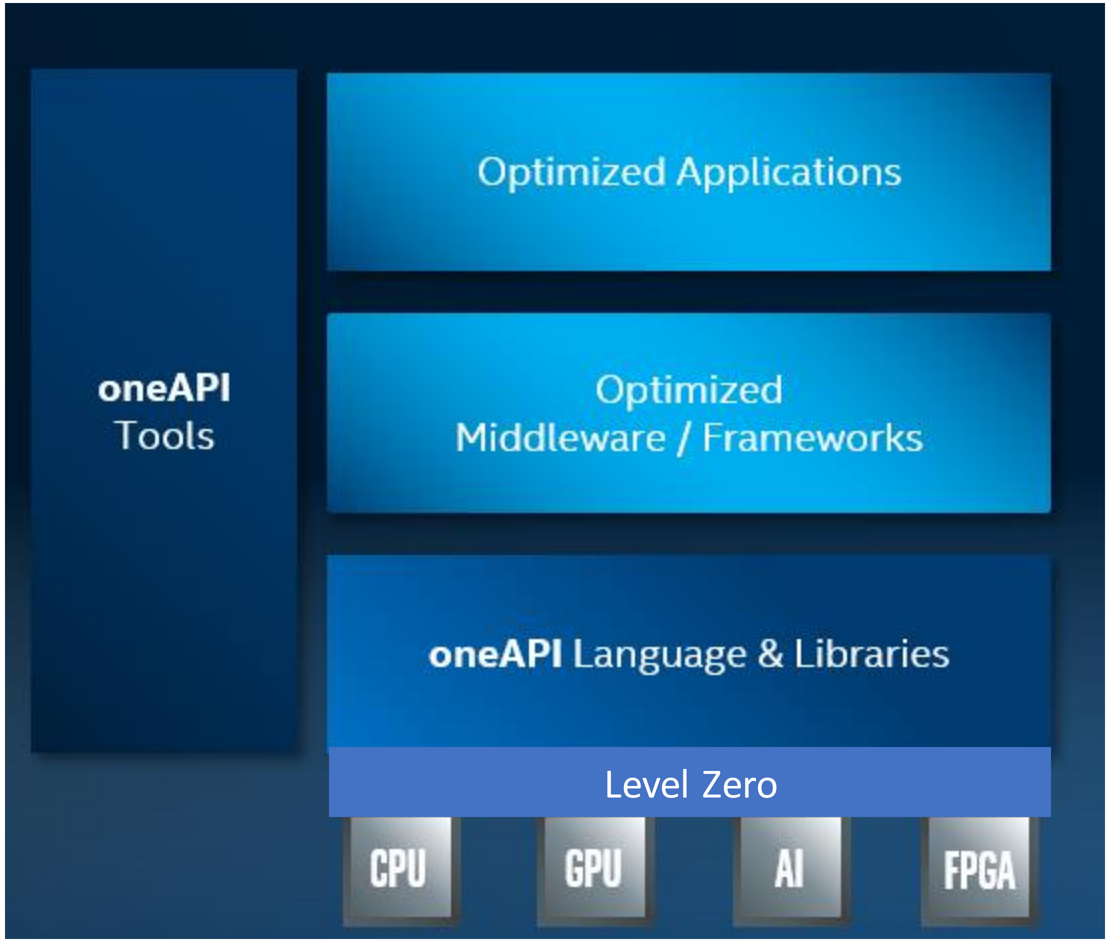
Why DPC++?

- Data Parallel C++

  = ISO C++ and Khronos SYCL and extensions

- Vision:
  - Fast moving open collaboration feeding into SYCL
    - Open source implementation
    - Goal to become upstream LLVM
  - DPC++ extensions aim to become core SYCL, or Khronos extensions
  - DPC++ supports the broader oneAPI ecosystem of standards, including libraries and tooling



