Qualcomm
TECHNOLOGIES, INC

Snapdragon™ OpenCL General Programming and Optimization

80-N8592-1 L

August 29, 2014

Submit technical questions at:
https://support.cdmatech.com/

Confidential and Proprietary – Qualcomm Technologies, Inc.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures

Tables

Revision history

Revision	Date	Description
A	Dec 2011	Initial release
B	Aug 2012	Updated to include GPU and CPU guidelines
C	Oct 2012	Updated GPU and CPU guidelines
D	Aug 2013	Updated optimization guidelines
E	Nov 2013	Updated Section 2.3.2 and Section 3.1.4.1.2
F	Dec 2013	Numerous changes were made to this document; it should be read in its entirety.
G	Jan 2014	Updated Table 3-1
H	Jan 2014	Updated Section 4.1 and Chapter 6
J	Feb 2014	Updated Table 3-1, Section 5.5, Section 5.5.1, and Chapter 6
K	Jun 2014	Added Section 3.4, updated Table 6-1, added Section 6.2 and Section 6.3
L	Aug 2014	Updated Table 1-1 and Sections 1.1 and 3.3 (merged document with 80-ND791-8)

Note: There is no Rev. I, O, Q, S, X, or Z per Mil. standards.

1 Introduction

1.1 Purpose

NOTE: Numerous changes were made to this document; it should be read in its entirety.

This document provides guidelines for developing and optimizing OpenCL apps for Snapdragon™ 400-, 600-, and 800-based mobile platforms that include the Adreno™ 300 and 400 series Graphics Processing Units (GPUs). The Adreno 300 series GPUs, or Adreno 3xx, include, e.g., the Adreno 320 GPU, which is embedded within the Snapdragon™ 600 (8064T) processors.

This document provides guidelines on how to develop and effectively optimize OpenCL apps that target the Adreno 3xx GPU with references pointing to the Snapdragon 600 platform as an example. For more information on optimization techniques to enhance performance of the OpenCL app on Snapdragon, see [S1].

This document is intended for OEMs and ISVs who may choose to adapt their apps to the Qualcomm Technologies, Inc. (QTI) Snapdragon platform.

1.2 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

Code variables appear in angle brackets, e.g., `<number>`.

Commands to be entered appear in a different font, e.g., **`copy a:*.* b:`**.

Shading indicates content that has been added or changed in this revision of the document.

1.3 References

Reference documents are listed in Table 1-1. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Table 1-1 Reference documents and standards

Ref.	Document	
Qualcomm Technologies		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1
Q2	*Presentation: Introduction to Adreno Profiler for OpenCL*	80-ND791-5
Q3	*Application Note: CL-GL Interop Usage Guidelines*	80-NK985-1

Ref.	Document	
Standards		
S1	*The OpenCL Specification*	The Khronos Group Inc. Versions 1.1, 1.2, and 2.0 http://www.khronos.org/opencl/
Resources		
R1	*OpenCL Programming Guide*, Addison-Wesley Publishing Company Chapter 7, "Display Lists" Chapter 8, "Drawing Pixels, Bitmaps, Fonts, and Images"	http://ube.ege.edu.tr/~ozturk/graphics/opengl_book/ch7.htm and http://ube.ege.edu.tr/~ozturk/graphics/opengl_book/ch8.htm

1.4 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://support.cdmatech.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

1.5 Acronyms

For definitions of terms and abbreviations, see [Q1].

2 Snapdragon Architecture

2.1 Snapdragon overview

Snapdragon is one of the most powerful and widely used processors in today's Android™ and Windows smartphones and tablets. Snapdragon processors bring together the best-in-class mobile components on a single chip, ensuring that Snapdragon-based devices deliver the latest mobile user experiences in an extremely power-efficient, integrated solution.

Snapdragon is a multiprocessor system that includes components such as a multimode modem, CPU, GPU, DSP, location/GPS, multimedia, power management, RF, optimizations to software and operating systems, memory, connectivity (Wi-Fi, Bluetooth®), etc.

For a list of current commercial devices that include Snapdragon processors and to learn more about Snapdragon processors, go to http://www.qualcomm.com/snapdragon/devices.

2.1.1 OpenCL on Snapdragon

This section focuses on programming the GPU of the Snapdragon processor for general-purpose data parallel computation using OpenCL.

OpenCL on GPU is supported on the Adreno 300, 400 series GPU, and is fully conformant to the OpenCL standard. Conformance certification can be found at www.khronos.org, or contact your CE representative for further details.

2.2 Platform architecture GPU

The Adreno GPU in Snapdragon processors is generally used for rendering graphical apps, but it is also a powerful general-purpose SIMD processor capable of processing many simultaneous threads at once. The capabilities of the GPU can be leveraged using OpenCL to perform data-parallel computations.

There is little difference in how an app utilizes the OpenCL API to address the compute capabilities of the GPU and the CPU. All OpenCL API calls execute on the host (CPU), and a command queue is used to communicate work for the device to perform. However, there are significant differences in the architecture of the GPU that impact how the workloads should be broken up.

When both graphics and OpenCL capabilities of the GPU are used, the GPU is timeshared by these two functionalities. It switches between the compute and graphics context at job boundaries, in a cooperative multitasking fashion. If a graphics shader is executed and a compute kernel is pending execution, the graphics shader must complete before the GPU executes the compute kernel. Likewise, if a compute kernel is executed and a graphics shader is pending execution, the compute kernel must complete before the GPU starts executing the next graphics shader. Therefore, to keep the graphics responsive, it is recommended that long-running compute kernels are avoided. A reasonable upper bound on compute kernel execution time would be tens of milliseconds.

3 Software Developer Guides

The OpenCL implementation exposes OpenCL-GPU, in which OpenCL kernels execute on the Adreno GPU. Figure 3-1 shows the OpenCL architecture.

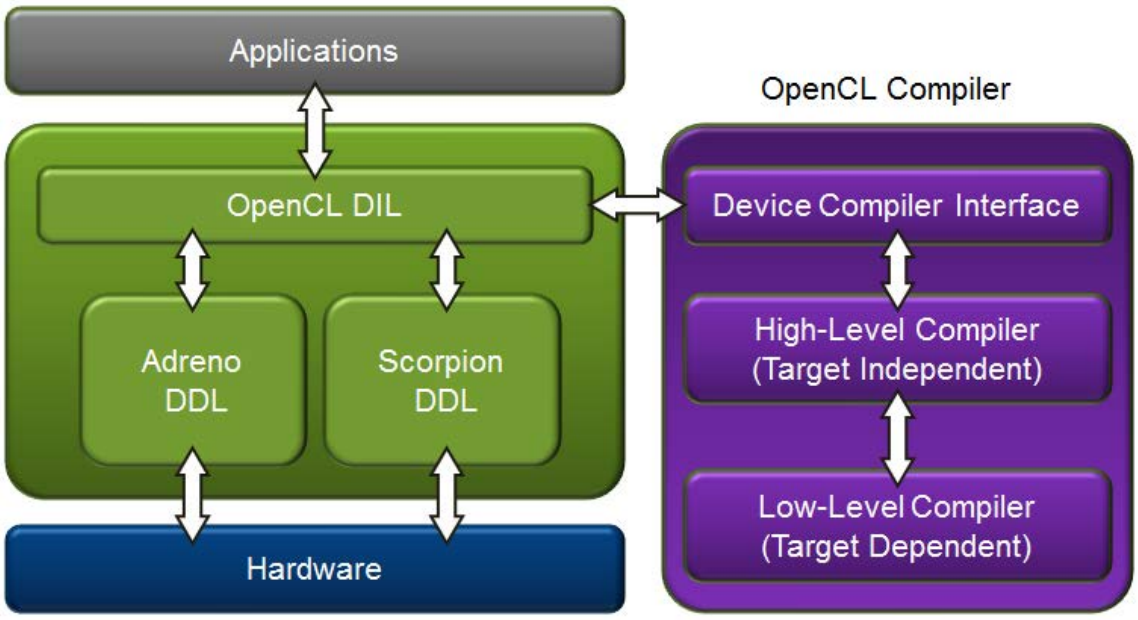


Figure 3-1 OpenCL architecture

This chapter also provides guidelines on optimal use of OpenCL-GPU and interoperation between CPU and GPU.

3.1 OpenCL – GPU extensions

In addition to supporting the baseline of OpenCL 1.1, 1.2 embedded profile per [S1], the Snapdragon OpenCL implementation supports several additional features, which are supported through extensions. Table 3-1 lists available OpenCL extensions on Snapdragon chipsets for the GPU.

Per the OpenCL standard, users should query chipsets with the clGetDeviceInfo API to retrieve the list of extensions supported by each device.

Table 3-1 OpenCL extensions

Extension	Description	Adreno GPU
cl_khr_global_int32_base_ atomics	Atomic operations on 32-bit integers in global memory (add, sub, xchg, inc, dec, cmpxchg)	Supported

Extension	Description	Adreno GPU
cl_khr_global_int32_ extended_atomics	Additional atomic operations on 32-bit integers in global memory (min/max/and/or/xor)	Supported
cl_khr_local_int32_base_ atomics	Atomic operations on 32-bit integers in local memory (add, sub, xchg, inc, dec, cmpxchg)	Supported
cl_khr_local_int32_extended_ atomics	Additional atomic operations on 32-bit integers in global local (min/max/and/or/xor)	Supported
cl_khr_byte_addressable_ store	Allows writing to char and short-based types	Supported
cles_khr_int64	Support for 64-bit integers (long, ulong, long*n*, ulong*n*)	Not supported
cl_khr_fp16	Support for fp16 half data types (half, half2, half4, half8, half16)	Supported
cl_khr_gl_sharing	Shares OpenCL image data with texture and buffer objects	Supported
cl_qcom_ion_host_ptr	Augments the functionality provided by clCreateBuffer, clCreateImage2D, clCreateImage3D allowing apps to specify a new flag, CL_MEM_ION_HOST_PTR_QCOM; this new flag maps memory pointed to by Ion host_ptr to the GPU address space; hence, usage of this flag prevents a copy of the host memory to the device and vice versa	Supported
cl_qcom_limited_printf	Introduces the printf built-in function to OpenCL kernels on the earlier release of CL 1.1; this functionality is part of the CL 1.2 specification on the newer release	Supported
Cl_khr_egl_event	Allows creating OpenCL event objects linked to EGL fence sync objects, potentially improving efficiency of sharing images and buffers between the two APIs	Supported
EGL_IMG_image_plane_ attribs	Allows creating an EGL image from a single plane of a multiplanar Android native image buffer	Supported
EGL_KHR_cl_event	Allows creating an EGL fence sync object linked to an OpenCL event object, potentially improving efficiency of sharing images between the two APIs	Supported
Cl_qcom_extended_images	Allows app to allocate 2D and 3D images up to a maximum of 8k x 8k size images	Supported (only for Adreno A3x products)

3.2 Interoperability guidelines

3.2.1 Stability when using compute and graphics interoperability

The Adreno GPU is normally used to render graphics. If graphics rendering is in progress, launching OpenCL kernels on the GPU forces timesharing the GPU with the graphics process. To ensure undefined behavior, it is important to note the following.

Limited execution time on GPU

To prevent rouge kernels from locking up the GPU, the GPU is reset if the OpenCL kernel execution time exceeds the default timeout value set by QTI. There are generally two factors that determine the kernel execution time:

- Nature of the kernel, e.g., long loops or use of built-ins
- Size of the NDRange

For the purpose of debug only, QTI provides a mechanism to alter this default timeout so that customers can quickly isolate rogue kernels. The timeout value can be changed via the debug file system in the kernel. This can be done by echoing the value to /data/debugfs/kgsl/kgsl-3d0/wait_timeout.

The hash script below shows how to mount and set this value.

```
if  [[ `adb shell ls /data/debugfs` =~ "No such file or directory" ]]; then
     echo "Creating /data/debugfs"
     adb shell mkdir /data/debugfs
 fi
 if  [[ `adb -s $dev shell ls  /data/debugfs/kgsl/kgsl-3d0/wait_timeout`
=~ "No such file or directory" ]]; then
      echo "Mounting debugfs"
      adb -s $dev shell "mount -t debugfs debugfs /data/debugfs"
 fi
 printf "dev:%-10s""Setting kgsl timeout to $timeout on device\n" $dev
 adb -s $dev shell "echo $timeout >
                              /data/debugfs/kgsl/kgsl3d0/wait_timeout"
```

Multithreaded apps that use CL-GL interop extensions

To obtain detailed guidelines on using CL-GL interop extensions, see [Q3].

