IWOCL 2024

The 12th International Workshop on OpenCL and SYCL

Towards a Unified Group Abstraction for SYCL

James Brodman, Intel

Ben Ashbaugh, Michael Kinsner, Steffen Larsen, Greg Lueck, John Pennycook, Roland Schulz – Intel

Gordon Brown – Codeplay

APRIL 8-11, 2024 | CHICAGO, USA | IWOCL.ORG

Our Ideal: A Group *Concept* (C++20)

```
// 1) Developers can define their own algorithms working over any kind of group
void foo(sycl::group auto& g, ...) {
  for (int i = g.get_item_id(); i < N; i += g.get_item_range()) { ... }
}

// 2) Groups can be partitioned across boundaries that are meaningful to applications
sycl::group auto partition1 = syclx::fixed_partition<4>(another_group);

// 3) Safe to use group algorithms in diverged control flow
sycl::group auto partition2 = syclx::logical_partition(another_group, condition);
if (condition) {
  sycl::group auto tangle = syclx::entangle(another_group);
  foo(tangle); // or foo(partition2)
}

// 4) Possible to discover dynamic run-time groups of work-items opportunistically
sycl::group auto active = syclx::opportunistic_group();

// 5) Groups can be partitioned along boundaries defined by the hardware/execution model
sycl::group auto root = syclx::this_work_item::get_root_group<1>();
sycl::group auto wg = syclx::scoped_partition<sycl::memory_scope::work_group>(root);
```

Step 1) Revisiting `sycl::group`

Addressing developer feedback

Recap: Core Functionality of SYCL Groups

```
class group {
 public:
  using id_type = id<1>;
  using range_type = range<1>;
  using linear_id_type = uint32_t;
  static constexpr int dimensions = 1;

  static constexpr memory_scope fence_scope = memory_scope::work_group;

  id_type get_local_id() const;
  range_type get_local_range() const;
  linear_id_type get_local_linear_id() const;
  linear_id_type get_local_linear_range() const;

  id_type get_group_id() const;
  range_type get_group_range() const;
  linear_id_type get_group_linear_id() const;
  linear_id_type get_group_linear_range() const;

  bool leader() const;
};
```

Required to define group algorithms interfaces

← Required for **`group_barrier()`**

Required to identify:
1) Work-item index within the group
2) Group index within implicit "parent" group

Renaming Existing Functionality

- `sycl::group` ⇒ `sycl::work_group`
  - SYCL, OpenCL and SPIR-V all talk about “work-groups”
  - “group” in SYCL means either **`group`** or “group of work-items”

- `get_local_id()` ⇒ `get_item_id()`
  - What is the ID local to? Is it still a group ID?
  - Contrast between **`get_item_id()`** and **`get_group_id()`** is more obvious

- We believe this will make SYCL easier to teach in the long run

Step 2) Extending the Group Classes

Enabling new use-cases

Use-Case: Device-wide Barriers and Algorithms

SYCL 2020

```
for (int offset = N / 2; offset > 0; offset /= 2) {
  q.parallel_for(N, sycl::nd_item<1> it) {
    value[it.get_global_id()] += value[it.get_global_id() + offset];
  } // all work-items must finish executing the kernel before the next iteration begins
}
```

SYCL 2020 + "Root Groups"

```
q.parallel_for(N, sycl::nd_item<1> it) {
  auto root = it.get_root_group();
  for (int offset = N / 2; offset > 0; offset /= 2) {
    value[root.get_item_id()] += value[root.get_item_id() + offset];
    sycl::group_barrier(root); // block until all work-items on the device reach the barrier
  }
}
```

Use-Case: Arbitrary Hierarchical Parallelism

SYCL 2020

```
q.parallel_for(N, sycl::nd_item<1> it) [[sycl::reqd_sub_group_size<32>]] {
  auto partition_group_id = it.get_sub_group().get_item_id() / 4;     // user-defined partition
  auto partition_item_id  = it.get_sub_group().get_item_id() % 4;
  float sum = value[it.get_global_id()];
  for (int offset = 4 / 2; offset > 0; offset /= 2) {
    sum += sycl::permute_over_group(it.get_sub_group(), sum, offset); // manual algorithm
  }
}
```

SYCL 2020 + "Fixed Size Partitions"

```
q.parallel_for(N, sycl::nd_item<1> it) [[sycl::reqd_sub_group_size<32>]] {
  auto partition = syclx::fixed_partition<4>(it.get_sub_group());      // user-defined partition
  float sum = sycl::reduce_over_group(partition, sum, sycl::plus<>()); // existing algorithm
}
```

Use-Case: Divergent Control Flow (Manual)

SYCL 2020

```
q.parallel_for(N, sycl::nd_item<1> it) {
  float x;
  if (condition) { x = value[it.get_global_id()]; }
  else { x = 0; } // must be set to the identity

  float sum = sycl::reduce_over_group(it.get_sub_group(), x, sycl::plus<>()); // must be converged
}
```

SYCL 2020 + "Logical Partitions"

```
q.parallel_for(N, sycl::nd_item<1> it) {
  auto partition = syclx::logical_partition(it.get_sub_group(), condition); // prepare to diverge
  if (condition) {
    float x = value[it.get_global_id()];
    float sum = sycl::reduce_over_group(partition, x, sycl::plus<>());      // divergence is safe
  }
}
```

