Qualcomm Technologies, Inc.

Qualcomm® Snapdragon™ Mobile Platform OpenCL General Programming and Optimization

80-NB295-11 Rev. C

February 13, 2023

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

Revision history

Revision	Date	Description
A	November 2017	Initial release.
B	December 2022	Numerous changes were made to this document; it should be read in its entirety.
C	February 2023	Editorial updates.

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

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, `cp armcc armcpp`.

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, open a technical support case at https://support.qualcomm.com/.

You will need to register for a Qualcomm ID account and your company must have support enabled to access our Case system.

Other systems and support resources are listed on https://qualcomm.com/support.

If you need further assistance, you can send an email to qualcomm.support@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 better understand the OpenCL standard, 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 to help developers to exploit the massive computing power available in modern heterogeneous systems and facilitate application development across platforms.

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

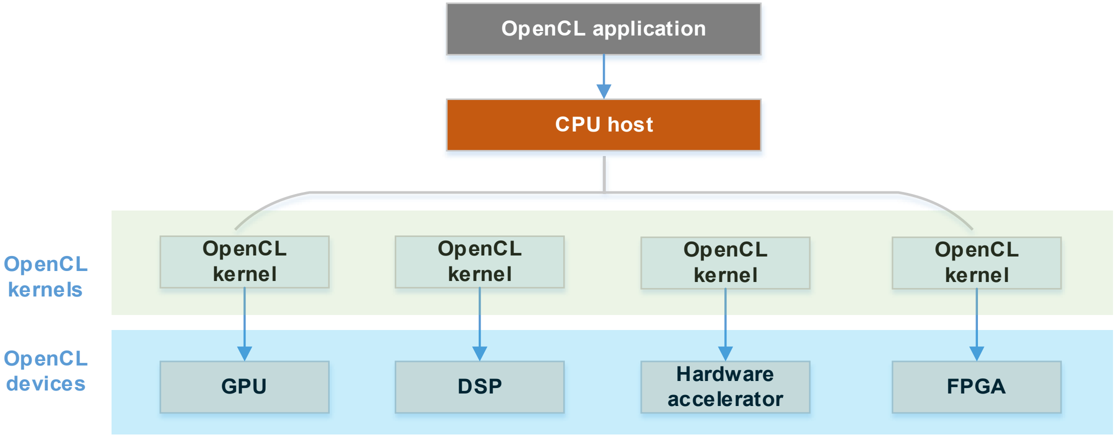


Figure 2-1 Heterogeneous system using OpenCL

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

- A host CPU that is a commander/master that manages and controls the application.
- OpenCL devices, including GPU, DSP, FPGA, and a hardware accelerator.
- Kernel codes that 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 immensely powerful and the top mobile GPUs reach the level of console/discrete GPUs in raw computing power. Developing applications using the computing power of GPUs without knowing their low-level details and maintaining applications' compatibility across different SOCs became a big challenge for developers.

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

Adreno GPUs have greatly accelerated many use cases via OpenCL, demonstrating 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 the host for resource management, kernel dispatch, and other tasks, while the OpenCL C language is used to write kernels that execute on OpenCL devices. The OpenCL API and OpenCL C language is quickly reviewed in the following sections.

2.3.1 OpenCL API functions

The OpenCL API functions can be classified into platform layer and runtime. The following tables 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. Check the extensions supported by the device.
Context	Context management, such as context creation, retain, and releases

Table 2-2 OpenCL runtime layer functionality

Functionality	Details
Command queue management	Used to communicate between the device and host and can have many queues in an application.

Functionality	Details
Create and build OpenCL programs and kernels	Is the kernel loaded and built successfully?
Prepare data for the 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 is 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 start 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 OpenCL 3.0 with the provisional SPIR-V 1.2 standard contains many improvements over the previous generation. Refer to References for more details.

OpenCL defines two profiles (embedded profile and full profile). Embedded profile 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 do not use vendor-proprietary or platform-specific extensions or features.

The program portability of OpenCL is ensured by the 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 not portable. As a high-level computing standard, the hardware implementation of OpenCL is vendor dependent. Different hardware vendors have different device architectures, each with advantages and disadvantages. Therefore, an OpenCL application written and optimized for one vendor's platform is unlikely to perform similarly 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. Therefore, 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 in old code can run on new versions of OpenCL. As some API functions may be deprecated in newer versions, the macros such as `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. Similar rules apply if OpenCL 2.x macros are used with the OpenCL 3.0 header file. OpenCL extensions may not carry forward to new devices, and applications using extensions need to examine if the new devices support them. This will be discussed in more detail in Chapter 9

3 OpenCL on Snapdragon

Snapdragon is one of the leading 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 a highly 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.

3.1 OpenCL on Snapdragon

Adreno GPUs have fully embraced OpenCL since the A3x GPUs. As OpenCL evolves with different versions and profiles, the support of OpenCL on Adreno GPUs also evolves. Table 3-1 shows the OpenCL versions and profiles available on the Adreno GPUs.

Table 3-1 Adreno GPUs and their OpenCL support

GPU series	Adreno A3x	Adreno A4x	Adreno A5x	Adreno A6x	Adreno A7x
OpenCL version	1.1	1.2	2.0	2.0	3.0
OpenCL profile	Embedded	Full	Full	Full	Full

Within the same OpenCL version or profile, some features and capabilities may vary across the Adreno GPU families, such as the availability of extensions, maximum dimensions of image objects, image formats, etc. A detailed list of the properties can be queried through the relevant OpenCL API functions, such as `clGetDeviceInfo` or `clGetPlatformInfo`.

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

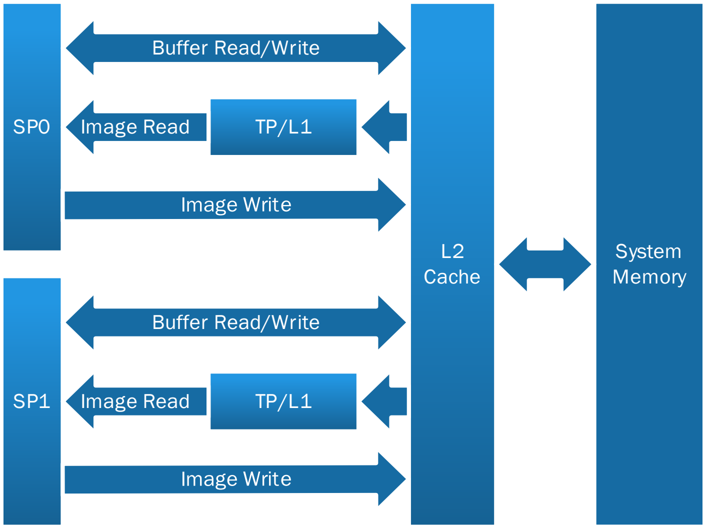


Figure 3-1 High-level architecture of the Adreno GPUs for OpenCL

Adreno GPUs support a rich set of graphics and compute APIs, including the OpenGL ES, OpenCL, DirectX, and Vulkan, etc. Figure 3-1 illustrates a high-level view of the Adreno hardware architecture for OpenCL, where the graphics-only hardware modules are skipped. There are many differences between each generation of Adreno GPUs, while for OpenCL the differences are minor. The key hardware modules for OpenCL execution are as follows:

- Shader (or streaming) processor (SP)
  - Core block of Adreno GPUs. Contains the key hardware modules such as the 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 in OpenCL, and therefore, executes one or multiple workgroups.
  - 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 are two SPs.
  - SPs load and store data through level 2 (L2) cache for buffer objects and image objects defined with the `__read_write` qualifier (OpenCL 2.0+ feature).
  - SPs load data from the texture processor/L1 module for read-only image objects.
- Texture processor (TP) and the L1 cache

  - The L1 cache is read-only.
  - TP fetches data from the L1 cache or the L2 cache in case the L1 cache misses.
  - TP performs texture operations based on the kernel's request, such as texture fetch and filtering.
- L2 cache
  - Supports read and write.
  - Handles the following requests:
    - Buffer data load/store from SPs.
    - Image data store.
    - Image data load from the L1 cache.
  - SP and TP are just two of many clients of the L2 cache.

3.2.2 Waves and fibers

The smallest unit of execution in Adreno GPUs is called fiber. One fiber corresponds to one work item in OpenCL. A collection of fibers that always execute in lockstep is called a wave. Here are some properties of waves and fibers:

- SP and waves:
  - SP can accommodate multiple active waves at a time.
  - SP can execute ALU instructions on one or more waves simultaneously.
  - Each wave can make independent forward progress, irrespective of the status of the other waves.
  - In newer Adreno GPUs, a SP can execute waves that belong to different workgroups.
  - A wave corresponds to a sub-group in OpenCL.
- The number of fibers and wave size:
  - Wave size depends on the Adreno GPU series and tiers as well as the compiler; values could be 8, 16, 32, 64, 128, etc. This can be queried through an API function (e.g., `sub_group_size()`).
  - Adreno GPUs support two modes, full wave mode and half wave mode. See Section 9.2.1 for more details.
  - Once compiled, the wave size of a kernel is fixed for a given GPU.
- The number of waves and workgroup size:
  - The maximum number of waves pipelined in the workgroup is hardware and kernel dependent.
  - For a given kernel, the maximum workgroup size is the product of the maximum allowed number of waves and the wave size.
    - Generally, the more complex the kernel is, the more registers it requires.

    - The total registers are fixed for a given GPU. The more registers a kernel requires, the fewer maximum allowed waves.
  - A workgroup may have 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.

OpenCL 1.x does not expose the concept of waves, while OpenCL 2.0 and above allow applications to use waves through the extension called `cl_khr_subgroups`. Many subgroup functions have become core features since OpenCL 2.1. OpenCL 3.0 has introduced a set of new reduction and shuffle functions for subgroups through KHR extensions. In addition, Adreno GPUs support a suite of other subgroup/shuffle functions via vendor extensions. See Section 9.2 for more details.

3.2.3 Latency hiding

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

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

This way, SP is primarily busy and working “full time” as the latency, or the dependency can be well hidden. A kernel could be latency bound if it does not have enough waves to hide latencies.

3.2.4 Level-2 (L2) cache

L2 cache is an essential block between the system memory and SPs. Here are a few key points to understand how it works:

- When a work item reads a byte from the system memory, the memory system does not only load the byte but the entire cache line that contains the byte. The cache line could be evicted by GPU hardware at some point to store a new cache line.
- When a work item writes a byte to the system memory, the memory system needs to load the cache line, modify it, and then write it out at some point.
- The kernel should use the data in the cache line as much as possible before the cache line is evicted.
- Cache sizes and the cache line size may vary across different GPU generations and tiers, though most Adreno GPUs have a cache line size of 64 bytes.
  - Developers can query a list of cache configurations through the `clGetDeviceInfo` API function. These include the following information:

    - Cache types (read-only, or read-write, `CL_DEVICE_GLOBAL_MEM_CACHE_TYPE`).
    - Size of the cache line in bytes (`CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE`, e.g., 64 bytes).
    - Cache size in bytes (`CL_DEVICE_GLOBAL_MEM_CACHE_SIZE`, e.g., 256KB).
  - □ Developers can derive the number of cache lines available in the GPU by dividing the cache size by cache line size (e.g., 4096 cache lines with a cache size of 256KB and a cache line size of 64 bytes).
- Cache thrashing, i.e., some cache lines must be evicted before reusing, may severely hurt performance if the L2 cache is overloaded.
  - □ Based on the cache size, developers may reduce or limit the workload of workgroups to avoid filling up the cache lines too quickly and improve the cache locality.

3.2.5 Workgroup assignment

A typical OpenCL kernel launches multiple workgroups. Adreno GPUs assign each workgroup to an SP, and each SP processes one or more workgroups simultaneously. If there are any, the remaining workgroups are queued for SPs to execute. Multiple SPs cannot process one workgroup.

In earlier Adreno GPUs, one SP can only process one workgroup at a time, and one workgroup must complete execution before another one can start on the SP. The premium tiers of Adreno A6x and A7x have lifted the restrictions and fully supported the concurrent workgroup execution per SP.

Take the 2D range in Figure 3-2 as an example and assume this is a GPU with 4 SPs. Figure 3-3 shows how different SPs process workgroups. In this example, there are nine workgroups. Assuming that no concurrent workgroups are running, each workgroup is executed by one SP. There are four waves per workgroup, and the wave size is 16.

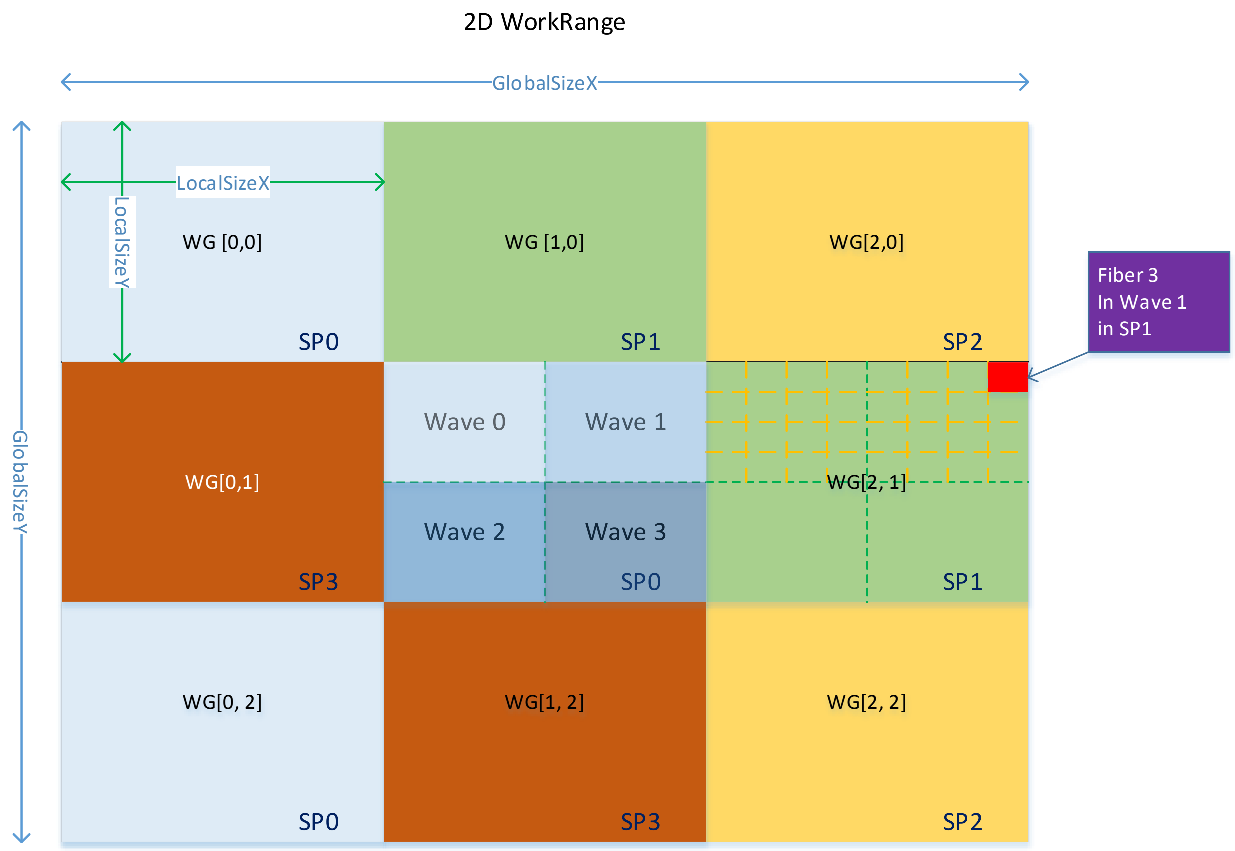


Figure 3-2 Example of workgroup layout and dispatch in Adreno GPUs

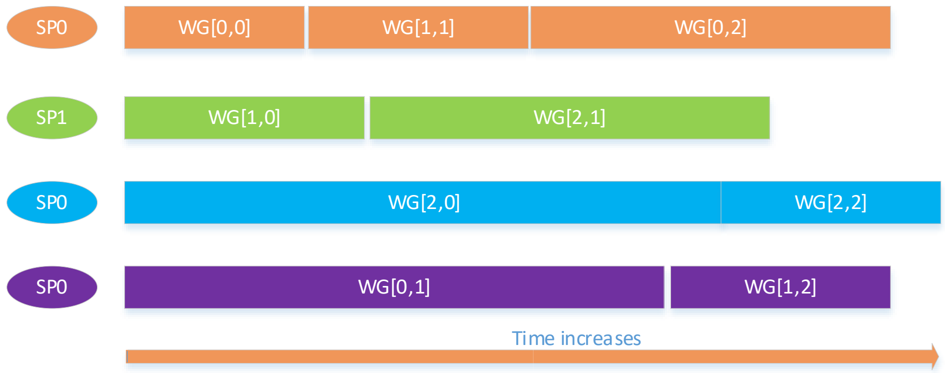


Figure 3-3 Example of workgroup allocation to SPs

The OpenCL standard neither defines the order of workgroup launching/execution, nor the method for workgroup synchronization. For Adreno GPUs, developers cannot assume the launch order of workgroups or waves in SPs.

3.2.6 Coalesced access

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, to improve the data load/store efficiency. Without coalesced access support, hardware must perform the load/store operation per each individual request, resulting in excessive requests and performance loss.

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, or within a certain address range (e.g., 128-bit range). 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.

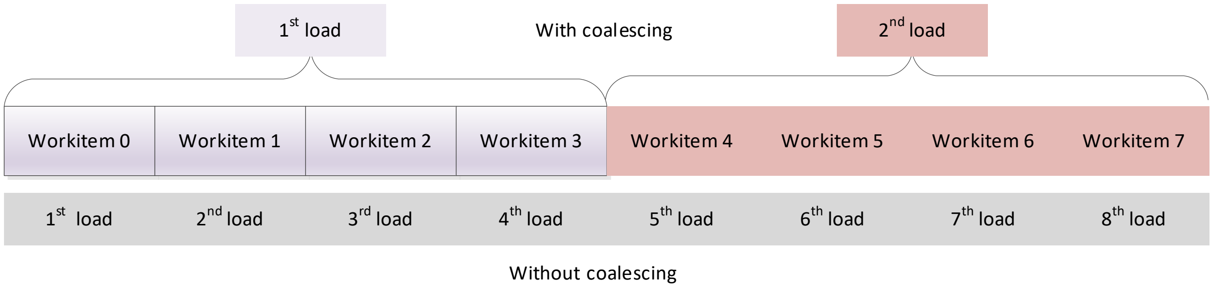


Figure 3-4 Illustration of coalesced vs. non-coalesced data load

3.3 Context switching between graphics and compute workload

3.3.1 Context switch

If a high priority task, such as graphics user interface (UI) rendering, is required while a low priority workload is running on Adreno GPUs, 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.

3.3.2 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 reset and leads to unpredictable consequences. Usually, the UI rendering on Android devices occurs with a fixed frequency, e.g., every 30 milliseconds. A long-running compute kernel could cause the UI to lag and be unresponsive, hurting the user experience. As a rule of thumb, the kernel execution time should be tens of milliseconds.

3.4 Support of OpenCL standard features

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, and A6x GPUs support OpenCL 2.0 full profile. From the OpenCL 1.1 embedded profile to the OpenCL 1.2 full profile, most 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 (KEK), etc. Table 3-2 lists the major differences on OpenCL profile support across the Adreno GPUs.

Table 3-2 Standard OpenCL features supported on Adreno GPUs

Features	**OpenCL 1.1 embedded** **Adreno A3x**	**OpenCL 1.2 full** **Adreno A4x**	**OpenCL 2.0 full** **Adreno A5x, A6x;** **OpenCL 3.0 in A7x and above**
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

3.5 Key features for OpenCL 2.0

3.5.1 Shared virtual memory (SVM)

As one of the most important features in OpenCL 2.0, SVM allows developers to write code that can share complex data structures such as linked lists or trees between the host and the device. Prior to OpenCL 2.0, a pointer created by the host may not be accessed directly by the kernel on the device and the data associated with the pointer cannot be shared.

With the support of SVM in OpenCL 2.0, the host and devices can share pointers and complex data structures that may contain pointers. In addition, SVM in OpenCL 2.0 also defines the memory consistency model so that the host and kernel can interact with each other using atomics for synchronization. This allows developers to write highly interactive parallel code between device and host with minimum synchronization costs.

Prior to OpenCL 2.0, host and device can only synchronize at certain points that must be explicitly specified in the code.

Since the Adreno A5x GPUs, the OpenCL 2.0 full profile has been fully supported, including the advanced SVM features such as the fine-grained buffer SVM with atomics support. There might be slight differences on the features available on the platform. It is recommended to use the API functions, `clGetDeviceInfo`, with the token `CL_DEVICE_SVM_CAPABILITIES` to query the exact features that are available on the platform.

In OpenCL 3.0, SVM becomes an optional feature through the feature macro mechanism, but the latest premium Adreno GPUs continue to support it.

3.5.2 OpenCL kernel enqueue kernel (KEK)

The KEK feature allows the work items in a kernel, which is the parent kernel, to enqueue a child kernel, without using the CPU host. This feature helps the workloads that are not known upfront and only are available during run-time. A work item can conditionally enqueue another kernel dependent on the runtime results. A typical use case can be the K-means clustering.

Adreno GPUs achieve the KEK feature without having GPU pass information to the CPU host to enqueue the new kernel, therefore saving the overhead of communication. In Adreno GPUs the child kernel will not start execution until the parent kernel is complete, though the OpenCL standard allows a child kernel to execute without waiting for its parent kernel to complete.