3.3 Performance optimization

Performance optimization is achieved at three levels to allow the most efficient usage of memory bandwidth, on-chip memory, and GPU general-purpose registers, which result in optimal performance for the Snapdragon platform.

The three levels include:

- Algorithm level
- API level
- Kernel level

3.3.1 Algorithm

Before porting to OpenCL, gain a thorough understanding of the algorithm. A thorough understanding of the application pipeline and analysis of data flow dependency helps determine where bottlenecks occur. Accordingly, you can customize code to minimize or hide bottlenecks. A few examples of how to achieve this are:

- Split kernels if they allow better data parallelization since data parallelization is the key to any Single Instruction Multiple Data (SIMD) architecture.
- Check data types in each stage of the application pipeline and make sure that the data type that is used is consistent across the entire pipeline.
- Use shorter data types as much as possible to reduce memory fetch/bandwidth and increase the number of Arithmetic Logic Units (ALU) available for execution.
- Merge kernels if the memory bandwidth can be reduced; e.g., if two subsequent kernels need the same source memory objects, it would make sense to read the memory object only once within a single kernel.
- Understand if a kernel is memory bound or ALU bound; this can also be achieved by using the OpenCL scrubber metrics in the Adreno Profiler.
- If an application uses both OpenGL and OpenCL on a GPU, there are some guidelines that must be followed to allow for optimal utilization of the GPU, thereby maximizing performance. See [Q3] for additional information.
- It is crucial to understand how the waves (threads) are scheduled and executed on the GPU; see Figure 3-2.
  - Assuming that the application has a work group size of 48 with a maximum of 8 items allowed per wave, then there are 6 waves that are executed in this work group for a device with 4 Shader Processors (SP). Each SP executes 2 waves simultaneously, which makes a total of 16 work items being executed simultaneously in a single SP.
  - If the number of ALU units is doubled per SP, the wave size increases from 8 to 16.

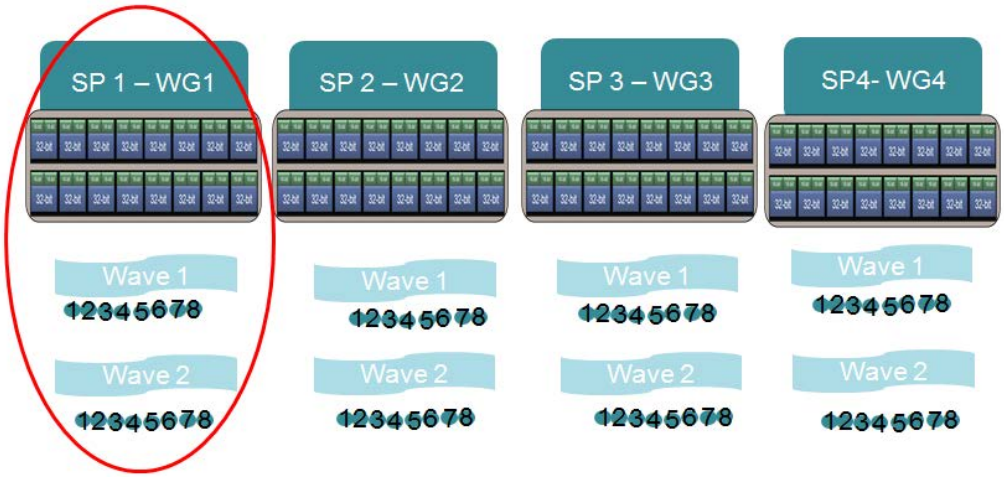


Figure 3-2 Scheduled and executed waves on a device with 4 SP with 16 32-bit ALUs

- Switching from one wave to the next occurs when there is a data dependency (read/write) that must be completed before proceeding. The Adreno processor hides data fetch/write latencies by switching from one wave to the next within the same work group.

For an example, see the "Improve algorithm" example in Section 3.3.4.1.

NOTE: The number of SPs/ALUs shown above is an arbitrary example for illustration only and does not represent any specific Adreno.

3.3.2 API

3.3.2.1 OpenCL memory mode

Table 3-2 OpenCL memory model

Memory	Definition	Latency	Note
Local	Shared by all work items in a work group	Medium	GPU hardware
Constant	Constant for all work items in a work group	Low when in GPU; high when in system memory	GPU or system memory
Private	Private to a work item	Based on where the memory is allocated by the compiler	GPU and/or system memory
Global	Accessible by all work items in all work groups	High	System memory

Because the constant, local, and private memory resides on chip memory, it is recommended in cases where there is frequent or multiple access of the same data element that these memory locations be made use of as much as possible. These memory regions have size limitations; therefore, if a larger chunk of data is declared, part or all of that chunk is stored in global memory. These on-chip memory regions can be used by using the address space qualifiers __constant, __local, and __private. For additional details, see Section 6.5 in [S1].

Figure 3-3 shows the physical location of OpenCL memory types in Adreno 320 and the Snapdragon 600 chipset.

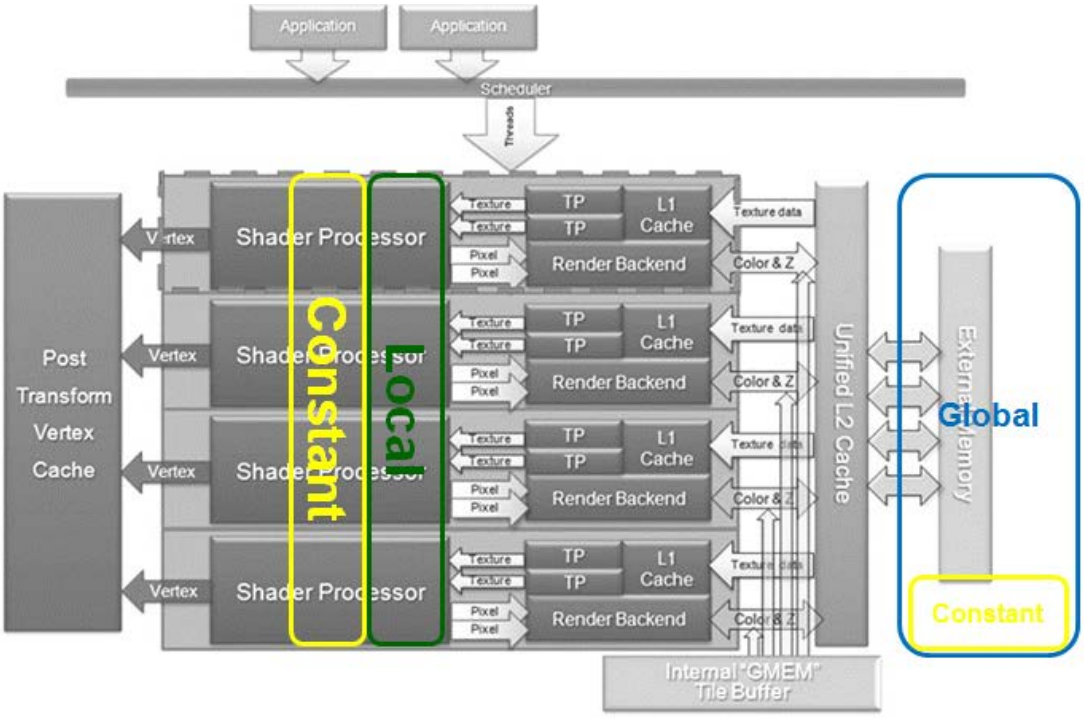


Figure 3-3 Physical location of OpenCL memory types in Adreno

3.3.2.2 Memory objects

There are two types of memory objects in OpenCL, buffer and image.

Buffer

A buffer object stores a one-dimensional collection of elements. Elements of a buffer object can be a scalar data type (such as an int, float), vector data type, or a user-defined structure. A buffer object is created using the following function:

```
cl_mem clCreateBuffer (cl_context context,
     cl_mem_flags flags,
     size_t size,
     void *host_ptr,
     cl_int *errcode_ret)
```

The buffer objects are stored in global memory and accessed via the L2 cache.

Image

An image object is used to store a one-, two-, or three-dimensional texture, frame buffer, or image. The elements of an image object are selected from a list of predefined image formats. A 2D image object is created using the following function:

```
cl_mem clCreateImage2D (cl_context context,
     cl_mem_flags flags,
     const cl_image_format *image_format,
     size_t image_width,
     size_t image_height,
     size_t image_row_pitch,
     void *host_ptr,
     cl_int *errcode_ret)
```

Image objects are accessed via the hardware texture pipeline via the L1 cache.

Image formats supported on Adreno 3xx series are:

```
CL_RGBA, CL_UNORM_INT8,
CL_RGBA, CL_UNORM_INT16,
CL_RGBA, CL_SIGNED_INT8,
CL_RGBA, CL_SIGNED_INT16,
CL_RGBA, CL_SIGNED_INT32,
CL_RGBA, CL_UNSIGNED_INT8,
CL_RGBA, CL_UNSIGNED_INT16,
CL_RGBA, CL_UNSIGNED_INT32,
CL_RGBA, CL_HALF_FLOAT,
CL_RGBA, CL_FLOAT,
CL_BGRA, CL_UNORM_INT8,
CL_BGRA, CL_UNORM_INT16,
CL_BGRA, CL_SIGNED_INT8,
CL_BGRA, CL_SIGNED_INT16,
CL_BGRA, CL_SIGNED_INT32,
CL_BGRA, CL_UNSIGNED_INT8,
CL_BGRA, CL_UNSIGNED_INT16,
CL_BGRA, CL_UNSIGNED_INT32,
CL_BGRA, CL_HALF_FLOAT,
CL_BGRA, CL_FLOAT,
CL_RG, CL_SIGNED_INT8,
CL_RG, CL_UNSIGNED_INT8,
CL_R, CL_UNORM_INT16,
CL_R, CL_SIGNED_INT16,
CL_R, CL_SIGNED_INT32,
CL_R, CL_UNSIGNED_INT16,
CL_R, CL_UNSIGNED_INT32,
```

```
CL_R, CL_HALF_FLOAT,
CL_R, CL_FLOAT,
```

Advantages of using an image over a buffer object

- Leverage the built-in texture engine

  OpenCL supports two filters, CLK_FILTER_NEAREST and CLK_FILTER_LINEAR. For CLK_FILTER_LINEAR, the proper combination of image type allows the GPU to do automatic linear interpolation using its built-in texture engine.

  For example, assume an image is type CLK_NORMALIZED_COORDS_TRUE and CL_UNORM_INT16, i.e., image date is 2-byte unsigned short. Function call Read_imagef():

  - Reads the pixels
  - Interpolates
  - Converts and normalizes it to the range of [0, 1]

  This is very convenient if you want to do 2D/3D bi/tri-linear interpolation.

- Leverage usage of L1 cache
- Better handling of boundary conditions
- Supports numerous image format/data type combinations listed under “Image” in Section 3.3.2.2.

For an example, see “Utilize texture engine for buffer” in Section 3.3.4.1.

3.3.2.3 Critical memory optimizations to consider

The Adreno system consists of shared memory, which can be effectively used by OpenCL without the need for copying data between the application processor and the GPU. Avoiding memory copy is one of the key optimization that might be needed for optimal performance.

3.3.2.3.1 Choose mapping over copying to avoid memory copy when used within OpenCL context (when allocated within OpenCL)

Enable CL_MEM_ALLOC_HOST_PTR so that memory objects can be mapped instead of needing to be copied. This flag specifies that the buffer should be allocated from host-accessible memory. This is accomplished by setting cl_mem_flags input in clCreateBuffer as follows:

```
Buffer = clCreateBuffer(
   Context,
   CL_MEM_ALLOC_HOST_PTR,
size_t size, void *host_ptr,
cl_int *errcode_ret);
```

To use the buffer by the apps processor side, the buffer should be mapped, written to, and then unmapped. This is the only method to avoid copying of data.

Corollaries of this choice are:

- Avoid CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR if possible, as the driver incurs additional memory copies every time a memory object is mapped or unmapped.
- When calling clCreate{Buffer/Image}, create memory objects for either read-only or write-only from the host, i.e., avoid using read-write. For this pass

  For this pass, CL_MEM_HOST_READ_ONLY, CL_MEM_HOST_WRITE_ONLY, or CL_MEM_HOST_NO_ACCESS
- Use clEnqueueMapBuffer to get a pointer to a buffer, write to the buffer by host, then use clEnqueueUnMapBuffer to return buffer control to kernel (instead of using clEnque{read/write}{buffer/Image} ().

