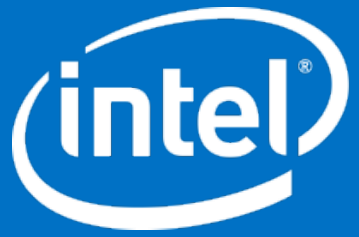


Intel Open-source SYCL compiler project

Konstantin Bobrovskii, Intel

2019 Khronos Embedded Vision Summit

Agenda

SYCL intro

Intel Open-Source SYCL project

Q&A

SYCL Intro

What is SYCL?

Single-source heterogeneous programming using STANDARD C++

- Use C++ templates and lambda functions for host & device code

Aligns the hardware acceleration of OpenCL with direction of the C++ standard

Developer Choice

The development of the two specifications are aligned so code can be easily shared between the two approaches

C++ Kernel Language

Low Level Control

'GPGPU'-style separation of device-side kernel source code and host code

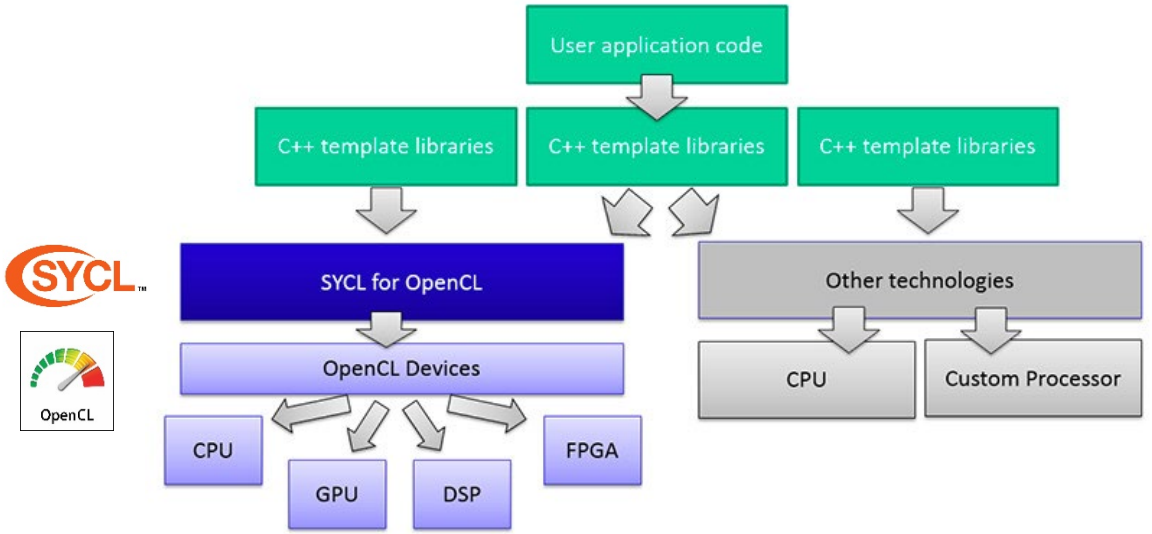
OpenCL

Single-source C++

Programmer Familiarity

Approach also taken by C++ AMP and OpenMP

SYCL™



Why SYCL? Reactive and Proactive Motivation:

Reactive to OpenCL Pros and Cons:

- OpenCL has a well-defined, portable execution model.
- OpenCL is too verbose for many application developers.
- OpenCL remains a C API and only recently supported C++ kernels.
- Just-in-time compilation model and disjoint source code is awkward and contrary to HPC usage models.

Proactive about Future C++:

- SYCL is based on purely modern C++ and should feel familiar to C++11 users.
- SYCL expected to run ahead of C++Next regarding heterogeneity and parallelism. ISO C++ of tomorrow may look a lot like SYCL.
- Not held back by C99 or C++03 compatibility goals.

