Mali™ GPU

Version: 1.0

Application Optimization Guide

ARM®

ARM DUI 0555A (ID033111)

Mali GPU
Application Optimization Guide

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
30 March 2011	A	Confidential	First release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
Mali GPU Application Optimization Guide

List of Tables
Mali GPU Application Optimization Guide

List of Figures
Mali GPU Application Optimization Guide

Preface

This preface introduces the *Mali GPU Application Optimization Guide*. It contains the following sections:

About this book

This book is for the Mali-200, Mali-300 and Mali-400 MP *Graphics Processor Units* (GPUs).

Intended audience

This book is written for application developers who are developing or porting applications on to platforms with Mali GPUs. This guide assumes application developers have some knowledge of 3D graphics programming but does not assume they are experts.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this for an introduction to optimizing for Mali GPUs.

Chapter 2 *Optimization Checklist*

Read this for a list of things to check for before starting a full optimization process.

Chapter 3 *The Optimization Process*

Read this for a description of a full optimization process.

Chapter 4 *Optimization Techniques*

Read this for a list of optimization techniques that are more complex than those in Chapter 2.

Glossary Read this for definitions of terms used in this book.

Conventions

The typographical conventions that this book can use are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`

Additional reading

This section lists publications by ARM and by third parties.

See Mali Developer Center, `http://www.malideveloper.com`, for access to Mali GPU developer documentation.

See Infocenter, `http://infocenter.arm.com`, for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *Mali GPU Performance Analysis Tool User Guide* (ARM DUI 0502)
- *Mali GPU OpenGL ES Application Development Guide* (ARM DUI 0363)

Other publications

This section lists relevant documents published by third parties:

- EGL 1.4 Specification - http://www.khronos.org
- OpenGL ES 1.1 Specification - http://www.khronos.org
- OpenGL ES 2.0 Specification - http://www.khronos.org

Feedback

ARM welcomes feedback on this document.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- the title, Mali GPU Application Optimization Guide
- the number, ARM DUI 0555A
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the Mali GPU Application Optimization Guide. It contains the following sections:

1.1 About optimization

This section describes optimization. It contains the following sections:

- *What is optimization?*
- *Why optimize?*
- *Aims of optimization*.

1.1.1 What is optimization?

Graphics is about making things look good. Optimization is about making things look good with the least computational effort.

Optimization is the process of taking an application and making it more efficient. For graphical applications this typically means modifying the application to make it faster.

1.1.2 Why optimize?

A low frame rate means the application appears to jump. This gives the impression of low quality, and makes applications such as games difficult to play.

You can use optimization to improve the frame rate of an application. This makes using the application a better, smoother experience.

Note

A consistent frame rate is typically more important than a high frame rate. A frame rate that varies gives a worse impression than a relatively low but consistent frame rate.

1.1.3 Aims of optimization

There are many different things you can optimize for. For example:

- increasing the frame rate
- making content more detailed
- reducing power consumption
  - using less memory bandwidth
  - using fewer clock cycles per frame.
- reducing memory foot print
- reducing download size.

The different optimizations are often interlinked. For example, you can optimize for a faster frame rate but keep the application at the pre-optimization frame rate on the target platform. This reduces power consumption even though you have not directly optimized for it. Power is saved because the GPU requires less time to compute frames and can rest for longer periods.

Optimizing to reduce memory footprint is not a typical optimization, but it can be useful because smaller applications are more cacheable. In this case, making the application smaller can also have the effect of making the application frame rate higher.

Note

This guide primarily concentrates on making the application frame rate higher. Where appropriate the other types of optimization are mentioned.

1.2 The graphics pipeline

The Mali-200, Mali-300 and Mali-400 MP GPUs implement a graphics pipeline supporting the OpenGL ES and OpenVG *Application Programming Interfaces* (APIs). This section describes the OpenGL ES graphics pipeline, it contains the following sections:

- *OpenGL ES Graphics pipeline overview*
- *Initial processing*
- *Per-vertex operations*
- *Rasterization and fragment shading* on page 1-4
- *Blending and framebuffer operations* on page 1-4.

1.2.1 OpenGL ES Graphics pipeline overview

Figure 1-1 shows a typical flow for the OpenGL ES graphics pipeline.

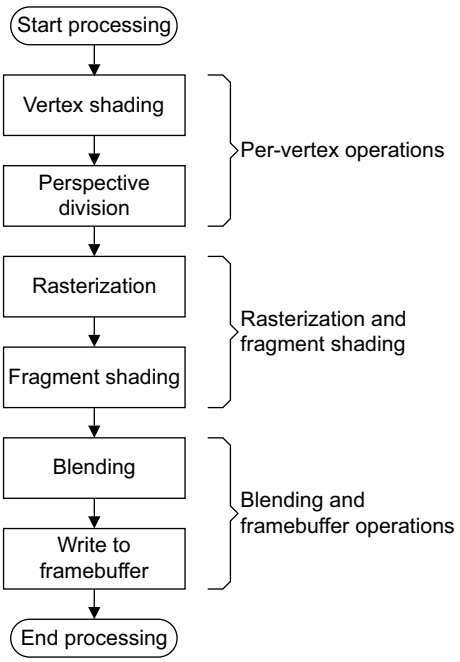

Figure 1-1 Graphics pipeline flow

Mali GPUs use data structures and functional blocks to implement the OpenGL ES graphics pipeline.

1.2.2 Initial processing

The API-level drivers for OpenGL ES or OpenVG create data structures in memory for the GPU and configure the hardware for each scene.

The software:

- generates data structures and texture descriptors
- creates command lists for the *Geometry Processor* (GP)
- compiles shaders on demand.

1.2.3 Per-vertex operations

The geometry processor runs a vertex shader program for each vertex.

This shader program performs:

- lighting
- transforms
- viewport transformation
- perspective transformation
- vertex assembly of graphics primitives
- polygon list builds for the pixel processor.

1.2.4 Rasterization and fragment shading

The *Pixel Processor* (PP) reads in the state information, polygon lists, and transformed vertex data. It performs rasterization and texturing.

The pixel processor rasterizes the polygons and runs fragment shaders. The rasterizer takes the coefficients from the triangle setup unit and applies equations to create fragments. A fragment shader program is then run on each fragment to calculate the color of the fragment.

1.2.5 Blending and framebuffer operations

The pixel processor produces the final display data for the framebuffer after processing the tile buffer. To increase processing speed, each pixel processor processes a different tile.

The blending unit blends the fragments with the color already present at the corresponding location in the tile buffer.

The pixel processor tests the fragments and updates the tile buffer. The pixel processor calculates if fragments are visible or hidden and stores the visible fragments in tile buffers.

The pixel processor writes the contents of the tile buffer to the framebuffer after the tile has been completely rendered.

1.3 The Mali GPU hardware

This section describes the main hardware components of the Mali GPU hardware. It contains the following sections:

- *Tile based rendering*
- *Mali GPU hardware components*
- *The Geometry processor* on page 1-6
- *The Pixel processors* on page 1-6
- *L2 cache controller* on page 1-6.

1.3.1 Tile based rendering

Mali GPUs use *tile-based deferred* rendering.

The Mali GPU divides the framebuffer into tiles and renders it tile by tile. Tile based rendering is efficient because values for pixels are computed using an on-chip memory. This technique is ideal for mobile devices because it requires less memory bandwidth and less power than traditional rendering techniques.

1.3.2 Mali GPU hardware components

A Mali GPU is typically used in a mobile or embedded environment to accelerate 2D and 3D graphics. The graphics are produced using an OpenGL ES or OpenVG graphics pipeline. See *The graphics pipeline* on page 1-3.

Mali GPUs are configurable so they can contain different components. The types of components a Mali GPU can contain are:

- geometry processor
- pixel processors
- *Memory Management Unit*s (MMU)
- *Power Management Unit* (PMU)
- L2 cache.

Table 1-1 shows the number of geometry and pixel processors in the different Mali GPUs.

Table 1-1 Possible Mali GPU components

Mali GPU	Geometry Processor	Pixel Processors
Mali-55	-	1
Mali-200	1	1
Mali-300	1	1
Mali-400	1	1-4

Note

This document does not apply to the Mali-55 or Mali T-604 GPUs.

Figure 1-2 on page 1-6 shows a Mali-400 MP GPU.

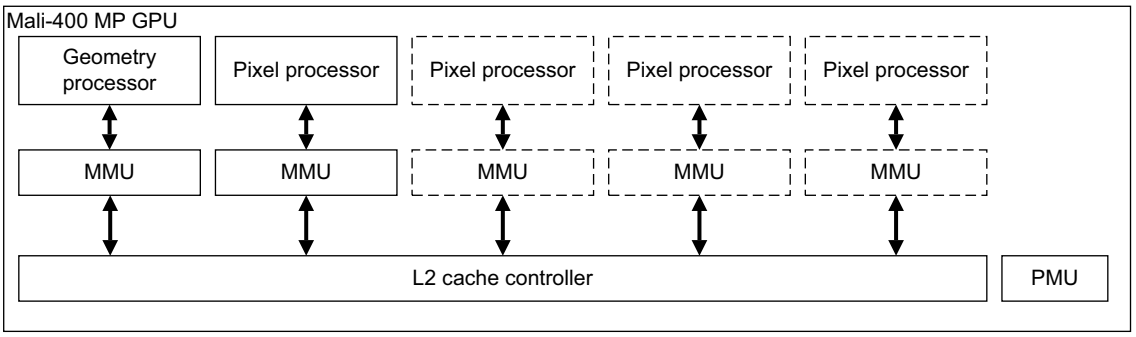

Figure 1-2 Mali-400 MP GPU

A general-purpose CPU runs the operating system, graphics applications, and manages tasks for the Mali GPU.

1.3.3 The Geometry processor

The geometry processor handles the vertex processing stage of the graphics pipeline. It generates lists of primitives and accelerates the building of data structures, such as polygon lists and packed vertex data, for the pixel processor.

1.3.4 The Pixel processors

The pixel processors handle the rasterization and fragment processing stages of the graphics pipeline. They use the data structures and lists of primitives generated by the geometry processor to produce the framebuffer result that the screen displays as a final image.

Note

The Mali-200 and Mali-300 GPUs have one pixel processor. The Mali-400 MP GPU can contain up to four pixel processors.

1.3.5 L2 cache controller

An L2 cache controller is available with the Mali-300 and Mali-400 MP GPUs. An L2 cache reduces memory bandwidth usage and power consumption.

An L2 cache is designed to hide the cost of accessing memory. Main memory is typically slower than the CPU so L2 can increase performance considerably in some applications.

1.4 Differences between desktop systems and mobile devices

Mobile and embedded systems must balance compute power, battery life and cost. This means mobile platforms are constrained in terms of available resources such as:

- CPU capabilities
- memory capacity
- memory bandwidth
- power consumption
- physical size.

Desktop systems do not have these limitations so application developers have much more compute resources to utilise.

Mali GPUs are typically used in mobile or embedded systems so it is important to be aware of these differences if you are porting a graphics application from a desktop platform.

Some graphically rich applications are initially developed for desktop platforms and are later ported to embedded or mobile platforms. The reduction in available resources means that the application is unlikely to work at the same performance level as it does on the desktop platform.

Optimization enables your application to get closer to the performance level it achieves on a desktop platform.

1.5 Differences between mobile renderers

The section describes differences between mobile renderers. It contains the following sections:

- *Differences with other mobile GPUs*
- *Differences with software renderers*.

1.5.1 Differences with other mobile GPUs

All GPUs have different optimization points. Do not assume applications optimized for one platform automatically perform well on another.

One specific optimization to do for Mali GPUs is to sort objects or triangles into front-to-back order in your application. This reduces overdraw, see *Avoid overdraw* on page 4-5.

1.5.2 Differences with software renderers

If your application runs on existing mobile devices with a software rendering 3D engine, the application might not run well on a Mali GPU. This is because the optimizations for using a GPU are different from those for software rendering 3D engines.

To obtain high performance with a GPU, your application requires re-optimization. In particular ensure you do not use a large number of draw calls per frame. For more information, see *Minimize draw calls* on page 4-2.

Chapter 2
Optimization Checklist

This chapter provides a checklist to go through before starting a full optimization process. It contains the following sections:

Note

These techniques can have a very large impact on performance, so ensure you have checked these before moving onto the following chapters.

