PowerVR

Performance Recommendations

Filename	:	PowerVR.Performance Recommendations
Version	:	PowerVR SDK REL_17.2@4910709a External Issue
Issue Date	:	30 Oct 2017
Author	:	Imagination Technologies Limited

Contents

List of Figures

1. Introduction

PowerVR SGX and PowerVR Rogue are Graphics Core architectures from Imagination Technologies designed specifically for shader-based APIs such as OpenGL ES 2.0, 3.x, and Vulkan (compatibility depends on the graphics core and available driver). Due to their scalable architectures, the PowerVR family spans a huge performance range.

1.1. Document Overview

The purpose of this document is to serve as recommendation and advice for developers who wish to get the best graphics performance from a PowerVR SGX or PowerVR Rogue enabled device. Throughout the document, the specific recommendations for PowerVR SGX and PowerVR Rogue are marked as appropriate.

1.2. The Golden Rules

The golden rules are a set of more generic performance recommendations that developers should seek to implement, as well as observe as many of the techniques and principles mentioned to help produce well-behaved, high performance graphics applications. These rules are detailed in the document entitled "PowerVR Performance Recommendations: The Golden Rules", which is supplied with the PowerVR SDK.

1.3. Optimal Development Approach

It is crucial to adopt the practices identified in this document from the very start of development in order to save much time and effort later. Once an application is implemented to a near-final state, the process of iteration depicted in Figure 1 should be adopted. The main benefit of this approach is that time is not wasted and graphics quality is not comprised by making changes that do not benefit performance.

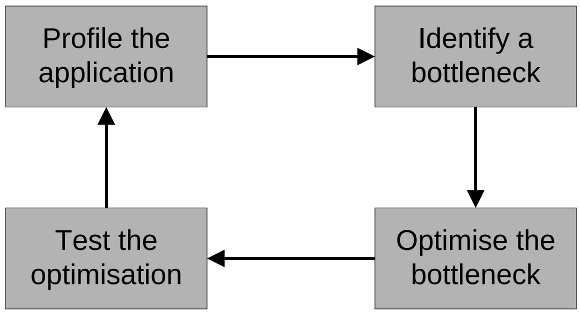

Figure 1. Cyclical profiling

1.4. Understanding Rendering Bottlenecks

It is a common misconception that the same actions can speed up any application. For example:

- **Polygon count reduction**: If the bottleneck of the application is fragment processing or texture bandwidth then the only result of this action will be to reduce the graphical quality of the application without improving rendering speed. In fact, if simpler models cause more of the render target to be covered by a material with complex fragments, then this can actually slow down an application.
- **Reduce rendering resolution**: In this case, if the fragment processing workload of your application is not the bottleneck then this will also only serve to reduce the quality of the graphics in your application without improving performance.

In reality, it is only once the limiting factor of an application is determined by profiling with the correct tools, that optimisation work should be applied. It is also important to realise that once work has been done then the application requires re-profiling in order to determine whether the work actually improved performance, and whether the bottleneck is still at the same stage of the graphics pipeline. It may be that the limiting stage in rendering is now at a different place and further optimisation should be targeted accordingly.

1.5. Optimising Applications for PowerVR Graphics Cores

1.5.1. Know the Target

Before diving into tools or performance recommendations, it is important to consider the capabilities and characteristics of the target device.

Graphics Architecture

A basic understanding of how API calls are processed by the driver, inserted into the graphics hardware's command stream, and converted into coloured pixels goes a long way. It provides an immediate appreciation of why certain graphics API calls are costly and how submitted calls will map to the graphics hardware's processing pipeline.

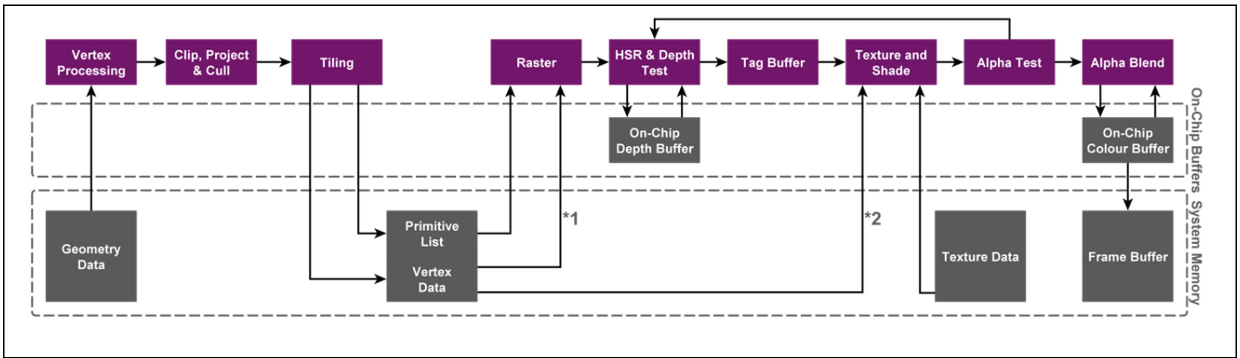

Figure 2. PowerVR TBDR Graphics Architecture

It is recommended to read the following documents in order to become familiar with the PowerVR graphics architecture:

- PowerVR Hardware Architecture Overview for Developers
- PowerVR Instruction Set Reference

These documents are packaged with the PowerVR SDK & Tools, and can also be found online here.

Mobile Graphics APIs

Mobile graphics APIs are a subset of their desktop counterparts, with imposed restrictions and specific features to suit the performance characteristics of mobile devices and the batteries that power them. Although the latest APIs, such as OpenGL ES 3.2, Vulkan and the Android Extension Pack, have brought many of the desktop and console features to low power devices, there are still differences that need to be considered.

Many of the recommendations in this document (as well as those in the PowerVR Performance Recommendations: The Golden Rules and PowerVR Supported Extensions documents) apply to all mobile graphics architectures. Of course, these documents also detail PowerVR specific behaviour and describe OpenGL ES extensions exposed by the PowerVR reference driver for advanced hardware features.

Thermal Design Power (Embedded Devices)

In this document we mainly focus on optimisations specifically for the PowerVR graphics core. However, reducing an application's workload on the CPU can be beneficial to the entire System on

Chip (SoC) – which includes the graphics core. This is because the CPU and graphics core share Thermal Design Power (TDP), meaning a reduction in CPU workload can not only reduce power draw for the entire SoC, but also reduce thermal output. This will help to prevent thermal throttling which would reduce the amount of power being sent to the SoC, potentially increasing the performance of the graphics core.

One of the primary advantages of the Vulkan API if used correctly is that it can reduce CPU workload. This is in part due to Vulkan drivers being very efficient and lightweight, which reduces overheads when using the API. This means that Vulkan has the potential to reduce the overall power consumption and thermal output of the entire SoC.

1.5.2. Analysing an Application's Performance

The Process of Optimising Graphics Applications

Optimising graphics applications seems like a straightforward process. Although the steps in the diagram below may seem obvious, both beginners and experienced developers have in the past made simple mistakes that have cost large amounts of development time to resolve. Developers tend to run their analysis tools, identify a bottleneck, modify their application and consider the work done. One of the most important stages of optimisation though is to verify the change has actually improved performance. Without analysing performance after a modification, it's easy for new (and possibly worse) bottlenecks to creep their way into a renderer.

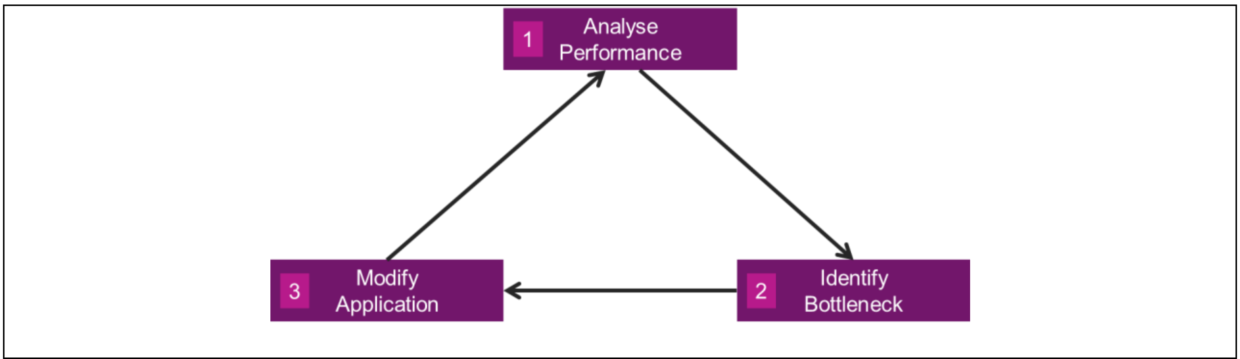

Figure 3. Optimisation Process for Graphics Applications

The Right Tools for the Job

The PowerVR SDK includes profilers, debuggers and a variety of other analysis tools to help developers track down issues. Here's a quick overview of the key utilities:

- **PVRMonitor:** Renders a real-time overlay of the CPU and graphics stats on Android devices with PowerVR graphics cores.
- **PVRTune:** Remote graphics performance analyser (server on the target, GUI analysis tool on your development machine). Captures timing data and counters, such as hardware unit loads and throughput data, in real-time.
- **PVRShaderEditor:** Off-line shader editor & performance analyser. Generates disassembly in real-time for SGX and Rogue graphics cores.
- **PVRTrace:** OpenGL ES 1.x, 2.0 & 3.x capture and analysis tool.

For more information regarding our entire suite of development tools, please visit our PowerVR Tools landing page here.

2. Optimising Geometry

2.1. Geometry Complexity

It is important that an appropriate level of geometry complexity be used for each object or portion of an object. Here are some examples of wasteful usage:

- Using a large number of polygons on an object that will never cover more than a small area of the screen.
- Using polygons for detail that will never be seen due to camera angle, or culling.
- Using large numbers of primitives for objects that may be drawn with fewer primitives. For example, using hundreds of polygons to render a single quad.

Shader techniques such as bump mapping should be considered to minimise geometry complexity, but still maintain a high level of perceived detail. Other techniques such as “Level of Detail” should also be used. This is especially true for things such as reflection passes where higher amounts of geometry may not be visible.

2.2. Primitive Type

For optimal performance on PowerVR Graphics Cores, a mesh with static attribute data should:

- Use indexed triangle lists
- Interleave VBO attribute data
- Not include unused attributes

For optimal vertex shader execution performance, meshes transformed by the same vertex shader (even if compiled into different shader programs) must have the same VBO attribute data layout. The PVRGeoPod tool can be used to generate vertex data which is optimised for cache coherency.

On some devices, padding each vertex to 16 byte boundaries may also improve performance.

2.3. Data Types

When passing data into a shader (i.e. uniforms, varyings, attributes) always consider their usage. If the data is intended to be used in math operations then always use a floating point data type to prevent unnecessary type conversions between integer and float. If the data is intended to be used solely as an integer e.g. an index to an array, then the data will not require a conversion.

Note that this implicit conversion is performed in the Unified Shader Core (USC) and costs a few additional cycles, and so should be avoided. Additionally the throughput (ops/cycle) of various data types should also be considered, see below (fastest to slowest):

- FP16 – mediump (see section 4.5.2)
- FP32 – highp (see section 4.5.1)
- INT

The conversion will be performed if the application uses an integer attribute type in a floating point operation, otherwise there is no additional cost associated with integer types.

The choice of attribute data type is a trade-off between shader cycles, bandwidth/storage requirements and precision. It is important that type conversion is considered as bandwidth is always at a premium.

There are a couple of exceptions to this rule:

- The data type 10F11F11F will require conversion to be performed by the hardware.
- The data type 8 bit S/UNORM may not require conversion depending on the hardware you are deploying to.

Precision requirements should be checked carefully, the byte and short types are often sufficient, even for position information. For example, scaled to a range of 10m the short types give a precision of 150 µm. Scaling and biasing those attribute values to fit a certain range can often be folded into other vertex shader calculations, e.g. multiplied into a transformation matrix.

2.4. Interleaving Attributes

Two ways exist to store vertex data in memory. These are either:

- The data is stored with all the information, position, normals and so on pertaining to a given vertex in a single block. This is followed by all the information pertaining to the next vertex, and so on.
- The data can be stored in a series of arrays, each containing all the information of a particular type for each vertex. For example, an array of positions, an array of normals and so on.

The first of these two options is called “interleaving”. In general data should be interleaved as this provides better cache efficiency, and thus better performance. However, two major caveats exist to this rule:

- Interleaving should not be used if several meshes share only some vertex attributes from an array of vertex attributes. In this case, separating the shared attributes into their own array may result in better performance compared to duplicating the shared attributes in order to interleave them with the shared ones.
- Interleaving should also not be used if a single attribute will be updated frequently, outside of the Graphics Core, while the other attributes remain the same. In this instance, data that will not be updated should be interleaved, while data that will be updated is held in a separate array.

2.5. Vertex Buffer Objects – OpenGL ES

Vertex Buffer Objects (VBO) are the preferred way of storing vertex and index data. Since VBO storage is managed by the driver there is no need to copy an array from the client side at every draw call and the driver is able to perform some transparent optimisations.

Pack all the vertex attributes that are required for a mesh into the same VBO unless a mixture of static and dynamic attributes are being used. Do not create a VBO for every mesh, it is a good idea to group meshes that are always rendered together in order to minimise buffer rebinding, this also has the benefit of improving batching.

As the graphics chip tends to process multiple frames at a time, the driver has to internally allocate multiple buffers for dynamic VBOs so that each frame has a unique dynamic buffer associated with it.

