IWOCL & SYCLCon 2022

Optimizing AI Pipelines with OpenVINO™ and SYCL

Nico Galoppo

Principal Engineer – AI Performance

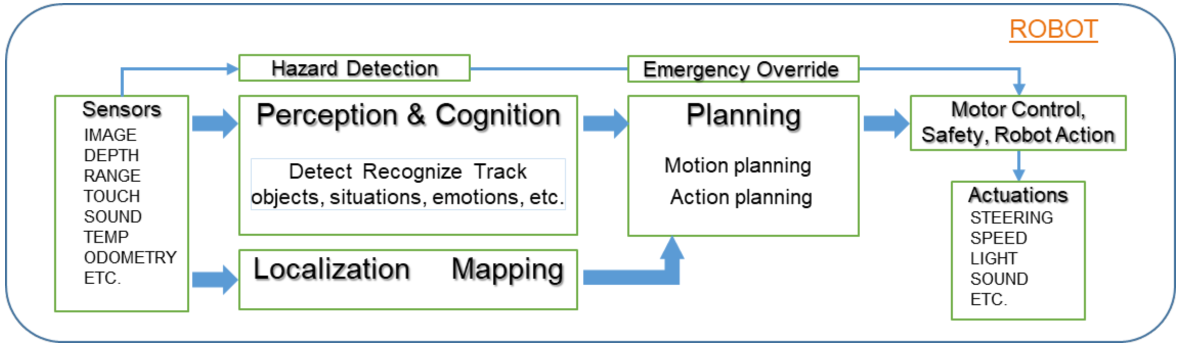
Intel® Internet of Things Group

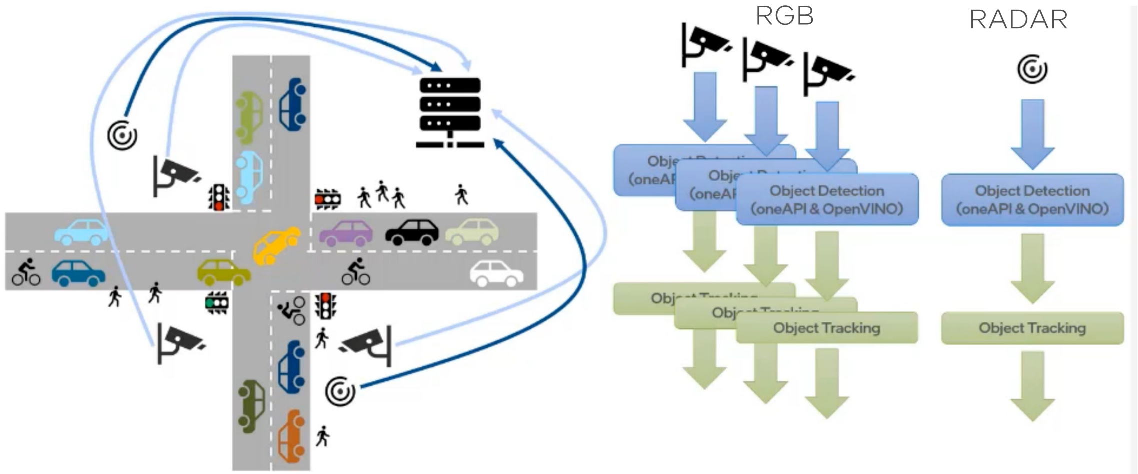
intel®

Modern AI Pipelines

Complex pipelines of multi-modal data-parallel processing

- Autonomous Mobile Robots
- Realtime Traffic Monitoring

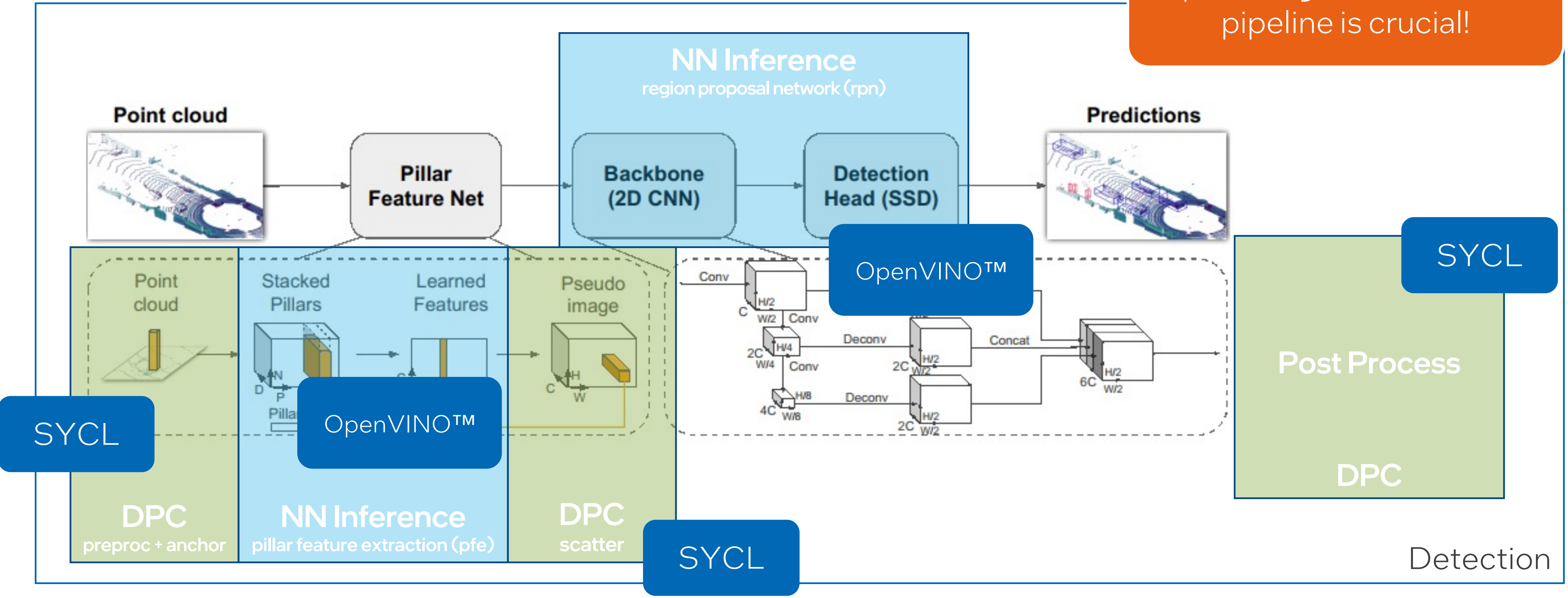




Case Study – PointPillars: Object Detection from Point Clouds

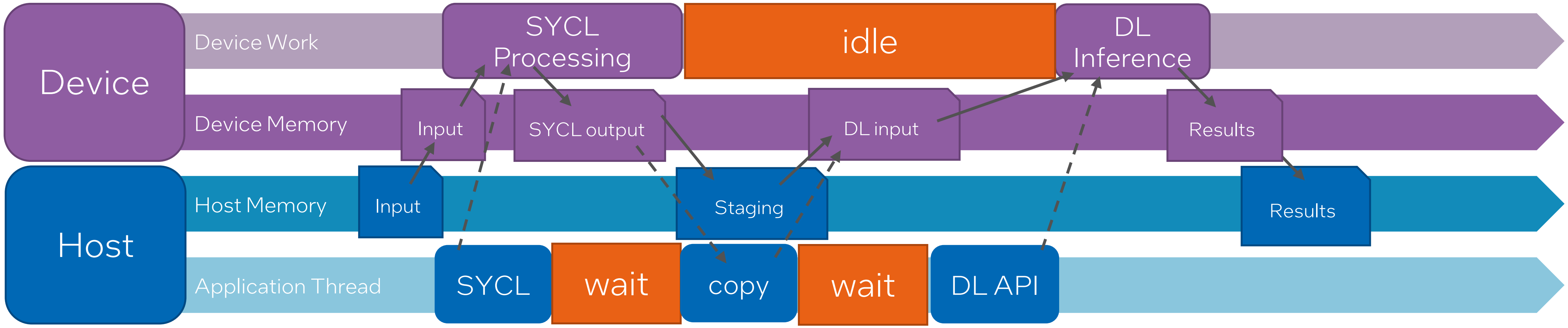
A mix of data-parallel compute & NN inference

