**Introduction**

The goal of this article is to provide a short introduction to the new OpenCL 2.0 atomics functionality and to discuss some caveats in the atomics usage and applicability to various GPU programming tasks.

**Atomic functions**

Atomic operations are well-known and important parallel programming primitives. They are main building blocks for synchronization primitives and lock-free algorithms. Atomics guarantee no interference between threads during single memory location modification.

OpenCL 1.2 implements atomic operations using built-ins that operate on regular integer data types and that can also be mixed with regular operators.

See the following simple example that illustrates the OpenCL 1.2 syntax:

```c
__global uint *counter;
*counter = 0;     //initialize variable with zero
uint old_val = atomic_inc( counter );  //make atomic increment on it
```

While the code, utilizing OpenCL 1.2 atomics, compiles and works just fine on 2.0 devices, the OpenCL 2.0 specification brings a whole bunch of changes and new functionality for atomic functions. Those additions mainly aim to achieve compatibility with C++11 standard, but some specificity exists. Let’s discuss what it means for us.

**OpenCL 2.0 Atomics Overview**

Unlike the OpenCL 1.2 specification the OpenCL 2.0 spec completely separates atomics from other language constructs. Memory affected by atomic operations is treated separately, so if you want to use OpenCL 2.0 atomics, you have to use variables of dedicated data type - `atomic_int`, `atomic_float`, `atomic_ptrdiff_t`, and so on. You cannot use regular OpenCL operators (=,+,-,>,< and other) on such data types. Instead, all operations on the atomics have to be performed explicitly with appropriate built-ins:

- atomic variables have to be initialized using the `atomic_init()` function. Global atomic variables of a program scope (this scope is also introduced in OpenCL 2.0 specification) should be initialized with `ATOMIC_VAR_INIT()` macro.
- to use variable value in regular calculations, it should be loaded from variable using `atomic_load()` function
- to assign a constant or regular variable value to atomic variable, `atomic_store()` should be used
- arithmetic, logical, min/max operations have to be performed using the `atomic_fetch_...()` function family
comparison and exchange operations should use the \texttt{atomic\_exchange()} and \texttt{atomic\_compare\_exchange\_...()} functions

there is no \texttt{atomic\_bool} type, but Boolean-like \texttt{atomic\_flag} type exists, which can be used by two corresponding functions: \texttt{atomic\_flag\_clear()} and \texttt{atomic\_flag\_test\_and\_set()}

With all those changes, the example code above renders into the following:

\begin{verbatim}
__global atomic\_uint *counter;
atomic\_init(counter, 0); //initialize variable with zero

//make atomic increment on it using settings equivalent to OpenCL 1.2 code
uint old_val = atomic\_fetch\_add\_explicit(counter, 1, memory\_order\_relaxed, memory\_scope\_device);
\end{verbatim}

What is behind this new explicit syntax? The main functional difference between OpenCL 1.2 and OpenCL 2.0 standards is that you can now control atomic operation memory synchronization ordering and scope. All built-ins mentioned above have explicit (built-in names have “\_explicit” suffix) and regular flavors (built-in names without the suffix). Explicit functions have arguments that let you specify \texttt{memory\_order} and \texttt{memory\_scope}, while regular ones just use default order mode and scope.

This addition makes new OpenCL 2.0 atomics syntaxically and functionally identical to the CPU atomics that are produced by C++ 11–compliant compilers.

New atomic arguments and their default values will be briefly discussed below. To get detailed description of the new atomic features and syntax, refer to section 6.13.11 of the OpenCL 2.0 C Specification (https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf).

Of course, since both memory ordering and memory scope are closely interrelated, it’s hard to discuss them separately, but we have to choose a starting point anyway. So let’s start from memory ordering, and discussion on memory scope will follow.

\section*{Memory Ordering}

While atomic memory is guaranteed to have special treatment, often might be a need for additional synchronization on regular, non-atomic memory during atomic operations. For example, if a set of work-items implements a sort of producer-consumer scenario, the work-items need not just a communication logic, which can be implemented by atomic operation, but consumers need to be sure that memory objects prepared by producers are guaranteed to be in valid state at a given moment.

First new atomic function parameter determines when and how such an additional synchronization is performed. Memory order is an enumerated type that enables you to specify one of the following modes:
- **memory_order_relaxed** – generally, provides best performance, since it doesn’t introduce any additional memory synchronization, only atomicity is guaranteed. Global atomic counter and image histogram calculation are examples of this ordering type usage.