Due to this driver behaviour it is generally better for performance if dynamic vertex data (i.e. changes on a per-frame basis) is split from the static vertex data and placed into a separate VBO, and then only the dynamic VBOs need to be resubmitted per frame rather than the entire vertex data set.

If there is a mesh where only some of the vertex data is dynamic (for example, a skinned character in a game) then a VBO should be created that contains the static data and use calls to `glVertexAttribPointer` to resubmit the dynamic vertex data.

On a similar note, a VBO that will never change should always set the `STATIC_DRAW` flag while a VBO whose contents will change should never set it.

An alternative approach to dynamically updating vertex data involves creating a kind of ring buffer, where all the vertex data is stored in a single buffer, and the API function `glMapBufferRange` is used to map portions of the buffer for updating. This allows the application to update vertex data stored in one region of a vertex buffer while the graphics core is drawing from a separate region of the same vertex buffer. Note that data should only be overwritten when it is no longer being used by the graphics core, meaning fences should be employed to determine when it is 'safe' to update a region of buffer.

2.6. Draw Call Size

When using the OpenGL ES API, draw calls should contain a significant amount of work for the graphics core to process through the number of vertices, heavy shader arithmetic computation, or both. This is in order to negate the API overhead incurred by calling the function in the first place. Many 'light' draw calls with little work for the graphics core to process each time can incur unnecessary overhead and potentially lead to a CPU bottleneck.

When an application uses the Vulkan API correctly, this issue is much less critical due to the API being much more efficient. Command buffer submission results in significantly less driver overhead compared to OpenGL ES.

2.7. Triangle Size

On PowerVR hardware an application should try to avoid small triangle sizes, in terms of pixel coverage (area), especially dipping below 32 pixels per primitive. Rendering triangles smaller than this size will impact the efficiency of rasterization, which could potentially lead to a bottleneck.

In addition submitting many tiny triangles will likely result in the hardware spending a large percentage of time processing them in the vertex stage, due to the number of triangles rather than size. This is specifically with the tile accelerator (TA) fixed function hardware, which could potentially result in a TA bottleneck. Many small triangles will result in an increased number of accesses to the parameter buffer, which is located in system memory. This will increase the memory bandwidth footprint.

Also note that sub-pixel triangles (triangles that cover less than a single pixel) will be discarded by the hardware.

Parameter Buffer

The tile accelerator (TA) on PowerVR graphics cores generates a list containing pointers to each vertex passed in from the application. This structure determines which tile each vertex resides in. This list is called the parameter buffer (PB) and is stored in system memory.

2.8. Face Culling

An application should enable face culling wherever possible, and correctly set the appropriate face to be culled. Enabling face culling significantly reduces the amount of load on the graphics hardware. This is due to the significant reduction in the number of polygons that must be processed by hardware. The tiling hardware (TA) does not need to bin as many triangles and the ISP does not need to rasterize as many triangles. This can help performance significantly on highly complex workloads with many hundreds of thousands or millions of polygons and reduce the likelihood of bottlenecks in these fixed function pipeline stages.

2.9. Sorting Geometry

2.9.1. Distance

On PowerVR hardware there is no performance benefit to be gained by sorting opaque geometry based on distance from the camera. This is because the hidden surface removal (HSR) hardware will detect and remove occluded (opaque) geometry from the pipeline automatically before fragment processing begins. In fact performing this operation would be a waste of resources.

However for blended geometry, sorting is necessary to get the correct results and for alpha tested geometry it may actually be beneficial to sort. If the application uses opaque, alpha tested and alpha blended objects in a scene, then we advise that the application renders the opaque objects first, followed by the alpha tested objects and finally the transparent geometry.

2.9.2. Render State

There are performance gains to be had by sorting geometry based on render state, for example common materials, shaders, render targets or resources. Sorting the geometry based on this method can significantly reduce the amount of resources that are switched (operations which may incur significant overheads) by the driver during rendering of the frame. This can reduce the amount of work that the driver performs, therefore reducing CPU workload. It can also reduce the amount of time the graphics core spends idle waiting for resources to be ready before being able to render.

Render to Texture

For maximised performance the preferred method for rendering to textures in OpenGL ES is through the use of Frame Buffer Objects (FBOs) with textures as attachments.

An application should **always** render to frame buffer objects (FBO) in series, this means submitting all calls for one FBO before moving to the next. This serves to significantly reduce unnecessary system memory bandwidth usage, caused by flushing partially completed renders to system memory when the target FBO is changed, and additionally it will minimise state changes. For optimal performance, attachments should be unique to each FBO and attachments should not be added or removed once the FBO has been created.

Note that PowerVR hardware exposes a unique extension to efficiently perform down sampling on frame buffer attachments, see section 6.7 for further information.

2.10. Z Pre-pass

On PowerVR hardware there is no performance benefit to rendering a low-poly geometry Z pre-pass to save fragment processing later. Performing this operation would be a waste of clock cycles and memory bandwidth. This is because the PowerVR hidden surface removal (HSR) hardware will detect and remove occluded (opaque) geometry from the pipeline automatically during rasterization before fragment processing begins.

3. Optimising Textures

3.1. Texture Size

It is a common misconception that bigger textures always look better. A 1024x1024 texture that never takes up more than a 32x32 area of the screen is a waste of both storage space and reduces cache efficiency. A texture's size should be based on its usage; there should be a 1 pixel to 1 texel mapping when the object that it is mapped to is viewed from the closest allowable distance.

Before considering reducing the resolution of your texture assets to save storage space, you should apply texture compression. If the quality of the lossy texture compression is unacceptable, you can then consider using an 8 or 16 bit per pixel uncompressed format. If you still need to reduce the storage space of your assets, you should then consider reducing the resolution of your images.

3.1.1. Demystifying NPOT

If a 2D texture has dimensions which are a power-of-two (i.e. width and height are 2^n and 2^m for some m and n), then the texture is said to be a POT texture (power-of-two). If they are not it is said to be an NPOT texture (non-power-of-two). This section seeks to clarify the status of NPOT textures in OpenGL ES.

OpenGL ES Support

NPOT textures are supported as required by the OpenGL ES specifications. However, it is necessary to point out the following:

- NPOT textures are not supported in OpenGL ES 1.1 implementations.
- NPOT textures are supported in OpenGL ES 2.0 implementations, but only with the wrap mode of `GL_CLAMP_TO_EDGE`.
    - The default wrap mode in OpenGL ES 2.0 is `GL_REPEAT`. This must be specifically overridden in an application to `GL_CLAMP_TO_EDGE` for NPOT textures to function correctly.
    - If this wrap mode is not correctly set then an "invalid texture" error will occur. Likewise a driver error may occur at runtime on newer drivers, to highlight the need to set a wrap mode.

GL_IMG_texture_npot

An extension exists (`GL_IMG_texture_npot`) to provide some of the functionality found outside of the core OpenGL ES specification. This extension allows the use of the following filters for NPOT textures:

- `LINEAR_MIPMAP_NEAREST`
- `LINEAR_MIPMAP_LINEAR`
- `NEAREST_MIPMAP_NEAREST`
- `NEAREST_MIPMAP_LINEAR`

It also allows the calling of `glGenerateMipmapOES` with an NPOT texture to generate NPOT MIP-maps. Like all other OpenGL ES extensions, the application should check for this extension's presence before attempting to load and use it.

Guidelines

Finally, a few additional points should be considered when using NPOT textures:

- POT textures should be favoured over NPOT textures for the majority of use cases as this gives the best opportunity for the hardware and driver to work optimally.

- A 512x128 texture will qualify as a POT texture, not an NPOT texture, where rectangular POT textures are fully supported.
- 2D applications, such as a browser or other application rendering UI elements where an NPOT texture is displayed with a one-to-one texel to pixel mapping, should see little performance loss from the use of NPOT textures other than possibly at upload time.
- To ensure that texture upload can be optimally performed by the hardware, use textures where both dimensions are multiples of 32 pixels.

The use of NPOT textures may cause a drop in performance during 3D rendering. This can vary depending upon MIP-map levels, size of the texture, texture usage and the target platform.

3.2. Texture Compression

Modern applications have become graphically intensive. Certain types of software, such as games or navigation aids, often need large amounts of textures in order to represent a scene with satisfying quality. Texture compression can save or allow better utilisation of bandwidth, power, and memory without noticeably losing graphical quality and should be used as much as possible. PowerVR hardware offers a specific form of texture compression called “PVRTC” which should be used as much as possible.

PVRTC is PowerVR’s proprietary texture compression scheme. It uses a sophisticated amplitude modulation scheme to compress textures. Texture data is encoded as two low-resolution images along with a full resolution, low bit-precision modulation signal. More information can be found in the whitepaper:

- Fenney, S. (2003) 'Texture Compression Using Low-Frequency Signal Modulation' *SIGGRAPH Conference*.

Additionally PVRTC supports both opaque (RGB) and translucent (RGBA) textures, unlike other formats such as S3TC that require a dedicated, larger form to support full alpha channels. It also boasts a very high image quality for competitive compression ratios of 4 bits per pixel (PVRTC 4bpp) and 2 bits per pixel (PVRTC 2bpp).

3.2.1. PVRTexTool

PVRTexTool (Figure 4) is a utility for compressing textures, which is an important technique that ensures the lowest possible texture memory overhead at application run-time. The PVRTexTool package includes a library, command-line and GUI tools, and a set of plug-ins. Plug-ins are available for Autodesk 3ds Max, Autodesk Maya, and Adobe Photoshop.

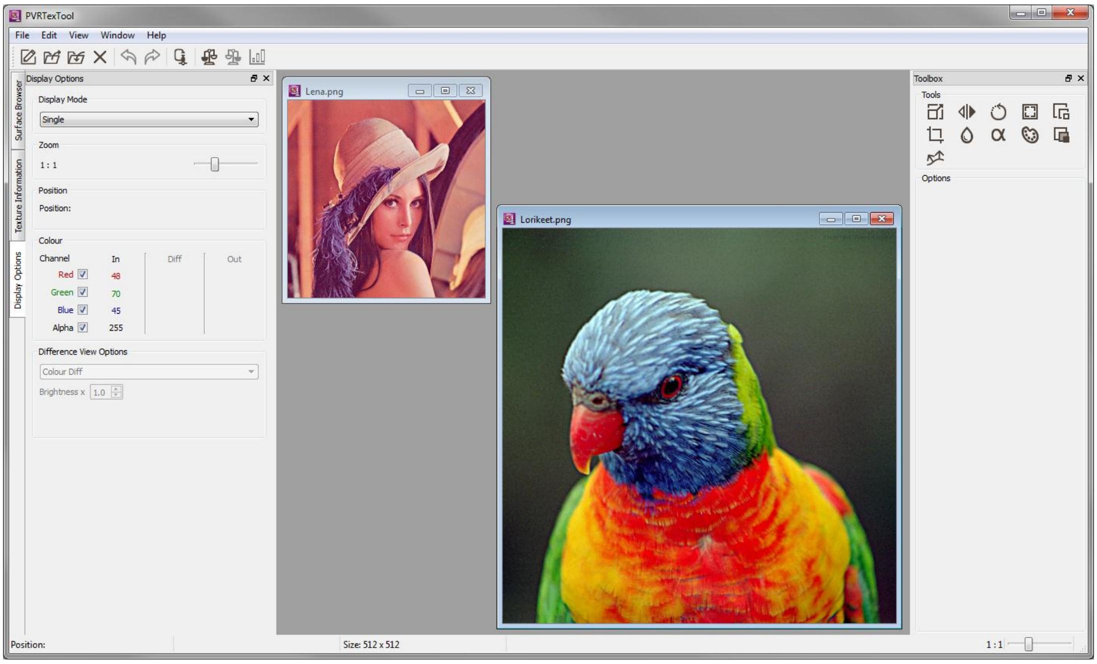

Figure 4. PVRTexTool GUI

Each component is capable of converting to a variety of popular compressed texture formats such as PVRTC and ETC, as well as all of the core texture formats for a variety of different APIs. They also include a number of advanced features to pre-process the image data, for example border generation, colour bleeding and normal map generation.

Textures can be saved to DDS, KTX, or PVR, which is Imagination's PowerVR Texture Container format which benefits from full public specification, support for custom metadata, as well as complete and optimised resource loading code in the PVRTools. Key features include:

- Supports all core texture formats in OpenGL ES, Vulkan and DirectX 11.1.
- PVRTC, ASTC, ETC and DXT texture compression.
- Outputs to PVR, KTX, or DDS files.
- Pre-processing textures for efficient rendering.
- Normal map generation.
- Composition and visualisation of cube maps.
- Optimised font to texture creation.
- Creation of texture arrays.

The latest version of the tool can be downloaded here.

Note: Texture arrays are allocated as contiguous blocks of memory. In OpenGL ES (only) modifying any texel within any element of the array will cause the driver to ghost the entire texture array. The KHR_debug logging will report when these ghosting events occur. In Vulkan all synchronisation is under the applications control.

3.2.2. Why use PVRTC?

In any given situation, the best texture format to use is the one that gives the required image quality at the highest rate of compression. The smaller the size of the texture data, the less bandwidth is required for texture fetches. This reduces power consumption, can increase performance, and allows for more textures to be used for the same budget. The smallest RGB and RGBA format currently available on all PowerVR Graphics Cores is PVRTC 2bpp and therefore it should be considered for every texture in an application. Larger formats (such as PVRTC 4bpp) should only be used if the image quality provided by a particular PVRTC 2bpp image does not have sufficient quality. On the latest PowerVR graphics cores, ASTC compression is also available.

Performance Improvement

The smaller memory footprint of PVRTC means less data is transferred from memory to the Graphics Core allowing for major bandwidth savings. In situations where memory bandwidth is the limiting factor in an application's performance, PVRTC can provide a significant boost. In addition PVRTC improves cache (on-chip memory) efficiency, because it takes less space to store the data in the cache. This can reduce the number of cache evictions and improve cache hit-rate.

