oneAPI Technical Advisory Board Meeting

December 14, 2021

Virtual Meeting

Agenda

Duration	Topics
5 minutes	Introduction
30 minutes	Unified runtime
30 minutes	Distributed programming
25 minutes	Domain specific libraries

Welcome and Thanks

- A unique opportunity to steer the parallel programming ecosystem

- A problem worth solving
  - Multi-architecture, avoiding lock-in to 1 specific hardware architecture
  - Direct and library-based programming
  - Extending existing models
  - Performant

- Your leadership, input, and feedback is critical

A good year for SYCL

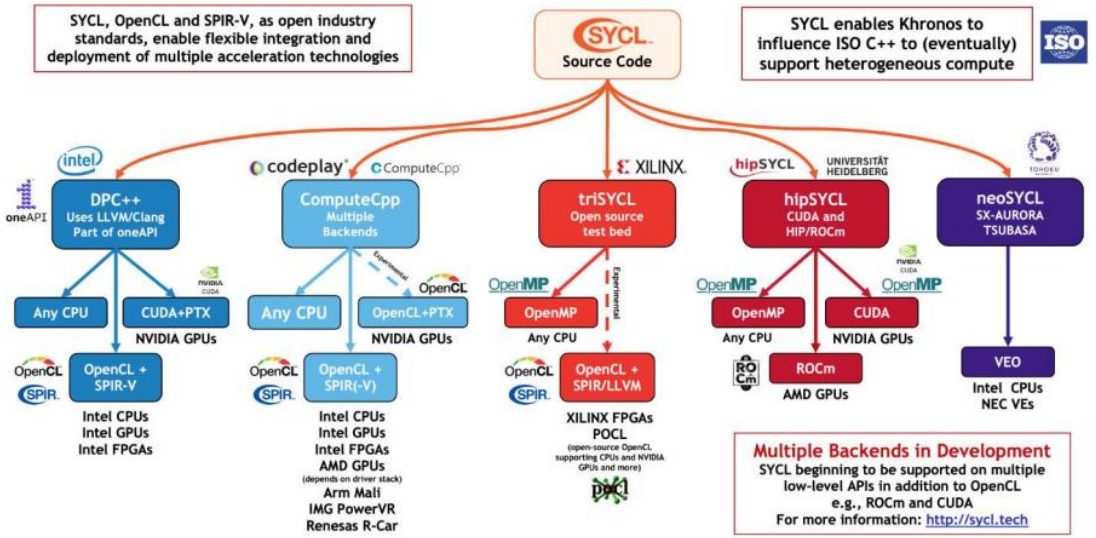
SYCL 2020 has released!

- **Feb 9, 2021:** Public release of SYCL 2020
  - Thanks to everyone from TAB that has helped to steer the direction!
  - Also thanks to the Khronos WG members (many are also in TAB), that have worked hard for years now toward this release
- Numerous DPC++ extensions are now in the core SYCL specification

DPC++ = ISO C++ + SYCL + extensions

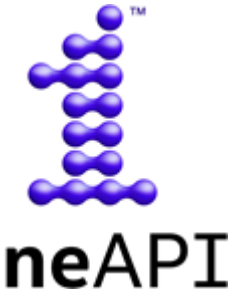
- **Future:** SYCL 2020 conformance tests to be written + released

The SYCL Ecosystem is Growing



https://www.khronos.org/sycl

Four TABS



DPC++ TAB

10/27: distributed computing
9/22: dynamic selection
8/25: level zero
7/28: property lists
5/26: invoke SIMD
4/21: oneDPL
3/24: SYCL implementation
2/24: SYCL implemenation

oneMKL tab

10/06: data fitting
7/14: sparse matrix multiply
6/16: sparse matrix multiply
5/19: random number generation
3/24: FFT
2/24: Sparse BLAS
1/27: batched Linear Algebra

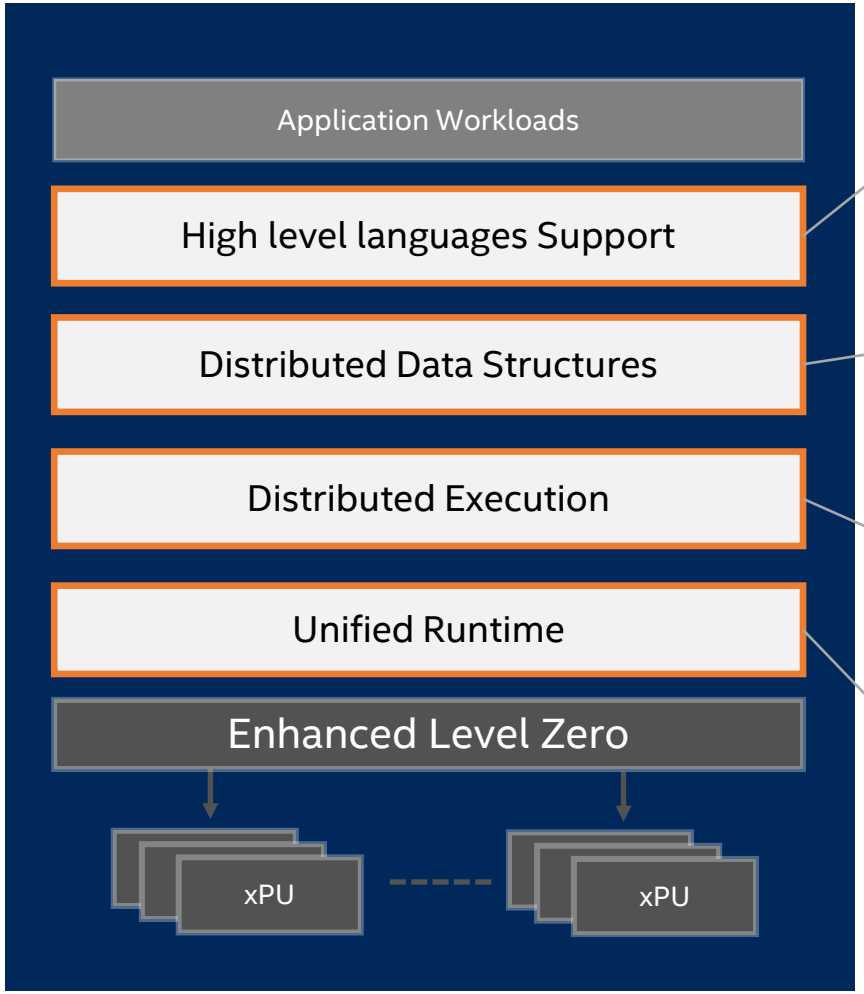
oneAI TAB

11/10: SYCL in AI (Codeplay)
8/10: DPC++/oneTBB/oneDAL
5/20: Antares (Microsoft),
Tensorflow (Google)
2/11: oneDNN

oneIPL TAB

11/29: Kickoff

Where we are headed:
distributed heterogeneous programming



Simplify Distributed Programming Specification Through High-Level Operators

Enable Distributed Programming on Application-Level Abstractions

Enable locality-aware Shared Memory programming for Distributed Computing

Unified Runtime that enables parallel SW modules to work together more effectively

