Image Special Interest Group

Session 1
June 21, 2023

The Image SIG discussion 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 per 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.

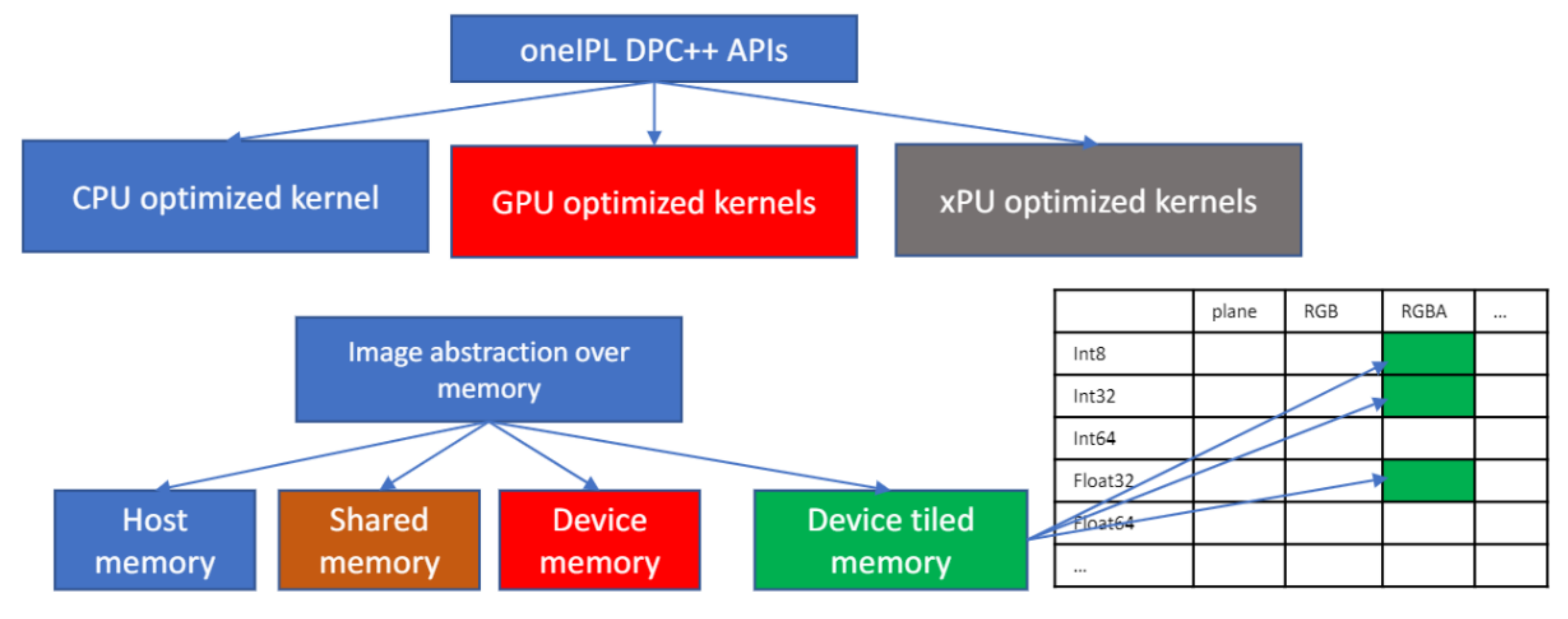
oneIPL programming language

- SYCL 2020 – based on C++17
- oneIPL primitives – classes for data abstractions + functional API
- API shall be compatible with SYCL 2020 compliant compiler implementation

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 the following image processing features:

- Resize bilinear/bicubic/Lanczos/etc.
- Color conversions of RGBA-RGB images to Grayscale/NV12/etc.
- 2D filters (Gaussian, Bilinear, Median, Convolutional, Separable)
- Normalization
- Auxiliary functions

[Provision Spec v0.6 is published.]

oneIPL Spec and Domains

oneIPL Spec v0.6 | oneIPL Spec v0.7 | Future updates

Basics
- Image/Accessors
- Allocators
- Type conversions
- Batch processing
- Arithmetic

Color conversions
- RGB(A) ⟷ NV12
- RGB(A) ⟷ RGBP
- RGB(A) ⟷ GRAY
- RGBA ⟷ RGB
- Other formats

Filters
- Sobel
- Gaussian
- Bilateral
- Median
- Other filters

Geometry
- Resize
- Mirror
- Warp affine
- Warp perspective
- Other transforms

Batch and 3D
- Batch resize and mirror
- Batch transforms
- Batch convert
- Batch filters
- 3D operations

oneIPL – oneAPI interface for image processing (providing Intel® IPP features)

