Qualcomm Technologies, Inc.

Qualcomm® Snapdragon™ Mobile Platform OpenCL General Programming and Optimization

80-NB295-11 A

November 3, 2017

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

Revision history

Revision	Date	Description
A	November 2017	Initial release

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document provides guidelines for OEMs, ISVs, and third-party developers for developing and optimizing OpenCL applications on the Qualcomm® Snapdragon™ 400-, 600-, and 800-based mobile platforms and chipsets.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

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

Button and key names appear in bold font, for example, click **Save** or press **Enter**.

1.3 Technical assistance

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

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

2 Introduction to OpenCL

This chapter discusses key concepts of the OpenCL standard and seeks to convey fundamental knowledge of OpenCL for application development on mobile platforms. To understand the OpenCL standard in more detail, refer to *The OpenCL Specification* in References. Developers with prior OpenCL knowledge and experience may skip this chapter and move to the next ones.

2.1 OpenCL background and overview

Developed and maintained by the Khronos group, OpenCL is an open and royalty free standard for cross platform parallel programming in heterogeneous systems. It is designed in a way that helps developers to exploit the massive computing power available in modern heterogeneous system and greatly facilitate application development across platforms.

Qualcomm® Adreno™ GPU series on Snapdragon platforms have been one of the earliest mobile GPUs that fully support OpenCL.

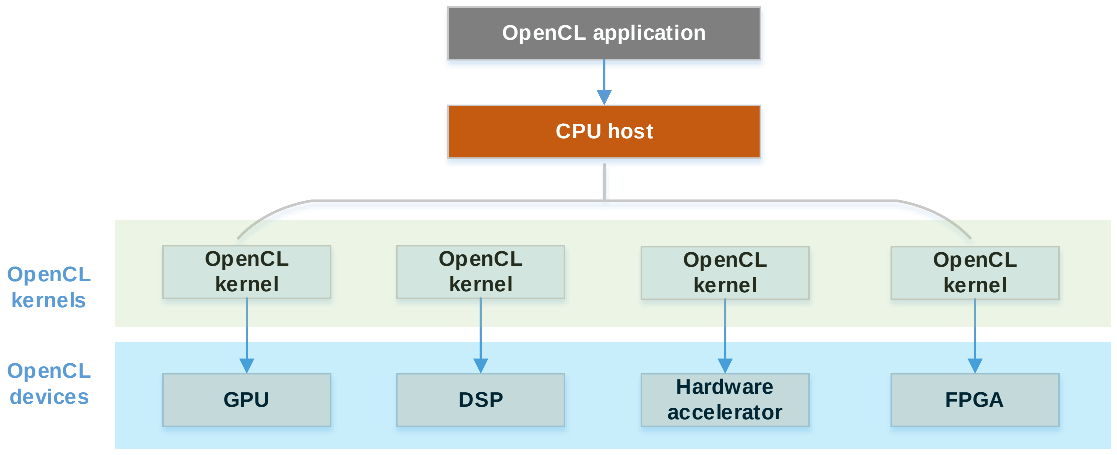

Figure 2-1 Heterogeneous system using OpenCL

Figure 2-1 shows a typical heterogeneous system that supports OpenCL. In this system, there are mainly three parts:

- A host CPU that is essentially a commander/master that manages and controls the application.
- Multiple OpenCL devices, including GPU, DSP, FPGA, and a hardware accelerator.
- Kernel codes which are compiled and loaded by the host to OpenCL devices to execute.

2.2 OpenCL on mobile

In recent years, the mobile system-on-chips (SOCs) have advanced significantly in computing power, complexity, and functionality. GPUs in the mobile SOCs (mobile GPUs) are very powerful and some of the top mobile GPUs reach the level of console/discrete GPUs in terms of raw computing power.

This poses challenges for developers: how to effectively make use of the computing power and quickly develop applications without knowing the low-level details of the GPUs, while maintaining application compatibility across different SOCs?

Created to tackle these problems, OpenCL allows developers to easily leverage computing power of mobile SOCs thanks to its cross-platform support. By using OpenCL, mobile SOCs can easily enable advanced use cases in many fields, such as image/video processing, computer vision, machine learning, etc.

In QTI, many use cases have been successfully accelerated using OpenCL with Adreno GPUs, which demonstrated excellent performance, power, and portability. It is highly recommended to use OpenCL with GPUs to accelerate their applications for Snapdragon SOCs.

2.3 OpenCL standard

The OpenCL standard primarily contains two components, the OpenCL runtime API and the OpenCL C language. The API defines a set of functions running on host for resource management, kernel dispatch and many other tasks, while the OpenCL C language is used to write kernels that execute on OpenCL devices. The OpenCL API and OpenCL C language will be reviewed in the following sections.

2.3.1 OpenCL API functions

The OpenCL API functions can be classified into two categories, platform layer and runtime. Table 2-1 and Table 2-2 summarize the high-level functionality of the platform layer and the runtime layer respectively.

Table 2-1 OpenCL platform layer functionality

Functionality	Details
Discover the platform	Is there an OpenCL platform available?
Discover OpenCL devices	Is OpenCL available on GPU, CPU, or other devices
Query the OpenCL device information	Global memory size, local memory size, maximum workgroup size, etc. Also check the extensions supported by the device.
Context	Context management, such as context creation, retain, and releases

Table 2-2 OpenCL run time layer functionality

Functionality	Details
Command queue management	Used to communicate between device and host and can have many queues in an application
Create and build OpenCL programs and kernels	Is the kernel loaded and built successfully
Prepare data for kernel to execute, create memory objects and initialize them	What memory flag to use? Is there a way to do zero copy memory object creation?
Create a kernel call and submit it to the compute device	What workgroup size to use?
Synchronization	Memory consistency
Resource management	Deliver results and release resources

Understanding of the two layers are essential for writing OpenCL applications. Refer to References for more details.

2.3.2 OpenCL C language

As a subset of the C99 standard, the OpenCL C language is used to write kernels that can be compiled and executed on devices. Developers with C language programming experience can easily get started with OpenCL C programming. However, it is crucial to understand the differences between the C99 standard and the OpenCL C language to avoid common mistakes. Here are the two key differences:

- Some features in C99 are not supported by the OpenCL C language due to hardware limits and the OpenCL execution model. Examples are function pointers and dynamic memory allocation (`malloc/calloc`, etc.).
- The OpenCL C language extends the C99 standard in several aspects so that it can better serve its programming model and facilitate development, for example:
  - It adds built-in functions to query the OpenCL kernel execution parameters.
  - It has image load/store functions that can leverage the GPU hardware.

2.3.3 OpenCL versions and profiles

The current OpenCL v2.2 with the provisional SPIR-V 1.2 standard contains many improved features. Refer to References for more details.

OpenCL defines two profiles, embedded profile and full profile. Embedded profile mainly targets mobile devices, which typically have lower precision capability, and fewer hardware features than traditional computing devices such as desktop GPUs. For a list of the key differences between embedded and full profile, refer to References.

2.4 OpenCL portability and backward compatibility

2.4.1 Program portability

As a strictly defined computing standard, OpenCL has good program portability. OpenCL applications written for one vendor’s platform should run well on other vendors’ platforms, if they are not using any vendor-proprietary or platform-specific extensions or features.

The program portability of OpenCL is ensured by Khronos’ certification program, which requires OpenCL vendors to pass rigorous conformance tests on their platform before they claim it is OpenCL “conformant.”

2.4.2 Performance portability

Unlike program portability, OpenCL performance is generally not portable. As a high-level computing standard, the hardware implementation of OpenCL is vendor dependent. Different hardware vendors have different device architectures and each one may have its advantages and disadvantages. As a result, an OpenCL application written and optimized for one vendor’s platform is unlikely to have the same performance as on other vendors’ platforms.

Even for the same vendors, different generations of their GPU hardware may vary in micro-architectures and features, which could lead to noticeable performance differences for OpenCL programs. As a result, applications optimized for older generations of hardware often require fine-tuning to exploit the full capacity of newer generations.

2.4.3 Backward compatibility

OpenCL fully embraces backward compatibility to ensure the investment on old code can run on new versions of OpenCL with no problems. Note that as some API functions may be deprecated in newer versions, the macros `CL_USE_DEPRECATED_OPENCL_1_1_APIS` or `CL_USE_DEPRECATED_OPENCL_1_2_APIS` need to be defined if the OpenCL 1.1 or 1.2 deprecated APIs are used with the OpenCL 2.x header file.

OpenCL extensions are not guaranteed to be carried forward to new devices, so applications using extensions need to examine if the new devices support them.

3 OpenCL on Snapdragon

Snapdragon is one of the most powerful and widely used mobile platforms in today's Android operating system and the Internet of Things (IOT) market. The Snapdragon mobile platform brings together 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. Generally used for rendering graphics applications, Adreno GPUs in Snapdragon processors are also powerful general-purpose processors capable of handling many computationally intensive tasks, such as image and video processing, and computer vision. The capabilities of the GPU can be leveraged using OpenCL to perform data-parallel computations.

3.1 OpenCL on Snapdragon

OpenCL is fully supported on the Adreno A3x, A4x, and A5x GPUs, and is fully conformant with the OpenCL standard. OpenCL has different versions and profiles, and different Adreno GPUs may support different OpenCL versions, as shown in Table 3-1.

Table 3-1 Adreno GPUs with OpenCL support

GPU series	Adreno A3x	Adreno A4x	Adreno A5x
OpenCL version	1.1	1.2	2.0
OpenCL profile	Embedded	Full	Full

Besides the difference in OpenCL versions and profiles, there are other properties that may vary across Adreno GPUs, such as the supported extensions and maximum dimensions of image objects, etc. A full list of the details can be obtained by calling the OpenCL API function `clGetDeviceInfo`.

3.2 Adreno GPU architecture

This section provides a high-level overview of the Adreno architecture relevant to OpenCL.

3.2.1 Adreno high-level architecture for OpenCL

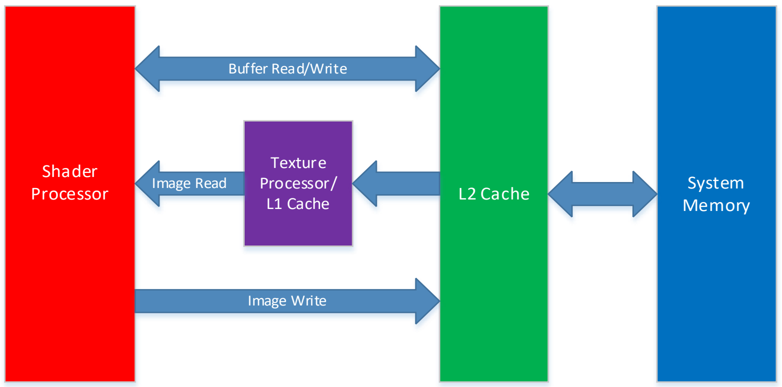

Figure 3-1 High-level architecture of the Adreno A5x GPUs for OpenCL

Adreno GPUs support many graphics and compute APIs, including the OpenGL ES, OpenCL, DirectX, and Vulkan, etc. Figure 3-1 illustrates a high-level view of the Adreno A5x hardware architecture for OpenCL, where the graphics related hardware modules are skipped. There are many differences between A5x and other Adreno GPUs, while for OpenCL the differences are relatively minor.

The key hardware modules for OpenCL execution are as follows:

- Shader (or streaming) processor (SP)
  - Core block of Adreno GPUs. Contains many hardware modules, including arithmetic logic unit (ALU), load/store unit, control flow unit, register files, etc.
  - It executes graphics shaders (e.g., vertex shader, fragment shader, and compute shader) and compute workload such as OpenCL kernels.
  - Each SP corresponds to one or more OpenCL Compute Units.
  - Adreno GPUs may contain one or more SPs, dependent on GPU series and tiers. A low-tier chipset may have a single SP, while a high or premium tier chipset may have more SPs. In Figure 3-1, there is only one SP.
  - SPs load and store data through L2 cache for buffer objects and image objects defined with the `__read_write` qualifier (OpenCL 2.0 feature).
  - SPs load data from texture processor/L1 module for read-only image objects.
- Texture processor (TP)
  - Performs texture operations, such as texture fetch and filtering based on kernel's request.
  - TP is coupled with L1 cache, which fetches data from UCHE in case of texture data cache miss.
- Unified L2 Cache (UCHE)

  - Responds to SP's load/store requests for buffer objects and L1 cache's data load requests for image objects.

3.2.2 Waves and fibers

In Adreno GPUs, the smallest unit of execution is called a fiber. One fiber corresponds to one work item in OpenCL. A collection of fibers which always execute in lock-step is called a wave. The SP can accommodate multiple active waves at a time. Each wave can generally make independent forward progress, irrespective of the status of the other waves. Note that:

- Wave size, or the number of fibers in a wave, is generally fixed for a given GPU and kernel.
- Wave size in Adreno GPUs depends on the GPU series and tiers as well as the compiler; values could be 8, 16, 32, 64, 128, etc.
- A workgroup may be executed by one or multiple waves, dependent on the workgroup size. For example, one wave is sufficient if the workgroup size is less than or equal to the wave size. Typically more waves are better as they allow better latency hiding.
- SP can execute ALU instructions on one or more waves simultaneously.
- The maximum number of waves that can be pipelined in the workgroup is hardware dependent. Typically, Adreno GPUs support up to 16 waves.
- Given a kernel, the maximum number of active waves in one SP depends on the kernel's register footprint and register file size, which again depends on GPU series and tiers.
- Generally, the more complex the kernel is, the fewer active waves.
- Given a kernel, the maximum workgroup size is the product of the maximum allowed number of waves and the wave size.

OpenCL 1.x does not expose the concept of waves, while OpenCL 2.0 allows applications to use it through the `cl_khr_subgroups` extension, which is supported from the Adreno A5x GPUs.