Use-Case: Divergent Control Flow (Implicit)

SYCL 2020

```
q.parallel_for(N, sycl::nd_item<1> it) {
  float x;
  if (condition) { x = value[it.get_global_id()]; }
  else { x = 0; } // must be set to the identity

  float sum = sycl::reduce_over_group(it.get_sub_group(), x, sycl::plus<>()); // must be converged
}
```

SYCL 2020 + "Tangles"

```
q.parallel_for(N, sycl::nd_item<1> it) {
  if (condition) {
    auto partition = syclx::entangle(it.get_sub_group());                  // get all diverged items
    float x = value[it.get_global_id()];
    float sum = sycl::reduce_over_group(partition, x, sycl::plus<>());     // divergence is safe
  }
}
```

Use-Case: Opportunistic (Discoverable) Parallelism

CUDA

```
void increment(int* ptr, int x) {
  auto active = cg::coalesced_threads();          // get the active threads
  int sum = cg::reduce(active, x, cg::plus<>()); // sum across active threads
  if (active.thread_rank() == 0) {                // elect a leader
    atomicAdd(ptr, sum);                          // only the leader does the atomic
  }
}
```

SYCL 2020 + "Opportunistic Groups"

```
void increment(int* ptr, int x) {
  auto active = syclx::opportunistic_group();                    // get the active work-items
  int sum = sycl::reduce_over_group(active, x, sycl::plus<>()); // sum across active work-items
  if (active.leader()) {                                         // elect a leader
    sycl::atomic_ref<...>(ptr) += sum;                           // only the leader does the atomic
  }
}
```

Use-Case: Partitioning Along Hardware Boundaries

SYCL 2020

```
sycl::group wg = it.get_work_group();    // group of all work-items in the same work-group
sycl::sub_group sg = it.get_sub_group(); // group of all work-items in the same sub-group

// compute global sub-group index manually
size_t global_sg_id = wg.get_group_id() * sg.get_group_range() + sg.get_group_id();
```

SYCL 2020 + "Scoped Partitions"

```
auto root = syclx::this_work_item::get_root_group<1>(); // group representing all work-items

// partition the root-group into sub-groups, treating root-group as its parent
auto global_sg = syclx::scoped_partition<sycl::memory_scope::sub_group>(root);

size_t global_sg_id = global_sg.get_group_id(); // get global sub-group index (in the root-group)!
```

Unresolved issue: how does a multi-dimensional group decompose into sub-groups?

Step 3) Defining the Group Concept(s)

Bringing it all together

Concept Idea 1: A Group of Work-Items

```
template <typename T>
concept indexable_item_group = requires(T g)
{
  typename T::id_type;
  typename T::range_type;
  typename T::linear_id_type;
  typename T::linear_range_type;

  requires std::signed_integral<decltype(T::dimensions)>;

  { g.get_item_id() } -> std::same_as<typename T::id_type>;
  { g.get_item_range() } -> std::same_as<typename T::range_type>;
  { g.get_item_linear_id() } -> std::same_as<typename T::linear_id_type>;
  { g.get_item_linear_range() } -> std::same_as<typename T::linear_range_type>;

  { g.get_group_id() } -> std::same_as<typename T::id_type>;
  { g.get_group_range() } -> std::same_as<typename T::range_type>;
  { g.get_group_linear_id() } -> std::same_as<typename T::linear_id_type>;
  { g.get_group_linear_range() } -> std::same_as<typename T::linear_range_type>;
};
```

All existing groups satisfy **`indexable_item_group`**.
Applicable to arbitrary future user-constructed groups.

Concept Idea 2: A Group of Work-Items with a Barrier?

```
template <typename T>
concept coordination_item_group = requires(T g)
{
  requires indexable_item_group<T>;
  requires std::same_as<decltype(T::fence_scope), const sycl::memory_scope>;

  { g.leader() } -> std::same_as<bool>;

  { sycl::group_barrier(g) } -> std::same_as<void>;

  { g.can_synchronize() } -> std::same_as<bool>;
};
```

← Must also implement **`get_item_id()`** & co.

← An overload of **`group_barrier`** must be available.

← A specific group instance might not support a barrier. (e.g., based on launch parameters or partitioning.)

All existing groups satisfy **`coordination_item_group`**.
Only applicable to user-constructed groups which provide an overload for **`group_barrier`**.

Future Work

Unresolved issues

Feedback Requested

- Which group types are the most useful?
- What should the concept(s) be called? **`sycl::group`** is taken. ☹
- Do you expect all groups to support algorithms?
  Would you be willing to register runtime information?

```
sycl::group auto g = user_defined_partitioning_algorithm(another_group);
if (not g.has_scratch()) {
    g.register_scratch((void*) user_allocated_memory);
}
if (not g.has_barrier()) {
    g.register_barrier((std::barrier*) user_allocated_barrier);
}
float sum = sycl::reduce_over_group(g, x, std::plus<>()); // UB if scratch or barrier missing?
```

Summary

- 3-step plan to reach C++20 Concepts for groups of work-items
  1. Improve teachability of existing group functionality
  2. Explore root groups; fixed-size and logical partitions; tangles; opportunistic groups; and scoped partitions
  3. Define the necessary Concepts

- We need your feedback
  - Critical to ensure we don't create a Concept too soon
  - Early implementations available at https://github.com/intel/llvm/

Disclaimers


intel