Coming next year: domain specific libraries

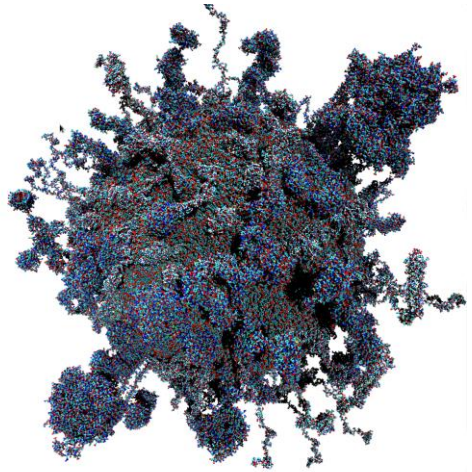
oneIPL: image processing **** new TAB formed last month ****

oneSPL: signal processing

oneDTL: data transformation

Ray tracing: **** new TAB coming in 2022 ****

Ray Tracing Specification Introduced

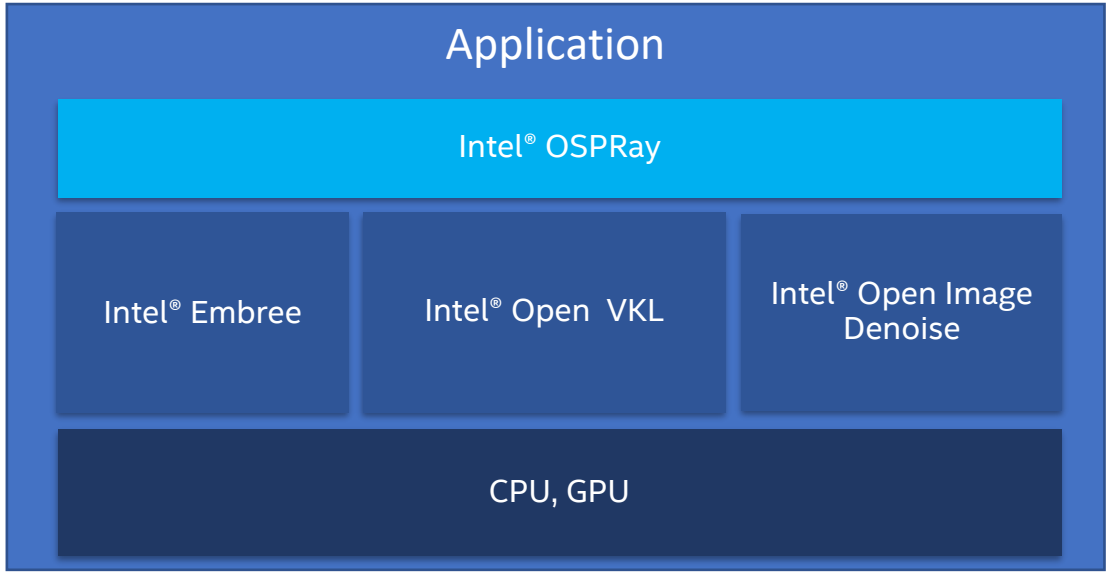


https://www.embree.org/images/gallery/ospTachyon-organelle.png

- oneAPI Specification Version 1.1
- Ray tracing and high-fidelity rendering and computation routines for use in a wide variety of 3D graphics uses including, film and television photorealistic visual effects and animation rendering, scientific visualization, high-performance computing computations, gaming, and more.
- Domains
  - Geometric ray tracing computations
  - Volumetric computation and rendering
  - Path guided ray tracing
  - Image denoising
- Execution on a wide variety of computational platforms - CPU, GPU, HPC
- TAB coming in 2022

oneAPI

Recommended Intel® Libraries



Agenda

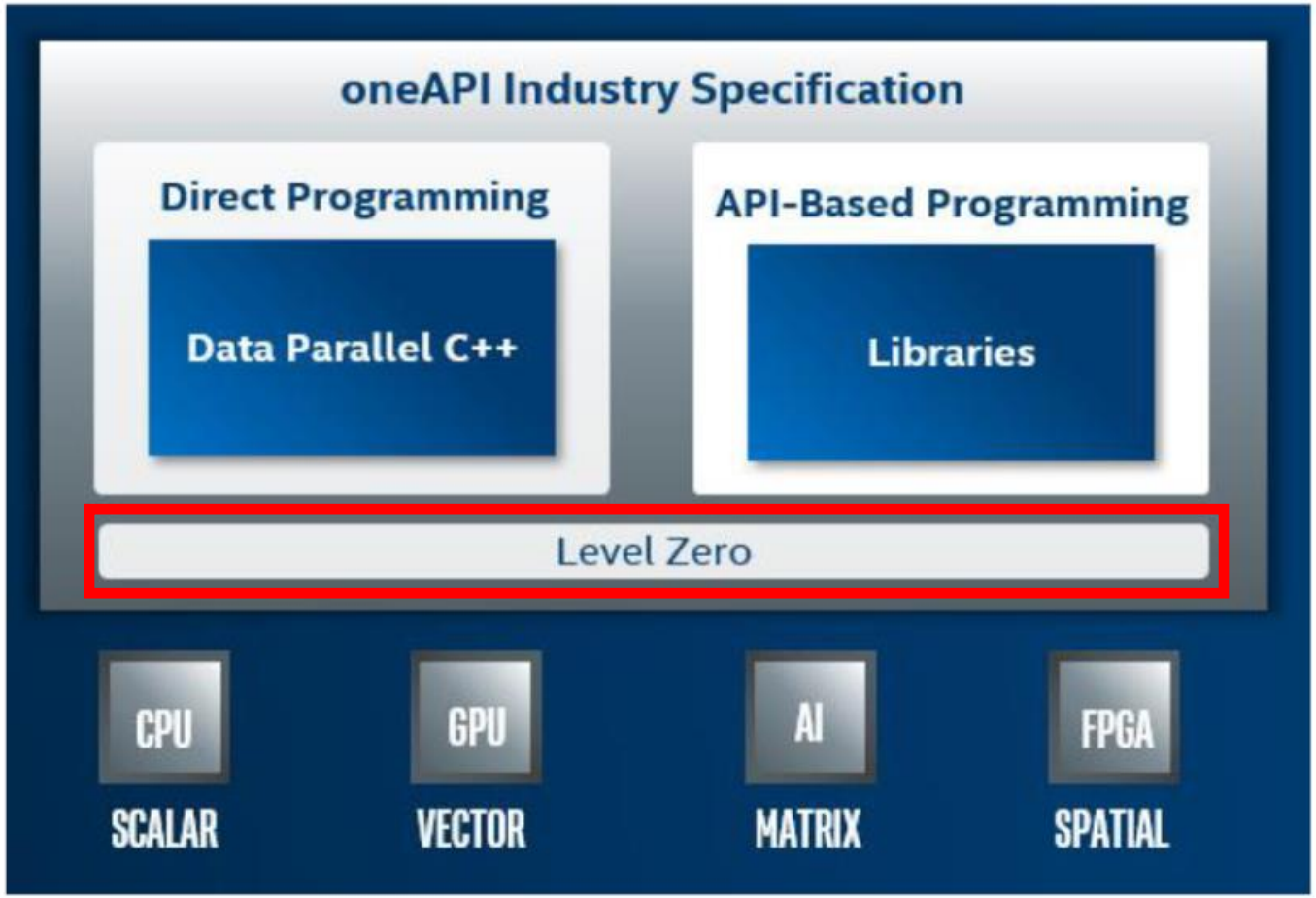
Duration	Topics
5 minutes	Introduction
30 minutes	Unified runtime
30 minutes	Distributed programming
25 minutes	Domain specific libraries

