ARM® Mali™ GPU

版本： 2.0

OpenGL ES 应用程序优化指南

ARM DUI 0555B (ID051413)

ARM Mali GPU

OpenGL ES 应用程序优化指南

版本信息

本指南已进行了下列更改。

更改记录

日期	发布	保密性	更改
2011 年 3 月 30 日	A	非机密	第一版
2013 年 5 月 14 日	B	非机密	第二版

产品状态

本文档中的信息是最终版，即用于开发完成的产品。

网址

http://www.arm.com

ARM DUI 0555B

ID051413

目录

ARM Mali GPU OpenGL ES 应用程序优化指南

附录 A Mali GPU 性能计数器

前言

本前言介绍 *ARM® Mali™ GPU OpenGL ES 应用程序优化指南*。其中包含下列小节：

关于本指南

本指南适用于 Mali-200、Mali-300、Mali-400 MP 和 Mali-450 MP *图形处理器单元* (GPU)。

目标读者

本指南是为正在开发或移植应用程序到 Mali GPU 平台的应用程序开发人员而撰写的，撰写时假定应用程序开发人员拥有一些 3D 图形编程知识，但未到专家级别。

使用本指南

本指南分为下列章节：

第 1 章 *简介*

本章提供 Mali GPU 优化简介。

本章介绍优化的概念，说明优化的原因和可以优化的对象，还提供了图形流水线概述，介绍了桌面和移动平台开发之间的区别。

第 2 章 *优化检查表*

本章列举开始完整优化流程前要检查的一系列事项。这些是一些相对简单的优化技巧，能够带来巨大的改变。我们先将它们列举出来，提醒您注意这些基本、却很重要的技巧。

第 3 章 *优化流程*

本章介绍整个优化流程，借助流程图说明诊断和解决性能问题的流程。流程图分步介绍进行测量、确定瓶颈，以及使用相关优化来消除瓶颈的流程。

第 4 章 *测量与查找瓶颈*

本章介绍如何测量您的应用程序，以及如何查找性能瓶颈。

第 5 章 *优化工作流程*

本章提供一系列流程图，指导您解决一系列常见的性能问题。每一流程图都逐步指导您诊断问题，并选择优化以消除瓶颈。

第 6 章 *应用处理器优化*

本章为性能受到应用处理器限制的应用程序提供优化列表。

第 7 章 *API 级别优化*

本章为性能受到 API 使用限制的应用程序提供优化列表。

第 8 章 *顶点处理优化*

本章为性能受到顶点处理限制的应用程序提供优化说明。

第 9 章 *片段处理优化*
本章为性能受到片段处理限制的应用程序提供优化列表。

第 10 章 *带宽优化*
本章为性能受到带宽限制的应用程序提供优化列表。

第 11 章 *其他优化*
本章提供未分类到其他章节的优化的列表。

附录 A *Mali GPU 性能计数器*
本章介绍 Mali GPU 性能计数器。

术语表
*ARM 术语表*列举了 ARM 文档中使用的术语，还提供这些术语的定义。*ARM 术语表*不包含行业标准术语，除非 ARM 的含义与公认的含义不同。
请参见 *ARM 术语表*，网址为 http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html。

约定
如需本指南中所用约定的介绍，请见以下小节：
- *排版约定*。

排版约定
下表介绍了排版约定：

排版约定

样式	用途
斜体	介绍特殊术语，以及表示交叉引用和引文。
粗体	突出显示界面元素，如菜单名称等。表示信号名称。适当时也用于描述性列表中的术语。
等宽字体	表示您可以在键盘上输入的文本，如命令、文件和程序名称，以及源代码。
等宽字体	表示命令或选项允许使用的缩写。您可以输入标有下划线的文本，而不输入完整的命令或选项名称。
等宽*斜体*	表示等宽字体文本中的参数，该参数可以替换为具体的值。
等宽粗体	表示语言关键字，在示例代码之外使用。
< 和 >	出现在代码或代码片段中时，用于括起汇编程序语法可替换的术语。例如：MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
SMALL CAPITALS	在正文中用于一些具有特定技术含义的术语，它们在 *ARM 术语表*中定义。例如，实施定义、特定于实施、未知和无法预测。

其他读物
本节提供 ARM 和第三方的出版物列表。

若要访问 ARM 文档，请参见信息中心网站 http://infocenter.arm.com。

ARM 出版物

本指南包含特定于本产品的信息。如需其他相关信息，请参见下列文档：

- *ARM® Mali™ GPU OpenGL ES 应用程序开发指南* (ARM DUI 0363)。
- *ARM®Cortex®-A 序列程序员指南*(ARM DEN 0013)。

其他出版物

本节提供第三方出版的相关文档列表：

- *EGL 1.4 规范*，http://www.khronos.org。
- *OpenGL ES 2.0 规范*，http://www.khronos.org。

反馈

ARM 欢迎您就本产品和其文档提供反馈。

有关本产品的反馈

如果您对本产品有任何意见或建议，请联系您的供应商，并提供：

- 产品名称。
- 产品修订版或版本。
- 一份说明，包含尽可能多的信息。适用时，请附上症状和诊断过程。

有关内容的反馈

如果您有内容方面的意见，请发送电子邮件到 errata@arm.com。请提供：

- 标题。
- 编号，ARM DUI 0555B。
- 您的意见所适用的页码。
- 您的意见的扼要说明。

ARM 也欢迎您就增补和改进提供一般建议。

备注

ARM 仅在 Adobe Acrobat 和 Acrobat Reader 测试了本 PDF 文件，无法担保使用其他 PDF 阅读器时的文档显示质量。

第 1 章
简介

本章介绍 *ARM® Mali™ GPU OpenGL ES 应用程序优化指南*。其中包含下列小节：

备注

本指南主要适用于 Mali-200、Mali-300、Mali-400 MP 和 Mali-450 MP GPU。其中所述的大多数技巧也适用于 Mali-T600 系列 GPU。

1.1 关于优化

图形是为了美化事物。优化则是利用最少的计算投入让事物拥有美丽的外观。对于计算动力和内存带宽有限并需要节省功率的移动设备而言，优化尤为重要。

优化是处理应用程序并提高其效率的过程。对于图形应用程序，这通常意味着修改应用程序以提高其速度。

低帧率意味着应用程序看上去跳跃不定。这会带来糟糕的印象，让游戏等应用程序难以操控。您可以使用优化来提高应用程序的帧率。这可让应用程序拥有更好、更顺畅的使用体验。

持续稳定的帧率通常比高帧率更为重要。与相对较低却又稳定的帧率相比，变化不定的帧率会带来更加糟糕的印象。

优化具有多个不同目标，例如：

- 提高帧率。
- 让内容拥有更多细节。
- 降低功耗。
  - 使用较少的内存带宽。
  - 每帧使用较少的时钟周期。
- 减小内存占用。
- 减小下载大小。

不同的优化通常是互相关联的。例如，您可以将帧率优化用作节省功率的手段。为此，您可以针对更高的帧率优化应用程序，但将帧率限制到较低的水平。这可以降低功耗，因为 GPU 需要更少的时间来计算帧，可在更长的期间内保持空闲。

减少应用程序内存占用不是典型的优化，但也有用，因为较小的应用程序更容易缓存。在这种情形中，让应用程序变小也具有提高应用程序速度的作用。

备注

本指南主要侧重于提高应用程序帧率。适当时，也会提及其他优化类型。

1.2 如何使用本指南

本指南旨在帮助您创建更加出色的应用程序。可以用它来帮助优化现有的应用程序，也可在开发应用程序时使用其中的技巧。

您可以在开发过程中的任何阶段优化应用程序。最佳做法是从良好的设计开始，并在开发过程中将优化技巧运用到您知道属于计算密集型的应用程序区域。

您可以通过下列方式使用本指南：

- 若要改善现有应用程序的性能，或在接近开发尾声时这么做，请参见下一章：
  - 第 2 章 *优化检查表*。
- 若要了解优化流程，请参见下列章节：
  - 第 3 章 *优化流程*。
  - 第 4 章 *测量与查找瓶颈*。
- 若要用作参考指南，通过示例工作流程带领您完成整个优化流程，请参见下列章节：
  - 第 5 章 *优化工作流程*。
- 若要了解优化技巧或用作参考书，请参见下列章节：
  - 第 6 章 *应用处理器优化*。
  - 第 7 章 *API 级别优化*。
  - 第 8 章 *顶点处理优化*。
  - 第 9 章 *片段处理优化*。
  - 第 10 章 *带宽优化*。
  - 第 11 章 *其他优化*。

备注

这些章节根据处理器类型来划分优化。不过，许多优化并不针对于一种处理器，也可以应用到其他处理器。

1.3 Mali GPU 硬件

本节介绍 Mali GPU 硬件的主要组成部分。其中包含下列小节：

- *基于区块的渲染*。
- *Mali GPU 硬件组件*。
- *顶点处理器*（第 1-5 页）。
- *片段处理器*（第 1-5 页）。
- *L2 缓存控制器*（第 1-5 页）。

1.3.1 基于区块的渲染

Mali GPU 使用*基于区块的延迟*渲染。

Mali GPU 将帧缓冲区划分为区块，逐个区块进行渲染。基于区块的渲染效率较高，因为可通过片上内存计算像素的值。这一技术是移动设备的理想选择，因为与传统的渲染技术相比，它要求的内存带宽比较少，功耗也较小。

1.3.2 Mali GPU 硬件组件

Mali GPU 通常在移动或嵌入式环境中用于加速 2D 和 3D 图形。其图形是通过 OpenGL ES 图形流水线生成的。请参见*图形流水线*（第 1-6 页）。

Mali GPU 是可配置的，因此包含多个不同组件。Mali GPU 可以包含的组件类型有：

- 顶点处理器。
- 片段处理器。
- *内存管理单元* (MMU)。
- *电源管理单元* (PMU)。
- L2 *缓存*。

表 1-1 显示不同 Mali GPU 中的组件。

表 1-1 可能的 Mali GPU 组件

Mali GPU	顶点处理器	片段处理器	MMU	PMU	L2 缓存
Mali-200	1	1	1	-	-
Mali-300	1	1	2	1	8 KB
Mali-400 MP	1	1-4	每处理器一个	1	0 KB-256 KB
Mali-450 MP	1	1-8	每处理器一个	1	64 KB-256 KB，每像素处理器块

备注

本指南不能直接用于 Mali-T600 系列 GPU，但所述的大部分优化技巧也适用于这些 GPU。

图 1-1（第 1-5 页）显示的是 Mali-400 MP GPU。

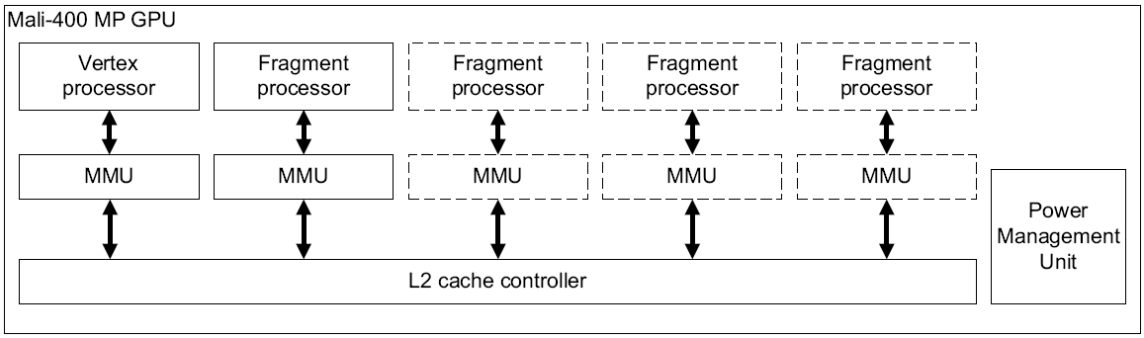

图 1-1 Mali-400 MP GPU

通用应用处理器运行操作系统、图形应用程序，以及 Mali GPU 驱动程序。

1.3.3 顶点处理器

顶点处理器负责图形流水线的顶点处理阶段。它为片段处理器生成原语列表并加速数据结构的构建，如多边形列表和打包顶点数据。

1.3.4 片段处理器

片段处理器负责图形流水线的栅格化和片段处理阶段。它们使用由顶点处理器生成的数据结构和原语列表，生成显示在屏幕上的帧缓冲区结果。

备注

Mali-200 和 Mali-300 GPU 包含一个片段处理器。Mali-400 MP GPU 最多可以包含四个片段处理器。Mali-450 MP GPU 最多可以包含八个片段处理器。

1.3.5 L2 缓存控制器

Mali-300、Mali-400 MP 和 Mali-450 MP GPU 包含一个或多个 L2 缓存控制器。L2 缓存可以减少内存带宽使用并降低功耗。

L2 缓存设计为隐藏访问内存的成本。主内存通常要慢于 GPU，因此 L2 缓存可以在部分应用程序中大幅提高性能。

1.4 图形流水线

Mali-200、Mali-300、Mali-400 MP 和 Mali-450 MP GPU 实施的图形流水线支持 OpenGL ES 1.1 和 2.0 *应用程序编程接口* (API)。本节介绍 OpenGL ES 图形流水线，其中包含下列小节：

- *OpenGL ES 图形流水线概述*。
- *初始处理*。
- *每顶点运算*。
- *栅格化和片段着色*（第 1-7 页）。
- *混合和帧缓冲区运算*（第 1-7 页）。

1.4.1 OpenGL ES 图形流水线概述

图 1-2 显示 OpenGL ES 2.0 图形流水线的典型工作流程。

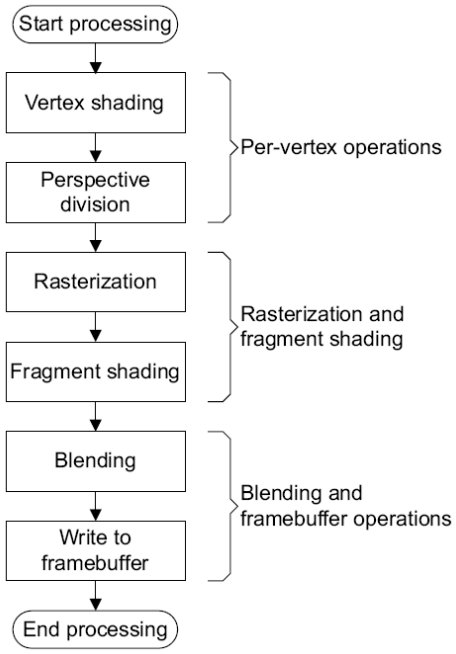

图 1-2 OpenGL ES 图形流水线工作流程

Mali GPU 使用数据结构和硬件功能块来实施 OpenGL ES 图形流水线。

1.4.2 初始处理

OpenGL ES 或 OpenVG 的 API 级驱动程序在内存中为 GPU 创建数据结构，并为每个场景配置硬件。

软件：

- 为*渲染状态词*(RSW) 和纹理描述符生成数据结构。
- 为顶点处理器创建命令列表。
- 根据需要编译着色器。

1.4.3 每顶点运算

顶点处理器为每个顶点运行顶点着色器程序。

该着色器程序执行：

- 照明。
- 变换。
- 视口变换。
- 透视变换。

顶点处理器也执行下列处理：

- 汇编图形原语的顶点。
- 构建多边形列表。

1.4.4 栅格化和片段着色

片段处理器执行下列运算：

读取数据 片段处理器读取状态信息、多边形列表和变换后的顶点数据。它们在三角形设置单元中处理，以生成系数。

栅格化多边形

栅格化程序从三角形设置单元中获取系数，应用等式以创建片段。

执行片段着色器

片段着色器程序在每个片段上执行，计算该片段的颜色。

1.4.5 混合和帧缓冲区运算

片段处理器在处理区块缓冲区后为帧缓冲区生成最终的显示数据。为提高处理速度，每个片段处理器处理不同的区块。

混合单元利用区块缓冲区中对应位置已存在的颜色混合片段。

片段处理器测试片段，并更新区块缓冲区。片段处理器计算片段是可见还是隐藏，然后将可见片段存储在区块缓冲区中。

在区块完全渲染后，片段处理器将区块缓冲区的内容写入帧缓冲区。

1.5 桌面系统和移动设备之间的区别

移动和嵌入式系统必须权衡计算动力、电池续航时间和成本。这意味着与桌面平台相比，移动平台中下列资源受到限制：

- 计算能力。
- 内存容量。
- 内存带宽。
- 功耗。
- 物理尺寸。

桌面系统没有这些限制，因此应用程序开发人员有许多倍的计算资源可以利用。

Mali GPU 通常在移动或嵌入式系统中使用；因此，如果要从桌面系统移植图形应用程序，您务必要了解这些区别。

一些图形丰富的应用程序最初是为桌面平台开发的，然后再移植到嵌入式或移动平台。可用资源的减少意味着应用程序无法以桌面平台上的相同性能水平运行。

优化可以让您的应用程序更加接近它在桌面平台上实现的性能水平。

请参见*移植桌面应用程序到移动设备检查表*（第 2-10 页）。

1.6 移动渲染器之间的区别

本节说明移动渲染器之间的区别。其中包含下列小节：

- *与其他移动 GPU 的区别*。
- *与软件渲染器的区别*。

1.6.1 与其他移动 GPU 的区别

所有 GPU 都有不同的优化点。许多优化是通用的，但不能推断针对一个平台优化的应用程序会自动地在另一平台上运行良好。

例如，ARM 建议您在应用程序中按照从前到后的顺序排列对象或三角形。这样可以尽早剔除片段，减轻片段处理器上的负载，并减少过度绘制。

这一优化不是 Mali GPU 独有的，它也适用于一些其他移动 GPU 和桌面 GPU。

1.6.2 与软件渲染器的区别

如果您的应用程序运行在具有软件渲染器的现有移动设备上，它可能无法在 Mali GPU 上良好运行。这是因为用于 GPU 的优化与用于软件渲染器的优化大不相同。

要在使用 GPU 时获得高性能，您的应用程序可能需要重新优化，尤其要确保每一帧不要使用大量的绘制调用。一起分批处理对象可以减少绘制调用的数量。

如需更多信息，请参见*尽可能减少绘制调用* （第 7-2 页）。

第 2 章
优化检查表

本章提供在开始完整的优化流程前需要核查的检查表。其中包含下列小节：

备注

这些技巧对性能有非常大的影响，因此请确保核查后再进入后续的章节。

2.1 关于优化检查表

应用程序性能不佳的原因有很多。优化 3D 应用程序是个复杂的主题，许多不同的技巧可以在不同的情形中使用。

然而，许多性能问题都能相对轻松地修复。其中大多数需要相对简单的优化技巧，您可以使用这些技巧改善图形的性能和质量。

本章列举多个能修复许多基本问题的技巧。请务必先核查该列表，然后再尝试更加高级的优化。

2.2 检查表

本节提供要在应用程序中检查的事项列表。其中包含下列小节：

2.2.1 检查显示设置

确保显示系统的设置正确，并且应用程序与之相匹配。如果有不匹配，系统可能会执行像素格式转换，并且可能会执行位块传送以进行修正。用于转换的资源无法供应用程序使用，因此会对应用程序性能造成负面影响。

检查下列设置：

- 确保应用程序具有正确的绘制表面

  应用程序请求绘制表面时，可能无法获得其请求的表面类型。为了避免获得错误的表面，请在返回潜在表面时进行检查，并且仅接受正确的表面。

  如需演示如何通过 EGLConfig 排序的示例代码，请访问 *Mali 开发者中心*，网址为 http://malideveloper.arm.com/。

- 确保帧缓冲区分辨率和颜色格式与显示控制器兼容。

下列建议适用于使用 Linux OS FBDEV 的平台：

- 确保帧缓冲区不超过屏幕的分辨率。
- 确保帧缓冲区不超过屏幕的色深。
- 确保绘制表面格式与帧缓冲区格式相同。

2.2.2 尽可能使用直接渲染

位块传送是一种代价高昂的运算，不仅费时，而且会消耗许多内存带宽。避免它可以大幅提升性能。

将图形直接绘制到帧缓冲区的过程叫做直接渲染。请尽可能使用直接渲染来避免位块传

送并提高应用程序的性能。使用直接渲染根据操作系统不同而异，因此请参阅操作系统的文档，了解其是否可用和具体用法。

2.2.3 使用具有正确设置的正确工具

使用正确的工具或具有正确设置的工具能够大幅影响性能。确保您使用的是应用了最新更新，并且根据您的设备进行了适当设置的正确工具：

使用最新的工具

利用开发工具的最新版本编译您的应用程序。这可确保应用程序能从最新稳定性改进和性能优化中获益。

在工具更新后重新生成所有内容

如果换用工具或工具版本，请务必重新编译所有内容，让软件的所有部分都从更改中获益。

针对正确的架构进行构建

应用处理器架构有不同的版本。要确保最佳的性能，请务必针对正确的版本进行构建。如果针对较旧的架构版本构建，然后在较新的版本上运行，性能可能会降低。

使用硬件中的设施

如果您的平台具备硬件浮点、*顶点浮点* (VFP) 或 NEON™ 功能，确保编译器进行相应的设置。另外，也请考虑使用利用这些硬件功能的库。

备注

如果操作系统支持硬件浮点运算，请确保整个系统和支持库都构建为支持该功能。

优化发布版本

确保在发布时，将编译器设置为生成为速度优化的二进制文件。这可提供最佳的性能。

2.2.4 删除调试信息

收集调试信息有助于更正错误，但需要内存和计算资源。收集调试信息的过程通常对性能有负面影响。

确保关闭调试后再发布应用程序。只有应用程序中需要调试功能时，才使调试保持打开。

如需其他发布前检查，请参见*最终发布检查表*（第 2-12 页）。

使用最少的 printf() 调用

printf() 调用可能会很慢。您可以仅在数量相对较多的帧后再显示帧率，从而防止它们影响应用程序性能。例如，每 100 个帧，而不是每个或每两个帧进行 printf() 调用。

如果在 Android OS 上使用 logcat，您可以使用较多的调用，因为其对性能影响非常小。

调用 glGetError() 不要超过一帧一次

对 glGetError() 的每次调用都需要时间进行处理。每帧大量进行此调用会消耗大量的计算资源，足以限制应用程序的帧率。确保调用 glGetError() 不要超过一帧一次。

您可以使用 #define 宏为开发版本生成调试代码，并在发布版本中删除。

备注

如果应用程序在您进行性能测量时收集调试信息，这些测量很可能不准确。

2.2.5 避免无限命令列表

本节介绍无限命令列表及其造成的问题。

延迟渲染过程涉及将命令放入列表中。如果不清除帧之间的缓冲区，命令列表将持续扩大。这会导致 Mali GPU 重复已为前面的帧完成的工作。这些工作显然是不必要的。

备注

- 只有应用程序渲染到 pixmapsurface 或 pbuffersurface 等表面，而且不在帧结束时清除命令列表，此问题通常才会成为问题。
- 如果应用程序使用*帧缓冲区对象* (FBO)，则此问题不会成为问题。
- 渲染到 eglWindowSurface 的应用程序会在每次调用 eglSwapBuffers() 时自动结束帧。