Optimizing the end-to-end pipeline is crucial!



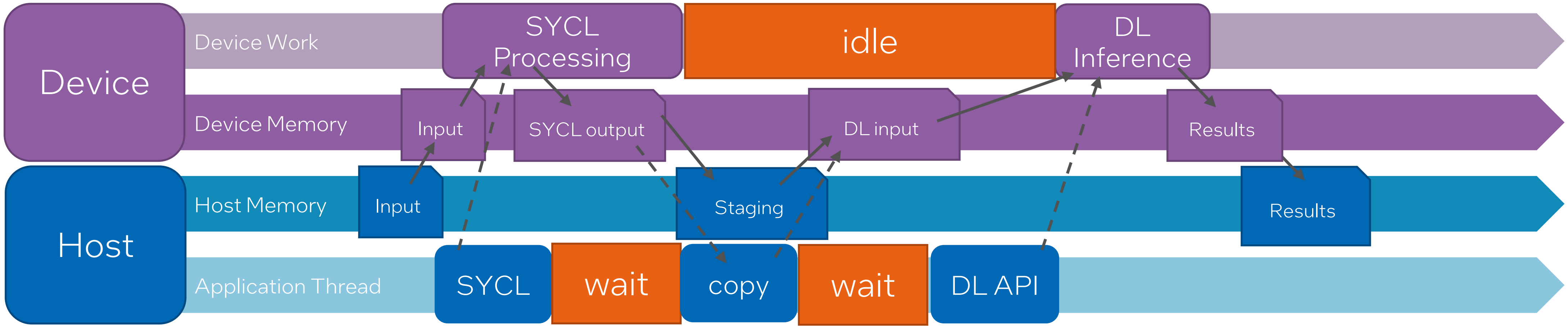
End-to-end Optimization of Mixed Pipelines

- Potential inefficiencies:
  - Unnecessary memory copies
  - Synchronization bubbles: explicit waits on the host application thread



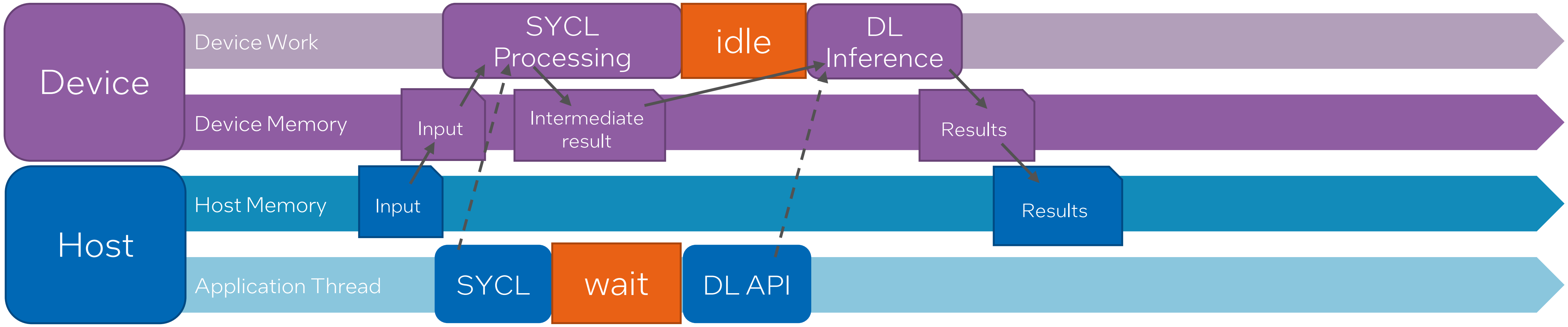
End-to-end Optimization of Mixed Pipelines

- Optimize memory transfer time: avoid unnecessary copies
  - Between host & device on integrated platforms
  - Share device memory between APIs
  - Requires the concept of common "handles" across APIs



