高通骁龙的手机平台上 Opencl 的通用编程和优化

目录

1 前言

1.1 目的

这篇文档的主要目的是，向原始设备制造商（OEMs），独立软件供应商（ISVs），第三方开发者们，提供在基于高通骁龙 400 系列、600 系列，和 800 系列的手机平台和芯片上进行开发和优化 Opencl 应用程序的一些准则。

1.2 惯例

函数声明，函数名字，类型声明，属性，和代码示例会用不同的字体格式出现，比如

```
#include
```

变量会用尖括号表示，比如 `< number>`

命令会用不同的格式出现，比如 **`copy a: *.* b:.`**

按钮和键盘名字会用粗体表示，比如点击 **Save** 或者按下 **Enter**

（按照翻译作者也就是我自己的理解，会针对有些不好理解的地方进行一些补充说明，会使用斜体）

1.3 技术支持

针对这篇文档中内容的支持和解释，您可以向高通技术部提交疑问，地址是 https://createpoint.qti.qualcomm.com/ 。

如果您无法访问这个技术支持页面，您可以在技术支持页面进行注册再访问或者向邮箱 support.cdmatech@qti.qualcomm.com 发送邮件。

2　Opencl 的简介

这一章主要讨论 Opencl 标准中的关键概念和在手机平台上开发 Opencl 程序的基础知识。如果想知道关于 Opencl 更详细的知识，请查阅参考文献中的《The OpenCL Specification》。对于已经有 OpenCL 的基础知识和经验的开发者可以跳过这一章，直接跳到下一章阅读即可。

2.1 OpenCL 背景和概述

Opencl 是由 Khronos group 开发和维护的一个开源的和完全免费的标准，针对是如何在异构系统上进行跨平台的程序并行。OpenCL 设计理念是帮助开发者利用最新的异构系统的巨大的计算能力，并使跨平台的应用开发变的容易。

骁龙平台上使用的高通 Adreno GPU 系列是最早全面支持 Opencl 的手机 GPU 之一。

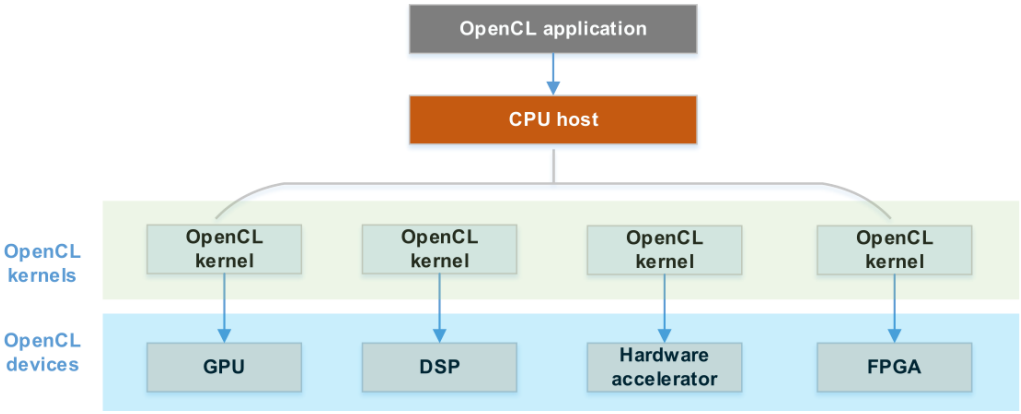

图 2-1 使用 Opencl 的异构系统

图 2-1 表示了一个支持 Opencl 的典型异构系统。这个系统包含了 3 个主要部分：

- 一个主 CPU，实质上就是一个指挥者/控制者，管理和控制应用程序。
- 多个 OpenCL 设备，包括 GPU，DSP，FPGA 和一个硬件加速器。
- Kernel 代码，主设备编译 Kernel 代码，并下载到 OpenCL 设备中执行。

2.2 手机上的 OpenCL

最近这些年，手机的片上系统（SOCs）已经在计算能力、复杂度和功能性上取得了显著的进步。手机 SOCs 上的 GPU（手机的 GPU）是非常强大的，某些顶级的手机 GPU 的原始计算能力能够达到控制台/分布式 GPU（*电脑的 GPU*）的水平。

开发者将面临这样的挑战：如何有效的利用 GPU 如此强大的计算能力，如何在不知道 GPU 的底层实现细节的情况下快速开发应用程序，同时能保持应用程序在不同 SOCs 上兼容性？

OpenCL 的创造就是为了解决上面的问题，OpenCL 的跨平台支持能够让开发者很方便的利用手机 SOCs 上的计算能力。通过使用 OpenCL，许多领域的高级用例能够方便的在手机

SOCs 上使用，比如图像/音频处理，计算机视觉和机器视觉等。

在高通，在 Andreno GPUs 上使用 OpenCL 已经成功加速了许多案例，也展示了出色的性能、功耗和可移植行。针对在骁龙 SOCs 上开发的应用程序，强烈建议在 GPU 上使用 OpenCL 使其加速。

2.3 OpenCL 标准

OpenCL 标准主要包两个方面：OpenCL 的实时运行的 API 和 OpenCL 的 C 语言规范（就是.cl 文件）。API 定义了一系列运行在 Host 上的函数，主要包括资源管理，内核分发（将 kernel 函数分发到不同的 GPU 上运行）以及许多其他的任务；OpenCL 的 C 语言是用来写 kernel 函数的，kernel 函数是运行在 OpenCL 设备（*OpenCL 设备参见图 2-1*）上。API 和 C 语言将会在接下来的章节中说明。

（*对照图 2-1，OpenCL 就是定义一些 API，这些运行在主 CPU 上，将任务划分成一个个的 kernel 函数，将 kernel 函数分发到 OpenCL 设备上运行。*）

2.3.1 OpenCL 的 API 函数

OpenCL 的 API 函数可以分成两种，平台层和实时运行层。表 2-1 和表 2-2 分别总结了平台层和实时运行的一些高阶功能。

表 2-1 OpenCL 平台层的功能

功能	详细描述
发现平台	检查当前的 OpenCL 平台是否可用
发现 OpenCL 设备	在 GPU，CPU 或者其他设备上找到可用的 OpenCL 设备
查询 OpenCL 设备的信息	查询 OpenCL 设备信息包括：全局内存大小（global memory size），本地内存大小（local memory size），最大的工作组数量（maximun workgroup size）等。并且检查该设备支持的扩展功能（OpenCL 定义了标准功能和扩展功能）。
上下文	上下文管理，比如上下文（context）的创建，保留和释放

表 2-2 OpenCL 的实时运行层的功能

功能	详细描述
命令队列的管理（Command queue）	命令队列用于 OpenCL 设备*(比如 GPU)*和主设备*(比如主 CPU)*之间的通信，一个应用程序中可以有多个队列。
创建和编译 OpenCL 程序和 kernel（内核），（编译.cl 文件）	检查 kernel 是否下载和编译正确
为 kernel 准备要执行的数据，创建内存对象，并对其初始化	使用什么样的内存标志*（比如只读只写等）*？是否有能直接创建 0 拷贝的内存*（0 拷贝在第 7 章将会详细说明）*？
创建一个 kernel 调用，并将它提交到对应的 OpenCL 设备上	使用多大的 workgroup（工作组）？
同步	内存同步*（需要等 OpenCL 运行完后再进行结果拷贝）*。
资源管理	传递运行结果*（将 OpenCL 设备上运行结果拷贝到主设备）*和释放资源。

理解这两个 API 层是写 OpenCL 应用程序的基本要求。参照参考文档获取更多细节信息。

2.3.2 OpenCL 的 C 语言*（规定如何写.cl 文件）*

作为 C99 标准的一个子集，OpenCL 的 C 语言是用来写能编译和能在设备*（以后 OpenCL 设备就简称设备）*上运行的 kernel 函数的。有 C 语言编程经验的开发者能够很快上手 OpenCL 的 C 语言编程。但是，为了避免一些常见的错误，理解 C99 标准和 OpenCL C 语言之间的差别也是至关重要的。下面是两个关键的不同点：

- 由于硬件的限制和 OpenCL 的执行模型，一些 C99 的特性在 OpenCL 上并不支持。比如函数指针，动态内存分配（malloc/calloc 等）
- OpenCL 语言在某些方面扩展了 C99 标准，是为了更好的服务编程模型和方便开发。比如：
  - OpenCL 添加了内建函数来查询 OpenCL 内核的执行参数。
  - 为了更好的使用 GPU 硬件，添加了图片加载和存储函数。

2.3.3 OpenCL 的版本和概述

当前的 OpenCL v2.2 和临时 SPIR-V 1.2 标准包含了许多改进的特性。可参照参考目录获取更多细节。

OpenCL 定义了两种简介*（不好翻译）*，嵌入式的简介和完整的简介。嵌入式的简介主要是用于手机设备，相比于传统的计算设备比如台式机的 GPUs，手机设备的计算精度更低，硬件特性更少。参考文档列出了嵌入式简介和完整简介之间的主要不同点。

2.4 OpenCL 的可移植性和向后兼容性

2.4.1 程序的可移植性

作为一个被严格定义的计算标准，OpenCL 有很好的可移植性。如果程序没有使用任何供应商特有的特性或者平台特有的扩展或特性，针对一个供应商平台写的 OpenCL 程序可以很好地在另一个供应商平台上运行。

OpenCL 程序的兼容性已经被 Khronos 的验证程序保证了。如果 OpenCL 供应商声称他们是符合 OpenCL 标准的，Khronos 的验证程序会要求 OpenCL 供应商在他们的平台上通过严格的一致性测试。

2.4.2 性能的可移植性

不像程序的可以执行，OpenCL 的性能并不是可移植的。作为一个高级别的计算标准，OpenCL 的硬件实现是取决于供应商的。不同的硬件供应商有不同的硬件架构，每一种架构都有它自己的优势和劣势。所以，针对某一个供应商平台开发和优化的 OpenCL 的应用程序，在另一个供应商的平台上可能不会有同样的性能

甚至对于同一个供应商，他们的不同系列的 GPU 硬件在微观架构和特性上都会有所不同，这样也会导致 OpenCL 程序表现出显著的性能差异。所以，针对老一代的硬件优化的程序经常需要进行一些调整，来充分发挥新一代硬件的运算能力。

2.4.3 向后兼容性

OpenCL 能够完全向后兼容，来保证针对 OpenCL 旧版本的代码能够毫无问题的运行在新版本的 OpenCL 上。不过需要注意，因为有些 API 函数在新版本已经废弃不使用了，所以如果包含了 OpenCL2.x 版本头文件中并且使用了 OpenCL 1.1 或者 OpneCL1.2 中过时的 APIs，那么需要定义宏 CL_USE_DEPRECATED_OPENCL_1_1_APIS 或者 CL_USE_DEPRECATED_OPENCL_1_2_APIS。

OpenCL 的扩展并不保证在新的设备上能够继续使用，所以使用扩展功能的应用程序必须检查新的设备是否支持他们。

3 在骁龙上使用 OpenCL

在今天安卓操作系统和 IOT(Internet of Things)市场上，骁龙是性能最强的也是最被广泛使用的芯片。骁龙的手机平台将最好的组件组合在一起放到了单个芯片上，这样保证了基于骁龙平台的设备将带来极致的功耗效率和集成的解决方案，从而带来最新的手机用户体验。

骁龙是一个多处理器系统，包含比如多模解调器（multimode modem），CPU，GPU，DSP，位置/GPS，多媒体，电源管理，RF，针对软件和操作系统的优化，内存，可连接性（Wi-Fi，蓝牙）等。

如果想了解当前包含骁龙处理器的消费设备清单，或者想知道关于骁龙处理器更多其他方面的内容，请浏览网站 http://www.qualcomm.com/snapdragon/devices。Adreno GPUs 一般在渲染图像的应用程序中使用，同时它也拥有一般处理器的处理大量计算复杂的任务的能力，比如图像和音频的处理、计算视觉。使用 OpenCL 执行数据并行的计算能够充分发挥 GPU 的能力。

3.1 在骁龙上使用 OpenCL

Adreno A3x，A4x 和 A5x 系列的 GPUs 已经能充分支持 OpenCL，并且已经完全符合 OpenCL 标准。OpenCL 有不同的版本和简介，所以不同系列的 Anreno GPUs 可能支持不同的 OpenCL 版本，如表 3-1 所示：

表 3-1 支持 OpenCL 的 Adreno GPUS

GPU 系列	Adreno A3x	Adreno A4x	Adreno A5x
OpenCL version	1.1	1.2	2.0
OpenCL 简介	Embedded	Full	Full

除了支持的 OpenCL 的版本和简介不同之外，不同的 Adreno GPUs 还可能有其他的不同的性质，比如支持的扩展功能不同，支持的图片对象最大维度不同等。可以通过调用 API 函数 clGetDeviceInfo 获取完整的设备细节信息。

3.2 Andreno GPU 架构

这章将从高层次的角度总体介绍一下 Adreno 中与 OpenCL 相关的架构。

3.2.1 与 OpenCL 相关的 Adreno 的架构

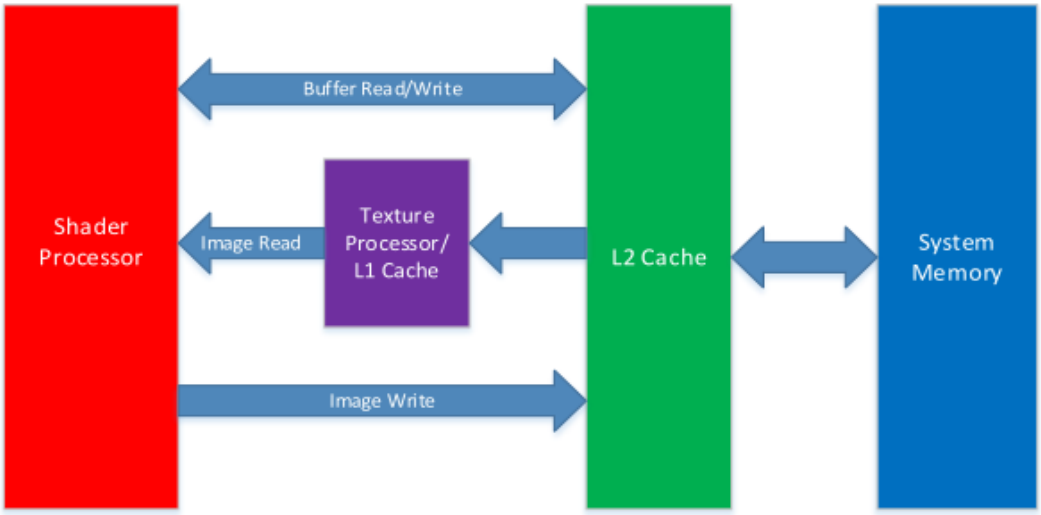

图 3-1 A5x GPUs 中与 OpenCL 相关的架构

Adreno GPUs 支持许多图像和计算 API，包括 OpenGL ES，OpenCL，DirectX 和 Vulkan 等。图 3-1 阐述了与 OpenCL 相关的 Andreno A5x 硬件总体架构，其中省略了图像相关的硬件模块。A5x 和其他 Andreno 系列的 GPU 有许多不同，但是在 OpenCL 上的差异很小。

执行 OpenCL 的关键硬件模块如下：

- SP (Shader or Streaming processor 着色或者流媒体处理器*（图像渲染的相关名词）*)
  - Andreno GPU 的核心部分。包含了许多硬件模块，包括算术逻辑单元（ALU），装载/存储单元，控制流单元，寄存器文件等。
  - 执行图像的着色程序（比如三角着色，片段着色，计算着色）*（图像渲染中使用的）*和执行计算任务比如 OpenCL kernels。
  - 每一个 SP 对应一个或者多个 OpenCL 的计算单元。
  - 根据 GPU 的系列和等级不同，Adreno GPUs 可能包含一个或者多个 SP。低级的芯片组可能只有一个 SP，高级的芯片组可能有更多的 SP。在图 3-1 中，只有一个 SP。
  - 对于使用__read_write （OpenCL2.0 的特性）限定词定义的缓冲区对象和图片对象，SP 将通过 L2 缓存装载和存储。
  - 对于只读的图像对象，SP 通过 texture processor/L1 模块来装载数据。

- TP（Texture processor 纹理处理器*（图像渲染的相关名词）*）
  - 执行纹理操作，比如基于内核的请求进行纹理的获取和过滤。
  - TP 是与 L1 cache 结合在一起，当出现纹理数据缓冲区未命中时，L1 cache 从 UCHE*(Unified L2 Cache 下面讲)*中获取数据。
- Unified L2 Cache（UCHE）
  - 负责为 SP 从缓冲区*（Buffer 对象）*对象存储/装载数据，负责为 L1 cache 请求装载图像对象（Image 对象）。*（如图 3-1，就当 SP 请求从缓冲区装载/存储数据，通过 UCHE ,当 L1 cache 请求装载图像对象时，通过 UCHE）*