为防止命令列表持续扩大，请确保应用程序在绘制新的帧之前清除下列缓冲区：

- 颜色缓冲区。
- 深度缓冲区。
- 模板缓冲区。

您可以使用下列命令清除这些缓冲区：

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTHBUFFER_BIT | GL_STENCILBUFFER_BIT );
```

备注

您必须同时清除所有这些缓冲区。

2.2.6 避免停止图形流水线的调用

一些 OpenGL ES 函数调用从帧缓冲区中读取。为此，Mali GPU 必须先渲染整个图像，然后您才能从中读取。这一操作导致图形流水线停止，因此有可能会降低性能。

避免下列 OpenGL ES 调用：

- glReadPixels()
- glCopyTexImage()
- glTexSubImage()

2.2.7 不要每帧编译着色器

为每个帧编译着色器是可能的。但这会降低应用程序的性能，因为着色器编译需要应用处理器和内存资源。

在应用程序启动时编译着色器更为高效。由于该操作仅在应用程序启动时需要资源，因此不会降低应用程序运行时的性能。

也可以将应用程序和预编译的着色器一起打包。这些着色器仅需要在运行时进行关联，所以需要相对很小的运行时计算资源。

备注

预编译的着色器仅对编译时所针对的 GPU 有用。

2.2.8 使用 VSYNC

垂直同步 (VSYNC) 将应用程序的帧率与屏幕显示速率同步。VSYNC 是个有用的技巧，这是因为：

- 它可通过去除撕裂提高图像质量。
- 它可防止应用程序产生帧的速度快于屏幕可以显示的速度，从而可以降低功耗。

如需更多信息，请参见*使用 VSYNC*（第 11-8 页）。

备注

在测量性能时，请勿使用 VSYNC。

2.2.9 使用平台适用的图形资源

与桌面系统相比，一些移动平台的小屏幕拥有相对较低的分辨率。

如果将桌面应用程序移植到移动平台，精细细节的视觉冲击可能很少。因此，您可以简化图形资源。例如，您可以进行下列更改：

- 减小纹理的大小和位深。
- 减小几何复杂度。
- 简化或删除几乎没有可见影响的特效。

这些更改可以减少内存带宽使用，实现更高的性能。如需有关简化的更多信

息，请参见*使用近似*（第 11-2 页）。

2.2.10 不要使用 24 位纹理

对于高位深纹理，请使用 16 位或 32 位纹理，而不要使用 24 位纹理。

24 位纹理无法完美装入缓存中。使用 24 位纹理可能造成数据使用多个缓存行，因此对性能和内存带宽有负面影响。

16 位和 32 位纹理可以毫无问题地装入缓存中，因此不会遭遇此类性能问题。

备注

对于大多数纹理而言，最好使用纹理压缩，而不是高位深纹理。压缩纹理使用较少的内存带宽，因此可提高性能。如需更多信息，请参见*使用纹理压缩*。

2.2.11 使用纹理映射

纹理映射这种技巧可以同时：

- 改善图像质量。
- 提高性能。
- 减少内存带宽使用。

您可以在 OpenGL ES 中通过一行代码，命令 Mali GPU 驱动程序在运行时生成纹理映射。此外，您也可利用 *Mali GPU 纹理压缩工具*预生成纹理映射。

纹理映射这种方式可以轻松地提高内存带宽受限应用程序的性能。

备注

一些应用程序在启用纹理映射时表现出非常显著的性能改善。

2.2.12 使用纹理压缩

纹理压缩这种技巧可以减少纹理在内存中占用的大小。纹理压缩可以：

- 提高性能。
- 提高纹理可缓存能力。
- 减少内存带宽使用。

Mali GPU 驱动程序支持 *Ericsson 纹理压缩* (ETC1)。这种纹理压缩类型广泛用于 OpenGL ES 版本 1.1 和 2.0。您可以使用 *Mali GPU 纹理压缩工具*创建 ETC1 压缩纹理。

如需更多信息，请参见*使用纹理压缩*（第 10-6 页）。

2.2.13 减少内存带宽使用

内存带宽要求许多功率，与桌面系统相比，移动设备中有很大限制。带宽很容易成为限制应用程序性能的瓶颈。基于此原因，务必要保持较低的带宽使用量：

- 带宽是一种共享资源，因此用量太大会以无法预料的方式限制整个系统的性能。例如，图形内存与应用内存共享，因此 GPU 的带宽用量高可导致应用处理器性能降低。
- 访问缓存中的数据可以降低用电量，提高性能。如果应用程序必须大量读取内存，请使用纹理映射和纹理压缩等技巧来确保数据对缓存友好。请参见*使用纹理映射*，同时请参见*使用纹理压缩*。

备注

判断内存带宽是否造成问题比较困难。请参见*判断内存带宽是否为问题所在*（第 4-18 页）。

减少内存带宽使用的方式有许多。

- 激活背面剔除。
- 利用视域剔除。
- 确保纹理不会太大。
- 使用适合屏幕上对象的纹理分辨率。
- 尽可能使用位深较低的纹理。
- 如果纹理不包含锐利的细节，则使用分辨率较低的纹理。
- 仅针对具体对象使用三线性过滤。
- 利用*细节级别* (LOD)。

如需更多信息，请参见第 10 章 *带宽优化*。

2.2.14 使用顶点缓冲区对象

顶点缓冲区对象 (VBO) 是一种数据存储机制，可以让应用程序在 GPU 内存中存储和操控数据。VBO 可以大幅削减顶点带宽开销，因此能提供显著的性能改善。

如果每一帧都将数据发送到 GPU，则无论是否变化它都会被复制。使用 VBO 可以避免此类复制，因为在 GPU 内存中存储数据意味着无需复制。

备注

您也可以使用索引缓冲区对象对索引缓冲区采取相同的做法。

2.2.15 确保应用程序没有应用处理器约束问题

如果应用程序存在应用处理器约束问题，图形系统会在等待处理图形数据时闲置。在这种情形中，您必须优化应用程序代码。提高图形性能的优化对总体性能没有影响。

应用程序可能会在以下方面存在应用处理器约束问题：

- 应用程序逻辑计算过于密集。
- 应用程序没有优化使用 API，造成驱动程序负载过重。
- 应用程序逻辑和驱动程序的组合。

优化应用程序代码的方式有多种：

- 优化 API 使用。
- 对齐数据。
- 使用循环优化。
- 使用快速数据结构。
- 使用矢量指令。

如需更多信息，请参见第 7 章 *API 级别优化*和第 6 章 *应用处理器优化*。

备注

如果图形应用程序来源于桌面平台并迁移到移动平台，它更有可能存在应用处理器约束问题。

2.3 移植桌面应用程序到移动设备检查表

如果将桌面应用程序移植到配有 Mali GPU 的移动设备，则整个检查表都适用。不过，尤其要注意如下所列：

- 按照从前到后顺序绘制不透明对象。
- 避免大量三角形。
- 避免长着色器。
- 避免高位深和高分辨率纹理。
- 使用纹理压缩。

2.4 检查系统设置

系统设置是否正确对应用程序性能至关重要。系统设置不是最佳时，即使优化很好的应用程序也会运行不佳。

备注

系统设置不正确是常见的错误，因此请确保予以检查。如果无法更改这些设置，请通知您的系统供应商。

检查缓存是否打开

现代系统都使用缓存来增强性能。如果系统中没有开启缓存，性能会大大削减。

检查应用处理器和 GPU 时钟设置是否正确

如果应用处理器和 GPU 的时钟设置不正确，则任何应用程序都无法以最高性能运行。此外，如果应用处理器或 GPU 的时钟设置得太高，系统的功耗可能会非常高。

检查应用处理器和 GPU 是否处于全功率模式

应用处理器和 GPU 都拥有低速、低功耗模式，能够节省处理器未在使用时的功耗。对于性能较高的应用程序，请确保处理器处于全功率模式，以获得最高的性能。

确保 GPU 时钟没有根据应用处理器负载而缩放

如果 GPU 时钟根据应用处理器负载而缩放，应用程序的性能可能会受到影响。这是因为应用处理器和 GPU 忙碌的时间可能会不同。如果 GPU 忙碌而应用处理器不忙，则降低 GPU 的时钟会降低性能。

独立控制应用处理器和 GPU 的时钟可以修复此问题。

2.5 最终发布检查表

表 2-1 列出了在发布应用程序前您可以快速检查的项目。

表 2-1 最终发布检查表

检查	其他信息
是否已启用缓存？	请参见*检查系统设置*（第 2-11 页）。
是否已关闭调试？	请参见*删除调试信息*（第 2-4 页）
是否已删除了停止流水线的调用？	请参见*避免停止图形流水线的调用*（第 2-5 页）
是否已启用背面剔除？	请参见*通过剔除减少绘制表面*（第 10-10 页）和*避免过度绘制*（第 9-2 页）
是否启用了纹理映射？	请参见*使用纹理映射*（第 2-7 页）
是否使用了压缩纹理？	请参见*使用纹理压缩*（第 2-7 页）
是否启用了 VSYNC？	请参见*使用 VSYNC*（第 2-6 页）
是否使用了最新的工具？	请参见*使用具有正确设置的正确工具*（第 2-4 页）
工具的配置是否正确？	
是否生成了优化的二进制文件？	

第 3 章
优化流程

本章介绍优化流程。其中包含下列小节：

优化流程涉及进行性能测量，识别瓶颈，并运用适当的技巧来消除瓶颈。

备注

第 5 章 *优化工作流程*中提供了如何使用此流程的示例。

3.1 优化流程中的步骤

本节介绍优化流程中的步骤。其中包含下列小节：

3.1.1 关于优化流程

优化流程涉及识别应用程序中的瓶颈，并运用各种技巧来消除瓶颈。

优化流程中包含多个步骤：

1. 从应用程序获取性能读数。
2. 分析读数，查找瓶颈。
3. 确定适当的优化类型。
4. 选择并应用优化。
5. 获取性能读数，确保优化发挥作用。

这些步骤显示在图 3-1 中。

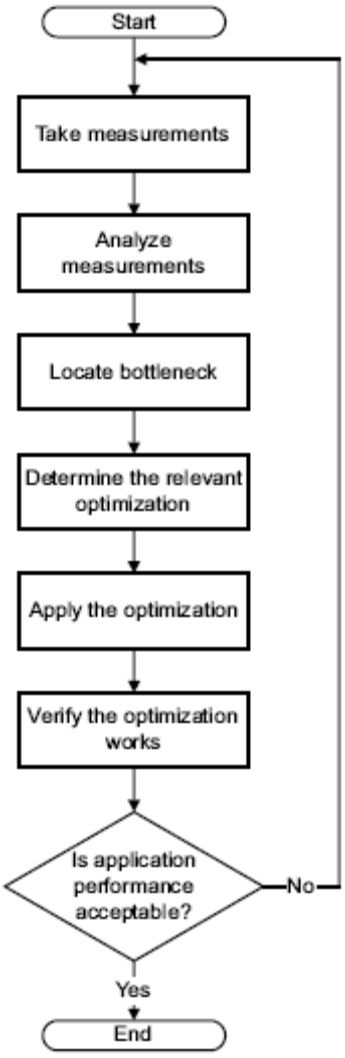

图 3-1 优化流程步骤

备注

优化流程可以显露不同区域中一系列不同的瓶颈，因此可能需要多次执行这一流程，才能将瓶颈全部消除。请参见*瓶颈在处理器之间移动*（第 3-9 页）。

3.1.2 进行测量

要进行优化，第一步是从应用程序进行测量。它们可以让您判断问题区域。在测量时请遵守下列规则，确保测量准确无误：

- 仅从配有 Mali GPU 的硬件设备进行测量。只有真实的硬件才能提供准确的性能测量数据。
- 确保在测量时关闭 VSYNC。如果启用，结果可能会不准确。

您可以使用 DS-5 Streamline 从 Mali GPU 计数器中获取读数，并在应用程序运行时记录相关的数据。也可以使用其他工具收集性能信息。

如需更多信息，请参见第 4 章 *测量与查找瓶颈*。

3.1.3 查找瓶颈

要查找降低性能的瓶颈，您必须分析测量结果。您可以使用工具来帮助进行分析：

DS-5 Streamline

DS-5 Streamline 以图表加时间线形式显示来自 Mali GPU 和应用处理器的计数器值。请参见*利用 DS-5 Streamline 查找瓶颈*（第 4-6 页）。

其他工具 也可以使用其他工具进行性能测量。测量结果的显示根据所用工具的不同而异。请参见*利用其他工具查找瓶颈*（第 4-13 页）。

您可以使用图表和其他数据显示查找性能瓶颈。找到瓶颈时，您可以：

- 进行额外测量，隔离确切问题区域。
- 应用一种或多种优化。

如需更多信息，请参见第 4 章 *测量与查找瓶颈*。

3.1.4 确定优化

要应用的优化因瓶颈不同而异。您可能找不到瓶颈的确切原因，但可以找到其影响最大的地方。通常，应用程序会在以下方面之一受到约束：

- 应用程序代码。
- 误用 API。
- 使用阻止 API 调用。
- 顶点处理。
- 三角形设置。
- 片段处理。
- 内存带宽。

如需更多信息，请参见*优化列表*（第 4-19 页）。

3.1.5 应用优化

应用优化可能会涉及修改应用程序代码和图形资源。可以从 *Mali 开发者中心*下载工具来帮助您进行此流程中的部分工作，其网址为 http://malideveloper.arm.com。

3.1.6 验证优化

优化可能不会总是获得预期的结果。利用已应用的优化重新运行该应用程序，对优化进行验证。

有可能某种优化对应用程序性能影响甚微。这可能意味着：

- 应用程序中存在限制性能的其他瓶颈。
- 测量结果有误导性，造成应用了错误的优化。难以测量真正的瓶颈时，可能会出现这一情况。

如果帧时间的差异很小，请考虑进行更多测量并进行分析。

3.1.7 重复优化流程

优化流程可以显露一系列不同的瓶颈。您可能必须多次执行该流程，才能全部消除这些瓶颈，并将性能提高到所需的水平。

在重复优化流程时，您可能会发现新的瓶颈。在优化系统的一个部分时，其他区域中可能会出现新的瓶颈。如需更多信息，请参见*瓶颈在处理器之间移动*（第 3-9 页）。

3.2 常规优化建议

本节介绍一般的优化建议。其中包含下列小节：

- *利用不同方式进行试验*。
- *使用帧时间而不是 FPS 进行比较*。
- *设定计算预算并据此进行测量*（第 3-7 页）。
- *瓶颈在处理器之间移动*（第 3-9 页）。

3.2.1 利用不同方式进行试验

不同的 GPU 实施具有不同的资源，可能使用各种不同版本的 Mali GPU 驱动程序。这些差异以不同的方式影响性能，因此务必要利用不同的图形编程和优化方式进行试验，从而获得最高的性能。

不同的应用程序对优化有不同的反应方式。具体的优化在一个应用程序中可能对性能产生显著的影响，在其他应用程序中可能影响甚微或毫无影响。

进行优化时，不要假设所有的优化始终都能提高性能。图形流水线中包含多个组件和不同资源，它们都可能成为瓶颈。不解决瓶颈的优化在当前的瓶颈解决之前不会有任何作用。

优化之间常常会存在取舍，所以要使用不同的技巧进行试验，了解哪一优化最适合您的应用程序。

3.2.2 使用帧时间而不是 FPS 进行比较

每秒帧数 (FPS) 是一项简单和基本的性能测量指标，但对于优化效用而言，帧时间更佳。

帧时间是线性测量指标，每秒帧数则是非线性的。线性测量指标让计算变得更加容易。

图 3-2 显示每秒帧数与帧时间的对比。此图表显示了 FPS 测量指标的非线性本质。

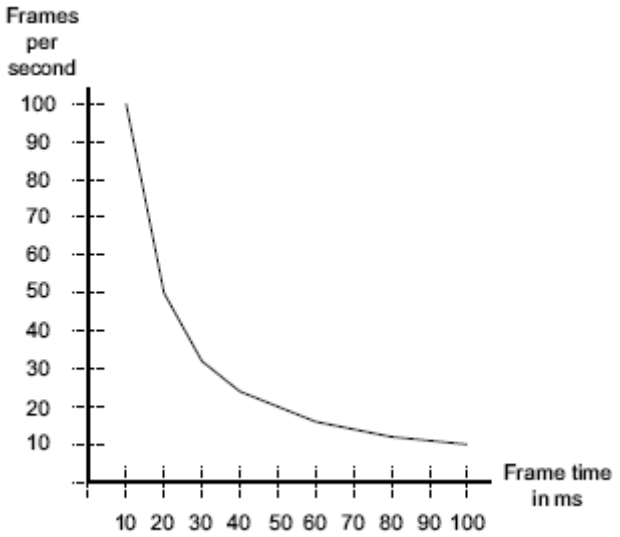

图 3-2 帧时间和 FPS

如果知道与不同优化对应的各个时间变化，您可以将这些时间加起来，获得总的改善结果。

如果使用 FPS 作为测量指标，您无法将结果相加，因为它们不是线性的。尝试将它们相加会得出不正确的总数。

表 3-1 显示不同 FPS 测量指标 A 和 B 之间的一系列比较。

每次测量 FPS 变化量都不同，而每次帧时间变化量却是相同的。

例如，从 100 FPS 到 200 FPS 涉及 100 FPS 或 5ms 的差值。但从 20 FPS 到 22.2 FPS，其差值为 2.2 FPS，时间却依然是 5ms。在测量优化的影响时，帧时间的线性本质更加容易处理。

表 3-1 每秒帧数和帧时间的区别

FPS 变化	FPS 差值	帧时间差值
20 到 22.2	2.2	5ms
50 到 66.6	16.6	5ms
100 到 200	100	5ms

3.2.3 设定计算预算并据此进行测量

处理器中存在您无法超越的最高性能限制。如果将应用程序所做的计算与最高值进行对比，您可以发现应用程序尝试做的计算是否太多。

设定可以作为测量基准的计算预算很有用。可用的确切预算取决于不同的因素，例如：

- 平台中 GPU 的类型。
- GPU 的配置。
- 可用的内存带宽。
- 色深。
- 图像分辨率。
- 所需的帧率。您可以为

以下项目设置预算：

三角形数　　每帧的三角形数上限。

应用处理器周期数

应用程序逻辑和驱动程序中花费的时间，以时钟周期数表示。

顶点处理器周期数

可用顶点着色器的平均长度，以周期数表示。

片段处理器周期数

可用像素着色器的平均长度，以周期数表示。

确保在计算这一项时考虑了过度绘制。过度绘制的系数通常为 2.5 倍，因此请将平均长度除以 2.5。

内存带宽

内存带宽包含写入内存或从中读取的任何数据。这包括：

- 纹理数据的每像素字节数。
- 属性数据类型的大小。
- 顶点数。
- 位块传送。
- 向帧缓冲区写入或从中读取的次数。

计算片段着色器预算

得出片段着色器预算的计算为：

1. Mali GPU 片段处理器数乘以 Mali GPU 时钟速度。
   这可得出每秒片段处理器周期数的理论最大值。

   将结果乘以 0.8，得出更为实际的每秒可用片段处理器周期数。此为结果 A。

2. 帧高度乘以帧宽度。这会得出每帧的像素数。

   将此结果乘以所需的帧率。这会得出每秒所需的像素数。

   为了将平均过度绘制考虑在内，将此值乘以 2.5。这会得出每秒所需的片段数。此为结果 B。

3. 将结果 A 的值除以结果 B 的值。

   得出的结果是每个片段着色器可以有的平均周期数。

您不必让所有片段着色器都这么长。例如，您可以对距离摄像头较近的对象使用较长、较复杂的着色器，对距离较远的对象使用较短、较不复杂的着色器。

您可以使用着色器编译器判断着色器需要的周期数。请参见*来自其他 Mali GPU 工具的测量*（第 4-13 页）。

备注

不要假设片段处理器周期数与片段处理器指令数相等。Mali GPU 中的处理器每个周期可以进行许多运算。

3.2.4 瓶颈在处理器之间移动

本节介绍瓶颈如何在处理器之间移动，以及理想应用程序的性能分析。其中包含下列小节：

- *瓶颈如何在处理器之间移动*。
- *理想应用程序性能分析*（第 3-10 页）。

瓶颈如何在处理器之间移动

在应用优化时，应用程序的性能瓶颈可能会在不同处理器之间移动。分析工具中的读数可以告诉您瓶颈有可能移到哪个位置，以及处理器是否利用不足。

