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

By Adam Lake (Intel) (https://software.intel.com/en-us/user/334331), Updated September 16, 2014 Translate

Downloads

Download Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics (/file/408528/download) [PDF 673KB]

Download OpenCL Zero Copy code sample (/file/407919/download) [ZIP 22.4KB]

Introduction

This document provides guidance to OpenCL™ developers who want to optimize applications running on Intel® processor graphics. Specifically, this document shows you how to minimize the memory footprint of applications and reduce the amount of copying on buffers in the shared physical memory system of an Intel® System on Chip (SoC) solution. It also provides working source code to demonstrate these principles.

The OpenCL 1.2 Specification includes memory allocation flags and API functions that developers can use to create applications with minimal memory footprint and maximum performance. This is accomplished by eliminating extra copies during execution, referred to as *zero copy* behavior. This document augments the OpenCL API specification by giving guidance specific to Intel processor graphics.

Key Takeaway

To create zero copy buffers, do one of the following:

- Use **CL_MEM_ALLOC_HOST_PTR** and let the runtime handle creating a zero copy allocation buffer for you
- If you already have the data and want to load the data into an OpenCL buffer object, then use **CL_MEM_USE_HOST_PTR** with a buffer allocated at a 4096 byte boundary (aligned to a page and cache line boundary) and a total size that is a multiple of 64 bytes (cache line size).

When reading or writing data to these buffers from the host, use **clEnqueueMapBuffer()**, operate on the buffer, then call **clEnqueueUnmapMemObject()**. This paper contains code samples to demonstrate the best known practices on Intel® platforms.

Motivation

Memory management within the GPU driver has a complicated set of memory usage scenarios that need to be considered. Applications can inform the driver of their usage by specifying flags during memory allocation as well as through specific memory access or transfer APIs called during runtime. Sometimes driver implementations need to create or manage internal copies of memory buffers to facilitate servicing these API calls. For example, internal memory buffer copies might be created to support the memory layout preferred by the CPU or GPU or to improve caching behavior. Such copies may be necessary in these scenarios, but they can detrimentally impact performance. Application developers need device-specific knowledge in order to know how to avoid these copies.

Definitions

Before going into the technical details, here are some definitions of terms used in this article.

- **Host memory:** Memory accessible on the OpenCL host.
- **Device memory:** Memory accessible on the OpenCL device.
- **Zero copy**: Refers to the concept of using the same copy of memory between the host, in this case the CPU, and the device, in this case the integrated GPU, with the goal of increasing performance and reducing the overall memory footprint of the application by reducing the number of copies of data.
- **Zero copy buffers**: Buffers created via the **clCreateBuffer()** API that follow the rules for zero copy. This is implementation dependent so the rules on one device may be different than another.
- **Shared Physical Memory**: The host and the device share the same physical DRAM. This is different from shared *virtual* memory, when the host and device share the same virtual addresses, and is not the subject of this paper. The key hardware feature that enables zero copy is the fact that the CPU and GPU have shared *physical* memory. Shared physical and shared virtual memories are not mutually exclusive.
- **Virtual Memory:** The memory model used by the operating system to give the process perceived ownership of its own dedicated memory space. Pointers that programmers operate on are not physical memory addresses but instead virtual addresses that are part of a virtual address space. The platform handles conversions between these virtual addresses and the physical memory address.
- **Intel processor graphics**: The term used when referring to current Intel graphics solutions. Product names for Intel GPUs integrated in SoC include Intel® Iris™ graphics, Intel® Iris™ Pro graphics, or Intel® HD Graphics depending on the exact SoC. For additional hardware architecture details see the Intel® Gen 7.5 Compute Architecture document referenced at the end of this document or http://ark.intel.com/ (http://ark.intel.com/).

Intel® Processor Graphics with Shared Physical Memory

Intel processor graphics shares memory with the CPU. Figure 1 shows their relationship. While not shown in this figure, several architectural features exist that enhance the memory subsystem. For example, cache hierarchies, samplers, support for atomics, and read and write queues are all utilized to get maximum performance from the memory subsystem.