3.2.2 Waves 和 fibers*（高通内部定义的概念，直接用英文，具体意义会在章节中详细解释）*

在 Adreno GPUs 中，最小的执行单元叫 fiber。一个 fiber 对应 OpenCL 中的一个 work item*（工作项，opencl 中的概念，因为代码中也会经常用到，所以直接用英文表示）*。以“固定步调”一起运行的一组 fiber 叫做 wave。SP 可以同时容纳多个已经激活的 wave。每一个 wave 与前面的程序独立，并且与其他 waves 的运行状态无关。需要注意以下几点：

- wave 的大小，或者说是在一个 wave 中 fbers 的数量，对于指定的 GPU 和 kernel，这个数量固定的。*（GPU 给出最大值，kernel 函数给出当前需要运行的值）*。
- 在 Adreno GPU 中，wave 的大小依赖于 GPU 的系列和编译器，一般值是 8,16,32,54,128 等。
- 一个 workgroup*（工作组，opencl 中的概念）*可以被一个或者多个 waves 执行，这主要由 workgroup 的大小决定。比如说，如果一个 workgroup 的大小小于或者等于 wave 的大小，只需要一个 wave 就可以。当然越多的 wave 当然越好，因为 wave 之间能够更好地隐藏延迟。*（像 cpu 上的流水线，比如一个 wave 执行完 load 数据，开始计算，另一个 wave 就可以开始执行 load 数据操作。这样第二个 wave 的 load 操作时间就被隐藏了。）*
- SP 可以在一个或者多个 waves 上同时执行 ALU 指令。
- 在一个 workgroup 中，可以流水线运行的最大的 wave 的个数是由硬件决定的。典型的，Adreno GPUs 支持 16 个 waves。
- 给定一个 kernel 函数，在一个 SP 上能激活的最大 waves 的个数是由 kernel 的寄存器占用和寄存器文件的大小决定*（就是虽然硬件最大支持 16 个 waves 同时运行，但是如果 kernel 函数占用寄存器过多，寄存器不够用，可能只有 8 个 waves 同时运行）*，同时也由 GPU 的系列和等级决定。
- 一般地，kernel 函数越复杂，可激活的 waves 越小。*（这里提供了优化思路，将复杂的 kernel 函数拆成多个简单的 kernel 函数，能够提高并行速度）*
- 给定一个 kernel 函数，最大的 workgroup 大小是 wave 的大小和允许的 wave 的最大个数的乘积。

OpenCL 1.x 的文档中并没有暴露 wave 上的概念，在 OpenCL2.0 中，已经允许应用程序通过 cl_khr_subgroups 扩展功能使用 wave，不过是从 Adreno A5x GPU 开始支持的。

3.2.3 延迟隐藏

延迟隐藏是 GPU 有效并行化处理中最强大的一个特点，使得 GPU 达到一个很高的吞吐量。举例如下：

- SP 开始执行第 1 个 wave。
- 在执行完几个 ALU 指令后，这个 wave 需要从外存中（可能是全局/本地/私有内存）获取更多的数据进行接下来的处理，而这些数据当前没有获取到。
- SP 为这个 wave 发送数据获取数据请求。
- SP 切换到已经准备好的第 2 个 wave 开始执行。

- SP 继续执行第 2 个 wave，直到第 2 个 wave 依赖的内容（*比如数据或者寄存器之类*）没有准备好。
- SP 可能会切换到第 3 个 wave，或者切回到第 1 个 wave 执行，如果第一个 wave 已经可执行（数据已经获取到）的话。

在这种方式下，SP 一直处于非常忙碌的状态，而且就像是全部时间都在工作，或者说外部依赖项已经被很好隐藏了。

3.2.4 workgroup 分配

一个典型的 OpenCL kernel 需要用到多个 workgroup。在 Adreno GPU 中，每一个 workgroup 被分配给一个 SP，一般地，在同一时间内每一个 SP 只能运行一个 workgroup。如果有剩下的 workgroup，会在 GPU 中排队等待执行。

用图 3-2 所示的 2 维 workgroup 为例，并假设该 GPU 有 4 个 SP。图 3-3 表示了这些 workgroup 如何被分配到不同的 SP 上。在这个例子中，共有 9 个 workgroup，并且每一个都由一个 SP 上执行。每一个 workgroup 有 4 个 wave，wave 的大小是 16.

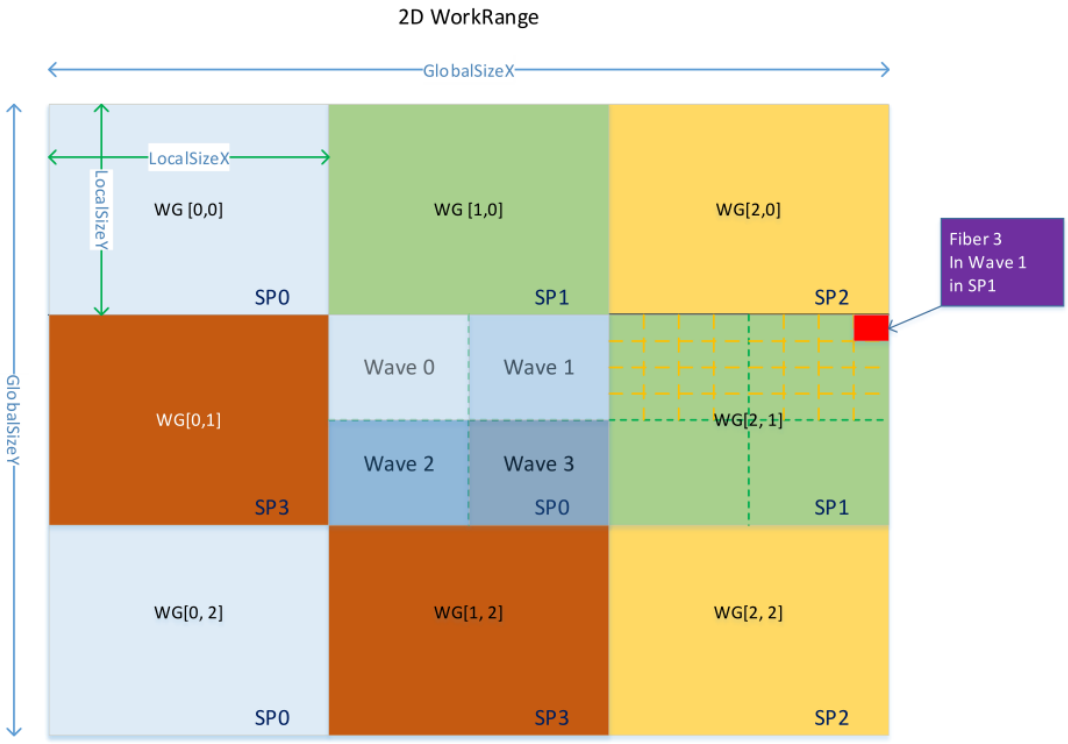

图 3-2　workgroup 的布局和分派到在 Andreno GPU 上执行的例子

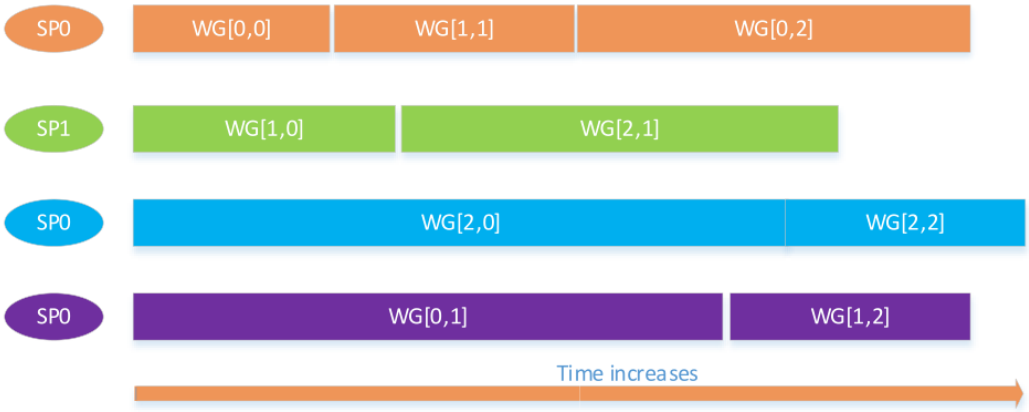

图 3-3　workgroup 被分配到 SP 上执行的例子

OpenCL 标准既没有定义 workgroup 启动和执行的顺序，也没有定义 workgroup 之间的同步机制。对于 Adreno GPUs，开发者不能假设 workgroup 在 SP 上按照指定的顺序启动。同样地，wave 也不能假设按照指定的顺序启动。

在大部分的 Adreno GPUs 中，一个 SP 在同一个时间只能运行一个 workgroup，而且一个 workgroup 必须完成后，另一个 workgroup 才能开始。但是在更高级和更新的 GPU 系列中，比如 Adreno A540 GPU，一个 SP 上可以执行多个 workgroup。

3.3 Adreno A3x,A4x 和 A5x 在 OpenCL 上的不同点

每一个新的 Adreno GPU 系列将会在 OpenCL 的特性和性能上带来很大的提升。这一章将讨论影响 OpenCL 性能的关键改变。

3.3.1 L2 cache

从 Adreno A320 和 Adreno 330 GPUs 到 Adreno A420, A430,A530 和 A540 GPUs，为了更好的效率和性能，L2 cache 的架构进行了极大的改进，同时还增加了 L2 cahe 的容量。

3.3.2 Local memory 本地内存

从 Adreno A3x 到 A4x 和 A5x 系列，Local memory 在容量，装载/存储吞吐量和合并访问（coalesced access）上有所提升。表 3-2 表示了在不同系列上合并访问的不同点。

表 3-2　本地内存性能总结

GPUs	Adreno A3x	Adreno A4x	Adreno A5x
合并访问	不支持	不支持	支持，每次操作可以由最多 4 个 work item 装载/存储 128 位。

合并访问是 OpenCL 和 GPU 并行计算中的一个重要概念。从本质上说，它指的是，基础硬件能够将多个 work item 的数据装载/存储请求合并成一个请求，从而提升数据的装载/存储效率。如果没有合并访问的支持，硬件必须针对每个单独的请求进行装载和存储的操作，这将会导致较差的性能。

图 3-4 阐述了合并访问和非合并访问之间的不同。为了能够将多个 work items 的请求结合在一起，请求的数据地址必须是连续的。在合并访问中，Adreno GPUs 能够在一次处理中给 4 个 work items 装载数据，而非合并访问中，需要 4 次处理才能够装载同样数量的数据。

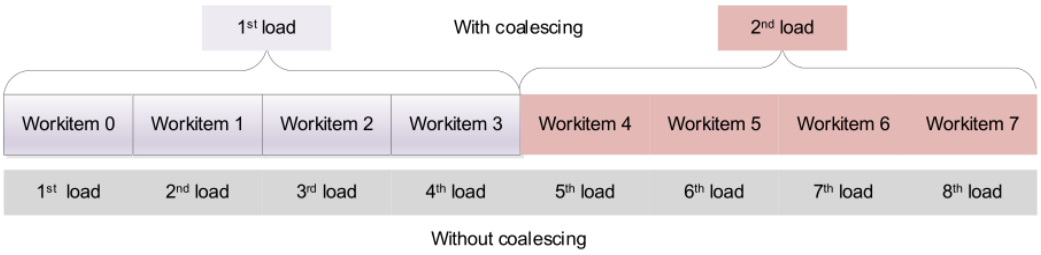

图 3-4 数据装载中合并访问 vs 非合并访问

3.4 图像和计算任务之间的上下文切换

3.4.1 上下文切换

在 Adreno GPUs 中，如果一个高优先级的任务，比如图像用户界面（UI）渲染进行了请求，同时一个低优先级任务正在 GPU 上运行，那么后者将会被暂停，然后 GPU 切换到高优先级的任务上执行。当高优先级的任务执行完毕，低优先级的任务将会被恢复。这种任务的切换叫做上下文切换。上下文切换是非常耗时的，因为需要复杂的硬件和软件操作。然而，这又是很重要的一个特性，能够使得紧急的和对时间要求严格的任务及时完成，比如自动类的应用程序。

3.4.2 限制 kernel/workgroup 在 GPU 上的执行时间

有时候，一个计算任务的 kernel 函数可能会执行一段时间，或者可能会触发一个警告而导致 GPU 重启。为了避免这些不可预测的行为，并不建议 kernel 函数中有需要长时间才能完成的 workgroup。通常情况下，在 Android 设备上，UI 渲染经常发生，比如每 30 ms，所以一个长时间运行的 kernel 可能会引起 UI 滞后或者没有响应，从而导致用户体验不好。理想的 kernel 执行时间是根据实际情况而定的。不过，一个比较好的通用准则是，一个 kernel 的执行时间应该在 10ms 的数量级上。

3.5 OpenCL 标准上的相关提升

Adreno A3x GPU 支持 OpenCL 1.1 的嵌入式版简介，Adreno A4x GPU 支持 OpenCL 1.2 的完整版，Andreno A5x GPU 上支持 OpenCL 2.0 完整版。

从 OpenCL 1.1 的嵌入式版简介到 OpenCL 1.2 的完整版，主要的改变是在软件上，而不是在硬件上，比如提升了 API 函数。

不过，从 OpenCL 1.2 的完整版到 OpenCL 2.0 的完整版，引进了许多新的硬件特性，比如 SVM(共享的虚拟内存)，kernel-enqueue-kernel，等。表 3-3 列出了 3 个 Adreno 系列的 GPU 上，在 OpenCL 支持上的不同点。

表 3-3 在 Adreno GPU 上的支持的标准 OpenCL 的特性

特性	Adreno A3x 支持的嵌入式 OpenCL1.1	Adreno A4x 支持的完全 OpenCL1.2	Adreno A5x 支持的完整 OpenCL2.0
分开编译和链接对象	不支持	不支持	支持
舍入模式	舍入为 0	舍入到最接近的偶数	舍入到最接近的偶数
kernel 中编译	不支持	支持	支持
1 维的纹理,1 维/2 维的图片数组	不支持	不支持	支持（仅在合并获取时）
共享虚拟内存	不支持	不支持	支持
管道	不支持	不支持	支持
转载/存储图像	不支持	不支持	支持
嵌套并行	不支持	不支持	支持
KEK （Kernel-enqueue-kernel）	不支持	不支持	支持
通用内存空间	不支持	不支持	支持
C++ 原子操作	不支持	不支持	支持

3.6 OpenCL 扩展

除了支持 OpenCL 的核心功能外，Adreno OpenCL 平台还通过扩展支持许多额外的功能特性，从而能够进一步提高了 OpenCL 的可用性和能够充分使用 Adreno GPU 的先进的硬件能力。指定 Adreno GPU 上可用的扩展功能可以通过函数 `clGetPlatformInfo` 函数查询。扩展功能的文档可以在 QTI 开发者网页上获取。（网址 https://developer.qualcomm.com）

4. Adreno OpenCL 的程序开发

这章将简要讨论一些开发 Adreno OpenCL 应用程序的基本要求，下面将会介绍如何调试和统计程序性能。

4.1 安卓平台上开发 OpenCL 程序

目前，Adreno GPU 主要是在安卓操作系统和在部分 Linux 系统上支持 OpenCL。为了开发带 OpenCL 的安卓 app，开发者必须熟悉 android 软件开发套件（SDK） 和本地开发套件（NDK *用来运行 C/C++ 的*）。更多关于 Android SDK 和 NDK 的信息，可分别参考 https://developer.android.com/index.html 和 https://developer.android.com/ndk/index.html。

在这章和接下来的章节中，我们假设是在 Android 平台上进行开发，并且开发者对 Android SDK 和 NDK 有相关经验。基于 Linux 的 app 开发与之类似。

下面是关于在骁龙平台上开发 OpenCL 的一些必要条件：

- 支持 OpenCL 的骁龙设备。并不是所有的骁龙设备都支持 OpenCL。可参考表 3-1 获取更多细节。
- OpenCL 软件。在 Adreno GPUs 上的 OpenCL 需要依赖 QTI 合适的库文件。
  - 检查设备上是否安装了 OpenCL 库。核心库是 libOpenCL.so，通常会放在设备的/vendor/lib 目录下。
  - 一些生产商会选择并不包含 OpenCL 软件（比如 Google 的 Nexus 和 Pixel devices）。
- OpenCL 必须运行在 NDK 层。
- 对于开发和测试来说，root 访问权限并不是必须的，不过如果需要使用高性能模式在 SOC 上运行，就会需要 root 权限。

表 4-1 在 Adreno GPUs 上开发 OpenCL 的条件

条目	要求	备注
设备	Adreno A3x/A4x/A5x GPUs	
操作系统	安卓，Linux	只有某些 Linux 平台支持 OpenCL
需要的设备软件	设备上的 libOpenCL.so	可能有些设备上没有
开发要求	Adreno NDK/SDK	OpenCL 代码需要运行在 NDK 层上
设备的 root 访问权限	通常不需要	在高性能模式下需要

4.2 调试工具

由于 GPU 的并行执行特点，调试 OpenCL 核通常是比较具有挑战的。Adreno GPUs 支持在 kernel 函数内部调用 `printf` 函数，这个对于调试来说非常有用。使用 `printf` 时，建

议减少工作负载，因为 printf 会减慢代码的执行，输出有条件限制的变量从而避免打印出太多次。比如说，可能只使能一个有问题的 workgroup，甚至是单个有问题的 work item（通过在 CLEnqueueNDRangeKernel 中设置合适的偏移）。

知道设备的软件版本号是很重要的，因为某些错误或者问题或许已经在新版本中解决了。使用 API 函数 clGetDeviceInfo，可以查询软件（驱动）和编译器的版本。可查看参考文献获取更多信息。

4.3 骁龙分析器（Snapdragon Profiler）

骁龙分析器是由 QTI 提供的一个统计分析工具，可以运行在 Windows，Mac 和 Linux 平台，能够让开发者分析在 Android 上运行的骁龙处理器的一些信息，包括 CPU，GPU，DSP，内存，功率，热量，网络数据等。它支持 OpenCL 和许多图像的 APIs，比如 OpenGL ES 和 Vulkan。为了获取更多细节，请参考 https://developer.qualcomm.com/software/snapdragon-profiler.