Power Consumption

Memory accesses are one of the primary causes of increased power consumption on mobile devices where battery life is of the upmost importance. The bandwidth savings and better cache performance resulting from the use of PVRTC both contribute to decreasing the quantity and magnitude of memory accesses. These in turn reduce the power consumption of an application.

3.2.3. Image File Compression vs. Texture Compression

Developers are familiar with compressed image file formats such as JPG or PNG. It is important to be aware of the distinction between these forms of "storage" compression and the texture compression discussed in this document.

The primary requirement of storage compression schemes is that files compressed using them should occupy as small an amount of storage in a file system as possible. There is no requirement that the data stay compressed while in use. The result is that storage-based image file formats tend to produce very small file sizes, often for very high (or lossless) image quality, but at the cost of immediate decompression on use. This immediate decompression, usually to 24/32bpp means that the image, while small on disk, consumes large amounts of bandwidth and memory at runtime.

Texture compression schemes, such as PVRTC are designed to be directly usable by the Graphics Core. The texture data exists in storage, in memory, and when transferred to the graphics hardware itself, in the compressed format. The only step in which full-precision colour values are extracted from a compressed state is when dedicated texture sampling hardware inside the graphics accelerator passes texel values to the shader processing units. A graphical representation of this can be seen in Figure 5.

This allows all the advantages mentioned in Section 3.2.2, but puts some limits on the form the compression technique may take. In order to allow for direct use by the graphics accelerator a texture format should be optimised for random access, with a minimal size of data from which to retrieve each texel's values. Consequently, texture compression schemes are usually fixed bitrate with very high data locality. Image file formats are not constrained by these requirements and thus can often achieve higher compression ratios and image quality for a given data size.

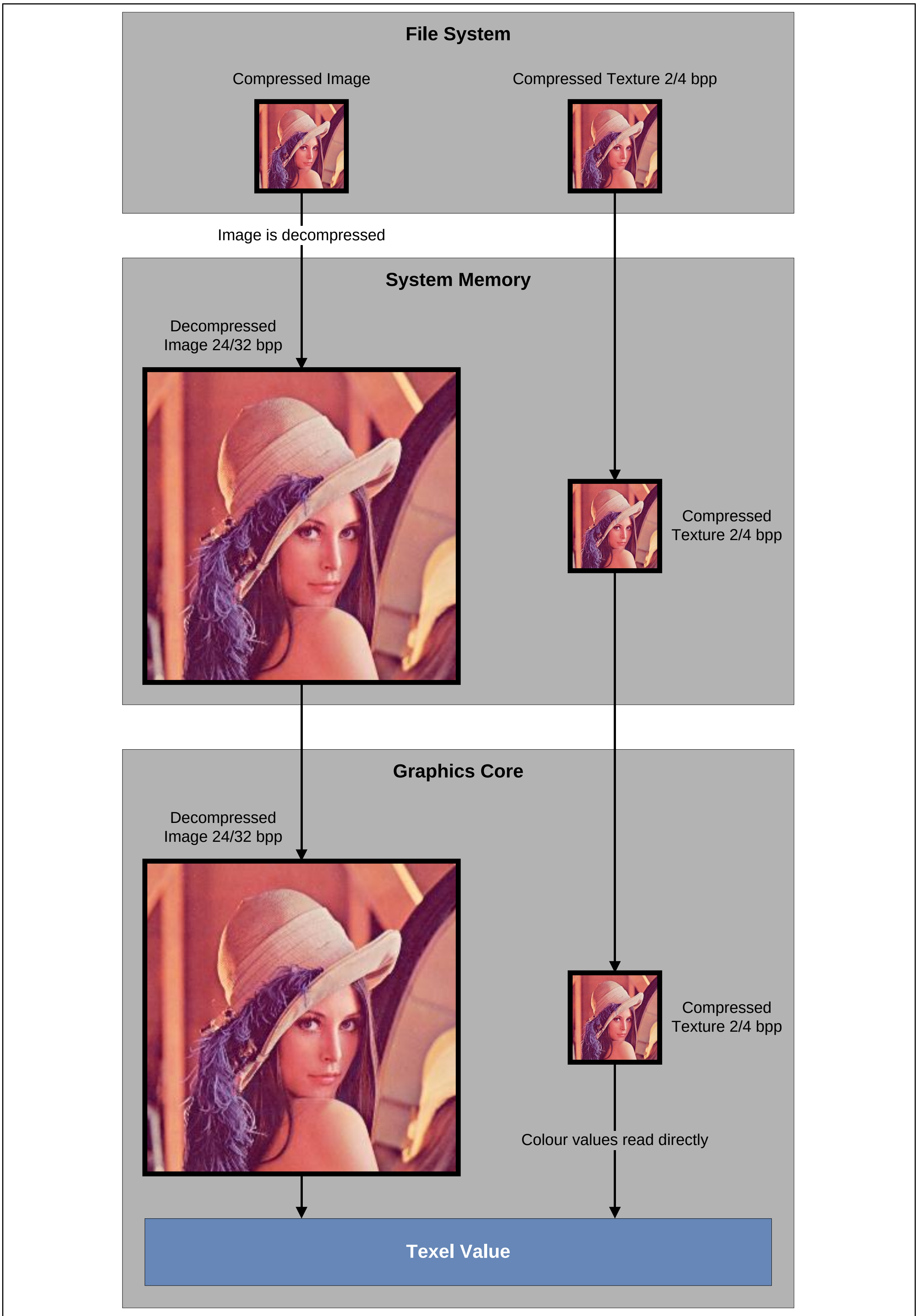

Figure 5. Image file compression vs. texture compression

3.3. Mipmapping

Mipmaps are smaller, pre-filtered variants of a texture image, representing different levels-of-detail (LOD) of a texture. By using a minification filter mode that uses mipmaps, the Graphics Core can be set up to automatically calculate which LOD comes closest to mapping the texels of a mipmap to pixels in the render target, and use the right mipmap for texturing.

3.3.1. Advantages

Using mipmaps has two important advantages:

- Increases performance by massively improving texture cache efficiency, especially in cases of strong minification.
- Improves image quality by countering the aliasing that is caused by the under-sampling of textures that do not use mipmapping.

The single limitation of mipmapping is that it requires approximately a third more texture memory per image. Depending on the situation, this cost may be minor when compared to the benefits in terms of rendering speed and image quality.

There are some exceptions where mipmaps should not be used. Specifically, mipmapping should not be used where filtering cannot be applied sensibly, such as for textures that contain non-image data such as indices or depth textures. It should also be avoided for textures that are never minified, for example, UI elements where texels are always mapped one-to-one to pixels.

3.3.2. Generation

Ideally mipmaps should be created offline using a tool like PVRTexTool, which is available as part of the PowerVR Graphics SDK. It is possible to generate mipmaps at runtime, which can be useful for updating the mipmaps for a render to texture target. In OpenGL ES this can be achieved using the function `glGenerateMipmap`. In Vulkan there is no such built in function, a developer must generate them manually. This will not work with PVRTC textures which must have their mipmaps generated offline. A decision must be made as to which cost is the most appropriate, the storage cost of offline generation, or the runtime cost (and increased code complexity in the case of Vulkan) of generating mipmaps at runtime.

3.3.3. Filtering

Finally, it should be noted that the lack of filtering between mipmap levels can lead to visible seams at mipmap transitions, a form of artefact called “mipmap banding”. Tri-linear filtering in OpenGL ES can be achieved by using the filter mode `GL_LINEAR_MIPMAP_LINEAR`. In Vulkan the filtering mode should be set to `VK_SAMPLER_MIPMAP_MODE_LINEAR` for tri-linear filtering. This can effectively eliminate these seams, for a price (see Section 3.4.1), and thus achieve an even higher image quality.

3.4. Texture Sampling

3.4.1. Texture Filtering

Texture filtering can be used to increase the image quality of textures used in 3D scenes. However, as the complexity of the filtering used increases, so will the associated cost as more samples are required.

The common techniques employed for texture filtering are:

- nearest
- bilinear
- cubic
- tri-linear
- anisotropic

Each technique above gives an increased image quality over the previous, at an increasing cost.

Performance can be gained by using an appropriate level of filtering, following the principle of "good enough" (see "PowerVR Performance Recommendations: The Golden Rules") For instance, not using anisotropic if tri-linear is acceptable, or not using tri-linear if bilinear is acceptable.

Filtering works by either taking a single sample, in the case of nearest filtering or multiple samples (involving multiple texture fetch operation) and combining (interpolating) them in order to produce as good a sampling value as possible to use in fragment calculations.

Retrieving multiple values requires more data to be fetched, possibly from disparate areas of memory and so cache performance and bandwidth use can be affected. For instance, when tri-linear filtering is used eight texel fetches are required, compared to only four for bilinear filtering or one for nearest filtering. This means the texture processing unit in the Graphics Core must spend more time and bandwidth fetching and filtering the required data as the complexity of the filtering increases.

The graphics core will attempt to hide memory access by scheduling USC tasks. However, if there is not enough work to hide the memory latency, then the texture fetches may cause the processing of a fragment to stall while the data is fetched from system memory. If the data is already in cache, then memory latency is much less an issue. Also more complex filtering techniques will result in additional data being transferred across the system memory bus in order to render a frame. Note that when performing independent texture reads, texture sampling can begin before the execution of a shader and so the latency of the texture fetch can be avoided, as the data is ready before shader execution.

On PowerVR hardware bilinear filtering is always hardware accelerated, including shadow sampling - sampling a texture with depth comparison activated - `sampler2DShadow`. In the case of shadow sampling the depth comparison operation is performed in software with USC instructions appended ("patched") to the fragment shader. The exact cost of the depth comparison operation will vary depending on the exact hardware the application is deployed to. A developer may determine the cost of the operation by using PVRShaderEditor (see section 4.1) and setting the appropriate GLSL compiler.

3.4.2. Texel Fetch

In certain cases performing a `texelFetch` operation can be considerably faster than calling the `texture` function. For example if the application is performing an expensive sampling operation, such as anisotropic filtering, it will likely be faster to perform a `texelFetch` operation over a `texture` operation, although this should be verified through profiling. On PowerVR hardware both operations are driven by dedicated hardware known as the texture processing unit (TPU) – in some special cases `texelFetch` may translate to a DMA operation.

3.4.3. Dependent Texture Read

A dependent texture read is a texture read in which the texture coordinates depend on some calculation within the shader instead of on a varying. As the values of this calculation cannot be known ahead of time it is not possible to pre-fetch texture data and so stalls in shader processing occur.

Vertex shader texture lookups always count as dependent texture reads, as do texture reads in fragment shaders where the texture read is based on the `.zw` channels of a varying. On some driver and platform revisions `Texture2DProj()` also qualifies as a dependent texture read if given a `Vec3` or a `Vec4` with an invalid `w`.

The cost associated with a dependent texture read can be amortised to some extent by hardware thread scheduling, particularly if the shader in question involves a lot of mathematical calculations. This process involves the thread scheduler suspending the current thread and swapping in another thread to process on the USC. This ‘swapped’ thread will process as much as possible, with the original thread being swapped back once the texture fetch is complete. Note that while the hardware will do its best to hide memory latency, dependent texture reads should be avoided wherever possible for good performance.

Further information on the functioning of the Coarse Grain Scheduler and thread scheduling within PowerVR hardware can be found in the “PowerVR Hardware Architecture Guide for Developers”.

Dependent texture reads are significantly more efficient on PowerVR Rogue Graphics Cores than SGX. However, there are still small performance gains to be had. For this reason, applications should always calculate coordinates before fragment shader execution unless the algorithm relies on this functionality.

3.4.4. Wide Floating Point Textures

For textures that exceed 32 bits per texel, each additional 32 bits is counted as a separate texture read. This also applies to half float texture with 3 or 4 components as well as float textures with 2 or more components. These larger formats should be avoided unless necessary for a particular effect.

3.5. Texture Uploading

When a non-compressed texture is uploaded to the graphics hardware the input data is in linear scan-line format (a compressed texture is uploaded block-by-block). Internally, PowerVR hardware uses its own layout to improve memory access locality and improve cache efficiency. Reformatting of the data is done on chip by dedicated hardware and thus is very fast. However it is still recommended that a few steps be taken to minimise the cost of this reformat:

- Textures should be uploaded during non-performance critical periods, such as initialisation. This helps avoid the frame rate dips associated with additional texture loading.
- Avoid uploading texture data mid-frame to a texture object that has already been used for that frame.
- Consider performing a “warm-up” step after texture uploads have been performed. Once again, this helps avoid the frame rate dips associated with texture loading.

3.5.1. Texture Warm-up

The warm-up step mentioned before ensures that textures are fully uploaded immediately. By default, `glTexImage2D` does not perform all the processing required to upload immediately. Instead, the texture is fully uploaded the first time it is used. It is possible to force an upload by drawing a series of triangles off screen or otherwise obscured with the texture object in question bound and so marked for use. Performing this for all textures in a scene will avoid the cost and potential stutters when they are uploaded on first use.

3.5.2. Texture Formats and Precision

In general, textures should be read as `lowp` (see Section 4.5.7). The exceptions to this are half float textures which should be read as `mediump`, and float and depth textures which should be read as `highp`.

3.6. Mathematical Look-ups

Sometimes it can be a good idea to encode the results of a complex function into a texture and use it as a look-up table instead of performing the calculations in a shader. However, this will only provide a boost in performance if a bottleneck has been identified in the processing of the shader in question, and bandwidth is free to perform the texture lookup. If the function parameters, and thus the texture coordinates in the look-up table, vary wildly between adjacent fragments then cache efficiency will suffer. As a significant amount of work must be saved for this to be an optimisation, profiling should be performed to determine if the results of using look-up tables are acceptable.

