Introduction to SYCL

Konstantin Bobrovskii, Intel

October 2019

Agenda

SYCL overview

SYCL programming model

Intel's SYCL implementation

SYCL overview

What is SYCL?

Single-source heterogeneous programming using STANDARD C++ 11

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

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

Developer Choice

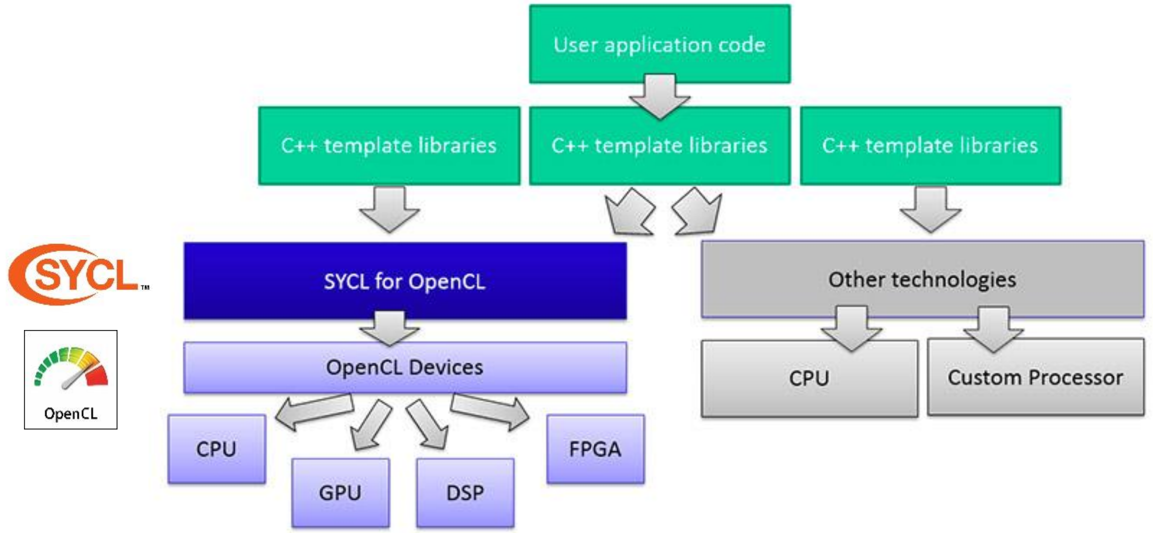
The development of the two specifications are aligned so code can be easily shared between the two approaches

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 source compilation 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.

SYCL C++ device code features

Supported features

+ templates
+ classes
+ operator overloading
+ static polymorphism
+ lambdas
+ pointer structure members (under USM)
+ function pointers (in flight)

Unsupported features

– dynamic memory allocation
– dynamic polymorphism
– runtime type information
– exception handling
– mutable static variables

SYCL programming model

SYCL vector addition example

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main () {
…
  // Device Buffers
  buffer<float, 1> buf_a(array_a, range<1>(count));
  buffer<float, 1> buf_b(array_b, range<1>(count));
  buffer<float, 1> buf_c(array_c, range<1>(count));

  queue myQueue;
  myQueue.submit([&](handler& cgh) {
    // Data accessors
    auto a = buf_a.get_access<access::read>(cgh);
    auto b = buf_b.get_access<access::read>(cgh);
    auto c = buf_c.get_access<access::write>(cgh);

