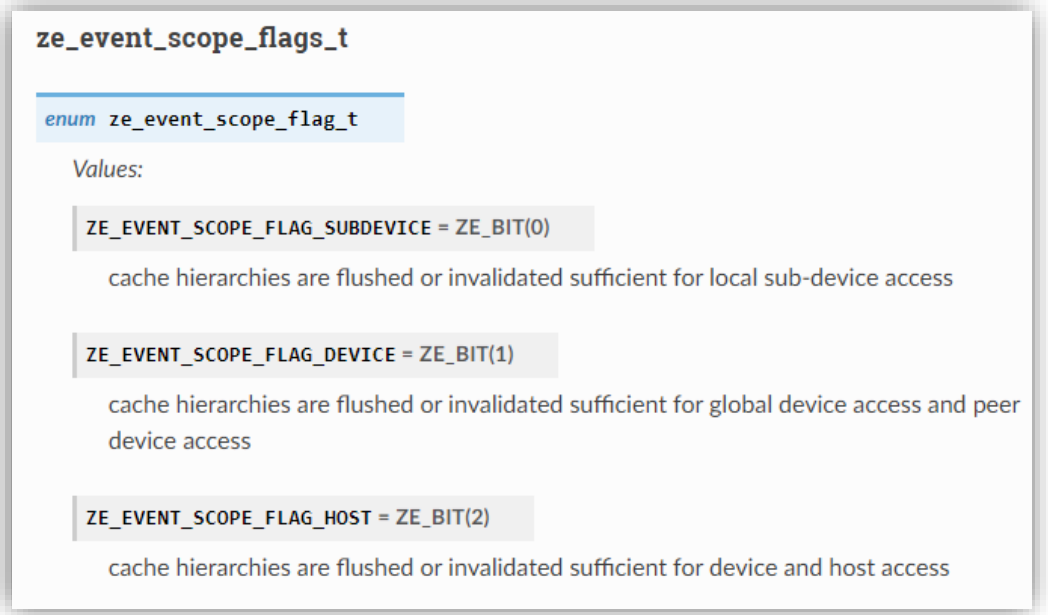
Taking Level Zero to Next Level

Advanced Features and Optimization Techniques

intel®

Optimization Techniques – Events Scope

- Events are fine-grain synchronization primitives to signal the completion of a command in a list.
- An event may:
  - Signal, indicating a command has completed
  - Be Waited upon, to wait until dependency command has completed
  - By default, apply only an execution barrier
- Events scope may be used to minimize amount of cache flushes and invalidations during signaling or waiting.



ze_event_scope_flags_t

enum `ze_event_scope_flag_t`

Values:

`ZE_EVENT_SCOPE_FLAG_SUBDEVICE` = ZE_BIT(0)

cache hierarchies are flushed or invalidated sufficient for local sub-device access

`ZE_EVENT_SCOPE_FLAG_DEVICE` = ZE_BIT(1)

cache hierarchies are flushed or invalidated sufficient for global device access and peer device access

`ZE_EVENT_SCOPE_FLAG_HOST` = ZE_BIT(2)

cache hierarchies are flushed or invalidated sufficient for device and host access