4. Optimising Shaders

4.1. PVRShaderEditor

To demystify shader optimisation, we provide a GUI utility called PVRShaderEditor (Figure 6) to share a wealth of off-line performance analysis data for developers as shaders are being written.

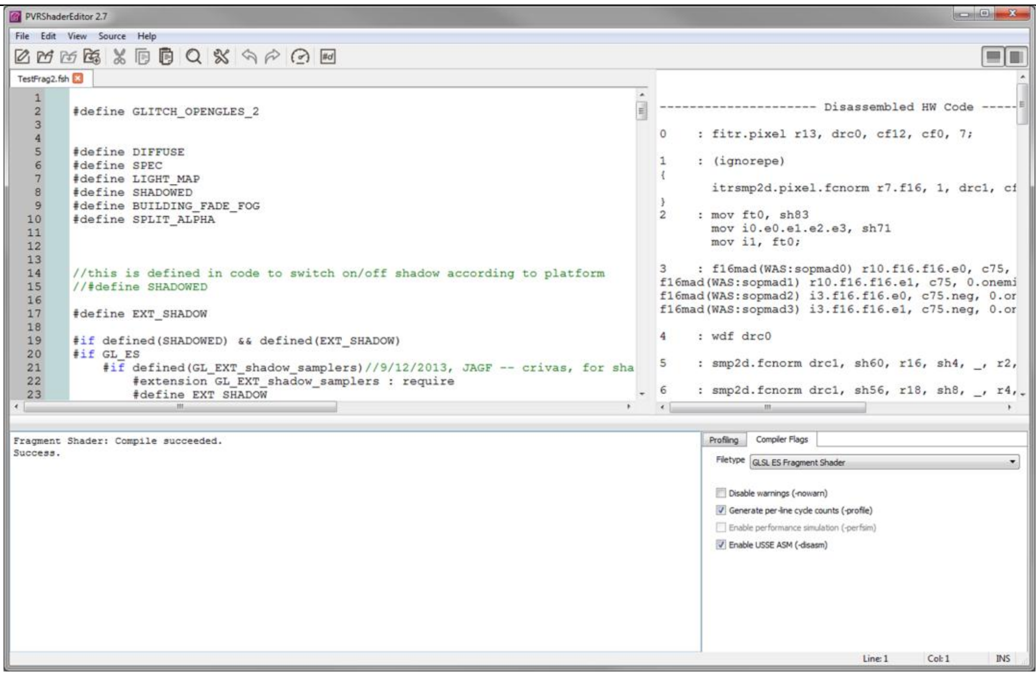

Figure 6. PVRShaderEditor GUI

Additionally, we provide shader disassembly for PowerVR Rogue graphics cores within the tool so developers can see the exact unified shader core (USC) instructions that have been generated by the compiler for the shader in question. Key features of the tool include:

- Syntax highlighting for GLSL ES, GLSL, PFX, HLSL and OpenCL Kernels.
- Supports PowerVR SGX and Rogue offline GLSL ES compilers.
- Per-line cycle count estimates.
- Simulated performance estimates (PowerVR Series5 and Series5XT graphics cores only).
- Full dissembled USC code, including FP16 disassembly.
- Supports Khronos reference GLSL compiler (optionally compiled to SPIR-V).

The latest version of PVRShaderEditor can be downloaded from our website here.

4.1.1. GLSL Optimiser

GLSL optimiser is a stand-alone C++ library, based on the Mesa GLSL compiler that is used in the many game engines to optimise GLSL shaders for mobile platforms. The tool can be found on Github here.

GLSL optimiser can automatically perform some graphics core independent GLSL shader optimisations. However keep in mind that the tool will not perform any hardware specific optimisations, but rather generic optimisations which will likely (but not always) improve performance on most platforms. In order to verify any shader optimisations performed by the tool, use PVRShaderEditor.

4.2. Choose the Right Algorithm

For complex shaders that run for more than a few cycles, picking the right algorithm is usually more important than low-level optimisations. It is highly recommended that a fast, well designed algorithm be favoured over small performance tweaks to a poor algorithm. Bear in mind that although increasingly powerful, mobile graphics hardware is not designed to handle some of the latest techniques in desktop and console shaders. As such, a reduction in complexity will likely be needed from some of these techniques for mobile shader implementations.

4.3. Know Your Spaces

A common mistake in vertex shaders is to perform unnecessary transformations between model space, world space, view space and clip space. If the model-world transformation is a rigid body transformation, i.e., it only consists of rotations, translations, mirroring, lighting and similar, calculations can be performed directly in model space. Transforming uniforms such as light positions and directions to model space is a per-mesh operation, as opposed to transforming the vertex position to world or view space once per vertex and so is an optimisation. In cases where a particular space must be used, e.g. for cube map reflections, it is often best to use this single space throughout.

4.4. Flow Control

PowerVR hardware offers full support for flow control in both vertex and fragment shaders without the need to explicitly enable an extension. When conditional execution depends on the value of a uniform variable, this is called "static flow control", and the same shader execution path is applied to all vertex or fragment instances in a draw call. "Dynamic flow control", on the other hand, refers to conditional execution based on per-fragment or per-vertex data, e.g. textures or vertex attributes.

Static flow control can be used to combine many shaders into one big "uber-shader". Thorough profiling should be done when taking this approach however, as a performance advantage may not be gained. A better solution when an uber-shader is desired is to use pre-processor defines to create separate shaders from one larger shader at build time, effectively creating many smaller shaders from a single original source file.

Using dynamic branching in a shader has a non-constant overhead that depends on the exact shader code. Dynamic branching is therefore unpredictable in its effect on performance. In general, the following specific points should be considered:

- Make use of conditionals to skip unnecessary operations when the condition is met in a significant number of cases.
- Do not branch to `discard` (see "PowerVR Performance Recommendations: The Golden Rules").
- **Series5 and Series5XT only**: Avoid branching to a texture read as samplers in dynamic branches qualify as "dependent texture reads" and will harm performance.

4.4.1. Discard

Applications should avoid the use of the `discard` operation in the fragment shader, as doing so will not improve performance. This is because some of the benefits of our TBDR architecture will be lost when discard is used, so if possible prefer alpha blending over discard. Note that this is a general problem across many tile based platforms and applies to many mobile/embedded graphics cores not just PowerVR devices.

4.4.2. Shader Group Vote – OpenGL ES

OpenGL ES 3.0 provides a new extension `GL_EXT_shader_group_vote`. This extension is designed to allow divergent code (i.e. branching) in shader programs to be optimised. Consider first how the graphics core (a SIMD processor) executes shaders which are commonly grouped together into a set of shader invocations that all must take the same code path. In compute this is known as a local work group.

Consider the code snippet below. If even a single shader in the local work group diverges from all other active shaders in the local work group (i.e. condition is 'true') then all other threads in local work group must also execute the `do_fast_path()` code path. This will usually leave a majority of threads in the local work group dormant. Once the function `do_fast_path()` returns, all of the active shaders in the local work group must then also execute the `do_general_path()` code path, meaning the local work group executes both code paths.

```
if (condition) {
        result = do_fast_path();
}
else {
        result = do_general_path();
}
```

With the same example but using the `allInvocationsEXT` function (see below), the `allInvocationsEXT` function will return the same value for all invocations of the shader in the local work group. This means the group will either execute the `do_fast_path()` or the `do_general_path()` but not both paths. It achieves this by computing the Boolean value across the local work group. The implementation uses this result to decide which path to take for all active threads in the local work group.

```
if (allInvocationEXT(condition)) {
        result = do_fast_path();
}
else {
        result = do_general_path();
}
```

The `GL_EXT_shader_group_vote` extension exposes three new built-in shader functions:

- `bool anyInvocationEXT(bool value)` - returns `true` if and only if `value` is true for at least one active invocation in the local work group.
- `bool allInvocationsEXT(bool value)` - returns `true` if and only if `value` is true for all active invocations in the local work group.
- `bool allInvocationsEqualEXT(bool value)` - returns `true` if `value` is the same for all active invocations in the group.

Further details on this extension can be found on the Khronos extensions page here.

4.5. Demystifying Precision

PowerVR hardware is designed with support for the multiple precision features of graphics APIs such as OpenGL ES and Vulkan. Three precision modifiers are included in the API spec for OpenGL ES 2.0 onwards and Vulkan, namely `mediump`, `highp`, and `lowp`. Lower precision calculations can be performed faster, but need to be used carefully to avoid trouble with visible artefacts being introduced. The best method of arriving at the right precision for a given value is to begin with `lowp` or `mediump` for everything (except samplers) then increase the precision of specific variables until the visual output is as desired.

4.5.1. Highp

Float variables with the `highp` precision modifier will be represented as 32 bit floating point (FP32) values, whereas integer values range from 2^{31}-1 to -2^{31}. This precision should be used for all position calculations, including world, view, and projection matrices, as well as any bone matrices used for skinning where the precision, or range, of `mediump` is not sufficient. It should also be used for any scalar calculations that use complex built-in functions such as `sin`, `cos`, `pow`, `log`, etc.

4.5.2. Mediump

Variables declared with the `mediump` modifier are represented as 16 bit floating point (FP16) values covering the range [65520, -65520]. The integer values cover the range [2^{15}-1, -2^{15}].

It is advised that an application uses FP16 wherever appropriate as it typically offers a performance improvement over FP32, and should be considered wherever FP32 would normally be used. This is as long as the precision is sufficient and the maximum and minimum values will not be overflowed, as visual artefacts may be introduced.

Using medium precision (FP16) in shaders can result in a significant improvement in performance over high precision (FP32). This is due to the dedicated FP16 SOP (sum of products) arithmetic pipeline, which can perform two SOP operations in parallel per cycle, theoretically doubling the throughput of floating point operations. The FP16 SOP pipeline is available on most PowerVR Rogue graphics cores – depending on the exact variant. Additionally some Rogue cores (such as Series6 XT) also provide a FP16 MAD (multiply, add) arithmetic pipeline, which can perform two MAD operations in parallel per cycle, again significantly improving performance compared to high precision.

A developer can verify the improvements of using medium precision by opening the shader in PVRShaderEditor and selecting the appropriate compiler for the target device.

4.5.3. Lowp

SGX

A variable declared with the `lowp` modifier will use a 10 bit fixed point format on SGX allowing values in the range [-2, 2] to be represented to a precision of 1/256. The integer values are in the range of [2^9 -1, -2^9]. This precision is useful for representing colours and any data read from low precision textures, such as normals from a normal map. Care must be taken not to overflow the maximum or minimum value of `lowp` precision, especially with intermediate results.

Rogue

On PowerVR Rogue devices `lowp` is represented as a 16 bit floating point value, meaning `lowp` and `mediump` have identical representations as far as the hardware is concerned.

4.5.4. Swizzling

Swizzling is the act of accessing or reordering the components of a vector out of order. Some examples of swizzling can be found next:

```
a = var.brg;                          // Swizzled – Out of order access
b = vec3(var.g, var.b, var.r);        // Swizzled – Out of order access
c = vec3(vec4);                       // Not swizzled – Dropping a component does not change
                                      // access order
d.gr = a.gr + b.gr                    // Not swizzled – This will be optimized to a
                                      // non-swizzled form
```

Swizzling costs performance on Series5 (lowp only) and Series5XT (all precisions) due to the additional work required to reorder vector components. As PowerVR Rogue is scalar based, swizzling is a significantly cheaper operation.

4.5.5. Attributes

The per-vertex attributes passed to a vertex shader should use a precision appropriate to the data-type being passed in. For example, `highp` would not be required for a float whose maximum value never goes above 2 and for which a precision of 1/256 would be acceptable.

4.5.6. Varyings

Varyings represent the outputs from the vertex shader which are interpolated across a triangle and then fed into the fragment shader. Varyings are significantly cheaper than performing per-fragment operations to calculate data that could have been passed in from a vertex shader via a varying.

However a developer should keep the following considerations in mind when using varyings:

- Each varying requires additional space in the parameter buffer, and additional processing time to perform interpolation.
- Varying outputs are stored in on-chip memory; therefore having too many may introduce register pressure and potentially reduce shader occupancy, i.e. reduce the maximum number of concurrent shader executions per unified shader core (USC).

Packing Varyings

Packing multiple varyings together, for example packing two `Vec2` into a single `Vec4` should suffer no performance penalty and will save varyings. Exclusively on PowerVR Series5 and Series5XT, co-ordinate varyings which are packed into the `.zw` channel of a `Vec4` will always be treated as a dependent texture read and should be avoided (see Section 3.4.3).

4.5.7. Samplers

Samplers are used to sample from a texture bound to a certain texture unit. The default precision for sampler variables is `lowp`, and generally this is good enough. Two main exceptions exist to the `lowp` rule. If the sampler will be used to read from either a depth or float texture then it should be declared with `highp`. On the other hand, if the sampler will be used to read from a half float texture then it should be declared as `mediump`.

4.5.8. Uniforms

Uniform variables represent values that are constant for all vertices or fragments processed as part of a draw call, meaning they should be used to pass data that can be computed once on the CPU and then not changed for the duration of a draw call. Unlike attributes and varyings, uniform variables may be declared as arrays.

Using uniforms are significantly cheaper than using varyings; however a developer should keep the following considerations in mind when using uniforms:

- A certain number of uniforms (uniform storage varies between graphics cores) can be stored in registers on-chip. Large uniform arrays will be stored in system memory and accessing them comes at a bandwidth and execution time cost.
- Redundant uniform updates in between draw calls should be avoided.

