Tutorial: Using Shared Virtual Memory

Intel® SDK for OpenCL™ Applications

OpenCL Sample Application Code

Legal Information

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Overview

This sample demonstrates the fundamentals of using Shared Virtual Memory (SVM) capabilities in OpenCL™ applications. The SVM Basic code sample uses the OpenCL 2.0 APIs to query SVM support and manage SVM allocations for the selected OpenCL 2.0 device.

The sample code implements an algorithm to demonstrate pointer sharing between host and device with OpenCL SVM features. Advanced topics like use of atomics within SVM allocations and associated performance considerations are out of the scope of this tutorial.

About This Tutorial	This tutorial demonstrates an end-to-end workflow you can ultimately apply to your own applications: • Check SVM availability • Allocate SVM memory. Access SVM memory in host code • Pass pointers to SVM memory to OpenCL C device code
Estimated Duration	10-15 minutes.
Learning Objectives	After you complete this tutorial, you should be able to: • Allocate SVM memory • Use the SVM memory in host- and device code.
More Resources	Intel SDK for OpenCL Applications documentation: - Optimization Guide - User's Guide OpenCL Specification Version 2.0 http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

About Shared Virtual Memory (SVM)

Intel's Shared Virtual Memory capabilities can be programmed via OpenCL 2.0's Shared Virtual Memory (SVM) APIs and OpenCL C language support. OpenCL SVM enables the host and device portions of OpenCL applications to seamlessly share pointers and complex data-structures. OpenCL 2.0 also defines memory model consistency guarantees for SVM.

OpenCL 2.0 defines three types of SVM:

- **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 and stores* within OpenCL *buffer* memory objects.
- **Fine-Grained system SVM**: Sharing occurs at the granularity of *individual loads/stores* occurring *anywhere within the host memory*.

In Coarse-Grained buffer SVM, consistency is enforced at synchronization points and with map/unmap commands to drive updates between the host and the device. This form of SVM is similar to non-SVM use of memory; however, it lets host and device share a single region of virtual memory address space containing pointer-based data structures (such as linked-lists), which was not possible in OpenCL versions lower than 2.0.

Coarse-Grained buffer SVM is the minimum required by the core OpenCL 2.0 specification. So if there is an OpenCL 2.0 device in your OpenCL platform, it should support the necessary functionality to run the sample code. The other two (fine-grained) levels described above are optional OpenCL 2.0 features. Consider

coarse-grained buffer SVM as a compatibility option to be able to run on any OpenCL 2.0 device.

Even the basic coarse-grained buffer type of SVM lets applications avoid duplicating data structure representations between the host and each OpenCL device. Thus, SVM can save extra memory copying and eliminate the need for fragile data structure marshalling code and its overhead. Pointers initialized on the host can be used “as is” on the device side in OpenCL C kernels. This is the main benefit of the shared address space that SVM provides.

Fine-grained buffer SVM eliminates the need to call map/unmap OpenCL API functions on the host to access SVM allocations. This simplifies the programming experience in real applications compared to coarse-grained SVM.

Besides a shared virtual address space, fine-grained SVM gives the application the capability to seamlessly read and write *the same region* of memory from both the host and from the device *simultaneously.* That is true for any non-overlapping modifications with granularity of byte. To modify *the same bytes* of memory concurrently, applications should synchronize between device(s) and the host via OpenCL 2.0-defined atomic operations applied on variables in SVM allocations. This expands the programmability of OpenCL 2.0 platforms opening the door to true heterogeneous programming.

Those advanced topics are out of the scope for this basic tutorial, which focuses on the key API functions necessary for allocation and use of SVM buffers.

See Also

OpenCL 2.0 Specification:

- SVM Introduction: *3.3.3 Memory Model: Shared Virtual Memory*
- SVM Host API: *5.6 Shared Virtual Memory*

Prerequisites

Before you start with the tutorial, make sure your system meets the following requirements.

