Towards Alignment of Parallelism in SYCL and ISO C++

John Pennycook, Intel Corporation

Ben Ashbaugh, James Brodman, Michael Kinsner, Steffen Larsen, Greg Lueck, Roland Schulz, Michael Voss
Parallel Evolution of SYCL, ISO C++ and OpenCL

Maintaining alignment between these specifications requires constant, ongoing effort!
Motivating Use-Case: Global Synchronization

Should this code work? Does it?

```cpp
template <size_t Dimensions>
void arrive_and_wait(size_t expected, sycl::group<Dimensions> wg, ...) {
    // Wait for all work-items in the group before signaling arrival
    sycl::group_barrier(wg);

    // Elect one work-item to synchronize with other groups
    if (wg.leader()) {
        // Signal that this group has arrived at the barrier
        atomic_counter++;

        // Spin while waiting for all groups to arrive
        while (atomic_counter.load() != expected) {}
    }

    // Wait for the leader to finish synchronizing with other groups
    sycl::group_barrier(wg);
}
```

**Assumption 1:**
Leader of the work-group makes progress while other work-items wait at second barrier.

**Assumption 2:**
Every work-group leader makes progress.

Empirical evidence for support under “occupancy-bound execution” by Sorensen et al. on multiple GPUs
Motivating Use-Case: Sub-group Specialization

Should this code work? Does it?

```cpp
void produce(sycl::local_ptr<example::concurrent_queue> tasks)
{
    if (sg.leader())
    {
        tasks->push(...);
    }
}

void consume(sycl::local_ptr<example::concurrent_queue> tasks)
{
    if (sg.leader())
    {
        work = tasks->pop();
    }
    foo(work);
}
```

**Assumptions:**
Leader of every sub-group makes progress, while other work-items wait at a barrier (not shown).

**NB:** Assumptions only relevant for sub-groups in the same work-group.

Empirical evidence for support of “warp specialization” by Bauer et al. on NVIDIA GPUs.
## Forward Progress Guarantees in ISO C++

<table>
<thead>
<tr>
<th>Guarantee</th>
<th>Concurrent</th>
<th>Parallel</th>
<th>Weakly Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eventually executes its first step</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Makes progress after executing its first step</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mental Model</th>
<th>Provided By</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS threads</td>
<td>std::thread</td>
<td>par</td>
</tr>
<tr>
<td>Tasks</td>
<td>par</td>
<td>par_unseq</td>
</tr>
</tbody>
</table>
## Forward Progress Guarantees in SYCL?

<table>
<thead>
<tr>
<th>Guarantee</th>
<th>Work-item in parallel_for</th>
<th>Work-item in ND-Range parallel_for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eventually executes its first step</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Makes progress after executing its first step</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Makes progress when other work-items hit a barrier</td>
<td>N/A</td>
<td>?</td>
</tr>
</tbody>
</table>

**SYCL 2020, Revision 6, Section 3.8.3.4:**

“A SYCL implementation must execute work-items concurrently and must ensure that the work-items in a group **obey the semantics of group barriers**, but are **not required to provide any additional forward progress guarantees**”

† Not “concurrent forward progress guarantees”!
“Blocking with Forward Progress Guarantee Delegation” in ISO C++

// Assume calling thread has concurrent forward progress guarantees
std::for_each(std::par_unseq, c.begin(), c.end(), [&](auto x) {
  ...
  // Each invocation has weakly parallel forward progress guarantees
}); // Calling thread blocks with forward progress delegation

at least one thread is temporarily strengthened to have concurrent forward progress guarantees
## Forward Progress Guarantees in SYCL (Revisited)

<table>
<thead>
<tr>
<th>Guarantee</th>
<th>Work-item in parallel_for</th>
<th>Work-item in ND-Range parallel_for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eventually executes its first step</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Makes progress after executing its first step</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Makes progress when other work-items hit a barrier</td>
<td>N/A</td>
<td>✓</td>
</tr>
</tbody>
</table>

Proposed Fixes to Section 3.8.3.4:

1. "Each work-item ... is a separate thread of execution, providing at least weakly parallel forward progress guarantees."
2. "When a work-item arrives at a group barrier acting on group G, implementations must eventually select and potentially strengthen another work-item in group G that has not yet arrived at the barrier."
Hypothetical: SYCL Implemented with ISO C++

```cpp
template <typename Kernel>
void handler::parallel_for(sycl::nd_range<1> ndr, Kernel f) {

    std::vector<size_t> groups = { 1, 2, ..., ndr.get_group_range()[0] };
    std::vector<size_t> items = { 1, 2, ..., ndr.get_local_range()[0] };

    // Create a thread of execution providing parallel forward progress guarantees per work-group
    std::for_each(std::execution::par, std::begin(groups), std::end(groups), [&](size_t group_id) {
        // Create a thread of execution providing weakly parallel forward progress guarantees per work-item
        std::for_each(std::execution::par_unseq, std::begin(items), std::end(items), [&](size_t item_id) {
            // Invoke the user supplied kernel function object
            sycl::nd_item<1> item = sycl::detail::make_nd_item<1>(group_id, item_id);
            f(item);
        });
    });
}
```

NB: Not all threads of execution are created at the same time.

