Data Parallel C++
TAB Meeting #2

oneAPI Technical Advisory Board - January 28, 2020

Agenda

1. Review from SC19 TAB meeting
2. Group collectives library direction
3. Simplifying interface for common patterns

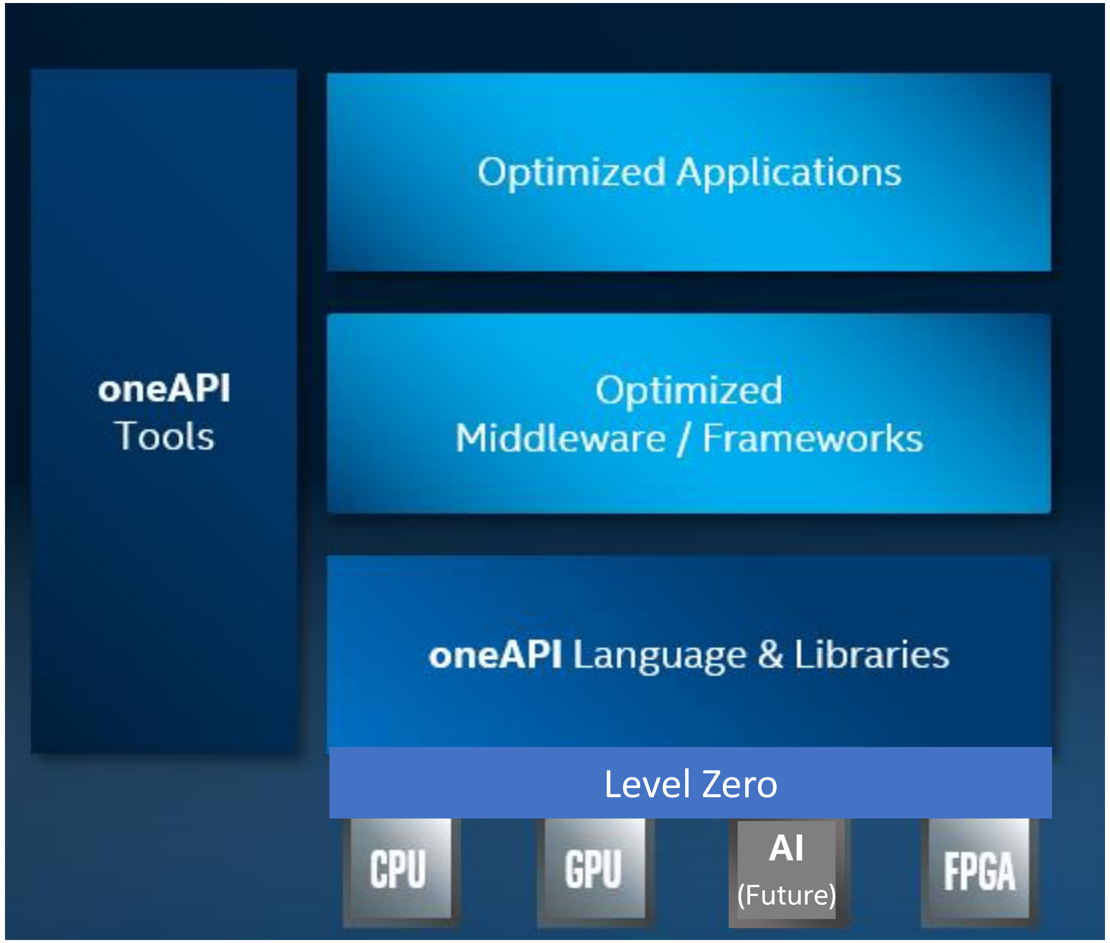
1. Review after SC19 TAB Meeting

oneAPI Technical Advisory Board - January 28, 2020

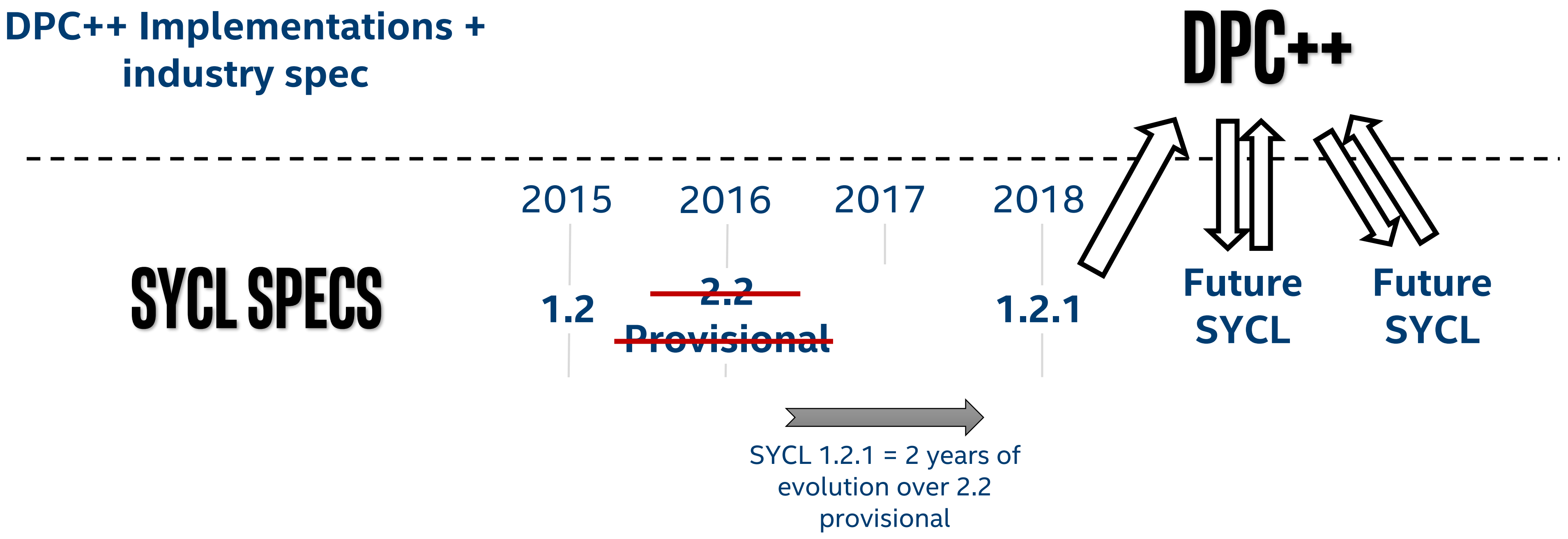
Why DPC++?

- 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



Ongoing relationship: DPC++ and SYCL



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
Invocation directly on queue	Simplification
CTAD deduction guides	Simplification
Required WG size	Optimization

Follow up from SC19 Meeting

1. Deterministic reductions
   - Will be adding control for varying scopes of determinism. Some other changes prioritized first
   - Will broadcast when ready for feedback

2. C++ version in SYCL and DPC++
   - We have pushed into next SYCL spec a process for minimum C++ version bumping
   - Open source DPC++ implementation is ToT LLVM - supports very modern C++

3. "Why is the base OpenCL version 1.2 instead of 2.0?"
   - The Khronos SYCL working group has made public two proposals for the next SYCL spec
   - One of these describes generalization of the backend used by SYCL, including OpenCL version
   - https://github.com/KhronosGroup/SYCL-Shared/tree/master/proposals

Follow up from SC19 Meeting (2)

4. “Can we make a language distinction between loops with completely independent iterations?”

- When a functor is passed to **parallel_for**, which takes as its argument **id** instead of **item**, then no collectives (e.g. barrier) are available