下面是骁龙分析器针对 OpenCL 进行统计分析而提供的关键特性：

1. 分析器有一个 kernel 分析器，可以对一个指定的 kernel 进行静态分析。它会提供比如寄存器占用空间，指令总数和每一种类型指令的数量等信息，这些可以帮助开发者更好的优化 kernels。

2. 分析器会对指定的 OpenCL 应用程序提供 OpenCL 的 API 调用轨迹和日志。这个功能可以让开发者从 API 层面就发现和解决瓶颈，同样也可以调试程序*（通过查看调用轨迹和日志）*。

3. 分析器会提供一些信息，比如 GPU 繁忙率，ALU 使用率，L1/L2 cache 命中率等，这些是开发者分析 kernel 函数性能问题的基本信息。

4. 分析器支持基于命令行的应用程序，同样支持 Android GUI app。

4.4 性能统计

给定一个 app，准确统计他的性能是很困难的。GPU 计时和 GPU 计时是两个常用的方法，它们的不同点将会在接下来的章节中讨论。

4.4.1 CPU 计时

CPU 计时是用来测量在主设备端调用 OpenCL 的完整执行时间。可以通过使用标准的 C/C++语言库中提供的任何日期或者时间的函数来实现。下面是一个使用 gettimeofday 的例子。

```
#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)));
}
```

OpenCL 的实时运行队列 API 函数可以分为阻塞调用和非阻塞调用。对于非阻塞调用，使用 CPU 计时需要考虑以下几点：

- 非阻塞调用意味着，主设备提交完任务（任务通常是放在另外一个 CPU 线程中排队等待执行）后，就会执行下一条指令，而不是等待函数执行完毕。
  - kernel 执行的 API 函数，clEnqueueNDRangeKernel，是一个非阻塞函数。
- 对于非阻塞调用，真正的执行时间并不是函数调用的前后时间差。

当使用一个 CPU 计时器在 host 端测量 kernel 执行时间的时候，必须保证这个函数完全执行完，可以通过使调用 clWaitforEvent（如果针对非阻塞调用有一个 eventID）或者 clFinish 保证。同样的规则适用于调用内存交换函数的时候。

4.4.2 GPU 计时

所有的 OpenCL 队列函数的调用，可以有选择的向 host 返回一个事件对象，OpenCL 的性能统计 API 函数可以使用这个事件对象来查询执行时间。Adreno GPUs 用它们自己的时钟和计时器来测量函数执行流程，并且 GPU 的执行时间是由 GPU 硬件计数器决定的，与操作系统无关。

要使能 GPU 计时器的功能，需要对当前的命令队列，在函数 clCreateCommandQueue 或者 clSetCommandQueueProperty 中合适位置设置标志 CL_QUEUE_PROFILING_ENABLE。而且，必须为队列函数提供一个事件对象。一旦函数执行完，可以使用 API 函数 clGetEventProfilingInfo 来获取命令队列的性能统计信息。

对于一个 clEnqueueNDRangeKernel 调用，使用 clGetEventProfilingInfo 函数并设置 4 个统计参数，包括 CL_PROFILING_COMMAND_(QUEUED，SUBMIT，START,和 END）后，可以提供一个在 Andreno GPU kernel 启动延迟和 kernel 执行时间的精确的快照，如图 4-1 所示。

- 前两个参数 CL_PROFILING_COMMAND_(QUEUED 和 SUBMIT)之间的差，给出软件的开销和 CPU cache 操作的开销。OpenCL 软件可能选择一个 kernel 先加入队列，然后与接下来入队的其他 kernel 一起提交，如果队列中的 kernel 函数足够多。开发者可以使用 clFlush 来加速提交。
- CL_PROFILING_COMMAND_(SUBMIT 和 START）的差可以给出 GPU 正在运行的其他工作的时间。
- CL_ PROFILING _COMMAND_(START 和 END）之间的差就是 kernel 在 GPU 上的

运行时间。

开发者主要就是缩小实际的 kernel 的运行时间。对于其他两个比较难控制的时间来说，这个的提升相对简单些。

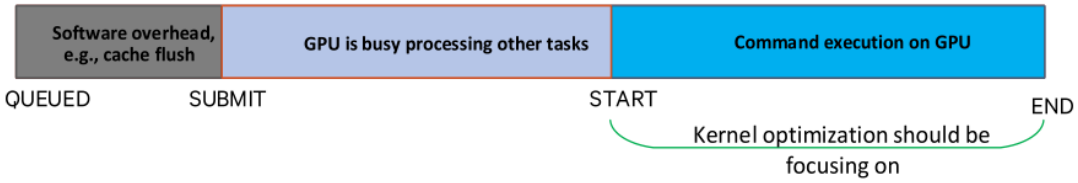

图 4-1 在 Adreno GPUs 中 clEnqueueNDRange 中的性能统计标志

4.4.3 GPU 计时 vs. CPU 计时

GPU 和 CPU 计时器都可以用来统计性能性能，那对于应用程序来说使用哪一个来呢？尽管 GPU 计时器能能够精确的测量出 GPU 执行时间，但是一些硬件操作时间（比如 cache 刷新）和一些软件操作（比如 CPU 和 GPU 的同步）是不在 GPU 的时钟系统中的。所以，对于 kernel 执行，GPU 计时统计的性数据看起来比 CPU 计时统计的要好。下面是两个实际操作中的建议：

- GPU 计时应该用来衡量 kernel 的优化效果。GPU 计时能够准确的统计出，从 GPU 执行的角度，通过这些优化的步骤能提升了多少性能。
- CPU 计时应该用来测量整个程序的端对端的性能（就是整个程序的性能）。如果 OpenCL 程序是整个应用程序流水线的一部分，那应该使用 CPU 计时来统计。

4.4.4 高性能模式

骁龙的 SOC 拥有先进的动态时钟和电源管理机制，这种机制能够自动控制系统使系统在特定的场景中运行在低功耗模式下，这种模式能够减少耗电。通常情况下，如果有高强度的工作负荷，系统会自动提升时钟频率和电压，使得设备进入所谓的高性能模式，从而提升性能和满足高工作负荷。

对于 OpenCL 优化，如果系统动态调整了时钟频率，理解和统计性能是十分困难的。因此，为了性能的精度和一致性，建议使能高性能模式。

如果没有设置高性能模式，在一系列的 OpenCL kernel 中，第一个 kernel 函数通常会表现出更长的启动延迟和更慢的执行时间。所以，必须在启动真正的 GPU 任务之前，需要用一些简单的 kernel 热身 GPU。

一个 OpenCL kernel 的性能不是只依赖于 GPU。在 CPU 上运行 API 函数跟在 GPU 上执行的 kernel 一样重要。所以，为了达到最佳性能，CPU 和 GPU 都需要使能高性能模式。另外，为减少来自 UI 渲染的接口调用，建议：

- 保证当前正在统计的性能的应用程序已经渲染了整个屏幕，这样其他的动作不能再更新屏幕。

- 如果是一个内部的程序，保证 SurfaceFlinger 并没有在安卓上运行。这样就能保证，只有正在被统计性能的应用函数在使用 CPU 和 GPU。

对于 A3x, A4x 和 A5x GPU，使能高性能模式的一系列命令有很大的区别。可参考 Section A 获取更多细节。

4.4.5 GPU 的频率控制

应用程序可以使用 cl_qcom_perf_hit 扩展功能来控制 GPU 频率。当创建 OpenCL 上下文时，这个扩展功能允许应用程序设置一个性能提示属性。性能的等级可以使 HIGH，NORMAL 和 LOW. NORMAL 性能等级会使能动态的时钟和电压控制。HIGH 和 LOW 性能等级将会禁止这种动态控制，强制 GPU 分别运行在最高和最低频率上。

注意：性能等级仅仅是一个提示。驱动会尝试遵循这个提示，但是，其他一些因素，比如热量控制，或者外部的应用程序或服务都可以否决这个提示。这个性能提示的扩展功能给了应用程序一些平衡功率/性能弹性。但是，必须要小心的使用，因为这个对 SOC 层的功耗消耗有很大的影响。

5. 性能优化的概述

这章提供了一个 OpenCL 应用程序优化的总体概述。更多的细节将会在接下来的章节中找到。

注意： OpenCL 程序的优化是具有挑战性的。相比初始的程序开发工作，经常需要做更多的工作。

5.1 性能移植性

就像在 2.4.2 节中讨论的那样，在不同的架构之间，OpenCL 一般都没有很好的性能移植性。针对某一个平台，特别是针对某个 GPU 优化的 OpenCL 应用程序，移植到 Adreno GPU 上后可能没有相同的性能。编程指南和其他 OpenCL 厂商的最佳做法，可能对 Adreno GPU 完全不适用。因此，针对在 Adreno GPU 上的优化，通读整个文档是非常重要的。此外，针对一种 Adreno GPU 优化的应用程序可能需要经过部分调整或者优化，才能在其他系列的 Adreno GPU 上达到最佳性能。

5.2 优化的总体视角

优化一个 OpenCL 的应用程序可以简单的分为一下三个级别，从高到低：

- 应用程序/算法
- API 函数
- kernel 函数

一个 OpenCL 优化问题本质上就是如何最优的使用内存带宽和计算能力，包括：

- 以最优方式使用全局内存，本地内存，寄存器和 cache。
- 以最优的方式发挥计算资源的作用，比如 ALU 和 texture 操作。

这章接下来的部分将会集中在应用程序级别的优化。其他层的优化将会在接下来的章节中讨论。

5.3 对使用 OpenCL 进行初始的评估

在盲目使用 OpenCL 之前，开发者需要先判断当前的应用程序是否适合用 OpenCL 优化。下面是一些适合在 GPU 上加速的程序的典型特点：

- 大量的输入数据
  - 对于少量的输入数据，CPU 和 GPU 之间的开销可能抵消掉了 OpenCL 优化带来性能提升。
- 计算密集
  - GPU 拥有很多计算单元，而且他最高的计算能力，*gflops*，通常比 CPU 高出很多。为了充分利用 GPU，应用程序需要有许多复杂的计算。
- 适合并行化计算

- 工作任务可以被划分为互相独立的小单元，每一个小单元任务的处理并不会影响其他的单元任务。
- 需要使用并行化任务充分利用 GPU 的隐藏内存延迟的能力，这是 GPU 最关键的一个能力。

- 有限的分支控制流
  - GPU 并没有像 CPU 那样，设计地能够处理有效的分支控制。如果使用了大量的条件判断和分支操作，CPU 可能会更合适。

5.4 将 CPU 代码移植到 GPU OpenCL

通常情况下，对于需要转成 OpenCL 的代码，开发者可能已经有一个基于 CPU 版本的参考程序。假设这个程序包含了许多小的功能模块。将每个模块分别对应一个 OpenCL kernel 函数，这样看起来很方便，但是，这种情况下的性能可能不是最优的。需要考虑一下几个事实：

- 在某些情况下，将 CPU 的几个功能模块合并成一个 OpenCL 函数可能会有更好的性能，如果合并能减少 GPU 和内存之间的数据流量。
- 在某些情况下，将一个复杂的 CPU 功能函数模块分解成几个小的简单的 OpenCL kernel，可能会对单个的 kernel 有更好的并行性和对整体有更好的性能。
- 开发者可能需要调整数据结构来适合新的数据流，这种新的数据流方式可以减少整体的数据量。

5.5 GPU 和 CPU 任务的并行

为了充分使用 SOC 的计算性能，当 GPU 执行一个 kernel 函数时，应用程序可能会将指定的任务分配到 CPU 上。当设计这种结构和分配任务时，下面是需要考虑的几点：

- 让 CPU 运行适合在 CPU 上运行的部分，比如分支控制和顺序操作。
- 避免出现 GPU 空闲等待 CPU 执行完成的情况，或者相反情况。
- CPU 和 GPU 之间的数据共享很耗时。所以，试着将一些轻量的 CPU 任务分配给 GPU，尽管这些任务并不是合适 GPU，这样是为了避免数据传输。

5.6 瓶颈分析

识别和分析瓶颈是至关重要的，因为这会使注意力集中到需要优化的区域。瓶颈导致拖延而且经常是应用程序中最慢的部分。不管其他的部分是多么有效率，应用程序的整体性能将会被最慢的那个部分限制，比如瓶颈部分。在瓶颈解决之前，关注其他部分是没有意义的。

5.6.1 识别瓶颈

通常情况，一个 kernel 要么是内存限制要么是计算限制（也可以说是 ALU 限制）。一个简单判别技巧是，按如下方式操作 kernel 代码并将它运行到设备上：

- 如果增加许多计算并不改变性能，那么这不是计算限制。
- 如果加载大量的数据并不改变性能，那么这不是内存限制。

在 4.3 节中讨论的骁龙 profiler 也可以用来识别瓶颈。

5.6.2 解决瓶颈

一旦一个瓶颈被确认了，可以使用不同的策略来解决它：

- 如果是一个 ALU 计算瓶颈的问题，找到方法减少计算复杂度和计算次数，比如在精度要求不高的情况下啊，使用更快的数学函数和内嵌的数学函数，或者使用 16 位浮点数代替 32 位浮点数。
- 如果是一个内存瓶颈的问题，试着提升内存访问效率，比如并行访问/存储，利用本地内存，或者 texture cache（比如，用只读的 image 对象替代缓冲区对象）。使用更短的数据类型来实现在 GPU 和全局内存中之间存储/装载，这样能够节省内存流量。

细节的问题将会在接下来的章节中讨论。

注意： 随着优化的进展，瓶颈可能会改变。如果内存限制被解决了，内存限制就会变成 ALU 限制，或者反之。为了获取最佳的性能，需要进行许多来来回回的迭代。

5.7 API 层面的性能优化

OpenCL 的 API 函数是执行在 CPU 端的，主要是管理资源和控制程序的运行。尽管，一般来说，在计算复杂度方面 API 函数相对于 kernel 的执行是很小的，但是 API 函数不恰当使用将会带来巨大的性能损失。下面是一些建议，能够帮助开发者避免一些常见的陷阱。

5.7.1 合理安排 API 函数的调用

耗时的 API 函数应该放在合适的位置上，避免他们阻塞或者影响 GPU 上的启动工作。一些 OpenCL API 函数需要耗费很长的时间去执行，所以必须在执行的循环外面调用。比如，下面的函数将会消耗大量的时间执行。

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

为了减少在应用程序启动阶段的执行时间，使用 `clCreateProgramWithBinary` 来替代 `clCreateProgramWithSource`。 可以参考 5.7.3 章节获取更多信息。

注意：如果 `clCreateProgramWithBinary` 失败，不要忘记返回然后重新编译源码。坦白来着，这种情况会经常发生，如果 OpenCL 软件进行了不兼容的更新。

- 避免在 NDRange 调用之间，创建和释放内存对象。因为 clCreate{Image|Buffer}的执行时间和请求内存的大小有关系（如果使用了 host_ptr 的话）。
- 如果可能，使用 Android ION 的内存分配。clCreate{Buffer|Image2D}会使用一个 ION 指针来创建内存对象，而不是分配新内存然后进行拷贝。章节 7.4 中讨论了如果使用 ION 内存。
- 在 OpenCL 中，尝试重复使用内存和上下文对象，避免创建新的对象。总的来说，host 端需要做一些轻量级的工作，在启动 GPU kernel 的时候，避免阻塞 GPU 的执行。

