Towards Alignment of Parallelism in SYCL and ISO C++

S. John Pennycook
john.pennycook@intel.com
Intel Corporation
USA

Ben Ashbaugh
ben.ashbaugh@intel.com
Intel Corporation
USA

James Brodman
james.brodman@intel.com
Intel Corporation
USA

Michael Kinsner
michael.kinsner@intel.com
Intel Corporation
Canada

Steffen Larsen
steffen.larsen@intel.com
Intel Corporation (UK) Limited
United Kingdom

Greg Lueck
gregory.m.lueck@intel.com
Intel Corporation
USA

Roland Schulz
roland.schulz@intel.com
Intel Corporation
USA

Michael Voss
michaelj.voss@intel.com
Intel Corporation
USA

ABSTRACT

SYCL began as a C++ abstraction for OpenCL concepts, whereas parallelism in ISO C++ evolved from the algorithms in the standard library. This history has resulted in the two specifications using different terminology to describe parallelism, which is confusing to developers and hinders the SYCL community's efforts to influence the direction of C++ through experiments and proof points. Critically, SYCL does not provide mechanisms for developers to reason about specific device behaviors that may impact the execution of parallel programs, such as the forward progress guarantees at various levels of the execution model hierarchy. The N-dimensional range (ND-range) execution model currently defined by SYCL extends the C++ model, but does not relate it to concepts or formalisms of C++ parallelism.

This paper presents: (1) a detailed analysis of parallelism terminology in SYCL and ISO C++; (2) proposed modifications to the SYCL standard, to align with C++17; and (3) a generalized abstract ND-range execution model introducing the notion of hierarchical forward progress guarantees. To demonstrate the potential impact of these changes, we outline a new extension to SYCL enabling developers to understand and potentially control device behavior across the hierarchy. Although discussed in the context of SYCL, the changes outlined in this paper have broader implications for all languages building upon an ND-range model (e.g. OpenCL). Our abstract hierarchical execution model applies generally to modern data parallel languages, many of which don't yet comprehend the hierarchical nature of the hardware architectures that they target.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**.

IWOCL '23, April 18–20, 2023, Cambridge, United Kingdom

ACM ISBN 979-8-4007-0745-2/23/04.
https://doi.org/10.1145/3585341.3585371

KEYWORDS

forward progress guarantees, hierarchical parallelism, execution models, SYCL, OpenCL, C++

ACM Reference Format:
S. John Pennycook, Ben Ashbaugh, James Brodman, Michael Kinsner, Steffen Larsen, Greg Lueck, Roland Schulz, and Michael Voss. 2023. Towards Alignment of Parallelism in SYCL and ISO C++. In *International Workshop on OpenCL (IWOCL '23), April 18–20, 2023, Cambridge, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3585341.3585371

1 INTRODUCTION

One of SYCL's stated goals [19] is to produce proof-points that can influence the evolution of heterogeneous computing in ISO C++. Realizing this vision requires close alignment between the two standards, and for SYCL itself to continually evolve and align with new C++ features and best practices.

At the time of writing, much of the terminology used by the latest SYCL specification (SYCL 2020 Revision 6) [18] to describe parallelism and concurrency has been drawn from multiple revisions of the OpenCL specification [17]. This terminology predates the introduction of a parallelism library and associated concepts in C++17 [20].

Any differences in the terminology used by the SYCL and C++ descriptions of parallelism have the potential to be problematic, creating unintended sources of ambiguity for readers familiar with both specifications. Such differences could also act as a barrier to SYCL's ability to influence the future direction of C++, where appropriate.

In this paper, we explore some of the differences between the SYCL and C++ descriptions of several important concepts and demonstrate that bringing the language specifications closer together enables a more tightly defined execution model for SYCL. Specifically, we examine:

- The relationship between C++ concepts like *threads of execution* and SYCL concepts like *work-items* and *work-groups*.
- How to accurately describe the forward progress guarantees of SYCL programs using the concepts of *weakly parallel*, *parallel* and *concurrent* forward progress from C++.

- Why the definition of group barriers in SYCL must be modified to remain compatible with work-items providing only weakly parallel forward progress guarantees.
- Why introducing the concept of *blocking with forward progress guarantee delegation* to SYCL is necessary to understand the interaction between host code and device kernels.

We conclude with an early look at a proposed extension to SYCL, designed to expose low-level details of the underlying backend and its execution model to expert developers. The extension consists of:

- A new mental model for N-dimensional range (ND-range) kernels, in which work-items behave as if they are executed within a hierarchical arrangement of threads of execution.
- A set of queries exposing a device's ability to provide specific forward progress guarantees, and any limitations on a kernel's launch configuration that must be respected.
- A set of compile-time properties enabling developers to embed any required forward progress guarantees directly into a kernel's definition, thereby preventing it from being launched on incompatible devices.

Although this work has been performed in the context of SYCL, we believe that our analysis and recommendations also apply to OpenCL and other languages building on the ND-range concept (e.g. SPIR-V [22]).