备注

在 DS-5 Streamline 中，可以在图表显示中直接看到瓶颈。瓶颈就是最繁忙的图表。

图 3-3 显示应用程序运行时系统中的不同部分的帧率柱形图。

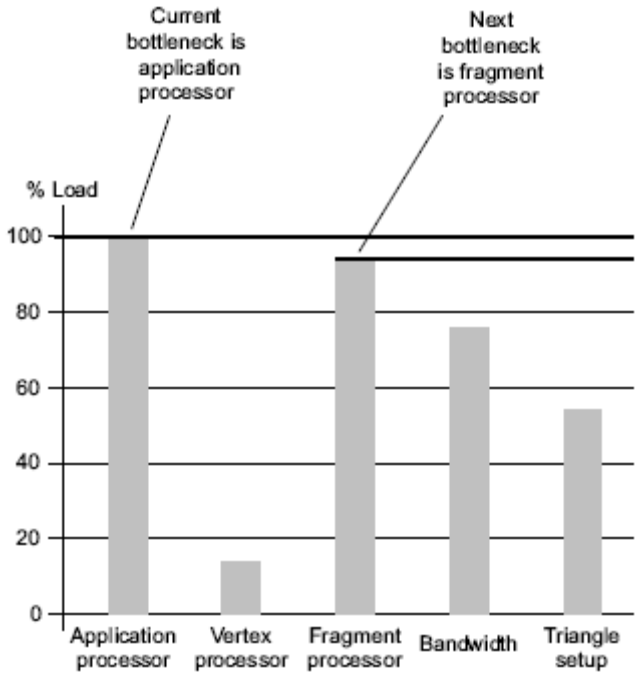

图 3-3 不同系统元素的帧率限制

比较这些柱形表明：

- 如果您优化性能瓶颈，下一个瓶颈是第二低柱形的组件。在这一情形中，瓶颈为应用处理器，下一瓶颈为片段处理器。
- 顶点处理器的图形的值远高于总体 GPU 帧率。这表明瓶颈不在顶点处理器上。差值很大也表明其利用不足。

如果处理器有空余的处理容量，请思考是否有任何处理运算可以移到该处理器上。例如，您或许可以将应用处理器或片段处理器的运算移到顶点处理器上。

理想应用程序性能分析

理想的应用程序受到各个元素的限制几乎是相等的。图 3-4 等柱形图表示该应用程序充分利用了所有元素。

在这种情形中，一种优化可能无法对性能造成很大影响，您需要多种优化才能获得更高、更稳定的帧率。

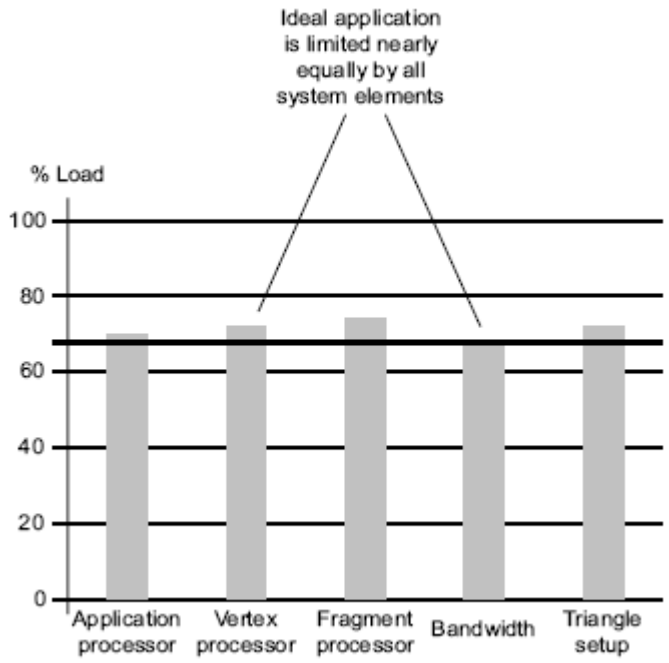

图 3-4 受到相等限制的理想应用程序

第 4 章

测量与查找瓶颈

本章介绍如何测量您的应用程序，以及如何查找性能瓶颈。其中包含下列小节：

4.1 关于测量与查找瓶颈

本章介绍测量与查找瓶颈的步骤。

此处所述的步骤主要以使用 DS-5 Streamline 进行测量为基础。您可以使用 DS-5 Streamline 的免费社区版进行测量。要获得该社区版，请访问 *Mali 开发者中心*，网址为 http://malideveloper.arm.com。

也可以使用其他工具来进行测量和解读测量结果。相关的技巧也进行了说明。您可以使用其他工具代替 DS-5 Streamline，或者作为额外的信息来源。

4.2 测量与查找瓶颈的步骤

要进行测量并查找瓶颈，请使用下列步骤：

进行首次测量并以图表形式查看

使用分析工具进行首次测量。先测量最重要的计数器，让您对可能存在性能瓶颈的位置有个概念。首先测量下列计数器：

- GPU 顶点活动。
- GPU 片段活动。
- <*应用处理器*> 指令：已执行数。

以图表形式查看计数器值

使用分析工具将计数器值绘制为图表。DS-5 Streamline 自动将计数器值绘制为图表加时间线。

通过比较图表隔离问题区域

您可以比较不同组件的图表，找出系统中最繁忙的部分。

如何比较图表取决于您使用的具体工具。如需更多信息，请参见*分许图表*（第 4-5 页）。

深入研究以找到确切问题

确定了系统中最繁忙的部分后，您可以进行进一步测量来隔离确切的问题区域。

4.3 测量

本节介绍测量。其中包含下列小节：

- *准确测量*。
- *首先测量的 Mali GPU 计数器*。

4.3.1 准确测量

要确保测量始终准确，请执行下列操作：

从设备进行性能测量

要准确测量，务必从设备进行性能测量。任何模拟或近似的方式都可能会产生误导性的测量结果。

例如，您或许能够使用桌面工作站进行开发，但此系统的相对优势很可能与移动设备大相径庭。

确保已禁用 VSYNC

确保在测量时已禁用 VSYNC。启用 VSYNC 时进行测量会产生不准确的结果。

4.3.2 首先测量的 Mali GPU 计数器

要找到可能存在瓶颈的位置，请先测量最重要的计数器。分析工具可以从 GPU 中的硬件计时器获取数据，并以图表形式显示结果。

表 4-1 显示最重要的 Mali GPU 计数器。

表 4-1 要首先查看的 Mali GPU 计数器

测量	DS-5 Streamline 中的计数器	备注
总帧率	-	来自您的应用程序
顶点处理器	GPU 顶点活动	-
片段处理器	GPU 片段活动	-
栅格化的片段数	Mali GPU 片段处理器 *X*：栅格化片段数	-
着色器指令	Mali GPU 片段处理器 *X*：已完成指令数	已执行的着色器指令
纹理缓存命中数	Mali GPU 片段处理器 *X*：纹理缓存命中数	-
纹理缓存未命中数	Mali GPU 片段处理器 *X*：纹理缓存未命中数	-
OpenGL ES 绘制调用数	glDrawElements 统计信息：对 glDrawElements 的调用数	-
	glDrawArrays 统计信息：对 glDrawArrays 的调用数	

备注

DS-5 Streamline 不直接测量帧率，但您可以间接测量。请参见 *分析 DS-5 Streamline 中的图表* （第 4-9 页）。

4.4 分析图表

DS-5 Streamline 和其他工具能以图表形式显示计数器数据。这提供了一种简单的视觉方式来识别瓶颈。

分析图表时请注意下列一般规则：

- 首先测量并绘制最重要的计数器。它们可以让您了解有可能存在性能瓶颈的位置。
- 图表可以通过太高、太低或覆盖较大区域来指出问题。确切的诊断取决于您测量的计数器，以及分析工具显示图表的方式。
- 图表可能是变化不定的。不同处理器的相对性能可能会在帧与帧之间发生变化。
- 查看较长时段内的平均值，找出可改善总体性能的位置。
- 确定了使用最为密集的处理器后，下一步是通过更多测量确定问题。
- 如果想要优化具体的场景，请逐个帧查看性能。
- 您可能无法通过查看图表来完全隔离问题。若是如此，您可以使用其他技巧来找出确切的问题。请参见*利用其他工具查找瓶颈*（第 4-13 页）。

4.5 利用 DS-5 Streamline 查找瓶颈

本节介绍如何使用 DS-5 Streamline 分析数据和查找瓶颈。其中包含下列小节：

- *关于 DS-5 Streamline*。
- *DS-5 Streamline 中的 GPU 计数器*（第 4-8 页）。
- *分析 DS-5 Streamline 中的图表*（第 4-9 页）。
- *DS-5 Streamline 显示片段处理器使用量高*（第 4-10 页）。
- *DS-5 Streamline 放大显示*（第 4-11 页）。
- *DS-5 Streamline 显示函数列表*（第 4-12 页）。

备注

DS-5 Streamline 需要启用了性能测量的 Mali GPU 驱动程序。激活 Mali GPU 驱动程序中的性能测量对正确编写的应用程序的性能影响可忽略不计。

4.5.1 关于 DS-5 Streamline

DS-5 Streamline 这款工具可以提供有关应用程序运行状况的信息。您可以使用 DS-5 Streamline 实时收集应用处理器和 Mali GPU 中性能计数器的数据。DS-5 Streamline 将计数器数据显示为一系列图表。

您可以将 DS-5 Streamline 用于：

- 从应用处理器和 Mali GPU 采集计数器数据。
- 保存采集的数据以便回放显示。
- 查看时间线，其中显示
  - 一段时间的 GPU 活动。
  - 每个进程的 GPU 活动。
  - 帧缓冲区中随时间的变化。
- 在图表和表格中显式各个性能计数器的值。
- 观察这些值如何随时间改变。
- 评估各个帧的性能。
- 查看处理器活动的图表。
- 查看堆栈轨迹。
- 查看应用程序性能分析。

您可以自定义 DS-5 Streamline，以读取和显示不同计数器中的数据。您可以相互比较这些图表，以便判断主导性能的因素和可能存在性能瓶颈的位置。

第 4-7 页上的图 4-1 显示 DS-5 Streamline 中显示了来自多个不同计数器的图表。

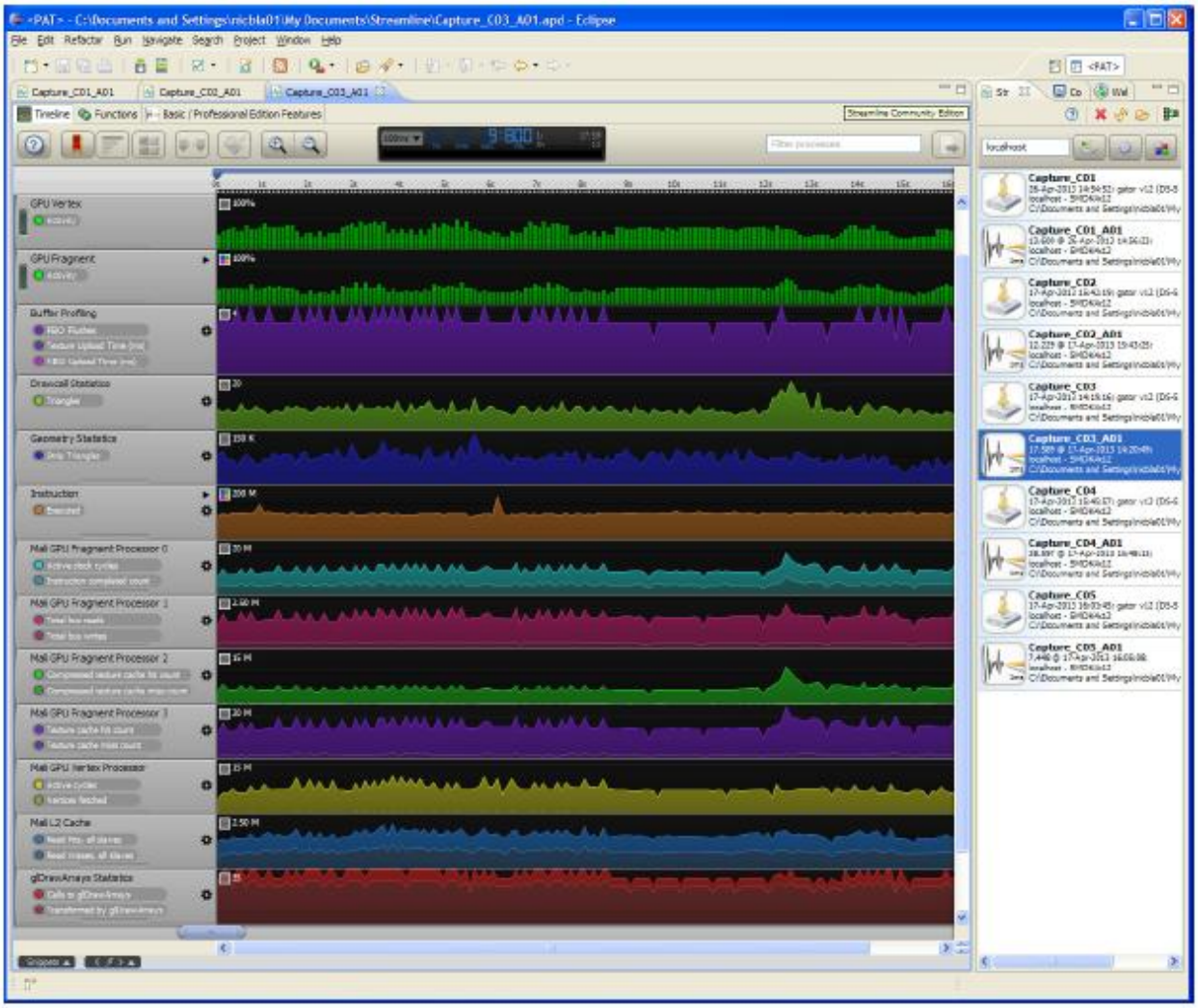

图 4-1 DS-5 Streamline

4.5.2 DS-5 Streamline 中的 GPU 计数器

DS-5 Streamline 从应用处理器和 Mali GPU 中的硬件计数器采集数据。它将这些结果显示为图表。

图 4-2 显示**计数器配置**窗口，其中您可以选择要在 DS-5 Streamline 中记录和显示的计数器。

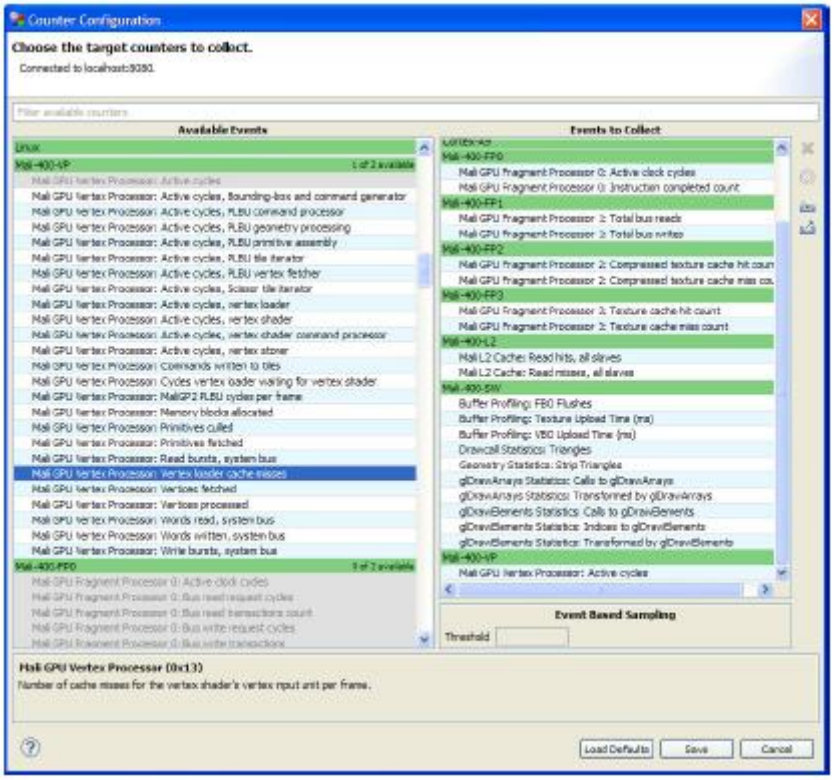

图 4-2 DS-5 Streamline 计数器

也可以添加显示值之间关系的虚拟计数器。例如：

- 缓存命中与未命中比。
- 根据定义条件生成值的触发器。如需更多信息，请参

见 DS-5 Streamline 文档。

4.5.3 分析 DS-5 Streamline 中的图表

您可以在 DS-5 Streamline 中查看如下项目：

- 应用程序使用的所有处理器的活动图表。
- 显示为调用图表的堆栈轨迹。
- 本机应用程序性能分析。

您可以从 DS-5 Streamline 中的图表进行下列观察：

- 短期任务产生较小的活动高峰。
- 密集任务产生长时间的处理器高使用量。
- 哪一处理器在什么时间上处于活动状态。

备注

DS-5 Streamline 不直接测量帧率。您可以直接在应用程序中测量，或者在 DS-5 Streamline 中通过如下方式间接测量：

- 查找 EGLswapbuffers 等 API 调用。
- 查找处理器图表中的重复模式。

分析 DS-5 Streamline 中的图表：

1. 查看下列图表：
   - GPU 顶点活动。
   - GPU 片段活动。
   - <*应用处理器*> 指令：已执行数。
2. 分析图表：
   - 查找图形最高、最长的处理器。此处理器的使用最为密集。
   - 如果难以找出用时很长的单个处理器，则问题可能在于带宽使用过量或图形流水线停滞。
   - 确定了使用最为密集的处理器后，进行更多测量来隔离问题。请参见第 5 章 *优化工作流程*。
   - 如果所有图表都显示忙碌，则您的应用程序正在充分利用 Mali GPU。

备注

为获得最准确的测量，请放大到 EGLSwapBuffers() 调用，使用卡尺来隔离帧。

4.5.4 DS-5 Streamline 显示片段处理器使用量高

图 4-3 显示 DS-5 Streamline 中所显示的多个图表。您可以看到，Mali GPU 片段处理器和绘制调用统计图表的窗口中部有一个活动高峰。其他图表则比较平坦。

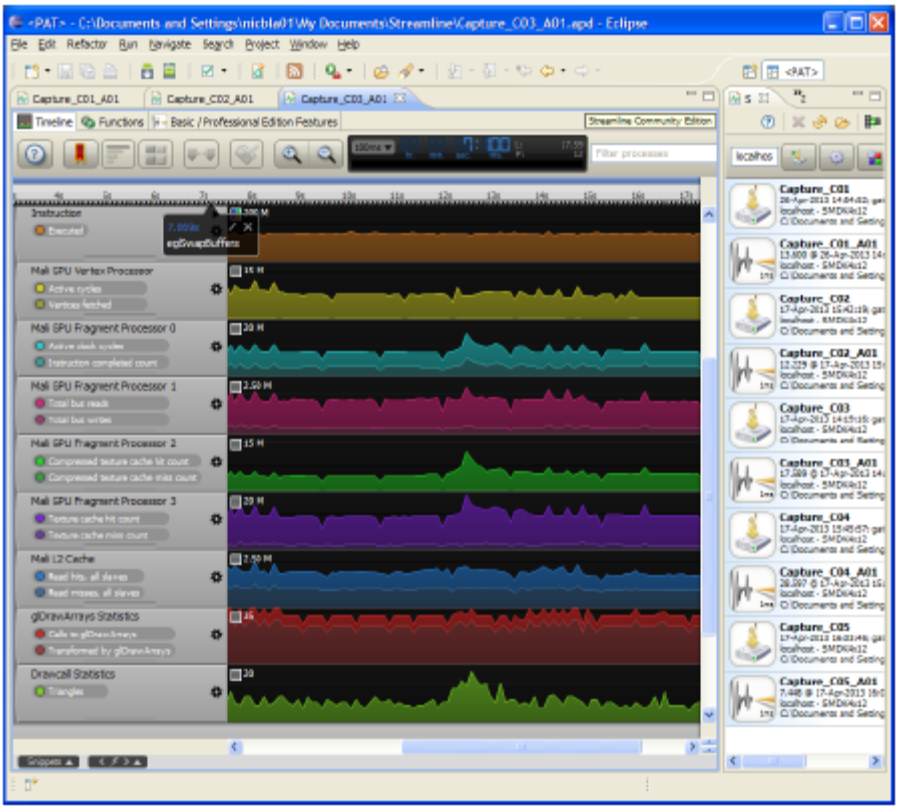

图 4-3 片段处理器使用量高

4.5.5 DS-5 Streamline 放大显示

图 4-4 显示 DS-5 Streamline 中的放大画面。活动高峰属于各个帧。

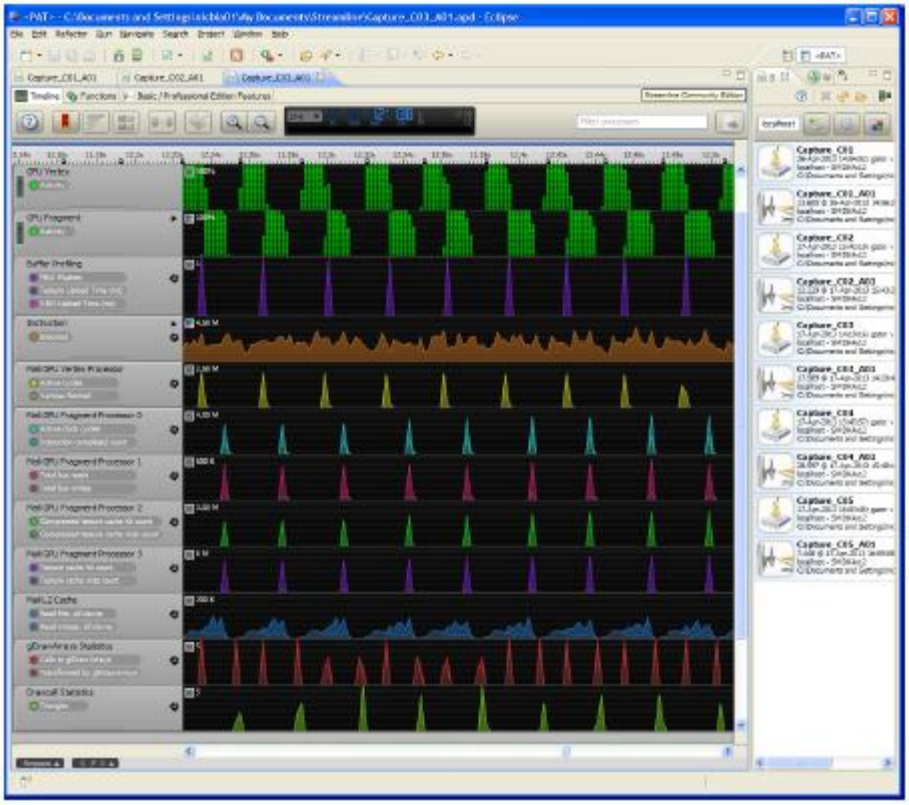

图 4-4 片段处理器使用量高放大画面

4.5.6 DS-5 Streamline 显示函数列表

图 4-5 显示 DS-5 Streamline 中显示了函数列表和使用情况统计。

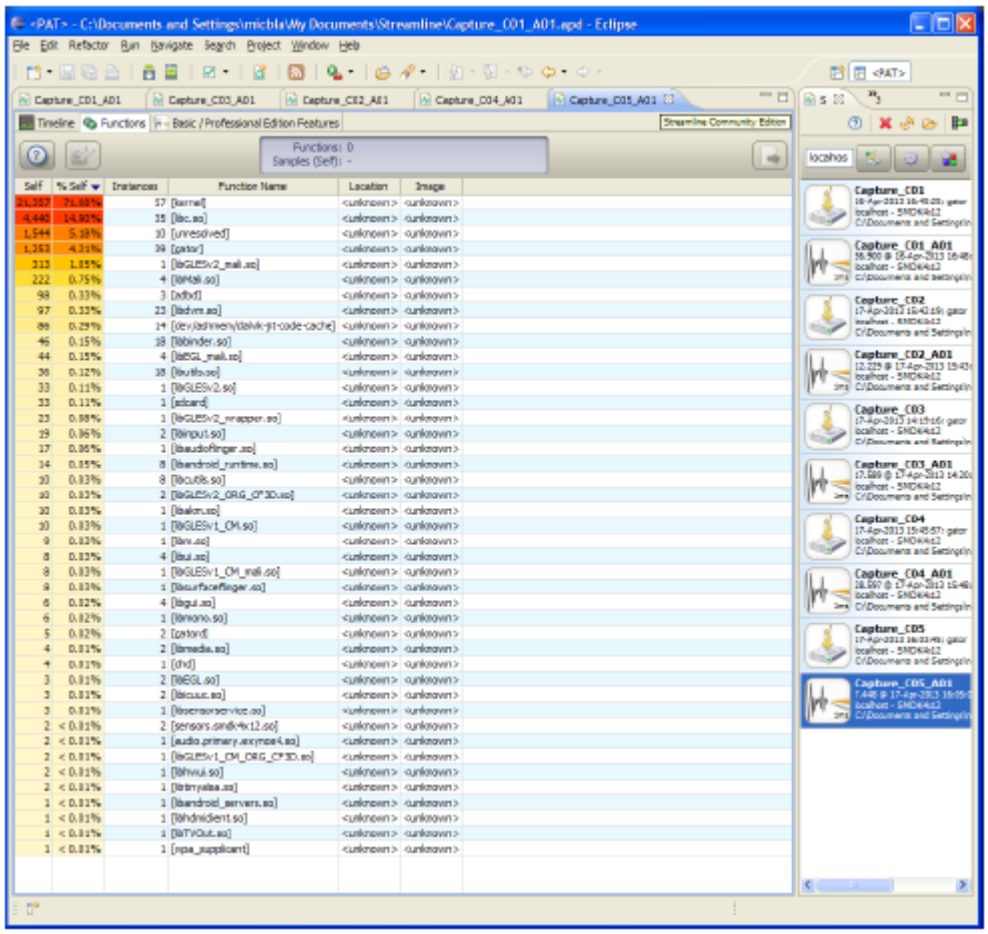

图 4-5 DS-5 函数列表

4.6 利用其他工具查找瓶颈

本节介绍如何使用其他工具和 DS-5 Streamline 之外的信息来源查找瓶颈。其中包含下列小节：

- *不使用分析工具进行测量*。
- *来自其他 Mali GPU 工具的测量*。
- *来自调试工具的信息*（第 4-14 页）。
- *通过比较查找问题区域*（第 4-14 页）。
- *通过比较查找问题区域的技巧*（第 4-14 页）。

4.6.1 不使用分析工具进行测量

如果无法访问 DS-5 Streamline 或启用了性能测量的 Mali GPU 驱动程序，则难以进行准确的测量。不过，只要您的应用程序中存在帧时间或帧率计数器，依然可以获取有用的信息。

备注

如果无法访问启用了性能测量的 Mali GPU 驱动程序，请联系您的设备供应商。

进行测量：

1. 在应用程序中激活帧率测量。
2. 在应用程序中运行具有代表性且完全可重复的序列。
3. 修改应用程序。
4. 在应用程序中运行同一序列，并进行测量。
5. 通过不同的修改，重复第 3 和第 4 步。

通过在应用程序中进行更改并重新运行完全相同的序列，您可以获得应用程序不同区域的一系列测量结果。

备注

要确定进行更改的类型，请参见*通过比较查找问题区域的技巧*（第 4-14 页）。

进行更改后，测量中的帧率可能会在某些情形中保持相似结果，而在其他情形中呈现不同的结果。

- 如果帧率变化很小，则应用程序不太可能在这一区域受到约束。
- 如果帧率变化很大，则应用程序最有可能在这一区域受到约束。

4.6.2 来自其他 Mali GPU 工具的测量

您可以使用 Mali GPU 软件工具获取测量结果和其他有用信息。

Mali GPU Offline Shader Compiler 中的 -v 选项可提供有关顶点和片段着色器的信息。

例如，下列命令的输出结果显示 *my_shader.frag* 所要执行的最小和最大周期数（假设没有缓存未命中）。

malisc -v *my_shader.frag*

您可以使用这些数据算出每秒可以着色的顶点数或片段数最大值。这在使用片段着色器预算时很有用处。请参见*设定计算预算并据此进行测量*（第 3-7 页）。

Mali GPU Offline Shader Compiler 和其他工具在 *Mali 开发者中心*提供，网址为 http://malideveloper.arm.com。

4.6.3 来自调试工具的信息

您可以从 OProfile 等性能分析工具中判断应用处理器的利用情况。性能分析工具可以提供有关应用程序行为的信息，还可区分应用程序和驱动程序的使用情况。这些区分可以让您得出问题是出在应用程序逻辑中还是对 OpenGL ES API 的使用中。

静态代码分析工具可以识别代码是否复杂。复杂的代码更有可能执行缓慢，也容易出错。

4.6.4 通过比较查找问题区域

您可以通过将代码或资源类型替换为不使用计算或内存带宽资源的版本，查找确切的问题区域。很大的性能差异表明问题所在区域。

例如，下列过程说明如何查找问题着色器：

1. 空着色器是不执行任何操作的着色器。将所有着色器替换为空着色器，并测量之前和之后的性能差别。如果没有性能差别，则问题不在着色器上。如果差别很大，则存在与着色器相关的问题。
2. 将着色器分为 A、B 两半，重新测试。
   a. A 部分作为空着色器，B 部分作为原始着色器，进行测试。
   a. A 部分作为原始着色器，B 部分作为空着色器，进行测试。
   c. 比较结果。
3. 判断性能影响最大的那一半。将这一半分为两半，再重复第 2 步。继续这一过程，直到您找到具体的问题着色器。
4. 优化该着色器，然后全部使用原始着色器再次测量性能。
   如果依然有问题，则可能存在其他问题着色器。继续重复此过程，直到您优化了所有问题着色器。

备注

这一过程中务必要有条理。一次调查一种类型的代码或资源。如果试图一次调查多种类型，您的测量可能会不准确。

4.6.5 通过比较查找问题区域的技巧

*通过比较查找问题区域*中所述的技巧可用于调查许多区域。这些技巧要求禁用代码和资源。此外，您也可以作如下尝试：

更改分辨率

更改应用程序的分辨率，测量帧率差别。
如果帧率变化与分辨率成反比，即分辨率加倍时性能减半，您的应用

程序受到片段处理器约束或者带宽限制。

如果帧率不变，则应用程序存在应用处理器约束或顶点处理器约束问题。测量 CPU 使用情况，以判断属于哪一种情况。

更改纹理大小

将纹理的大小改为 1 x 1。如果帧率提高，则纹理缓存命中率太低。这表明您的纹理太大，或者应用程序受到带宽限制。

使用存根驱动程序

存根驱动程序将 OpenGL ES 驱动程序替换为不执行任何操作的驱动程序。使用存根驱动程序时产生的帧率表明不使用驱动程序时的应用处理器使用性能。

如果使用存根驱动程序时帧率不变，则应用程序受到应用处理器约束。

如果帧率确实改变，则应用程序可能使该驱动程序负载过重。

缩短着色器长度

如果着色器过长，它可能会造成帧率降低。尝试较短的着色器，并测量变化。

如果帧率提高，则着色器长度可能尝试做得太多或太长。

备注

每个片段可用的周期数与帧大小和帧率成反比；也就是说，分辨率加倍或帧率加倍时，每个片段可用的周期数会减半。

使用空的片段着色器

空着色器可以指出您的应用程序是否受着色器约束。

空着色器不执行工作。将着色器替换为空着色器，再测量性能。如果性能大幅提高，则应用程序有可能受到着色器约束。

此项测试也会降低带宽用量，所以性能变化很大可能也表明带宽使用过量。

更改顶点数

使用 3D 场景中对象的简单版本，减少顶点的数量。如果性能差异很大，则表明应用程序可能使用过多数量的顶点，或者受到带宽约束。

更改纹理的位深

如果减小纹理的位深时应用程序性能会提高，则内存带宽可能是问题所在。

更改绘制表面的位深

如果减小表面位深时性能会提高，则内存带宽可能是问题所在。

如果此测试在某项具体设置上产生大幅性能提升，则系统设置可能不正确。

减少绘制调用

使用过多数量的绘制调用是常见的问题。尝试将相同数量的工作移到较少

的绘制调用中，查看性能是否改善。

减少状态更改

尝试减少状态更改，查看性能是否改善。

4.7 隔离具体问题区域

找到了瓶颈所处的一般区域后，下一阶段是确定瓶颈的具体原因。本节介绍如何隔离问题区域。其中包含下列小节：

- *应用程序受应用处理器约束*。
- *应用程序受顶点处理约束*。
- *应用程序受片段处理约束*（第 4-18 页）。
- *判断内存带宽是否为问题所在*（第 4-18 页）。