    // Kernel
    cgh.parallel_for<class vec_add>(count, [=](id<> i)
      {
        c[i] = a[i] + b[i];
      }
    );
  });
…
}
```

Buffer Objects

Command Queue

Data Accessors

Kernel Code

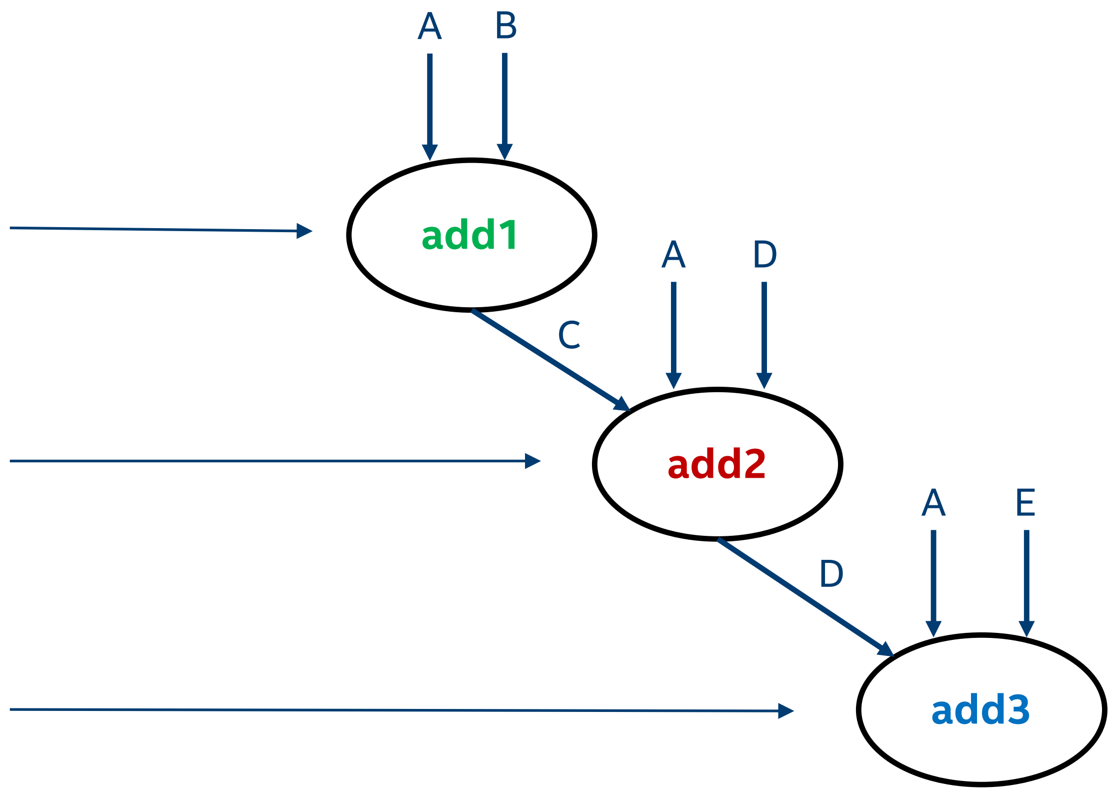
Kernel scope

Command group scope

Application scope

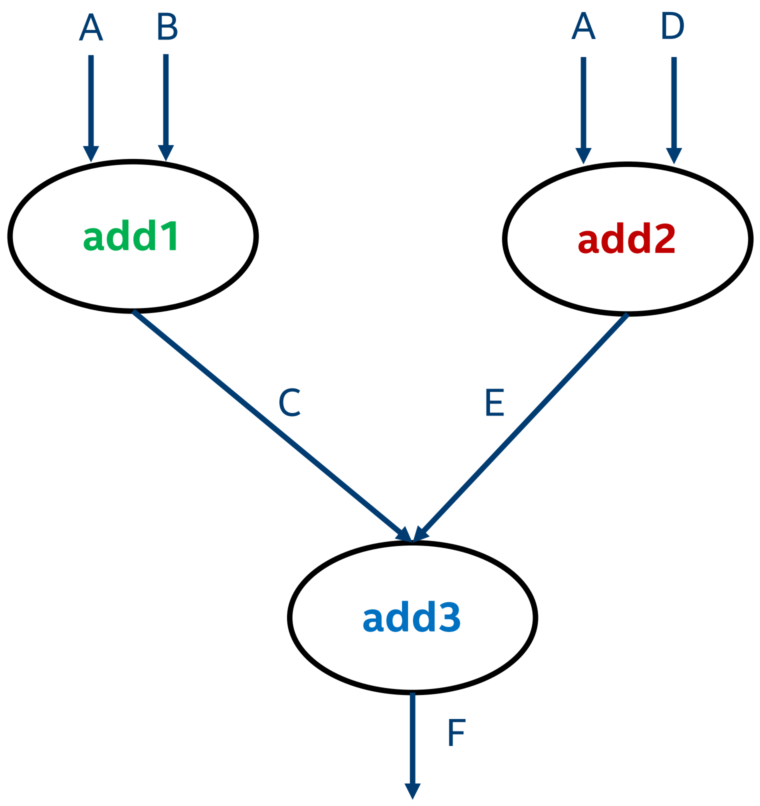
SYCL Example: Graph of Asynchronous Executions

```
 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]; });
});
```



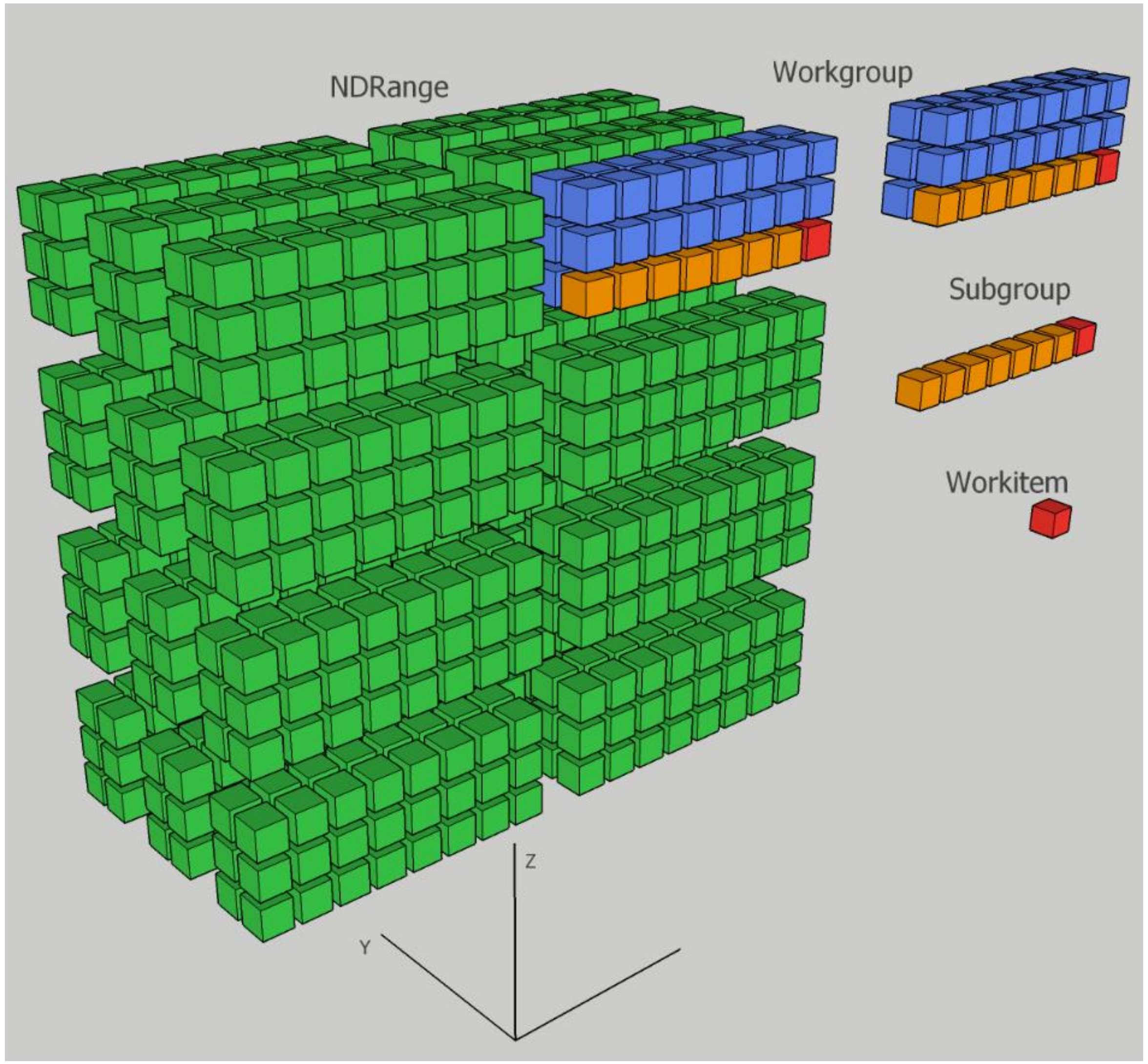
SYCL Example: Graph of Asynchronous Executions