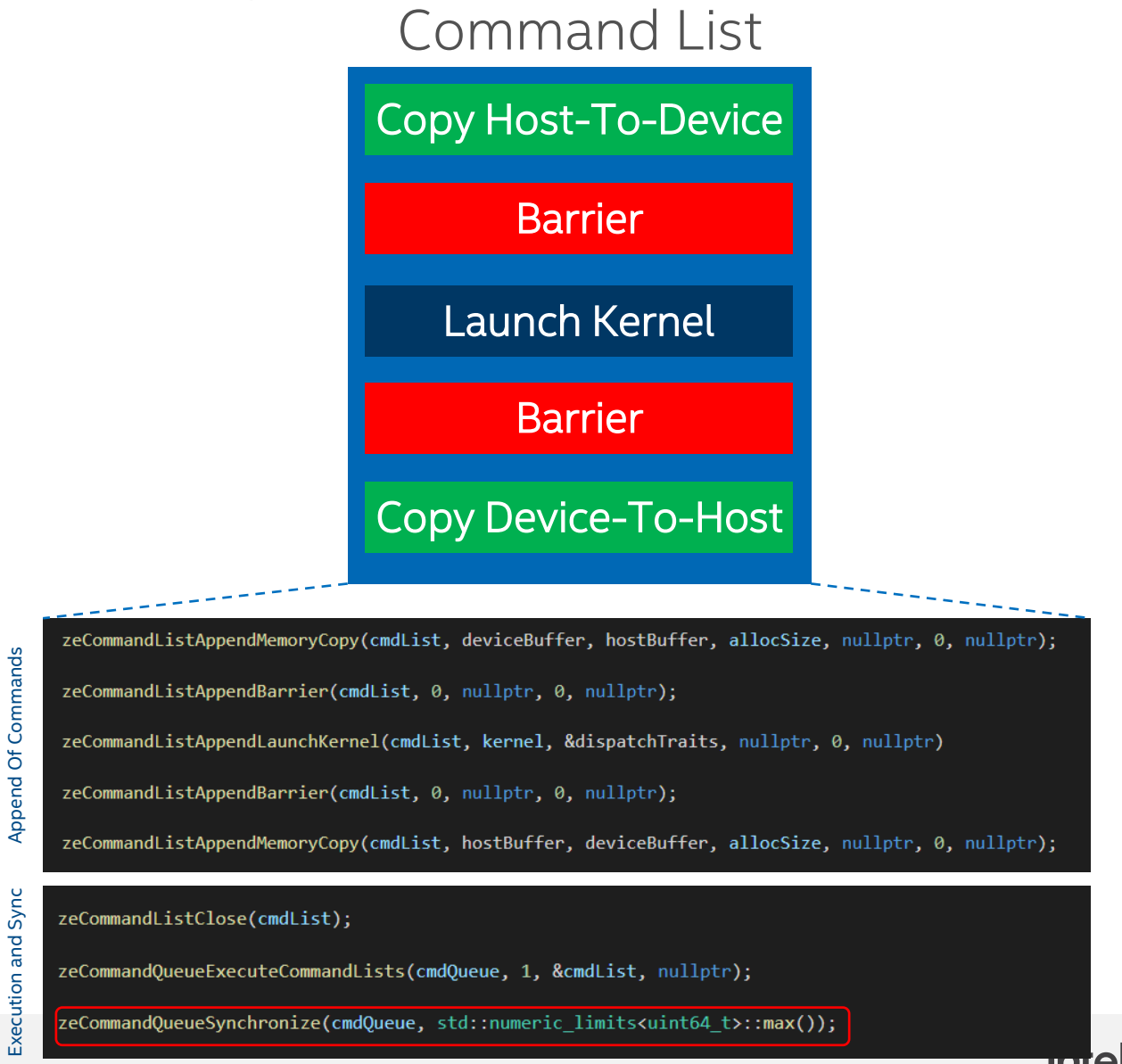
https://spec.oneapi.io/level-zero/latest/core/api.html?highlight=scope#ze-event-scope-flags-t

Optimization Techniques – Command List Reuse

- **Command List Reuse**
  - Command list may be submitted multiple time
  - Record ONCE – Replay MULTIPLE times
  - Drastically reduces host overhead (creation cost paid only once)
- **Command List Reuse requirements:**
  - Command list must have the same arguments and kernels
  - Any one-time state (like events) require reset prior to reuse
  - Command list cannot be referenced by device (must not be in use, flag to relax this requirement is WIP)

Optimization Techniques – Command List Reuse

- Let's start simple, barriers as sync points
  - Copy data from host to device
    - Use host USM memory for input data
  - Do compute operations
  - Download data back to host
    - Use host USM memory for output data
  - Whenever there is data dependency do barrier
    - It will ensure that everything prior to barrier is done