2.1 About the optimization checklist

Applications can under-perform for a number of reasons. Optimizing 3D applications is a complex topic with many different techniques that can be used in different circumstances.

Many performance problems can be fixed relatively easily. This chapter lists some examples of this type of problem, and describes techniques to fix them.

Most of these are relatively simple optimization techniques that you can use to improve performance and quality of graphics.

Note

Some optimizations conflict and do not work well if they are used together. For example, if you use depth sorting to reduce overdraw, attempting to also reduce state changes can conflict with this optimization.

2.2 Check the display settings

This section describes what to check for display settings. It contains the following sections:

- *About display settings*
- *Data conversions caused by incorrect settings*
- *Configuring display settings to avoid conversions* on page 2-5.
- *Ensure your application has the correct drawing surface* on page 2-5.

2.2.1 About display settings

For a system to function, the following settings must be configured:

Drawing format The drawing format is the color format that your application draws with. The application draws graphics on the drawing surface using the drawing format.

Drawing surface The drawing surface is an area in memory that your application draws graphics into. The surface can be in different configurations known as configs. Each config has different settings for resolution and color depth.

Framebuffer This is a data structure that contains the data that is sent to the screen for display.

Display controller The display controller is a hardware component that sends data from the framebuffer to the screen.

Screen The screen is the physical display device that you look at. This has a maximum resolution and color depth.

2.2.2 Data conversions caused by incorrect settings

This section describes the data conversions caused by incorrect settings and the resources they require. It contains the following sections:

- *Types of conversions*
- *Resources used by conversions* on page 2-5.

Types of conversions

The following types of data conversions can be triggered by incorrect settings:

Color format conversion

Color format conversion happens when one format does not match another. This can happen if:

- the drawing surface format does not match the framebuffer format
- the framebuffer and screen do not match.

Image scaling Drawing to a different resolution from the screen might cause the surface to be scaled to the correct resolution.

Memory copies Memory copies or blitting happen when data is moved. Color format conversions and image scaling can also trigger blitting. To avoid this use direct rendering. See *Use direct rendering if possible* on page 2-7

Resources used by conversions

Data conversion operations require resources, including:

- compute resources on the GPU or CPU
- memory
- memory bandwidth.

Resources used for conversions cannot be used by applications. This has a negative impact on the performance of your application.

2.2.3 Configuring display settings to avoid conversions

Data conversions are not necessary if the system and your applications have correct and compatible settings. Ensure the following:

Ensure the framebuffer resolution and color format are compatible with the display controller.

The display controller might be able to display different formats and resolutions. If the framebuffer is not in a resolution and format that is compatible, a color format conversion is performed.

The following advice applies to platforms that use Linux `FBDEV`:

Ensure the framebuffer does not exceed the resolution of the screen.

Trying to display something larger than the resolution of the screen shall result in the image being rescaled or not being shown correctly.

Ensure the framebuffer does not exceed the color depth of the screen.

If the screen has a 16-bit color depth, using a 32-bit color framebuffer results in the display being drawn incorrectly or a color format conversion.

Converting between 32 and 16 color depths can be a very expensive process. It is typically done by the GPU, but in some cases it must be performed by the CPU. This reduces valuable compute resources.

Using a 16-bit display uses less memory and bandwidth than a 32-bit display. If however your system is limited to a 32-bit display, do not write 16-bit display data to save memory. Converting 16-bit data to 32-bit data can be an expensive process.

Ensure the drawing surface format is the same as the framebuffer format.

If the drawing surface format is different from the framebuffer format, a conversion is required to display it.

Note

If you are using double or triple buffering, there are multiple framebuffers in memory but only one is displayed at a time.

2.2.4 Ensure your application has the correct drawing surface

When your application requests a drawing surface it might not get the type of surface it requested. This means you might get a higher color depth than you requested. To avoid getting the wrong surface, check potential surfaces as they are returned and only accept the correct one.

For example, if you request a RGB565 surface you are presented with a list of configs. If you pick the first config it might be an RGBA8888 surface. This is obviously not the surface you want. If you iterate through the configs returned you can select the RGB565 format directly and avoid the incorrect formats.

For example code that shows how to sort through EGLConfigs, see the *Mali Developer Center*, `http://www.malideveloper.com/`.

2.3 Use direct rendering if possible

Typically graphics are drawn to an off-screen buffer and then blitted into the frame buffer for display.

Blitting is an expensive operation that takes time and consumes a lot of memory bandwidth. You can improve performance significantly by avoiding it.

The process of drawing graphics directly into the framebuffer is called direct rendering. If possible, use direct rendering to avoid blitting and increase the performance of your application. Using direct rendering is OS-specific, so see the documentation for your OS to check if it is available and how to use it.

2.4 Use the correct tools with the correct settings

This section describes using the correct tools. It contains the following sections:

- *Use the latest tools*
- *Rebuild everything after a tools update*
- *Build for the correct architecture*
- *Use the facilities in your hardware*
- *Optimize your release build*.

2.4.1 Use the latest tools

Compile your application with the latest versions of your development tools. This ensures your application benefits from the latest optimizations and bug fixes.

2.4.2 Rebuild everything after a tools update

If you change tools or versions of tools, ensure you recompile everything so everything gets the same benefits from the changes.

2.4.3 Build for the correct architecture

There are different versions of CPU architectures. To ensure the best performance ensure you build for the correct version. If you build for an older architecture version and run on a newer version, performance can be reduced.

2.4.4 Use the facilities in your hardware

If your platform has hardware floating point, *Vector Floating Point* (VFP), or NEON™, ensure the compiler is set to build for it. Also consider using libraries that take advantage of these hardware features.

Note

If your operating systems supports hard floating point, ensure the entire system and support libraries are built to support it.

2.4.5 Optimize your release build

Ensure that for release, you set your compiler to produce binaries optimized for speed. These provide the best performance.

2.5 Remove debugging information

Gathering debugging information is useful for correcting errors but it requires memory and compute resources. However the process of gathering debugging information typically has a negative impact on performance.

Ensure you switch off debugging before releasing your application. For other pre-release checks, see *Final release check list* on page 2-28.

Use minimal `printf()` calls

`printf()` calls can be very slow. You can prevent them from impacting application performance by only displaying the frame rate after a relatively large number of frames. For example, make a `printf()` call every 100 frames, not one every second or one every frame.

Do not call `glGetError()` more than one time per frame

Every call to `glGetError()` takes time to process. A large number of these per frame consumes sufficient compute resources to limit the frame rate of the application. Ensure you make no more than one `glGetError()` call per frame.

Note

You can use `#define` macros to build the debug code for development builds and remove it for release builds.

Note

If the application is gathering debugging information while taking measurements, these measurements are likely to be inaccurate.

2.6 Avoid infinite command lists

This section describes infinite command lists and the issues they can cause.

The process of deferred rendering involves placing commands into lists. If you do not clear buffers between frames, the command lists can keep growing. This causes the Mali GPU to repeat work already completed for previous frames. This is obviously more work than necessary.

Note

- This issue is typically only a problem if your application renders to a surface such as a `pixmapsurface`, or `pbuffersurface`, and it does not clear the command lists at the end of a frame.
- This issue is not a problem if your application uses *Frame Buffer Objects* (FBO).
- An application that renders to a `eglWindowSurface` naturally ends the frame every time it calls `eglSwapBuffers`.

To prevent command lists growing, ensure your application clears the following buffers before drawing a new frame:

- color buffers
- depth buffers
- stencil buffers.

You must clear all these buffers at the same time.

2.7 Avoid calls that stall the graphics pipeline

There are a number of OpenGL ES function calls that can cause the graphics pipeline to stall. These can have a significant impact on performance so avoid using them.

Mali GPUs uses deferred rendering. The GPU receives all draw calls, then draws the frame.

Some OpenGL ES function calls read from the framebuffer. To do this the Mali GPU must first render the entire image before you can read back from it. This operation is likely to be slow.

The following OpenGL ES calls are best avoided because they can stall the graphics pipeline:

glReadPixels()

This call stalls the graphics pipeline and this can lead to a significant performance reduction. Avoid calls to `glReadPixels()` if at all possible.

glCopyTexImage()

`glCopyTexImage()`

ARM recommends that if you are using OpenGL ES 2.0, you use frame buffer objects instead of this call. This enables you to draw directly to a texture rather than using a copy.

glTexSubImage()

This call stalls the pipeline if you attempt to modify a pixmap image that has not completed rendering.

Note

This call only stalls if you are rendering to a pixmap.

2.8 Do not compile shaders every frame

It is possible to compile shaders for every frame. This reduces the performance of your application because shader compilation requires CPU and memory resources.

It is more efficient to compile shaders when your application starts. This only requires resources when your application starts, so it does not reduce the performance of your application while it is running.

You can also ship your application with pre-compiled shaders. These only require linking at runtime so require relatively little runtime CPU resources.

2.9 Use VSYNC

This section describes *Vertical Synchronization* (VSYNC) and the issues it can cause. It contains the following sections:

- *About VSYNC*
- *Using VSYNC*
- *Potential issues with VSYNC* on page 2-14
- *Triple buffering* on page 2-15.

2.9.1 About VSYNC

VSYNC synchronizes the frame rate of your application with the screen display rate.

VSYNC is a useful technique because it improves image quality by removing tearing. It also prevents the application producing frames faster than the screen can display them. You can use this to save power.

Note

Ensure you deactivate VSYNC before doing any other optimizations. If VSYNC is enabled frame rate measurements are likely to be incorrect and can lead you to applying incorrect optimizations.

2.9.2 Using VSYNC

To use VSYNC, optimize your application for the highest possible frame rate. The aim is to have the application frame rate significantly higher than the screen display rate.

For example, assume your application produces frames at 40 *Frames Per Second* (FPS) and the screen display rate is 30 FPS. Figure 2-1 shows that 4 frames are generated for every 3 screen display updates.

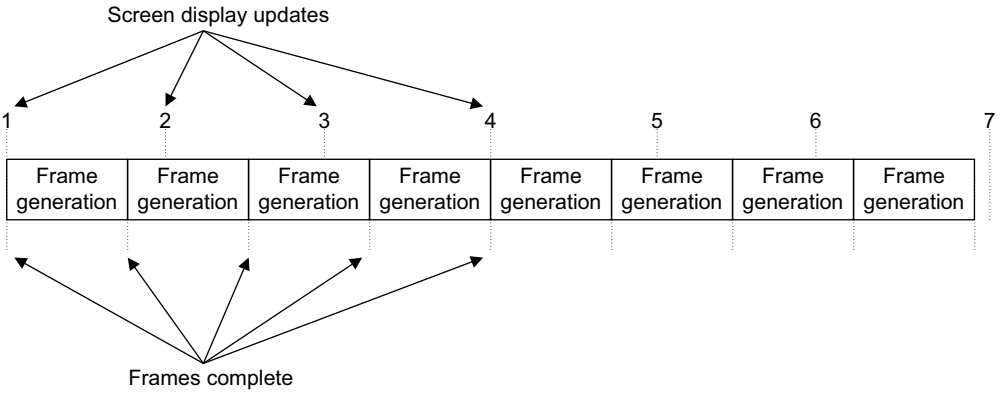

Figure 2-1 Screen updates and frame completes

Activating VSYNC locks the application frame rate to 30 FPS. The application generates a frame then stops generating any new graphics until after the frame is displayed. When the frame is displayed on screen the application starts the next frame. This process is shown in Figure 2-2 on page 2-14. Power is saved because the GPU is not active between the end of frame generation and the screen display update.

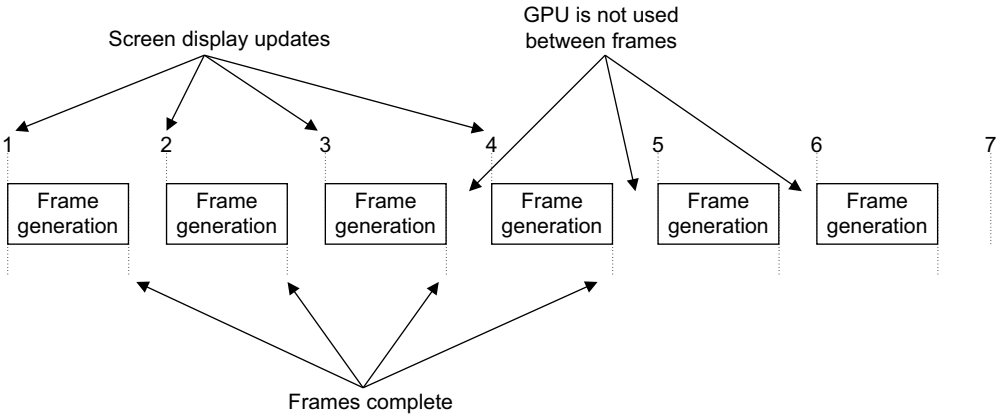

Figure 2-2 Screen updates and frame completes with VSYNC

2.9.3 Potential issues with VSYNC

VSYNC might not be appropriate for applications that have a highly variable frame rate. This is because VSYNC can cause sudden drastic drops in frame rate. This happens if the time for an application to produce a frame is longer than the time between screen display updates. The frame can then only be displayed at the next screen update. This has the effect of halving the frame rate. This is shown in Figure 2-3.

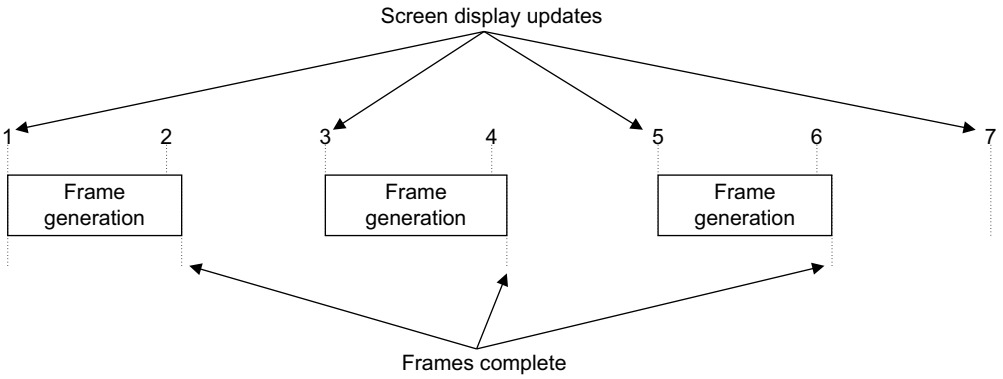

Figure 2-3 Screen updates and frame completes with VSYNC reducing frame rate

Note

Only activate VSYNC as a final step after you have performed any other optimizations. Measure the performance of your application after you activate VSYNC to ensure there are no sudden frame drops.

2.9.4 Triple buffering

Triple buffering is a technique that uses three buffers to reduce or avoid the problems VSYNC can cause. The three buffers are:

The frontbuffer

The frontbuffer holds the image currently being displayed.

The middlebuffer

The middlebuffer holds a completed image until the front buffer is ready to display it. When the current frame finishes displaying, the frontbuffer and middlebuffer switch.

The backbuffer

The backbuffer holds the image being drawn. When the GPU finishes drawing the backbuffer and middlebuffer switch.

Using three buffers decouples the drawing process from the display process. This means that a consistent high frame rate is achieved with VSYNC, even if frame drawing takes take longer than the frame display time.

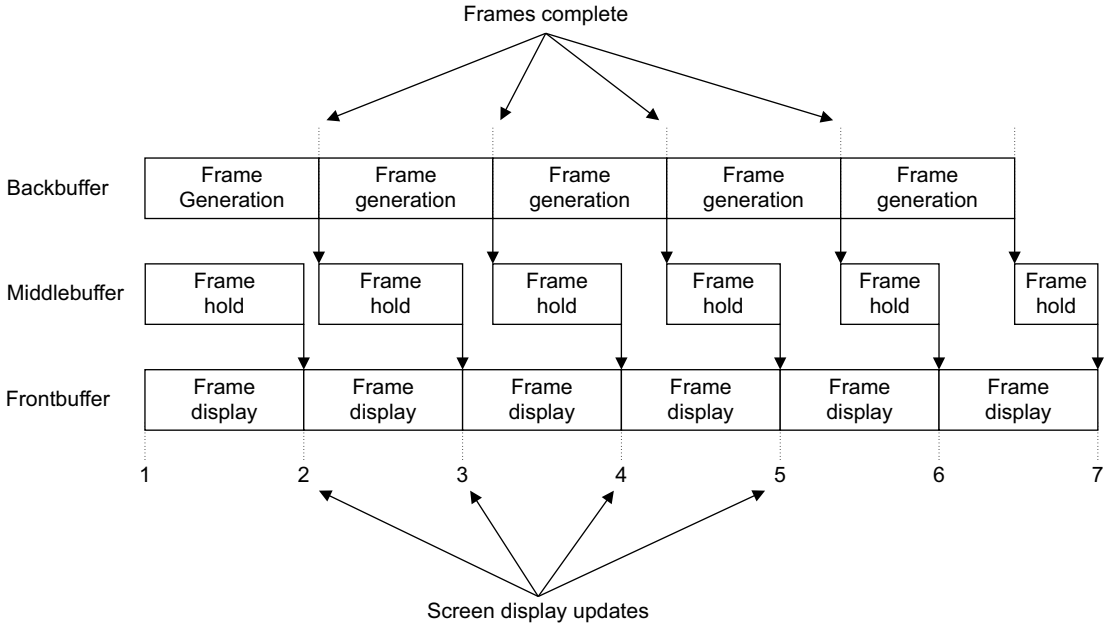

Figure 2-4 Screen updates with triple buffering and VSYNC

Triple buffering with VSYNC is shown in Figure 2-4. The frame generation process takes longer than the frame display, but the display frame rate remains high because the GPU can keep drawing while the middle buffer holds a completed frame. A frame might be dropped occasionally.

Note

Triple buffering requires a third buffer the size of the frame buffer. This can be a significant amount of memory if your application is drawing to a high resolution.

2.10 Use graphics assets appropriate for your platform

Mobile platforms typically have small screens with relatively low resolutions compared to desktop systems.

If you are porting a desktop application to a mobile platform, fine details are likely to have little visual impact. This enables you to simplify graphics assets. You can make changes such as:

- reducing the size and bit depth of textures
- reducing geometry complexity
- simplifying or removing effects that have little visible impact.

These changes reduce memory bandwidth usage. This enables higher performance.

For more information on simplification, see *Use approximations to improve performance* on page 4-7.

2.11 Do not use 24-bit textures

For high bit depth textures, use 16-bit or 32-bit textures rather than 24-bit textures.

24-bit textures do not fit neatly into cache. Using 24-bit textures can cause data to use more than one cache line and this has a negative impact on performance and memory bandwidth.

16-bit and 32-bit textures fit into caches without problems so they do not suffer from these performance issues.

Note

For most textures, it is better to use texture compression than high bit depth textures. Compressed textures can improve performance because they use less memory bandwidth. For more information, see *Use texture compression* on page 2-19.

2.12 Use mipmapping

Mipmapping is a very useful technique that:

- reduces memory bandwidth usage
- increases texture cacheability
- increases image quality
- potentially provides very large performance gains.

When your application draws an object on screen, the image drawn can be very different in size. It can range from filling the screen to being a small object in the distance.

If a single texture is used, the density that the texture is sampled at, or *texture sampling density*, is only correct when the object drawn is similar to the size of the texture. If the object size and the texture size do not match, the texture sampling density is incorrect. This produces artifacts that reduce image quality.

Mipmapping gets around this problem by taking a high resolution texture and scaling it to multiple smaller sizes known as *mipmap levels*. When an object is drawn with mipmapping enabled, the texture with the closest size to the object is used. This means the object always has a texture of a matching size to take samples from and the texture sampling density is therefore correct. This reduces image artifacts and produces a higher quality image.

If the texture sampling density is correct, the texels that are sampled are close to one another in memory. This means the texture data is more cacheable. Increased cacheability reduces memory bandwidth usage and increases performance.

You can instruct the Mali GPU driver to generate mipmaps at runtime or you can pre-generate the mipmaps with the *Mali GPU Texture Compression Tool*.

Mipmapping requires about 33% more memory than a non-mipmapped texture.

Note

You can generate mipmaps from uncompressed textures in OpenGL ES with a single line of code. This is an easy way to improve the performance of memory bandwidth limited applications.

2.13 Use texture compression

This section describes texture compression. It contains the following sections:

2.13.1 About texture compression

Texture compression reduces the size of textures in memory. Texture compression:

- increases performance.
- reduces memory bandwidth usage
- increases texture cacheability

The Mali GPU drivers support *Ericsson Texture Compression* (ETC1). This is widely used with OpenGL ES versions 1.1 and 2.0. ETC1 compression removes data so it is described as *lossy*. There can be a reduction in image quality compared to uncompressed textures. ETC1 texture compression provides a compression ratio of 4:1 compared to RGB565.

You can compress images with the *Mali GPU Texture Compression Tool*. The tool provides before and after compressed images so that you can compare the changes. It also provides an image that shows the difference between the compressed and uncompressed images. You can download the *Mali GPU Texture Compression Tool* from the Mali Developer Center, `http://www.malideveloper.com`.

Note

OpenGL ES 1.1 supports a compressed texture format called *paletted textures*. This format is designed to save memory and not boost performance. In some instances paletted textures can reduce performance.

2.13.2 Suitability of textures for texture compression

Texture compression works well with some content. It provides compression with no noticeable image changes in some cases, but in others the differences are very noticeable.

Texture compression works for:

- photographic content
- diffuse maps
- environment maps
- specular maps.

Texture compression does not usually work well for:

- normal maps
- fonts
- gradients between different hues.

Note

These are only guidelines. Experiment with your content to see if compression provides acceptable results.

2.13.3 Using ETC1 with transparent objects

ETC1 does not support the alpha channel in images so you cannot directly use texture compression on images with transparency.

You can use ETC1 texture compression and transparency by breaking the texture into two textures:

- A texture containing the color information.
- A single channel texture containing the transparency information. This transparency can be compressed because it is not represented as a transparent image.

These textures are compressed separately. They are recombined to create the original texture with transparency in a shader.

For more information and other methods of using texture compression with transparency, see the *Mali Developer Center*, `http://www.malideveloper.com/`.

2.14 Reduce memory bandwidth usage

This section explains why it is important to keep memory bandwidth usage low and describes methods to reduce it. It contains the following sections:

Textures use a large amount of memory bandwidth. The following sections list optimizations that you can use to reduce the memory bandwidth used for textures.

2.14.1 About reducing bandwidth

Memory bandwidth in mobile devices is very restricted compared to desktop systems. It can easily become a bottleneck limiting the performance of your application. For this reason, try to minimize your use of memory bandwidth:

- Bandwidth is a shared resource so using too much can limit the performance of the entire system in unpredictable ways. For example, graphics memory is shared with application memory so high bandwidth usage by the GPU can degrade CPU performance.
- Accessing external memory requires a lot of power so reducing bandwidth usage reduces power consumption.
- Accessing data in cache reduces power usage and can increase performance. If your application must read from memory a lot, use techniques such as mipmapping and texture compression to ensure your data is cacheable. See *Use mipmapping* on page 2-18, and see *Use texture compression* on page 2-19.

Note

Determining that memory bandwidth is causing problems is difficult. See *Diagnosing when memory bandwidth is a problem* on page 3-19.

2.14.2 Activate back face culling

Back face culling is an OpenGL ES option that you can enable. If it is enabled, the GPU removes the back facing triangles in objects so they are not drawn.

2.14.3 Utilize view frustum culling

Any object that is outside the view frustum is not visible and can be culled. Any object that is behind the view frustum can also be culled.

2.14.4 Ensure textures are not too large

Ensure your textures are not too large by using the minimum resolution and color depth required to create the effect you want.

If textures are too large they use more memory bandwidth and produce lower quality images.

Note

If you must use large textures ensure you use mipmapping and texture compression to reduce bandwidth usage. See *Use mipmapping* on page 2-18, and see *Use texture compression* on page 2-19.

