White Paper

OpenCL* and OpenGL* Sharing

This white paper is the fifth in a series of papers describing how to best utilize underlying Intel® processors and Intel integrated graphics using OpenCL*. Please refer to the OpenCL Optimization Guide and other up-to-date information at http://www.intel.com/software/opencl.

The 3rd generation Intel® Core™ processor family now supports OpenCL* 1.1 for both the CPU and Intel® HD Graphics. The Intel® SDK for OpenCL* Applications provides developers with the ability for the first time to use the OpenCL standard to utilize the compute resources of both the CPU and Processor Graphics. OpenCL gives developers programming on Intel platforms another choice and OpenCL provides unique interoperability with respect to sharing buffers with other graphics APIs like DirectX*, OpenGL*, and Intel® Media SDK.

This article focuses on the interoperability between OpenCL and OpenGL, now available on the 3rd gen Intel Core processors.

Developers should carefully evaluate their choice of API and extensions based on the task at hand and the hardware features exposed by APIs that programmer is inclined to use. Tasks such as video encoding, decoding and transcoding are best handled by the Intel Media SDK as it can utilize dedicated hardware that is not yet exposed to OpenCL APIs. If you are interested in zero copy OpenGL sharing then shared context with another device CPU is not supported. Please refer to limitation of shared context for further details.

OpenCL or OpenGL Shading Language (GLSL)

Programmers have choice to perform computations on GPU using GLSL or OpenCL kernels. OpenCL has some advantages over GLSL. OpenCL kernels operate on workgroups whereas GLSL is at the per-pixel level. Using OpenCL, programmers can take advantage of local memory, constant memory, and numerous built-in math functions, atomic operations, thread synchronization APIs that are not available to shaders. OpenCL kernels can reuse data from one stage to the next using local memory (to avoid bank conflicts, see Recommendations section of Optimizing OpenCL* Usage (Intel® Processor Graphics) of Optimization Guide). Programmers also can synchronize kernels and use events to process work across devices (CPU and processor graphics).

GLSL may be a good fit for applications where shaders (small code operating on pixels) are running on vertices, fragments and end result is to display image (as GLSL fits well with graphics pipeline and can utilize processor graphics hardware better). OpenCL is comparatively new and the OpenCL compiler is still evolving, whereas GLSL compilers have been around for a while. You can use the Intel® Graphics Performance Analyzers and Intel® VTune™ Amplifier XE to determine which would work best for your code.

Using OpenCL along with GLSL is another option to consider. Create/Pass data using Vertex Buffer Objects /Texture Buffers or frame buffers to the GPU using OCL kernels. Then draw avoiding data copy from/to host memory to/from the GPU by utilizing OpenCL/OpenGL interoperability. Careful balancing of work assigned to various devices is required. Programmers should not starve or overload a device as there are now so many ways to divide work between the CPU and GPU(s).

Anatomy of an OpenCL and OpenGL Application

The architecture of the OpenCL/OpenGL application depends on the underlying OS and toolkits/threading models used. OpenGL is a state-based API, so making OpenGL calls across various threads requires device contexts and render context to be current for OpenCL APIs to allow shared context creation. This makes the host-side application architecture more like a producer/consumer model. For animation/physics type applications, use OpenCL (GPU + CPU) to create data (producer) and then render it using OpenGL (consumer) and for an image post-processing type application, share OpenGL rendered frame buffers (producer) with OpenCL kernels to apply effects (consumer).

Graphics device resources are best utilized if OpenGL commands are issued using single thread context for rendering. OpenCL allows creating shared context between the CPU and the GPU so use all available resources to create data to render while making sure that the graphics device has enough bandwidth left to do other tasks as execution units are also being utilized by other applications and there is no pre-emptive multitasking available on GPUs.

The scheme shown in Figure 1 is better if you decide to distribute data creation work among devices and then draw using OpenCLshared context using the Intel Processor Graphics. Use the scheme shown in Figure 2 if you think the Intel Processor Graphics is underutilized as this scheme saves the cost of transferring data from main memory to the Intel Processor Graphics.

If you are implementing effects using OpenCL post processing, use the Figure 2 scheme so you can simply run kernels right after OpenGL is done producing frame buffers without having to copy data back to host memory. If your program uses shaders for post processing, consider converting them to OpenCL kernels as unlike shaders running on each pixel, OpenCL kernels operate on multiple pixels in each work item.