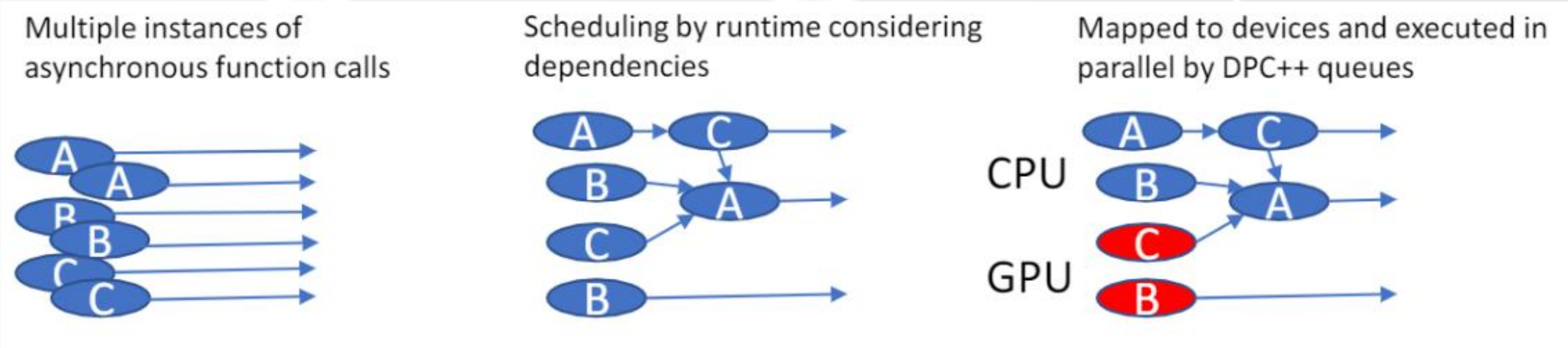
sycl::queue

images

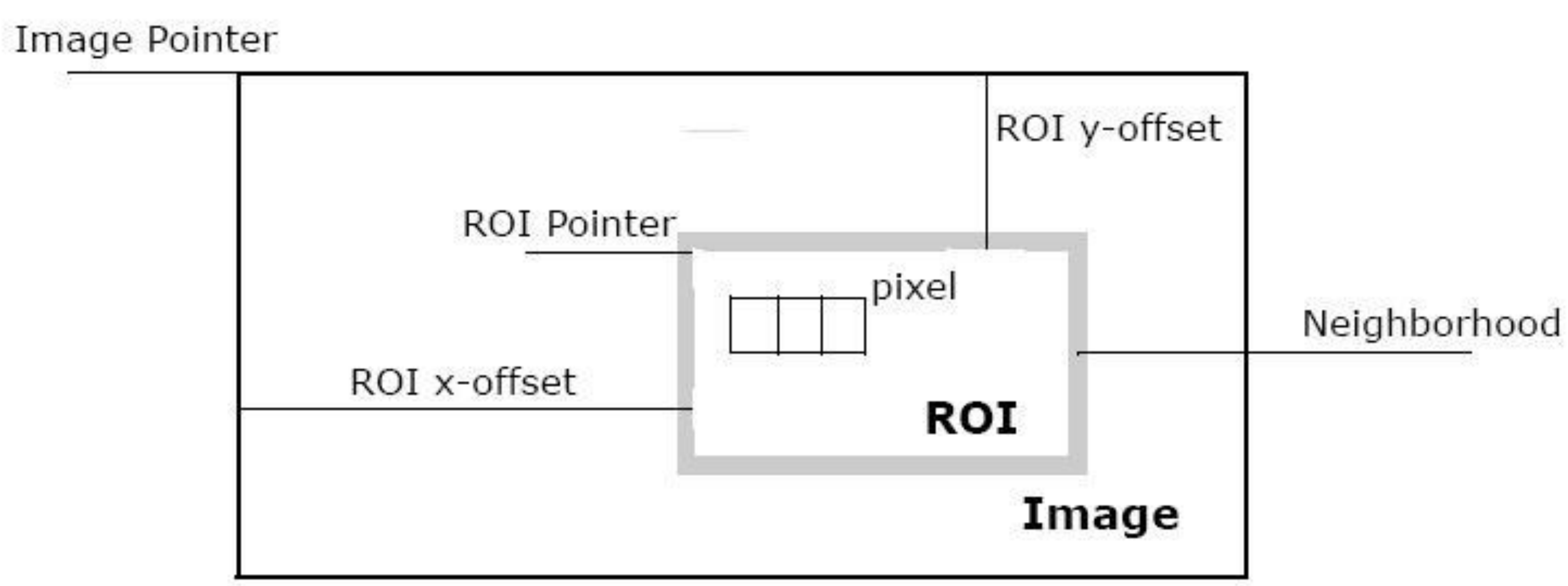
```
event_t event = bilateral(queue, src, dst, spec, border, events);
```

