Message fences

Document #: D2535R0

Date: 2-2-2022

Audience: SG1

Reply-to: dlustig@nvidia.com, ogiroux@nvidia.com

Abstract

Message fences place evaluations on objects into *happens before* but not *strongly happens before*. Object fences are message fences that restrict their effects to specific objects.

Tony tables

Before	After

`x = 1;`
`atomic_thread_fence(memory_order_release);`
`a.store(1, memory_order_relaxed);`
	`x = 1;`
	`atomic_object_fence(memory_order_release, x);`
	`a.store(1, memory_order_relaxed);`

`while(a.load(memory_order_relaxed) != 1);`
`atomic_thread_fence(memory_order_acquire);`
`assert(x == 1);`
	`while(a.load(memory_order_relaxed) != 1);`
	`atomic_object_fence(memory_order_acquire, x);`
	`assert(x == 1);`

(slower)	(faster)

Revisions

R0

- This is the first revision

Motivation

C++ is the lingua franca of accelerated computing. Accelerated computing platforms often contain sophisticated memory hierarchies, with specialized features such as scratchpad memories, complex interconnection fabrics, and other hallmarks of distributed systems in spite of being programmed using a standard shared address space paradigm.

In many deployments, there are situations where communication needs to be established between two threads alone, with no other threads participating at any time in the past or future.

The canonical example is the very same message passing paradigm that is commonly used to explain memory models. Each message is strictly a point-to-point communication: no third-party thread will ever directly observe the contents of that message. As such, there is no need for transitivity (or in memory model fencing parlance, "cumulativity") for these messages.

On large distributed systems connected by complex fabrics, and/or on architectures that internally resemble distributed systems, transitivity can be expensive to enforce. Message fences introduce a non-transitive synchronization mechanism that suffices for message passing scenarios and that can be implemented on many systems more cheaply than traditional transitive fences and ordering mechanisms.

Message fences can be further optimized by restricting their applicability only to certain objects explicitly specified by the programmer. These object fences can take advantage of hardware storage-specific fencing mechanisms that may be cheaper to execute than standard synchronization applied across the entire memory space. Object fences provide a way to write code that remains portable even when using non-standard storage declarations such as CUDA's `__shared__`.

Existing Implementations

A number of existing programming models provide synchronization for message passing paradigms.

On many existing implementations with shared memory, transitivity is provided by default, and implementations will therefore pay the cost of enforcing transitivity even though it is not always needed when implementing a message passing paradigm. On larger MPI deployments [mpi] built on top of complex network fabrics, message passing can be implemented using point-to-point communication over the network, and transitivity with other prior communication will not be established.

OpenMP [openmp] provides a flush construct `#pragma omp flush [memory-order-clause] [(list)] new-line` which allows expert programmers to specify the specific list of objects being flushed. The `memory-order-clause` can be `release`, `acquire`, or `acq_rel`, but establishing `seq_cst` ordering is not an option. To be fair, the OpenMP manual also includes the following warning:

> Use of a flush construct with a list is extremely error prone and users are strongly discouraged from attempting it.

OpenCL [opencl] provides distinct *happens before* ordering relationships for local memory (a private per-work-group memory space) and for global memory (visible to all work-groups as well as to the host). If both are viewed as part of a larger flat address space, then synchronization on local memory can be seen as a form of message or object fence that establishes transitive ordering among objects in local memory but with no connection to any ordering established among objects in global memory.

A conservative implementation of object fences simply discards the object information and becomes a message fence:

```
template<class... Objects>
void atomic_object_fence(memory_order order, Objects const&...) noexcept
{
    atomic_message_fence(order);
}
```

A conservative implementation of message fences simply falls back on standard thread fences:

```
template<>
void atomic_message_fence(memory_order order) noexcept
{
    atomic_thread_fence(order);
}
```

Discussion

These rules suffice for establishing point-to-point communication as was motivated earlier:

1. Message fences are strictly non-transitive: they intentionally "forget history" when establishing ordering relationships in order to make the synchronization cheaper.
2. Object fences further evaluations on objects not named, thereby creating a form of "per-object happens before" relationship, in order to make the synchronization cheaper still.

However, they do not provide a means of "re-integrating" the synchronized objects into the transitive ordering established by *strongly happens before*. The memory model would need to become dramatically more complex to capture such relationships, and so instead we simply disallow it. Ordinary fences and atomic operations continue to place evaluations on these objects into *strongly happens before* as normal.

Because message and object fences make no attempt to establish transitive ordering, they also make no attempt to restore sequential consistency. By their nature, no total order is being established. Message fences executing concurrently may operate in an completely unsynchronized manner on optimized implementations; e.g., operating on two different scratchpad memories. As such, because it would have no real meaning, `memory_order_seq_cst` is also simply disallowed on message fences.

For simple objects, many implementations will have sufficient knowledge about the storage backing the object that they will be able to emit an optimized mapping. For example, an object fence targeting an object in GPU scratchpad memory will be able to emit a fence specifically targeting scratchpad memory. However, if an object fence specifies more complex objects and/or multiple objects backed by multiple different storage locations, then the implementation may have no choice but to take the conservative implementation path.

A more conservative version of this proposal could omit message fences, and instead provide only object fences.

Proposed Wording

This wording is relative to N4901.

31.11 Fences [atomics.fences]

1. This subclause introduces synchronization primitives called fences. **An invocation of an `atomic_thread_fence()` is a *thread fence*. An invocation of an `atomic_message_fence()` is a *message fence*. An invocation of an `atomic_object_fence()` is both a *message fence* and an *object fence*.**
2. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*. A fence with release semantics is called a *release fence*.
3. A *release* ***thread*** *fence* A synchronizes with an *acquire* ***thread*** *fence* B if there exist atomic operations X and Y , both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.
4. A *release* ***thread*** *fence* A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation
5. An atomic operation A that is a release operation on an atomic object M synchronizes with an *acquire* ***thread*** *fence* B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.
6. **An evalution A on object O happens before another evaluation B on the same object if there exist fences X and Y such that A is sequenced before X, X is invoked with O if it is an object fence, Y is invoked with O if it is an object fence, Y is sequenced before B, and X would synchronize with Y if they were both thread fences.**

```
extern "C" void atomic_thread_fence(memory_order order) noexcept;

extern "C" void atomic_message_fence(memory_order order) noexcept;

template<class... Objects>
void atomic_object_fence(memory_order order, Objects const&... objects) noexcept;
```

7. ***Preconditions*: if it is a *message fence*, `order != memory_order::seq_cst`.**
8. *Effects*: Depending on the value of `order`, this operation:

   (8.1) — has no effects, if `order == memory_order::relaxed`;

   (8.2) — is an acquire fence, if `order == memory_order::acquire` or `order == memory_order::consume`;

   (8.3) — is a release fence, if `order == memory_order::release`;

   (8.4) — is both an acquire fence and a release fence, if `order == memory_order::acq_rel`;

   (8.5) — is a sequentially consistent acquire and release fence, if `order == memory_order::seq_cst`.

References

[mpi] MPI, a Message Passing Interface Standard, version 3.1 https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[opencl] OpenCL, the OpenCL Specification, v3.0.10 https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

[openmp] OpenMP, *flush construct* https://www.openmp.org/spec-html/5.0/openmpsu96.html#x127-4920002.17.8