2.14.5 Use a texture resolution that fits the object on screen

Use a texture resolution that is appropriate to the size of the object as it appears on screen. If the object is small on screen, you can use a small texture. For example if the object only takes up 100 by 100 pixels when it appears on screen, you do not require a texture any larger than 100 by 100 pixels.

Note

If the texture is displayed in a 1:1 pixel precise manner, image filtering or mipmaps are not necessary. Disable these to save memory and bandwidth.

2.14.6 Use low bit depth textures where possible

It is sometimes possible to use low bit depth textures with little or no visible loss of definition. Experiment with 4-bit or 8-bit per pixel textures to see if they work with your application. If these do not work, try 16-bit formats such as RGB565 or RGBA5551. Only use 32 bit RGBA8888 textures if there is no other option.

2.14.7 Use lower resolution textures if the texture does not contain sharp detail

If the texture does not contain sharp detail try a lower resolution texture.

If the texture contains a small section of high detail you might be able to use a standard resolution texture for the high detail part and a low resolution texture for the rest.

2.14.8 Textures and lighting maps do not have to be the same size

Different textures are used for different purposes. These are not required to be the same resolution even if they are on the same object. Consider each texture individually and use the lowest resolution and bit depth required to get the effect you want.

2.14.9 Consider if tri-linear filtering is necessary for every object

Trilinear filtering is useful for large surfaces that extend away from the viewer. For example floors, walls and airport runways.

Trilinear filtering can have a significant bandwidth cost, so avoid it on objects where it makes little visual difference.

On a small screen trilinear filtering is likely to have very little visual impact. Consider if it is worth the extra computations and bandwidth.

Note

If your application is bandwidth bound, disabling tri-linear filtering is a useful performance-quality trade-off.

2.14.10 Utilize dynamic level of detail

Dynamic Level of Detail (LOD) is a family of techniques that lower the resolution of geometry and textures for objects as they move away from the camera. These techniques reduce bandwidth usage.

For more information on LOD, see *Use dynamic level of detail* on page 4-8.

2.15 Use Vertex Buffer Objects

A *Vertex Buffer Object* (VBO) is a data storage mechanism that enables an application to store and manipulate data in GPU memory. VBOs provide a large reduction in overhead so can provide a considerable performance increase.

If you send data to the GPU every frame it is copied whether it has changed or not. Using VBOs avoids these copies because storing data in the GPU memory means no copies are required.

2.16 Ensure your application is not CPU bound

This section describes how to determine if your application is CPU bound and how to optimize application code. It contains the following sections:

- *Determining if your application is CPU bound*
- *Optimize application logic*
- *Use loop optimizations* on page 2-26
- *Align data* on page 2-26.

If an application is CPU bound, making graphics optimizations has no impact on performance. Determine if your application is CPU bound and make the relevant optimizations before moving onto graphics optimizations.

It is relatively rare for an application processing to be the slowest component but it can and does happen. This is especially true if the application originated on a desktop platform and you are moving it to a mobile platform.

2.16.1 Determining if your application is CPU bound

An approximate way to determine if your application is CPU bound is to change the graphics output to a very low resolution.

At a very low resolution the GPU does relatively little work. If the frame rate does not change there is a good probability the bottleneck is in the CPU.

The application can be CPU bound in the following areas:

The application logic uses too much processing power

To determine if your application is CPU bound in the application logic, remove the draw calls and swapbuffers commands by making them into comments, or use a stub driver.

If there is no change or only a small change in performance the limitation is probably the application logic.

Use a profiler to determine what areas of the application logic are causing the issue and optimize this code.

The application is overloading the driver

If the low resolution output test indicates that the bottleneck is in the CPU but the application logic is not the problem, then the application might be using the OpenGL ES API in an sub-optimal way. See *Minimize draw calls* on page 4-2, *Do not compile shaders every frame* on page 2-12, and *Use Vertex Buffer Objects* on page 2-24.

A combination of application logic and driver

Sometimes the application is not bound in one area, but it is the combination of both that causes the application to be CPU bound.

Note

These are only approximate methods for determining if the application is CPU bound and might not always be reliable. To determine the source of problems with more accuracy you must do a full optimization process.

2.16.2 Optimize application logic

There are many reasons application logic might not perform well and many ways to optimize it. See *Application optimizations* on page 4-15.

2.16.3 Use loop optimizations

Software loops are used extensively to process graphics and application data. Applications typically spend a large portion of running time in loops so they are a good target for optimizing your code. See *Optimize loops* on page 4-9.

2.16.4 Align data

The OpenGL ES standard requires data to be copied from the application to the driver. If you align your data to eight bytes it is more cacheable so the copies are faster.

You must also align data before the Mali GPU can use it. If you enforce alignment in code the Mali GPU driver does not have to align the data during the copy. This improves performance because there is no overhead aligning the data.

Note

- Ensure you align data when you import data from header files. This might require a compiler-specific command.
- Aligned data is more cacheable on GPUs and CPUs so it is a good idea to always align data to 8 bytes if possible.

2.17 Check system settings

It is critical for application performance that your system is set up correctly. Even well optimized applications run badly if your system settings are incorrect.

Incorrect system settings are a common error so ensure you check them. If you cannot change the settings, inform the vendor of your system.

Check the caches are switched on

Modern systems all use cache to boost performance. If the caches in the system are not switched on there is a large performance reduction.

Check the CPU and GPU clock settings are correct

No application can run at maximum performance if the clock settings for the CPU or GPU are not correct. Alternatively, if the clocks for the CPU or GPU are set too high the system is likely to use too much power.

Check the CPU and GPU are in full power mode

CPU and GPUs all have low speed, low power modes that save power when the processors are not in use. For high performance applications, ensure the processors are in full power mode for maximum possible performance.

Ensure the GPU clock is not scaled according to CPU load

If the GPU clock is scaled according the CPU load the performance of applications are likely to suffer. This is because the CPU and GPU might be busy at different times. If the GPU is busy when the CPU is not, lowering the clock of the GPU reduces performance.

Control the clock of the CPU and GPU independently to fix this problem.

2.18 Final release check list

Table 2-1 provides a list of items you can quickly check before releasing an application.

Table 2-1 Final release check list

Check	Additional Information
Are caches enabled?	See *Check system settings* on page 2-27.
Did you switch off debugging?	See *Remove debugging information* on page 2-9
Have you removed pipeline stalling calls?	See *Avoid calls that stall the graphics pipeline* on page 2-11
Is back face culling enabled?	See *Activate back face culling* on page 2-21 and *Avoid overdraw* on page 4-5
Is mipmapping enabled?	See *Use mipmapping* on page 2-18
Are you using compressed textures?	See *Use texture compression* on page 2-19
Is VSYNC enabled?	See *Use VSYNC* on page 2-13
Did you use the latest tools?	See *Use the correct tools with the correct settings* on page 2-8
Are your tools configured correctly?	
Did you build an optimized binary	

Chapter 3
The Optimization Process

Optimization involves taking performance measurements, identifying bottlenecks, and applying appropriate techniques to remove them. This chapter describes this process. It contains the following sections:

3.1 About the optimization process

The optimization process involves identifying bottlenecks in applications, then using techniques to remove them.

There are a number of steps in the optimization process:

- take performance readings from your application
- analyze the readings to locate the bottleneck
- identify the types of optimization that are appropriate
- select and apply an optimization
- take performance readings to ensure the optimization works.

The steps are shown in Figure 3-1.

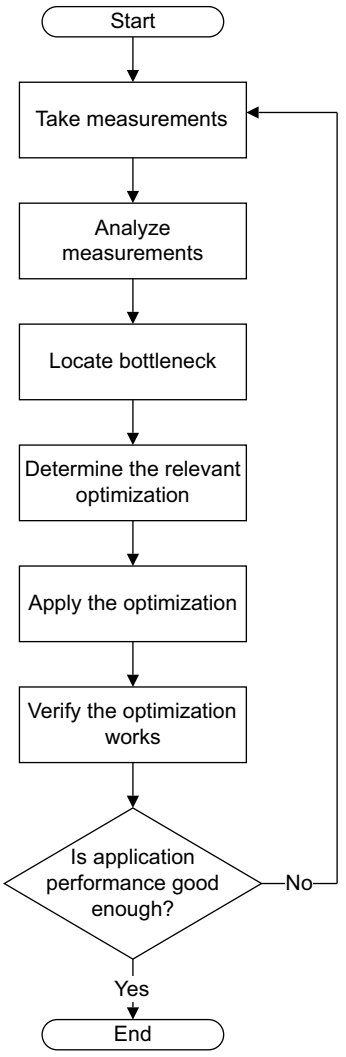

Figure 3-1 Optimization process steps

Note

The optimization process is likely to reveal a series of different bottlenecks, so you might have to go through the process a number of times to remove them all. See *Bottlenecks move between processors* on page 3-6.

3.2 General optimization advice

This section contains general optimization advice. It contains the following sections:

3.2.1 The general principle of try it and see

Different applications can react to optimizations in very different ways. In one application a specific optimization might have a large impact on performance, whereas in another applications it might have a small impact.

If you are optimizing, do not assume all optimizations are always going to increase performance. There are often trade-offs between optimizations, so try out different techniques to see what works best for your application.

In general, experiment with different approaches for all optimizations and graphics programming.

3.2.2 Use frame time instead of FPS for measurements

Frames per second is a simple and basic measurement of performance, but frame time is a better measure of optimization effectiveness.

The reason frame time is a better measurement is that frame time is a linear measure whereas frames per second is non-linear. Linear measurements also make calculations easier.

Figure 3-2 shows frames per second plotted against frame time. This graph illustrates the non-linear nature of FPS measurements.

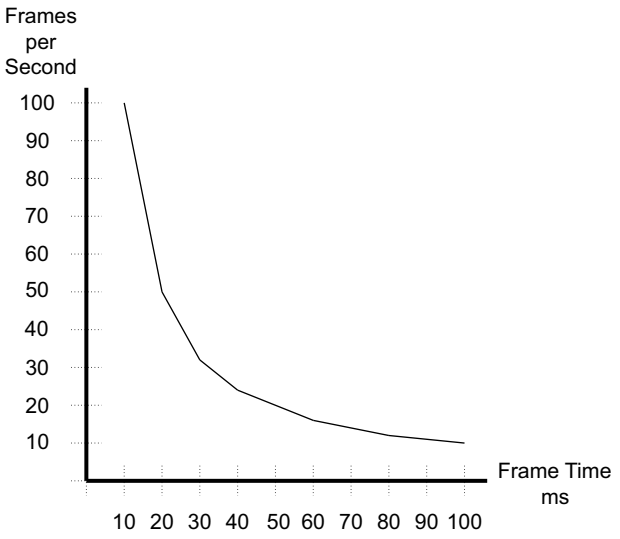

Figure 3-2 Frame time and FPS

If you know the individual time changes corresponding to different optimizations, you can add the times together to get the total improvement.

If you are using FPS as a measurement, you cannot add them together because their non-linear nature. Any attempt to add them gives an incorrect total.

Table 3-1 shows a series of comparisons between different FPS measurements A and B.

The FPS changes by a different amount for every measurement, but the frame time changes by the same amount every time. For example, going from 100 FPS to 200 FPS involves a difference of 100 FPS or 5ms. However going from 20 FPS to 22.2 FPS is a difference of 2.2 FPS but this is also 5ms. The linear nature of frame time is easier to work with when you are measuring the impact of optimizations.

Table 3-1 Difference between frames per second and frame time

Frame time A	FPS A	Frame time B	FPS A	FPS difference	Frame time difference
70ms	14.3	65ms	15.4	1.1	5ms
60ms	16.7	55ms	18.2	1.5	5ms
50ms	20	45ms	22.2	2.2	5ms
40ms	25	35ms	28.6	3.6	5ms
30ms	33.3	25ms	40	6.7	5ms
25ms	40	20ms	50	10	5ms

Table 3-1 Difference between frames per second and frame time (continued)

Frame time A	FPS A	Frame time B	FPS A	FPS difference	Frame time difference
20ms	50	15ms	66.6	16.6	5ms
15ms	66.6	10ms	100	33.4	5ms
10ms	100	5ms	200	100	5ms

3.2.3 Set a computation budget and measure against it

It is useful to set a computation budget that you can measure against. There are maximum performance limits in processors that you cannot exceed. If you compare the computations your application is doing against the maximum values, you can see if your application is trying to do too much.