2 RELATED WORK

Our exploration of SYCL's execution model builds on our previous exploration of its memory model [2]. Our proposed changes to the memory model, group barriers and atomic references were accepted into SYCL 2020. However, our changes also increased the importance of further alignment between SYCL and C++17; the concept of *making progress* is shared between the execution model and the memory model (specifically with regards to atomics), and so it is paramount that the two models are consistently defined.

Sorensen et al. have published several papers [24–26] investigating ways to describe the execution model of general-purpose GPUs, with a focus on the forward progress guarantees of different work-groups and the safety of implementing device-wide barriers. Their occupancy-bounded execution (OBE) model defines the progress guarantees of work-groups with respect to device occupancy limits. In this paper, we propose a way to reason about the forward progress guarantees for all groups and work-items in an ND-range.

There are proposals (P2300 [12], P2500 [1]) to extend ISO C++ with new features enabling asynchronous offloading of parallel algorithms. Neither proposal (explicitly) considers hierarchical parallelism. This paper is a first step towards understanding how to align SYCL with these proposals; additional work is required to understand how P2300 concepts such as *senders*, *receivers* and *schedulers* can interact with ND-range parallelism.

3 MOTIVATION

3.1 Evolving Hardware Architectures

SYCL programs can be mapped to multiple architectures, via multiple backends. In order to support such a wide variety of hardware and implementation options, SYCL and other similarly portable programming languages/frameworks define a core feature set that represents the capabilities of the lowest common denominator. Features that cannot be supported everywhere are made available as optional features, to bridge the gap to more capable devices.

```
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 other groups to arrive
    while (atomic_counter.load() != expected) {}
  }

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

Figure 1: A device-wide barrier implemented using SYCL atomic references and group barriers.

Many modern architectures are capable of providing stronger forward progress guarantees than the architectures that existed when portable language specifications like SYCL were first introduced. We've recently seen other programming languages undergo some quite significant changes to reflect this, as discussed below. Curiously, each language has adopted a different solution to the problem, focusing on providing different guarantees—we surmise that these changes have been driven by the concerns of different groups of vendors and/or developers, and are ultimately associated with changes in specific hardware architectures.

OpenCL 2.0 [16] introduced the concept of *independent forward progress*, but did so only for sub-groups. Then, in CUDA 9.0 [7], NVIDIA introduced a new feature called *independent thread scheduling* that strengthened the forward progress guarantees of individual CUDA threads when executing on GPUs based on the Volta microarchitecture (or later). Most recently, the OpenMP language committee have revealed plans to introduce a new *progress unit* concept in OpenMP 6.0 [6], which will allow developers to reason about the mapping of OpenMP threads to hardware threads. All of these solutions ignore the hierarchical nature of modern parallel architectures, and consequently none of them are flexible enough to cover the range of hardware that SYCL targets.

3.2 Software Specialization

It is not uncommon for developers to *specialize* their applications, deliberately writing non-portable code in exchange for improved performance [23]. However, divergence in hardware capabilities can potentially lead to unintentional specialization, when developers unwittingly design algorithms that rely on device-specific behavior (e.g. "warp-synchronous programming" [10]) that is not covered by the language specification (i.e. *undefined behavior*).

Consider the code in Figure 1, which uses SYCL atomic references and group barriers to construct a device-wide barrier that synchronizes all work-items executing the same kernel. This code is syntactically valid and can be compiled by any conforming SYCL

compiler. However, there are many device/implementation configurations where the code will not execute correctly, because it makes invalid assumptions about the SYCL execution model. Although often surprising to developers used to CPUs with strong forward progress guarantees, the SYCL specification allows implementations to keep executing the loop that checks the value of `atomic_counter` on a single work-item forever, preventing other work-items from signaling that they have arrived at the barrier. Providing implementations some flexibility in situations like these is what enables SYCL to support a wide variety of devices.

But, due to this flexibility, there are also device/implementation configurations where the code *will* execute correctly, or will only execute correctly some of the time [24]. There is therefore a risk that developers will not learn that their code is non-portable until they attempt to run it on a new machine or make seemingly unrelated changes (e.g. running a different problem size).

The types of developers writing such kernels are typically expert programmers, and so readers may be tempted to dismiss the issue as impacting only a very small number of developers. It is important to remember that many developers interact with SYCL via a higher-level abstraction, such as a library (e.g. the oneAPI DPC++ Library [11] or the oneAPI Deep Neural Network Library [15]), a portability framework (e.g. Kokkos [13] or RAJA [5]), or a domain-specific framework (e.g. TensorFlow [14]). Providing more information to these higher-level abstractions may enable expert programmers to further tune for specific hardware, improving performance without exposing the average programmer to any additional complexity or the risk of deadlock.

4 TERMINOLOGY

The introduction of parallelism to the C++ standard library (which includes support for parallel versions of algorithms like `for_each`) brought with it some new terminology for describing the scheduling of parallel work and the interactions between parallel workers.

4.1 Threads of Execution and Work-items

The C++17 specification defines the behavior of parallel and concurrent programs in terms of *threads of execution*, which are said to make progress when performing certain operations (including, but not limited to, atomic operations).

When an appropriate execution policy is provided, each invocation of a function object handed to one of the standard parallel algorithms can be executed by a different thread of execution. This aligns very well with how SYCL currently describes the execution of kernel functions in Section 3.2:

> "Each instance of the SYCL kernel function will be executed as a single, though not necessarily entirely independent, flow of execution ..."

We believe that threads of execution are equivalent to the SYCL concept of work-items, and therefore recommend that the SYCL specification clarifies that work-items are threads of execution.

Note that the term *execution agent* is used elsewhere in the C++17 specification to describe a broader class of parallel worker that includes threads of execution and implementation- or user-defined workers (e.g. tasks). Our discussion focuses on threads of execution because agents are not mentioned in the context of execution policies or parallel algorithms. Furthermore, assuming that work-items are threads of execution ensures that they also meet the definition of execution agent.

4.2 Forward Progress Guarantees

The C++17 specification additionally defines three different classes of forward progress guarantees that can be associated with a given thread of execution (henceforth referred to as "a thread" for brevity):

