The **SPIR-V** memory and execution model

Covering SPIR-V for OpenCL 1.2, 2.0 and 2.1

Version: 1.0

Document Revision: 5

Khronos SPIR Working Group
Lee Howes

Acknowledgements

The SPIR-V specification is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

John Kessenich, LunarG
Lee Howes, QUALCOMM
Robert J. Simpson, QUALCOMM

1. Glossary

Acquire semantics (Acquire Memory Semantics): One of the memory order semantics defined for synchronization operations. Acquire semantics apply to atomic operations that load from memory. Given two units of (see Unit Of Execution), **A** and **B**, acting on a shared atomic object **M**, if **A** uses an atomic load of **M** with acquire semantics to synchronize-with an atomic store to **M** by **B** that used release semantics, then **A**'s atomic load will occur *before* the effects of any *subsequent* operations by **A** are visible to **B**. Memory orders *release*, *sequentially consistent*, and *acquire_release* all include *release semantics* and effectively pair with a load using acquire semantics.

Acquire-release semantics (both **Acquire** and **Release** Memory Semantics): A memory order semantic for synchronization operations (such as atomic operations) that has the properties of both acquire and release memory orders. It is used with read-modify-write operations.

Atomic operations: Operations that at any point, and from any perspective, have either occurred completely, or not at all. Memory orders associated with atomic operations may constrain the visibility of loads and stores with respect to the atomic operations (see *relaxed semantics*, *acquire semantics*, *release semantics*, *acquire release semantics* or *sequentially consistent semantics*).

Barrier: A barrier is a mechanism for blocking an entity or a set of entities There are three types of *barriers* – a command-queue barrier, a work-group barrier and a sub-group barrier.

- Mechanisms in some host API to synchronize between SPIR-V Program Instances.
- ControlBarrier instructions that synchronize the execution sets of SPIR-V Invocations.
- MemoryBarrier instructions that synchronize a SPIR-V invocation with the flow of data through the memory system.

Compute Device Memory: This refers to one or more memories attached to the device executing the SPIR-V Program Instance.

Concurrency: A property of a system in which a set of tasks in a system can remain active and make progress at the same time. To utilize concurrent execution when running a program, a programmer must identify the concurrency in their problem, expose it within the source code, and then exploit it using a notation that supports concurrency.

Constant Memory (UniformConstant storage class**)**: A region of *global memory* whose content remains constant during the execution of a SPIR-V Program. The *host* allocates and initializes memory objects placed into *constant memory*.

Control Barrier: A control ordering operation. An OpControlBarrier instruction. A control barrier blocks some set of invocations from progressing until all of them reach the same point in the execution.

Control flow: The flow of instructions executed by a work-item. Multiple logically related work items may or may not execute the same control flow. The control flow is said to be *converged* if all the Invocations in the set execution the same stream of instructions.   In a *diverged* control flow, the work-items in the set execute different instructions. At a later point, if a diverged control flow becomes converged, it is said to be a re-converged control flow.

Converged control flow: see **control flow**.

Data race: The execution of a SPIR-V Program Instance contains a data race if it contains two actions in different Units of Execution where (1) one action modifies a memory location and the other action reads or modifies the same memory location, and (2) at least one of these actions is not atomic, or the corresponding memory scopes are not inclusive, and (3) the actions are global actions unordered by the global-happens-before relation or are local actions unordered by the local-happens before relation.

Diverged control flow: see *control flow*.

Generic address space: An address space that include the *private*, *local*, and *global* address spaces available to a device. The generic address space supports conversion of pointers to and from private, local and global address spaces, and hence lets a programmer write a single function that at compile time can take arguments from any of the three named address spaces.

Global Happens before: see *happens before*.

Global Memory: A memory region accessible to all *work-items* executing in a *context*. It is accessible to the *host* using *commands* such as read, write and map. *Global memory* is included within the *generic address space* that includes the private and local address spaces.

Happens before: An ordering relationship between operations that execute on multiple units of execution. If an operation A happens-before operation B then A must occur before B; in particular, any value written by A will be visible to B.We define two separate happens-before relations: *global-happens-before* and *local-happens-before*. These are defined in section 3.6.

Image Object: A *memory object* that stores a two- or three- dimensional structured array. Image data can only be accessed with read and write functions. The read functions use a *sampler*. OpTypeImage.

Invocation: A single running instance of a SPIR-V program: an OpenCL work-item or GLSL shader invocation.

The *image object* encapsulates the following information:

- Dimensions of the image.
- Description of each element in the image.
- Properties that describe usage information and which region to allocate from.
- Image data.

The elements of an image are selected from a list of predefined image formats. "Image Format" in the SPIR-V specification.

Local Memory (WorkgroupLocal storage class): A memory region associated with a *work-group* and accessible only by *work-items* in that *work-group*. *Local memory* is included within the *generic address space* that includes the private and global address spaces.

Memory Barrier: A memory ordering operation without an associated atomic object. An OpMemoryBarrier instruction. A memory barrier can use the *acquire semantics*, *release semantics*, *acquire release semantics* or *sequentially consistent semantics*.

Memory Consistency Model: Rules that define which values are observed when multiple units of execution load data from any shared memory plus the synchronization operations that constrain the order of memory operations and define synchronization relationships. The memory consistency model in SPIR-V is based on the memory model from the ISO C11 programming language.

Memory Regions (or Pools): A distinct address space. *Memory regions* may overlap in physical memory though the API may treat them as logically distinct. Logically distinct *memory regions* are denoted as *private*, *local*, *constant,* and *global*.

Memory Scopes: These memory scopes define a hierarchy of visibilities when analyzing the ordering constraints of memory operations. They are defined by the values of the memory_scope enumeration constant. Current values are **Invocation** (memory constraints only apply to a single SPIR-V Invocation and in practice apply only to image operations)**, Subgroup** (memory-ordering constraints only apply to Invocations executing in a sub-group), **Workgroup** (memory-ordering constraints only apply to Invocations executing in a work-group), **Device** (memory-ordering constraints only apply to Invocations executing on a single device) and **CrossDevice (**memory-ordering constraints only apply to Invocations and other Units of Execution executing across multiple devices and when using shared virtual memory).