To build and run the sample application, you need

- A processor based on Intel® microarchitecture Broadwell. See the list of supported processors in the SDK release notes.
- Intel OpenCL Driver for Intel Graphics (which already includes Intel OpenCL 2.0 runtime for CPU device) available at the OpenCL Drivers and Runtimes for Intel® Architecture page.
- Microsoft Visual Studio* 2010 and higher.
- Intel® SDK for OpenCL™ Applications 2014 and higher, available at the SDK main page.

Navigation Quick Start

This tutorial includes sample code that you can compile using Microsoft Visual Studio 2010 and higher. Find the relevant solution file in the **sample root directory** > **SVMBasic** subfolder.

Building and Running Code Sample

To build the SVM Basic code sample,

1. Double-click the solution file (`*.sln`) relevant to your Visual Studio version.
2. Select **Build** > **Build Solution**.

Then to run the application,

1. Select a project file in the Visual Studio **Solution Explorer**.
2. If the sample application is not set as a startup project, right-click the project and select **Set as StartUp Project**.
3. Press **Ctrl+F5** to run the application.

To run the application in **Debug** mode, press **F5**.

You can also run the sample application using the command-line interface:

1. Run the command prompt.
2. Switch to the directory, where the solution file resides.
3. Go to the directory according to the platform configuration:
   - \Win32 - for Win32 configuration
   - \x64 - for x64 configuration
4. Open the appropriate project configuration (Debug or Release).
5. Run the sample by entering `SVMBasic.exe`.

Controlling the Sample Application

The sample executable is a console application. You can control OpenCL platform, device type, and array size with the dedicated command-line options. The sample uses default parameters if you run it without specifying any command-line options.

Command-Line Options		
Short Form	**Full Form**	**Description**
`-h`	`--help`	Shows command-line options with descriptions.
`-p`	`--platform`	Selects the platform, the devices of which are used. **(Default value: Intel)**
`-t`	`--type`	Selects an OpenCL device to run by *type.* First device of the specified type will be picked. The following options are relevant: • `cpu` • `gpu` Combine the `-t` option with the `-p` option, which specifies OpenCL platform. **(Default value: gpu)**
`-d`	`--device`	Selects an OpenCL device by *number* (or name). This option combines well with the previous ones. For example, if you have multiple devices of the same *type* specified with `–t`, you can select the particular device to run using `-d`. **(Default value: 0)**
`-s`	`--size`	Amount of input floating-point numbers to process with the OpenCL kernel. **(Default value: 1048576)**

Sample Implementation

For illustrative purposes, the sample implements a traversal algorithm that handles a data structure populated with pointers. Core functionality is placed in the following files:

- `svmbasic.cpp` – OpenCL host code
- `svmbasic.cl` – OpenCL C kernel device code
- `svmbasic.h` – Definition of the structure type, arrays of which are used by both the device and the host code.

Code Execution Scenario

The sample code executes according to the following scenario:

1. Selects the OpenCL platform and device according to the specified command-line arguments.
2. Checks SVM availability for the selected OpenCL device. If SVM capabilities are not available the application exits immediately.
3. Allocate SVM memory by creating two arrays in two SVM buffers on the host side.
4. Access SVM memory on the host by mapping the newly-created arrays on the host and populating them via pointers in the host address space. The map/unmap pair is only required for coarse-grained buffer SVM.
5. Pass pointers to SVM memory to the device to enable the OpenCL C kernel to access SVM memory.
6. The OpenCL C kernel reads the shared memory and traverses the arrays using those shared pointers. With SVM, such pointers work seamlessly in OpenCL C kernels and point to the same data, just as they do in the host code.
7. The OpenCL C kernel performs arithmetic operations with values from the SVM buffers and writes to the dedicated output buffer.
8. The kernel-written data is read in the host code and validated against a CPU reference implementation to prove its correctness.

Steps 2-6 are specific for SVM, while the rest of the steps are common infrastructure. This tutorial document focuses on the SVM-specific steps only.

Using SVM in Your Application

The rest of the tutorial sections guide you through the process of using SVM in your application. The following steps correspond to the scenario described in the Code Execution Scenario section.

