oneIPL Technical Advisory Board

Tech session #2

February 3rd, 2022

Agenda

- Introduction & open questions (10 min)
- Technical discussion (45 min)

  oneIPL Image Data Abstraction:

  1) Hardware-accelerated images and data formats
  2) oneIPL image data abstraction
  3) oneIPL image interoperability with USM
  4) Memory allocation and temporary images
- Closing words and next plans (5 min)

https://spec.oneapi.io/oneipl/latest/index.html - oneIPL specification (current version: v0.5)

The oneIPL TAB rules

DO NOT share any confidential information or trade secrets with the group

DO keep the discussion at a High Level

- Focus on the specific Agenda topics
- We are asking for feedback on features for the oneIPL specification (e.g. requirements for functionality and performance)
- We are NOT asking for the feedback on any implementation details

Please submit the feedback in writing on GitHub in accordance to Contribution Guidelines at spec.oneapi.io. This will allow Intel to further upstream your feedback to other standards bodies, including The Khronos Group SYCL specification.

Introduction of TAB members

- **Robert Schneider (PhD)**,
  Principal Key Expert
  Diagnostic Imaging
  Siemens Healthiness

- **SungShik Baik**,
  Principle Engineer, PC engineer
  Ultrasound System R&D
  Samsung Medison

- **Kangsik Kim**,
  Principle Engineer,
  Ultrasound signal processing architect
  Ultrasound System R&D
  Samsung Medison

- **Ashish Uthama**,
  Principal Software Engineer
  Image Processing
  Mathworks

- **Mark Rabotnikov**,
  Lead software engineer,
  Advanced Development group,
  Enterprise Diagnostics Informatics
  Philips

- **Tim van der Horst**,
  C++ Software Designer,
  Interventional Guided Therapy
  Systems R&D - Imaging & Image
  Processing
  Philips

- **Sohrab Amirghodsi**,
  Principal Compute Scientist
  Photoshop ART
  Adobe

- **Guoyi Zhou**,
  Head of the Medical Innovation
  Research Center
  SonoScape

- **Zhilei Zhu**,
  Xinje

- **Yizhi Li,**
  Computer Vision Software Architect
  HuaRay

- **Victor Getmanskiy**,
  oneIPL architect,
  Intel Performance Libraries,
  Intel

- **Maksim Shabunin**,
  AI Framework Engineer,
  OpenVINO Core Engineering / OpenCV,
  Intel

- **Sergey Ivanov**,
  AI Framework Engineer,
  OpenCV/G-API,
  Intel

Open question for oneIPL TAB

Accuracy across devices is different:

- The CPU/GPU devices support different IEEE754 compliance
- Standard libs has no claims on correct rounding (math library)
- The order of operations impacts the result since the algorithms also has different flows on different devices.

oneIPL spec specifies the precision of computations within supported computation datatype – ComputeT, which is a template parameter of the oneIPL functions, example:

```
template <    typename ComputeT = float,
              typename SrcImageT,
              typename DstImageT>
sycl::event resize_lanczos(
                        sycl::queue&                   queue,
                        SrcImageT&                     src,
                        DstImageT&                     dst,
                        const resize_lanczos_spec&     spec        = {},
                        const std::vector<sycl::event>& dependencies = {})
```

Request the feedback on accuracy of calculations CPU and GPU :

- Are there any specific expectations from image processing perspective?
- How important is the accuracy for different use-cases?
- Are there any criteria on the results similarity across devices?
- Are there any image similarity metrics not related to accuracy, which required to be fulfilled like PSNR?

oneIPL specification

- SYCL 2020 – based on C++17
- oneIPL primitives - class data abstractions + functional API
- API shall be compatible with SYCL 2020 compliant compiler implementation
- Current provisional spec version is 0.5, the spec v0.6 is in progress

oneIPL specification

What's new is coming in oneIPL spec v0.6:

- Replace ipl::formats -> ipl::layouts
- Image constructors changed to remove dependency on implementation
- Default image allocator shall be USM
- Methods to image auxiliary classes moved to image API
- Switched to generic template parameters
- Gaussian filter with separated sigma for x and y axis
- Normalize without sycl::buffer in spec