Modification Order: All modifications to a particular atomic object M occur in some particular **total order**, called the **modification order** of M. If A and B are modifications of an atomic object M, and A happens-before B, then A shall precede B in the modification order of M. Note that the modification order of an atomic object M is independent of whether M is in local or global memory.

Pipe: The *pipe* memory object conceptually is an ordered sequence of data items. A pipe has two endpoints: a write endpoint into which data items are inserted, and a read endpoint from which data items are removed. At any one time, only one kernel instance may write into a pipe, and only one kernel instance may read from a pipe. To support the producer consumer design pattern, one kernel instance connects to the write endpoint (the producer) while another kernel instance connects to the reading endpoint (the consumer).

Private Memory: A region of memory private to a *work-item*. Variables defined in one *work-item's private memory* are not visible to another *work-item*.

Program Instance: The collection of SPIR-V invocations comprising a single API dispatch of a running SPIR-V program.

Re-converged Control Flow: see *control flow*.

Remainder work-groups: When the work-groups associated with a kernel-instance are defined, the sizes of a work-group in each dimension may not evenly divide the size of the NDRange in the corresponding dimensions. The result is a collection of work-groups on the boundaries of the NDRange that are smaller than the base work-group size. These are known as *remainder work-groups*.

Relaxed Consistency: A memory consistency model in which the contents of memory visible to different *work-items* or *commands* may be different except at a *barrier* or other explicit synchronization points.

Relaxed Semantics (Relaxed memory semantics): A memory order semantic for atomic operations that implies no order constraints. The operation is *atomic* but it has no impact on the order of memory operations.

Scope inclusion: Two actions **A** and **B** are defined to have an inclusive scope if they have the same scope **P** such that: (1) if **P** is Subgroup, and **A** and **B** are executed by Invocations within the same sub-group, or (2) if **P** is Workgroup, and **A** and **B** are executed by Invocations within the same work-group, or (3) if **P** is Device, and **A** and **B** are executed by Invocations on the same device, or (4) if **P** is CrossDevice, if **A** and **B** are executed by Invocations or other Units of Execution on one or more devices that can share SVM memory with each other and the host process.

Sequenced before: A relation between evaluations executed by a single unit of execution. Sequenced-before is an asymmetric, transitive, pair-wise relation that induces a partial order between evaluations. Given any two evaluations A and B, if A is sequenced-before B, then the execution of A shall precede the execution of B.

Sequential consistency: Sequential consistency interleaves the steps executed by each unit of execution. Each access to a memory location sees the last assignment to that location in that interleaving.

Sequentially consistent semantics (SequentiallyConsistent Memory Semantic): One of the memory order semantics defined for synchronization operations. When using sequentially-consistent synchronization operations, the loads and stores within one unit of execution appear to execute in program order (i.e., the sequenced-before order), and loads and stores from different units of execution appear to be simply interleaved.

Shared Virtual Memory (SVM): An address space exposed to both the host and the devices executing SPIR-V Programs within a single execution environment. SVM causes addresses to be meaningful between the host and all of the devices within a context and therefore supports the use of pointer based data structures in OpenCL kernels. It logically extends a portion of the global memory into the host address space therefore giving work-items access to the host address space. There are three types of SVM in OpenCL **Coarse-Grained buffer SVM**: Sharing occurs at the granularity of regions of OpenCL buffer memory objects. **Fine-Grained buffer SVM**: Sharing occurs at the granularity of individual loads/stores into bytes within OpenCL buffer memory objects. **Fine-Grained system SVM**: Sharing occurs at the granularity of individual loads/stores into bytes occurring anywhere within the host memory.

SIMD: Single Instruction Multiple Data. A programming model where a SPIR-V Program Instance is executed concurrently on multiple *processing elements* each with its own data and a shared program counter. All *processing elements* execute a strictly identical set of instructions.

SPIR-V Program: A SPIR-V entry point the functions it calls and operations it performs as defined in the SPIR-V binary and that may be launched by one of the valid execution environments.

SPMD: Single Program Multiple Data. A programming model where a SPIR-V Program Instance is executed concurrently on multiple *processing elements* each with its own data and its own program counter. Hence, while all computational resources run the same *kernel* they maintain their own instruction counter and due to branches in a *kernel*, the actual sequence of instructions can be quite different across the set of *processing elements*.

Sub-group: Sub-groups are an implementation-dependent grouping of work-items within a work-group. The size and number of sub-groups is implementation-defined.

Sub-group Barrier. See *Barrier*.

SVM Buffer: A memory allocation enabled to work with Shared Virtual Memory (SVM). Depending on how the SVM buffer is created, it can be a coarse-grained or fine-grained SVM buffer. Optionally it may be wrapped by a Buffer Object. See *Shared Virtual Memory (SVM)*.

Synchronization: Synchronization refers to mechanisms that constrain the order of execution and the visibility of memory operations between two or more units of execution.

Synchronization operations: Operations that define memory order constraints in a program. They play a special role in controlling how memory operations in one unit of execution (such as work-items or, when using SVM a host thread) are made visible to another. Synchronization operations include *atomic operations* and *memory barriers*

Synchronization point: A synchronization point between a pair of commands (A and B) assures that results of command A happens-before command B is launched (i.e. enters the ready state) .

Synchronizes with: A relation between operations in two different units of execution that defines a memory order constraint in global memory (*global-synchronizes-with*) or local memory (*local-synchronizes-with*).

Unit of execution: a generic term for a process, OS managed thread running on the host (a host-thread), SPIR-V Invocation (see Invocation) or any other executable agent that advances the work associated with a program.

Work-group: A collection of related *work-items* that execute on a single *compute unit*.   The *work-items* in the group execute the same *kernel-instance* and share *local memory* and *work-group functions*.

Work-group Barrier. See *Barrier*.

Work-group Function: A function that carries out collective operations across all the work-items in a work-group. Available collective operations are a barrier, reduction, broadcast, prefix sum, and evaluation of a predicate.   A work-group function must occur within a *converged control flow*; i.e. all work-items in the work-group must encounter precisely the same work-group function.

2. The SPIR-V Execution Model

2.1 Invocation

A single execution path from a SPIR-V entry point is called an invocation. How invocations are produced is up to the runtime system and particular execution model in use as defined by the high-level runtime. Invocations may be identified by one of various builtin variables depending on the execution model in use.