3.6 Critical features for OpenCL 3.0

The Khronos group has officially released the OpenCL 3.0 standard. The most noticeable change is many required and mandatory features in prior OpenCL standards become optional. The motivation behind this mechanism is to improve the foundation and influence of the OpenCL ecosystem. By lowering the bar of passing OpenCL conformance, many devices, such as embedded devices, FPGA, and other dedicated hardware, can be certified as OpenCL 3.0 conformant, without supporting some high complexity features (e.g., floating-point arithmetic, shared virtual memory, KEK, etc).

Starting from A7x, Adreno GPUs will support OpenCL 3.0. Ideally, developers should be able to run their OpenCL 1.x and 2.x applications on Adreno GPUs with OpenCL 3.0 support.

3.6.1 Feature macros

OpenCL 3.0 has introduced a mechanism called feature macro. It is to categorize majority of the standard features in older versions into different feature sets, and developers need to query whether the features are supported before using them. It is important to query for functionality correctness and code portability.

3.6.2 Backward compatibility

OpenCL applications developed for versions older than 3.0 should work without changes, as backward compatibility is typically guaranteed. However, developers should start writing code against the latest OpenCL 3.0 version to have better support in the long term.

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

Adreno GPUs support OpenCL mainly on the Android operating system (OS) and selected Linux systems. To develop an Android app that runs with OpenCL, developers need to use the Android software development kit (SDK) and the Android native development kit (NDK). Refer to https://developer.android.com/index.html and https://developer.android.com/ndk/index.html for the Android SDK and NDK, respectively. Throughout this chapter and the following chapters, it is assumed that the development is on the 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 the 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 the device.
    - Some vendors may choose not to include the OpenCL software (e.g., 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 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 Adreno OpenCL SDK and Adreno OpenCL machine learning SDK

Developers can find the latest Adreno OpenCL SDK and the Adreno OpenCL machine learning SDK at https://developer.qualcomm.com/software/adreno-gpu-sdk/tools. The OpenCL SDK features code examples and documents that help developers to understand and effectively use the latest Adreno OpenCL features. With a proprietary set of API functions and hand-optimized kernels, the machine learning SDK helps developers use the Adreno GPU for the machine learning inference and training applications. 0

4.3 Debugging tools and tips

Debug of OpenCL application is often challenging due to the parallel nature of GPU execution. For kernel debugging, OpenCL supports the `printf` function, which is similar to the standard `printf` in c99 with a few minor differences. It is recommended to reduce the workload by printing only the necessary variables (using conditions to limit the output), as `printf` typically 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, developers can use API functions such as `clGetDeviceInfo` or `clGetPlatformInfo`. For more details, refer to References.

Debugging tips on Adreno GPUs

- Use barrier or fence inside the kernel to prevent the compiler from reordering the code before/after it.
- If error codes are returned, check the OpenCL specification for more information.
- Isolate the problem by debugging one work item/pixel/workgroup/kernel.
  - For example, set `global_work_size = [1]`, and `global_offset` to the pixel coordinate [x].
- If a crash is observed in an API function/kernel, here are a few things to check:
  - Invalid memory address.
  - If other API functions have issues.
  - If the memory is not updated as expected.
  - If there is an overflow or the memory buffer size is incorrect.
    - Try one pixel only.
  - If the kernel execution is not complete yet.
    - Use `clFinish/clWaitforEvent` to make sure the kernel execution is complete.
  - If results are incorrect and unstable.

- □ If different work items write to the same memory address, or there is missing synchronization or barrier.

4.4 Snapdragon profiler (SDP)

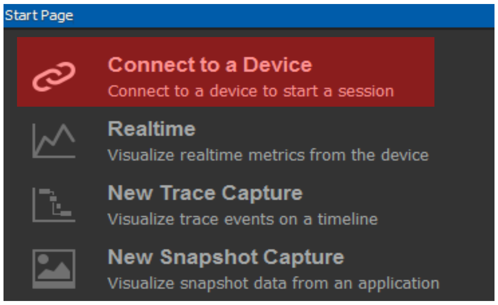
The Snapdragon profiler is a free profiling tool that runs on Windows, Mac, and Linux platforms and allows developers to analyze CPU, GPU, DSP, memory, power, thermal, and network data of Snapdragon processors. It supports OpenCL and many graphics APIs, such as OpenGL ES and Vulkan. For more details, refer to https://developer.qualcomm.com/software/snapdragon-profiler, where the executable can be downloaded for Windows, MacOS, and Linux, along with its user guide. A brief YouTube video introduction is available at Capturing OpenCL applications in Snapdragon profiler. The following are some key features offered by the Snapdragon profiler for OpenCL profiling.

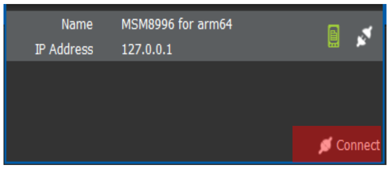
- The profiler has a kernel analyzer that allows developers to do static analysis for a given kernel. It provides information such as register footprint and the number of instructions to help developers optimize kernels.
- The profiler provides OpenCL API traces and logs for a given OpenCL application. 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 the command line-based applications, as well as Android GUI apps.

This section briefly presents how to profile an OpenCL application.

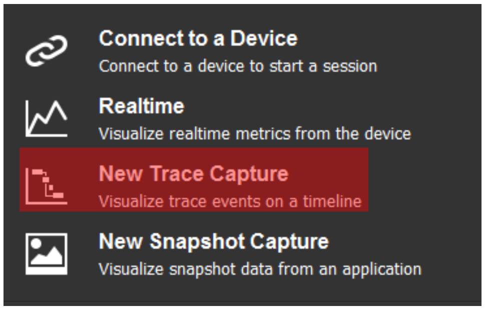
4.4.1 Steps to use SDP

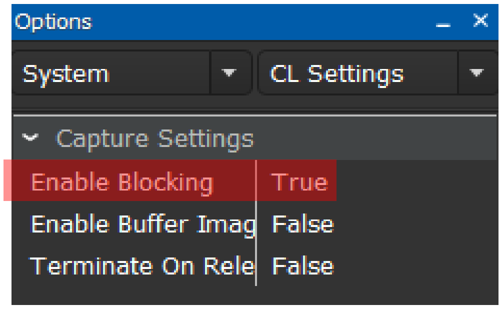
1. Connect the profiler with the device.


Start Page
Connect to a Device
Connect to a device to start a session
Realtime
Visualize realtime metrics from the device
New Trace Capture
Visualize trace events on a timeline
New Snapshot Capture
Visualize snapshot data from an application


Name MSM8996 for arm64
IP Address 127.0.0.1

2. Set up the configuration for OpenCL profiling.
   a. Choose the OpenCL Layout.
   b. Enable Blocking must be set to True.
   c. Choose New Trace Capture.

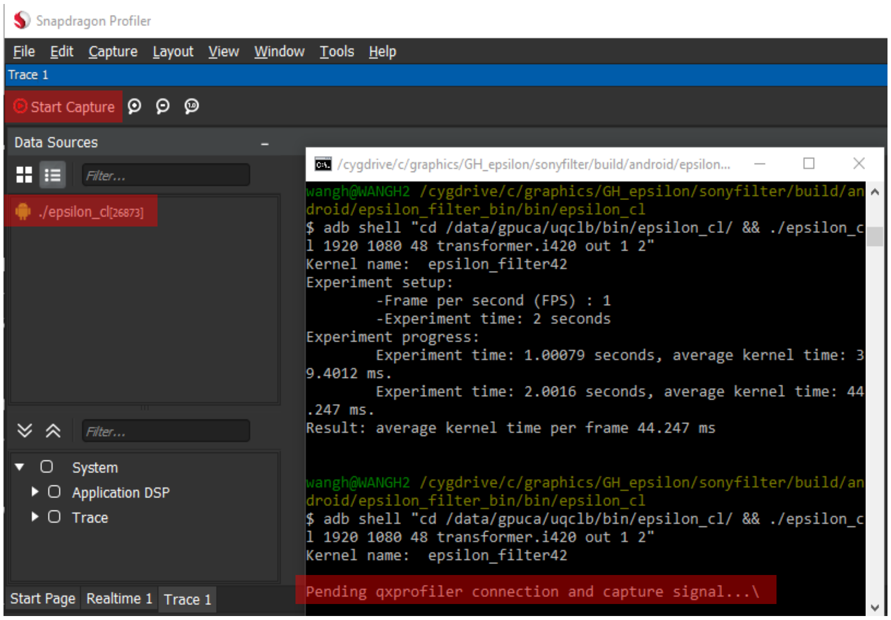

Connect to a Device
Connect to a device to start a session
Realtime
Visualize realtime metrics from the device
New Trace Capture
Visualize trace events on a timeline
New Snapshot Capture
Visualize snapshot data from an application



Enable Blocking True
Enable Buffer Imag False
Terminate On Rele False

3. Launch the OpenCL application in a command window.

   a. A message similar to “Pending qxprofiler connection and capture signal…\” should pop up in the command window where the application is launched, waiting for the next step.


Snapdragon Profiler
File Edit Capture Layout View Window Tools Help
Trace 1
Start Capture
Data Sources
Filter...
./epsilon_cl[26873]
Filter...
System
Application DSP
Trace
Start Page Realtime 1 Trace 1
/cygdrive/c/graphics/GH_epsilon/sonyfilter/build/android/epsilon...

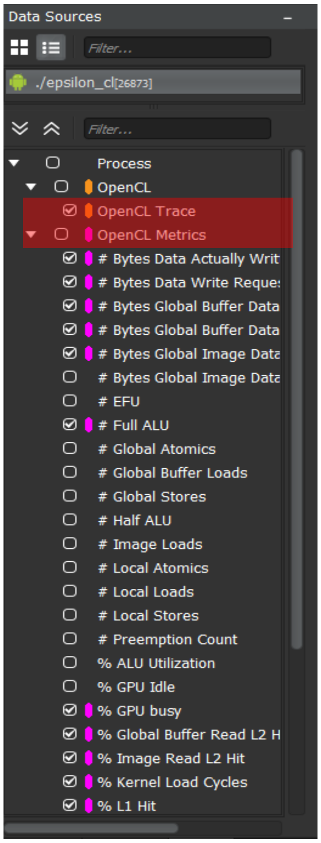
```
wangh@WANGH2 /cygdrive/c/graphics/GH_epsilon/sonyfilter/build/an
droid/epsilon_filter_bin/bin/epsilon_cl
$ adb shell "cd /data/gpuca/uqclb/bin/epsilon_cl/ && ./epsilon_c
l 1920 1080 48 transformer.i420 out 1 2"
Kernel name:  epsilon_filter42
Experiment setup:
        -Frame per second (FPS) : 1
        -Experiment time: 2 seconds
Experiment progress:
        Experiment time: 1.00079 seconds, average kernel time: 3
9.4012 ms.
        Experiment time: 2.0016 seconds, average kernel time: 44
.247 ms.
Result: average kernel time per frame 44.247 ms


wangh@WANGH2 /cygdrive/c/graphics/GH_epsilon/sonyfilter/build/an
droid/epsilon_filter_bin/bin/epsilon_cl
$ adb shell "cd /data/gpuca/uqclb/bin/epsilon_cl/ && ./epsilon_c
l 1920 1080 48 transformer.i420 out 1 2"
Kernel name:  epsilon_filter42

Pending qxprofiler connection and capture signal...\
```

   b. Ideally, the OpenCL application should be detected and shown on the left panel of the profiler GUI.

4. Choose the metrics to profile.



Data Sources

Filter...

./epsilon_cl[26873]

Filter...

- □ Process
  - □ OpenCL
    - ☑ OpenCL Trace
    - □ OpenCL Metrics
      - ☑ # Bytes Data Actually Writ
      - ☑ # Bytes Data Write Reque:
      - ☑ # Bytes Global Buffer Data
      - ☑ # Bytes Global Buffer Data
      - ☑ # Bytes Global Image Datε
      - □ # Bytes Global Image Datε
      - □ # EFU
      - ☑ # Full ALU
      - □ # Global Atomics
      - □ # Global Buffer Loads
      - □ # Global Stores
      - □ # Half ALU
      - □ # Image Loads
      - □ # Local Atomics
      - □ # Local Loads
      - □ # Local Stores
      - □ # Preemption Count
      - □ % ALU Utilization
      - □ % GPU Idle
      - ☑ % GPU busy
      - ☑ % Global Buffer Read L2 H
      - ☑ % Image Read L2 Hit
      - ☑ % Kernel Load Cycles
      - ☑ % L1 Hit

   a. Click the application name, then OpenCL Trace and OpenCL Metrics appear in the bottom left panel.

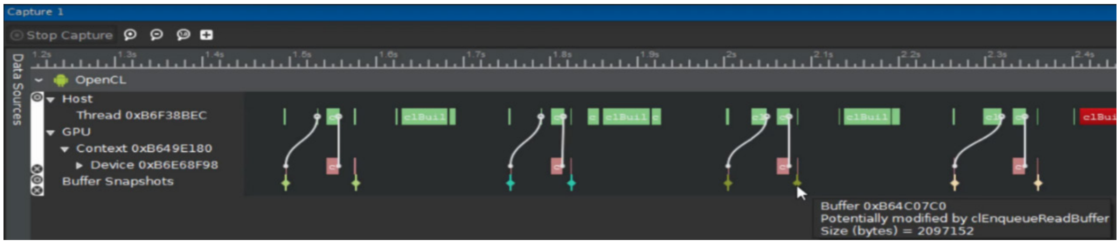
   b. Choose the trace and metrics for profiling.

5. Collect the results.

   a. Click **Start to capture** and the application should resume its execution until completion.

   b. The history of API functions is available in the main window and can be zoomed in/out.

   c. Profiling metrics are shown in the bottom window.



Buffer 0xB64C07C0
Potentially modified by clEnqueueReadBuffer
Size (bytes) = 2097152

   d. A new trace session can be initiated by clicking **Capture->New trace**.

   e. Results can be exported to a CSV file for offline analysis.

4.4.2 How to interpret the metrics from SDP

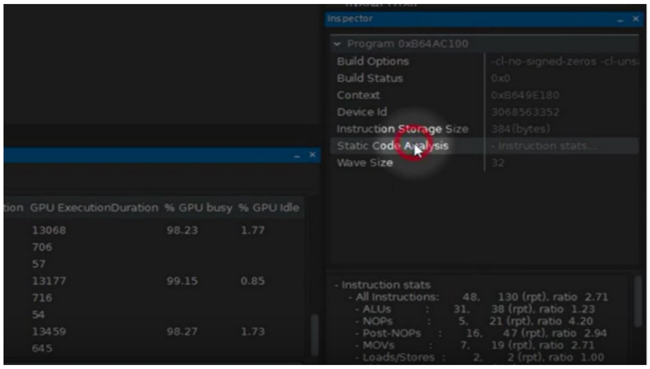
Here are a few critical metrics for OpenCL application profiling.

- ALU utilization %.
  - A low value may indicate the kernel could be memory bound.
  - Improve the efficiency of the data load/store.
- L2 global buffer read %.
  - A low value may indicate L2 cache is not well used (possible cache thrashing).
  - Balance the workload of the workgroups and tune the workgroup size.
- L1 hit %.
  - If a kernel does not use image objects, it will be 0.
- GPU busy % and idle %.
  - GPU should be fully busy when executing kernels.
  - Ideally the busy % should be close to 100%.
  - Anything below 90% should be a warning sign that something is not right in the host that causes the GPU to idle while the CPU is busy.
    - Use an event-based pipeline and reduce synchronization between the device and the host.

4.4.3 How to effectively use the profiler

- Identify the bottleneck.
  - If a metric and the performance do not improve while other metrics are improved, the unchanged metric may be the bottleneck.
- ALU bound vs memory bound vs latency bound:
  - Most real-life cases are memory bound.
  - Some could be ALU bound, in some convolution or matrix multiplication cases.
  - Latency bound means there may not be enough waves to hide latency
    - See Section 6.1 for more information on how to tune workgroup sizes.
- Compare the data from the counter to the theoretical numbers.
  - Help identify cache thrashing:
    - Developers may check the amount of data loaded into the GPU.
    - If the number of bytes exceeds the theoretically required data, cache thrashing will likely happen.
  - Developers can design some simple micro-benchmarks to help understand the GPU behavior.

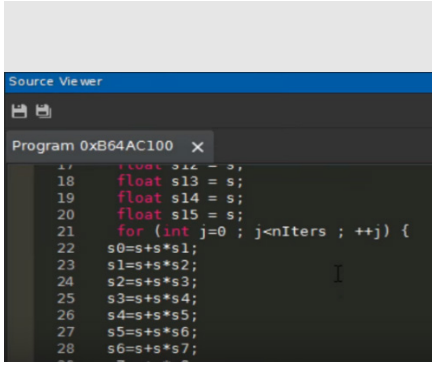
4.4.4 SDP: static code analysis



Build Options	-cl-no-signed-zeros -cl-uns
Build Status	0x0
Context	0xB649E180
Device Id	3068563352
Instruction Storage Size	384(bytes)
Static Code Analysis	- Instruction stats...
Wave Size	32

tion	GPU ExecutionDuration	% GPU busy	% GPU Idle
	13068	98.23	1.77
	706		
	57		
	13177	99.15	0.85
	716		
	54		
	13459	98.27	1.73
	645		

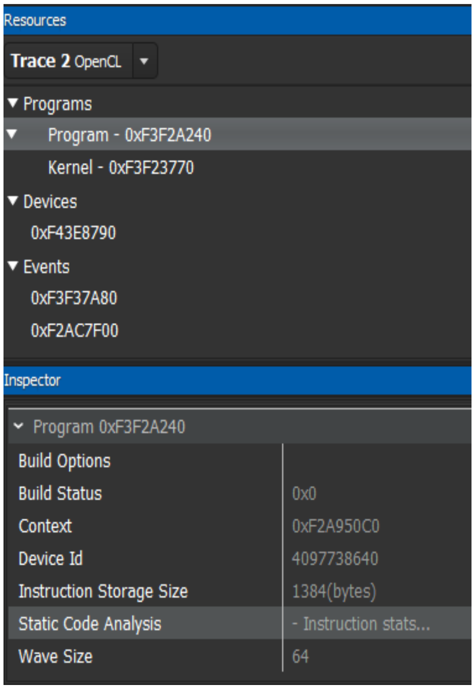
```
- Instruction stats
  - All Instructions:    48,   130 (rpt), ratio  2.71
    - ALUs          :   31,    38 (rpt), ratio  1.23
    - NOPs          :    5,    21 (rpt), ratio  4.20
    - Post-NOPs     :   16,    47 (rpt), ratio  2.94
    - MOVs          :    7,    19 (rpt), ratio  2.71
    - Loads/Stores  :    2,     2 (rpt), ratio  1.00
```



```
    float s13 = s;
    float s14 = s;
    float s15 = s;
    for (int j=0 ; j<nIters ; ++j) {
   s0=s+s*s1;
   s1=s+s*s2;
   s2=s+s*s3;
   s3=s+s*s4;
   s4=s+s*s5;
   s5=s+s*s6;
   s6=s+s*s7;
```

The profiler features a static code analyzer tool that developers can use to derive essential information about the kernel. The following are some of the most important ones.

- All instructions.
- ALU/half ALUs/EFUs.
- Buffer load/store.
  - LDG/STG: global buffer load/store.
  - LDL/STL: local memory load/store.
  - LDP/STP: private memory load/store (this often indicates a register spilling).
- Full registers and half registers.
  - Determine the number of waves.
  - The more registers, the fewer active waves.
- Maximum number of waves/maximal waves.
  - Optimize kernel/reduce complexity if maximal waves <4.



Resources

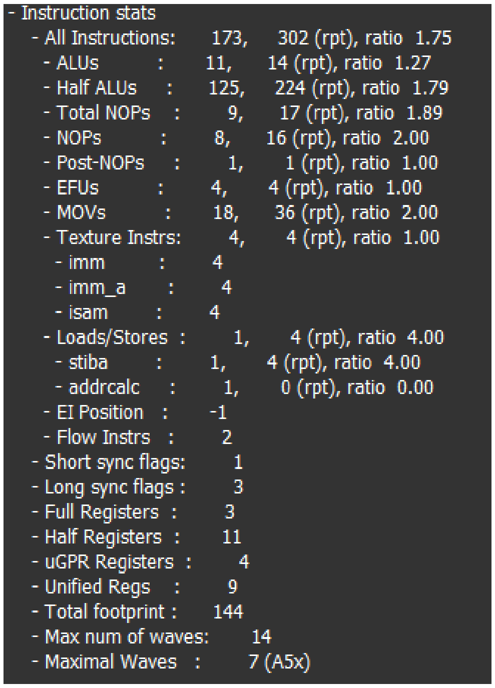
Trace 2 OpenCL

- Programs
  - Program - 0xF3F2A240
    - Kernel - 0xF3F23770
- Devices
  - 0xF43E8790
- Events
  - 0xF3F37A80
  - 0xF2AC7F00

Inspector

Program 0xF3F2A240

Build Options	
Build Status	0x0
Context	0xF2A950C0
Device Id	4097738640
Instruction Storage Size	1384(bytes)
Static Code Analysis	- Instruction stats...
Wave Size	64



```
- Instruction stats
  - All Instructions:     173,    302 (rpt), ratio  1.75
   - ALUs        :     11,     14 (rpt), ratio  1.27
   - Half ALUs   :    125,    224 (rpt), ratio  1.79
   - Total NOPs  :      9,     17 (rpt), ratio  1.89
   - NOPs        :      8,     16 (rpt), ratio  2.00
   - Post-NOPs   :      1,      1 (rpt), ratio  1.00
   - EFUs        :      4,      4 (rpt), ratio  1.00
   - MOVs        :     18,     36 (rpt), ratio  2.00
   - Texture Instrs:    4,      4 (rpt), ratio  1.00
    - imm        :      4
    - imm_a      :      4
    - isam       :      4
   - Loads/Stores :     1,      4 (rpt), ratio  4.00
    - stiba      :      1,      4 (rpt), ratio  4.00
    - addrcalc   :      1,      0 (rpt), ratio  0.00
   - EI Position :     -1
   - Flow Instrs :      2
  - Short sync flags:   1
  - Long sync flags :   3
  - Full Registers  :   3
  - Half Registers  :  11
  - uGPR Registers  :   4
  - Unified Regs    :   9
  - Total footprint : 144
  - Max num of waves:     14
  - Maximal Waves   :     7 (A5x)
```

The following are the most critical issues developers should handle among all the data.