Pseudocode of a hypothetical implementation for illustrative purposes only.
# New Mental Model: A Hierarchy of Threads

<table>
<thead>
<tr>
<th>Host</th>
<th>Root-Group</th>
<th>Work-Group</th>
<th>Sub-Group</th>
<th>Work-Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The **host** has at least one thread and creates one thread per **root-group**.
- The **root-group** creates one thread per **work-group**.
- Each **work-group** creates one thread per **sub-group**.
- Each **sub-group** creates one thread per **work-item**.

Each thread **blocks with forward progress guarantee delegation** on its children.
New Mental Model: Mapping to OpenCL 1.x

<table>
<thead>
<tr>
<th>Host</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root-Group</td>
<td>Weakly Parallel</td>
</tr>
<tr>
<td>Work-Group</td>
<td>Weakly Parallel</td>
</tr>
<tr>
<td>Sub-Group</td>
<td>Weakly Parallel</td>
</tr>
<tr>
<td>Work-Item</td>
<td>Weakly Parallel</td>
</tr>
</tbody>
</table>

- At least one \{root-group, work-group, sub-group, work-item\} makes progress.
- Individual \{root-group, work-group, sub-group, work-item\}s have no guarantees.

Each thread **blocks with forward progress guarantee delegation** on its children.
New Mental Model: Mapping to OpenCL 2.x†

<table>
<thead>
<tr>
<th></th>
<th>Host</th>
<th>Concurrent</th>
<th>Root-Group</th>
<th>Weakly Parallel</th>
<th>Work-Group</th>
<th>Weakly Parallel</th>
<th>Sub-Group</th>
<th>Concurrent</th>
<th>Weakly Parallel</th>
<th>Work-Item</th>
<th>Weakly Parallel</th>
</tr>
</thead>
</table>

- At least one \{root-group, work-group\} makes progress.
- Every sub-group in an executing work-group makes progress.
- At least one work-item per sub-group makes progress.
- Individual \{root-group, work-group, work-item\}s have no guarantees.

† Assuming support for CL_DEVICE_SUB_GROUP_INDEPENDENT_FORWARD_PROGRESS.
Designing an Extension: High-Level Goals

- Don’t change the default behavior of existing SYCL programs
- Give implementations flexibility, not additional implementation burden
- Let developers decide how to balance P3 trade-offs
- Allow developers to reason about where code will run correctly
- Enable opt-in to strong guarantees to enable new algorithms
- Re-use existing mechanisms for querying device capabilities

Portability

Performance

Productivity
Using the Extension: Declaring Requirements

```cpp
struct MyKernel {
    // Kernel function calls arrive_and_wait
    // (Other functionality omitted)
    void operator()(sycl::nd_item<1> it) {
        ...
        arrive_and_wait(num_work_groups, it.get_group());
        ...
    }

    // Kernel Properties: Declare requirements
    auto get(sycl::properties_tag) {
        return sycl::properties {
            sycl::work_group_progress
            <sycl::forward_progress_guarantee::concurrent,
             sycl::execution_scope::root_group>
        };
    }

    size_t num_work_groups;
};
```

Requirements are embedded in the kernel.

“At least one work-item in each work-group created by the same root-group must provide concurrent forward progress guarantees.”
Using the Extension: Submitting the Kernel

```cpp
try {
    // Kernel Launch: Attempt to use a fixed ND-range
    auto range = sycl::nd_range<1>{num_wg * wg_size, wg_size};
    q.parallel_for(range, MyKernel(num_wg));
} catch (...) {
    // Fall back to an alternative kernel implementation
    // or exit with an error
}
```

Requirements are extracted automatically from the kernel definition.

Implementation throws an exception if the requirements cannot be satisfied.
Using the Extension: Querying Support

// Device Queries: Check support for requirements
using query = sycl::info::device::forward_progress_guarantee
  <sycl::forward_progress_guarantee::concurrent,
   sycl::execution_scope::root_group>;
sycl::device dev = q.get_device();
auto capability = dev.get_info<query>();
if (capability >= sycl::forward_progress_guarantee::concurrent)
{
  // Kernel Launch Queries: Determine valid ND-range size
  using size_query = sycl::info::kernel::max_work_group_size;
  using num_query = sycl::info::kernel::max_num_work_groups;
  auto bundle = sycl::get_kernel_bundle(q.get_context());
  auto kernel = bundle.get_kernel<class MyKernel>();
  auto wg_size = kernel.get_info<size_query>(q);
  auto num_wg = kernel.get_info<num_query>(q, wg_size);

  // Kernel Launch: Use results from queries
  auto range = sycl::nd_range<1>{num_wg * wg_size, wg_size};
  q.parallel_for(range, MyKernel(num_wg));
}
else { ... } //Fallback path as before (see previous slides)
Summary

▪ We’ve bridged the gap between SYCL and C++17 parallelism
  • Fixed underdefined aspects of SYCL by reusing proven terminology/concepts
  • Defined a way to reason about hierarchical forward progress guarantees
  • Proposed new features to state assumptions/requirements and query support

▪ Ongoing effort to maintain alignment and influence other standards
  • Explore interaction between ND-range kernels and std::execution
  • Apply our learnings to OpenCL, SPIR-V, Vulkan
  • Feedback welcome at https://github.com/intel/llvm/pull/7598
Disclaimers & Notices

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Khronos® is a registered trademark and SYCL™ and SPIR™ are trademarks of The Khronos Group Inc.

Code included in this presentation is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD