QUALCOMM

Qualcomm Technologies, Inc.

Qualcomm® Adreno™ OpenCL Programming Tips

Optimizing OpenCL code on the Adreno GPU

October 2016

Qualcomm Technologies, Inc.

Qualcomm Technologies, Inc.

5775 Morehouse Drive

San Diego, CA 92121

U.S.A.

Table of Contents

Introduction

This document contains concise programming tips for optimizing the performance of OpenCL apps running on the Qualcomm® Adreno™ GPU. It is an abbreviated version of a longer programming guide that is still in progress; in the meantime, developers can use this to optimize their apps.

This document assumes that the reader is sufficiently familiar with OpenCL to have already created an app and is now looking for ways to tune the app's performance. Accordingly, we omit mention of procedures for setting up the development environment and of the references and resources commonly used in OpenCL programming.

Note also that not every application will perform better on the GPGPU. Some applications are prohibitively complex or run better with serial computing on the CPU. You should evaluate the probability of performance improvement before investing time and resources in porting to the GPGPU.

Memory

Most kernels are memory-bound rather than compute-bound. A big part of performance optimization depends on performing load/store between the GPU and system memory effectively.

Vectorization and coalescing

Since the bit width of Adreno is 128, we recommend that you load/store in batches of 128 bits per transaction.

There is no benefit to using ***vload8*** or ***vload16*** over a multiple of ***vload4***.

Coalescing global memory accesses between adjacent work items often leads to better utilization of memory bandwidth.

Image vs. buffer memory objects (*clCreateImage* vs. *clCreateBuffer*).

We strongly recommend loading from image objects (***read_image***) to take advantage of several features:

- The advantage of L1 cache (no L1 for buffer object load)
- The availability of built-in bilinear filtering, if needed
- Automatic hardware handling of boundary conditions (e.g., *CLAMP_TO_EDGE*)

(NOTE: If the image object is declared with ***__read_write*** qualifier, then those features are not available.)

However, we recommend using a buffer object for a more-flexible access pattern (for example, byte-addressable load/store) and for reading and writing within a kernel.

For most applications, it is a good idea to read from image objects and write to buffer objects.

Global memory (GM) vs. local memory (LM)

LM has lower latency than GM, but be aware of the overhead in accessing LM.

A local barrier, if used, will add latency when synchronizing across work items.

Moving data from GM to LM requires intermediate register storage, which can increase the register footprint.

If GM data is cached in L2, an adequate, work group-sized kernel can hide the latency completely. In that case, LM may not show any benefit.

We recommend using LM for intermediate storage between two stages and for caching input data used more than three times.

Private memory

If possible, the compiler will store private memory objects in registers. Otherwise, the compiler will spill to local memory and finally to system memory. At each stage of spilling, performance is lost.

For private arrays, we recommend trying LM instead of private memory.

Register spilling can be detected by indication of "ldp" or "stp" instructions from the Kernel Analyzer in the Qualcomm® Snapdragon™ Profiler.

Constant memory

Up to 3KB of constant memory can be stored in dedicated, constant RAM, which is directly accessible by ALUs. The rest is stored in system memory.

The compiler will attempt to promote constant variables and arrays to constant RAM. However, due to space limitations some constants may not get promoted.

A constant array defined in the kernel program is generally promoted to constant RAM. But a constant array passed in as a kernel argument by a pointer will be stored in constant RAM only if its **size in bytes** (e.g., 1024 in the expression below) is specified using the attribute

__kernel void foo(__const float f __attribute__((max_constant_size(1024))))*

Divergent (or random) access for a constant array may impact performance. In that case, converting to a 1D image object may improve performance.

Zero memory copy (Avoiding memory copy)

To avoid a hidden memory copy, use ***cl_mem_flags*** flag ***CL_MEM_ALLOC_HOST_PTR*** when creating a memory object with ***clCreateBuffer*** or ***clCreateImage***.

When mapping a buffer object or an image object to host memory space, use ***clEnqueueMapBuffer***, ***clEnqueueMapImage*** and ***clEnqueueUnmapMemObject*** APIs.

Be aware that even with zero copy there is a cost to using *Map/Unmap* calls, because the driver has to flush or invalidate the CPU cache as required.

To avoid memory copy when sharing data between different external components and the GPU, use ION memory and the extension *cl_ion_qcom_host_ptr*. Alternatively, use Android native buffers and the extension *cl_qcom_android_native_buffer_host_ptr* to avoid memory copy. Android native buffers are based on ION.

Work group size and shape

It is VERY IMPORTANT to tune 2D work group size by profiling different sizes and shapes (e.g., 8x8, 4x16, 2x32). Work group shape is as important as work group size.

Do not expect *local_work_size=NULL* to result in the best performance in all cases. The driver attempts to pick a reasonable work group size but that will rarely be the optimal size.

In general, the larger the work group size, the better. Larger work groups are better at hiding memory latency, but keep in mind that they may lead to cache thrashing and lower performance if too large. The upper limit on the work group size for a particular kernel is determined by a number of factors, including register usage and the presence of barrier instructions.

If the application will run on multiple products, then be sure to test on as many of them as possible. Do not assume that performance will be uniform for a given chipset, platform or manufacturer.

Data type and bit width

We recommend shorter data types. For example, use *short* instead of *int*. A shorter data type reduces data storage and bandwidth as well.

Regarding half vs. float, 16-bit float (half) is natively supported in Adreno and offers double the throughput of 32-bit float.

DO NOT USE *size_t* in kernel code. *size_t* may result in a 64-bit data type and 64-bit arithmetic operations; 32-bit arithmetic is much faster than 64-bit arithmetic. For example, instead of *size_t x = get_global_id(0)*; use *int x = get_global_id(0)*.

Math functions

Use the built-in *native_** functions as much as possible and always declare “fast-math” (*-cl-fast-relaxed-math*).

Avoid integer divide or modulo, if possible.

Use *mul24* or *mad24* if precision is sufficient. 32-bit integer multiplication is emulated using three instructions.

Use built-in OpenCL math functions (e.g., normalize, rsqrt, pow, max) instead of writing your own functions.

Miscellaneous kernel optimization

Manually unrolling a loop may improve performance.

Avoid divergent branching and excessive conditional checks.

We recommend local atomic over global atomic, so in your testing try local atomic first, then global atomic.

Using a work group barrier is costly.

Using a generic pointer (CL 2.0) could lead to an increase in the register footprint. If the compiler cannot locate the physical type of the generic pointer, it may add instructions to convert between generic and physical types, which is expensive. If you know the pointer types, specify them.

Use ***as_typeN*** to unpack/pack data into other types.

API-level tuning

Avoid creating or releasing memory objects in the middle of GPU execution.

Build kernel source code offline and load the binary at runtime to reduce latency when the application launches.

Use event-driven pipeline and non-blocking function calls.

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

Avoid using memory copy APIs.

Avoid unnecessary synchronization.

Avoid ***clFinish***. Use only when absolutely necessary, or at the very end of a pipeline.

Regarding kernel merging, combine multiple kernels into one to reduce memory traffic when possible. That often saves power.

Regarding kernel splitting, if a kernel is too complex and has a very small work group size, try splitting the kernel into multiple kernels to increase parallelism.

If you are using CL-GL interop, use the Server Side Sync feature (***cl_khr_egl_event*** and ***egl_khr_cl_event*** extensions).

Additional resources

We encourage developers to use Snapdragon Profiler to analyze CPU, GPU, DSP, memory, power, thermal and network data, so they can find and fix performance bottlenecks. Snapdragon Profiler is available on the Qualcomm Developer Network at no charge to registered developers.