3.2.3 Latency hiding

Latency hiding is one of the most powerful characteristics of GPU for efficient parallel processing, and enables GPU to achieve high throughput. Here is an example:

- SP starts to execute the 1st wave.
- After a few ALU instructions, this wave requires additional data from external memory (could be global/local/private memory) to proceed, which is not available.
- SP sends data fetch requests for this wave.
- SP switches execution to the 2nd wave which is ready to execute.
- SP continues to execute the 2nd wave to a point where external dependency is not ready.
- SP may switch to the 3rd wave, or switch back to the 1st wave, if the data for the 1st wave is available.

In this way, SP is mostly busy and working like "full time" as the latency, or the dependency, can be well hidden.

3.2.4 Workgroup assignment

A typical OpenCL kernel launches multiple workgroups. In Adreno GPUs, each workgroup is assigned to an SP, and each SP typically processes one workgroup at a time. The remaining workgroups, if there is any, are queued in GPU for execution.

Take the 2DRange in Figure 3-2 as an example, and assume this is a GPU with 4 SPs. Figure 3-3 shows how the workgroups are allocated to different SPs. In this example, there are 9 workgroups in total and each of them is executed by one SP. And there are four waves per workgroup, and the wave size is 16.

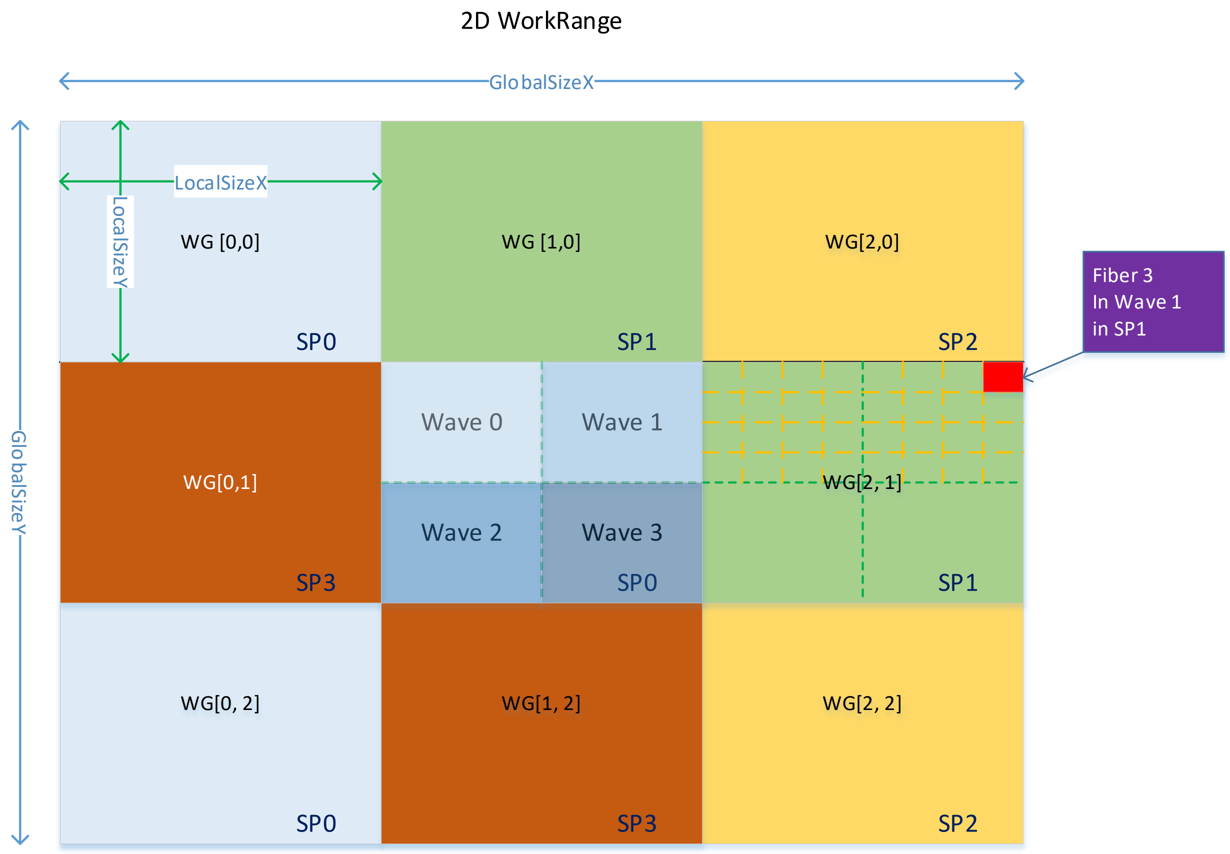

Figure 3-2 An example of workgroup layout and dispatch in Adreno GPUs

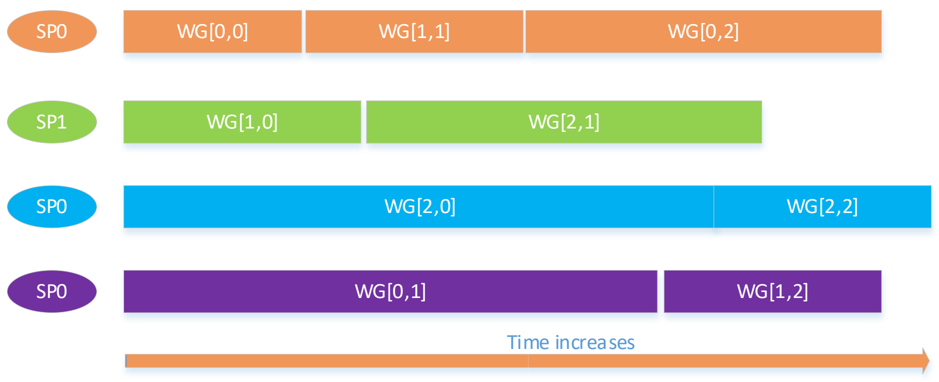

Figure 3-3 An example of workgroup allocation to SPs

The OpenCL standard neither defines the order of workgroup launching/execution, nor the method for workgroup synchronization. For Adreno GPUs, developers cannot assume that workgroups are launched in certain orders on the SPs. It is also true for waves.

In most Adreno GPUs, one SP can only process one workgroup at a time, and the workgroup must be completed before another one can be started. While in high tier or newer series, such as the Adreno A540 GPU, multi-workgroup execution per SP is supported.

3.3 Adreno A3x, A4x, and A5x differences on OpenCL

Each new series of Adreno GPUs brings numerous improvements to OpenCL features and performance. This section discusses key changes that affect OpenCL performance.

3.3.1 L2 cache

The L2 cache architecture has been improved significantly for better efficiency and performance from the Adreno A320 and A330 GPUs to the Adreno A420, A430, A530 and A540 GPUs, in addition to size increases.

3.3.2 Local memory

Local memory has been improved from the Adreno A3x to A4x and A5x series, including the size capacity, load/store throughput, and coalesced access. Table 3-2 shows the difference of coalesced access on different series.

Table 3-2 Local memory performance summary

GPUs	Adreno A3x	Adreno A4x	Adreno A5x
Coalesced	No	No	Yes, load/store up to 128 bits by up to 4 work items per operation

Coalesced access is an important concept for OpenCL and GPU parallel computing. Basically, it refers to the case where the underlying hardware can combine and merge the data load/store requests by multiple work items into one request, so that the data load/store efficiency is improved. Without coalesced access support, the hardware must perform the load/store operation per each individual request, resulting in inferior performance.

Figure 3-4 illustrates the difference between coalesced data load vs. non-coalesced. To combine the requests from multiple work items, the addresses of the data generally need to be consecutive. In the coalesced case, Adreno GPUs can load data for four work items in one transaction, while without coalesced it would take four transactions for the same amount of data.

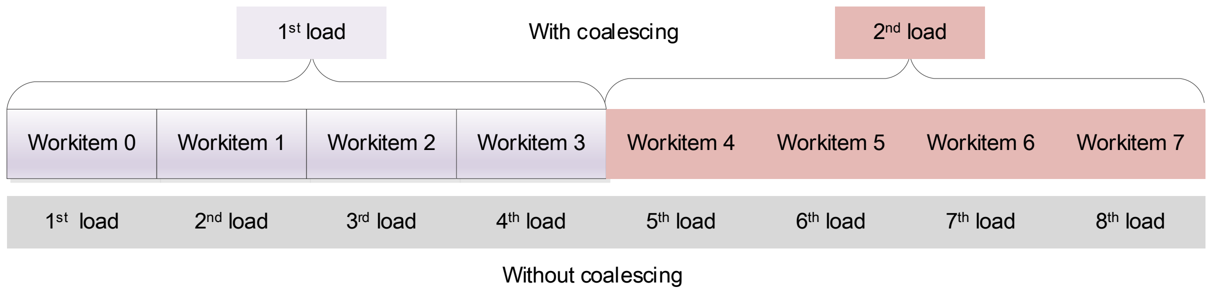

Figure 3-4 Illustration of coalesced vs. non-coalesced data load

3.4 Context switching between graphics and compute workload

3.4.1 Context switch

In Adreno GPUs, if a high priority task, such as graphics user interface (UI) rendering, is required while a low priority workload is running on GPU, the latter one could be forced to pause so that GPU switches to the high priority workload. When the high priority task is completed, the lower one is resumed. This type of workload switch is called *context switch*. Context switch is generally expensive, as it requires complex hardware and software operations. However, it is an important feature to enable the emerging and advanced timing critical tasks such as automobile applications.

3.4.2 Limit kernel/workgroup execution time on GPU

Sometimes a compute kernel may be running for an excessive period and trigger an alert that causes the GPU to be reset. To avoid unexpected behaviors, it is discouraged to have compute kernels with workgroups that take too long to complete. Usually the UI rendering on Android devices occurs constantly, e.g., every 30 milliseconds, and a long-running compute kernel could cause UI to be lagging and unresponsive, and therefore hurt user experience. The ideal execution time would be case dependent. However, a general rule of thumb is that the kernel execution time should be in the range of tens of milliseconds.

3.5 OpenCL standard related improvement

The Adreno A3x GPUs support OpenCL 1.1 embedded profile, while the Adreno A4x GPUs support OpenCL 1.2 full profile, and the Adreno A5x GPUs support OpenCL 2.0 full profile.

From the OpenCL 1.1 embedded profile to the OpenCL 1.2 full profile, the majority changes are on software rather than hardware, such as improved API functions.

From the OpenCL 1.2 full profile to the OpenCL 2.0 full profile, however, there are many new hardware features introduced, such as the shared virtual memory (SVM), kernel-enqueue-kernel, etc. Table 3-3 lists the major differences on OpenCL profile support across the three Adreno GPUs.

Table 3-3 Standard OpenCL features supported in Adreno GPUs

Features	OpenCL 1.1 Embedded Adreno A3x	OpenCL 1.2 Full Adreno A4x	OpenCL 2.0 Full Adreno A5x
Separate compilation and linking of objects	No	Yes	Yes
Rounding mode	Rounding to zero	Rounding to nearest even	Rounding to nearest even
Built in kernels	No	Yes	Yes
1D texture, 1D/2D image array	No	Yes	Yes
Shared virtual memory	No	No	Yes (coarse grain only)
Pipe	No	No	Yes
Load-store image	No	No	Yes
Nested parallelism	No	No	Yes
Kernel-enqueue-kernel (KEK)	No	No	Yes
Generic memory space	No	No	Yes
C++ atomics	No	No	Yes

3.6 OpenCL extensions

In addition to supporting the core OpenCL functionality, the Adreno OpenCL platform supports many additional features through extensions, which improve the OpenCL usability and expose advanced hardware capabilities in Adreno GPUs. The extensions available on a given Adreno GPU may be queried using `clGetPlatformInfo`. Documentation for these extensions is available on the QTI Developer Network website (https://developer.qualcomm.com).

4 Adreno OpenCL application development

This chapter briefly discusses some basic requirements for Adreno OpenCL application development, followed by how to debug and profile applications.

4.1 OpenCL application development on Android

Currently, Adreno GPUs support OpenCL mainly on the Android operating system (OS) and on select Linux systems. To develop an Android app that runs with OpenCL, developers need to get familiar with the Android software development kit (SDK) and the native development kit (NDK). Refer to https://developer.android.com/index.html and https://developer.android.com/ndk/index.html for Android SDK and NDK respectively.

Throughout this chapter and the following chapters, it is assumed that the development is on Android platform and the developers have experience on Android SDK and NDK. The app development on Linux should be similar.

There are several prerequisites for OpenCL development on Snapdragon platform:

- Snapdragon devices with OpenCL support. Not all Snapdragon devices support OpenCL. Refer to Table 3-1 for more details.
- OpenCL software. OpenCL on Adreno GPUs relies on QTI proprietary libraries.
  - Check if the device has the OpenCL libraries installed.
    - The core library is **libOpenCL.so**, which is usually located at */vendor/lib* on device.
  - Some vendors may choose not to include the OpenCL software (for example, Google's Nexus and Pixel devices).
- OpenCL must run at the NDK layer.
- Root access privilege is not necessary for development and testing, but it may be required for running the SOCs in Performance mode.

Table 4-1 summarizes the key requirements for OpenCL development with Adreno GPUs.

Table 4-1 Requirements of OpenCL development with Adreno GPUs

Items	Requirements	Note
Devices	Adreno A3x/A4x/A5x GPUs	
Operating system	Android, Linux	Only select Linux platforms support OpenCL.
Device software requirement	libOpenCL.so on device	Some devices may not have it
Development requirement	Adreno NDK/SDK	OpenCL code needs to run at NDK layer
Root privilege on device	Not required generally	Required for performance mode

4.2 Debugging tools

Debugging OpenCL kernel is often challenging due to the parallel nature of GPU execution. The Adreno GPUs support the `printf` function inside kernels, which is very useful for debug. To use `printf`, it is recommended to reduce the workload, print variables with conditions, and avoid printing out too many variables, as `printf` slows down code execution. For example, one may only enable the problematic workgroup, or even the single problematic work item (by setting proper offsets in the function `CLEnqueueNDRangeKernel`)

It is important to know the software version of the device as some bugs or issues may have been fixed in the newer releases. To query the software (driver) and compiler version, an API function called `clGetDeviceInfo` can be used. For more details, refer to References.