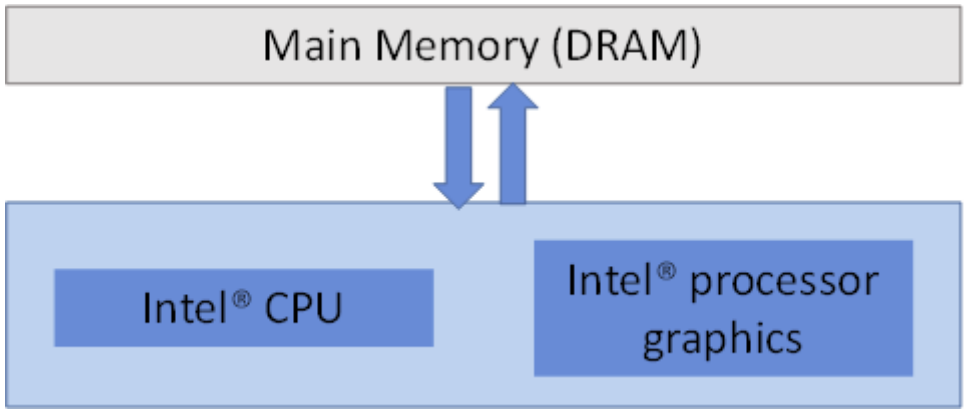

Figure 1.*Relationship of the CPU, Intel® processor graphics, and main memory. Notice a single pool of memory is shared by the CPU and GPU, unlike discrete GPUs that have their own dedicated memory that must be managed by the driver.*

Benefits of Zero Copy

With Intel processor graphics, using zero copy always results in better performance relative to the alternative of creating a copy on the host or the device. Unlike other architectures with non-uniform memory architectures, memory shared between the CPU and GPU can be efficiently accessed by both devices.

Memory Buffers in OpenCL

The performance of buffer operations in OpenCL can be different on different OpenCL implementations. Here, we clarify the behavior on Intel processor graphics.

Creating Buffers with clCreateBuffer()

One use case for OpenCL is *when memory is already populated on the host* and you want the device to read this data. In this case use the flag **CL_MEM_USE_HOST_PTR** to create the buffer. This may be the case when we are using OpenCL with existing codebases. When using the **CL_MEM_USE_HOST_PTR** flag, if we want to guarantee a zero copy buffer on Intel processor graphics, we need to ensure that we adhere to two device-dependent alignment and size rules. We must create a buffer that is aligned to a 4096 byte boundary and have a size that is a multiple of 64 bytes. Also note that if we write into this buffer, we will overwrite the original contents of the buffer. For clarity we include a short code sequence at the end of this section to test a buffer to determine if it meets this criteria.

CL_MEM_USE_HOST_PTR: Use this when a buffer is already allocated as page-aligned with **_aligned_malloc()** instead of **malloc()** and a size that is a multiple of 64 bytes:

```
1 int *pbuf = (int *)_aligned_malloc(sizeof(int) * 1024, 4096);
```

Using **_aligned_malloc()** requires the use of **_aligned_free()** when deallocating. On Linux*, Android*, and Mac OS* see documentation for **mem_align()** or **posix_memalign()**. Do not use **free()** on memory allocated with **_aligned_malloc().** Create the buffer and its associated **cl_mem** object using:

```
1 cl_mem myZeroCopyCLMemObj = clCreateBuffer(ctx,…CL_MEM_USE_HOST_PTR…);
```

A second case is *when the data will be generated on the device* but may be read back on the host. In this case leverage the **CL_MEM_ALLOC_HOST_PTR** flag to create the data. Do not worry about using the example code below to test that the memory is sized and allocated at a proper base address, the runtime will handle this for you.

A third case is when data is generated on the host but your application is in control of the initialization of the buffer. In this case you create the buffer, then initialize it. For example, suppose you are reading in input from a file. The difference between this case and the use of **CL_MEM_USE_HOST_PTR** is if the

buffer has already been populated. To initialize the contents of the buffer, use the OpenCL map and unmap API functions described later.