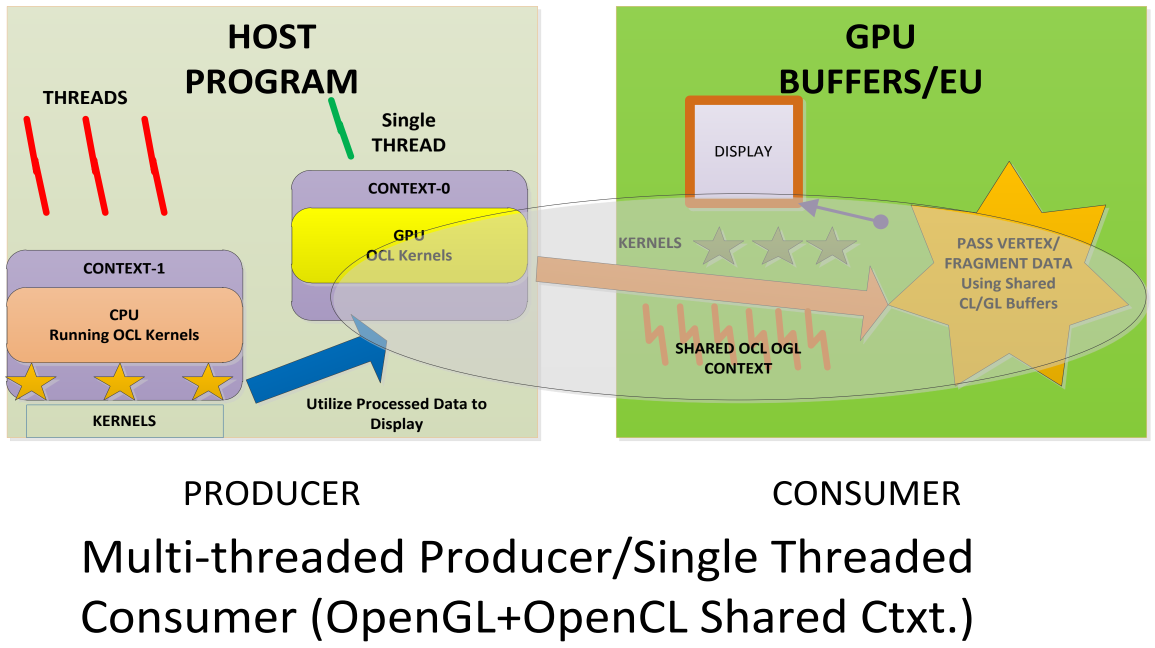
HOST PROGRAM
THREADS
Single THREAD
CONTEXT-0
GPU OCL Kernels
CONTEXT-1
CPU Running OCL Kernels
KERNELS
Utilize Processed Data to Display
GPU BUFFERS/EU
DISPLAY
KERNELS
PASS VERTEX/ FRAGMENT DATA Using Shared CL/GL Buffers
SHARED OCL OGL CONTEXT
PRODUCER
CONSUMER
Multi-threaded Producer/Single Threaded Consumer (OpenGL+OpenCL Shared Ctxt.)