  - A user can use this form of kernel functor to signal that iterations/work-items are independent
- Does this already sufficiently address the request for a form of invocation function that has independent iterations?

2. Groups Library

oneAPI Technical Advisory Board - January 28, 2020

Group Algorithms Based on C++17 `<algorithm>`

- C++17 algorithms operate over a range defined by a pair of iterators using an execution policy indicating how an algorithm can be parallelized:

```
float x = std::reduce(std::par, data.begin(), data.end(),
                      0, std::plus<>());
```

- SYCL group algorithms operate over work-item data or a range defined by a pair of `multi_ptrs` using a group describing which work-items to use:

```
float x = sycl::reduce(group, data[i], 0, std::plus<>());
float x = sycl::reduce(group, data.begin(), data.end(), 0, std::plus<>());
```

- All members of a group must call algorithm functions together.

Group Algorithms: Example Kernels

```
h.parallel_for(nd_range<1>(32, 32), [=](nd_item<1> it) {
    int item_sum = 0;
    for (int i = it.get_local_id(0); i < 32; i += it.get_local_range(0)) {
      item_sum += input[i];
    }
    group g = it.get_group();
    int group_sum = reduce(g, item_sum, 0, std::plus<>());
    if (it.get_local_id(0) == 0) {
      output = group_sum;
    }
});

h.parallel_for(nd_range<1>(32, 32), [=](nd_item<1> it) {
    group g = it.get_group();
    int group_sum = reduce(g, input, input + 32, 0, std::plus<>());
    if (it.get_local_id(0) == 0) {
      output = group_sum;
    }
});
```

- Examples use only one work-group to keep things simpler.
- First example shows reduction of work-item data.
- Second example shows reduction of data held in memory.