```
 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

SYCL execution model



1,2,3-D index space (NDRange)

- Work item – single element of the index space
- Kernel is invoked for each work item in the NDRange
  - API to query coordinates within NDRange to partition data or specialize execution

2 levels of grouping

- Work group ~ “Block” in CUDA. A team of work-items. Can be 3D
- Sub-group (Intel vendor extension) ~ “Warp” in CUDA
  - Always 1D (along lowest dim)
  - Work items might execute in lock-step
  - Might make IFP with respect to each other

Grouping helps scaling. A grouping level may define

- Synchronization domain – barriers across work items within the group
- Memory scoping – memory shared/accessible only by work items within the group
- Group-wide operations

Memory Model Highlights: memory kinds

Global memory

- accessible to all work-items in all work-groups. Read/write, may be cached, persistent across kernel invocations

Constant memory

- a region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

Local Memory

- shared between work-items in a single work-group and inaccessible to work-items in other work-groups. Example: SLM on Gen

Private Memory

- is a region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item. Example: Register File on Gen

Memory Model Highlights: buffers and images

On host they exist as real objects and can map to multiple device objects

On device – accessors of appropriate kind are used

Buffers – “usual memory”

- element can be of any std layout and trivially copyable type, can get a raw device pointer in device code

Images – “image memory”

- limited set of formats for image elements following popular GPU image formats
- “Special memory” - can’t get a plain pointer to contents in device code
- if mapped to GPU H/W, access can go through faster caches than with buffers.

Parallelism: forms of parallel_for

Simplest – no work groups, flat work-item id

```
cg.parallel_for<class K>(range<1>(numWIs), [=](id<1> index) {
  acc[index] = 42.0f;
});
```

Full – work-item id hierarchical, group operations available

```
cg.parallel_for<class K>(nd_range(range(numWGs), range(wgSize)), [=](nd_item<1> it) {
  acc[it.get_global()] = 42.0f;
  it.barrier(access::fence_space::global);
});
```

Single task – execute kernel in one work-item

```
cg.single_task<class K>([=]() { acc[0] = 43.0f; });
```

Hierarchical parallelism

Hierarchical parallelism (HP) – explicit scopes of parallel code, unlike OpenCL or CUDA

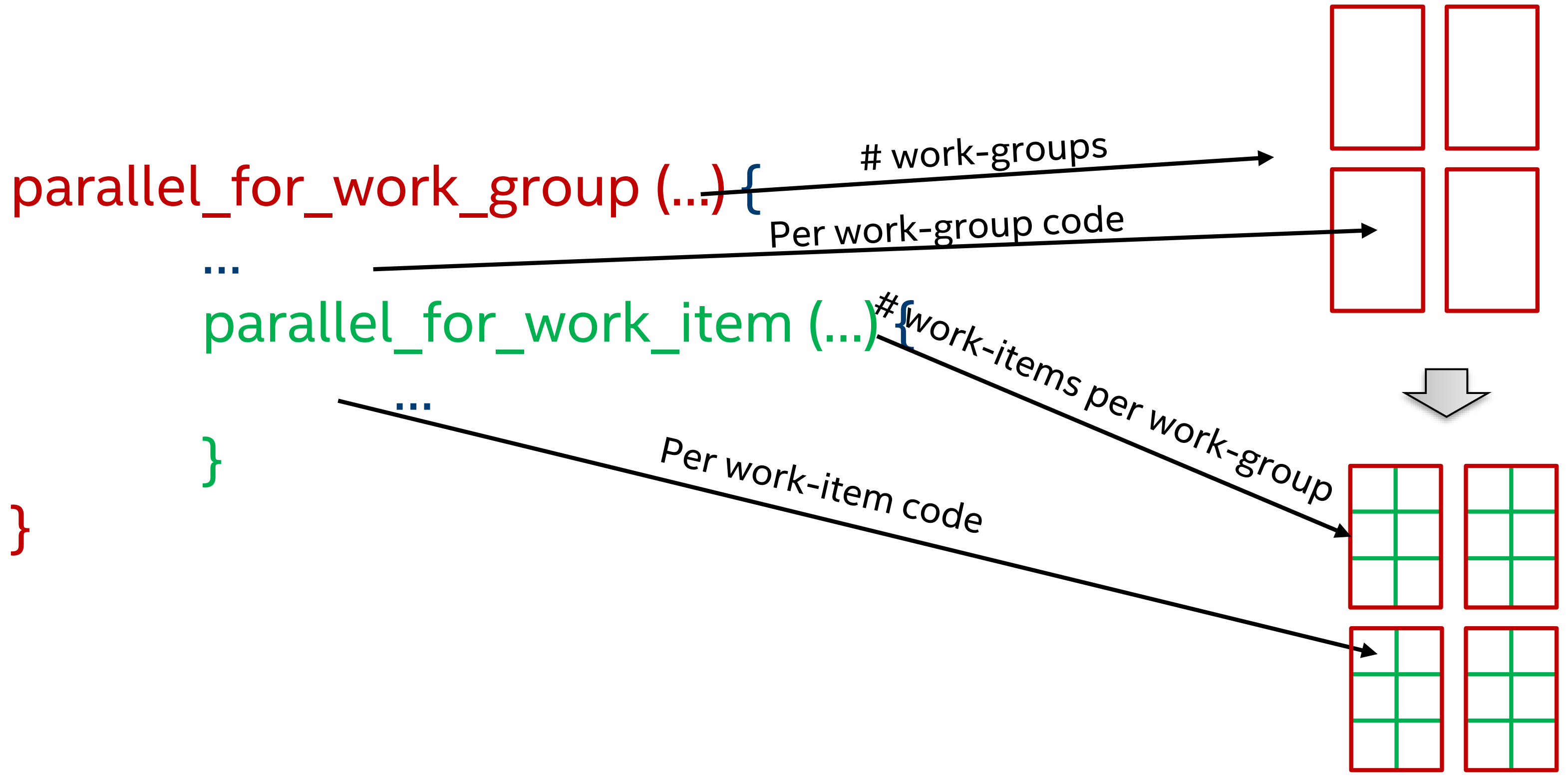
- Data and code semantics vary depending on scope
- Maps more to OpenMP, where scopes are explicit as well