End-to-end Optimization of Mixed Pipelines

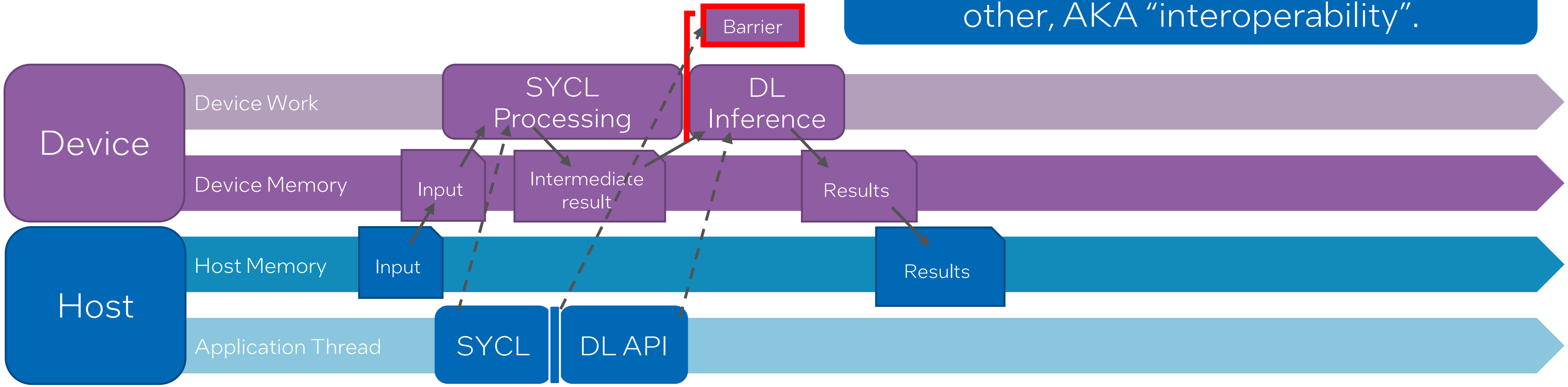
- Optimize memory transfer time: avoid unnecessary copies
  - Between host & device on integrated platforms
  - Share device memory between APIs
  - Requires the concept of common "handles" across APIs
- Minimize synchronization bubbles
  - Avoid explicit waits on the application thread between different API calls
  - Requires common work queue or the ability to schedule events across APIs



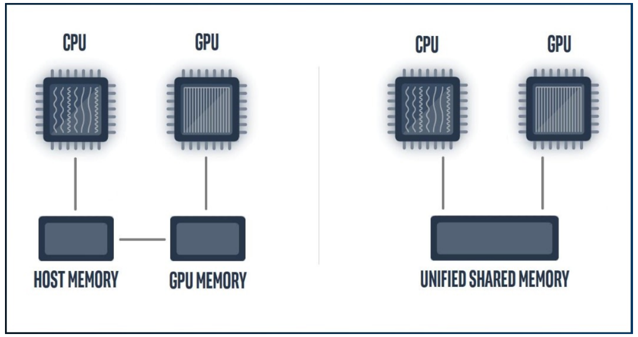
End-to-end Optimization of Mixed Pipelines

- Optimize memory transfer time: avoid unnecessary copies
  - Between host & device on integrated platforms
  - Share device memory between APIs
  - Requires the concept of common "handles" across APIs
- Minimize synchronization bubbles
  - Avoid explicit waits on the application thread between different API calls
  - Requires common work queue or the ability to schedule events across APIs

Solution: allow the APIs to "talk" to each other, AKA "interoperability".



API Overview



- SYCL 2020:
  - Supports many backends, including OpenCL, Level Zero
  - Expanded interoperability
  - Unified Shared Memory (USM) – **common memory handles!**
- OpenVINO™: open-source deep learning toolkit & inference runtime
  - Supports multiple device types with plugin design
  - Plugins can support multiple backends, e.g. GPU plugin supports OpenCL & Level Zero backends and USM
  - DL inference scheduled through asynchronous inference requests

Our Optimization Tools: Interoperability APIs

SYCL Interop APIs with supported backends (incl. OpenCL)

SYCL object → Backend object

```
template<backend Backend, class T>
backend_return_t<Backend, T> get_native(const T &syclObject);
```

Backend object → SYCL object

```
template<backend Backend>
queue make_queue(const backend_input_t<Backend, queue> &backendObject,
                 const context &targetContext,
                 const async_handler asyncHandler = {});
```

Our Optimization Tools: Interoperability APIs

OpenVINO™ Interop APIs (currently with OpenCL & VAAPI)

- RemoteContext: wraps native backend context
  - Create from native handle or get from OpenVINO™ runtime plugin

```
cl_context ctx = get_cl_context();
ov::intel_gpu::ocl::ClContext gpu_context(core, ctx);
```

- RemoteTensor: wraps native backend memory handles
  - Create form native handle or allocation by OpenVINO™ runtime plugin
  - Native handle types include USM pointers, cl_mem, cl::Buffer/cl::Image2D
  - Inherits from ov::Tensor – can be used with all standard OpenVINO™ inference request APIs

```
void* shared_buffer = allocate_usm_buffer(input_size);
auto remote_tensor = gpu_context.create_tensor(in_element_type, in_shape, shared_buffer);
```

USM pointers are supported by OpenVINO™ and SYCL!

A Recipe for OpenVINO™ SYCL Interoperability

Basic principle: use a common backend!
Here we'll use OpenCL as the backend.
Enforce backend with SYCL_DEVICE_FILTER.

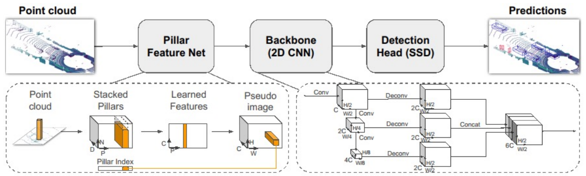
A Recipe for OpenVINO™ SYCL Interoperability

Use common backend: OpenCL
Use SYCL_DEVICE_FILTER

Create	Create a RemoteContext from SYCL queue.
Compile	Compile an OpenVINO™ DL model using the RemoteContext.
Allocate	Allocate USM memory (device or shared).
Create	Wrap all DL in/output USM in RemoteTensor objects.
Assign	Assign RemoteTensors as inference in/outputs.
Sync	Move host thread synchronization to device side synchronization

```
ov::Core core;
cl_command_queue q =
  sycl::get_native<sycl::backend::opencl>(sycl_queue);
ov::RemoteContext remote_context =
  ov::intel_gpu::oclClContext(core, q);

auto compiled_network = core.compile_model(model, remote_context);

float * dev_input = sycl::malloc_device<float>(size, sycl_queue);
ov::RemoteTensor input_tensor =
  remote_context.create_tensor(element_type, shape, dev_input);

float * dev_output = sycl::malloc_device<float>(size, sycl_queue);
ov::RemoteTensor output_tensor =
  remote_context.create_tensor(element_type, shape, dev_output);

auto infer_request = compiled_network.create_infer_request();
infer_request.set_input_tensor(0, input_tensor);
infer_request.set_tensor(output_name, output_tensor);

// ... generate input tensor with SYCL kernel...

infer_request.start_async();

clEnqueueBarrierWithWaitList(q, 0, nullptr, nullptr);

// ... more processing with SYCL...
```