(1) Threads with *concurrent forward progress guarantees* eventually execute their first step. Once the first step has been executed, the thread continues to make progress until it terminates.
(2) Threads with *parallel forward progress guarantees* are not required to execute their first step. If the first step is executed, the thread continues to make progress until it terminates.
(3) Threads with *weakly parallel forward progress guarantees* effectively have no guarantees.

When SYCL talks about the forward progress guarantees of different work-items, the phrasing implies that these guarantees are a pairwise relationship. For example, Section 3.8.3.4 says that:

> "... forward progress guarantees with respect to other work-items is implementation-defined."

However, if work-items are equivalent to threads, then this phrasing is incorrect — each individual work-item should provide its own forward progress guarantees, independent of the existence of other work-items. The same section also says that implementations must:

> "... execute work-items concurrently[1] 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."

We propose to instead define work-items as providing weakly parallel forward progress guarantees, and to clarify the scheduling guarantees provided by barriers. Specifically, we propose that the definition of group barriers be rewritten to include the below:

> "All implementations must additionally ensure that a work-item arriving at a group barrier does not prevent other work-items in the same group from making progress. 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."

This tighter language also clarifies that it is safe to elect a group leader to perform some operation on behalf of the other work-items in the group, so long as those work-items are waiting at a barrier. Strictly speaking, without this change there is no requirement for the leader of a group to make progress while other members wait for it to arrive. In reality, all SYCL implementations already implement this behavior since it is required for barriers to work as expected.

[1] i.e. "at the same time", not "with concurrent forward progress guarantees".

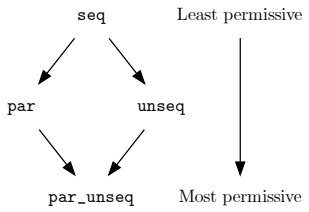

Figure 2: The partial ordering of execution policies defined in the `std::execution` namespace.

4.3 Blocking with Forward Progress Guarantee Delegation

Finally, the C++17 specification provides a mechanism called *blocking with forward progress guarantee delegation* that enables developers to ensure threads with parallel and weakly parallel forward progress guarantees eventually perform the work assigned to them. When one thread blocks on the completion of another thread with weaker forward progress guarantees, the weaker thread is temporarily strengthened.

Having established that SYCL device kernels are executed by threads with weakly parallel forward progress guarantees, it becomes apparent that the SYCL specification requires a mechanism to ensure that device code eventually executes. We propose to leverage the definition of blocking with forward progress guarantee delegation, such that the threads used to execute device code may be strengthened when a host thread blocks upon their completion (e.g. using `queue::wait` or similar). This not only clarifies the behavior of SYCL CPU devices that share resources with the host, but also clarifies that developers cannot typically rely upon concurrent execution of host and device code.

5 EXPRESSING REQUIREMENTS

Both C++ and SYCL offer multiple ways for developers to express asynchrony and parallelism in their applications. Understanding which of these approaches is most suitable for different use-cases is key to developer productivity.

In C++, different features correspond to different forward progress guarantees, allowing developers to express algorithmic requirements for correctness. Implementations are strongly recommended to ensure that `std::thread`s provide concurrent forward progress guarantees, and developers can influence the forward progress guarantees associated with threads implicitly created by a library's implementation of a parallel algorithm by selecting different *execution policies*.

C++17 defines four such policies (`seq`, `par`, `unseq` and `par_unseq`) in the `std::execution` namespace. These policies are partially ordered (as shown in Figure 2): `seq` requires a fully sequential implementation; `par` permits parallel execution; `unseq` permits unsequenced execution of invocations within a single thread of execution; and `par_unseq` permits both parallel execution and unsequenced execution. Although the specification does not mandate that implementations create multiple threads of execution for any of these policies, implementations that do so are required to provide parallel and weakly parallel forward progress guarantees for the `par` and `par_unseq` execution policies, respectively.

The schedulers proposed by P2300 [12] also have features related to forward progress guarantees. Although developers cannot request the creation of threads with specific guarantees, each scheduler advertises the guarantees that it can satisfy via a call to `get_forward_progress_guarantee()`.

In SYCL, the two forms of `sycl::parallel_for` accepting a `sycl::range` and `sycl::nd_range` have slightly different execution models. In the former, each work-item has weakly parallel forward progress guarantees; in the latter, each work-item has weakly parallel forward progress guarantees and some scheduling guarantees in the presence of group barriers. SYCL does not provide a way to request stronger forward progress guarantees on devices which support them, nor a way to query the guarantees that can be satisfied on specific devices. As a result, developers cannot reason about the ability of devices to safely execute certain synchronization/coordination patterns (such as critical sections and producer-consumer relationships) without deadlocking. This also makes SYCL an unsuitable target for a general-purpose implementation of the C++17 parallel algorithms, since only algorithms using the `par_unseq` execution policy can be safely translated into SYCL kernels and executed by SYCL work-items.

6 TOWARDS AN EXTENSION

As discussed in the previous section, SYCL does not yet provide a way to reason about a device's ability to provide strong forward progress guarantees. For certain algorithms, developers must therefore choose between either: a) writing high performance kernels that make unsafe assumptions about forward progress guarantees, which may fail when moved to other implementations, backends, devices, driver versions, etc; or b) writing portable kernels that do not make such assumptions, but which may fail to take advantage of the capabilities of some hardware.

In this section, we lay the groundwork for a SYCL extension that would allow developers to adopt a more balanced approach, eliminating the need to choose between performance and portability for these kernels. Targeting an extension allows us to prove out terminology, developer mental model, and implementability before folding changes into the SYCL specification itself.

6.1 Mental Model

The design of our proposed extension stems from a thought experiment, where SYCL is implemented using only features provided by C++. In such an implementation, we could employ a hierarchical arrangement of `std::for_each` calls to represent the hierarchical arrangement of SYCL work-items. Figure 3 shows what this may look like in pseudo-code, using one `std::for_each` for work-groups, and another for work-items. Note that a real implementation would be more complex than this, as our simple pseudo-code implementation ignores several SYCL features (e.g. sub-groups, group barriers and multi-dimensional ranges).

This thought experiment highlights several interesting points:

(1) In addition to there being threads associated with each work-item, there are also threads associated with each work-group.

