OpenCL™ 2.0 Shared Virtual Memory Overview

Contents

Introduction 3
SVM Features and Types 3
    Detecting the Supported SVM Type 5
Overview of SVM Features 6
    Identifying an SVM Buffer Using a Regular Pointer 6
    Shared Virtual Address Space 8
    “Map-free” Access 10
    Fine-Grained Coherent Access 11
    Fine-Grained Synchronization 12
    Sharing the Entire Host Address Space 14
    Implicit Use of Any SVM Allocation 15
Conclusion and Key Takeaways 17
References 17

Introduction

One of the remarkable features of OpenCL™ 2.0 is *shared virtual memory (SVM)*. This feature enables OpenCL developers to write code with extensive use of pointer-linked data structures like linked lists or trees that are shared between the host and a device side of an OpenCL application. In OpenCL 1.2, the specification doesn't provide any guarantees that a pointer assigned on *the host side* can be used to access data in the kernel on *the device side* or vice versa. Thus, data with pointers in OpenCL 1.2 cannot be shared between the sides, and the application should be designed accordingly, for example, with indices used instead of pointers. This is an artifact of a separation of address spaces of the host and the device that is addressed by OpenCL 2.0 SVM.

OpenCL 2.0 SVM enables the host and device portions of an OpenCL application to seamlessly share pointers and complex pointer-containing data-structures. Moreover, as described in this article, SVM is more than just about shared address space. It also defines memory model consistency guarantees for SVM allocations. This enables the host and the kernel sides to interact with each other using atomics for synchronization, like two distinct cores in a CPU. This is an important addition to OpenCL 2.0's shared address space support and is targeted to fulfill the needs of developers who need tighter synchronization between the host and the device beyond enqueuing commands onto an OpenCL queue and synchronizing through events.

Note that efficient implementation of all OpenCL 2.0 SVM features requires dedicated hardware coherency support such as enabled in the new Intel® Core^{TM} M processor family and future generations of Intel Core Processors with Intel® Graphics Gen8 compute architecture. See the Compute Architecture of Intel Processor Graphics Gen8 [PDF] article for more information. Not all OpenCL platforms support all SVM features defined by the OpenCL 2.0 specification, so the SVM features are organized in different *feature classes* of SVM support. The OpenCL 2.0 specification defines a minimum level of SVM support that is required for all OpenCL 2.0 implementations while other features are marked as optional. The host application should query the OpenCL implementation to determine which level of SVM is supported and route to the specific application code path that uses that level.

This article describes all required and optional features provided by the Khronos specification without focusing on any particular OpenCL platform.

SVM Features and Types

In its purest form, SVM enables CPU and GPU code to share a pointer rich data-structure by simply passing a single root pointer. However, OpenCL 2.0 shared virtual memory includes a number of features to enable varying degrees of hardware support and application control. The following list contains SVM *features* that can be considered separately. Each of them may have a self-contained goodness while being used in an application, though the features are not completely independent. Each feature will be described in more detail in later sections of this article.

- **Shared virtual address space** between the host and a kernel on a device allows sharing pointer-based data structures between the host and the device.
- **Identifying an SVM buffer using a regular pointer** without having to create a separate `cl_mem` object via the `clCreateBuffer` function. This helps to integrate OpenCL into a legacy C/C++ program and to easily manage OpenCL memory resources on the host.
- **"Map-free" access** to SVM allocations on the host side simplifies OpenCL host programming by eliminating the necessity to use map/unmap commands.
- **Fine-grained coherent access** to an SVM allocation from the host during accessing the same SVM allocation from the kernel on the device side in the same time. This allows the host and the device kernel to concurrently make modifications to adjacent bytes of a single SVM allocation.

- **Fine-grained synchronization:** Concurrent modification of the same bytes from the host and from the kernel on the device using atomics enables light-weight synchronization and memory consistency between the host and the device without enqueueing new commands in an OpenCL command queue.

- **Implicit use of any SVM allocation:** Pointers in one SVM allocation can point to other SVM allocations. Minimum level of SVM support requires that such indirectly referenced allocations should be bound to a kernel's execution context or need to be explicitly passed as kernel parameters. One of the advanced SVM features allows not passing all such indirectly used SVM allocations to kernels and using any number of them implicitly.

- **Sharing the entire host address space** provided by an operating system seamlessly, without creating an SVM buffer for it.

The OpenCL 2.0 specification classifies these features into three levels of SVM support that are called *SVM types*. Each SVM type provides a sub-set of the features listed above. The levels are differentiated by two important characteristics:

- **Buffer allocation vs. System allocation**. How SVM allocation is done: allocation by an operating system function (like the `malloc` function, operator new, or another function), or explicit creation of an SVM buffer with an OpenCL API function (`clSVMAlloc`).

- **Coarse-grained vs. Fine-grained**. What granularity of access is supported for sharing: as individual memory locations or as whole regions of memory buffers.

Characteristics above are combined into three types of SVM:

1. **Coarse-Grained buffer SVM**: Sharing occurs at the granularity of regions of OpenCL *buffer* memory objects. Cross-device atomics are not supported.

2. **Fine-Grained buffer SVM**: Sharing occurs at the granularity of *individual loads and stores* within OpenCL *buffer* memory objects. Cross-device atomics are optional.