oneAPI Technical Advisory Board Meeting: **Unified Runtime**

12/14/2021

oneAPI Foundation – Level Zero

- Foundational layer to oneAPI
- Proposing new Unified Runtime API to deliver on this definition
- New runtime capabilities can support higher-level languages and libraries
- Runtime can facilitate access to variety of cross-API platforms

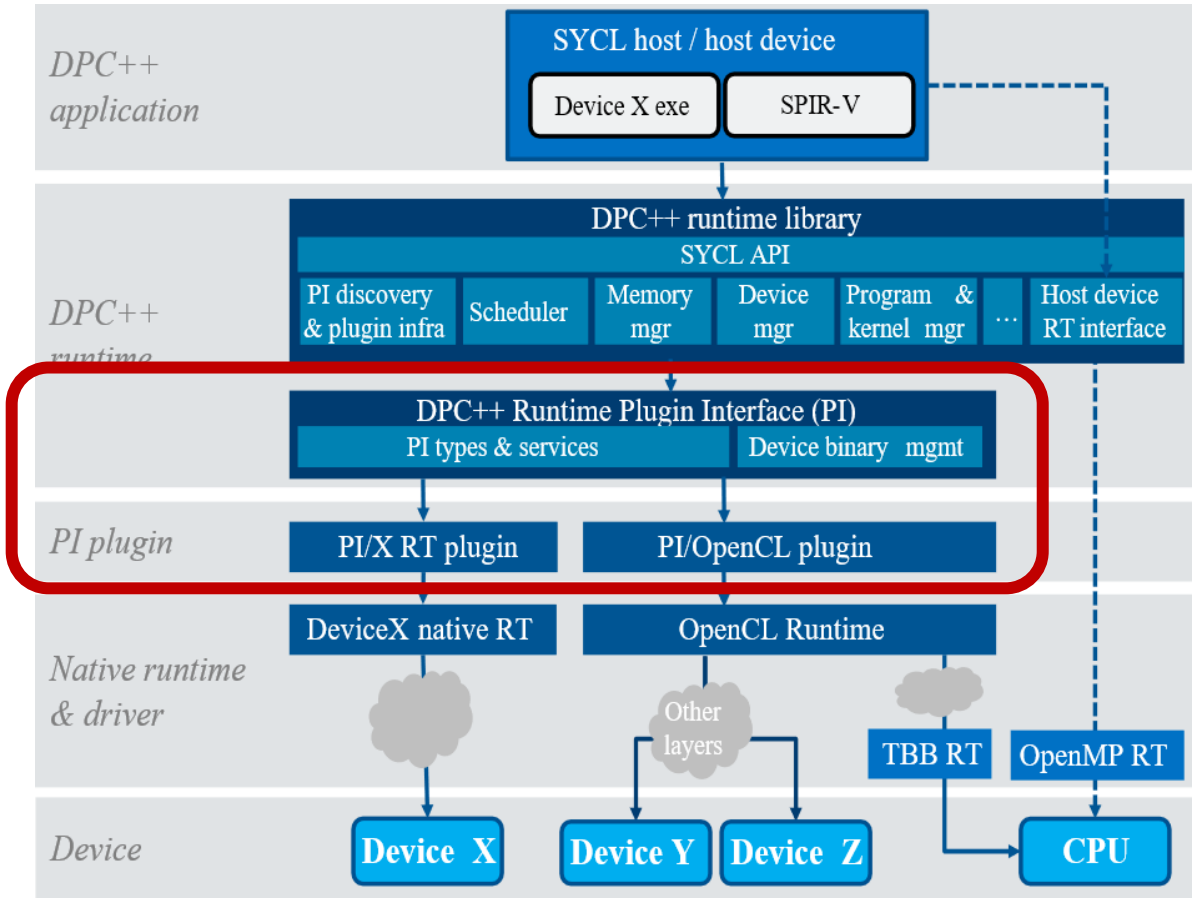


Does oneAPI need a Unified Runtime?

- Why is Level Zero as a foundation not already sufficient?
  - Level Zero provides a low-level abstraction to a device.
  - Can we provide richer constructs for cross-language and cross-API enabling?
- What are the Design principles?
  - Is it Deterministic – How would you support tuning heuristics?
  - Is it Stateless – How would you support callbacks to be registered?
- What features are missing to support your favorite X-language?
- Do we need special support for the CPU?
  - CPU device driver or resource management?
  - Leverage common components -- hwloc / memkind / numa?