Async call

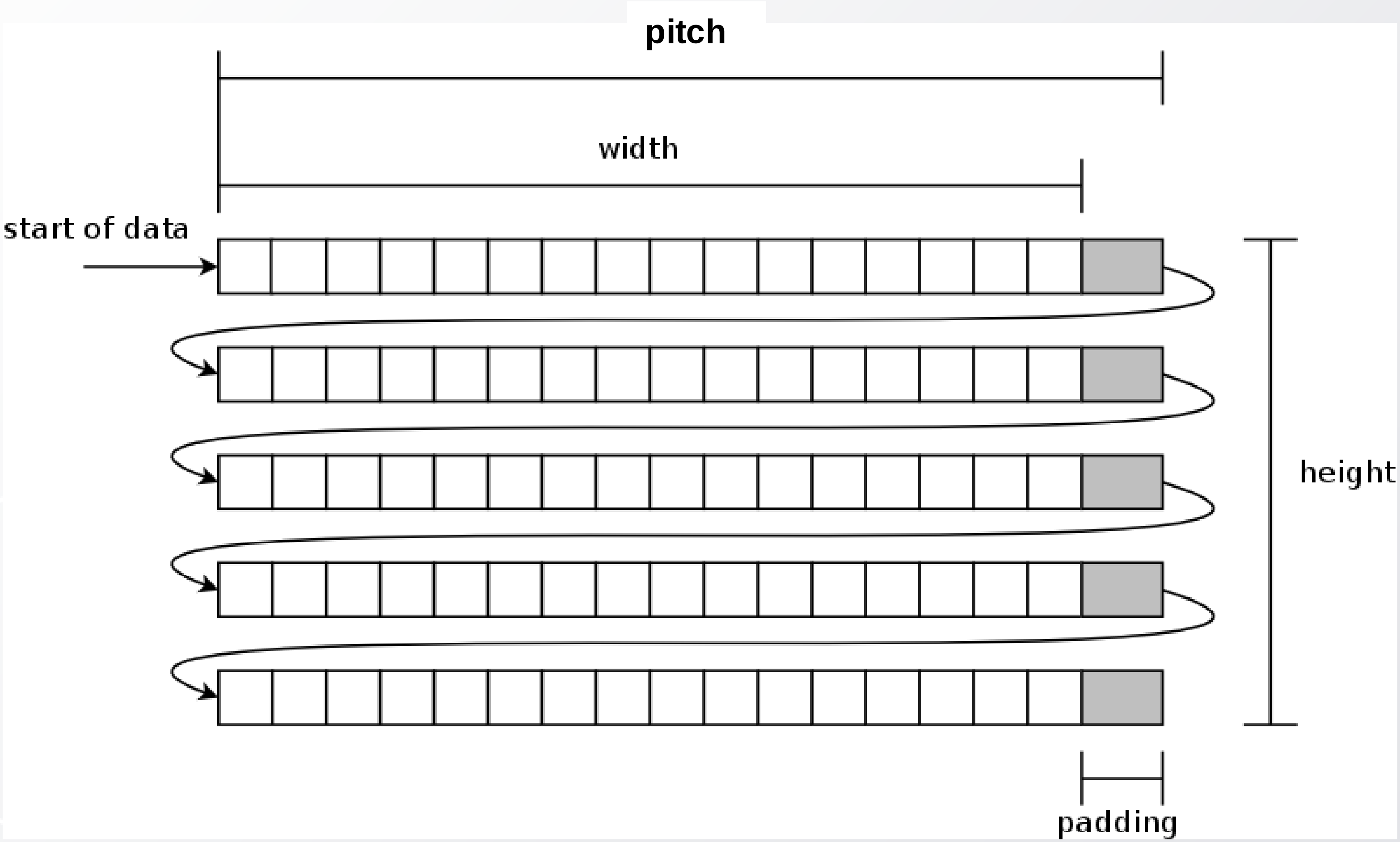
APIs are based on DPC++ and sycl::queue to be able to construct processing pipelines and include oneAPI calls based on DPC++ queue targeted to difference xPUs. Calls are asynchronous and scheduled by device runtime.



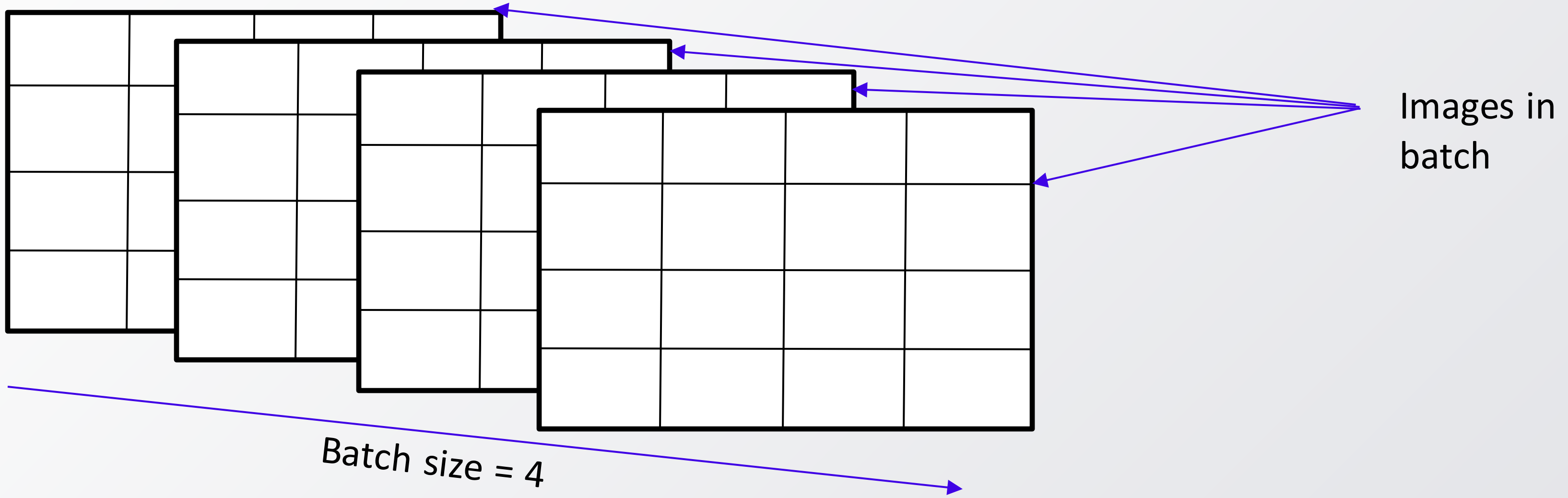
Basic terminology: Image and Region of Interest (ROI)