OpenCL Example (w/ C++ Wrappers):

nstream.cl

```
__kernel void nstream(
    int length,
    double s,
    __global double * A,
    __global double * B,
    __global double * C)
{
    int i = get_global_id(0);
    A[i] += B[i] + s * C[i];
}
```

nstream.cpp

```
cl::Context gpu(CL_DEVICE_TYPE_GPU, &err);
cl::CommandQueue queue(gpu);

cl::Program program(gpu,
    prk::opencl::loadProgram("nstream.cl"), true);

auto kernel = cl::make_kernel<int, double, cl::Buffer,
    cl::Buffer, cl::Buffer>(program, "nstream", &err);

auto d_a = cl::Buffer(gpu, begin(h_a), end(h_a));
auto d_b = cl::Buffer(gpu, begin(h_b), end(h_b));
auto d_c = cl::Buffer(gpu, begin(h_c), end(h_c));

kernel(cl::EnqueueArgs(queue, cl::NDRange(length)),
    length, scalar, d_a, d_b, d_c);
queue.finish();
```

SYCL Example

```
sycl::gpu_selector device_selector;
sycl::queue q(device_selector);

sycl::buffer<double> d_A { h_A.data(), h_A.size() };
sycl::buffer<double> d_B { h_B.data(), h_B.size() };
sycl::buffer<double> d_C { h_C.data(), h_C.size() };

q.submit([&](sycl::handler& h)
{
     auto A = d_A.get_access<sycl::access::mode::read_write>(h);
     auto B = d_B.get_access<sycl::access::mode::read>(h);
     auto C = d_C.get_access<sycl::access::mode::read>(h);

     h.parallel_for<class nstream>(sycl::range<1>{length}, [=] (sycl::item<1> i) {
          A[i] += B[i] + s * C[i];
     });
});
q.wait();
```

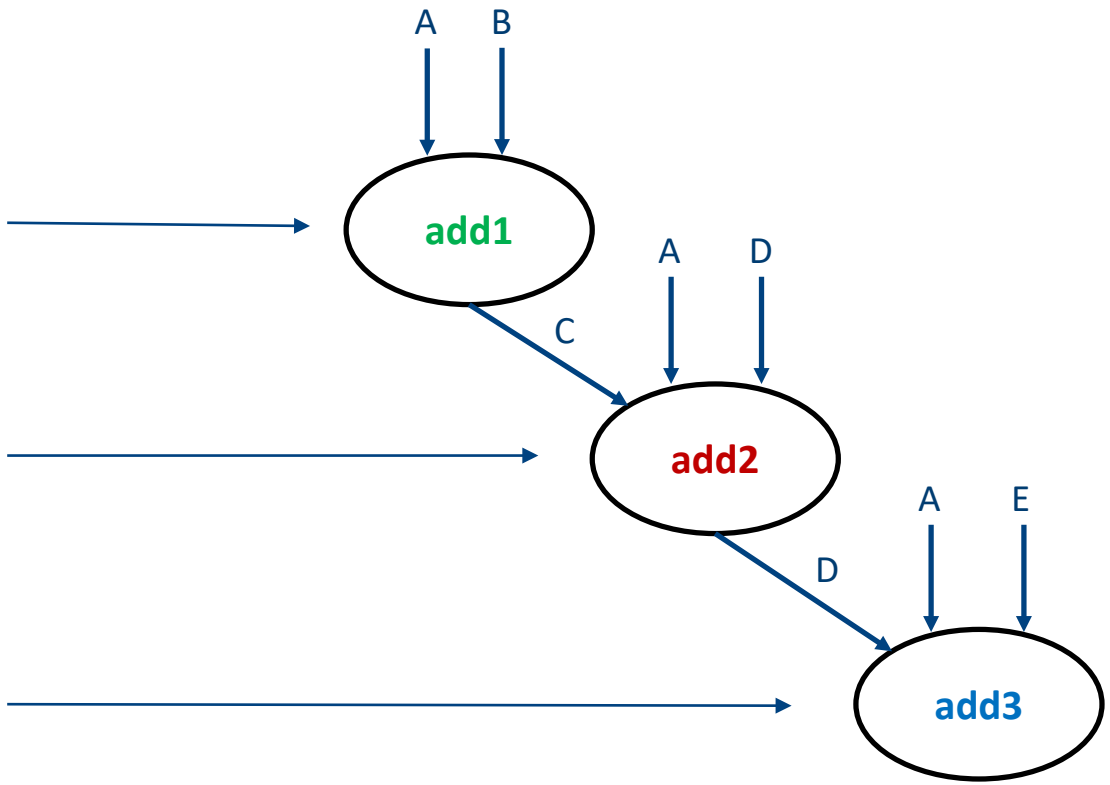
Retains OpenCL's ability to easily target different devices in the same thread.