3. **Fine-Grained system SVM**: Sharing occurs at the granularity of *individual loads/stores* occurring *anywhere within the host memory*. Cross-device atomics are optional.

	Coarse-grained	**Fine-grained**
Buffer allocation	1. Coarse-grained buffer SVM (no SVM atomics)	2. Fine-grained buffer SVM (optional SVM atomics)
System allocation	*Not applicable*	3. Fine-grained system SVM (optional SVM atomics)

The higher the level of SVM is, the more features it provides, and it may also require dedicated support from hardware, operating system, or device driver. So developers shouldn't expect that the highest level of SVM is supported on all devices and all OpenCL platforms. In fact, while coarse-grained buffer SVM is required to be implemented on all OpenCL 2.0 platforms, the other levels are optional.

Cross-device atomics or *SVM atomics* are atomic functions and fence operations that can be applied to coordinate concurrent access to memory locations in SVM allocations by the host and kernels. Support for atomics is optional for both fine-grained types. SVM atomics are not supported in coarse-grained type of SVM.

To more clearly describe which SVM type has support for a specific SVM feature, the following table maps the SVM features to SVM types.

SVM Feature	**SVM Type**				
	Coarse-grained buffer	Fine-grained buffer		Fine-grained system	
		w/o atomics	with atomics	w/o atomics	with atomics
Shared virtual address space	✓	✓	✓	✓	✓
Identifying an SVM buffer using a regular pointer	✓	✓	✓	✓	✓
“Map-free” access		✓	✓	✓	✓
Fine-grained coherent access		✓	✓	✓	✓
Fine-grained synchronization			✓		✓
Implicit use of any SVM allocation				✓	✓
Sharing the entire host address space				✓	✓

Detecting the Supported SVM Type

SVM availability and its highest supported type for a given device ID is queried with the `clGetDeviceInfo` OpenCL 2.0 API function passing `CL_DEVICE_SVM_CAPABILITIES` constant. The level of SVM support is returned through a pointer to a variable of type `cl_device_svm_capabilities`.

```
cl_device_svm_capabilities caps;

cl_int err = clGetDeviceInfo(
    deviceID,
    CL_DEVICE_SVM_CAPABILITIES,
    sizeof(cl_device_svm_capabilities),
    &caps,
    0
);
```

If the OpenCL device identified by deviceID doesn't support OpenCL 2.0, the returned err value is CL_INVALID_VALUE. Such return value indicates that SVM is not supported at all. Otherwise err is CL_SUCCESS and value returned in caps variable is a bit-field that describes a combination of the following values:

- CL_DEVICE_SVM_COARSE_GRAIN for coarse-grained buffer SVM
- CL_DEVICE_SVM_FINE_GRAIN_BUFFER for fine-grained buffer SVM
- CL_DEVICE_SVM_FINE_GRAIN_SYSTEM for fine-grained system SVM
- CL_DEVICE_SVM_ATOMICS for atomics support

To detect a specific SVM type together with the cross-device atomics availability, the following expressions can be used. If a specific expression is true, the corresponding SVM type is supported by the device.

SVM Type	**Expression**
No SVM support	`err == CL_INVALID_VALUE`
Coarse-grained buffer	`err == CL_SUCCESS && (caps & CL_DEVICE_SVM_COARSE_GRAIN)`
Fine-grained buffer	`err == CL_SUCCESS && (caps & CL_DEVICE_SVM_FINE_GRAIN_BUFFER)`
Fine-grained buffer with atomics	`err == CL_SUCCESS && (caps & (CL_DEVICE_SVM_FINE_GRAIN_BUFFER | CL_DEVICE_SVM_ATOMICS))`
Fine-grained system	`err == CL_SUCCESS && (caps & CL_DEVICE_SVM_FINE_GRAIN_SYSTEM)`
Fine-grained system with atomics	`err == CL_SUCCESS && (caps & (CL_DEVICE_SVM_FINE_GRAIN_SYSTEM | CL_DEVICE_SVM_ATOMICS))`

Alternatively, if the application has already queried for OpenCL 2.0 support and found that it is available, SVM coarse-grained buffer is also supported by default. In that case, it isn't necessary to detect it by calling clGetDeviceInfo if only this type of SVM is required by the application.

Overview of SVM Features

The following sections describe each of the SVM features. For each feature, a tag in a green box specifies the minimum SVM level required to use the feature.

Identifying an SVM Buffer Using a Regular Pointer

Coarse-grained buffer

OpenCL 1.2 requires identifying and managing memory resources such as buffers or images through an explicit host interface. This interface requires using an identifier of type cl_mem in all operations with an OpenCL buffer, such as passing it to kernels and mapping it for access on the host side. If the application needs the host to access an OpenCL buffer, it must specify this cl_mem handle to each operation. This complicates application code and makes it harder to use legacy code that accesses memory using conventional pointers.

OpenCL 2.0 SVM simplifies OpenCL programming by enabling access to memory resources using regular pointers rather than these `cl_mem` objects.

SVM buffers are created with the `clSVMAlloc` function. Much like the `malloc` function or the new operator, `clSVMAlloc` returns a conventional C/C++ pointer:

```
void* p = clSVMAlloc (

  context,            // an OpenCL context where this buffer is available

  CL_MEM_READ_ONLY,   // access mode for the kernel and other options; here
                     // only read-only access is required

  size,               // amount of memory to allocate (in bytes)

  0                   // alignment in bytes (0 means default)
);
```

The `clSVMAlloc` function creates an SVM allocation in a given OpenCL context. Though the function returns a conventional pointer, it can only be used in the specified context.