- **memory_order_acquire** – acquire memory fence is inserted right before atomic operation, so all write results of other work-items within operation scope become visible to current work-item before atomic operation starts.

- **memory_order_release** – release fence is inserted right after atomic operation, so write results of the current work-item immediately become visible to others once atomic operation finishes.

- **memory_order_acq_rel** – both acquire and release fences are inserted.

- **memory_order_seq_cst** – same as the previous one, but in addition to fences, all atomic accesses of this mode together with their synchronization steps are serialized into single global sequence within a given scope.

The resulting table that represents additional synchronization barriers inserted depending on the ordering mode specified, looks like this:

<table>
<thead>
<tr>
<th>Memory order</th>
<th>Fence before</th>
<th>Fence after</th>
<th>Serialized access</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_order_relaxed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memory_order_acquire</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memory_order_release</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>memory_order_acq_rel</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>memory_order_seq_cst</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

For acquire-release synchronization there is no single global ordering of operations. While memory fences order memory operations for a single variable, accesses to different variables in different work-items are still parallel and thus globally unordered. Because of this, different work-items can have different views to the same variable. This scheme is well suited for “one consumer-multiple producers” or “one producer-multiple consumers” scenarios. The following scheme depicts the situation:

Sequential consistency makes all accesses globally ordered. Of course, this introduces additional overhead, but guarantees that all work-items have the same view to all variables at any moment of time. Sequential ordering may be necessary for “multiple producer-multiple consumer” situations where all consumers must observe the actions of all producers occurring in the same order. The picture below illustrates this:
The default OpenCL 2.0 ordering mode is `memory_order_seq_cst`. It is important to keep this in mind if you decide to port your existing code from OpenCL 1.2 to OpenCL 2.0 standard. Because 1.2 doesn’t provide any additional memory synchronization ability, most lightweight `memory_order_relaxed` might seem to be a natural candidate for backwards compatible default mode, ensuring maximal performance. But OpenCL 2.0 is aligned with C++ 11 standard, which might somewhat contradict the OpenCL 1.2 programmer expectations.

Short recommendation on choosing the appropriate memory ordering mode for a given application is the following – use default `memory_order_seq_cst` to be sure that maximal C++ 11 compliant memory consistency is achieved, but if you need just atomicity and maximal performance – use `memory_order_relaxed`.

For additional explanation on memory ordering, please refer to sections 3.3.4 and 3.3.5 of the OpenCL 2.0 Specification ([https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf](https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf)).

### Memory Scope

Another OpenCL 2.0 atomics parameter that could be specified is memory scope of operation. It determines a set of work-items that are affected by atomicity and memory ordering constraints for a given operation. Memory scope enumerated type can specify the following scopes:

- `memory_scope_work_item`
- `memory_scope_work_group`
- `memory_scope_device`
- `memory_scope_all_svm_devices`

Two cases you will mainly deal with are `memory_scope_work_group` and `memory_scope_device`. The first one guarantees atomicity and result consistency only between work-items of a single work-group, while the second one affects all simultaneously running work-items on the device.

One implication is that, because of local memory nature, only `memory_scope_work_group` is relevant for atomic variables placed in the local memory, and thus scope argument is just ignored for them.
Compared to the device scope, the work-group scope generally provides more optimization opportunities for the compiler and the OpenCL runtime. So you can expect significant performance benefits from it, especially when using it together with `memory_order_relaxed`.

The default OpenCL 2.0 memory scope is `memory_scope_device`, since both OpenCL 1.2 standard and C++ 11 expectations coincide here.

The `memory_scope_work_item` has very limited usage. It is used together with the new OpenCL 2.0 read-write images feature only, and provides synchronization ability between write and read image calls for one work-item. For inserting synchronization fences into image pipeline, the `atomic_work_item_fence()` function is used.

The last mode, `memory_scope_all_svm_devices`, is used mainly for lightweight synchronization between devices that share common virtual address space, and thus should be discussed separately.

References

3.3.4 Memory Model: Memory Consistency Model (in Khronos OpenCL 2.0 API Specification)

3.3.5 Memory Model: Overview of atomic and fence operations (in Khronos OpenCL 2.0 API Specification)

6.13.11 Atomic Functions (in Khronos OpenCL 2.0 C Programming Language Specification)

OpenCL™ 2.0 Shared Virtual Memory Overview