Accessors create DAG to trigger data movement and represent execution dependencies.

Parallel loops are explicit like C++ vs. implicit in OpenCL.
Kernel code does not need to live in a separate part of the program.

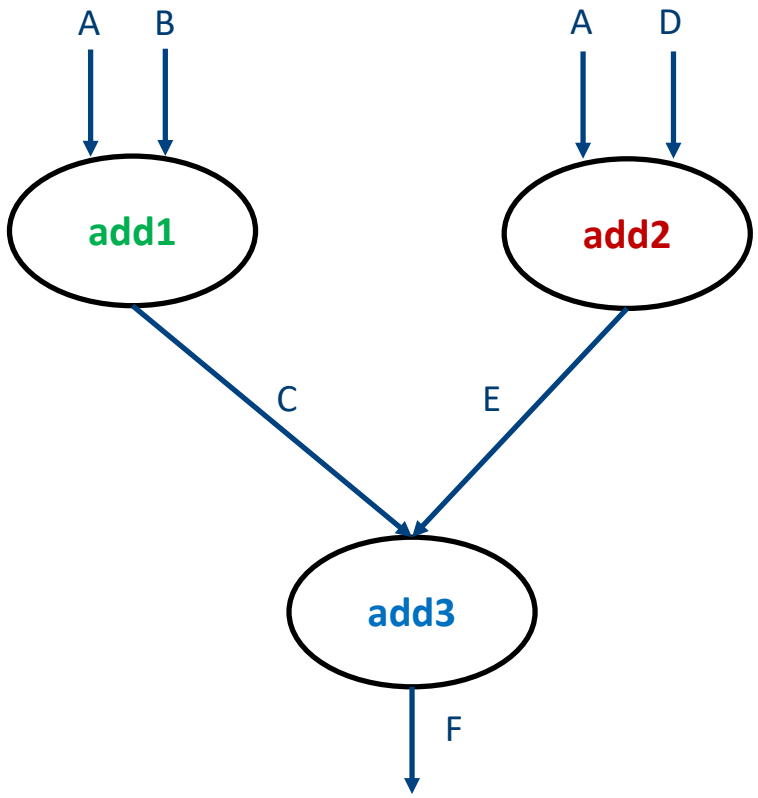
SYCL Example: Graph of Asynchronous Executions

```
 myQueue.submit([&](handler& cgh) {
      auto A = a.get_access<access::mode::read>(cgh);
      auto B = b.get_access<access::mode::read>(cgh);
      auto C = c.get_access<access::mode::discardwrite>(cgh);
      cgh.parallel_for<class add1>( range<2>{N, M},
          [=](id<2> index) { C[index] = A[index] + B[index]; });
});

myQueue.submit([&](handler& cgh) {
      auto A = a.get_access<access::mode::read>(cgh);
      auto C = c.get_access<access::mode::read>(cgh);
      auto D = d.get_access<access::mode::write>(cgh);
      cgh.parallel_for<class add2>( range<2>{P, Q},
          [=](id<2> index) { D[index] = A[index] + C[index]; });
});

myQueue.submit([&](handler& cgh) {
      auto A = a.get_access<access::mode::read>(cgh);
      auto D = d.get_access<access::mode::read>(cgh);
      auto E = e.get_access<access::mode::write>(cgh);
      cgh.parallel_for<class add3>( range<2>{S, T},
          [=](id<2> index) { E[index] = A[index] + D[index]; });
});
```