Check SVM Availability

You can check SVM availability using the `checkSVMAvailability` function, which queries `CL_DEVICE_SVM_CAPABILITIES` from `clGetDeviceInfo`. See `svmbasic.cpp` for details.

```
bool checkSVMAvailability (cl_device_id device)
{
    cl_device_svm_capabilities caps;
    cl_int err = clGetDeviceInfo(
        device,
        CL_DEVICE_SVM_CAPABILITIES,
```

```
        sizeof(cl_device_svm_capabilities),
        &caps,
        0);
    return err == CL_SUCCESS;
}
```

This returns `true` if at least coarse-grained buffer SVM is available. `cl_device_svm_capabilities` is a bit-field that describes a combination of the following values:

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

OpenCL 2.0 specification requires that `CL_DEVICE_SVM_COARSE_GRAIN_BUFFER` is supported by all OpenCL 2.0 devices.

For example, if your application requires fine-grained buffer support, the `return` statement should be replaced with

```
  return err == CL_SUCCESS && (caps & CL_DEVICE_SVM_FINE_GRAIN_BUFFER);
```

`false` might be returned in the case when the device is not an OpenCL 2.0 device (an OpenCL 1.2 device for example). In this case `clGetDeviceInfo` should respond with an error because `CL_DEVICE_SVM_CAPABILITIES` constant is an invalid value for OpenCL implementation version lower than OpenCL 2.0.

Allocate SVM Memory

Create two synthetic data structures - arrays to be allocated and filled:

- the first array should consist of `Element` structures (refer to `svmbasic.h` for the structure definition),
- the second one of `float` values.

Create the first array in the SVM memory with the `clSVMAlloc` function. This function returns a regular pointer in the host address space. At the same time, you can pass this pointer to the kernel to be used like a regular OpenCL buffer. Pointers to the memory in the SVM allocation are mapped to the same bytes on the host and on the device sides.

The arguments for `clSVMAlloc` are similar to `clCreateBuffer`. See the example below:

```
Element* inputElements = (Element*)clSVMAlloc(
    context,              // an OpenCL context where this buffer is
                          // available
    CL_MEM_READ_ONLY,
    size*sizeof(Element), // amount of memory to allocate (in bytes)
    0                     // alignment in bytes (0 means default)
);
```

In case of fine-grained buffer SVM, while allocating SVM memory, you need to pass `CL_MEM_SVM_FINE_GRAIN_BUFFER` as an extra flag for `clSVMAlloc`, like in the following example:

```
Element* inputElements = (Element*)clSVMAlloc(
    context,              // an OpenCL context where this buffer is
                          // available
```

```
    CL_MEM_READ_ONLY | CL_MEM_SVM_FINE_GRAIN_BUFFER,
    size*sizeof(Element), // amount of memory to allocate (in bytes)
    0                     // alignment in bytes (0 means default)
);
```

Do it for all allocations in case of using the fine-grained buffer SVM.