Invocations may not be considered to be entirely independent. While control flow through the invocation is treated largely independently by the SPIR-V code, multiple invocations may be mapped to lanes of a SIMD thread, or to independent threads of execution that do not guarantee forward progress. As such both generator and consumer must be aware of these constraints before making assumptions about behavior.

2.2 Sub-groups and Work-Groups and NDRanges

SPIR-V supports a hierarchy of execution concepts around the core *Invocation*.
Sub-groups are collections of invocations, and work-groups collections of sub-groups.

The index space supported by SPIR-V is called an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three. The NDRange is decomposed into work-groups forming blocks that cover the Index space.

Work-groups are N-dimensional with the same number of dimensions as the surrounding NDRange. The Invocations of a given work-group execute concurrently on the device and may synchronize and communicate.

Each work-group is decomposed into sub-groups. Sub-groups are one-dimensional entities with an implementation-defined collection of invocations.

Not all execution models will support all levels of the execution hierarchy. All execution models support *Invocations*. The memory model is described in terms of this hierarchy.

2.3 Execution Model: Execution of kernel-instances

Given a work-pool fed by some host API, a device will pull groups of invocations from the work-pool and execute them on one or several compute units in any order; possibly clustering interleaving execution of work-groups from multiple commands. A conforming implementation may choose to serialize the invocations so a correct algorithm cannot assume that invocations will execute concurrently except where the exposure of sub-groups and work-groups provides limits guarantees. There is no safe and portable way to synchronize across the independent execution of invocations except within sub-groups and work-groups since once in the work-pool, they can execute in any order.

When sub-groups are exposed, the invocations within a single sub-group execute concurrently but not necessarily in parallel (i.e. they are not guaranteed to make independent forward progress). Therefore, only high-level synchronization constructs (e.g. sub-group functions such as barriers) that apply to all the work-items in a sub-group are well defined and included in SPIR-V. In the absence of synchronization operations, invocations within a sub-group may be serialized. In the presence of sub-group operations, invocations within a sub-group may be serialized before any given sub-group operations, between dynamically encountered pairs of sub-group operations and between a work-group operations and the end of the kernel.

The work-items within a single work-group execute concurrently but are only guaranteed to make independent progress in the presence of sub-groups and device support. In the absence of this capability, only high-level synchronization constructs (e.g. work-group functions such as control barriers) that apply to all the work-items in a work-group are well defined and included in SPIR-V for synchronization within the work-group.

When work-groups and sub-groups are both exposed, invocations execute concurrently within a given work-group. When divided into sub-groups, with appropriate device support the sub-groups within a work-group may make independent forward progress with respect to each other even in the absence of work-group barrier operations. In this situation, sub-groups are able to internally synchronize using barrier operations without synchronizing with each other and may perform operations that rely on runtime dependencies on operations other sub-groups perform.

In the absence of independent forward progress of constituent sub-groups, invocations within a work-group may be serialized before, after or between work-group or sub-group synchronization functions.

2.4 Execution Model: Synchronization

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution. Consider the following three domains of synchronization exposed by SPIR-V instructions or between SPIR-V Program Instances:

- Sub-group synchronization: Contraints on the order of execution for Invocations in a single sub-group
- Work-group synchronization: Constraints on the order of execution for Invocations in a single work-group
- Command synchronization: Constraints on the order of SPIR-V Program Instances launched for execution

Synchronization across all **Invocations** within a single **sub-group** is carried out using a group instruction with execution scope **Subgroup**. These functions carry out collective operations across all the Invocations in a sub-group. A sub-group function must occur within a converged control flow; i.e. all **Invocations** in the sub-group must encounter precisely the same sub-group function. For example, if a work-group function occurs within a loop, the Invocations must encounter the same sub-group function in the same loop iterations. All the work-items of a sub-group must execute the sub-group function and complete reads and writes to memory before any are allowed to continue execution beyond the sub-group function. Synchronization between sub-groups must either be performed using work-group functions, or through memory operations. Using memory operations for sub-group synchronization should be used carefully as forward progress of sub-groups relative to each other is only supported optionally by API implementations.

Synchronization across all **Invocations** within a single **work-group** is carried out using a group operations with execution scope **Workgroup**. These functions carry out collective operations across all the Invocations in a work-group. A work-group function must occur within a converged control flow; i.e. all Invocations in the work-group must encounter precisely the same work-group function. For example, if a work-group function occurs within a loop, the Invocations must encounter the same work-group function in the same loop iterations. All the Invocations in a single work-group must execute the work-group function and complete reads and writes to memory before any are allowed to continue execution beyond the work-group function. Work-group functions that apply between work-groups are not provided by SPIR-V since SPIR-V does not define forward-progress or ordering relations between work-groups, hence collective synchronization operations are not well defined.

Synchronization between entire SPIR-V Program Instances is defined in terms of distinct **synchronization points**. How these are managed is up to the execution environment. It is guaranteed that at any such point all work-items taking part in the same **Program Instance** and all their side effects will have completed execution.

The resulting *happens-before* relationships are a fundamental part of the SPIR-V memory model. When applied at the level of commands, they are straightforward to define at a language level in terms of ordering relationships between different commands. Ordering memory operations inside different commands, however, requires rules more complex than can be captured by the high level concept of a synchronization point.

3. The SPIR-V Memory Model

The SPIR-V memory model describes the structure, contents, and behavior of the memory accessible to a SPIR-V Program Instance. The model allows a programmer to reason about values in memory as the host program and multiple SPIR-V Invocations execute. This chapter describes the model in its most general form. Like the instruction set, different execution environments will subset the model to restrict the guarantees provided to SPIR-V code.

A SPIR-V program executes within some bounding environment exposing a set of units of execution. The host program runs as one or more host threads managed by the operating system running on the host. There may be multiple devices within the environment which all have access to a shared set of memory objects and which may also have access to host memory. On a single device, multiple work-groups may execute in parallel with potentially overlapping updates to memory. Finally, within a single work-group, multiple work-items concurrently execute, once again with potentially overlapping updates to memory.