Constant Calculations

The PowerVR shader compiler is able to extract calculations based on constant values (for example uniforms) from the shader and perform these calculations once per draw call.

4.5.9. Conversion Costs

When performing arithmetic operations on multiple precisions within the same calculation it is likely that values will have to be “packed” or “unpacked”. Packing is the act of taking a higher precision value and placing into a lower precision variable while unpacking is the reverse and involves taking a lower precision value and placing it into a higher precision variable.

Where possible precisions should be kept the same for an entire calculation as each pack and unpack has a cost associated with it. This cost can be further amortised by writing shaders in such a way that all higher precision calculations are performed together, at the top of the shader, and all lower precision calculations performed at the bottom. This ensures that variables are not repeatedly packed and unpacked. It also ensures that variables are not all unpacked into `highp` thereby losing any benefit of using lower precision.

Additionally using fixed point values in an arithmetic operation will result in the graphics core performing a type conversion. This should be avoided as additional cycles will be introduced to the shader.

4.6. Scalar Operations

It is very easy to accidently vectorise a calculation. Hence, one should be wary of vectorising scalar operations where it cost more cycles for the same output. For example:

```
highp vec4 v1, v2;
highp float x, y;

// Bad
v2 = (v1 * x) * y; // vector * scalar followed by vector * scalar totals 8 scalar muladds

// Good
v2 = v1 * (x * y); // scaler * scalar followed by vector * scalar totals 5 scalar muladds
```

4.7. Constant Data in Shaders

If used correctly the `const` keyword can provide a significant performance boost. For example, a shader that declares a `const` array outside of the `main()` block can perform significantly better than the same shader with the array not marked as `const`, even if the array could be treated as such. Another example would be the use of a `const` value to reference an array member. In this example, if the value is `const` the Graphics Core can know ahead of time that the number will not change and data can be pre-fetched prior to the shader being run.

4.8. Geometry / Tessellation Shaders

In general tessellation and geometry shaders should be avoided if possible, as the hardware will be required to 'bin' (place into tiles) many more vertices which are produced by these pipeline stages. This results in many more writes to the parameter buffer which is located off-chip in system memory. In addition geometry and tessellation will usually result in increased pressure on the rasterization hardware, due to the increased number of triangles. The exact impact on performance will depend on the exact graphics core that the application is being deployed to. Profiling with PVRTune would reveal the impact of using geometry and/or tessellation shaders.

If an application must use tessellation and/or geometry shaders on PowerVR hardware, it should be noted that the shaders may be hardware accelerated, may be emulated or may not be supported at all, depending on the exact graphics core that the application is being deployed to, see the list below:

- Series 6XE graphics cores do not support either the tessellation or geometry shader extension and therefore these stages in the graphics pipeline are not available on this platform.
- Series 6, Series 6XT, Series 7XE and Series 8XE graphics cores have native support (i.e. hardware acceleration) for geometry shaders and provide execution of tessellation shaders through software emulation.
- Series 7XT and Series 8XT graphics cores provide native support (i.e. hardware acceleration) for both geometry and tessellation shaders.

While geometry shaders are not optimal when used to introduce new geometry, they can be used to cull geometry, which may result in some moderate performance gains. This should be verified through profiling using our PVRTune tool which is available in our SDK.

5. Optimising Specific Techniques

5.1. Using Multiple Render Targets Efficiently

Multiple Render Targets (MRTs) are available in a variety of APIs, and are supported on PowerVR Rogue hardware. MRTs allow a developer to render images to multiple render target textures at once. These textures can then be used for such things as inputs to other shaders, applied to a 3D model, presented to the screen and so on. A common use case for MRTs is deferred shading, whereby the lighting calculations are stored in multiple render targets and then used to light the scene after it has been drawn.

Tile based architectures such as PowerVR hardware can efficiently use render targets by storing per-pixel render target data for a single tile (i.e. 32 x 32 pixels) entirely in on-chip memory, also known as Pixel Local Storage (PLS). This has the advantage of significantly reducing system memory access compared to immediate mode renderers.

On most PowerVR devices the recommended maximum size per pixel for a render target is 128 bits plus a depth attachment. On some graphics cores the amount of available memory for PLS may be increased to 256 bits plus a depth attachment.

It is highly recommended that applications do **not** exceed the amount of available per pixel storage as this will result in the render target data being spilled out into system memory. This is extremely expensive as the render target data will need to be read for each fragment from system memory when a shader accesses the data stored in the render target. This essentially negates one of the main benefits of tile based rendering, and costs huge amounts of memory bandwidth and performance. In addition exceeding the per pixel storage will also likely result in reduced USC occupancy, meaning the maximum number of active threads (shaders) executing in parallel per USC will be severely limited, resulting in reduced efficiency and performance.

On PowerVR hardware applications are able to use a variety of render target formats. If the per pixel render target data is able to fit into on-chip memory, then all texture accesses are handled by the on-chip memory bus and therefore all formats equally provide the same performance. This is due to the fact that no transactions from system memory to the chip are required to load and store the data.

In addition to memory transaction and performance considerations when render targets spill in system memory, not all render target formats will be supported at 'full rate' - over the system memory bus. Therefore transfer rates may be further reduced depending on the format and the Texture Processing Unit (TPU) available in the graphics core – see the list below:

- RGBA8 can be accessed at full rate.
- RGB10A2 can be accessed at almost full rate.
- RG11B10 can be accessed at half rate
- RGBA16F can be accessed at half rate
- RGBA32F can be accessed at quarter rate (no bilinear filtering)

5.1.1. Recommended HDR Texture Formats

There are several texture formats available, which can be used to store HDR texture data. Each format has its benefits and drawbacks. This section aims to discuss several HDR suitable texture formats that are currently available for use by developers.

The appropriate HDR texture format will depend on several factors such as available memory bandwidth, precision (quality), alpha support etc.

The following table details various attributes of HDR suitable texture formats:

Table 1. HDR Suitable Texture Formats

Texture Format	Bandwidth Cost	USC Cost	Filtering	Precision	Alpha
RGB10A2	Same as RGBA8	None	Hardware accelerated, slightly slower than RGBA8.	RGB channels have greater precision over RGBA8 at the cost of alpha precision.	Supports alpha (only 4 unique values)
RGBA16F	2x RGBA8	None	Hardware accelerated but performs at half the rate of RGBA8.	Far greater precision than RGBA8 (2^{16} values per channel).	Supports alpha.
RG11B10F	2x RGBA8 (internally stored as RGBA16F)	None	Hardware accelerated but performs at half the rate of RGBA8.	Same as RGBA16F.	Does not have an alpha channel.
RGBA32F	4x RGBA8	None	Hardware accelerated but performs at quarter the rate of RGBA8 and only supports nearest sampling.	Vastly greater precision than any other format (2^{32} values per channel).	Supports alpha.
RGBM* (RGBA8)	Same as RGBA8	Moderate USC cost for encoding / decoding the data.	Hardware does not natively support filtering on this format.	Encoding algorithm improves the range of values that can be represented by the RGB channels compared to standard RGBA8.	No alpha – sacrificed to provide improved RGB range.
RGBdiv8* (RGBA8)	Same as RGBA8	Slightly more complex than RGBM to encode / decode the data.	Hardware does not natively support filtering on this format.	Encoding algorithm improves the range of values that can be represented by the RGB channels compared to standard RGBA8.	No alpha – sacrificed to provide improved RGB range.

**See section 3.9.1*

For HDR texture formats which are natively supported by the hardware we recommend using either RGB10A2 or RGBA16F which has increased bandwidth. These textures provide a good balance between quality, performance (filtering) and memory bandwidth usage.

RGBM & RGBdiv8

Both RGBM and RGBdiv8 texture formats require the developer to implement encoding and decoding functionality into the shader as these formats are not natively supported by the hardware. This costs additional USC cycles, therefore if an application is USC limited it should not employ these formats.

These formats do have the advantage that they cost very little in terms of memory bandwidth i.e. they cost the same bandwidth as RGBA8, therefore if an application is bound by memory bandwidth it may be useful to explore these formats. Further information on the RGBM format can be found here, more information on RGBdiv8 can be found here.

5.2. Preferred Lighting Solution

There are several lighting techniques which an application may employ, each with their own costs and benefits. Choosing the most optimal algorithm for the task may improve performance significantly. Here is a list of the common lighting techniques and their usage scenarios:

- Forward shading is the recommended method to be used for a small number of light sources. ‘Small’ is less than 10 point lights, anything more and deferred will be faster.
- Traditional deferred shading is the recommended method to be used for many light sources. This technique is highly efficient on PowerVR hardware, because the per pixel data for the tile stored in the G-Buffer render targets can be stored in on-chip memory (pixel local storage). As a result, the fragment shaders can reuse the data already stored in on-chip memory, significantly reducing accesses to system memory and significantly improving performance.
- For all compute based techniques such as tiled deferred and forward+, it is worth keeping in mind that the contents of the G-buffer will need to be written out to system memory. This is costly for almost all mobile and embedded platforms including PowerVR graphics cores, due to the limited amount of system memory bandwidth available.

5.3. Preferred Shadowing Solution

There are many techniques for creating shadows. One technique that performs exceptionally well on PowerVR hardware is stencil shadowing, due to the fact that the hardware is very good at handling stencil buffers. This is because the data is stored in on-chip memory and never has to be written out to system memory, therefore if hard shadows are acceptable then it is recommended to employ a stencil shadowing algorithm.

Techniques that require results to be written to off-chip memory such as shadow mapping will generally perform worse than techniques that can be computed entirely in on-chip memory.

5.4. MSAA Performance

On PowerVR hardware Multi-Sampled Anti-Aliasing (MSAA) can be performed directly in on-chip memory before being written out to system memory, which saves valuable memory bandwidth. In general, MSAA is considered to cost relatively little performance. This is true for typical games and UIs, which have low geometry counts but very complex shaders. The complex shaders typically hide the cost of MSAA and have a reduced blend workload.

2x MSAA is virtually free on most PowerVR graphics cores (Rogue onwards), while 4x MSAA+ will noticeably impact performance. This is partly due to the increased on-chip memory footprint, which results in a reduction in tile dimensions (for instance 32 x 32 -> 32 x 16 -> 16 x 16 pixels) as the number of samples taken increases. This in turn results in an increased number of tiles that need to be processed by the tile accelerator hardware, which then increases the vertex stages overall processing cost.

The concept of ‘good enough’ should be followed in determining how much anti-aliasing is enough. An application may only require 2x MSAA to look ‘good enough’, while performing comfortably at a consistent 60 FPS. In some cases there may be no need for anti-aliasing to be used at all e.g. when the target device’s display has high PPI (pixels per-inch).

Performing MSAA becomes more costly when there is an alpha blended edge, resulting in the graphics core marking the pixels on the edge to “on edge blend”. On edge blend is a costly operation, as the blending is performed for each sample by a shader (i.e. in software). In contrast, on opaque edge is performed by dedicated hardware, and is a much cheaper operation as a result. On edge blend is also ‘sticky’, which means that once an on-screen pixel is marked, all subsequent blended pixels are blended by a shader, rather than by dedicated hardware. In order to mitigate these costs, submit all opaque geometry first, which keeps the pixels “off edge” for as long as possible. Also, developers should be extremely reserved with the use of blending, as blending has lots of performance implications, not just for MSAA.

5.5. Preferred Analytical AA Solution

Analytical anti-aliasing algorithms such as fast approximate anti-aliasing (FXAA) or sub-pixel morphological anti-aliasing (SMAA) are shader-based techniques. These types of anti-aliasing techniques use analytics to detect and blur sharp geometric features. They are post-processing algorithms which are performed in screen space, and generally have a fixed cost i.e. a full-screen pass. They do require more memory bandwidth, which is usually at a premium on mobile and embedded devices.

On PowerVR hardware the recommended analytical anti-aliasing solutions are as follows (best to worst performance):

- Fast approXimate Anti-Aliasing (FXAA) – single full screen pass.
- Conservative Morphological Anti-Aliasing (CMAA).
- Morphological Anti-Aliasing (MLAA).
- Sub-pixel Morphological Anti-Aliasing (SMAA).

5.6. Screen Space Ambient Occlusion

Screen Space Ambient Occlusion (SSAO) is a technique used for efficiently approximating how each point in the scene is affected by ambient light. SSAO is implemented in a fragment shader, which executes once per fragment i.e. a single full screen pass. In its simplest form the algorithm samples the depth buffer (stored in a texture) around the current pixel and attempts to estimate the occlusion for the current fragment. However this approach results in hundreds of random access to the depth texture stored in system memory, which will inevitably thrash the cache and result in poor performance.

If an application implements SSAO, it is recommended that the algorithm implements a form of hierarchical Z buffer (Hi-Z, HZB) optimisation. Briefly this involves performing a hierarchical Z pre-pass to build a MIP chain for the hardware calculated depth buffer. Building the depth MIP-chain involves progressively taking either the minimum or maximum depth value for a tile of pixels (e.g. 4 x 4) and storing the value in the next MIP level.

Hierarchical Z buffer optimisation significantly improves the rate of convergence for ray intersection (reducing the number of steps to find an intersection) and significantly improves cache efficiency. This is because a small region defining the working area at each level of the MIP chain will likely reside in cache, which significantly reduces accesses to system memory. Overall this results in improved performance and massively reduced system memory bandwidth. A white paper explaining one such approach can be found here.