4.7.1 应用程序受应用处理器约束

应用程序可能会在以下方面存在应用处理器约束问题：

应用程序逻辑使用太多处理能力

要判断应用程序是否在应用程序逻辑中受应用处理器约束，请删除绘制调用和交换缓冲区命令，可以将它们替换为注释或使用存根驱动程序。如果性能变化很小或没有变化，则可能是应用程序逻辑中存在限制。

使用性能分析程序确定应用程序逻辑的哪些区域表现不佳，再优化相关代码。

请参见第 6 章 *应用处理器优化*。

应用程序使驱动程序负载过重

如果低分辨率输出测试表明瓶颈在应用处理器中，但应用程序逻辑不是问题所在，应用程序可能以欠优化的方式使用 OpenGL ES API，例如：

- 绘制调用太多。
- 状态更改太多。
- 流水线停滞。

请参见第 7 章 *API 级别优化*。

应用程序逻辑和驱动程序负载过重的组合

有时候，应用程序不是在一个方面受到约束，而是两个方面的组合造成应用程序受到应用处理器约束。若是如此，您必须优化应用程序逻辑和 OpenGL ES API 使用。

备注

- 您可以使用 DS-5 或 OProfile 等性能分析程序区分应用程序和驱动程序开销。
- 这些只是用来判断应用程序是否受应用处理器约束的近似方式，可能不是始终都可靠。要更加准确地判断问题的来源，请执行完整的优化流程。

4.7.2 应用程序受顶点处理约束

如果应用程序受顶点处理约束，问题可能出现在下列方面之一：

- 顶点数量太多。
- 顶点着色器太长。
- 顶点着色器太复杂。
- 三角形设置时间较长。

- *多边形列表构建器单元* (PLBU) 时间较长。

请参见*顶点处理器约束问题*（第 5-11 页）。

4.7.3 应用程序受片段处理约束

如果应用程序受片段处理约束，问题可能出现在下列方面之一：

片段处理约束问题：

问题出在片段处理中。问题通常为：

- 过度绘制较高。
- 纹理读取数太多。
- 纹理缓存未命中率较高。

片段着色约束问题：

问题出在着色器中。问题通常为：

- 着色器太长。
- 着色器太复杂。
- 着色器太长、太慢。
- 着色器有太多分支。

请参见第 9 章 *片段处理优化*。

4.7.4 判断内存带宽是否为问题所在

内存带宽影响一切，难以直接测量。所以也难以诊断其是否为瓶颈。

带宽使用过量可以表现为处理器中的其他限制。如果其中一个处理器限制了性能，但优化似乎没有作用，则问题可能出在带宽上。

如果应用程序受带宽约束，则问题最有可能是：

- 纹理。
- 过度绘制。

请参见*带宽约束问题*（第 5-23 页），另请参见第 10 章*带宽优化*。

4.8 优化列表

本节列举本指南中介绍的各种优化，并注明它们所适用的系统部分。其中包含下列小节：

- *应用程序处理优化列表*。
- *API 优化列表*。
- *顶点处理优化列表*（第 4-20 页）。
- *片段处理优化列表*（第 4-20 页）。
- *带宽优化列表*（第 4-21 页）。
- *其他优化列表*（第 4-22 页）。

在确定了问题原因后，您必须应用优化以进行修复。使用本节中的列表，确定要使用的优化。

4.8.1 应用程序处理优化列表

表 4-2 显示本指南中列出的应用程序处理优化。

表 4-2 本指南中列出的应用程序处理优化

优化
使用具有正确设置的正确工具（第 2-4 页）
删除调试信息（第 2-4 页）
确保应用程序没有应用处理器约束问题（第 2-8 页）
检查系统设置（第 2-11 页）
对齐数据（第 6-2 页）
优化循环（第 6-3 页）
使用矢量指令（第 6-5 页）
使用快速数据结构（第 6-6 页）
考虑替代算法和数据结构（第 6-7 页）
使用多处理（第 6-8 页）
使用近似（第 11-2 页）

4.8.2 API 优化列表

表 4-3 显示本指南中列出的 API 优化。

表 4-3 本指南中列出的 API 优化

优化
检查显示设置（第 2-3 页）
尽可能使用直接渲染（第 2-3 页）
避免无限命令列表（第 2-5 页）
避免停止图形流水线的调用（第 2-5 页）
不要每帧编译着色器（第 2-6 页）

表 4-3 本指南中列出的 API 优化（续）

优化
使用 VSYNC（第 2-6 页）
使用顶点缓冲区对象（第 2-8 页）
尽可能减少绘制调用（第 7-2 页）
尽可能减少状态更改（第 7-7 页）
确保图形流水线保持运行（第 7-8 页）

4.8.3 顶点处理优化列表

表 4-4 显示本指南中列出的顶点处理优化。

表 4-4 本指南中列出的顶点处理优化

优化
使用 VSYNC（第 2-6 页）
使用平台适用的图形资源（第 2-6 页）
减少内存带宽使用（第 2-7 页）
使用顶点缓冲区对象（第 2-8 页）
检查系统设置（第 2-11 页）
优化循环（第 6-3 页）
减少顶点数量（第 8-2 页）
使用剔除（第 8-3 页）
使用法线贴图模拟精细几何（第 8-5 页）
使用细节级别（第 8-6 页）
避免过度绘制（第 9-2 页）
使用近似（第 11-2 页）

4.8.4 片段处理优化列表

表 4-5 显示本指南中列出的片段处理优化。

表 4-5 本指南中列出的片段处理优化

优化
使用 VSYNC（第 2-6 页）
使用平台适用的图形资源（第 2-6 页）
不要使用 24 位纹理（第 2-6 页）
使用纹理映射（第 2-7 页）
使用纹理压缩（第 2-7 页）

表 4-5 本指南中列出的片段处理优化（续）

优化
减少内存带宽使用（第 2-7 页）
优化循环（第 6-3 页）
使用矢量指令（第 6-5 页）
使用细节级别（第 8-6 页）
减少纹理带宽（第 9-2 页）
避免过度绘制（第 9-2 页）
缩短着色器（第 9-4 页）
简化着色器（第 9-4 页）
减少分支数量（第 9-4 页）
其他片段着色器问题（第 9-5 页）
使用近似（第 11-2 页）

4.8.5 带宽优化列表

表 4-6 显示本指南中列出的带宽优化。

表 4-6 本指南中列出的带宽优化

优化
检查显示设置（第 2-3 页）
尽可能使用直接渲染（第 2-3 页）
使用 VSYNC（第 2-6 页）
使用平台适用的图形资源（第 2-6 页）
不要使用 24 位纹理（第 2-6 页）
使用纹理映射（第 2-7 页）
使用纹理压缩（第 2-7 页）
减少内存带宽使用（第 2-7 页）
使用顶点缓冲区对象（第 2-8 页）
使用细节级别（第 8-6 页）
避免过度绘制（第 9-2 页）
优化纹理（第 10-3 页）
使用纹理映射（第 10-5 页）
使用纹理压缩（第 10-6 页）
仅在必要时使用三线性过滤（第 10-8 页）
通过避免过度绘制减小带宽（第 10-9 页）

表 4-6 本指南中列出的带宽优化（续）

优化
通过剔除减少绘制表面（第 10-10 页）
利用细节级别减小带宽（第 10-11 页）
使用近似（第 11-2 页）

4.8.6 其他优化列表

有时，您可以通过将计算移到利用不足的资源进行优化。这是一种有用的优化技巧，可以充分地利用您的计算资源。

表 4-7 显示本指南中列出的其他优化。

表 4-7 本指南中列出的其他优化

优化
检查显示设置（第 2-3 页）
尽可能使用直接渲染（第 2-3 页）
使用具有正确设置的正确工具（第 2-4 页）
删除调试信息（第 2-4 页）
检查系统设置（第 2-11 页）
使用近似（第 11-2 页）
检查显示设置（第 11-5 页）
使用 VSYNC（第 11-8 页）
充分利用低效的资源（第 11-11 页）

请参见*充分利用低效的资源*（第 11-11 页）和*瓶颈在处理器间移动*（第 3-9 页）。

第 5 章
优化工作流程

本章提供如何查找和解决多个常见性能问题的示例。它将逐步指导您诊断问题，并选择优化以解决问题。

本章包含下列小节：

备注

本章未介绍如何查找和解决所有的性能问题；但是，您可以使用相似的流程查找其他问题。

5.1 关于优化工作流程

本章提供一系列流程图，逐步指导您诊断性能问题，并应用相关优化以修复问题。图 5-1 显示了总体流程图。标有颜色的方块表示您可以在流程图中采取的一个途径。

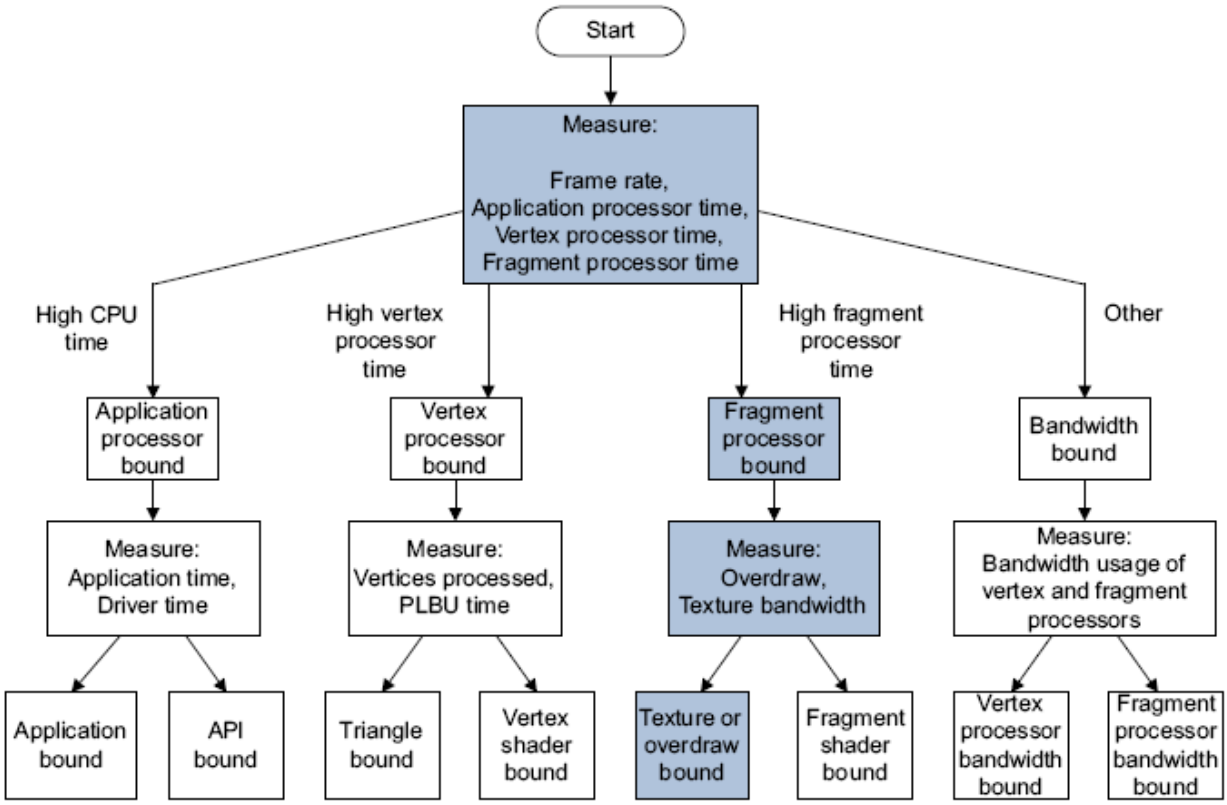

图 5-1 工作流程概述

下列的步骤基于第 3 章 *优化流程*中所述的流程：

1. 从一些基本测量开始。请参见*测量应用程序*（第 5-3 页）。
2. 比较结果，判断哪些区域需要额外调查。
3. 前往相关的章节，遵循所述的过程操作。每一小节中都包含显示相关工作流程的流程图。
4. 每一小节都会介绍您可以用于诊断问题的一系列测量，并建议解决问题的方式。此外还会提供更多信息的链接。
5. 完成这一流程时，测量应用程序以确保优化发挥作用。如果性能不够高，则再次执行该流程。要全面优化应用程序，您可能必须多次执行该流程。

5.2 测量应用程序

本节介绍如何对应用程序进行初步性能测量，以及如何判断需要额外测量的区域。其中包含下列小节：

- *对真实硬件进行测量*。
- *利用 DS-5 Streamline 进行测量*（第 5-4 页）。
- *判断问题区域*（第 5-5 页）。

图 5-2 显示最上层工作流程。

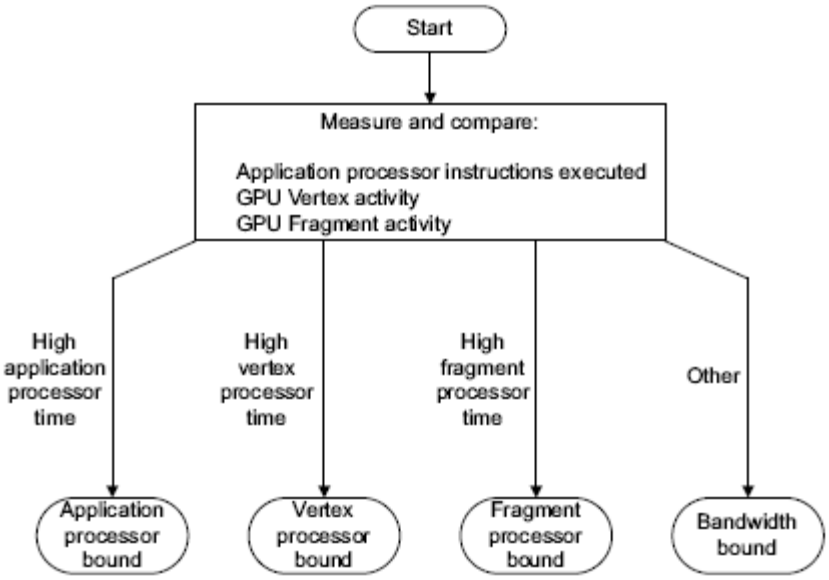

图 5-2 最上层工作流程

5.2.1 对真实硬件进行测量

您必须使用配有 Mali GPU 的硬件设备。只有真实的硬件才能提供准确的性能测量数据。

下列小节说明如何利用 DS-5 Streamline 测量真实的硬件。

您可以从 *Mali 开发者中心*下载 DS-5 Streamline，网址为 http://malideveloper.arm.com。

如果没有 DS-5 Streamline，您依然可以通过其他方式查找性能问题。请参见*利用其他工具查找瓶颈*（第 4-13 页）。

5.2.2 利用 DS-5 Streamline 进行测量

如果使用 DS-5 Streamline，您需要：

- 配有 Mali GPU 的硬件设备。
- 启用了性能测量的 Mali GPU 驱动程序。

要利用 DS-5 Streamline 测量性能，请执行下列操作：

1. 将设备连接到您的工作站，进行初始测量。
2. 查看下列图表：
   - GPU 顶点活动。
   - GPU 片段活动。
   - <*应用处理器*> 指令：已执行数。
3. 相互比较这些图表。查找活动最多、用时最多的处理器。

图 5-3 显示 DS-5 Streamline，其中为应用处理器、顶点处理器和片段处理器绘制了一系列图表图形。

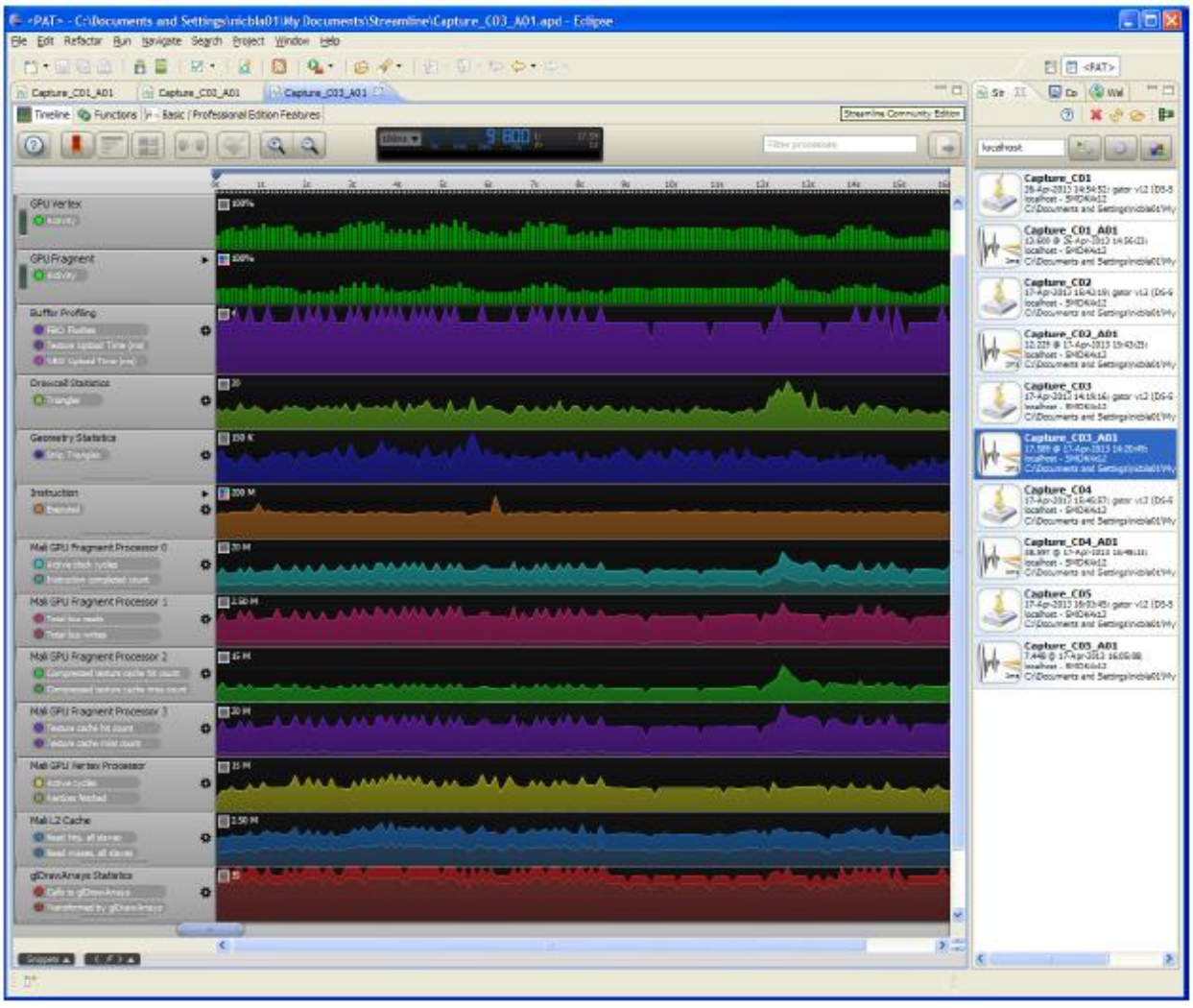

图 5-3 DS-5 Streamline

5.2.3 判断问题区域

要判断问题区域，您必须找出系统的哪一部分速度最慢或用时最长。具体方式取决于您所使用的测量工具：

- DS-5 Streamline 显示的图表说明了不同处理器的活动。您可以比较不同的图表，直接查看活动最多的处理器。

  如果某一部分运行缓慢，它就会使用较多的处理器时间。您可以辨别活动最多的处理器，这最有可能是问题区域所在。

- 如果 DS-5 Streamline 画面中不显示任何正忙碌的具体处理器，而显示不同处理器的图表中存在缺口，则问题可能是 API 使用而导致的停滞。这属于应用处理器问题。

- 如果问题似乎不是其中一个，您的应用程序可能受到带宽约束。
  如果您可以确定问题区域但优化没有任何影响，则应用程序也可能受到带宽约束。请参见*带宽约束问题*（第 5-23 页）。

在确定了问题区域后，请前往本章中相关的小节，以隔离问题的原因：

- 如果是应用处理器问题，请参见*应用处理器约束问题*（第 5-6 页）。
- 如果是顶点处理器问题，请参见*顶点处理器约束问题*（第 5-11 页）。
- 如果是片段处理器问题，请参见*片段处理器约束问题*（第 5-15 页）。
- 如果是带宽问题，请参见*带宽约束问题*（第 5-23 页）。

备注

带宽使用过量问题比较难以直接判断，因为它常常表现为其他处理器存在问题。当应用程序受到带宽约束时，系统中的所有处理器都会受到负面影响。

5.3 应用处理器约束问题

本节介绍如何诊断最为常见的应用处理器约束问题。其中包含下列小节：

- *关于应用处理器约束问题*（第 5-8 页）。
- *检查问题属于应用程序约束还是 API 约束*（第 5-8 页）。
- *应用程序约束*（第 5-8 页）。
- *API 约束*（第 5-8 页）。
- *检查绘制调用是否太多*（第 5-9 页）。
- *检查 VBO 使用情况*（第 5-9 页）。
- *检查是否有流水线停滞*（第 5-9 页）。
- *检查状态更改是否太多*（第 5-9 页）。
- *其他应用处理器约束问题*（第 5-10 页）。图 5-4（第 5-7 页）显示本小节的流程。

如果应用程序在应用处理器上执行软件中的许多处理工作，它可能会受到应用处理器约束。如果应用程序源自桌面平台并且您将其迁移到移动平台，则尤其会如此。如果应用程序受到应用处理器约束，则图形优化对性能不会有任何影响。

DS-5 Streamline 可以让您查看：

- 进行了哪些 API 调用。
- API 函数被调用了多少次。
- API 函数中花费的时间。

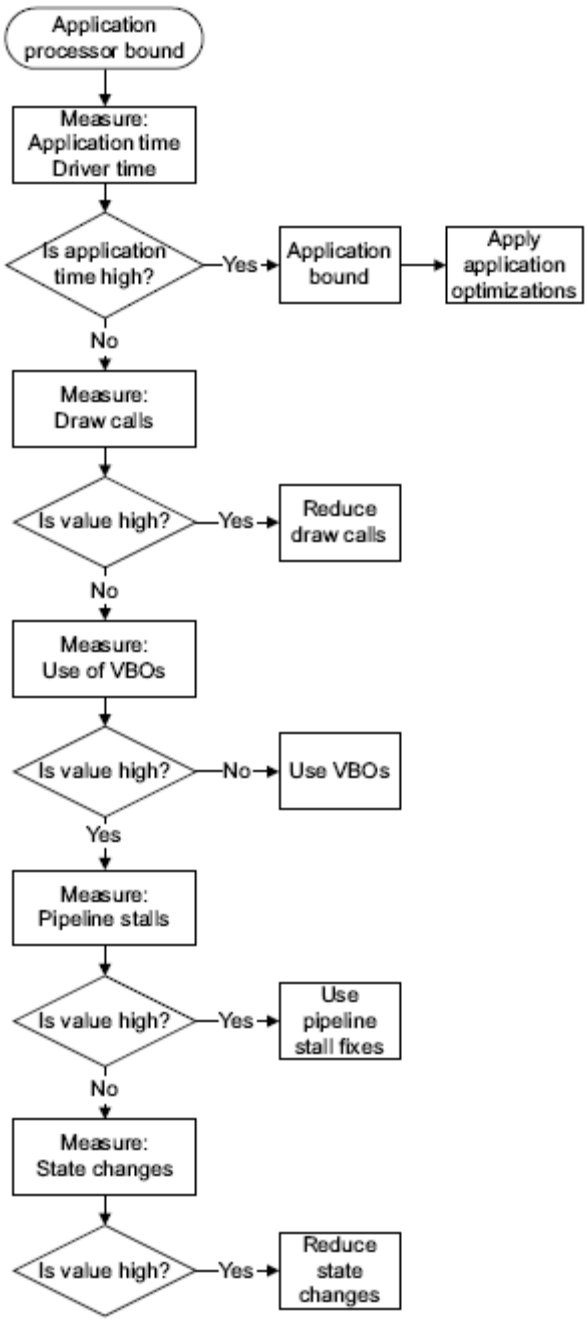

图 5-4 应用处理器时间较长工作流程

5.3.1 关于应用处理器约束问题

应用处理器可能会受应用程序约束或 API 约束：

- 如果应用程序没有优化，系统的其余部分将闲置以等待应用程序生成数据。
- 如果应用程序没有以最佳方式使用 OpenGL ES API，它可能会受到 API 约束。这可能会造成系统中的图形处理器停滞，或者使驱动程序太过忙碌。

5.3.2 检查问题属于应用程序约束还是 API 约束

测量应用程序中花费的时间，再测量驱动程序中花费的时间，然后进行比较。

查看 API 调用的时间线：

- 如果应用程序花费大量时间忙于处理应用程序代码而不进行很多 API 调用，则问题最有可能出在应用程序本身。请参见*应用程序约束*。
- 如果应用程序进行许多 OpenGL ES API 调用，则应用程序可能会调用得太多。请参见 *API 约束*。
- 如果应用程序不忙于处理应用程序代码，而且进行的 OpenGL ES API 调用数量相对较少，则应用程序可能没有以最佳的方式使用 OpenGL ES API。请参见 *API 约束*。
- 如果不同处理器的图表中存在缺口，但没有具体的处理器处于忙碌状态，则问题可能是由于 API 使用导致停滞。请参见 *API 约束*。

5.3.3 应用程序约束

如果应用程序时间太长，性能会受到限制，因为应用程序没有足够快速地产生命令。

有许多应用程序优化技巧来改善应用程序的性能。例如：

- 使用近似。
- 代码优化。
- 快速数据结构。
- 替代算法。
- 矢量化。

如需更多信息，请参见第 6 章 *应用处理器优化*。

5.3.4 API 约束

如果驱动程序时间太长，性能会受到限制，因为驱动程序无法产生足够的命令。驱动程序出现负载过重的原因有许多。通常是因为没有以最佳方式使用 OpenGL ES API，或者应用程序进行过多数量的 OpenGL ES API 调用。

有许多正确的方式来使用 OpenGL ES API，但一些方式产生的结果要优于另一些。下列小节介绍要查找的问题。

5.3.5 检查绘制调用是否太多

测量下列 OpenGL ES 绘制调用：

- glDrawElements 统计信息：对 glDrawElements 的调用数
- glDrawArrays 统计信息：对 glDrawArrays 的调用数

如果图形很高，则性能受到限制，因为应用程序没有足够快速地产生命令。绘制调用具有相对较高的开销，因此调用数量太多可能会使驱动程序负载过重。

影响性能的每帧绘制调用数取决于设备中的应用处理器。

通常，每帧几千个绘制调用会造成性能大幅下降。将每帧绘制调用数量保持在几百个或更少的范围内，可以维持较高的性能。

为尽可能减小开销，将来自不同绘制调用的元素一起批处理，形成一个绘制调用。请参见*尽可能减少绘制调用*（第 7-2 页）。

5.3.6 检查 VBO 使用情况

测量计数器 BufferProfiling：VBO 上传时间 (ms)。

如果图形在多个帧后达到高峰，然后在一个或多个帧上降到零或较小的数值，则 VBO 的使用可能是正确的。

如果图形持续位于低值或零，则您使用的 VBO 数量可能不够，或根本未在使用。

如果不使用 VBO，则必须每一帧传输数据，这会限制应用程序的性能。VBO 可以减小这一开销，实质性地提高应用程序的性能。

请参见*使用顶点缓冲区对象*（第 2-8 页）。

5.3.7 检查是否有流水线停滞

检查应用处理器和 GPU 处理器是否同时都处于活动状态。如果没有，则流水线可能出现停滞。

通常，数据流过流水线，多个数据元素立即得到处理。当流水线中必须彻底完成一个阶段的处理后才移到下一个阶段时，就会出现流水线停滞。如果流水线停滞，则没有处理器同时活动，性能就会受到最慢运算的限制。

为防止流水线停滞，请避免使用下列 OpenGL ES 调用：

- glReadPixels()
- glCopyTexImage()
- glTexSubImage()

请参见*确保图形流水线保持运行*（第 7-8 页）。