The memory model must precisely define how the values in memory as seen from each of these units of execution interact so a programmer can reason about the correctness of SPIR-V programs. We define the memory model in four parts.

- Memory regions: The distinct memories visible to the host and the devices that share a context.
- Memory objects: The objects defined and managed by the execution environment.
- Shared Virtual Memory (SVM): A virtual address space exposed to both the host and the devices within a context.
- Consistency Model: Rules that define which values are observed when multiple units of execution load data from memory plus the atomic/fence operations that constrain the order of memory operations and define synchronization relationships.

3.1 Memory Model: Fundamental Memory Regions

Memory in OpenCL is divided into two parts.

- **Host Memory:** The memory directly available to the host. The detailed behavior of host memory is defined outside of the SPIR-V specification. Memory objects move between the Host and the devices through functions within the execution environment API or through a shared virtual memory interface.
- **Device Memory:** Memory directly available to SPIR-V units of execution.

Device memory consists of four named address spaces or *memory regions*:

- **Global Memory (WorkgroupGlobal, Uniform** Storage Classes): This memory region permits read/write access to all work-items in all work-groups running on any device within a context. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.
- **Constant Memory (UniformConstant** Storage Class): A region of global memory that remains constant during the execution of a kernel-instance. The host allocates and initializes memory objects placed into constant memory.
- **Local Memory (WorkgroupLocal** Storage Class): A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group.
- **Private Memory (PrivateGlobal, Function** Storage Class): A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.
- **Image Memory (Input**, **Output**, Image Storage Class): A region of memory contained within image objects. Orderings on image data are not involved in the happens-before orders of the memory model.

The **Generic** storage class may behave as *global*, *local* or *private* memory depending on the underlying value.

The memory regions and their relationship are summarized in Figure 1. Local and private memories are always associated with a particular device. The global, constant and image memories, however, are shared between all devices within a given execution environment.

The named address spaces available to a device are disjoint meaning they do not overlap. This is a logical relationship, however, and an implementation may choose to let these disjoint named address spaces share physical memory.

To reduce errors when code needs to call functions that take pointers from multiple address spaces, the *global*, *local*, and *private*, address spaces belong to a single *generic address space* if the appropriate operations are enabled by the execution environment.

The *constant* and *image* address spaces are disjoint from the generic address space.

The addresses of memory associated with memory objects in *global* memory may or may not be preserved between executing kernel or shader instances, between a device and the host, and between devices. In this regard global memory acts as a global pool of memory objects rather than an address space. On shared virtual memory allocations, where supported, this does not apply.

SVM causes addresses to be meaningful between the host and all of the devices within a context hence supporting the use of pointer based data structures. It logically extends a portion of the global memory into the host address space giving work-items access to the host address space. On platforms with hardware support for a shared address space between the host and one or more devices, SVM may also provide a more efficient way to share data between devices and the host. Details about SVM are presented in Figure 3.3.

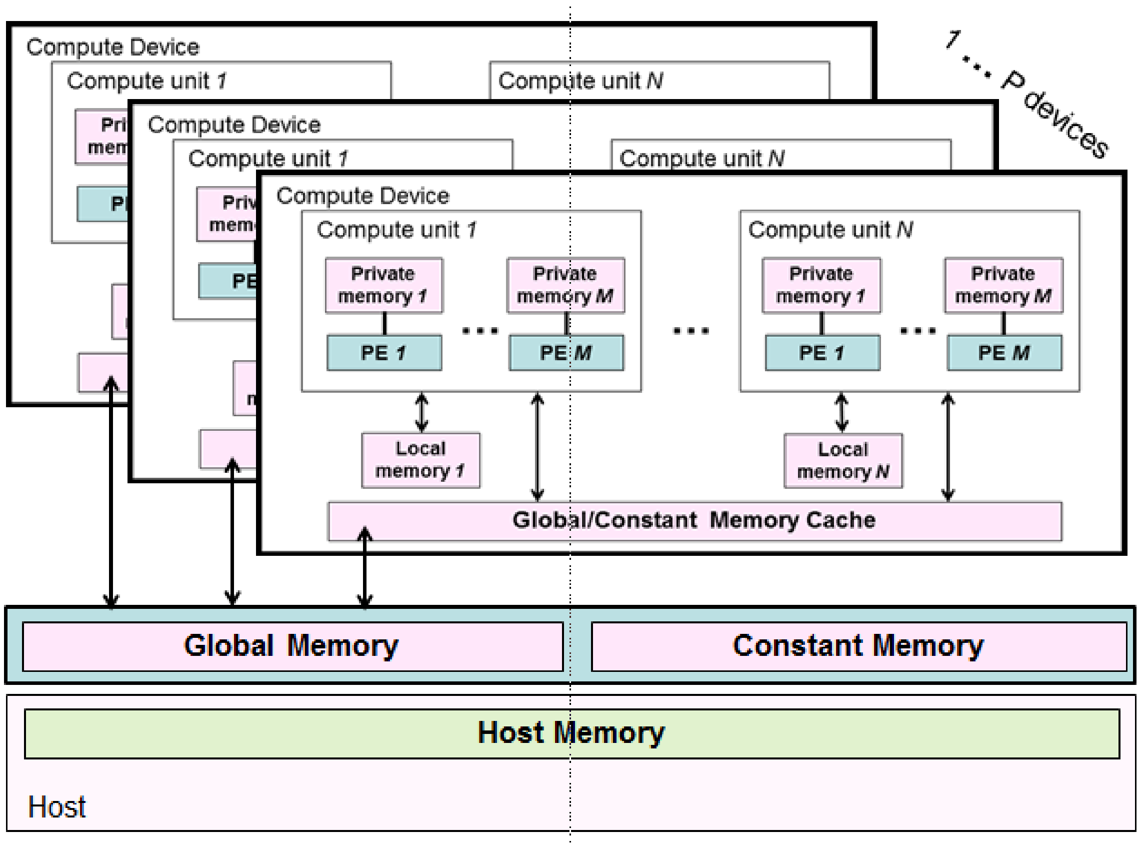

Figure 1 The named address spaces exposed in a SPIR-V environment. Global and Constant memories are shared between the one or more devices within the environment, while local and private memories are associated with a single device.