Unified Runtime API

DPC++ Runtime Plugin Interface

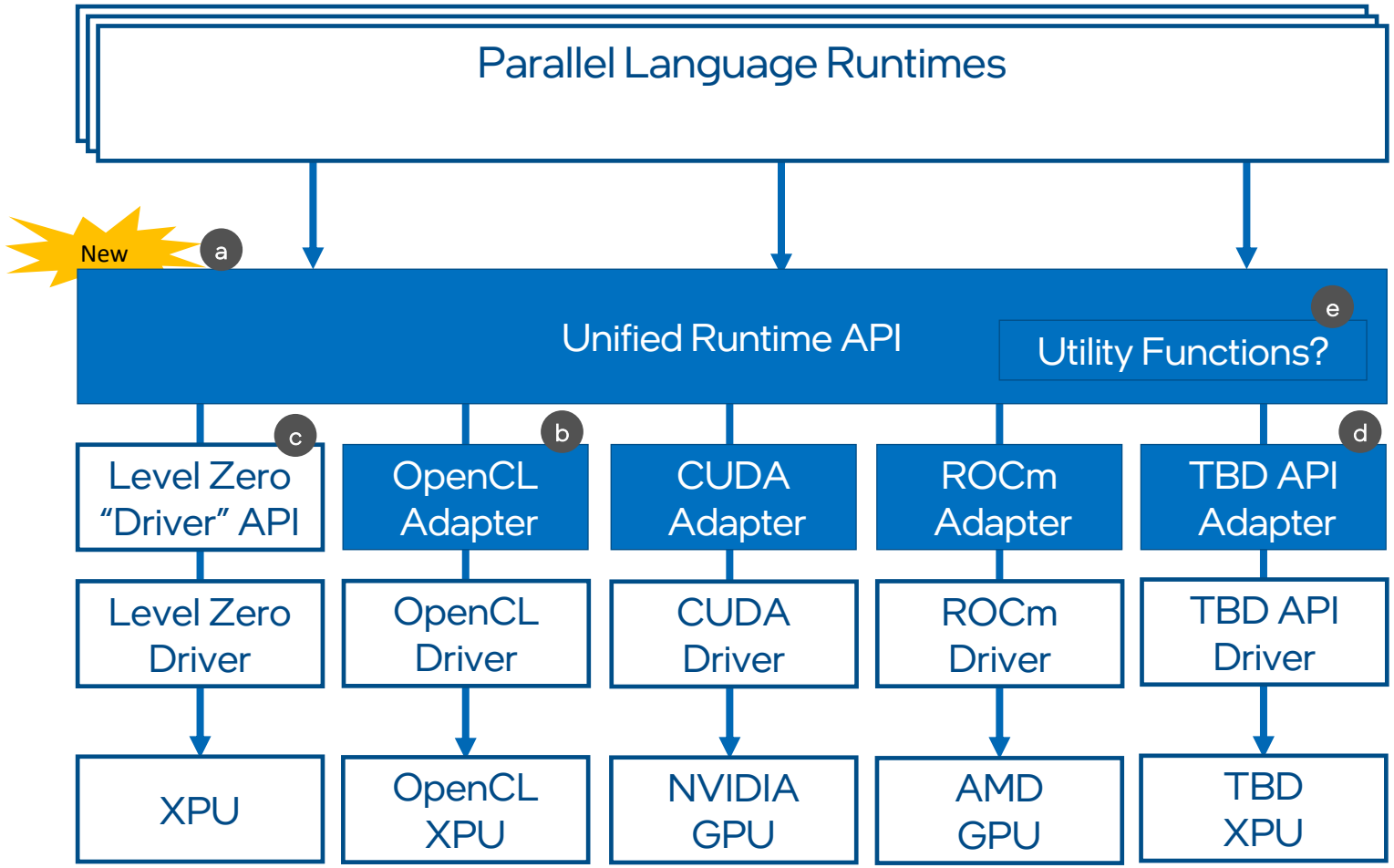


Problem Statement:

- Plugin Interface is an implementation detail
  - No formal specification
- Plugin Interface is only usable by the DPC++ Runtime
  - Other language runtimes must duplicate functionality
- Plugin Interface has deep OpenCL heritage

Diagram Source: https://github.com/intel/llvm/blob/sycl/sycl/doc/PluginInterface.md
(Note: slightly out-of-date!)

Proposal: Unified Runtime API



a. Refactor and formalize Plugin Interface into Unified Runtime APIs for use by multiple language runtimes
b. Refactor existing plugins into Unified Runtime API Adapters
c. Refactor, generalize, and modernize adapter interface
d. Enable new backends by implementing new Unified Runtime API Adapter
e. Eventually: Move common utility functionality to Unified Runtime API?

Questions for TAB: Unified Runtime API

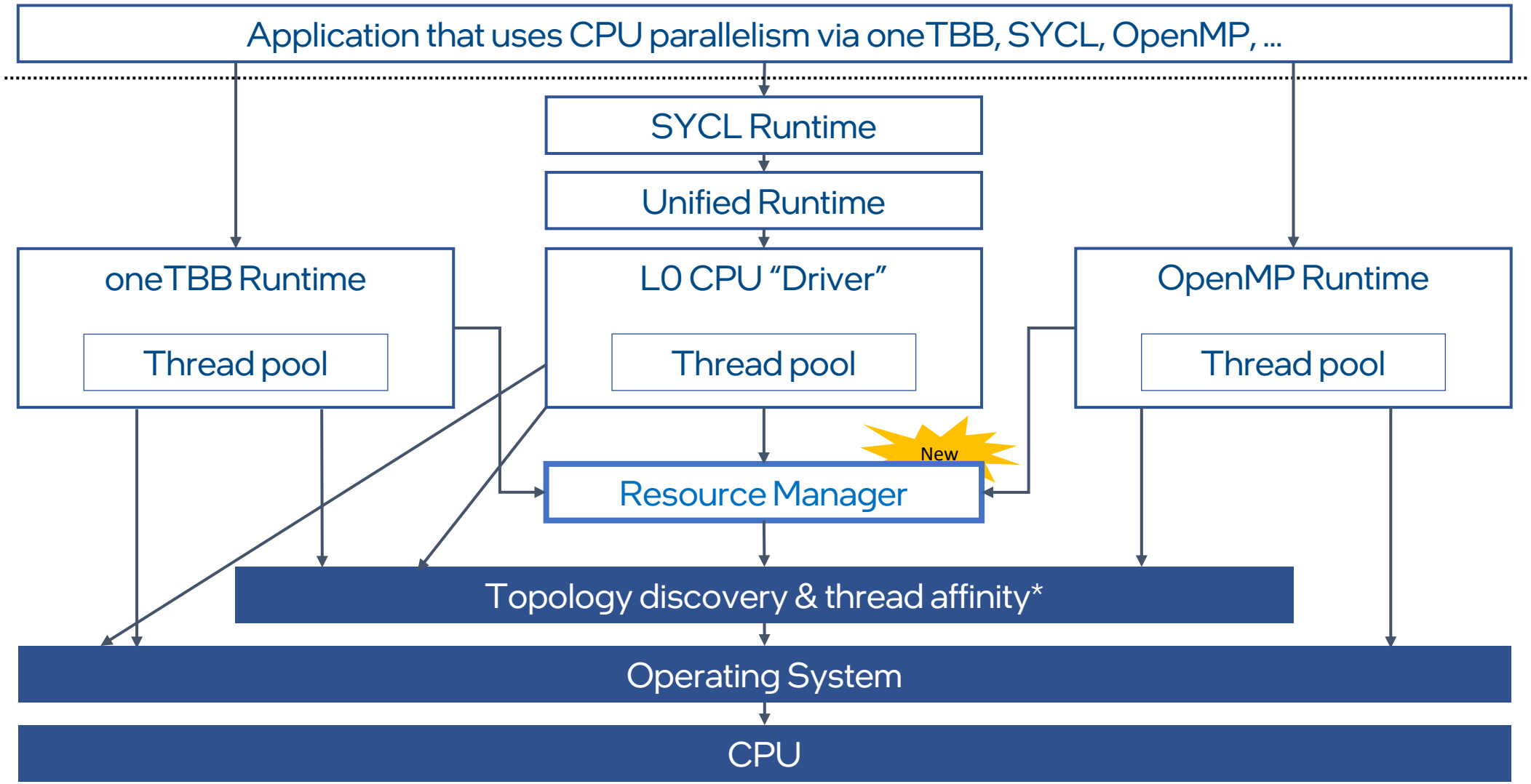
- Should this be designed as:
  - Extensions to the existing Level Zero specification?
  - A partitioning of Level Zero into “Runtime” and “Driver” sections?
  - An entirely new layer in the oneAPI software stack?
- What features would you need that are missing from Level Zero?
- Is an Adapter Interface spec needed to enable more backends?
  - Why would you add a new backend instead of using Level Zero?
- What other backend choices should consider in the design?
- What ease of use behaviors would simplify developers experience?