5.7. Ray-Marching

If an application implements a ray-marching graphical effect such as Screen Space Reflections (SSR), we recommended that the algorithm implements an optimal sampling technique which takes as few samples as possible to achieve the desired quality. One possible approach to reducing the number of samples is to perform the ray marching in a half-screen resolution buffer i.e. down sampled. PowerVR hardware exposes an extension in OpenGL ES for hardware accelerated down sampling, see section 6.7. This approach could also be coupled with a hierarchical Z buffer (similar to the technique discussed in section 5.6) to accelerate the ray marching further.

In place of the relatively expensive ray marching algorithms, it may be worth considering using a parallax corrected local cube map method. This could be used as either a fall back technique, or even as the main technique as it may be 'good enough' visually for the application and is considerably cheaper to perform. In addition the application could use simple, cheap planar reflections for flat surfaces.

An article discussing an optimal SSR technique can be found here and an article discussing a local cube map can be found here.

5.8. Separable Kernels

Many post-processing techniques such as full screen blur (with a Gaussian blur) or gather and scatter operations like motion blur, depth of field and bloom etc. can be implemented with multiple separated kernels. The downside to using a multi-pass algorithm is that they can be very inefficient in terms of memory bandwidth. This is due to increased round trips to system memory i.e. write-out, read-back, write-out, read-back etc. This results in significantly increased memory bandwidth usage and power consumption, and may result in poor performance.

The alternative solution is to use a single kernel (i.e. a single pass – brute force) to achieve the desired graphical effect. However, condensing the algorithm into that single pass may actually result in even worse performance than the multi-pass technique. This is because the algorithm may require many more samples when performing in single pass mode to achieve the same level of quality. This will result in increased system memory bandwidth usage over the multi-pass.

An example of this is Gaussian blur, which is commonly implemented as a multi-pass technique i.e. a horizontal and vertical pass. This significantly simplifies the complexity of the algorithm when compared to a single pass approach, which requires significantly more samples. However there are full screen blur techniques that work with a single pass which have been proven to be efficient such as Epic’s single pass circular based filter algorithm instead of a two pass Gaussian. More information can be found here.

In order to choose the ideal algorithm (single or multi-pass) a developer should profile the algorithm to determine which technique provides the most efficient usage of system memory bandwidth and USC.

5.9. Efficient Sprite Rendering

Rendering sprites efficiently may seem like a trivial exercise. However, without careful consideration an application may be unresponsive and sluggish due to poor graphics performance. Traditional sprite rendering tends to see textures drawn using alpha blending on to quads. These quads will consist of large areas of alpha, either completely transparent (i.e. alpha value of 0), or partial alpha. Areas which are completely transparent are traditionally discarded using either the `discard` keyword or alpha testing, while areas of partial alpha undergo blending. Both of these have some form of impact on performance versus fully opaque objects meaning that a large number of sprites being drawn inefficiently can seriously harm performance.

The `discard` keyword (see “PowerVR Performance Recommendations: The Golden Rules”) should be avoided in favour of the much faster alpha blending.

Even when favouring alpha blending, performance can still be affected if there are a large number of sprites. One method to minimise the impact of several layers of blended sprites is to increase the geometry complexity of the sprites in order to reduce the amount of wasted transparent fragments. For example, if a sprite is circular in shape and is rendered using the most optimal fitting quad, 22% of the fragments processed are redundant. Significant performance improvements can be gained by reducing the wasted transparency by increasing geometry complexity.

PowerVR hardware has excellent vertex processing capabilities and is designed to handle large amounts of geometry data, far in excess of what is present in most sprite based applications. As such, increasing complexity should have minimal performance impact and any impact this may have is most likely outweighed by the savings of rendering less transparency. If we increase the complexity of the previous case of a perfectly fitting quad around a circular sprite to that of a dodecagon (twelve sided polygon) we can reduce the amount of wasted fragment processing to just 3%.

Assuming radius r of 64

$$A = 1 - \frac{\pi r^2}{(2r)^2}$$

$$A = 0.214$$

$$A = 21.4\%$$

vs

$$A = 1 - \frac{\pi r^2}{12\left(2 - \sqrt{3}\right)r^2}$$

$$A = 0.029$$

$$A = 2.9\%$$

Figure 7. Increasing complexity and reducing processing

Additionally the application should also consider splitting opaque and alpha blended objects (such as UI elements) that appear in the scene into separate draw submissions. In order for the rasterization to be performed as efficiently as possible, the elements should be rendered in the following order:

1. opaque scene elements
2. alpha blended scene elements
3. alpha blended UI elements.

5.10. Physically Based Rendering & Per-Pixel LOD – Rogue Performance

Physically Based Rendering (PBR) is a forward and deferred render compatible lighting model that aims to better represent real world light behaviour. It is more costly to calculate than traditional diffuse, specular, and ambient lighting, but it is very appealing to artists as it makes it easier to specify complex material properties. PBR art pipelines are rapidly becoming the norm in AAA titles.

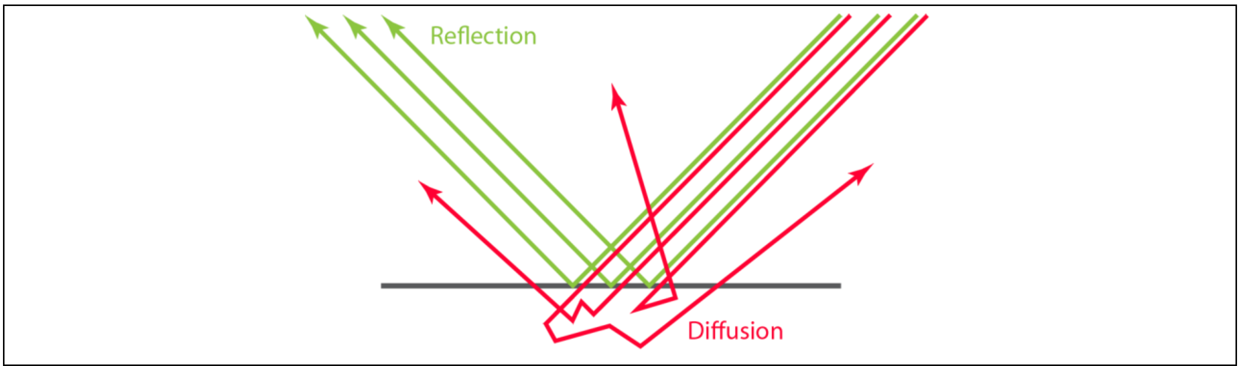

Figure 8. PBR Theory

Per-Pixel Texture LOD

PBR pairs each object in a scene with a roughness/gloss map. This texture allows artists to alter the surface roughness and glossiness across an object rather than having a consistent surface roughness or glossiness across the entire object. An example use is to add areas of dull rust to a shiny pistol, or to describe the properties of a rubber grip, all within a single draw call.

To add an element of reflectivity, environment maps are applied to all objects. Each environment map contains progressively blurrier surfaces towards the bottom of the chain. The sampled roughness value is used to calculate which MIP level of the environment map should be sampled.

Why is this approach a problem for Rogue?

Rogue sub-divides a fragment shader USC task into 2×2 blocks of spatially aligned pixels. A primary reason for doing this is so gradients can be calculated across a pixel-quad to determine how texture filtering should be applied. It is also optimised for the standard rendering case where a LOD value is calculated for a pixel-quad based on the calculated gradients. This allows the graphics core to batch texture sample operations for the pixel-quad into a single TPU request.

When texture LOD is specified per-pixel (passed in via a varying), the graphics core has to assume that each pixel in the quad has a unique LOD. This causes the USC to issue a TPU request for each pixel instead of the entire quad (which is denoted by the USC instruction - `pplod`), which in turn causes 1/4 TPU throughput.

This behaviour could lead to a memory bandwidth bottleneck in some applications. Shown below is an example fragment shader which shows how an application can get around this behaviour by using branching and performing bilinear filtering in software. By branching to a textureLod operation with a constant value (as the LOD parameter), the compiler will no longer make assumptions about the LOD of each pixel, meaning the compiler will not automatically fetch a sample per pixel in the pixel group. Note however the workaround described below increases the number of USC instructions significantly. Therefore it is important to profile the application before implementing the workaround to determine if the application is bandwidth or USC limited, as this workaround may negatively impact performance if the application is already USC limited. Decreasing memory bandwidth in an application that is USC limited would yield no performance benefits.

The Workaround

There is a GLSL workaround to avoid the 1/4 speed path. However, it introduces dynamic branching and additional instructions.

```
#version 310 es

in mediump float LOD;
in mediump vec3 TexCoords;

uniform lowp samplerCube EnvMap;

layout (location = 0) out lowp vec4 oColour;

mediump vec4 envSample(lowp samplerCube envMap_, mediump vec3 texCoords_, mediump float LOD_){
        mediump vec4 mip0;
        mediump vec4 mip1;

        if(LOD_ <= 4.0){
                if(LOD_ <= 2.0){
                        mip1 = textureLod(envMap_, texCoords_, 1.0);
                }else{ // LOD_ > 2.0
                        mip1 = textureLod(envMap_, texCoords_, 3.0);
                }
        }else{ // LOD_ > 4.0
                if(LOD_ <= 6.0){
                        mip1 = textureLod(envMap_, texCoords_, 5.0);
                }else{ // LOD > 6.0
                        mip1 = textureLod(envMap_, texCoords_, 7.0);
                }
        }

        if(LOD_ <= 3.0){
                if(LOD_ <= 1.0){
                        mip0 = textureLod(envMap_, texCoords_, 0.0);
                }else{ // LOD_ > 1.0
                        mip0 = textureLod(envMap_, texCoords_, 2.0);
                }
        }else{ // LOD_ > 3.0
                if(LOD_ <= 5.0){
                        mip0 = textureLod(envMap_, texCoords_, 4.0);
                }else{ // LOD_ > 5.0
                        mip0 = textureLod(envMap_, texCoords_, 6.0);
                }
        }

        bool isEven = ((int(LOD_) & 1) == 0);
        mediump float fractVal = fract(LOD_);
        mediump float invFractVal = 1.0 - fractVal;
        mediump float mixVal = isEven ? fractVal : invFractVal;
        return mix(mip0, mip1, mixVal);
}

void main() {
        oColour = envSample(EnvMap, TexCoords, LOD);
}
```

6. OpenGL ES Specific Optimisations

6.1. glClears & glColorMask

It is important that an application avoids a partial clear (partial colour mask) at the start of a frame for two important reasons:

1. The previous frame has to be read in. This is performed by a full screen primitive reading it in as a texture.
2. This texture must be masked out by the partial clear, which is done by submitting another full screen primitive as a blend.

It is important to note that this will result in two overdraws before we even begin working on the frame. If the colour mask is changed to full and `glClear`, this counts as a state change for the colour mask. A state change requires a flush to be performed on the tile accelerator and the clear becomes another full screen primitive. This adds the second overdraw.

In the case of one full clear (no partial colour masks) at the start of frame, the “fast clear” path is followed. This marks the whole frame as a set colour and does nothing, so no full screen primitive is required, resulting in no pixels being drawn at all.

Note that PVRTrace GUI emulates this behaviour.

6.1.1. Invalidating Frame Buffer Attachments

To prevent unnecessary memory transfers when rendering to a frame buffer object using the OpenGL ES API, it is recommended that the application invalidates frame buffers through the use of the function `glInvalidateFramebuffer` e.g. `GL_DEPTH_ATTATCHMENT`, `GL_STENCIL_ATTATCHMENT`. Calling this function tells the driver to discard the contents of the specified frame buffer attachments, and therefore the driver does not need to store the contents of the frame buffer attachments into system memory. This can save huge amounts of memory bandwidth and improve performance significantly as by default OpenGL ES will preserve frame buffer attachments.

Additionally an application should call `glClear` e.g. `GL_DEPTH_BUFFER_BIT`, `GL_STENCIL_BUFFER_BIT`, specifying the buffers to clear. Calling this function will tell the driver that it does not need to load the contents of the attachments specified from system memory, again saving a huge amount of memory bandwidth.

6.2. Draw*Indirect and MultiDraw*IndirectEXT

6.2.1. Draw*Indirect

A standard OpenGL ES draw call requires the programmer to pass the parameters of the draw via the function’s arguments. With the Draw*Indirect calls, the programmer can instead pass in a structure containing the draw parameters. The great thing about this structure is that it doesn’t have to be populated by the CPU – a programmer can use the graphics driver and SSBOs to populate the structure. This enables the application to issue a draw without any CPU-side involvement.

Example Use Case: Batched Draws

For optimal performance, applications should batch draws by state to reduce the number of API calls. The problem here is that a separate draw call needs to be issued for each object in that batch and draw calls have a CPU overhead it is better to avoid. With Draw*Indirect, developers can populate an SSBO with the vertex data of all draws that share the same state. With this SSBO, only a single Draw*Indirect needs to be made.

Example Use Case: Particle Systems

Another use case could be a particle system where the developer doesn’t want to allocate a big array for particles up front. Instead, the developer could use a compute shader to determine how many particles need to be rendered each frame. For really complex particle systems, they could also remove particles from the render if they are obscured by opaque objects.

6.2.2. MultiDraw*IndirectEXT

These API calls are very similar to Draw*Indirect. The key difference is that an array of Draw*IndirectCommand structures can be passed into each draw call.

Example Use Case: Occlusion Culling

In complex 3D navigation systems, draw calls tend to be grouped by map tiles. If a map tile intersects the view frustum, all draw calls within the tile are issued to the graphics core. This can be optimised with occlusion queries to further reduce the number of draw calls that are issued. With MultiDraw*IndirectEXT a developer can do even better than occlusion queries. A compute shader can be used to populate an array of Draw*IndirectCommand structures. Those can then be used to issue a single draw call for many objects sitting in many different tiles.