A programmer may use the features of the memory consistency model (section 3.4) to manage safe access to global memory from multiple work-items potentially running on one or more devices. In addition, when using shared virtual memory (SVM), the memory consistency model may also be used to ensure that host threads safely access memory locations in the shared memory region.

3.2 Memory Model: Memory Objects

The contents of global memory are *memory objects*. Memory objects fall into three distinct classes.

- **Buffer**: A memory object stored as a block of contiguous memory and used as a general purpose object to hold byte addressable data.
- **Image**: An image memory object holds one, two or three dimensional images. The formats are based on the standard image formats used in graphics applications. An image is accessible by specific image access instructions in SPIR-V.

- **Pipe**: The *pipe* memory object conceptually is an ordered sequence of data items. A pipe has two endpoints: a write endpoint into which data items are inserted, and a read endpoint from which data items are removed. Pipes have sized packets and are addressable by packet.

Data transfer into and out of memory objects from the host is defined by the rules of the execution model and its runtime. SPIR-V code may assume that data is available according to these rules combined with the SPIR-V memory model.

3.3 Memory Model: Shared Virtual Memory

Shared virtual memory (SVM) enables sharing of addresses across the host and SPIR-V modules. SVM only applies to buffer memory objects where pointer-based addressing is available. The SPIR-V memory model defines three types of shared virtual memory:

- **Coarse-Grained buffer SVM**: Sharing occurs at the granularity of regions of buffer memory objects. Consistency is enforced at explicit synchronization points defined by the execution environment's host runtime and are not under the control of SPIR-V instructions.

- **Fine-Grained SVM**: Sharing occurs at the granularity of individual loads/stores into bytes within memory objects or system memory allocations dependening on execution model rules. Loads and stores may be cached. Consistency is guaranteed at synchronization points defined by the execution environment's host API and are not under the control of SPIR-V instructions..

- **Fine-Grained SVM with atomics**: Sharing occurs at the granularity of individual loads/stores into bytes occurring anywhere within the host memory. Memory consistency is guaranteed across multiple devices by individual atomic operations within SPIR-V code.

Whether using any of the above forms of SVM or simple buffer access, available atomic operations withint SPIR-V code maintain consistent order within a single device.

3.4 Memory Model: Memory Consistency Model

The SPIR-V memory model tells programmers and compiler developrs what they can expect from a SPIR-V implementation; which memory operations are guaranteed to happen in which order and which memory values each read operation will return. The memory model tells compiler writers which restrictions they must follow when implementing compiler optimizations; which variables they can cache in registers and when they can move reads or writes around a barrier or atomic operation. The memory model also tells hardware designers about limitations on hardware optimizations; for example, when they must flush or invalidate hardware caches.

The memory consistency model in SPIR-V is based on the memory consistency rules defined by the C11 language. To help make the presentation more precise and self-contained, we include modified paragraphs taken verbatim from the ISO C11 international standard. When a paragraph is taken or modified from the C11 standard, it is identified as such along with its original location in the C11 standard.

3.5 Memory Model: Overview of atomic and memory barrier operations

The SPIR-V specification defines a number of *synchronization operations* that are used to define memory order constraints in a program. They play a special role in controlling how memory operations in one unit of execution (such as work-items or, when using SVM a host thread) are made visible to another. There are three types of synchronization operations in SPIR-V; *atomic operations*, *control barriers* and *memory barriers*. These are represented by the *OpAtomic**, *OpControlBarrier* and *OpMemoryBarrier* builtins.

Atomic operations operate on memory and are indivisible; that is, they either occur completely or not at all. Barrier operations do not directly on memory but take part in the memory ordering rules of the consistency model. All of these operations are used to order memory operations between units of execution and hence they are parameterized with *Scope* and *Memory Semantics* parameters that control the details of how the operations interact with the system's memory orders.

An atomic operation on one or more memory locations is an acquire operation, a release operation, a relaxed operation or both an acquire and release operation. An atomic operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. Atomic read-modify-write operations may perform multiple underlying operations atomically and thus may carry more than one of these semantics.

Defined in the *Memory Semantics* bitfield are the orders *Acquire* (used for reads), *Release* (used for writes), *Acquire-Release*, *Sequentially Consistent* and *Relaxed*. These are used for simple communication between units of execution using shared variables.

For instructions where it is relevant, the Storage Class bit field may also be provided. Each storage class maps in to the *Local*, *Global*, *Constant*, *Private* or *Image* memory region as described earlier. *Local* and *Global* refer to the happens-before relationships below. An operation that is parameterized by a memory semantic that maps into the *Local* memory region will operate on local memory and take part in the *local-happens-before* relation. An operation that is parameterized by a storage class that maps into the *Global* memory region will operate on global memory and take part in the *global-happens-before* relation. An operation parameterized by both *Local* and *Global* will take part in both relations, if supported by the execution environment.

Finally, defined in the *Scope* field are the scopes *CrossDevice*, *Device*, *WorkGroup*, *SubGroup*, and *Invocation*. These define the distance at which an *atomic*, *control barrier* or *memory barrier* influences memory orders, or how they influence execution synchronization for *control barriers*.

Scopes have meaning when applied to *local* memory or to *global* memory. When used on global memory visibility is bounded by the capabilities of that memory. When applied to *coarse-grained memory* (either coarse SVM or non-SVM allocations) or to *fine-grained* SVM, operations parameterized with *CrossDevice scope* will behave as if they were parameterized with *Device scope*. When used on local memory, visibility is bounded by the work-group and, as a result, scopes wider than *WorkGroup* are invalid.

Two actions **A** and **B** are defined to have an *inclusive scope* if they have the same *scope* **P** such that:

- **P** is *SubGroup* and **A** and **B** are executed by Invocations within the same sub-group.
- **P** is *WorkGroup* and **A** and **B** are executed by Invocationswithin the same work-group.
- **P** is *Device* and **A** and **B** are executed by Invocationson the same device.
- **P** is *CrossDevice* if **A** and **B** are executed by Invocations or other Units of Execution on one or more devices that can share SVM memory with each other and the host process.