Putting Theory into Practice – PointPillars Optimization



SYCL Processing

OpenVINO™ Inference

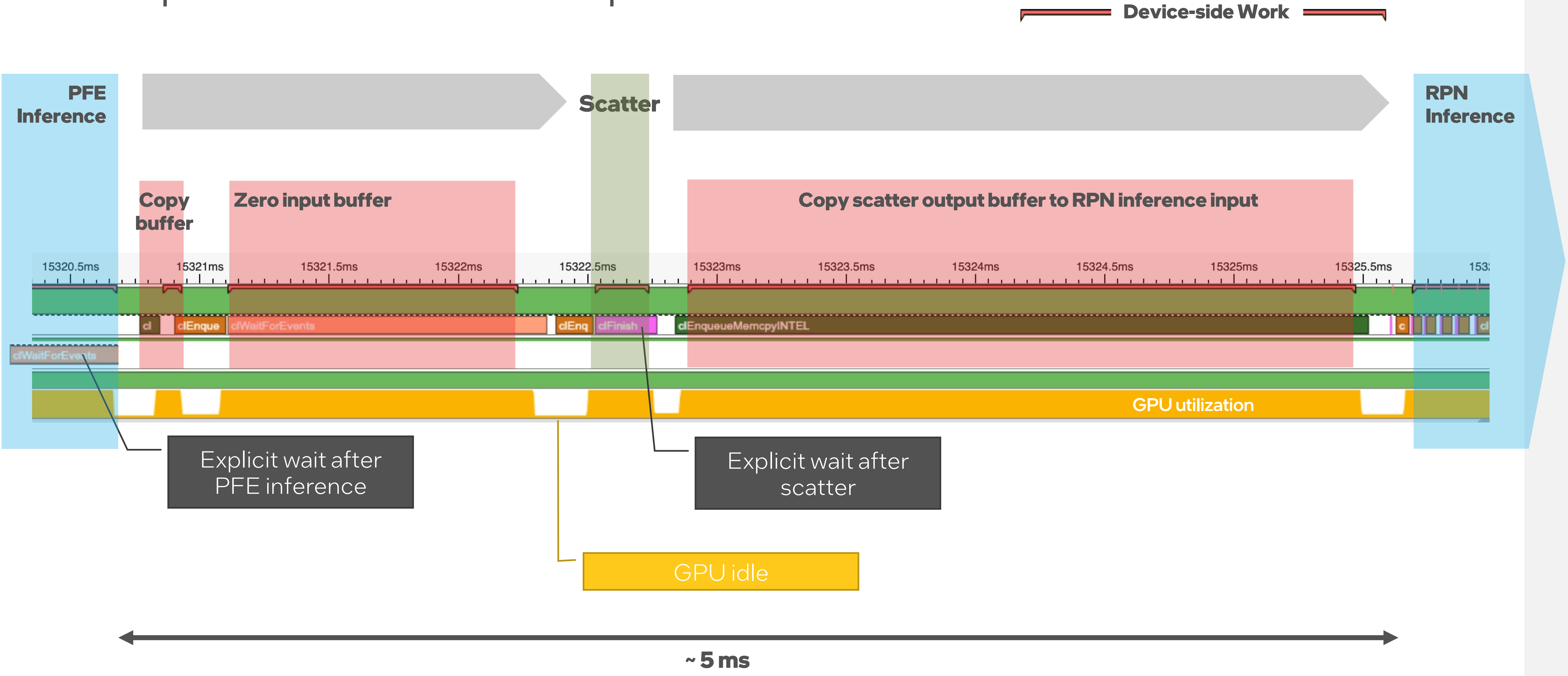
Pre-Process → Create Anchors → PFE Inference → Scatter → RPN Inference → Post-Process

Optimize transition between each stage:

1. Share SYCL output memory with inference input → remove memory copies
2. Share inference output memory as SYCL input → remove memory copies
3. Remove waits on the application thread → increase device utilization

Code: PointPillars OneAPI Sample

Deep Dive – Before Optimization



Step 1: Share Scatter Output with RPN Inference

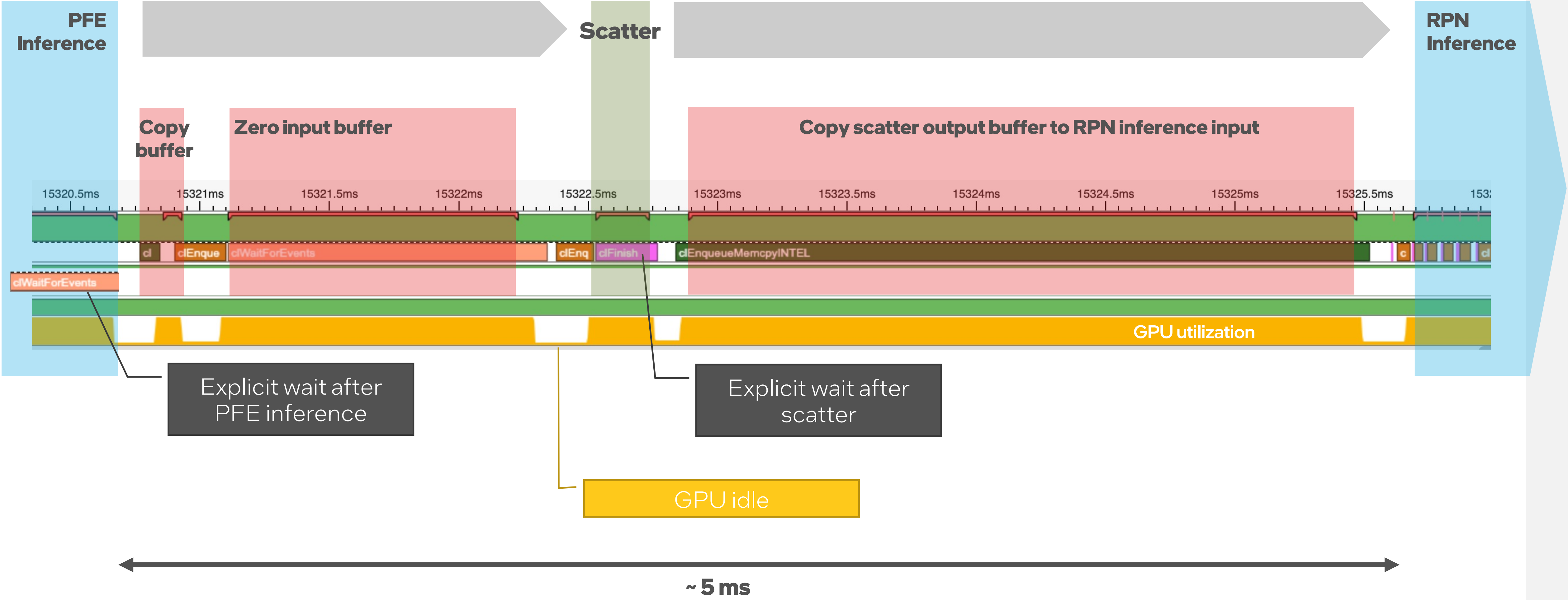
```cpp
auto rpn_exe_network = core.compile_model(model);

// ...

DoScatter(dev_pfe_output, dev_scatter_output);

// A Tensor is a host-side representation of the memory, implies
// map/unmap, i.e. copy of host memory to device memory when inference
// request is submitted.
ov::Tensor input_tensor = rpn_infer_request.get_input_tensor(idx);
sycl_queue.memcpy(input_tensor.data(), dev_scatter_output, scatter_output_size)
  .wait(); // Explicit wait on the host thread!

rpn_infer_request.start_async();
rpn_infer_request.wait();
```