```
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) {

    // NB: An additional for_each would be required here
    //     to support non-trivial sub-groups. Omitted for brevity.

    // Create a separate 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
      auto item = detail::make_nd_item<1>(group_id, item_id);
      f(item);

    });

  });

}
```

Figure 3: A simplified hypothetical implementation of SYCL implemented with C++17 parallel algorithms.

(2) Threads at different levels of the hierarchy may have different forward progress guarantees, since they may have been created by algorithms using different execution policies.
(3) Threads at the outer level block with forward progress guarantee delegation on threads at the inner level.
(4) Not all threads are created at the same time. The conditions in which a given thread is created at work-item scope is dependent on the forward progress guarantees associated with some thread at work-group scope.

Modelling program behavior as if kernels are executed by an implementation like the one above thus gives us the necessary framework to describe and reason about forward progress guarantees in SYCL.

Figure 4 shows a visualization of our proposed view of the thread hierarchy in SYCL device kernels, for a kernel with two work-groups, two sub-groups per work-group, and four work-items per sub-group. For simplicity, only the left-most branches in the figure are expanded.

At the root, we introduce a new group type (a "root-group") containing all of the work-items executing the kernel. This root-group is associated with a single thread, and creates additional threads for each work-group in the kernel. Each work-group then creates threads for each of its constituent sub-groups, and each sub-group creates a thread for each of its constituent work-items. Each work-item thread is responsible for executing a single invocation of the device kernel function.

6.2 Mapping to Existing Models

For our new mental model to be suitable for SYCL, it must be possible to map it to the various backends that SYCL implementations can already target. Table 1 shows two possible mappings: one where

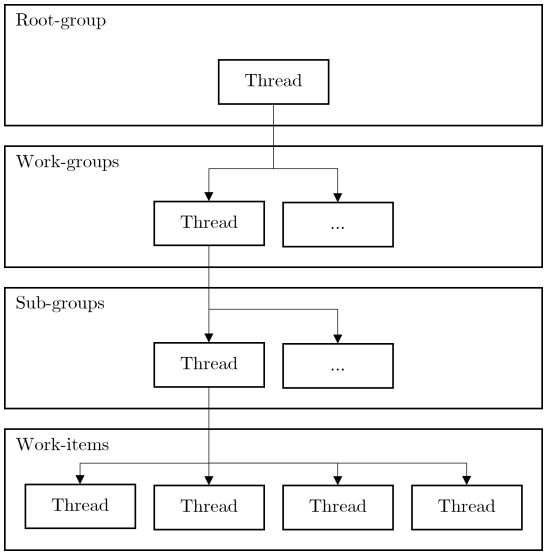

Figure 4: Proposed thread hierarchy for SYCL device kernels.

Table 1: The forward progress guarantees expected to be provided by SYCL implementations targeting OpenCL backends.

Thread	OpenCL 1.x	OpenCL 2.x
Host	Concurrent	Concurrent
Work-group	Weakly parallel	Weakly parallel
Sub-group	Weakly parallel	Concurrent
Work-item	Weakly parallel	Weakly parallel

SYCL targets an OpenCL 1.x backend; and another where SYCL targets an OpenCL 2.x backend with support for sub-group independent forward progress. In both cases, we assume that the host thread is the one running `main`, and that it provides concurrent forward progress guarantees.

6.2.1 OpenCL 1.x. The mapping to OpenCL 1.x is the simplest possible mapping, because OpenCL 1.x does not provide any scheduling guarantees for work-items beyond a guarantee that some work-items will eventually begin executing when the kernel is submitted for execution. As a result, we can simply describe all the threads executing device code as providing weakly parallel forward progress guarantees.

6.2.2 OpenCL 2.x. The mapping to OpenCL 2.x is only slightly more complicated, with each thread at sub-group scope now providing concurrent forward progress guarantees. Note that since the OpenCL specification does not use C++17 terminology to describe forward progress guarantees, the mapping is open to interpretation. Our decision to use concurrent here is based on Section 3.2.2 of the OpenCL 2.2 specification, which states:

> "Sub-groups execute concurrently[2] within a given work-group and with appropriate device support ...

[2] i.e. "at the same time", not "with concurrent forward progress guarantees".

> may[3] make independent forward progress with respect to each other, with respect to host threads and with respect to any entities external to the OpenCL system but running on an OpenCL device, even in the absence of work-group barrier operations."