CPU Resource Manager API

Resource Manager Philosophy

- Goal: Enable coordination and composition of CPU usage among SYCL, oneTBB, and other runtimes that opt-in such as OpenMP or thread pools
- Manages permissions for resources, not threads
- Key characteristics:
  - A resource permit is granted to run a # of threads on a specific subset of hardware
  - Permits can be (re)negotiated for more and fewer resources
  - Support policies for space and time partitioning
- Composable resource usage is shared responsibility
  - Assumes the runtimes that opt-in are well-behaved and respect the permits
  - The runtimes decide what to ask for, how to map threads to CPUs, etc.

CPU execution path architecture



* There are existing libraries such as hwloc that support topology discovery and thread affinity. An implementation might select one of these.

Questions for TAB: CPU Resource Manager

- Should CPU resource management be addressed by oneAPI?
  - Is it enough to have a solution for a single application, not system-wide?
  - Will non-oneAPI runtimes opt-in to such a resource manager?
- Does the general approach outlined here seem reasonable?
  - What aspects or use cases you think it must not miss?
  - Should applications have a choice of CPU management policies?
- Should this be a publicly specified API, or just implementation?
  - If specified, does this CPU centric API belong to Level Zero?
  - Should we also specify a HW discovery API, or just use hwloc?

XPU Dynamic Selection API

High-Level API example

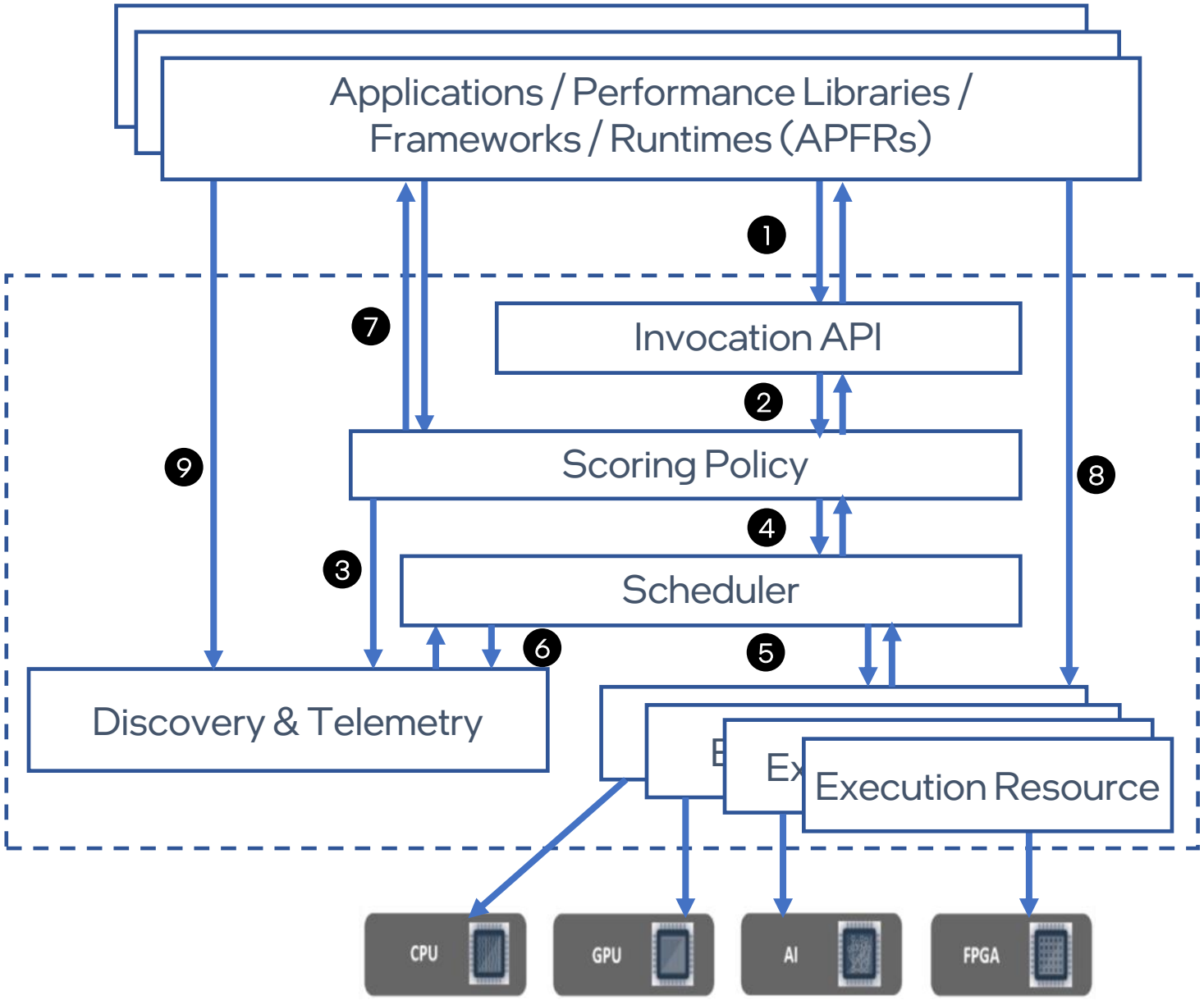
```
const int num_items = 1000000;
DoubleVector a(num_items), b(num_items), c(num_items);
initialize(a, b, c);

sycl::queue gpu_queue(sycl::gpu_selector{});
sycl::queue cpu_queue(sycl::cpu_selector{});

ds::round_robin_policy p{gpu_queue, cpu_queue};

for (int i = 0; i < 100; ++i) {
  ds::invoke_async(p, [&](sycl::queue q) {
                        return f(q,num_items,a,b,c);
                      });
}
ds::wait_for_all(p);
```