CL_MEM_ALLOC_HOST_PTR: Use this flag when you have not yet allocated the memory and want OpenCL to ensure that you have a zero copy buffer.

```
1 buf = clCreateBuffer(ctx, ….CL_MEM_ALLOC_HOST_PTR, ….)
```

Table 1.*Different application scenarios showing which flags to pass to* ***clCreateBuffer()*** *to enable the use of zero copy buffers on Intel® processor graphics*

Flag(s)	When to use the flag(s) to enable a zero copy scenario
CL_MEM_USE_HOST_PTR	• Buffer was already created in existing application code and the alignment and size rules were followed when the buffer was allocated, or you want control over the buffer allocation and do not want to rely on OpenCL. • In cases when you don't want to incur the cost of a copy that would take place with **CL_MEM_ALLOC_HOST_PTR | CL_MEM_COPY_HOST_PTR**. • In cases when data can be safely overwritten by OpenCL or you know the data will not be overwritten because your application controls any writes to the buffer.
CL_MEM_ALLOC_HOST_PTR	• You want the OpenCL runtime to handle the alignment and size requirements. • In cases when you may be reading data from a file or another I/O stream. • A brand new application being written to use OpenCL and not a port from existing code. • Buffer will be initialized in host or device code and not by a library decoupled from your control. • *Don't forget to map and unmap the buffer during initialization.*

Flag(s)	When to use the flag(s) to enable a zero copy scenario
CL_MEM_ALLOC_HOST_PTR | CL_MEM_COPY_HOST_PTR	• You want the OpenCL runtime to handle the size and alignment requirements. • In cases when you may be reading or writing data from a file or another I/O stream and aren't allowed to write to the buffer you are given. • Buffer is not already in a properly aligned and sized allocation and you want it to be. • You are okay with the performance cost of the copy relative to the length of time your application executes, for example at initialization. • Porting existing application code where you don't know if it has been aligned and sized properly. • The buffer used to create the OpenCL buffer needs the data to be unmodified and you want to write to the buffer

In summary, most cases can use **CL_MEM_ALLOC_HOST_PTR** on Intel processor graphics. Do not forget when initializing the buffer contents to first map the buffer, write to the buffer, and then unmap the buffer. In some cases the data may already be in an aligned and properly sized allocation, allowing you to use **CL_MEM_USE_HOST_PTR**.

This short function in C verifies that the pointer and size of the allocation adheres to the alignment and size rules:

```
01 unsigned int verifyZeroCopyPtr(void *ptr, unsigned int
   sizeOfContentsOfPtr)
02 {
03     int status; //so we only have one exit point from function
04     if((uintptr_t)ptr % 4096 == 0) //page alignment and cache
   alignment
05     {
06         if(sizeOfContentsOfPtr % 64 == 0) //multiple of cache size
07         {
08             status = 1;
09         }
10         else status = 0;
```

```
    }
    else status = 0;
    return status;
```

Accessing the Buffer on the Host

When directly accessing any buffer on the host, zero copy buffer or not, you are required to map and unmap the buffer in OpenCL 1.2. See below and the sample code for details.

Accessing the Buffer on the Device

Accessing the buffer on the device is no different than any other buffer; no code change is required. You only need to worry about the host-side interaction and the map and unmap APIs.

Use clEnqueueMapBuffer() and clEnqueueUnmapMemObject()

The APIs **clEnqueueReadBuffer()**, **clEnqueueWriteBuffer()**, and **clEnqueueCopyBuffer()** are not recommended, especially for large buffers since they require the contents of the buffer to be copied. Sometimes, however, these APIs might be beneficial, for example, if you are reading the contents out of the buffer because you want to reuse it immediately on the GPU in a double buffering scenario. In this case, it is useful to make a host-side copy and let the device continue to operate on the original buffer. To facilitate host read or write access to a memory buffer that has been shared with Intel processor graphics, use the APIs **clEnqueueMapBuffer()** and **clEnqueueUnmapMemObject()**.