Algorithm Priority

- Initial focus is on matching OpenCL/OpenMP functionality:
  - OpenCL:
    - `any_of, all_of, none_of`
    - `broadcast` (no C++17 algorithm equivalent)
  - OpenMP:
    - `reduce`
    - `exclusive_scan, inclusive_scan`

- We plan to take this extension to Khronos for the next SYCL spec
  – Other algorithms (e.g. `find, sort, shift`) to follow in future extensions.

Towards a Generic Group Abstraction

- Add traits to group class:

```
template <int Dimensions = 1>
class group
{
public:
  typedef id<dimensions> id_type;
  typedef range<dimensions> range_type;
  typedef size_t linear_id_type;
  static constexpr int dimensions = Dimensions;
};
```

- Motivation:
  - Simplifies definition of functions in group algorithms library
  - Groundwork for generic programming supporting other group types (e.g. sub-groups, groups of active work-items, arbitrary user-defined groups)

Function Objects

- Provide transparent (if C++14) function objects for supported operators.
- Function objects supported natively by C++ are defined as aliases:
  - `sycl::plus (+)`
  - `sycl::multiplies (*)`
  - `sycl::bit_and (&), sycl::bit_or (|), sycl::bit_xor (^)`
  - `sycl::logical_and (&&)`
  - `sycl::logical_or (||)`
- New function objects added for common HPC cases:
  - `sycl::minimum`
  - `sycl::maximum`

Summary

- Group algorithms library design reflects our long-term direction:
  - Enables support for STL-like functions at all levels of the hierarchy
  - Step towards group concepts and programming with generic groups

- Your feedback would be much appreciated, specifically on:
  - Alignment with C++17/20 algorithms
  - Algorithm priority

3. Simplifications For Common SYCL Patterns

oneAPI Technical Advisory Board - January 28, 2020

Presentation Goals

1. Review SYCL 1.2.1 verbosity in simple example
   - Acknowledge importance of code simplicity, and DPC++ work feeding to SYCL

2. Show direction that we're moving for reduction of verbosity

3. Gather feedback on directions

Generally: Reduce verbosity of code

- Historically
  - SYCL goal has been to enable expressive and generic programming
  - Simple code patterns can be quite verbose

- Pushing now to
  - Simplify developer input for common patterns, while still enabling generic coding

- Multiple directions in work
  - Target fixes to future SYCL specifications directly where possible
    - Presentation at upcoming Khronos F2F
  - Build extensions to prove solutions, and simplify DPC++ code now

Directions

- Multiple simplifications in flight
  1. cl::sycl ⇒ sycl
  2. Optional kernel lambda names
  3. Class template argument deduction (CTAD)
     - buffer<int, 2> b(ptr, range<2>(5, 5)); ⇒ buffer b(ptr, range(5, 5));
  4. Implicit conversion of id to size_t
     - sg.get_local_id()[0] ⇒ sg.get_local_id()
  5. range and nd_range simplifications: Under discussion
     - cgh.parallel_for(range{N}, ... ⇒ cgh.parallel_for(N, ...
  6. queue.parallel_for and variants without submit()
     - Q.parallel_for(R, {prev_event}, [=](id<1> idx) { ... });
  7. Buffer construction: buffer<int> b(v.data(), v.size()); ⇒ buffer b(v);
  8. Enable auto in kernel call

Hello world of SYCL 1.2.1

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;

int main () {
    cl::sycl::buffer<int,1> B(N);

    cl::sycl::queue{}.submit([&](cl::sycl::handler &h) {
        auto a = B.get_access<cl::sycl::access::mode::write>(h);
        h.parallel_for<class mykern>(cl::sycl::range<1>(N), [=](cl::sycl::id<1> i) {
            a[i] = i[0];
        });
    });

    auto a = B.get_access<cl::sycl::access::mode::read>();
    for (int i=0; i<N; ++i) std::cout << a[i] << '\n';

    return 0;
}
```

- Wordy/redundant
- Lots to remember or copy/paste
- Examples are a combo of language + style

oneAPI

Hello world with implemented simplifications

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;

int main () {
    cl::sycl::buffer<int,1> B(N);

    cl::sycl::queue{}.submit([&](cl::sycl::handlerauto &h) {
        auto a = B.get_access<cl::sycl::access::mode::write>(h);
        h.parallel_for<class mykern>(cl::sycl::range<1>(N), [=](cl::sycl::id<1> i) {
            a[i] = i[0];
        });
    });

    auto a = B.get_access<cl::sycl::access::mode::read>();
    for (int i=0; i<N; ++i) std::cout << a[i] << '\n';

    return 0;
}
```