4.3 Snapdragon Profiler

The Snapdragon Profiler is a profiling tool provided by QTI that runs on Windows, Mac, and Linux platforms and allows developers to analyze CPU, GPU, DSP, memory, power, thermal, network data of Snapdragon processors running Android. It supports OpenCL and many graphics APIs, such as OpenGL ES and Vulkan. For more details, refer to https://developer.qualcomm.com/software/snapdragon-profiler.

The following are some key features offered by the Snapdragon Profiler for OpenCL profiling.

- The profiler has a kernel analyzer which allows developers to do static analysis for a given kernel. It provides information such as register footprint, total instructions, and the number instructions for each type of operations, etc., to help developers better optimize kernels.
- The profiler provides OpenCL API traces and logs for a given OpenCL application. It allows developers to identify and resolve bottlenecks from the API level, as well as debug the application.
- The profiler provides information such as GPU busy ratio, ALU utilization ratio, L1/L2 cache hit ratio, etc., which is essential for developers to identify performance issues in kernels.
- The profiler supports command line based applications, as well as Android GUI apps.

4.4 Performance profiling

Given an app, it is critical to profile its performance accurately. Two commonly used methods, CPU timer and GPU timer, and their key differences are discussed in the following sections.

4.4.1 CPU timer

CPU timer is used for measuring the full execution time of OpenCL calls from the host side. This can be achieved by using any of the date and time functions that are part of the standard library of the C/C++ programming language. An example is to use `gettimeofday` as follows:

```
#include <time.h>
#include <sys/time.h>
void main() {
struct timeval start, end;
gettimeofday(&start, NULL); /*get the start time*/
/*Execute function of interest*/ {      . . .
```

```
        clFinish(commandQ);
 }
gettimeofday(&end, NULL); /*get the end time*/
/*Print the total execution time*/
printf("%ld\n", ((end.tv_sec * 1000000 + end.tv_usec)
 - (start.tv_sec * 1000000 + start.tv_usec)));
}
```

The OpenCL runtime enqueue API functions can be categorized to blocking calls and nonblocking calls. And for nonblocking calls CPU timer must be used with care:

- Nonblocking call means that the host proceeds to the next instruction after its submission (which usually is queued for execution in another CPU thread), rather than wait for the function call to complete.
  - The kernel execution API function, `clEnqueueNDRangeKernel`, is a nonblocking function.
- For nonblocking calls, the real execution time is not the time difference between the function call.

When using a CPU timer to measure the kernel execution time from host side, one must make sure the function is complete by using either the `clWaitforEvent` call (if there is an event ID for the nonblocking call), or `clFinish`. The same rule applies to the memory transfer calls.

4.4.2 GPU timer

All OpenCL enqueue function calls optionally return an event object to the host, which can be used by the OpenCL profiling APIs to query the execution time. Adreno GPUs have their own clock and timer to measure the function execution flow, and the GPU execution time is determined by GPU hardware counters that are independent of the operating system.

To enable the GPU timer functionality, the `CL_QUEUE_PROFILING_ENABLE` flag needs to be set in the property argument of either `clCreateCommandQueue` or `clSetCommandQueueProperty` for the current command queue. Also, an event object must be provided to the enqueue function. Once the function is completed, the API function `clGetEventProfilingInfo` shall be used to obtain the profiling information of the command execution.

For a `clEnqueueNDRangeKernel` call, using the `clGetEventProfilingInfo` function with the four profiling parameters, including `CL_PROFILING_COMMAND_(QUEUED, SUBMIT, START,` and `END)`, can provide an accurate picture of the kernel launch latency and kernel execution time in Adreno GPUs, as shown in Figure 4-1.

- The difference between the first two parameters, `CL_PROFILING_COMMAND_(QUEUED` and `SUBMIT)`, gives an idea of the software overhead, and the overhead of CPU cache operations. The OpenCL software may choose to queue the kernel first, and submit it along with several following kernels in the queue later, for example, when the number of kernels in the queue is large enough. Developers may use the `clFlush` function to speed up the submission.
- The difference between `CL_PROFILING_COMMAND_(SUBMIT` and `START)` can give an idea of many other jobs GPU is processing.
- The actual kernel execution time on GPU is the difference between `CL_PROFILING_COMMAND_(START` and `END)`.

Developers should focus on minimizing the actual kernel execution time. This can be improved relatively easier as compared to the other two timers that are typically hard to control.

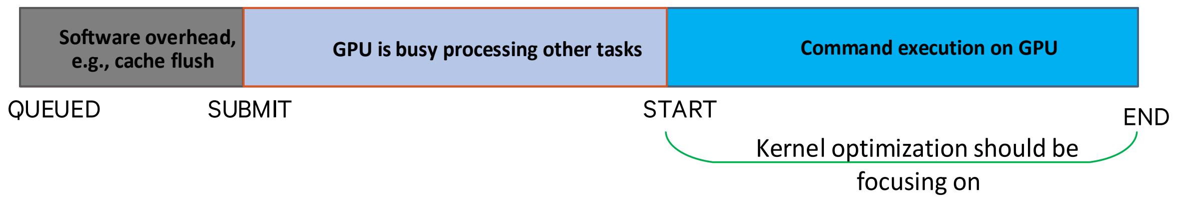
Software overhead, e.g., cache flush
GPU is busy processing other tasks
Command execution on GPU
QUEUED
SUBMIT
START
END
Kernel optimization should be focusing on

Figure 4-1 Profiling flags for the `clEnqueueNDRange` call in Adreno GPUs

4.4.3 GPU timer vs. CPU timer

As both GPU and CPU timer can be used to profile the performance, which one should be used for an application? Though GPU timer can accurately measure the GPU execution time, some hardware operations (e.g., cache flush) and some software operations (e.g., synchronization between CPU host and GPU) are out of the GPU clock system. As a result, the GPU timer is likely to report a better performance number than CPU timer for kernel execution. Here are the two practices recommended:

- GPU timer should be used to measure the kennel optimization. GPU timer can tell exactly how much improvement is achieved by the optimization steps from a GPU execution perspective.
- CPU timer should be used to measure end-to-end performance for the whole application. This is important if the OpenCL program is only part of a whole application pipeline.

4.4.4 Performance mode

Snapdragon SOCs have advanced dynamical clock and voltage control mechanism which automatically controls the system so that it runs at power saving mode to save battery under certain scenarios. Typically, if there is intensive workload, the system may automatically raise clock rate and voltage, pushing the device into so-called *performance mode* to boost the performance and meet the workload demand.

For OpenCL optimization, it would be difficult to understand and profile the performance if the system dynamically changes clock rate. Therefore, it is recommended to enable performance mode for the sake of profiling consistence and accuracy.

In the absence of performance mode settings, the first OpenCL kernel in a sequence typically shows greater launch latency and slower execution time. One may need to use some simple kernels to warm up the GPU before launching the real GPU workload.

The performance of an OpenCL kernel is not solely dependent on GPU. The API functions running at CPU are as critical as the kernel execution on GPU. To achieve the best performance, both CPU and GPU should have performance mode enabled. In addition, to reduce the interference from UI rendering, it is recommended that:

- 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 that `SurfaceFlinger` is not running on Android. This ensures that the CPU and GPU are being solely used by the application being profiled.

The sequence of commands needed to enable performance mode are slightly different for the Adreno A3x, A4x and A5x GPUs. Refer to Section A for more details.

4.4.5 GPU frequency controls

The application can leverage the `cl_qcom_perf_hint` extension to control GPU frequency. This extension allows the application to set a performance hint property when creating the OpenCL context. The performance level can be `HIGH, NORMAL` and `LOW`. The `NORMAL` perf level leaves the dynamic clock and voltage control enabled. The `HIGH` and `LOW` performance levels disable the dynamic control and force the GPU to run at their maximum and minimum frequencies respectively.

NOTE: The performance levels are just a hint. The driver attempts to respect these hints but factors such as thermal controls or external applications or services can override these hints. The perf hint extension gives the application some flexibility in the power/performance tradeoff. However, it should be used carefully as it has major implications for SOC level power consumption.

5 Overview of performance optimizations

This chapter provides a high-level overview of OpenCL application optimization. More detailed discussions can be found in the next few chapters.

NOTE: Optimization of an OpenCL application can be challenging. It often requires considerably more efforts than initial development..

5.1 Performance portability

As discussed in Section 2.4.2, OpenCL generally does not have good performance portability across different architectures. OpenCL applications that have been optimized on other platforms, especially on discrete GPUs, are unlikely to perform well on Adreno GPUs. The programming guide and best practices from other OpenCL vendors may not be applicable for Adreno GPUs at all. Therefore, it is extremely important to read through this entire document for optimization work on Adreno GPUs. Also, an OpenCL application optimized for one Adreno GPU may need extra tuning or optimization to achieve optimal performance on other Adreno GPUs.

5.2 High-level view of optimization

Optimization of an OpenCL application can be roughly categorized into the following three levels from top to bottom:

- Application/algorithm
- API functions
- Kernel optimization

An OpenCL optimization problem is essentially a problem of how to optimally utilize the memory bandwidth and computing power, including

- The optimal ways to use global memory, local memory, registers and caches, etc.
- The optimal ways to leverage computing resources such as the ALU and texture operations.

The application level optimization strategy is addressed in the remaining sections of this chapter. Other levels are presented in the following chapters.

5.3 Initial evaluation for OpenCL porting

It is important for developers to assess whether an application is suitable for OpenCL prior to porting it blindly. Following are the typical characteristics of a good candidate for OpenCL acceleration on GPU:

- Large input data set
  - The overhead between CPU and GPU may overshadow the performance gain of OpenCL for small input data sets.
- Computationally intensive
  - GPUs have many computing units (ALUs) and its peak computing power, *gflops*, is usually a lot higher than CPU. To fully utilize the GPU, applications should have reasonably high computational complexity.
- Parallel computation friendly
  - The workload may be partitioned into independent small units, and processing of each unit does not affect the others.
  - Parallelized task is needed to fully utilize GPU's capability for memory latency hiding, a key benefit of using GPU.
- Limited divergent control flow
  - GPU is not designed to handle divergent control flow as efficiently as CPU. If the use case requires a lot of conditional check and branching operations, CPU may be more suitable.

5.4 Port CPU code to OpenCL GPU

Typically, developers may already have a CPU based reference program for OpenCL porting. Assume the program consists of many small functional modules. While it seems convenient to convert each module to an OpenCL kernel on a one-by-one mapping basis, the performance is unlikely to be optimal. It is important to consider the following factors:

- In some cases, merging multiple CPU functional modules into one OpenCL kernel can lead to better performance if doing so reduces data traffic between GPU and memory.
- In some cases, splitting a complex CPU functional module into multiple simpler OpenCL kernels can yield better parallelization of individual kernels and better overall performance.
- Developer may need to modify data structures to tailor the data flow in a way that can reduce overall data traffic.

5.5 Parallelize GPU and CPU workloads

To fully utilize the compute power of the SOC, the application may delegate certain tasks to the CPU while the GPU is executing a kernel. Here are a few points to consider when designing such topology and allocating the workload:

- Allow the CPU to run the part that is best suitable for the CPU, such as divergent control flow and sequential operations.
- Avoid situations where the GPU is idle and waits for the CPU to complete, or vice versa.
- Data sharing between the CPU and GPU can be expensive. Instead, try shifting lightweight CPU tasks to the GPU, even though it may not be GPU friendly, to eliminate the need for data transfer.

5.6 Bottleneck analysis

It is crucial to identify and analyze bottlenecks, as this leads to focus areas for optimization. Bottlenecks cause stalls and are often the slowest stages in the application. No matter how efficient other stages are, performance of the application is limited by its slowest stages, i.e., the bottlenecks. It may not make sense to pay any attention to areas until the bottleneck is resolved.

5.6.1 Identify bottlenecks

Typically, a kernel is either memory-bound or computation-bound (also known as ALU bound). One simple trick is to manipulate the kernel codes and run on device as follows:

- If adding a lot more computing does not change performance, it may not be compute bound.
- If loading excessive data does not change performance, it may not be memory bound.

The Snapdragon Profiler, as discussed in Section 4.3 can be used to identify the bottlenecks.

5.6.2 Resolve bottlenecks

Once the bottleneck is identified, different strategies can be used to resolve it:

- If this is an ALU bound problem, find ways to reduce the complexity and the number of calculations, such as using fast relaxed math or native math where the precision requirements are not high, and using 16-bit floating point format instead of 32-bit floating point format.
- If this is a memory bound problem, try to improve memory access, such as vectorizing load/store, utilizing local memory or texture cache (e.g., use read-only image object in place of buffer object). Using shorter data types to load/store data between GPU and global memory can be beneficial for saving memory traffic.

Details are described in following chapters.

NOTE: The bottleneck could shift as optimization progresses. A memory bound problem could become an ALU bound problem if the memory bottleneck is resolved, or vice versa. Many back and forth iterations are necessary to obtain the optimal performance.

5.7 API level performance optimization

The OpenCL API functions are executed on the CPU host to manage resources and control application execution. Although, in general, API functions are lighter than kernel execution from computational complexity perspective, improper use of API functions could lead to big performance penalty. Here are a few points that can help developers avoid some common pitfalls.

5.7.1 Proper arrangement of API function calls

Expensive API functions should be properly placed so that they do not block or affect the launching of workload to GPU. Some OpenCL API functions take a long time to execute and should be called outside of the execution loop. For example, following functions can take a lot of time to execute:

```
clCreateProgramWithSource()
clBuildProgram()
clLinkProgram()
clUnloadPlatformCompiler()
```

- To reduce the execution time during application startup, use `clCreateProgramWithBinary` instead of `clCreateProgramWithSource`. See Section 5.7.3 for more details. Section

NOTE: Do not forget to fall back to building from source when `clCreateProgramWithBinary` fails. This could occur, though rarely, if the OpenCL software has incompatible updates.

- Avoid creating or releasing memory objects between `NDRange` calls. The execution time of `clCreate{Image|Buffer}` is related to the amount of memory requested (if `host_ptr` is used).
- If possible, use Android ION memory allocator. `clCreate{Buffer|Image2D}` can create memory objects with an ION pointer instead of allocating additional memory and copying it. Section 7.4 discusses how to use the ION memory.
- Try to reuse memory and context objects in OpenCL to avoid creating new objects. Overall, the host should be doing a lightweight work during the GPU kernel launch to avoid stalling GPU’s execution.