Optimization Techniques – Command List Reuse: Completion

- During submission , you may pass a fence
  - This fence is signaled when command list is done, telling applications list is ready to be reused.

zeCommandQueueExecuteCommandLists

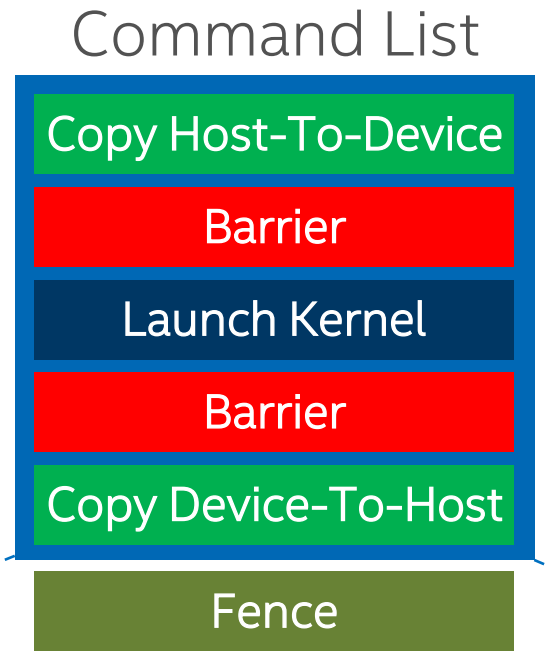
ZE_APIEXPORT ze_result_t ZE_APICALL
zeCommandQueueExecuteCommandLists(ze_command_queue_handle_t *hCommandQueue*, uint32_t *numCommandLists*, ze_command_list_handle_t **phCommandLists*, ze_fence_handle_t *hFence*)

Executes a command list in a command queue.

Parameters

- `hCommandQueue` : handle of the command queue
- `numCommandLists` : number of command lists to execute
- `phCommandLists` : [range(0, numCommandLists)] list of handles of the command lists to execute
- `hFence` : [optional] handle of the fence to signal on completion

https://spec.oneapi.io/level-zero/latest/core/api.html?highlight=executecommand#zecommandqueueexecutecommandlists



Append Of Commands

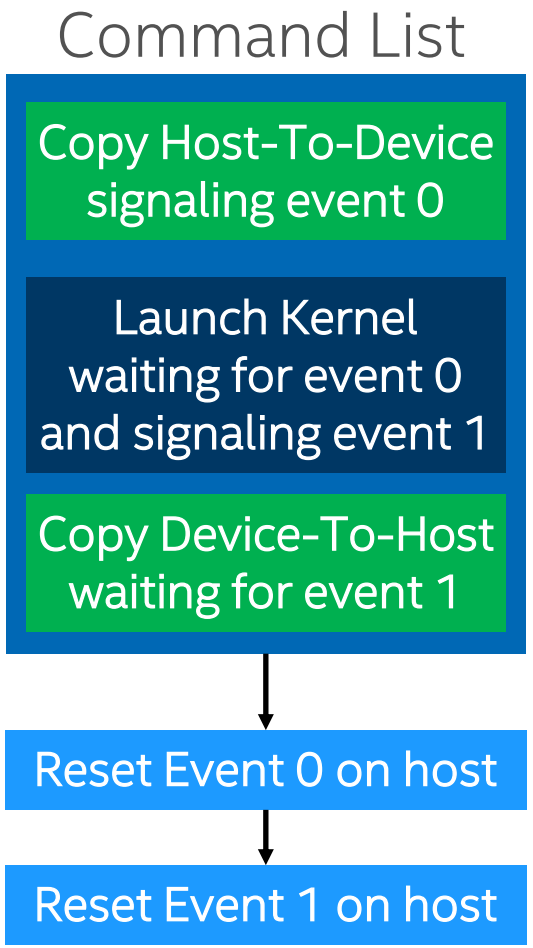
```
zeCommandListAppendMemoryCopy(cmdList, deviceBuffer, hostBuffer, allocSize, nullptr, 0, nullptr);
zeCommandListAppendBarrier(cmdList, 0, nullptr, 0, nullptr);
zeCommandListAppendLaunchKernel(cmdList, kernel, &dispatchTraits, nullptr, 0, nullptr)
zeCommandListAppendBarrier(cmdList, 0, nullptr, 0, nullptr);
zeCommandListAppendMemoryCopy(cmdList, hostBuffer, deviceBuffer, allocSize, nullptr, 0, nullptr);
```