- Works with DPC++ beta04
- Better, but not there yet
- **Auto handler + default dim buffer** always worked, but people don't use them (and they should)

Hello world with implemented simplifications (2)

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;

int main () {
    sycl::buffer<int> B(N);

    sycl::queue{}.submit([&](auto &h) {
        auto a = B.get_access<sycl::access::mode::write>(h);
        h.parallel_for(sycl::range(N), [=](cl::sycl::id<1> i) {
            a[i] = i[0];
        });
    });

    auto a = B.get_access<sycl::access::mode::read>();
    for (int i=0; i<N; ++i) std::cout << a[i] << '\n';

    return 0;
}
```

- Works with DPC++ beta04
- Better, but not there yet
- Auto handler + default dim buffer always worked, but people don't use them (and they should)

Where we’ll be with changes already in flight

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;

int main () {
    sycl::buffer<int> B(N);

    sycl::queue{}.submit([&](auto &h) {
        auto a = B.get_access<sycl::access::mode::write>(h);
        h.parallel_for(sycl::range(N), [=](cl::sycl::id<1>auto i) {
            a[i] = i[0];
        });
    });

    auto a = B.get_access<sycl::access::mode::read>();
    for (int i=0; i<N; ++i) std::cout << a[i] << ‘\n’;

    return 0;
}
```

- Better, but not there yet
- With USM, significantly shorter still: Q.parallel_for, and no accessor

Where we'll be with changes already in flight (2)

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;

int main () {
    sycl::buffer<int> B(N);

    sycl::queue{}.submit([&](auto &h) {
        auto a = B.get_access<sycl::access::mode::write>(h);
        h.parallel_for(N, [=](auto i) {
            a[i] = i;
        });
    });

    auto a = B.get_access<sycl::access::mode::read>();
    for (int i=0; i<N; ++i) std::cout << a[i] << '\n';

    return 0;
}
```

- Better, but not there yet
- With USM, significantly shorter still: Q.parallel_for, and no accessor
- Major remaining problem = accessors

Example with USM

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;

int main () {
    sycl::queue q;
    int *B = (int *)sycl::malloc_shared(N*sizeof(int), q);

    q.parallel_for(N, [=](auto i) { B[i] = i; }).wait();

    for (int i=0; i<N; ++i) std::cout << B[i] << '\n';

    sycl::free(B, q);
    return 0;
}
```

- Should anything in this USM example be simpler?

Accessor Simplifications

1. Modify enums (simplify)
   - sycl::access::target::global_buffer ⇒ sycl::target::global
   - sycl::access::mode::read ⇒ sycl::access_mode::read

2. Make the enums unnecessary in typical code
   1. Default accessor mode is read_write (conservative)
   2. Optimized modes in accessor name, and not template parameter

+ Maintain backward compatibility

New Accessor Forms

New	SYCL 1.2.1 (Old forms)
accessor	accessor<sycl::access::mode:read_write>
read_accessor	accessor<sycl::access::mode:read>
write_accessor	accessor<sycl::access::mode:write>
discard_write_accessor	accessor<sycl::access::mode:discard_write>
discard_read_write_accessor	accessor<sycl::access::mode:discard_read_write>
atomic_accessor	accessor<sycl::access::mode:atomic>

Most example and tutorial code should use simply “accessor”

New Buffer `get_access` Forms

New	SYCL 1.2.1 (Old forms)
get_accessor	get_access<sycl::access::mode:read_write>
get_read_accessor	get_access<sycl::access::mode:read>
get_write_accessor	get_access<sycl::access::mode:write>
get_discard_write_accessor	get_access<sycl::access::mode:discard_write>
get_discard_read_write_accessor	get_access<sycl::access::mode:discard_read_write>
get_atomic_accessor	get_access<sycl::access::mode:atomic>

