IWOCL & SYCLcon 2021

Toward a Better Defined SYCL Memory Consistency Model

Presenter: Ben Ashbaugh

Co-Authors: James Brodman, Michael Kinsner, Gregory Lueck, John Pennycook, Roland Schulz

intel®

Agenda

- Why does a memory model matter?
- What was in SYCL 1.2.1?
- Other Memory Models
  - C++
  - OpenCL 2.0
  - Vulkan
- What is in SYCL 2020?
- Topics for the next SYCL version?

intel.

Why does a memory model matter?

- Your code may not execute exactly as you wrote it!
- Possible areas of uncertainty:
  - The **Compiler** may reorder instructions
  - The **Hardware** may execute instructions out-of-order
  - There may be **Caching** effects (e.g. multi-level caches, device-side caches)
- A **Memory Model** is a contract to constrain this uncertainty:
  - Guarantees behavior for application developers
  - Describes valid and invalid optimizations for implementers

Example:

```
int X = 0, Y = 0;
```

Thread #1

```
X = 1;
Y = 2;
```

Thread #2

```
if (Y == 2) {
    // OK? No ☹
    assert(X == 1);
}
```

A Memory Model describes how to make this OK!

Alternate Title:

Better Memory Model A SYCL We Need!

What was in SYCL 1.2.1?

- SYCL 1.2.1 Memory Model derived from OpenCL 1.2
  - Can guarantee memory consistency within a work-group with a fence or barrier
  - … probably.

Work-item 1

```
X = 1;
item.mem_fence();
// Note: atomic store:
Y.store(2);
```

Work-item 2 (same work group)

```
// Note: atomic load:
if (Y.load() == 2) {
  item.mem_fence();
  assert(X == 1); // OK!
}
```

What was in SYCL 1.2.1?

- SYCL 1.2.1 Memory Model derived from OpenCL 1.2
  - Can guarantee memory consistency within a work-group with a fence or barrier
  - No memory consistency guarantees across work-groups or kernels!

Work-item 1

```
X = 1;
item.mem_fence();
// Note: atomic store:
Y.store(2);
```

Work-item 2 (different work group)

```
// Note: atomic load:
if (Y.load() == 2) {
  item.mem_fence();
  assert(X == 1); // Not OK
}                     ☹
```

