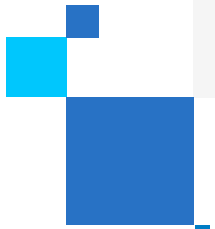
Towards Alignment of Parallelism in SYCL & C++: Identifying and Closing the Gaps

John Pennycook

Acknowledgements:
Ben Ashbaugh, James Brodman, Mike Kinsner, Greg Lueck, Steffen Larsen, Roland Schulz, Mike Voss



intel®

Disclaimers & Notices

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

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

Definitions and Terminology

Forward Progress Guarantees in ISO C++

- **Concurrent** forward progress guarantees
  - Eventually executes its first step
  - Makes progress if it hasn't terminated
- **Parallel** forward progress guarantees
  - Not required to execute its first step*
  - Makes progress if it has executed its first step
- **Weakly parallel** forward progress guarantees
  - No guarantees*

Real definitions are available at the end of the presentation.

Blocking with Forward Progress Guarantee Delegation

```
// 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
```

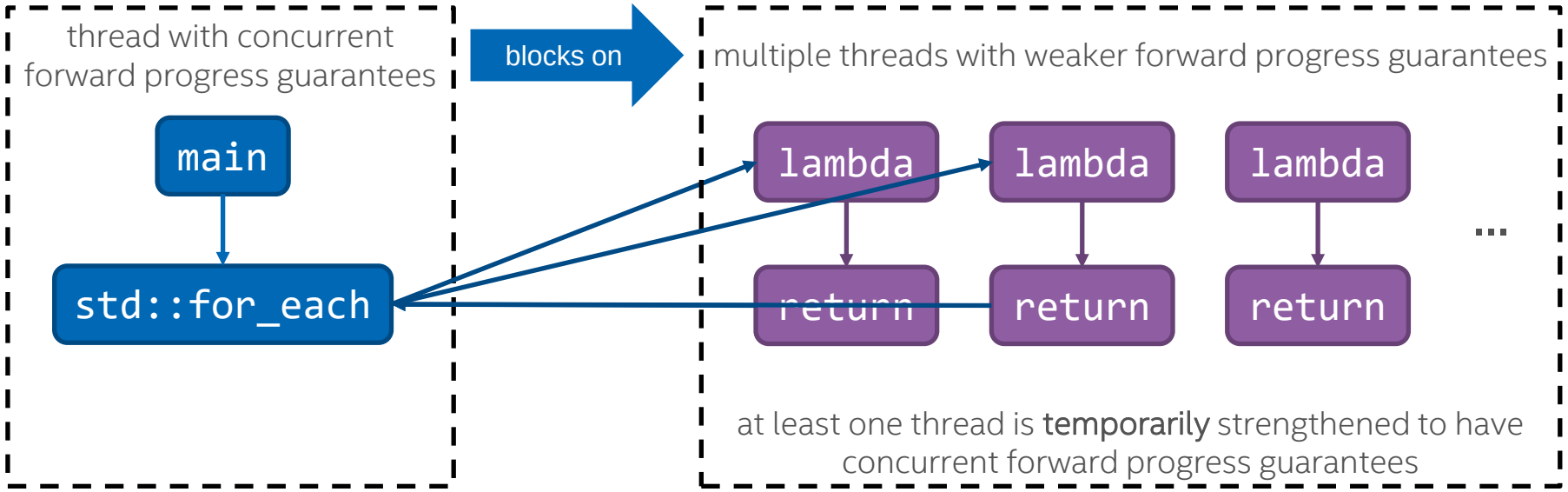


Diagram inspired by: Olivier Giroux, Forward Progress in C++. Real definitions are available at the end of the presentation.

Forward Progress Guarantees in SYCL 2020

- Section 3.8.3.4: Forward progress
  - "…forward progress with respect to other work-items is implementation-defined."
- Section 4.15.3: Atomic references
  - "…due to the lack of forward progress guarantees between work-items in SYCL"
- ≈20 uses of "concurrent", possibly implying *concurrent forward progress.*
- No guarantee that "blocking the host" leads to progress on the device (or progress of **`host_task`**s).