The exact resources available depend on different factors such as:

- the type of GPU in your platform
- the configuration of the GPU
- color depth
- image resolution
- the required frame rate.

You can set a budget for:

- pixel processor cycles
- geometry processor cycles
- CPU cycles
- memory bandwidth.

3.2.4 Bottlenecks move between processors

This section describes how bottlenecks move between processors and the profile of an ideal application. It contains the following sections:

- *How bottlenecks move between processors*
- *Ideal application profile* on page 3-7.

How bottlenecks move between processors

The performance bottleneck in an application can move between the different processors as optimizations are applied. The readings from Performance Analysis Tool can tell you where the bottleneck is likely to move to and if a processor is under-used.

Figure 3-3 shows a bar graph of frame rates for different parts of a system running an application.

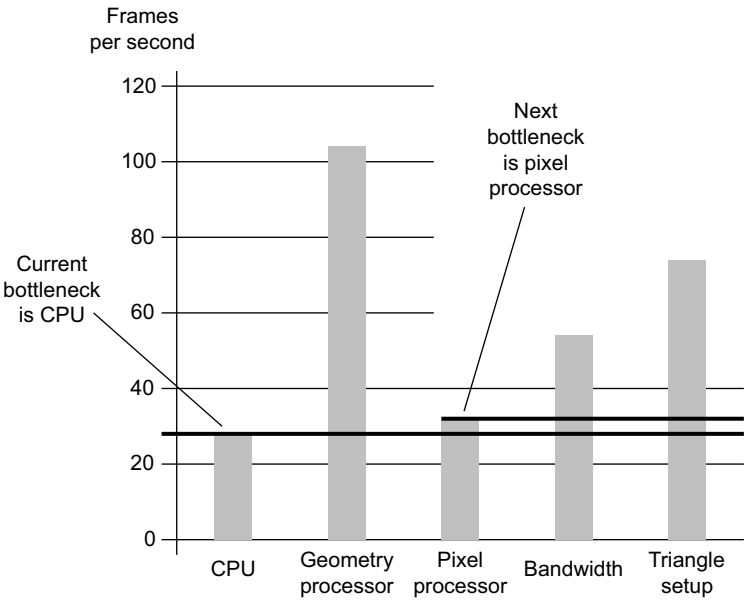

Figure 3-3 Frame rate limitations of different system elements

Comparing the bars indicates the following:

- If you optimize the performance bottleneck, the next bottleneck is the component with second lowest bar. In this case the bottleneck is the CPU and the next bottleneck is the pixel processor.
- These graph of the geometry processor has a much higher value than the overall GPU Frame Rate. This indicates that the bottleneck is not in the geometry processor. The large difference also indicates it is under-used.

  If a processor has spare processing capacity, consider if there are any processing operations that you can move to it. For example, you might be able to move operations from the CPU or pixel processor to the geometry processor.

Ideal application profile

An ideal application is limited approximately equally by all elements. A bar graph such as Figure 3-4 indicates the application is making good use of all elements.

In this case a single optimization is not likely to make a large impact on performance and you require multiple optimizations give a higher and more stable frame rate.

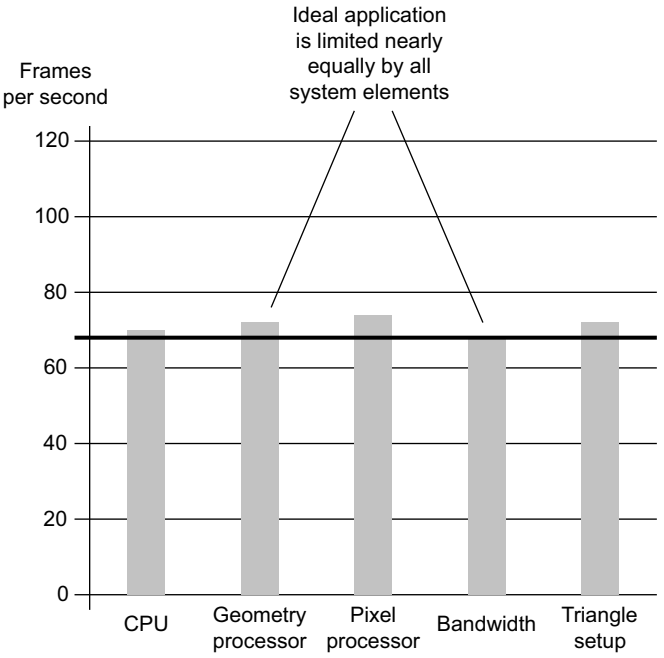

Figure 3-4 Ideal application equally limited

3.3 The optimization process steps

This section describes the steps in the optimization process. It contains the following sections:

- *Take measurements*
- *Locate the bottleneck*
- *Determine the optimization*
- *Apply the optimization* on page 3-9
- *Verify the optimization* on page 3-9
- *Repeat the optimization process* on page 3-9

3.3.1 Take measurements

To optimize, take measurements from your application and determine the problem areas.

You can take measurements in the following ways:

- Use the Mali instrumented drivers to take readings from counters and record data about the application while it is running. This data is saved to files you can view later.
- Use a selection of techniques and tools to gather performance information.

For more information, see *Taking measurements with the instrumented drivers* on page 3-10 and see *Locating bottlenecks with other tools* on page 3-16.

Note

You must switch VSYNC off when optimizing and especially when you are taking measurements. See *Use VSYNC* on page 2-13.

3.3.2 Locate the bottleneck

To locate a bottleneck that is reducing performance, you must analyze your measurements.

Use the Performance Analysis Tool to analyze the data files produced by the instrumented drivers.

The Performance Analysis Tool can show the counter data as easy to interpret graphs. You can also customise the display to show graphs of different counters against each other. This is a very useful feature for locating bottlenecks.

When you have located the bottleneck you can:

- Take additional measurements to isolate the exact problem area. Some bottlenecks are difficult to find so might require more investigation
- Apply one or more optimizations.

For more information, see *Locating bottlenecks with the Performance Analysis Tool* on page 3-10 and *Locating bottlenecks with other tools* on page 3-16.

3.3.3 Determine the optimization

The optimization to apply depends on the bottleneck. You might not find the exact cause in the application, but you can find out where it has the greatest impact. Typically, this is one of:

- pixel processor
- geometry processor
- CPU

- triangle setup
- API issues such as blocking calls
- memory bandwidth.

For more information, see *Determining the relevant optimization* on page 3-20.

3.3.4 Apply the optimization

Applying the optimization might involve modifying application code and art assets. You can download tools to assist you with some parts of this process this from Mali Developer Center, `http://www.malideveloper.com`.

3.3.5 Verify the optimization

Attempts at optimization might not always work as well as required. You must verify that the optimization works correctly. To verify the optimization, run the application again with the optimization applied.

It is possible for an optimization to work but have very little effect on application performance. This can mean the following:

- There are other bottlenecks in the application limiting performance.
- The measurements were misleading and the wrong optimization was applied. This can happen if the real bottleneck is difficult to measure.

If there is only a small difference to frame time, consider taking more measurements and analyzing them.

3.3.6 Repeat the optimization process

An optimization process can reveal a series of different bottlenecks. You might have to go through the process a number of times to remove all of them and get performance up to the required level.

As you repeat the optimization process you are likely to find new bottlenecks. Bottlenecks can move between the geometry processor, the pixel processors, and the CPU. For more information, see *Bottlenecks move between processors* on page 3-6.

3.4 Locating bottlenecks with the Performance Analysis Tool

This section describes using the Performance Analysis Tool to locate bottlenecks. It contains the following sections:

3.4.1 Taking measurements with the instrumented drivers

The instrumented drivers are a special version of the Mali GPU drivers. These drivers include additional functionality that enables them to record and store performance data. This includes data about how your application is using the OpenGL ES API and the Mali GPU hardware.

The Performance Analysis Tool can analyze the data produced by the instrumented drivers to identify bottlenecks for optimization. See *Locating bottlenecks with the Performance Analysis Tool*.

The instrumented drivers save the following types of files:

.ds2 `.ds2` performance data files contain the following information:
- a description of the performance counters
- numerical performance counter data
- images from the framebuffer.

.log `.log` files list the OpenGL ES calls your applications makes when it is running. You can analyze these to see how many call of what type your application makes. You can use this information to determine if the application makes too many calls or makes intensive calls too often.

3.4.2 About the Performance Analysis Tool

The Mali GPU Performance Analysis Tool is a tool that provides you with information about how well your application performs.

When you run your application on a Mali GPU with instrumented drivers, data is gathered and saved to `.ds2` performance data files.

You can use the Performance Analysis Tool to:

- capture counter data from the instrumented driver
- save captured data for replay
- view the contents of the framebuffer
- display the values of individual performance counters in graphs and tables
- observe how the values change over time
- assess the performance of each frame.

You can customise the Performance Analysis Tool so you can view the counter values as graphs and compare them against each other. This helps you determine what factors are dominating performance and where performance bottlenecks are likely to be. You can also add virtual counters that show relationships between values.

See *Mali GPU Performance Analysis Tool User Guide*.

3.4.3 GPU counters

The Performance Analysis Tool can capture data from the hardware counters in the GPU and display the results as graphs.

Looking at the most important counters first gives you an idea of where performance bottlenecks are likely to be found. Table 3-2 shows the most important counters to consider in the Mali-200 GPU.

Table 3-2 The most important counters to consider in the Mali-200 GPU

Counter	Location in the Performance Analysis Tool	Notes
Overall frame rate	-	From your application
Geometry Processor Frame rate	`MaliGP2/Performance/Frame rate`	-
Pixel Processor Frame rate	`Mali200/Performance/Frame rate`	-
Fragments rasterized	`Mali200/HW counters/Fragment rasterized count`	-
Shader instructions	`Mali200/HW counters/Instructions completed count`	Shader instructions executed
Texture Bandwidth	`Mali200/Bandwidth/Texture BW`	-
Texture cache hit count	`Mali200/HW counters/Texture cache hit count`	-
Texture cache miss count	`Mali200/HW counters/Texture cache miss count`	-
OpenGL ES draw calls	`GLES Profiling/glDrawElements` `GLES Profiling/glDrawArrays`	-

Counters for the Mali-400 MP are named slightly differently. Pixel processor counters are listed per core. Table 3-3 shows the first counters to consider in the Mali-400 MP GPU.

Table 3-3 The most important counters to consider in the Mali-400 MP GPU

Counter	Location in the Performance Analysis Tool	Notes
Overall frame rate	-	From your application
Geometry Processor Frame rate	`Mali-400 GP/Performance/Frame rate`	Per GPU
Pixel Processor Frame rate	`Mali-400 PP/Performance/Frame rate`	Per core
Fragments rasterized	`Mali-400 PP/HW counters/Core <number>/Fragment rasterized count`	Per core
Shader instructions	`Mali-400 PP/HW counters/Core <number>/Instructions completed count`	Shader instructions executed per core
Texture Bandwidth	`Mali-400 PP/Bandwidth/Texture BW`	Per GPU
Texture cache hit count	`Mali-400 PP/HW counters/Core <number>/Texture cache hit count`	Per core
Texture cache miss count	`Mali-400 PP/HW counters/Core <number>/Texture cache miss count`	Per core
OpenGL ES draw calls	`GLES Profiling/glDrawElements` `GLES Profiling/glDrawArrays`	-

3.4.4 Analyzing graphs

The Performance Analysis Tool enables you to view counters as graphs. You can determine problem areas by comparing graphs against each another.

1. In the Performance Analysis Tool, display the following counters as graphs:
   - **Geometry Processor Frame Rate**
   - **Pixel Processor Frame Rate**
2. Compare the overall GPU Frame Rate to the graphs. You can make a number of comparisons to locate performance bottlenecks:
   a. Look for a processor that has a frame rate number similar to the overall GPU Frame Rate. This means that processor might be the limiting factor. Consider optimizing for this processor first.
   b. If no processor has a similar frame rate, look for the lowest frame rate. This indicates the processor with the lowest performance. Optimize for this processor, optimizing for other processors has no impact on performance.
   c. If there is no under-performing processor look for the graph with the most similar shape to the overall frame rate graph. This is likely to be the dominant performance factor.
   d. If the overall GPU frame rate and other frame rate counters show no correlation, investigate the CPU.
   e. If it is difficult to find a correlation, investigate memory bandwidth. Bandwidth limitations can impact all results in unpredictable ways.