Most example and tutorial code should construct “accessor”, not use getters

Hello world with new accessor forms

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;

int main () {
    sycl::buffer<int> B(N);

    sycl::queue{}.submit([&](auto &h) {
        sycl::accessor a(B,h); ~~auto a = B.get_access<sycl::access::mode::write>(h)~~;
        h.parallel_for(N, [=](auto i) {
            a[i] = i;
        });
    });

    sycl::accessor a(B); ~~auto a = B.get_access<sycl::access::mode::read>()~~;
    for (int i=0; i<N; ++i) std::cout << a[i] << '\n';

    return 0;
}
```

Hello world with new accessor forms (2)

oneAPI

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;

int main () {
    sycl::buffer<int> B(N);

    sycl::queue{}.submit([&](auto &h) {
        sycl::accessor a(B,h);
        h.parallel_for(N, [=](auto i) {
            a[i] = i;
        });
    });

    sycl::accessor a(B);
    for (int i=0; i<N; ++i) std::cout << a[i] << '\n';

    return 0;
}
```

Simple factoring out for examples

oneAPI

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int N = 32;
using namespace sycl;

int main () {
    buffer<int> B(N);
    queue Q;

    Q.submit([&](auto &h) {
        accessor a(B,h);
        h.parallel_for(N, [=](auto i) {
            a[i] = i;
        });
    });

    accessor a(B);
    for (int i=0; i<N; ++i) std::cout << a[i] << '\n';

    return 0;
}
```

- This code runs a kernel on a device, and manages execution order and data movement from device to host
- What else here should be simplified?

oneAPI™

2b: Group Library Backup

Broadcast

- ```
  template <typename Group, typename T>
  T broadcast(Group g, T x);
  ```
- ```
  template <typename Group, typename T>
  T broadcast(Group g, T x, Group::linear_id_type
  local_linear_id);
  ```
- ```
  template <typename Group, typename T>
  T broadcast(Group g, T x, Group::id_type local_id);
  ```
  ```

Reduce

- `template <typename Group, typename T, class BinaryOp>`
  `T reduce(Group g, T x, BinaryOp binary_op);`

- `template <typename Group, typename V, typename T, class BinaryOp>`
  `T reduce(Group g, V x, T init, BinaryOp binary_op);`

- `template <typename Group, typename Ptr, class BinaryOp>`
  `Ptr::element_type reduce(Group g, Ptr first, Ptr last, BinaryOp binary_op);`

- `template <typename Group, typename Ptr, typename T, class BinaryOp>`
  `Ptr::element_type reduce(Group g, Ptr first, Ptr last, T init, BinaryOp binary_op);`

Any/All/None

- ```
  template <typename Group>
  bool any_of(Group g, bool predicate);
  ```

- ```
  template <typename Group, typename T, class UnaryPredicate>
  bool any_of(Group g, T x, UnaryPredicate p);
  ```

- ```
  template <typename Group, typename Ptr, class UnaryPredicate>
  bool any_of(Group g, Ptr first, Ptr last, UnaryPredicate p);
  ```
  ```

Exclusive Scan

- ```
  template <typename Group, typename T, class BinaryOp>
  T exclusive_scan(Group g, T x, BinaryOp binary_op);
  ```
- ```
  template <typename Group, typename V, typename T, class BinaryOp>
  T exclusive_scan(Group g, V x, T init, BinaryOp binary_op);
  ```
- ```
  template <typename Group, typename InPtr, typename OutPtr, class BinaryOp>
  OutPtr exclusive_scan(Group g, InPtr first, InPtr last, OutPtr d_first, BinaryOp
  binary_op);
  ```
- ```
  template <typename Group, typename InPtr, typename OutPtr, typename T, class BinaryOp>
  OutPtr exclusive_scan(Group g, InPtr first, InPtr last, OutPtr d_first, T init,
  BinaryOp binary_op);
  ```

Inclusive Scan

- `template <typename Group, typename T, class BinaryOp>`
  `T inclusive_scan(Group g, T x, BinaryOp binary_op);`

- `template <typename Group, typename V, typename T, class BinaryOp>`
  `T inclusive_scan(Group g, V x, BinaryOp binary_op, T init);`

- `template <typename Group, typename InPtr, typename OutPtr, class BinaryOp>`
  `OutPtr inclusive_scan(Group g, InPtr first, InPtr last, OutPtr d_first, BinaryOp binary_op);`

- `template <typename Group, typename InPtr, typename OutPtr, class BinaryOp, typename T>`
  `T inclusive_scan(Group g, InPtr first, InPtr last, OutPtr d_first, BinaryOp binary_op, T init);`