Hierarchy levels:

- Current SYCL has 2: work group and work item
- Future SYCL may generalize to more levels

```
myQueue.submit([&](handler & cgh) {
  // Issue 64 work-groups of 8 work-items each
  cgh.parallel_for_work_group<class example_kernel>(
    range<3>(4, 4, 4), range<3>(2, 2, 2), [=](group<3> myGroup) {
    // [workgroup code]
    int myLocal; // this variable is shared between workitems;
                 // will be instantiated for each work-item separately
    private_memory<int> myPrivate(myGroup);

    // Issue parallel work-items. The number issued per work-group is determined
    // by the work-group size range of parallel_for_work_group. In this case,
    // 8 work-items will execute the parallel_for_work_item body for each of the
    // 64 work-groups, resulting in 512 executions globally/total.
    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
      //[work-item code]
      myPrivate(myItem) = 0;
    });

    // Implicit work-group barrier

    // Carry myPrivate value across loops + “flexible range” for workitems
    myGroup.parallel_for_work_item(range<3>(7, 7, 7), [&](h_item<3> myItem) {
      //[work-item code]
      output[myItem.get_global_id()] = myPrivate(myItem);
    });
    //[workgroup code]
  });
});
```

Hierarchical parallelism (logical view)


parallel_for_work_group (…) {
…
parallel_for_work_item (…) {
…
}
}
# work-groups
Per work-group code
# work-items per work-group
Per work-item code

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

SYCL vs. CUDA VS. OpenCL

SYCL	OpenCL	CUDA
NDRange		Grid
Work group		Block
Subgroup (ext)		Warp
Work item		Thread
USM (ext)	SVM	UM
Auto kernel dependence mgmt. via accessors and buffers	-	Semi-auto (CUDA Graph)
Hierarchical parallelism	-	-
C++ in kernel code	experimental	C++ in kernel code
Support for any device		- (Nvidia only)
Kernel as a lambda	experimental	experimental (--expt-extended-lambda)

"-" means "not supported"

OpenCL interop

SYCL spec requires devices to interoperate with OpenCL and provides APIs

- kernel functions can be defined by traditional OpenCL C kernels
- equivalent OpenCL object can be retrieved from (almost) any SYCL object
  - allowing using it with OpenCL API functions
- SYCL objects feature constructors which take OpenCL objects
  - however in some cases the developer is responsible for maintaining lifetime consistency between OpenCL objects and SYCL objects.

Intel’s SYCL implementation

Intel SYCL Github project & links

SYCL github project – SYCL 1.2.1 + extensions:

- Repo: https://github.com/intel/llvm branch : sycl
- Supports CPU, Intel GPU (Gen8+), Intel FPGA (Arria 10) and Host devices. Any OpenCL 2.0-compatible device should work too. *Contact your Intel rep for details regarding supported hardware.*
- Getting started guide https://github.com/intel/llvm/blob/sycl/sycl/doc/GetStartedWithSYCLCompiler.md

Khronos SYCL resources: https://www.khronos.org/sycl/resources

The latest SYCL language specification: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

Intel extensions to SYCL

- **Unified Shared Memory (USM).** Major productivity tool to avoid manual memory management.

https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/USM/USM.adoc

- **NDRange subgroups.** A performance tool for manual device code vectorization.

https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/SubGroupNDRange/SubGroupNDRange.md

- **Ordered queue.** cl::sycl::queue is out-of-order. cl::sycl::ordered_queue may simplify usage and porting to SYCL from CUDA and OpenCL.

https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/OrderedQueue/OrderedQueue.adoc

- **Unnamed lambdas.** *-fsycl-unnamed-lambda* makes class X in parallel_for<class X> optional. This enables, for example, implementing Kokkos parallel_for construct via SYCL's parallel_for.

- **NDRange reduction.** API for generic reduction across NDRange.

https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/Reduction/Reduction.md

- **Function Pointers (in flight).** SPIRV extension for now, language extension will follow.

https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/SPIRV/SPV_INTEL_function_pointers.asciidoc

Intel SYCL Extension: USM

A pointer has the same representation and refers to the same location on all devices under USM. Automatic data movement between host and devices, direct pointer usage w/o accessors, device memory over-subscription.

4 Levels:

- **Explicit. E**xplicit allocation of device memory and data copying. Device memory not accessible on host.
- **Restricted.** Adds allocation of host and shared memory. No explicit data copying in shared memory, but no concurrent access from host and device.
- **Concurrent.** Adds concurrent access to shared memory. Optionally allows device memory oversubscription.
- **System.** Does not require use of special allocator - malloc'ed memory is covered by USM. Allows oversubscription.