Proposed SYCL Bug Fixes and Clarifications

https://github.com/KhronosGroup/SYCL-Docs/pull/300

- Establish that work-items are *threads of execution*.
- "no guarantees" ⇒ *weakly parallel progress guarantees*.
  - Guarantee SYCL programs eventually make progress, via blocking with delegation.
- Replace misleading uses of "concurrent":
  - e.g. "must execute work-items concurrently" ⇒
    "must ensure work-items in a group obey the semantics of group barriers"
- Introduce intentional uses of "concurrent":
  - e.g. "there is no guarantee that host and device will execute concurrently"

Expressing Requirements

Expressing Requirements in ISO C++

- C++ threads and `std::async(std::launch::async, ...)`
  - Strongly recommended to have *concurrent forward progress guarantees*

- C++17 execution policies
  - `std::execution::par` ⇒ *parallel forward progress guarantees*
  - `std::execution::unseq` ⇒ *weakly parallel forward progress guarantees* + unsequenced

- P2300 schedulers
  - `get_forward_progress_guarantee()` advertises scheduler capabilities

Expressing Requirements in SYCL

- `sycl::parallel_for(sycl::range, ...)`
  - All work-items have *weakly parallel forward progress guarantees*

- `sycl::parallel_for(sycl::nd_range, ...)`
  - All work-items have *weakly parallel forward progress guarantees* **and**
  - Support for group barriers

- No way to reason about or request stronger progress guarantees

Use-Case: Global Synchronization via Atomics

```
template <size_t Dimensions>
void arrive_and_wait(size_t expected, sycl::group<Dimensions> wg, ...)
{
  sycl::group_barrier(wg);

  // Elect one work-item to synchronize with other groups
  if (wg.leader()) {
    atomic_counter++;

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

  sycl::group_barrier(wg);
}
```

Assumption:
Leader of every work-group makes progress while other work-items wait at barrier.

Use-Case: Sub-group Specialization

```
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);
  ...
}
```

Assumption:
Leader of each sub-group makes progress while other work-items wait at a barrier.

NB: Code doesn't care about sub-groups in other work-groups!