5.7.2 Use event-driven pipeline

The OpenCL enqueue API functions may accept an event list that specifies all the events that must be complete before the current API function starts to execute. Meanwhile, the enqueue API functions can also emit an event ID to identify themselves. The host simply submits the API functions and kernels to GPU for execution without worrying about their dependency and completeness, if the dependency is correctly specified in the event list parameters. By using this method, the overhead of launching API function calls is reduced significantly, as the software can schedule the functions in its best way and the host does not have to interfere in between the API function calls. Therefore, it is highly desirable to streamline the API functions using event driven pipeline. In addition, developers should note:

- Avoid blocking API calls. A blocking call stalls the CPU to wait for the GPU to finish, then stalls the GPU before the next `clEnqueueNDRangeKernel` call. Blocking API calls is useful mostly for debugging.
- Use callback functions. Starting with OpenCL 1.2, many API functions are enhanced or modified to accept user-defined callback functions to handle events, and this asynchronous call mechanism allows more efficient pipeline execution as the host is now more flexible to handle the events.

5.7.3 Kernel loading and building

Loading and building kernel source at runtime can be expensive. Some applications may prefer to generate source code on-the-fly, as some parameters may not be available upfront. This may be fine if generation and compilation of the source code do not affect GPU execution. But in general, dynamic source code generation is discouraged.

Instead of building the source code on-the-fly, a better way is to build the source code offline and use the binary kernel only. When the application is loaded, the binary kernel code is loaded as well. Doing so would significantly reduce the overhead of loading code from disk.

If the application targets different tiers of Adreno devices, different versions of the binary code are needed. A few things to note regarding compatibility:

- Binary code can be used only for the specific GPU for which it was compiled. If a binary was built on a device that has the Adreno A530 GPU, it cannot be used on a device that has the Adreno A540 GPU.
- Backward compatibility is available across compiler versions. Newer versions of the compiler may support an older binary, provided that the target GPU is the same.

If an incompatible binary kernel is found, use `clCreateProgramWithSource` as a fall back solution.

5.7.4 Use in-order command queues

The Adreno OpenCL platform has support for out-of-order command queues. However, there is a greater overhead due to the dependency management required for implementing out-of-order command queues. The Adreno software pipelines commands sent to an in-order queue. Therefore, it is good practice to use in-order command queues rather than out-of-order command queues.

6 Workgroup size performance optimization

Workgroup size tuning is a simple and effective optimization method for many kernels. This chapter presents the fundamental information on workgroup size, such as how to get workgroup size, why workgroup size is important, some common practices on optimal workgroup selection and tuning are also discussed.

6.1 Obtain the maximum workgroup size

The maximum workgroup size of a kernel on a device can be queried by using the following API function after running `clBuildProgram`:

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

The actual workgroup size used by `clEnqueueNDRangeKernel` cannot exceed `maxWorkGroupSize`. If the workgroup size is not specified by the application, the Adreno OpenCL software may select a maximum working workgroup size.

6.2 Required and preferred workgroup size

A kernel may be written in a way that certain workgroup size is required or preferred. OpenCL provides following methods for requesting specific workgroup size to the compiler:

- Use the `reqd_work_group_size` attribute

  The `reqd_work_group_size(X, Y, Z)` attribute passes in a specific 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)
     {        . . .        }
```

- Use the `work_group_size_hint` attribute

  The OpenCL software attempts to use the specified size, but does not guarantee the actual workgroup 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)
     {         . . .         }
```

In most cases, the compiler cannot guarantee that it generates the optimal machine code if the workgroup size restriction is imposed. Also, the compiler may have to spill registers to the system's random-access memory (RAM), if it cannot meet the required workgroup size using the on-chip registers. Therefore, the use of these two attributes is discouraged, unless the kernel is written in a way that a specified workgroup size is mandatory for the kernel to work.

NOTE: For cross-platform compatibility purposes, it is not a good practice to write kernel that relies on a fixed workgroup size or layout.

6.3 Factors affecting the maximum workgroup size

If no workgroup size attributes are specified, the maximum workgroup size of a kernel depends on many factors:

- Register footprint of the kernel. Generally, the more complex the kernel is, the larger the register footprints, and smaller the maximum workgroup size. Factors that could raise register footprint are as follows:
  - Packing more workload for each work item
  - Control flow
  - High precision math functions (e.g., not using the native math functions or fast math compilation flag)
  - Local memory, if this leads to allocating additional registers to temporarily store source and destination of load/store instructions.
  - Private memory, e.g., an array defined for each work item.
  - Loop unrolling
  - Inline functions
- Size of the general purpose register (GPR) File
  - Adreno low tiers may have smaller register file size.
- Barriers in kernel
  - If a kernel does not use barriers, the maximum workgroup size can be set to DEVICE MAXIUMUM in the Adreno A4x and A5x series, regardless of the register footprint.

6.4 Kernels without barrier

Traditionally, all work items in the workgroup have been required to be resident on the GPU at the same time. For kernels that have heavy register footprint, this can restrict their workgroup size to be well below the device maximum.

Starting from the Adreno A4x series, kernels without barrier can have the maximum workgroup size that Adreno supports, typically 1024, despite their complexity. As there is no synchronization required between waves, for these types of kernels, a new wave can start to execute when an old wave is complete.

In this case, having the maximum workgroup size does not mean that they have good parallelism. A kernel without barriers could be so complex that only a few waves run in parallel inside SP, which would result in poor performance. Developers should continue to optimize and minimize the register footprint, regardless of the maximum workgroup size obtained from the `clGetKernelWorkGroupInfo` function.

6.5 Workgroup size tuning

This section describes general guidelines in selecting the best workgroup size and shape.

6.5.1 Avoid using default workgroup size

If a kernel call does not specify the workgroup size, the OpenCL software will find a working workgroup size using some simple mechanism. Developers should be aware that the default workgroup size is unlikely to be optimal. It is always a good practice to try different workgroup size and shape layout (for 2D/3D) manually and find the optimum one.

6.5.2 Large workgroup size, better performance?

This is true for many kernels, as increasing the workgroup size allows more waves to be resident on the SP, which often translates to better latency hiding and improved SP utilization.

However, for some kernels, performance may deteriorate with increasing workgroup sizes. An example is when larger workgroup size results in increased cache thrashing, due to poor data locality and access patterns. The locality problem is also acute for texture accesses, because the texture cache is typically smaller than the unified L2 cache. Ultimately, it is the nature of data access within the kernel that determines the best workgroup size and shape.

6.5.3 Fixed vs. dynamic workgroup size

For performance portability across devices, avoid making assumptions that one workgroup size fits all, and avoid hard coding workgroup size. A specific workgroup size and layout that works best on one device may be suboptimal on another. Therefore, given a kernel, it is advised to profile different workgroup sizes for all devices that the kernel can execute with, and select the best one for each device at runtime.

6.5.4 One vs. two vs. three-dimensional (1D/2D/3D) workgroup

The dimensionality of a kernel may have performance implication. Dependent on the data access pattern by work items, in some cases a 2D kernel may have better data locality in cache, which leads to better memory access and better performance. While in other cases, a 2D kernel may result in worse cache thrashing than 1D kernel. It would be good to try different dimensions with the kernel for optimal performance.

6.6 Other topics on workgroup size

6.6.1 Global work size and padding

OpenCL 1.x requires that the global worksize of a kernel needs to be a multiple of its workgroup size. If the application specifies a workgroup size that does not meet this condition, the `clEnqueueNDRangeKernel` call will return an error. In such a case, the application can pad the global work size such that it becomes a multiple of the user specified workgroup size.

NOTE: OpenCL 2.0 lifts this restriction, and the global worksize does not have to be multiple of the workgroup size, which is called *non-uniform* workgroups.

Ideally, the workgroup size in its first dimension should be a multiple of the wave size (32 for example) to fully utilize the wave resources. If this is not the case, consider padding the workgroup size to meet this condition, keeping in mind that the global worksize may also need to be padded for OpenCL 1.x.

6.6.2 Brute force search

Due to the complexity involved in workgroup size selection, experimentation is often the best way to find the optimal size and shape.

One option is to use a warm-up kernel that has similar complexity as the actual workload (but perhaps using smaller workload) to dynamically search the optimal workgroup size at the start of the application. The selected workgroup size is then used for the actual kernel. Many commercial benchmarks rely on this method.

6.6.3 Avoid uneven workload across workgroups

Some applications may be written in a way that uneven workload across workgroups could occur. For instance, some region based image processing use cases may have some regions that need a lot more processing than the other regions. This should be avoided as the performance may be unpredictable. In addition, it can complicate the context switch, if any single workgroup takes too long to finish.

A method to overcome this issue is to adopt a two-stage processing strategy. The first stage may collect the interesting points and prepare data for the second stage to process. The workload is more deterministic, which might make it easier to distribute equally across workgroups.

6.6.4 Workgroup synchronization

OpenCL does not guarantee the execution order of workgroups and does not define a mechanism to do workgroup synchronization. It is not recommended to have the application depend on the workgroup orders.

In practice, it is possible to do limited synchronization across workgroups by using atomic functions or other methods. For example, the application may allocate a global memory object that is updated atomically by work items from different workgroups. A workgroup can monitor the memory object that is updated by the other workgroups. By this way, it is possible to achieve limited workgroup synchronization.

7 Memory performance optimization

Memory optimization is the most important and effective OpenCL performance technique. A significant number of applications are memory bound rather than compute bound. Mastering memory optimization is therefore essential for OpenCL optimization. In this chapter, the OpenCL memory model is reviewed, followed by the best practices.

7.1 OpenCL memories in Adreno GPUs

OpenCL defines four types of memory—namely, global, local, constant, and private memory—and understanding the differences between them is essential. Figure 7-1 illustrates a conceptual layout of these four types of memory.

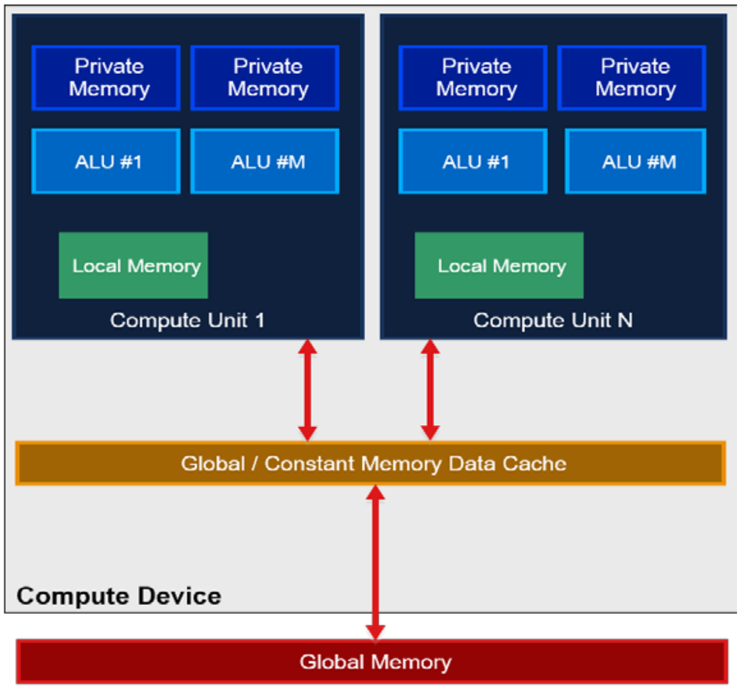

Figure 7-1 OpenCL conceptual memory hierarchy

The OpenCL standard only defines these memory types conceptually, and how they are implemented is vendor-specific. The physical locations may be different from its conceptual location. For example, private memory objects may be allocated in the off-chip system RAM which is far away from GPU.

Table 7-1 lists the definition of the four types of memory, and their latency and physical location in Adreno GPUs. In Adreno GPUs, both local and constant memory are supported by on-chip RAM and have much shorter latency than the off-chip system RAM.

Generally, it is recommended to use local and constant memory for data that need frequent access to take advantage of the short latency property. More details are presented in the following sections.

Table 7-1 OpenCL memory model in Adreno GPUs

Memory	Definition	Relative latency	Location
Local	Shared by all work items in a work group	Medium	On-chip, inside SP
Constant	Constant for all work items in a work group	Low for on-chip allocation, and high otherwise	On-chip if it can fit in. Otherwise in system RAM
Private	Private to a work item	Based on where the memory is allocated by the compiler	In SP as register or local memory or in system RAM (compiler determined)
Global	Accessible by all work items in all work groups	High	System RAM

7.1.1 Local memory

Adreno GPUs support fast on-chip local memory, while the local memory size varies from series/tiers to series/tiers. Prior to using local memory, it is a good practice to query how much local memory is available per workgroup for the device using the following API:

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

Here are the guidelines for using local memory.

- Use local memory to store data that is used repeatedly or to store intermediate results between two stages within a kernel.
    - An ideal scenario is when work items access the same content multiple times, and more than twice
        - For example, consider a window-based motion estimation using object matching for certain video processing. Suppose each work item processes a small region of 8x8 pixels within a search window of 16x16 pixels, leading to a lot of data overlap between neighboring work items. Local memory can~~can~~ be a good fit to store the pixels in this case to reduce the redundant fetch.
- Barriers used for data synchronization across work items may be expensive
    - If there is data exchange between work items, for example, work item A writes data to local memory and work item B reads from it, a barrier operation is required due to the relaxed memory consistency model in OpenCL.
    - Barrier often results in a synchronization latency that stalls ALUs, leading to lower ALU utilization.
    - In some situations, caching data into local memory leads to synchronization latency that washes out the benefits of using local memory. In such a case, using global memory directly—to avoid barrier—may be a better option.
- Use vectorized local memory load/store
    - Using a vectorized load of up to 128 bits (e.g., `vload4_float`) that is 32-bit aligned is recommended.