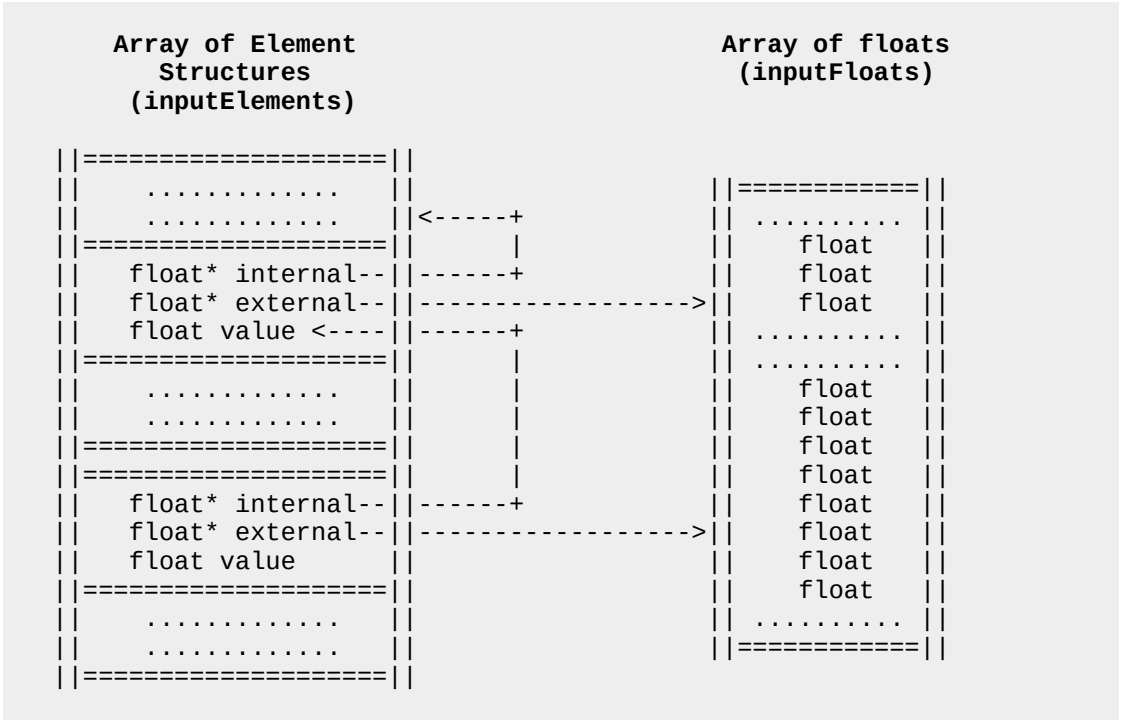
Each structure of the first array (`struct Element`) has two pointers:

- The first pointer (*internal*) points to the value field of another `Element` from the same array.
- The second pointer (*external*) points to a floating point value in the separate array.

You create the second array in a similar way:

```
float* inputFloats = (float*)clSVMAlloc(
    context,              // context where this buffer is available
    CL_MEM_READ_ONLY,     // fine-grained: CL_MEM_SVM_FINE_GRAIN_BUFFER
    size*sizeof(float),   // amount of memory to allocate (in bytes)
    0                     // default alignment (0 means default)
);
```

Pointer values of the `Element` array entries are set randomly. The data structures do not reflect any real usage scenario, but are illustrative for a simple device-side traversal. After initialization the linked data structure appears as illustrated below:



The two arrays described above, are created separately to illustrate two different ways of passing pointers to SVM allocations to the kernel:

- The first array (`inputElements`) is passed to the kernel as one of the kernel arguments with the `clSetKernelArgSVMPointer` OpenCL 2.0 API function.

- The second array (`inputFloats`) is used by the kernel *indirectly* and should be also made known to the kernel with the `clSetKernelExecInfo` OpenCL 2.0 new API (refer to Passing SVM Pointers to Kernel section below for details).

Access SVM Memory in Host Code

Much like `new` or `delete` operators, `clSVMAlloc` returns a conventional C/C++ pointer. Once allocated, OpenCL 2.0 platforms with *fine-grained* SVM support may just start using the pointer directly like any conventional C/C++ pointer.

In contrast to fine-grained SVM, coarse-grained-only enabled platforms should perform special steps to use allocated SVM memory on the host. In this case, although `clSVMAlloc` function returns a regular host pointer, according to the OpenCL 2.0 specification you cannot access this SVM memory in the host code without *mapping* it. To map the memory region, you should call `clEnqueueSVMMap`:

```
clEnqueueSVMMap(
    queue,          // OpenCL queue
    CL_TRUE,        // block on this command until map is done
    CL_MAP_WRITE,   // map for writing on the host
    inputElements,  // pointer to the beginning of the SVM region to map
    sizeof(Element)*size,  // the size of the SVM region to map
    0, 0, 0
);
```

The code should map the SVM region for initialization, so you should use `CL_MAP_WRITE`. To be able to use the region in the OpenCL kernel, after initialization of `inputElements`, you need to *unmap* the region:

```
clEnqueueSVMUnmap(
    queue,
    inputElements,
    0, 0, 0
);
```

By using map/unmap commands as synchronization points you coordinate the ownership over the SVM allocations between the host and OpenCL device(s), hence keeping the content of the allocations consistent. Refer to paragraph *5.6.2 "Memory consistency for SVM allocations"* of the OpenCL 2.0 specification for details.