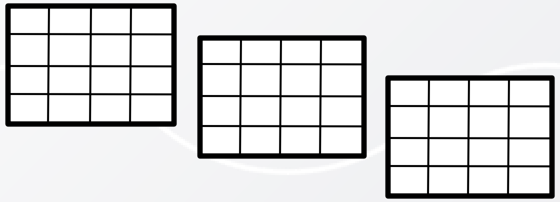
Basic terminology: Step/Pitch/Stride



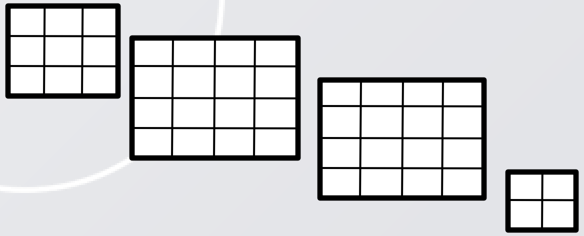
Basic terminology: Batch



Uniform batch (all images have same size)



Non-Uniform batch (images have different sizes)



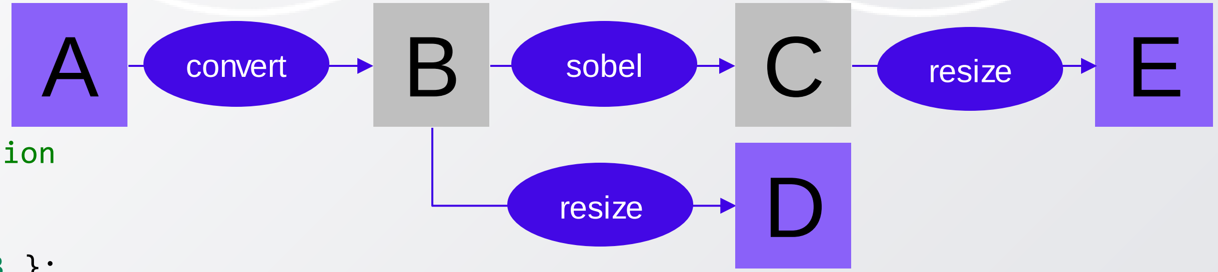
oneIPL programming model

- API was simplified compared to the one discussed a year ago.
- src and dst arguments are of class Image type and defining either image or region of interest (ROI).
- spec argument defines Image-independent parameters like filter kernel size, scalar factors, etc.
- Image-dependent arguments follows before dependencies (e.g., border_val defines the border pixel value required for constant borders in algorithms supporting such border)
- Dependencies argument defines a vector of events of the kernels which shall be completed before the execution of the algorithm
- Example of gaussian algorithm API (SFINAE in the template signature provide the example of compile-time checks and not part of the spec):

```
template <typename ComputeT = float,
          typename SrcImageT,
          typename DstImageT
          detail::enable_for_fp<ComputeT>                          = 0,
          detail::enable_for_same_formats<SrcImageT, DstImageT>    = 0,
          detail::enable_for_same_data_types<SrcImageT, DstImageT> = 0,
          detail::enable_for_nonplanar<SrcImageT>                  = 0>
sycl::event gaussian(sycl::queue&                       queue,
                     SrcImageT&                         src,
                     DstImageT&                         dst,
                     const gaussian_spec<ComputeT>&     spec,
                     const typename SrcImageT::pixel_t& border_val   = {},
                     const std::vector<sycl::event>&    dependencies = {})
```