Selection policy and universe

Rotates through the universe at each invoke_async

Waits for all work to complete

Conceptual Flows of Dynamic Selection (DS)



1. APFRs submit work that is automatically assigned to an Execution Resource via the Invocation API.
2. The Invocation API uses a Scoring Policy that implements the logic for selecting an Execution Resource. There will be support for standard and custom Scoring Policies.
3. The Scoring Policy obtains information about the state of the platform and application via Discovery & Telemetry.
4. The Scoring Policy interacts with the Execution Resources via the Scheduler, which defines the Execution Resources, objects for waiting for work and a submit function. There will be support for standard and custom Schedulers.
5. The Scheduler interacts with the execution targets via an Execution Resource object.
6. The Scheduler both uses and provides support for Telemetry and Discovery functions.
7. APFRs can use Scoring Policies to select an Execution Resource if they choose to manage submissions directly.
8. APFRs can submit work directly to Execution Resources obtained via a Scoring Policy.
9. APFRs can use Discovery directly to learn about the state of the platform or application. APFRs that submit directly to Execution Resources (edge 8), may need to report specific events and metrics via the Telemetry APIs.

Questions for TAB: XPU Dynamic Selection

- How would you use dynamic selection of device/execution resource?
- Is this the right level of abstraction, or is a lower-level API better?
  - How would you see this as delivered in the Unified Runtime?
- What information is needed to make dynamic selection effective?
  - What scoring policies would be useful/applicable for your use cases?
  - Do you think useful policies can be implemented without knowing the content of the user-functor? If not, what information is needed?
- Should you invoke different functions on each device?
  - How would you prefer to express that?

Questions for TAB: What Else?

- Why is Level Zero as a foundation not already sufficient?
  - Level Zero provides a low-level abstraction to a device.
  - Can we provide richer constructs for cross-platform and cross-API enabling?
- What are the Design principles?
  - Is it Deterministic – How would you support tuning heuristics?
  - Is it Stateless – How would you support callbacks to be registered?
- What is missing to support your favorite X-language?
- Do we need special support for the CPU?
  - CPU device driver or resource management?
  - Leverage common components -- hwloc / memkind / numa?

oneAPI Distributed Programming

oneAPI Tab Meeting

Dec 14th

David Ozog, Maria Garzaran, Robert Cohn

Vision for a Distributed oneAPI Architecture

Primary Goals:

- Inherent support for SPMD libraries in oneAPI
- Portability across fabrics and GPU vendors (OFI + SYCL)
- Open standard APIs (OpenSHMEM / MPI)
- XPU-initiated communication in one-sided style
- Integration of communication with SYCL graphs
- Explore higher-level distributed programming models

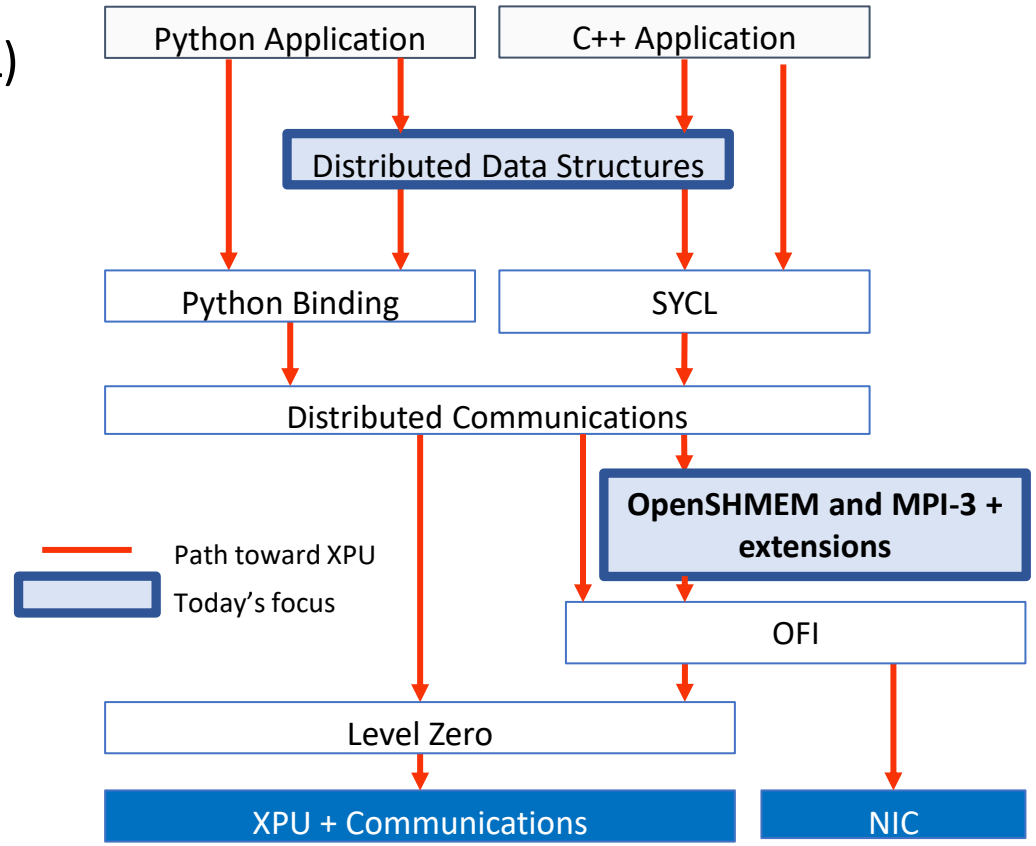
Why XPU-initiated comms:

- Better for communication/computation overlap
- Reduce GPU kernel launch overheads
- Supports strong scaling with less complexity

Why integration with SYCL graphs:

- Generate a task graph to offload to the GPU
- Communication primitives can be a node in the task graph

Software Stack:



Outline

1. XPU-initiated communication
2. Integration with SYCL graphs
3. Beyond MPI/OpenSHMEM in SYCL

XPU-Initiated Comms with OpenSHMEM (or MPI)

```
shmem_init();
int *src = (int *) shmem_malloc(array_size * sizeof(int));
int *dst = (int *) shmem_malloc(array_size * sizeof(int));

q.parallel_for( nd_range<1>{N, N}, [=]( nd_item<1>idx ) ) {
    /* Do some work on src buffer */
    do_work(src, idx, chunk_size);

    /* Pass some data to a neighboring device (in a ring fashion) */
    shmem_putmem_nbi(&dst_put[idx*chunk], &src[idx*chunk_sz], sizeof(int), (shmem_my_pe() + 1) % shmem_n_pes());
    shmem_quiet();
}); /* ... */
```

0 1 2 3

Open Questions:

0 It's simplest to restrict each process to own "symmetric" memory within a single *particular* XPU. Any other use-cases?