    - Vectorized data load/store is discussed in more details in Section 7.2.2.
- Allow each work item to participate in local memory data load rather than use one work item to do the entire load
    - Avoid having only one work item to load/store entire local memory for the workgroup.
- Avoid using the function called `async_work_group_copy`. It is often difficult for compiler to generate the optimal code to load local memory and better for developers to write code that manually loads data into local memory.

7.1.2 Constant memory

Adreno GPUs support on-chip constant memory. It has the best latency and superior performance among the four types of memory. Constant memory is usually used in the following cases:

- Scalar and vector variables defined with `constant` qualifier are usually stored in constant RAM.
- An array defined with `constant` qualifier is stored in constant RAM, if it is defined in program scope (e.g., the compiler can determine its size) and there is sufficient space is constant RAM.
- Scalar or vector data types for kernel arguments are stored in constant RAM. For example, `coeffs` in the following example will be stored in constant RAM:

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

- Scalar and vector variables and arrays which are qualifies by `__constant` but do not fit into constant RAM, will be allocated in system RAM.
- To have an array defined in kernel argument load into constant RAM, an attribute called `max_constant_size(N)` must be provided to indicate the size of the constant array, where `N` represents the number of bytes required. In the following example 1024 bytes in the constant RAM is allocated for the variable `foo`:

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

Specifying the `max_constant_size` attribute is important. Without this attribute, the array is stored in the off-chip system RAM, since the compiler does not know the size of the array and cannot promote it to the on-chip RAM.

NOTE: This feature is only supported for 16-bit and 32-bit arrays; 8-bit arrays are not supported. Also, if there is insufficient space in constant memory to allocate the array, it is stored in the off-chip system RAM instead.

NOTE: Constant RAM may not be optimal for an array that is dynamically indexed, and divergently accessed by work items. For example, if one work item fetches index 0 and the next one fetches index 20, constant memory is inefficient. Using an image object is a better choice in this case.

7.1.3 Private memory

In OpenCL, private memory is private to each work item and not accessible by other work items in the workgroup. Physically, private memory could reside in on-chip registers or off-chip system RAM. The exact location depends on several factors and here are a few typical cases:

- Scalar variables are stored in registers, which is the fastest memory.
  - If no enough registers, private variables will be allocated in system RAM.
- Private arrays may be stored in:
  - Local memory, though it is not guaranteed
  - Off-chip system RAM, if they exceed local memory capacity

Storing private memory into off-chip system RAM is highly undesirable as system RAM is slower and private memory access patterns are not cache friendly, especially if the amount of private memory per work item is large. Here are several recommendations:

- Avoid defining any private array in kernels. Try to use vector if possible.
- Replace private array with global or local array and design its layout so that the access of the array elements can be coalesced across multiple work items. The cache performance could be a lot better.
- Use vectorized private memory load/store, i.e., try to load/store up to 128-bit per transaction.

7.1.4 Global memory

OpenCL application can use two types of global memory objects, buffer and image, and both use the off-chip system RAM. As compared to a buffer object which is simple one-dimensional data array stored in system RAM, image object is an opaque memory object in the sense that developers cannot assume the layout or format of how data are stored internally. When an image object is created, the software arranges the data in certain ways for GPU to access more efficiently. The optimal ways to use them are quite different and discussed in the following sections.

7.1.4.1 Buffer

Buffer objects store one-dimensional collection of elements, which can be scalar data types (such as integer, floating point), vector data type, or user-defined structures. A buffer object is created using the following API function:

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

Buffer objects are stored in global memory and accessed via the L2 cache in Adreno GPUs. In this function, the most important parameter is `cl_mem_flags`. OpenCL allows many different flags for this function, and how to select and combine these flags is extremely important for performance. Here are a few points:

- Some flags may incur an extra memory copy. Try to use zero-copy flags that are discussed in Section 7.4.

- Some flags are meant for desktop/discrete GPUs with dedicated GPU memory.
- Use the flags that are most accurate. The general idea is that as the flags are more restrictive, the OpenCL driver can find better configurations for the object and improve its performance. For instance, it can impose a cache flush policy (write-through, write-back, etc.) that best fits the memory object. Section 7.4.2 has more details on the cache policy and its implication for performance. Here are a few examples:
  - If the memory is read-only by host, then use `CL_MEM_HOST_READ_ONLY`.
  - If the memory has no access by host, then use `CL_MEM_HOST_NO_ACCESS`.
  - If the memory is for host to write only, use `CL_MEM_HOST_WRITE_ONLY`.

7.1.4.2 Image

An image object is used to store a one-, two-, or three-dimensional texture, frame buffer, or an image data, and the layout of data inside the image object is opaque. In practice, the content in the object does not have to be associated with an actual image data. Any data can be stored as image object to utilize the hardware texture engine and its L1 cache in Adreno.

An image object is created using the following API:

```
cl_mem clCreateImage(cl_context context,
                     cl_mem_flags flags,
                     const cl_image_format *image_format,
                     const cl_image_desc *image_desc,
                     void *host_ptr,
                     cl_int *errcode_ret)
```

Notice that `cl_mem_flags` for image has the similar rule of thumb to the buffer object as discussed in the previous section.

There are many image formats and data types supported in Adreno GPUs. From the Adreno A3x GPUs to the Adreno A5x GPUs, new pairs of image format and data type have been added. Users may use the function `clGetSupportedImageFormats` to get a full list of the supported image format/data type.

To fully utilize the memory bandwidth, it is recommended to use the pairs whose length is of 128-bit, e.g., `CL_RGBA/CL_FlOAT, CL_RGBA/CL_SIGNED_INT32`, etc.

7.1.4.3 Using image object vs. buffer object

Using image object over buffer object has following advantages:

- Leverage the texture engine hardware.
- Leverage the L1 cache
- Handling of image boundary is built-in.
- Supports numerous image format and data type combinations listed under “Image” in Section 7.1.4, with support for automatic format conversions.

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

For example, assume an image is of type `CLK_NORMALIZED_COORDS_TRUE` and `CL_UNORM_INT16`, i.e., image date is 2-byte unsigned short. Function call `read_imagef` will do the following:

- Reads pixels from image object (which is then cached in L1 cache).
- Interpolates neighboring pixels in hardware.
- Converts and normalizes it to the range of [0, 1]

This is convenient for bilinear or trilinear interpolation operations.

Sometimes a buffer object may be a better choice:

- More flexible data access:
  - Image object only allows access at the pixel size boundary, for example, 128-bit for RGBA and 32-bit/channel image object.
  - For buffer object, Adreno supports byte addressable access. For example, 128-bit data could be loaded from any byte address in a buffer object, if it is not out of the buffer boundary.
- If L1 becomes the bottleneck.
  - For example, there is serious L1 cache thrashing, which makes L1 cache access inefficient.
- A buffer object can be read and write inside kernels. Although image object can also be read and write from OpenCL 2.0, its performance is generally worse due to synchronization requirement.

Table 7-2 Buffer vs. image in Adreno GPUs

Features	Buffer	Image
L2 cache	Yes	Yes
L1 cache	No	Yes
Support read and write object	Yes	No in OpenCL 1.x Yes, in OpenCL 2.x (synchronization required)
Byte addressable	Yes	No
Built-in hardware interpolation	No	Yes
Built-in boundary handling	No	Yes
Support image format/sampler	No	Yes

7.1.4.4 Use of both Image and buffer objects

Instead of using texture object only or buffer object only, a better way is to have both UCHE⇔SP and UCHE⇔TPL1⇔SP paths fully utilized. As TPL1 has L1 cache, storing L1 the most frequently used but relatively small amount of data is a good practice.

7.1.4.5 Global memory vs. local memory

One use case of local memory is to load data into local memory first, synchronize to make sure the data are ready, and then the work items in the workgroup can use it for processing. However, using global memory may be better than LM due to the following reasons:

- May have better L2 cache hit ratio and better performance
- Code is simpler than local memory and have a larger work group size

7.2 Optimal memory load/store

In previous sections we discussed the general guidance on how to use different type of memory. In this section, we will go over a few key and general points that are critical for performance regarding memory load/store.

7.2.1 Coalesced memory load/store

Coalesced load/store refers to the capability of combining load/store requests from multiple neighboring work items, as mentioned in Section 3.3.1 for local memory access. Coalesced access is also important for global memory load/store.

Coalesced store works in a similar manner to read, except that load is a 2-way process (request and respond), while store is a 1-way process. Therefore, coalesced load is generally more critical than store.

In Adreno GPUs, hardware coalesced access has been enabled gradually since the Adreno A4x GPUs, as shown in Table 7-3. Private memory does not support coalesced access.

Table 7-3 Coalesced access support in Adreno GPUs

Load/store	Adreno A3x	Adreno A4x	Adreno A5x
Global memory coalesced load	No	No	Yes
Global memory coalesced store	No	Yes	No
Local memory coalesced load/store	No	No	Yes

7.2.2 Vectorized load/store

Vectorized load/store refers to multiple data load/store in a vectorized way for single work items. This is different from coalesced access, which is for multiple work items. Here are a few key points to use vectorized load/store:

- For each work item, it is recommended to load data in chunk of multiple bytes, e.g., 64-bit/128-bit, the bandwidth can be better utilized.
  - For example, multiple 8-bit data can be manually packed into one element (e.g. 64-bit/128-bit), which is loaded using `vloadn`, and then unpacked using `as_typeN` function (e.g., `as_char16`).
  - See the vectorized operation example in Section 9.2.3.
- For optimal SP to L2 bandwidth performance, load/store memory address should be 32-bit aligned.
- There are two methods to do vectorized load/store:

- Use built-in function (`vload/vstoren`), which is well defined in OpenCL.
- Alternatively, pointer cast can be used to do vectorized load/store as follows:
  ```
  char *p1; char4 vec;
  vec = *(char4 *)(p1 + offset);
  ```

- It is recommended to use vectorized load/store instructions that take up to 4 components. Vectorized load of data type with more than 4 components would be divided into multiple load/store instructions with each of them taking no more than 4 components.
- Avoid loading too many data in one work item.
  - Loading too many data may result in higher register footprints, which could lead to a smaller work group size and hurt performance. In the worst cases, it may cause register spilling, i.e., compiler must use system RAM to store variables.

NOTE: Vectorized ALU calculations can also improve performance, though generally not as much as the one from vectorized memory load/store.

7.2.3 Optimal data type

Data type is important as it affects not just memory traffic but also the ALU operations. Here are a few rules for data type:

- Check data types in each stage of the application pipeline and make sure that the data type used is consistent across the entire pipeline.
- Use shorter data types if possible to reduce memory fetch/bandwidth, and increase the number of ALU available for execution.

7.2.4 16-bit floating (half) vs. 32-bit floating

Using of half data type instead of floating data type is highly recommended as Adreno GPUs have dedicated hardware to accelerate half data type calculation. The `gflops` of half ALUs is almost twice of full ALUs. Here are some rules:

- 16-bit half has limited precision support. It can only precisely represent data within much narrowed range.
  - For example, it can only accurately represent [0, 2048] on integer values.
- Convert half to float for calculation if half data calculation causes unacceptable precision loss. But store the results as half data type.

7.3 Atomic functions

A set of local and global atomic functions are defined in OpenCL, and Adreno GPUs natively support all of them in hardware. Here some rules when using atomic functions:

- Avoid frequently updating a single global atomic memory address by work items from single or multiple workgroups, as atomic operations are serialized operations and their performance is usually not good as parallel operations.
- Try to use local atomic first and have a single update to global memory atomically.

7.4 Zero copy

Adreno OpenCL provides a few mechanisms to avoid costly memory copy that could occur at host side. Dependent on how the memory object is created, there are a few different options to avoid excessive copy.

7.4.1 Use map over copy

Assume that the OpenCL application has full control over the data flow, i.e., the target and source memory object creation are all managed by the OpenCL application. This is the simplest case and memory copy can be avoided by using the steps as follows:

- When creating a buffer/image object, use the flag `CL_MEM_ALLOC_HOST_PTR`, and follow the steps as follows:
  - First set `cl_mem_flags` input in `clCreateBuffer`:

    ```
    cl_mem Buffer = clCreateBuffer(context,
                                   CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR,
                                   sizeof(cl_ushort) * size,
                                   NULL,
                                   &status);
    ```
  - Then use the map function to return a pointer to the host:

    ```
    cl_uchar *hostPtr = (cl_uchar *)clEnqueueMapBuffer(
                                   commandQueue,
                                   Buffer,
                                   CL_TRUE,
                                   CL_MAP_WRITE,
                                   0,
                                   sizeof(cl_uchar) * size,
                                   0, NULL, NULL, &status);
    ```
  - Host updates the buffer using the pointer `hostPtr`.
    - For example, host can fill camera data or read data from disk into the buffer
  - Unmapped the object:

    ```
      status = clEnqueueUnmapMemObject(
                 commandQueue,
                 Buffer,
                 (void *) hostPtr,
                 0, NULL, NULL);
    ```
  - The object can be used by OpenCL kernels.

`CL_MEM_ALLOC_HOST_PTR` is the only method to avoid copying of data in this scenario. With the other flags, such as `CL_MEM_USE_HOST_PTR` or `CL_MEM_COPY_HOST_PTR`, driver will have to do additional memory copy for GPU to access.

7.4.2 Avoid memory copy for objects allocated not by OpenCL

7.4.2.1 ION memory extensions

If a memory object is initially created outside the scope of the OpenCL API and is allocated using ION/Gralloc, the `cl_qcom_ion_host_ptr` extension can be used to create a buffer/image object, which maps its ION memory to GPU accessible memory without incurring extra copy.

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

7.4.2.2 QTI Android native buffer (ANB) extension

In many camera and video processing use cases, ANB (allocated by `gralloc`) must be shared. Sharing is possible because the buffers are based on ION. However, to use the ION path, the developer needs to extract internal handles from these buffers, which requires access to QTI's internal headers. The `cl_qcom_android_native_buffer_host_ptr` extension offers a more straightforward way to share ANBs with OpenCL without needing access to QTI headers. This enables ISVs and other third-party developers to implement zero copy technique for ANBs.