Before

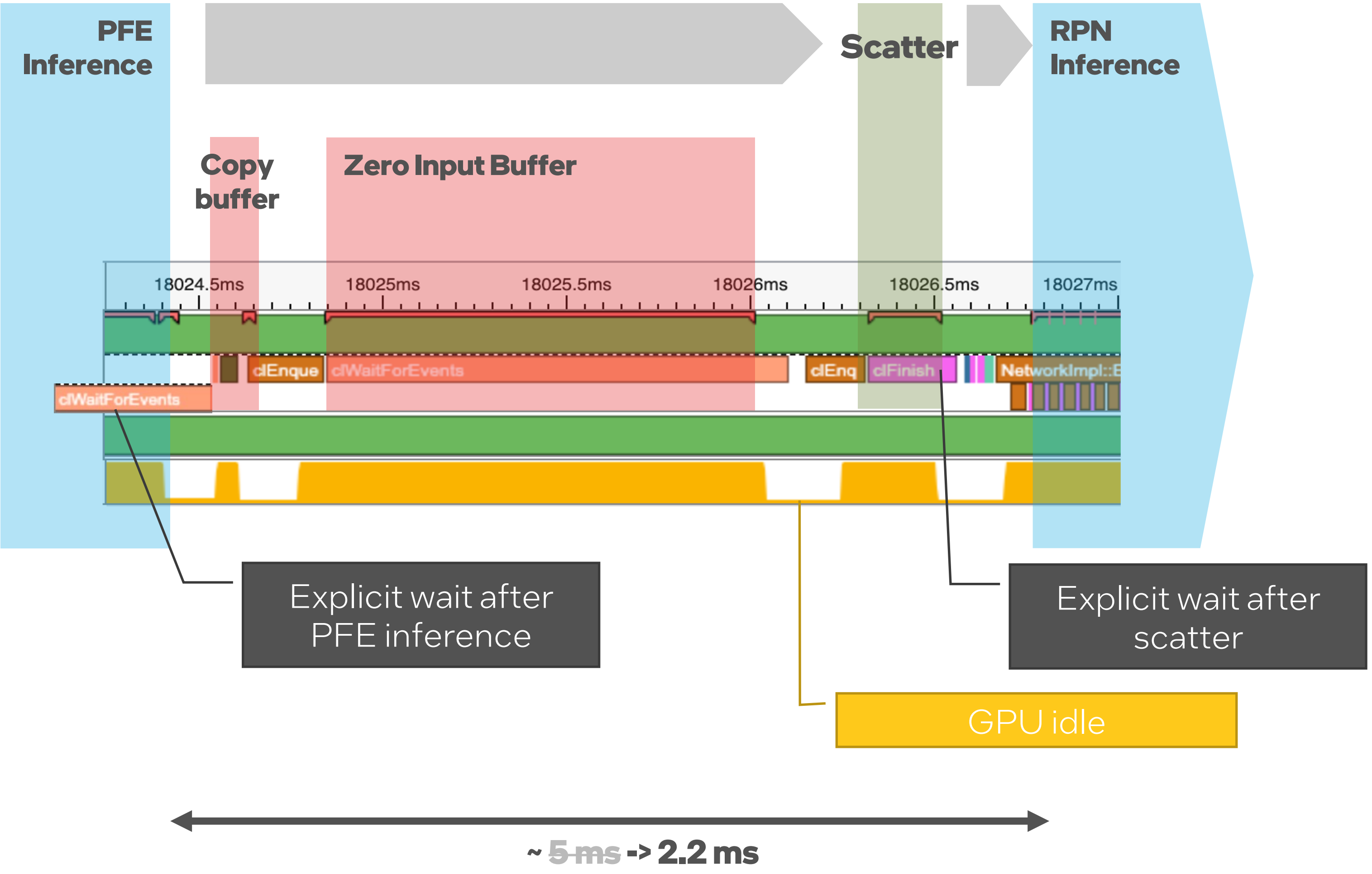
```cpp
// 1. Create RemoteContext
ov::Core core;
cl_command_queue q =
  sycl::get_native<sycl::backend::opencl>(sycl_queue);
ov::RemoteContext remote_context =
  ov::intel_gpu::oclClContext(core, q);

// 2. Compile model using RemoteContext
auto rpn_exe_network = core.compile_model(model, remote_context);

// 3. Allocate USM memory
float * dev_scatter_output = sycl::malloc_device<float>(size, sycl_queue);

// 4. Create RemoteTensors to wrap USM memory
ov::RemoteTensor scatter_output_tensor =
  remote_context.create_tensor(element_type, shape, dev_scatter_output);

// ...

DoScatter(dev_pfe_output, dev_scatter_output);

// 5. Assign RemoteTensors to inference input
rpn_infer_request.set_input_tensor(idx, scatter_output_tensor);

rpn_infer_request.start_async();
rpn_infer_request.wait();
```

After

Before Optimization



Step 1: Share Scatter Output with RPN Inference



Removed memory copy, 2.8 ms savings!