Hardware-accelerated images and data formats

sycl::image supported formats

```
enum class image_format {
    ~~r8g8b8a8_unorm,~~
    ~~r16g16b16a16_unorm,~~
    r8g8b8a8_sint,
    r16g16b16a16_sint,
    r32b32g32a32_sint,
    r8g8b8a8_uint,
    r16g16b16a16_uint,
    r32b32g32a32_uint,
    r16b16g16a16_sfloat,
    r32g32b32a32_sfloat,
    ~~b8g8r8a8_unorm,~~
};
```

ipl::image supported formats

Layout/ type	plane	channel3	channel4	plane3	sub420i	sub420	sub422, …
int8_t			🟩		N/A	N/A	Future extension
int16_t			🟩		N/A	N/A	
int32_t			🟩		N/A	N/A	
uint8_t			🟩				
uint16_t			🟩		N/A	N/A	
uint32_t			🟩		N/A	N/A	
half			🟩		N/A	N/A	
float			🟩		N/A	N/A	
double					N/A	N/A	
bf16,…	Future extension				N/A	N/A	

🟩 GPU hardware-accelerated (available in SYCL)

N/A Not available

oneIPL image formats supported by SYCL

Currently there is no support for sycl2020 images in compilers, so HW images are used only inside implementation and there is no option to have any public API in spec working with SYCL images.

SYCL images are under refinement, it is a high chance that sycl2020 images would remain as POC only, since it is currently very restricted

	SYCL 1.2.1	SYCL 2020
rgb -> channel3	✓	✗
bgr -> channel3	✗	✗
rgba -> channel4	✓	✓
bgra -> channel4	✓	✓
nv12 -> sub420i	✗	✗
yuv420 -> sub420	✗	✗
grayscale -> plane	✓	✗

HW images, data formats and types coverage

- Image data, layout, region of interest (ROI) are specified in **ipl::image class.** Layout, data type, and memory are defined at compile-time.
- The supported image formats and data types are defined by the matrix of combinations, each algorithm in the specification contain such matrix.
- Generic layouts is channel count – rows (1,3,4 channels). They are mapped to the formats: 1 – plane or grayscale, 3 – RGB, BGR, 4 – RGBA, BGRA, …
- Additional layouts supported selectively – 3 planes for R, G, B, subsampled YUV formats (like NV12), etc.
- Generic datatypes – 8u-32u – unsigned integer, 8s-32s – signed integer, fp16-fp64 – floating-point

Layout	8u	8s	16u	16s	32u	32s	fp16	fp32	fp64
plane									
channel3									
channel4	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
plane3									
sub420i		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
sub420		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

36 generic image formats

extra formats for selected API

oneIPL Image Data Abstraction

Important change in spec v0.6 (TBD):
Formats -> Layouts

```
template <layouts Layout, typename DataT, typename AllocatorT>
class image final;
```

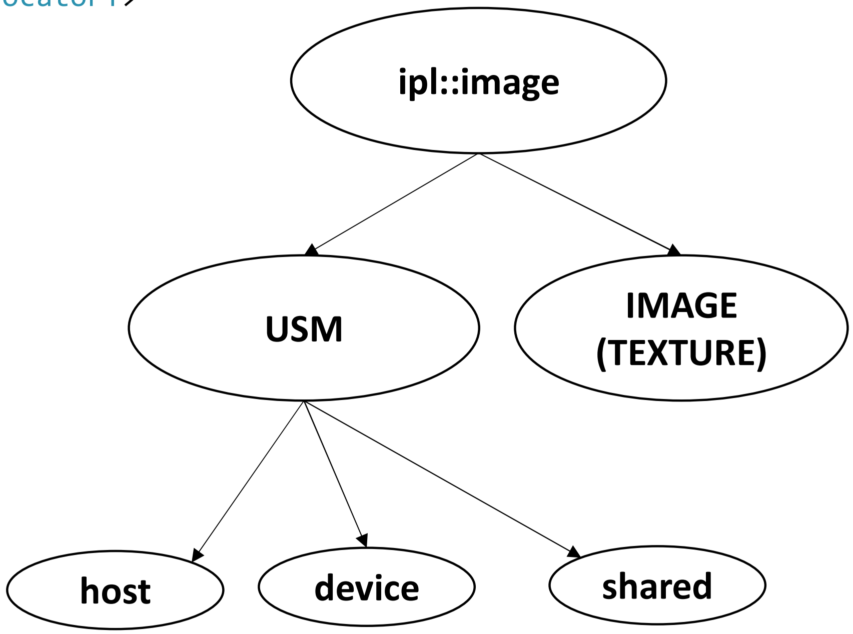
Reasons:

1. **Layout is part of compile-time dispatch and defines algorithm in kernel, multiple formats are mapped to single layout:**

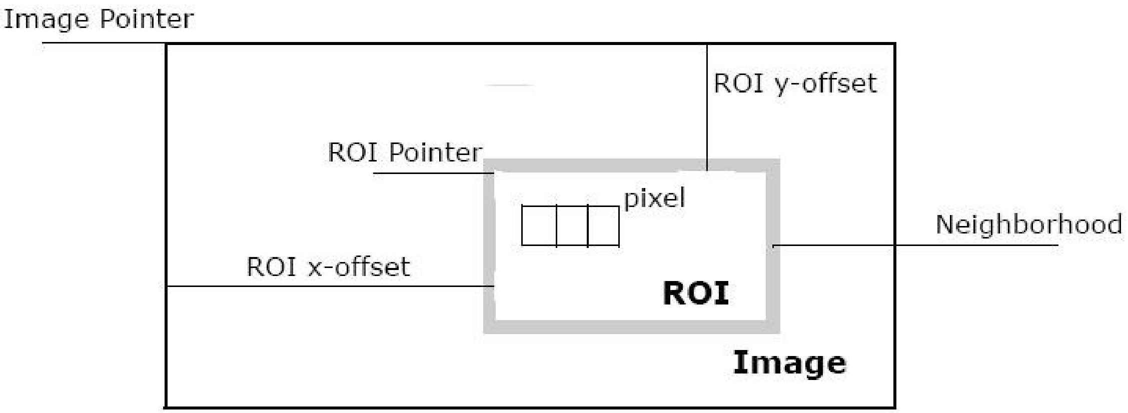
```
channel4 layout ->
rgba/bgra/argb/abgr/cmyk/… formats
```

2. Align to industry standard approach (OpenCV, python libraries, Intel® Integrated Performance Primitives/NVIDIA Performance Primitives)

Significantly affected functions:
Color conversions, which requires specific color format



Basic terminology: region of Interest (ROI)



ROI is a rectangular part of image targeted to processing.

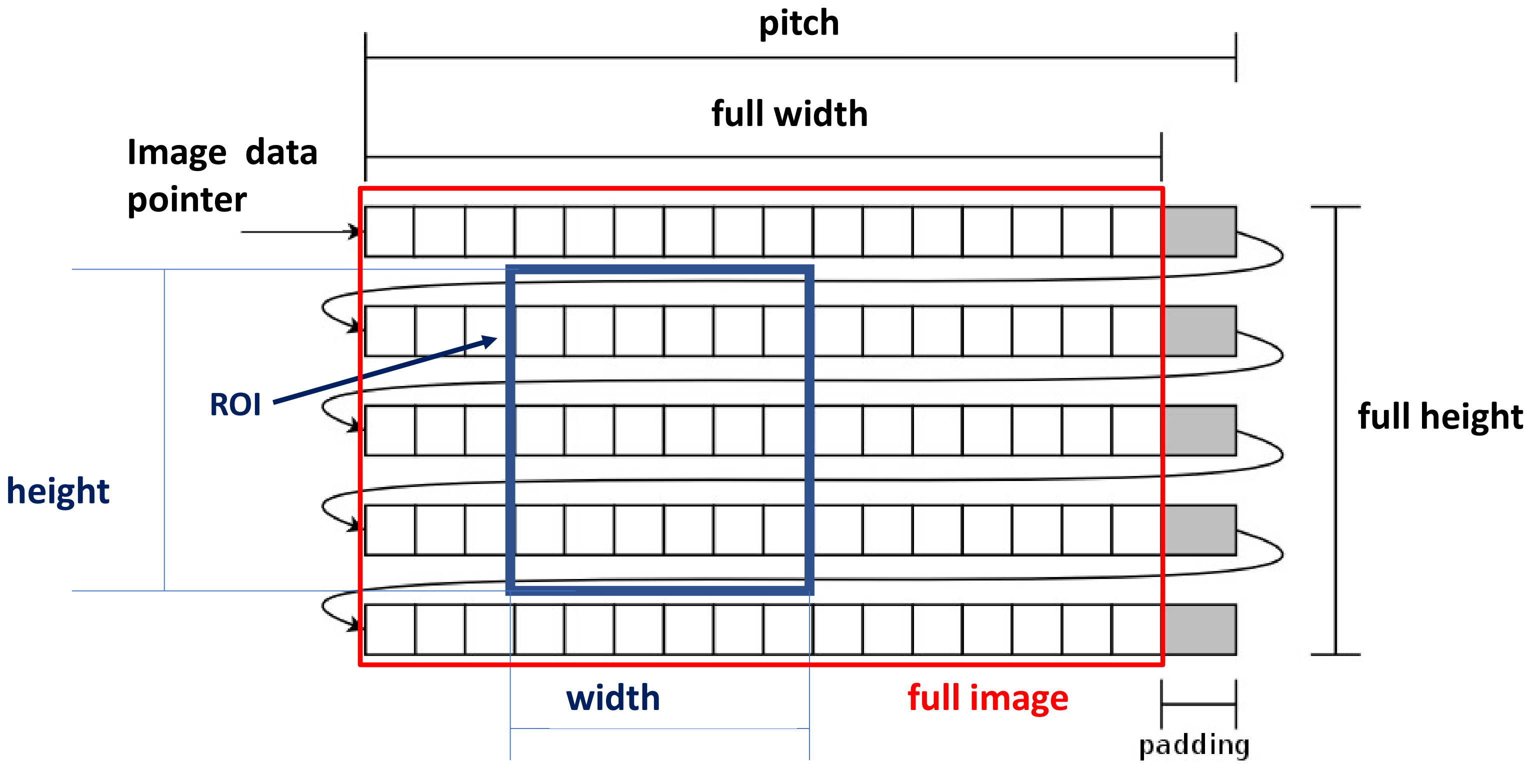
- Defined by rectangle (2d offset and size)
- Rectangle shall not intersect image
- If ROI is not specified, the image still has a ROI with zero offset equal to image size
- So the **image abstraction** is always a **ROI** placed inside some 2D data.

```
struct roi_rect {
    explicit roi_rect(const sycl::id<2>& offset, const sycl::range<2>& size);
    roi_rect(const sycl::range<2>& size);

    sycl::id<2>    offset; ///< 2D offset of ROI
    sycl::range<2> size;   ///< 2D size of ROI
};
```