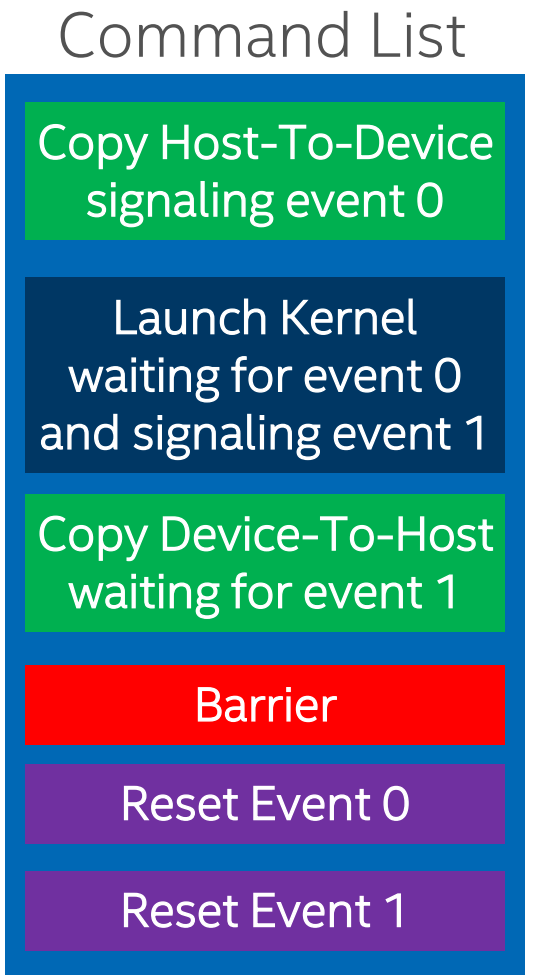
Execution and Sync

```
zeCommandListClose(cmdList);
zeCommandQueueExecuteCommandLists(cmdQueue, 1, &cmdList, fence);
zeFenceHostSynchronize(fence, timeout);
```