5.7.2 使用事件驱动的流水线方式

OpenCL 中入队的 API 函数可能会接收一个事件列表的参数，这个参数表示在当前的 API 函数开始执行之前，列表中的所有的事件必须执行完。同时，这个 API 函数同样可以产生一个时间 ID 来识别他们自己。如果事件列表参数正确的表示了依赖关系，那么 host 端只需简单地将 API 函数和 kernel 提交给 GPU 执行，而不需要操心他们之间的依赖关系和完成情况。通过这种方法，启动一个 API 函数的调用开销将会显著减少，因为软件能够按照最优方式去调度这些函数并且 host 端不需要在 API 函数调用之间进行连接*（换句话说，就是不需要调用完一个后，host 等待他执行完再调用另一个 API 函数）*。因此，通过使用事件驱动的方式使得 API 函数的执行像流水线的方式，这种方法是非常推荐的。另外，开发者主要注意：

- 避免阻塞的 API 调用。一个阻塞的调用会是 CPU 停下来等待 GPU 执行完成，进而在下一次的 clEnqueueNDRangeKernel 的调用之前阻塞了 GPU。阻塞 API 调用通常用在调试过程中。
- 使用回调函数。从 OpenCL1.2 开始，对许多 API 函数进行了增强和修改，API 函数能够接受自定义的回调函数去处理事件，而且因为 host 端能够更灵活的处理事件，这种异步的调用机制会使流水线更有效地执行。

5.7.3 kernel 的装载和编译

实时的装载和编译 kernel 源码是非常耗时的。因为一些参数可能无法提前获取，所以一些应用程序宁愿运行过程中编译源码。如果生成和编译源码并不影响 GPU 执行，那么这是可行的。但是，一般情况下，不建议动态地生成源码。

取代实时编译源码，一个更好的方式是离线编译源码，然后直接使用二进制 kernel。当应用程序装载时，二进制的 kernel 代码也同样被装载。使用这个将会显著降低从磁盘中装载代码的开销。

如果应用程序是用在不同系列的骁龙设备上，那么就需要不同的版本的二进制代码。考虑到兼容性问题，需要注意以下几点：

- 针对某一种 GPU 编译的二进制的代码只能在该 GPU 上使用。如果一个二进制是在 Adreno A530 的 GPU 的设备上编译的，那么这个二进制代码不能被用在 Adreno A540 的 GPU 上。
- 在编译器版本之间，向后的兼容性是可以达到的。新版本的编译一般会支持旧版本的二进制，不过目标 GPU 是要一样的。

如果发现了一个不兼容的二进制 kernel，使用 `clCreateProgramWithSource` 作为一个备用解决方法。

5.7.4 使用有顺序的命令队列

Adreno OpenCL 平台支持乱序的命令队列。然而，在实施乱序的命令队列时需要进行依赖之间的管理，这样会导致很大的开销。Adreno 软件流水命令可以发出一个顺序队列。因此，使用顺序的命令队列是比使用乱序的更好的一种的选择。

6 工作组大小的性能优化

对于许多 kernels 来说，工作组大小的调整会是一种简单有效的方法。这章将会介绍基于工作组大小的基础知识，比如如何获取工作组大小，为什么工作组大小非常重要，同时也会讨论关于最优工作组大小的选择和调整的一般方法。

6.1 获取最大的工作组尺寸

在运行完 clBuildProgram 后，使用下面的 API 函数可以查询设备的最大工作组尺寸。

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

在 clEnqueueNDRangeKernel 中使用的实际工作组尺寸不能超过 maxWorkGroupSize。如果应用程序没有指定工作组大小，Adreno OpenCL 软件可能会选择最大的工作组尺寸。

6.2 需要的和优先的工作组尺寸

每一个 kernel 函数都有他需要或者优先的工作组大小。对于需要的工作组大小，OpenCL 通过下面方法提供给编译器。

- 使用 reqd_work_group_size 属性。

作为需求，reqd_work_group_size(X, Y, Z) 属性会传入一个指定的工作组尺寸。如果指定的工作组大小不能满足将会返回一个错误。

比如，如果要求 16x16 的工作组尺寸：

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

- 使用 work_group_size_hint 属性

OpenCL 会尝试使用这个指定的尺寸，但是不保证真实的大小与指定的一致。比如，提示使用 64x64 工作组尺寸：

```
__kernel __attribute__(( work_group_size_hint (64, 4, 1) ))
```

```
void myKernel( __global float4 *in, __global float4 *out)
{  . . .  }
```

在许多情况下，当工作组尺寸严格指定时，编译器不能保证能编译出最优的机器代码。而且，如果片上寄存器不能满足要求的工作组尺寸时，编译器可能会需要将寄存器溢出到系统的 RAM 内存上。因此，这两种属性并不建议使用，除非必须指定工作组尺寸，kernel 才能运行。

注意：为了交叉编译的兼容性，将 kernel 写成依赖固定工作组的尺寸或者布局，并不是一个好的做好。

6.3 影响工作组最大尺寸的因素

如果没有指定工作组尺寸的属性，一个 kernel 的最大工作组尺寸依赖以下的几个因素：

- kernel 的寄存器使用。一般来说，kernel 越复杂，寄存器使用越多，支持的最大的工作组尺寸越小。过多地使用寄存器的原因如下：
  - 一个工作项中有过多的工作任务。
  - 有控制流
  - 高精度的数学函数（比如，没有使用内部函数或者快速数学运算的编译选项 -fastmath）
  - 本地内存，如果需要分配额外的寄存器暂时存储装载和存储指令中源和目的地址。
  - 私有内存，比如为每一个工作组定义了一个数组
  - 循环展开
  - 内联函数

- 通用寄存器的大小
  - Adreno 低级系列的 GPU 可能有更少的寄存器数量。

- kernel 中的栅栏（Barrier）

如果一个 kernel 没有使用栅栏（barrier），在 Adreno A4x and A5x 系列中，在不用考虑寄存器使用的情况下，工作组最大可以设置为 DEVICE MAXIUMUM。

6.4 没有 barrier 的 kernels

以前地，一个 workgroup 中所有的 work item 要求在同一时间同时驻留在 GPU 上。对于大量消耗寄存器的 kernel，这将会限制他们的最大工作组尺寸，并将会远远小于设备支持的最大工作组尺寸。

从 Adreno A4 系列起，不需要考虑寄存器的使用情况，没有 barrier 的 kernel 就可以有 Adreno 支持的最大工作组尺寸，一般是 1024。对于这种类型的 kernel（没有 barrier）来说，因为不需要 wave 之间进行同步，所以当一个旧的 wave 执行完毕，新的 wave 就可以开始执行了。

在某些情况下，拥有最大的 workgroup 尺寸并不意味着他们有最好的并行性。一个没有 barriers 的 kernel 可能会因为太复杂导致只有很少的 wave 在 SP 上并行执行，这将会导致性能降低。开发者需要继续优化和减少寄存器使用，不考虑从 `clGetKernelWorkGroupInfo` 函数中获取到的最大的 workgroup 尺寸。

6.5 工作组尺寸的调整

这个部分将会介绍一些在选择最优的工作组尺寸和形状时通用的指导准则。

6.5.1 避免使用默认的工作组大小

如果一个 kernel 调用没有指定 workgroup 的尺寸，那么 OpenCL 会用简单的方法找一个能用的工作组尺寸。开发者必须要要意识到，这种默认的工作组尺寸通常不是最优的。有效的做法是，手动尝试使用不同的工作组大小和维度（2D/3D），然后找出最优的一个。

6.5.2 越大的工作组尺寸，越好的性能？

对于许多 kernel 来说这是正确的，因为增加工作组尺寸能够允许更多的 wave 运行在 SP 上，这样能够更好地隐藏延迟和提升 SP 的使用。

然而，对于某些 kernel 来说，增加工作组尺寸可能会导致性能退化。一种情况是，由于不良的数据局部性和访问模式，越大的工作组尺寸将导致越多的 cache 垃圾。这个数据局部性的问题在使用 texture 获取时更加严重，因为 texture cache 比统一的 L2 cache 要小。最终，决定最优的工作大小和维度的本质是 kernel 的数据获取。

6.5.3 固定的 vs. 动态的工作组尺寸

为了不同设备之间的性能兼容性，避免假设一个工作组尺寸能够适合所有的设备，避免对 workgroup 尺寸固定编码。一个指定的工作组大小和维度在一个设备上是最优的，在另一个设备上可能是次优的。因此，给定一个 kernel，建议针对 kernel 能够执行的所有设备统计出不同的 workgroup 尺寸，然后在运行时对每个设备选出一个最优的。

6.5.1 一维 vs 二维 vs 三维 workgroup（1D/2D/3D）

kernel 的维度可以会影响性能。取决于 work item 的数据获取方式，在某些情况下，一个 2D 的 kernel 可能会在 cache 上有更好的数据本地性*（数据在 cache 上）*，导致更好的内存

获取和更好的性能。然而在其他情况下，一个 2D kernel 比 1D 会产生更多的 cache 垃圾。建议尝试使用不同的维度，从而获取最优的性能。

6.6 关于 workgroup 的其他话题

6.6.1 全局的 work size 和填充

OpenCL 1.x 要求一个 kernel 的全局 worksize 必须是 workgroup 尺寸的倍数。如果应用程序指定的 workgroup 尺寸不满足这个条件，那么 `clEnqueueNDRangeKernel` 的函数调用将会返回一个错误。在这种情况下，应用程序可以填充全局 worksize，保证它是用户指定的 workgroup 尺寸的倍数。

注意： OpenCL 2.0 取消的这个限制，而且 global worksize 并不需要必须是 workgroup size 的倍数，这种被叫做非归一化的 workgroup。

理想情况是，workgroup 尺寸的第一个维度是 wave 尺寸的倍数（比如说 32），这样能充分利用 wave 的资源。如果不是这种情况，可以考虑填充 workgroup 的大小来满足这个条件，需要记住，在 OpenCL 1.x 中，全局的 worksize 必须填充（保证是 workgroup 的倍数）。

6.6.2 残酷地寻找

因为 workgroup 尺寸选择的复杂性，经验常常是发现最优大小和维度的最好方法。

一种选择是，在程序开始时，使用一个与实际的工作任务相同复杂度（但是一般使用比较简单的任务）的唤醒功能的 kernel 去动态的寻找最优的 workgroup 尺寸。然后将这个选出来的 workgroup 尺寸用在实际的 kernel 中。很多商业的标准检查程序就是使用的这种方法。

6.6.3 在 workgroup 中避免不均匀的工作负载

一些应用程序可能被写成，在不同工作组中出现不均衡的负载。比如说，基于区域的图像处理的用例中会出现一些区域需要比其他区域多很多处理的情况。这种情况需要避免，因为这会导致性能的不可预测性。另外的，如果单个 workgroup 任务需要太长时间运行的话，会导致上下文切换变的复杂。

解决这个问题的方法是，使用两个阶段处理策略。第一个阶段可能会收集感兴趣的点和为第二阶段准备数据。工作负载越具有确定性，在不同 workgroups 中进行均等的分配将会更简单。

6.6.4 工作组的同步

OpenCL 并不能保证 workgroup 的执行顺序，而且也没有定义一个工作组同步的机制。不建议有需要依赖工作组顺序的程序。

在实际情况下，可以使用 atomic 函数或者其他方法，在 workgroup 之间可以进行有限的同步。比如说，一个应用程序可能分配了一个全局内存对象，这个对象需要被不同的工作组中的 workitem 更新。一个 workgroup 可以管理一个由其他 workgroup 更新的内存对象。通过这种方式，可能会实现有限的工作组同步。

7 内存性能优化

内存优化是最重要也是最有效的 OpenCL 性能优化技术。大量的应用程序是内存限制而不是计算限制。所以，掌握内存优化的方法是 OpenCL 优化的基础。在这章中，将会回顾 OpenCL 的内存模型，然后是最优的实践方法。

7.1 在 Adreno GPU 中的 OpenCL 内存模型

OpenCL 定义了四种内存类型——也就是，global（全局的），local（本地的），constant（常量的），和 private（私有的） 内存，理解这些内存的不同点是基本要求。图 7-1 展示了四种内存概念上的设计图。

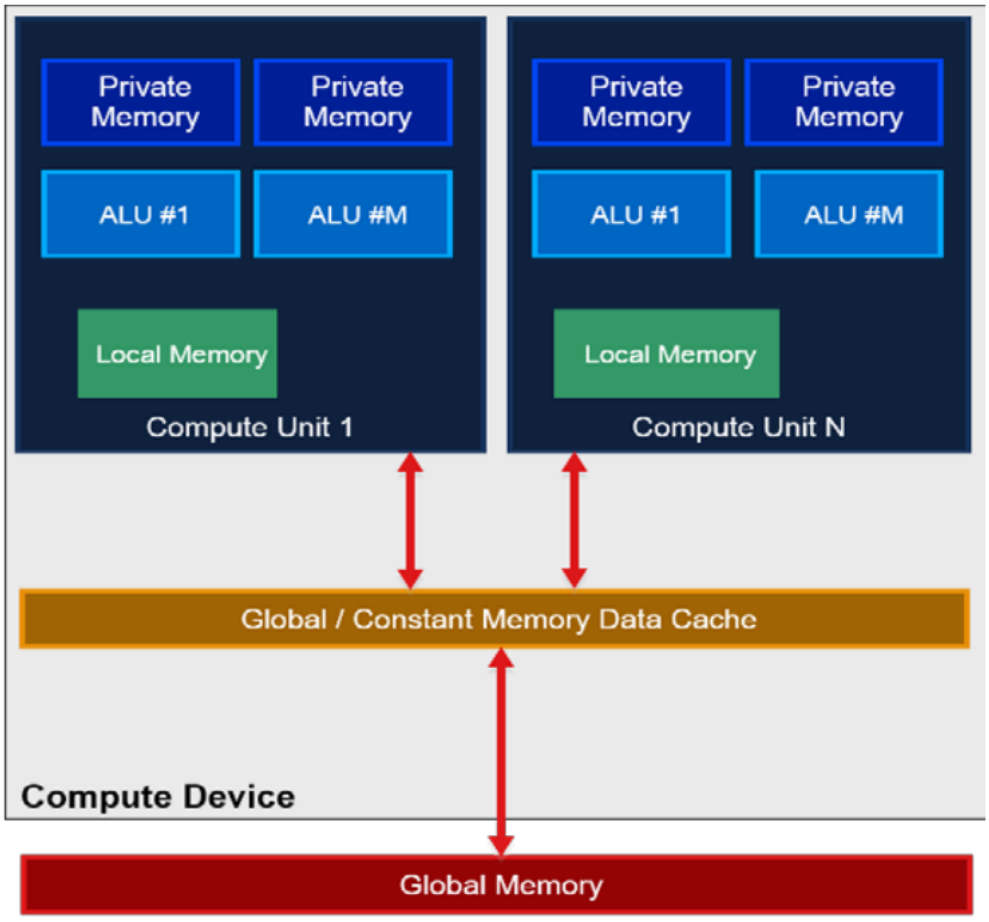

图 7-1 OpenCL 概念上的内存结构

OpenCL 标准只在概念上定义了这些内存，至于如何实现是由厂商自己定义的。物理上的位置可能与概念上的位置有所不同。比如，private 内存对象可能会被放在片外 RAM 上，离 GPU 很远。

表 7-1 列出了在 Adreno GPU 上 4 种内存的定义，以及他们的延迟和物理位置。在 Adreno GPU 上，local 和 constant 内存被放在了片上 RAM 上，比片外 RAM 有更短的延迟。

通常地，对经常要访问到的数据，建议使用 local 和 constant 内存，以便能更好地利用这种短延迟的特性。更多的细节将会在接下来的章节中说明。

表 7-1 Adreno GPU 上的 OpenCL 内存模型

内存类型	定义	相对延迟性	位置
Local	被一个work group 内的所有 work item 共享	中等	片上，SP 内部
Constant	work group 内部所有的 work item 的常量数据	当放到片内是，延迟低。否则，延迟比较大	如果能放下，会放在片内，否则放到系统 RAM 中。
Private	对于 work item 单独拥有	由编译器决定将它分配到哪块内存	在 SP 上放在寄存器或者 local 内存或者系统 RAM(由编译器决定)
Global	可以被所有的 work group 中的所有 work item 访问	大	系统 RAM

7.1.1 local 内存

Adreno GPUs 支持快速的片上 local 内存，不同系列/等级中的 local 内存大小都有所不同。在使用本地内存之前，比较好的做法是，先使用下面的 API 函数，查询一下该设备上每一个 workgroup 可以使用的本地内存有多大。

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

下面是使用本地内存的一些方法：

- 使用本地存储反复使用的数据，或者存储 kernel 函数中两个阶段之间的临时结果。
  - 一种理想的场景是，work items 多次获取相同的内容数据时，而且大于 2 次。
    - 比如说，考虑这种情况，在某种视频处理中，使用对象匹配方法的滑窗方法。 假设每个 work item 使用带 16x16 像素的滑窗处理 8x8 的像素区域，那么相邻的 work item 之间有大量的数据重叠。在这种情况下，本地内存很适合存储像素，减少了多余的数据获取。