- Try all means to remove STP/LDP, if there is one.
  - See Section 7.1.4 for more details.
- Optimize the code to reduce the “total footprint” and increase the “maximum number of waves.”
  - These are highly correlated.
  - The smaller the footprint, the more waves the GPU can execute in parallel and the better the performance.

4.5 Performance profiling

Given an application, 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.

4.5.1 CPU timer

Developers should use the date and time functions in the standard library of the C/C++ programming language to measure the full execution time of OpenCL calls from the host side. 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)));
    }
```

Some of the OpenCL runtime API functions with “enqueue” in their names accept a flag parameter to indicate whether it is a blocking or a non-blocking call. For non-blocking calls, the CPU timer must be used carefully.

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

When using a CPU timer to measure the kernel execution time from the host side, developers must make sure the function is complete by using either the `clWaitforEvent` call (if there is an event ID for the non-blocking call) or `clFinish`. The same rule applies to the API functions for memory copy.

4.5.2 GPU timer

The OpenCL enqueue function calls optionally return an event object to the host, which the OpenCL profiling APIs can use to query the execution time. Adreno GPUs have a clock and timer to measure the function execution flow. The GPU execution time is provided by GPU hardware counters independent of the operating system.

To enable the GPU timer functionality, developers must set the `CL_QUEUE_PROFILING_ENABLE` flag 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 execution is complete, developers can use the API function `clGetEventProfilingInfo` 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 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 the GPU is processing.
- The actual kernel execution time on the GPU differs between `CL_PROFILING_COMMAND_(START` and `END)`.

Developers should focus on minimizing the actual kernel execution time, which is relatively more straightforward than the other two timers, which are typically hard to control.

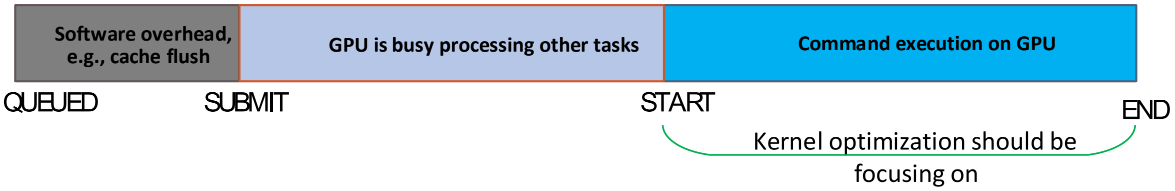

Software overhead, e.g., cache flush
GPU is busy processing other tasks
Command execution on GPU
QUEUED
SUBMIT
START
END
Kernel optimization should be focusing on

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

The kernel-enqueue-kernel (see Section 3.5.2) feature in OpenCL 2.0 introduced a new profiling flag called `CL_PROFILING_COMMAND_COMPLETE`. This flag returns the current device time counter in nanoseconds when the command identified by the event and any child commands enqueued by this command on the device have finished execution.

4.5.3 GPU timer vs. CPU timer

Developers can use GPU and CPU timers to profile performance. Though the GPU timer can accurately measure the GPU execution time, some hardware operations (e.g., cache flush) and software operations (e.g., synchronization between CPU host and GPU) are out of the GPU clock system. Therefore, the GPU timer is likely to report a better performance number than the CPU timer for kernel execution. The following are the two practices recommended.

- Use the GPU timer to measure kernel optimization. The GPU timer can tell precisely the amount of improvement each optimization step achieves from a GPU execution perspective.
- Use the CPU timer to measure the end-to-end performance of the application if the OpenCL program is only part of an entire application pipeline.

4.5.4 Performance mode

Snapdragon SOCs have an advanced dynamical clock and voltage control mechanism that automatically controls the system to run at power saving mode to save battery under specific scenarios. Typically, suppose there is an intensive workload. In that case, the system may automatically raise the clock rate and voltage, pushing the device into so-called performance mode to boost performance and meet the workload demand.

Given an OpenCL application, it would be difficult to understand and profile its performance if the system dynamically changes the clock rate. Therefore, developers should enable the performance mode for the sake of profiling consistency and accuracy.

Without the performance mode settings, the first OpenCL kernel in a sequence typically shows more significant launch latency and slower execution time. Developers can use simple kernels to warm up the GPU before launching the actual GPU workload.

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

- Ensure that the application renders full screen so that no other activity updates the screen.
- If it is a native application, be sure that `SurfaceFlinger` is not running on Android. This ensures that the application is profiled solely by CPU and GPU.

The sequence of commands needed to enable performance mode is slightly different for the different families of Adreno GPUs. Refer to Section A for more details.

4.5.5 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 power and clock 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, developers should use it carefully as it has significant implications for SOC-level power consumption.

5 Overview of performance optimizations

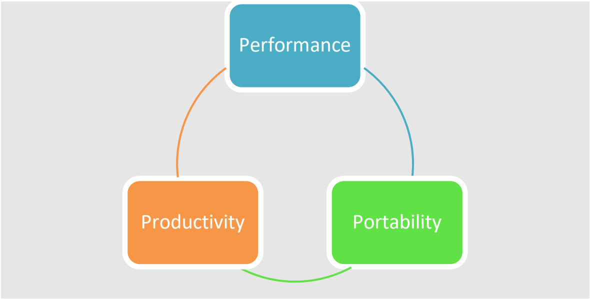
This chapter provides a high-level overview of OpenCL application optimization. More details are in the following few chapters.

Optimization of an OpenCL application can be challenging. It often requires more effort than the initial development.

5.1 Performance portability

As discussed in Section 2.4.2, the performance of OpenCL heavily depends on the low-level architecture of the hardware platform and generally is not portable across different architectures. OpenCL applications optimized on other platforms, especially on discrete GPUs, are unlikely to perform well on mobile GPUs. Therefore, other OpenCL vendors' best practices may not apply to Adreno GPUs. Reading through this entire document for optimization work on Adreno GPUs is crucial. Also, an OpenCL application optimized for an Adreno GPU may need further tuning and optimization to achieve optimal performance on other Adreno GPUs.

Performance, productivity, and portability are often the three things that developers need to balance. It is difficult to achieve all three goals simultaneously, as they depend on the time, budget and objective. For example, better portability requires more generic code, as some specific optimization tips may only apply to some GPUs. Similarly, achieving better productivity, i.e., to be able to rapidly develop, optimize, and deploy an application, usually requires a sacrifice in performance or portability. Developers must make careful decisions based on their priorities and objectives.



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, caches, etc.
- The optimal ways to leverage computing resources such as the ALU and texture operations.

The application-level optimization strategy is in the remaining sections of this chapter, while the other levels are in the following chapters.

5.3 Initial evaluation for OpenCL porting

Developers must assess whether an application is suitable for OpenCL before porting it blindly. Following are the typical characteristics of a good candidate for OpenCL acceleration on GPUs:

- Extensive input data set.
  - The communication overhead between CPUs and GPUs may overshadow the performance gain of OpenCL for small input data sets.
- Computationally intensive.
  - GPUs have many computing units (ALUs), and their peak computing power is usually much higher than CPUs. To fully utilize a GPU, applications should have reasonably high computational complexity.
- Parallel computation friendly.
  - A workload may be partitioned into small independent units, and the processing of each unit does not affect the others.
  - Parallelized tasks are needed for GPUs to hide latency.
- Limited divergent control flow.
  - GPUs generally do not handle divergent control flow as efficiently as CPUs. If the use case requires a lot of conditional checks and branching operations, CPUs 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 crucial 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.

- Developers may need to modify data structures to tailor the data flow and reduce overall traffic.

5.5 Parallelize GPU and CPU workloads

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

- Have the CPU execute the part that is best suited 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 if they require a lot of synchronization or data transfer. Instead, try shifting lightweight CPU tasks to the GPU, even though it may not be GPU-friendly, to eliminate the overhead.

5.6 Bottleneck analysis

Bottlenecks are the slowest stages that developers should spend the most time on. No matter how efficient all the other stages are, an application's performance is limited by its bottlenecks. Identifying and analyzing bottlenecks is crucial and not always straightforward. This section briefly discusses identifying and resolving bottlenecks.

5.6.1 Identify bottlenecks

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

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

As discussed in Section 4.3, the Snapdragon profiler 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 amount of computing.
  - Use fast relaxed math or native math.
  - Use the 16-bit floating point format instead of the 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 the following chapters.

NOTE: The bottleneck could shift as optimization progresses. A memory-bound problem could become an ALU-bound problem after resolving the memory bottleneck, or vice versa. Several backs and forth iterations are necessary to obtain the optimal performance. Also, bottlenecks could be different for different GPUs.

5.7 API level performance optimization

The OpenCL API functions to manage resources and control application execution mostly run on the CPU host. Though those functions are not as demanding as kernel executions, improper use of API functions could lead to a hefty performance penalty. The following are several 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 the execution loop. For example, the following functions can take a considerable amount 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.

Do not forget to fall back to building from source when `clCreateProgramWithBinary` fails. However, this could occur if the OpenCL software has incompatible updates.

- Avoid creating or releasing memory objects between the kernel calls. The execution time of `clCreate{Image|Buffer}` is related to the amount of memory requested (if `host_ptr` is used).
- If possible, use the 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. The host should be doing lightweight work during the GPU kernel launch to avoid stalling GPU's execution.

5.7.2 Use an 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. Using this method significantly reduces the overhead of launching API function calls. The software can schedule the functions best, and the host does not have to interfere with the API function calls. Therefore, it is highly desirable to streamline the API functions using an event-driven pipeline. In addition, developers should note the following:

- Avoid blocking API calls. Blocking calls requires the host CPU to wait for the GPU to finish, then stall the GPU before the next `clEnqueueNDRangeKernel` call. Blocking API calls are useful mainly 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. This asynchronous call mechanism allows more efficient pipeline execution as the host is now more flexible in handling the events.

5.7.3 Kernel compiling and building

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

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

5.7.4 Backward compatibility of binary kernel

Different versions of the binary code are needed if the application targets different tiers of Adreno devices.

- Binary code can be used only for the specific GPU for which it is compiled. For example, a binary compiled for Adreno A730 GPU cannot be used for Adreno A740 GPU.
- There is no guarantee that for the same device, a binary built from one version of OpenCL software can be reused directly with a newer version of OpenCL software.
- If the binary kernel is incompatible, use `clCreateProgramWithSource` as a fallback solution.

5.7.5 Use in-order command queues

The Adreno OpenCL platform supports the out-of-order command queues. However, there is a significant overhead due to the dependency management required for implementing out-of-order command queues. The Adreno software efficiently pipelines the commands for an in-order queue. Therefore, using in-order command queues rather than out-of-order ones is good practice.

6 Top kernel optimization tips on Adreno GPUs

This chapter provides a few top OpenCL optimization tips for Adreno GPUs, with more details and other information to be described in the following chapters. All the suggestions in this chapter should have the highest priority, and developers should try them before trying others when doing kernel optimization.

6.1 Workgroup performance optimization

A kernel's workgroup size and shape considerably impact performance, and tuning workgroup sizes is a simple and effective performance optimization method. This section presents the essential information on workgroups, including how to obtain the workgroup size given a kernel, why the workgroup size tuning is necessary, and standard practices on optimal workgroup size tuning.

6.1.1 Obtain the maximum workgroup size

Developers should always query the maximum workgroup size of a kernel on a device by using the following API function after running `clBuildProgram`:

```
size_t maxWork-groupsize;
clGetKernelWorkgroupInfo(myKernel,
                         myDevice,
                         CL_KERNEL_WORK_GROUP_SIZE,
                         sizeof(size_t),
                         &maxWork-groupsize,
                         NULL);
```

The actual workgroup size used by `clEnqueueNDRangeKernel` cannot exceed `maxWork-groupsize`. If the workgroup size is unspecified by the application, the Adreno OpenCL software will select a default and working workgroup size.

6.1.2 Required and preferred workgroup size

A kernel may require or prefer a certain workgroup size to work properly or effectively. OpenCL provides a few attributes to allow a kernel to request or ask for a specific workgroup size from the compiler:

- Use the `reqd_work_group_size` attribute.

The `reqd_work_group_size(X, Y, Z)` attribute explicitly requires a specific workgroup size. A compilation error is returned if the specified workgroup size cannot be satisfied by the compiler.

For example, to require a 16x16 workgroup 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 hinted workgroup size but does not guarantee the actual workgroup size matches the hint. For example, to hint at a workgroup size of 64x4:

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

In most cases, with the workgroup size restrictions, the compiler cannot guarantee that it generates the optimal machine code. Also, the compiler may have to spill registers to system memory if it cannot meet the required workgroup size using the on-chip registers. Therefore, developers are discouraged from using these two attributes unless the kernel needs the workgroup size to work correctly.

Writing a kernel that relies on a fixed workgroup size or layout is unsuitable for cross-platform compatibility and portability purposes.

6.1.3 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 (the required number of registers). Generally, the more complex a kernel is, the larger the register footprints and the 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 temporarily allocating additional registers to 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 sizes.
- Barriers in the kernel.

□ If a kernel does not use barriers, its maximum workgroup size can be set to the DEVICE MAXIMUM in the Adreno A4x, A5x, A6x, and A7x series, regardless of the register footprint.

□ See Section 6.1.4 for more details.

□ See Section 6.1.4 for more details.

6.1.4 Kernels without a barrier (steaming mode)

Traditionally, all work items in the workgroup are required to be resident on the GPU simultaneously. For kernels with a heavy register footprint, this can restrict their workgroup size to 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 between waves, a new wave can start to execute when an old wave is complete for these types of kernels.

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

6.1.5 Workgroup size and shape tuning

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

6.1.5.1 Avoid using the 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 good practice to manually try different workgroup sizes and shape layouts (for 2D/3D) and find the optimum one.

6.1.5.2 Large workgroup size, better performance?

This is true for most 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, some kernels may have worse performance with increasing workgroup sizes. An example is when a 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 L2 cache. Finding the best workgroup size and shape requires a lot of trial and error.

6.1.5.3 Fixed vs. dynamic workgroup size

For performance portability across devices, avoid assumptions that one workgroup size fits all and hard-coded workgroup size. A workgroup size and layout that works best on one GPU may be suboptimal on another one. Therefore, developers should profile different workgroup sizes for all devices with which the kernel can execute and select the best one for each device at runtime.

6.1.5.4 One vs. two vs. three-dimensional (1D/2D/3D) kernel

A kernel can support up to three dimensions. And the choice of its dimensionality may have performance implications. Compared to a 1D kernel where each work item has only 1D indices (e.g., global ID, local ID, etc.), a 2D kernel has an extra set of these built-in indices and potentially has a performance boost if these indices help save some calculation.

Depending on the data access pattern by work items, a 2D kernel may have better data locality in the cache, leading to better memory access and performance. While in other cases, a 2D kernel may result in worse cache thrashing than a 1D kernel. It would be good to try different dimensions with the kernel for optimal performance. Ideally, the workgroup size on the first dimension should be multiples of `sub_group_size`, which is especially important if the kernel has divergence.

6.1.6 Other topics on workgroup

6.1.6.1 Global work size and padding

OpenCL 1.x requires that a kernel's global worksize be 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.

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 (e.g., 32) to fully utilize the wave resources. If this is not the case, consider padding the workgroup size to meet this condition.

6.1.6.2 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 with similar complexity as the actual workload (but perhaps a smaller workload) to dynamically search the optimal workgroup size at the start of the application and then use the selected workgroup size for the actual kernel. Commercial benchmarks rely on this method.

6.1.6.3 Avoid uneven workload across workgroups

Applications may have uneven workload distribution across workgroups. For instance, region-based image processing may have regions that take more resources to process than others. Distributing them evenly to workgroups may create a balance issue. It can also complicate the context switch if a single workgroup takes too long to finish.

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

6.1.6.4 Workgroup synchronization

OpenCL does not guarantee the execution order of workgroups and does not define a mechanism for workgroup synchronization. Developers should never assume the order of the workgroups running on GPUs.

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

6.1.6.5 Persistent thread

Launching a workgroup takes time for GPU hardware, and the cost would affect the performance if the number of workgroups is large. This is especially expensive if the workload of each workgroup is light. Therefore, instead of launching a lot of workgroups, developers may reduce the number of workgroups and increase the workload of each workgroup. In extreme cases, a kernel can use just one workgroup per SP, with many iterations, to complete the same task many workgroups do. This so-called “persistent thread” can minimize the cost of workgroup launches in hardware and improve performance. A caveat of this approach is that context switch may be affected, as discussed in Section 3.3 The workload of a workgroup should not affect the other high-priority applications, such as the rendering of the user interface (UI).

6.2 Use image objects instead of buffer objects

OpenCL supports buffer and image objects (and pipe objects from OpenCL 2.0). The one-dimensional buffer objects are a natural choice for many developers due to their simplicity and flexibility, such as the support of pointers, byte-addressable access, etc. Image objects, which store one-, two- or three-dimensional data with a predefined image format, are preferred for Adreno GPUs over buffer objects due to various reasons:

- Adreno GPUs have a powerful texture engine and dedicated level 1 cache that can load data in image objects effectively.
- Using images allows hardware to handle out-of-boundaries read automatically.
- Adreno GPUs support numerous pairs of image formats and data type combinations.
- Adreno GPUs support bi-linear or tri-linear interpolation operations.

In many use cases, developers can expect a good performance boost when replacing buffer objects with image objects at the expense of a slightly more complex host code and a loss of flexibility.

A more detailed description is in Section 7.1.5.3

6.3 Vectorized load/store and coalesced load/store

Adreno GPUs support up to 128-bit read/write of global/local memory and image per load/store transaction. To maximize the memory load efficiency, each work item ideally should use the vectorized load/store functions, such as `vload4/vstore4` with four 32-bit data, and `read_image(f,i,ui,h)/write_image(f,i,ui,h)` with `CL_RGBA` and `float/int32/uint32/half` data type. This is helpful for memory-bound use cases.

Adreno GPUs support the hardware coalesced load/store. For example, suppose each work item loads 16-bit data whose addresses are consecutive in memory. Adreno GPUs can combine the request of a certain number of neighboring work items to minimize the number of load requests. However, compared with the 128-bit vectorized load/store, this coalescing is less effective.

A more detailed description is in Section 7.2.2

6.4 Constant memory

Adreno GPUs support fast on-chip constant memory and using it could drastically reduce kernel execution time. In most cases, the compiler can automatically use constant memory to store some variables, such as a constant array. However, in some cases, developers need to provide more information so that compiler can decide whether constant memory can be used. For example, for the following kernel,

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

the buffer `foo`, a global memory object, can be promoted to the fast on-chip constant memory if the compiler can determine that its size, as specified via the `max_constant_size` attribute, does not exceed the available constant memory.

It could lead to a further performance boost if the ALU operations with elements in `foo` are uniform, i.e., the work items in a subgroup or workgroup use the same component from `foo` for computing. This is because the content in constant memory can broadcast into ALUs in no time for fast ALU computing. All other memories (global, local, and private) must go through the lengthy load/store path to move the data into registers before using them for ALU computing.

A more detailed description is in Section 7.1.3

6.5 Local memory

Local memory is on-chip physical memory in Adreno GPUs. Using local memory does not necessarily lead to performance improvement. The following are a few things that developers should note:

- Try to do a 128-bit vectorized load/store.
- Local memory should store the most frequently used data.
  - If used once or very few times, local memory may hurt performance.
- Subgroup functions for reduction and shuffle operations are recommended. Subgroup functions allow work items to share and exchange data without using local memory.
  - Subgroup functions may not go through the lengthy load/store path.
- Extensive use of local memory may restrict the number of concurrent workgroup executions and therefore hurt latency hiding.

A more detailed description is in Section 7.1.2

7 Memory performance optimization

Memory optimization is the most critical and effective OpenCL performance technique. Many applications are memory bound rather than compute-bound. Mastering memory optimization is, therefore, essential for OpenCL optimization.

7.1 OpenCL memories in Adreno GPUs

OpenCL defines four types of memory (global, local, constant, and private), and understanding their differences is essential for performance optimization. Figure 7-1 illustrates a conceptual layout of these four types of memory.

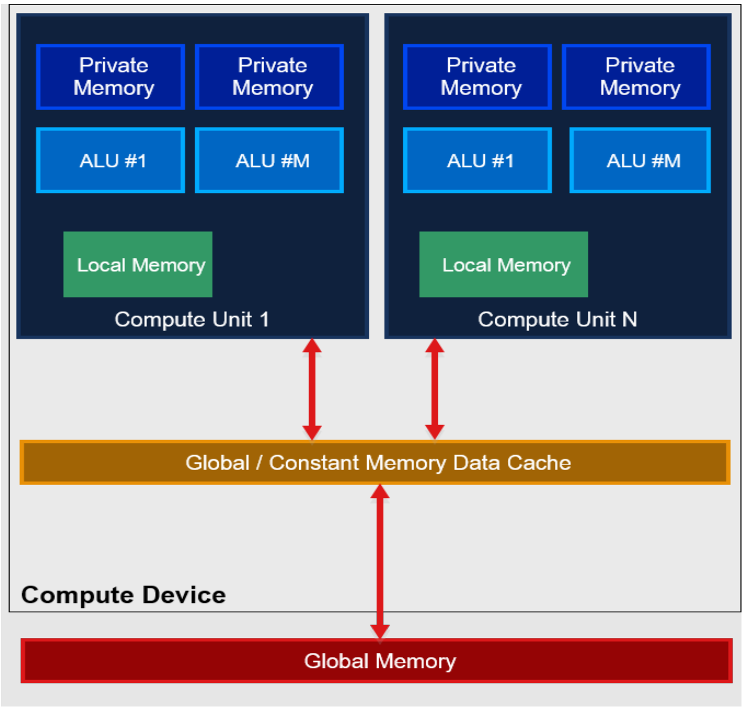


Figure 7-1 OpenCL conceptual memory hierarchy

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

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

Generally, a kernel should use local and constant memory for data that needs frequent access to take advantage of the short latency property. More details are 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 workgroup	Medium	On-chip, inside SP
Constant	Constant for all work items in a workgroup	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 compiler allocates the memory	In SP as register or local memory or in system RAM (compiler determined)
Global	Accessible by all work items in all workgroups	High	System RAM

7.1.1 Lifecycle of the memory content

One typical question is how to pass a memory object's content from one kernel to the next. For instance, how to share the content of a kernel’s local memory with the following kernel. The following are the concepts that developers should follow:

- Local memory is per workgroup, and the life cycle of its content ends once the workgroup execution is complete. Therefore, sharing one workgroup's local memory content or from one kernel to another is impossible.
- Constant memory content is consistent across all work items in workgroups. Once a kernel execution is complete, the content is likely overwritten by other tasks running on the GPU, such as graphics workload.
- Individual work items own private memory, which cannot be shared once the work item execution is complete.
- Global memory, however, is backed by the buffer and image objects that hosts create and accessible by both the host and GPU. Therefore, it is accessible through different kernels if the objects are not released.

Therefore, global memory objects are the right way to pass data from one kernel to another. For other memories, developers should never assume their content from one kernel on the GPU is accessible by the following kernels.

7.1.2 Local memory

Adreno GPUs support fast on-chip local memory, while the local memory size varies from series/tiers to series/tiers. Before 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, ... )
```

The following are the guidelines for using local memory.