Explicit & Restricted are supported today.

Intel SYCL Extension: USM – Simple Example

```
…
float* a = (float*) sycl::malloc_shared(100*sizeof(float), dev, ctxt);
float* b = (float*) sycl::malloc_shared(100*sizeof(float), dev, ctxt);

for (int i = 0; i < 100; i++) {
  a[i] = func();
}

q.submit([&](handler &cgh) {
  cgh.parallel_for<class foo>([=](id<1> i) {
    b[i] = 3.14 * a[i];
});});


q.wait();

for (int i = 0; i < 100; i++) {
  … = b[i];
}
…
```

- No accessors, direct pointer usage in the kernel
- Special malloc will not be needed in higher USM levels

Intel SYCL Extension: NDRange subgroup

A tool for manual code vectorization. No new scope like `parallel_for_work_item` today – future SYCL will likely add.
SIMD loop approximate example:

```
cgh.parallel_for<class X>(..., [&](nd_item item)
{
  sub_group sg = item.get_sub_group();
  for (int v = sg.get_local_id(); v < N; v += sg.get_local_range()) {
    // use v to index access to per SIMD-lane data
    ... sg.subgroup_api_call(...); ...
  }
});
```

APIs

- local id and range, barrier, any, all, broadcast, reduce, *scan, shuffle*, load, store

Intel SYCL Features

Offline compilation (aka Ahead-Of-Time). Produce native device code at compile time. Intel GPU, CPU and Intel FPGA are supported.

Windows support. Quality already close to Linux.

Separate compilation and linking. Allows the device program to span multiple separately compiled translation units. Interface to build systems remains almost as simple as w/o offload through “fat” objects and “fat” binaries.

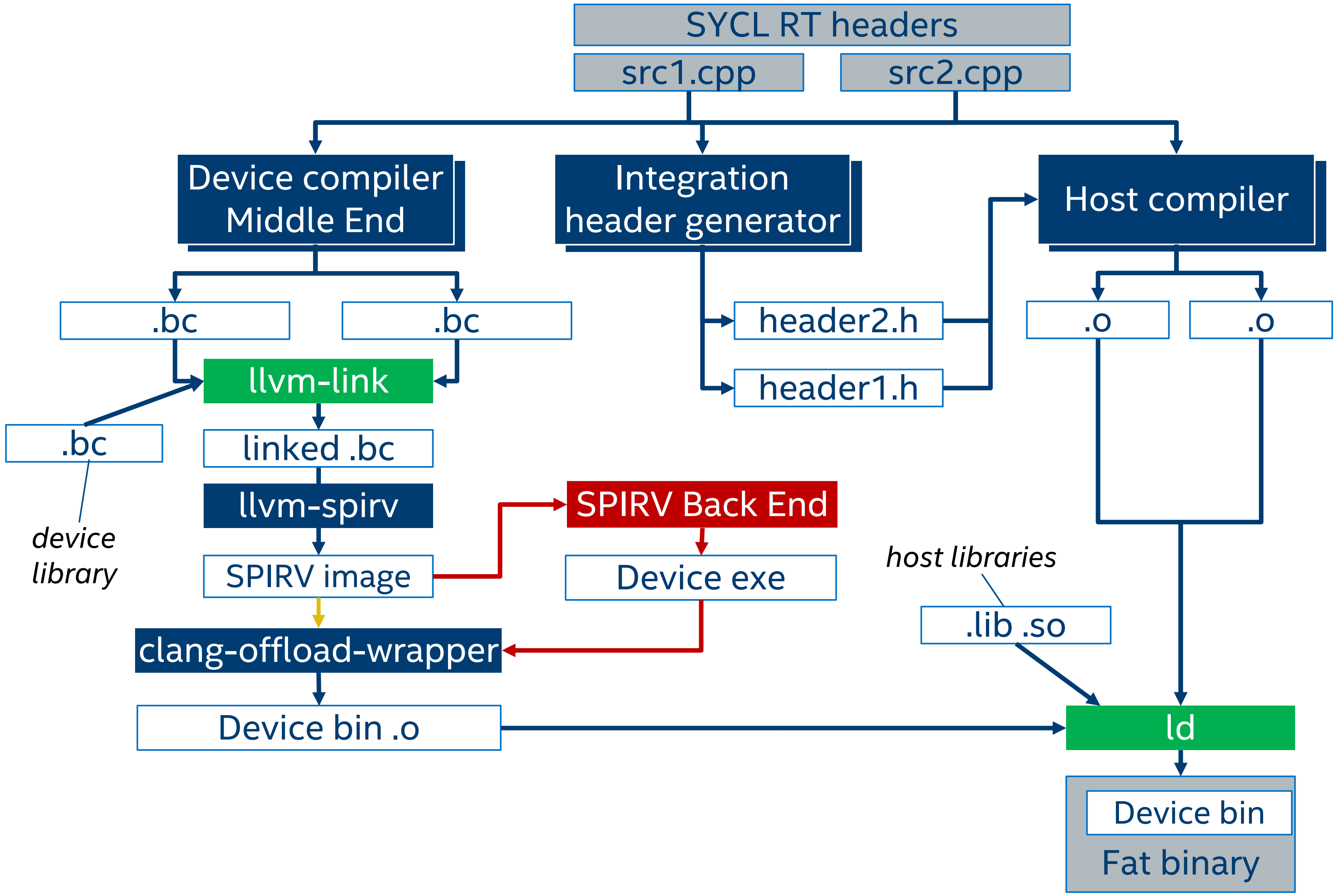
Static device code libraries. Intel SYCL can aggregate host + device code into “fat” static libraries.

Device code distribution per modules. Compiler can auto-split device code into multiple modules to reduce JITting time and binary size (in flight)

Generic pointers. Not visible to the user. Makes it possible to reuse existing C++ libraries in SYCL device code. Requires SPIRV BEs to support generic AS. Language changes are coming too, maybe part of future SYCL spec.