oneAPI Tools

Optimized Applications

Optimized Middleware / Frameworks

oneAPI Language & Libraries

Level Zero

CPU

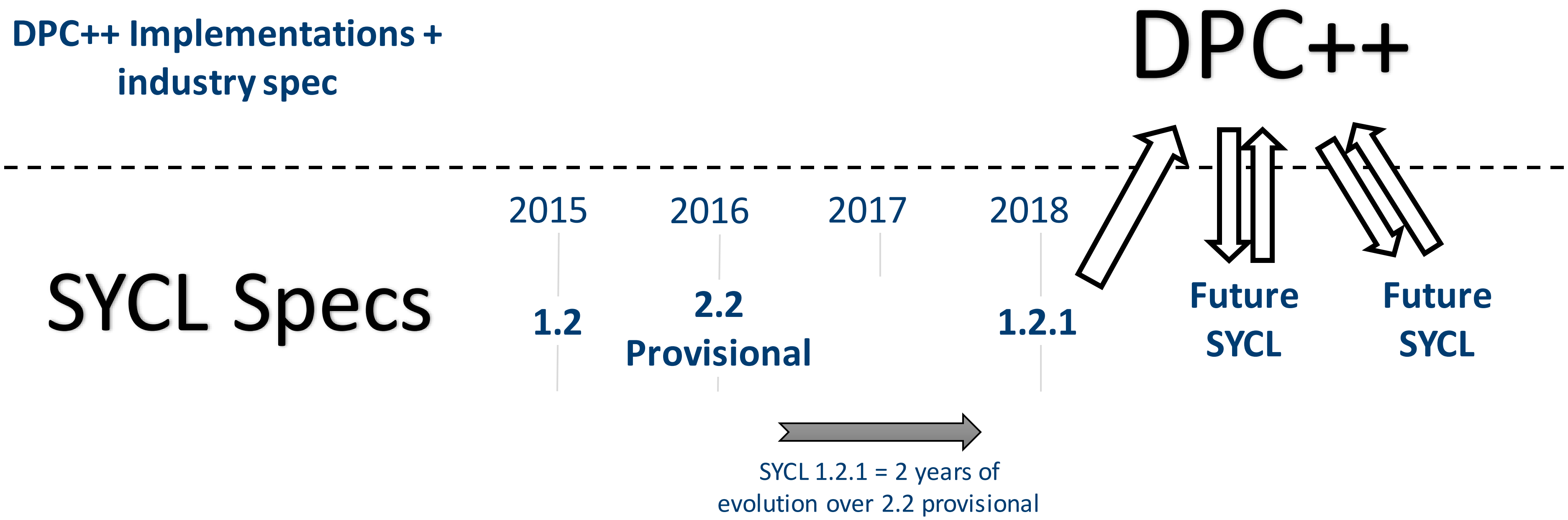
GPU

AI

FPGA

oneAPI

Ongoing relationship: DPC++ and SYCL



DPC++ Implementations + industry spec

DPC++

SYCL Specs

2015 2016 2017 2018

1.2

2.2 Provisional

1.2.1

Future SYCL

Future SYCL

SYCL 1.2.1 = 2 years of evolution over 2.2 provisional

Today: DPC++ Extensions over SYCL

Evolving landscape

- SYCL 1.2.1 is public
- A number of published extensions on Intel GitHub
  - DPC++ open source project building first implementation

Extension	Purpose
USM (Unified Shared Memory)	Pointer-based programming
Sub-groups	Cross-lane operations
Reductions	Efficient parallel primitives
Work-group collectives	Efficient parallel primitives
Pipes	Spatial data flow support
Argument restrict	Optimization
Optional lambda name for kernels	Simplification
In-order queues	Simplification
Class template argument deduction and simplification	Simplification

Unified Shared Memory (USM)

- SYCL 1.2.1 provides a buffer abstraction for memory
  - Powerful and elegantly expresses data dependences
- However…
  - Replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers
- USM provides a pointer-based alternative in DPC++
  - Simplifies porting to an accelerator
  - Gives programmers the desired level of control
  - Complementary to buffers