- Use local memory to store data repeatedly or 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 video processing. Suppose each work item processes a small region of 8x8 pixels within a search window of 16x16 pixels, leading to data overlap between neighboring work items. In this case, local memory can be an excellent fit to store the pixels 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 OpenCL's relaxed memory consistency model.
  - A barrier often results in a synchronization latency that stalls ALUs, leading to lower utilization.
  - In some situations, caching data into local memory leads to synchronization latency that washes out the benefits of using local memory. Using global memory directly to avoid barriers may be a better option in such a case.
- Use vectorized local memory load/store.
  - A vectorized load of up to 128 bits (e.g., `vload4_float`) that is 32-bit aligned is recommended.
  - See more detail on vectorized data load/store 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 the entire local memory for the workgroup.
- Avoid using the function called `async_work_group_copy`. It is often tricky for the compiler to generate the optimal code to load local memory, and better for developers to write code that manually loads data into local memory.

7.1.3 Constant memory

Adreno GPUs support on-chip constant memory, which can provide superior performance among the four types of memory if appropriately used. Constant memory is usually used in the following cases:

- Scalar and vector variables defined with `constant` qualifiers.
- An array with a `constant` qualifier if defined in the program scope (e.g., the compiler can determine its size), fit into constant memory.
- Kernel arguments that are of scalar or vector data types. For example, `coeffs` in the following example will be stored in constant memory:

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

- Scalar and vector variables and arrays with `__constant` but do not fit into constant memory will be allocated in system memory.

The following is an essential recommendation for constant memory. If a kernel has these two characteristics:

- A small array as kernel argument, e.g., the coefficients of a 5x5 Gaussian filter.
- The array's elements are read uniformly across the subgroup or workgroup.

Its performance can be significantly improved if the array can be loaded into constant memory via an attribute called `max_constant_size(N)`. The attribute is to specify the maximum number of bytes required for this array. In the following example, 1024 bytes in the constant memory are allocated for the variable `foo`:

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

Specifying the `max_constant_size` attribute is essential. Without this attribute, the array is stored in the off-chip system memory because the compiler does not know the size of the buffer and cannot promote it to the on-chip constant memory. This feature only supports 16-bit and 32-bit arrays, i.e., 8-bit arrays are not supported. Also, if the buffer is too large to fit in constant memory, it is stored in the off-chip system memory instead.

Constant memory 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.

7.1.4 Private memory

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

- Scalar variables are stored in registers, which is faster than other memory.
    - If there are insufficient registers, private variables will be in system memory.
- Private arrays may be stored in:
    - Local memory, though it is not guaranteed.
    - Off-chip system memory if they exceed local memory capacity.

Storing private memory into off-chip system memory is highly undesirable due to two reasons:

- The latency of system memory is a lot higher than the GPR.
- Private memory access patterns are not cache-friendly, especially if the amount of private memory per work item is significant.

It is recommended to:

- Avoid defining any private array in kernels. Try to use vectors if possible.

- Replace a private array with a global or local array and design its layout so that the access of the array elements can coalesce across multiple neighboring work items. The cache performance could be better.
- Use vectorized private memory load/store, i.e., try to load/store up to 128-bit per transaction, using `vload4/vstore4` to load four 32-bit elements each time.

7.1.5 Global memory

OpenCL supports buffer and image objects using the off-chip system RAM. Compared to a buffer object, a simple one-dimensional data array stored in system RAM, an image object is an opaque memory object in which developers do not see how the underlying data is stored. When an image object is created, the software arranges the data in specific ways for the GPU to access efficiently. The optimal ways to use them are different and discussed in the following sections.

7.1.5.1 Buffer objects

Buffer objects store a one-dimensional collection of elements: scalar data types, vector data types, or user-defined structures. The buffer object's content is loaded or written by kernels via the L2 cache in Adreno GPUs. 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)
```

In this function, `cl_mem_flags` is a vital flag that developers must use with care, as it could significantly impact performance. OpenCL allows many different flags for this function, and for Adreno GPUs, here are a few key points:

- Some flags may incur an extra memory copy. Try to use the zero-copy flags described in Section 7.4.
- Some flags are for desktop/discrete GPUs with dedicated GPU memory.
- Use the most accurate flags:
    - The general idea is that the more restrictive flags are, the better chance that the OpenCL software can find the best configuration for the object.
    - For instance, the OpenCL software can apply a cache flush policy (write-through, write-back, etc.) that best fits the memory object with the most negligible overhead on cache flushes.
    - Section 7.4.2 details the cache policy and its implication for performance. Here are a few examples:
        - If the memory is read-only by the host, then use `CL_MEM_HOST_READ_ONLY`.
        - If the memory has no access by the host, then use `CL_MEM_HOST_NO_ACCESS`.
        - If the memory is for the host to write only, use `CL_MEM_HOST_WRITE_ONLY`.

7.1.5.2 Image objects

An image object stores a 1D, 2D, or 3D texture, frame buffer, or image data, and the data layout inside the image object is opaque. In practice, the content in the object does not have to be associated with actual image data. Any data can be stored as an 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 have a similar rule of thumb to the buffer object discussed in the previous section.

Many image formats and data types are supported in Adreno GPUs. And over generations, more image formats and data type pairs have been added. Developers may use the function `clGetSupportedImageFormats` to get a complete list of image formats/data types available on the device. To fully utilize the memory bandwidth, developers should use pairs whose length is 128-bit, e.g., `CL_RGBA/CL_FlOAT`, `CL_RGBA/CL_SIGNED_INT32`, etc.

Adreno GPUs also support formats not in OpenCL standards through the vendor extensions, such as the YUV and compressed formats. In addition to new formats, many new functions are also hardware accelerated for image objects, such as box filtering, SAD, and SSD. For more details, please refer to Chapter 9 .

7.1.5.3 Image object vs. buffer object

As mentioned in section 6.2 Adreno GPUs work better with image objects than buffer objects due to various advantages, including a powerful texture engine, dedicated L1 cache, and automatic handling of out-of-boundary access. Adreno GPUs support many image formats and data type combinations and can do automatic format conversions.

OpenCL supports two sampler filters, `CLK_FILTER_NEAREST` and `CLK_FILTER_LINEAR`. For `CLK_FILTER_LINEAR`, the proper combination of image types allows the GPU to do automatic bilinear/trilinear interpolation using its built-in texture engine. For example, assume an image is `CLK_NORMALIZED_COORDS_TRUE` and `CL_UNORM_INT16`, i.e., the image data is 2-byte unsigned short. To execute `read_imagef,` Adreno GPUs do the following:

- Reads pixels from the image object via the L1 cache.
- Perform the interpolation using all required pixels.
- Converts and normalizes it to the range of [0, 1].

This is convenient for bilinear/trilinear interpolation operations. Adreno GPUs also support bi-cubic interpolation through a vendor extension. For more details, please refer to Section 9.3.4 Sometimes, however, a buffer object may be a better choice:

- Buffer allows more flexible data access:

  - Image objects only allow access at the pixel size boundary, for example, 128-bit for RGBA and 32-bit/channel image object.
  - Adreno supports flexible access for buffer objects, where a pointer provides excellent flexibility on data access.
- The L1 cache becomes the bottleneck.
  - For example, there is severe L1 cache thrashing, which makes L1 cache access inefficient.
- A buffer object allows read and write inside kernels. A read-and-write image (an image object with `__read_write` qualifier) in kernels is supported by OpenCL 2.0. Due to synchronization requirements, the performance of read-and-write images may be worse than buffers on some old-generation Adreno GPUs.

Table 7-2 Buffer vs. image in Adreno GPUs

Features	Buffer	Image
L2 cache	Yes	Yes
L1 cache	Unavailable on most GPUs	Yes
Support read-and-write in kernel	Yes	No in OpenCL 1.x Yes, in OpenCL 2.x (synchronization required)
Use of pointer	Yes	No
Built-in hardware interpolation	No	Yes
Built-in boundary handling	No	Yes
Support image format/sampler	No	Yes

7.1.5.4 Use of both Image and buffer objects

Instead of using texture or buffer objects only, a better way is to have both L2 cache⇔SP and L2 cache⇔TPL1⇔SP paths fully utilized. As TPL1 has the L1 cache, it is a good practice to have L1 store the most frequently used but relatively small amount of data.

7.1.5.5 Global memory vs. local memory

One typical 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 local memory due to the following reasons:

- It may have a better L2 cache hit ratio and better performance
- Code is simpler than local memory and has a larger workgroup size

7.2 Optimal memory load/store

In previous sections, we discussed the general guidance on how to use a different type of memory. In this section, we will review a few key points 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 3.2.4, for local memory access. Coalesced access is also vital for global memory load/store.

Coalesced store works similarly to read, except that load is a 2-way process (request and respond), while the store is a 1-way process that mostly does not block kernel execution. And for most use cases, the data load is much larger than the data store. Therefore, coalesced load is generally more critical than store.

Adreno GPUs support coalesced access to global and local memory, but not to private memory.

7.2.2 Vectorized load/store

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

- Each work item should load data in a chunk of multiple bytes, e.g., 64/128-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 10.3.3.
- For optimal SP to L2 bandwidth performance, the memory address for load/store should be 32-bit aligned.
- There are two methods to do vectorized load/store:
  - Use built-in function (`vload/vstoren`).
  - Alternatively, pointer cast can be used to do vectorized load/store as follows:

```
char *p1; char4 vec;
vec = *(char4 *)(p1 + offset);
```

- Use vectorized load/store instructions that take up to four components. Vectorized data type load with more than four components would be divided into multiple load/store instructions, each taking no more than four components.
- Avoid loading too much data in one work item.
  - Loading too much data may result in higher register footprints, leading to a smaller workgroup size and hurting performance.
  - In the worst cases, it may cause register spilling, i.e., the compiler must use system RAM to store variables.

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

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

- Check data types in each stage of the application pipeline and ensure 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 vs. 32-bit data type

Using 16-bit data type instead of 32-bit data type is highly recommended on Adreno GPUs due to two reasons:

- The computing capability (measured in gflops) of 16-bit ALUs operations is twice of the 32-bit ones, thanks to Adreno's dedicated hardware acceleration logic for 16-bit ALU computing.
- Load/store of 16-bit data vs. 32-bit data can save half of the bandwidth.

In particular, the 16-bit floating-point, also called half floating (FP16), is highly desirable for some machine learning and image processing use cases. Note that the data range and precision of 16-bit half are more restricted than 32-bit floating data (FP32). For example, it can only accurately represent [0, 2048] on integer values. Developers must be aware of the precision loss problem.

Another way to use 16-bit is to load/store data as 16-bit, while the computing part can be 32-bit if the precision loss is unacceptable. This would save half of the memory traffic compared to using 32-bit data.

7.3 Atomic functions in OpenCL 1.x

OpenCL 1.x supports local and global atomic functions, including `atomic_add`, `atomic_inc`, `atomic_min`, `atomic_max`, etc. Note that the atomic functions discussed here differ from those in shared virtual memory (SVM) in Section 7.5 . Adreno GPUs support all of them in hardware. Here are some rules when using atomic functions:

- Avoid frequently doing atomic operations on a single global/local memory address by many work items.
  - Atomic operations are serialized and non-separable operations that may require lock and unlock on the memory address.
  - Therefore, atomic on a single address by many work items is not desirable.
- Try to make a reduction first, e.g., to use local atomic first and have a single update to global memory atomically.
  - In Adreno GPUs, each SP has its own local memory atomic engine. Doing local atomic first helps reduce the access contention if using global memory atomic if their addresses are the same.

7.4 Zero copy

Adreno OpenCL provides a few mechanisms to avoid costly memory copies that could occur on the host side. Depending on how the memory object is created, a few options exist to prevent excessive copies. This section describes a few basic approaches to achieving zero-copy, and Section 7.5 presents a more advanced technique with shared virtual memory (SVM).

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. For this simple case, 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);
```

  - The host updates the buffer using the pointer `hostPtr`.
    - For example, the host can fill camera data or read data from the disk into the buffer.
  - Unmap the object:

```
          status = clEnqueueUnmapMemObject(
                              commandQueue,
                              Buffer,
                              (void *) hostPtr,
                              0, NULL, NULL);
```

  - OpenCL kernels can use the object.

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

7.4.2 Avoid memory copy for objects allocated not by OpenCL

7.4.2.1 ION/dmabuf memory extensions

Suppose a memory object is initially created outside the scope of the OpenCL API and is allocated using ION/DMA-BUF. In that case, developers can use the `cl_qcom_ion_host_ptr` or `cl_qcom_dmabuf_host_ptr` extensions to create buffer/image objects, which map into the GPU-accessible memory without incurring extra copy.

Samples that illustrate the use of `cl_qcom_ion_host_ptr` and `cl_qcom_dmabuf_host_ptr` can be provided upon request.

7.4.2.2 QTI Android native buffer (ANB) extension

ANB (allocated by `gralloc`) must be shared in many camera and video processing use cases. Sharing is possible because the buffers are based on ION. However, developers need to extract internal handles from these buffers to use the ION path, 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 techniques for ANBs.

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

7.4.2.3 Android Hardware Buffer (AHB) extension

Like the ANB extension described above, the `cl_qcom_android_ahardwarebuffer_host_ptr` extension offers an easy way to share AHBs with OpenCL without extracting internal ION handles and thus implement zero-copy AHB applications.

A sample that illustrates the use of the `cl_qcom_android_ahardwarebuffer_host_ptr` extension can be provided on request.

7.4.2.4 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`) 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 Shared virtual memory (SVM)

As an essential and advanced feature introduced to the OpenCL 2.0 standard, SVM allows the host and the device to share and access the same memory space and avoid excessive data copy, e.g., accessing the host pointer on the OpenCL device is now possible.

SVM has several types that a GPU can choose to support. Starting from Adreno A5x GPUs, the coarse-grain SVM and the more advanced fine-grain buffer SVM with atomics are all supported.

- For coarse-grain SVM, memory consistency is only guaranteed at the synchronization points using map/unmap functions, i.e., `clEnqueueSVMMap` and `clEnqueueSVMUnMap`.
  - Coarse-grain SVM is, therefore, similar to the zero-copy technique described in Section 7.4.1 , as they all require map and unmap operations.
  - Still, coarse-grain SVM allows applications to use and share pointer-based data structures across the host and the device.
- Fine-grain buffer SVM eliminates the requirement of map/unmap synchronization in coarse-grain SVM.
  - Fine-grain buffer SVM is a “map-free” SVM, i.e., the host and the device can simultaneously modify the same memory region.
    - Still, it requires some level of synchronization.
  - Dependent on the data access pattern between the host and device, different types of synchronization may be needed.
    - If there is no read-write dependency on the same piece of data between the host and the device, e.g., the host and the device are working on a separate portion of an SVM memory object, no atomic/fence is needed.
      - In this case, memory consistency is ensured at OpenCL synchronization points, e.g., after a `clFinish` call, all the data will be up-to-date.
    - If there is dependency or requirement on memory access order like one data is modified by the host and the device needs to use the new data, then atomic or fence is required.
      - When creating it, the SVM buffer must have the flag `CL_MEM_SVM_ATOMICS`.
      - Inside the kernel, `memory_scope_all_svm_devices` must be used.
      - A set of C11-style atomic functions must be used with appropriate memory scope, order, and atomic flags.

Developers need to evaluate the pros and cons of SVM carefully. As an advanced feature, an exemplary and performant implementation of SVM for GPUs typically requires sophisticated hardware design. There is a hidden cost for implementing all these advanced data sharing and synchronization, which developers may not be aware of. The bar for observing the benefits of SVM in complex real-life use cases is relatively high. Developers should use SVM with great care, especially if there is a lot of data dependency between the host and the device. The synchronization cost in this type of use case may undermine the gain of having a shared virtual memory space.

7.6 Improve the GPU’s L1/L2 cache usage

To have good cache usage, developers should follow the rules as follows:

- Understand the impact of data load/store:
  - Many kernels load a lot more data from the global memory than the data to be stored. Therefore, it is essential to improve the data locality and reduce the demand for cache lines by doing a coalesced load, vectorized load, using images, etc.
  - However, the data store may also have a significant performance impact.
    - For data store, a cache line must first be loaded from the system memory, modified, and then written out.
    - If the locality of the data store is poor, e.g., data are written into too many cache lines, the memory system must load many cache lines for the update.
    - Coalesced write is essential for performance as it can improve locality and reduce the demand for cache lines from the memory system.
- Check and avoid cache thrashing for better cache usage efficiency.
  - Cache thrashing means a cache line is evicted before it can be fully used and then must be fetched again. This can lead to severe performance penalty.
  - Snapdragon Profiler can provide information about cache access, such as the number of bytes for load/store and the cache hit/miss ratio.
  - If the number of bytes to load into the L2 cache is much higher than expected by the kernel, there may be 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:
    - Tune the workgroup size and shape.
    - Change access pattern, e.g., change the dimensionality of the kernel.
    - If there is cache thrashing when using loops, adding atomics or barriers in the loop may reduce the chance of cache thrashing.

Profiling tools rely on hardware performance counters to produce metrics about cache usage. Because the performance counters are designed to relay information about the hardware, derived metrics such as the L1/L2 cache hit ratio may give non-intuitive results. For example, it is possible to see large negative values for % L2 hit rates, indicating that more data is being loaded into the cache than is requested. In cases like these, the programmer should focus on the relative change in the profiling metric value between optimizations rather than the absolute value of the metric.

7.7 CPU cache operations

Modern SOCs have multi-level caches, and the Snapdragon SOC is no exception. It is helpful for developers to know some basics of GPU/CPU cache operations in the SOCs.

The OpenCL driver must flush or invalidate the CPU cache at appropriate times for cacheable memory objects, which 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 software has a sophisticated CPU cache management policy that 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.

Here are a few rules to minimize the cost of CPU cache operations:

- An application should be structured so that processing is not frequently moved back and forth between CPU and GPU.
- In addition, the application should allocate memory objects so that the data subject to back-and-forth CPU and GPU access is in a different memory object from the data with only one access transition.
- The memory objects should be created using the appropriate CPU cache policy for their intended usage.
    - The driver will select the CPU cache policy when allocating memory for buffer or image objects. 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 ION and ANB mechanisms (see Section 7.4.2 , the application has more direct control over CPU cache policy.
    - When importing these objects into OpenCL, the application must correctly set the CPU cache policy flag.

7.8 Best practices to reduce power/energy consumption

Power and energy are significant factors 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 requirements. 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, e.g., storing data in constant or local memory, using shorter data types, reducing data precision, 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, the event-driven pipeline reduces the overhead of host and device communication. Avoid creating too many OpenCL objects and creating or releasing OpenCL objects between kernel executions.

8 Kernel performance optimization

This chapter presents more details on kernel optimization, which could overlap with the top optimization tips in Chapter 6 and memory optimization in Chapter 7 .