6.2.3. Instancing

Instancing is extremely useful for drawing many hundreds or thousands of objects that share the same vertex data, but have different world transformations.

Consider the example of drawing many thousands of leaf objects that are very simple in terms of geometry. With the 'non-instanced' approach, the application would need to loop `X` times calling `glDraw*` on the same object each time. This is extremely expensive in terms of API overhead, even if the geometry is relatively simple in nature. This is because each time a draw call is issued the CPU has to spend time instructing the graphics core about how to draw the object. The actual rendering may be extremely fast but the API overhead completely cripples performance.

In the same scenario described previously but using the 'instanced' approach, the application needs only to call a single API function `glDraw*Instanced` once. This then allows the application to draw the object `X` number of times. The instanced function behaves almost identically to `glDraw*` but takes an extra parameter, `primcount`, which tells the graphics core how many instances of the object it should render. This approach results in significantly more efficient behaviour.

In order to achieve optimal performance when implementing instancing, we recommended wherever possible to use a power of two instance divisor. The result of doing so is a reduction in the number of instructions required to stream the data to the unified shader cores (USCs), effectively eliminating a potential bottleneck.

6.3. PBO Texture Uploads

PBOs are Pixel Buffer Objects. They were introduced in OpenGL ES 3.0 and enable applications to map GL driver allocated textures into the applications address space. Once mapped, the application can then read from or write to the texture from the CPU.

6.3.1. Optimal Texture Updates with PBOs

PBOs can be used to reduce the number of memory copies required to transfer data to memory accessible to the graphics core. For example, if a very fast upload of texture data from file was required, a PBO could be created and directly load the contents of the file into this memory. However, if the file was loaded into application memory first and then copied into the PBO, there would be as many memory copies performed as a call to `glTexImage2D` would.

Transfer Queue (TQ) Tasks

If `glTexStorage` has been used to define the texture, transfer tasks for PBO writes will be kicked when `glTexImage` is called. Note that the PBO must be unmapped before any GL calls are issued for the texture. Failing to do so will result in an error. If `glTexStorage` hasn't been used, the transfer task will be deferred to the first draw call that uses the modified texture.

Note that even if there is already a copy of the texture in graphics memory, the driver will have to TQ copy the mapped region of the texture from twiddled graphics memory to the driver's PBO buffer.

If the application does not need to preserve the mapped region, it may specify the `GL_MAP_INVALIDATE_RANGE_BIT` access flag when calling `glMapBufferRange`. If the entire texture can be invalidated, then the application can use the `GL_MAP_INVALIDATE_BUFFER_BIT` flag.

6.4. Rogue Specific

6.4.1. Using glTexStorage2D & glTexStorage3D

`glTexStorage2D` & `glTexStorage3D` were introduced in OpenGL ES 3.0. They provide a mechanism to define immutable-format textures. These are textures where the format and dimensions of all levels cannot be altered after their creation. The main benefit of immutable-format textures is that they reduce the amount of validation the driver has to perform. Texture format validation is performed up front and only once for all texture levels.

6.5. VAOs, UBOs, Transform Feedback Buffers and SSBOs in OpenGL ES

This section provides a quick reference.

6.5.1. Vertex Array Objects (VAOs)

Encapsulates bound vertex state e.g. `glVertexAttribPointer`. Binding a VAO applies all of the encapsulated state to the global GL state. The ID Zero ('0') is reserved by GL to represent the default VAO. Always use VAOs when working with SGX or Rogue.

APIs

- **OpenGL ES 1.x & 2.0:** Extension (`GL_OES_vertex_array_object`), which is discussed in greater detail in the PowerVR Supported Extensions document.
- **OpenGL ES 3.x:** Core.

6.5.2. Uploading Uniforms (Uniform Buffer Objects)

Instead of uploading uniform data from client memory (e.g. `glUniformMatrix4fv` and `glUniform1f`), a Uniform Buffer Object (UBO) allows a developer to store uniform data in an OpenGL ES buffer object.

There are a few different approaches that can be used for supplying uniform data to shaders in OpenGL ES. The most optimal method will vary depending on the use case. Here are some general guidelines which can be considered when uploading shader uniforms to the graphics core:

- If there are only a small number of uniforms, then setting the uniforms directly – for instance through functions such as `glUniformMatrix4fv` and `glUniform1f` - is usually the most efficient approach. This saves the overhead incurred when using a buffer.
- If there are many uniforms that are changing in bulk, then a Uniform Buffer Object (UBO) is the most efficient method to use. Applications can map and un-map buffers to modify them. There are several slots available to bind UBOs, so an application may use more than one for each draw i.e. split static and dynamic data.
- If there are several uniforms changing in bulk at the same time, and a small number i.e. one or two uniforms changing at different frequencies, then the most efficient method is to split those uniforms out of the UBO and update them on their own separately from the rest of the buffer. This is because UBOs come with the general disadvantage of all buffers. Essentially if they are modified in any way then the driver will need to ensure that any previous operations are complete before updating the buffer, otherwise the entire buffer must be ghosted (copied).

In general it is recommended to use UBOs whenever they are suitable according to the guidelines above when deploying to platforms with Rogue graphics cores. That said, UBOs are not recommended when deploying to platforms with SGX graphics cores.

APIs

- **OpenGL ES 1.x:** Not exposed.
- **OpenGL ES 2.0:** Exposed (`IMG_uniform_buffer_object`).
- **OpenGL ES 3.x:** Core.

6.5.3. Transform Buffer Objects

These buffers are used for transform feedback. When these are bound, post-transform vertex data is automatically resolved to the buffer by the graphics hardware. The buffers can be written to/read by the graphics hardware, but cannot be accessed by the CPU. A developer should always use these buffers when working on Rogue, but they are not available for SGX architectures.

APIs

- **OpenGL ES 1.x & 2.0:** Not exposed.
- **OpenGL ES 3.x:** Core.

6.5.4. SSBOs – Shader Storage buffer Objects

SSBOs are similar to UBOs. For example, storage blocks are defined in GLSL and SSBOs are bound to SSBO binding points. Unlike UBOs, SSBOs:

- Can be written to by the graphics core.
- Can be used as compute kernel input/output.
- Can be much larger than UBOs (MBs instead of KBs).
- Have variable storage up to the range bound for the given buffer. The actual size of the array, based on the range of the buffer bound, can be queried at runtime in the shader using the length function on the unbounded array variable.

Although SSBOs have similar features to UBOs and transform feedback buffers, the flexibility comes at a cost. SSBO reads may be more costly than UBOs, as data is fetched from system memory like a buffer texture instead of being pre-loaded into shader registers as UBOs would. Unlike transform feedback buffers that are written to automatically when bound, SSBOs need to be written to explicitly in shader code. SSBOs are not available on SGX.

When using PowerVR Rogue graphics cores we recommended favouring UBOs and transform feedback buffers where possible, as they take optimal paths through the pipeline. SSBOs are best suited for draw indirect/dispatch compute indirect use cases.

APIs

- **OpenGL ES 1.x, 2.0 & 3.0:** Not available.
- **OpenGL ES 3.1+:** Core.

6.6. Synchronisation

The most efficient method for hardware to schedule tasks is vertex processing executing in parallel to fragment tasks. In order to achieve this, the application should aim to remove functions which cause synchronisation between the CPU and graphics core wherever possible. In OpenGL ES this includes synchronisation functions such as `glReadPixels`, `glFinish`, `eglClientWaitSync`, `glWaitSync`.

- On PowerVR hardware calling the function `glFlush` results in a `NOP`, i.e. no work is carried out.
- On PowerVR hardware calling the function `glFinish` flushes (kicks) all outstanding renders in a context, as per the OpenGL ES specification.
- On PowerVR hardware calling the function `eglClientWaitSync` flushes (kicks) all outstanding renders in a context. It then waits (blocks the calling thread or until x nanoseconds have passed) for the specified sync object to be signalled. In other words, it waits for the work to be completed.

- Calling the function `glClientWaitSync` results in similar behaviour to calling the function `eglClientWaitSync`.
- Calling the function `glWaitSync` results in similar behaviour to calling the function `eglClientWaitSync`, however currently the OpenGL ES specification does not support a timeout.

6.6.1. Multithreading in OpenGL ES

Synchronisation between OpenGL ES threads is done by `eglMakeCurrent`. It performs the following:

- It binds the supplied context to the current rendering thread and the supplied draw/read surfaces.
- If the calling thread already has the current rendering context, then all outstanding operations are flushed and the context is marked as no longer current.
- If the draw and read parameters are set to `EGL_NO_SURFACE` and context is set to `EGL_NO_CONTEXT` then the current context is released from the calling thread without assigning it to a new one.

Usually the driver has to flush all outstanding operations unless the currently bound context is released and then rebound. In this latter case, all outstanding operations have already been kicked. Also the driver has to wait for operations to finish if the context/surface pairs are broken up and paired up with a different context/surface e.g. surface kept context changed or context kept surface changed. In the case of releasing the current context and surfaces without assigning a new one, the driver has to flush all outstanding operations but doesn't need to wait for them. Therefore calls to `eglMakeCurrent` should be kept to a minimum.

Using multi-threaded rendering usually has no performance benefits, and in some cases it can lead to worse performance. For example, the worst use case is to frequently bind a single graphics context to different threads (using `eglMakeCurrent`). In this case the API calls have the same cost as a single threaded render as the API call submission is serialised. However there is the additional overhead of the context switch, which means that performance will be less optimal than a single threaded renderer.

For the best possible performance, rendering threads should be created at start up. A primary thread should be used for all rendering. Additional threads created with a shared context should only be used for shader compilation and buffer data upload. The number of background threads should be kept to a minimum, preferably one thread per-CPU core. Creating threads in excess will lead to unmaintainable, hard to debug code.

6.7. Frame-buffer Down Sampling

PowerVR hardware provides an efficient fast path for down sampling frame buffer attachments, which can be extremely useful for certain post-processing techniques such as bloom and screen space reflections etc. The OpenGL ES function `glFramebufferTexture2DDownsampleIMG` allows an application to attach a buffer to a frame buffer which is at a lower resolution than the frame buffer itself and have the hardware automatically down sample the attachment. All PowerVR hardware exposes at least a 2 x 2 downscale. Other down sampling modes may be available depending on the exact hardware, and it is possible for an application to query this information at runtime.

For further information on this extension please refer to the following Khronos extension page here.

6.8. Pixel Local Storage Extension

Graphics techniques such as deferred lighting are often implemented by attaching multiple colour render targets to a frame buffer object, rendering the required intermediate data, and then sampling from this data as textures. While flexible, this approach even when implemented optimally (see section 5.1) still consumes a large amount of system memory bandwidth, which comes at a premium on mobile devices.

OpenGL ES 3.x provides the extension `shader_pixel_local_storage(2)` which enables communication between fragment shader invocations which cover the same pixel location. This extension enables applications to store the intermediate per-pixel data on-chip, for example the "G-

Buffer" in a deferred lighting pass. This memory can be read from and written to by shader invocations at the same pixel location.

The extension enables tile based renderers such as PowerVR graphics cores to efficiently make use of tile memory, as these intermediate "buffers" are never allocated or written out to system memory, as they only exist in on-chip memory.

This extension is extremely beneficial for mobile and embedded systems, and using it correctly will result in a significant reduction in memory bandwidth usage. Most techniques (such as deferred lighting) that write intermediate data out to system memory and then sample from it at the same pixel location can be optimised using this extension.

For further information on this extension please refer to the following Khronos extension page here.

7. Vulkan Specific Optimisations

A Brief Introduction

Vulkan is a new generation graphics and compute API, which has been built upon components from AMD's Mantle API – which was donated to Khronos by AMD. Vulkan is a highly efficient, streamlined and modern API designed to take advantage of current and future device architectures. In addition Vulkan works on a wide variety of platforms such as desktop PCs, consoles, mobile devices and embedded devices.

Vulkan is designed from the ground up to take advantage of modern CPU architecture such as multi-core and multi-threaded systems. Rendering work can be spread over many logical threads - our Vulkan Gnome Horde demo in our SDK shows this aspect of the API very nicely.

The application does not necessarily need to implement multi-threading to take advantage of Vulkan, the API itself is very efficient which results in extremely low CPU overhead. Vulkan requires less work to be carried out by the CPU to instruct the graphics core, which significantly reduces CPU usage. For mobile devices this can reduce thermal output and power consumption.

Vulkan feels more akin to modern object orientated programming languages as the rendering state is packaged into easily manageable independent API objects, rather than one giant global state machine as found in OpenGL ES.

Tile based renderers (such as PowerVR devices) in particular can enjoy several advantages that Vulkan brings to the table such as:

- Explicitly declared dependencies – all dependencies are explicitly declared ahead of time by the application. This means the driver has the opportunity to execute the commands in the most optimal way. This work had to be done on-the-fly in OpenGL ES and would usually never be optimal as the driver has to guess. Importantly for tile based renderers the driver is able package the work into tiler and rasterizer tasks which are directly consumable by the underlying hardware.
- Fine grained synchronisation – applications have much finer control over synchronisation (compared to OpenGL ES) between objects and memory. This means that the driver can build a comprehensive dependency chain, which means only caches that absolutely need to be flushed are, and operations that absolutely need to be completed are waited on. This essentially means the graphics core can be utilised more efficiently as work can be scheduled in advance (i.e. tiler tasks) to get a head start while waiting on a fragment task.
- Render passes – the render pass object describes a render from start to finish. They disallow any operations that would definitely cause a mid-frame flush during rasterization, which could stall the graphics hardware. This also means that the graphics core can more effectively use on-chip storage due to the fact that intermediate frame buffer attachments that do not need to be stored are never written back to system memory. This can significantly reduce memory bandwidth and save power.
- Explicit render state – the Vulkan driver is aware of the entire render state through pipeline objects ahead of time, which means that shaders can be better optimised based on input/output and various fixed function states such as blending. Operations such as shader patching whereby code must be added to a shader to perform a certain graphical operation, such as alpha blending on PowerVR, can now be done in advance before the shader is compiled, because the entire render state is already known. This can improve the graphics cores efficiency and reduce hitching as this extra work no longer needs to be carried out at draw time.