3.6 Memory Model: Memory Ordering Rules

Fundamentally, the issue in a memory model is to understand the orderings in time of modifications to objects in memory. Modifying an object or calling a function that modifies an object are side effects, i.e. changes in the state of the execution environment. Evaluation of an expression in general includes both value computations and initiation of side effects. [C11 standard, Section 5.1.2.3, paragraph 2, modified]

We assume that the SPIR-V language and the host programming languages have a sequenced-before relation between the evaluations executed by a single unit of execution. This sequenced-before relation is an asymmetric, transitive, pair-wise relation between those evaluations, which induces a partial order among them. Given any two evaluations **A** and B, if **A** is sequenced-before **B**, then the execution of **A** shall precede the execution of **B**. (Conversely, if **A** is sequenced-before **B**, then **B** is sequenced-after **A**.) If **A** is not sequenced-before or sequenced-after **B**, then **A** and **B** are unsequenced. Evaluations **A** and **B** are indeterminately sequenced when **A** is either sequenced-before or sequenced-after **B**, but it is unspecified which. [C11 standard, Section 5.1.2.3, paragraph 3, modified]

NOTE: sequenced-before is a partial order of the operations executed by a single unit of execution (e.g. a host thread or work-item). It generally corresponds to the source program order of those operations, and is partial because of the undefined argument evaluation order of SPIR-V operations in some cases.

In the SPIR-V language, the value of an object visible to a work-item W at a particular point is the initial value of the object, a value stored in the object by W, or a value stored in the object by another work-item or host thread, according to the rules below. Depending on details of the host programming language, the value of an object visible to a host thread may also be the value stored in that object by another work-item or host thread. [C11 standard, Section 5.1.2.4, paragraph 2, modified]

Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. [C11 standard, Section 5.1.2.4, paragraph 4]

Let us define an atomic object **M** as a memory object that is touched at any point by an *OpAtomic** SPIR-V operation. All modifications to a particular atomic object **M** occur in some particular total order, called the modification order of **M**. If **A** and **B** are modifications of an atomic object **M**, and **A** happens-before **B**, then **A** shall precede **B** in the modification order of **M**, which is defined below. Note that the modification order of an atomic object **M** is independent of whether **M** is in local or global memory. [C11 standard, Section 5.1.2.4, paragraph 7, modified]

A release sequence begins with a release operation **A** on an atomic object **M** and is the maximal contiguous sub-sequence of side effects in the modification order of **M**, where the first operation is **A** and every subsequent operation either is performed by the same work-item or host thread that performed the release or is an atomic read-modify-write operation. [C11 standard, Section 5.1.2.4, paragraph 10, modified]

SPIR-V's local and global memories are disjoint. Kernels may access both kinds of memory while host threads may only access global memory. Furthermore, the *memory semantics* argument of the *OpControlBarrier* operation specifies which memory operations the function will make visible: these memory operations can be, for example, just the ones to local memory, or the ones to global memory, or both. Since the visibility of memory operations can be specified for local memory separately from global memory, we define two related but independent relations, *global-synchronizes-with* and *local-synchronizes-with*. Certain operations on global memory may global-synchronize-with other operations performed by another work-item or host thread. An example is a release atomic operation in one work- item that global-synchronizes-with an acquire atomic operation in a second work-item. Similarly, certain atomic operations on local objects in kernels can local-synchronize- with other atomic operations on those local objects. [C11 standard, Section 5.1.2.4, paragraph 11, modified]

We define two separate happens-before relations: global-happens-before and local-happens-before.

A global memory action **A** global-happens-before a global memory action **B** if

- **A** is sequenced before **B**, or
- **A** global-synchronizes-with **B**, or
- For some global memory action **C**, **A** global-happens-before **C** and **C** global-happens-before **B**.

A local memory action **A** local-happens-before a local memory action **B** if

- **A** is sequenced before **B**, or
- **A** local-synchronizes-with **B**, or
- For some local memory action **C**, **A** local-happens-before **C** and **C** local-happens-before **B**.

A SPIR-V execution environment shall ensure that no program execution demonstrates a cycle in either the “local-happens-before” relation or the “global-happens-before” relation.

NOTE: The global- and local-happens-before relations are critical to defining what values are read and when data races occur. The global-happens-before relation, for example, defines what global memory operations definitely happen before what other global memory operations. If an operation **A** global-happens-before operation **B** then **A** must occur before **B**; in particular, any write done by **A** will be visible to **B**. The local-happens-before relation has similar properties for local memory. Programmers can use the local- and global-happens-before relations to reason about the order of program actions.

A visible side effect **A** on a global object **M** with respect to a value computation **B** of **M** satisfies the conditions:

- **A** global-happens-before **B**, and
- there is no other side effect **X** to **M** such that **A** global-happens-before **X** and **X** global-happens-before **B**.

We define visible side effects for local objects **M** similarly. The value of a non-atomic scalar object **M**, as determined by evaluation **B**, shall be the value stored by the visible side effect **A**. [C11 standard, Section 5.1.2.4, paragraph 19, modified]

The execution of a program contains a data race if it contains two conflicting actions A and B in different units of execution, and

- (1) at least one of **A** or **B** is not atomic, or **A** and **B** do not have inclusive memory scope, and
- (2) the actions are global actions unordered by the global-happens-before relation or are local actions unordered by the local-happens-before relation.

Any such data race results in undefined behavior. [C11 standard, Section 5.1.2.4, paragraph 25, modified]

We also define the visible sequence of side effects on local and global atomic objects. The remaining paragraphs of this subsection define this sequence for a global atomic object **M**; the visible sequence of side effects for a local atomic object is defined similarly by using the local-happens-before relation.

The visible sequence of side effects on a global atomic object **M**, with respect to a value computation **B** of **M**, is a maximal contiguous sub-sequence of side effects in the modification order of **M**, where the first side effect is visible with respect to **B**, and for every side effect, it is not the case that **B** global-happens-before it. The value of **M**, as determined by evaluation **B**, shall be the value stored by some operation in the visible sequence of **M** with respect to **B**. [C11 standard, Section 5.1.2.4, paragraph 22, modified]

If an operation **A** that modifies an atomic object **M** global-happens before an operation **B** that modifies **M**, then **A** shall be earlier than **B** in the modification order of **M**. This requirement is known as write-write coherence.