Basic terminology: pitch, width, height

Image data (memory)



oneIPL Image Data Abstraction

oneapi::ipl::image class is basic data abstraction for image data. oneIPL provides single abstraction over different memory types: host, device, shared and special GPU images (textures).

```
template <layouts Layout, typename DataT, typename AllocatorT>
class image final
{
public:
// … consructors, etc.

// spec v0.5
    auto get_full_image() const->image; // previously get_whole_image
    auto get_roi(const roi_rect& roi_rect) const->image;
    std::size_t get_pitch() const;     // pitch (bytes)
    std::size_t get_size() const;      // size  (bytes)
// spec v0.6 (moved from image_helper class)
    data_t* get_pointer();                 // pointer (USM only)
    AllocatorT get_allocator() const;      // allocator
    bool is_roi() const;                   // true if ROI != image
    const roi_rect& get_roi_rect() const; // ROI rectangle
    const sycl::id<2>& get_offset() const;// ROI offset

    std::size_t get_width() const;         // width
    std::size_t get_height() const;        // height
    const sycl::range<2>& get_range() const; // range<2>{heigh, width}


    std::size_t get_full_width() const;    // full width of image data
    std::size_t get_full_height() const;   // full height of image data
    sycl::range<2> get_full_range() const; // range<2>{full_heigh, full_width}

};
```