1 Host SHMEM and device SHMEM have different memory models. Are users ok with OpenSHMEM API extensions?

2 `shmem_quiet()` and `shmem_fence()` (even *get*) could be expensive within a kernel, e.g. Should they be supported?

3 What are the key routines for XPU-initiated subset OpenSHMEM (put, get, wait?, put with signal? Teams?)

XPU-Initiated Communication with MPI

- MPI Forum is proposing the usage of Partition Communication to support XPU-Initiated Communication
  - MPI programs retain the two-sided communication paradigm, but use one-sided communication within the kernel
  - Split Communication in two phases:
    a) Setup calls execute in the host (guarantees that buffers for send/receive data are ready)
    b) Ready calls execute in the GPU to indicate the data itself is ready to be sent

Partition Communication supports XPU-initiated Communication

Host code (CPU)

```
MPI_Psend_init(void, *buf, …  tag,
num_partitions, &request);

for (...) {
  MPI_Pstart(&request);
  kernel( …., request);
  MPI_Wait(&request);
}
MPI_Request_free(&request);
```

New call in the host. Guarantees buffer on receiver is ready

Kernel (GPU)

```
kernel(..., MPI_Request request) {
  Q.submit (… {
    int i = my_partition[my_id];
    /* Compute and fill partition i then mark ready: */
    MPI_Pready(i, request);
  })
  Q.wait();
}
```

Comm calls from the XPU kernel

Outline

1. XPU-initiated communication
2. Integration with SYCL graphs
3. Beyond MPI/OpenSHMEM in SYCL/DPC++

SHMEM/MPI calls are offloaded to the GPU scheduler

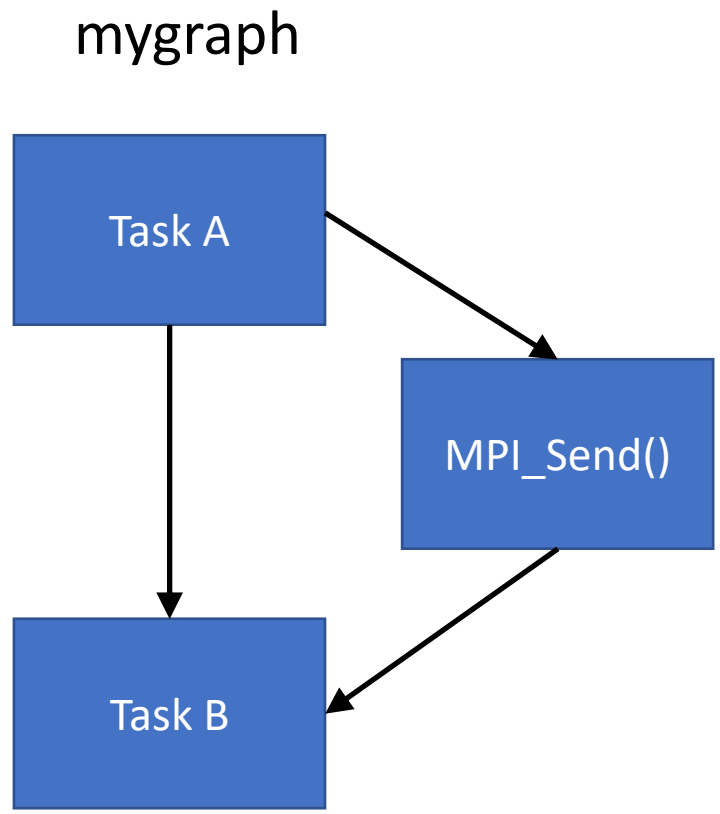
```
Task_Graph mygraph;

// Task A
auto eA = mygraph.Enqueue ({
    h.parallel_for(… );
});

// MPI_Send that depends on Task A
Auto eB = MPI_Send(…, eA);

// Task B that depends on A
auto eC = mygraph.Enqueue ( {
     depends_on(eA, eB};
     h.parallel_for(… );
});
mygraph.execute(q);
```

Enqueue a compute task in the graph

Insert a comm task in the graph. These could be MPI or SHMEM

Express dependences between tasks

Offload graph to the GPU scheduler



This is just a conceptual example, not valid SYCL code

Beyond MPI/SHMEM: Productivity and scale up/scale out

oneAPI

- What are the most promising technologies for productive scaling to multiple GPU's and/or multiple nodes?
  - Answer is different depending on type of user/application, small scale vs large scale
  - implicit vs explicit parallelism, SPMD vs controller/worker
- Language/Runtime support: Kokkos remote spaces, Berkeley Container Library, Distributed Data Structures, Legion, Charm++
- High-level languages: Python, MATLAB, Julia: distributed arrays, tasking
- Optimized primitives: MPI collectives, Xccl
- Domain specific libraries/frameworks: Spark, Horovod, Slate/Scalapack

Domain specific libraries

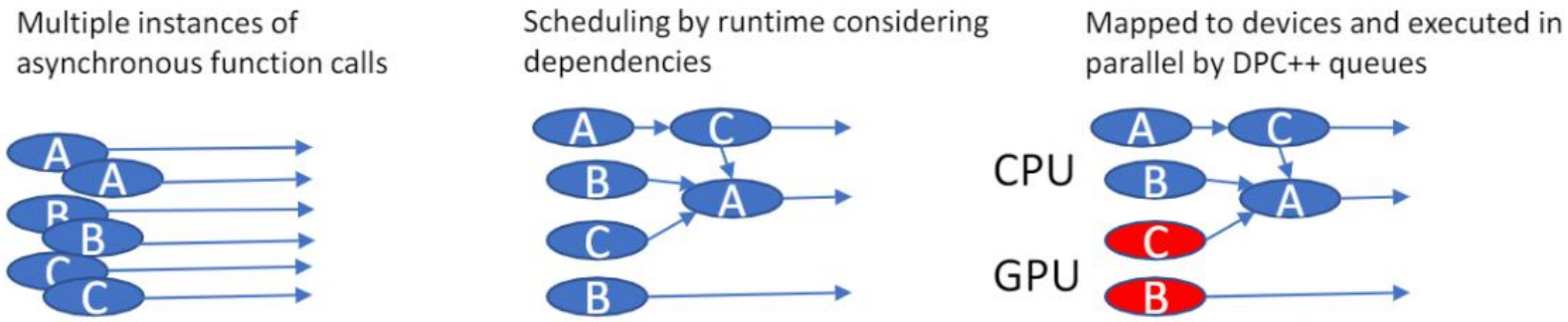
oneIPL: image processing **** new TAB formed in November 2021 ****