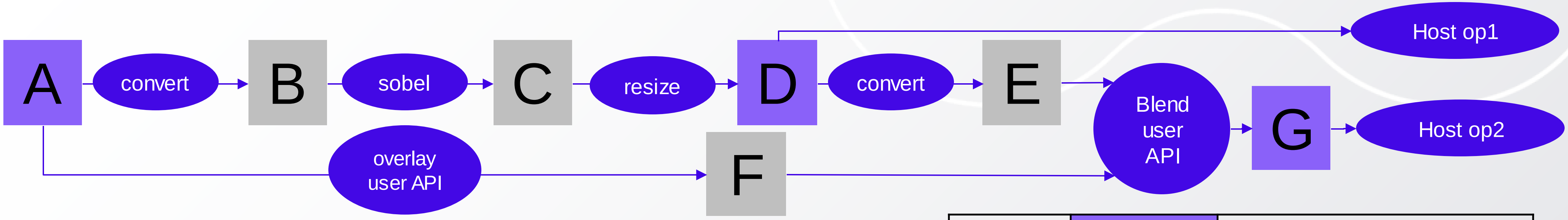
oneIPL API – example

```
template <layouts Layout>
using image_t = ipl::image<Layout, std::uint8_t>;

// Size of source and destination buffers after color conversion
const sycl::range<2> src_size{ src_image_data.get_range() };
const auto           dst_size{ 2 * src_size };
const ipl::roi_rect  dst_roi_rect{ dst_size / 6, dst_size / 3 };

// Create queue
auto queue = examples::make_queue();
// Create allocators
ipl::shared_usm_allocator_t s_allocator{ queue };
ipl::device_usm_allocator_t d_allocator{ queue };

// Source RGBA image data and destination images
image_t<layouts::channel4> src_image{ queue, p_data, src_size };             // A.Source image data
image_t<layouts::plane>    gray_image{ src_size, d_allocator };              // B.Gray image data
image_t<layouts::plane>    sobel_image{ src_size, d_allocator };             // C.Sobel image data
image_t<layouts::plane>    sobel_image_roi{ sobel_image, { src_size * 2 / 3 } };   // C.Sobel image roi
image_t<layouts::plane>    resized_image{ dst_size, s_allocator };           // D. Resized sobel image data
auto                       resized_image_roi = resized_image.get_roi(dst_roi_rect); // E. Resized image roi

// Run pipeline: convert to grayscale -> sobel -> resize_lanczos -> resize_lanczos
(void)ipl::rgb_to_gray(queue, src_image, gray_image);
(void)ipl::sobel_3x3(queue, gray_image, sobel_image);
(void)ipl::resize_lanczos(queue, sobel_image_roi, resized_image);
(void)ipl::resize_lanczos(queue, gray_image, resized_image_roi);
```



Legend		
	A	Input/output
	B	Temporary memory
	operation	Operation

oneIPL API – example with more operations



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<layouts::channel4,  uint8_t> src_image{ src_image_data.get_pointer(), src_size, image_alloc }; // A (image)
ipl::image<layouts::plane, uint8_t> gray_image{ src_size, device_alloc };    // B (device USM)
ipl::image<layouts::plane, uint8_t> sobel_image{ src_size, device_alloc };   // C (device USM)
ipl::image<layouts::plane, uint8_t> resized_image{ dst_size, shared_alloc }; // D (shared USM)
ipl::image<layouts::channel4, uint8_t> res_rgba_image{ dst_size,  image_alloc };// E (image)
ipl::image<layouts::channel4,  uint8_t> ovr_image{ dst_size, image_alloc };  // F (image)
ipl::image<layouts::channel4,  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);
ipl::rgb_to_gray(queue, src_image, gray_image);
ipl::sobel_3x3(queue, gray_image, sobel_image);
ipl::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**

12

oneIPL programming model

- Image data, layout, region of interest (ROI) are specified in **ipl::image class.** Layout and 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 spec contain such matrix.
- Generic layouts is channel count – rows (1ch, 3ch, 4ch). They are mapped to the formats 1ch – plane or grayscale, 3ch – RGB, BGR, 4ch – RGBA,BGRA,…
- Additional layouts supported selectively – P3 (3 planes for R,G,B), subsampled YUV formats, etc.
- Generic datatypes – 8u-32u – unsigned integer, 8s-32s signed integer, fp16-fp64 – floating points

	8u	8s	16u	16s	32u	32s	64u	64s	fp32	fp64	fp16
Plane (C1 in IPP)											
Channel3 (C3)											
Channel4 (C4)											
Plane3 (P3)											
NV12		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

33 generic image formats

extra formats

oneIPL programming model

- Some format-type combinations are device specific (if CPU supports FP64, and GPU doesn't), implementation shall exist only for CPU, and in case GPU supports FP16, and CPU doesn't – only for GPU).
- Each function has own support matrix in spec and it is filed based on current support request. Type support shall be explained in **reference documentation for spec implementation.**

gaussian	8u	8s	16u	16s	32u	32s	32f	64f	16f
1									
3									
4									
P3									

bilateral	8u	8s	16u	16s	32u	32s	32f	64f	16f
1									
3									
4									
P3									

resize_lin	8u	8s	16u	16s	32u	32s	32f	64f	16f
1									
3									
4									
NV12									

resize_super	8u	8s	16u	16s	32u	32s	32f	64f	16f
1									
3									
4									
P3									