This statement would clearly not be true if sub-groups provided weakly parallel forward progress guarantees. We believe that parallel forward progress guarantees would not be sufficient either, since in the absence of work-group barrier operations there would be no guarantee that all sub-groups in a work-group would ever begin executing.

6.2.3 Other Devices and Backends. Sorensen et al. [24–26] have previously demonstrated that many GPUs obey their OBE model. Under OBE, any work-group that begins execution continues executing until it terminates; in our mapping, this is equivalent to each work-group thread providing parallel forward progress guarantees.

Many of these same GPUs can be targeted by a backend that supports some form of *cooperative launch* mechanism (e.g. CUDA's `cudaLaunchCooperativeKernel` and Level Zero's `zeCommandListAppendLaunchCooperativeKernel`). A cooperative launch ensures that all work-groups are created together, and requires that the total number of work-groups does not exceed the maximum number of work-groups that the device can execute simultaneously. OBE paired with a cooperative launch is equivalent to each work-group thread providing concurrent forward progress guarantees.

The forward progress guarantees provided at each level of our hierarchy are thus not necessarily a static property of a given device-backend configuration, and may depend on other factors (e.g. how a kernel is launched, or the size of its ND-range). This is an important observation, and one that must be accounted for in the design of any new language features relating to forward progress guarantees.

6.3 Coordination Scopes

There is a subtlety in the OpenCL 2.x mapping from Table 1 that deserves further examination. Recall from Section 6.1 that the creation of a particular thread may be dependent upon a thread previously created at an enclosing (broader) scope. This effectively means that there are two ways for a developer to reason about the forward progress guarantees associated with sub-groups, corresponding to two different use-cases.

Coordination of sub-groups within a single work-group (which CUDA developers refer to as *warp specialization* [3, 4, 21]) is safe in OpenCL 2.x because it only requires that all sub-groups within the same work-group provide concurrent forward progress guarantees. However, coordination of sub-groups across multiple work-groups is not safe, because the work-groups responsible for creating the sub-groups only provide weakly parallel forward progress guarantees. The dependency upon another thread's creation has effectively weakened each sub-group's forward progress guarantees; there is no guarantee that a given sub-group will ever execute its first step, because there is no guarantee that its parent work-group will.

In order to simplify discussion of requirements that span multiple levels of the hierarchy, we introduce the concept of a *coordination scope*. Pairing guarantees with a coordination scope effectively collapses multiple levels of the hierarchy, replacing the guarantees provided at each level with the weakest guarantee provided at any level. To demonstrate the behavior of coordination scopes, let us revisit the two sub-group use-cases discussed above.

The first use-case requires developers to reason about the forward progress guarantees provided by sub-groups at work-group coordination scope (because only sub-groups within the same work-group are involved in the coordination). Since there are no intermediate levels between work-groups and sub-groups, there are no levels to collapse, and the forward progress guarantees provided by sub-groups at work-group coordination scope in OpenCL 2.x are therefore equivalent to the forward progress guarantees provided by sub-groups.

The second use-case requires developers to reason about the forward progress guarantees provided by sub-groups at root-group coordination scope (because sub-groups in different work-groups are still part of the same root-group). There is one intermediate (work-group) level between the root-group and sub-groups that must be collapsed. The forward progress guarantees provided by sub-groups at root-group coordination scope in OpenCL 2.x are therefore only weakly parallel, representing the weakest of the guarantees provided by work-groups and sub-groups.