Step 2: Share PFE Output with Scatter Input

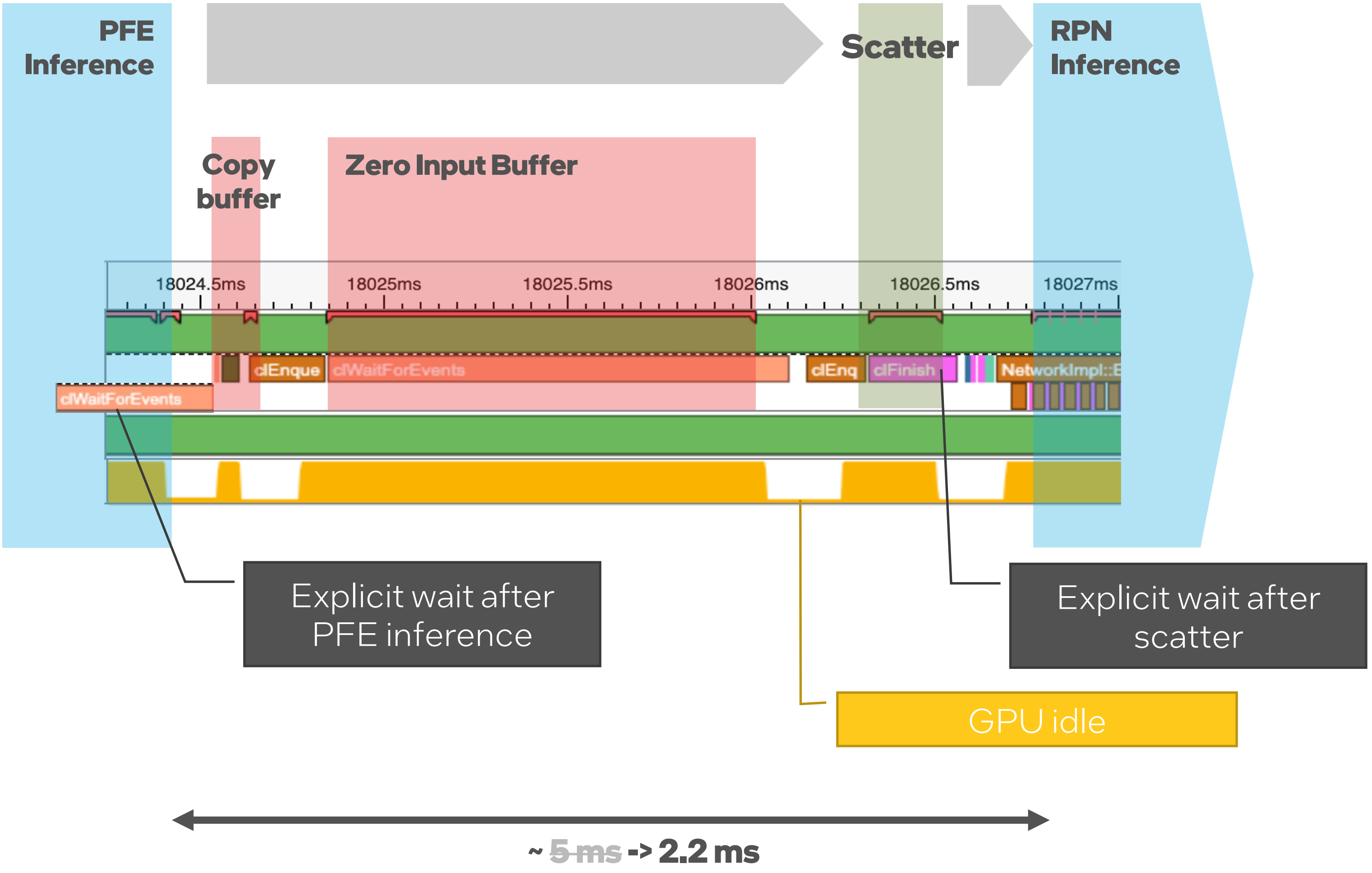
```
ov::Tensor output_tensor = pfe_infer_request_.get_tensor(output_name);
  .wait(); // Explicit wait on the host thread!

pfe_infer_request.start_async();
pfe_infer_request.wait();

sycl_queue.memcpy(dev_pfe_output, output_tensor.data(), size)

DoScatter(dev_pfe_output, dev_scatter_output);
```

Before

```
// 4. Create RemoteTensors to wrap USM memory
ov::RemoteTensor pfe_output_tensor =
  remote_context.create_tensor(element_type, shape, dev_pfe_output);

// 5. Assign RemoteTensors to inference output
pfe_infer_request.set_tensor(name, pfe_output_tensor);

pfe_infer_request.start_async();
pfe_infer_request.wait();

DoScatter(dev_pfe_output, dev_scatter_output);
```