5.3.8 检查状态更改是否太多

查看下列 OpenGL ES API 调用的时间线：

- glEnable()
- glDisable()

状态更改具有相对较高的开销。数量太多时会造成驱动程序负载过重。

每个状态更改都要求传输数据，而且有处理开销。如果应用程序有大量的状态更改，这可导致驱动程序负载过重，从而限制性能。

要将开销降到最低，最好尽可能进行最少的状态更改。可能时，一起批处理状态更改，以减少其数量。

请参见*尽可能减少状态更改*（第 7-7 页）。

5.3.9 其他应用处理器约束问题

如果应用程序受到应用处理器约束，但问题不是本节中所列之一，请查看其他优化。请参见第 6 章*应用处理器优化*。

另一可能是内存带宽使用过量导致问题。请参见第 10 章 *带宽优化*。

5.4 顶点处理器约束问题

本节介绍如何诊断最为常见的顶点处理器约束问题。其中包含下列小节：

- *检查顶点着色器时间*（第 5-12 页）。
- *检查顶点是否太多*（第 5-13 页）。
- *检查 PLBU 时间是否较长*（第 5-13 页）。
- *检查是否有剔除的原语*（第 5-14 页）。
- *检查 VBO 使用情况*（第 5-14 页）。
- *其他顶点处理器约束问题*（第 5-14 页）。图 5-5（第 5-12 页）显示本小节的工作流程。

备注

在实际的应用程序中，顶点处理器成为瓶颈并不常见。

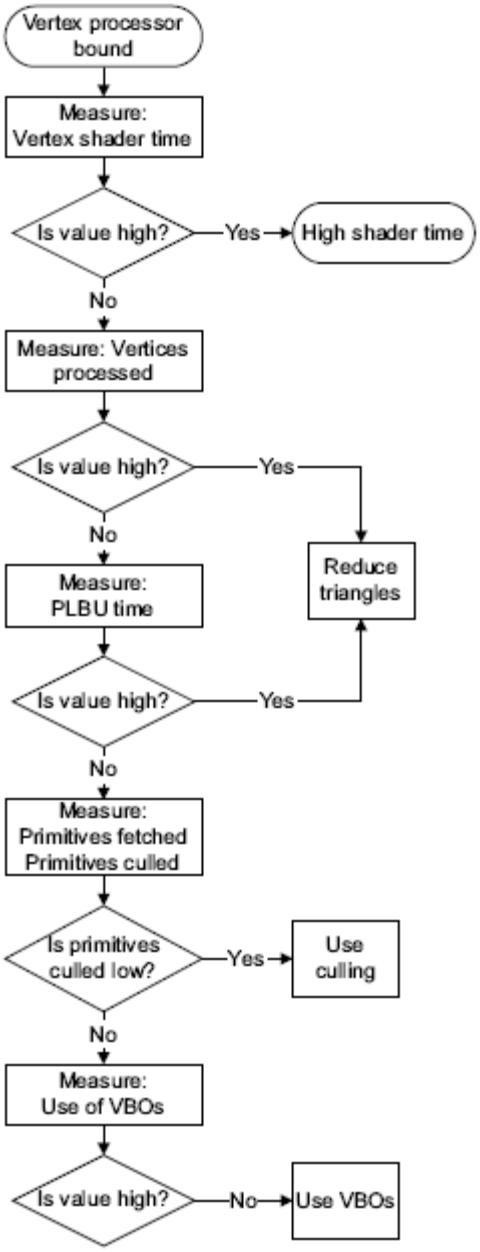

图 5-5 顶点处理器时间较长工作流程

5.4.1 检查顶点着色器时间

测量顶点处理器计数器 Mali GPU 顶点处理器：活动周期数，顶点着色器。

如果图形持续较高，则应用程序受到顶点着色器约束。可能出现这一情况的原因有多种。要确定原因，请测量下列顶点处理器计数器：

- Mali GPU 顶点处理器：活动周期数
- Mali GPU 顶点处理器：活动周期数，顶点着色器
- Mali GPU 顶点处理器：顶点加载器缓存未命中数

您可以使用这些计数器，查看着色器是否有下列问题：

着色器是否太长？

如果满足下列条件，则顶点着色器太长：

- 活动周期数顶点着色器的值比活动周期数的值低。
- 顶点加载器缓存未命中数的值较高。

如果顶点着色器太长，请尝试将它缩短。

着色器是否太复杂？

如果满足下列条件，则着色器太复杂：

- 活动周期数顶点着色器的值与活动周期数的值接近。
- 顶点加载器缓存未命中数的值较低。

如果顶点着色器太复杂，请作如下尝试：

- 简化着色器。
- 应用算法优化。
- 考虑是否可将着色器部分或完全移到片段处理器或应用处理器。

着色器是否太长且太复杂？

如果测量结果表明某着色器太长，但优化产生的影响相对很小，则该着色器可能既太长又太复杂。此时，您必须同时针对这方面进行优化。

着色器是否有太多分支？

分支在 Mali GPU 中的开销相对较低，但分支数量太多会使着色器太多或太复杂。

5.4.2 检查顶点是否太多

测量顶点处理器计数器 Mali GPU 顶点处理器：已处理顶点数

如果图形持续较高，则应用程序可能使用了太多三角形，请参见

- *检查 PLBU 时间是否较长*。
- *第 8 章 顶点处理优化*。

5.4.3 检查 PLBU 时间是否较长

检查*多边形列表构建器单元* (PLBU) 时间，具体操作为测量顶点处理器计数器 Mali GPU 顶点处理器：活动周期数，PLBU 几何处理。

如果图形持续较高，则应用程序可能使用了太多三角形。

如果应用程序受三角形约束，则它具有太多原语、顶点或三角形。要修复此问题，请尝试减少三角形数量。

要减少三角形数量，您可以：

- 使用较少的对象。
- 使用较简单的对象。

- 取消镶嵌对象。
- 剔除三角形。

您可以使用其他技巧来减少三角形数量。请参见*检查是否有剔除的原语*，另请参见第 8 章 *顶点处理优化*。

备注

PLBU 时间较长可能表示您没有启用剔除。若是如此，顶点处理器就必须为每一帧处理所有三角形。此情形中，可能不是三角形数量太多，而是正在处理的数量太多。请参见*使用剔除*（第 8-3 页）。

5.4.4 检查是否有剔除的原语

您可以通过剔除最终图像中不显示的三角形，减少场景中的三角形数量。

测量顶点处理器计数器 Mali GPU 顶点处理器：剔除的原语数。

如果剔除的原语数图形较低，则应用程序可能未充分利用剔除。确保*背面剔除*和*深度测试*都为活动状态。

如果剔除的原语数图形较高，则原因可能有多种：

- 应用程序可能使用过多数量的三角形。
- 应用程序可能没有使用视域剔除。
- 应用程序可能让 Mali GPU 执行太多剔除。在应用程序中粗糙地剔除一些大对象，然后再发送到 Mali GPU。

如需更多信息，请参见*使用剔除*（第 8-3 页）。

5.4.5 检查 VBO 使用情况

测量软件计数器 BufferProfiling：VBO 上传时间 (ms)。

如果图形在多个帧后达到高峰，然后在一个或多个帧上降到零或相对较小的数值，则 VBO 的使用是正确的。

如果值持续较低或为零，则您使用的 VBO 数量可能不够，或根本未在使用。

如果不使用 VBO，则必须每一帧传输数据，这会限制应用程序的性能。VBO 可以减小这一开销，实质性地提高应用程序的性能。

请参见*使用顶点缓冲区对象*（第 2-8 页）。

5.4.6 其他顶点处理器约束问题

如果应用程序受到顶点处理器约束，但问题不是本节中所列之一，那么请查看其他优化。

请参见第 8 章 *顶点处理优化*和第 11 章 *其他优化*。

5.5 片段处理器约束问题

本节介绍如何诊断和解决片段处理器约束问题。片段处理器可能会受到片段处理器或片段着色器程序所耗用的数据的约束。本节包含下列小节：

- *检查片段处理器约束问题*（第 5-16 页）。
- *检查片段着色器约束问题*（第 5-19 页）。

5.5.1 检查片段处理器约束问题

本节逐一指导您完成一系列测量，以诊断片段处理器约束问题。其中包含下列小节：

- *检查纹理带宽*（第 5-17 页）。
- *检查纹理是否过大*（第 5-17 页）。
- *检查压缩纹理读取数*（第 5-17 页）。
- *检查是否有过度绘制*（第 5-18 页）。
- *其他片段处理器约束问题*（第 5-18 页）。

图 5-6 显示本节的工作流程。

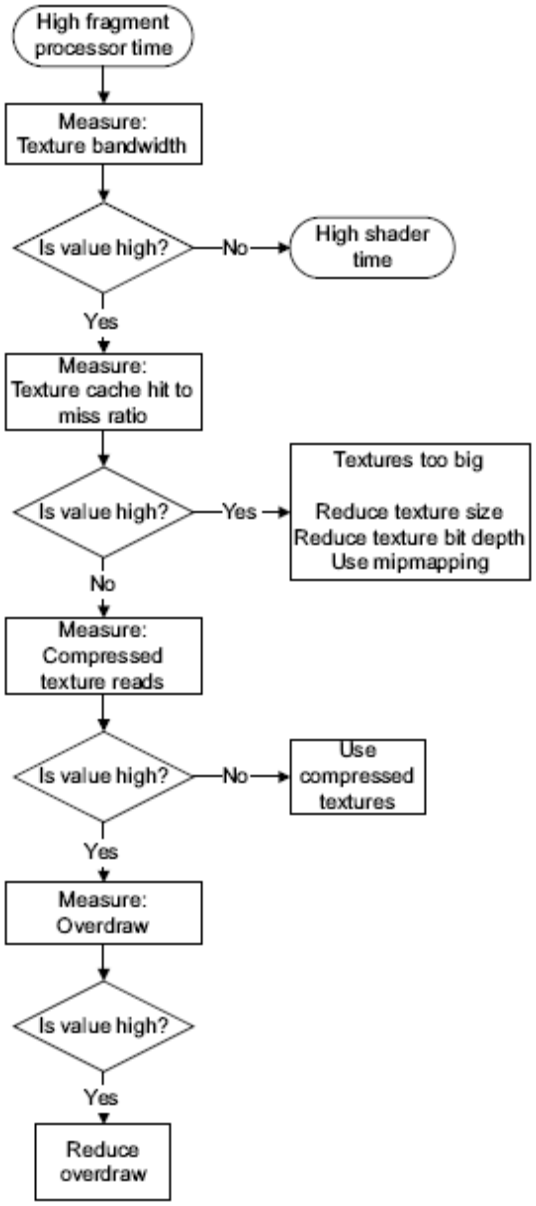

图 5-6 片段处理器时间较长工作流程

检查纹理带宽

测量下列计数器。

- 片段处理器：总线读取总数
- 片段处理器：纹理描述符读取数

如果总线读取总数和纹理描述符读取数的图表都较低，则应用程序受到片段着色器约束。请参见*检查片段着色器约束问题*（第 5-19 页）。

如果总线读取总数和纹理描述符读取数的图表都较高，则应用程序受到纹理约束。请参见*检查纹理是否过大*。

检查纹理是否过大

测量下列计数器：

- Mali GPU 片段处理器 *X*：纹理缓存命中数。
- Mali GPU 片段处理器 *X*：纹理缓存未命中数。

纹理缓存未命中率通常是纹理缓存命中率的 10%。如果比这高很多，则可能存在多个问题：

- 纹理太大。
- 纹理位深太高。
- 应用程序未使用纹理映射。

如果满足上述任何条件，则应用程序有可能存在内存带宽的问题。

请参见第 9 章 *片段处理优化*，以及第 10 章 *带宽优化*。

检查压缩纹理读取数

测量下列计数器：

- Mali GPU 片段处理器 *X*：纹理缓存命中数
- Mali GPU 片段处理器 *X*：压缩纹理缓存命中数

分析和比较计数器的值：

- 如果压缩纹理缓存命中数的图形为零，则应用程序不在使用压缩纹理。
- 从纹理缓存命中数中减去压缩纹理缓存命中数。这可得出未压缩纹理缓存的命中数。如果此值大大低于压缩纹理缓存命中数，则您使用的压缩纹理数量很少。请考虑对纹理进行更多压缩。
- 如果压缩纹理缓存命中数的图表比纹理缓存未压缩读取数高很多，则应用程序正在使用压缩纹理，但是：
  - 纹理可能太大。
  - 应用程序未使用纹理映射。
  - 纹理数量可能太多。

纹理使用大量内存带宽。如果带宽用量太大，着色器就无法获取足够的数据，导致停滞。压缩纹理可以减少内存带宽用量。这可提高性能，同时降低功耗。请参见*使用纹理压缩*（第 10-6 页）。

检查是否有过度绘制

测量计数器 Mali GPU 片段处理器 *X*：穿过 z/模板片段数。

将结果除以一个帧中的像素数。这可得出过度绘制系数。

过度绘制系数为 1 时表示没有过度绘制，但这很罕见。过度绘制通常为 2.5 左右，但因应用程序而异。如果内容大多为不透明的，并且过度绘制系数大于 2.5，则性能可能受到影响。请考虑使用下列技巧，减小过度绘制系数：

- 启用深度测试。
- 启用背面剔除，避免渲染不可见的表面。

绘制顺序对减少过度绘制很重要：

1. 按照深度排列场景对象。
2. 按照从前到后顺序绘制不透明对象。
3. 按照从后到前顺序绘制透明对象。

请参见*避免过渡绘制*（第 9-2 页）和*使用剔除*（第 8-3 页）。

备注

具有透明内容的场景过度绘制系数高是正常的。

其他片段处理器约束问题

如果应用程序受到片段处理器约束，但问题不是本节中所列之一，那么请查看其他优化。请参见第 9 章 *片段处理优化*和第 11 章 *其他优化*。

另一种可能是应用程序受到带宽约束。请参见 *带宽约束问题*（第 5-23 页）。

5.5.2 检查片段着色器约束问题

片段着色器时间较长可能是由多个问题造成的。本节介绍这些问题。其中包含下列小节：

- *确认问题是在片段着色器中*（第 5-20 页）。
- *检查着色器是否太长*（第 5-21 页）。
- *检查着色器是否太复杂*（第 5-21 页）。
- *检查着色器是否太长且太复杂*（第 5-21 页）。
- *检查分支是否太多*（第 5-22 页）。
- *其他片段着色器问题*（第 5-22 页）。

图 5-7（第 5-20 页）显示本小节的工作流程。

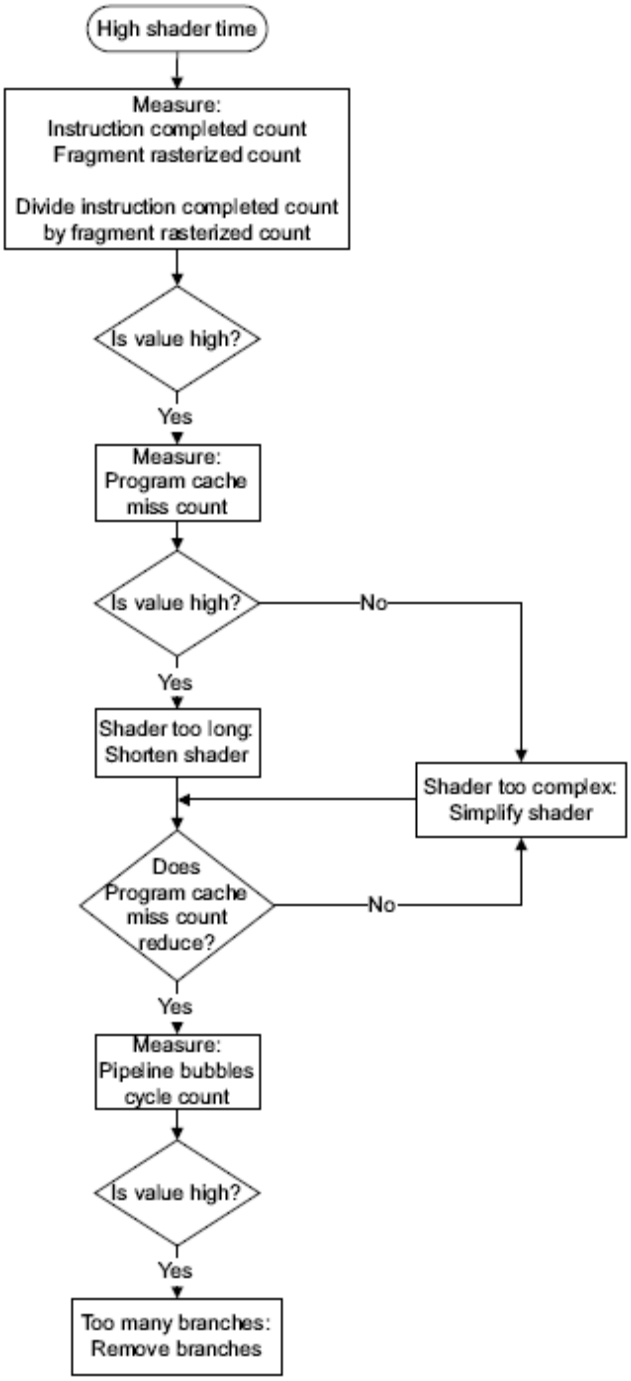

图 5-7 片段着色器时间较长工作流程

确认问题是在片段着色器中

测量下列片段着色器硬件计数器：

- Mali GPU 片段处理器 *X*：穿过 z/模板片段数。
- Mali GPU 片段处理器 *X*：已完成指令数。

将已完成指令数的值除以穿过 z/模板片段数的值。这会得出每个片段完成的指令字平均数。

如果此值较高，则应用程序的片段着色器时间较长。计为高的值的类型视您的 Mali GPU 的确切配置而定。每个片段可用周期数取决于着色器核心的数量及其时钟速度。要得出每个片段可用的周期数，您必须计算着色器预算。请参见*设定计算预算并据此进行测量*（第 3-7 页）。

备注

Mali GPU 每个周期执行多条指令。请不要假设一行代码等于一条指令。

检查着色器是否太长

测量下列硬件计数器：

- Mali GPU 片段处理器 *X*：程序缓存未命中数。
- Mali GPU 片段处理器 *X*：程序缓存命中数。

通常程序缓存未命中数的值比较低。常常是程序缓存命中率的 0.01% 或以下。

如果图形较高，则片段着色器程序太长。若是如此，请缩短着色器，再重新测量计数器。如果结果没有变化，则着色器太长且太复杂。请参见*检查着色器是否太长且太复杂*。

检查着色器是否太复杂

测量下列硬件计数器：

- Mali GPU 片段处理器 *X*：程序缓存未命中数。
- Mali GPU 片段处理器 *X*：程序缓存命中数。

通常程序缓存未命中数的值比较低。常常是程序缓存命中率的 0.01% 或以下。

如果程序缓存未命中数的图表非常低，则片段着色器太复杂。请尝试：

- 简化着色器。
- 算法优化。
- 考虑是否可将着色器部分或完全移到顶点处理器或应用处理器。

重新测量总体性能。如果性能没有改善，则着色器太长且太复杂。请参见*检查着色器是否太长且太复杂*。

检查着色器是否太长且太复杂

如果检查了着色器程序是否太长或太复杂，并且优化没有很大影响，则着色器程序可能既太长又太复杂。此情形中，请尝试简化着色器并缩短其长度。

如果对长度的优化影响甚微，请再次测量程序缓存未命中数。如果该值已下降，则大小优化已发挥作用，但着色器依然太复杂。在这种情形中，您还必须简化着色器。请参见*片段着色器优化*（第 9-4 页）。

检查分支是否太多

测量片段处理器硬件计数器 Mali GPU 片段处理器 *X*：流水线气泡周期数。

如果 Mali GPU 片段处理器 *X*：流水线气泡周期数的图形很高，则着色器可能有太多分支。

备注

分支在 Mali GPU 上不太可能是问题，因为分支的计算开销相对较低。

其他片段着色器问题

如果应用程序受到片段着色器约束，但问题不是本节中所列之一，那么请查看其他优化。请参见第 9 章 *片段处理优化*和第 11 章 *其他优化*。

5.6 带宽约束问题

本节介绍如何判断应用程序是否受带宽约束，以及如何减少带宽使用。其中包含下列小节：

- *测量纹理缓存命中与未命中比*（第 5-25 页）。
- *检查位块传送*（第 5-25 页）。
- *测量最大带宽*（第 5-25 页）。
- *对比应用程序带宽和最大可用带宽*（第 5-26 页）。
- *片段处理器带宽约束*（第 5-26 页）。
- *顶点处理器带宽约束*（第 5-27 页）。

如果应用程序使用的内存带宽超过可用量，则应用程序受带宽约束。判断应用程序是否受带宽约束比较困难，因为它影响应用程序和图形流水线的所有部分。

您可以使用多种技巧来确认应用程序是否受带宽约束。图 5-8（第 5-24 页）显示本小节的工作流程。

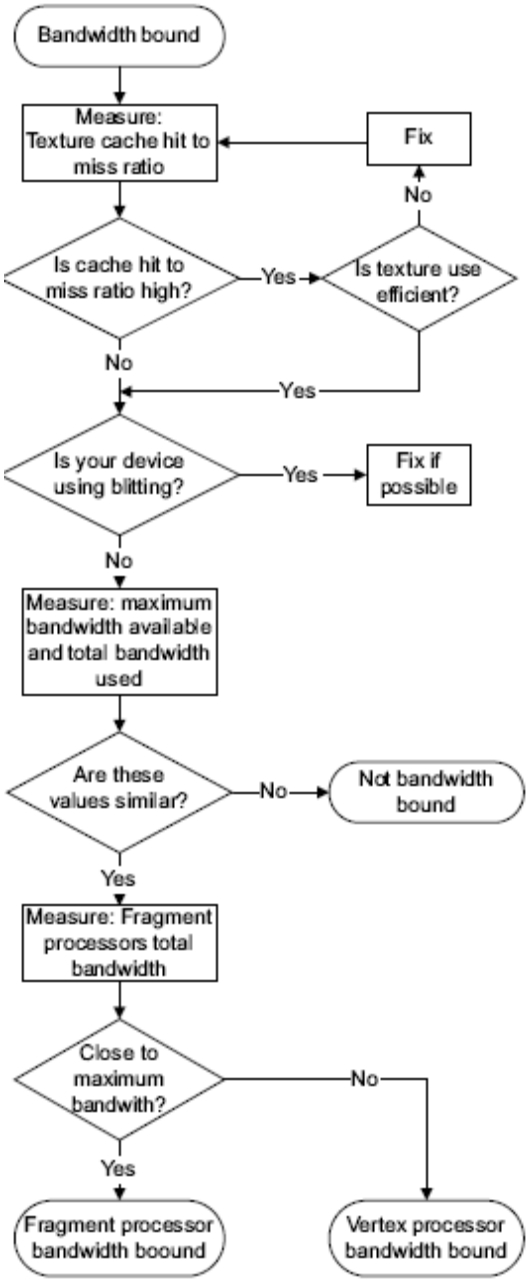

图 5-8 带宽约束工作流程

5.6.1 测量纹理缓存命中与未命中比

使用最多内存带宽的是纹理。使用太多纹理带宽的副作用是纹理缓存用量很高。

测量下列片段着色器硬件计数器：

- Mali GPU 片段处理器 *X*：纹理缓存命中数。
- Mali GPU 片段处理器 *X*：纹理缓存未命中数。

通常，纹理缓存命中与未命中比约为 10 比 1。比值越高越好，越低则越差。

比值较低意味着缓存使用高于正常值。其原因可能是：

- 使用的纹理太大。
- 使用的大纹理数太多。
- 应用程序未使用压缩纹理。
- 应用程序未使用纹理映射的纹理。

如果应用程序中存在这些问题，请先修复它们，然后再继续。修复这些问题可以减少内存带宽使用，而且也可能会修复总体性能问题。

5.6.2 检查位块传送

检查系统是否在位块传送图像数据。位块传送使用内存带宽，而且这可能是带宽使用过量的原因。

要检查位块传送，请测量软件计数器：EGL 计数器：位块传送时间。

如果系统在位块传送高分辨率帧缓冲区，此操作的带宽用量可能会达到每秒数百兆字节。

系统设置不正确时可能会出现位块传送。请参见*检查显示设置*（第 11-5 页）。

备注

系统可能会作为其显示系统的一部分执行位块传送。这种位块传送无法避免。

5.6.3 测量最大带宽

如果可以确定应用程序受带宽约束，请查找带宽使用过量的来源以进行修复。查找带宽使用过量的来源：

1. 计算出最大可用带宽。
2. 将它与系统的不同部分比较，找出哪一部分使用过多带宽。

如果不知道设备上的最大可用带宽，您可以通过运行尽可能多地使用带宽的测试应用程序得出结果。确保测试应用程序使用：

- 最高可用分辨率。
- 最高可用位深。
- 尺寸很大、位深很高的纹理。
- 16x 抗锯齿。
- 无纹理压缩。

- 无纹理映射。

- 无 VSYNC。

测试应用程序的宗旨是尽可能多地使用带宽，使内存系统过载。如果测试应用程序运行时帧率很低，则其工作正常。

在测试应用程序运行期间，测量下列计数器：

- Mali GPU 顶点处理器：字读取数，系统总线。
- Mali GPU 顶点处理器：字写入数，系统总线。
- Mali GPU 片段处理器 *X*：总线读取总数。
- Mali GPU 片段处理器 *X*：总线写入总数。

备注

如果 Mali GPU 有多个片段处理器，请确保测量其全部。

将测量结果相加，再乘以 8。其结果是最大可用带宽，单位是 MB/秒。这一测量结果包含缓存使用，因此有可能看起来高于系统中的最大理论内存带宽。

5.6.4 对比应用程序带宽和最大可用带宽

运行您的应用程序，测量下列计数器：

- Mali GPU 顶点处理器：字读取数，系统总线。
- Mali GPU 顶点处理器：字写入数，系统总线。
- Mali GPU 片段处理器 *X*：总线读取总数。
- Mali GPU 片段处理器 *X*：总线写入总数。

备注

如果 Mali GPU 有多个片段处理器，请确保测量其全部。

将测量结果相加，再乘以 8。其结果是总带宽用量，单位是 MB/秒。

将这些值相加，将结果与您在*测量最大带宽*（第 5-25 页）从测试应用程序中测量的最大可用带宽值比较。如果数字接近，则您的应用程序使用太多带宽。

比较这些值，查看哪个值最高：

- 如果 Mali GPU 片段处理器使用的带宽最多，请参见*片段处理器带宽约束*。
- 如果 Mali GPU 顶点处理器使用的带宽最多，请参见*顶点处理器带宽约束*（第 5-27 页）。

5.6.5 片段处理器带宽约束

如果应用程序由于片段处理器使用过多带宽而受到约束，问题可能源自下列一个或多个原因：