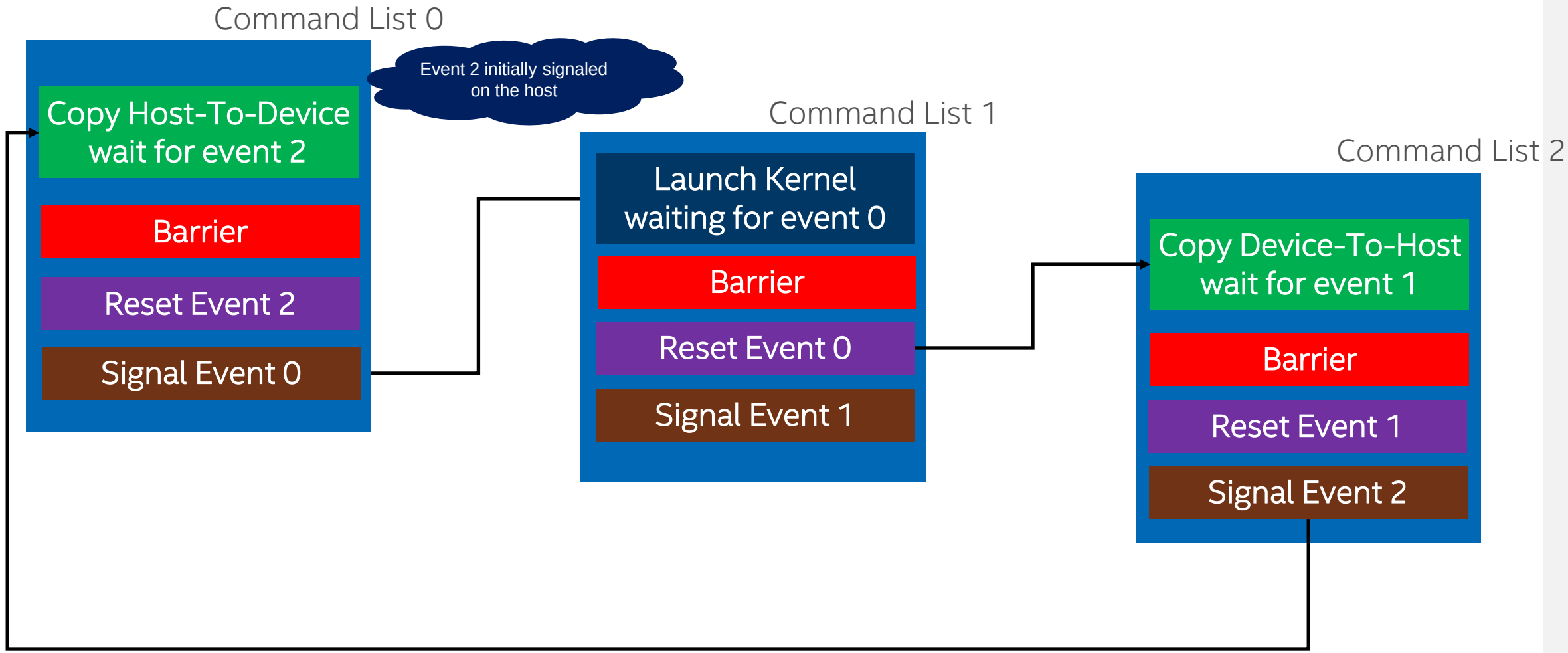
Optimization Techniques – Command List Reuse: Events

- Events can also be used in reusable command lists
  - Prior to reuse events need to be reset to non signaled state
  - Can be done on host (left figure)
    - zeEventHostReset
  - Can be done in command list itself (right figure)
    - zeCommandListAppendEventReset