```
enum class forward_progress_guarantee {
  concurrent,
  parallel,
  weakly_parallel
}
```

Figure 5: The `sycl::forward_progress_guarantee` enumeration type.

7 PROPOSED EXTENSION FOR FORWARD PROGRESS GUARANTEES

In this section, we propose a new set of language features for encoding our new mental model into SYCL. We believe that other languages could adopt similar extensions, but their design may need some modification.

To aid readability, we show the features from our proposed extension in the `sycl` namespace, rather than a more verbose `sycl::ext::<vendorstring>` namespace that would be required by a real implementation.

7.1 Scopes and Progress Guarantees

Our extension introduces two new enumeration types, used to formulate queries and represent their results. The first of these enumeration types, `sycl::forward_progress_guarantee`, is taken from P2300 [12] and its values are shown in Figure 5. The possible values of the second enumeration type, `sycl::execution_scope`, are shown in Figure 6.

An `execution_scope` is used to define the level of the hierarchy of interest to a query (henceforth the *target* scope) and the desired *coordination* scope (as defined in Section 6.3).

7.2 Device Queries

Inspired by the scheduler query from P2300 [12], we propose to introduce the device query shown in Figure 7. The intent of this

[3]The `cl_khr_subgroups` extension from OpenCL 2.0 requires sub-groups to make "independent forward progress", but this is optional in the OpenCL 2.1 core feature.

```
enum class execution_scope {
  work_item,
  sub_group,
  work_group,
  root_group,
}
```

Figure 6: The `sycl::execution_scope` enumeration type.

```
template <sycl::execution_scope Scope,
          sycl::execution_scope CoordinationScope>
struct sycl::info::device::forward_progress_guarantee;
```

Figure 7: The device information descriptor for the forward progress device query.

```
template <sycl::forward_progress_guarantee Guarantee,
          sycl::execution_scope CoordinationScope>
inline constexpr
work_group_progress_key::value_t<Guarantee, CoordinationScope>
work_group_progress;

template <sycl::forward_progress_guarantee Guarantee,
          sycl::execution_scope CoordinationScope>
inline constexpr
sub_group_progress_key::value_t<Guarantee, CoordinationScope>
sub_group_progress;

template <sycl::forward_progress_guarantee Guarantee,
          sycl::execution_scope CoordinationScope>
inline constexpr
work_item_progress_key::value_t<Guarantee, CoordinationScope>
work_item_progress;
```

Figure 8: Compile-time kernel properties describing required forward progress guarantees.

query is to enable expert developers to write their own device specialization and kernel dispatch mechanisms.

The query returns the *strongest* forward progress guarantees that a device can provide for threads at the specified target scope, given a coordination scope. We choose to return the strongest guarantees for simplicity, since any device that can satisfy a request for strong guarantees could trivially satisfy weaker guarantees.

Note that we do not provide any way to query whether a device provides stronger than weakly parallel forward progress guarantees *by default*. We believe that it will be better in the long run to require explicit requests for behaviors not mandated by the SYCL specification, to discourage unsafe assumptions and to ensure developers remain aware of additional factors that may affect forward progress guarantees (see Section 6.2.3). This philosophy is aligned with SYCL's approach to features like atomic references, where there is no default memory order or scope.

7.3 Kernel Properties

The DPC++ compiler already supports SYCL extensions providing compile-time properties [8] and an ability to embed such properties into kernel definitions [9]. At the time of writing, the most common use-case for these kernel properties is to set the desired sub-group size, using a `sycl::sub_group_size<S>` property in place of the existing (but less robust) `[[sycl::reqd_sub_group_size(s)]]` kernel attribute.

```
// Maximum number of work-items in the kernel
template <uint32_t Dimensions>
struct max_work_item_sizes;

// Maximum work-group size and number of work-groups
struct max_work_group_size;
struct max_num_work_groups;

// Maximum sub-group size and number of sub-groups
struct max_sub_group_size;
struct max_num_sub_groups;
```

Figure 9: Initial set of kernel launch queries.

Owing to the flexibility of compile-time property lists, we can leverage them here to declare requirements for specific forward progress guarantees. Figure 8 shows three new properties, allowing developers to request specific forward progress guarantees for a combination of target scope (encoded in the property name) and coordination scope (as a template parameter).

As with device queries, coordination scopes provide a shorthand for simply and succinctly making requests that span multiple levels of the hierarchy. For example, a kernel compiled with the `sycl::work_item_progress<sycl::execution_scope::root_group, sycl::forward_progress_guarantee::parallel>` property will require all work-items across all sub-groups and work-groups to provide parallel forward progress guarantees (equivalent to CUDA independent thread scheduling).

Note that we deliberately do not define a `sycl::root_group_progress` property, since it would be impossible to provide a broader coordination scope. Specifying the forward progress guarantees required at the highest level of the hierarchy would be equivalent to specifying the guarantees of the whole kernel relative to other kernels, to other devices, and to host code—and such guarantees are necessary to safely pass messages between devices. Although this is an important use-case, we have decided to defer the solution to future work. Progress requirements at the kernel level may require additional attention to other parts of SYCL (e.g. command-group scheduling) and may ultimately be better expressed with an alternative mechanism or distinct compile-time properties.

7.4 Kernel Launch Queries

As discussed in Section 6.2.3, some devices may only be able to satisfy requests for specific forward progress guarantees if the developer promises to obey specific launch limits. An initial set of queries providing the values of these launch limits for specific kernels (henceforth *kernel launch queries*) is given in Figure 9.

This initial set of kernel launch queries is based on our implementation experience with four backends: an OpenCL backend for Intel CPUs, an OpenCL backend for Intel GPUs, a Level Zero backend for Intel GPUs, and a CUDA backend for NVIDIA GPUs. As we gain implementation experience with additional devices and backends, it may be necessary to introduce support for additional limits via additional queries.

In many of the backends listed above, the launch limit depends on more than just the kernel. In all cases, the limit is expected to be device- (or perhaps even queue-) specific. In some cases, the limit may even depend on some other component of the launch configuration. For example, the maximum number of work-groups