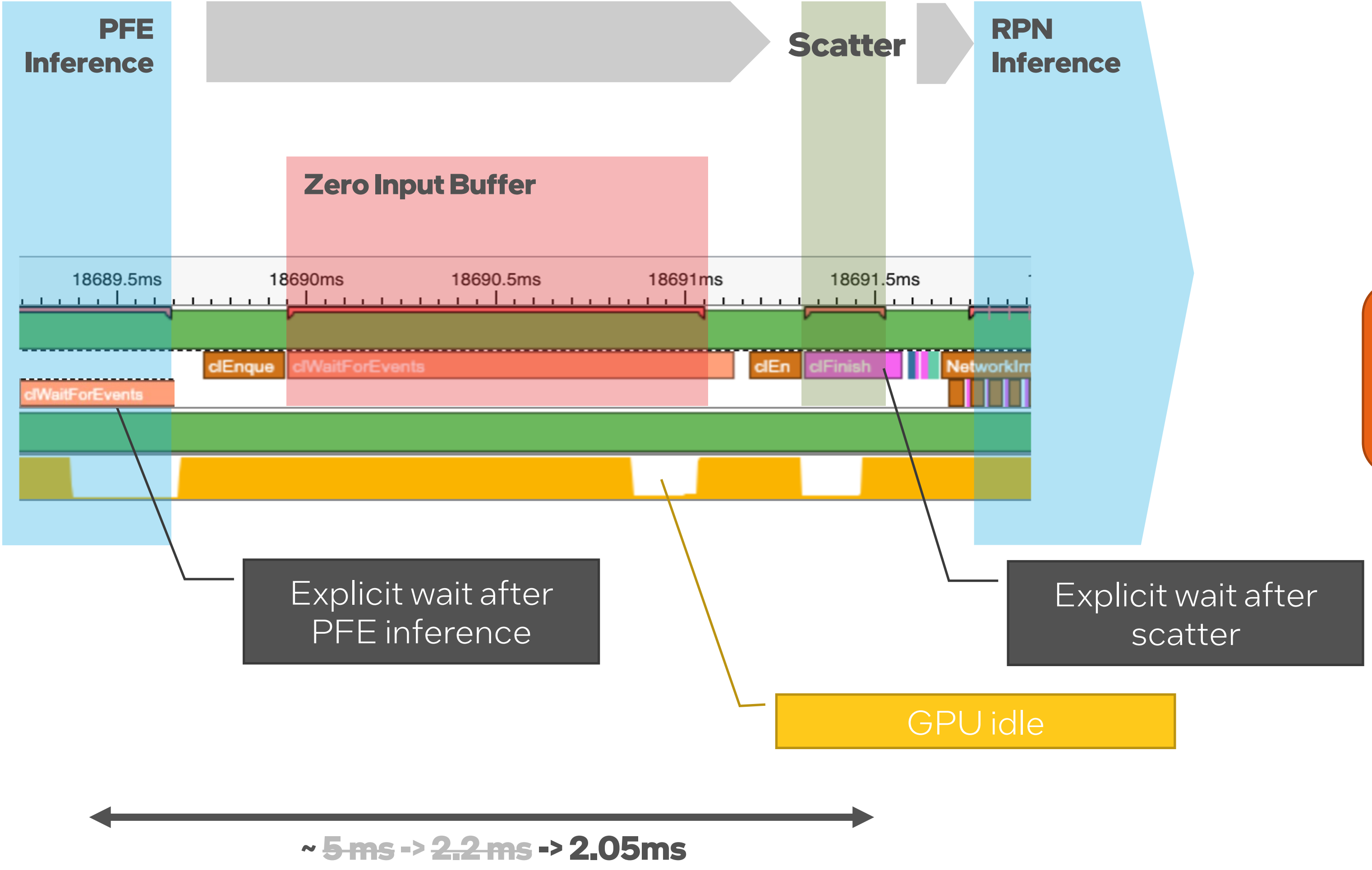
After

Step 2: Share PFE Output with Scatter Input



Before Optimization

Step 2: Share PFE Output with Scatter Input



Removed 2nd memory copy, +0.15 ms savings!

Step 3: Remove Waits on the Application Thread

SYCL & OpenVINO™ runtime are using the same OpenCL command queue under the hood.

Replace host thread waits with device side barriers on the shared queue between API calls (using OpenCL API).

```
cl_command_queue q =
    sycl::get_native<sycl::backend::opencl>(sycl_queue);
```

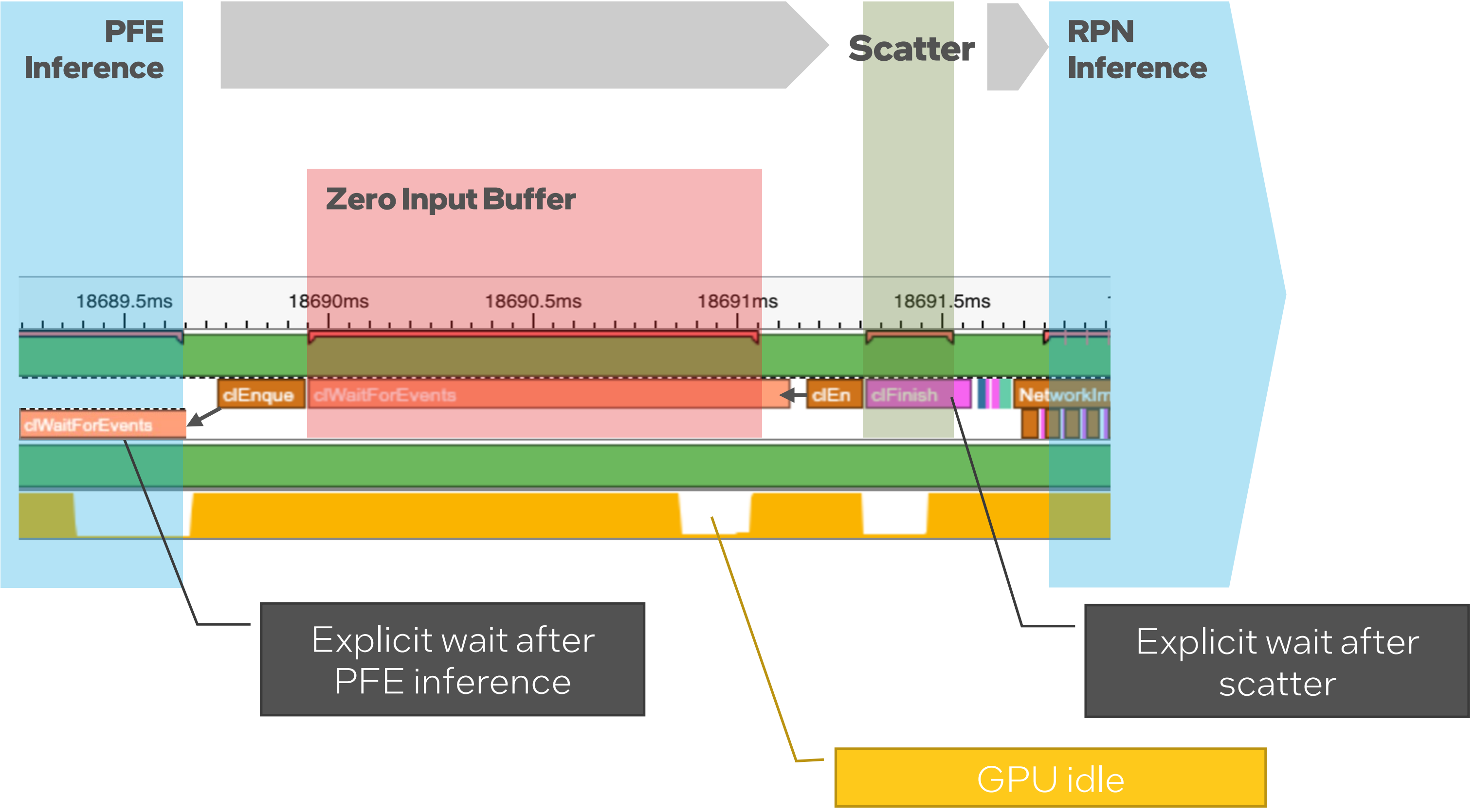
```
ov::RemoteTensor pfe_output_tensor =
    remote_context.create_tensor(element_type, shape, dev_pfe_output);

pfe_infer_request.set_tensor(name, pfe_output_tensor);
pfe_infer_request.start_async();

pfe_infer_request.wait();

DoScatter(dev_pfe_output, dev_scatter_output);

sycl_queue.wait();

ov::RemoteTensor scatter_output_tensor =
    remote_context.create_tensor(element_type, shape, dev_scatter_output);
rpn_infer_request.set_input_tensor(idx, scatter_output_tensor);

rpn_infer_request.start_async();
```