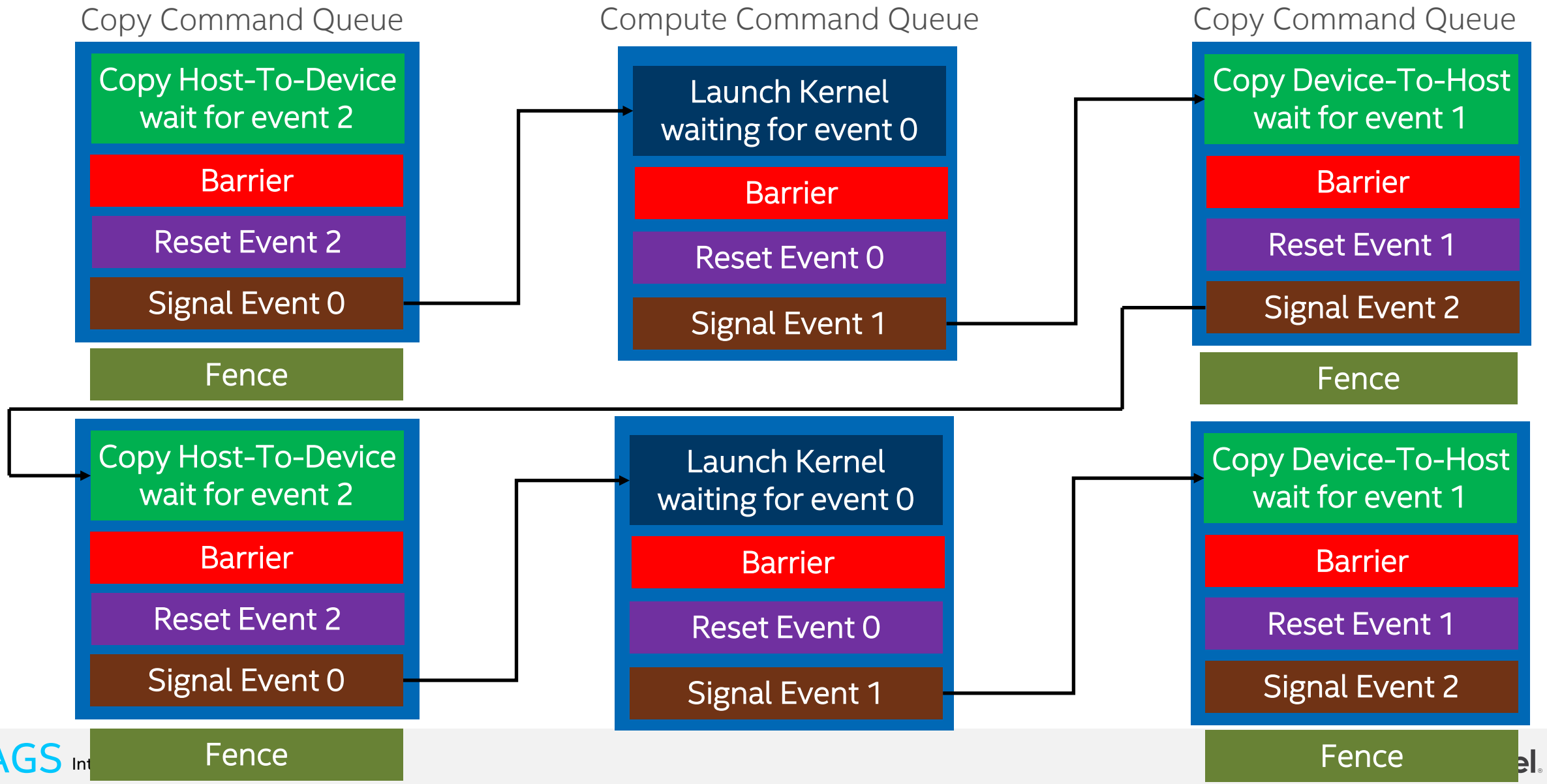




Optimization Techniques – Command List Reuse: Events



Optimization Techniques – Command List Reuse: Events



Optimization Techniques – Updating scalars via Host USM

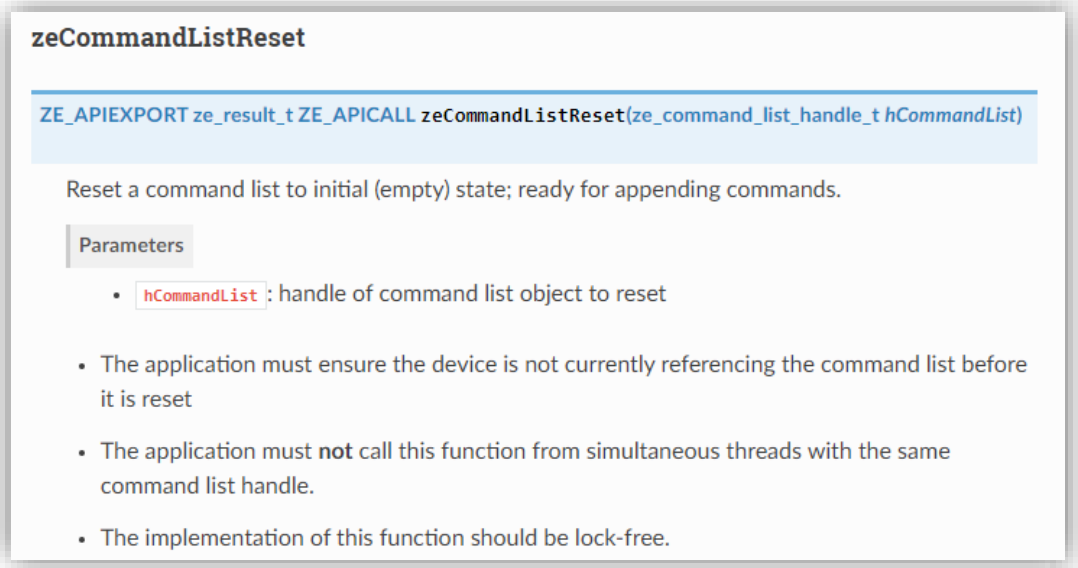
- Host USM may be used as Kernel Argument (e.g., scalarViaHostUsm in second kernel on figure on the right)
- While command list is not active, host may update the memory
- This allows to reuse command lists in more scenarios
- Caller needs to ensure that caches are invalidated prior to execution of this kernel (event wait scope with Host visibility)

```
__kernel void usefulStuff(
__global int *outBuffer, int scalar)
{
  outBuffer[get_global_id(0)] = scalar;
}
```

```
__kernel void usefulStuff2(
    __global int *outBuffer,
    __global const int* scalarViaHostUsm)
{
  int scalar = scalarViaHostUsm[0];
  outBuffer[get_global_id(0)] = scalar;
}
```

Optimization Techniques – Command List Recycle

- **Command List may be Recycled as well**
- Recycle allows to reuse internal driver resources, technique:
  1. Wait until command list is completed
  2. Reset command list -> zeCommandListReset
  3. Reprogram from scratch for new kernels
  4. Close it and submit again
- Each command list is a set of new resources. Reset:
  - Avoids creation of too many new command lists
  - Maximizes Command List Reuse & Recycle
  - Maximizes number of kernels in single command list



zeCommandListReset

ZE_APIEXPORT ze_result_t ZE_APICALL **zeCommandListReset**(ze_command_list_handle_t *hCommandList*)

Reset a command list to initial (empty) state; ready for appending commands.

Parameters

- `hCommandList`: handle of command list object to reset

- The application must ensure the device is not currently referencing the command list before it is reset
- The application must **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