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 used. It is hard to answer as there are many factors involved. Here are a few high-level guidelines:

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

Developers can try the following:

- Splitting a big kernel into small kernels could yield better data parallelization.
- Fuse multiple kernels into one kernel (kernel fusion) if data traffic can be reduced with good parallelization, e.g., workgroup size is reasonably large.

8.2 Compiler options

Many compiler build options for the APIs `clCompileProgram` and `clBuildProgram` are available for performance optimization. With these options, developers can enable some functionality for their purpose. For example, using `-cl-fast-relaxed-math` compiles the kernel using fast math rather than the OpenCL conformant math with much higher precision per OpenCL specification:

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

Refer to Section 5.6.4 of *The OpenCL Specification* found in References for more detail.

In addition, Adreno GPUs can support some Adreno-specific options to enable certain features, as discussed in Chapter 9 .

8.3 Conformant vs. fast vs. native math functions

The OpenCL standard defines many math functions in the OpenCL C language. By default, all the math functions must meet the IEEE 754 single precision floating-point math requirements, as the OpenCL specification requires. Adreno GPUs have a built-in hardware module, the elementary function unit (EFU), to accelerate some primitive math functions. Many math functions that EFU does not directly support have been either optimized by combining EFU and ALU operations or emulated using complex algorithms by the compiler. The following table shows a list of OpenCL-GPU math functions categorized based on relative performance. It is a good practice to use the high-performance ones, e.g., 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 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, native_divide`;
  - Here is an example of using 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	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 costs 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 a lack of knowledge. In these cases, developers 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 unroll loops manually.

8.5 Avoid branch divergence

Generally, GPUs are not efficient when work items in the same wave follow different execution paths. Some work items may have to be masked for divergent branches, 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.

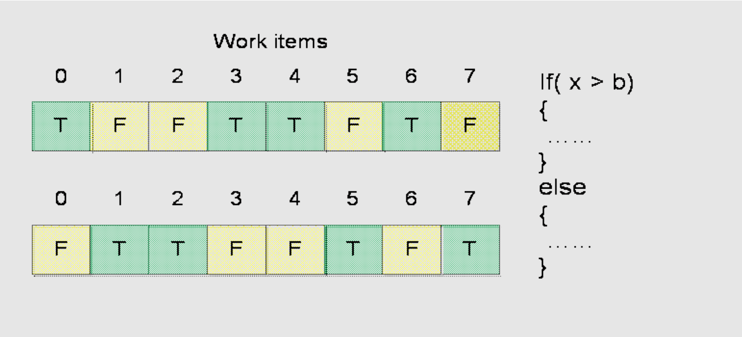


Figure 8-1 Pictorial representation of divergence across two waves

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. Developers may convert some simple divergent or conditional check operations to fast ALU operations. One example in Section 10.3.6 shows how a ternary operation, executed by the expensive control flow logic, is converted to a fast 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 an image's boundaries. To better handle boundaries, the following options should be considered:

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

8.7 Avoid the use of size_t

In many cases, a 64-bit memory address poses a complication for OpenCL kernel compilation, and developers must be careful. Developers should avoid defining variables as type of `size_t` inside kernels. For the 64-bit OS, a variable defined as `size_t` inside kernels may have to be handled as a 64-bit long. Adreno GPUs must use two 32-bit registers to emulate 64-bit. Therefore, having `size_t` 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 with the best knowledge of data types for better register usage and code quality.

8.8 Generic vs. named memory address space

A feature called generic memory address space has been introduced since OpenCL 2.0. Before OpenCL 2.0, a pointer must specify its memory address space, such as local, private, or global. This feature allows pointers not to set their address space in kernels. GPU determines the real address space during the kernel execution. This feature enables developers to reuse and reduce code base, particularly useful for tasks like library development.

The use of generic memory address space may incur a performance penalty due to the hardware's cost associated with identifying memory space. Here are a few tips with the memory address space:

- Developers should explicitly specify the memory address space if it is known upfront. This would reduce compiler ambiguity and avoid GPU hardware costs to identify the real memory space.
- Try to avoid using divergent memory address space. For a uniform case, the compiler may be able to extract the memory space and avoid having the hardware identify its memory space.
- Prepare different versions with different memory address spaces.

8.9 Subgroup

The new subgroup functions in OpenCL 2.0 provide a finer granularity control over work items than workgroups. A workgroup has one or multiple subgroups, and the subgroup in Adreno GPUs is precisely mapped to the wave concept. Compared to 1D/2D/3D workgroups, subgroups have only one dimension. Similar to workgroups, a set of functions allows work items to query their local ID and other parameters in a subgroup.

The power of subgroups is that OpenCL introduces a rich set of functions that allows work items in subgroups to share data and do various operations across the work items in subgroups. Without the feature, data sharing across work items may have to rely on local or global memory, which is typically expensive.

It is up to the hardware vendors to choose how to implement the subgroup functions. It could be either accelerated via hardware or through software emulation. In Adreno GPUs, many of the subgroup functions are hardware accelerated.

Besides the subgroup functions in core OpenCL, there are also KHR extensions on subgroups in OpenCL 3.0. It is essential to check the availability of the extensions before using them.

There are broadly two types of subgroup functions: reduction and shuffle.

- Reduction: Adreno has hardware support for the reduction functions, much faster than doing reduction through local memory.
- Shuffle: shuffle allows the data to be passed from one work item to another. It typically supports shuffle-up, shuffle-down, and generic shuffle.

Besides the support of standard subgroup functions, Adreno GPUs support subgroup reduction and shuffle via vendor extensions. See Section 0for more details.

8.10 Use of union

Although a standard feature in the OpenCL kernel language, union is inefficient on Adreno GPUs. The compiler needs to allocate extra registers to handle the different-sized members; therefore, the performance is typically worse than the regular kernel without a union. Developers should avoid using union on Adreno GPUs.

8.11 Use of struct

A struct can make code easier to understand and organize as an excellent way to group a set of related variables into one place. Despite the pros, using struct on Adreno GPUs may incur some inefficiency and is not always recommended. Here are a few tips with struct:

- Try to avoid pointers inside a struct.
- Assign individual members explicitly rather than assigning the whole struct variables to another.
- Choose *struct of array* or *array of struct*.
  - One key consideration is whether the choice can ease the bottleneck of the kernel.
  - For example, if the array can be arranged so that the data load from memory has better coalescing, then struct of array is a better choice.
  - Array of struct might be a better choice if the members in struct lead to good cache locality.

The decision will depend on each use case's characteristics.

8.12 Miscellaneous

Many other optimization tips that look minor could improve kernel performance. Here are a few things developers may give a try:

- Pre-calculate values that do not change within the kernels.
  - Calculating a value that can be precalculated outside kernels is wasteful.
  - Pre-calculated values can be passed to the 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 32-bit integer multiplication requires more than one instruction.
  - If integer numbers are within the 24-bit range, using `mul24` is faster than direct 32-bit multiplication.
- Reduce EFU functions.
  - For example, the code is as follows:

    ```
    r = a / select(c, d, b<T)
    ```

    Where `a, b`, and `T` are float variables, `c` and `d` are constant and can be rewritten as follows:

    ```
    r = a * select(1/c, 1/d, b<T)
    ```

    This 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 very expensive in 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 Adreno GPUs.
- 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 a global optimization perspective.
  - `mem_fence` may prevent the compiler from shuffling/mixing the code before and after.
  - `mem_fence` allows developers to manipulate some of the code for profiling and debugging.
- Use 16-bit ALU computation instead of 8-bit. 8-bit may have to be converted to 16-bit or 32-bit ALU operations as Adreno GPUs do not have general 8-bit ALU support.
- Use bit shift operations instead of multiplication, if possible.

9 OpenCL extensions in Adreno GPUs

An OpenCL platform or device may support or expose features that are not adopted into the core standard through the extension mechanism, to further improve their usability, expose hardware capability, and create a differentiated experience. The OpenCL standards support several types of extensions based on their adoption status and the vendors adopting them:

- KHR extensions are approved by the OpenCL standard working group but are optional for vendors to support. The strings representing these extensions start with `cl_khr`.
  - These extensions typically have more than one vendor supported and have conformance tests vendors that must pass if they claim to support.
  - The specification of the KHR extensions is available at the Khronos' official OpenCL website.
  - Some of the KHR extensions could become core features in a newer version of the OpenCL standard.
- EXT extensions are approved by the OpenCL standard working group but are optional for vendors to support. The strings representing these extensions start with `cl_ext.`
  - These are more for experimental purposes and do not usually have conformance tests.
  - These EXT extensions could become KHR or even core features.
- Vendor extensions have specific syntax and typically only run on a particular vendor's platform. Their names typically have the vendor's company name acronym inside.
  - For example, OpenCL vendor extensions in Adreno GPUs have "`qcom`" inside, e.g., `cl_qcom_accelerated_image_ops`.

In addition, there could be also private or internal extensions, which a vendor may not make public but could be available to customers. An extension could be targeting an OpenCL platform or an OpenCL device:

- The extensions available on a given platform can be obtained via the API function `clGetPlatformInfo` with the parameter of `cl_platform_info` set to `CL_PLATFORM_EXTENSIONS.`
- The extensions available for an OpenCL device can be obtained via the API function `clGetDeviceInfo` with the parameter of device info set to `CL_DEVICE_EXTENSIONS`.

Both functions will return a list of name strings for the available extensions. There are caveats when using the extensions:

- An extension could have different versions.

- Vendor extensions can be deprecated or de-featured by the vendor.
- An extension can become a core feature of the OpenCL standard. If an extension is unavailable, check if the core specification already adopts it.
- Developers must query the list of extensions available on the platform before using an extension for better portability.

The following table shows the KHR and vendor extensions supported on Snapdragon 888 devices. The detailed documentation for the extensions are available in the Adreno OpenCL SDK, which can be downloaded from the QTI developer network website (https://developer.qualcomm.com).

In the following few sections, the KHR extensions will be skipped and a high-level overview of the vendor extensions available in Adreno GPUs is given.

Table 9-1 List of extensions supported on Snapdragon 888 devices

Type of extensions	Extensions	Reference
KHR extensions	cl_khr_3d_image_writes cl_img_egl_image cl_khr_byte_addressable_store cl_khr_depth_images cl_khr_egl_event cl_khr_egl_image cl_khr_fp16 cl_khr_gl_sharing cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomicscl_khr_local_int32_extended_atomics cl_khr_image2d_from_buffer cl_khr_mipmap_image cl_khr_srgb_image_writes cl_khr_subgroups cl_khr_integer_dot_product cl_ext_image_from_buffer cl_ext_image_requirements_info	The OpenCL™ extension specification: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_Ext.html

Type of extensions	Extensions	Reference

Vendor extensions	`cl_qcom_create_buffer_from_image`
	`cl_qcom_ext_host_ptr`
	`cl_qcom_ion_host_ptr`
	`cl_qcom_perf_hint`
	`cl_qcom_other_image`
	`cl_qcom_subgroup_shuffle`
	`cl_qcom_vector_image_ops`
	`cl_qcom_extract_image_plane`
	`cl_qcom_android_native_buffer_host_ptr`
	`cl_qcom_protected_context`
	`cl_qcom_priority_hint`
	`cl_qcom_compressed_image`
	`cl_qcom_ext_host_ptr_iocoherent`
	`cl_qcom_accelerated_image_ops`
	`cl_qcom_reqd_sub_group_size`
	`cl_qcom_recordable_queues`
	`cl_qcom_android_ahardwarebuffer_host_ptr`
	`cl_qcom_bitreverse`
	`cl_qcom_create_buffer_from_image`
	`cl_qcom_dot_product8`
	`cl_qcom_ml_ops`
	`cl_qcom_extended_query_image_info`
	`cl_qcom_filter_bicubic`
	`cl_qcom_onchip_global_memory`
		Adreno OpenCL SDK documentation (check Qualcomm Developer's network)

9.1 OS-dependent vendor extensions

This section presents the extensions that extend the functionality of OpenCL runtime with no new functions required in the OpenCL kernel language. Some extensions may depend on the Android OS and its low-level design, which could change. It is essential to frequently check the OpenCL SDK of the Qualcomm developer network to keep up-to-date with the extension details The extensions covered in other parts of this document will be skipped. For instance, the details of multiple zero-copy extensions are in Section 7.4 .

9.1.1 Performance hint (cl_qcom_perf_hint)

This extension allows applications to request the performance level desired for device(s) in an OpenCL context. A higher performance implies higher frequencies on the device. Three levels of performance hint, including high, normal, and low, are supported with flags, `CL_PERF_HINT_HIGH_QCOM`, `CL_PERF_HINT_NORMAL_QCOM`, and `CL_PERF_HINT_LOW_QCOM`, respectively.

- `CL_PERF_HINT_HIGH_QCOM` is the default setting for devices in an OpenCL context (requests the highest performance level from the device).
- `CL_PERF_HINT_NORMAL_QCOM` is a balanced performance setting that is set dynamically by the GPU frequency and power management.

- CL_PERF_HINT_LOW_QCOM requests a performance setting that prioritizes lower power consumption.

There are two ways to use this extension:

- Specify the perf hint flag at the context property parameter when creating the context using the clCreateContext function. Here is an example:

```
cl_context_properties properties[] =
{CL_CONTEXT_PERF_HINT_QCOM,

CL_PERF_HINT_LOW_QCOM, 0};

clCreateContext(properties, 1, &device_id, NULL, NULL, NULL);
```

- Use clSetPerfHintQCOM, a separate API function, to set the perf hint property for an existing context. This function can be used to set or update the CL_CONTEXT_PERF_HINT_QCOM property irrespective of whether it was specified at the context time as one of the context properties. Here is an example:

```
clSetPerfHintQCOM(context, CL_PERF_HINT_NORMAL_QCOM);    .
```

9.1.2 Priority hint for context creation (cl_qcom_priority_hint)

This extension allows applications to specify the desired priority for enqueued kernels to be submitted to the device(s) in an OpenCL context. Like the performance hint flags, it defines three levels of priority:

- High priority, CL_PRIORITY_HINT_HIGH_QCOM, which implies that enqueued kernels may be submitted to the device for processing before other enqueues on other contexts that have lower priority.
- Normal priority, CL_PRIORITY_HINT_NORMAL_QCOM, a default behavior. To select the priority that would otherwise be used for the context if this extension is not used.
- Low priority, CL_PRIORITY_HINT_LOW_QCOM, contrary to the high priority, the kernels are submitted to the device after the other kernels on other contexts that have higher priority.

The hint should be provided at context creation using clCreateContext as a context property.

9.1.3 Recordable command queue (cl_qcom_recordable_queues)

The kernel enqueue function call, clEnqueueNDRangeKernel, is the key and demanding function that dispatch kernels to the GPU hardware, as it requires the application to configure and validate many kernel arguments, such as global work sizes, workgroup size, event dependencies, etc. For use cases where kernels have to be enqueued for repetitive processing, such as video processing applications where each frame repeatedly executes the same kernels in sequence, developers can minimize the overhead of setting the parameters continually.

- Instead of repeating these operations, this extension introduces a new set of API functions for recording sequences of kernel enqueues. A sequence only needs to be recorded once but can be dispatched multiple times. The arguments to any kernel in a recorded sequence, referred to as a recording, may be modified without re-

recording the entire command sequence. Therefore, the extension can save CPU power and improve dispatch latency. Here is how the extension can be used:

- A command queue is needed for recording, which must be created with the recordable property `CL_QUEUE_RECORDABLE_QCOM` with `clCreateCommandQueue`.
- A recording object must be created using a function called `clNewRecordingQCOM`.
- To start a recording use the standard enqueue function, `clEnqueueNDRangeKernel`, with this recording object created with `clNewRecordingQCOM`.
- Currently only `clEnqueueNDRangeKernel` can be recorded.
- `clEndRecordingQCOM` is used to finish the recording.
- `clEnqueueRecordingQCOM` is used to enqueue all kernels in the recordable queue for the GPU to execute, which requires the recording object, and a “live” command queue.
  - This function can be used to update parameters of the kernels in the recordable queue.
  - The live command queue for this function is different from the one for recording and must be an in-order command queue.
  - A mechanism is defined to specify the kernels to update, and the list of parameters to change, and the new value for each parameter to be changed.
  - All kernel parameters could be changed, including the kernel arguments, global sizes, etc.
  - The application can choose not to update the recording.
- This extension does not work with the kernel enqueue kernel (KEK) feature or `printf`.
- There has been an ongoing effort in the Khronos’ OpenCL working group to standardize a KHR extension that supports similar features while being more generic.
  - Going forward this extension will be supported in addition to the Khronos extension, once the Khronos extension is finalized.

9.1.4 cl_qcom_protected_context

This extension allows applications to create so called protected OpenCL contexts. An OpenCL command-queue that is created on a protected context is implicitly considered protected as well. Protected OpenCL contexts enable use of the Content Protection feature available on specific Qualcomm GPUs. The main purpose of this feature is to separate memory into protected and unprotected zones and prevent copying of data from the protected to the unprotected. To use this feature, a context with the `CL_CONTEXT_PROTECTED_QCOM` property must be created:

```
   cl_context_properties properties[] = {CL_CONTEXT_PROTECTED_QCOM, 1, 0};

   protected_context = clCreateContext(properties, 1, &device_id, NULL,
NULL, &err);
```

Once a protected command queue using the context is created and then used throughout the application:

```
   protected_queue = clCreateCommandQueue(protected_context,     device_id,
0, &err);
```

There are essentially two ways to create a protected memory object on Android:

- A protected graphics buffer can be allocated using the `GRALLOC_USAGE_PROTECTED` usage flag and can be accessed in OpenCL by using the `cl_qcom_android_native_buffer_host_pointer` extension with `clCreateBuffer` or `clCreateImage2D`.
- A protected ION allocation can be created from the protected heap using the `ION_SECURE` flag and can be accessed in OpenCL by using the `cl_qcom_ion_host_pointer` extension with `clCreateBuffer` or `clCreateImage2D`.

In both cases, the buffer is considered a protected memory object by OpenCL. An OpenCL application that enqueues kernels with one or more protected memory objects as arguments can only do so on a protected command-queue.

9.1.5 cl_qcom_create_buffer_from_image

In OpenCL, an image object is of an opaque data structure, which is managed by functions defined in the OpenCL API. The underlying details of how data in image objects are stored are unavailable to developers. Unlike buffer objects, which can be directly accessed using pointers, the built-in image read and write functions like `image_readf/image_writef` must be used inside kernels to access the image data.

In some cases, developers may want to access image data as raw pointers in the OpenCL C language instead of using the built-in image read/write functions. This extension can benefit the use cases as follows:

- Reading from or writing to EGL external images exposed indirectly to OpenCL through OpenGL and OpenCL interop extensions.
- An experienced developer may want to read/write multiple pixels with a single memory load/store operation.
- The data in the image object may have to be simultaneously fed into a kernel that only accepts it as a buffer object, and another kernel that only accepts it as an image object.

With the extension, to create a new raw buffer object from an existing image object, a function as follows is presented:

```
        cl_mem clCreateBufferFromImageQCOM(cl_mem image, cl_mem_flags
flags, cl_int *errcode_ret)
```

Where `image` is a valid image with some limits. There are requirements on image type, layout, and the concurrent read/write accesses:

- It supports all images except the following:
  - □ Image type `CL_MEM_OBJECT_IMAGE1D_BUFFER`, or
  - □ Images created using `CL_MEM_USE_HOST_PTR`.
- Data layout:

    - The buffer that is returned references the data store allocated for `image` and points to the origin pixel in this data store.
    - The data layout is equivalent to what is produced by `clEnqueueMapImage` when `origin` is `(0, 0, 0)` and `region` is `(width, height, depth)`.
    - The `image` from which the buffer is created is called the `parent_image` of the buffer.
- To access the pixel data in the returned buffer correctly, the client must query the parent image’s two parameters using `clGetDeviceImageInfoQCOM`:
    - Row pitch, using `CL_BUFFER_FROM_IMAGE_ROW_PITCH_QCOM`.
    - Slice pitch, using `CL_BUFFER_FROM_IMAGE_SLICE_PITCH_QCOM`.
- Concurrency of reading and writing:
    - Undefined:
        - Concurrent reading from and writing to both a buffer object and its `parent_image` is undefined.
        - Concurrent reading from and writing to buffer objects created with the same `parent_image` is undefined.
    - Supported:
        - Only concurrent reading from both a buffer object and its `parent_image` object.
        - Concurrent reading from multiple buffer objects created from the same image is defined.

Refer to the extension document in the Adreno OpenCL SDK for more details and examples on how to use it.

9.1.6 cl_qcom_onchip_global_memory

This extension provides functionality to create OpenCL buffers and images that reside in faster-access on-chip memory (hereafter referred to as `onchip_global_memory`) on Adreno GPUs. Once created, these objects can be used in OpenCL kernels in the same way as regular global memory objects are used.

If `cl_qcom_other_image` is present, this extension further allows the creation of planar images from an OpenCL buffer created in `onchip_global_memory`.

By default, the contents of `onchip_global_memory` are not preserved after kernel exit. However, an application can use `onchip_global_memory` to pass data between two or more kernels by leveraging the `cl_qcom_recordable_queues` extension. Contents of `onchip_global_memory` will be valid for the entire duration of the enqueue of a recording. Applications can therefore link the output of one kernel to the input of the next kernel using `onchip_global_memory` as long as these kernels are part of the same recording. `onchip_global_memory` is not preserved across different enqueues of the same recording. The first kernel in the recording must treat the `onchip_global_memory` as uninitialized.

By using buffers and images allocated in `onchip_global_memory` in lieu of global allocations, applications could achieve power and performance improvements. An example would be to use onchip_global_memory for intermediate buffers in a pipeline of kernels.