NOTE: For more details on use of buffer and images, see [R1]. Also, [S1] provides details regarding use of the extensions and flags for buffer and images.

3.3.2.3.2 Avoid memory copy for memory allocated out of scope of OpenCL

Approaches to avoiding memory copy for memory allocated out of scope of the OpenCL are:

- Using Ion memory extension

  This extension can be used for any use cases where shared memory between the host and GPU is required. When using Ion extensions to share memory, explicit synchronization is required.
- Using QTI Android native buffer extension

  This extension allows accessing Android native buffers in OpenCL. It is easier to use than the Ion extension and also does not require access to internal QTI header files.
- EGL image path – using the standard EGL extensions

  If the external allocation is an EGLImage or an EGLImage can be created for it, then this path can be used to share it with OpenCL. There are other EGL extensions that have been designed to work with this extension to allow efficient synchronization as well as YUV processing.

Ion memory extensions

If memory was initially allocated outside the scope of the OpenCL API and it is allocated using Ion/Gralloc, use the cl_qcom_ion_host_ptr extension, which maps the Ion memory space to that of the GPU memory space provided the file descriptor is available. Follow these steps:

1. Call ‘int fd = open("/dev/ion", O_RDWR);’ to get the file descriptor.
2. Set up a sequence of IOCTL calls to ensure the Ion object is created properly.
3. Run ADB root before launching your application.

NOTE: Use of Ion memory through the Qualcomm extension to avoid memory copy is illustrated by a detailed sample code that can be provided on request.

Using Qualcomm Android native buffer extensions

In many use cases, e.g., camera and video processing, Android native buffers (allocated by Gralloc) must be shared. Sharing is possible because the buffers are based on Ion. However, to use the Ion path, the programmer would need to extract internal handles from these buffers, which requires access to Qualcomm's internal headers. cl_qcom_android_native_buffer_host_ptr is a private extension that offers a more straightforward way to share Android native buffers with OpenCL without needing access to internal headers. This enables ISVs and other third-party developers to implement zero-copy technique for Android native buffers.

NOTE: A sample that illustrates use of Qualcomm Android buffer extensions can be provided on request.

Using standard EGL extensions

cl_khr_egl_image is a Khronos extension that allows creating an OpenCL image from an EGLImage. The main benefits that come with this are:

- It is standard. Code written to use this technique will most likely work for other GPUs that support it.
- EGL/CL extensions (cl_khr_egl_event and EGL_KHR_cl_event) that are designed to work with this extension make more efficient synchronization possible.
- YUV processing is a little easier with the EGL_IMG_image_plane_attribs extension.

NOTE: A sample that illustrates use of the EGLImage path through standard EGL extensions is available on request.

3.3.2.3.3 Avoid create/release memory objects between NDRange calls to avoid any chance of causing GPU stalls

3.3.2.4 Use event-driven pipeline

Suggestions for how to schedule CPU and GPU usage are:

- Parallelize CPU and GPU execution and do not let CPU stall for one clEnqueueNDRangeKernel. The CPU should just initiate these calls and check the final results.
- To streamline the execution process, put all functions on GPU to execute, even if some may not be GPU-friendly.
- Avoid blocking API calls. Blocking calls stalls the CPU to wait for the GPU to finish, then stalls the GPU before the next NDRange call. Blocking API calls is useful only for debugging.

3.3.2.5 Move expensive API calls to initialization phase

Some OpenCL functions take a long time to execute and should almost always be called outside of the main loop.

```
clBuildProgram()
clCreate...()
clLinkProgram()
clUnloadPlatformCompiler()
```

To reduce execution time during startup, use clCreateProgramWithBinary instead of clCreateProgramWithSource. However, be ready to fall back to building from source for situations such as recently updated drivers.

See Section 3.3.4.3 for more details about a sample application for this use case.

The execution time of clCreate{Image|Buffer} is a linear function of the amount of memory requested. Smaller pixel formats are preferred, as they use less memory bandwidth.

Android uses the Ion memory allocator. clCreate{Buffer|Image2D} is able to create memory objects with Ion pointers instead of allocating additional memory and needlessly copying the data into it.

If over 100 OpenCL APIs are called in a given main loop iteration, the CPU takes a long time to execute and becomes a bottleneck.

3.3.2.6 Compiler options

OpenCL supports some compiler options that are defined in Section 5.6.4 of [S1]. A section is dedicated to performance optimization compiler options. With these options, the compiler attempts to improve performance. Compiler options are passed in through the APIs clCompileProgram and clBuildProgram.

Multiple options can be combined:

```
clBuildProgram( myProgram,
            numDevices,
            pDevices,
            “-cl-fast-relaxed-math -qcom-accelerate-16-bit”,
            NULL,
            NULL );
```

3.3.2.6.1 QTI-specific options

qcom-accelerate-16-bit

This option sets a flag to the driver to use 16-bit ALUs instead of 32-bit ALUs. This can be done when the number of 16-bit operations are considerably more than the number of 32-bit operations in the kernel; e.g.:

```
clBuildProgram(program, 0, NULL, "-qcom-accelerate-16-bit", NULL, NULL);
```

NOTE: This compiler option is set automatically from compiler ver 24.02 and later.

qcom-sched-rule=2

This option activates a compiler instruction scheduling heuristic to try to minimize the register usage, e.g.:

```
clBuildProgram(program, 0, NULL, "-qcom-sched-rule=2", NULL, NULL);
```

3.3.3 Kernel

This section identifies kernel-level optimizations that can help improve performance.

3.3.3.1 Memory load/store

A very important factor that determines overall performance is the usage of load/store functionality. Memory load is a two-way process; SPs send a request, then the cache responds to the request. If there is a cache miss, the data is fetched from global memory. However, memory write is a one-way process, and noncoalesced write could impact performance. Some ways to optimize load/store are:

- Coalesced memory load/store
  - L2 cache can combine the memory request into one burst if they are coalesced.
  - Read access from L2 to SP is not combined across multiple work items; access is one work item at a time.
  - Bandwidth between SP and L2 is 128 bits per clock.
- Allow all work items to participate in local memory load from global memory
  - Vectorize local memory load manually (compiler does not do this automatically).
  - If vectorization is not possible, use async_work_group_copy; this may help improve performance.

3.3.3.2 Vectorization

- For optimal SP to L2 bandwidth performance, align read access to a 32-bit address and write access to a 128-bit address.
- Vectorized load/store with coalesced memory access generally yields significant performance gain.
- Vectorized data load/store (vload/vstoreN) such that each load/store instruction can process up to 128-bit data.
- Alternatively, use cast (supported).
  - char *p1; char4 vec;
  - vec = *(char4 *)(p1 + offset);

Vectorized load/store of a larger data type is more optimal than a small data type; e.g., a load of uint2* is more optimal than uchar8*. However, this has been modified in more recent versions of compiler, i.e., 24.00.02 and later, where a vectorized load of both data types perform equally well.

CAUTION: ALU calculations are scalar since the SPs in Adreno 3xx and later are scalar-based. Hence, a vectorized ALU benefit is not as much as a vectorized memory load/store.

CAUTION: Vectorization may lead to too many pixels in one work item that may result in a higher register footprint and lead to a smaller work group size, or even private memory may be used. This may result in lower performance.

For example, using the type float16 increases register footprint but does not have performance gain in terms of ALU operation.

See the vectorized operation example in Section 3.3.4.1.

3.3.3.3 Workgroup size

Optimal workgroup (WG) size depends first on performance portability across devices.

If extreme performance is required, brute-force selection of a WG size would be needed. Steps for performing brute-force selection are provided below.

When attempting to pick the appropriate WG size, note that:

- Increasing the WG size generally improves the occupancy of the shader cores.

  This in turn improves the ability to hide memory access latencies, and results in significant performance gains for memory-bound kernels (see Figure 3-4).

  However, due to hardware resource constraints, e.g., GPR file size and local memory size, the peak occupancy and hence WG size may be restricted for certain kernels.

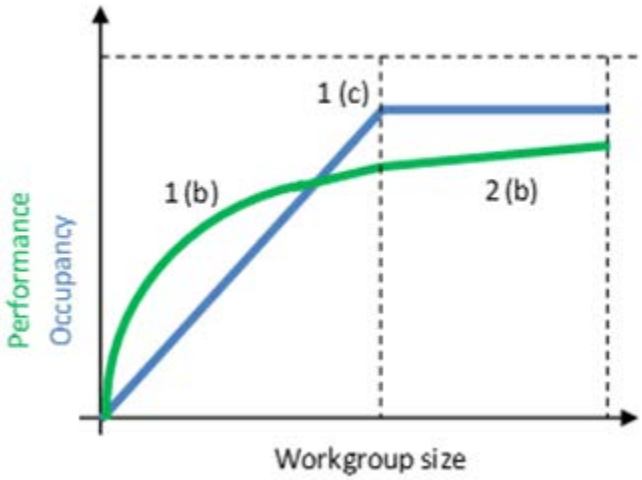


Figure 3-4 Theoretical relationship between workgroup size, occupancy, and performance

- For certain kernels, it is possible to increase WG size further, beyond the occupancy limit.
  - This is allowed only if no intra-WG synchronization (barriers) or local memory usage.
  - Performance improves slightly as wave drain latency is amortized across more waves.
- Noticeable variations in performance can result from small changes in WG size. These variations are caused by:
  - Utilization of the shader cores, which is a function of the number of WGs (Figure 3-5)
  - SIMD wave utilization, which is a function of WG size and wave mode (Figure 3-6)
  - Effective cache utilization, which is a function of WG size and shape, and the data access pattern of kernel instances

Figure 3-5 Overall shader core utilization

Figure 3-6 SIMD/wave utilization

- For some kernels, performance may deteriorate with increasing WG sizes.

  The waves in a WG are not necessarily scheduled for execution in a predefined order, and larger WGs containing more waves may result in increased cache thrashing depending on the data locality and access patterns.

  The locality problem is especially acute for texture accesses, because the texture cache is typically smaller than the unified load/store cache.

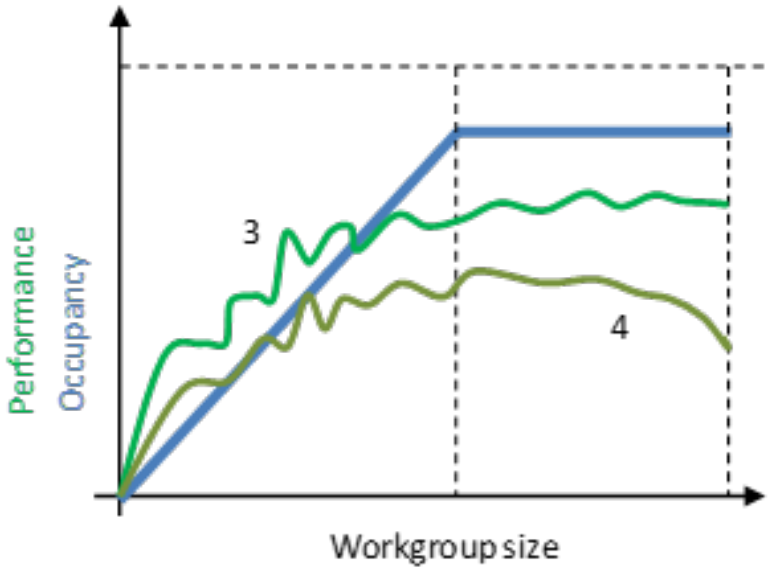


Figure 3-7 Local variations and/or drop in performance due to varying hardware utilization metrics

3.3.3.3.1 Performance portability

For performance portability across devices and computational domains, avoid making assumptions about the ideal workgroup size for your compute kernels. Instead, let the device driver/compiler on the device select a suitable workgroup size. This is achieved by passing NULL as the workgroup size parameter in clEnqueueNDRangeKernel.

3.3.3.3.2 Extreme performance optimization

If extreme performance is required, brute force selection of a workgroup size is necessary. The procedure for performing brute force selection is detailed in the section below.

Estimation of ideal workgroup size is not straightforward and thus not advisable if it can be avoided. Also, this type of implementation is *not* portable across devices.

Local workgroup size

The number of work items in one work group is also known as a local work size. For optimal kernel performance, the larger the workgroup size, the better, as this allows for more parallelization and better latency hiding.