NOTE: A sample that illustrates use of the `cl_qcom_android_native_buffer_host_ptr` extension can be provided on request.

7.4.2.3 Using standard EGL extensions

The `cl_khr_egl_image` extension creates an OpenCL image from an EGL image. 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.

7.5 Improve cache usage

To have a good cache usage, the following rules should be followed:

- Check cache thrashing and cache usage efficiency. Snapdragon Profiler can provide cache access information, such as the number of bytes for load/store and cache hit/miss ratio.
  - If the number of bytes to load into UCHE is a lot higher than what is expected by the kernel, it is possible that there is cache thrashing.
  - Metrics such as the L1/L2 hit/miss ratios can tell how well the cache is used.
- Avoid thrashing by doing the following:
  - Adjust the workgroup size, such as reduce workgroup size
  - Change access pattern, e.g., change the dimensionality of kernel.

- If there is cache thrashing when using loops, adding atomics or barrier in the loop may reduce the chances of thrashing.

7.6 CPU cache operations

For memory objects that are cacheable, the OpenCL driver must flush and/or invalidate the CPU cache at appropriate times. This ensures that both the CPU and the GPU see the most current copy of the data when they attempt to access it. For example, the CPU cache must be invalidated when mapping the output buffer of a kernel for reading by the host CPU. The OpenCL driver has a sophisticated CPU cache management policy which attempts to minimize the number of cache operations by tracking data visibility on a per memory object basis and by deferring operations to the extent possible. For example, there could be a CPU cache flush on an input buffer right before a kernel is launched.

CPU cache operations have a very measurable cost, which can be observed as a delta between `CL_PROFILING_COMMAND_QUEUED` and `CL_PROFILING_COMMAND_SUBMIT` for `clEnqueueNDRangeKernel`, as shown in Figure 4-1. In some cases, the execution time of `clEnqueueMapBuffer/Image and clEnqueueUnmapBuffer/Image` can increase. The cost of a CPU cache operation generally increases linearly along with memory object size.

To minimize the cost of CPU cache operations, the application pipeline should be structured so that processing is not moved back and forth between CPU and GPU. In addition, the application should allocate memory objects in such a way that the data which is subject to back and forth CPU and GPU access is in a different memory object from the data which has only one access transition.

The memory objects should be created using the CPU cache policy that is appropriate for their intended usage. When allocating memory for buffer objects or image objects, the driver will select the CPU cache policy. The default CPU cache policy is write-back. However, if either `CL_MEM_HOST_WRITE_ONLY` or `CL_MEM_READ_ONLY` is specified in flags, the driver will assume that the application does not intend to read the data using the host CPU. In that case, the CPU cache policy is set to write-combine.

For externally allocated memory objects such as with ION and ANB mechanisms, the application has more direct control over CPU cache policy. When importing these objects into OpenCL the application must set the CPU cache policy flag correctly.

7.7 Use of SVM

The Adreno A5x GPUs support coarse-grain SVM, a key feature in the OpenCL 2.0 full profile. With SVM, the host and device memory addresses are identical. SVM feature in OpenCL 2.0 allows the memory to be shared across host and device easily, and accessing the host pointer on OpenCL device is now possible.

For coarse-grain SVM, accessing the memory by host or devices is limited at the synchronization points (map/unmap). This can greatly facilitate applications that need to work with pointer based data structure across host and device.

7.8 Best practices to reduce power/energy consumption

Power and energy is a major factor for mobile applications. Applications with optimal performance may not have the best power/energy performance and vice versa. Therefore, it is important to understand power/energy and performance requirement. There are several tips on reducing power and energy consumption for OpenCL:

- Try all means to avoid memory copy, for example, using ION memory to achieve zero-copy, and using `CL_MEM_ALLOC_HOST_PTR` when creating buffers with `clCreateBuffer`. Also avoid using the OpenCL APIs that do data copy.
- Minimize the memory transaction between host and device. This can be achieved by storing memory in constant memory or local memory, using shorter data type, reducing data precision, and eliminating private memory usage, etc.
- Optimize kernels and improve their performance. The faster a kernel can run, generally the less energy or power it would consume.
- Minimize software overhead. For example, use event-driven pipeline to reduce the host and device communication overhead. Avoid creating too many objects, and avoid creating or releasing objects in between kernel execution.

8 Kernel performance optimization

This section presents tips on kernel optimization.

8.1 Kernel fusion or splitting

A complex application may contain a lot of stages. For OpenCL porting and optimization, one may ask how many kernels should be developed. It is hard to answer as there are many factors involved. Here are a few high-level criteria:

- Good balance between memory and compute
- Enough waves to hide latency
- No register spilling

These could be achieved by doing the followings:

- Split a big kernel into multiple small kernels, if doing this yields better data parallelization.
- Fuse multiple kernels into one kernel (kernel fusion), if memory traffic can be reduced and parallelization can be maintained, e.g., workgroup size is reasonably large.

8.2 Compiler options

OpenCL supports some compiler options that are defined in Section 5.6.4 of *The OpenCL Specification* found in References. Compiler options are passed in through the APIs `clCompileProgram` and `clBuildProgram`. Multiple options can be combined:

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

With these options, developers can enable some functionality for its own purpose. For example, using `-cl-fast-relaxed-math` allows the kernel to be built using fast math rather than the OpenCL conformant math that has much higher precision requirement per OpenCL specification.

8.3 Conformant vs. fast vs. vs. native math functions

The OpenCL standard defines many math functions in the OpenCL C language and by default all the math functions must meet the IEEE 754 single precision floating-point math requirements, as required by the OpenCL specification. Adreno GPUs have a built-in hardware module, the elementary function unit (EFU), to accelerate some primitive math functions. Many math functions that are not directly supported by EFU have been either optimized by combining EFU and ALU operations, or emulated using complex algorithms by the compiler. Table 8-1 shows a list of OpenCL-GPU math functions categorized based on their relative performance. It is a good practice to use the high-performance ones, for example, functions in category A.

Table 8-1 Performance of OpenCL math functions (IEEE 754 conformant)

Category	Implementation	Functions (refer to the OpenCL standard for more details)
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, 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, tanpi, tgamma`

Alternatively, developers may choose to use native or fast math instead of conformant math functions, if the application is not precision sensitive. Table 8-2 summarizes the three options for using math functions.

- For fast math, enable `-cl-fast-relaxed-math` in the `clBuildProgram` call.
- Use native math 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;`
  - Here is an example to use native math:
    - Original: `int c = a / b;` // `Both a and b are integers`
    - Use native instruction: `int c = (int)native_divide((float)(a), (float)(b));`

Table 8-2 Math function options based on precision/performance

Math functions	Definition	How to use	Precision requirement	Performance	Typical use cases
Conformant	Follow IEEE 754 single precision floating-point math requirement	Default	Strict	Low	Scientific computing, precision sensitive use cases
Fast	Fast math with lower precision	Kernel build option: `-cl-fast-relaxed-math`	Medium	Medium	Many image, video, and vision use cases
Native	Directly calculated by hardware	Use `native_function` instead of function in kernel	Low, vendor dependent	High	Image, video and vision use cases if not sensitive to precision loss

8.4 Loop unrolling

Loop unrolling is generally a good practice as it could reduce instruction execution cost and improve performance. The Adreno compiler can typically unroll loops automatically based on some heuristics. However, it is also possible that the compiler may choose not to fully unroll loops, based on factors such as register allocation budget, or the compiler cannot unroll it due to lack of some knowledge. In these cases, developer may give the compiler a hint, or manually force it to unroll the loops as follows:

- A kernel may give a hint by using `__attribute__((opencl_unroll_hint))` or `__attribute__((opencl_unroll_hint(n)))`.
- Alternatively, a kernel can use directive `#pragma unroll` to unroll loops.
- The last option is to manually unroll loops

8.5 Avoid branch divergence

Generally, GPUs are not efficient when work items in the same wave follow different execution paths. For divergent branches, some work items may have to be masked, resulting in lower GPU occupancy, as shown in Figure 8-1. Also, the conditional check code like `if-else` usually invokes the control flow hardware logic which is expensive.

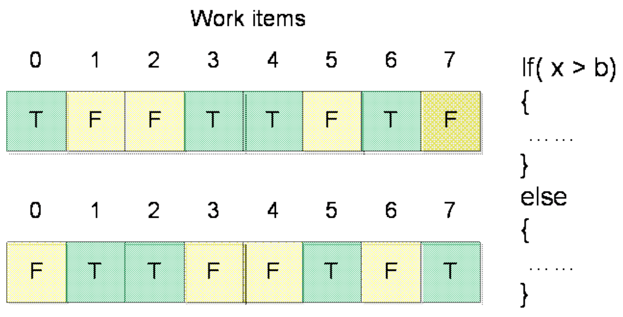

Figure 8-1 Pictorial representation of divergence across two waves

There are some methods to avoid or reduce divergence and conditional checks. At the algorithm level, one may group the work items that fall into one branch into one nondivergent wave. At the kernel level, some simple divergent/conditional check operations can be converted to fast ALU operations. One example is shown in section 9.2.6, where a ternary operation handled by the expensive control flow logic is converted to an ALU operation. Another method is to use functions like `select,` which may use fast ALU operations instead of the control flow logic.

8.6 Handle image boundaries

Many operations may access pixels out of image's boundaries, e.g., filtering, transform, etc. To better handle boundaries, the following options should be considered:

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

8.7 32-bit vs. 64-bit GPU memory access

From the Adreno A5x GPUs, the 64-bit OS is becoming dominant, and many Adreno GPUs are enabled to support the 64-bit OS. The most important change in the 64-bit OS is that the memory space can be well over 4GB and the CPU supports 64-bit instruction sets.

While the GPU can access 64-bit memory space, its use incurs extra complexity and there may be performance penalty.

8.8 Avoid use of size_t

A 64-bit memory address poses complication for OpenCL kernel compilation in many cases and developers need to be careful. It is highly recommended to avoid defining variable as type of `size_t` inside kernels. For the 64-bit OS, a variable defined as `size_t` inside kernels may have to be treated as 64-bit long. Adreno GPUs must use two 32-bit registers to emulate 64-bit. Therefore, having `size_t` type of variables requires more register resources, which often translates to performance degradation due to less active waves and smaller workgroup sizes. So, developers should use 32-bit or shorter data types instead of `size_t`.

For the built-in functions in OpenCL that return `size_t`, the compiler may try to derive and limit the scope based on its knowledge. For example, `get_local_id` returns the result as `size_t`, though `local_id` is never more than 32-bit. In this case, the compiler uses a short data type instead. However, it is generally a good practice to provide the compiler the best knowledge on data type, so that it can generate the optimal code.

8.9 Generic memory address space

OpenCL 2.0 introduces a new feature called *generic memory address space*, in which a pointer does not have to specify its address space. Prior to OpenCL 2.0, a pointer must specify its memory address space, which can be local, private, or global. With generic memory address space, a pointer could be dynamically assigned to different memory address spaces.

While this feature allows developers to reduce their code base and reuse existing code, using of generic memory address space may have slight performance penalty because the GPU SP hardware needs to figure out the real memory space dynamically. It is recommended to precisely define the memory address space if developers clearly know its memory space. This would reduce ambiguity for compiler, and result in better machine code and improved performance.

8.10 Miscellaneous

Many other optimization tips, which look minor but could improve kernel performance, are as follows:

- Precalculate values that do not change within the kernels.
  - It is wasteful to calculate a value that can be precalculated outside of the kernels.
  - Precalculated values can be passed to kernel through kernel arguments or with `#define`.
- Use the fast integer built-in functions. Use `mul24` for 24-bit integer multiplication, and `mad24` for 24-bit integer multiplication and accumulation.
  - Adreno GPUs have native hardware support for `mul24`, while the 32-bit integer multiplication is emulated using more instructions.
  - If there are integer numbers within 24-bit range, using `mul24` is faster than direct 32-bit multiplication.
- Reduce EFU functions.
  - For example, the code like `r=a/select(c,d,b<T)`, where `a`, `b`, and `T` are float variables, `c` and `d` are constant, can be rewritten as `r=a*select(1/c,1/d, b<T)`, which avoids the reciprocal EFU function as `1/c` and `1/d` can be derived at compilation time by the compiler.

- Avoid divide operations, especially integer divide.
  - Integer divide is extremely expensive on Adreno GPUs.
  - Instead of using divide, do a reciprocal operation using `native_recip`, as described in Section 8.3.
- Avoid integer module operation, which is expensive.
- For constant arrays, such as lookup tables, filter taps, etc., declare them outside the kernel scope.
- Use `mem_fence` functions to split/group code sections.
  - The compiler has complex algorithms to generate the optimal code from global optimization perspective.
  - `mem_fence` can be used to prevent the compiler from shuffling/mixing the code before and after it.
  - `mem_fence` allows developers to manipulate some portion of the code for profiling and debugging.
- Use bit shift operations instead of multiplication.

9 OpenCL optimization case studies

In this chapter, a few examples are presented to demonstrate how to optimize using the techniques discussed in early chapters. In addition to a few simple code snippet demonstrations, two well-known image processing filters, Epsilon filter and Sobel filters, are step-by-step optimized by using many of the practices discussed in previous chapters.

9.1 Application sample code

9.1.1 Improve algorithm

This example demonstrates how to simplify an algorithm to optimize its performance. 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.