Graphs can be volatile. The relative performance of the different processors can change from one frame to another:

- look for the average rate over time to find where to make general performance improvements
- look at performance on a frame by frame basis to find areas in specific scenes to optimize.

Note

- A graph can indicate the general problem area. You might want to take more measurements to isolate the exact problem. See *Finding exact problem areas* on page 3-18.
- Also ensure that you have VSYNC off when taking measurements because this can create incorrect measurements.

3.4.5 Specific problems areas to look at

The following are typical problems areas:

Pixel Processor bound problems:

- high texture cache miss rate
- overdraw.

Pixel Shader bound problems:

- shader instruction high cache miss rate
- shader too slow
- too many branches.

Geometry Processor bound problems:

- high shader time
- high PLBU time

Note

It is unusual for the geometry processor to be the bottleneck in real applications.

CPU bound problems:

- too many draw calls.
- too many state changes
- slow application logic.

Note

CPU profilers such as OProfile can often distinguish between application and driver overhead.

API problems:

- pipeline stalls
- driver too busy.

3.4.6 Additional counters to examine

The following counters can indicate performance problems:

Cache counters

There are counters for the different caches in the GPU including:

- Varying cache conflict miss count
- Palette cache miss count
- Load/store cache miss count
- Texture cache conflict miss count
- Varying cache miss count
- Vertex cache miss count
- Texture cache miss count
- Program cache miss count.

OpenGL ES and EGL API counters

- glDrawElements
- glDrawArrays
- Draw call statistics
- Geometry statistics.

EGL

- Timing
- Blitting.

3.5 Locating bottlenecks with other tools

This section describes other sources of information. It contains the following sections:

- *Taking measurements without the Performance Analysis Tool*
- *Measurements from other Mali GPU tools*
- *Information from debugging tools* on page 3-17.

3.5.1 Taking measurements without the Performance Analysis Tool

If you do not have access to instrumented drivers for a Mali GPU, it is difficult to take exact measurements and you cannot analyze them with the Performance Analysis Tool. It is however still possible to get useful information provided that you have a frame time or frame rate counter in your application.

Note

If you do not have access to instrumented drivers contact your device vendor. They might be able to provide instrumented drivers for the Mali GPU.

To take measurements:

1. Activate frame time measurement in your application.
2. Run a representative, exactly repeatable sequence, in your application.
3. Modify the application.
4. Run the same sequence in the application and take measurements.
5. Repeat steps 3 and 4 with different modifications.

By making changes in your application then re-running an identical sequence, you obtain a series of measurements of different areas of your application.

Note

To determine the type of changes to make, see *Areas to investigate* on page 3-18.

The frame time in the measurements is likely to remain similar in some cases but different in others.

- if the frame time changes very little, the application is not likely to be bound in that area
- if the frame rate changes dramatically, the application is most likely bound in that area.

3.5.2 Measurements from other Mali GPU tools

You can use a number of Mali GPU software tools to obtain measurements and other useful information.

For example, use the -v option with the *Mali GPU offline shader compiler* gives you information about your vertex and fragment shaders.

Executing `malisc -v my_shader.frag` produces output that shows the minimum and maximum number of cycles that `my_shader.frag` takes to execute, assuming no cache misses.

You can use these figures to work out the maximum number of vertices or fragments that can be shaded per second.

The *Mali GPU offline shader compiler* and other tools are available from the Mali Developer Center, `http://www.malideveloper.com`.

3.5.3 Information from debugging tools

You can obtain about CPU utilisation from profiling tools such as OProfile. Profiling tools can tell you about the behavior of your application and distinguish between application CPU usage and driver CPU usage.

Static code analysis tools can identify if code is complex. Complex code is more likely to be slow and prone to errors.

3.6 Finding exact problem areas

This section describes how to exactly locate the code or asset causing a performance problem. It contains the following sections;

- *Technique for locating the exact problem areas*
- *Areas to investigate*
- *Diagnosing when memory bandwidth is a problem* on page 3-19.

Note

This section describes a drill-down technique and what to investigate. You do not require the Performance Analysis Tool to use this technique.

3.6.1 Technique for locating the exact problem areas

You can find exact problem areas by replacing code or assets types with versions that use no compute or memory bandwidth resources. A large performance difference indicates a problem area.

The following procedure explains how to find problem shaders:

1. Replace all shaders with null shaders and measure the performance difference. If there is not much performance difference then the problem is not a shader. If there is a big difference then there is a problem with shaders.
2. Divide the shaders into two halves, A and B, and test again.
   a. test with the A half as null shaders and the B half as the original shaders
   b. test with the A half as the original shaders and the B half as null shaders
   c. compare the results.
3. Determine the half with the largest performance impact. Divide this half into two and repeat step 2. Continue this process until you have located the problem shader.
4. Optimize the shader and measure performance again with all the original shaders present. If there is still a problem there might be other problems shaders. Continue repeating the process until you have optimized all the problem shaders.

Note

It is important to be systematic in this process. Investigate one type of code or asset at a time. If you try investigating more than one type at a time your measurements are likely to be inaccurate.

3.6.2 Areas to investigate

There are a number of areas you can investigate with the technique described in *Technique for locating the exact problem areas*. You can follow the technique exactly by disabling code and assets. Alternatively you can try using simpler versions.

Change resolution

Change the resolution of your application and measure the frame rate difference.

If the frame rate scales with the inverse of the resolution, that is the performance halves when you double the resolution, your application is pixel processor bound.

Change texture size

Change the size of your textures to 1 by 1. If the frame rate increases, the texture cache hit rate is too low.

Use a stub driver

A stub driver replaces the OpenGL ES driver with a driver that does nothing. The frame rate produced when using a stub driver indicates the CPU usage. If the frame rate does not change from using the normal driver your application is CPU bound.

Reduce shader length

If you shader is too long it can reduce your frame rate. Try shorter shaders and measure the changes.

If the frame rate increases the shader length might be trying to do too much, or is too long.

Note

The number of cycles the Mali pixel processor spends shading is proportionate to the frame size and frame rate. If you double the frame size the number of cycles available halves.

Use an empty fragment shader

An empty or null shader indicates if you are shader bound.

A null shader does no work. Replace your shaders with null shaders and measure performance. If performance rises sharply your application is likely shader bound.

Change number of vertices

A large change when using simpler models indicates your application might be using too many vertices or is bandwidth bound.

Change the bit depth of textures

If application performance rises when you reduce the bit depth of textures, memory bandwidth might be a problem.

Change the bit depth of the drawing surface

If application performance rises with a lowering of texture bit depth memory bandwidth might be a problem.

Reduce draw calls

Using too many draw calls is a common problem. Moving the same amount of work into a smaller number of draw calls might indicate this is a problem area.

Reduce state changes

Reducing the number of state changes might indicate this is a problem area.

3.6.3 Diagnosing when memory bandwidth is a problem

Memory bandwidth impacts everything so it is difficult to diagnose if this is the problem.

Memory bandwidth is difficult to measure without dedicated tools such as the *Performance Analysis Tool*. It is therefore difficult to confirm bandwidth usage is a bottleneck.

Bandwidth overuse can appear to be other limitations in processors. If one of the processors is limiting performance and optimizations do not appear to be having any effect it might be bandwidth causing the problem.

3.7 Determining the relevant optimization

This section lists the optimizations and the areas of a system they impact. It contains the following sections:

- *Optimization lists*
- *Miscellaneous optimizations* on page 3-21.

3.7.1 Optimization lists

When you have determined the cause of the problem you must apply an optimization to fix it. This section lists the optimizations described in this guide and indicates what parts of the system they apply to. Use the lists to determine what optimizations to use. Table 3-4 shows the optimizations listed in this guide.

Some optimizations impact multiple parts of the system, these are listed in every area they impact.

Table 3-4 Optimizations described in this guide

Optimization	Geometry Processor	Pixel Processor	CPU	Triangle setup	API	Memory Bandwidth
Optimizations listed in Chapter 2 *Optimization Checklist*						
Check the display settings on page 2-4	-	-	Y	-	-	Y
Use direct rendering if possible on page 2-7	-	-	Y	-	-	Y
Use the correct tools with the correct settings on page 2-8	-	-	Y	-	-	-
Remove debugging information on page 2-9	-	-	Y	-	-	-
Avoid infinite command lists on page 2-10	-	-	-	-	Y	-
Avoid calls that stall the graphics pipeline on page 2-11	-	-	-	-	Y	-
Do not compile shaders every frame on page 2-12	-	-	Y	-	-	-
Use VSYNC on page 2-13	Y	Y	Y	Y	Y	Y
Use graphics assets appropriate for your platform on page 2-16	Y	Y	-	-	-	Y
Do not use 24-bit textures on page 2-17	-	Y	-	-	-	Y
Use mipmapping on page 2-18	-	Y	-	-	-	Y
Use texture compression on page 2-19	-	Y	-	-	-	Y
Reduce memory bandwidth usage on page 2-21	-	-	-	-	-	Y
Use Vertex Buffer Objects on page 2-24	-	-	Y	-	Y	Y
Ensure your application is not CPU bound on page 2-25	-	-	Y	-	-	-
Check system settings on page 2-27	Y	Y	Y	Y	-	Y
Optimizations listed in Chapter 4 *Optimization Techniques*						
Minimize draw calls on page 4-2	-	-	Y	-	Y	-
Minimize state changes on page 4-4	-	-	Y	-	Y	-

Table 3-4 Optimizations described in this guide (continued)

Optimization	Geometry Processor	Pixel Processor	CPU	Triangle setup	API	Memory Bandwidth
Avoid overdraw on page 4-5	-	Y	-	-	-	Y
Use approximations to improve performance on page 4-7	Y	Y	Y	Y	-	Y
Use dynamic level of detail on page 4-8	Y	Y	-	Y	-	Y
Optimize loops on page 4-9	-	Y	Y	-	-	-
Use vector instructions on page 4-12	-	Y	Y	-	-	-
Ensure the graphics pipeline is kept running on page 4-14	-	-	-	-	Y	-
Application optimizations on page 4-15	-	-	Y	-	-	-

3.7.2 Miscellaneous optimizations

You can sometimes optimize by moving computations to under-used resources. This is not a direct answer to a performance problem, but it is a useful optimization technique.

See *Make use of under-used resources* on page 4-13 and see *Bottlenecks move between processors* on page 3-6.

Chapter 4
Optimization Techniques

This chapter describes a number of optimization techniques that are more complex than those listed in Chapter 2 *Optimization Checklist*. It contains the following sections:

4.1 Minimize draw calls

This section describes minimizing draw calls. It contains the following sections:

- *About minimizing draw calls*
- *Limitations on combined draw calls*
- *Combining textures in a texture atlas*.

4.1.1 About minimizing draw calls

Every OpenGL ES API function call imposes an overhead. It is easy to limit the performance of your application by making too many function calls.

You can reduce the overheads imposed by grouping uniforms and draw calls together and passing them in a smaller number of API calls.

OpenGL ES draw calls are calls to the following functions:

- `glDrawArrays()`
- `glDrawElements()`.

Draw calls impose CPU processing overhead. Your application can easily become CPU-bound if it makes too many of them.

The overheads are because of the operations that the functions have to do. The processing required by draw calls include allocating memory, copying data, and processing data. The overhead is the same whether you draw a single triangle or thousands of triangles in a draw call.

If you combine multiple triangles into a single draw call the overhead is only imposed once rather than multiple times. This reduces the total overhead and increases the performance of your application.

You can combine triangles in a single draw call by using the following:

- triangle strips
- triangle lists
- index arrays.

You can additionally combine triangles by making combinations of combinations. For example, you can combine multiple triangle strips together. Adding degenerate triangles between the strips can assist this process.

4.1.2 Limitations on combined draw calls

There are limitations on combined draw calls. The combined triangles must:

- be in the same format
- use the same shader
- use the same GL state.

If you cannot join triangles because they use different shaders, consider if the effect generated by the shader makes a significant difference. If the difference is small, it might be worth removing the effect.