- 在不同的 work items 之间使用 barrier 进行同步很可能非常耗时
  - 如果存在 work item 之间的数据交换，比如说，work item A 向本地内存中写数据，work item B 从本地内存中读取数据，因为 OpenCL 的内存一致性模型

并不严格，所以需要一个 barrier 操作。

- Barrier 经常会导致同步延迟，从而阻塞 ALU，导致更低的 ALU 的使用效率。
- 在某些情况下，将数据缓冲到本地内存中可能会需要同步，同步产生的延迟将会抵消使用本地内存带来的性能提升。在这种情况下，直接使用全局内存，避免使用 barrier 可能是更好的选择。

- 使用向量化的装载和存储本地内存
  - 可以使用高达 128 位的向量装载方式，建议装载时 32 位对齐。
  - 在 7.2.2 节中，将会对向量化的存储和装载做更多细节地讨论。

- 让每个 work item 参与本地内存的数据装载，而不是用一个 work item 来完成整个装载任务。
  - 避免使用一个 work item 为整个 work group 装载或者存储所有的本地内存。
- 避免调用函数 async_work_group_copy。对编译器来说，自动生成装载本地内存的最优的代码是非常困难的，所以开发者通过代码手动装载数据到本地内存是一种更好的方式。

7.1.2 constant 内存

Adreno GPU 支持片上的 constant 内存。这种类型的内存在 4 种类型的内存中，有最好的延迟性和更高的性能。constant 内存一般用在以下情况中：

- 使用 constant 修饰符定义的常量或者向量将会被存储在 constant RAM 中。
- 使用 constant 修饰符定义的数组会被存储在 constant RAM 中，如果数组是在程序范围内定义（比如编译器可以决定它的大小）而且在 constant RAM 上有足够的空间。
- kernel 参数中的常量或者向量数据将会被存储在 constant RAM 中。比如，在下面的例子中，coeffs 将会被存储到 constant RAM 中

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

- 使用__constant 修饰的常量和向量变量以及数组，不适合放到 constant RAM 中，将会被放到系统 RAM 中。
- 如果要将 kernel 函数参数中定义的数组装载到 constant RAM 中，必须提供一个叫做 max_constant_size(N)的属性来表示 constant array 的大小，其中 N 表示需要的字节数（bytes）。在接下来的例子中，constant RAM 中的 1024 个字节将会被分配给 foo：

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

指定 max_constant_size 属性是非常重要的。如果没有这个属性，这个数组将会被存储到片外的系统 RAM 上，因为编译器并不知道数组的大小，从而不能将它放到片上 RAM 上。

注意： 这个特性只能支持 16 位和 32 位数据的数组，8 位数据的数组并不支持。另外，如果在 constant memeory 中没有足够的位置分配给数组，那么数组将会被存储到片外的系统 RAM 上。

注意： 对于动态索引的数组和被不同的 work item 访问的数组， constant RAM 并不是最优的。比如，一个 work item 获取索引 0，接下来的另一个 work item 获取索引 20，这种 constant 内存是无效的。在这种情况下，使用 image 对象是一个更好的选择。

7.1.3 private 内存

在 OpenCL 中，private 内存是每一个 work item 私有的，不能被 workgroup 中的其他 work item 访问。物理上，private 内存可以存在在片上寄存器或者片外系统 RAM 上。它的实际存放位置依赖某些因素，下面是一些典型的情况：

- 常量将会被存放在寄存器中，寄存器是最快的内存。
  - 如果没有足够的寄存器，私有变量将会被放到系统 RAM 中。
- 私有的数组将会被存储在：
  - 本地内存，但是这不能保证
  - 片外系统 RAM，如果数组超出了本地内存的容量

将 private 内存存储在片外系统 RAM 上是非常不理想的，因为系统 RAM 比较慢，而且私有内存的访问方式并不能很好地使用 cache，特别是当每个 work item 的 private 内存数量很大的时候。下面是一些建议：

- 避免在 kernel 中定义任何 private 数组。如果可能的话，尝试使用 vector。
- 用 global 或者 local 内存替代 private 数组，并设计成当多个 work item 获取数组元素时可以合并获取。
- 使用向量化进行 private 内存装载/存储，比如，在每次处理中尝试装载/存储高达 128 位的数据。

7.1.4 global 内存

OpenCL 应用程序能够使用两种类型的全局内存对象，buffer 和 image，这两种全局内存对象都是使用片外系统内存。buffer 对象是一个简单的一维数组，image 对象是一种模糊的内存对象，开发者不能假设数据在内存中存储的布局和格式。当一个 image 对象创建时，软件会将数据安排成 GPU 能够更有效的访问形式。使用它们的最有效方式是不同的，这个将会在接下来的章节中讨论。

7.1.4.1 Buffer

Buffer 对象存储一维元素的集合，这些元素可以是数值类型的数据（比如整型，浮点型），向量数据类型或者用户定义的数据结构。一个 buffer 对象可以使用以下的 API 函数创建：

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

Buffer 对象存储在 global 内存中，而且在 Adreno GPUs 中可以通过 L2 cache 访问。在这个函数中，最重要的参数是 cl_mem_flags。 OpenCL 允许这个函数有很多不同的标志，如果选择和结合这些标志对性能提升是非常重要的。下面是一些建议：

- 一些标志会导致额外的内存拷贝。尝试使用 zero-copy 标志，这个标志将会在 7.4 节中讨论。
- 一些标志是针对台式/分离式的 GPU，这些 GPUs 有自己专属的 GPU 内存。
- 使用最精确的标志。总的来说，标志越严格，OpenCL 的驱动就能找到越是适合的配置，从而提高性能。比如说，它可以找到最适合该内存对象的 cache 刷新的规则（写过去，写回来等）。7.4.2 节中有对 cache 规则和对性能影响的详细说明。下面是一些例子：
  - 如果内存只是被 host 端读，使用 CL_MEM_HOST_READ_ONLY
  - 如果内存无法被 host 端访问，使用 CL_MEM_HOST_NO_ACCESS
  - 如果内存只能被 host 端写，使用 CL_MEM_HOST_WRITE_ONLY

7.1.4.2 Image

一个 Image 对象是用来存储一维，二维，或者三维的纹理，帧缓冲，或者一个图像数据，image 对象中的数据布局是不透明的。实际上，这个对象中的内容并不需要必须与一个真实的图像数据有关系。任何数据都可以存储成 image 对象的格式，这样就可以在 Adreno 上使用硬件的 texture 引擎和它的 L1 cache。

一个 image 对象可以使用下面的 API 创建：

```
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)
```

注意，image 对象的 cl_mem_flags 与前面讨论的 buffer 对象中的标志有相同的规则。

Adreno GPUs 支持很多种图像格式和数据类型。从 Adreno A3x GPU 到 Adreno A5x GPU，添加了新的 image 格式和数据类型。用户可以使用函数 clGetSupportedImageFormats 去获取支持的图像格式和数据类型的完整列表。

为了充分使用内存带宽，建议使用数据长度是 128 位的标志对，比如 CL_RGBA/CL_FlOAT, CL_RGBA/CL_SIGNED_INT32 等。

7.1.4.3 使用 image 对象 使用 vs. buffer 对象

相比 buffer 对象，image 对象有以下优点：

- 能使用 texture 引擎硬件。
- 使用 L1 cache
- 内嵌有图像边界的处理。
- 支持大量的图像格式和数据类型的结合，已经在 7.1.4 节“Image”列出，同时支持自动的格式转换。

OpenCL 支持两种简单的滤波，CLK_FILTER_NEAREST 和 CLK_FILTER_LINEAR 。对于 CLK_FILTER_LINEAR，结合适合的 image 类型，能够让 GPU 使用内嵌的 texture 引擎做自动的双线性差值。

举例来说，假设一个图像的类型是 CLK_NORMALIZED_COORDS_TRUE 和 CL_UNORM_INT16，假设图像数据是 2 字节的 unsigned short 类型。read_imagef 的函数调用将会做以下的工作：

- 从 image 对象中读取像素点（这些像素点将被缓冲到 L1 cache 中）
- 在硬件上进行临近像素点差值。
- 转换并归一化到[0,1]。

对于双线性或者三线性差值操作来说，这个很方便。

有时，buffer 对象可能是个更好的选择：

- 更灵活的数据获取方式：
  - image 对象只能允许按像素大小的边界访问，比如，对于 RGBA 的 128 位，32 位/通道。
  - 对于 buffer 对象，Adreno 支持字节寻址访问。比如，在 buffer 对象中，在没有超过 buffer 边界的情况下，128 位数据可以从任何字节地址装载
- 如果 L1 是瓶颈
  - 比如说，出现很严重的 L1 cache 垃圾，使得 L1 cache 访问效率很低。
- 一个 buffer 对象可以在 kernel 中读和写。尽管 image 对象从 OpenCL2.0 开始，也能够读和写，但是由于同步的要求，它的性能很低。

表 7-2 Adreno GPU 上的 Buffer vs. image

特性	Buffer	Image
L2 Cache	Yes	Yes
L1 Cache	No	Yes
支持对象的读和写	Yes	No（在 OpenCL1.x） Yes（在 OpenCL2.x，有同步的要求）

按字节寻址	Yes	No
带内嵌的硬件插值	No	Yes
带内嵌的边界处理	No	Yes
支持 Image 格式和采样	No	Yes

7.1.4.4 同时使用 Image 和 buffer 对象

相比于仅使用 texture 对象或者 buffer 对象，一个更好的方式是同时能够充分使用 UCHE⇔ SP 和 UCHE⇔TPL1⇔SP 这种路径。因为 TPL1 有 L1 cache，将最常用的但是相对数量少的数据存储在 L1 上是一个好的方法。

7.1.4.5 Global 内存 vs. Local 内存

local 内存的一种使用法是，先将数据装载到本地内存，进行数据同步，保证数据已经可用，然后 work group 的 work item 用这些数据进行处理。但是，可能由于以下几个原因，使用全局内存可能比本地内存更好：

- 可能有更好的 L2 cache 的命中率和更好的性能。
- 代码会比使用本地内存时简单，而且会有更大的 work group 尺寸。

7.2 优化内存的装载/存储

在之前的章节中，我们讨论了如何使用不同类型的内存。在这节中，我们将仔细考察，内存的装载/存储对性能影响的一些关键的和普遍的重点。

7.2.1 合并的内存装载/存储

合并装载/存储指的是把多个相邻的 work item 装载/存储的请求合并的能力，这个已经在 3.3.1 节讨论 local 内存访问时提到过。合并访问对于 global 内存的存储/装载也很重要。

除了装载是两路处理（请求和响应），存储是一路处理这种情况以外，合并存储跟读操作的工作方式类似。因此，合并的装载比存储更严格。

在 Adreno GPUs 中，从 Adreno A4x 系列的 GPU 开始，硬件的合并操作逐渐被使能，如表 7-3 所示。Private 内存不支持合并访问。

表 7-3 在 Adreno GPUs 中支持合并访问的 GPUs 系列

装载/存储	Adreno A3x	Adreno A4x	Adreno A5x
global 内存合并装载	No	No	No
global 内存合并存储	No	Yes	No
本地内存合并装载/存储	No	No	Yes

7.2.2 向量化的装载/存储

向量化的装载/存储指的是一个 work item 使用向量化的方法同时装载/存储多个数据。这个与合并访问是不同的，合并访问时是多个 work item。下面是使用向量化装载/存储的一些关键点：

- 对于每一个 work item，建议同时装载一整块数据，比如 64bit/128bit，这样能够更好地利用带宽。
  - 比如，多个 8 位的数据可以手动打包成一个元素（比如 64 位或 128 位），这样可以使用 vloadn 装载，然后通过 as_typeN 函数（比如 as_char16）进行解包。
  - 可参考 9.2.3 中向量化操作的列子。

- 为了更好的优化 SP 到 L2 的带宽性能，装载/存储的内存地址必须是 32 字节对齐的。
- 有两种方法来向量化装载/存储
  - 使用内嵌的函数（`vloadn/vstoren`），这些函数已经在 OpenCL 中被很好的定义了。
  - 另外，指针的强制类型转换同样也可以用来向量化的装载/存储，如下所示：

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

- 建议在使用向量化的装载/存储指令时，最多使用 4 个元素（*vload4/store4*）。因为当向量化装载超过 4 个元素的数据类型时，将会被拆分成多个装载/存储指令，这些指令的操作数不会超过 4 个元素。

- 避免在一个 work item 中装载太多的数据

  - 装载太多的数据可能会使用更多的寄存器，从而导致更小的工作组尺寸，以及性能的损失。在最坏的情况下，会引起寄存器溢出，比如，编译器可能需要使用系统 RAM 来存储变量。

注意：向量化的 ALU 计算同样也可以提高性能，尽管一般来说，没有向量化的内存/存储对性能提升的多。

7.2.3 优化数据类型

数据类型也很重要，因为它不仅影响内存的繁忙程度，也会影响 ALU 的操作。下面是数据类型的一些规则：

1. 检查程序流中的每一个阶段的数据类型，保证在整个流程中每个阶段的数据类型一致。

2. 如果可能，使用更短的数据类型，这样可以减少内存的获取/带宽，而且可以提高可执行的 ALU 数量。

7.2.4 16 位宽的浮点 vs. 32 位宽的浮点

因为 Adreno GPUs 有专用的硬件来计算 half-float 数据类型，所以强烈建议使用 halt-float 来替代 float 数据类型。half ALU 的 gflops 几乎是 full ALUs 的两倍。下面是一些规则：

- 16 位宽的 half 的支持精度是有限的。它仅能够精确的表示很小范围的数据。
  - 比如，它只能够精确地表示在[0,2048]整数范围内的数据。
- 如果 half 的数据计算会导致不能接受的精度损失，那需要将 half 转成 float 精度。但是在存储时，仍然使用 half 数据类型。

7.3 原子函数

在 OpenCL 中定义了大量的本地的和全局的原子函数，而且 Adreno GPU 在硬件上就支持。下面是使用原子函数的一些规则：

1. 避免一个或者多个工作组频繁去更新一个单独全局原子内存地址，因为原子操作是顺序操作，而且他们的性能比如并行操作。

2. 首先应该尝试使用本地原子操作，然后再对全局内存进行一次原子更新。

7.4 0 拷贝

Andreno OpenCL 提供了一些机制来避免可能在 host 端发生耗时的内存拷贝。因为与内存对象创建的方式有关，所以有一些不同的方法来避免耗时的拷贝。

7.4.1 使用 map 替换拷贝

假设 OpenCL 应用程序对数据流有完全的控制权，比如目标和源内存对象的创建都是由 OpenCL 应用程序管理的。这是一种最简单的情况，可以使用以下几个步骤避免内存拷贝：

- 当创建一个 buffer/image 对象时，使用 `CL_MEM_ALLOC_HOST_PTR` 标志，然后执行以下的步骤：
  - 首先在 `clCreateBuffer` 函数中设置 `cl_mem_flags`：

```
cl_mem Buffer = clCreateBuffer(context,
                CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR,
                sizeof(cl_ushort) * size,
                NULL,
                &status);
```

- 然后使用 map 函数，给 host 返回一个指针

```
cl_uchar *hostPtr = (cl_uchar *)clEnqueueMapBuffer(
                        commandQueue,
                        Buffer,
                        CL_TRUE,
                        CL_MAP_WRITE,
                        0,
                        sizeof(cl_uchar) * size,
                        0, NULL, NULL, &status);
```

- host 使用指针 hostPtr 来更新 buffer
  - 比如，host 可以读取相机数据或者从硬盘中的数据填充到 buffer 中。
- 取消对象映射

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

- 然后这个对象可以被 OpenCL 的 kernel 函数使用

在这种场景下，CL_MEM_ALLOC_HOST_PTR 是避免数据拷贝的唯一方法。使用其他的标志，比如 CL_MEM_USE_HOST_PTR 和 CL_MEM_COPY_HOST_PTR，为了 GPU 能访问这些数据，驱动将必须要进行额外的内存拷贝

7.4.2 避免不是由 OpenCL 分配的对象的内存拷贝

7.4.2.1 ION 内存扩展

如果内存对象是在 OpenCL API 范围外被初始化创建的，而且是使用 ION/Gralloc 分配的，那么可以使用 cl_qcom_ion_host_ptr 这个扩展来创建 buffer/image 对象，这样会将 ION 内存映射到 GPU 可访问的内存，而且不会发生额外的拷贝。

注意：如果有需要，可以提供一个详细的简单的代码来阐述通过 QTI 扩展来使用 ION 内存而避免内存拷贝的方法。

7.4.2.2 QTI ANB(Android native buffer）扩展

在许多相机和音频处理的使用案例中，ANB（由 gralloc 分配的）必须共享。因为 buffer 是基于 ION 的，所以共享是有可能的。然而，为了使用 ION，开发者需要从从 buffer 中获取内部的句柄，这个需要访问 QTI 的内部头文件。`cl_qcom_android_native_buffer_host_ptr` 扩展提供了一个更直接的方式与 OpenCL 共享 ANB，并且不需要访问 QTI 的头文件。这样 ISV 和其他第三方的开发者能够在 ANB 上实现 0 拷贝。