7.1. Memory Types

Vulkan provides several memory types, the specification orders them based on their performance. An application should attempt to use the fastest memory types wherever possible. The list below contains the memory types and common use cases:

- The memory type `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` is recommended when allocating static device buffers (created once at initialisation) used to store data that is exclusively used by the device. This type of memory should be preferred as it will offer the best performance for the application, as it will provide the application with the fastest type of memory available.

- The memory type `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` is recommended when allocating buffers used to upload data to graphics device. The application should always prefer this memory type when implementing staging buffers which is the most optimal method for passing data to the graphics device. The memory allocation should avoid the flags `DEVICE_LOCAL`, `HOST_COHERENT` and `HOST_CACHED` when using this memory type for staging buffers.
- The memory type `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` is recommended when allocating buffers used to update per frame data, such as uniform buffers. This will provide the most optimal memory type available for performing coherent data transfers between the CPU and graphics hardware.

Note that the PowerVR driver does not currently support host cached coherent memory types, but this is subject to change in future driver releases.

Also note that all current PowerVR based platforms utilise Unified Memory Architecture (UMA), meaning that memory types are less critical than on a system employing a Discrete Memory Architecture (DMA). In addition buffers may be left mapped without negatively impacting performance.

Recommended Frequency of Allocation

Vulkan provides the function `vkAllocateMemory` to allocate memory objects. On PowerVR hardware we recommend that the function `vkAllocateMemory` is called as infrequently as possible. Memory allocations should likely be in the 10s of MB at a time and ideally called during initialisation. If the application requires finer grained allocations then the application should implement its own memory sub-allocation.

7.2. Pipelines

In Vulkan a pipeline object holds all of the graphics state information. They describe states such as:

- primitive type
- depth/stencil test
- blending
- which shaders to use
- vertex layout
- multi-sampling
- face culling
- polygon winding

Pipelines are created from a description of all shader stages and any relevant fixed function stages. This allows the shaders to be optimised based on their inputs and outputs, and eliminates expensive draw time state validation/error checking which is done when the object is created. This can increase performance and reduce hitching which removes unpredictability.

7.2.1. Pipeline Barriers

On PowerVR hardware Vulkan pipeline barriers are free if the application does not have to wait for work to be completed, and no data conversion is required.

The most efficient manner for the hardware to schedule tasks is vertex processing executing in parallel to fragment tasks. Therefore the application should aim to remove unnecessary pipeline dependencies and barriers wherever possible. Smart usage of sub-pass dependencies can help to avoid pipeline bubbles, where no work can be carried out. An example of this is sub-pass dependencies where the fragment stage is waiting on the vertex stage to finish.

In addition it can sometimes be beneficial to define a dependency by region, which allows for finer control over fence granularity. This is particularly the case if the barriers are being used in fragment stages to fence reads of an input attachment.

7.2.2. Pipeline Caching

Pipeline cache objects allow the result of pipeline construction to be reused between pipelines and between runs of an application.

When using the Vulkan API an application should always use pipeline caching wherever possible as this means the driver can compile subsequent pipelines much quicker. Ideally pipelines should be created at initialisation rather than during the main render loop. In addition using warm-up frames to cache pipeline objects before drawing is not needed as optimisation is carried out when the pipeline is created, rather than when it is loaded for use.

7.2.3. Derivative Pipelines

A pipeline derivative is a child pipeline created from a parent pipeline, where the child and parent are expected to share much commonality. The goal of derivative pipelines is that they are cheaper to create using the parent as a starting point, and that they are more efficient on either host or device to switch/bind between children of the same parent.

Currently the PowerVR graphics driver does not take advantage of derivative pipelines in Vulkan and therefore applications will see no performance benefits from using them. However it is advised to use Vulkan correctly (i.e. use derivative pipelines where applicable) so that the application can take advantage of the feature when future drivers supports it.

7.3. Descriptor Sets

In Vulkan the base binding unit is a ‘descriptor’ which represents a single binding, although descriptors are not bound individually. Instead they are grouped together into descriptor set objects, which are opaque objects that contain storage for a set of descriptors. Each descriptor set has a descriptor set layout which describes the resources (such as buffers and image resources – samplers) that will be bound when drawing. The descriptor set is bound before any drawing commands just like a vertex buffer or frame buffer.

7.3.1. Multiple Descriptor Sets

On PowerVR hardware using multiple Vulkan descriptors for a single draw call has minimal to no impact on performance, due to the driver being able to gather all descriptors on the graphics core. Therefore a scheme should be chosen that works the best for the particular use case in the application.

7.3.2. Pooled Descriptor Sets

On PowerVR hardware it is possible to allocate pooled descriptor sets in a fragmentation-less memory pool, depending on the how the descriptor sets and descriptor pool are constructed.

In order for the driver to perform a fragmentation-less allocate or free, all descriptor sets must be allocated with the same size and the descriptor pool must have the `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` flag set.

Note that if different sized descriptor sets are allocated the driver will fall back to a non-pooled memory scheme.

7.4. Push Constants

Vulkan provides a high speed path to modify constant data in pipelines known as push constants, which are expected to outperform memory-backed resource updates.

On PowerVR hardware push constants are guaranteed to be placed in fast constant buffers, which are located in on-chip memory. Also push constants have an efficient allocation mechanism in the command buffer, so an application should use push constants wherever it is appropriate.

Note that small statically indexed UBOs may also be placed in constant buffers, but this behaviour is not guaranteed.

7.5. Queues

The PowerVR driver exposes a universal graphics queue family (i.e. Vulkan graphics, compute & present) with two queues and a single separate sparse binding queue. The hardware will aim to parallelise work as much as possible depending on currently available resources such as overlapping vertex, fragment, and compute workloads depending on available resources. This can be verified by profiling the application with PVRTune.

While an application may use a single Vulkan queue, it may actually be beneficial to use multiple queues, since it is harder to accidentally serialise work (costing performance) across different queues.

7.6. Command Buffers

Vulkan command buffers are objects that are used to record various API commands, such as drawing operations and memory transfers into, which can then be subsequently submitted to a device queue for execution by the hardware. The advantage of this approach is that all of the hard work of setting up the drawing commands can potentially be done in advance and in multiple threads.

Vulkan provides two levels of command buffers:

- Primary command buffers, which can execute secondary command buffers, and which are submitted to queues
- Secondary command buffers, which can be executed by primary command buffers, and which are not directly submitted to queues.

7.6.1. Command Buffer Usage Flags

On PowerVR hardware we recommended that the command buffer usage flag is left unset (i.e. 0) unless the application requires specific behaviour. The following list contains the current usages flags that the API exposes and their appropriate usage cases:

- `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` – this flag informs the driver that the command buffer will be submitted multiple times. As a result the driver must perform a copy of the entire buffer after it is submitted to a queue. Applications should only set this flag if absolutely necessary.
  - Note that this method is faster than manually creating another identical buffer every frame. If the application requires this type of functionally, then we recommend that the flag is set on secondary command buffers, and primary command buffers are always rebuilt. This is because the copy of secondary command buffers is done when the secondary command buffer is recorded into the primary command buffer (i.e. `vkCmdExecuteCommands`) rather than when the buffer is submitted to a queue (i.e. `vkQueueSubmit`).
- `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` – this flag must be set if the command buffer is a secondary and it is executing inside a render-pass.
- `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` – currently this flag does not have any effect on how the driver processes the command buffer when submitted. However it is possible that future drivers may use this flag as a hint about how much time should be spent on compiling the command buffer. Therefore for maximum portability this should be set if the application intends to submit the command buffer only once.

7.6.2. Transient Command Buffers

Currently on PowerVR hardware there is no performance benefit gained from using Vulkan transient command buffers, however this subject to change with future driver updates.

7.6.3. Secondary Command Buffers

In Vulkan the use of secondary command buffers may benefit performance significantly. An application can build several secondary command buffers on separate threads i.e. preparing commands for the next frame, while the main thread is executing the primary command buffer. Once the secondary command buffers are ready by signalling through some sort of thread synchronisation, the work can be en-queued into the primary command buffer and the process can be repeated.

7.7. Render Pass

In Vulkan a render pass represents a collection of frame buffer attachments, sub-passes, and dependencies between the sub-passes, and describes how the attachments are used over the course of the sub-passes such as load & store operations. Render passes group together rendering commands that are all targeted at the same frame buffer into well-defined tasks – they represent the structure of the frame.

Render passes operate in conjunction with frame buffer objects. Frame buffers represent a collection of specific memory attachments that a render pass instance uses.

We advise that applications use as few render passes as possible, because changing render targets is a fundamentally expensive operation.

7.7.1. Sub-Passes

In Vulkan a sub-pass represents a phase of rendering that reads and writes a subset of the attachments in a render pass. Rendering commands are recorded into a particular sub-pass of a render pass instance.

Readable attachments are known as input attachments and contain the result of an earlier sub-pass at the same pixel location. An important property of input attachments is that they guarantee that each fragment shader only accesses data produced by shader invocations at the same pixel location.

On PowerVR hardware it is recommended to make use of sub-passes and sub-pass dependencies wherever appropriate, because the driver will aim to collapse them whenever possible. Meaning the sub-pass (i.e. attachments) will stay entirely in pixel local storage (on-chip memory) and will never be written out to system memory, thus significantly reducing transfers to system memory. This makes it a very efficient method for implementing a deferred lighting pipeline.

7.7.2. Load & Store Ops

Vulkan provides explicit control over load and store operations on frame buffer attachments. On PowerVR hardware the most optimal settings are defined below:

- Load operation – should preferably be set to either `VK_ATTACHMENT_LOAD_OP_DONT_CARE` or `VK_ATTACHMENT_LOAD_OP_CLEAR` wherever possible and is preferred over using the function `vkCmdClearAttachment`. This will inform the graphics driver that it does not need to load the buffers data from system memory, saving enormous amounts of bandwidth
- Store operation – should preferably be set to `VK_ATTACHMENT_STORE_OP_DONT_CARE`, unless the attachment is required for later use (i.e. preserve the data).

Note that in the rare situation where the graphics core enters smart parameter mode (SPM – occurs when the parameter buffer overflows, causing the driver to perform a partial render), a load/store operation will be performed to preserve the contents of the attachments; this operation must be carried out so that the state may be restored when rendering resumes.

7.7.3. Transient Attachments

Vulkan render pass objects may also contain transient attachments. This type of attachment can be read and written by one or more sub-passes but is ultimately discarded at the end of the pass which means the data is never written out to main memory saving valuable memory bandwidth.

Older PowerVR graphics drivers do not take advantage of Vulkan transient attachments - however it is advised that an application still makes use of this feature in Vulkan (i.e. use transient attachments), as devices with newer drivers will fully support this feature.

7.7.4. Optimal Number of Attachments

On PowerVR hardware it is recommended to have less than eight input and output frame buffer attachments across an entire render-pass.

7.7.5. Attachment Order

On PowerVR hardware the order of input attachments will not adversely affect performance. The driver will re-order the attachments as required, producing the optimal order for the graphics core when the render-pass is compiled.

7.8. MSAA

To perform MSAA optimally using Vulkan on PowerVR hardware, the application should use a lazily allocated MSAA frame buffer attachment, which will be used to render the scene to. The store operation flag should be set to `VK_ATTACHMENT_STORE_OP_DONT_CARE` as this tells the driver to discard the contents of this buffer after use to save memory bandwidth i.e. the buffer is not to be preserved. In addition there must be a second frame buffer attachment, which will be used to resolve multi-sampled image. This attachment should have the store operation flag set to `VK_ATTACHMENT_STORE_OP_STORE`.

This method ensures that the multi-sampled buffer is only allocated by the driver on-chip when required and the MSAA resolve is performed on-chip and only then written out to system memory.

```
// Multi-sampled frame buffer attachment that we render to.
attachments[0].format = swapChain.colorFormat;
attachments[0].samples = VK_SAMPLE_COUNT_2_BIT; // 2x MSAA
attachments[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
// No longer required after resolve, this will save memory bandwidth
attachments[0].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachments[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachments[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachments[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[0].finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

// The frame buffer attachment where the multi-sampled image will be resolved to.
attachments[1].format = swapChain.colorFormat;
attachments[1].samples = VK_SAMPLE_COUNT_1_BIT;
attachments[1].loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachments[1].storeOp = VK_ATTACHMENT_STORE_OP_STORE; // Store the resolved image
attachments[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachments[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachments[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[1].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

7.9. Image Layout

During image creation the initial image format does not affect performance in any way i.e. the format `VK_IMAGE_LAYOUT_UNDEFINED` or the format `VK_IMAGE_LAYOUT_PREINITIALIZED` may be used without any impact on application performance.

If the application is using an image as a specific attachment type to a frame buffer such as colour, stencil or depth etc, then the final image layout as defined in `VkAttachmentDescription` should be set to the appropriate optimal layout. This depends on the attachment usage such as `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` for use as a colour attachment.

8. Contact Details

For further support, visit our forum:
http://forum.imgtec.com

Or file a ticket in our support system:
https://pvrsupport.imgtec.com

To learn more about our PowerVR Graphics SDK and Insider programme, please visit:
http://www.powervrinsider.com

For general enquiries, please visit our website:
http://imgtec.com/corporate/contactus.asp