4.1.3 Combining textures in a texture atlas

To assist with the process of combining triangles, you can combine multiple textures together into a texture atlas. This is a single texture image that contains textures from components of different objects.

If you are drawing text on screen you use the same technique. Combine all the drawing calls into a single call and use a *font atlas* to provide the font images. This is an efficient technique because it enables you to draw all the text in a single draw call.

Note

Texture atlas based techniques typically do not work with repeating textures.

4.2 Minimize state changes

State changes are similar to draw calls in that there is an overhead imposed every time state is changed. To reduce this overhead, minimize the number of state changes your application makes.

You can reduce state changes by:

- Grouping together triangles or objects with the same texture.
- Using texture atlases. This enables you to draw complex single objects or multiple objects with a single texture. See *Combining textures in a texture atlas* on page 4-2.

4.3 Avoid overdraw

Overdraw occurs when graphical objects are drawn over the top of one another. The GPU uses both the compute resources and memory bandwidth required for fragments that cannot be seen. This is a waste of resources that negatively impacts the performance of applications. Avoid overdraw if possible.

You can use a number of methods to do this:

- *Use culling*
- *Sort objects and draw in front to back order*
- *Optimize the use of transparency* on page 4-6.

4.3.1 Use culling

Culling is the process of working out what parts of a scene are not visible and removing them so they are not drawn. There are a number of methods of culling. These range from very coarse types of culling performed by the application to very fine culling of individual triangles.

It is best to cull entire objects in the application and let the GPU cull individual triangles. Triangle culling is a highly intensive operation and if you use the CPU to do this it is likely to reduce the performance of your application.

Work out what parts of the world are visible

In applications such as mapping or games, only certain parts of the game world are visible at any given time. You can reduce GPU work by working out what is visible at different locations in the game before runtime. At runtime the application uses this information to determine what objects are visible. You can use techniques such as view frustum culling to do this.

Enable depth testing

Back face culling is an OpenGL ES option. Depth testing discards fragments that are behind already drawn fragments. This reduces the amount of computation required by the GPU.

Enable back face culling

Back face culling is an OpenGL ES option. When it is enabled the GPU removes the back facing triangles in objects so they are not drawn.

Calculate bounding boxes

You can reduce the number of runtime computations required for culling by using bounding boxes or spheres to contain complex objects. Your application uses these to calculate what objects might be visible and only sends information for the visible objects to the GPU.

Use view frustum culling

Any object that is outside the view frustum is not visible so can be culled. Any object that is behind the camera plane can also be culled.

4.3.2 Sort objects and draw in front to back order

Sort the objects in the scene by distance from the camera, then draw the objects in the frame in front to back order. If the objects are not in this order, the GPU can draw the same pixel multiple times until it draws the front most object. Sorting prevents this from happening.

You can save time sorting by keeping geometry in data structures such as *Binary Space Partition* (BSP) trees.

4.3.3 Optimize the use of transparency

Transparency is a useful effect but it can be resource intensive if there are many layers.

If you are using transparent objects you can reduce their drawing cost by:

- Ensuring transparent objects are drawn last. If a transparent object is not drawn last your application might have to waste resources drawing the object more than once.
- Minimizing the number of transparent layers visible through each other.
- Ensuring they occupy a small area on the screen.

4.4 Use approximations to improve performance

This section describes using approximations to improve performance. It contains the following sections:

- *General methods of approximation*
- *Specific methods of approximation*.

Many desktop applications use special effects to create high quality images. High quality effects require a lot of computations.

One method of optimizing is to use approximations in the graphics. This involves using faster techniques that create a similar, but not identical effect.

4.4.1 General methods of approximation

There are a number of general methods of approximation that you can use:

Use compute power where it has the greatest visual impact

Some effects are subtle and might be hardly visible on a mobile device. Make the best use of the available resources by using compute power on effects that have the most visible impact.

Simplify effects

You can optimize by removing or simplifying elaborate effects. Try changing complex effects to simpler effects that give a similar result.

Graphics are rarely required to be correct

Graphics are typically not required to be precise. You might be able to use more approximate, simpler computations that reduce correctness to achieve increased performance.

Simplify equations

Some shaders use complex equations. Try to use simpler, less compute intensive equations to achieve a similar effect.

Consider different algorithms

You can get large performance increases by changing to more efficient algorithms.

4.4.2 Specific methods of approximation

Shadows You can use projection shadows to generate soft shadows.

Lighting Save computations by using fewer lights or by reducing the distance they are visible from.

Blur For blur effects, take a low resolution texture and blur it. This is faster and uses less memory bandwidth than blurring a higher resolution texture.

Reflection There are graphics techniques that generate reflections with relatively little cost but impose limitations. Consider if you can work within the limitations and so use these techniques.

Glow Create a series of transparent white triangles around the object. Fade from fully opaque beside the object to fully transparent at the edge.

4.5 Use dynamic level of detail

Dynamic level of Detail (LOD) is a set of techniques that involve using different versions of an object with greater or less detail, depending on the distance from the camera.

A full resolution, fully detailed object is used when it is close to the camera. When the object is further away a lower resolution less detailed object is used. The same goes for textures. Textures are high resolution close up but lower resolution far away.

Applying dynamic level of detail to objects reduces the usage of both compute power and memory bandwidth.

You can choose the LOD technique for different distances from the camera, Use the lowest level of detail that is appropriate but try lower levels to see if it makes much difference. Using less details requires less memory and bandwidth.

4.6 Optimize loops

Loops are used for intensive computations in both application code and shaders. They take up a large amount of processing time. You can make application and shader code faster by optimizing loops.

Generally, the key to loop optimization is to make the loop do as little as possible per iteration to make the iterations faster. If the loop runs ten thousand times, reducing the loop by one instruction reduces the number of instructions executed by ten thousand.

Move repeated computations

If there are computations in a loop that can be done before the loop, move these computations to outside the loop.

Look for instructions that compute the same value over and over again. You can move this computation out of the loop.

Move unused computations

If there are computation in the loop that generate results that are not used within the loop, move these computations to outside the loop.

Avoid computations in the iteration if test

Every time a loop is iterated a conditional test determines if another iteration is required. Make this computation as simple as possible. Consider if the entire computation must be performed each time. If possible move any pre-computable parts outside the loop.

Simplify code

Avoid complex code constructs. It is easier for the compiler to optimize if your code is simpler.

Avoid cross-iteration dependencies

Try to keep the computations in iterations independent of other iterations. This makes a number of optimizations possible in the compiler.

Work on small blocks of data

Ensure the inner loops work on small blocks of data. Smaller blocks of data are more cacheable.

Minimize code size

Keeping loops and especially inner loops small makes them more cacheable. This can have a significant impact on performance. Reducing the size of loops, especially inner loops make the instructions more likely to be cached increasing performance.

Unroll loops

You can take the computations from many loop iterations and make them into a single big iteration. This saves if tests computations so it reduces the computational load.

Test how well unrolling the loop works at different sizes. Over a certain threshold the loop becomes too big for efficient caching and performance drops.

On some microprocessors loop unrolling can be done automatically so you do not have to do it manually in your code. However, automatic code unrolling is limited. If your CPU supports automatic code unrolling keeping the code small is the better option. Test to see what works best.

Avoid branches in loops

A conditional test in a loop generates at least one branch instruction. Branch instructions can slow a microprocessor down and are especially bad in loops. Avoid branches if possible and especially in inner loops.

Do not make function calls in inner loops

Function calls in loops generate at least two branches and can initiate program reads from a different part of memory. If possible, avoid making function calls in loops and especially in inner loops.

Consider if the data be part processed in the loop first, then the call made outside the loop.

Some functions calls can be avoided by copying the contents of the function into the loop. The compiler might do this automatically.

Make use of tools

Various CPU tools can help with optimizing applications. Use what is available.

Use vector instructions if possible

Vector instructions process multiple data items at once so you can use these to speed up a loop or reduce the number of iterations. For more information see *Use vector instructions* on page 4-12.

Note

If you are processing a very small number of items it might be faster not to use a loop.

4.7 Use fast data structures

All graphics and application processing depends on data. If the processor cannot read and write this at high speed then neither the CPU or GPU can process it at high speed.

Experiment with different types of data structures and measure to see what is fastest.

4.8 Use vector instructions

Many ARM CPU and Mali GPU processors include vector or *Single Instruction Multiple Data* (SIMD) instructions. These enable the processor to perform multiple operations with a single instruction.

Vector processing works by processing multiple operations in parallel with a single instruction. The number and type of operations you can do depends on the type of vector processor extension in your processor.

For example, an ARM processor with the NEON Media Processing Engine can do up to 4 32bit operations, 8 16-bit operations, or 16 8-bit operations simultaneously, depending on the implementation.

Using vector instructions can produce a very large performance boost for some operations. Use vector processing where possible. This increases performance and reduces code size making it more cacheable.

You can sometime use vector instruction in a loop as a form of loop unrolling. This can reduce the number of total iterations the loop must do by 4 or more times.

Note

If the number of data items being processed is not a multiple of the number of elements in the vector, you might require additional code to process the end and possibly start elements. This code is only executed once.

4.9 Make use of under-used resources

This section describes making use of under-used resources. It contains the following sections:

- *Moving operations from the pixel processor to the geometry processor*
- *Moving operations from the geometry processor to the pixel processor*
- *Using spare resources to increase image quality*
- *Using spare resources to save power*.

If a resource is under-used consider moving computations to it. To determine if you have under used resources you must analyze the performance of your application. For more information, see Chapter 3 *The Optimization Process*.

4.9.1 Moving operations from the pixel processor to the geometry processor

In many applications the geometry processor is under-used. If your application is pixel processor bound, it might be possible to move some of these computations to the geometry processor.

For example, you can move some types of pixel shading computations to the geometry processor by using varyings. Varyings are computed per vertex and are interpolated across a triangle. This requires less computations than computing values per pixel but sometimes produces lower visual quality for some effects. The quality difference can be relatively small so consider if per pixel operations are worth the additional computations.

A compromise is to split calculations across both processors. You can use the geometry processor to output varyings and compute per pixel differences in the pixel processor.

4.9.2 Moving operations from the geometry processor to the pixel processor

If you application is geometry processor bound and it uses a lot of geometry for detail, then you might be able to reduce it by using techniques that make surfaces appear more detailed than they really are. For example, normal maps are textures that represent surface normals. Shaders use them to give the impression of more surface detail without increasing the number of triangles.

4.9.3 Using spare resources to increase image quality

If your application is performing well you can use spare resources to improve image quality by for example, adding additional effects.

4.9.4 Using spare resources to save power

If your application is performing well you can opt not to use spare resources. Not using spare resources saves power because the GPU can switch off when there is no work to do.

4.10 Ensure the graphics pipeline is kept running

The graphics pipeline consists of the CPU, geometry processor, and pixel processor. For the pipeline to work efficiently, data must be kept flowing through it.

Ensure your application keep the graphics pipeline running, It can do this by performing multiple types of operations simultaneously and avoiding blocking calls.

If data is not kept flowing, the CPU, geometry processor, or pixel processor might idle, waiting for data output from another stage.

Figure 4-1 shows the impact of a stall on the graphics pipeline. The diagram shows 8 jobs processed in 11 steps. The steps are indicated by the letters A to K:

- Steps A to D show data passing through the pipeline correctly. Job 1 starts in the CPU at step A and moves along the pipeline in steps B to C. The results arrive in the framebuffer in step D.
- Steps E to G show the impact of job 5 stalling in the CPU. Job 4 moves through the pipeline as it is processed in steps F and G, but the geometry processor runs out of work at step F. At step G both the geometry and pixel processors have no work.
- While the pipeline is stalled, the GPU cannot produce more frames so the framebuffer contains the results of job 4 for steps G to J.
- At step H, job 5 finishes in the CPU and moves into the geometry processor at step I. Job 5 moves along the pipeline and arrives in the framebuffer in stage K.