注意：如果有需要，可以提供一个例子来阐述 `cl_qcom_android_native_buffer_host_ptr` 扩展的使用。

7.4.3 使用标准的 EGL 扩展

`cl_khr_egl_image` 扩展能够从 EGL 图像中创建一个 OpenCL image 对象。这样做的主要好处是：

- 这是一种标准，使用这种技术书写的代码能够最大程度上在其他支持 EGL 的 GPU 上工作。
- 由于 EGL/CL 扩展（`l_khr_egl_event` 和 `EGL_KHR_cl_event`）的设计，所以使用 EGL/CL 扩展的程序可能会实现更有效的并行。
- 对于 YUV 处理，使用 `EGL_IMG_image_plane_attribs` 扩展会更简单。

7.5 提高 cache 的利用

为了有更好的利用 cache，必须要遵守下面的一些规则：

- 检查 cache 的垃圾和 cache 的使用率。Snapdragon Profiler 可以提供 cache 访问的信息，比如装载/存储的字节数，cache 命中率/没命中率。
  - 如果装载进 UCHE 的字节数比 kernel 期望的多很多，那么可能存在 cache 垃圾。
  - 比如 L1/L2 命中/没命中率等指标能够提供 cache 的使用情况。

- 通过以下方法避免 cache 垃圾
  - 调整 workgroup 大小，比如减少 workgroup 的大小。
  - 改变访问模式，比如，改变 kernel 的维度。
  - 如果使用 loops 时产生 cache 垃圾，在循环中添加 `automics` 或者 `barrier` 可能会减少垃圾。

7.6 CPU 的 cache 操作

对于可以用 cahce 缓存的内存对象，OpenCL 驱动需要在合适的时间更新 cache 数据或者使 cache 数据无效。这能够保证当 CPU 和 GPU 尝试访问数据时，他们看到的是最新的数据拷贝。比如，当主 CPU 为了读数据，映射一个 kernel 的输出 buffer 时，那必须使 CPU cache 中的数据无效。OpenCL 的驱动程序有非常复杂的 CPU cache 管理机制，通过对每一块内存对象偏移的可视化跟踪和尽可能推迟额外的操作，尝试使用最少数量的 cache 操作。比如，在一个 kernel 启动前，可能会对输入 buffer 的 CPU cache 的进行刷新。

CPU cache 操作会有可以测量的损失，可以通过 `clEnqueueNDRangeKernel` 中的 `CL_PROFILING_COMMAND_QUEUED` 与 `CL_PROFILING_COMMAND_SUBMIT` 之间的差值查看，如图 4-1 显示的那样。在某些情况下，`clEnqueueMapBuffer/Image` 和 `clEnqueueUnmapBuffer/Image` 的执行时间可能会增加。一个 CPU cache 操作的耗时通常会随着内存对象的大小线性增加。

为了最小化 CPU cache 操作的耗时，必须对应用程序的流程进行仔细地安排，避免在 CPU 和 GPU 之间来回切换处理。而且，应用程序分配内存对象时，在 CPU 和 GPU 之间来回被访问的数据和只有一种访问的数据需要放在不同的内存对象里。

内存对象创建时必须使用 CPU cache 机制，这个机制需要跟他们的用途合适。当为 buffer 对象或者 image 对象分配内存时，驱动将会选择 CPU cache 机制。默认的 CPU cache 机制是 write-back。然而，如果使用了 `CL_MEM_HOST_WRITE_ONLY` 或 `CL_MEM_READ_ONLY` 标志，驱动将会认为应用程序不准备使用 host CPU 来读取数据。在这种情况下，CPU 的 cache 机制被设置为 write-combine。

对于外部的分配内存对象，比如使用 ION 和 ANB 机制，应用程序对 CPU cache 机制更直接的控制。将这些对象引入到 OpenCL 时，应用程序必须要正确设置 CPU cache 机制。

7.7 使用 SVM

Adreno A5x GPUs 支持粗粒度的 SVM，这是 OpenCL2.0 完整版简介中一个关键特性。使用 SVM 时，host 和 devie 的内存地址是相同。在 OpenCL2.0 中的 SVM 特性能够方便地实现 host 端和 device 端之间的内存共享，能通过在 OpenCL 设备上访问 host 指针。

对于粗粒度的 SVM，在同步时（map/unmap）,host 或者 devices 上访问内存将会被限制了. 对于需要在 host 端和 device 端都进行处理的数据结构指针 这类应用程序可以很好的利用这个特性。

7.8 减少电源/热量消耗的最好的经验

对于移动的应用程序，能源和热量是一个主要的考虑因素。高性能的应用程序可能没有最好的电源/热量的性能，而且反之亦然。因此，理解电源/热量和性能的要求是很重要的。

下面是一些在 OpenCL 上减少电源和热量消耗的几个提示：

- 使用所有的方法避免内存拷贝。比如，使用 ION 内存来实现 0 拷贝，而且，在使用 函数 `clCreateBuffer` 创建 buffers 使用标志 `CL_MEM_ALLOC_HOST_PTR`。另外，避免使用 OpenCL APIs 进行数据拷贝。
- 最小化 host 和 device 之间的内存传输。可以通过以下方式实现，在 constant 内存或者 local 内存存储内存数据，使用更短的数据结构，降低数据精度，剔除 private 内存的使用等。
- 优化 kernel 和提高他们的性能。kernel 运行的越快，消耗的能量和电量越少。
- 最小化软件的开销。比如，使用事件驱动的流程来减少 host 和 device 的通信开销。避免创建太多的对象，避免在 kernel 执行之间创建或者释放对象。

8 kernel 性能优化

这章将会说明一些 kernel 优化的小技巧。

8.1 kernel 合并或者拆分

一个复杂的应用程序可能包含很多步骤。对于 OpenCL 的移植性和优化，可能会问需要开发有多少个 kernel。这个问题很难回答，因为这涉及到很多的因素。下面是一些准则：

- 内存和计算之间的平衡。
- 足够多的 wave 来隐藏延迟。
- 没有寄存器溢出。

上面的要求可以通过执行以下操作实现：

- 如果这样做能够带来更好的数据并行，将一个大的 kernel 拆分成多个小的 kernel。
- 如果内存的流量能够减少而且同样能保证并行性，可以将多个 kernel 合并成一个 kernel，例如 workgroup 的尺寸能够足够地大。

8.2 编译选项

OpenCL 支持一些编译选项，参考文献的《The OpenCL Specification》的 5.6.4 节中进行了定义。编译选项可以通过 APIsclCompileProgram 和 clBuildProgram 传递。多个编译选项可以结合，如下所示。

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

通过这些选项，开发者能够针对他们自己的需求使能某些功能。比如，使用-cl-fast-relaxed-math，kernel 会编译成使用快速数学函数而不是 OpenCL 标准函数，每一个 OpenCL 的说明中 OpenCL 标准函数都有很高的精度要求。

8.3 一致性 vs. 快速 vs. vs. 内部的数学函数

OpenCL 标准在 OpenCL C 语言中定义了许多数学函数，默认情况下，因为 OpenCL 规范说明书的要求，所有的数学函数都必须满足 IEEE 754 单精度的浮点精度数学要求。Adreno GPU 有一个内嵌的硬件模块，EFU(elementary function unit 基本函数单元)，来加速一些初级

的数学函数。对于许多 EFU 不能直接支持的数学函数，可以通过结合 EFU 和 ALU 操作来优化，或者通过编译器使用复杂的算法来模拟进行优化。表 8-1 展示了 OpenCL-GPU 数学函数的列表，并按照他们的相对性能来分类的。使用更好性能的函数是个较好的方法，比如使用 A 类中的函数

表 8-1 OpenCL 数学函数的性能（符合 IEEE 754 标准）

类别	实现	函数（可参考 OpenCL 标准获取更多细节）
A	仅简单使用 ALU 指令	ceil, copysign,fabs, fdim, floor, fmax, fmin, fract, frexp, ilogb, mad, maxmag, minmag, modf, nan, nextafter, rint, round, trunk
B	仅使用 EFU，或者 EFU 机上简单的 ALU 指令	asin, asinpi, atan, atanh, atanpi, cosh, exp, exp2, rsqrt, sqrt, tanh
C	ALU,EFU,和位操作的结合	acos, acosh, acospi, asinh, atan, atan2pi, cbrt, cos, cospi, exp10, expml, fmod, hypot, ldexp, log, log10, loglp, log2, logb, pow, remainder, remquo, sin, sincos, sinh, sinpi
D	复杂的软件模拟	erf, erfc, fma, lgamma, lgamma_r, pown, powr, rootn, tan, tanpi, tgamma

另外，如果应用程序对精度不敏感的话，开发者可以选择使用内部的或者快速的数学函数来替代标准的数学函数。表 8-2 总结了使用数学函数时的 3 个选项。

- 使用快速函数时，在调用函数 `clBuildProgram` 时使能`-cl-fast-relaxed-math`。
- 使用内部的数学函数：
  - 许多函数有内部实现，比如：`native_cos`, `native_exp`,`native_exp2`, `native_log`, `native_log2`, `native_log10`, `native_powr`,`native_recip`, `native_rsqrt`, `native_sin`, `native_sqrt`, `native_tan`;
  - 下面使用内部数学函数的例子：
    - 原始的： `int c = a/b ; // a` 和 `b` 都是整数。
    - 使用内部指令：

```
int c = (int)native_divide((float)(a)),(float)(b));
```

表 8-2 基于精度/性能的数学函数选择

数学函数	定义	怎么使用	精度要求	性能	典型应用
标准	符合 IEEE754 单精度浮点要求	默认	严格	低	科学计算，对精度敏感的情况下
快速	低精度的快速函数	kernel 编译选项 `-cl-fast-relaxed-math`	中等	中等	许多图像，音频和视觉的用例中
内部	直接使用硬件计算	使用 native_function 替换 kernel 中的函数	低，与供应商有关	高	对精度损失不敏感的情况下的图像，音频，和视觉用例中

8.4 循环展开

循环展开通常是一个好方法，因为它能够减少指令执行的耗时从而提高性能。Adreno 编译器通常能基于试探法自动地将循环展开。然而，有时候编译器选择不将循环完全展开，因为基于考虑到，寄存器的分配预算，或者编译器因为缺少某些信息不能将它展开等因素。在这些情况下，开发者可以给编译器一个提示，或者手动的强制将循环展开，如下所示：

- kernel 可以使用 `__attribute__((opencl_unroll_hint))` 或者 `__attribute__((opencl_unroll_hint(n)))` 给出提示。
- 另外，kernel 可以直接使用`#pragma unroll` 展开循环。
- 最后一个选择是手动展开循环。

8.5 避免分支

一般地，当在同一个 wave 中的 work item 有不同的执行路径时，那么 GPU 就不是那么高效率。对于某些分支，一些 work time 必须执行，从而导致较低的 GPU 使用率，就像图 8-1 所示。而且，像 if-else 的条件判断代码通常会引起硬件的控制流逻辑，这个是非常耗时的。

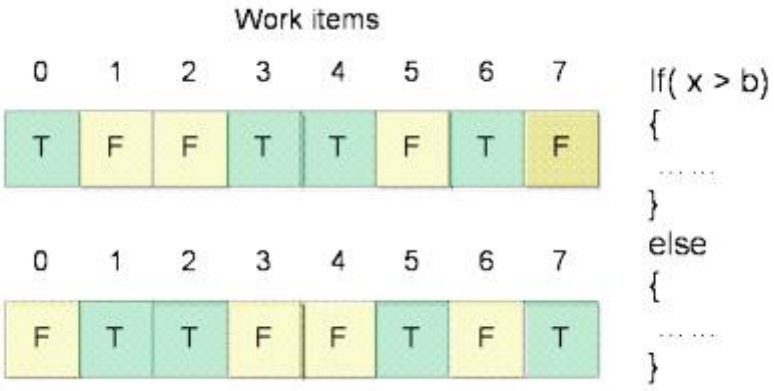

图 8-1 绘图表示出现在两个 wave 中的分支情况

有一些方法可以用来避免或者减少分支和条件判断。在算法层面，一种方法是将进入同一分支的 work item 组成一个不可分的 wave。在 kernel 层面，一些简单的分叉/条件判断可以转变成快速的 ALU 操作。在 9.2.6 节中一个例子中，有耗时的控制流逻辑的一个三元操作被转变成一个 ALU 操作。其他的方式是使用类似于 select 函数，这个可能会使用快速的 ALU 操作来替代控制流逻辑。

8.6 处理图像边界

许多操作可能会获取图像边界外的像素点，比如滤波，变换等。为了更好地处理边界，可以考虑下面的选择：

- 如果可能的话，对图像进行扩边。
- 使用带有合适的采样器的 image 对象（texture 引擎会自动处理这个）
- 编写单独的 kernel 函数去处理边界，或者让 CPU 处理边界。

8.7 32 位 vs. 64 位 GPU 内存访问

从 Adreno A5X GPU 开始，64 位操作系统逐渐成为主流，而且许多的 Adreno GPU 支持 64 位操作系统。64 位操作系统中最重要的改变是内存空间将能完全覆盖 4GB，而且 CPU 支持 64 位指令集。

当 GPU 可以获取 64 位内存空间时，它的使用将会引起额外的复杂性，而且可能会影响性能。

8.8 避免使用 size_t

64 位的内存地址在许多情况下会提升编写 OpenCL kernel 的复杂度，开发者必须要小心。强烈建议避免在 kernels 中定义 size_t 类型的变量。对于 64 位操作系统，在 kernel 中定义成 size_t 的变量可能会被当成 64 位长度的数据。Adreno GPUs 必须使用 32 位寄存器来模拟 64 位。因此，size_t 类型的变量会需要更多的寄存器资源，从而因为可用的 wave 变少和更小的 workgroup 大小导致性能退化。所以，开发者应该使用 32 位或者更短的数据类型来替代 size_t.

对于 OpenCL 中返回 size_t 的内嵌函数，编译器会根据它所知道的信息尝试推导并限制数据范围。比如， `get_local_id` 返回的数据类型为 size_t，尽管 `local_id` 永远不会超过 32 位。在这种情况，编译器尝试使用一个短的数据类型来替代。但是，更好的方法是，给编译器提供关于数据类型的最充分的信息，然后编译器可以产生更好的优化代码。

8.9 一般的内存空间

OpenCL 2.0 介绍了一个新的特性，叫做*一般性的内存地址空间*，在这个地址空间中，指针不需要指定它的地址空间，在 OpenCL2.0 之前，指针必须指定它的地址空间，比如指定为是 local，private，或者 global。在一般性的地址空间中，指针可以动态地被指定为不同的地址空间。

这个特性降低了开发者的代码基础而且能重复使用已经存在的代码，使用一般性的内存地址空间会有轻微的性能损失，因为 GPU SP 硬件需要动态的指出真正的地址空间。如果开发者清楚知道变量的内存空间，建议清晰地定义内存地址。这将会减少编译器的歧义，从而会有更好的机器代码进而提升性能。

8.10 其他

还有很多其他的优化技巧，这些技巧看起来很小，但是同样可以提高性能，这些技巧如下所示：

- 已经计算过的数据，而且不会在 kernel 中被改变的。
  - 如果一个数据可以在外面（*host 端*）计算好，那么放到 kernel 中计算会很浪费。
  - 已经计算好的数据可以通过 kernel 参数传递给 kernel，或者用`#define` 的方式。

- 使用快速的整型的内嵌函数。使用 mul24 计算 24 位的整型乘法，和使用 mad24 计算 24 位的整型乘加。
  - Adreno GPU 的内部硬件支持 mul24，而 32 位的整型乘法需要用更多的指令模

拟。

- 如果是在 24 位范围内的整型数据，使用 mul24 会比直接使用 32 位的乘法更快。

- 减少 EFU 函数
  - 比如，像 `r=a/select(c,d,b<T)` 这样的代码（其中 `a,b` 和 `T` 是浮点变量，`c` 和 `d` 是常数），可以写成 `r = a * select(1/c,1/d,b<T)`，这样会避免 EFU 中倒数函数，因为 `1/c` 和 `1/d` 可以在编译器编译阶段计算出来。

- 避免除法操作，特别是整型的除法。
  - 整型的除法在 Adreno GPUs 上是极其耗时的。
  - 不使用除法，可以使用 `native_recip` 计算倒数，像 8.3 节描述的那样。
  -

- 避免整型的模操作，这个也很耗时。

- 对于常数的数组，比如说查找表，滤波 tap 等，在 kernel 的外面进行声明。

- 使用 mem_fence 函数来分开或者组合代码段。
  - 编译器会从全局优化的角度，使用复杂的算法产生最优的代码。
  - mem_fonce 可以用来阻止编译器混排和混合前面或者后面的代码。
  - mem_fonce 可以让开发者单独操作代码的某个部分来进行优化和调试。