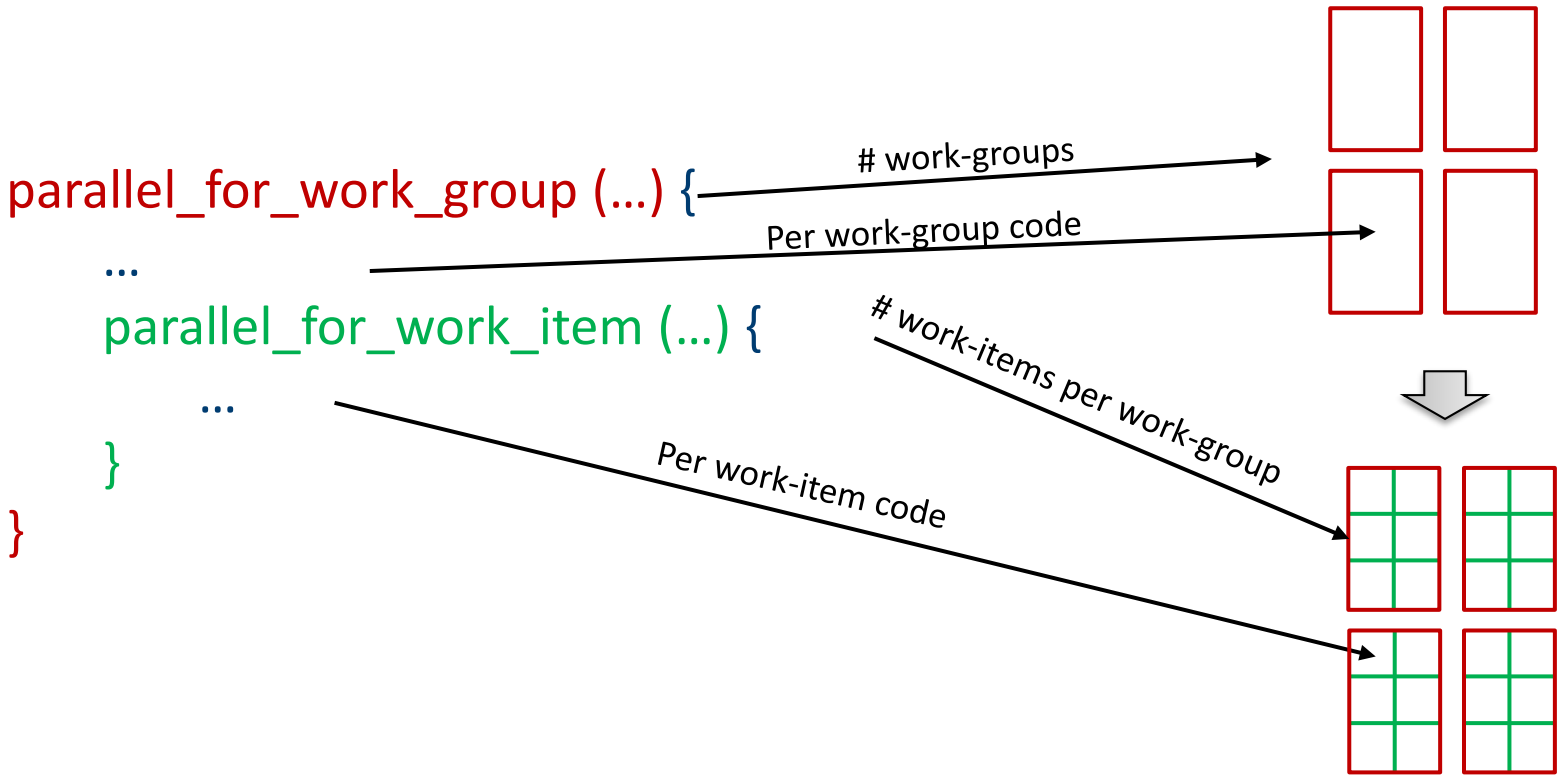
SYCL Example: Graph of Asynchronous Executions

```
 myQueue.submit([&](handler& cgh) {
      auto A = a.get_access<access::mode::read>(cgh);
      auto B = b.get_access<access::mode::read>(cgh);
      auto C = c.get_access<access::mode::discardwrite>(cgh);
      cgh.parallel_for<class add1>( range<2>{N, M},
          [=](id<2> index) { C[index] = A[index] + B[index]; });
});

myQueue.submit([&](handler& cgh) {
      auto A = a.get_access<access::mode::read>(cgh);
      auto D = d.get_access<access::mode::read>(cgh);
      auto E = e.get_access<access::mode::discardwrite>(cgh);
      cgh.parallel_for<class add2>( range<2>{P, Q},
          [=](id<2> index) { E[index] = A[index] + D[index]; });
});

myQueue.submit([&](handler& cgh) {
      auto C = c.get_access<access::mode::read>(cgh);
      auto E = e.get_access<access::mode::read>(cgh);
      auto F = f.get_access<access::mode::discardwrite>(cgh);
      cgh.parallel_for<class add3>( range<2>{S, T},
          [=](id<2> index) { F[index] = C[index] + E[index]; });
});
```