Other Memory Models

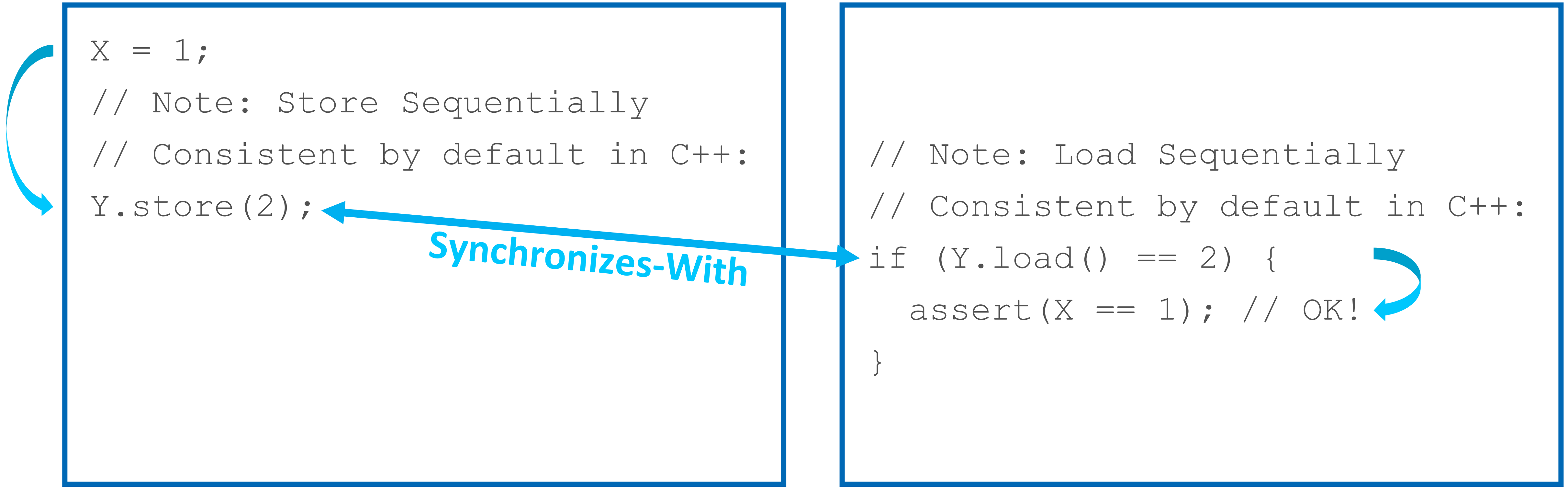
C++11

C++11 Memory Model

- C++11 defined behavior for multiple threads of execution
  - Many concepts directly applicable to SYCL data parallel programming
- Key definitions: Memory Location, Conflicting Action, Data Race
- Key definitions: Sequenced-Before, Synchronizes-With, Happens-Before
- Key definitions: Memory Order:
  - Sequential Consistency, Acquire and Release, Relaxed Consistency
- Key additions: **`std::atomic<>`**, **`atomic_thread_fence()`**

Example: C++11 Sequentially Consistent Atomics

```
int X = 0; std::atomic<int> Y(0);
```



Thread 1

```
X = 1;
// Note: Store Sequentially
// Consistent by default in C++:
Y.store(2);
```

Thread 2

```
// Note: Load Sequentially
// Consistent by default in C++:
if (Y.load() == 2) {
  assert(X == 1); // OK!
}
```

What does Sequential Consistency mean?

- Informally: There is one modification order, all threads agree what it is.

```
std::atomic<int> X(0), Y(0);
```

Writer Thread 1:

```
X.store(1);
```

Writer Thread 2:

```
Y.store(1);
```

Reader Thread 1:

```
// note: X first, then Y!
int A = X.load(); // If 1...
int B = Y.load(); // If 0...
```

Reader Thread 2:

```
// note: Y first, then X!
int C = Y.load(); // If 1...
int D = X.load(); // Must be 1!
```

Write to Y happened after write to X!

intel.

Example: C++11 Acquire-Release Atomics

```
int X = 0; std::atomic<int> Y(0);
```

Thread #1

```
X = 1;
Y.store(2, memory_order_release);
```

Synchronizes-With

Thread #2

```
if (Y.load(memory_order_acquire)
    == 2) {
  assert(X == 1); // OK!
}
```

Difference between Acquire-Release and Sequential Consistency:

- Informally: Acquire-Release orders memory stores within threads, but not across threads.

```
std::atomic<int> X(0), Y(0);
```

Writer Thread 1:

```
X.store(1, release);
```

Writer Thread 2:

```
Y.store(1, release);
```

Reader Thread 1:

```
// note: X first, then Y!
int A = X.load(acquire); // If 1
int B = Y.load(acquire); // If 0
```

Reader Thread 2:

```
// note: Y first, then X!
int C = Y.load(acquire); // If 1
int D = X.load(acquire); // ???
```

What about Relaxed Consistency?

- Relaxed Consistency provide read-modify-write atomicity…
  - … but no Happens-Before relation
  - Hence no memory consistency guarantees
- Still useful, but only in limited cases:
  - Reductions, Histograms, Bump Allocators, Counters, Progress Bars

```
std::atomic<int> counter(0);
counter.fetch_add(1, std::memory_order_relaxed); // x 100
assert(counter == 100);
```

C++11 Memory Model Limitations

- Assumption: All memory is visible to all threads
  - Not true for SYCL devices (may have device-specific memories)
  - Not true for SYCL address spaces (**`global`**, **`local`**, **`private`**)

Still, a good basis for a SYCL memory model.

Important to be compatible with C++ anyhow!

Other Memory Models

OpenCL 2.0

OpenCL 2.0 Memory Model

- Based upon C++11/C11
  - Reuses: Memory Location, Conflicting Action, Data Race, Memory Orders
  - Reuses: Sequenced-Before, Synchronizes-With, Happens-Before
- Key addition: Memory Scope
  - Work-group, Entire Device, All Devices
  - Constrains memory consistency to improve performance
- Key addition: Memory Flags
  - Global or Local (or both)
  - Constrains memory spaces to improve performance