To determine local work group size, first query work group sizes by calling clGetKernelWorkGroupInfo:

```
clGetKernelWorkGroupInfo( myKernel,
                  myDevice,
                  CL_KERNEL_WORK_GROUP_SIZE,
                  sizeof(size_t),
                  &maxWorkGroupSize,
                  NULL );
```

After getting the work group size, try different sizes using the size returned by the function as a maximum limit and then select the one that gives the best performance.

For example, assume:

```
maxWorkGroupSize = 256;
// For 1D workgroup
localWorkGroupSize[0] = 256; // Use maximum for 1D

// For 2D workgroup
localWorkGroupSize[0] = 16; // 16x16 = 256
localWorkGroupSize[1] = 16;

// Another option for 2D workgroup
localWorkGroupSize[0] = 64; // 64x4 = 256
localWorkGroupSize[1] = 4;

// Pass in the local work group size for execution
clEnqueueNDRangeKernel( myCmdQueue,
                  myKernel, dimension, NULL,
                  globalWorkgroupSize, // total work items
                  localWorkGroupSize, // determined local WG size
                  0, NULL, NULL );
```

In addition to being passed in through clEnqueueNDRangeKernel, work group size can also be embedded in the kernel function.

The attribute reqd_work_group_size(X, Y, Z) passes in the specified work group size as a requirement. An error is returned if the specified work group size cannot be satisfied.

For example, to require 16x16 work group size:

```
__kernel
__attribute__(( reqd_work_group_size(16, 16, 1) ))
     void myKernel( __global float4 *in, __global float4 *out)
     {
     . . .
     }
```

The attribute work_group_size_hint(X, Y, Z) passes in the specified work group size as a hint. The driver tries to use the hinted size, but does not guarantee the actual work group size matches the hint.

For example, to hint 64x4 work group size:

```
__kernel
__attribute__(( work_group_size_hint (64, 4, 1) ))
     void myKernel( __global float4 *in, __global float4 *out)
     {
     . . .
     }
```

Some additional pointers on how to choose the work group size are:

- Performance of WxH is not necessarily the same as HxW.
- The workgroup size depends on the register footprint.
- Passing NULL as the workgroup size does not guarantee optimal performance; for optimal performance, try all workgroup sizes.
- The local workgroup size must evenly divide the global work size.
- Pick the best workgroup size from empirical execution times.
- Brute force approach impractical if NDRange is not fixed, or for data-dependent execution.
- Other items that could lead to a smaller workgroup size are:
  - Use of local memory
  - More pixels to process in each work item
  - Control flows, e.g., boundary check
  - Kernel arguments such as sampler
  - Conformant math functions
- Best practices to consider:
  - Prefer workgroup sizes that yield ≥ CL_DEVICE_MAX_COMPUTE_UNITS number of workgroups for optimal workgroup-level parallelism.
  - Query the device for CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_ MULTIPLE. Prefer workgroup sizes that are multiples of this value.

3.3.3.3.3 Global WG size

The global workgroup size significantly affects performance. Ideally, the global workgroup size in each dimension should be a multiple of 32. For example, instead of enqueuing 1920x1080 work items, it may be good to enqueue 1920x1088 with an image that needs cropping or padding. The beginning of the kernel function could check whether the current work item falls within the boundaries of the image, and return if it is out of the boundaries.

If the global work size is an odd number (e.g., a prime number), the legal choices for a work group size are limited since the recommended global work group size is a multiple of a local work group size. If needed, manually add a kernel argument to ignore the last padded global work items.

GPGPU is good for parallel processing of large amount of data and a computationally intensive workload, but not for a small workload. For small workloads, the gain of GPGPU over CPU may be overshadowed by the overhead of launching kernels on GPU. As a rule of thumb, if the global workgroup size contains at least 10,000 work items, it would help to compensate the overhead of submitting a kernel.

See Section 3.3.4.3 for more details about a sample application for this use case.

3.3.3.4 Local, global, and constant memory

Local memory

Local memory resides on the chip. Use the address space qualifiers_local to access this on-chip memory region. For additional details on address space qualifiers, see Section 6.5 in [S1].

It is recommended to use this type of memory if data is used repeatedly. An ideal flow loads data from global to local memory, and then work items read and/or write to the local memory multiple times, e.g., a window-based motion estimation/object match, where each work item needs to fetch local memory data with a moving window.

For data exchange between work items, work item A writes data to local memory and work item B reads it. This usually comes with a barrier operation before the local memory read due to the relaxed memory consistency in OpenCL.

Work items may write to local memory multiple times, and ultimately write the data to global memory. Allow each work item to participate in local memory data load rather than using one work item to do the entire load.

Vectorized local memory read/write is supported similar to that of global memory, which is recommended to be used. Because the SP to GPU cache is 128 bits wide, using a vectorized load of 128 bits 32-bit aligned data into local memory is beneficial.

Function async_work_group_copy() can be used to better utilize the hardware feature to do data load and store between global and local memory. See Section 6.12.10 of [S1] for additional details.

Limitations that application developers need to be aware off while trying to use this memory region are:

- The driver reports an error if the local memory used by the kernel being enqueued is out of the hardware limit. Query for the size of the local memory by using the API:

```
clGetDeviceInfo(deviceID, CL_DEVICE_LOCAL_MEM_SIZE, .. )
```

- If time dependencies exist between work items, pay attention to the possible need to synchronize their execution so that as the work items complete and the same results are produced. Work group barriers can be used to complete this task. However, this synchronization is costly and using global memory may be a better option if too many synchronizations are needed. See Section 5.10 in [S1] for more information on work group barriers.

- Local memory does not support unaligned access while Global memory does.

For example, see the Adreno SDK: MatrixMatrixMul sample.

Global memory

Use global memory instead of local memory for nonrepeated data reads.

- May have better L2 cache hit ratio and better performance
- Simpler code than when local memory is being used
- Could result in a larger work group size

Constant memory

For better performance, consider declaring lookup tables as global scope variables in the __constant address space, especially when the lookup tables are larger than the available constant RAM on the GPU. This can be done for read-only arguments of any size to the kernels as well. An example of how this can be done is shown below.

```
//
// Good performance:
//
__constant float foo_fast[] = {...};
__kernel void myFastKernel(__global float* bar)
{
    // Access constant lookup table 'foo_fast' here.
}

//
// Poor performance for constant parameter that's larger than constant RAM:
//
__kernel void mySlowKernel(__constant float* foo_slow, __global float* bar)
{
    // Access lookup table 'foo_slow' here.
}
```

To utilize the constant memory, the compiler must able to determine the size of the constant variables. The size of the variable declared as constant cannot exceed the maximum constant memory allowed. For the above case, an attribute, max_constant_size(N), can also be provided to the compiler to indicate the size of the constant data for the compiler. N represents number of bytes of the variable.

For example, passing 1024 bytes variable foo:

```
__kernel void myFastKernel(
     __constant float foo*
     __attribute__( (max_constant_size(1024)) )
{
. . .
}
```

Maximum constant size can be queried as follows:

```
GetDeviceInfo( deviceId,
          CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE,
          (VOID*)&maxConstantSize,
          sizeof(maxConstantSize) );
```

Note that since argument parameters declared as scalar and vector data types are stored in constant RAM, smaller-size LUT can also be declared as follows:

```
__kernel void myFastKernel((__global float* bar, float8 coeffs)
{
   //coeffs will be mapped to constant RAM
}
```

Constant RAM is *not* used if:

- The array is dynamically indexed with multiple accesses. However, there is no adverse impact if single access is made.
- Any data type other than float is used, e.g., uint, etc.
- There is more than one kernel in the file.

The above three cases are fixed on compiler ver 24.02 and later. To determine the compiler version, use the following command:

```
adb logcat | grep –irn “compiler”
```

- Use of vector declaration.
- Vector loading (by vload() or pointer access) is used.

The above two cases will be fixed in upcoming releases of the compiler. Until then, do not use these cases in your application if you want to use constant RAM.

3.3.3.5 Native math functions

When being executed with conformant math, most math functions require very high precision and are expensive. If high precision is not needed and faster execution is, do one of the following:

- Use native math functions; the hardware supports some elementary functions natively with reduced accuracy, which are significantly faster than built-in functions.
  - Math functions that have native implementation are native_cos, native_exp, native_exp2, native_log, native_log2, native_log10, native_powr, native_recip, native_rsqrt, native_sin, native_sqrt, native_tan; e.g.:
    - Original – const int edge_deno = a / b;
    - Use native instruction – const int edge_deno = (int)native_divide((float)(a), (float)(b));

  The above native usage is fairly accurate when the denominator is small.
- Enable -cl-fast-relaxed-math in the clBuildProgram() call, unless absolute accuracy in math operations is required.

Table 3-2 shows a list of OpenCL-GPU math functions categorized based on their relative complexity and that use a combination of ALU and EFU hardware support or complex emulation.

Table 3-3 OpenCL-GPU math functions performance categories

Performance category	Function name
A (simple using ALU instructions only)	ceil
	copysign
	fabs
	fdim
	floor
	fmax
	fmin
	fract
	frexp
	ilogb
	mad
	maxmag
	minmag
	modf
	nan
	nextafter
	rint
	round
	trunc
B (EFU only or EFU plus simple ALU instructions)	asin
	asinpi
	atan
	atanh
	atanpi

Performance category	Function name
	cosh
	exp
	exp2
	rsqrt
	sqrt
	tanh
C (combination of ALU, EFU, and bit maneuvering)	acos
	acosh
	acospi
	asinh
	atan2
	atan2pi
	cbrt
	cos
	cospi
	exp10
	expm1
	fmod
	hypot
	ldexp
	log
	log10
	log1p
	log2
	logb
	pow
	remainder
	remquo
	sin
	sincos
	sinh
	sinpi
D (complex software emulation)	erf
	erfc
	fma
	lgamma
	lgamma_r
	pown
	powr
	rootn
	tan

Performance category	Function name
	tanpi
	tgamma

Therefore, while programing the kernel, the function preference based on performance should be in the order shown in the above table, which is functions using:

- ALU instructions only
- EFU only or EFU plus simple ALU instructions
- A combination of ALU, EFU, and bit maneuvering
- Complex software emulation

3.3.3.6 Avoid divergence

The GPU is not efficient when different work items in the same wave follow different control flow paths. When all work items in the same wave follow the same path, there is no performance penalty, since all work items within a hardware thread share the same program counter. For branches, some work items may have to be masked, resulting in lower GPU occupancy.

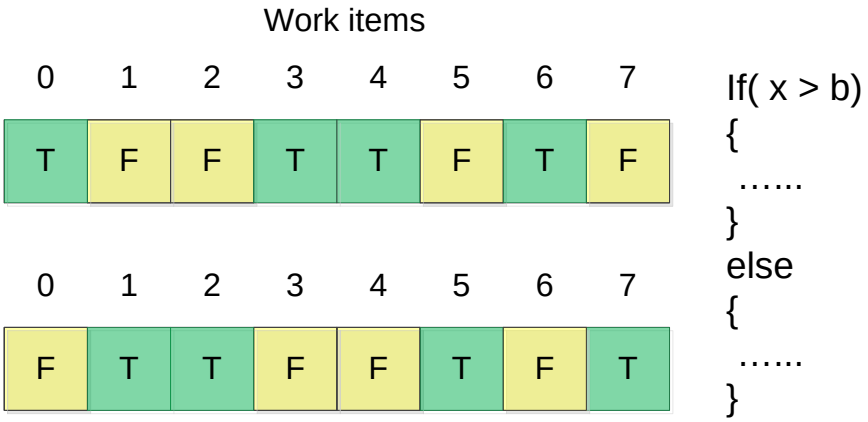


Figure 3-8 Pictorial representation of divergence across two waves

3.3.3.7 Handling image boundaries

Many operations may go out of the image boundaries, e.g., filtering, transform, etc. For handling the boundaries better, add boundary check-in kernels. This may result in poor performance, as the boundary check causes divergent branches, reduces work group size, etc.

A better solution than adding boundary checks in the kernel would be to try the following:

- Pad the image upfront, if possible.
- Use image object with proper samplers (texture engine handles it automatically).
- Write separate kernels to handle boundaries.

3.3.3.8 Optimal use of cache

For optimal cache usage:

- Avoid loops within each work item and use more work items resulting in more work groups.
- Add atomics to sync all waves.

Metrics such as the L1 read/write hit rate and L2 read/write hit rate help determine how well the cache is being used. These metrics are available in the Adreno Profiler's OpenCL scrubber's metrics section.