Gcc or 3rd party compiler as a host compiler (in flight)

Intel SYCL Features: Separate Compilation



→ JIT compilation

```
$ clang++ -fsycl \
  src1.cpp src2.cpp -lOpenCL
```

→ AOT compilation

```
$ clang++ \
  -fsycl-targets=spir64_gen-… \
  src1.cpp src2.cpp -lOpenCL
```

Note: the simplest scenario is shown, when compilation and linkage are 1 invocation

General Optimization Tips for Intel GPU

Topic	Tips
EU utilization	Make sure the ND range is big enough to engage all H/W threads, vectorization multiplies the minimum required by vector length.
Divergent code	Avoid it if possible replacing with min/max or compute a predication flag.
Load balancing	Make sure all instances of the kernel execute roughly the same time, otherwise the slowest will keep the device under-utilized.
Conditional execution	Avoid boundary condition checking in the code via handling it outside the kernel. Data structures can be padded.
Dynamic local indexing	Compiler can generate much better code for small local array access if indices are constant or can be known at compile time after unrolling. Avoid data-dependent indexing
Loops	Compiler reduces control flow overhead by unrolling simple loops and inlining simple functions. Avoid complicated loops and functions
Small kernels	It is profitable to make kernel code small enough to fit into the instruction cache. On the other hand, must do enough work to amortize offload overhead.
Data blocking	Reduce global memory accesses by prior fetching frequently accessed data into a small local array. Compiler will try to allocate it on registers.