- SYCL queues are out-of-order by default – data dependencies order kernel executions
- Will also be able to use in-order queue policies to simplify porting

Hierarchical parallelism (logical view)



- Fundamentally top down expression of parallelism
- Many embedded features and details, not covered here

Intel Open-Source SYCL Project

Motivation, location, architecture, plugin interface, future directions

Motivation

Why SYCL

- SYCL is an open standard
- Can target any offload device
- Expressive and highly productive

Why Open source

- Promote SYCL as the offload programming model
- Foster collaboration and innovation in the industry

Location and information

Github: https://github.com/intel/llvm/tree/sycl - fork from llvm repo

Primary project goal is contribution to llvm.org:

- RFC: https://lists.llvm.org/pipermail/cfe-dev/2019-January/060811.html

First changes to the clang driver are already committed:
https://reviews.llvm.org/D57768

Detailed plan for upstream: https://github.com/intel/llvm/issues/49

Overview of Changes atop LLVM/Clang

SYCL runtime. Core runtime components, SYCL API definition and implementation

Compilation driver. Two-step host + device compilation, separate compilation and linking, AOT compilation

Front End. Device compiler diagnostics, device code marking and outlining, address space handling, integration header generation,

Tools.

- **clang-offload-bundler**. Generates “fat objects”. Updated to support SYCL offload kind and partial linking to enable “fat static libraries”.
- **clang-offload-wrapper**. New tool to create fat binaries – host + device executables in one file. Portable analog of OpenMP offload linker script.