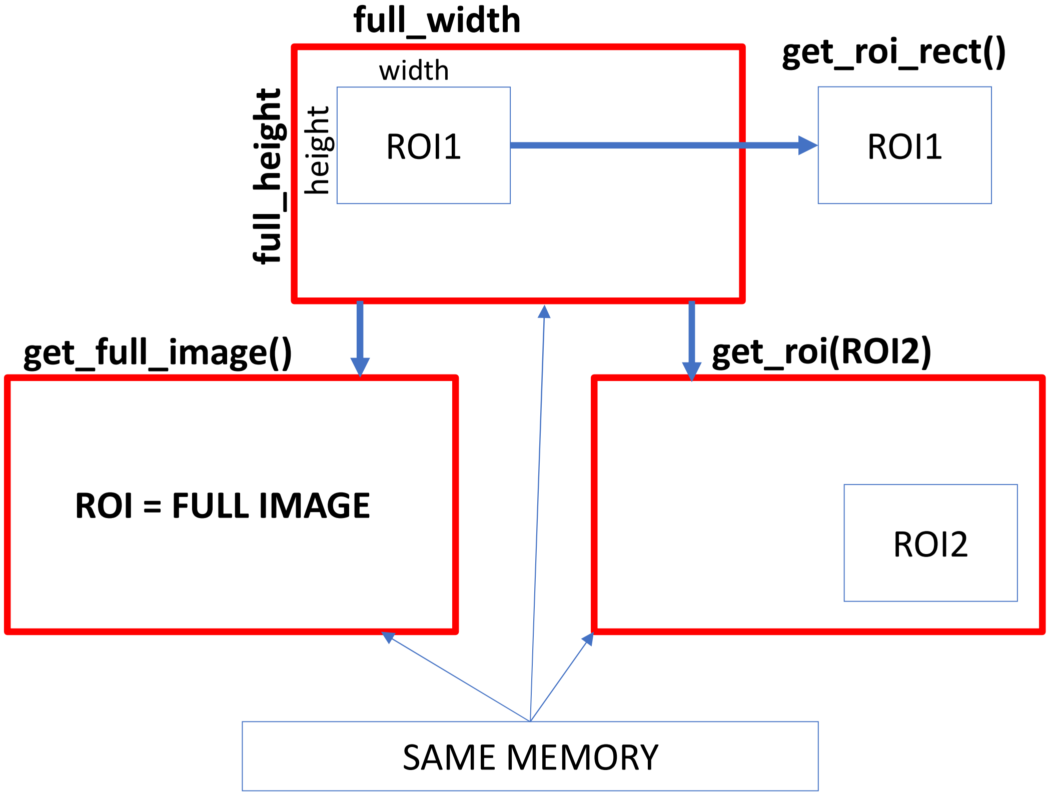


Image constructors

- The generic image constructor allow to create image object over memory specifying size, pitch and ROI.

```
explicit image(sycl::queue&                    queue,
               data_t*                         image_data,
               const sycl::range<2>&           image_size,
               std::size_t                     pitch,
               const roi_rect&                 roi_rect,
               AllocatorT                      allocator,
               const std::vector<sycl::event>& dependencies = {});
```

Image constructors (simplified)

- To simplify image creation there are overloads with no extra parameters (including overloads with **no allocator**):

```
explicit image(sycl::queue&                         queue,
               data_t*                              image_data,
               const sycl::range<2>&                image_size,
               std::size_t                          pitch,
               const std::vector<sycl::event>& dependencies = {});
```

No ROI

```
explicit image(sycl::queue&                         queue,
               data_t*                              image_data,
               const sycl::range<2>&                image_size,
               const roi_rect&                      roi_rect,
               const std::vector<sycl::event>& dependencies = {});
```

No pitch

```
explicit image(sycl::queue&                         queue,
               data_t*                              image_data,
               const sycl::range<2>&                image_size,
               const std::vector<sycl::event>& dependencies = {});
```

No pitch and ROI

Image constructors: memory behavior

- Memory is specified by pointer and queue. There are 2 options:

1. Pointer and queue are **not connected** (Pointer is host and queue is device) -> allocate and copy memory.
2. Pointer and queue are **connected** (host/shared/device/image(texture)) and queue is device -> map memory.

```
explicit image(sycl::queue&                    queue,
               data_t*                         image_data,
               const sycl::range<2>&           image_size,
               std::size_t                     pitch,
               const roi_rect&                 roi_rect,
               AllocatorT                      allocator,
               const std::vector<sycl::event>& dependencies = {});
```

Image constructors: explicit allocator

- Passed **allocator** shall be associated with same **queue**, except **image_allocator_t** which doesn't require queue and specify host allocation only. For **image_allocator_t** queue parameter is ignored.
- **Dependencies** might be required in case of async allocation and copy operations

(if provided memory is used in other kernels).

```
explicit image(sycl::queue&                      queue,
               data_t*                           image_data,
               const sycl::range<2>&             image_size,
               std::size_t                       pitch,
               const roi_rect&                   roi_rect,
               AllocatorT                        allocator,
               const std::vector<sycl::event>& dependencies = {});
```

Image constructors: implicit allocation

- To simplify the allocator/queue connection there are overloads for constructors with no allocator parameter. Allocation shall be created internally in implementation, based on **queue.**

```
explicit image(sycl::queue&                    queue,
               data_t*                         image_data,
               const sycl::range<2>&           image_size,
               std::size_t                     pitch,
               const roi_rect&                 roi_rect,
               const std::vector<sycl::event>& dependencies = {});
```