Refer to the Adreno OpenCL SDK for the extension document and an example demonstrating usage of onchip_global_memory.

9.1.7 cl_qcom_extended_query_image_info

The two vendor extensions supported in Adreno GPUs, cl_qcom_other_image and cl_qcom_compressed_image (both covered in Section 9.3) allows developers to create planar and compressed images that can be used in kernels.

This extension, cl_qcom_extended_query_image_info, complements the above extensions by enabling applications to query image attributes such as image size, image element size, row pitch, slice pitch, and alignment based on the image format and image descriptor. It is not necessary to create an image to query these attributes.

This extension accepts both conventional RGBA images and non-conventional images such as NV12, TP10, MIPI packed, Bayer pattern, tiled, and compressed images.

Refer to the Adreno OpenCL SDK for the extension document and an example demonstrating usage of cl_qcom_extended_query_image_info.

9.2 Subgroup

OpenCL 2.0 introduced a family of subgroup functions through the KHR core extension called cl_khr_subgroups to facilitate data sharing among work items within subgroups and therefore provide a fine granularity for the collaborative work items to work together. Without it, data sharing across work items must rely on local or global memory, and also the expensive workgroup barrier synchronization for data consistency.

Many of the subgroup functions in the extension have been adopted as core feature in OpenCL 3.0, such as sub_group_broadcast, sub_group_reduce_<op>, sub_group_scan_exclusive_<op>, sub_group_scan_inclusive_<op>, where <op> can be add, min, or max.

Adreno GPUs have several subgroup-related extensions to provide extra functionality, as described in the following sections.

9.2.1 Subgroup size (wave size) selection

Adreno GPUs generally support two different subgroup sizes, half-wave size and full-wave size. A kernel can run at either mode, likely resulting in different performances. The compiler picks the optimal subgroup size based on heuristics that predict the performance of the kernel. In some cases the application may be able to get better results by overriding the compiler selection using this extension. This extension provides a kernel attribute that enables applications to specify a preferred subgroup size.

To use this extension, #pragma cl_qcom_reqd_sub_group_size must be enabled in the program and the wave size attribute as follows.

```
#pragma OPENCL EXTENSION cl_qcom_reqd_sub_group_size: enable
 __attribute__((qcom_reqd_sub_group_size("half")))
```

```
kernel void half_sub_group_kernel(...)
{     ...     }
  __attribute__((qcom_reqd_sub_group_size("full")))
  kernel void full_sub_group_kernel(...)
 {     ...     }
```

Here are a few key notes:

- `sub_group_size` varies across different tiers of Adreno GPUs, which could be as large as 128, and as small as 16.
- Developers should query the value of `sub_group_size` via the OpenCL device inquiry API before using it.
- Developers should write code in a way that can adapt to different wave sizes to maximize its portability.

Refer to the Adreno OpenCL SDK for the extension documentation and samples.

9.2.2 Subgroup shuffle

Adreno GPUs support several subgroup shuffle functions, such as `cl_qcom_subgroup_shuffle`, as shown in Table 9-2 . The objective of the shuffle functions is to transfer the data in `source_value` from the current work item to the destination work item. The key is to determine `sub_group_local_id`, the destination work item in the subgroup. This variable is dependent on a few parameters as follows:

- Whether the operation is up or down:
  - □ Up: the destination `sub_group_local_id = source sub_group_local_id + offset.`
  - □ Down: the destination `sub_group_local_id = source sub_group_local_id – offset.`
- The behavior when the calculated `sub_group_local_id` is out of range:
  - □ Drop off: no transfer occurs.
  - □ Rotate: rotate back. The new `sub_group_local_id` is like a module operation to the width of the subgroup operation.
- Width mode:
  - □ `CLK_SUB_GROUP_SHUFFLE_WIDTH_WAVE_SIZE_QCOM`: applies to the whole subgroup.
  - □ `CLK_SUB_GROUP_SHUFFLE_WIDTH_W4_QCOM`: perform a shuffle operation within a group of four work items in a subgroup.
  - □ `CLK_SUB_GROUP_SHUFFLE_WIDTH_W8_QCOM`: perform a shuffle operation within a group of eight work items in a subgroup.

To use the shuffle functions, add the pragma to the kernel:

```
#pragma OPENCL EXTENSION cl_qcom_subgroup_shuffle: enable
```

The data types supported in this extension include `uchar, char, ushort, short, uint, int`, `ulong`, `long`, `float`, and `half` (if `cl_khr_fp16` is supported). Vectors are not supported.

Table 9-2 Subgroup shuffle operations

Functions	General functionality
`qcom_sub_group_shuffle_up(<gentype> source_value, uint offset, qcom_sub_group_shuffle_width_modes_t width, <gentype> default_value)`	Transfers data from a work item in the subgroup to another work item in the same subgroup having a `sub_group_local_id` equal to `(source sub_group_local_id + offset)`. If the `sub_group_local_id` of the work item that is about to receive the data is more than the largest `sub_group_local_id` of the work item participating in the shuffle group, the data transfer is dropped for that work item.
`qcom_sub_group_shuffle_down(<gentype> source_value, uint offset, qcom_sub_group_shuffle_width_modes_t width, <gentype> default_value)`	Transfers data from a work item in the subgroup to another work item in the same subgroup having a `sub_group_local_id` equal to `(source sub_group_local_id - offset)`. If the `sub_group_local_id` of the work item that is about to receive the data is less than the smallest `sub_group_local_id` of the work item participating in the shuffle group, the data transfer is dropped for that work item.
`qcom_sub_group_shuffle_rotate_up(<gentype> source_value, uint offset, qcom_sub_group_shuffle_width_modes_t width, <gentype> default_value)`	Transfers data from a work item in the subgroup to another work item in the same subgroup having a `sub_group_local_id` equal to (`source sub_group_local_id + offset)`. If `sub_group_local_id` of the work item that is about to receive the data is more than the largest `sub_group_local_id` of the work item participating in the shuffle group, `sub_group_local_id` is wrapped back into the subgroup of work items participating in the shuffle operation.
`qcom_sub_group_shuffle_rotate_down(<gentype> source_value, uint offset, qcom_sub_group_shuffle_width_modes_t width, <gentype> default_value)`	Transfers data from a work item in the subgroup to another work item in the same subgroup having a `sub_group_local_id` equal to `(source sub_group_local_id - offset)`. If `sub_group_local_id` of the work item that is about to receive the data is less than the smallest `sub_group_local_id` of the work item participating in the shuffle group, `sub_group_local_id` is wrapped back into the subgroup of work items participating in the shuffle operation.
`qcom_sub_group_shuffle_xor(<gentype> source_value, uint offset, qcom_sub_group_shuffle_width_modes_t width, <gentype> default_value)`	Transfers data from a work item in the subgroup to another work item in the same subgroup whose `sub_group_local_id` is equal to `(source sub_group_local_id XOR offset)`. When all work items execute this call, it results in data exchange among them.

Here are a few key notes:

- Divergence:

  - These shuffle built-in functions need to be encountered by all work items in a subgroup that is participating in the shuffle operation.
  - Calling the shuffle function inside a conditional may result in undefined behaviors if the condition is not satisfied by all work items.
- Requirement of the number of work items:
  - When width is `CLK_SUB_GROUP_SHUFFLE_WIDTH_WAVE_SIZE_QCOM`, ensure that there are work items more or equal to the subgroup size available to execute.
  - When width is `CLK_SUB_GROUP_SHUFFLE_WIDTH_W4_QCOM`, ensure that there are four or more work items.
  - When width is `CLK_SUB_GROUP_SHUFFLE_WIDTH_W8_QCOM`, ensure that there are eight or more work items.
- Requirements on offset:
  - It must be constant. All work items participating in a shuffle operation should use the same offset. Any divergence in offset values across the work items will lead to undefined results.
  - The offset value passed into a shuffle operation should not exceed the width specified for that shuffle operation. Violation of this rule will lead to undefined results.
  - When width is `CLK_SUB_GROUP_SHUFFLE_WIDTH_WAVE_SIZE_QCOM`, the offset should be less than maximum subgroup size, as returned by `CL_KERNEL_MAX_SUB_GROUP_SIZE_FOR_NDRANGE_KHR`.
  - When width is `CLK_SUB_GROUP_SHUFFLE_WIDTH_W8_QCOM`, the offset should be less than eight.
  - When width is `CLK_SUB_GROUP_SHUFFLE_WIDTH_W4_QCOM`, the offset should be less than four.

9.3 Image related operations

Adreno GPUs have built-in hardware modules that accelerate common filtering operations, including convolution, box filtering, SAD (sum of absolute difference), and SSD (sum of square difference), etc. Use of these modules can lead to better performance or consume less power than using the common OpenCL kernel language.

9.3.1 Convolution operations

Convolution filter operation multiplies a matrix of image sample values to a matrix of filter weights and sums up the results to produce the output value. Refer to the Adreno OpenCL SDK for the extension documentation and samples. The extension requires the application to create a special image object with a detailed description of the weight matrix, as illustrated in the following example:

```
weight_img_desc.image_desc.image_type =CL_MEM_OBJECT_WEIGHT_IMAGE_QCOM;

weight_img_desc.image_desc.image_width = filter_size_x;

weight_img_desc.image_desc.image_height = filter_size_y;
weight_img_desc.image_desc.image_array_size = num_phases;
weight_img_desc.image_desc.image_row_pitch = 0; // must set to zero
weight_img_desc.image_desc.image_slice_pitch = 0; // must set to zero
weight_img_desc.weight_desc.center_coord_x = center_coord_x;
weight_img_desc.weight_desc.center_coord_y = center_coord_y;
 // specify separable filter. Default (flags=0) is 2D convolution filter
 weight_img_desc.weight_desc.flags = CL_WEIGHT_IMAGE_SEPARABLE_QCOM;
```

The weight matrix descriptor contains all the information required for the operation. Besides the regular image properties such as its width and height, the center coordinate of the weight matrix and the type of convolution, i.e., whether it is separable, are also required. In this extension, two types of convolution filters are supported, 2-D convolution filter and separable convolution filter.

- For 2-D convolution filter:
  - The filter is a 2-D matrix specified by a set of `(filter_size_x * filter_size_y)` weight elements.
  - The filter's center point, whose coordinates are based on the origin at the top left corner of the filter's spatial region.
  - Figure 9-1 illustrates a 2-D convolution filter having size of 10x10 and its center point is located at coordinates `(cx = 1, cy = 1)`.
    - The pixel located at (3, 2) in the image matrix multiples the pixel located at (1, 1) in the weight matrix.
    - The other pixels in the image multiple the corresponding pixels in the weight matrix.
    - The output value located at (3, 2) will be the sum of previous steps.

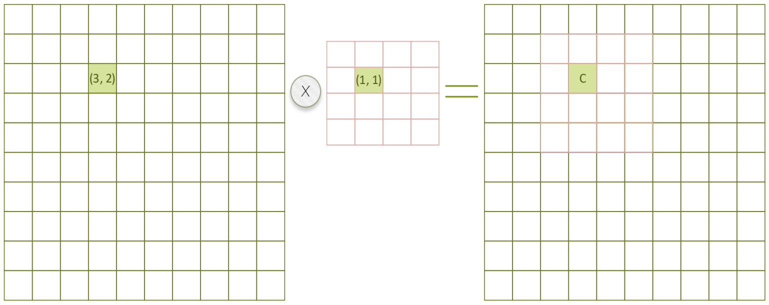


Figure 9-1 2-D convolution filter example

- Separable convolution filter:
  - The filter is a 2-D filter that can be specified by two 1-D filters in x and y directions such that their product yields the 2D filter.

  - A separable filter can be done by performing a 1D convolution of each row using the 1D horizontal filter, then followed by convolving each column of the result using the 1D vertical filter.
  - The result is mathematically the same as applying the 2D filter for each pixel directly.
  - For use cases with many phases, separable filters can offer a performance advantage as the number of computations is reduced by an order of magnitude.
  - Figure 9-2 shows an example of 2D filter and its associated separable ones.

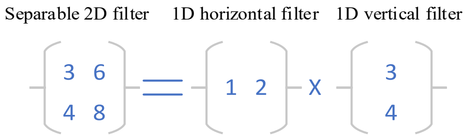


Figure 9-2 Example of separable 2D filter

Table 9-3 lists the key parameters and their descriptions and limits of the weight matrix.

Table 9-3 Requirements of the weight matrix

Parameters	Description and Requirement.
Image type flags	CL_MEM_READ_ONLY and CL_MEM_COPY_HOST_PTR. Weight images only support read access from the kernel.
image_format	Query using clGetSupportedImageFormats with the image type set to CL_MEM_OBJECT_WEIGHT_IMAGE_QCOM.
2D convolution filter: num_phases	num_phases = num_phase_x * num_phase_y. It requires that both values (num_phase_x = num_phase_y) must be to the power of two. The weight values of each filter phase are organized into a 2D slice size of (filter_size_x * filter_size_y).
num_phases for separable	num_phases = num_phase_x = num_phase_y (must be power of two). The weights of 1D horizontal and 1D vertical filters of each filter phase are organized into a 2D slice having slice_height = 2 and slice_width = max(filter_size_x, filter_size_y).
Weight slices if num_phases is greater than 1	The weight slices must be organized in a 2D array, which has the number of slices equal to num_phases. During the built-in execution, the phase value is calculated based on the fraction of subpixel offset of the coordinate of the filter's center point from the pixel center.

Limits for the convolution functions:

- The inputs <image>, <weight_image> and <sampler> must be uniform for all work items within a workgroup executing the function qcom_convolve_imagef.
- The number of filter phases must not exceed the platform's maximum number of phases, which can be queried using clGetDeviceInfo with the <param_name> set to CL_DEVICE_HOF_MAX_NUM_PHASES_QCOM.
- The filter_size_x/y must not exceed the platform's maximum filter sizes, which can be queried using clGetDeviceInfo with the <param_name> set to CL_DEVICE_HOF_MAX_FILTER_SIZE_X_QCOM and CL_DEVICE_HOF_MAX_FILTER_SIZE_Y_QCOM.
- The built-in convolution filter does not work on multi-plane planar images.

9.3.2 Box filter

Box filter is a linear operation taking the average of pixels within a spatial region on the source image covered by the box filter. A box filter is specified by `(box_filter_width, box_filter_height)`, and the coordinates where the center of the box filter is positioned. Mathematically the linear average of pixels produced by a box filter centered at coordinates (`x, y`) is calculated as follows:

```
box_filter(x, y) = sum(weight(i, j) * pix(i, j)) / (box_width *
box_height)
```

Where (`i, j`) are coordinates of the pixels covered by the box filter. It is important to note that some pixels may be only covered partially by the box. Therefore, `weight(i, j)` is adjusted based on the coverage of `pixel(i, j)` by the box. For example, if the pixel is 50% covered, the weight will be 0.5. Figure 9-3 shows an example of a 2x2 box filter applied to a 5x6 image. The weight is automatically calculated by the built-in hardware.

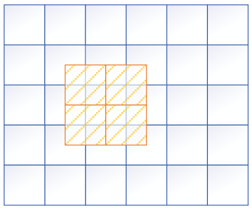


Figure 9-3 Box filtering example when pixels are partially covered

As the box filter is centered at coordinates (2, 2), there are nine pixels covered by the filter. Based on the pixels covered by the filter, the weights are adjusted as follows:

```
W(0, 0) = 0.25  W(1, 0) = 0.5  W(2, 0) = 0.25
W(0, 1) = 0.5   W(1, 1) = 1    W(2, 1) = 0.5
W(0, 2) = 0.25  W(1, 2) = 0.5  W(2, 1) = 0.25
```

Figure 9-4 shows a 3x3 filter where all the pixels from the source image are covered fully. Therefore, the weight of each pixel is equal for this filtering.

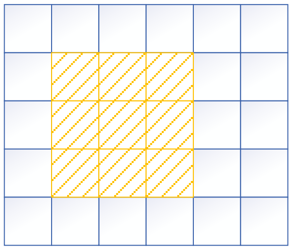


Figure 9-4 Box filtering example when pixels are fully covered

The built-in box filter function is defined as follows:

```
float4 qcom_box_filter_imagef(image2d_t image, sampler_t sampler, float2
coord, const qcom_box_size_t box_size);
```

- Argument `<image>` specifies a valid readable image, on which the box filter is applied.
- Argument `<sampler>` specifies the sampling mode for reading `<image>`.

- Argument `<coord>` specifies the coordinates of the point where the center of the box filter is positioned on the source image plane. Only floating-point coordinates can be used otherwise the kernel will fail to compile.
- Argument `<box_size>` specifies the `box_filter_width` and `box_filter_height`. `box_size` must be passed in as an OpenCL kernel argument and it must be fixed during the runtime of the kernel.

Limits for the functions:

- The inputs `<image>`, `<box_size>` and `<sampler>` must be uniform for all work items within a workgroup executing `qcom_box_filter_imagef` function.
- The `box_size.x/.y` must not exceed the platform's maximum box filter sizes, which can be queried using `clGetDeviceInfo` with the `<param_name>` set to `CL_DEVICE_HOF_MAX_FILTER_SIZE_X_QCOM` and `CL_DEVICE_HOF_MAX_FILTER_SIZE_Y_QCOM`.
- The `<box_size>` must be passed in as an OpenCL kernel argument and it must be fixed during the runtime of the kernel.
- The built-in box filter does not work on multi-plane planar images.

For more details and code examples, refer to the Adreno OpenCL SDK documentation.

9.3.3 Sum of absolute differences (SAD) and sum of square differences (SSD)

Block matching operation measures the correlation (or similarity) of a block within a target image to a reference block within a reference image. There are two error metrics used to measure the correlation of two image blocks: sum of absolute differences (SAD) and sum of square differences (SSD).

Suppose that there are two candidate blocks A and B, and we want to know which one matches block R the best. By computing SAD between A and R, and between B and R, we can select the block that leads to the least error, or the least SAD value. This can be generalized for searching the least SAD value for reference block R cross a set of N target blocks. The two functions are defined as follows:

```
float4 qcom_block_match_sadf(image2d_t target_image, sampler_t sampler,
float2 coord, uint2 region, image2d_t reference_image, uint2
reference_coord);
float4 qcom_block_match_ssdf(image2d_t target_image, sampler_t sampler,
float2 coord,uint2 region, image2d_t reference_image, uint2
reference_coord);
```

These two functions share similar requirements with the convolution and box filter operations in this section. In addition, `region` and `reference_coord`, which specify the size of the target and reference blocks on the images respectively, must be integers. Refer to the Adreno OpenCL SDK for the extension documentation and samples.

9.3.4 Bicubic filter

In addition to the filter modes defined in the OpenCL standard such as `CLK_FILTER_LINEAR` or `CLK_FILTER_NEAREST`, a new filter mode called `CL_FILTER_BICUBIC_QCOM` is added in the newer Adreno GPUs that allows developers to use the hardware accelerated bicubic interpolation. To use this feature, a pragma called `cl_qcom_filter_bicubic` enable is required in the kernel. With the filter mode `QCOM_CLK_FILTER_BICUBIC`, the image read function, `read_imagef`, returns a weighted average of a 4x4 square of image elements. Given the input coordinate `(x, y)` of a 2D image, the 4x4 square is obtained as follows:

Let

```
x0 = (int) floor(x - 1.5f);
x1 = x0 + 1;
x2 = x1 + 1;
x3 = x2 + 1;
```

and

```
y0 = (int) floor(y - 1.5f);
y1 = y0 + 1;
y2 = y1 + 1;
y3 = y2 + 1;
```

and

```
a = frac(x - 0.5f);
b = frac(y - 0.5f);
```

where `frac(x)` denotes the fractional part of `x` and is computed as `x - floor(x)`. Then, the weights are calculated as follows:

```
w_u0 = - 0.5f * a + 1.0f * (a * a) - 0.5f * (a * a * a);
w_u1 =   1.0f     - 2.5f * (a * a) +1.5f * (a * a * a);
w_u2 =   0.5f * a + 2.0f * (a * a) - 1.5f * (a * a * a);
w_u3 = - 0.5f * (a * a) + 0.5f * (a * a * a);
```

and

```
w_v0 = - 0.5f * b + 1.0f * (b * b) - 0.5f * (b * b * b);
w_v1 =   1.0f     - 2.5f * (b * b) + 1.5f * (b * b * b);
w_v2 =   0.5f * b + 2.0f * (b * b) - 1.5f * (b * b * b);
w_v3 =             -0.5f * (b * b) + 0.5f * (b * b * b);
```

The calculated image element value is as follows:

```
refOut = ((t00*w_v0 + t01*w_v1+ t02*w_v2 + t03*w_v3) * w_u0 +
          (t10*w_v0 + t11*w_v1+ t12*w_v2 + t13*w_v3) * w_u1 +
          (t20*w_v0 + t21*w_v1+ t22*w_v2 + t23*w_v3) * w_u2 +
          (t30*w_v0 + t31*w_v1+ t32*w_v2 + t33*w_v3) * w_u3);
```

Where `txy` is the image element at location `(x, y)` in the 2D image. If any of the selected `txy` in the above equations refers to a location outside the image, the border color is used as the color value for `txy`.

Note that the built-in hardware acceleration of bicubic has limited precision. Therefore, it is important to check the precision requirement of your application when using it. Refer to the Adreno OpenCL SDK for documentation and samples on `cl_qcom_filter_bicubic` for more details.