The allocation flags passed as the second argument to `clSVMAlloc` may be *OR*ed together, and are divided to two categories:

- The access mode required for kernel execution on the device, similar to ones for `clCreateBuffer`, it can be
  - **CL_MEM_READ_ONLY** – read-only memory when used inside a kernel
  - **CL_MEM_WRITE_ONLY** – memory is written but not read by a kernel
  - **CL_MEM_READ_WRITE** – memory is read and written by a kernel

- Parameters that allow specific operations on the memory available in advanced types of SVM:
  - **CL_MEM_SVM_FINE_GRAIN_BUFFER** – creates an SVM allocation that works correctly with fine-grained memory accesses (see sections Map-free access and Fine-grained simultaneous access in this document).
  - **CL_MEM_SVM_ATOMICS** – enables using SVM atomic operations to control visibility of updates in this SVM allocation (see section Fine-grained synchronization in this document).

Choosing the right value for the last argument of the `clSVMAlloc` function – *alignment* – is important for efficient operation on allocated SVM memory. The default value that is chosen by passing zero value, will work well if the application doesn't need stricter alignment requirements. However, a specific alignment value should be given if the allocated memory will be used for some data structure that requires alignment on a particular boundary.

Once allocated, OpenCL 2.0 platforms with *fine-grained* SVM support may just start using the returned pointer `p` directly like any conventional C/C++ pointer. However, platforms with coarse-grained support require special steps to use allocated SVM memory on the host: the host must map the memory object before accessing it and then unmap it afterwards. Refer to the Map-free access section of this document to understand the difference in more details.

The SVM pointer `p` is passed to an OpenCL `kernel` by calling the `clSetKernelArgSVMPointer` function that is similar to `clSetKernelArg` function used to pass regular `cl_mem` objects to the `kernel`:

```
clSetKernelArgSVMPointer(kernel, 0, p);
```

On the kernel side, there is no difference between a regular OpenCL 1.2 buffer passed as an argument and an SVM allocation. Both are represented as a pointer to the `global` address space:

```
kernel void mykernel (global float* p)
{
    . . .
}
```

To release SVM memory, the `clSVMFree` function is used:

```
clSVMFree (

  context,   // an OpenCL context used in corresponding clSVMAlloc call

  p          // a pointer to allocated with clSVMAlloc memory
);
```

If there is a need to synchronize the deallocation operation with OpenCL commands enqueued to command queue, there is another function that may serve better: `clEnqueueSVMFree`. It implements the same SVM memory deallocation as `clSVMFree`, with the addition that it is enqueued as a regular OpenCL command, for example, right after the kernel that uses that SVM memory.

In some cases using a regular pointer may be more troublesome than a conventional OpenCL 1.2 `cl_mem` object, for example in legacy OpenCL 1.2 code. In this case, for compatibility reasons, OpenCL 2.0 allows creating a `cl_mem` object on top of the memory previously allocated with `clSVMAlloc`. This is achieved by calling `clCreateBuffer` with CL_MEM_USE_HOST_PTR and passing the pointer that was returned from the `clSVMAlloc` function:

```
cl_mem buffer = clCreateBuffer (

  context,            // an OpenCL context where this buffer is available

  CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,   // access mode for the device and
                                            // other options

  size,               // amount of memory to allocate (in bytes)

  p,                  // pointer returned by clSVMAlloc

  &err                // resulting error code
);
```

By doing that, both `buffer` and `p` can be used to access the underlying SVM allocation.

Shared Virtual Address Space

Coarse-grained buffer

OpenCL 2.0 shared virtual memory, by its name, implies a shared address space. It means that the pointers assigned on the host can be seamlessly dereferenced in the kernel on the device side and vice versa. The pointers address the same data in this case. However, this is only true for pointers addressing data in SVM allocations and may not be true for OpenCL 2.0 regular buffer objects which aren't created as SVM allocations. From the kernel side, SVM allocations are represented as data in the `global` address space; hence only `global` pointers can be used for data sharing.

As OpenCL 1.2 doesn't support SVM allocations, so there is no guaranteed way to share pointers between the host and the devices. In OpenCL version 1.2 and lower, if an application needs to share a linked data structure (like a linked list or a tree) between the host and the device, *indices* rather than *pointers* should be used. This complicates managing *dynamically growing* data structures that spread across several separately allocated buffers because indices imply the need to use base addresses for relative access. And if the application needs linked data structures spread across several OpenCL buffers, an index is not enough to address the data.

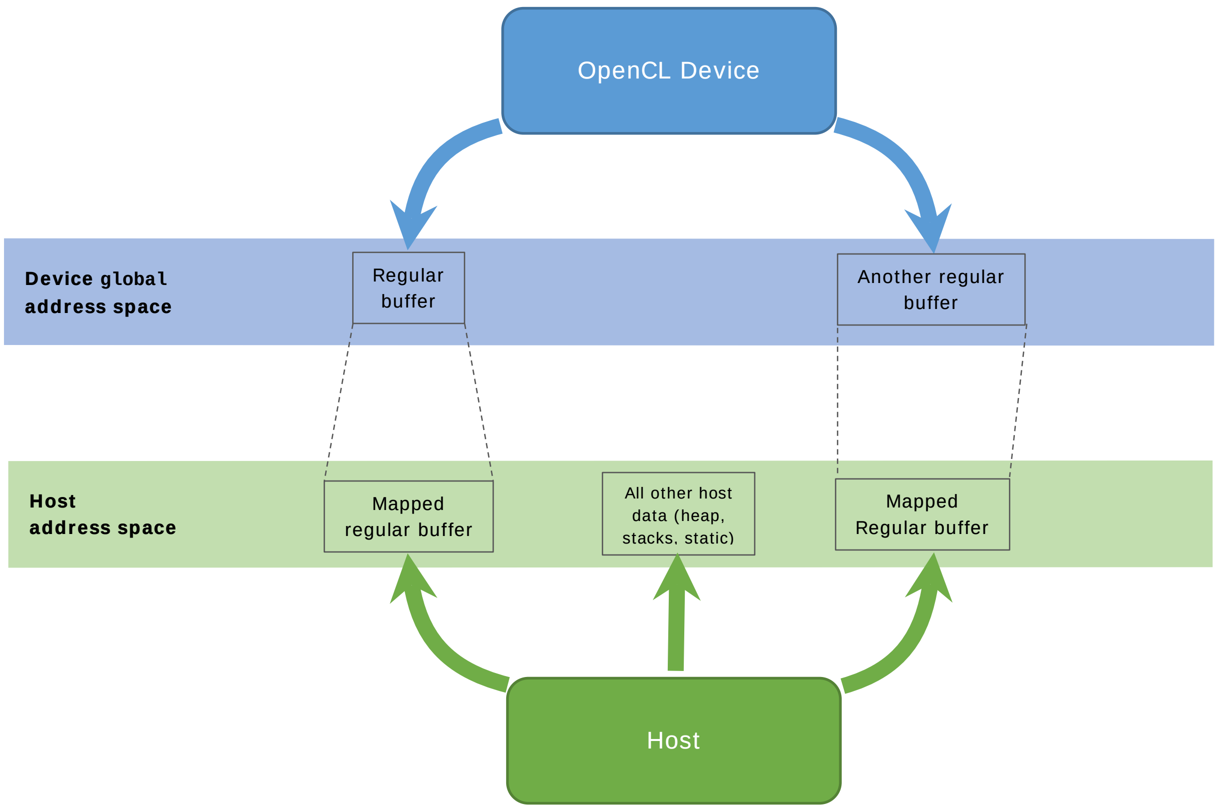


Figure 1: Schematic representation of the address spaces in OpenCL 1.2 and regular buffers with their mapped content.

An important observation is that a global address space pointer on an OpenCL 1.2 device may be represented in a way that is unlike a regular pointer on the host. The two pointers may have even different size. For example, a pointer on the device may be represented by a pair of a buffer index and an offset into that buffer. However, with OpenCL 2.0 SVM allocations, it is guaranteed that a global address space pointer on the device matches the pointer representation on the host.

Shared virtual memory shouldn't be confused with *shared physical memory* – they are different terms. The shared physical memory term is used when the host and a device use the same physical memory even if the virtual addresses they use don't match. This feature may be available on versions of OpenCL prior to OpenCL 2.0 depending on the vendor of OpenCL platform – it is not defined by the OpenCL specification. Shared physical memory enables efficient transfers of data between the host and the device, and may require following specific buffer allocation rules depending on the vendor.

Refer to Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics for guidelines on how to exploit the benefits of shared *physical* memory on Intel® Processor Graphics.

One of the important things to remember is that without SVM, even if data in a buffer is physically shared between the host and the device, the virtual addresses they use are not required to match. In fact, these two concepts – shared physical memory and shared virtual memory – are independent, and can be available separately or together depending on the vendor of OpenCL platform.

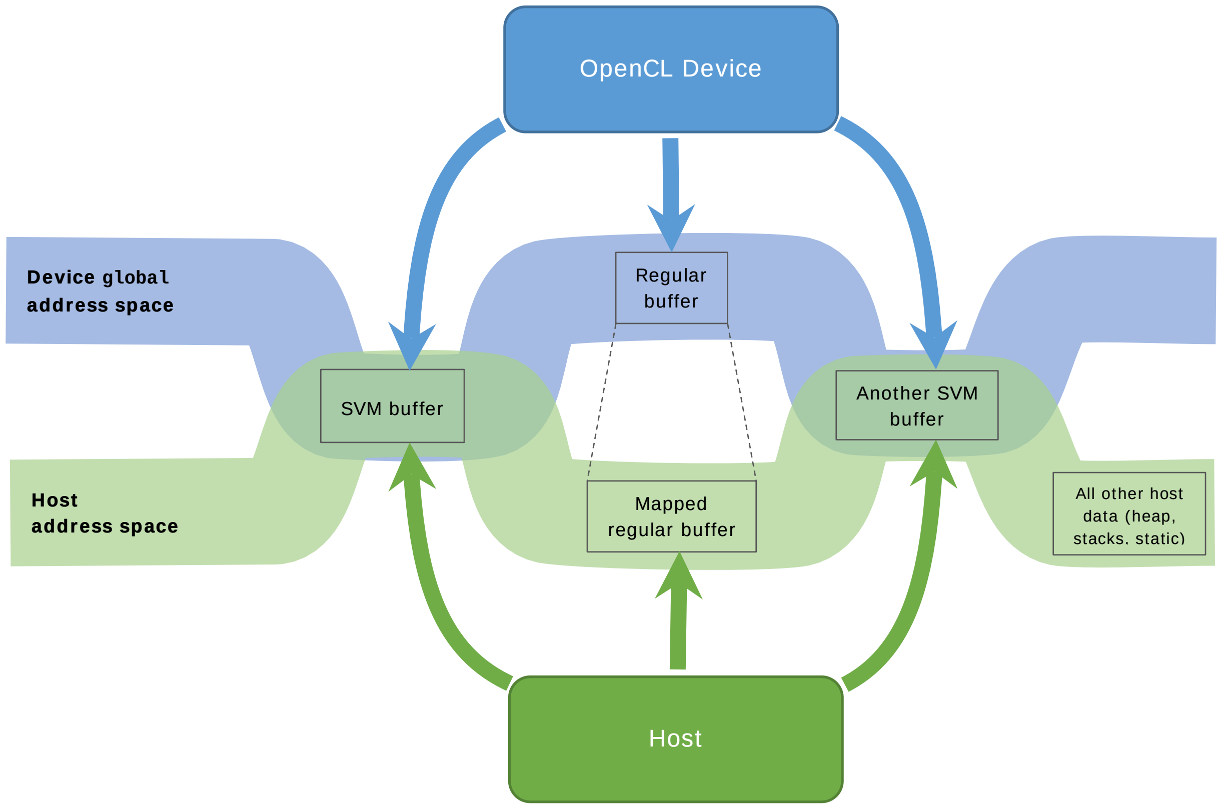


Figure 2: Schematic representation of the address spaces in OpenCL 2.0 and their overlap in areas where SVM buffers are allocated. A regular buffer and its mapped content are shown for comparison.

"Map-free" Access

Fine-grained buffer

Mapping/unmapping regions of an OpenCL buffer – SVM or not – is an important mechanism for host and device interaction. It is required when underlying hardware cannot resolve fine-grained accesses to a single OpenCL buffer from both sides. As an explicit mechanism, mapping/unmapping becomes too verbose when true fine-grained data exchange is needed between the host and the device.

Thanks to modern hardware, the OpenCL platform may free the application from doing explicit map/unmap commands. In this case the OpenCL platform enables doing accesses with any granularity to SVM allocation from both sides (the host and the device) leaving the burden of keeping coherent memory content to underlying hardware.

Map/unmap is required (coarse-grained SVM buffer)	**"Map-free" (fine-grained SVM buffer)**

`float* p = (float*)clSVMAlloc(…);`
 	
`clEnqueueSVMMap(…,`
`    CL_TRUE,  // block until map is done`
`    p, …);`
 	
`// Initialize SVM buffer`
`p[i] = …;`
 	
`clEnqueueSVMUnmap(…, p, …);`
 	
`clEnqueueNDRange(…);`
 	
`clEnqueueSVMMap(…,`
`    CL_TRUE,  // block until map is done`
`    p, …);`
 	
`// Read the data produced by the kernel`
`… = p[i];`
 	
`clEnqueueSVMUnmap(…, p, …);`
	`float* p = (float*)clSVMAlloc(…);`
	
	`// Initialize SVM buffer`
	`p[i] = …;`
	
	`clEnqueueNDRange(…);`
	
	`clFinish(…);`
	
	`// Read the data produced by the kernel`
	`… = p[i];`

When map/unmap is required – for coarse-grained SVM buffers and regular non-SVM buffers – a buffer is bound to the device side from the moment of creation. Each time the host reads or writes data in the buffer, it should explicitly enclose the access operators in map/unmap brackets. `clEnqueueSVMMap` is a request to give ownership over a specific region of SVM allocation to the host.

When map-free access available – in the fine-grained buffer SVM – there is no specific side that owns the content of the memory – any side can access it like two threads working on distinct cores on the CPU can access a piece of memory in the virtual address space of the process. Hence there is no need to map. This has a positive effect on application design, because now it doesn't require explicit the numerous and verbose map/unmap calls.

To create a SVM allocation that can operate with map-free fine-grained accesses, the `clSVMAlloc` function should be called with the **CL_MEM_SVM_FINE_GRAIN_BUFFER** memory flag, as in the following example:

```
  void* p = clSVMAlloc (
    context,            // an OpenCL context where this buffer is available
    CL_MEM_READ_WRITE | CL_MEM_SVM_FINE_GRAIN_BUFFER,
    size,               // amount of memory to allocate (in bytes)
    0                   // alignment in bytes (0 means default)
  );
```

Fine-Grained Coherent Access

Fine-grained buffer

Closely connected with map-free access, fine-grained simultaneous access provides the ability for the host and the device to modify the same region of memory simultaneously. It means that the host side can enqueue a kernel with the `clEnqueueNDRangeKernel` command and, without *waiting* until the kernel has finished execution, modify data in the same SVM allocation as the kernel side does:

Host side	**Kernel side**

```
float* p = (float*)clSVMAlloc(…);

clEnqueueNDRange(…, mykernel, …);
clFlush(…);

// Do not wait and modify some data
  p[0] = 0;
  p[2] = 2;
  p[4] = 4;

clFinish(…);
```

```
kernel void mykernel (global float* p)
{

  p[1] = 1;
  p[3] = 3;
  p[5] = 5;

}
```

```
// At this point (after clFinish) the host and the device have the same view of
// the memory at p, the first values of which are {0.f, 1.f, 2.f, 3.f, 4.f, 5.f}
```

Memory consistency is guaranteed at OpenCL synchronization points until the host and the device read and modify different bytes in the SVM allocation. If there is a need to modify the same bytes, or one side needs to read data written by another side, additional synchronization is required, like atomics and memory fences. This synchronization is needed to guarantee that the host and kernel will access consistent memory content.

After the kernel's execution completed, the SVM allocation's final memory content will be a combination of the modifications made by the kernel and the device even if those modifications are made in neighbor bytes when one byte is modified by the host, and another one is modified by the device.

Fine-Grained Synchronization

Fine-grained buffer + Atomics

One amazing feature of SVM is fine-grained synchronization between the host and SVM devices. With this feature, data written by one executable agent (host or device) can be made available to another agent without enqueuing any data transfer API commands like buffer read or map/unmap. Moreover, the agents may collaborate by executing concurrent atomic operations on the same variables placed in SVM allocations. The host and devices may also use memory fences to provide needed memory consistency. OpenCL 2.0 atomics are compatible with C++11 atomics.

Atomics applied on SVM allocation have the following properties that make them a powerful mechanism for host and the device synchronization:

- Access atomicity: transactional access to a particular variable of scalar type, like `int`. For example, with atomics the application can safely update the same integer variable from both the host and devices.
- Memory consistency: ensuring that reads or writes made to memory locations by one agent are visible to other agents and in the correct order. For example, if a circular queue is implemented in an SVM allocation, then insertion of a new queue item and the update of the queue's *next_item* pointer variable made by, say, the host, must be seen by a device in the right order. To provide this support, OpenCL 2.0 has several ordering rules that user may explicitly specify when using atomics.

To use atomics and fences, applications should specify memory_scope_all_svm_devices memory scope when calling atomic operations in the kernel. Also, it is required to allocate SVM memory with CL_DEVICE_SVM_ATOMICS:

```
void* p = clSVMAlloc (
  context,              // an OpenCL context where this buffer is available
  CL_MEM_READ_WRITE | CL_MEM_SVM_FINE_GRAIN_BUFFER | CL_MEM_SVM_ATOMICS,
  size,                 // amount of memory to allocate (in bytes)
  0                     // alignment in bytes (0 means default)
);
```

Once allocated this way, the resulting SVM memory can hold variables that can be used in atomic operations. Furthermore, the OpenCL 2.0 rules only guarantee memory consistency in that SVM memory. Data accesses where one executable agent (the host or the kernel on the device) writes data and another agent concurrently reads that data should only happen within such allocated SVM regions.

The following example illustrates concurrent initialization of an array with floating point numbers. The items are initialized concurrently by the host and the device. The index of an item is an atomically incremented counter shared between the host and the device in SVM area. Thanks to atomics, each element is initialized only once.

The host code:

```
// This variable will be used as shared atomically incremented counter
auto index = (std::atomic<cl_int>*)clSVMAlloc(…,
  CL_MEM_READ_WRITE | CL_MEM_SVM_FINE_GRAIN_BUFFER | CL_MEM_SVM_ATOMICS,
  sizeof(cl_int), 0
);

// Allocate area that will be concurrently written from both the host and
// the device side by index. This area will not be used for actual data
// exchange between the host and the device, hence it is not needed to be
// created with CL_MEM_SVM_ATOMICS flag.
auto p = (float*)clSVMAlloc (…,
  CL_MEM_READ_WRITE | CL_MEM_SVM_FINE_GRAIN_BUFFER,
  size*sizeof(float), 0
);

clSetKernelArgSVMPointer(kernel, 0, index);
clSetKernelArgSVMPointer(kernel, 1, p);

clEnqueueNDRangeKernel(…,
  kernel, …
  &size, …);    // global size matches with the number of elements in p

clFlush(…);

int localIndex;
while((localIndex =
     std::atomic_fetch_add_explicit(
       index, 1,
       std::memory_order_relaxed)
     ) < size)
{
  p[localIndex] = localIndex;
}

clFinish(…);

// At this point p is initialized in arbitrary order by the host and the device
// concurrently. Each element is initialized once.
```

The kernel code:

```
kernel void mykernel (global atomic_int* index, global float* p)
{
  int localIndex = atomic_fetch_add_explicit(
    index, 1,
    memory_order_relaxed,
    memory_scope_all_svm_devices
  );

  if(localIndex < get_global_size(0))
    p[localIndex] = localIndex;
}
```

For more information on atomic operations, refer to Using OpenCL™ 2.0 Atomics.

Sharing the Entire Host Address Space

Fine-grained system

OpenCL platforms that support *system* SVM allow a kernel on a device to use any data in the host address space. There is no need to call `clSVMAlloc` to allocate SVM memory as is required in the buffer flavors of SVM. Any memory available to the host – for example, obtained with the `malloc` function or the new operator – is also available for the kernel on the device.

This property of system SVM is important for applications that don't have control over memory allocation, such as ones that use libraries that allocate memory internally. Another good example is porting of existing C/C++ applications to OpenCL to enable them to run on a GPU. If the application is large and complex with many places where memory is allocated, it may be difficult to port it to use OpenCL 2.0 *buffer* SVM because each memory allocation used by the kernel should be rewritten to use the `clSVMAlloc` function. System SVM doesn't require this.

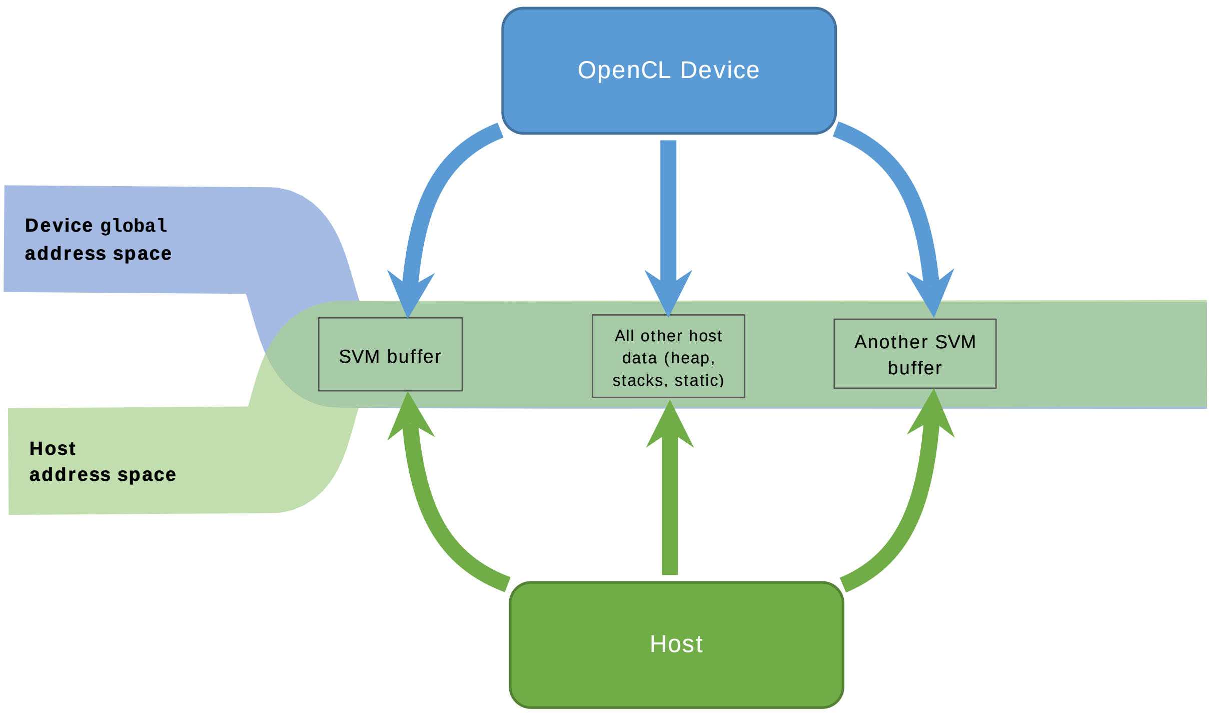
Although any data in the host address space may be used, data should be properly aligned as required by the OpenCL specification. Furthermore, stronger alignment rules may be required to make data accesses efficient depending on the OpenCL platform used.

The following code illustrates creating an SVM allocation and passing it to a kernel. The code from the left and from the right are different depending on availability of system SVM support. The code from the left side is required when there is no system SVM support available on the OpenCL platform. The code from the right is correct when the system SVM support is available. As system SVM allows to share any host data, there is no need to allocate memory with the `clSVMAlloc` function.

Buffer SVM allocation	**System SVM allocation**

`float* p = (float*)clSVMAlloc(`
`  context,`
`  CL_MEM_READ_WRITE |`
`    CL_MEM_SVM_FINE_GRAIN_BUFFER,`
`  size,`
	`// _aligned_malloc is one of the`
	`// methods to allocate aligned`
	`// memory to ensure efficient data`
	`// processing in the kernel`
	`float* p = (float*)_aligned_malloc(`
	`  size,`
	`  sizeof(cl_float16)`
	`);`

`  0`
`);`
 	
`clSetKernelArgSVMPointer``(`
`  mykernel, `**`p`**`, …);`
 	
`clEnqueueNDRange``(…, mykernel, …);`
	`clSetKernelArgSVMPointer``(`
	`  mykernel, `**`p`**`, …);`
	
	`clEnqueueNDRange``(…, mykernel, …);`



OpenCL Device
Device global address space
SVM buffer
All other host data (heap, stacks, static)
Another SVM buffer
Host address space
Host

Figure 3: Schematic representation of the address spaces in OpenCL 2.0 with support of the system SVM.

Implicit Use of Any SVM Allocation

Fine-grained system

Any buffer that is used by a kernel on a device should be passed to the kernel with `clSetKernelArg` in OpenCL 1.2 and higher. A similar requirement is also true for OpenCL 2.0 SVM allocations if fine-grained *system* SVM is not available on the platform. For platforms with only buffer SVM support – no matter whether this is coarse-grained or fine-grained SVM – the host application should call one of the two following functions to *explicitly* pass SVM allocations to a specific kernel:

- **`clSetKernelArgSVMPointer`**: to pass a pointer to an SVM allocation as a kernel argument;
- **`clSetKernelExecInfo`**: to pass pointers to all SVM allocations that can be reached and accessed by a specific kernel, but are not passed as kernel arguments. For example, this may happen when a pointer to one SVM allocation is stored in another SVM allocation. The call must be made for each kernel separately. Refer to the SVMBasic tutorial for more information [OpenCL 2.0 Shared Virtual Memory Code Sample](#).

Accessing SVM pointers that are not passed one of these ways is prohibited with *buffer* SVM. In the cases when many SVM pieces each allocated with a separate clSVMAlloc function should be used in the application, the requirement to notify each kernel about all allocations may turn to be too restrictive. Also, an OpenCL platform may limit the number of SVM allocations used per kernel.

One advantage of fine-grained *system* SVM is that the host is not required to make these notification calls to enable kernels to access SVM allocations not passed as kernel arguments. With *system* SVM, each kernel can access any pointer: ones explicitly allocated with clSVMAlloc and ones pointing to system-allocated memory anywhere in the host address space. If a kernel accesses many host memory locations by traversing pointers, then using system SVM is especially convenient because there is no need to specify each memory allocation with clSetKernelExecInfo for each kernel.

The following code illustrates the difference between buffer SVM and system SVM while passing two-element linked-list data structure to the kernel:

Explicit indirect use (buffer SVM allocation)	**Implicit indirect use (system SVM allocation)**

struct Node {
  float value;
  Node* next;
};
	struct Node {
	  float value;
	  Node* next;
	};

Node* **node1** = (Node*)**clSVMAlloc**(
  context,
  CL_MEM_READ_WRITE |
    CL_MEM_SVM_FINE_GRAIN_BUFFER,
  sizeof(Node),
  0
);
	Node* **node1** = **new** Node;

node1->value = 1.f;	**node1**->value = 1.f;

Node* **node2** = (Node*)**clSVMAlloc**(
  context,
  CL_MEM_READ_WRITE |
    CL_MEM_SVM_FINE_GRAIN_BUFFER,
  sizeof(Node),
  0
);
	Node* **node2** = **new** Node;

node2->value = 2.f;	**node2**->value = 2.f;
// Link **node1** to **node2** **node1**->next = **node2**; **node2**->next = 0;	// Link **node1** to **node2** **node1**->next = **node2**; **node2**->next = 0;
// Pass **node1** as a kernel argument **clSetKernelArgSVMPointer**(mykernel, **node1**, …);	// Pass **node1** as a kernel argument **clSetKernelArgSVMPointer**(mykernel, **node1**, …);

// Pass **node2**; it will be indirectly
// used by a kernel through **node1**->next
clSetKernelExecInfo(
  mykernel,
  CL_KERNEL_EXEC_INFO_SVM_PTRS,
  sizeof(**node2**),
  &**node2**
);
	// **node2** is not passed explicitly,
	// however kernel can still access it
	// through **node1**->next

clEnqueueNDRange(…, mykernel, …);	**clEnqueueNDRange**(…, mykernel, …);

If the kernel doesn't use any system-allocated SVM memory and all used buffer SVM allocations are passed to the kernel by one of the `clSetKernelArgSVMPointer` or `clSetKernelExecInfo` functions as described above, the application can optionally notify the runtime about this. This is achieved by calling `clSetKernelExecInfo` with `CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM = CL_FALSE`:

```
cl_bool flag = CL_FALSE;

clSetKernelExecInfo(
    mykernel,
    CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM,
    sizeof(flag),
    &flag
);
```

Conclusion and Key Takeaways

With OpenCL 2.0, the support for Shared Virtual Memory (SVM) introduces one of the most significant improvements for the programming model. Previously memory spaces of the host and OpenCL devices were distinct which added a lot of complexity to OpenCL host logic. Now the SVM bridges the gap, so that memory is accessible to both the host and OpenCL devices using a single pointer.

SVM is foremost a productivity feature that makes porting existing C/C++ code to the OpenCL simpler, especially for the pointer-linked data structures. But SVM is not only about eliminating the excess host OpenCL code, it also allows for tighter synchronization between host and OpenCL devices via using fine-grained coherent accesses to SVM memory with atomics.

There are different levels of SVM support depending on OpenCL platform hardware capabilities. It is highly important for developers to be aware of the differences between SVM types and design the host logic accordingly.

The higher level of SVM support – moving from coarse-grained buffer SVM to fine-grained system SVM – the more *productive* ways of host logic organization it provides. In the same time, using advanced levels of the SVM support makes host OpenCL code *less portable*, because not all the SVM features available on all OpenCL 2.0 platforms. Hence, selection of target SVM type for an OpenCL application is a tradeoff between *productivity* and *portability*.

Find out more on the SVM and associated topics using the resources in the section below.

References

OpenCL terms and definitions in Khronos OpenCL 2.0 API Specification Glossary

3.3.3 Memory Model: Shared Virtual Memory (in Khronos OpenCL 2.0 API Specification)

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)

3.3.6 Memory Model: Memory Ordering Rules (in Khronos OpenCL 2.0 API Specification)

5.6 Shared Virtual Memory (in Khronos OpenCL 2.0 API Specification)

Compute Architecture of Intel Processor Graphics Gen8 [PDF]

Get started with OpenCL 2.0 API

OpenCL 2.0 Shared Virtual Memory Code Sample

Using OpenCL™ 2.0 Atomics

Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics

Using OpenCL™ 2.0 sRGB Image Format