OpenCL 2.0 Memory Model Examples

```
// An acquire-release operation for all work-items in the work-group:
atomic_fetch_add_explicit(pointer_to_atomic_value, operand,
    memory_order_acq_rel, memory_scope_work_group);

// A sequentially consistent operation for all work-items on all SVM devices:
atomic_fetch_add_explicit(pointer_to_atomic_value, operand,
    memory_order_seq_cst, memory_scope_all_svm_devices);

// A release fence for global memory and local memory, for all work-items in
// the work-group:
atomic_work_item_fence(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE,
    memory_order_release, memory_scope_work_group);
```

OpenCL 2.0 Memory Model Limitations

- Cumbersome to use:
  - Added atomic types: `atomic_int`, `atomic_float`, etc.
  - Doesn't (easily) support mixed atomic and non-atomic access
- **Separate Happens-Before relations adds significant complexity!**
  - A flag in one address space cannot enforce consistency in another address space!
  - Generic address space adds even more complexity

Refer to: Overhauling SC Atomics in C11 and OpenCL (Batty et al.)

Implications of Separate Happens-Before:

```
global int* x; local atomic_int* y;
```

Work-item 1

```
// Note: global memory:
*x = 1;
// Note: local memory:
atomic_store_explicit(y, 2,
  memory_order_release);
```

No Happens-Before!

Synchronizes-With

Work-item 2

```
if (atomic_load_explicit(y,
      memory_order_acquire)
      == 2) {
    assert(*x == 1); // Not OK
}
```

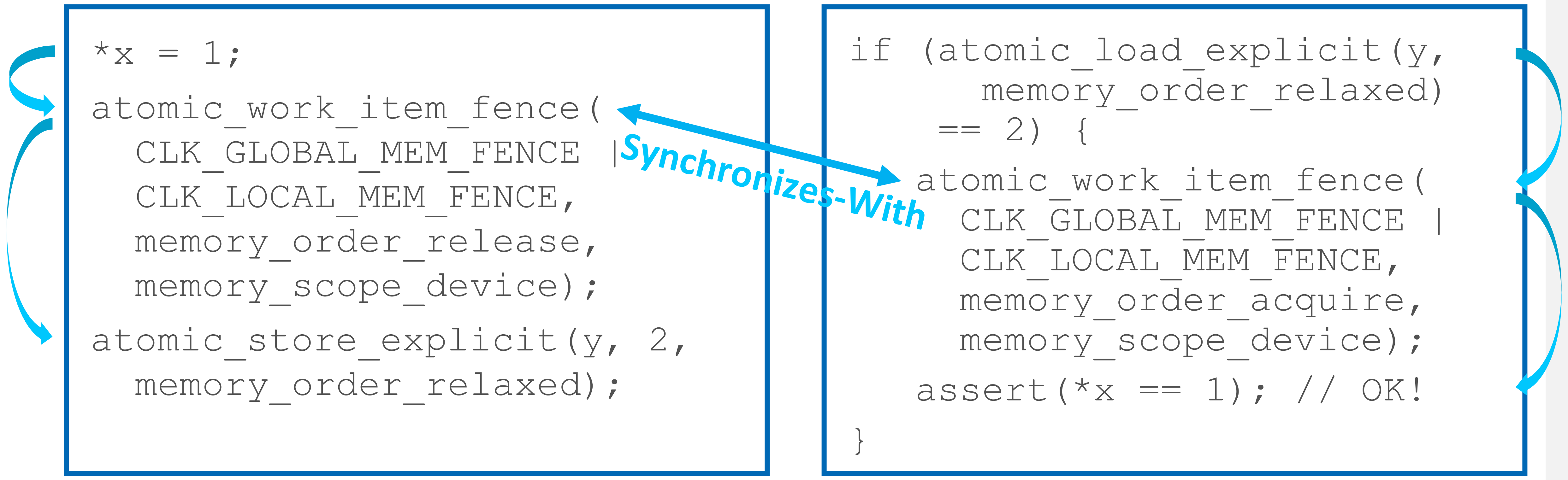
☹

No Happens-Before!

intel.