As it has been already said before, using `clEnqueueSVMMap` and `clEnqueueSVMUnmap` commands is necessary for *coarse-grained* buffer SVM only.

Pass SVM Pointers to a Kernel

Pointers to SVM memory may be passed to the OpenCL C kernel in two ways. The first one is to pass the SVM pointer as a kernel argument (such as `inputElements` below) with `clSetKernelArgSVMPointer()`. This is similar to passing a conventional OpenCL buffer cl_mem object as a regular argument to the kernel:

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

This pointer will point to the first argument of the kernel (pointer to the array of `Element`, see the `svmbasic.cl` file for details):

```
kernel void svmbasic (global Element* elements, global float *dst)
{
    . . .
}
```

The second method of passing SVM allocations is used when an SVM allocation is accessed *implicitly* by the OpenCL C Kernel (for example, when it is pointed to by pointers within another SVM allocation). This is the case for `inputFloats` usage in the kernel, as the sample does not pass `inputFloats` pointer directly as a kernel argument. Instead, `inputFloats` elements are used through pointers stored in `inputElements`:

```
kernel void svmbasic (global Element* elements, global float *dst)
{
    int id = (int)get_global_id(0);

    float internalElement = *(elements[id].internal);
    float externalElement = *(elements[id].external);
    dst[id] = internalElement + externalElement;
}
```

In this case, according to OpenCL 2.0 specification (refer to section 5.9.2 'Setting Kernel Arguments') the SVM pointer must be specified using `clSetKernelExecInfo` function with parameter `CL_KERNEL_EXEC_INFO_SVM_PTRS`:

```
clSetKernelExecInfo(
    kernel,
    CL_KERNEL_EXEC_INFO_SVM_PTRS,
    sizeof(inputFloats),
    &inputFloats
);
```

You need to do that for each non-argument SVM pointer in your program that is used in the kernel.

Note that using `clSetKernelExecInfo` is a necessary step for both coarse-grained and fine-grained *buffer* SVM allocations, but not for fine-grain *system* SVM allocations, which operates on the full range of the system virtual memory addresses.

Summary

This tutorial demonstrated an end-to-end workflow you can apply to your own application.

Step	Tutorial Recap	Key Tutorial Take-aways
Check SVM availability	Use the `checkSVMAvailability` function to query OpenCL device capabilities.	Make sure your system contains OpenCL 2.0 devices, so that you are able to utilize the SVM feature.
Allocate SVM memory	Use the `clSVMAlloc` function to create two arrays: an array of `Element` structures and an	By allocating pointers to SVM memory, you create a pool of memory to be available for the host

	element of float values.	and device sides of the application in a shared address space.
Access SVM memory in host code	In case of fine-grained SVM: use the pointers to SVM memory as you would use conventional C/C++ pointers. In case of coarse-grained SVM: use the `clEnqueueSVMMap` function to enable the host side of the application to use the allocated SVM memory.	In case of coarse-grained SVM only, to be able to access the allocated SVM memory on the host side of the application, you need to map the memory. Make sure you unmap the memory after using it in the host side, so that the device side of the application is also able to use the memory region. Fine-grained SVM doesn't require those steps.
Pass SVM memory to the kernel	Use the `clSetKernelArgSVMPointer` to pass the SVM pointer explicitly as a kernel argument, or use the `clSetKernelExecInfo` function with the `CL_KERNEL_EXEC_INFO_SVM_PTRS` parameter to pass the SVM pointers implicitly.	Even indirectly accessed (not through kernel arguments) SVM should be passed to the kernel via the `clSetKernelExecInfo`.

Exposed OpenCL APIs

The sample application focuses on the following API functions:

- `clSVMAlloc`
- `clEnqueueSVMMap`
- `clEnqueueSVMUnmap`
- `clSetKernelArgSVMPointer`
- `clSetKernelExecInfo`
- `clSVMFree`