USM: Allocations and pointer handles

Type	Description	Accessibly By		Migratable To	
Device	Device allocation	Host	✗	Host	✗
		Device	✓	Device	✗
		Other device	?	Other device	✗
Host	Host allocation	Host	✓	Host	✗
		Any device	✓	Device	✗
Shared	Migrating allocation	Host	✓	Host	✓
		Device	✓	Device	✓
		Other device	?	Other device	?

```
auto A = (int*)malloc_shared(N*sizeof(int), …);
auto B = (int*)malloc_shared(N*sizeof(int), …);
…

q.submit([&](handler& h) {
  h.parallel_for(range<1>{N}, [=] (id<1> ID) {
    auto i = ID[0];
    A[i] *= B[i];
  });
});
```

USM: Dependencies

- **Explicit Scheduling**
  - Submitting a kernel returns an event
  - Wait on events to order tasks

```
auto E = q.submit([&] (handler& h) {
  auto R = range<1>{N};
  h.parallel_for(R, [=] (id<1> ID) {
    auto i = ID[0];
    C[i] = A[i] + B[i];
  });
});
E.wait();
```

- **DPC++ graph scheduling**
  - Building DAGs from events

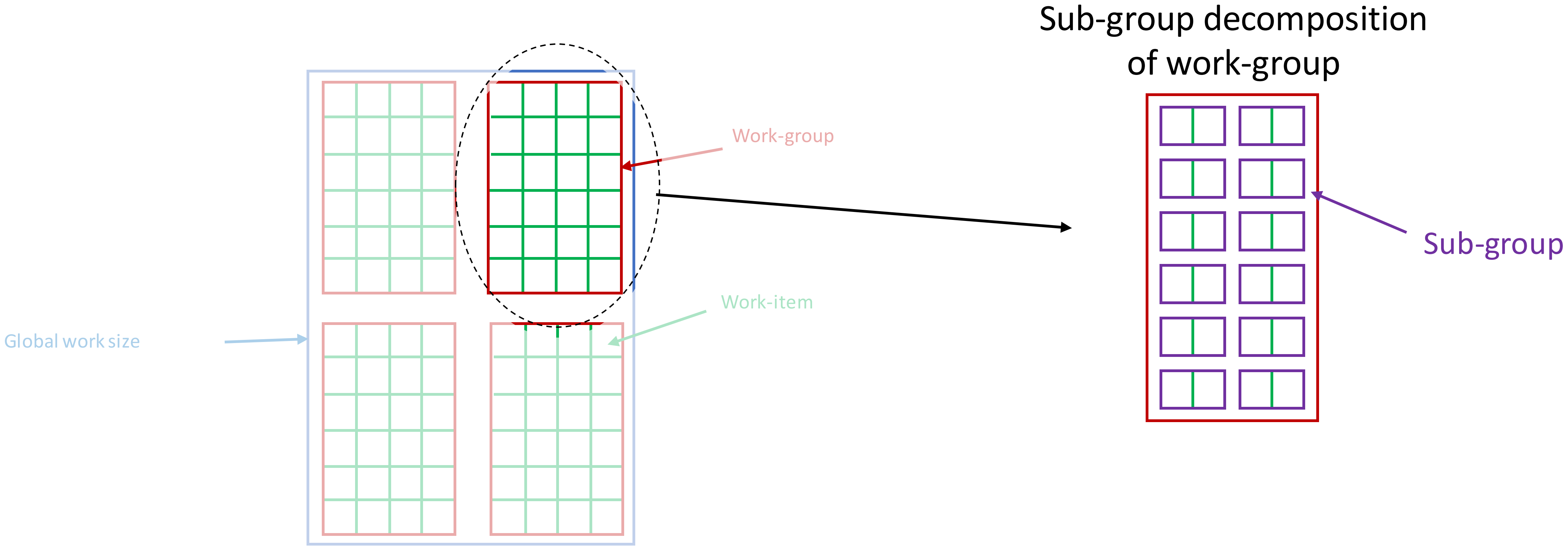
```
auto R = range<1>{N};

auto E = q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&] (handler& h) {
  h.depends_on(E);
  h.parallel_for(R, [=] (id<1> ID) {…});
});
```

Sub-groups

- Additional grouping of work-items, corresponding to SIMD and similar hardware mappings