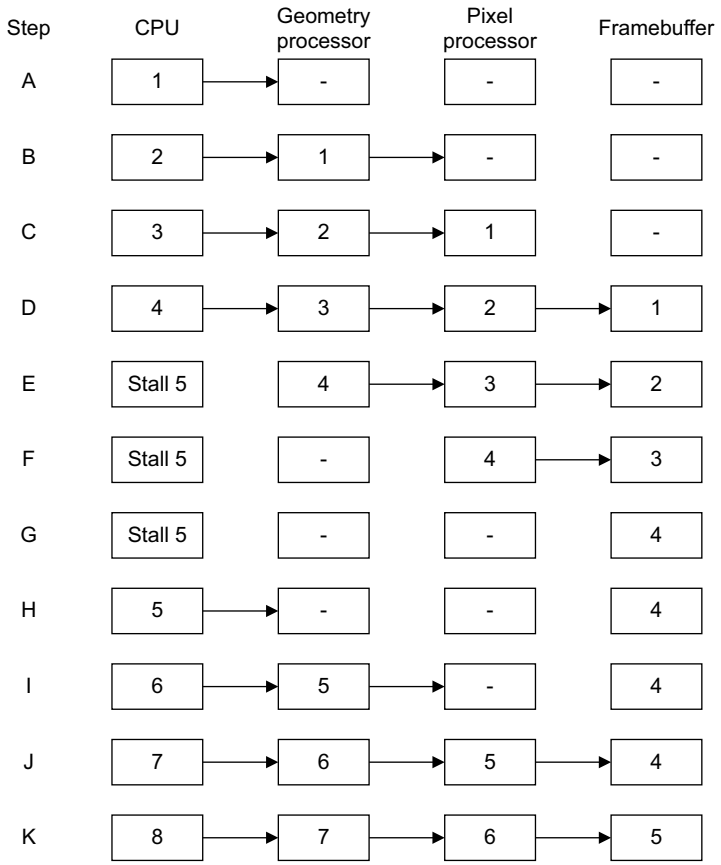

Figure 4-1 Graphics pipeline flow with stall

4.11 Application optimizations

There are many code level optimizations that can make an application perform better in specific areas. Consider using different algorithms and data structures. Changing these can make large performance gains.

Consider what problems you are trying to solve and see if you can solve them in different ways. For example:

- if you are modelling a landscape as a grid, an alternative is an irregular mesh.
- if you are representing object positions by placing them in a grid, an alternative is to use a mathematical approach.
- if you are computing collision detection in three dimensions, consider if the third dimension can be approximated or ignored. This can reduce many types of computations.

These are examples. Experiment with different approaches and measure the results to see how well they work.

If you cannot think of alternative approaches, describe the problem to someone else and see how they would solve it. If their solution is different then that is an alternative you can consider.

The act of describing the solution can be sufficient to bring new ideas to mind. This can assist you in thinking about possible alternatives.

Glossary

This glossary describes some of the terms used in Mali graphics processor documents from ARM Limited.

Anti-aliasing The process of removing or reducing aliasing artifacts, primarily jagged polygon edges, from an image. Anti-aliasing is particularly important for low-resolution displays. There exist several techniques to perform anti-aliasing, see multi-sampling and super-sampling.

Alpha Blending Alpha blending is a combination of two colors permitting transparency effects in computer graphics. The value of alpha in the color code ranges from 0.0 to 1.0, where 0.0 represents a fully transparent color, and 1.0 represents a fully opaque color.

API *See* Application Programming Interface.

Application Programming Interface (API)
A specification for a set of procedures, functions, data structures, and constants that are used to interface two or more software components together. For example, an API between an operating system and the application programs that use it might specify exactly how to read data from a file.

API driver A specialized driver that controls graphics hardware. Examples are OpenGL ES driver and OpenVG driver.

Architecture The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.

Blending A process where two sets of color and alpha values are merged together to form a new set of color and alpha values for a fragment.

See also Alpha-blending

Blitting Blitting copies an image into a buffer and optionally performs alpha-blending (if the alpha channel is present) to merge the source and destination areas together.

See also Copying, Alpha-blending, Blending.

Clipping The process of taking a triangle and geometrically removing parts of it that are outside a specific volume.

Copying Copying duplicates an image from a buffer to the display surface, overwriting the original contents of the display surface.

See also Blitting.

Device driver An operating system component that communicates with the graphics hardware.

Display subsystem The display that a final image is viewed on and the associated software that controls the operation of the display.

Draw mode The OpenGL ES APIs support several ways of specifying the primitives to draw, that is, different draw modes. The primitives can be specified individually or as a connected strip or fan. They can also be non-indexed, meaning that vertices are passed in a vertex array and processed in order, or indexed, meaning that an index array is used to look up vertices in a vertex array.

Early-Z A Z-testing scheme that performs the actual Z-test before texturing or fragment shading when it is safe to do so, increasing performance and reducing the required bandwidth.

EGL driver *See* Native platform graphics interface.

ESSL *See* OpenGL ES Shading Language.

ESSL compiler The compiler that translates shaders written in ESSL, into binary code for the shader units in the graphics processor. There are two versions of ESSL compiler - the on-target compiler and the offline compiler

ETC *See* Ericsson Texture Compression (ETC1)

Ericsson Texture Compression (ETC1)
A 4 *bit-per-pixel* (bpp) texture compression algorithm.

Fixed-function pipeline
A process that uses standard functions to draw graphics on fixed-function graphics hardware. For example, OpenGL ES 1.1 implements a fixed-function pipeline.

Fragment A fragment consists of all data, such as depth, stencil, texture, and color information, required to generate a pixel in the framebuffer. A pixel is usually composed of several fragments. A fragment can be multi-sampled and super-sampled.

Fragment processor A programmable component of the pixel processor, that runs fragment shaders.

Fragment shader A program running on the fragment processor that calculates the color and other characteristics of each fragment.

Framebuffer A memory buffer containing a complete frame of graphical data, produced by the GPU.

Geometry processor A geometry processor, such as the MaliGP2, executes vertex shaders that typically contain transform and lighting calculations, and generates lists of primitives for a pixel processor to draw.

Graphic application A custom program that executes in the Mali graphics system and displays graphics content.

Graphics driver A software library implementing OpenGL ES or OpenVG, using graphics accelerator hardware. See also OpenGL ES driver and OpenVG driver.

Graphics pipeline The series of functions, in logical order, that must be performed to compute and display computer graphics.

Instrumented drivers Alternative graphics drivers that are used with the Mali GPU. The Instrumented drivers include additional functionality such as error logging and recording performance data files for use by the Performance Analysis Tool.

Mali Binary Asset Exporter
A converter tool for Windows that converts XML-based COLLADA documents to the Mali Binary Asset format for use with the Mali Demo Engine. The Mali Binary Asset Exporter is a component of the Mali Developer Tools.

Mali Demo Engine The Mali Demo Engine is a component of the Mali Developer Tools. The Mali Demo

Engine library enables you to develop 3D graphics applications more easily than using OpenGL ES alone.

Mali Demo Engine Library
A C++ class framework for developing OpenGL ES 2.0 applications for the Mali GPU.

Mali MMU A full-featured *Memory Management Unit* (MMU) that is present on Mali GPUs, such as the Mali-200 GPU.

Mali Surface The destination for Mali output. It can potentially be for color, depth, stencil, but in the EGL sections it is only related to color output buffers.

Microprocessor *See* Processor.

Mipmap A pre-calculated, optimized collection of bitmap images that accompanies a main texture, intended to increase rendering speed and reduce artifacts.

Multi-ICE A JTAG-based tool for debugging embedded systems.

Multi-sampling An anti-aliasing technique where each pixel in the framebuffer is split into multiple samples corresponding to different positions within the area covered by the pixel. Each fragment produced for the pixel is duplicated onto each sample, and operations such as alpha-blending and depth testing is performed on a per-sample basis. In the final image, the color of each pixel is the average between the colors of the samples for that pixel.

The Mali pixel processors support multi-sampling at four samples per pixel with negligible performance impact.

Native platform graphics interface (EGL) driver
A standardized set of functions that communicate between graphics software, such as OpenGL ES or OpenVG drivers, and the platform-specific window system that displays the image.

Offline Compiler A command line tool that translates vertex shaders and fragment shaders written in the ESSL into binary vertex shaders and binary fragment shaders that you can link and run on the graphics processor.

On-target compiler A component of the OpenGL ES 2.0 driver that translates shaders provided by the application in source form, into binary shader code, at runtime.

OpenGL ES driver On graphics systems that use the OpenGL ES API, the OpenGL ES driver is a specialized driver that controls the graphics hardware.

OpenGL ES Shading Language (ESSL)
A programming language used to create custom shader programs that can be used within a programmable pipeline, on graphics hardware. You can also use pre-defined library shaders, written in ESSL.

OpenVG driver On graphics systems that use the OpenVG API, the OpenVG driver is a specialized driver that controls the graphics hardware.

Render Target *See* Mali Surface

Performance Analysis Tool
A fully-customizable GUI tool that displays and analyzes performance data files produced by the Instrumented drivers, together with framebuffer information.

See also Instrumented drivers, Performance data file.

Performance counter Data produced by the Instrumented drivers and the GPU hardware, that can be displayed and analyzed as statistical information in the Performance Analysis Tool.

Performance data file Files that contain a description of the performance counters, together with the performance counter data in the form of a series of values and images. Performance data files are saved in `.ds2` format and can be loaded directly into the Performance Analysis Tool.

Pixel A pixel is a discrete element that forms part of an image on a display. The word pixel is derived from the term Picture Element.

Pixel processor A Mali pixel processor performs rendering operations to produce a final image for display.

Primitive A basic element that is used, with other primitives, to generate images. A primitive can be a point, a line, a triangle, or a quad. Properties of primitives are defined through the use of vertices. Each primitive is divided into fragments so that there is one or more fragments for each pixel covered by the primitive.

See also Vertex.

Processor A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.

Programmable pipeline
A process that uses custom programs to draw graphics on programmable graphics hardware. For example, OpenGL ES 2.0 implements a programmable pipeline.

Quad A rendering primitive with four vertices.

Rasterization The process of identifying the fragments of each triangle that cover each pixel on the display screen.

Rasterizer A unit or method to convert a geometrical description of primitives supported by Mali (point, line, triangle, or quad), to fragments. The rasterizer works on line equations generated in the triangle setup phase of Mali GPU pixel processors.

Sample A sample refers to a value or set of values at a point in space. The defining point of a sample is that it is a chosen value out of a continuous signal. In the context of graphics, the sample point is usually in the middle of a pixel, and what is sampled is the geometry descriptions of polygons.

Scissoring A process that restricts rendering to a specified rectangular region of a rendering surface.

Shader A program, usually an application program, running on the graphics processor, that calculates some aspect of the graphical output. See fragment shader and vertex shader.

Shader Library A set of shader examples, tutorials, and other information, designed to assist with developing shader programs for the Mali graphics processor. The Shader Library is a component of the Mali SDK.

Shading language A programming language used to define custom shader programs to run on programmable graphics hardware. Different graphics APIs support different shading languages.

SoC System-on-Chip.

Sub-pixel Full-color displays are made by combining red, green, and blue light in varying degrees to produce different shades of colors. In a display with a fixed pixel structure, such as LCDs or plasma panels, the red, green, and blue light comes from adjacent cells in the display's physical structure. The light from these three subpixels, one for each color, combine to create a single pixel. There are also pixel structures that do not rely on three subpixels.

Super-sampling An anti-aliasing technique where the image is rendered in a higher resolution than the framebuffer and then scaled down before being written to the framebuffer.

Super-sampling An anti-aliasing technique where the image is rendered in a higher resolution than the framebuffer and then scaled down before being written to the framebuffer.

Texture Compression tool
A component of the Mali Developer Tools that you can use to compress textures and images, using the ETC algorithm.

Tile buffer A memory buffer inside the GPU that holds the framebuffer contents for the tile that is currently being rendered. The tile buffer can be accessed without using the memory bus.

Triangle setup unit
The triangle setup unit is a component of Mali pixel processors, such as the Mali-200 pixel processor. Triangle setup prepares primitives for rendering by calculating various data that is required to rasterize and shade the primitive.

Unaligned A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.

Vertices This is the plural form of the word vertex.

See also Vertex.

Vertex A set of data defining the properties of one point of a primitive. For example, a point primitive, an endpoint of a line primitive, or a corner of a triangle primitive.

Vertex attributes The data provided by the application, to define a vertex.

Vertex shader A program running on the geometry processor, that calculates the position and other characteristics, such as color and texture coordinates, for each vertex.