3.3.3.9 Maximizing ALU utilization

One of the primary goals of optimization is to ensure that the ALU utilization is as high as possible. A computation-intensive application with minimal amount of data reads and writes, which indicates not much of a dependency on data bus latency, can run at up to 80% ALU utilization. Such high ALU utilization also indicates that further optimizing such an application would be a hard task with very minimal improvements.

Some other ways to make sure you achieve high ALU utilization are:

- Precalculate values that do not change within the kernels. It is wasteful to calculate a value that can be precalculated outside of the kernels, as it is redundant to calculate the same value across all work items.
- All work items run the same code. This is to avoid branching or divergence as mentioned before.
- Adreno has native 16-bit, 24-bit integer multiplication support. For short or 24-bit integer multiplication, use mul24(); 32-bit integer multiplication is emulated.
- Use #define to pass any constant scalar to a kernel if the value of the constant scalar is known at kernel build time. For constant arrays (such as LUT, filter taps, etc.) declare the constant array outside the kernel scope. Implementation should follow the guidelines in "Constant Memory" in Section 3.3.3.4 when there is a need to pass constant array pointer. Use OpenCL built-in functions instead of writing your own functions since these are compiler-optimized, e.g., clamp. See Section 6.12 in [S1] for more details on built-in functions.

3.3.4 Sample application code

3.3.4.1 Optimization examples

Example – Improve algorithm

NOTE: This example is excerpted from Adreno SDK: ImageBoxFilter. For full sample code, see the Adreno SDK.

Given an image, apply a simple 8x8 box blurring filter on it.

```
// Original kernel before optimization
__kernel void ImageBoxFilter(__read_only image2d_t source,
                             __write_only image2d_t dest,
```

```
                                sampler_t sampler)
{
   ... // variable declaration

   for( int i = 0; i < 8; i++ )
   {
        for( int j = 0; j < 8; j++ )
        {
             coor = inCoord + (int2) (i - 4, j - 4 );
             // !! read_imagef is called 64 times per work item
             sum += read_imagef( source, sampler, coor);
        }
   }
   // Compute the average
   float4 avgColor = sum / 64.0f;

   ... // write out result
}
```

To reduce texture access, the above kernel is split into two passes. The first pass calculates the 2x2 average for each work item and saves the result to an intermediate image. The second pass uses the intermediate image for the final calculation.

```
// First pass: 2x2 pixel average
__kernel void ImageBoxFilter(__read_only image2d_t source,
                             __write_only image2d_t dest,
                             sampler_t sampler)
{
   ... // variable declaration

   // Sample an 2x2 region and average the results
   for( int i = 0; i < 2; i++ )
   {
        for( int j = 0; j < 2; j++ )
        {
             coor = inCoord - (int2)(i, j);
             // 4 read_imagef per work item
        sum += read_imagef( source, sampler, inCoord - (int2)(i, j) );
        }
   }
   // equivalent of divided by 4, in case compiler does not optimize
   float4 avgColor = sum * 0.25f;
   ... // write out result
}
```

```
// Second Pass: final average
__kernel void ImageBoxFilter16NSampling(__read_only image2d_t source,
                                __write_only image2d_t dest,
                                sampler_t sampler)
{
   ... // variable declaration
   int2 offset = outCoord - (int2)(3,3);

   // Sampling 16 of the 2x2 neighbors
   for( int i = 0; i < 4; i++ )
   {
        for( int j = 0; j < 4; j++ )
        {
             coord = mad24((int2)(i,j), (int2)2, offset);
             // 16 read_imagef per work item
             sum += read_imagef( source, sampler, coord );
        }
   }
   // equivalent of divided by 16, in case compiler does not optimize
   float4 avgColor = sum * 0.0625;
   ... // write out result
}
```

The modified algorithm accesses the image buffer 20 (4+16) times per work item, which is significantly less than the original 64 read_imagef.

Example – Vectorized operation

NOTE: This example is excerpted from Adreno SDK: MatrixMatrixAdd. For full sample code, see the Adreno SDK.

Given two MxM matrices, perform addition.

```
// Original kernel before optimization
__kernel void MatrixMatrixAddSimple(const int matrixRows,
                                    const int matrixCols,
                                    __global float* matrixA,
                                    __global float* matrixB,
                                    __global float* MatrixSum)
{
   int i = get_global_id(0);
   int j = get_global_id(1);
   // Only retrieve 4 bytes from matrixA and matrixB.
   // Then save 4 bytes to MatrixSum.
```

```
    MatrixSum[i*matrixCols+j] =
         matrixA[i*matrixCols+j] + matrixB[i*matrixCols+j];
}

// Modified kernel for optimization
__kernel void MatrixMatrixAddOptimized2(const int rows,
                                        const int cols,
                                        __global float* matrixA,
                                        __global float* matrixB,
                                        __global float* MatrixSum)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    // Utilize built-in function to calculate index offset
    int offset = mul24(j, cols);
    int index  = mad24(i, 4, offset);

    // Vectorize to utilization of memory bandwidth for performance gain.
    // Now it retieves 16 bytes from matrixA and matrixB.
    // Then save 16 bytes to MatrixSum
    float4 tmpA = (*((__global float4*)&matrixA[index]));
    float4 tmpB = (*((__global float4*)&matrixB[index]));
    (*((__global float4*)&MatrixSum[index])) = (tmpA+tmpB);
    // Since ALU is scalar based, no impact on ALU operation.
}
```

Example – Utilize texture engine for buffer

NOTE: This sample code is excerpted from Adreno SDK: DotProduct. For the full sample, see the Adreno SDK.

Given 5 million pairs of vectors, calculate dot product for each pair.

```
// Original kernel before optimization
__kernel void DotProduct(__global const float4 *a,
                         __global const float4 *b,
                         __global float *result)
{
    // a and b contain 5 million vectors each
    // Arrays are stored as linear buffer in global memory
    result[gid] = dot(a[gid], b[gid]);
}

// Modified kernel for optimization
__kernel void DotProduct(__read_only image2d_t c,
```

```
                        __read_only image2d_t d,
                        __global float *result)
{
   // Image c and d are used to hold the data instead of linear buffer
   // read_imagef goes through the texture engine
   int2 gid = (get_global_id(0), get_global_id(1));
   result[gid.y * w + gid.x] =
        dot(read_imagef(c, sampler, gid), read_imagef(d, sampler, gid));
}
```

This case uses the texture engine to improve frequent data access. It is a fairly simple example, but the technique can be applied to many situations in which data access is the potential bottleneck.

3.3.4.2 Adreno SDK

The Adreno SDK contains a list of samples that demonstrate many usages of OpenCL. Many samples also employ the optimization techniques described in this document. The Adreno SDK can be downloaded from https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-gaming-graphics-adreno.

For a list of optimized samples in the SDK, see [Q2].

Some of the samples featured in the Adreno SDK are:

- BandwidthTest
- ClothSimCLGLES
- Concurrency
- DeviceQuery
- DotProduct
- FFT1D
- FFT2D
- ImageBoxFilter
- ImageMedianFilter
- ImageRecursiveGaussianFilter
- ImageSobelFilter
- InteropCLGLES
- MatrixMatrixAdd
- MatrixMatrixMul
- MatrixTranspose
- MatrixVectorMul
- ParallelPrefixSum

- ParticleSystemCLGLES
- PostProcessCLGLES
- VectorAdd

3.3.4.3 More examples

More samples applications are under development and will be released through the Adreno SDK in the near future. For status of the following samples, contact any of QTI's OpenCL customer support representatives.

NOTE: Applications designated with an asterisk (*) have been completed and are available upon request from a QTI customer support representative.

- Usage of Ion memory for images and buffers*

  This application clearly demonstrates how to create and use Ion memory when Ion memory is already created by the Android framework or another layer. This helps illustrate a usage model for zero memory copy.
- Usage of Ftrace markers for use in measurement of timing (in user application)*
- Usage of Android tools to do plot timing
- Proving tools to analyze from a command line
- Usage of embedding kernel binary in application source code to illustrate the performance improvement during loading*
- Sample to allow application developers to experiment with work group sizes (global and local) and illustrate the register footprint that are consequences with the various work group sizes*
- Usage of constant memory/local memory
  - Sample code for lookup tables use case using constant memory
  - Sample code for fitting filters into combination of constant memory plus local memory (without spilling into global memory)
  - Some mechanism to identify if constant memory is being used
- Kernel samples that are optimized to demonstrate performance increase on CLBenchmark
- Work group size padding and tuning
- Usage of data compression and packing
- Usage of data format conversion with zero penalty

3.3.5 Measuring performance

3.3.5.1 CPU timers

CPU timers can be used for measuring the execution time of OpenCL calls that are executed only on the CPU. This can be achieved by using any of the C date and time functions that are part of the standard library of the C programming language.

An example is to use gettimeofday():

```
#include <time.h>
#include <sys/time.h>

main() {
struct timeval start, end;
/*get the start time*/
gettimeofday(&start, NULL);

/*Execute function of interest*/
   {
   . . .
   }

/*get the end time*/
 gettimeofday(&end, NULL);

/*Print the amount of time taken to execute*/
 printf("%ld\n", ((end.tv_sec * 1000000 + end.tv_usec)
          - (start.tv_sec * 1000000 + start.tv_usec)));
}
```

3.3.5.2 GPU timers

The GPU execution time is determined by hardware counters that are independent of the operating system.

All enqueue calls optionally return an event object that can be used for measuring the GPU execution times. These event objects are enabled and used by clGetEventProfilingInfo(). For example, see Adreno SDK: VectorAdd. To enable profiling, set the CL_QUEUE_PROFILING_ENABLE flag in the properties argument of either clCreateCommandQueue or clSetCommandQueueProperty. The execution time on GPU is provided by difference in the clGetEventProfilingInfo() calls with the parameters CL_PROFILING_COMMAND_START and CL_PROFILING_COMMAND_END.

This can be achieved by programming these events around the specific clEnqueueNDRange(), or an easier method would be to use the Adreno Profiler tool to profile the openCL application, which will automatically provide all the CPU and GPU timing information for the entire application. For more information on how to use the Adreno Profiler, see the Adreno Profiler user guide (downloaded with Adreno Profiler as indicated in Section 4.1) or use the help section within the Adreno Profiler tool itself.

3.3.5.3 Performance mode

During optimization, it may be necessary to:

- Ensure that the application being profiled renders full screen so that no other activity is updating the screen. If it is a native application, be sure SurfaceFlinger is not running. This ensures that the CPU and GPU are being solely used by the application being profiled.
- Measure the execution time in the Performance mode of the CPU and GPU to avoid fluctuating measurements that may be caused by CPU and GPU clock control mechanisms, which are always running in the background to optimize power consumption.

  The sequence of commands needed to enable Performance mode are outlined below. These commands are not persistent and are effective until the next system restart. The command sets ensure disabling the CPU frequency fluctuations and controlling data buses that influence data transfer rates.

3.3.5.3.1 CPU settings for performance

```
/*disabling mpdecision keeps all CPU cores ON*/
adb shell stop mpdecision

/*Set performance mode for all CPU cores. In this case, a dual core CPU*/
adb shell "echo performance >
/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor"
adb shell "echo 1 > /sys/devices/system/cpu/cpu1/online"
adb shell "echo performance >
/sys/devices/system/cpu/cpu1/cpufreq/scaling_governor"
```

3.3.5.3.2 GPU settings for performance

Disabling power scaling policy

Newer targets support the following method for disabling power scaling:

```
"echo performance > /sys/class/kgsl/kgsl-3d0/devfreq/governor"
```

Legacy targets support the following method for disabling power scaling:

```
/*Disable power scaling policy for GPU*/

adb shell "echo none > /sys/class/kgsl/kgsl-3d0/pwrscale/policy"
```

Disabling GPU sleep

Force the GPU clocks/bus vote/power rail to always on.

- Keep clocks on until the idle timeout forces the power rail off.
- Keep the bus vote on permanently.
- Keep the gfx power rail on permanently.

The command sequence for newer targets is:

```
adb shell “echo 1 > /sys/class/kgsl/kgsl-3d0/force_clk_on”
adb shell “echo 1 > /sys/class/kgsl/kgsl-3d0/force_bus_on”
adb shell “echo 1 > /sys/class/kgsl/kgsl-3d0/force_rail_on”
```

The command sequence for legacy targets is:

```
/*Disable GPU from going into sleep*/
adb shell "echo 0 > /sys/class/kgsl/kgsl-3d0/pwrnap"

/*Or Set a very high timer value for GPU sleep interval*/
adb shell "echo 10000000 > /sys/class/kgsl/kgsl-3d0/idle_timer"
```

In the profiling version of the application, introduce clFinish() calls to synchronize between the CPU and GPU. This ensures that the true execution times are obtained for all calls that may be blocking or nonblocking.