- 使用位移操作替换乘法。

9 OpenCL 优化用例的学习

在这一章中，将会用一些例子来展示如何使用之前章节中讨论的技术来进行优化。除了一些小的简单代码片段的展示外，还有两个熟知的图像滤波处理，Epsilon 滤波和 Sobel 滤波，将会使用之前章节中讨论的方法进行一步一步地优化。

9.1 应用程序的代码样本

9.1.1 提升算法

这个例子说明了如何简化代码来提升性能。给定一张图片，对它进行 8x8 的 box 模糊滤波。

优化前的原始 kernel 代码：

```
__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
}
```

为了减少 texture 访问，将上面的 kernel 函数分成两个步骤。第一个步骤，每个 work item 计算一个 2x2 的平均值，并将结果存放到临时的 image 中。第二个步骤使用临时的 image 进行最终的计算。

修改后的 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
}
```

修改后的算法每个 work item 只获取 20（4+16）次 image buffer，明显比原始的 64 次 read_imagef 获取次数少很多。

9.1.2 向量化的装载和存储

这个例子说明如何在 Adreno GPUs 上使用向量化的装载和存储而更好地利用了带宽。

优化前的原始代码：

```
__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];
}
```

修改后的代码

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

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

新的 kernel 使用 float4 来实现向量化的装载/存储。因为向量化，新的 kernel 的 global work size 是之前的 kernel 的 1/4.

9.1.3 使用 image 替代 buffer

这个例子是对给定 500 万对向量计算每一对向量的点积。原始代码是使用 buffer 对象，修改完后，使用 texture 对象（read_imagef）来提升常用数据的访问性能。这是一个简单的例子，但是使用到的技术可以适用到许多类似情况，这些情况下 buffer 对象的访问不如 texture 对象访问有效。

优化前的原始 kernel 函数	优化后的 kernel 函数

__kernel void DotProduct(__global const float4 *a, __global const float4 *b,__global float *result){ // a and b contain 5 million vectors each
// Arrays are stored as linear buffer in global memory
result[gid] = dot(a[gid], b[gid]);
}
	__kernel void DotProduct(__read_only image2d_t c, __read_only image2d_t d, __global float *result){
	// Image c and d are used to hold the data instead of linear buffer
	// read_imagef goes through the texture engine
	int2 gid = (get_global_id(0), get_global_id(1));
	result[gid.y * w + gid.x] = dot(read_imagef(c, sampler, gid), read_imagef(d, sampler, gid));
	}

9.2 Epsilon 滤波

Epsilon 滤波被广泛用在图像处理中用来减少蚊式噪音，这种噪音是一种发生在高频区域（比如图像的边）的缺陷。这种滤波是非线性的和基于点式的低通滤波，支持空间的变化，而且只有特定阈值的像素点会被滤掉。

在这个例子中，Epsilon 滤波仅被用在 YUV 图像中的 Y 通道，因为噪音经常在这个通道可见。另外，假设 Y 通道是连续存储的（NV12 格式），与 UV 通道的存储分开。Figure9-1 阐述了滤波算法的两个基本步骤：

- 对于一个要滤波的像素点，计算 9x9 的范围内所有像素点到中心像素点的绝对差值。
- 如果绝对值小于一个阈值，这个邻居像素点就用来平均。注意，阈值通常是在程序中预先定义的一个常量。

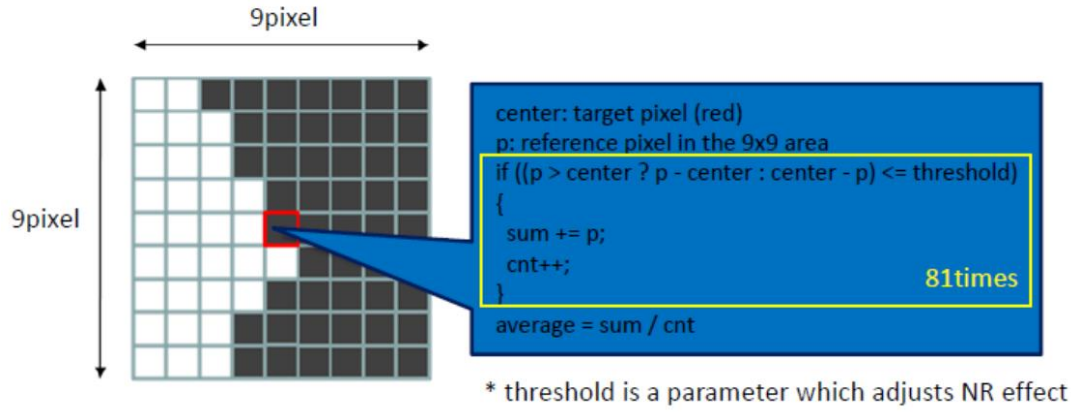

图 9-1 Epsilon 滤波算法

9.2.1 第一次的实现

这个程序计算的 YUV 图片的分辨率是 3264x2448(宽度是 3264，高度是 2448），每个像素点是 8 位的数据宽度。这里说的性能数据是基于 Snadragon 810(MSM8994,Adreno 430)的高性能模式。

下面是第一次实现的参数和策略：

- 使用 OpenCL 的 image 对象替代 buffer 对象。
  - 使用 image 替代 buffer 能够避免边界检测和充分利用 Adreno GPUs 中的 L1 cache。
- 使用 `CL_R | CL_UNORM_INT8` 图片格式/数据类型
  - 单个通道，比如这里的 Y 通道，而且读取到 SP 的像素点，都被 Adreno GPUs 内建的 texture 管道归一化到[0,1]。
- 每个 work item 处理一个输出的元素。
- 使用 2D 的 kernel，global 的 worksize 大小被设置为[3264,2448].

在这个实现中，每个 work item 需要获取 81 个浮点类型的像素点。在 Adreno A430 上这个实现的性能数据作为进一步优化的基准。

9.2.2 数据包优化

通过比较计算量和数据装载量，这个例子明显是内存瓶颈。因此，主要的优化是如何提升数据的装载效率。

首先需要注意到，使用 32 位的浮点来表示像素点是对内存的浪费。对许多图像处理算法来说，8 位或者 16 位的数据类型就够用了。因为 Adreno GPUs 对 16 位浮点类型有内嵌的硬件支持，比如 half 或者 fp16，所以可以使用下面的优化方法：

- 使用 16 位浮点数据类型来替换 32 位浮点
  - 现在每个 work item 获取 81 个 half 数据

- 使用 CL_RGBA| CL_UNORM_INT8 图像类型或者数据类型
  - 使用 CL_RGBA 去装载 4 个通道能更好的的利用 TP 的带宽。
  - 使用 read_imagef 替代 read_imageh。TP 会将数据自动转换成 16 位 half 数据。
- 每个 work item
  - 每行使用 3 个 half4 向量
  - 输出一个已经被处理的像素点
  - 对于每一个输出的像素点，访问的内存数是 3x9 = 27 （half4）
- 性能提升 1.4x

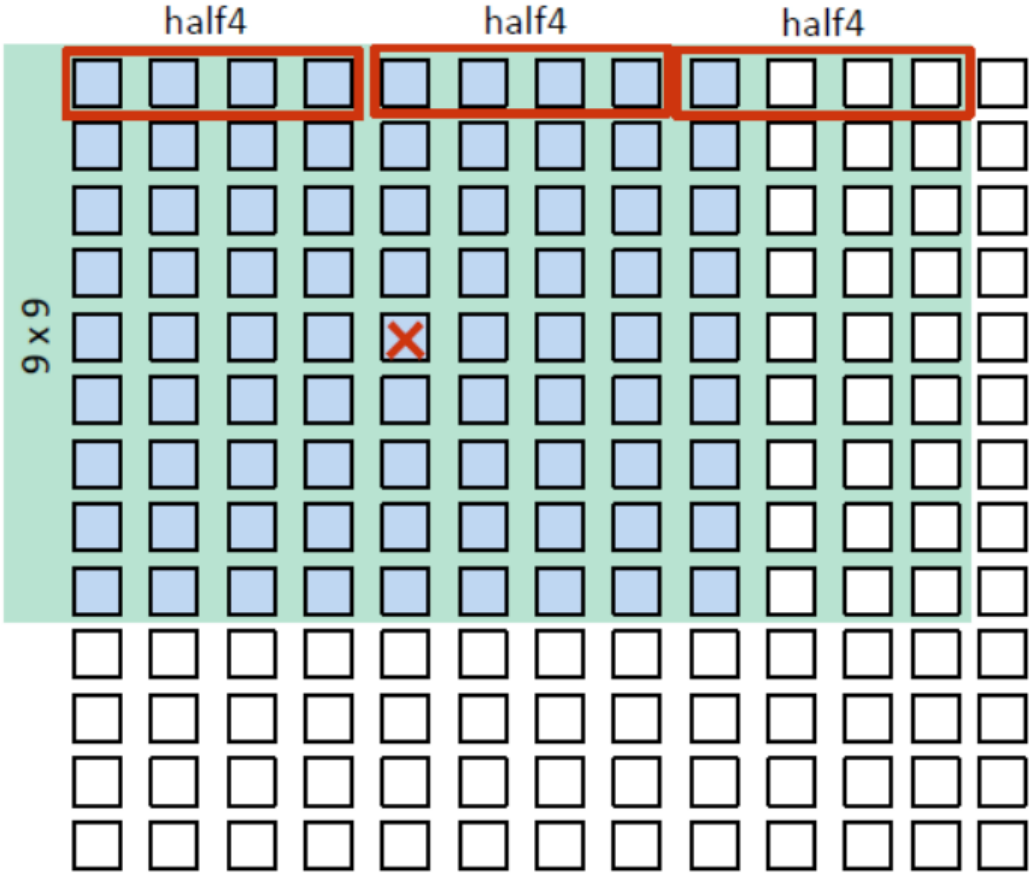

图 9-2 使用 16 位 half 类型打包数据

9.2.3 向量化装载/存储优化

在之前的步骤中，只要计算一个输出像素，就有很多邻居像素点需要装载。可以通过额外装载一些像素点，那么久可以滤波更多的像素点，如下所示：

- 每个 work item
- 每一行读 3 个 `half4` 的向量
  - 输出 4 个像素
  - 每个输出像素点需要获取的内存数量是：3x9/4 = 6.75(`half4`)
- 全局的 work size： （width/4）x height
- 对每一行循环展开。
- 在每一行内，使用活动窗口的方法。

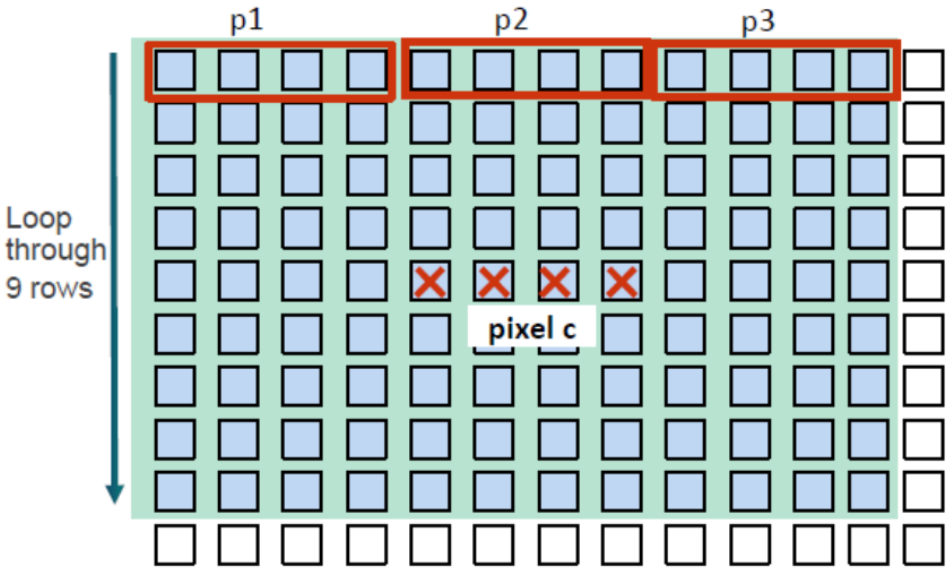

图 9-3 每一个 work item 滤波更多的像素点

图 9-3 阐述了通过装载额外的像素点，处理多个像素的方法。以下是几个具体的步骤：