oneSPL: signal processing

oneDTL: data transformation

Dedicated TABs are planned for 2022

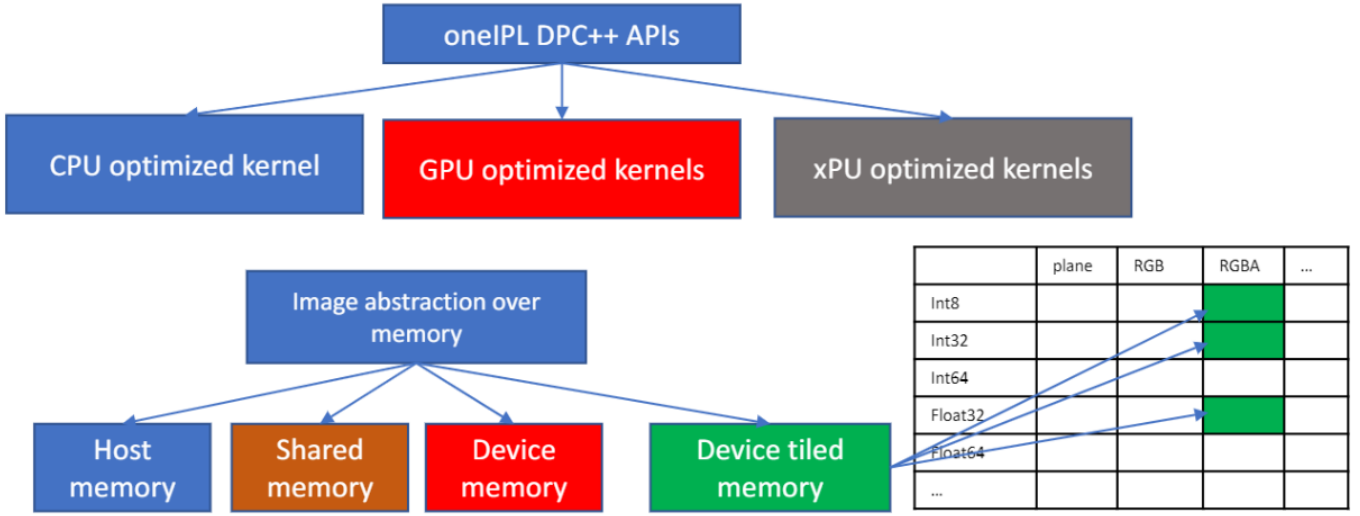
APIs are based on DPC++ and sycl::queue to be able to construct processing pipelines and include any one API calls based on DPC++ queue targeted to difference xPUs. Calls are asynchronous and scheduled by runtime.



oneIPL overview

oneIPL provides DPC++ API for image processing functionality working on XPU.

oneIPL API provides C++ abstraction over image data, which maps to the most accelerated memory available for format and data type.

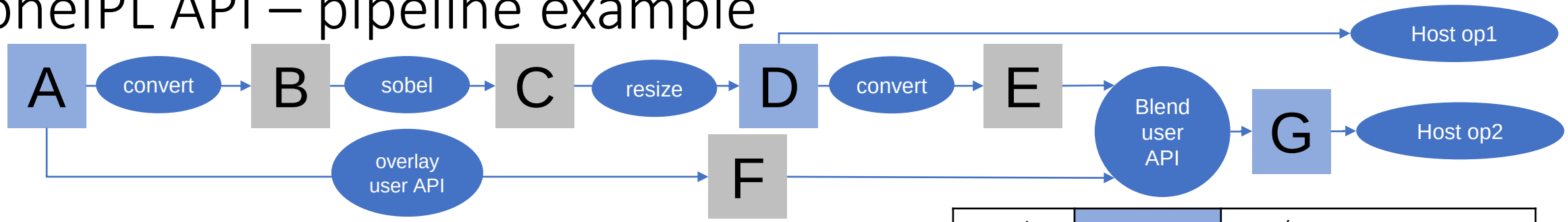


Initial API includes:

- Resize bilinear/bicubic/Lanczos/etc.
- Color conversion of RGBA-RGB images to Grayscale/NV12/etc.
- Gaussian filter
- Normalize
- Auxiliary functions

Provision Spec v0.5 is published.

oneIPL API – pipeline example



Legend		
	A	Input/output
	B	Temporary memory
	operation	Operation

```
// Creation of I/O for pipeline steps
// Source rgba image (sycl-image based) data and destination images (usm-based)
ipl::image<formats::rgba,  uint8_t> src_image{ src_image_data.get_pointer(), src_size, image_alloc }; // A (image)
ipl::image<formats::plane, uint8_t> gray_image{ src_size, device_alloc };            // B (device USM)
ipl::image<formats::plane, uint8_t> sobel_image{ src_size, device_alloc };           // C (device USM)
ipl::image<formats::plane, uint8_t> resized_image{ dst_size, shared_alloc };         // D (shared USM)
ipl::image<formats::plane, uint8_t> res_rgba_image{ dst_size,  image_alloc };        // E (image)
ipl::image<formats::rgba,  uint8_t> ovr_image{ dst_size, image_alloc };              // F (image)
ipl::image<formats::rgba,  uint8_t> dst_image{ dst_size, image_alloc };              // G (image)

// Run pipeline: convert to grayscale -> sobel -> resize_lanczos
generate_overlay(queue, src_image, ovr_image.get_usm_pointer(), dst_size);
convert(queue, src_image, gray_image);
sobel_3x3(queue, gray_image, sobel_image);
resize_lanczos(queue, sobel_image, resized_image);
convert(queue,  resized_image, res_rgba_image);
blend(queue, resized_image, ovr_image, dst_imge, {0.2});
```

Questions:

- **Priorities of related functionality**
- **Typical related use cases and pipelines**

oneSPL overview

oneSPL provides DPC++ API for signal processing functionality working on XPU.

Buffer and USM are expected to be source of 1-d input.

Initially proposed functionality includes:

- Arithmetic (specific functions)
- Vector initialization (specific functions)
- Convert functions
- Filtering functionality

Provision Spec is planned to be published in Q1'22.

oneDTL overview

oneDTL provide DPC++ API for data processing functionality initially focused on data compression.
Provision Spec v0.5 is published.

Example of OneDTL compression function usage:

```
// Create input and output objects
 data_compression::zfp_field<float, dim>             original_data{ input_buffer, dimension };
 data_compression::zfp_fixed_rate<float, dim>        mode{ 7.7 };
 data_compression::zfp_compressed_stream<float, dim> compressed_data{ dimension, mode,
                                                                      data_compression::zfp_index_type::offset};
// Run compression
 data_compression::encode(queue, original_data, compressed_data);
```