4 Software Development Tools

QTI provides a suite of tools to facilitate development and porting of apps to OpenCL. These include the Adreno Profiler and Adreno Software Development Kit (SDK).

4.1 Adreno Profiler

The Adreno Profiler tool facilitates with profiling the OpenCL app to identify optimization opportunities through the use of many performance measurement metrics and static analysis of OpenCL kernels. The Adreno Profiler tool can be downloaded from https://developer.qualcomm.com/. See [Q2] for guidelines on how to use the Adreno Profiler tool.

4.2 Adreno SDK

The Adreno SDK provides a set of sample OpenCL use cases that are commonly used in most image processing-based use cases. Some of the use cases are highly optimized for use in the Snapdragon platform.

In future revisions of the SDK, additional optimized samples and other samples that illustrate optimization techniques specific to the Snapdragon platform will be available. See Chapter 6 for more details about the SDK.

5 Android SDK/NDK Installation Guide

This chapter describes the installations and guidelines to program OpenCL apps using the Android NDK, as well as details to create a package with the Android SDK.[1]

The Android NDK and SDK are supported on Windows, OSX, and Linux.

The following guidelines describe installation for an OSX system. See the Android Developer's site for installation details for your particular environment.

5.1 Determine location for installed packages

Components installed are:

- Apache Ant
- Android NDK
- Android SDK

Determine at the start of installation where these components should be located on your system.

The following variables are used to reference the installation location:

- APACHE_ANT – /usr/local/bin/apache_ant/bin/ant
- ANDROID_NDK – /Volumes/work/android_ndk
- ANDROID_SDK – /Volumes/work/android_sdk

5.2 Configuring paths

On OSX, edit ~/.bashrc and add the following block to define the paths. This facilitates the remainder of the installation.

```
#--------------------------------------
# Android Development Setup
#--------------------------------------
export ANDROID_SDK=/Volumes/work/android_sdk/android-sdk-macosx
export ANDROID_NDK=/Volumes/work/android_ndk/android-ndk-r8b
export APACHE_ANT=/usr/local/bin/apache_ant
# Setup paths for Android Development tools.
export PATH=$ANDROID_SDK/tools:$PATH
export PATH=$ANDROID_SDK/platform-tools:$PATH
export PATH=/usr/local/bin/apache_ant/bin:$PATH
```

[1]The Android NDK and SDK are available through the Android Developer site.

```
export PATH=$ANDROID_NDK/toolchains/arm-linux-androideabi-
4.4.3/prebuilt/darwin-x86/bin:$PATH
```

5.3 Installing Apache Ant

Apache Ant is open source software that is used to package the components of an app into the <app>.apk package.

1. Download the package from http://ant.apache.org/bindownload.cgi.
   a. Select the current release of Ant.
   b. Select the tar.gz archive.
2. Copy the downloaded file (apache-ant-1.8.4-bin.tar.gz) to a temp area, e.g., ~/temp/apache_ant.
3. Open a terminal window.
4. Extract the archive:

   ```
   tar -xvf apache-ant-1.8.4-bin.tar.gz
   ```

5. Create a directory to store the Ant release.

   ```
   mkdir -p  $APACHE_ANT
   ```

6. Copy the distribution.

   ```
   cp -r ~/temp/apache_ant  $APACHE_ANT
   ```

7. Test Apache Ant.

   ```
   ant -version
   Apache Ant(TM) version 1.8.4 compiled on May 22 2012
   ```

NOTE: Use Apache Ant to build the APK from a command line.

5.4 Installing Android SDK

To install Android SDK:

1. Go to the Android standard developer site http://developer.android.com/sdk/index.html.
2. Go to http://developer.android.com/sdk/installing/adding-packages.html.
3. After unpacking the installer, cd to $ANDROID_SDK/tools and execute: android sdk.
   - This downloads all the required Android packages. The process may take some time.
   - If you are prompted to enter a user ID/password for third-party packages, click **Enter** to skip them.

4. Ensure that $ANDROID_SDK is defined to point to the location of your SDK installation and exported to the path of the Android SDK tools directory.

```
export ANDROID_SDK=/Volumes/work/android_sdk/android-sdk-macosx
export PATH=$ANDROID_SDK/tools:$PATH
```

5.5 Installing Android NDK

To install Android NDK:

1. Go to the Android standard developer site http://developer.android.com/tools/sdk/ndk/index.html.
2. Select the download for your platform.

NOTE: Instructions for Mac OS X are displayed. NDK Rev 9c 64 bit (android-ndk-r9c-darwin-x86_64.tar.bz2) was used.

3. Untar the archive. Ensure that $ANDROID_SDK is defined to point to the location of your NDK installation.
4. Configure for OpenCL.
5. Select the platform API version you will use for your projects, e.g., android-14.
6. OpenCL header files are part of the Adreno SDK, but OpenCL library is not and must be added to the build path.

5.5.1 Installing OpenCL library in Android NDK

To install the OpenCL library:

1. Do one of the following:
   - Locate the libOpenCL.so library from your Android build distribution. It is located in out/target/product/msm8960/system/vendor/lib.
   - Pull the libOpenCL.so library from the QTI development device.

```
adb pull /system/vendor/lib/libOpenCL.so
```

2. Install the library to your NDK lib directory, e.g., $ANDROID_NDK/platforms/android-14/arch-arm/usr/lib/libOpenCL.so.

5.6 OpenCL development with the Android NDK/SDK

Key files required for the Android SDK/NDK are shown in this section. Refer to documentation on the Android developer site for additional information.

Determine the location for your project. This can be any location, as your system has been set up with paths to find the NDK/SDK required components in the previous installation steps.

The project directory in the example is **<project>**.

AndroidManifest.xml

The Android manifest is used to pull in all components in the SDK package. The AndroidManifest.xml file for the sample hello-cl app is:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.qualcomm.hello_cl_gpu"
      android:versionCode="412"
      android:versionName="4.12">
        <uses-sdk android:minSdkVersion="8" />
        <uses-feature android:glEsVersion="0x00020000"/>
        <supports-screens android:smallScreens="true"
                                android:normalScreens="true"
                                android:largeScreens="true"
                                android:anyDensity="true"/>
        <uses-permission android:name="android.permission.INTERNET"   />
        <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
        <uses-permission android:name="android.permission.WAKE_LOCK" />
    <application android:icon="@drawable/hello_cl_icon"
                                android:label="@string/app_name"
                                android:theme=
                                      "@android:style/Theme.NoTitleBar.Fullscreen"
                                android:debuggable="true"
                                android:hasCode="true">
                <!-- Our activity is the built-in NativeActivity framework class.
             This will take care of integrating with our NDK code. -->
                <activity android:name="android.app.NativeActivity"
                          android:launchMode="singleInstance"
                          android:label="@string/app_name">
                <meta-data android:name="android.app.lib_name"
                             android:value="hello_cl_gpu" />
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

```
        </application>
</manifest>
```

jni/Application.mk

The Application.mk file for the sample hello-cl app defines the compiler versions and Android platform version, e.g.:

```
APP_ABI := armeabi-v7a
APP_PLATFORM := android-14
APP_STL := gnustl_static
APP_CPPFLAGS += -fexceptions -frtti
```

jni/Android.mk

The Android.mk file for the sample hello-cl app is your makefile to build the NDK components that are included in the SDK.

The output should be a library (.so) file, in which you define the name, that will be included in your SDK package through the AndroidManifest.xml file.

The following example is taken from the hello-cl sample app:

```
LOCAL_PATH:= $(call my-dir)
JNI_DIR := $(call my-dir)

SRC_PATH:= ..
MODULE_SRC_FILES += $(SRC_PATH)/src/hello_cl_android_native.cpp
MODULE_SRC_FILES += $(SRC_PATH)/src/hello_cl.c
MODULE_INC_FILES := inc

include $(CLEAR_VARS)
LOCAL_CFLAGS += -DUSE_GPU -DOS_ANDROID_NATIVE
LOCAL_CFLAGS += -DANDROID_CL
LOCAL_SRC_FILES:= $(MODULE_SRC_FILES)
LOCAL_C_INCLUDES := $(MODULE_INC_FILES)
LOCAL_MODULE:= hello_cl_gpu

LOCAL_LDLIBS := \
-lOpenCL \
-lEGL \
-lGLESv2 \
-llog \
-landroid

LOCAL_STATIC_LIBRARIES := android_native_app_glue
```

```
include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
```

5.6.1 Additional SDK project definitions

Basic components are required for the simplest app to work properly.

Icon

Use Paintbrush or another drawing utility to create a 48x48 pixel icon. In our app, we specified the icon name as drawable/hello_cl_icon. The file should be located in <project>/res/drawable/<icon>.png.

Strings

Commonly used constants should be stored in a resource file. Place these in the <projects>/res/values directory, e.g., res/values/string_resources.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string
        name="app_name"
        >hello_cl</string>
</resources>
```

5.6.2 Build process

5.6.2.1 Updating your project

To build your project, some key components must be defined. This is done through the Android update command, e.g.:

```
android update project -p . -n projectname -t android-14
```

Output

```
Updated and renamed default.properties to project.properties
Updated local.properties
Added file ./build.xml
Added file ./proguard-project.txt
```

5.6.2.2 Building NDK components

From your project directory, run:

```
$ANDROID_NDK/ndk-build
```

Output

```
Gdbserver      : [arm-linux-androideabi-4.6] libs/armeabi-v7a/gdbserver
Gdbsetup       : libs/armeabi-v7a/gdb.setup
Compile++ thumb  : hello_cl_gpu <= hello_cl_android_native.cpp
In file included from jni/../src/hello_cl_android_native.cpp:27:0:
/Volumes/work/android_ndk/android-ndk-r8b/platforms/android-14/arch-
arm/usr/include/jni.h:592:13: note: the mangling of 'va_list' has changed
in GCC 4.4
Compile thumb  : hello_cl_gpu <= hello_cl.c
Compile thumb  : android_native_app_glue <= android_native_app_glue.c
StaticLibrary  : libandroid_native_app_glue.a
SharedLibrary  : libhello_cl_gpu.so
Install        : libhello_cl_gpu.so => libs/armeabi-v7a/libhello_cl_gpu.so
```

This shows that the library libhello_cl_gpu.so was successfully built. Integrate this component into the SDK package.

In the AndroidManifest.xml file for the sample hello-cl app, the following library package is specified to be included in the SDK:

```
<meta-data android:name="android.app.lib_name"
                            android:value="hello_cl_gpu" />
```

5.6.2.3 Packaging the app

Building the app is the final step. This is done with Apache Ant from your project directory. Run ant debug.

Output

```
<snip>
-do-debug:
 [zipalign] Running zip align on final apk...
     [echo] Debug Package:
/Volumes/qj/vendor/qcom/proprietary/gles/opencl_tests/src/cl_sdk_example/bi
n/cl_sdk_example-debug.apk
[propertyfile] Updating property file:
/Volumes/qj/vendor/qcom/proprietary/gles/opencl_tests/src/cl_sdk_example/bi
n/build.prop
```

```
[propertyfile] Updating property file:
/Volumes/qj/vendor/qcom/proprietary/gles/opencl_tests/src/cl_sdk_example/bi
n/build.prop
[propertyfile] Updating property file:
/Volumes/qj/vendor/qcom/proprietary/gles/opencl_tests/src/cl_sdk_example/bi
n/build.prop
[propertyfile] Updating property file:
/Volumes/qj/vendor/qcom/proprietary/gles/opencl_tests/src/cl_sdk_example/bi
n/build.prop

-post-build:

debug:

BUILD SUCCESSFUL
Total time: 4 seconds
```

5.6.2.4 Installing and running the app

In the sample hello-cl-gpu app, the process was kept as simple as possible, i.e., the app simply outputs the status to the console.

5.6.2.4.1 Installing the app

Set up your target device, adb remount. This makes the file system writable.