BACKUP

SYCL Compiler Architecture

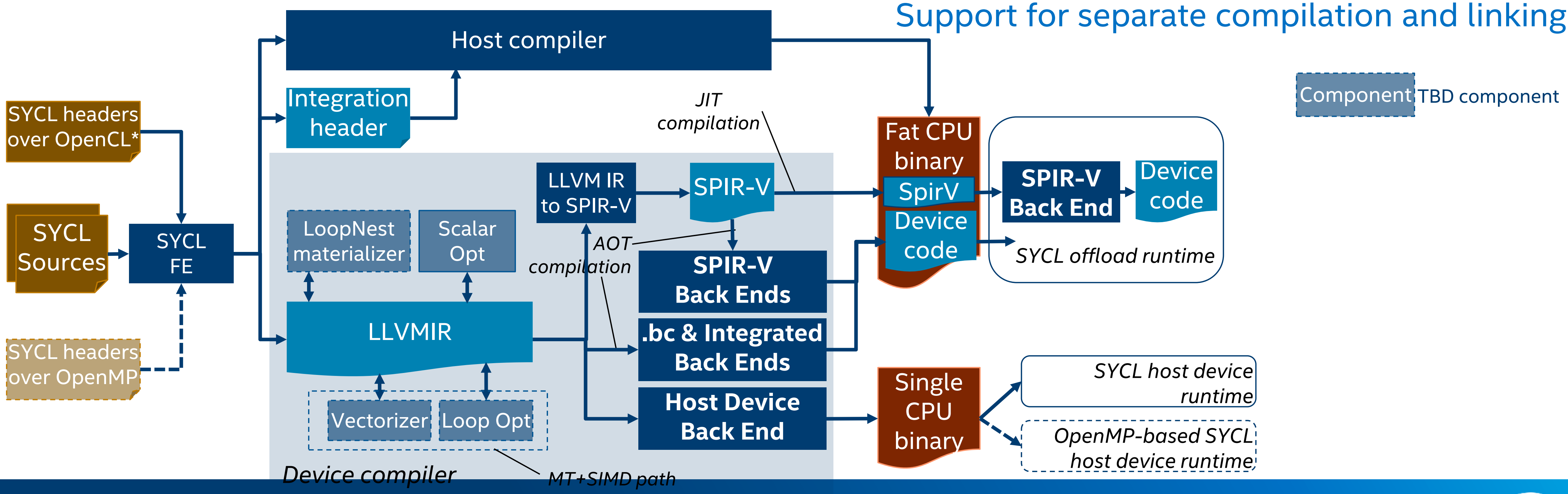
Single host + multiple device compilers

3rd-party host compiler can be used

- Integration header with kernel details

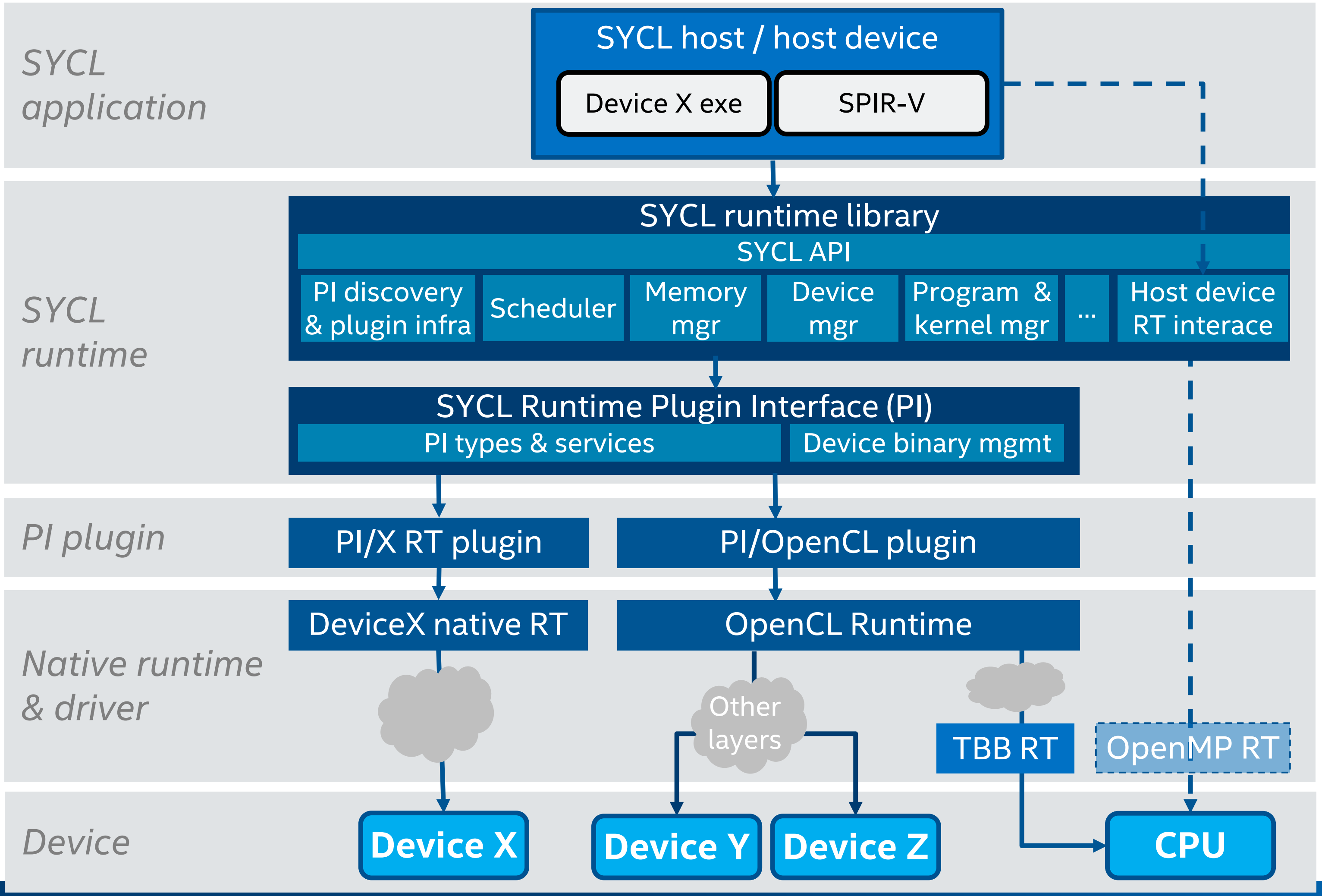
Support for JIT and AOT compilation

Support for separate compilation and linking



* OpenCL dependence is to be moved under the Plugin Interface

SYCL Runtime Architecture



Modular architecture

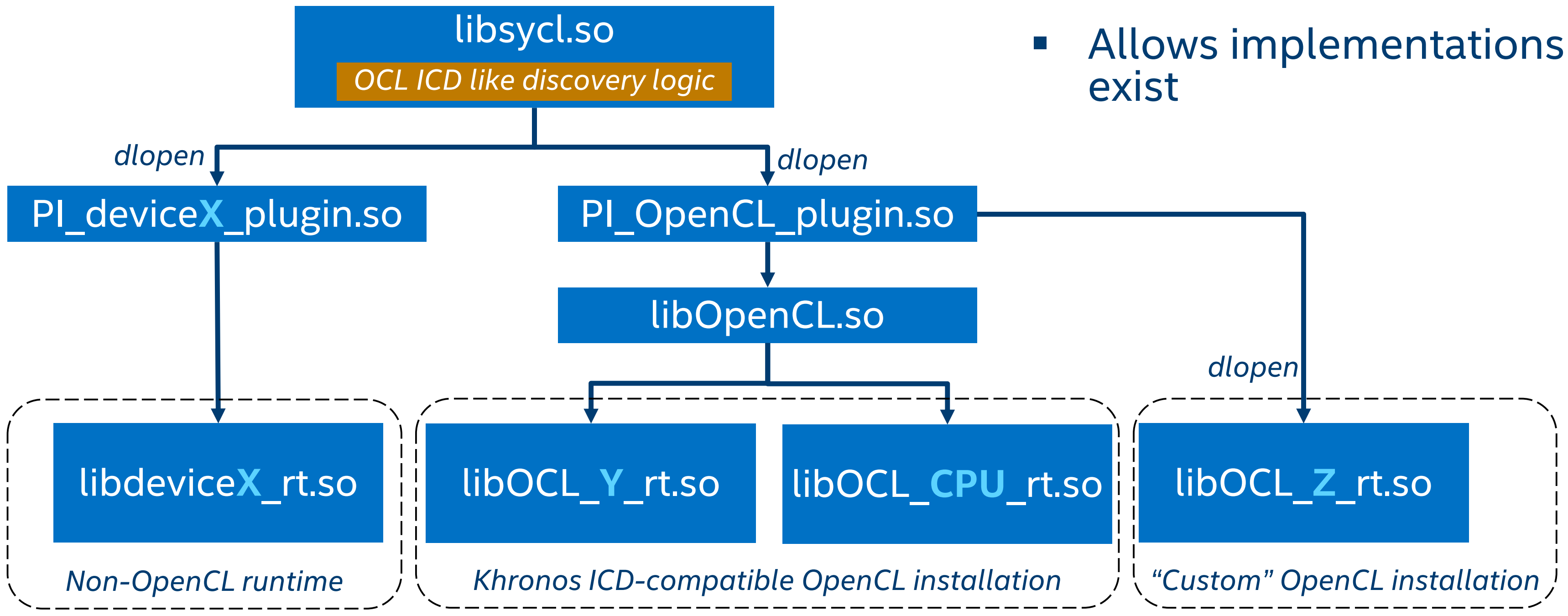
Plugin interface to support multiple back-ends