9.3.5 Enhanced vector image operations

In the standard OpenCL, the image read/write functions such as `read_imagef/write_imagef` and can only read or write one pixel (one or multiple components, dependent on the image format) in a single operation. The `cl_qcom_vector_image_ops` extension introduces a new set of OpenCL built-in functions that allow reading and writing of a group of OpenCL image elements in a single operation. They allow the application to either read in or write out a single component across multiple image elements. They are therefore called vector image operations and can offer potential performance gains as well as ease of development.

The built-in functions work across a range of input image formats, and their name indicates the data type of the returned value and their access pattern. Here are some examples of vector image read built-in functions:

9.3.5.1 2x2 read

`qcom_read_imageX_2x2` operations read four elements in the form of a 2x2 vector from the input image:

```
float4 qcom_read_imagef_2x2(image2D_t image, sampler_t sampler, float2
coord, int compid);

half4 qcom_read_imageh_2x2(image2D_t image, sampler_t sampler, float2
coord, int compid);

uint4 qcom_read_imageui_2x2(image2D_t image, sampler_t sampler, float2
coord, int compid);

int4 qcom_read_imagei_2x2(image2D_t image, sampler_t sampler, float2
coord, int compid);
```

The base point specified by `<coord>` is the upper left corner of this vector. `element[0]` is the lower left element. The four output elements are ordered counterclockwise starting from `element[0]` in the 2x2 vector. Specifically, `element[1]` is the lower right, `element[2]` is the upper right, and `element[3]` is the upper left (the base point), as shown in Figure 9-5.

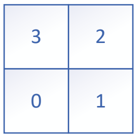


Figure 9-5 2x2 vector image read

9.3.5.2 4x1 read

`qcom_read_imageX_4x1` operations read four elements in the form of a 4x1 vector from the input image:

```
float4 qcom_read_imagef_4x1(image2d_t image, sampler_t sampler, float2
coord, int compid);

half4 qcom_read_imageh_4x1(image2d_t image, sampler_t sampler, float2
coord, int compid);

uint4 qcom_read_imageui_4x1(image2d_t image, sampler_t sampler, float2
coord, int compid);

int4 qcom_read_imagei_4x1(image2d_t image, sampler_t sampler, float2
coord, int compid);
```

Denote `element[0]` as the element located at the base point `<coord>`. The four output elements are ordered from left to right starting from `element[0]` in the `4x1` vector. Specifically, `element[0]` is the leftmost element, followed by `element[1]`, `element[2]`, and `element[3]`.

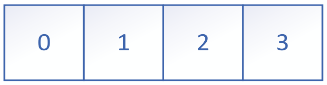


Figure 9-6 4x1 vector image read

9.3.5.3 Image write

A new set of built-in vector image write functions, using the naming convention as `qcom_write_image##datatypev_##pattern_suffix##_format_suffix##`, are supported in Adreno GPUs. The function names explicitly specify the image format that it supports, the vector format, and the format and plane of the destination image. There are many flavors of the image write functions. Here are a few examples:

```
qcom_write_imagefv_2x1_n8n00(image2d_t image, int2 coord, float
color[2])

qcom_write_imagefv_2x1_n8n01(image2d_t image, int2 coord, float2
color[2])

qcom_write_imagefv_2x1_n10p00(image2d_t image, int2 coord, float
color[2])

qcom_write_imagehv_3x1_n10t00(image2d_t image, int2 coord, half
color[3])

qcom_write_imageuiv_4x1_u10m00(image2d_t image, int2 coord, uint
color[4])
```

Some functions require specific image formats such as `NV12_Y` and `TP10_UV`. Y only and UV only images are single-plane derivatives of multi-plane planar images. They can be created by using the `cl_qcom_extract_image_plane` extension.

It is important to understand that the functions must be used with the supported image type and data type. Otherwise, the return will be undefined. For example, the functions `qcom_read_imagef_2x2` and `qcom_read_imagef_4x1` to read floating point only supports images created with formats such as `CL_FLOAT, CL_HALF_FLOAT, CL_UNORM_XX` and `CL_QCOM_UNORM_INT10`. There are some special rules for the vector read and write of images in YUV formats:

- A 4x1 read of U or V values will end up returning the U or V values that correspond to the selected Y pixels.
  - A 4x1 read of the U plane centered at Y00 will return `(U00, U00, U01, U01)`, the four U values that are corresponding to the four Y values.
- A 2x2 read of the U or V planes returns four distinct U or V values. For example, in reference to the following 4x4 image:
  - A 2x2 read centered at the same point will return `(U10, U11, U01, U00)`.

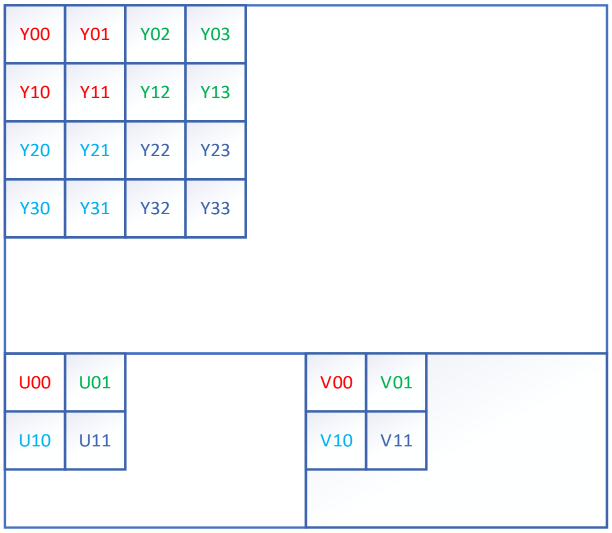


Figure 9-7 Vector read and write for YUV image

Refer to the Adreno OpenCL SDK for documentation and samples on `cl_qcom_vector_image_ops`.

9.3.6 Compressed image support

An extension called `cl_qcom_compressed_image` is supported in Adreno GPUs, which allows images to be read and written in a proprietary compressed format designed by Qualcomm. In addition to the savings of memory bandwidth, it may also reduce power and energy consumption, which is especially useful for many camera and video use cases as typically they are very data demanding. To use the extension, the host must use the functions in Table 9-4 to query the format availability, and its channel and data type support information, as they might be different across different Adreno GPUs. The method to use the compressed image format is like the regular image formats:

- It supports two filtering modes, `CLK_FILTER_LINEAR` and `CLK_FILTER_NEAREST`.
- It supports several different image channels, such as `CL_RGBA`.
- It supports data types, such as `CL_UNORM_INT8`.

A typical use case and workflow of using the compressed format is to have the GPU read the compressed data produced by other modules in the SOC (e.g., camera module). This is often coupled with the zero copy techniques (i.e., either using ION memory or the Android native buffer techniques). Table 9-4 shows the steps of these two approaches, which are very similar except some flags and `enum` types.

Table 9-4 Steps to use compressed image

Step	ION	Android Native buffer
Query supported format.	errcode = clGetSupportedImageFormats(context, CL_MEM_READ_ONLY | CL_MEM_COMPRESSED_IMAGE_QCOM, CL_MEM_OBJECT_IMAGE2D, num_format_list_entries, format_list, &num_reported_image_formats);	

Create a buffer to hold the image data.	cl_mem_ion_host_ptrcompressed_ionmem = {0}; // Initialize ION buffer attributes compressed_ionmem.ext_host_ptr.allocation_type = CL_MEM_ION_HOST_PTR_QCOM;compressed_ionmem.ext_host_ptr.host_cache_policy =CL_MEM_HOST_UNCACHED_QCOM; compressed_ionmem.ion_filedesc = ion_info_fd.file_descriptor; // file descriptor for ION
	compressed_ionmem.ion_hostptr = ion_info.host_virtual_address; // hostptr returned by ION
		cl_mem_android_native_buffer_host_ptr compressed_ANBmem = {0};GraphicBuffer *gb; // previously created compressed_ANBmem.ext_host_ptr.allocation_type = CL_MEM_ANDROID_NATIVE_BUFFER_HOST_PTR_QCOM; compressed_ANBmem.ext_host_ptr.host_cache_policy = CL_MEM_HOST_WRITEBACK_QCOM; // the hostptr to a native buffer and gb is an Android GraphicBuffer
		compressed_ANBmem.anb_ptr = gb->getNativeBuffer();

Create an image object for the application to use.	cl_image_format image_format = {0};
	cl_image_desc image_desc = {0};
	cl_int errcode = 0;
	// Set image format
	image_format->image_channel_order = CL_QCOM_COMPRESSED_RGBA;
	image_format->image_channel_data_type = CL_UNORM_INT8;
	// Set image parameters
	image_desc->image_width = 128;
	image_desc->image_height = 256;
	image_desc->image_row_pitch = 0; // must be 0 for compressed images
	image_desc->image_slice_pitch = 0; // must be 0 for compressed images
	// Create a compressed image
	compressed_rbga_image =clCreateImage(context, CL_MEM_EXT_HOST_PTR_QCOM | CL_MEM_READ_ONLY, image_format, image_desc, (void*)compressed_ionmem, &errcode);
		cl_image_format image_format = {0};
		cl_image_desc image_desc = {0};
		cl_int errcode = 0;
		// Set image format
		image_format->image_channel_order = CL_QCOM_COMPRESSED_RGBA;
		image_format->image_channel_data_type = CL_UNORM_INT8;
		// Set image parameters
		image_desc->image_width = 128;
		image_desc->image_height = 256;
		image_desc->image_row_pitch = 0; // always 0 for compressed images
		image_desc->image_slice_pitch = 0; // always 0 for compressed images
		// Create a compressed image
		compressed_rbga_image = clCreateImage(context, CL_MEM_EXT_HOST_PTR_QCOM | CL_MEM_READ_ONLY, image_format, image_desc, (void*)compressed_ANBmem, &errcode);

OpenCL on Adreno GPUs can decode and read a compressed image and write it to another image with the same compressed format. However, a compressed image can only be read or written inside kernels. The in-place read/write feature, i.e., `CL_MEM_KERNEL_READ_AND_WRITE` is not supported for compressed images. Refer to the Adreno OpenCL SDK for documentation and samples on `cl_qcom_compressed_image`.

9.4 Machine learning on Adreno GPUs

9.4.1 Qualcomm neural processing SDK (SNPE/QNN)

The Qualcomm neural processing SDK (SNPE/QNN) is a mature solution for ML workloads on edge computing. The proprietary, closed-source SDK has achieved tremendous success. It provides customers a large suite of tools and SDKs to accelerate neural networks by using all available computing devices on Snapdragon, including CPU, GPU, and DSP. Manufacturers and developers have adopted the neural processing SDK because of its high commercial quality.

Some advanced developers still prefer to run their ML workloads specifically on Adreno GPUs. They can take advantage of the recently released Qualcomm Adreno OpenCL ML SDK for customization, flexibility, and acceleration on Adreno GPUs.

9.4.2 OpenCL ML SDK for Adreno GPUs

Developers can accelerate many common machine learning operations through the `cl_qcom_ml_ops` extension. Qualcomm optimized operations can offer a significant performance advantage. They support both inference and training. Refer to the Adreno OpenCL MK SDK for documentations and samples. Addtiional details are available from the blog. Additional details are available from the blog: OpenCL Machine Learning Acceleration on Adreno GPU - Qualcomm Developer Network. Here are some elements of the SDK:

- An OpenCL extension, `cl_qcom_ml_ops`, is a core part of the SDK. Available in some of the Adreno A6x GPUs and all A7x GPUs, this extension provides a comprehensive set of API functions to enable many critical ML operations.
- The functions contain highly optimized kernels that fully leverage the Adreno GPU's hardware capability.
- Besides the API functions, the extension also defines the required data structures, tokens, tensor objects, and memory management mechanisms to facilitate using the API functions.
- The SDK provides documentation and samples to help developers leverage this functionality. It features a model conversion tool to convert the model files using the standard ML networks, such as TensorFlow and PyTorch, to the models the APIs can directly consume in the ML extension.
- With the SDK, developers can easily port and adapt their ML application developers using ML Ops instead of writing their OpenCL kernels, which could see performance gains. Refer to the SDK documents and code examples to find more details.

9.4.3 Tensor virtual machine (TVM) and the cl_qcom_ml_ops extension

Recently TVM, a well-known and very active open-source compiler framework for deep learning workloads, has added the support of Adreno's `cl_qcom_ml_ops` extension. It has lowered the bar of using the SDK for developers and helped provide a quick generation and prototyping of the ML network running on Adreno GPUs.

9.4.3.1 Why TVM

TVM can automatically generate several OpenCL kernel implementations for a given ML operation or layer. It can then use an ML-based tuning methodology to find the best-performing OpenCL kernels from a large search space. TVM can do op-level and graph-level optimizations on ML models to generate high-performance OpenCL kernel implementations for a wide range of hardware modules. And, because it is open source, TVM is backed by a large, active community with members from both industry and academia.

9.4.3.2 How TVM works with the cl_qcom_ml_ops extension

The TVM community has introduced BYOC as a way of embedding the high-performance kernels from vendor acceleration libraries (like Adreno) into the main code generated by TVM. Therefore, we are taking advantage of BYOC to integrate the `cl_qcom_ml_ops` extension into TVM for an end-to-end solution.

Although the `cl_qcom_ml_ops` extension is powerful, its proprietary APIs come with a learning curve. The TVM and `cl_qcom_ml_ops` integration is more straightforward than using the SDK on its own. With this integration, developers do not need to understand the specification, the header files, or which APIs to call. It allows developers to start using OpenCL ML on day one without learning about the API definition.

9.4.3.3 How to use TVM with the cl_qcom_ml_ops extension

The Adreno OpenCL ML integration into TVM has been open-sourced and up-streamed. The integration allows developers to easily import deep learning models from the frameworks that TVM supports, such as TensorFlow, PyTorch, Keras, CoreML, MXnet, and ONNX. It uses the graph-level optimizations of TVM and the Adreno OpenCL ML library kernels as much as possible. For any kernels or operators not supported by the `cl_qcom_ml_ops` extension, BYOC allows a fallback option for any back end supported by TVM.

See the TVM repo with OpenCL ML SDK and the blog: Accelerate your machine learning networks using TVM and the Adreno OpenCL ML APIs on Adreno GPUs - Qualcomm Developer Network for more details.

9.4.4 Other features for ML

9.4.4.1 Support of bfloat16 data

The bfloat16 (brain floating point) floating-point format uses 16 bits to represent an approximate dynamic range of 32-bit floating-point numbers by retaining eight exponent bits. Still, it supports only 8-bit precision rather than 24-bit FP32 format. bfloat16 can be used to reduce the data storage requirement and also speed up some machine learning algorithms. Refer to the Adreno OpenCL SDK for more information.

9.5 Other enhancements

9.5.1 Enhancement of 8-bit operations

The `cl_qcom_dot_product8` extension introduces a new set of OpenCL built-in functions for calculating dot products with a pair of four 8-bit components, followed by saturating the addition of the dot product result with a 32-bit accumulator. For this feature, the following compiler #pragma must be enabled:

```
#pragma OPNCL EXTENSION cl_qcom_dot_product8 : <behavior>
```

Two functions are defined. One is for unsigned 8-bit integer and the other is for signed.

Table 9-5 Accelerated 8-bit vector operations

Function	Description	Examples

`int qcom_udot8_acc(uint p0, uint p1, int acc);`	Assume `p0` and `p1` can each be interpreted as four **unsigned** 8-bit components and that acc is interpreted as a **signed** 32-bit accumulator.
	`qcom_udot8_acc` sums the products of the components and adds the accumulator to get a **signed** 32-bit result, using **signed** saturation.
		Calculating an unsigned dot product of (11, 22, 33, 44) and (55, 66, 77, 88) and an accumulator of 9:
		`uchar p0a = 11; uchar p0b = 22;`
		`uchar p0c = 33; uchar p0d = 44;`
		`uchar p1a = 55; uchar p1b = 66;`
		`uchar p1c = 77; uchar p1d = 88;`
		`uint  p0  = (p0a << 24) | (p0b << 16) | (p0c << 8) | p0d;`
		`uint  p1  = (p1a << 24) | (p1b << 16) | (p1c << 8) | p1d;`
		`int   acc = 9;`
		`int result = qcom_udot8_acc(p0, p1, acc);`

`int qcom_dot8_acc(uint p0, uint p1, int acc);`	Assume `p0` and `p1` can be interpreted as four **signed** 8-bit components and four **unsigned** 8-bit components, respectively and that acc is interpreted as a **signed** 32-bit accumulator.
	`qcom_dot8_acc` sums the products of the components and adds the accumulator to get a **signed** 32-bit result, using **signed** saturation.
		Calculating a signed dot product of (-11, 22, -33, 44) and (55, 66, 77, 88), and an accumulator of 9:
		`uchar p0a = -11; uchar p0b =  22;`
		`uchar p0c = -33; uchar p0d =  44;`
		`uchar p1a =  55; uchar p1b =  66;`
		`uchar p1c =  77; uchar p1d =  88;`
		`uint  p0  = (p0a << 24) | (p0b << 16) | (p0c << 8) | p0d;`
		`uint  p1  = (p1a << 24) | (p1b << 16) | (p1c << 8) | p1d;`
		`int   acc = 9;`
		`int result = qcom_dot8_acc(p0, p1, acc);`

Comparable functionality is also available from the Khronos extension - `cl_khr_integer_dot_product`.

9.5.2 cl_qcom_bitreverse

This extension introduces a new OpenCL built-in function for the accelerated reversal of bits in an unsigned integer. Applications using this built-in function may get a performance benefit relative to applications that use other methods to reverse bits. For this feature, the pragma `cl_qcom_bitreverse` must be enabled:

```
 #pragma OPNCL EXTENSION cl_qcom_bitreverse : enable  //disable
```

Once enabled, the function `qcom_bitreverse` can be used to reverse the order of bits. Here is an example:

```
        uint  input  = 0x1248edbf;
        uint  output = qcom_bitreverse (input); //output = 0xfdb71248
```

Refer to the Adreno OpenCL SDK for documentation and samples on `cl_qcom_bitreverse.`

10 OpenCL optimization case studies

This chapter presents a few examples to demonstrate optimization using the techniques discussed in the earlier chapters. In addition to a few simple code snippet demonstrations, two well-known image processing filters, Epsilon filter and Sobel filters, are optimized step-by-step by using many practices discussed in the previous chapters.

10.1 Resources and blogs

A few blogs discuss use case optimization, which is publicly available at the Qualcomm Developer Network. Here are a few of them that developers can refer to:

Table 10-1 Blogs on OpenCL optimizations and other resources

Topics	Links
Summary of OpenCL optimization	https://developer.qualcomm.com/blog/opencl-optimization-stop-leaving-compute-cycles-table
Use case study: Epsilon filter	https://developer.qualcomm.com/blog/opencl-optimization-accelerating-epsilon-filter-adreno-gpu
Use case study: Sobel filter	https://developer.qualcomm.com/blog/opencl-optimization-accelerating-sobel-filter-adreno-gpu
Use case study: matrix multiplication	https://developer.qualcomm.com/blog/matrix-multiply-adreno-gpus-part-1-opencl-optimization
	https://developer.qualcomm.com/blog/matrix-multiply-adreno-gpus-part-2-host-code-and-kernel
OpenCL ML SDK	OpenCL Machine Learning Acceleration on Adreno GPU - Qualcomm Developer Network

The use cases discussed in this section are the Epsilon filter and the Sobel filter, which are partially covered in the blogs.

10.2 Application sample code

10.2.1 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 middle 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, significantly less than the original 64 `read_imagef` accesses.

10.2.2 Vectorized load/store

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

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 retrieves 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 more recent kernel should be ¼ of the original kernel because of this vectorization.

10.2.3 Use image instead of buffer

This example calculates a dot product for each pair, given five million pairs of vectors. The original code uses a buffer object and 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 where buffer object access is not as efficient as texture object access.

Original kernel before optimization	Modified kernel for optimization

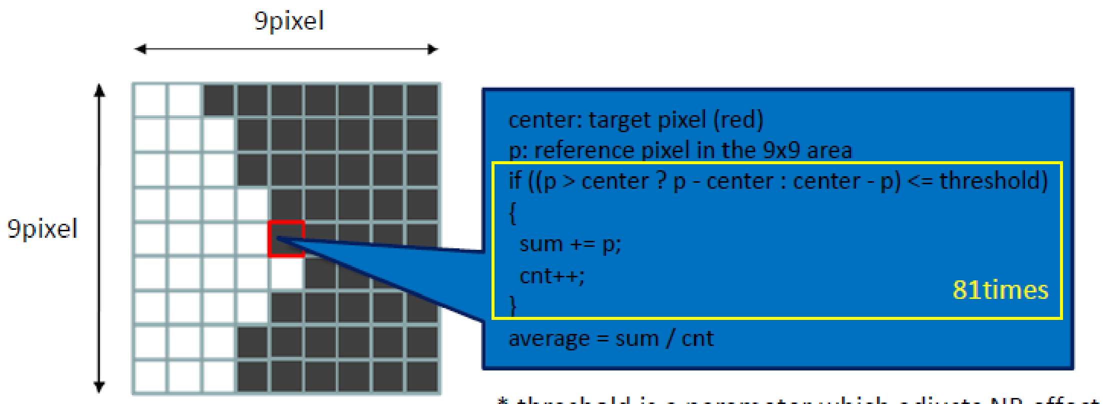
`__kernel void DotProduct(__global const float4 *a, __global const float4 *b,__global float *result){// a and b contain 5 million vectors each`
`// Arrays are stored as linear buffer in global memory`
`result[gid] = dot(a[gid], b[gid]);`
`}`
	`__kernel void DotProduct(__read_only image2d_t c, __read_only 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));`
	`}`

10.3 Epsilon filter

Epsilon filter is used widely in image processing to reduce mosquito noise, a type of impairment occurring at a 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 a certain threshold are filtered.