Sub-group Collectives

- Key cross-lane collectives
  - Broadcast
  - Shuffle
  - Barrier
  - Load/Store
  - Reduce/Scan
  - Vote/Ballot

```
// Compute a histogram of indices using atomics
h.parallel_for(nd_range<1>{N}, [=](nd_item<1> it) {
  // Get handle to this item's sub-group
  dpcpp::sub_group sg = it.get_sub_group();

  // Load the (index, value) pair for this item
  auto i = it.get_global_id(0);
  int idx = index[i]; int x = values[i];

  // If all elements in the group are the same,
  // use a sub-group reduction and one atomic
  if (sg.all(idx == sg.broadcast(idx, 0))) {
    int sum = sg.reduce(x, std::plus<>());
    if (sg.get_local_id() == 0) {
      histogram[idx].fetch_add(sum);
    }
  }
  // Otherwise, use an atomic for each work-item in the sub-group
   else if (sg.any(idx != sg.broadcast(idx, 0))) {
    histogram[idx].fetch_add(x);
  }
});
```

Reductions

- Common parallel pattern
  - Non-trivial to code across architectures, problem sizes
  - User reduction operators

```
// Compute a dot-product by reducing all computed values using standard plus functor
Q.submit([&](handler&  h) {
   auto a = a_buf.get_access<access::mode::read>(h);
   auto b = b_buf.get_access<access::mode::read>(h);
   auto sum = accessor<int,0,access::mode::write,access::target::global_buffer>(sum_buf,  h);

   cgh.parallel_for(nd_range<1>{N,  M}, reduction(sum,  0, plus<int>()),  [=](nd_item<1>  it, auto& sum) {
      int i = it.get_global_id(0);
      sum += (a[i] * b[i]);
   });
});
```

Work-group Collectives

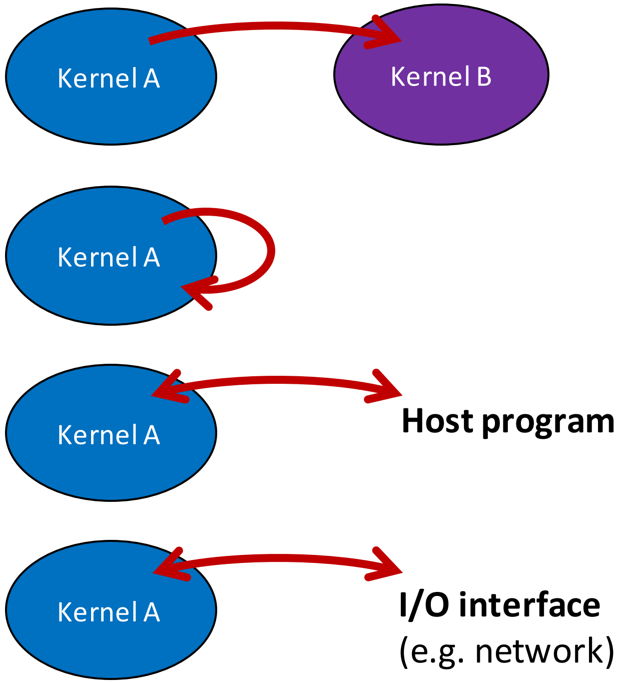
- Additional work-group scope collective operations
  - Broadcast
  - Vote
  - Ballot
  - Reduce
  - Scan

```
template <typename T>
T broadcast(T x, id<1> local_id) const
```

```
bool any(bool predicate) const
```

Data Flow Pipes

- Data with control sideband
  - Fine-grained information transfer and synchronization
  - Important on spatial architectures
- Required for FPGA, optional feature otherwise