Towards an Extension

A new mental model: a hierarchy of threads of execution

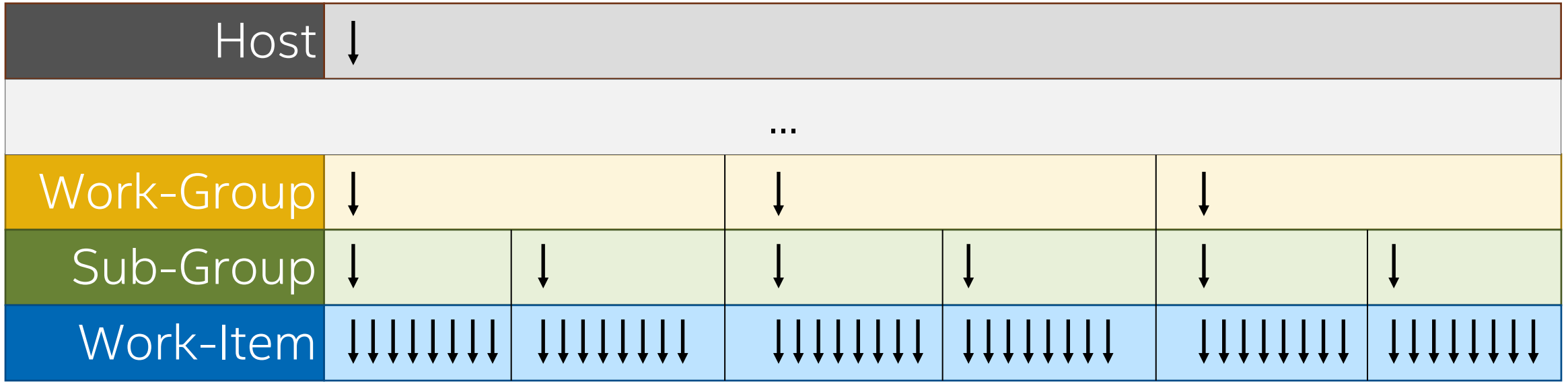
Hypothetical: SYCL Implemented with ISO C++

```
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 separate 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 separate thread of execution providing parallel forward progress guarantees per work-item
    std::for_each(std::execution::par, 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!

First Step: A Hierarchy of Threads of Execution



- The **host** C++ program has at least one *thread of execution*.
- Potentially many intermediate layers (e.g. device, sub-device, kernel)
- Each work-group creates one *thread of execution* per sub-group.
- Each sub-group creates one *thread of execution* per work-item.

Each thread of execution *blocks with forward progress delegation* on its children.

Second Step: A Hierarchy of Guarantees

Host	Host Progress Guarantee
	…
Work-Group	Work-Group Progress Guarantee
Sub-Group	Sub-Group Progress Guarantee
Work-Item	Work-Item Progress Guarantee

- New model enables us to talk about progress guarantees at **each level**.
- Flexibility ⇒ wide platform and backend support.

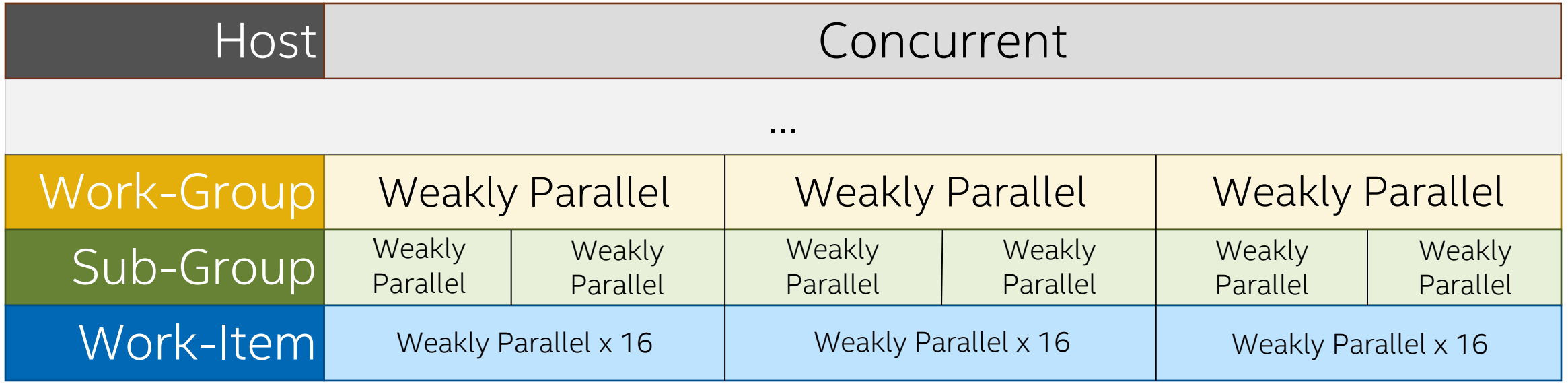
Mapping Example: OpenCL 1.x

Host	Concurrent
	…
Work-Group	Weakly Parallel
Sub-Group	Weakly Parallel
Work-Item	Weakly Parallel

- At least one {work-group, sub-group, work-item} must make progress.
- Individual {work-group, sub-group, work-item}s have no guarantees.

Each thread of execution *blocks with forward progress delegation* on its children.

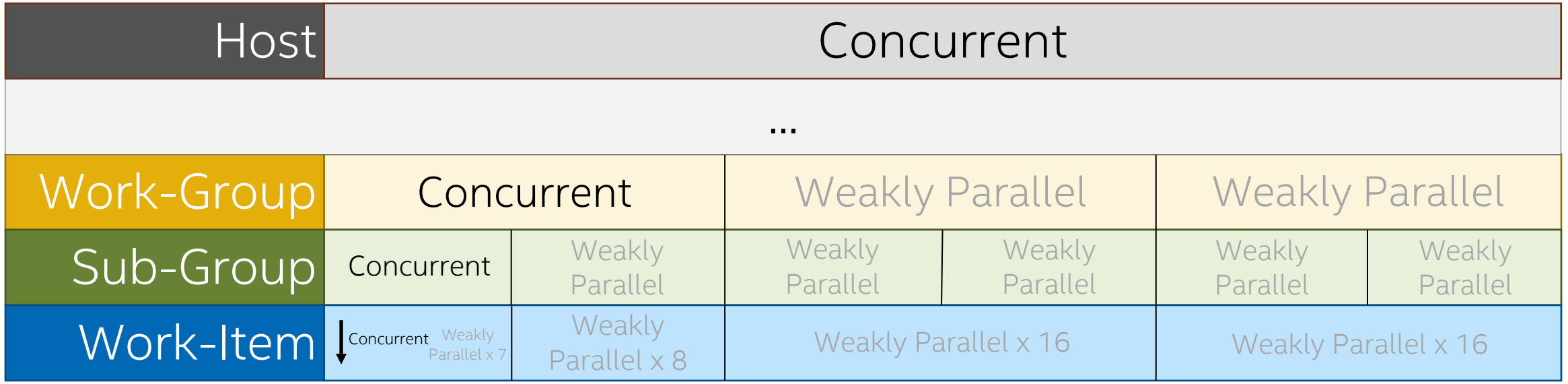
Mapping Example: OpenCL 1.x



- At least one {work-group, sub-group, work-item} must make progress.
- Individual {work-group, sub-group, work-item}s have no guarantees.