LLVM. New `sycldevice` environment to customize spir target

SYCL Compiler Architecture

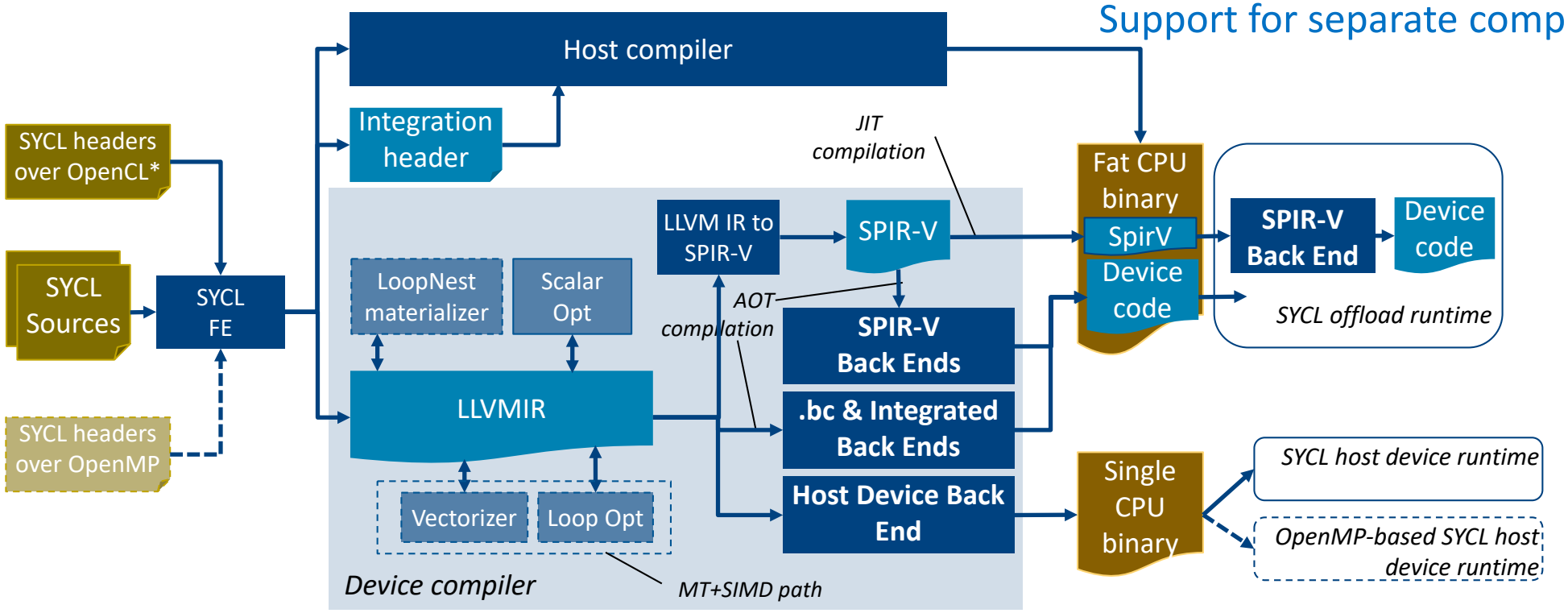
Single host + multiple device compilers

3rd-party host compiler can be used

- Integration header with kernel details

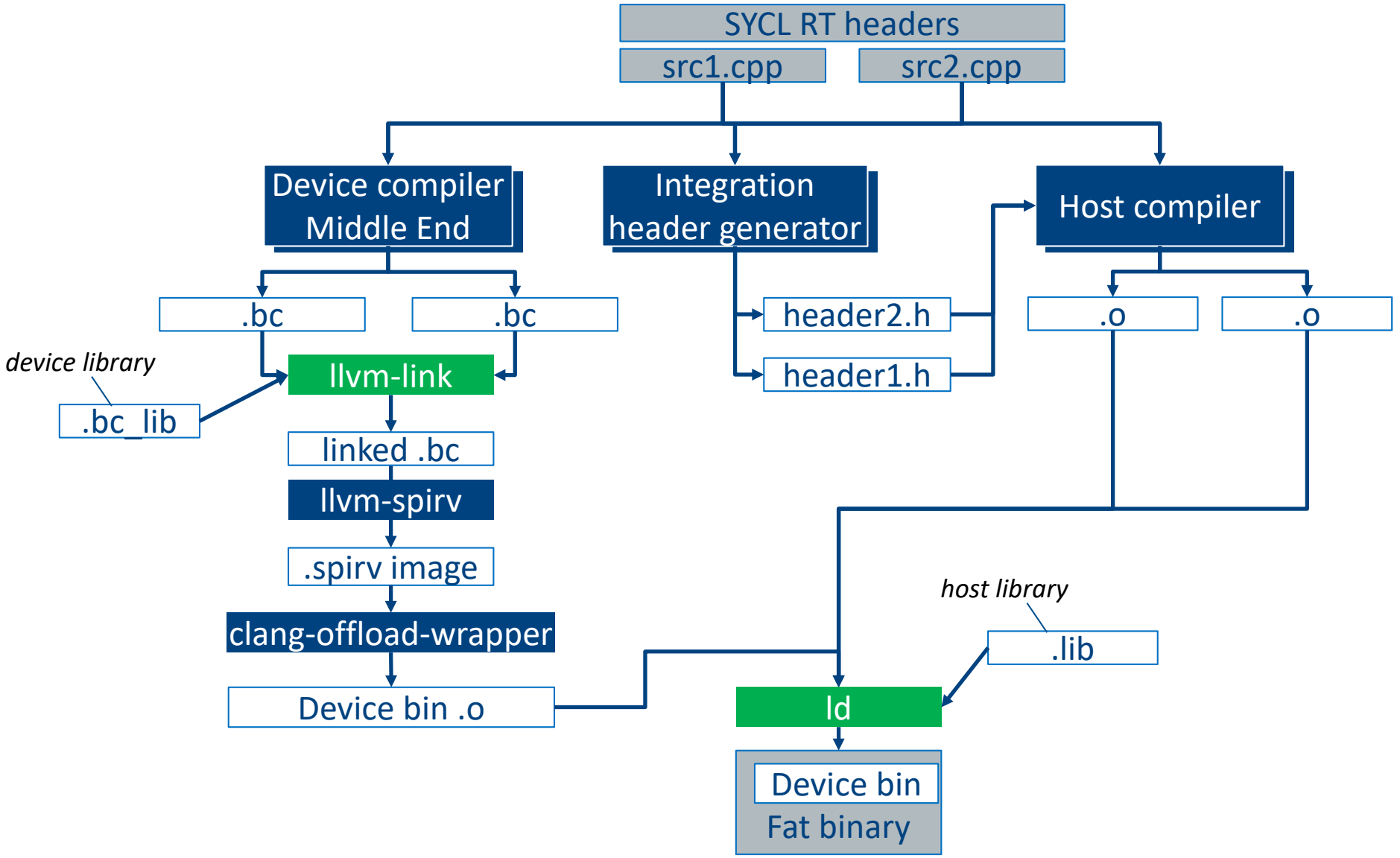
Support for JIT and AOT compilation

Support for separate compilation and linking

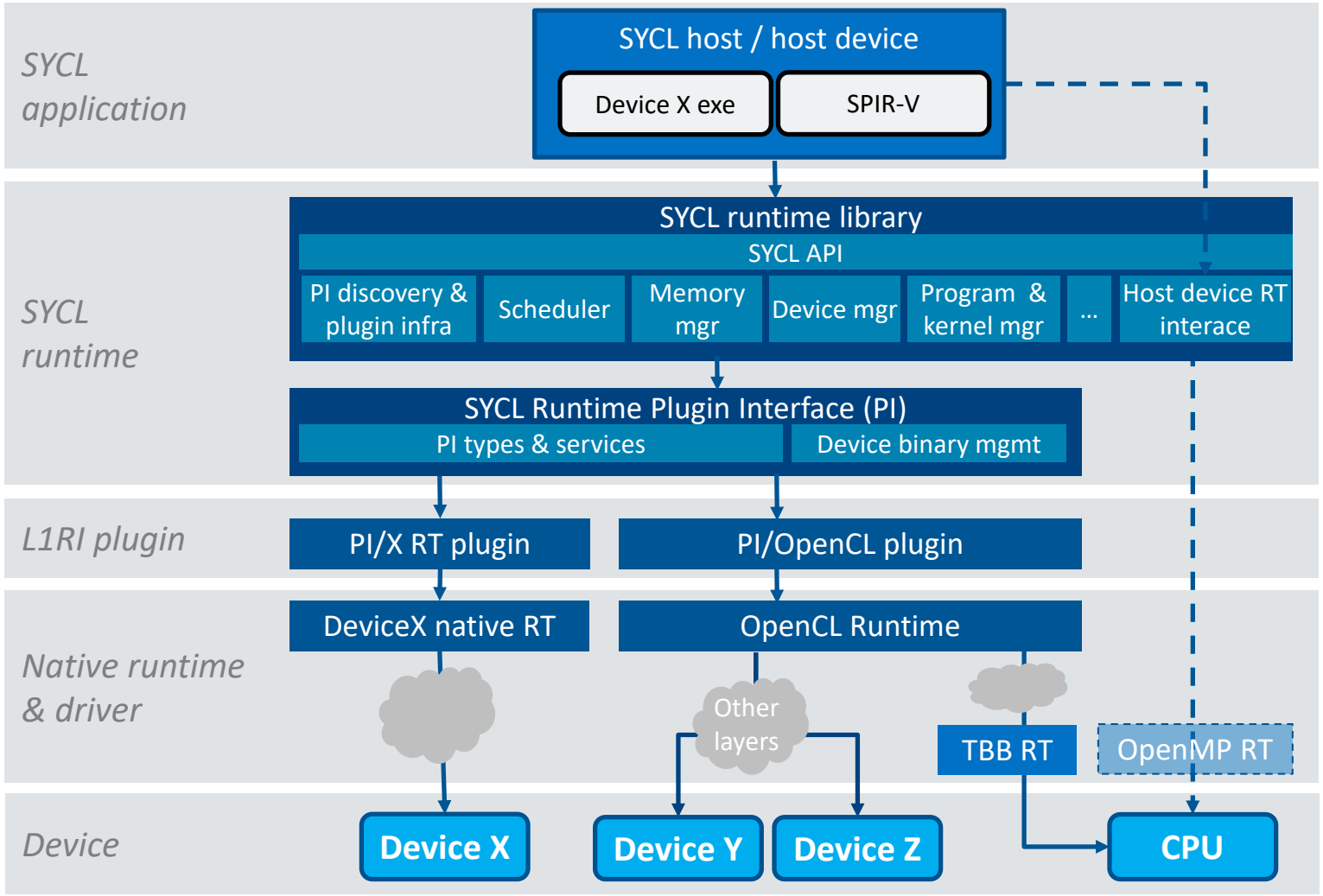