Kernel A

Kernel B

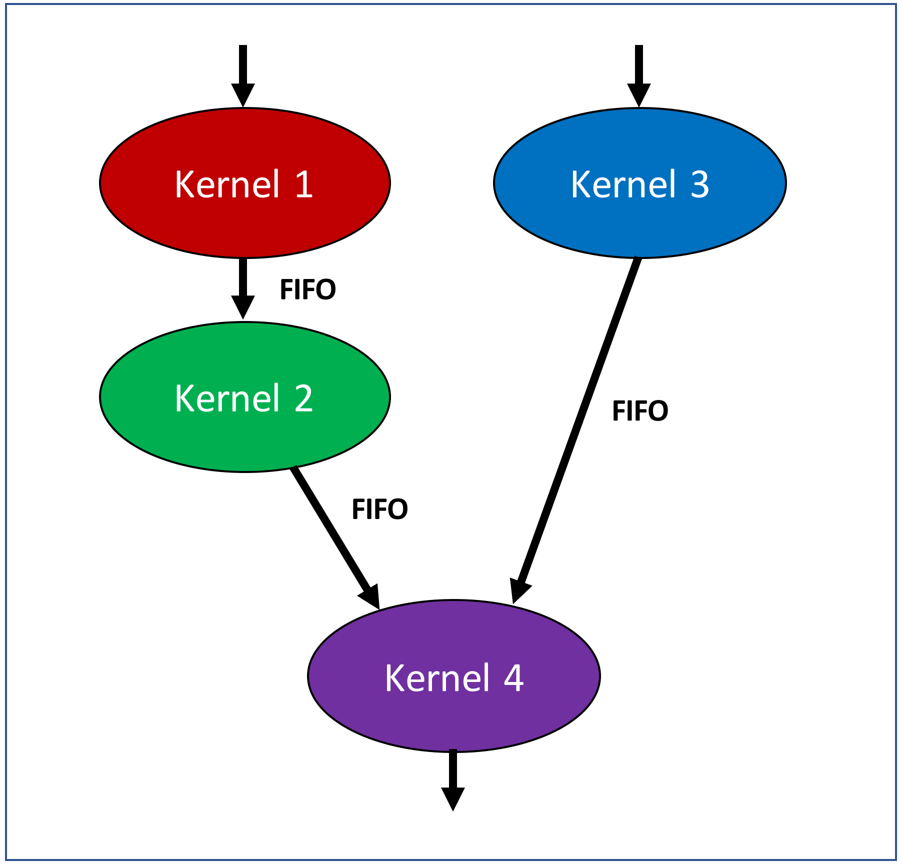
Kernel A

Kernel A

Host program

Kernel A

I/O interface
(e.g. network)



Kernel 1

Kernel 3

FIFO

Kernel 2

FIFO

FIFO

Kernel 4

Kernel Argument Restrict

- C99 restrict-like functionality sometimes critical for performance
  - No obvious place to attach restrict to kernel args (lambdas, functors)
  - ISO C++ hasn't agreed on equivalent to C99 restrict (alias sets, etc)

Lambda

```
cgh.parallel_for<class lambda_foo>(
        range<1>(N), [=](id<1> wiid) [[dpcpp::kernel_args_restrict]] {
        int id = wiid[0]; acc1[id]=id; acc2[id]=id*2;
 });
```

Functor

```
class functor_foo {
    ... void operator()(item<1> item) [[dpcpp::kernel_args_restrict]] {
        int id = item[0]; buf1_m[id]=id; buf2_m[id]=id*2;
  } };
```

Optional Lambda Naming

- Separate compilation led SYCL to require naming of lambdas
  - Too verbose, and a problem for libraries
- DPC++ extension makes lambda names optional
  - Explicit names still useful when debugging

From:

```
h.parallel_for<class my_kernel_name>(R, [=](id<1> idx) { writeResult[idx] = idx[0]; });
```

To:

```
h.parallel_for                      (R, [=](id<1> idx) { writeResult[idx] = idx[0]; });
```

In-order Queues

- DPC++ queues are Out-Of-Order
  - Allows expressing complex DAGs
- Linear task chains are common
  - DAGs are overkill and add verbosity
- Simple things should be simple to express
  - In-order semantics express the linear task pattern easily

```
// With Ordered Queues
ordered_queue q;
auto R = range<1>{N};

q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});
```

Other verbosity reductions

- Enabling class template argument deduction (CTAD)

SYCL 1.2.1:

```
void foo() {
  int arr[64];
  buffer<int,2> B(arr,range<2>(8,8));
}
```

Redundant encoding

With DPC++ extension:

```
void foo() {
  int arr[64];
  buffer B(arr, range(8,8));
}
```

- Generally working to simplify
  - Combination of additions and using newer C++ features