In this implementation, the Epsilon filter is applied only to the intensity (Y) component of YUV images, as the noise is mainly visible. Also, it assumes that the Y component is consecutively stored (NV12 format), separated from the UV component. There are two basic steps, as illustrated in Figure 10-1 .

- 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. The threshold usually is a constant predefined in the application.



```
center: target pixel (red)
p: reference pixel in the 9x9 area
if ((p > center ? p - center : center - p) <= threshold)
{
  sum += p;
  cnt++;
}
average = sum / cnt
```

* threshold is a parameter which adjusts NR effect

Figure 10-1 Epsilon filter algorithm

10.3.1 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 checks 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 must access 81 floating point pixels. The performance of Adreno A430 GPU is used as the baseline for further optimization.

10.3.2 Data pack optimization

By comparing the amount of computation and data load, it is easy to conclude that this is a memory-bound use case. Thus, the leading 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 types 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 four channels to utilize the TP bandwidth better.
  - 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.

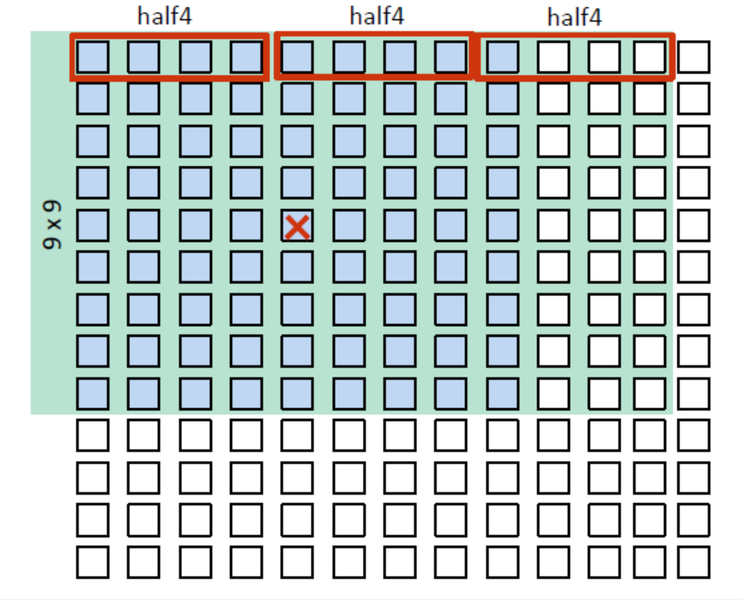


Figure 10-2 Data pack using 16-bit half (fp16) data type

10.3.3 Vectorized load/store optimization

In the previous step, only one pixel is output with 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.

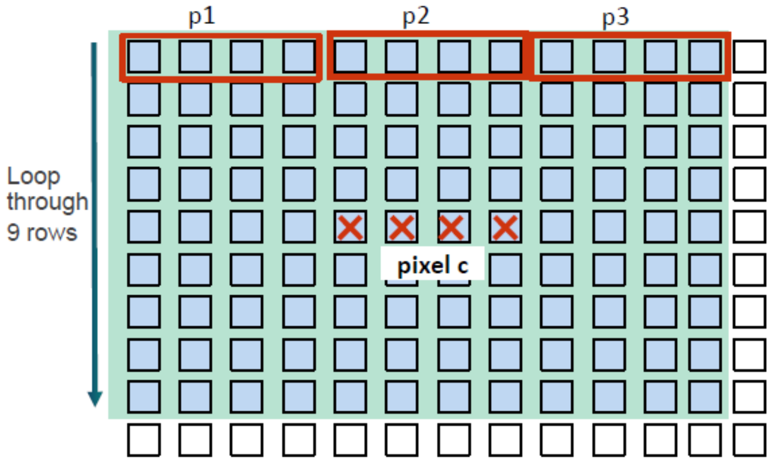


Figure 10-3 Filtering more pixels per work item

Figure 10-3 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.

10.3.4 Further increase workload 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`).

Number of memory access per output pixel: `4x9/8 = 4.5 (half4).`

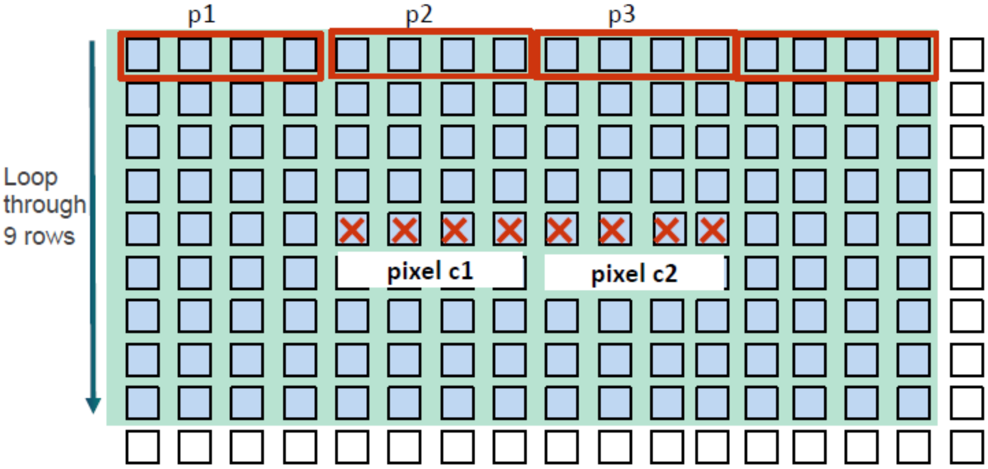


Figure 10-4 Process 8 pixels per work item

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

- The cache hit ratio does not change much, as it is already excellent in the previous step.
- More registers are required, resulting in fewer waves, which hurts the hiding of parallelism and latency.

For experimental purposes, 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 10-5 shows that each work item does the following:

- Reads 6 `half4` vectors per row.
- Outputs 16 pixels.
- The number of memory access per output pixel is 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, seriously hurting the performance.

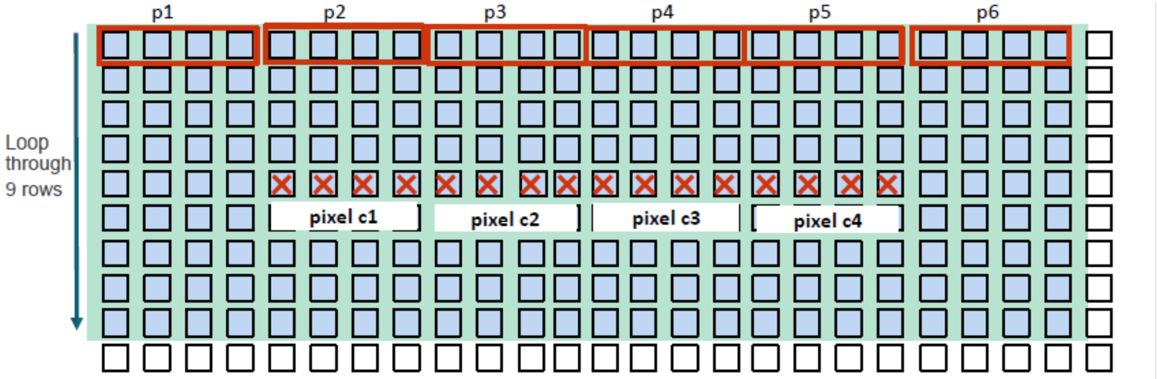


Figure 10-5 Process 16 pixels per work item

10.3.5 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, the surrounding pixels for the 9x9 filtering are loaded into local memory, shown in Figure 10-6 .

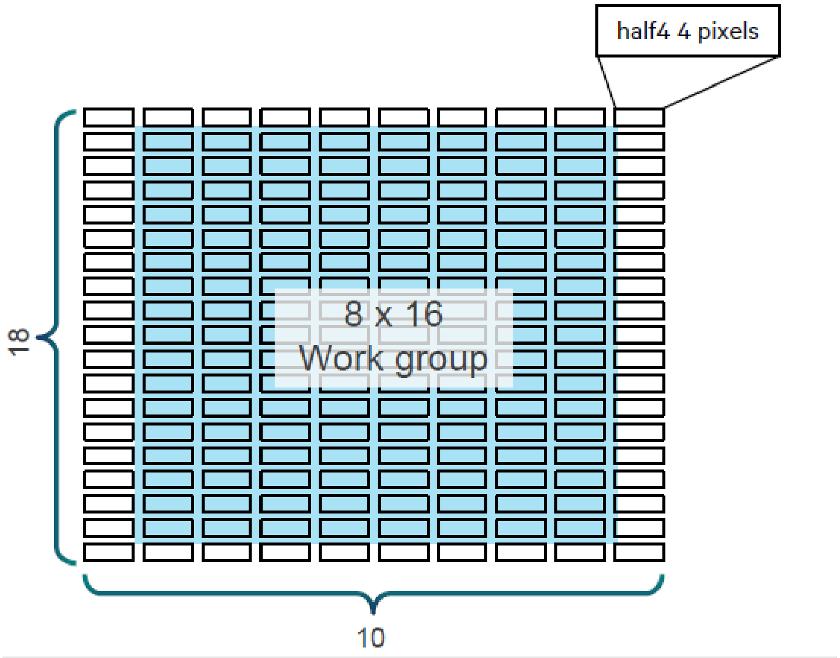


Figure 10-6 Using local memory for Epsilon filtering

Table 10-2 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 10.4.4.

Table 10-2 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, local memory often requires barrier synchronization inside workgroups and 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 better if it has a high cache hit ratio.

10.3.6 Branch operations optimization

The Epsilon filter needs to make a comparison between the 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 10.3.2, and the performance is improved to 5.4x from 3.4x of the baseline.

The critical 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. However, the variable `Cond` may have different values, while the old one uses some costly hardware logic to handle the divergence.

10.3.7 Summary

The optimization steps and their performance numbers are summarized in Table 10-3 . Initially, the algorithm is memory bounded. By doing data packing and vectorized load, it becomes more ALU bound. In summary, the critical optimization for this use case is to load data optimally. Many memory-bound use cases could be accelerated by using similar techniques.

Table 10-3 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		

Step	Optimizations	Image format	Data type In kernel	Vector processing	Speedup
2	Use half type in kernel	CL_R | CL_UNORM_INT8	half	1-pixel/work item	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)			4-pixel/work item	2.4 X
4-1-1	Use LM, increase workgroup size, workgroup size: 8x24				2.8 X
4-1-2	Remove branching operations. Use LM, workgroup size: 8x24				2.9 X
4-2	Remove branching operations				5.4 X

The OpenCL performance of the Epsilon filter with three different resolutions is shown in Table 10-4 . The gains are more visible for larger images. For an image of 3264x2448, a 5.4x performance boost is observed, compared to 4.3x for an image of 512x512 using the initial OpenCL code. A fixed cost is associated with kernel execution regardless of workload, and its weight in the overall performance becomes lower as the workload increases.

Table 10-4 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

10.4 Sobel filter

The Sobel filter, also called the 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 for the horizontal direction and the other for the vertical direction, as shown in Figure 10-7 .

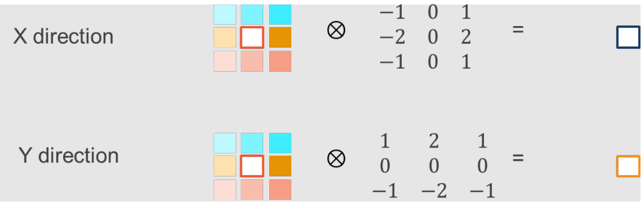


Figure 10-7 Two directional operations in Sobel filter

10.4.1 Algorithm optimization

The Sobel filter is a separable filter that 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 10-8 Sobel filter separability

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

10.4.2 Data pack optimization

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

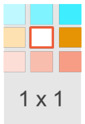


Figure 10-9 Process one pixel per work item: load 3x3 pixels per kernel

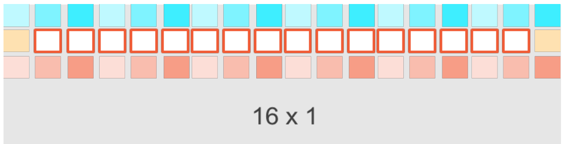


Figure 10-10 Process 16x1 pixels: load 18x3 pixels

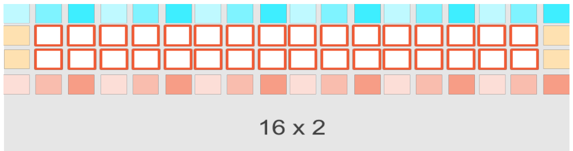


Figure 10-11 Process 16x2 pixels, load 18x4 pixels

The following table summarizes the total number of bytes and average bytes required in each case. For the first case in Figure 10-9 , 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 10-10 and Figure 10-11 . This often reduces data traffic from global memory to GPU and results in better performance.

Table 10-5 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

10.4.3 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`, `char4`, etc. Table 10-6 shows the number of load/store requests for the vectorized cases (assuming the pixel data type is 8-bit char).

Table 10-6 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.

10.4.4 Performance and summary

After applying the two optimization steps, a significant performance boost is observed, as shown in Figure 10-12 , in which the original (single-pixel/work item) on MSM8992 (Adreno 418) is normalized to 1.

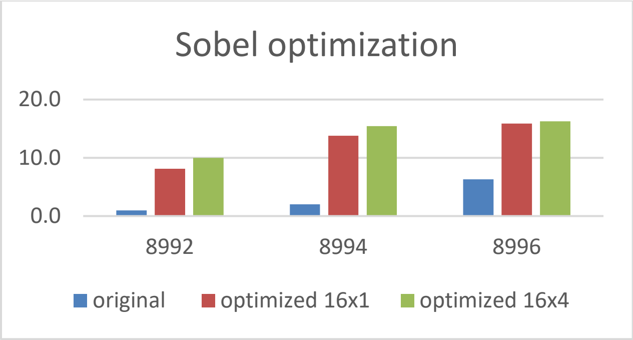


Figure 10-12 Performance boost by using data pack and vectorized load/store

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

- Data packing improves memory access efficiency.
- Use vectorized load/store 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 data overlap that can be reused. Therefore, using local memory does not necessarily improve performance. There could be other options to boost the performance, for example, using texture over a global buffer.

10.5 Summary

A few examples and code snippets 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. Not all results can be exactly reproduced due to compiler and driver upgrades. But generally, a similar performance boost should be achieved with these optimization steps.

11 Summary

This document intends to provide detailed guidance on how to optimize OpenCL programs with Adreno GPUs. Much 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 for doing the same task, it is crucial to read through and understand the guidelines and techniques for Adreno GPUs. Many factors that look minor could have significant performance impacts. These are not easy to tackle without hands-on exercise and training.

Due to time constraints and other factors, not all topics are presented. Developers can find more information via the publicly available OpenCL SDK at the Qualcomm developer network, which is more frequently updated.

Future releases of this document will include more topics.

A How to enable performance mode

Root access is usually required for Android devices (adb root, adb remount) to enable performance mode. These commands need to be 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
```

A.3 Adreno A6x GPU and Adreno A7x GPU

Script to enable performance mode:

```
# usage:   bash run-android.sh <deviceName>
# example: bash run-android.sh sdm845

if [ "$1" == "" ]
then
   echo "Please specify device name (e.g. sdm845) from the list of all
available devices:"
   adb devices -l
   exit 1
fi
export chosen_device=`adb devices -l | grep $1 | cut -d' ' -f1`

if [ "$chosen_device" != "" ]
then
   echo "Found chosen_device $MSM_Device with device id ${chosen_device}"
   export ANDROID_SERIAL=$chosen_device
   adb wait-for-device
   adb root
   adb wait-for-device
   adb remount
   adb wait-for-device
else
   echo "Could not find ${1} attached! Here's a list of all available
devices"
   adb devices -l
   exit 1
```

```
fi

adb wait-for-device
adb root
adb wait-for-device
adb remount
adb wait-for-device
adb shell stop thermal-engine

# CPU clusters online, set to peak
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 "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"
# CPU Bus BW
adb shell "echo -n disable > /sys/devices/soc/soc:qcom,bcl/mode"
adb shell "echo performance > /sys/class/devfreq/soc:qcom,cpubw/governor"

# GPU Bus BW
adb shell "echo performance > /sys/class/devfreq/soc:qcom,gpubw/governor"
#adb shell mount -t debugfs none /sys/kernel/debug
# GPU core freq
adb shell "echo performance > /sys/class/kgsl/kgsl-3d0/devfreq/governor"
# Do adb shell cat /sys/class/kgsl/kgsl-3d0/devfreq/available_frequencies
to query available freqs
# 0:710, 1:675, 2:596, 3:520, 4:414, 5:342, 6:257, 7:180 Mhz
adb shell "echo 0 > /sys/class/kgsl/kgsl-3d0/min_pwrlevel"
adb shell "echo 0 > /sys/class/kgsl/kgsl-3d0/max_pwrlevel"
adb shell "echo 100000000 > /sys/class/kgsl/kgsl-3d0/idle_timer"

# DDR peak freq
# Alternate way to set DDR to peak frequency. This works on Napaliv1
device, but not (yet) on v2
# adb shell "cat /sys/kernel/debug/clk/measure_only_bimc_clk/clk_measure"
-> Need to multiply this number by 2
adb shell "echo "{class: ddr, res: fixed, val: 1800}" >
/sys/kernel/debug/aop_send_message"
```

Example of how to use the script and the output:

```
$ bash perf-mode.sh msmnile
Found chosen_device  with device id 29b1feeb
adbd is already running as root
remount succeeded
adbd is already running as root
remount succeeded
/system/bin/sh: can't create /sys/devices/soc/soc:qcom,bcl/mode: No such
file or directory
/system/bin/sh: can't create /sys/class/devfreq/soc:qcom,cpubw/governor:
No such file or directory
```

Developers can safely ignore the error message (last two lines above). Some of the `/sys` node names are not valid on specific Android devices.

To verify that performance has indeed gone into effect, run the following script using the same device name as the argument.

```
# usage:   bash run-android.sh <deviceName>
# example: bash run-android.sh sdm845

if [ "$1" == "" ]
then
   echo "Please specify device name (e.g. sdm845) from the list of all
available devices:"
   adb devices -l
   exit 1
fi
export chosen_device=`adb devices -l | grep $1 | cut -d' ' -f1`

if [ "$chosen_device" != "" ]
then
   echo "Found chosen_device $MSM_Device with device id ${chosen_device}"
   export ANDROID_SERIAL=$chosen_device
   adb wait-for-device
   adb root
   adb wait-for-device
   adb remount
   adb wait-for-device
else
   echo "Could not find ${1} attached! Here's a list of all available
devices"
   adb devices -l
   exit 1
fi

adb wait-for-device
adb root
adb wait-for-device
adb remount
adb wait-for-device

adb shell "cat /sys/devices/soc/soc:qcom,bcl/mode"
adb shell "cat /sys/class/devfreq/soc:qcom,cpubw/governor"
adb shell "cat /sys/class/devfreq/soc:qcom,gpubw/governor"
adb shell "cat /sys/devices/system/cpu/cpu?/online"
adb shell "cat /sys/devices/system/cpu/cpu?/cpufreq/scaling_governor"
adb shell "cat /sys/devices/system/cpu/cpu?/cpufreq/scaling_cur_freq"

#adb shell mount -t debugfs none /sys/kernel/debug
while [ 1 ]; do
    #adb shell "cat /sys/class/kgsl/kgsl-3d0/devfreq/governor"
    adb shell "cat /sys/class/kgsl/kgsl-3d0/devfreq/cur_freq"
    adb shell "cat
/sys/kernel/debug/clk/measure_only_bimc_clk/clk_measure"
    adb shell "cat /sys/kernel/debug/clk/bimc_clk/measure"
    #adb shell "cat /sys/class/kgsl/kgsl-3d0/idle_timer"
    #adb shell "cat /sys/devices/system/cpu/cpu?/online"
    #adb shell "cat /sys/devices/system/cpu/cpu?/cpufreq/scaling_governor"
    #adb shell "cat /sys/devices/system/cpu/cpu?/cpufreq/scaling_cur_freq"
    sleep 0.5
done
```

This script runs in a loop (press Ctrl+C to exit) and last few lines will print out the GPU clock frequency (585 Mhz for the device below), as shown in the following example:

```
$ bash list-clocks.sh msmnile
Found chosen_device  with device id 29b1feeb
adbd is already running as root
remount succeeded
adbd is already running as root
remount succeeded
cat: /sys/devices/soc/soc:qcom,bcl/mode: No such file or directory
cat: /sys/class/devfreq/soc:qcom,cpubw/governor: No such file or directory
performance
1
1
```

```
1
1
1
1
1
1
performance
performance
performance
performance
performance
performance
performance
performance
1785600
1785600
1785600
1785600
2419200
2419200
2419200
2841600
585000000
cat: /sys/kernel/debug/clk/measure_only_bimc_clk/clk_measure: No such file
or directory
cat: /sys/kernel/debug/clk/bimc_clk/measure: No such file or directory
585000000
cat: /sys/kernel/debug/clk/measure_only_bimc_clk/clk_measure: No such file
or directory
cat: /sys/kernel/debug/clk/bimc_clk/measure: No such file or directory
```

B References

B.1 Related documents

Title	Number
Standards	
The OpenCL™ Overview - The Khronos Group Inc	https://www.khronos.org/api/opencl
The OpenCL™ Specification: API	https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html
The OpenCL™ Specification: C	https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html
The OpenCL™ Extension Specification	https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_Ext.html
Resources	
The 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 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
UI	user interface