```
/* Available only when arguments correspond to Param::value_types */
template <typename Param, typename...T >
typename Param::return_type get_info(T... args) const;
```

Figure 10: An extended form of `kernel::get_info` accepting additional, query-specific, arguments.

that can be executed concurrently often depends on the developer's desired work-group size and local memory requirements.

To address this, we propose replacing SYCL's existing `kernel::get_info` function with the variant shown in Figure 10. This variant allows each kernel query to define not only its return type, but also the type and number of arguments it expects. This design is primarily intended to shield SYCL from unknown future changes in launch limits, but could also be used to simplify the SYCL specification by combining the information descriptors from the `kernel` and `kernel_device_specific` namespaces.

To maximize portability, it is important that the extension allows for some flexibility in implementations. The combination of device queries and launch limits provide that flexibility. If an implementation wants to support all SYCL kernels without any performance guarantees, it can limit support to trivial cases (e.g. requiring all kernels to be launched with a single work-group). If an implementation would prefer to advertise support for only configurations that can achieve high levels of performance, it can opt-out of supporting specific configurations via the device queries in Section 7.2.

7.5 Usage Examples

To demonstrate how the elements of our extension fit together, let us revisit the example from Figure 1. Recall that such a device-wide barrier function is not guaranteed to work on all SYCL devices, since it relies on specific forward progress guarantees for work-groups. The usage examples in this section show how our extension enables developers to use such functions in a safe and portable manner.

Generally speaking, safe submission of a kernel containing a call to our `arrive_and_wait` function requires four steps:

(1) Using compile-time properties to associate a kernel's required forward progress guarantees with its definition.
(2) Using device queries to check that the device supports the forward progress guarantees required by a kernel.
(3) Using kernel launch queries to determine any device-specific limitations on launching the kernel correctly.
(4) Launching the kernel with a valid configuration.

Figure 11 shows step (1) for a kernel expressed as a function object. The `get` function and `properties_tag` are used by the implementation to extract the user-supplied compile-time property list, and the properties are associated with every instance of `MyKernel`.

Figure 12 shows step (4) for this kernel, omitting steps (2) and (3) in favor of a `try-catch` block. The kernel submission in step (4) will throw an exception if the kernel requirements cannot be satisfied or the ND-range is invalid, and we expect several real-life use-cases will rely on this behavior rather than querying device capabilities or kernel launch requirements.

Figure 13 shows a more complex example that includes steps (2) and (3), where the code explicitly tests for device support and selects an ND-range based on device capabilities. We expect this

```
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());
    ...
  }

  // (1) 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;

};
```

Figure 11: An example of encoding forward progress guarantee requirements via compile-time properties.

```
try {
  // (4) 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
}
```

Figure 12: An example of protecting against unsafe calls to `arrive_and_wait` via a `try-catch` block.

```
// (2) 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)
{
  // (3) 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);