纹理 通常，纹理读取是内存带宽用量中最大的部分。减少纹理带宽使用的方式有多种：

- 减少纹理数量。

- 降低纹理分辨率。

- 降低纹理位深。
- 使用纹理映射。
- 使用纹理压缩。

过度绘制

当绘制的像素重叠时就会出现过度绘制。这会浪费带宽，因为被覆盖的像素是不可见的。您可以通过减少过度绘制来减少带宽使用。请参见*避免过度绘制*（第 9-2 页）。

三线性过滤

三线性过滤涉及读取多个纹理来生成一个像素，这会使用大量的带宽。只有绝对必要时，才可为对象使用三线性过滤。

片段着色器太复杂

具有许多中间状态的复杂着色器可以填满缓存内存，导致其清空到主内存。如果出现此情形，则每次从内存读取或向其写入数据时都会使用内存带宽。

请参见第 10 章 *带宽优化*。

5.6.6 顶点处理器带宽约束

如果应用程序由于顶点处理器使用过多带宽而受到约束，则问题可能是下列之一：

三角形数量太多

三角形数量太多可导致带宽使用过量。这鲜有发生，除非您有非常复杂的场景。请参见*减少顶点数量*（第 8-2 页）。

如果不使用剔除，则三角形也可能会使用过多带宽。如果不使用剔除，顶点处理器会处理从未绘制的三角形。请参见*使用剔除*（第 8-3 页）。

顶点着色器太复杂

具有许多中间状态的复杂着色器可以填满缓存内存，导致其清空到主内存。如果出现此情形，则每次向内存写入数据或从中读取时都会使用内存带宽。

读取非本地化数据

如果数据广泛分散在各种数据结构中，GPU 可能会将数据加载到未曾使用的缓存中。避免松散顶点阵列等数据结构，始终尝试将数据放到一起，使其更加容易缓存。

第 6 章
应用处理器优化

有许多优化技巧来改善运行于应用处理器的代码的性能。

本章介绍应用处理器优化。其中包含下列小节：

6.1 对齐数据

OpenGL ES 标准要求数据从应用程序复制到驱动程序。如果将数据对齐为八字节，则它更容易被缓存。这可加快复制，减少内存带宽使用。

您还必须在 Mali GPU 可以使用数据之前对齐数据。如果您在代码中实施对齐，则 Mali GPU 驱动程序不必在复制期间对齐数据。这可以改善性能，因为进行复制时就没有对齐数据的开销了。

备注

- 确保在从标头文件导入数据时对齐数据。这可能需要特定于编译器的命令。
- 数据对齐后更容易在 GPU 和应用处理器中缓存，因此最好始终尽可能将数据对齐为八字节。

6.2 优化循环

循环在应用程序代码和着色器中用于密集计算。它们可占用非常大比例的处理时间。您可以通过优化循环，大幅加快应用程序和着色器代码。

通常，循环优化的关键是使循环在每个迭代上尽可能执行最少的工作，以便加快各个迭代。如果一个循环运行一万次，该循环每减少一个指令，要求的指令数就会少一万。

备注

务必对代码执行性能分析，以便您可以找出使用最多计算能力的循环。仅优化需要优化的循环。

移动重复的计算

如果循环中有可以在该循环之前进行的计算，请将这些计算移到循环之外。

查找不断重复计算相同值的指令。您可以将这些计算移到循环之外。

移动未使用的计算

如果循环中某些计算产生的结果不在该循环中使用，请将这些计算移到循环之外。

避免迭代 if 测试中的计算

每次循环迭代时，条件测试就会判断是否需要再次迭代。尽可能简化这一计算。思考是否每次都必须执行整个计算。若有可能，将可以提前计算的部分移到循环之外。

简化代码

避免复杂的代码结构。代码越简单，编译器就越容易优化该代码。

避免交叉迭代依赖项

尽力使迭代中的计算独立于其他迭代。这可让编译器进行优化，反之可能就不行。

处理小块数据

确保内循环处理小块的数据。数据块越小，越容易缓存。

尽可能减小代码

使循环（尤其是内循环）变小，可让它们更容易缓存。这对性能有重大影响。减小循环大小可以让指令更有可能被缓存，因而能提高性能。

展开循环

您可以从许多循环迭代中提取计算，把它们变为数量较少的大迭代。这可通过减少 if 测试计算数量，降低计算负载。

测试不同大小上展开循环的运行状况。超过一定阈值后，循环就会变得太大而不能高效缓存，从而造成性能下降。

编译器可以帮您展开循环，所以您可能不必在代码中手动进行。一些应用处理器可以在硬件中执行循环展开。两种情形中您都值得测试，查看是否必须手动展开。

然而，自动代码展开是有限的。如果应用处理器支持自动代码展开，最好使代码尺寸保持较小。进行测试，了解最佳的方式。

避免循环中分支

循环中的条件测试通常会产生分支指令。分支指令可能会使应用处理器变慢，在循环中尤其不佳。尽可能避免分支，尤其是在内循环中。

不要在内循环中进行函数调用

循环中的函数调用至少产生两个分支，而且会发起从内存不同部分的程序读取。尽可能避免在循环中调用函数，尤其是在内循环中。

考虑是否可以先在循环中处理部分数据，然后在循环之外调用。

一些函数调用可以通过将函数内容复制到循环中予以避免。编译器或许可以对一些调用自动执行此操作，但对于部分编译器，您可以通过编译器指令强制执行。

使用正确的工具

各种应用处理器工具可以帮助优化应用程序。有这些工具时请加以利用。

尽可能使用矢量指令

矢量指令可以同时处理多个数据项目，您可以利用它们来加速循环或减少迭代数量。如需更多信息，请参见*使用矢量指令*（第 6-5 页）。

备注

如果要处理的项目数量非常少，不使用循环可能会更快。

6.3 使用矢量指令

许多 ARM 应用处理器和 Mali GPU 包含矢量或*单指令多数据* (SIMD) 指令。它们可以让处理器利用一条指令处理多个运算。对于一些运算，使用矢量指令可以大大提升性能，因此请尽可能利用矢量处理。

矢量处理的运作方式为通过一条指令并行处理多个运算。您可以进行的运算数量和类型取决于处理器中矢量处理器扩展的类型。

例如，具有 NEON 媒体处理器引擎的 ARM 处理器最多可同时处理 4 个 32 位运算、8 个 16 位运算，或者 16 个 8 位运算，视具体实施而定。Mali GPU 中的片段处理器具有相似的能力。

矢量指令的其他优点还有：

- 矢量运算可以减小代码大小，使其更容易被缓存。
- 有时，您可以在循环中使用矢量指令，作为循环展开的形式。这可将循环必须执行的迭代总数减少 4 倍或以上。

  备注

  如果被处理的数据项数量不是矢量中元素数量的倍数，您需要额外代码来处理起始或结尾元素。此代码仅执行一次。

  例如，如果您要处理包含 1002 个数据项的阵列，仅需要执行 250 次迭代，因为您一次可以处理 4 个项目。剩余的 2 个项目需要额外的代码进行处理。

6.4 使用快速数据结构

所有图形和应用程序处理都依赖数据。如果处理器不能高速读取数据，应用处理器就无法高速处理该数据。

试验不同类型的数据结构并进行测量，查看哪种类型速度最快。尝试如下所列：

避免指针追踪

一些数据结构使用指针在数据结构中的不同元素之间导航。读取数据元素涉及首先读取一个地址，然后从该地址获取数据项和下一元素的地址。这意味着读取数据会在每次读取元素时产生内存延迟。如果数据在内存间分散，延迟就会增加，读取速度就会降低。

如果数据的性质要求这样的数据结构，请：

- 尝试使元素包含尽可能多的数据。
- 使多个元素聚集成小的阵列。

使用规则的线性数据结构

阵列可以被快速读取，因为它们拥有规则的结构，可以线性读取或从内存进行流传输。

本地化数据，使其更容易缓存

如果数据元素并列排在内存中，读取一个元素可能会将另一个带入缓存中。这可提高访问该数据的速度并节省功耗，因为它能节约内存事务。

优化数据大小，使其能装入缓存中

尽力使数据元素的大小为 2 的幂，这可使它们更容易缓存。

6.5 考虑替代算法和数据结构

有许多代码级优化可以使应用程序提高特定区域内的性能。使用不同的算法和数据结构有时可以大大提高性能。

考虑您要尝试解决的问题，了解是否能够通过不同方式进行解决。例如：

- 如果要将风景建模为网格，可尝试使用不规则网格作为备选。
- 如果要在网格中表示对象位置，备选方案是使用数学方式。
- 对于冲突检测，可以将对象分组到具有边框的层次结构中。从最上层测试冲突速度较快。只有最上级中存在冲突时，您才需要在层次结构中进行深层测试。
- 如果在计算三维中的冲突检测，可检查是否能够使用近似或者忽略第三维。这可减少许多类型的计算。

以上是示例。试验不同的方法并测量结果，查看它们的效果如何。

如果想不到备选方法，请向别人描述问题，看看他们会如何解决。如果他们有不同的解决方案，那么这或许就是您可以考虑的备选方案。

描述解决方案本身就足以带来新的想法。这可以帮助您想出可能的备选方案。

6.6 使用多处理

许多现代移动设备具有多个应用处理器。多处理或许能大幅提高应用程序代码的性能。

多处理要求数据能被并行访问。

您可以通过下列方式实现这一点：

- 拆分数据，分配到各个线程。
- 在线程之间共享数据，使用互斥防止冲突。
- 使用并发数据结构。也就是能够被多个线程同时读取的数据结构。

第 7 章

API 级别优化

本章介绍 API 级别优化。其中包含下列小节：

7.1 尽可能减少绘制调用

本节介绍如何尽力减少绘制调用。其中包含下列小节：

7.1.1 关于尽可能减少绘制调用

进行过多数量的绘制调用很容易限制应用程序的性能。如果应用程序每一帧进行成百上千个绘制调用，请尝试减少其数量。

OpenGL ES 绘制调用会调用下列函数：

- glDrawArrays()
- glDrawElements()

性能限制的原因在于每一个绘制调用都有处理开销。绘制调用所需的处理包括分配内存、复制数据，以及其他数据处理。无论绘制的三角形是一个还是几千个，开销是相同的。

如果应用程序发出大量绘制调用，这些绘制调用的开销可以使应用程序变得受应用处理器约束。

将一致项和绘制调用分组到一起，并通过较小数量的 API 调用进行传递，您就能减少或消除性能限制。

如果将多个三角形合并到一个绘制调用，其开销仅会发生一次而不是多次。这可减少应用程序的总开销，并提高其性能。

图 7-1 显示一系列传递相对较少数据的 API 调用。图 7-2（第 7-3 页）显示一对 API 调用，每个传递大量数据。

图 7-2（第 7-3 页）中的处理与图 7-1 中所示的相同，但花费的时间较短，因为两个绘制调用的开销要比四个的开销低。

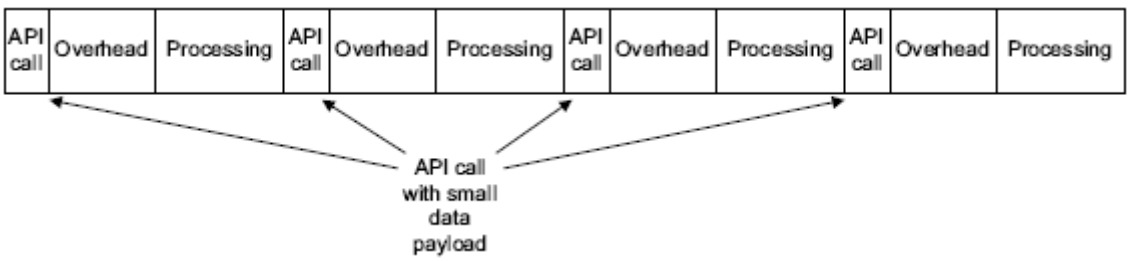

图 7-1 数据有效负载较小的 API 调用

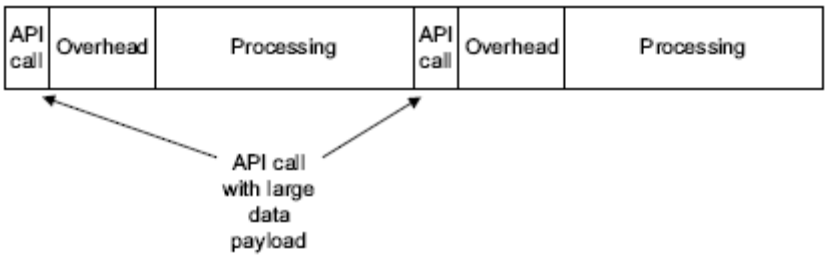

图 7-2 数据有效负载较大的 API 调用

您可以使用下列数据结构合并单个绘制调用中的三角形：

- 三角形条。
- 三角形列表。
- 索引阵列。

您可以将合并项再次合并，进一步合并三角形。例如，您可以将多个三角形条合并到一起。您可以在条之间添加退化三角形，辅助这一过程。

7.1.2 合并绘制调用的限制

合并绘制调用存在限制。合并的三角形必须：

- 格式相同。
- 使用同一着色器。
- 使用同一 GL 状态。

如果因为三角形使用不同着色器而无法合并它们：

- 考虑着色器产生的特效是否带来显著差异。如果差异比较小，可能值得去除这一独立特效。
- 您也可以在一个着色器中使用不同特效，然后使用控制变量在它们之间选择。

  备注

  这会增加着色器的大小，而且选择特效需要额外的计算。

7.1.3 将纹理合并为纹理拼图

为帮助完成合并三角形的过程，您可以将多个纹理合并为一个纹理拼图。此为单个纹理图像，其中包含来自不同对象的组件的纹理。图 7-3 中显示的纹理拼图包含用于绘制图 7-4 中所示标志的所有纹理。

备注

基于纹理拼图的技巧通常不适用于重复的纹理。

图 7-3 标志的纹理拼图

图 7-4 游戏中的标志

7.1.4 合并多个纹理拼图

如果绘制的对象使用相同的着色器，您可以将多个纹理拼图合并为一个较大的合并纹理拼图。图 7-5 显示一组标志的合并纹理拼图，其中包含图 7-3（第 7-4 页）中所示的纹理拼图。

图 7-5 具有多个标志的纹理拼图

7.1.5 将文本纹理合并为字体拼图

如果在屏幕上绘制文本，您可以将文本放到字体拼图中。然后，您可以将所有文本绘制调用合并为一个调用，并使用*字体拼图*提供字体图像。这是一个高效的技巧，因为它可让您在单个绘制调用中绘制所有文本。图 7-6 显示字体拼图的一个部分。

图 7-6 字体拼图

7.2 尽可能减少状态更改

状态更改与绘制调用相似，每次状态有变化时都会产生开销。为减少此类开销，请尽可能减少应用程序所做的状态更改数量。

您可以通过下列技巧减少状态更改：

- 将具有相同纹理的三角形或对象分组在一起。
- 将状态相同的对象分组在一起，一次性对它们进行更改。
- 使用纹理拼图。这可让您绘制复杂的单一对象，或者绘制具有单一纹理的多个对象。请参见*将纹理合并为纹理拼图*（第 7-4 页）。
- 检查应用程序中的状态是否必须要更改，仅进行必要的状态更改。
- 删除冗余的 glEnable() 与 glDisable() 对。

  例如，将此类型的结构：

  — glEnable()

  <do operation 1>

  glDisable()

  glEnable()

  <do operation 2>

  glDisable()

  替换为：

  — glEnable()

  <do operation 1>

  <do operation 2>

  glDisable()

7.3 确保图形流水线保持运行

本节介绍如何避免使图形流水线停滞。其中包含下列小节：

- *图形流水线*。
- *避免停止图形流水线的调用*（第 7-9 页）。

7.3.1 图形流水线

图形流水线由应用处理器、顶点处理器、片段处理器和帧缓冲区组成。要让图形流水线高效运作，数据必须在其中保持流动。

如果数据不能保持流动，应用处理器、顶点处理器或片段处理器可能会空闲，等待来自另一阶段的数据输出。

您可以通过同时执行多种类型的运算并避免阻止性调用，让流水线保持运行。

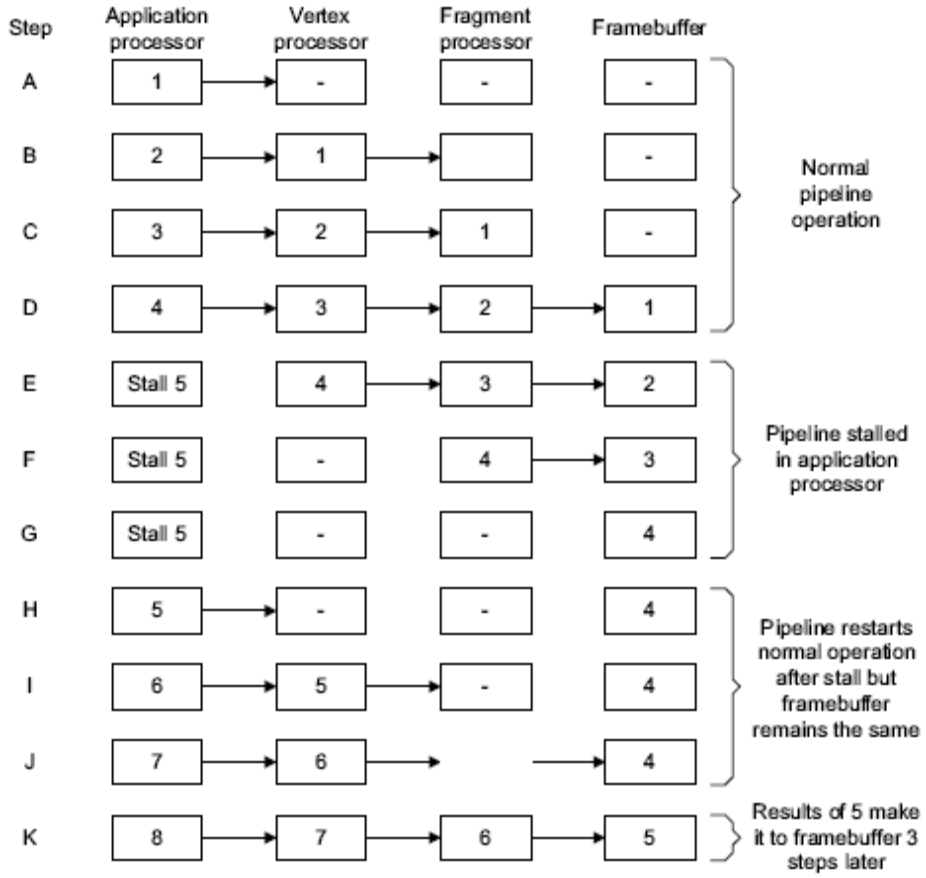

图 7-7 具有停滞的图形流水线工作流

图 7-7 显示了停滞对图形流水线的影响。示意图中显示在 11 个步骤中处理 8 个作业。这些步骤用字母 A 到 K 表示：

第 A 到 D 步第 A 到 D 步显示数据正常通过流水线。

作业 1 在应用处理器中启动（第 A 步），并沿着流水线移动（第 B 到 C 步）。结果在第 D 步到达帧缓冲区。

第 E 到 G 步 第 E 到 G 步显示作业 5 在应用程序处理器中停滞造成的影响。

作业 4 在第 F 和 G 步中处理时穿过流水线，但顶点处理器在第 F 步时耗尽工作。第 G 步，几何和片段处理器都没有工作。

第 G 到 J 步 第 G 到 J 步，帧缓冲区包含作业 4 对第 G 到 J 步的结果。这是因为 GPU 无法在流水线停滞时产生更多帧。

第 H 步 在第 H 步时，作业 5 在应用处理器中完成，并在第 I 步移到顶点处理器中。

作业 5 沿着流水线移动，在第 K 步中到达帧缓冲区。

第 E 到 G 步中的停滞使得帧缓冲区保留第 G 到 J 步中的相同图像。在实际的应用程序中，这可产生帧率骤降，并可能产生应用程序跳跃不定。

7.3.2 避免停止图形流水线的调用

有多个 OpenGL ES 函数调用可导致图形流水线停滞。请避免使用这些调用，因为它们对性能有重大影响。

一些 OpenGL ES 函数调用从帧缓冲区中读取。为此，Mali GPU 必须先渲染整个图像，然后您才能从中读取。此类运算有可能会缓慢。

最好避免下列 OpenGL ES 调用，因为它们可能会使图形流水线停滞：

glReadPixels()

此调用可使图形流水线停滞，造成性能大幅降低。若有可能，避免对 glReadPixels() 的调用。

glCopyTexImage()

如果您使用 OpenGL ES 2.0，则 ARM 建议您使用 FBO，不要使用此调用。它们可以让您直接绘制到纹理，而不是使用副本。

glTexSubImage()

这通常不会使流水线停滞；但是，如果您试图修改尚未完成渲染的像素映射或 FBOrendertarget，它就能使其停滞。

第 8 章
顶点处理优化

本章介绍一系列针对顶点处理受约束的应用程序的优化。其中包含下列小节：

8.1 减少顶点数量

您可以使用下列方式来减少顶点数量：

- 使用较少的对象。
- 使用较少的多边形。
- 使用较简单的对象。
- 取消镶嵌对象。
- 使用三角形条。
- 使用三角扇形。
- 剔除三角形。请参见*使用剔除*（第 8-3 页）。

8.2 使用剔除

剔除是删除场景中不可见的部分的过程，以便 Mali GPU 不必绘制这些部分。剔除的方式有多种。其范围广泛，包括由应用程序执行的非常粗糙的剔除类型，以及由 Mali GPU 执行的非常精细的各个三角形和片段剔除。

备注

最好是在应用程序中剔除完整对象和绘制调用，而让 GPU 剔除各个三角形。在具有许多三角形的场景中，三角形剔除可能是非常密集的运算。如果使用应用处理器执行，有可能会降低应用程序的性能。

启用背面剔除

背面剔除是一个 OpenGL ES 选项。启用时，GPU 会删除对象背面的三角形，以便不绘制它们。

启用深度测试

深度测试是一个 OpenGL ES 选项，它会丢弃之前绘制的片段后面的片段。这可减少 GPU 所需的计算量。

使用边界形状和视域剔除