```
Read center pixel c;  //读取中心像素点 c
For row = 1 to 9, do:  //从行 1 到行 9，执行以下操作
read data p1;        //读取数据 p1
Perform 4 computation with pixel c;  //对 c 执行 1 次计算
read data p2;        //读取数据 p2
Perform 4 computations with pixel c;  //对 c 执行 4 次计算
read data p3;        //读取数据 p3
Perform 4 computations with pixel c;  //与 c 执行 4 次计算
end for             //结束循环
write results back to pixel c.  //将结果写回中心像素点
```

通过这些步骤，性能比基准性能提高了 3.4x。

9.2.4 进一步提升每个 work item 的工作量

提升每个 work item 的工作量，性能可能会有所提升。下面是一些可选的操作：

- 读取多个 `half4` 向量，提升输出的像素个数到 8 个。

- 全局的 work size ： width/8 x height
- 每个 work item
  - 每一行读 4 个 half4 向量。
  - 输出 8 个像素
  - 每个输出像素点需要访问的内存次数为： 4x9/8 = 4.5 (`half4`)

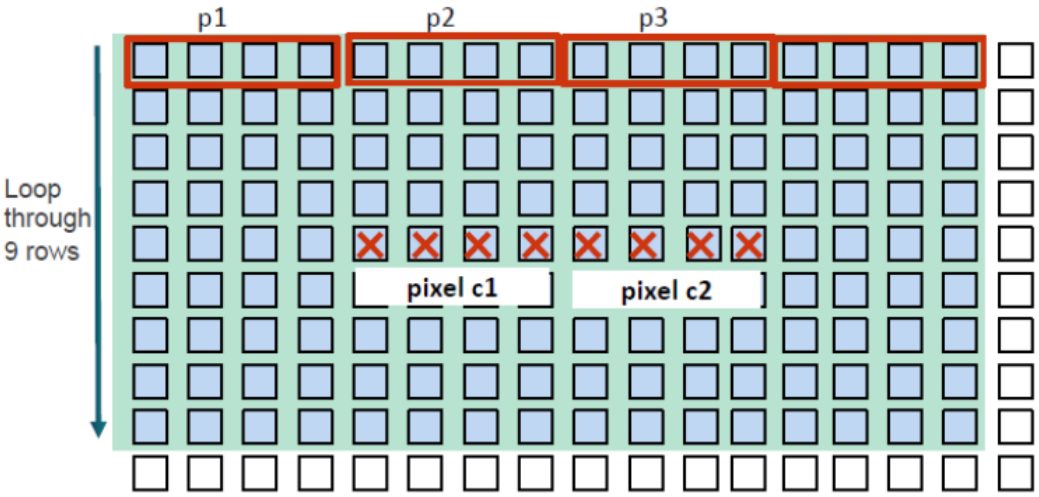

图 9-4 每个 work item 处理 8 个像素点

这些改变仅提升了很少的性能（仅提升 0.1x），下面是为什么这么做不起作用的原因：

- 在 cache 命中率上没有很大的改变，这个在之前的步骤上已经做的很好了。
- 更多的寄存器被使用，导致了更少的 wave，这将会损失平行性和延迟的隐藏性。

为了实验的目的，下面的方法可以装载更多的像素点：

- 读取更多的 `half4` 的向量，提升输出像素点的个数到 16.
- 全局的 work size： width/16 x height

图 9-5 表示了每个 work item 会执行以下几个步骤：

- 每行读 6 个 `half4` 个向量。
- 输出 16 个像素点
- 每个输出像素点需要访问的内存次数为：6x9/16 = 3.375(`half4`)

经过这些改变后，性能从 3.4x 退化到 0.5x。装载更多的像素点进入 kernel 中会引起寄存器溢出，这将会严重地损害性能。

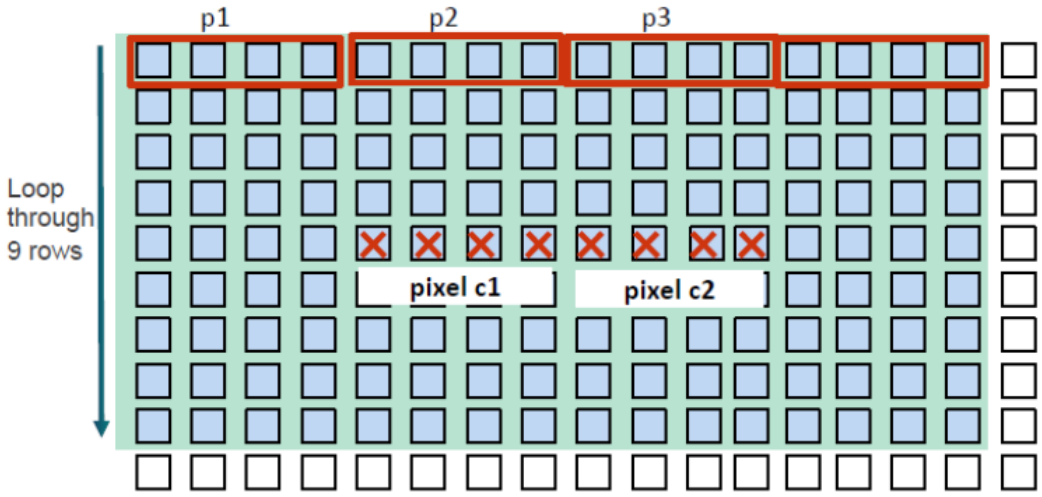

图 9-5 每个 work item 处理 16 个像素

9.2.5 使用本地内存优化

本地内存比全局内存有更短的延迟，因为本地内存是片上内存。一个选择是，将像素点装载进本地内存，避免重复从全局内存中加载。而且，对于要处理的中心元素，由于 9x9 范围的滤波，所以它周围的元素也需要，所以如图 9-6 所示，装载进内存。

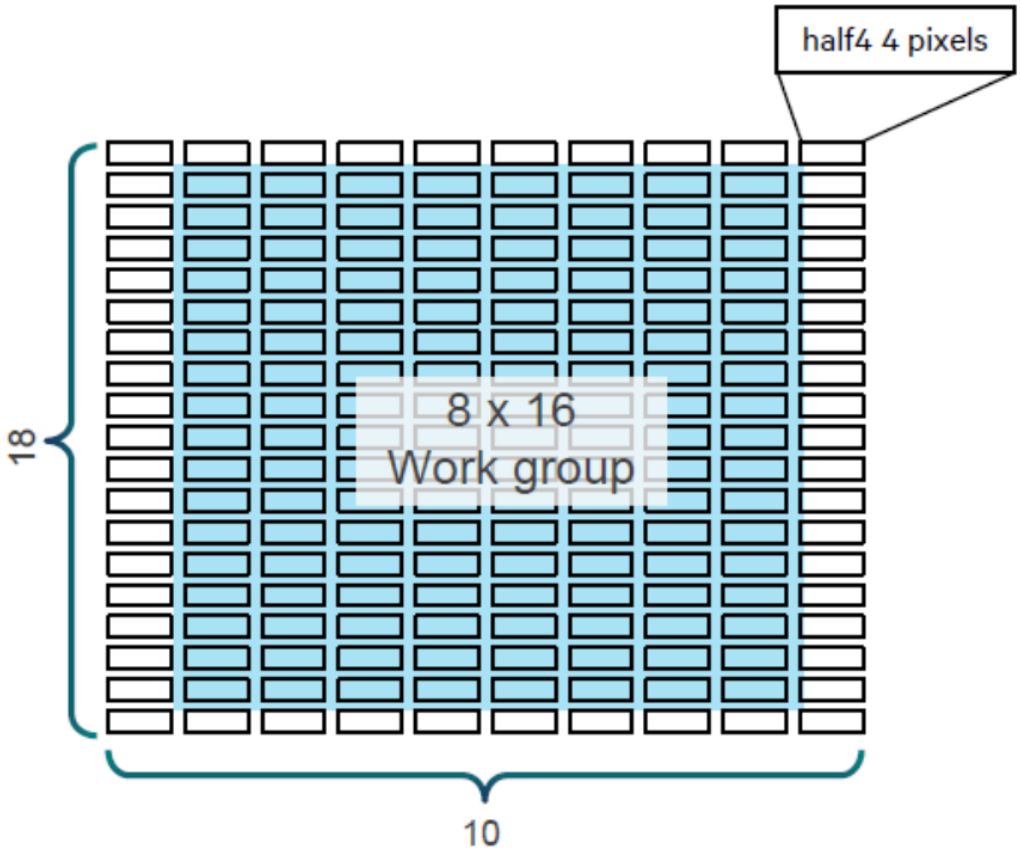

图 9-6 使用本地内存进行 Epsilon 滤波

表 9-1 列出了两种情况的设置和他们的性能。整体性能比原始的要更好，然而从 9.3.4 节中可以看出，并没有获得最好的性能。

表 9-1 使用本地内存的性能

	情况 1	情况 2
workgroup	8x16	8x24
本地内存大小（byte）	10x18x8=1440	10x26x8 = 2080
性能	2.4x	2.8x

如 7.1.1 节所讨论的，本地内存使用时，通常需要使用 barrier 进行 workgroup 中 work item 之间的同步，这样会导致性能不比使用全局内存时的好。而且，如果同步导致许多开销的话，它可能会有更差的性能。在这种情况下，如果全局内存有高 cache 命中率的话，那么全局内存可能是一个更好的选择。

9.2.6 分支操作的优化

Epsilon 滤波需要进行像素之间的比较，如下：

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

三元素符 ?: 是发生在硬件上的分支，因为同一个 wave 中的不是所有的 fiber 都会执行相同的分支。分支操作可以被 ALU 操作替代，如下所示：

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

这种优化方法之前就应用在 9.2.2 章中描述过的一个例子中，性能从 3.4x 提升到 5.4x。

这个操作的关键差异是，新代码是在高度并行的 ALU 下执行，而且在同一个 wave 中所有的 fiber 基本上执行的是同一块代码，尽管变量 Cond 可能有不同的值，而原来的代码会使用非常耗时的硬件逻辑来处理分支。

9.2.7 总结

表 9-2 总结了优化的步骤和性能数据。最初，算法是内存瓶颈的。通过数据打包，向量化装载，它变成 ALU 瓶颈。总的来说，对于这个例子的关键是优化装载数据的方式。许多其他的内存瓶颈的用例可以使用相似的技术来加速。

表 9-2 优化和性能的总结

步骤	优化方法	imag 类型	kernel 中的数据类型	向量化操作	速度提升
1	最初的 GPU 实现	CL_R | CL_UNORM_INT8	float	1-pixlel/work item	
2	在 kernel 中使用 half 类型	CL_R | CL_UNORM_INT8	half		1.0x
3	数据打包	CL_RGBA| CL_UNORM_INT8			1.4x
4	向量化处理 循环展开			4-pixel/work item （halt4 output）	3.4x
5	每个 work item 处			8-pixel/work	3.5x

	理更多的像素			item	
6	每个 work item 处理更多的像素			16-pixel/work item	0.5x
4-1	使用 LM (workgroup 大小为 8x16)				2.4x
4-1-1	使用 LM (workgroup 大小为 8x24)			4-pixel/work item	2.8x
4-1-2	使用 LM 移除分支操作，workgroup 大小为 8x24				2.9x
4-2	移除分支操作				5.4x

表 9-3 中展示了在三种分辨率的图片上使用 Epsilon 滤波后的 OpenCL 性能。从图中可以看出，越大的图像，性能提升越大。对于一个 3264x2448 的图像，可以看到有 5.4x 的性能提升，相比之言，在 512x512 的图像上，优化后的代码性能比最初的 OpenCL 代码的性能，只有 4.3x 的提升。这是很容易理解的，因为在不考虑任务量的情况下，消耗的时间与 kernel 的执行时间是正相关的，而且任务量越大，他在整个性能数据中的权重越小。

表 9-3 不同分辨率的图片的性能统计

图片的分辨率		512x512	1920x1080	3245x2448
像素点的个数		0.26MP	2MP	8MP
设备（A430）	GPU 的最初实现结果	1x	1x	1x
	GPU 优化后的结果	4.3x	5.2x	5.4x

9.3 Sobel 滤波

Sobel 滤波，也被称作 Sobel 操作，用在很多图像处理和计算机视觉算法的边界检测中。它使用两个 3x3 的 kernels 与原始图片结合，近似得出导数。这里有两个 kernel：一个负责水平方向，一个负责垂直方向，如图 9-7 所示：

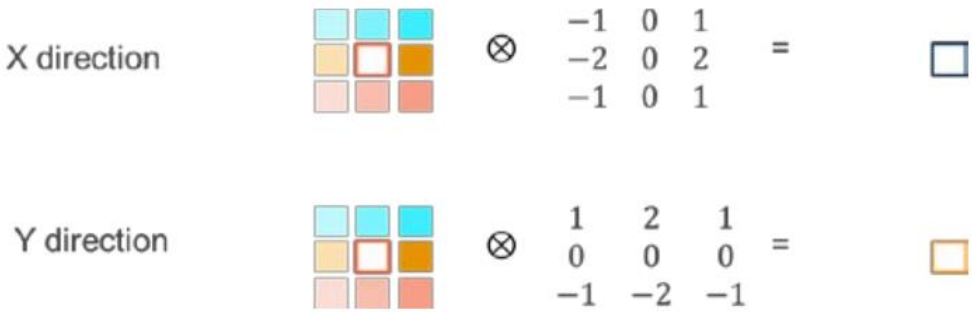

图 9-7 Sobel 滤波的两个方向操作

9.3.1 算法优化

Sobel 的滤波是一个可分离的滤波器，可以如下分解：

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

图 9-8 Sobel 滤波分离

相比于不可分离的 2D 滤波，一个可分离的 2D 滤波器可能将复杂度从 O（n^2）降低到 O（n）。由于 2D 的高复杂性和计算的耗时，理想的情况就是使用可分离的滤波器。

9.3.2 数据打包的优化

尽管可分离的滤波，已经减少了很多计算，但是每一个要被滤波的像素点所需要的像素点个数是一样的，比如对 3x3kernel 来说，需要 8 个邻居像素点加上中心像素点。可以明显地看出来，这个一个内存瓶颈的问题。所以，如果有效的将像素点装载进 GPU 是性能优化的关键。下面的图片中阐述了 3 种可以使用的选择：

图 9-9 每个 work item 处理一个像素点，每个 kernel 装载 3x3 个像素点

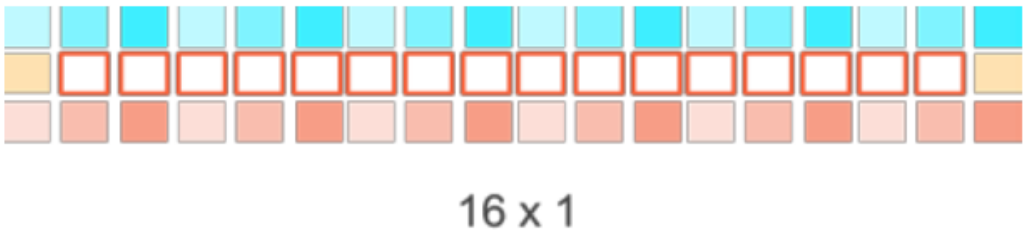

图 9-10 处理 16x1 个像素点，装载 18x3 个像素点

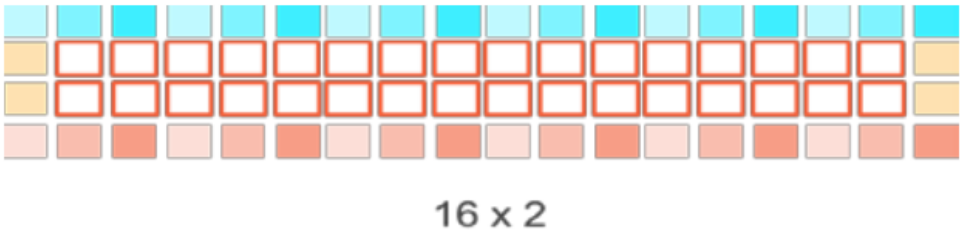

图 9-11 处理 16x2 个像素点，装载 18x4 个像素点

下面的表格中总结了每种情况下总的字节数和平均字节数。对于表 9-9 所述的第一个种情况，每个 work item 只对一个像素点进行 Sobel 滤波。随着每个工作项处理的像素点的个数的增加，需要装载的数据数量将会减少，如 9-10 和 9-11 所示。这将会减少全局内存到 GPU 的数据量，从而提升性能。

表 9-4 3 种情况下数据装载/存储的数量

	1 pixel/work item	16x1 pixels/work item	16x2 pixels/work item
总的输入字节数	9	54	72
平均输入字节数	9	3.375	2.25
平均存储字节数	2	2	2

9.3.3 向量化的装载/存储优化

对于 16x1 和 16x2 这两种情况，装载/存储的次数可以通过使用 OpenCL 中的向量化装载存储函数进行进一步的减少，比如 `float4`,`Int4` 和 `char4` 等。表 9-5 表示了使用了向量化的情况下，需要的装载/存储的次数（假设像素的数据类型是 8-bit）。

表 9-5 使用向量化的装载/存储方法需要的装载和存储次数

	16x1 向量化	16x2 向量化
装载	6/16=1.375	8/32=0.374
存储	2/16=0.125	4/32 = 0.125

下面是一小段向量化装载的代码：

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

如下，在边界处，需要装载两个像素

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

注意： 每个工作项处理像素点的数量提升可能会导致严重的寄存器使用空间的压力，从而导致寄存器溢出到私有内存和性能下降。

9.3.4 性能统计和总结

在使用了两种优化步骤之后，可以看到性能的显著提升，如图 9-12 所示，图中在 MSM8994(Adreno418）上的原始算法性能（每个 work item 处理单个像素点）被归一化为 1.

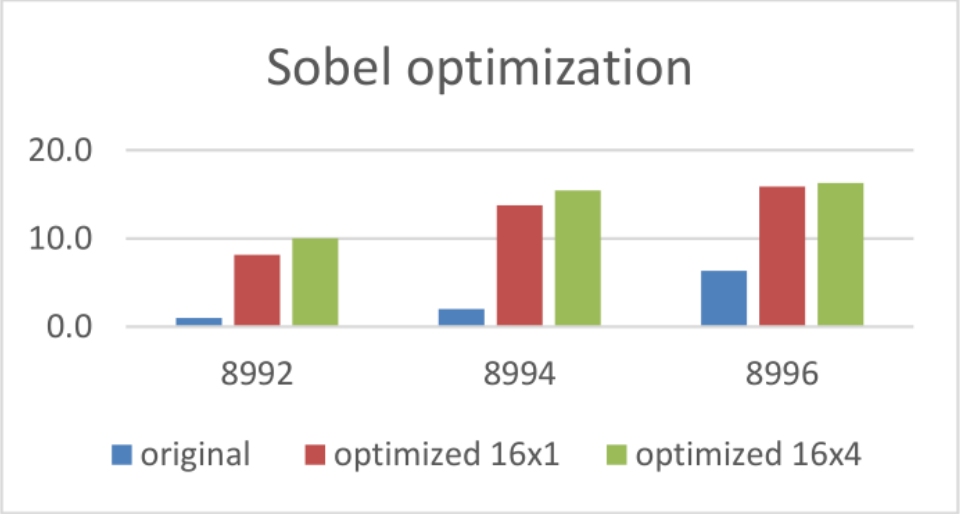

图 9-12 通过使用数据打包和向量化装载/存储带来的性能提升

为了总结，下面是这个用例优化的几个关键点：

- 数据打包提升了内存访问效率
- 向量化装载/存储是减少内存繁忙的关键点。
- 在这种情况下，优先选择更短的数据类型比如整型或者 char 型。

在这种情况下，没有使用本地内存。数据打包和向量化的装载/存储已经最小化了可复用数据的重叠。因此，使用本地内存并不能提升性能。

可能还存在其他的提升性能的选项，比如使用 texture 来替换 global buffer。

9.4 总结

在这一章节中提供了一些简单的例子和代码片段来证实了前几章说明的优化规则，并且指出了性能是如何改变的。开发者可以尝试在真实的设备上使用这些步骤。由于编译器和驱动的升级，这些结果可能不会被准确的重现。但是，一般地，通过这些优化步骤，肯定会实现同样的性能提升。

10 总结

这篇文档主要是介绍了关于在 Adreno GPUs 上优化 OpenCL 代码的详细方法。文档中提供的大量信息能够帮助开发者理解 OpenCL 基础和 Adreno 结构，还有最重要的，掌握 OpenCL 优化技能。

OpenCL 优化经常是具有挑战性的而且需要大量的尝试和试错。因为每个供应商对同一个任务可能都有他自己的最好的实践方法，所以通读这个文档，并对 Adreno GPUs 的优化准则和方法有深入的了解都是很重要的。许多看起来次要的因素可能对性能有很大的影响。不幸地是，不亲自动手操作和实践会很难解决这些问题（比如发现这些次要的因素）。

由于时间的限制和一些其他原因，有一些方法没有涉及到。Adreno GPUs 支持大量的扩展，这些扩展可能会显著地提升性能而且增加了许多其他的功能。比如说，最近的 Adreno GPU 支持的某些专属的图片格式，指的是，从图片的单个处理器上获取原始图片和音频，压缩成这种图片格式，然后进行直接和有效的处理。这些可以保存一些人为的操作，同样也可以提升带宽的使用。

该文档的以后版本可能会包含更多的主题。