If a value computation **A** of an atomic object **M** global-happens-before a value computation **B** of **M**, and **A** takes its value from a side effect **X** on **M**, then the value computed by **B** shall either equal the value stored by **X**, or be the value stored by a side effect **Y** on **M**, where **Y** follows **X** in the modification order of **M**. This requirement is known as read-read coherence. [C11 standard, Section 5.1.2.4, paragraph 22, modified]

If a value computation **A** of an atomic object **M** global-happens-before an operation **B** on **M**, then **A** shall take its value from a side effect **X** on **M**, where **X** precedes **B** in the modification order of **M**. This requirement is known as read-write coherence.

If a side effect **X** on an atomic object **M** global-happens-before a value computation **B** of **M**, then the evaluation **B** shall take its value from **X** or from a side effect **Y** that follows **X** in the modification order of **M**. This requirement is known as write-read coherence.

3.6.1 Memory Ordering Rules: Atomic Operations

This and following sections describe how different program actions in SPIR-V Programs and host code contribute to the local- and global-happens-before relations. This section discusses ordering rules for SPIR-V's atomic operations.

Given the *SequentiallyConsistent*, *Acquire*, *Release* and *Relaxed memory semantics* applied to *OpAtomic**, the following rules apply:

- For *Relaxed*, no operation orders memory.
- For *Release*, *AcquireRelease* and *SequentiallyConsistent*, a store operation performs a release operation on the affected memory location.
- For *Acquire*, *AcquireRelease* and *SequentiallyConsistent*, a load operation performs an acquire operation on the affected memory location. [C11 standard, Section 7.17.3, paragraphs 2-4, modified]

Certain built-in functions synchronize with other built-in functions performed by another unit of execution. This is true for pairs of release and acquire operations under specific circumstances. An atomic operation **A** that performs a release operation on a global object **M** global-synchronizes-with an atomic operation **B** that performs an acquire operation on **M** and reads a value written by any side effect in the release sequence headed by **A**. A similar rule holds for atomic operations on objects in local memory: an atomic operation **A** that performs a release operation on a local object **M** local-synchronizes-with an atomic operation **B** that performs an acquire operation on **M** and reads a value written by any side effect in the release sequence headed by **A**. [C11 standard, Section 5.1.2.4, paragraph 11, modified]

NOTE: Atomic operations specifying order *Relaxed* are relaxed only with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object. [C11 standard, Section 7.17.3, paragraph 8]

There shall exist a single total order **S** for all *sequentially consistent* operations that is consistent with the modification orders for all affected locations, as well as the appropriate global-happens-before and local-happens-before orders for those locations, such that each memory_order_seq operation **B** that loads a value from an atomic object **M** in global or local memory observes one of the following values:

- the result of the last modification **A** of **M** that precedes **B** in **S**, if it exists, or
- if **A** exists, the result of some modification of **M** in the visible sequence of side effects with respect to **B** that is not memory_order_seq_cst and that does not happen before **A**, or
- if **A** does not exist, the result of some modification of **M** in the visible sequence of side effects with respect to **B** that is not memory_order_seq_cst. [C11 standard, Section 7.17.3, paragraph 6, modified]

Let X and Y be two *sequentially consistent* operations. If X local-synchronizes-with or global-synchronizes-with Y then X both local-synchronizes-with Y and global-synchronizes-with Y.

If the total order **S** exists, the following rules hold:

- For an atomic operation **B** that reads the value of an atomic object **M**, if there is a *sequentially consistent* fence **X** sequenced-before **B**, then **B** observes either the last *sequentially consistent* modification of **M** preceding **X** in the total order **S** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 9]

- For atomic operations **A** and **B** on an atomic object **M**, where **A** modifies **M** and **B** takes its value, if there is a *sequentially consistent* fence **X** such that **A** is sequenced-before **X** and **B** follows **X** in **S**, then **B** observes either the effects of **A** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 10]

- For atomic operations **A** and **B** on an atomic object **M**, where **A** modifies **M** and **B** takes its value, if there are *sequentially consistent* fences **X** and **Y** such that **A** is sequenced-before **X**, **Y** is sequenced-before **B**, and **X** precedes **Y** in **S**, then **B** observes either the effects of **A** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 11]

- For atomic operations **A** and **B** on an atomic object **M**, if there are *sequentially consistent* fences **X** and **Y** such that **A** is sequenced-before **X**, **Y** is sequenced-before **B**, and **X** precedes **Y** in **S**, then **B** occurs later than **A** in the modification order of **M**.

NOTE: *sequentially consistent* order ensures sequential consistency only for a program that is (1) free of data races, and (2) exclusively uses *sequentially consistent* synchronization operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, *sequentially consistent* memory barriers ensure a total order only for the barriers themselves. Memory barriers cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications.

Atomic read-modify-write operations should always read the last value (in the modification order) stored before the write associated with the read-modify-write operation. [C11 standard, Section 7.17.3, paragraph 12]

<u>Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation.</u>

Note: Under the rules described above, and independent to the previously footnoted C++ issue, it is known that *x == y == 42* is a valid final state in the following problematic example:

```
global x = 0;
local y = 0;

unit_of_execution_1:
... [execution not reading or writing x or y, leading up to:]
int t = atomic_load(&y, acquire);
atomic_store(&x, t, release);

unit_of_execution_2:
... [execution not reading or writing x or y, leading up to:]
int t = atomic_load(&x, acquire);
atomic_store(&y, release);
```

This outcome is justified by a cycle that is "split" between the *local-happens-before* and *global-happens-before* relations.

This is not useful behavior and implementations should not exploit this phenomenon. It should be expected that in the future this may be disallowed by appropriate updates to the memory model description by the SPIR-V committee.

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time. [C11 standard, Section 7.17.3, paragraph 16]

If the SPIR-V code uses *CrossDevice* scope on atomic operations, and if the execution environment and device both support cross device atomics, then C11-compatible atomic operations in host code will synchronize at *CrossDevice* scope with SPIR-V atomic operations.

3.6.2 Memory Ordering Rules: MemoryBarrier Operations

This section describes how the SPIR-V fence operations contribute to the local- and global-happens-before relations.