Each thread of execution *blocks with forward progress delegation* on its children.

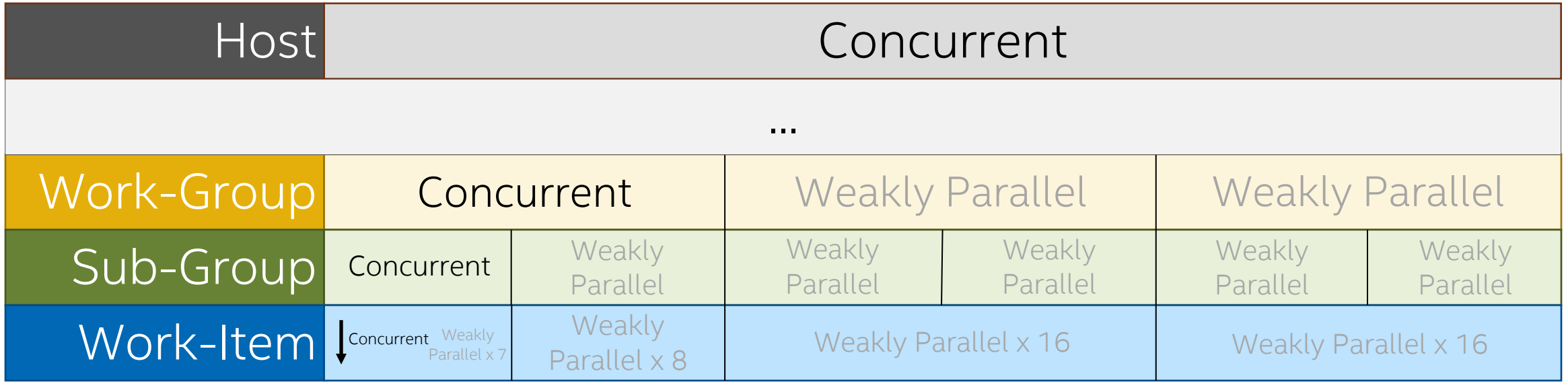
Mapping Example: OpenCL 1.x



- At least one {work-group, sub-group, work-item} must make progress.
- Individual {work-group, sub-group, work-item}s have no guarantees.

Each thread of execution *blocks with forward progress delegation* on its children.

Mapping Example: OpenCL 1.x



- At least one {work-group, sub-group, work-item} must make progress.
- Individual {work-group, sub-group, work-item}s have no guarantees.

Remember: the strengthening is not permanent.
This allows implementations to stop executing a work-item and switch to another.

Mapping Example: OpenCL 2.x
(with Sub-Group Independent Forward Progress)

Host	Concurrent
	…
Work-Group	Weakly Parallel
Sub-Group	Concurrent (?)
Work-Item	Weakly Parallel

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

Mapping Considerations for Other Backends

- Precise mapping is specific to a device, not the backend:
  - e.g. **OpenCL**: CPUs vs GPUs
  - e.g. **HIP**: AMD GPUs vs NVIDIA GPUs
  - e.g. **CUDA**: pre-Volta vs Volta
- Important considerations:
  - Eager vs lazy submission
  - Support for “cooperative” kernels
  - Mapping of hardware “threads”

Extension Sketch

- Several questions developers may ask:
  1. What are the guarantees for threads created **within** some nested scope?
  2. What are the guarantees for **all** {work-items, sub-groups, work-groups}?
  3. What are the forward progress requirements of **this kernel**?

- Answering these questions requires *queries* and/or *properties*.

- Following slides show high-level ideas; syntax needs work!

Proposal for 1) Querying Scoped Guarantees

Return the strongest guarantee that this device can provide at *Scope:*