Covered in fp64-capable device

Covered in fp16-capable device

Supported

Not supported, **if implemented, spec will be updated in next version**

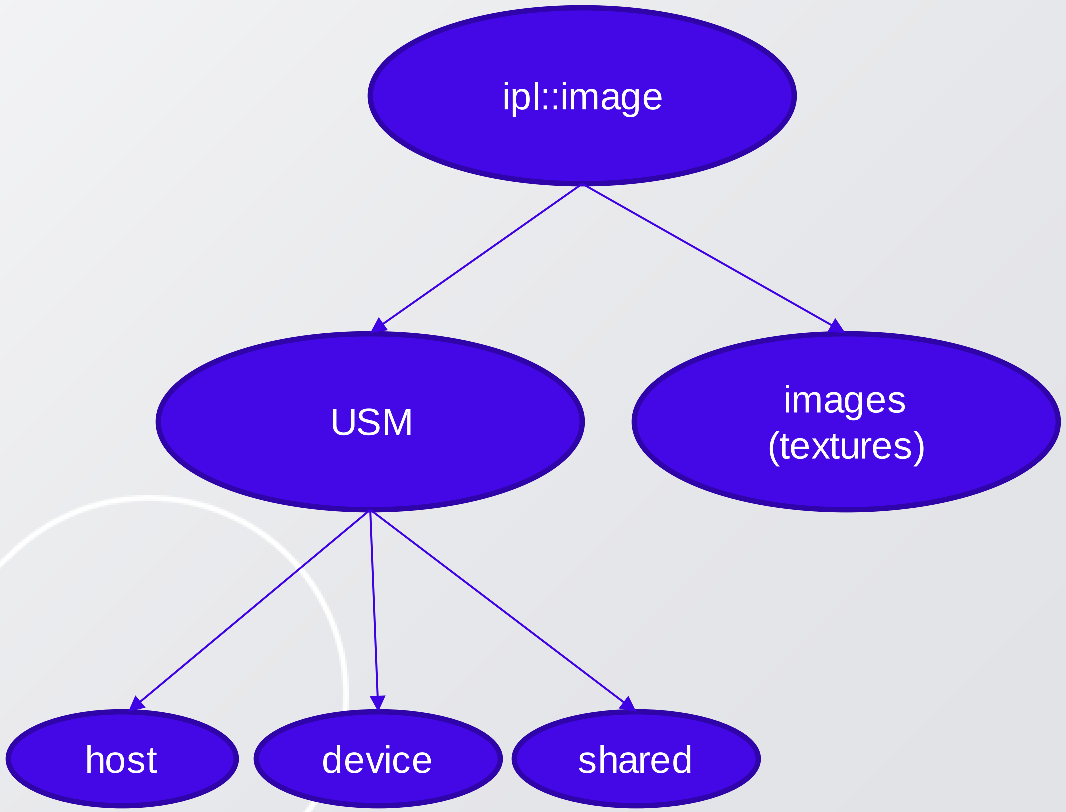
- Extra formats are supported selectively based on use-cases. E.g. at the figure gaussian and bilateral have P3 support, and only resize has NV12 support.

- Generic formats are supported for multiple devices.

oneIPL Image 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 memory – textures.

```
template <layouts Layout, typename DataT, typename AllocatorT>
class image final {
public:
using image_impl_t::image_impl_t;
using allocator_t = AllocatorT;
using pixel_t = pixel_layout_t<DataT, Format>;

image(const image_impl_t& image_impl);
image(image_impl_t&& image_impl);

// special constructors
...
auto operator=(const image_impl_t& image_impl)->image&;
auto operator=(image_impl_t&& image_impl)->image&;

auto get_roi(const roi_rect& roi_rect) const->image;
…
};
```



oneIPL Image Abstraction

Image special constructors

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

```
sycl::queue                    queue;
ipl::shared_usm_allocator_t allocator{ queue };
const sycl::range<2>           image_size{ height, width };
const ipl::roi_rect            roi_rect{ image_size / 6,
image_size / 3 };

// Source 3-channel image data
ipl::image<ipl::layouts::channel3, std::uint8_t> src_image{
queue, p_image_data, image_size, allocator };
// ROI of the source image
ipl::image<ipl::layouts::channel3, std::uint8_t> dst_image{
src_image, roi_rect };
```

oneIPL Image Abstraction

oneIPL supports different type of memory – device, host, shared and special GPU texture (image) memory. Memory type is accessible explicitly via user allocation or via allocator passed to ipl::image constructor, if no external memory is provided. Allocator argument is special tag. By default, allocator is selected as shared or texture depending on supported formats depending on device.

```
template <formats Format, typename DataT,
          typename AllocatorT=select_image_allocator_t<Format, DataT>>
class image;
```

```
Image allocation tags defined in oneIPL spec:
  ipl::host_usm_allocator_t   host USM
  ipl::device_usm_allocator_t device USM
  ipl::shared_usm_allocator_t shared USM
  ipl::image_allocator_t      image memory (special allocator, allocation done
                                            in SYCL by using sycl::image type)
```

Image formats supported by SYCL 2020 Provisional

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

Image format support is limited in the current SYCL spec, so only 4-channel images can be dispatched to Tiled memory on the device.

How does oneIPL compare to IPP?

oneIPL spec is a specification for the DPC++ interfaces covering subset of IPP features for image processing, which introduces

- Data abstraction for 2D images,
- Async execution on devices based on SYCL queue,
- USM and image memory support, memory management, allocators
- Error handling based on exceptions.

Code comparison (IPP C API vs oneIPL DPC++ API)

```
IppStatus status = ippStsNoErr;
Ipp32f* pSrc = NULL, * pDst = NULL;     /* Pointers to source/destination images */
int srcStep = 0, dstStep = 0;          /* Steps, in bytes, through the source/destination images
*/
Ipp32u kernelRadius = 2;
Ipp32f sigma = 0.35f;
IppiBorderType borderType = ippBorderMirror;
IppiSize roiSize = { WIDTH, HEIGHT };  /* Size of source/destination ROI in pixels */
IppFilterGaussianSpec* pSpec = NULL;   /* context structure */
Ipp32u kernelSize = kernelRadius * 2 + 1;
Ipp8u* pBuffer = NULL;                 /* Pointer to the work buffer */
Ipp8u* pInitBuf = NULL;                /* Pointer to the Init buffer */
int iTmpBufSize = 0, iSpecSize = 0, iInitBufSize = 0;    /* Common work buffer size */
Ipp32f borderValue = 0;
int numChannels = 1;
int bufStep = 0;

// Here shall be some code function code allocating and making image data
check_status(status = ippiFilterGaussianGetSpecSize(kernelSize, ipp32f, numChannels,
    &iSpecSize, &iInitBufSize))

pSpec = (IppFilterGaussianSpec*)ippiMalloc_8u_C1(iSpecSize, 1, &bufStep);
pInitBuf = ippiMalloc_8u_C1(iInitBufSize, 1, &bufStep);

check_status(status = ippiFilterGaussianInit(roiSize, kernelSize, sigma, borderType,
    ipp32f, numChannels, pSpec, pInitBuf));

check_status(status = ippiFilterGaussianGetBufferSize(roiSize, kernelSize, ipp32f,
    numChannels, &iSpecSize, &iTmpBufSize))

pBuffer = ippiMalloc_8u_C1(iTmpBufSize, 1, &bufStep);

check_status(status = ippiFilterGaussian_32f_C1R(pSrc, srcStep, pDst, dstStep, roiSize,
    borderType, &borderValue, pSpec, pBuffer))
```

Algorithm parameters

Algorithm call

```
using namespace oneapi::ipl;

const auto kernelRadius = 2;
const auto sigma = 0.35;
const auto borderType = border_type::mirror;
const sycl::range<2> size{ HEIGHT, WIDTH };
const gaussian_spec spec{ kernelRadius, sigma, border_type::mirror};
try
{

    sycl::queue queue{ sycl::cpu_selector{} };

    // Making image obgects on top of raw memory
    image<layouts::plane, float> src_image{ queue, pSrc, size };
    image<layouts::plane, float> dst_image{ queue, pDst, size };

    (void)gaussian(queue, src_image, dst_image, spec);
    queue.wait_and_throw();

}
catch(...)
{
    // Process error
}
```

Key differences:

- Yellow marks is common steps
- Language C vs C++ (DPC++/SYCL)
- IPP C API requires extra arguments and initializations and more code
- Error handling STATUS code in C/Exception in SYCL API
- IPP DPC++ API is asynchronous, call is as part of pipeline executed on device

Code comparison (IPP C API vs IPP DPC++ API)

```
    // Classical IPP code doing Gaussian Filter
    IppStatus status = ippStsNoErr;
    Ipp32f* pSrc = NULL, * pDst = NULL;     /* Pointers to source/destination images */
    int srcStep = 0, dstStep = 0;          /* Steps, in bytes, through the
source/destination images */
    Ipp32u kernelRadius = 2;
    Ipp32f sigma = 0.35f;
    IppiBorderType borderType = ippBorderMirror;
    IppiSize roiSize = { WIDTH, HEIGHT };  /* Size of source/destination ROI in pixels */
    IppFilterGaussianSpec* pSpec = NULL;   /* context structure */
    Ipp32u kernelSize = kernelRadius * 2 + 1;

    // Initialization IPP sequence is skipped here with some declarations of variables
    check_status(status = ippiFilterGaussian_32f_C1R(pSrc, srcStep, pDst, dstStep,
roiSize,
      borderType, &borderValue, pSpec, pBuffer))

    // SYCL code for xPU doing Gaussian Filter
    using namespace oneapi::ipl::experimental;
    const auto borderType = border_type::mirror;
    const sycl::range<2> size{ HEIGHT, WIDTH };
    const gaussian_spec spec{ kernelRadius, sigma, border_type::mirror};
    try
    {

        sycl::queue queue{ sycl::cpu_selector{} };

        // Making image obgects on top of raw memory
        image<layouts::plane, float> src_image{ queue, pSrc, size };
        image<layouts::plane, float> dst_image{ queue, pDst, size };

        (void)gaussian(queue, src_image, dst_image, spec);
        queue.wait_and_throw();
    }
    catch(...)
    {
        // Process error
    }
```

Both SYCL code and classical IPP code can be used in single DPC++ application.

If a lot of features are on CPU, they can be used from C APIs and some selected features for xPU can be used additionally. The code can be split across files and linked to single application as well.

This approach will allow to integrate xPU calls in addition to existing features for existing customers. It requires anyway to rebuild the C application using DPC++ compiler and introduce dependency to new xPU libraries and DPC++ SYCL runtime.

It doesn’t not allow to construct pipelines executed on device which is possible only combining the calls of oneAPI APIs like xPU IPP APIs, oneMKL, oneDAL, etc.

The pipelines with multiple DPC++ kernels queues is the target use case to new xPU interfaces.

NPP vs DPC++ oneIPL APIs (NPP snippet is from production code)

Real project using NPP (open3d) - lots of code with runtime checks. Number of IF constructs is N_TYPES * N_LAYOUTS in each functions. Performance impact – N_IFS * number of calls. Having a loop would spend time on Ifs at each iteration.

```
void Resize(const open3d::core::Tensor& src_im,
    open3d::core::Tensor& dst_im, ...) {
    // ... here is some conversion code
    auto dtype = src_im.GetDtype();
    auto context = MakeNPPContext();
#define NPP_ARGS
    static_cast<const npp_dtype *>(src_im.GetDataPtr()),              \
            src_im.GetStride(0) * dtype.ByteSize(), src_size, src_roi, \
            static_cast<npp_dtype *>(dst_im.GetDataPtr()),             \
            dst_im.GetStride(0) * dtype.ByteSize(), dst_size, dst_roi, \
            it->second, context

    if (dtype == core::UInt8) {
        using npp_dtype = Npp8u;
        if (src_im.GetShape(2) == 1) {nppiResize_8u_C1R_Ctx(NPP_ARGS);}
        else if (src_im.GetShape(2) == 3) {nppiResize_8u_C3R_Ctx(NPP_ARGS);}
        else if (src_im.GetShape(2) == 4) {nppiResize_8u_C4R_Ctx(NPP_ARGS);}
    }
    else if (dtype == core::UInt16) {
        using npp_dtype = Npp16u;
        if (src_im.GetShape(2) == 1) {nppiResize_16u_C1R_Ctx(NPP_ARGS);}
        else if (src_im.GetShape(2) == 3) {nppiResize_16u_C3R_Ctx(NPP_ARGS);}
        else if (src_im.GetShape(2) == 4) {nppiResize_16u_C4R_Ctx(NPP_ARGS);}
    }
    else if (dtype == core::Float32) {
        using npp_dtype = Npp32f;
        if (src_im.GetShape(2) == 1) {nppiResize_32f_C1R_Ctx(NPP_ARGS);}
        else if (src_im.GetShape(2) == 3) {nppiResize_32f_C3R_Ctx(NPP_ARGS);}
        else if (src_im.GetShape(2) == 4) {nppiResize_32f_C4R_Ctx(NPP_ARGS);}
        // other branches ...
    }
}
```

oneIPL code – no checks, compile-time dispatching. Customer code needs to be written in C++ with templates, but gives significant performance advantage for such example. Also, code is much simpler compared to C-style.

```
// Data types and function signatures shall be changes in customer code to
have benefits from C++ APIs
// otherwise code will looks the same as in original sample
template<typename T, layouts L>
void Resize(image_descriptor<T>& src_image_descr, image_descriptor<T>&
dst_image_descr) {
    // Create queue
    sycl::queue queue{};

    // Source rgba image data (image_image)
    image<T, L> src_image{ src_image_descr.p_data,
src_image_descr.src_size, src_image_descr.src_roi };
    // Destination rgba image data (image_image)
    image<T, L> dst_image{ dst_image_descr.p_data,
src_image_descr.dst_size, src_image_descr.dst_roi };

    (void)resize_bilinear(queue, src_image, dst_image);
    queue.wait();
}
```

DPC++ API requires and produces less code in this case.
For many important use-cases processing works with **fixed format known at compile-time**:
Images are usually decoded to layout requested from decoder: bgr(a)/rgb(a)/nv12, etc. Conditional switching of the format with extra code is not required contrary to the C API case.

Thank you for attending!

- If you have content to post to oneAPI.io, please let us know
- Please feel free to extend invitations to others to join the Image SIG or other oneAPI Community Forums
- Join oneAPI on LinkedIn: https://www.linkedin.com/groups/14241252/