```
adb install -r  <app>.apk
```

Output

```
2314 KB/s (180337 bytes in 0.076s)
    pkg: /data/local/tmp/cl_sdk_example-debug.apk
Success
```

5.6.2.4.2 Running the app

1. Start a console and run:

   ```
   adb shell logcat
   ```

2. Find and click the app icon.

   For the hello-cl-gpu app, the following output appears on the console:

   ```
   V/hello_cl( 2393): Starting hello cl
   V/threaded_app( 2393): Start: 0x5c973398
   I/hello_cl( 2393): num_platforms: 1, platforms[0] = de763ed3
   I/hello_cl( 2393): Selected device is:
   ```

```
I/hello_cl( 2393): GPU
I/hello_cl( 2393): Device id: 5bc60350
I/hello_cl( 2393): clCreateContext passed
I/hello_cl( 2393): clCreateCommandQueue passed
I/hello_cl( 2393): clCreateProgramWithSource passed
I/hello_cl( 2393): clBuildProgram passed
I/hello_cl( 2393): Computed '1024/1024' correct values!
```

6 Adreno SDK for OpenCL

The Adreno SDK package includes numerous sample programs that demonstrate the use of OpenCL on the Adreno 3xx GPU on Android. The samples use the Android SDK and Android NDK. Each sample produces an Android app package file (APK) that can be installed and run on a Snapdragon device supporting OpenCL. This SDK can be downloaded from developer.qualcomm.com.

Table 6-1 lists OpenCL samples that are provided as part of the Adreno SDK. All samples illustrate the functional usages of openCL APIs and some samples also demonstrate optimization techniques for GPU. The samples incorporating both functional usage and optimization techniques are denoted with *.

For FFT1D, FFT2D, ImageMedianFilter, and ImageSobelFilter samples, less optimal versions of kernels are also included for comparison. Optimized kernels are associated with the QC_OPTS compiler flag and are defined by default.

Table 6-1 OpenCL samples

Sample name	Description
BandwidthTest	Reports the memory bandwidth for host→device, device→host, and device→device buffer copies
ClothSimCLGLES	Demonstrates OpenCL-OpenGL ES interoperability by computing a parallel implementation of a cloth simulation using OpenCL and writing the output results to OpenGL ES Vertex Buffer Objects (VBOs)
Concurrency*	Demonstrates concurrent use of the CPU and GPU using OpenCL; splits a matrix multiplication operation across the Adreno GPU and Krait CPU device
DeviceQuery	Reports into the log all of the OpenCL device information
DotProduct*	Performs dot product between two arrays of float4 vectors
FFT1D*	Parallel implementation of the 1D Fast Fourier Transform using a radix-2 algorithm
FFT2D*	Parallel implementation of the 2D Fast Fourier Transform based on using the radix-2 1D FFT; performs 1D FFT on the rows of a 2D array, transposes, performs 1D FFT on the columns of the 2D array, and transposes back for the final result
ImageBoxFilter*	Applies an 8x8 box filter to an input image and writes out the filtered image; demonstrates use of local memory and synchronization to increase performance on Adreno GPU
ImageMedianFilter*	Applies a 3x3 median filter to an input image and writes out the filtered image
ImageRecursiveGaussianFilter	Applies a recursive Gaussian filter to an input and writes out the filtered image
ImageSobelFilter*	Applies an edge detection Sobel filter to an input image and writes out the filtered image; converts the image to grayscale and then executes the Sobel filter

Sample name	Description
InteropCLGLES	Simple demonstration of OpenCL-OpenGL ES interoperability; outputs a sin wave to vertices of a VBO and draws the sin wave using OpenGL ES line primitives
MatrixMatrixAdd*	Performs addition to two large matrices
MatrixMatrixMul*	Demonstrates multiplication of two large matrices of size MxN and NxP; includes optimized version that uses local memory to cache intermediate fetched values
MatrixTranspose*	Performs transpose of MxN matrix and produces NxM matrix
MatrixVectorMul*	Performs multiplication of an array of 4x4 matrices with 4 component vectors
ParallelPrefixSum*	Parallel implementation of the exclusive prefix sum (scan) operation; demonstrates implementation for small and large power-of-2 arrays
ParticleSystemCLGLES	Computes a 2D particle simulation using OpenCL and renders the resultant VBOs using OpenGL ES point sprites
PostProcessCLGLES	Renders a scene to an OpenGL ES Framebuffer Object (FBO) and postprocesses the result using a Sobel edge detection filter with OpenCL; the result of the filter is written to an OpenGL ES texture that is rendered to the screen on a quad; this sample can also be compiled so it does not use cl/gl interop; see Section 6.2 for more details
VectorAdd	Performs the addition of arrays of two vectors; shows usage of CL kernel event profiling

6.1 Building and installing Adreno SDK OpenCL samples

This section covers how to build Adreno SDK OpenCL samples for Android. The Adreno SDK includes scripts to automatically build the samples. This section also describes how to build the samples manually. To build and install OpenCL samples on a Windows machine, Cygwin is required; it can be downloaded from cygwin.com.

6.1.1 Automatic build/install

The easiest way to build the Android samples is to run the script build.sh from Samples/OpenCL/Build/Android or build_android.sh under Samples/OpenCL. Running this script with no command line options from the shell builds all of the Adreno SDK samples. The default Android target to build is android-19. To specify an Android target ID, use the –at option.

To build all samples:

```
# ./build.sh
```

To build all samples for a specific target, e.g., android-14:

```
# ./build.sh –at android-14
```

To build an individual sample, use the –t command line option; e.g., to build the DeviceQuery sample:

```
# ./build.sh –t DeviceQuery
```

To build and install the sample on the device, use the –i command-line option; e.g., to build the DeviceQuery sample and install it on the connected device (using adb):

```
# ./build.sh –t DeviceQuery -i
```

The full list of command-line options for the build.sh script can be printed with the –h command line option. To install a previously built sample to the device, use the install.sh script. To install all the samples to your device, type:

```
# ./install.sh
```

To install an individual sample to the device, use the –t command-line option:

```
# ./install.sh –t DeviceQuery
```

6.1.2 Manual build/install

The samples can also be built manually. To compile a sample natively, use the ndk-build system (via Cygwin if using Windows).

1. Navigate to the Android/jni directory of the sample and run the following command:

   ```
   # $ANDROID_NDK/ndk-build
   ```

2. Go to the Android/ directory for the sample (up one from jni/) and type (where <SampleName> = the name of the sample, e.g., BandwidthTest if building the BandwidthTest sample):

   ```
   # android.bat update project -p . -n <SampleName> -t android-14'
   ```

NOTE: On Linux or MAC OS X, the command is “android” instead of “android.bat.”

   ```
   # ./InstallAssets.sh
   # ant debug
   ```

3. To install the sample on the connected device, type:

   ```
   # adb install -r bin/<SampleName>-debug.apk
   ```

6.1.3 Running Adreno SDK OpenCL samples

This section covers how to run OpenCL samples after they are installed on the device. It also describes how to run the samples automatically and manually.

6.1.3.1 Automatic run

The easiest way to run Android samples on your device is to run the script run_tests.sh. To run an individual sample, e.g., VectorAdd, type the following:

```
$ ./run_tests.sh -t VectorAdd
Deleting existing logfile
================================================================
 Running tests: VectorAdd
 Log: /logs/cl_examples/VectorAdd.log
 Timeout:  10 sec
================================================================
Issuing command to start app
adb shell  am start -S -n
com.qualcomm.VectorAdd/com.qualcomm.common.AdrenoNativeActivity -e DEVICE
gpu -e RUNTESTS true
Stopping: com.qualcomm.VectorAdd
Starting: Intent { cmp=
com.qualcomm.VectorAdd/com.qualcomm.common.AdrenoNativeActivity (has
extras) }

VectorAdd Result: PASSED
log file is /logs/cl_examples/VectorAdd.log
I/OpenCL 1.1 Embedded Samples( 1769): Parsing option DEVICE = 'gpu'
I/OpenCL 1.1 Embedded Samples( 1769): Parsing option RUNTESTS = 'true'
I/OpenCL 1.1 VectorAdd( 1769): OpenCL Platform: QUALCOMM
I/OpenCL 1.1 VectorAdd( 1769): Selected device: GPU
I/OpenCL 1.1 VectorAdd( 1769): OpenCL Device Name (0) : QUALCOMM Adreno(TM)
I/OpenCL 1.1 VectorAdd( 1769): Computed '1024' vector additions in
'0.001160' se
conds.
I/OpenCL 1.1 VectorAdd( 1769): RunTests: PASSED
```

6.1.3.2 Manual run

The samples can also be run manually from a command line. To run the sample named <SampleName>, type the following:

```
$ adb logcat -c
$ adb shell am start -n
com.qualcomm.<SampleName>/com.qualcomm.common.AdrenoNativeActivity -e
DEVICE gpu -e RUNTESTS true'
$ adb logcat | grep OpenCL
```

The following command-line options are supported by the samples:

- -e DEVICE [gpu|all]
- -e RUNTESTS [true|false]

To stop the sample, either close it from the Android user interface or type the following:

```
$ adb shell am force-stop com.qualcomm.<SampleName>
```

6.2 GL-CL interop examples

The sample set contains examples to show CL-GL interoperation. The purpose of CL-GL interoperation is to share memory between CL and GL operations. The sample PostProcessCLGLES is used to demonstrate the advantage.

6.2.1 Building and running PostProcessCLGLES

Turn off the default frame rate-limiting option by running the following command:

```
adb shell setprop debug.egl.swapinterval 0
```

This allows the app to render as fast as the GPU allows.

1. Open a shell and navigate to Samples/OpenCL/PostProcessCLGLES/Android/jni.
2. Run ndk-build clean.
3. Compile with the command:

   ```
   ndk-build
   ```

4. Go up one directory to the Android folder.
5. Copy the assets by running InstallAssets.sh or InstallAssets.bat.
6. Run ant debug and ant installed.
7. Run the app on a device and monitor the frame rate.

To build without CL-GL buffer sharing, compile step 3 with:

```
ndk-build APP_CFLAGS+=-DNO_SHARED_BUFFER
```

The frame rate is much lower for an app that is compiled without buffer sharing.

6.2.2 Using Adreno Profiler to examine differences

6.2.2.1 Adreno Profiler grapher

Because this is an OpenGL app, the first thing to examine is the frame rate.

Steps to start Adreno Profiler:

1. Start Adreno Profiler.
2. Click **Grapher** on the top tool bar to open the grapher panel.
3. In the Metrics Browser, click **Grapher Metrics** to display the metric list.
4. Expand EGL and double-click the FPS option to enable logging.
5. Start PostProcessCLGLES on the device and make sure it is running during profiling.
6. Click **Connect** on the tool bar and select the PostProcessCLGLES app from the popup dialog.
7. Let the app run for a few seconds and click **Disconnect**.
8. The grapher should plot FPS in real time.

Buttons for the previous steps are shown.

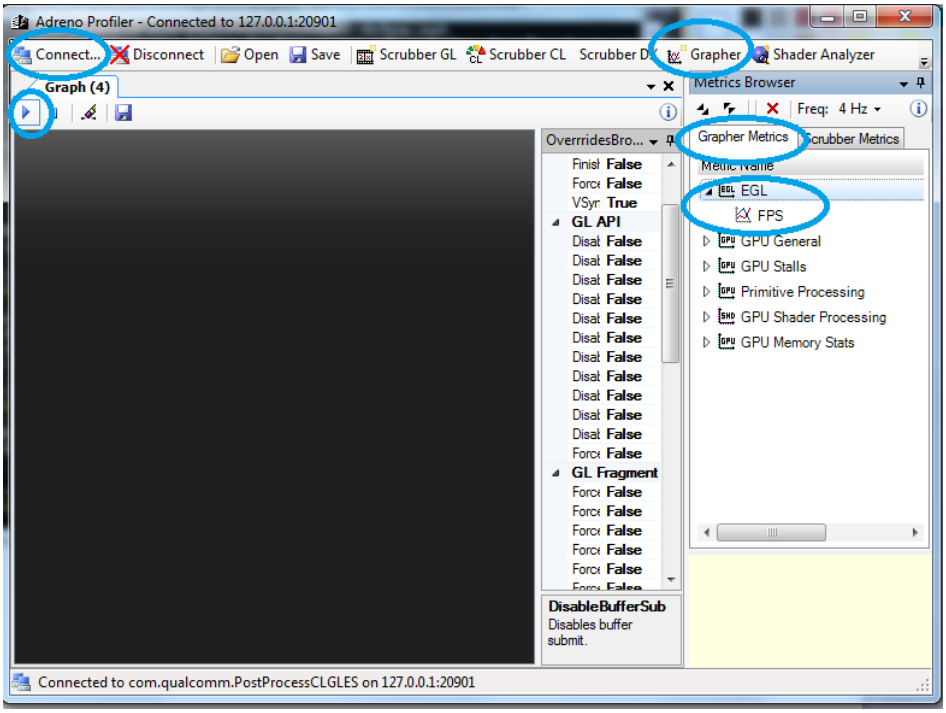


Figure 6-1 and Figure 6-2 show the differences between GL-CL interop buffer sharing and no GL-CS interop buffer sharing. The FPS range is much higher for an app that uses buffer sharing.

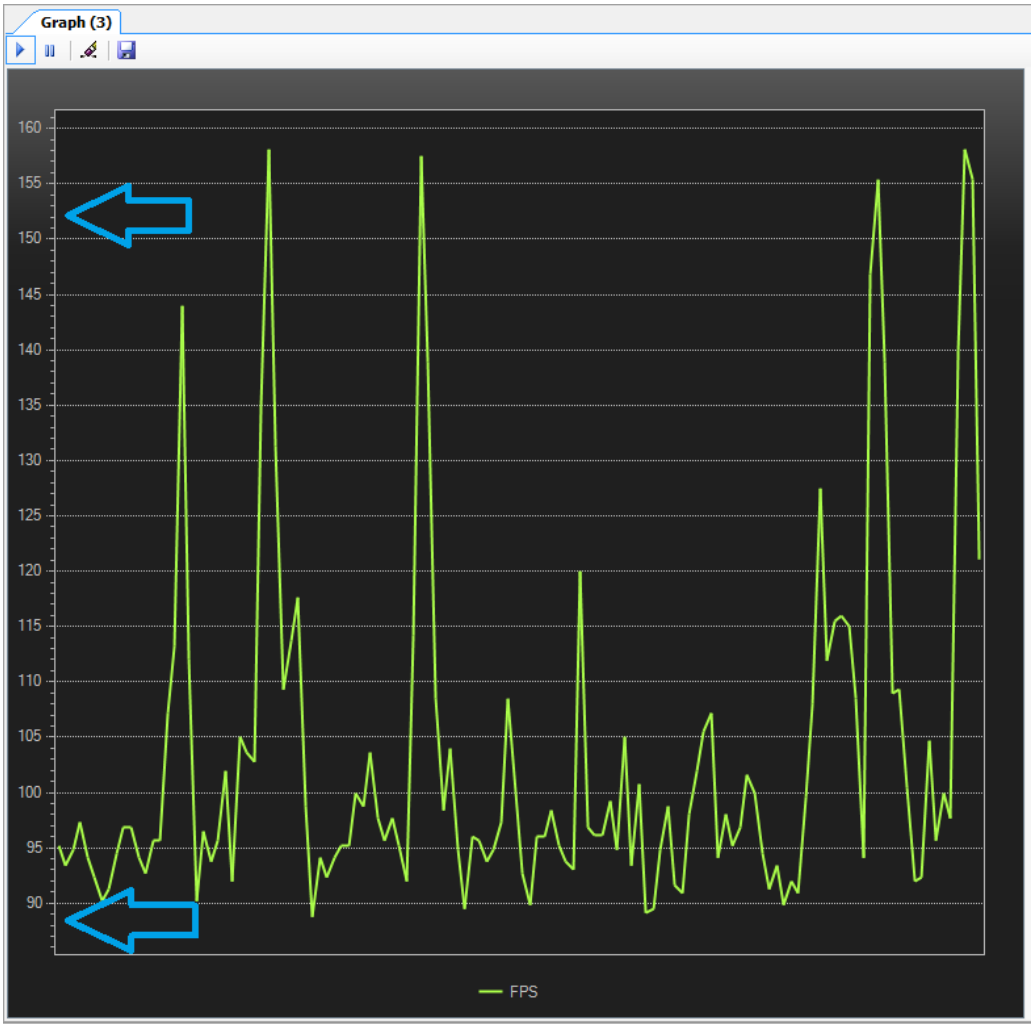


Figure 6-1 App with CL-GL interop buffer sharing, FPS in 90 to 155 range

Figure 6-2 App without CL-GL interop buffer sharing, FPS in 60 to 100 range

6.2.2.2 Adreno Profiler CL scrubber

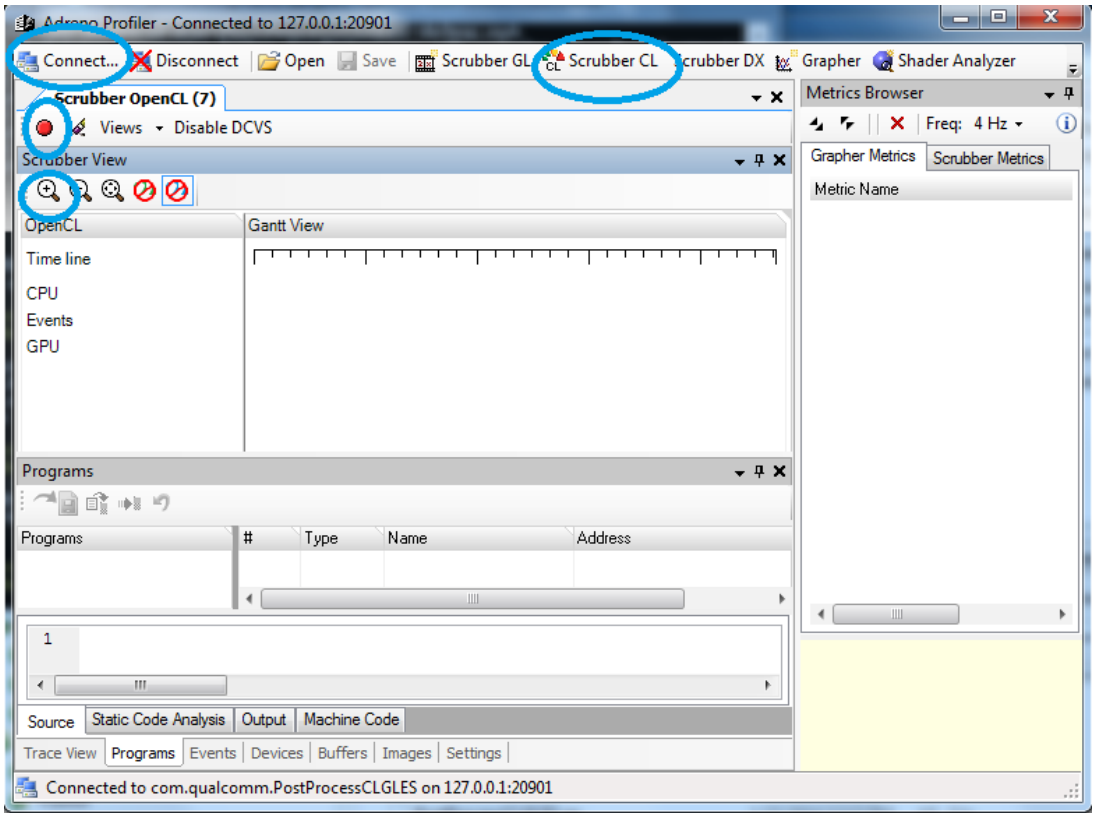
The CL scrubber captures useful data for a CL app. The API Gantt chart is discussed in this section.

1. Open a shell and run:

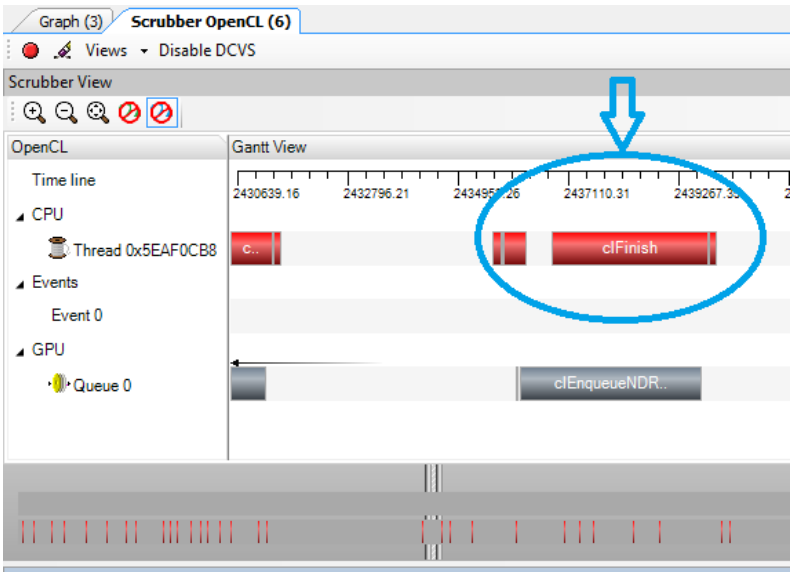
```
adb shell setprop ADRENO_PROFILER_ENABLE_OPENCL 1
```

2. Start up Adreno Profiler.
3. Click **Scrubber CL** on the top toolbar.
4. Start the PostProcessCLGLES app on the device and make sure it runs during profiling.
5. Click **Connect** and select PostProcessCLGLES from the popup dialog.
6. Click the red **Record** button in the scrubber panel.

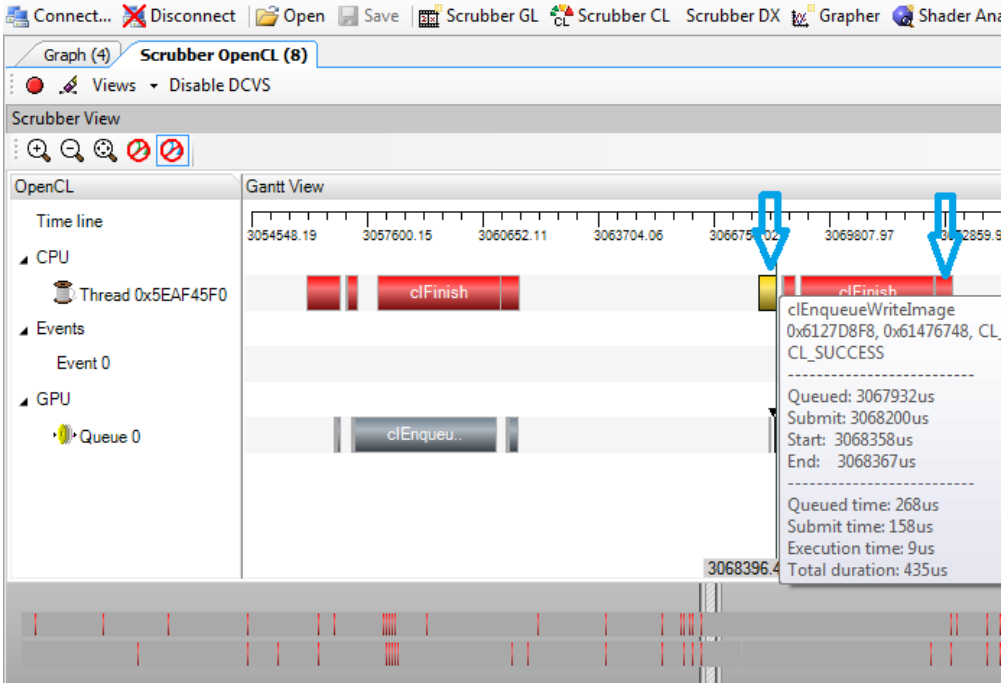
7. Let the app run for a couple of seconds and click **Record** again to stop.
8. In the Gantt chart window, continue to zoom in until the API marker bars are visible.



NOTE: No extra API call is needed to copy the buffer.



Mouse over the bar and the time spent in the copying buffer API call appears. Copying occurs for every kernel operation that increases the overall time.



6.3 Kernel profiling with clGetEventProfilingInfo

OpenCL has a built-in API to query the kernel event timestamp. These timestamps (in microseconds) can be used for kernel profiling. See the sample app VectorAdd for example usage.

Four types of events can be queried. A command's life occurs in the following order:

- CL_PROFILING_COMMAND_QUEUED – Indicates a command is queued to the host
- CL_PROFILING_COMMAND_SUBMIT – Marks submission of a command from the host to the device, most likely the GPU
- CL_PROFILING_COMMAND_START – Indicates a command starts the execution
- CL_PROFILING_COMMAND_END – Signals a command finishes the execution

The events are also presented in the Gantt chart under the CL scrubber of the Adreno Profiler. See Section 6.2.2 for an example of using the Adreno Profiler.

6.3.1 ASIC-specific remarks

- For an A3xx PL, timestamps retrieved are approximated due to dynamic clock scaling. To query the most accurate time, the best practice is to disable DCVS through Adreno Profiler under the CL scrubber tab.
- For the A4xx PL, the most current driver should support obtaining the accurate timestamp. Contact customer support for the status of the most current driver.