您可以使用边界性框体、球体和其他形状来包含复杂的对象，以此减少剔除所需的运行时计算量。

通过利用这些形状来计算哪些对象处于可见状态，应用程序可以删除不可见的对象，仅将可见对象的信息发送给 GPU。

- 任何在视域之外、不可见的对象都可以被剔除。
- 在摄像机平面后面的任何对象都可以被剔除。

算出视界中的哪些部分是可见的

在绘图或游戏等应用程序中，任何给定的时间都只有视界中的特定部分处于可见状态。您可以在应用程序中的不同位置上于运行时前算出视界中的哪些部分处于可见状态，以此减少 GPU 工作。在运行时，应用程序可使用此信息来判断哪些对象为可见，Mali GPU 就可仅处理这些对象。

图 8-1（第 8-4 页）显示游戏中的一个视界部分。如果不使用剔除，则 Mali GPU 必须处理所有这些数据。

图 8-1 不使用剔除的视界部分

图 8-2 显示了游戏视界中的同一部分，其灰色部分已被剔除。仅处理剩余的彩色部分。通过剔除灰色部分，Mali GPU 处理的数据量大大减少。

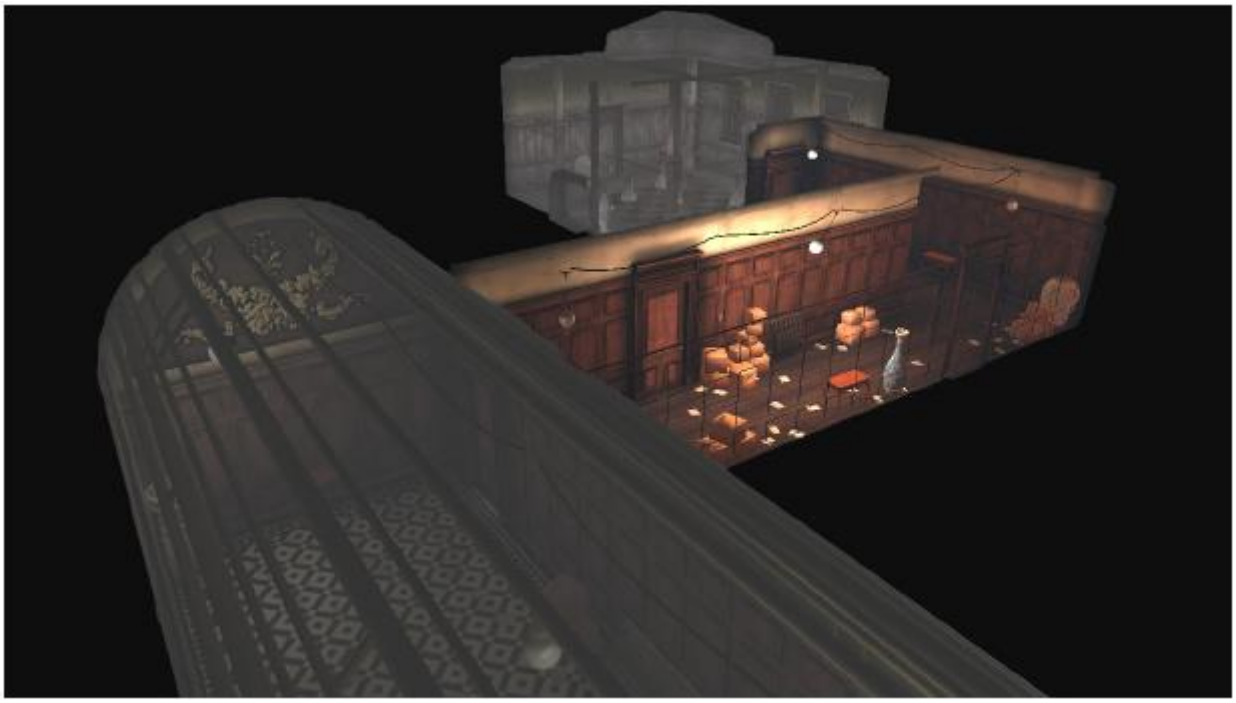

图 8-2 使用剔除的视界部分

8.3 使用法线贴图模拟精细几何

使用法线贴图模拟精细几何。这可以造成形状比实际更加复杂的印象。使用这种技巧，您可以让对象外观更佳，同时减少顶点处理器上的计算量。

图 8-3 和图 8-4 显示场景中使用的法线贴图。高亮部分是通过高光贴图生成的。

图 8-3 使用法线贴图的地板

图 8-4 使用法线贴图的天花板

8.4 使用细节级别

LOD（细节级别）是一组涉及使用细节较大或较小的各种对象版本的技巧，具体取决于对象离摄像机的距离。

当对象靠近摄像机时，显示分辨率完整、细节全面的对象。当对象处于较远的位置时，使用分辨率较低、细节较少的对象。图 8-5 和图 8-6 显示不同细节级别的小行星。

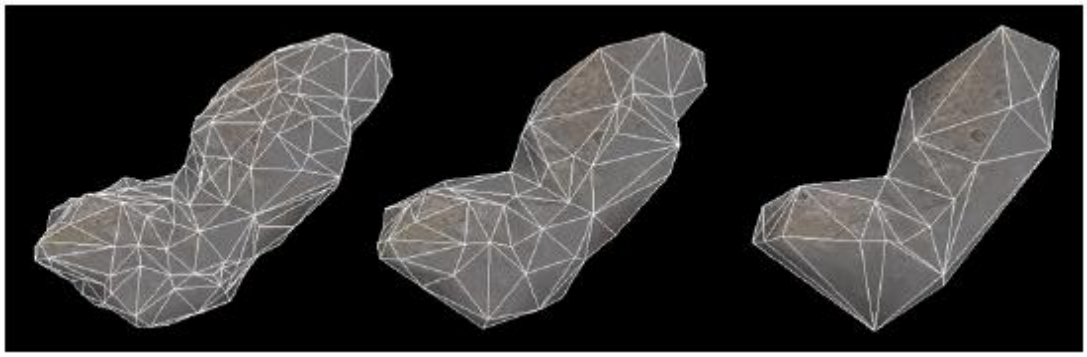

图 8-5 具有细节级别的线框小行星

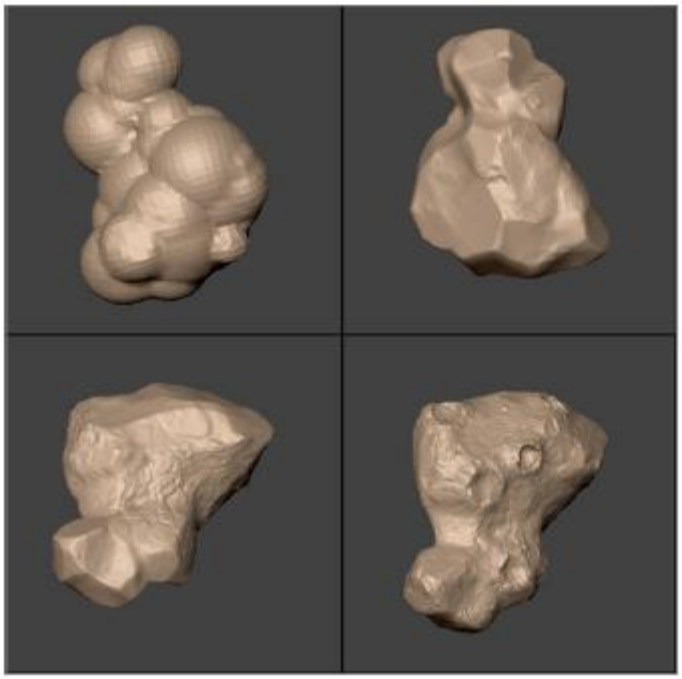

图 8-6 具有细节级别的小行星

同样的技巧适用于纹理。高分辨率纹理用于离摄像机较近的对象，低分辨率纹理用于较远的对象。

将 LOD 应用到对象可以同时减少计算功率和内存带宽的使用量。

您可以针对离摄像机的不同距离选择 LOD 技巧。使用适当的最低细节级别，但尝试较低的级别以查看是否有显著差异。使用较少的细节，要求的内存和带宽也较少。

可以使用的另一技巧是将对象替换为公告板。当对象在屏幕上变得很小时，小细节就会变得不可见。您可以将对象替换为带有含该对象图像的纹理的公告板。绘制看起来相似的事物需要较少的计算量。

例如，思考一间细节完整的房子，其窗户、窗台和窗帘都是 3D 对象。随着摄像机从房子拉远，各个细节可以被简化，因为细致的细节不再可见。与摄像机的距离最远时，房子在屏幕上变得非常小，您可以将它绘制为房子的平面图像，不带任何 3D 细节。

第 9 章
片段处理优化

本章介绍片段处理优化。其中包含下列小节：

9.1 片段处理器优化

本节介绍片段处理器优化。其中包含下列小节：

- *减少纹理带宽*。
- *避免过度绘制*。
- *其他片段处理器约束问题*（第 9-3 页）。

9.1.1 减少纹理带宽

纹理使用大量内存带宽。如果带宽用量太大，片段着色器就无法获取足够的数据，导致停滞。

可以减少纹理带宽的方式有多种。请参见第 10 章 *带宽优化*。

9.1.2 避免过度绘制

当 GPU 多次在同一像素上绘制时就会发生过度绘制。这意味着计算资源和内存带宽被用于最终帧中不可见的片段。这是资源浪费，而且会对应用程序性能带来负面影响。尽可能避免过度绘制。

您可以通过多种方式避免过度绘制：

启用深度测试

深度测试可丢失已绘制片段后面的任何片段。

启用背面剔除

背面剔除可删除对象背面不可见的三角形。

对非透明对象先排序后绘制

1. 确保已启用深度测试。这样此技巧才能起作用。
2. 根据与摄像机的距离，排列场景中对象的顺序。
   您可以通过在*二进制空间分区* (BSP) 树等数据结构中保留几何来节省排序时间。
3. 按照从前到后顺序绘制不透明对象。
4. 最后按照从后到前顺序绘制透明对象。

优化透明度使用

透明是一种有用的特效，但如果有许多图层，则会造成资源密集。

如果使用透明对象，您可以通过以下方式减少其绘制成本：

- 确保最后绘制透明对象。如果不最后绘制透明对象，您的应用程序可能必须多次绘制同一对象，造成资源浪费。
- 尽可能减少互相可见的透明图层数量。
- 在非透明对象之后绘制透明对象。
- 按照从后到前顺序绘制透明对象。

9.1.3 其他片段处理器约束问题

如果应用程序受到片段处理器约束，但问题不是本节中所列之一，那么请查看其他优化。请参见第 9 章 *片段处理优化*和第 11 章 *其他优化*。

另一可能是应用程序可能受到带宽约束。请参见第 10 章 *带宽优化*。

9.2 片段着色器优化

本节介绍片段着色器优化。其中包含下列小节：

- *缩短着色器*。
- *简化着色器*。
- *减少分支数量*。
- *其他片段着色器问题*（第 9-5 页）。

9.2.1 缩短着色器

如果着色器太长，它可导致指令缓存未命中，造成性能降低。如果片段着色器程序太长，请缩短着色器。

您可以使用许多标准代码优化技巧来缩短着色器。请参见*优化循环*（第 6-3 页）。

9.2.2 简化着色器

如果片段着色器太复杂，就会需要很长时间来执行其功能。有许多技巧可用于简化着色器。

算法优化

考虑是否可以将多个运算合并为一个。

移动计算

考虑是否可将着色器部分或完全移到顶点处理器或应用处理器。

利用纹理替代计算

您可以预先计算值并存储在纹理中，以此替代执行计算。这是一种有用的技巧，但会增加内存带宽用量。

您可以使用许多标准代码优化技巧来简化着色器。请参见*优化循环*（第 6-3 页）。

9.2.3 减少分支数量

分支在 Mali GPU 上的运算速度不慢，但它们在以下情形中可能会造成问题：

- 分支数量太多。
- 分支形成必须从缓存或 RAM 读取的代码块。

分支数量太多或许表明正在使用控制类型算法。控制算法通常在应用处理器上运行更佳。

考虑是否可使用较少的分支密集算法来获得相同的结果。另一选择是考虑将代码移到应用处理器。

如果分支形成必须从缓存或 RAM 读取的代码块，则着色器可能太长。

如果代码块之间存在相似性，您或许可以考虑合并它们，以一种计算方式来执行分支的功能。

9.2.4 其他片段着色器问题

如果应用程序受到片段着色器约束，但问题不是本节中所列之一，那么请参见第 11 章 *其他优化*。

第 10 章
带宽优化

本章说明务必要保持内存带宽用量低的原因，同时介绍降低带宽使用量的方法。其中包含下列小节：

10.1 关于减小带宽

与桌面系统中的 GPU 相比，移动设备中的内存带宽受到很大的限制。它很容易成为限制应用程序性能的瓶颈。基于此原因，尽可能尝试减少内存带宽使用量：

- 带宽是一种共享资源，因此用量太大会以无法预料的方式限制整个系统的性能。例如，图形内存与应用内存共享，因此 GPU 的带宽用量高可导致应用处理器性能降低。
- 访问外部内存需要许多功率，所以降低带宽使用可降低功耗。
- 访问缓存中的数据可以降低用电量，提高性能。如果应用程序必须大量读取内存，请使用纹理映射和纹理压缩等技巧来确保数据对缓存友好。请参见*使用纹理映射*（第 10-5 页），另请参见*使用纹理压缩*（第 10-6 页）。

备注

判断内存带宽是否造成问题比较困难。请参见*判断内存带宽是否为问题所在*（第 4-18 页）。

10.2 优化纹理

纹理通常会使用大量内存带宽，因此优化其使用是减小带宽的第一目标。本节介绍多种方法，用于优化纹理并减少其使用的内存带宽：

- *确保纹理不会太大*。
- *使用适合屏幕上对象的纹理分辨率*。
- *尽可能使用位深较低的纹理*。
- *如果纹理不包含锐利的细节，则使用分辨率较低的纹理*。
- *纹理和光照贴图不必大小相同*（第 10-4 页）。
- *减少纹理数量*（第 10-4 页）。

10.2.1 确保纹理不会太大

通过使用创建所需特效所需的最低分辨率和位深，确保纹理不会太大。

如果纹理太大，它们会使用更多内存带宽，生成质量较低的图像，而且对缓存不友好。

备注

如果应用程序需要大纹理，务必使用纹理映射和纹理压缩来减少带宽用量。请参见*使用纹理映射*（第 10-5 页），以及*使用纹理压缩*（第 10-6 页）。

10.2.2 使用适合屏幕上对象的纹理分辨率

根据对象在屏幕上显示的尺寸，使用对应的纹理分辨率。如果对象在屏幕上较小，您可以使用小纹理。例如，如果对象在屏幕上显示时仅占据
100 x 100 像素，就不需要使用大于 100 x 100 像素的纹理。

备注

如果纹理以 1:1 像素精度方式显示，就不需要图像过滤或纹理映射。禁用这些功能可节省内存和带宽。

10.2.3 尽可能使用位深较低的纹理

有时可以使用位深较低的纹理，可见清晰度损失较小或没有损失。试验每像素 4 位或 8 位的纹理，看它们是否适合您的应用程序。如果不行，可尝试 16 位格式，如 RGB565 或 RGBA5551。仅在没有其他选择时使用 32 位 RGBA8888 纹理。

备注

不要使用 24 位纹理。它们对缓存不友好；虽然小于 32 位纹理，但使用的带宽更多。

10.2.4 如果纹理不包含锐利的细节，则使用分辨率较低的纹理

如果纹理不包含锐利的细节，请尝试使用分辨率较低的纹理。

如果纹理包含一小部分高细节，您或许可以对高细节部分使用标准分辨率纹理，对其余部分使用低分辨率纹理。

10.2.5 纹理和光照贴图不必大小相同

不同的纹理有不同的用途。即使它们在同一对象上，也不必使用相同的分辨率。单独考虑每个纹理，使用最低的分辨率和位深来获得您需要的效果。

10.2.6 减少纹理数量

如果使用多个纹理，请查看是否能够减少纹理数量而获得相同的结果。例如，可考虑该特效是否能预焙到另一纹理上。

10.3 使用纹理映射

纹理映射是一种可能带来极大性能提升的技巧。纹理映射有诸多潜在优点：

- 减少内存带宽使用。
- 提高纹理可缓存能力。
- 提高图像质量。

当应用程序在屏幕上绘制对象时，绘制图像的大小可能会根据离摄像机的远近而有很大不同。可能是填满屏幕，也可能是远处的细小对象。

如果使用单个纹理，则只有在绘制的对象与纹理大小相似时，该纹理采样时的密度（或*纹理采样密度*）才正确。如果对象大小和纹理大小不匹配，则纹理采样密度不正确。这可造成失真，降低图像质量。

纹理映射可以解决此问题，它可获取高分辨率纹理并缩放到多个较小的尺寸，称为*纹理映射级别*。与无纹理映射的纹理相比，它需要的内存要多 33% 左右。图 10-1 中显示了一个具有多个纹理映射级别的图像。

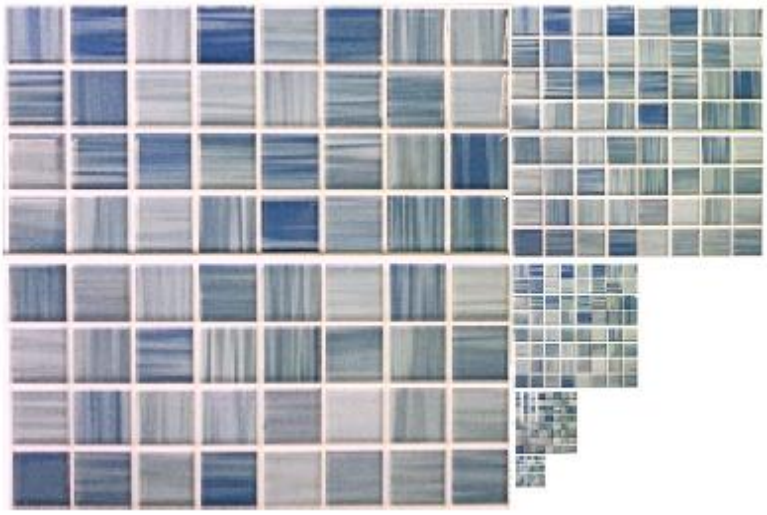

图 10-1 具有多个纹理映射级别的图像

当绘制启用了纹理映射的对象时，将使用大小与该对象最接近的纹理。这意味着对象始终具有大小匹配的纹理以从其采样，因此纹理采样密度就会正确。这可减少图像失真，生成质量更高的图像。

如果纹理采样密度正确，采样的纹素就会在内存中互相靠近，使得纹理数据更容易被缓存。缓存能力增强可降低内存带宽用量并提高性能。

您可以命令 Mali GPU 驱动程序在运行时生成纹理映射，或者也可使用 *Mali GPU 纹理压缩工具*预生成纹理映射。您可以从 *Mali 开发者中心*下载此工具，网址为 http://malideveloper.arm.com。

您可以通过下列函数调用，在运行时从 OpenGL ES 中的未压缩纹理生成纹理映射：

glGenerateMipmap()

10.4 使用纹理压缩

本节介绍纹理压缩。其中包含下列小节：

- *关于纹理压缩*。
- *纹理对纹理压缩的适合性*（第 10-7 页）。
- *使用具有透明度的 ETC1*（第 10-7 页）。

10.4.1 关于纹理压缩

纹理压缩可以减少纹理在内存中占用的大小。纹理压缩：

- 提高性能。
- 减少内存带宽使用。
- 提高纹理可缓存能力。

Mali GPU 驱动程序支持 *Ericsson 纹理压缩* (ETC1)。它广泛用于 OpenGL ES 版本 1.1 和 2.0。ETC1 压缩会删除数据，因此它被描述为*有损*压缩。与未压缩纹理相比，图像质量有所降低。与 RGB565 相比，ETC1 纹理压缩提供的压缩比为 4:1。

您可以使用 *Mali GPU 纹理压缩工具*压缩图像。该工具提供压缩前和压缩后的图像，因此您能对更改进行比较。它也提供可显示压缩图像和未压缩图像之间差异的图像。

您可以从 *Mali 开发者中心*下载 *Mali GPU 纹理压缩工具*，网址为 http://malideveloper.arm.com。

图 10-2 显示 Mali GPU 纹理压缩工具。

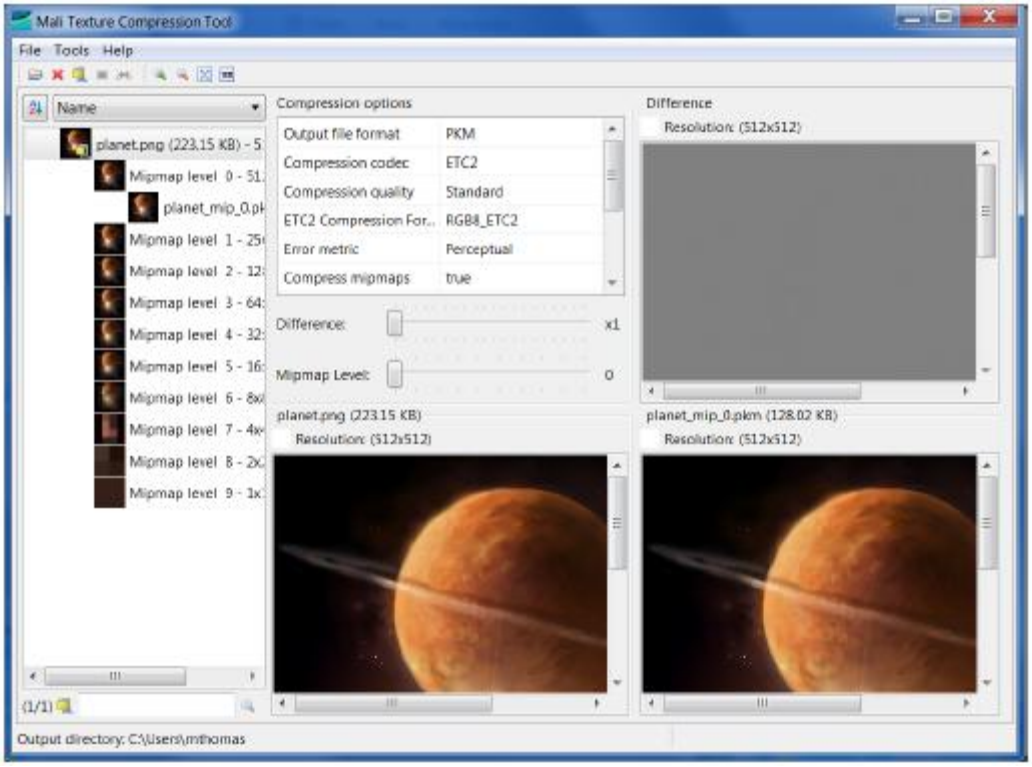

图 10-2 Mali GPU 纹理压缩工具

10.4.2 纹理对纹理压缩的适合性

纹理压缩对一些内容效果很不错。它在某些情形中提供的压缩没有任何明显的图像更改，但在另一些情形中，差别会非常明显。

纹理压缩通常适合：

- 照片内容。
- 漫射贴图。
- 环境贴图。
- 高光贴图。

纹理压缩通常不适合用于：

- 法线贴图。
- 字体。
- 不同色调间的渐变。

备注

以上仅仅是作为参考。对您的内容进行试验，查看压缩是否能提供可接受的结果。

10.4.3 使用具有透明度的 ETC1

ETC1 不直接支持在具有透明度的图像上压缩纹理，因为它不支持图像中的 Alpha 通道。

有多种技巧可以将颜色和透明度信息分开，再将它们存储在纹理拼图或单独的纹理中，从而使您能够使用具有透明度的 ETC1 纹理压缩。

图 10-3 显示颜色和透明度合并在纹理拼图中。

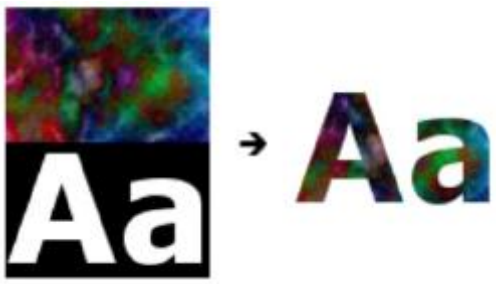

图 10-3 从纹理拼图中合并纹理，创造具有透明度的纹理

图 10-4 显示颜色和透明度存储为单独的纹理。

图 10-4 拆分合并的纹理，创造具有透明度的纹理

如需有关这些方法的更多信息和示例，请参见 *Mali 开发者中心*，网址为 http://malideveloper.arm.com/。

10.5 仅在必要时使用三线性过滤

三线性过滤可用于避免从观看者向远处延伸的大型平整表面上的失真。例如，地板、墙壁和机场跑道。

三线性过滤具有很大的带宽成本，因此要避免使用它，除非能够带来显著的视觉差别。

如果应用程序在小屏幕上运行，三线性过滤的视觉影响可能会非常小。若是如此，请考虑是否值得使用额外的计算和带宽。

备注

如果应用程序受带宽约束，禁用三线性过滤是有用的性能质量权衡。

10.6 通过避免过度绘制减小带宽

过度绘制涉及将片段多次绘制到同一像素。如果前面的片段遮挡了后面的片段，则绘制它们会浪费计算能力。

绘制片段使用带宽，因此减少过度绘制可降低带宽使用量。

请参见*避免过度绘制*（第 9-2 页）。

10.7 通过剔除减少绘制表面

您可以通过剔除三角形来减少绘制的表面数量。这可减少处理的三角形数量，并且减少过度绘制。

请参见*使用剔除*（第 8-3 页）。

10.8 利用细节级别减小带宽

LOD 是一组技巧，可以随着对象远离摄像机移动而降低几何和纹理的分辨率。这些技巧能够降低带宽使用量。

如需更多信息，请参见*使用细节级别*（第 8-6 页）。

第 11 章
其他优化

本章介绍其他章节中未涵盖的优化。其中包含下列小节：

11.1 使用近似

本节介绍如何使用近似来提高性能。其中包含下列小节：

- *一般近似方法*。
- *技巧性近似方法*。

许多桌面应用程序使用特效来创造高质量图像。高质量特效需要许多计算。

一种优化方式是在这些特效中使用近似。这涉及使用更快速的技巧创建类似但不完全相同的特效。

11.1.1 一般近似方法

下列是一组您可以使用的一般近似方法：

在能够带来最大视觉冲击的地方使用计算能力

一些特效是微妙的，在移动设备上可能很难看清。将计算能力用在效果最显眼的特效上，充分利用可用的资源。

简化特效

您可以通过去除或简化周密的特效来进行优化。尝试将复杂的特效更改为提供类似结果的简单特效。

图形很少要求正确性

人类的视觉系统常常不会注意到错误，因此图形通常不必对所有应用程序都做到精确无误。光线也可不准确，阴影也可有错或完全缺失。

您或许能够使用更近似、更简单的计算，通过降低准确度来提高性能。

简化等式

一些着色器使用复杂的等式。尝试使用较简单的、计算不是很密集的等式来获得类似的特效。

考虑不同的算法

有时，您可以通过换用更加高效的算法大幅提升性能。尝试不同的算法，看其表现如何。

11.1.2 技巧性近似方法

下列是一些技巧性近似方法：

阴影 您可以使用投影阴影来产生软阴影。

光照 减少使用的光线数量或缩短它们的可见距离，以此节省计算数量。

模糊 对于模糊特效，使用低分辨率纹理，再使它变模糊。与使较高分辨率的纹理变模糊相比，这速度更快，使用的内存带宽也较少。

您也可以使用景深等模糊特效，以便使用较低分辨率的纹理。

图 11-1 显示应用程序中的景深。前景中的路面细节比较锐利，随着

远离摄像机而逐渐模糊。

图 11-1 景深

光晕

在对象周围创建一系列透明的白色三角形。从对象边上完全不透明渐变到边缘完全透明。