Example use of **clEnqueueMapBuffer()**:

```
mappedBuffer = (float *)clEnqueueMapBuffer(queue, cl_mem_image,
CL_TRUE, CL_MAP_READ, 0, imageSize, 0, NULL, NULL, NULL);
```

Example use of **clEnqueueUnmapMemObject()**:

```
clEnqueueUnmapMemObject(queue, cl_mem_image, mappedBuffer, 0, NULL,
NULL);
```

Caveats on Other Platforms

The behavior described above may not be the same on all platforms. It is best to check the vendor's documentation. The OpenCL API specification provides a vendor the *ability* to create zero copy buffers. It does not *guarantee* to always return a zero copy buffer. In fact, the specification actually states that a copy may be created. In the documentation for **CL_MEM_USE_HOST_PTR**:

"OpenCL implementations are allowed to cache the buffer contents pointed to by host_ptr in device memory. This cached copy can be used when kernels are executed on a device."

An Observation about the Virtual Address Space

You may notice that the address you are validating to be on a 4096 byte page boundary is a virtual address boundary and not a physical address. Is this a problem? While in theory it could be an issue, after all the OS could have mapped a virtual address to any physical address, this does not happen in our implementation. You are assured that if the virtual address is page aligned, then the physical address is page aligned. More details on how and why this works is beyond the scope of this article.

Validating Zero Copy Behavior

One downside of the OpenCL 1.2 API is there is no runtime mechanism to validate that a copy has or has not occurred. For example, when a **clEnqueueMapBuffer()** executes, was a copy created or not? The way we verified these samples was to time a short program that declared the output buffer with **_aligned_malloc()** and compare the result when using **malloc()** with an image of size 1024x1024 over several iterations to verify that the aligned allocation was *significantly* faster from just before the map to

just after the unmap. These timings also include driver overhead. We have left the timing code in the sample if you want to verify this yourself.

A Zero Copy Example

We ported to OpenCL a well-known BSD licensed codebase that simulates ambient occlusion called AOBench available at: https://code.google.com/p/aobench/ (https://code.google.com/p/aobench/). We start with a straightforward mapping to OpenCL, the way any reasonable programmer would do as a first attempt. Next we show two different versions where we create the computed ambient occlusion resultant image as a zero copy buffer. The first shows the code when you want to allocate the buffer and pass this buffer to the OpenCL runtime. This might be common when you already have an existing application that makes use of **CL_MEM_USE_HOST_PTR**. The second is further simplified and is useful when you want the OpenCL runtime to handle the buffer allocation and uses **CL_MEM_ALLOC_HOST_PTR**. We have left it up to you to implement the third possibility: a short, but useful, exercise to create the buffer using **CL_MEM_ALLOC_HOST_PTR**, map the pointer, populate the buffer on the host, then unmap the pointer. We focus on the output image buffer for this example; other buffers could be treated similarly.

Sample File and Directory Structure

This section contains details on how this code is partitioned. The emphasis was to create a simple C example and not a product quality implementation. Most of the code is common across all of the samples.

Source Files:

- **Common/host_common.cpp**: Boilerplate to start up and manage an OpenCL context, compile the source, create queues, and handle general cleanup.
- **Common/kernels.cl**: OpenCL kernel code for this particular example. The contents of this file are of no significance other than demonstrating a complete application.
- **Include/host_common.h:** header file for **host_common.cpp**, various initialization values, and function and variable declarations.

- **Include/scene.h**: scene graph functions and variables used in this sample.