Earlier, we introduced synchronization primitives called fences. Fences can utilize the *Acquire* memory order, the *Release* memory order, or both. A memory barrier with acquire semantics is called an acquire barrier; a fence with release semantics is called a release barrier.

A global release barrier **A** global-synchronizes-with a global acquire barrier **B** if there exist atomic operations **X** and **Y**, both operating on some global atomic object **M**, such that **A** is sequenced-before **X**, **X** modifies **M**, **Y** is sequenced-before **B**, **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, and that the scopes of **A**, **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 2, modified.]

A global release barrier **A** global-synchronizes-with an atomic operation **B** that performs an acquire operation on a global atomic object **M** if there exists an atomic operation **X** such that **A** is sequenced-before **X**, **X** modifies **M**, **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, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 3, modified.]

An atomic operation **A** that is a release operation on a global atomic object **M** global-synchronizes-with a global acquire barrier **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**, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 4, modified.]

A local release barrier **A** local-synchronizes-with a local acquire barrier **B** if there exist atomic operations **X** and **Y**, both operating on some local 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, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 2, modified.]

A local release barrier **A** local-synchronizes-with an atomic operation **B** that performs an acquire operation on a local 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, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 3, modified.]

An atomic operation **A** that is a release operation on a local atomic object **M** local-synchronizes-with a local acquire barrier **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**, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 4, modified.]

Let **X** and **Y** be two barriers that each have both the *LocalMemory* and *GlobalMemory memory semantics* flags specified. **X** global-synchronizes-with **Y** and **X** local synchronizes with **Y** if the conditions required for **X** to global-synchronize with **Y** are met, the conditions required for **X** to local-synchronize-with **Y** are met, or both sets of conditions are met.

3.6.3 Memory Ordering Rules: Group Operations

SPIR-V includes collective operations across sets of work-items within inclusive scope. These are called work-group functions and are either *OpControlBarrier* or one of the *OpGroup** operations when provided with the *WorkGroup* scope or *SubGroup* execution scope. We will first discuss the control barrier. The other functions are discussed afterwards.

The control barrier function provides a mechanism for a kernel to synchronize the work-items within a single scope instance defined by the *WorkGroup* or *SubGroup* execution scope parameter provided to the operation: informally, each work-item of the scope instance must execute the barrier before any are allowed to proceed. It also orders memory operations to a specified combination of one or more address spaces such as local memory or global memory, in a similar manner to a fence.

To precisely specify the memory ordering semantics for barrier, we need to distinguish between a dynamic and a static instance of the call to a barrier. A call to a barrier can appear in a loop, for example, and each execution of the same static barrier call results in a new dynamic instance of the barrier that will independently synchronize the scope instance's Invocations.

A work-item executing a dynamic instance of a barrier results in two operations, both fences, that are called the entry and exit fences. These fences obey all the rules for fences specified elsewhere in this chapter as well as the following:

- The entry fence is a release fence with the same flags and scope as requested for the barrier.
- The exit fence is an acquire fence with the same flags and scope as requested for the barrier.
- For each work-item the entry fence is sequenced before the exit fence.
- If the flags have a storage class in the *Global* memory region set then for each Invocation the entry fence global-synchronizes-with the exit fence of all other Invocations in the same scope instance.
- If the flags have a storage class in the *Local* memory region set then for each Invocation the entry fence local-synchronizes-with the exit fence of all other Invocations in the same scope instance.

The use of the *OpGroup** implies sequenced-before relationships between statements within the execution of a single work-item in order to satisfy data dependencies. For example, a work item that provides a value to an *OpGroup** builtin function must behave as if it generates that value before beginning execution of that builtin. Furthermore, the generator of the SPIR-V code must ensure that all work items in the scope instance must execute the same *OpGroup* builtin* call site, or dynamic *OpGroup* builtin* instance.

3.6.4 Memory Ordering Rules: Host-side and Device-side Commands

Any running SPIR-V Program Instance is managed by some host API. Under the rules of non-SVM allocations or of SVM allocations without support for *CrossDevice* atomic operations memory orderings are strictly governed by the rules of this host API. The specifics for any given API are a part of the specification for that execution environment. Here we describe the orderings as they relate to the SPIR-V kernel code.

Memory ordering rules in this section apply to all memory objects (buffers, images and pipes) as well as to SVM allocations where no earlier, and more fine-grained, rules apply.

In the remainder of this section, we assume that each command **C** enqueued onto a command-queue has an associated logically signaling entity **S** that signals its execution status, regardless of whether **S** was returned to the unit of execution that enqueued **C**. We also distinguish between the API function call that enqueues a command **C** and creates the synchronizing object **S**, the execution of **C**, and the completion of **C** (which marks the S as complete).

The ordering and synchronization rules for API commands are defined as following:

1. If an API function call **X** enqueues a command **C**, then **X** global-synchronizes-with **C**.
2. If **S** is a synchronizing entity upon which a command **C** waits, then **S** global-synchronizes-with **C**. In particular, if **C** waits on an **S** that is tracking the execution status of the command **C1**, then memory operations issued by **C1** will global-happen-before memory operations issued by **C**.
3. If a command **C** has a signally entity **S** that signals its completion, then **C** global-synchronizes-with **S**.
4. For a command **C** launched from the host, if **C** has a synchronizing entity **S** that signals its completion, then **S** global- synchronizes-with an API call **X** that waits on **S**.
5. The start of a SPIR-V instance **K** global-synchronizes-with all operations in the work items of **K**. Note that this includes the execution of any atomic operations by the work items in a program using fine-grain SVM.
6. All operations of all work items of a kernel-instance **K** global-synchronizes-with the signaling entity **S** signalling the completion of **K**. Note that this also includes the execution of any atomic operations by the work items in a program using fine-grain SVM.

3.7 Memory model: Volatile

Memory operations modified by the *volatile* memory access semantic must be interpreted strictly according to the rules of the underlying abstract machine. The effect of this is that:

1. All volatile memory operations must be passed through by the compiler without combination or duplication.
2. Volatile memory operations either to a single memory location or to multiple memory locations must not be re-ordered.

Volatile has no effect on atomicity, or on relative ordering between volatile and non-volatile memory operations. Volatile may be combined with atomic modifiers. The combination of volatile and non-volatile memory operations to the same location is undefined.