反射

有一些图形技巧可以利用相对较少的开销生成反射，但会存在限制。思考您是否能在限制范围内处理，再使用这些技巧。

图 11-2（第 11-4 页）显示具有反射的场景。这些反射是通过绘制倒立棋子的几何来产生的。这一技巧不使用额外的几何内存，但只能用于平整的镜面。

图 11-2 带有反射的场景

11.2 检查显示设置

本节介绍如何检查显示设置。其中包含下列小节：

- *关于显示设置*。
- *设置不正确导致数据转换*（第 11-6 页）。
- *配置显示设置以避免转换*（第 11-6 页）。
- *确保应用程序具有正确的绘制表面*（第 11-7 页）。

11.2.1 关于显示设置

要让系统正常运作，必须配置下列设置：

绘制格式 绘制格式是您的应用程序用于绘制的颜色格式。
应用程序使用绘制格式在绘制表面上绘制图形。

绘制表面 绘制表面是内存中的一个区域，应用程序将图形绘制到其中。该表面可以使用不同的配置，称为 *EGLConfig*。每种配置拥有不同的分辨率和色深设置。

帧缓冲区 这是一种数据结构，包含要发送到屏幕进行显示的数据。

显示控制器 显示控制器是一种硬件组件，它将数据从帧缓冲区发送到屏幕。

屏幕 屏幕是您查看的物理显示设备。它具有最高分辨率和色深。

图 11-3 显示图像的显示步骤。

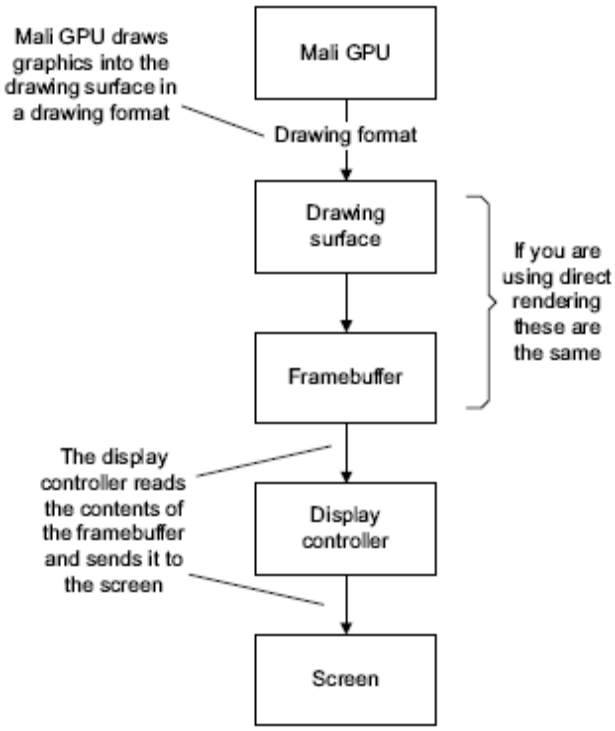

图 11-3 图像显示步骤

11.2.2 设置不正确导致数据转换

本节介绍设置不正确导致的数据转换，以及它们需要的资源。其中包含下列小节：

- *转换类型*。
- *转换使用的资源*。

转换类型

设置不正确可触发下列类型的数据转换：

颜色格式转换

当一种格式与另一种不匹配时，会发生颜色格式转换。它可在以下情形中发生：

- 绘制表面格式与帧缓冲区格式不匹配
- 帧缓冲区和屏幕不匹配

图像缩放

绘制到与屏幕不同的分辨率可能会导致表面缩放至正确的分辨率。

备注

如果应用程序无法产生高帧率，这对于高分辨率显示是有用的。生成较低分辨率的帧，再让系统把它缩放到完整分辨率。

内存拷贝

在数据移动时会发生内存拷贝或位块传送。颜色格式转换和图像缩放也可触发位块传送。要避免，请使用直接渲染。请参见*尽可能使用直接渲染*（第 2-3 页）。

转换使用的资源

数据转换运算需要的资源包括：

- GPU 或应用处理器上的计算资源。
- 内存。
- 内存带宽。

用于转换的资源无法供应用程序使用，因此会对应用程序性能造成负面影响。

11.2.3 配置显示设置以避免转换

如果系统和应用程序具有正确且兼容的设置，就不需要数据转换。确保如下所列：

确保帧缓冲区分辨率和颜色格式与显示控制器兼容

显示控制器或许能够显示不同的格式和分辨率。如果帧缓冲区的分辨率和格式不兼容，就会执行颜色格式转换。

下列建议适用于使用 Linux FBDEV 的平台：

确保帧缓冲区不超过屏幕的分辨率

试图显示大于屏幕分辨率的内容将导致图像被重新缩放，或者显示不正确。

确保帧缓冲区不超过屏幕的色深

如果屏幕为 16 位色深，使用 32 位颜色帧缓冲区会造成显示绘制不正确，或者导致颜色格式转换。

在 32 位和 16 位色深之间转换是一个代价非常高的过程。通常由 GPU 完成，但在一些情形中必须由应用处理器执行。这会减少宝贵的计算资源。

与 32 位显示相比，使用 16 位显示时所用的内存和带宽比较少。不过，如果系统限制为 32 位显示，则不要使用 16 位帧缓冲区来节省内存。将 16 位数据转换到 32 位数据代价很大。

确保绘制表面格式与帧缓冲区格式相同

如果绘制表面格式与帧缓冲区格式不同，则需要转换后才能显示。

备注

如果您使用双缓冲或三缓冲，则内存中有多个帧缓冲区，但每次仅显示一个。

11.2.4 确保应用程序具有正确的绘制表面

应用程序请求绘制表面时，可能无法获得其请求的表面类型。这意味着您可能需要高于请求的色深。为了避免获得错误的表面，请在返回潜在表面时进行检查，并且仅接受正确的表面。

例如，如果您请求 RGB565 表面，您会看到一张 EGLConfig 列表。如果选择第一项配置，它可能是 RGBA8888 表面。这显然不是您要的表面。如果您迭代返回的配置，您可以直接选择 RGB565 格式，避免不正确的格式。

如需演示如何通过 EGLConfig 排序的示例代码，请访问 *Mali 开发者中心*，网址为 http://malideveloper.arm.com/。

11.3 使用 VSYNC

本节介绍*垂直同步* (VSYNC) 及其造成的问题。其中包含下列小节：

- *关于 VSYNC*。
- *使用 VSYNC*。
- *VSYNC 潜在问题*（第 11-9 页）。
- *三缓冲*（第 11-10 页）。

11.3.1 关于 VSYNC

VSYNC 将应用程序的帧率与屏幕显示速率同步。

VSYNC 是一项有用的技巧，因为它可以通过去除撕裂提高图像质量。它也可防止应用程序产生帧的速度快于屏幕可以显示的速度。您可以用它来节省功率。

备注

在进行其他优化前，请务必先取消激活 VSYNC。启用 VSYNC 后，帧率测量可能会不正确，从而可能导致您应用不正确的优化。

11.3.2 使用 VSYNC

要使用 VSYNC，请针对最高的可能帧率优化您的应用程序。其目标是让应用程序帧率远高于屏幕显示速率。

例如，假设应用程序产生帧的速度为 40 *帧/秒* (FPS)，屏幕显示速率为 30 FPS。图 11-4 显示每 3 次屏幕显示更新产生 4 帧。

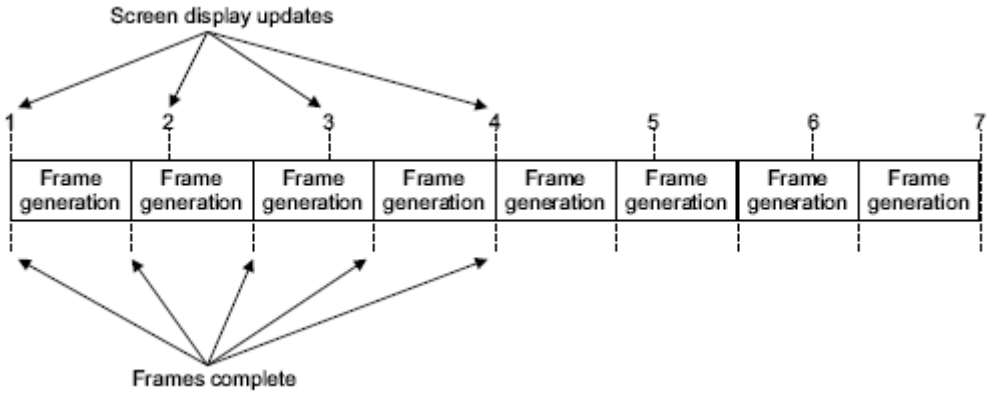

图 11-4 屏幕更新和帧完成

激活 VSYNC 会将应用程序帧率锁定为 30 FPS。应用程序产生一帧，然后停止产生新的图形，直到该帧显示出来为止。当帧显示在屏幕上时，应用程序启动下一帧。图 11-5（第 11-9 页）中显示了这一流程。由于 GPU 在帧产生结束和屏幕显示更新之间不活动，因此可以节省功率。

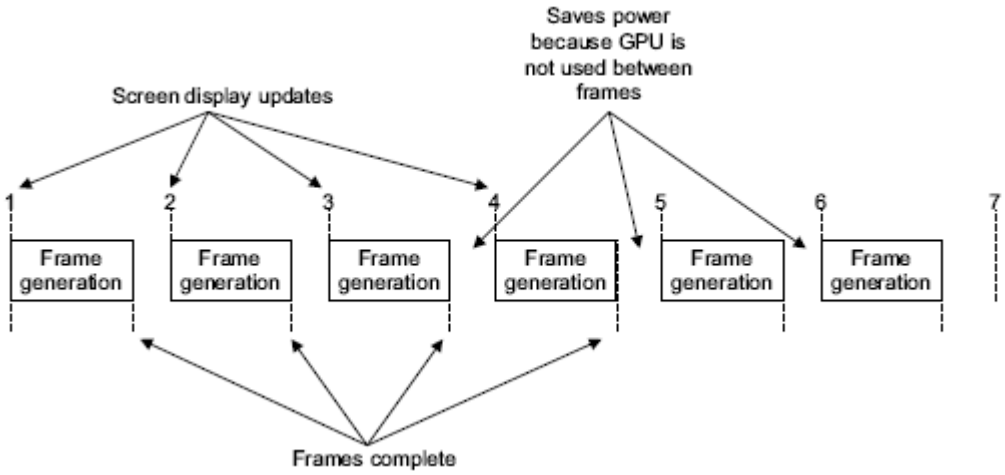

图 11-5 使用 VSYNC 时的屏幕更新和帧完成

11.3.3 VSYNC 潜在问题

VSYNC 可能不适用于帧率高度变化的应用程序，因为它可以表现为帧率中出现骤降。

当应用程序产生帧的时间比屏幕显示更新间更长时，会出现帧率下降。只有在屏幕下一次更新时，该帧才能显示出来。这具有二等分帧率的作用。其显示在图 11-6 中。

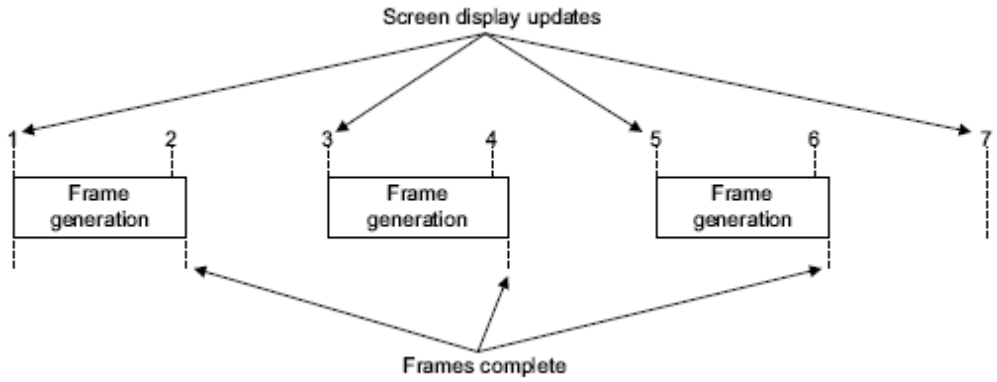

图 11-6 VSYNC 降低帧率时的屏幕更新和帧完成

备注

仅在您执行了任何其他优化后，再作为最后一步来激活 VSYNC。激活 VSYNC 后测量应用程序的性能，确保不会出现突然掉帧。

11.3.4 三缓冲

三缓冲这种技巧使用三个缓冲区来缓减或避免 VSYNC 可能导致的问题。这三个缓冲区为：

后置缓冲区

后置缓冲区存储正在绘制的图像。当 GPU 完成绘制时，后置缓冲区和中间缓冲区将切换。

中间缓冲区

中间缓冲区存储已完成的图像，直到前置缓冲区准备好显示它。当前帧完成显示时，前置缓冲区和中间缓冲区将切换。

前置缓冲区

前置缓冲区存储正在显示的图像。

使用三个缓冲区可以将绘制过程与显示过程分离。这意味着可利用 VSYNC 获得持续的高帧率，即使帧绘制用时长于帧显示时间。

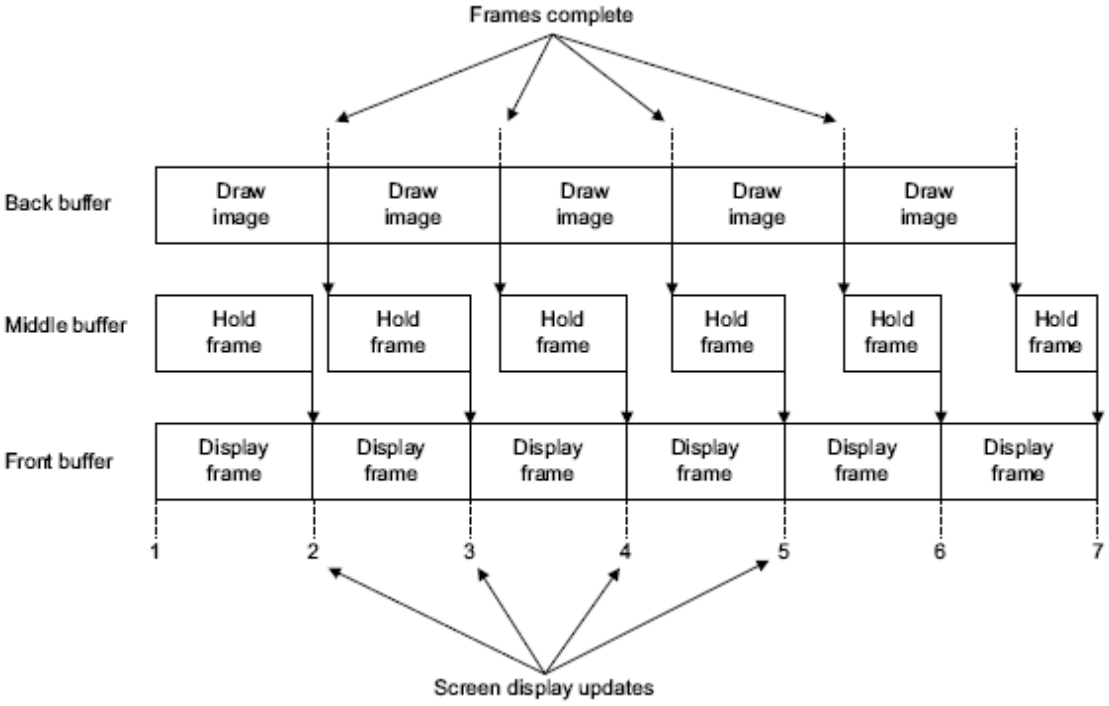

图 11-7 使用三缓冲和 VSYNC 时的屏幕更新

图 11-7 中显示三缓冲与 VSYNC 搭配。帧生成过程的用时比帧显示的长，但显示保持高帧率，因为 GPU 可以在中间缓冲区存储已完成的帧期间继续进行绘制。偶尔可能会掉帧。

备注

三缓冲要求三个大小与帧缓冲区相同的缓冲区。如果应用程序在较高的分辨率上绘制，这可能是大量的内存。

11.4 充分利用低效的资源

本节介绍如何充分利用低效的资源。其中包含下列小节：

- *使用空闲资源提高图像质量*。
- *使用空闲资源节省功率*。
- *将运算从片段处理器移到顶点处理器*。
- *将运算从顶点处理器移到片段处理器*。
- *将运算从应用处理器移到顶点处理器*（第 11-12 页）。

如果某一资源利用不充分，可考虑将计算移到其中。要判断是否有利用不充分的资源，您必须分析应用程序的性能。如需更多信息，请参见第 3 章 *优化流程*。

11.4.1 使用空闲资源提高图像质量

如果应用程序运行良好，您可以使用空闲资源，通过添加额外特效等方法来提高图像质量。

11.4.2 使用空闲资源节省功率

您可以选择不使用空闲资源。不使用空闲资源可以节省功率，因为 GPU 可以在不工作时关闭。

备注

在电池续航时间有限的移动设备中，这是一项有用的技巧。

11.4.3 将运算从片段处理器移到顶点处理器

在许多应用程序中，顶点处理器利用不充分。如果应用程序受片段处理器约束，或许可以将部分计算移到顶点处理器上。

例如，您可以通过使用可变，将一些类型的像素着色计算移到顶点处理器上。可变是按照顶点来计算的，并跨越三角形进行插值。与按照像素计算值相比，需要的计算比较少；但对一些特效，会产生较低的视觉质量。质量差异可能相对较小，所以可以考虑按像素运算是否值得额外的计算。

折衷的方案是将计算拆分到两个处理器上。您可以使用顶点处理器输出可变，并在片段处理器中计算每像素差异。

11.4.4 将运算从顶点处理器移到片段处理器

如果应用程序受顶点处理器约束，并且将大量几何用于细节，那么您或许可以使用让表面看起来比实际更富有细节的技巧减少几何。例如，法线贴图是呈现表面法线的纹理。着色器程序可以使用法线贴图造成更多表面细节的印象，而不会增加三角形数量。

请参见*使用法线贴图模拟精细几何*（第 8-5 页）。

11.4.5 将运算从应用处理器移到顶点处理器

您可以使用顶点处理器来计算动画。图 11-8 显示了演示中的一个帧，其中使用顶点着色器生成植物动画。

图 11-8 植物动画

附录 A
Mali GPU 性能计数器

本章提供 Mali GPU 性能计数器列表。其中包含下列小节：

A.1 顶点处理器性能计数器

表 A-1 列出监控顶点处理器的 Mali GPU 性能计数器。

表 A-1 顶点处理器性能计数器

计数器名称	说明
活动周期数	顶点处理器活动时每帧的周期数。
活动周期数，边界性框体和命令产生器	顶点处理器多边形列表构建器单元设置边界性框体和命令时在每帧上花费的活动周期数。这主要是图形原语。其中包括总线上等待花费的时间。
活动周期数，PLBU 命令处理器	顶点处理器 PLBU 命令处理器活动时每帧的周期数，包括等待信号量的时间。
活动周期数，PLBU 几何处理	顶点处理器 PLBU 活动时每帧的周期数，最终数据输出除外。这是通过准备列表命令的活动周期数。其中包括总线上等待花费的时间。
活动周期数，PLBU 原语汇编	顶点处理器 PLBU 执行原语汇编时每帧所花费的周期数。这不包括剪切或最终输出。但包括总线上等待花费的时间。
活动周期数，PLBU 区块迭代器	顶点处理器 PLBU 在边界性框体中迭代区块产生命令时每帧所花费的活动周期数，这主要是由于图形原语。其中包括总线上等待花费的时间。
活动周期数，PLBU 顶点提取器	顶点处理器 PLBU 提取顶点数据时每帧所花费的活动周期数。其中包括总线上等待花费的时间。
活动周期数，剪切区块迭代器	顶点处理器 PLBU 迭代区块以执行剪切时每帧所花费的活动周期数。其中包括总线上等待花费的时间。
活动周期数，顶点加载器	顶点加载器单元活动时每帧的周期数。
活动周期数，顶点着色器	顶点着色器单元活动时每帧的周期数。
活动周期数，顶点着色器命令处理器	顶点着色器命令处理器活动时每帧的周期数。这包括等待信号量的时间。
活动周期数，顶点存储器	顶点存储器单元活动时每帧的周期数。
写入区块的命令数	顶点处理器每一帧向片段处理器输入数据结构写入的命令数。 这些命令为八字节，主要为原语。
顶点加载器等待顶点着色器的周期数	顶点加载器在等待顶点着色器时每帧空闲的周期数。
Mali GP2 PLBU 每帧周期数	顶点处理器 PLBU 输出单元活动且写入片段处理器输入数据结构时每帧的周期数，包括在总线上等待花费的时间。
分配的内存块数	输出片段处理器输入数据结构每帧所需的溢出数据块数量。
剔除的原语数	由于从背面查看或者由于在视域之外，每帧被剔除的图形原语数。
提取的原语数	顶点处理器每帧所提取的图形原语数量。
读取突发数，系统总线	顶点处理器每帧从系统总线读取的突发数量。

顶点加载器缓存未命中数	顶点着色器顶点输入单元每帧缓存未命中数。

表 A-1 顶点处理器性能计数器（续）

计数器名称	说明
已提取顶点数	顶点处理器每帧提取的顶点数量。
已处理顶点数	顶点处理器每帧处理的顶点数量。
字读取数，系统总线	顶点处理器每帧从系统总线读取的 64 位词语总数。
字写入数，系统总线	顶点处理器每帧向系统总线写入的 64 位词语总数。
写入突发数，系统总线	顶点处理器每帧向系统总线写入的突发数量。

A.2 片段处理器性能计数器

表 A-2 列出监控片段处理器的 Mali GPU 性能计数器。

表 A-2 片段处理器性能计数器

计数器名称	说明
活动时钟周期数	多边形开头和 IRQ 之间活动的时钟周期数量。
总线读取请求周期数	总线读取请求信号为 HIGH 的周期数。
总线写入请求周期数	总线写入请求信号为 HIGH 的周期数。
总线读取交易数	总线接受的读取请求数。
总线写入交易数	总线接受的写入请求数。
压缩纹理缓存压缩命中数	压缩纹理的纹理缓存命中数。
压缩纹理缓存压缩未命中数	压缩纹理的纹理缓存未命中数。
通过 z 模板片段数	通过 Z 和模板测试的片段数。
栅格化片段数	栅格化的片段数量。片段数/(四元组数 x 4) 可以得出每四元组实际片段的平均数。
片段终止拒绝片段数	因被终止而从片段着色器退出的片段数。
转发片段终止拒绝片段数	被转发片段终止而终止的片段数。
已完成指令数	已完成的片段着色器指令字数。此为已处理片段数和着色器程序长度的函数。已完成指令数的公式为： (四元组数) x (每四元组中像素数) x (着色器中的指令数)。
加载未命中失败指令数	因为加载运算失败而未能完成的片段着色器指令数。
存储未命中失败指令数	因为存储运算失败而未能完成的片段着色器指令数。
纹理未命中失败指令数	因为纹理运算失败而未能完成的片段着色器指令数。
区块读取未命中失败指令数	因为从区块缓冲区读取失败而未能完成的片段着色器指令数。
可变未命中失败指令数	因为可变运算失败而未能完成的片段着色器指令数。
行数	从多边形列表读取的行数。
加载单元读取数	LOAD 子指令从总线读取的 64 位字数。
加载/存储缓存命中数	加载/存储缓存中的命中数。
加载/存储缓存未命中数	加载/存储缓存中的未命中数。
评估补丁数	为 EarlyZ 拒绝评估的补丁数。
EarlyZ/模板拒绝补丁数	被 EarlyZ 拒绝的补丁数。补丁可以为 8x8、4x4 或 2x2 个片段。
流水线气泡周期数	渲染活动时片段着色器中未使用的周期数。
像素矩形数	从多边形列表读取的像素矩形数。

表 A-2 片段处理器性能计数器（续）

计数器名称	说明
点数	从多边形列表读取的点数。
多边形数	从多边形列表读取的三角形数。
多边形列表读取数	多边形列表读取程序从总线读取的 64 位字数。
程序缓存命中数	程序缓存中的命中数。
程序缓存未命中数	程序缓存中的未命中数。
程序缓存读取数	从总线读取到片段着色器程序缓存中的 64 位字数。
栅格化四元组数	从栅格化程序输出的 2x2 四元组数。
RSW 读取数	从总线读取到渲染状态字寄存器中的 64 位字数。
PolygonListReader 停滞周期数	多边形列表读取程序等待输出被收集的时钟周期数。
三角形设置停滞周期数	TSC 等待输入的时钟周期数。
存储单元写入数	写入总线的 64 位字数。
纹理缓存冲突未命中数	请求的纹素不在缓存中并且其从内存检索的值必须覆盖较旧的缓存条目的次数。当存取模式无法被缓存服务时就会发生此情况。
纹理缓存命中数	请求的纹素在纹理缓存中被找到的次数。
纹理缓存未命中数	请求的纹素在纹理缓存中无法找到的次数。
纹理缓存未压缩读取数	从总线读取到未压缩纹理缓存中的 64 位字数。
纹理描述符重新映射读取数	从纹理描述符重新映射表中读取时从总线读取的 64 位字数。
纹理描述符读取数	从总线读取的包含纹理描述符的 64 位字数。
纹理映射器周期数	纹理运算周期数。
纹理映射器多通道数	由于需要更多纹理通道而循环的纹理运算数。
区块写回写入数	由写回单元写入总线的 64 位字数。
总线读取总数	从总线读取的 64 位总字数。
总线写入总数	写入总线的 64 位总字数。
统一重新映射读取数	从统一重新映射表中读取时从总线读取的 64 位字数。
可变缓存冲突未命中数	请求的可变不在缓存中并且其从内存检索的值必须覆盖较旧的缓存条目的次数。当存取模式无法被缓存服务时就会发生此情况。
可变缓存命中数	请求的可变在缓存中被找到的次数。
可变缓存未命中数	请求的可变在缓存中无法找到的次数。
可变读取数	从总线读取的包含顶点处理器生成的可变的 64 位字数。

表 A-2 片段处理器性能计数器（续）

计数器名称	说明
顶点缓存命中数	请求的顶点在缓存中被找到的次数。
顶点缓存未命中数	请求的顶点在缓存中无法找到的次数。
顶点缓存读取数	从总线读取到顶点缓存中的 64 位字数。