- **NotZeroCopy/main.c**: source file that contains the functions for doing a straightforward port of the sample code. The functions we are interested in are **initializeDeviceData()** and **runClKernels()**. In **initializeDeviceData()** we used a standard **malloc()** without forcing an alignment, resulting in a surface that will not support zero copy. Also, in **runCLKernels**() we used the standard API call **clEnqueueReadBuffer()**. Functionally, this is 100% correct, however it is not optimal for performance.
- **ZeroCopyUseHostPtr/main.c**: source file that contains functions that demonstrate modifications to NotZeroCopy. Notice the use of **_aligned_malloc()** instead of **malloc()** from **NotZeroCopy** in **initializeDeviceData()**. Also, when we call **clCreateBuffer()** we use the flag **CL_MEM_USE_HOST_PTR**.
- **ZeroCopyAllocHostPtr/main.c**: source file that contains functions that demonstrate the modifications required when using the allocation mechanism of the runtime. Specifically, notice the fact that now we do not even need to call **_aligned_malloc()** as we did in the **ZeroCopyUseHostPtr** example. Instead, we simply pass the flag **CL_MEM_ALLOC_HOST_PTR** and pass the size of the buffer we want to allocate.

The other files and directories are generated automatically by the Microsoft Visual Studio* IDE.

Microsoft Visual Studio 2012 Configuration:

The Microsoft Visual Studio 2012 solution, **OpenCLZeroCopy.sln**, has three projects: **NotZeroCopy.vcxproj**, **ZeroCopyAllocHostPtr.vcxproj**, and **ZeroCopyUsedHostPtr.vcxproj**. The **OpenCLZeroCopy.props** property sheet holds the settings specific to this project. These settings include: the system path to the **cl.h** header file, the pointer to the **OpenCL.lib** library to link to, and the pointer to the local include directory for this example. You may need to change these settings for your build environment.

Building and Running the Examples

Build Requirements

First, make sure you have downloaded and installed the Intel® SDK for OpenCL™ Applications available here: https://software.intel.com/en-us/vcsource/tools/opencl-sdk (https://software.intel.com/en-us/vcsource/tools/opencl-sdk). Also, be sure to install Microsoft Visual Studio 2012 (MSVC 2012) IDE. Next, open the solution file **OpenCLZeroCopy.sln** in MSVC 2012.

Paths in the Property Sheet

Instead of making changes for each build of each executable, MSVC supports the use of property sheets. If you change a property sheet, the change propagates to all builds that include this property sheet in their build settings. We have included figures here that show the path names on our system if you decide you need to change them. Alternatively, you can use the environment variable: $(INTELOCLSDKROOT), which defaults to C:\Program Files (x86)\Intel\OpenCL SDK\3.0\. You may have a more recent version. We have a relative path for the include files used across each of the executables and use the default installation location for the Intel OpenCL SDK. For more information on property sheets consult the MSDN documentation: http://msdn.microsoft.com/en-us/library/z1f703z5(v=vs.90).aspx (http://msdn.microsoft.com/en-us/library/z1f703z5(v=vs.90).aspx) .