Modified kernel:

```
// 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` accesses.

9.1.2 Vectorized load/store

This example demonstrates how to do vectorized load/store in Adreno GPUs to better utilize the bandwidth.

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:

```
__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])); // Alternatively
vload and vstore can be used in here
   float4 tmpB = (*((__global float4*)&matrixB[index]));
   (*((__global float4*)&MatrixSum[index])) = (tmpA+tmpB);
   // Since ALU is scalar based, no impact on ALU operation.
}
```

The new kernel is doing vectorized load/store using `float4`. The global work size in the newer kernel should be ¼ of the original kernel because of this vectorization.

9.1.3 Use image instead of buffer

This example calculates a dot product for each pair, given 5 million pairs of vectors. The original code uses a buffer object, which is modified to use a texture object instead (`read_imagef`) to improve frequent data access. It is a simple example, but the technique can be applied to many cases in which buffer object access is not as efficient as texture object access.

Original kernel before optimization	Modified kernel for 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]);`
`}`
	`__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));`
	`}`

9.2 Epsilon filter

Epsilon filter is used widely in image processing for the reduction of mosquito noise, which is a type of impairment occurring at high frequency region such as edges in images. The filter is essentially a nonlinear and point-wise low pass filter with space-varying support, and only the pixels with certain threshold are filtered.

In this implementation, the Epsilon filter is applied only on the intensity (Y) component of YUV images, as the noise is mainly visible in there. Also, it is assumed that the Y component is consecutively stored (NV12 format), which is separated from the UV component. There are two basic steps as illustrated by Figure 9-1.

- Given a pixel to be filtered, calculate the absolute difference value between the central pixel and each pixel in its neighboring 9x9 region.
- If the absolute difference is below a threshold, the neighboring pixel value is used for averaging. Note the threshold usually is a constant predefined in the application.

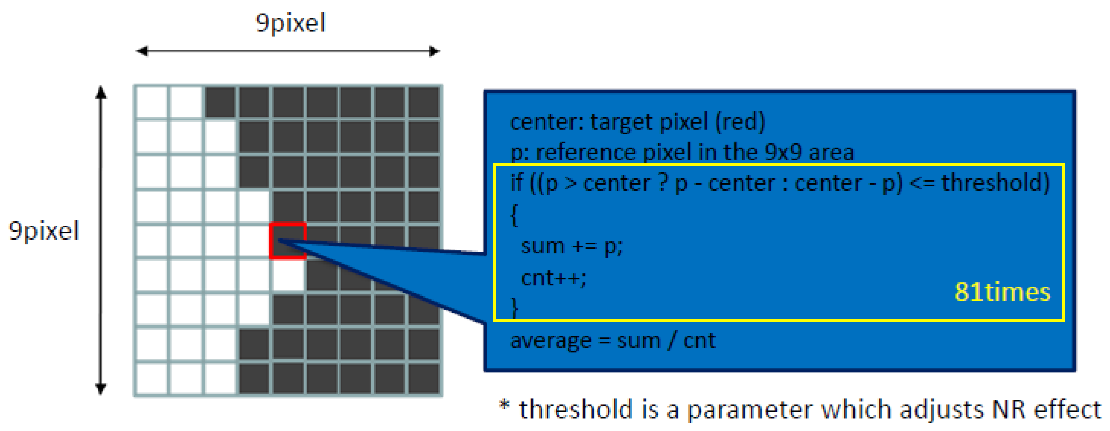
9pixel
9pixel
center: target pixel (red)
p: reference pixel in the 9x9 area
if ((p > center ? p - center : center - p) <= threshold)
{
sum += p;
cnt++;
}
81times
average = sum / cnt
* threshold is a parameter which adjusts NR effect

Figure 9-1 Epsilon filter algorithm

9.2.1 Initial implementation

This application targets YUV images with resolution of 3264x2448 (width = 3264, height =2448) with 8-bit per pixel. The performance numbers reported here are from Snapdragon 810 (MSM8994, Adreno 430) at performance mode.

Here are the initial implementation parameters and strategy:

- Use OpenCL image object instead of buffer
  - Using image over buffer can avoid some boundary check and leverage the L1 cache in Adreno GPUs
- Use `CL_R|CL_UNORM_INT8` image format/data type.
  - Single channel as this is for Y component only, and pixels read into SP is normalized into [0, 1] by the built-in texture pipe in Adreno GPUs.
- Each work item produces one output pixel
- Using 2D kernel and the global work size is set to [3264, 2448]

In the implementation, each work item needs to access 81 floating point pixel. The performance on the Adreno A430 GPU is used as the baseline for further optimization.

9.2.2 Data pack optimization

By comparing the amount of computation and data load, it is obvious that this is a memory bound use case. Thus, the main optimization should be how to improve the data load efficiency.

The first thing to note is that, using 32-bit floating (fp32) to represent pixel values is a waste of memory. For many image processing algorithms 8-bit or 16-bit data type could be sufficient. Since Adreno GPUs have native hardware support for 16-bit float data type, i.e., half or fp16, the following optimization options can be applied:

- Use 16-bit half data type instead of 32-bit float
  - Each work item now accesses 81 half data
- Use `CL_RGBA|CL_UNORM_INT8` image format/data type.
  - Using `CL_RGBA` to load 4 channels to better utilize the TP bandwidth.

- Replace `read_imagef` with `read_imageh.` TP converts the data into 16-bit half automatically.

- Each work item
  - Reads three `half4` vectors per row
  - Output one processed pixel
  - Number of memory access per output pixel: 3x9=27 (`half4`)
- Performance improvement: 1.4x

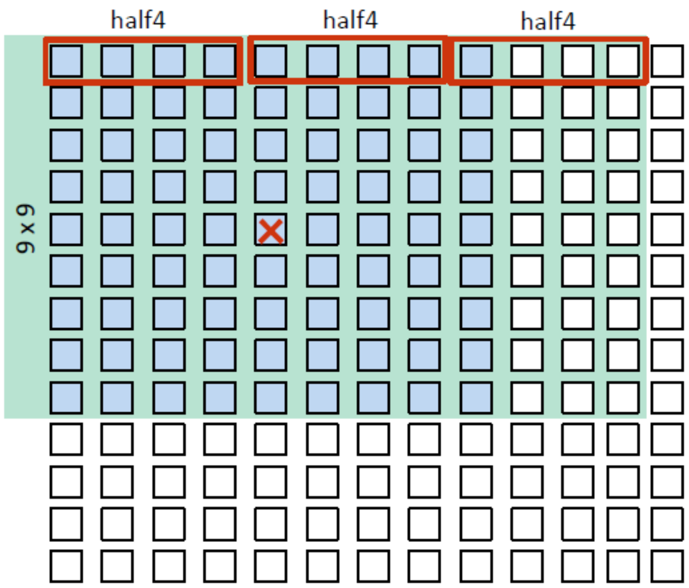

Figure 9-2 Data pack using 16-bit half (fp16) data type

9.2.3 Vectorized load/store optimization

In the previous step, only one pixel is output with so many neighboring pixels loaded. With a few extra pixels loaded, more pixels can be filtered as follows:

- Each work item
- Reads three `half4` vectors per row
  - Output four pixels
  - Number of memory access per output pixel: 3x9/4 = 6.75 (half4)
- Global work size: (width/4) x height.
- Loop unrolling for each row
- Inside each row, sliding window method is used.

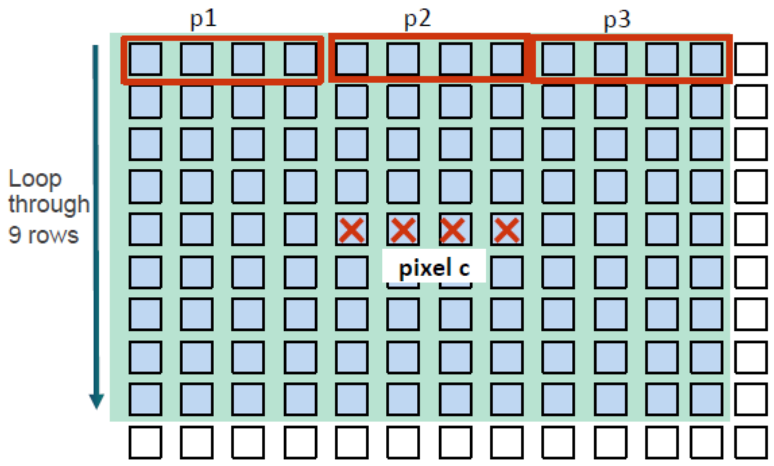

Figure 9-3 Filtering more pixels per work item

Figure 9-3 illustrates the basic diagram of how multiple pixels are processed with extra pixels loaded. Here are the steps:

```
Read center pixel c;
For row = 1 to 9, do:
read data p1;
Perform 1 computation with pixel c;
read data p2;
Perform 4 computations with pixel c;
read data p3;
Perform 4 computations with pixel c;
end for
write results back to pixel c.
```

After this step, the performance is improved by 3.4x from the baseline.

9.2.4 Further increase work load per work item

One may expect more performance boost by increasing the workload per work item. Here are the options:

- Read one more `half4` vector and increase the output pixel number to 8

- Global work size: width/8 x height
- Each work item
  - Reads four `half4` vectors per row
  - Outputs eight pixels
  - Number of memory access per output pixel: 4x9/8 = 4.5 (`half4`)

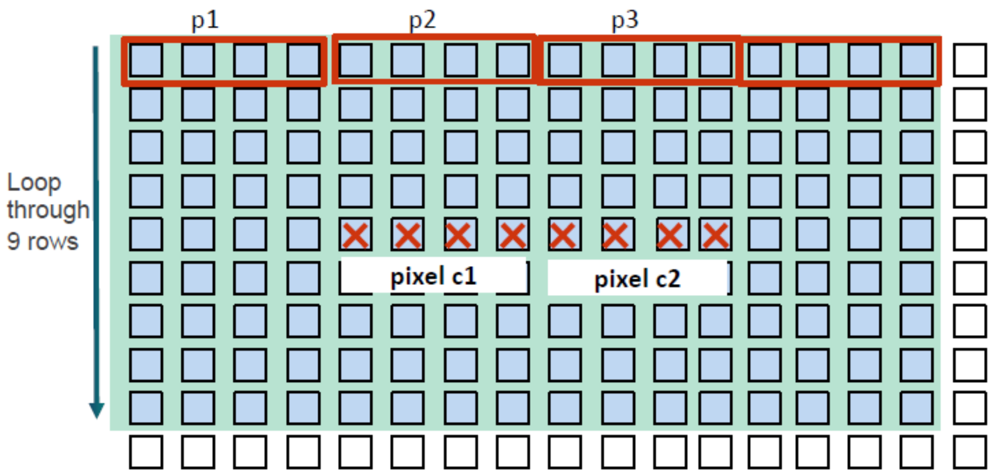

Figure 9-4 Process 8 pixels per work item

These changes lead to slight performance improvement by 0.1x. Here are the reasons why it does not work well:

- There is no much changes on cache hit ratio, which is already very good in previous step.
- More registers are used and less waves, which hurts the parallelism and latency hiding.

For experimental purpose, one may load even more pixels as follows:

- Read even more `half4` vectors and increase the number of output pixels to 16.
- Global work size: width/16 x height

Figure 9-5 shows that each work item does the following:

- Reads 6 `half4` vectors per row
- Outputs 16 pixels
- Number of memory access per output pixel: 6x9/16 = 3.375 (`half4`).

After these changes, the performance deteriorates from 3.4x to 0.5x of the baseline! Loading more pixels into one kernel causes register spilling, which seriously hurt the performance.

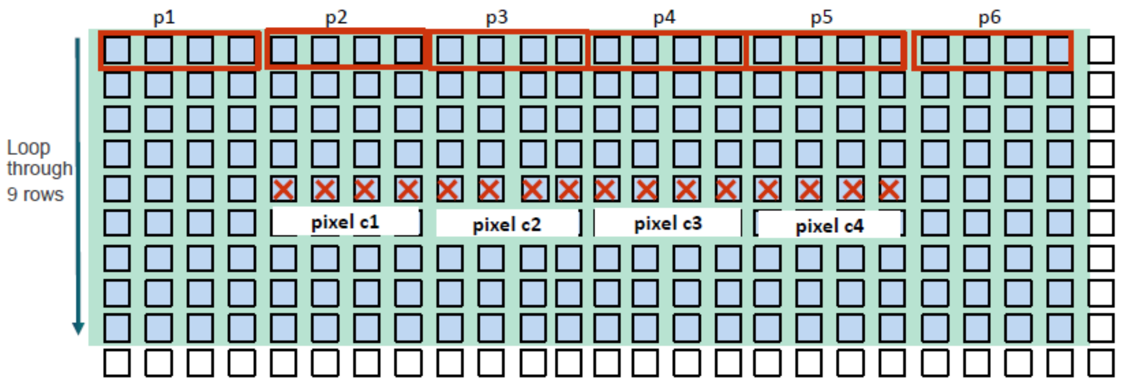

Figure 9-5 Process 16 pixels per work item

9.2.5 Use local memory optimization

Local memory has a much shorter latency than global memory as it is on-chip memory. One choice is to load the pixels into local memory, and avoid repeatedly loading from global memory. In addition to the center pixel to be processed, the surrounding pixels are also required for the 9x9 filtering and thus loaded into local memory, which is shown in Figure 9-6.

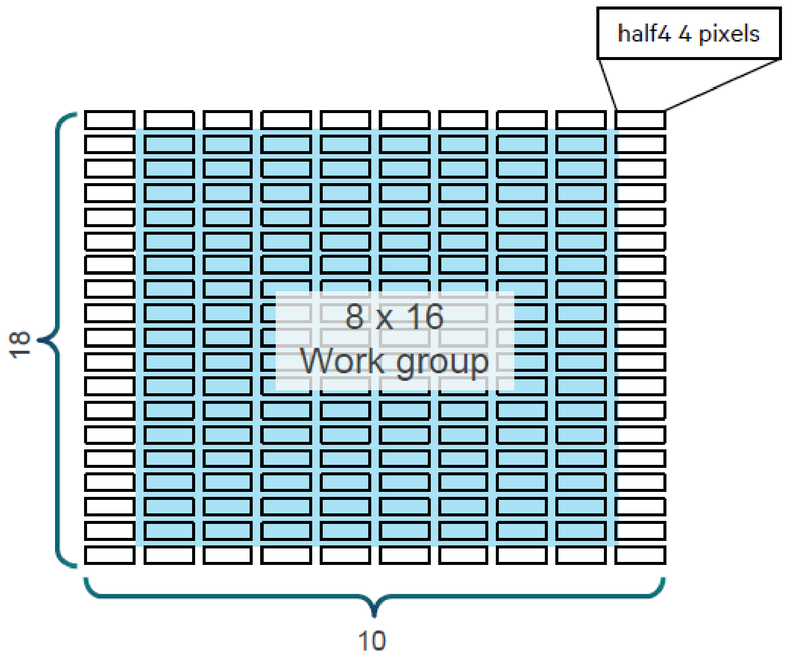

Figure 9-6 Using local memory for Epsilon filtering

Table 9-1 lists the setup of two cases and their performance. The overall performance is considerably better than the original one. However, they do not beat the best number from Section 9.3.4.

Table 9-1 Performance from using local memory

	Case 1	Case 2
Workgroup	8x16	8x24
Local memory size (byte)	10x18x8 =1440	10x26x8 = 2080
Performance	2.4x	2.8x

As discussed in Section 7.1.1, use of local memory often requires barrier synchronization inside workgroups and it does not necessarily yield better performance than global memory. Instead, it may perform worse if there is too much overhead. In this case, global memory could be a better choice if it has high cache hit ratio.

9.2.6 Branch operations optimization

The Epsilon filter needs to do comparison between pixels as follows:

```
Cond = fabs(c -p) <= (half4)(T);
sum += cond ? p : consth0;
cnt += cond ? consth1 : consth0;
```

The ternary operator `?:` incurs some divergence in hardware as not all fibers in a wave go to the same execution branch. The branching operation can be replaced by ALU operations as follows:

```
Cond = convert_half4(-(fabs(c -p) <= (half4)(T)));
sum += cond * p;
cnt += cond;
```

This optimization is applied on top of the one described in Section 9.2.2 and the performance is improved to 5.4x from 3.4x of the baseline!

The key difference is that the new code is executed in the highly parallelized ALU and all fibers in the wave essentially execute the same piece of code, though the variable `Cond` may have different values, while the old one is using some costly hardware logic to handle the divergence.

9.2.7 Summary

The optimization steps and their performance numbers are summarized in Table 9-2. Initially, the algorithm is memory bounded. By doing data packing and vectorized load, it becomes more ALU bounded. In summary, the key optimization for this use case is to load data in an optimal way. A lot of memory bound use cases could be accelerated by using similar techniques.

Table 9-2 Summary of optimizations and performance

Step	Optimizations	Image format	Data type In kernel	Vector processing	Speedup
1	Initial GPU implementation	CL_R | CL_UNORM_INT8	float	1-pixel/work item	
2	Use half type in kernel	CL_R | CL_UNORM_INT8	half		1.0 X
3	Data packing	CL_RGBA | CL_UNORM_INT8			1.4 X
4	Vectorized processing Loop unrolling			4-pixel/work item (half4 output)	3.4 X
5	More pixels per work item			8-pixel/work item	3.5 X
6	More pixels per work item			16-pixel/work item	0.5 X
4-1	Use LM (workgroup size: 8x16)				2.4 X
4-1-1	Use LM, increase workgroup size, workgroup size: 8x24				2.8 X

Step	Optimizations	Image format	Data type In kernel	Vector processing	Speedup
4-1-2	Remove branching operations Use LM, workgroup size: 8x24			4-pixel/work item	2.9 X
4-2	Remove branching operations				5.4 X

The OpenCL performance of Epsilon filter with three different resolutions are shown in Table 9-3. It shows that the performance gaining factor increases for larger images. For an image of 3264x2448, 5.4x performance boost is observed, as comparing to 4.3x for an image of 512x512 using the initial OpenCL code. This is understandable, as there is fixed cost associated with kernel execution regardless of workload, and its weight in the overall performance becomes lower as the workload is larger.

Table 9-3 Performance profiled for images with different resolutions

Image resolution		512x512	1920x1080	3246x2448
Number of pixels		0.26MP	2MP	8MP
Device (A430)	GPU initial results	1x	1x	1x
	GPU optimized	4.3x	5.2x	5.4x

9.3 Sobel filter

Sobel filter, also called Sobel operator, is used in many image processing and computer vision algorithms for edge detection. It uses two 3x3 kernels to combine with the original image to approximate the derivative. There are two kernels: one kernel is for horizontal direction and the other is for vertical direction, as shown in Figure 9-7.

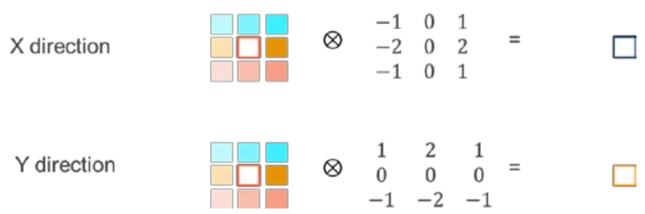

Figure 9-7 Two directional operations in Sobel filter

9.3.1 Algorithm optimization

The Sobel filter is a separable filter which can be decomposed as follows:

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & +1 \end{bmatrix}$$

Figure 9-8 Sobel filter separability

Compared with a nonseparable 2D filter, a 2D separable filter can lower the complexity from `O(n²)` to `O(n)`. It is highly desirable to use separable filters instead of nonseparable due to 2D's high complexity and computational cost.

9.3.2 Data pack optimization

Although computation is reduced considerably for the separable filter, the number of pixel required for filtering each point is the same, i.e., 8 neighboring pixels plus the center pixel for this 3x3 kernel. It is easily seen that this is a memory bound problem. So how to effectively load the pixels into GPU is the key for performance. Three options are as illustrated in the following figures:

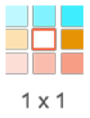

Figure 9-9 Process one pixel per work item: load 3x3 pixels per kernel

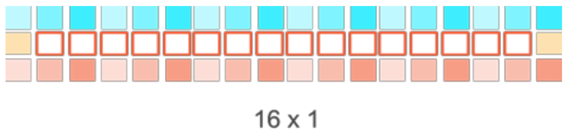

Figure 9-10 Process 16x1 pixels: load 18x3 pixels

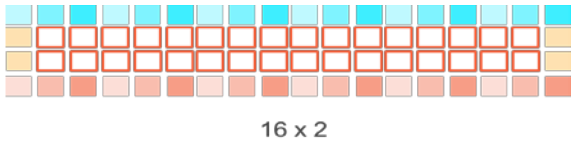

Figure 9-11 Process 16x2 pixels, load 18x4 pixels

The following table summarizes the total number of bytes and average bytes required in each case. For the 1st case in Figure 9-9, each work item only processes Sobel filtering on one pixel. As the number of pixels per work item increases, the amount of data to be loaded is reduced for cases shown in Figure 9-10 and Figure 9-11. This often leads to reduction of data traffic from global memory to GPU and results in better performance.

Table 9-4 Amount of data load/store for the three cases

	One pixel/work item	16x1 pixels/work item	16x2 pixels/work item
Total input bytes	9	54	72
Average input bytes	9	3.375	2.25
Average store bytes	2	2	2

9.3.3 Vectorized load/store optimization

The number of load/store for the cases of 16x1 and 16x2 can be further reduced by using the vectorized load store function in OpenCL, such as `float4`, `int4`, and `char4`, etc. Table 9-5 shows the number of load/store requests for the vectorized cases (assuming the pixel data type is 8-bit char).

Table 9-5 Number of loads and stores by using vectorized load/store

	16x1 Vectorized	16x2 vectorized
Loads	6/16=1.375	8/32=0.374
stores	2/16=0.125	4/32=0.125

A code snippet doing the vectorized load is as follows:

```
short16 line_a = convert_short16(as_uchar16(*((__global uint4
*)(inputImage+offset))));
```

There are two pixels to be loaded at the boundary as follows:

```
short2 line_b = convert_short2(*((__global uchar2 *)(inputImage + offset +
16)));
```

NOTE: Increases in the number of pixels processed by each work item may cause serious register footprint pressure, resulting in register spilling into private memory and performance degradation.

9.3.4 Performance and summary

After applying the two optimization steps, significant performance boost is observed, as shown in Figure 9-12, in which the original (single pixel/work item) on MSM8992 (Adreno 418) is normalized to 1.

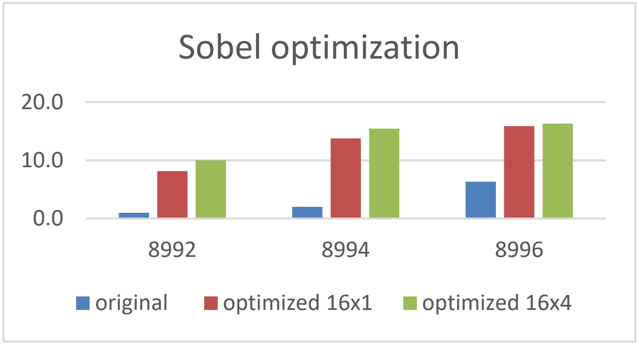

Figure 9-12 Performance boost by using data pack and vectorized load/store

To summarize, here are the key points for this use case optimization.

- Data packing improves memory access efficiency
- Vectorized load/store is key to reduce memory traffic
- Short type is preferred over integer or char type in this case

In this case, local memory is not used. The data pack and vectorized load/store have minimized the overlap of data that can be reused. Therefore, using local memory does not necessarily improve the performance.

There could be other options to boost the performance, for example, using texture over global buffer.

9.4 Summary

A few examples and code snippet are provided in this chapter to demonstrate the optimization rules presented in previous chapters and how the performance has changed. Developers should try to follow the steps with real devices. It is advised that not all results can be exactly reproduced due to compiler and driver upgrades. But generally, similar performance boost should be achieved with these optimization steps.

10 Summary

This document intends to provide a detailed guidance on how to optimize OpenCL programs with Adreno GPUs. A good amount of information has been provided to help developers understand the OpenCL fundamentals and Adreno architectures, and most importantly, master OpenCL optimization techniques.

OpenCL optimization is often challenging and requires a lot of trial and error. As each vendor may have its own best practices of doing the same task, it is important to read through and have in-depth understanding of the guide and practices for Adreno GPUs. Many factors that look minor could have significant performance impacts. These are unfortunately not easy to tackle without hands-on exercise and practices.

Due to time constraints and other factors, some topics are not covered. Adreno GPUs support a lot of extensions that can significantly boost performance and add extra functionalities. For instance, recent Adreno GPUs support some proprietary image formats which raw image and video data captured from image signal processor (ISP) can be compressed to for direct and efficient processing. This could save some manual handling as well as improve bandwidth usage.

Future releases of this document will include more topics.

A How to enable performance mode

To enable performance mode, root access is usually required for Android devices(`adb root; adb remount`). Note that these commands need to rerun if the system restarts.

A.1 Adreno A3x GPU

A.1.1 CPU settings

```
/*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"
```

A.1.2 GPU settings:

Disable power scaling policy:

- Newer targets support the following method to disable power scaling:

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

- Legacy targets support the following method to disable power scaling:

```
/*Disable power scaling policy for GPU*/
adb shell "echo none > /sys/class/kgsl/kgsl-3d0/pwrscale/policy"
```

Disable GPU sleep and 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 graphics 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"
```

A.2 Adreno A4x GPU and Adreno A5x GPU

```
adb shell "cat /sys/class/kgsl/kgsl-3d0/gpuclk"
adb shell "echo 0 > /sys/class/kgsl/kgsl-3d0/min_pwrlevel"
adb shell "echo performance >/sys/class/kgsl/kgsl-3d0/devfreq/governor"
adb shell "cd /sys/class/devfreq/qcom,cpubw.* && echo performance >
governor"
adb shell "echo performance >/sys/class/devfreq/qcom,cpubw.29/governor"
adb shell "echo 1 > /sys/devices/system/cpu/cpu0/online"
adb shell "echo 1 > /sys/devices/system/cpu/cpu1/online"
adb shell "echo 1 > /sys/devices/system/cpu/cpu2/online"
adb shell "echo 1 > /sys/devices/system/cpu/cpu3/online"
adb shell "echo 1 > /sys/devices/system/cpu/cpu4/online"
adb shell "echo 1 > /sys/devices/system/cpu/cpu5/online"
adb shell "echo 1 > /sys/devices/system/cpu/cpu6/online"
adb shell "echo 1 > /sys/devices/system/cpu/cpu7/online"
adb shell stop thermald
adb shell stop mpdecision
adb shell "echo performance >
/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor"
adb shell "echo performance >
/sys/devices/system/cpu/cpu1/cpufreq/scaling_governor"
adb shell "echo performance >
/sys/devices/system/cpu/cpu2/cpufreq/scaling_governor"
adb shell "echo performance >
/sys/devices/system/cpu/cpu3/cpufreq/scaling_governor"
```

```
adb shell "echo performance >
/sys/devices/system/cpu/cpu4/cpufreq/scaling_governor"
adb shell "echo performance >
/sys/devices/system/cpu/cpu5/cpufreq/scaling_governor"
adb shell "echo performance >
/sys/devices/system/cpu/cpu6/cpufreq/scaling_governor"
adb shell "echo performance >
/sys/devices/system/cpu/cpu7/cpufreq/scaling_governor"
adb shell "cat /sys/class/kgsl/kgsl-3d0/gpuclk"
adb shell "echo 1 > /sys/class/kgsl/kgsl-3d0/force_clk_on"
adb shell "echo 1000000 > /sys/class/kgsl/kgsl-3d0/idle_timer"
```

For GPU only:

```
adb shell echo 0 > /sys/class/kgsl/kgsl-3d0/min_pwrlevel
adb shell echo 0 > /sys/class/kgsl/kgsl-3d0/max_pwrlevel
```

B References

B.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Introduction to Adreno Profiler for OpenCL	80-ND791-5
CL-GL Interop Usage Guidelines	80-NK985-1
Standards	
The OpenCL Specification	The Khronos Group Inc. Versions 1.1, 1.2, and 2.0 http://www.khronos.org/opencl/
Resources	
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

B.2 Acronyms and terms

Acronym or term	Definition
ALU	arithmetic logic unit
ANB	Android native buffer
EFU	elementary function unit
GPR	general purpose register
IOT	internet of things
NDK	native development kit
OS	operating system-on-chip
RAM	random-access memory
SDK	software development kit
SOC	system-on-chip
SP	streaming or shader processor
SVM	shared virtual memory
TP	texture processor
UCHE	unified L2 cache
UI	user interface