Image constructors without pointer

There are constructors with no pointer which allocates empty images:

```
explicit image(sycl::queue& queue,
               const sycl::range<2>& image_size,
               std::size_t pitch,
               AllocatorT allocator);
```

And simplified overloads without pitch and allocator:

```
explicit image(sycl::queue& queue, const sycl::range<2>& image_size, std::size_t pitch);
explicit image(sycl::queue& queue, const sycl::range<2>& image_size);
```

Mapping multiple ROI to single image

- ROI images can be created as a mapping to existing image.

```
explicit image(const image& image, const roi_rect& roi_rect);
```

Example: processing multiple ROIs detected by Machine Learning algorithm on single frame.

```
ipl::image<layouts:channel4,  uint8_t, shared_usm_allocator_t>  src_image{ size, shared_allocator };
// some function returning ROIs container
auto roi_rects = detect_kernel(src_image);
for(auto& roi_rect: roi_rects)
{
   ipl::image<layouts:channel4,  uint8_t,  shared_usm_allocator_t> roi{ src_image, roi_rect };
   process_kernel(roi);

   // it can be done in simpler way via image method:
   process_kernel(src_image.get_roi(roi_rect));
}
```

oneIPL image and USM interoperability

- USM –> ipl::image: ipl::image constructed based on USM pointer either contains

```
explicit image(sycl::queue&                    queue,
               data_t*                         usm_pointer,
               const sycl::range<2>&           image_size,
               const std::vector<sycl::event>& dependencies = {});
```

- ipl::image –> USM: USM pointer can be returned via accessor, which works only for images based on USM:

```
data_t* get_pointer() const;
```

Example: write custom kernel using USM

```
sycl::event copy_kernel(image<layouts::plane, std::uint8_t>& src,
                        image<layouts::plane, std::uint8_t>& dst) {
  return queue.submit([&] (sycl::handler& cgh) {
      const auto* p_src = src.get_pointer();
      auto* p_dst = dst.get_pointer();
      const auto src_offset = src.get_offset();
      const auto dst_offset = dst.get_offset();
      cgh.parallel_for<class kernel>(image.get_range(),
        [=] (sycl::item<2> item) {
            sycl::id<2> id = it.get_global_id(); // coordinates relative to ROI
            src_pixel = src.get_pitch() * (src_offset[1] + id[1]) + src_offset[0] + id[0];
            dst_pixel = dst.get_pitch() * (dst_offset[1] + id[1]) + dst_offset[0] + id[0];
            p_dst[dst_pixel] = p_src[src_pixel];
        }
    }
}
```

Range is the same, but image data configuration is different (offset and pitch might be different for the same size).

oneIPL provide accessors for simplification of such access patterns and to avoid using direct address arithmetic

Next Steps

- All materials and minutes of meetings will be published on GitHub and will be available for the offline review (the offline feedback of invited TAB members will be also processed and discussed on next TAB meeting)
- The next technical discussion: February 17th
  - **Proposed new time: 7:30 AM PST**

Find more on https://spec.oneapi.io/versions/latest/introduction.html#contribution-guidelines
https://github.com/oneapi-src/oneAPI-tab

oneIPL Technical Advisory Board meetings

The goal is to provide the feedback and define future development of the specification.

First topics planned to discuss are at the table below, but it might be adjusted later.

Topic	Plan	Date
1) oneIPL overview	1) Programming model 2) Execution model 3) Image processing pipelines 4) Image data abstraction 5) Memory model	December 16th, 2021
2) oneIPL Image data abstraction	1) HW images and data formats and types coverage 2) IPL image data abstraction 3) Interoperability with USM 4) Memory allocation and temporary images	February 3rd, 2022
3) oneIPL Library design details	1) Domains 2) Reference code and optimized backends 3) Error handling mechanism 4) Interoperability with other oneAPI libraries	February 17th, 2022
4) oneIPL Functions overview	1) ML oriented APIs for image preprocessing 2) Data type support in the functions 3) Color formats and conversions	March 3rd, 2022

Resources

- https://www.oneapi.io/spec/ - oneAPI Specification
- https://spec.oneapi.io/oneipl/latest/index.html - oneIPL specification (current version: v0.5)
- https://github.com/oneapi-src/oneAPI-tab - GitHub with oneAPI TAB materials
- https://spec.oneapi.io/versions/latest/introduction.html#contribution-guidelines - oneAPI Specification contribution guidelines