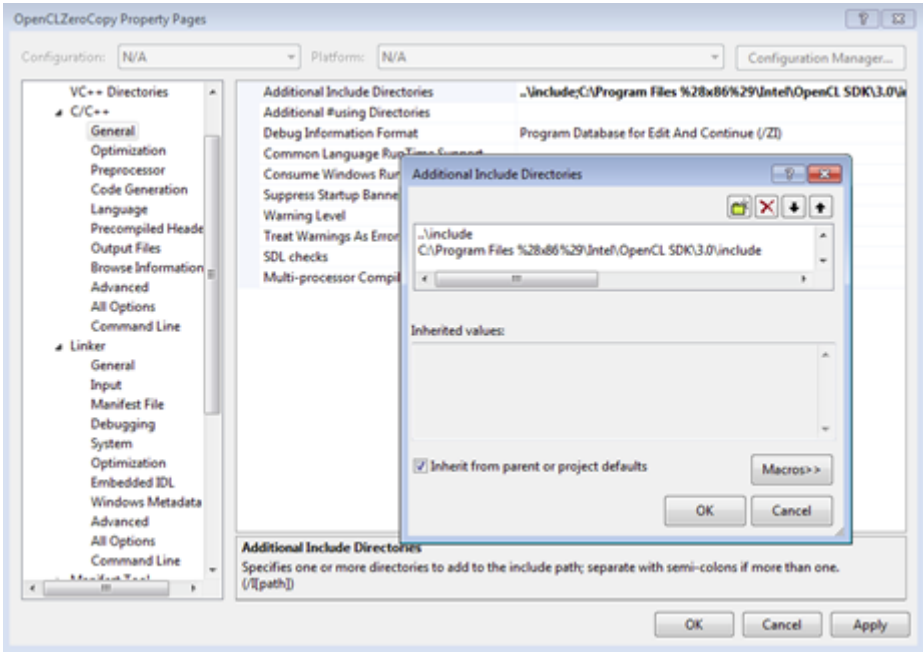

Figure 2.*Additional include directories used in this code sample.*

You can access the property manager from the *View* menu and select the *Property Manager* Menu item.