Correct Code with Separate Happens-Before:

```
global int* x; local atomic_int* y;
```



Work-item 1

```
*x = 1;
atomic_work_item_fence(
  CLK_GLOBAL_MEM_FENCE |
  CLK_LOCAL_MEM_FENCE,
  memory_order_release,
  memory_scope_device);
atomic_store_explicit(y, 2,
  memory_order_relaxed);
```

Work-item 2

```
if (atomic_load_explicit(y,
      memory_order_relaxed)
    == 2) {
   atomic_work_item_fence(
     CLK_GLOBAL_MEM_FENCE |
     CLK_LOCAL_MEM_FENCE,
     memory_order_acquire,
     memory_scope_device);
   assert(*x == 1); // OK!
}
```

Other Memory Models

Vulkan

Vulkan Memory Model

- Also based upon C++11/C11, with added Scopes and Storage Classes
- Key addition: Availability and Visibility

C++ Memory Order	Vulkan Memory Model SPIR-V Memory Semantics
`memory_order_relaxed`	None
`memory_order_acquire`	`MakeVisible | Acquire`
`memory_order_release`	`MakeAvailable | Release`
`memory_order_acq_rel`	`MakeAvailable | MakeVisible | AcquireRelease`
`memory_order_seq_cst`	**No mapping available!**

- Limitation: No support for default C++ semantics (`seq_cst`)

Table from: https://www.khronos.org/blog/comparing-the-vulkan-spir-v-memory-model-to-cs

What is in SYCL 2020?

SYCL 2020 Memory Model

- Significant improvement over the SYCL 1.2.1 memory model!
- Aligned well with C++20:
  - Reuses many terms and syntax: **`atomic_ref`** and **`atomic_fence`**
- Aligned well with OpenCL 2.0:
  - Includes memory scopes and memory regions for performance
  - Key difference: single happens-before relation for all memory regions!

sycl::atomic_ref

```
template <typename T,
          memory_order DefaultOrder,
          memory_scope DefaultScope,
          address_space Space = address_space::generic_space>
struct atomic_ref;
```

- `sycl::atomic_ref` is aligned with C++20, with additions:
  - Order and scope follow OpenCL, and can be used to improve performance
  - Address space associated with the referenced object
  - Default order defined per `sycl::atomic_ref`, and can be overridden per operation
- No support for synchronization (i.e. `wait`, `notify_one`, `notify_all`)

Device and Context Queries in SYCL 2020

- SYCL is targeting diverse architectures:
  - e.g. Some devices may not support sequential consistency
- Combinations of devices may have different capabilities:
  - e.g. Sequential consistency across context with multiple architectures!?
- SYCL 2020 adds queries to devices *and* contexts:
  - `info::context::atomic_memory_order_capabilities`, `info::context::atomic_memory_scope_capabilities`
  - `info::device::atomic_memory_order_capabilities`, `info::device::atomic_memory_scope_capabilities`

What does the future hold?

Problems to solve in SYCL-Next?

Similarities and Differences: C++ vs SYCL

- The default behavior of SYCL is now very close to C++17

- SYCL provides different consistency guarantees at different scopes:
  - Work-item code always follows *sequenced-before* rules
  - Groups of work-items can always enforce consistency via group barriers
  - Enforcing consistency via atomics (e.g. across work-items) depends on device capabilities

- The visibility of SYCL allocations is also tied to scopes
  - SYCL defaults to work-item visibility; C++ defaults to system visibility
  - Should SYCL-Next change its defaults to align with C++?

Forward Progress

```
q.parallel_for(nd_range(N, L), [=](nd_item<1> it) {
  auto sg = item.get_sub_group();
  if (!sg.leader()) {
    // Spin until flag is set
    while (atomic_ref<int, memory_order::acquire, memory_scope::device>(flag) == 0) {};
  } else {
    // Set flag after performing some work
    ...
    atomic_ref<int, memory_order::release, memory_scope::device>(flag) = 1;
  }
});
```

- The above code is valid SYCL 2020... but isn't guaranteed to work!
- New language features required to help developers?
  - Querying forward progress guarantees: work-items, sub-groups, work-groups?
  - Declaring that a kernel requires (assumes) specific guarantees for correctness?

Summary and Call to Action

Summary and Call to Action

- SYCL 2020 has a significantly improved memory model
  - Syntactically aligned with modern C++
  - Incorporates scopes and address spaces to improve performance
  - Add queries to support diverse device capabilities
- There is still work to be done! We're looking for your help!
  - Formal modeling to prove correctness
  - Forward progress guarantees to assure correctness and to improve portability
  - Other features that you need for your code bases?
- Many parts of SYCL could benefit from research projects. Reach out!

Thank you!

Notices & Disclaimers

intel®