Figure 1. *CPU Produces Data for GPU to Display*

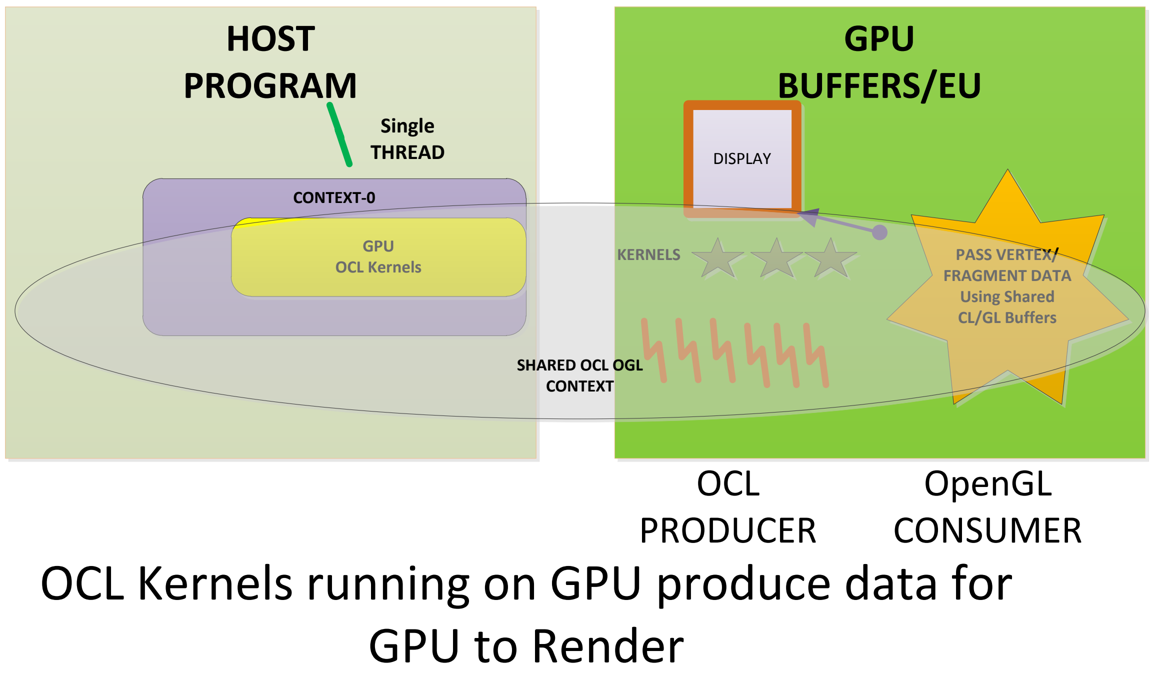

Figure 2. *OCL Kernels on GPU Produce and Consume Data*

General Graphics Device Considerations for Performance

Optimizing the work group size for the processor graphics is very important. Please refer to the section Work-group size recommendations summary in the OpenCL Optimization Guide for details.

Do not oversubscribe the GPU because if it gets bogged down, users will think the machine is unresponsive.

Use events to synchronize work and profiles using OpenCL profiling events often to detect bottlenecks.

OpenGL on Windows*, Linux*, and OSX (Multithreading Issues)

Refer to pertinent OS documentation on restrictions imposed by the underlying implementation. OpenGL is a state-based API and keeping the OpenGL context current for a given thread requires extra effort. Profile your application to detect if work is CPU bound or GPU bound to see if implementing parallelism on the CPU will be of any benefit or not.

If the application can benefit from rendering multiple scenes in parallel, use multiple GL contexts, one for each scene.

You may also like to explore GLUT APIs, GLEW APIs, and other bundled APIs, such as Qt, etc. to ease your development. See the OpenGL website for OS-related OpenGL implementation details.

OpenGL and OpenCL Synchronization BKMs

Since Acquire/Release are used for synchronization between OpenCL and OpenGL APIs, it is better to release OpenCL resources BEFORE OpenGL resources as OpenCL resources are created FROM OpenGL objects.

Try not to use multiple threads on the GPU while rendering/drawing using OpenGL. Keeping states coherent requires extra calls to OpenGL, i.e., create a render context or get the current context in the running thread as OpenGL restricts each context to a single thread.

glFinish and clFinish wait until all previously submitted OpenGL commands and OpenCL commands are done. If you have just one thread issuing these commands, control will return to your application after commands are finished as they are not really very fine grained commands. Maintaining an optimal control over responsiveness while keeping devices busy is not easy and requires multiple experiments with threading vs. amount of work submitted before glFinish/clFinish.

Use double buffering to manage memory resources better. By using double buffering, the CPU can create data while the GPU is rendering a previously submitted frame. If the GPU seems to be keeping up and can do more work, employ triple buffering to keep the producer and consumer busy or move some more data creation work to the GPU eliminating some data copying from host memory to the GPU.

Do not use immediate mode, use VBOs (Vertex Buffer Objects), OpenGL commands, such as glDrawArrays, glDrawElements, etc., as these can handle large buffers containing multiple geometric primitives resulting in fewer OpenGL calls. It is better to arrange all pertinent data for a given vertex, such as position, color, index, etc., together in VBOs (interleaved).

For Texture data, make sure the chosen format is supported by OpenCL and OpenGL drivers and doesn't require any conversions. Use double buffering to avoid texture data resource conflicts between the CPU and the GPU.

Conclusions

OpenCL and OpenGL sharing is a great way to design your application to take advantage of the simplicity of the OpenCL programming language while maximizing data on Intel Processor Graphics.

If your application requires complex long kernels, you can run these kernels on the CPU and the GPU using separate contexts and then use another context to transfer data to the GPU for drawing using OpenGL APIs.

OpenCL kernels are a lot more powerful with support for built-in math functions, atomics, and barriers and also expose underlying Intel Processor Graphics memory hierarchy better than GLSL kernels.

You can download the Intel® SDK for OpenCL Application 2012 at
www.intel.com/software/opencl

About the Author

Vinay Awasthi works as an Application Engineer for the Apple* Enabling Team at Intel at Santa Clara. Vinay has a Master's Degree in Chemical Engineering from the Indian Institute of Technology, Kanpur. Vinay enjoys mountain biking and scuba diving in his free time**.**

Notices

Optimization Notice