https://spec.oneapi.io/level-zero/latest/core/api.html?highlight=zecommandlistreset#zecommandlistreset

Optimization Techniques – Unified Shared Memory

- Host Allocations:
  - Can be accessed directly from device
  - Host allocation doesn't migrate to local memory
  - Lower bandwidth on device than local memory, but no explicit copy needed
  - Useful for output / input buffers
  - Useful for updating scalar values for kernels during command list reuse
- Device Allocations:
  - Cannot be accessed by the HOST
  - Require explicit memory transfers, but offers high bandwidth
- Choose wisely when to use given memory allocation type
  - Host <-> Device Transfers may be done asynchronously to compute using copy engines

Optimization Techniques – Avoid non–USM pointers

- Non-USM pointers have an overhead
  - Driver needs to ensure device can access non-USM pointer
  - This is expensive process which adds to overhead
- Prefer USM pointers over malloc/new/stack pointers
  - Replace malloc / new calls with zeMemAllocHost / zeMemAllocShared
  - Replace free / delete calls with zeMemFree calls
- Reuse command lists containing non-USM pointers
  - Do not recycle / destroy them as this will trigger non-USM pointer temporary resource cleanup

Advanced Features/Optimization Techniques

- Low-Latency Immediate Command Lists
  - Object comprising both a command queue and a command list
  - Commands appended to an immediate command list are submitted for execution immediately
- Inter-Process Communication and Inter-API Memory/Image Exchange
  - Event and memory handles can be exchanged with other processes in the system to be used in their own virtual address space
  - Memory and image handles used by other APIs in the same system can be exported to/imported from them.

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.


intel