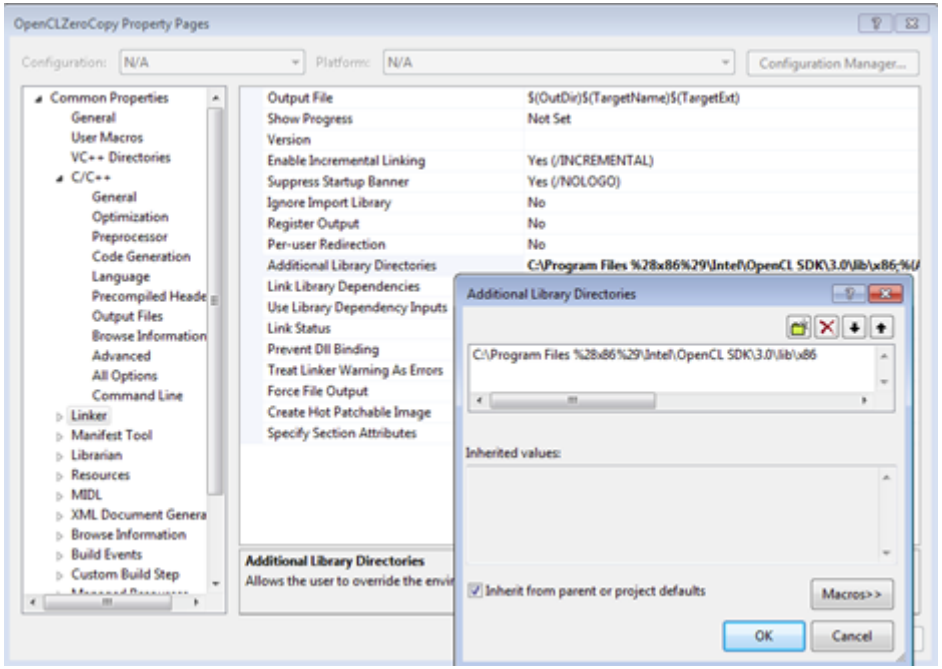

Figure 3.*Additional library directories used in this code sample.*

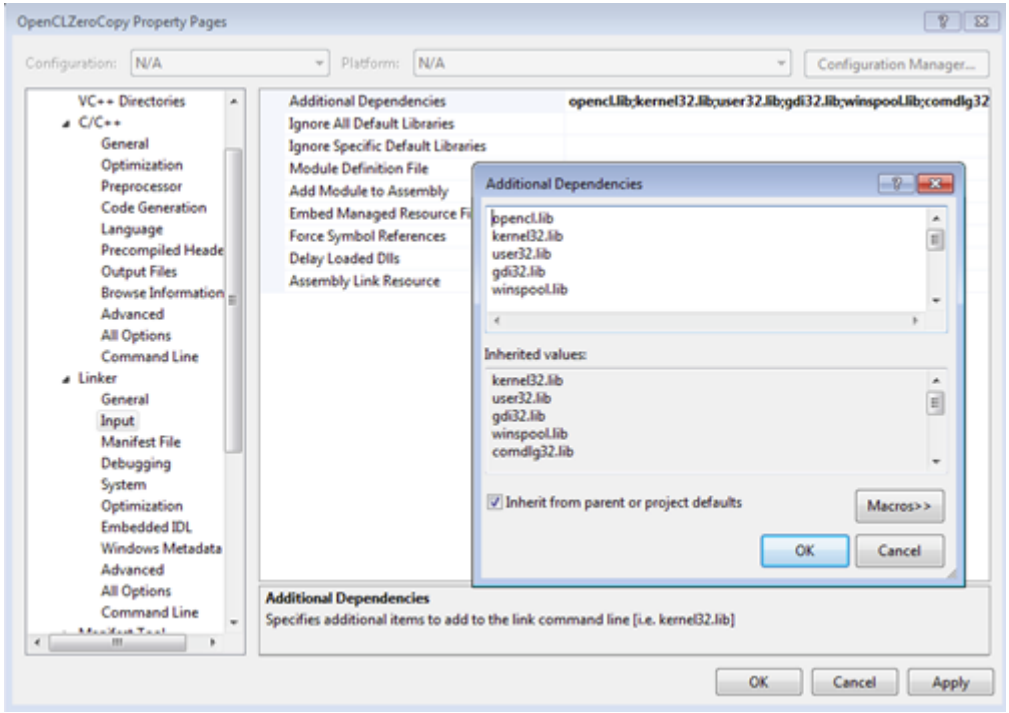

Figure 4.*Notice the opencl.lib file is added as an additional library.*

Running the Example

Build this sample code by selecting *Build->Build Solution* from the main menu. All of the executables should be generated. You can run them within Visual Studio directly or go to the Debug and/or Release directories that are located in the same location as the **OpenCLZeroCopy** solution file.

What's Coming in OpenCL 2.0: Shared Virtual Memory (SVM)

This paper has focused on understanding the use of buffers that can be shared on platforms that support shared physical memory (SPM) such as the Intel CPUs and Intel processor graphics. OpenCL 2.0 will have APIs to expose shared virtual memory on architectures that can support it. This will allow you to not just have a shared buffer for writing, but also to share virtual addresses on the CPU and GPU. For example, you could leverage SVM to update a scene graph on the CPU using a physics simulation then use the GPU to calculate the final image.

Acknowledgements

Lots of folks have encouraged the development of this and other collateral. Some provided feedback and some helped create the space to get it done: Stephen Junkins, Murali Sundaresan, David Blythe, Aaron Kunze, Allen Hux, Mike Macpherson, Pavan Lanka, Girish Ravunnikutty, Ben Ashbaugh, Sergey Lyalin, Maxim Shevstov, Arnon Peleg, Vadim Kartoshkin, Deepti Joshi, Uri Levy, and Shiri Manor.

References

1. OpenCL 1.2 specification: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf)
2. OpenCL 2.0 specification, composed of three books: the OpenCL C Language specification, the OpenCL Runtime API, and the OpenCL extensions: https://www.khronos.org/registry/cl/specs/ (https://www.khronos.org/registry/cl/specs/)
3. AOBench: https://code.google.com/p/aobench/ (https://code.google.com/p/aobench/)
4. Stephen Junkins' whitepaper: Intel® Gen 7.5 Compute Architecture: https://software.intel.com/sites/default/files/managed/f3/13/Compute_Architecture_of_Intel_Processor_Graphics_Gen7dot5_Aug2014.pdf (https://software.intel.com/sites/default/files/managed/f3/13/Compute_Architecture_of_Intel_Processor_Graphics_Gen7dot5_Aug2014.pdf). A must-read for anybody using OpenCL on Intel Processor Graphics platforms.

About the Author

Adam Lake – Adam works in the Visual Products Group as a Senior Graphics Architect and Voting Representative to the Khronos OpenCL Standards Body. He has worked on GPGPU programming for 12+ years. Previously he has worked in VR, 3D, graphics, and stream programming language compilers.

* Other names and brands may be claimed as the property of others.
OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission from Khronos.
Copyright © 2014 Intel Corporation. All rights reserved.

For more complete information about compiler optimizations, see our Optimization Notice (/en-us/articles/optimization-notice#opt-en).