* OpenCL dependence is to be moved under the Plugin Interface

Simplified Compilation Flow for JIT scenario



SYCL Runtime Architecture



Modular architecture

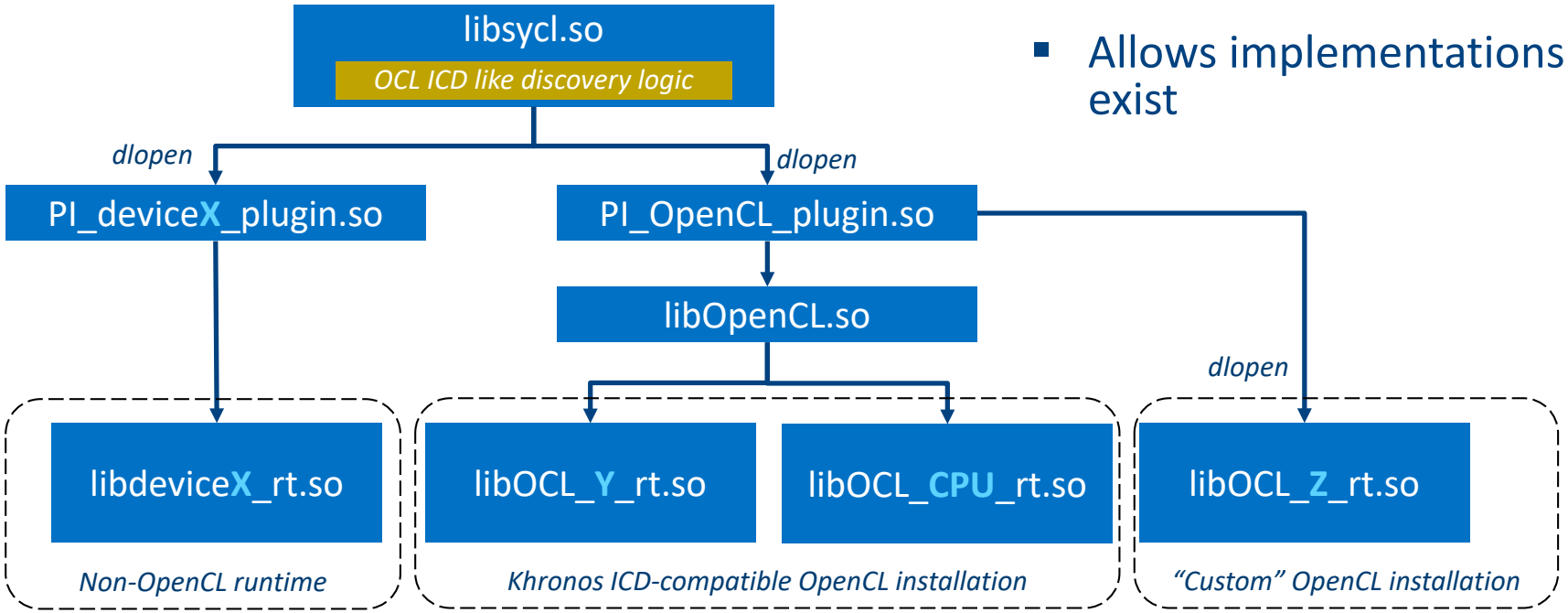
Plugin interface to support multiple back-ends

Support concurrent offload to multiple devices

Support for device code versioning

- Multiple device binaries in a single fat binary

SYCL Runtime Plugin Interface



Defines a programming model atop OpenCL concepts & model

Abstracts away the device layer from the SYCL runtime

- Allows implementations for multiple devices co-exist

Future plans

NDRange subgroups

Ordered queue

Ahead of time compilation

Scheduler improvements

Unified Shared Memory

Hierarchical parallelism

Generic address space

Windows support

Specialization constants

Documentation update

Fat static libraries

Device code distribution per cl::sycl::program

Q&A