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

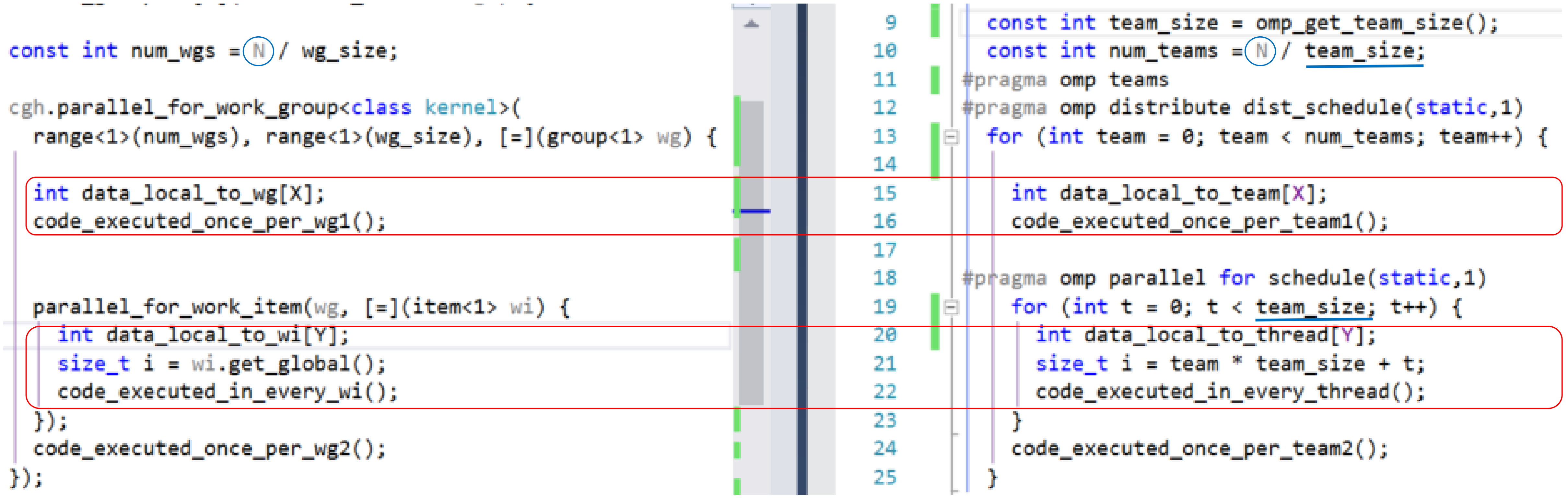
- Allows implementations for multiple devices co-exist

Convert OpenMP offload to SYCL offload

```
#pragma omp declare target
const float coeffs[] = {
  0.2f, 0.3f, 0.3f, 0.0f,
  0.1f, 0.5f, 0.5f, 0.0f,
  0.3f, 0.1f, 0.1f, 0.0f,
  0.0f, 0.0f, 0.0f, 0.0f };
#pragma omp end declare target

#pragma omp declare target
static void sepia_impl(const float* src, float* dst, int i) {
  for (int j = 0; j < 4; ++j) {
    float w = 0.0f;
    for (int k = 0; k < 4; ++k) {
      w += coeffs[4 * j + k] * src[i + k];
    }
    dst[i + j] = w;
  }
}
#pragma omp end declare target

void Sepia::execute_offload(float* image) {
  float* src_image = this->src_image;
#pragma omp target map(to: src_image[0:IMG_SIZE*4]) \
                   map(from: image[0:IMG_SIZE*4])
#pragma omp parallel for
  for (int i = 0; i < IMG_SIZE * 4; i += 4) {
    sepia_impl(src_image, image, i);
  }
}
```

```
static void sepia_impl(float *src, float *dst, int i) {
  const float coeffs[] = {
    0.2f, 0.3f, 0.3f, 0.0f,
    0.1f, 0.5f, 0.5f, 0.0f,
    0.3f, 0.1f, 0.1f, 0.0f,
    0.0f, 0.0f, 0.0f, 0.0f };
  i *= CHANNELS_PER_PIXEL;
  for (int j = 0; j < 4; ++j) {
    float w = 0.0f;
    for (int k = 0; k < 4; ++k) {
      w += coeffs[4 * j + k] * src[i + k];
    }
    dst[i + j] = w;
  }
}
void Sepia::execute_offload(float* image) {
  MyGpuSelector sel(pattern);
  queue q(sel);
  buffer<float, 1> image_buf(src_image, range<1>(IMG_SIZE));
  buffer<float, 1> image_buf_exp(image, range<1>(IMG_SIZE));
  q.submit([&](handler& cgh) {
    auto src = image_buf.get_access<sycl_read>(cgh);
    auto dst = image_buf_exp.get_access<sycl_write>(cgh);
    cgh.parallel_for<class s>(range<1>(IMG_SIZE), [=](id<1> i) {
      sepia_impl(src.get_pointer(), dst.get_pointer(), i.get(0));
    });
  });
}
```

1D Parallelism example – loose mapping to OpenMP



```
const int num_wgs = N / wg_size;

cgh.parallel_for_work_group<class kernel>(
  range<1>(num_wgs), range<1>(wg_size), [=](group<1> wg) {

  int data_local_to_wg[X];
  code_executed_once_per_wg1();


  parallel_for_work_item(wg, [=](item<1> wi) {
    int data_local_to_wi[Y];
    size_t i = wi.get_global();
    code_executed_in_every_wi();
  });
  code_executed_once_per_wg2();
});
```

```
  const int team_size = omp_get_team_size();
  const int num_teams = N / team_size;
#pragma omp teams
#pragma omp distribute dist_schedule(static,1)
  for (int team = 0; team < num_teams; team++) {

    int data_local_to_team[X];
    code_executed_once_per_team1();

#pragma omp parallel for schedule(static,1)
    for (int t = 0; t < team_size; t++) {
      int data_local_to_thread[Y];
      size_t i = team * team_size + t;
      code_executed_in_every_thread();
    }
    code_executed_once_per_team2();
  }
```

collapse(n) can be used to express
n-dimensional ||sm in OpenMP