Before

```
ov::RemoteTensor pfe_output_tensor =
    remote_context.create_tensor(element_type, shape, dev_pfe_output);

pfe_infer_request.set_tensor(name, pfe_output_tensor);
pfe_infer_request.start_async();

clEnqueueBarrierWithWaitList(q, 0, nullptr, nullptr);

DoScatter(dev_pfe_output, dev_scatter_output);

clEnqueueBarrierWithWaitList(q, 0, nullptr, nullptr);

ov::RemoteTensor scatter_output_tensor =
    remote_context.create_tensor(element_type, shape, dev_scatter_output);
rpn_infer_request.set_input_tensor(idx, scatter_output_tensor);

rpn_infer_request.start_async();
```

After

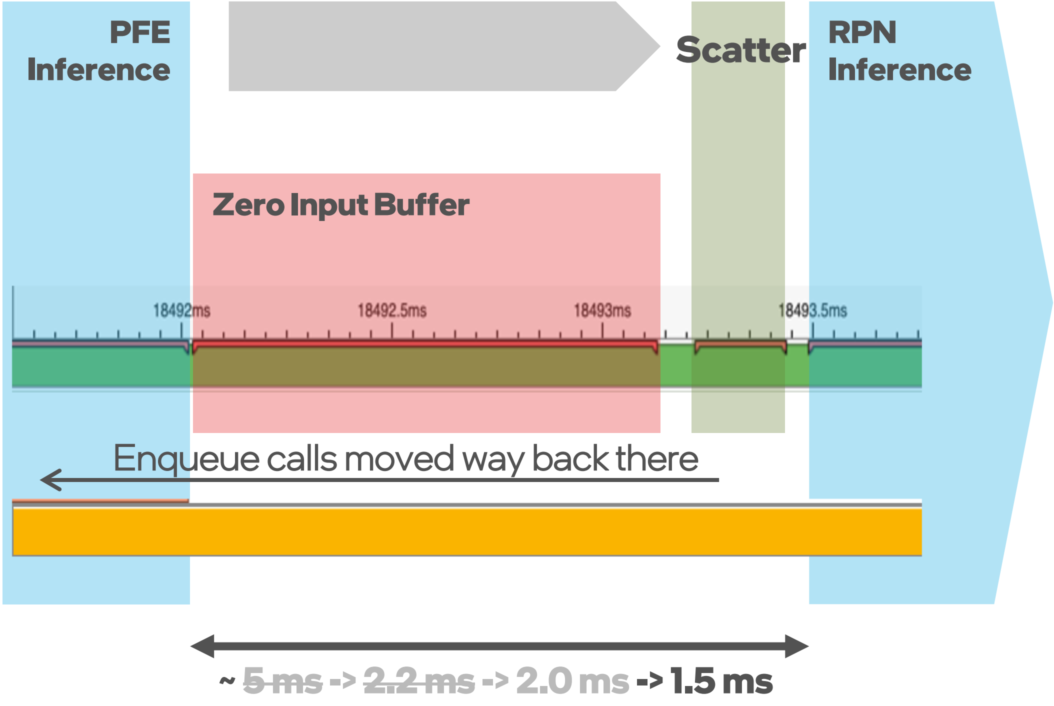
Step 3: Remove Waits on the Application Thread



~ ~~5 ms~~ -> ~~2.2 ms~~ -> **2.05ms**

Before Optimization

Step 3: Remove Waits on the Application Thread



100% GPU utilization
Overall gains: 5 ms →1.5 ms

OpenVINO™ Interoperability API Implementation Details

- Abstract base classes: RemoteContext & RemoteTensor
  - Interface can be implemented by any Inference Engine plugins, using any compute backend.
  - Currently OpenCL Buffer/Image2D and USM tensors implemented by the GPU plugin
- Context & Queue sharing
  - Allows for pipeline scheduling on app side and avoid blocking of host thread on waiting for completion of inference
- Limitations
  - Queue sharing cannot be combined with multiple concurrent queue optimizations in OpenVINO™.
  - No event/synchronization/dependencies mechanism. Application needs to manage the shared native backend queue manually.

Proposal: Synchronization Interop

- Could add direct support for SYCL event type
  - Not very flexible since the inference plugin would have to support SYCL
- Instead, follow RemoteContext/RemoteTensor pattern: RemoteEvent
  - Inference plugin only needs to support the the shared backend API

```
class RemoteEvent {
  // ...
  void wait();
  // ...
};

class InferRequest {
  // ...
  RemoteEvent start_async(std::vector<RemoteEvent>& dependencies);
};
```

```
DoScatter( ... );

clEnqueueBarrierWithWaitList(q, 0, nullptr, nullptr);
infer_request.start_async();
clEnqueueBarrierWithWaitList(q, 0, nullptr, nullptr);
```

```
sycl::event e = DoScatter( ... );

cl_event scatter_event = sycl::get_native<sycl::backend::opencl>(e);
ov::RemoteEvent dep = remote_context.create_event(event);

ov::RemoteEvent e = infer_request.start_async({dep});
```

Call to Action: Add Interoperability with SYCL to your API

- Lean on a common compute "backend" (OpenCL, Level Zero,…).
- Learn from OpenVINO™ Interoperability API.
  - Abstract context/memory/sync objects.
  - Implement derived instances for each supported backend.
- This approach provides maximum flexibility.
  - Your API does not need to "speak" SYCL, only the common backend API.
  - Naturally extends to interop with programming APIs beyond SYCL.

Thank you!

nico.galoppo@intel.com – Twitter: @ngaloppo

Call to Action

- OpenVINO™ Remote Tensor API documentation
- Examples of Interoperability in other APIs
  - oneDNN / SYCL interoperability
  - Kernel and API interoperability with OpenCL* and SYCL* technology
- Optimizations will be available on GitHub
  - PointPillars OneAPI Sample

intel®