  // (4) 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
{
  // Fall back to an alternative kernel implementation,
  // throw an exception, or exit with an error
}
```

Figure 13: An example showing usage of device queries and kernel launch queries in place of a `try-catch` block.

pattern to be common in libraries and frameworks that execute specialized code paths for different devices.

It may still be necessary or desirable to define higher-level properties and queries to simplify common use-cases (e.g. a dedicated "does this device support device-wide barriers" query). However, we believe that such functionality could be implemented in terms of the features introduced here—and likely as libraries, rather than as additional core features—and so we leave them to future work.

7.6 Performance, Portability and Productivity

Our proposed extension was designed to strike a good balance between performance, portability and productivity (P3) concerns. By leaving the default behavior of SYCL unchanged, developers can continue to write portable applications that are capable of running on any SYCL device. But by exposing lower-level controls for opt-in features, our extension enables individual developers to make different P3 trade-offs for different applications and use-cases.

Requiring instead that all SYCL implementations provide strong progress guarantees for all work-groups, sub-groups and work-items (or somehow automatically detect when such guarantees are necessary for correctness) would have placed significant burden on SYCL implementations. We believe it would also have decreased the performance of most current implementations, which rely on an assumption of weakly parallel forward progress guarantees to enable common optimizations (e.g. vectorization over work-items).

8 CONCLUSION AND FUTURE WORK

The terminology changes proposed by this work have already been approved by the Khronos SYCL Working Group and are expected to appear in the next revision of the SYCL specification (SYCL 2020 Revision 7). Keeping C++ and SYCL aligned will require constant and ongoing effort, but we believe that this effort is necessary for SYCL to succeed in influencing the future of heterogeneous computing within C++.

The extension discussed here is just one of the paths being considered to unify both execution models and the code patterns needed to achieve performance on modern parallel hardware. Like all extensions, it must prove itself before being adopted by the specification, and so we welcome community feedback at www.github.com/intel/llvm. As the design of the extension settles, and we gain more implementation experience, we will seek opportunities to apply our findings in other contexts (e.g. OpenCL, SPIR-V, Vulkan).

DISCLAIMERS

REFERENCES

[1] Ruslan Arutyunyan. 2022. *C++17 Parallel Algorithms and P2300.* Technical Report. http://wg21.link/p2500

[2] Ben Ashbaugh, James C Brodman, Michael Kinsner, Gregory Lueck, John Pennycook, and Roland Schulz. 2021. Toward a Better Defined SYCL Memory Consistency Model. In *International Workshop on OpenCL (IWOCL'21).* Association for Computing Machinery, New York, NY, USA, Article 20, 3 pages.

[3] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) *(SC '11).* Association for Computing Machinery, New York, NY, USA, Article 12, 11 pages.

[4] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: Leveraging Warp Specialization for High Performance on GPUs. *SIGPLAN Not.* 49, 8 (feb 2014), 119–130.

[5] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* IEEE, 71–81.

[6] OpenMP Architecture Review Board. 2022. *OpenMP Application Programming Interface Version 6.0 Preview 1.*

[7] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and Programmability. *IEEE Micro* 38, 2 (2018), 42–52.

[8] Intel Corporation. 2021. *sycl_ext_oneapi_properties.* Retrieved January 12, 2023 from https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_properties.asciidoc

[9] Intel Corporation. 2022. *sycl_ext_oneapi_kernel_properties.* Retrieved January 12, 2023 from https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_properties.asciidoc

[10] NVIDIA Corporation. 2022. Tuning CUDA Applications for Volta. Retrieved January 12, 2023 from https://docs.nvidia.com/cuda/pdf/Volta_Tuning_Guide.pdf

[11] Tom Deakin, Simon McIntosh-Smith, S. John Pennycook, and Jason Sewall. 2021. Analyzing Reduction Abstraction Capabilities. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* 33–44.

[12] Michał Dominiak, Lewis Baker, Lees Howes, Kirk Shoop, Michael Garland, Eric Niebler, and Bryce Adelstein Lelbach. 2021. *std::execution.* Technical Report. http://wg21.link/p2300

[13] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[14] Mehdi Goli, Luke Iwanski, and Andrew Richards. 2017. Accelerated machine learning using TensorFlow and SYCL on OpenCL Devices. In *Proceedings of the 5th International Workshop on OpenCL.* 1–4.

[15] Mehdi Goli, Kumudha Narasimhan, Ruyman Reyes, Ben Tracy, Daniel Soutar, Svetlozar Georgiev, Evarist M Fomenko, and Eugene Chereshnev. 2020. Towards Cross-Platform Performance Portability of DNN Models using SYCL. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* IEEE, 25–35.

[16] Khronos OpenCL Working Group. 2015. *The OpenCL Specification, Version 2.0, Revision 29.*

[17] Khronos OpenCL Working Group. 2022. *The OpenCL Specification, Version 3.0.*

[18] Khronos SYCL Working Group. 2022. *SYCL 2020 Specification (revision 6).*

[19] The Khronos Group Inc. 2023. SYCL Overview. Retrieved January 11, 2023 from https://www.khronos.org/sycl/

[20] ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). 1605 pages. https://www.iso.org/standard/68564.html

[21] Arpith Chacko Jacob, Alexandre E Eichenberger, Hyojin Sung, Samuel F Antao, Gheorghe-Teodor Bercea, Carlo Bertolli, Alexey Bataev, Tian Jin, Tong Chen, Zehra Sura, Georgios Rokos, and Kevin O'Brien. 2017. Efficient Fork-Join on GPUs Through Warp Specialization. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC).* 358–367.

[22] John Kessenich, Boaz Ouriel, and Raun Krisch. 2022. *SPIR-V Specification, Version 1.6, Revision 2.*

[23] S. John Pennycook, Jason D. Sewall, Douglas W. Jacobsen, Tom Deakin, and Simon McIntosh-Smith. 2021. Navigating Performance, Portability, and Productivity. *Computing in Science & Engineering* 23, 5 (2021), 28–38.

[24] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2016. Portable Inter-Workgroup Barrier Synchronisation for GPUs. *SIGPLAN Not.* 51, 10 (oct 2016), 39–58.

[25] Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. 2018. GPU Schedulers: How Fair Is Fair Enough?. In *29th International Conference on Concurrency Theory (CONCUR 2018).* 23:1–23:17.

[26] Tyler Sorensen, Lucas F. Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F. Donaldson. 2021. Specifying and Testing GPU Workgroup Progress Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 131 (oct 2021), 30 pages.