```
auto progress = device.get_info<info::scoped_progress<Scope>>();
```

Work-Group	Parallel
Sub-Group	Concurrent
Work-Item	Weakly Parallel

```
Scope = execution_scope::work_group ⇒ parallel
Scope = execution_scope::sub_group  ⇒ concurrent
Scope = execution_scope::work_item  ⇒ weakly_parallel
```

Proposal for 2) Querying Global Guarantees

Return the strongest guarantee that this device can provide at *Scope*, reflecting the guarantees that are provided at all enclosing scopes:

```
auto progress = device.get_info<info::progress<Scope>>();
```

Work-Group	Parallel
Sub-Group	Concurrent
Work-Item	Weakly Parallel

Scope = `execution_scope::work_group` ⇒ `parallel`
Scope = `execution_scope::sub_group` ⇒ `parallel` (because of work-groups!)
Scope = `execution_scope::work_item` ⇒ `weakly_parallel`

Proposal for 3) Kernel Requirements

- Compile-time properties can describe global or scoped progress guarantees required by a kernel for correctness, e.g.

```
template <sycl::execution_scope Scope, sycl::forward_progress_guarantee Guarantee>
inline constexpr progress_key::value_t<Scope, Guarantee> progress;

template <sycl::execution_scope Scope, sycl::forward_progress_guarantee Guarantee>
inline constexpr scoped_progress_key::value_t<Scope, Guarantee> scoped_progress;
```

- Expected implementation behavior:
  - Launch a kernel differently to satisfy requirements; or
  - Fail to launch kernel with unsatisfiable requirements

Summary

- Ongoing work to align SYCL 2020 with C++17 terminology
- Plan to expose details to developers via queries/properties
  - Ongoing work to finalize sketches and simplify final proposal
- For more information
  - [intro.progress] – C++ Draft
  - Forward Progress in C++ - Olivier Giroux @ CppNorth 2022


intel

Definitions:
Forward Progress Guarantees

- "For a thread of execution providing *concurrent forward progress guarantees*, the implementation ensures that the thread will eventually make progress for as long as it has not terminated."
- "For a thread of execution providing *parallel forward progress guarantees*, the implementation is not required to ensure that the thread will eventually make progress if it has not yet executed any execution step; once this thread has executed a step, it provides concurrent forward progress guarantees."
- "For a thread of execution providing *weakly parallel forward progress guarantees*, the implementation does not ensure that the thread will eventually make progress."

Definitions from: Draft C++ Standard

Definitions:
Blocking with Forward Progress Guarantee Delegation

- "When a thread of execution *P* is specified to *block with forward progress guarantee delegation* on the completion of a set *S* of threads of execution, then throughout the whole time of *P* being blocked on *S*, the implementation shall ensure that the forward progress guarantees provided by at least one thread of execution in *S* is at least as strong as *P*'s forward progress guarantees."

- Note: "It is unspecified which thread or threads of execution in *S* are chosen and for which number of execution steps. The strengthening is not permanent and not necessarily in place for the rest of the lifetime of the affected thread of execution."

Definitions from: Draft C++ Standard