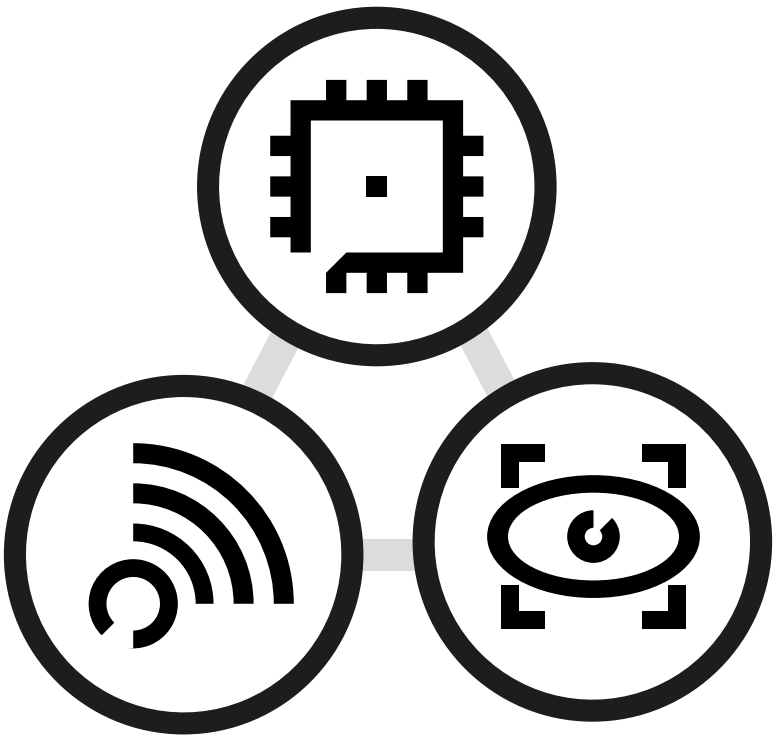
Introduction to PowerVR for Developers

Revision: 1.0
12/05/2021
Public



Published: 12/05/2021-19:19

Contents

1. Introducing PowerVR

This document provides developers with an overview of PowerVR, including the history, and details of the PowerVR graphics hardware architecture.

The PowerVR architecture is optimised for minimising memory bandwidth usage and reducing power consumption, while improving processing throughput.

PowerVR technology is developed and licensed by Imagination Technologies.

Important: It is assumed the reader is familiar with the 3D graphics programming pipeline of OpenGL, DirectX, Vulkan, or a similar 3D graphics programming API.

2. History of PowerVR

From the 80s to Present Day

A brief history of where PowerVR came from, and the journey to the present day

Imagination Technologies began life as VideoLogic back in 1985 working with video technology. During the 90s, the company branched into graphics under the PowerVR brand. The initial target was arcade machines, but before long a license was signed with NEC (now Renesas) to develop PC-based solutions. PowerVR technology began to appear in popular PC graphics cards such as the Matrox M3D, and Apocalypse 3Dx.

The desktop graphics market became very competitive during the late 90s. PowerVR was critically renowned for its quality and performance versus the more brute force approach applied by rivals. As a result, PowerVR won the coveted GPU slot in Sega's powerful and highly-regarded Dreamcast console. The PowerVR Series2 GPU used was the same as the Naomi arcade version, which had become widespread by this time. This made porting easier, so gamers could now enjoy playing many of their favourite arcade games at home.

In 1999, Videologic decided to re-brand as Imagination, to reflect the new focus on licensing IP.

In the early 2000s, Imagination went into partnership with ST MicroElectronics and Hercules, to make the 3D Prophet 4500 (Series3). Memory bandwidth had become a serious concern, but PowerVR's efficient architecture compensated well for this issue. As a result, this Kyro II-based card was able to outperform cards such as the GeForce2 in many areas, for a fraction of the price.

During the mid 2000s, Imagination changed direction to focus on mobile phones, correctly gambling that there was a real future in hardware 3D acceleration on

device. There was an early start in the first smartphones, such as the Nokia N95 and Sony Ericsson P1. However, PowerVR led the way as the GPU inside many of the groundbreaking all-screen devices that did away with the need for a stylus, and depended entirely on a smooth graphical interface.

The high performance with low power cost architecture meant it was naturally a PowerVR GPU (SGX - Series5) that could be found in the very popular PlayStation Vita handheld.

With such a solid reputation and continuous innovation, PowerVR technology began to crop up all over the mobile and embedded market. Amazon Fire tablets and sticks, early Samsung Galaxy phones, Samsung TVs, set top boxes, car instrument clusters and dashboard displays, smart ovens... the list continues to grow.

AR/VR, and automotive are just some of the new markets being targeted. Ray-tracing and neural networking are other areas where PowerVR is making impressive technological achievements.

3. Modern GPUs

How Does a GPU Differ From a CPU?

A modern System on Chip (SoC) often integrates both a Central Processing Unit (CPU) and a Graphics Processor Unit (GPU). They are designed differently depending on the type of data set they are more likely to be processing.

CPUs are optimised to execute large, heavily branched tasks on a few pieces of data at a time. A thread running on a CPU is often unique and is executed on its own, generally independent of all other threads. Any given processing element will process in just a single thread. Typical numbers of threads for a program on a CPU is commonly one to eight, up to a few tens at any period of time.

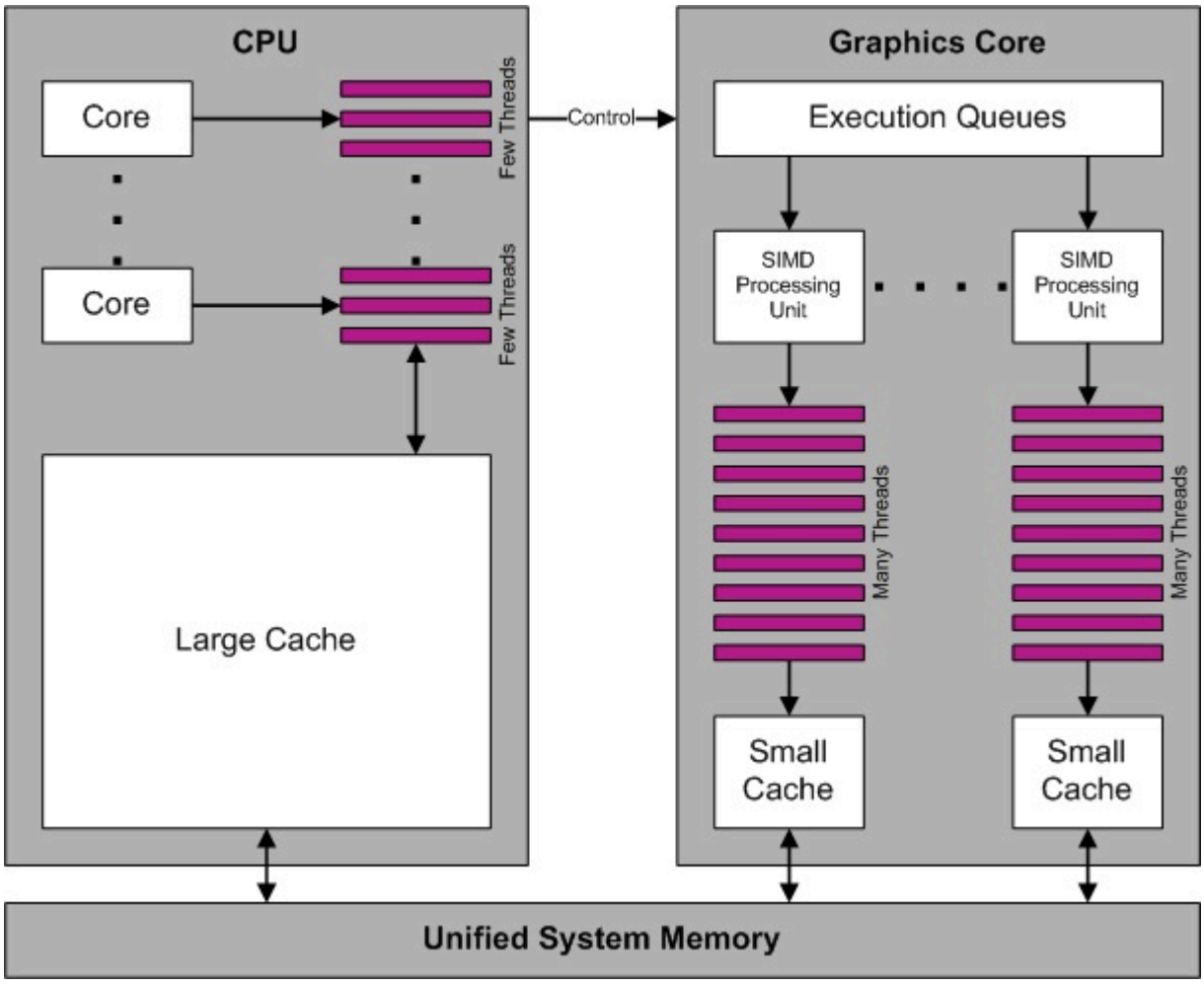
GPUs are optimised to work on the principle that the same piece of code will be executed in multiple threads, often numbering into the millions, to handle the large screen resolutions of today's devices. These threads differ only in input and normally follow the exact same execution steps.

Parallelism

Every graphics processor executes the same instruction on multiple threads concurrently, in the form of Single Instruction, Multiple Data (SIMD) processing.

The main advantage of the SIMD architecture is that significant numbers of threads can be run in parallel for a correctly structured application, and this is done with extremely high efficiency. SIMD architectures are usually capable of running many orders of magnitude more threads at once than a typical CPU.

SIMD is designed to operate on large coherent data sets and performs exceptionally well at this type of task. Algorithms that operate independently on a large coherent data set, such as graphics and image processing, are therefore well suited for this processor type.



Vector and Scalar Processing

Modern graphics core architectures feature multiple processing units which are either vector or scalar based. Both are supported by different versions of PowerVR architecture – Series 5 supporting vector, and Series 6, Series 7 and Series 8 supporting scalar.

- Scalar processing units operate on a single value per processing unit.
- Vector processing units work on multiple values per processing unit.

Vector

Vector processing can be very efficient, as the execution unit can work on multiple values at the same time rather than just one. For colour and vertex manipulation, this type of architecture is extremely efficient. Traditional rendering operations are, therefore, well suited to this architecture as calculations often operate on three or four elements at once.

The main drawback of vector architectures is that if scalar values or vectors smaller than the processor expects are used, the additional processing element width is wasted. The most common vector width is four, which means that a shader or kernel mainly operating on three component vectors will operate these instructions

with 75% efficiency. Having a shader that works on only one scalar at a time may take this number down to as low as 25%. This wastes energy and performance as parts of the processor are not doing any work. It is possible to optimise for this by vectorising code, but this introduces additional programmer burden.

Scalar

Scalar processors tend to be more flexible in terms of the operations that can be performed per hardware cycle, as there is no need to fill the additional processing width with data. Whilst vector architectures could potentially work on more values in the same silicon area, the actual number of useful results per clock will usually be higher in scalar architectures for non-vectorised code. Scalar architectures tend to be better suited to general purpose processing and more advanced rendering techniques.

4. PowerVR Architecture Overview

The PowerVR Advantage

PowerVR is the name of the graphics hardware IP family from Imagination Technologies. The ethos behind PowerVR has always been efficiency and technique, over brute force. All generations are based on Imagination’s patented Tile Based Deferred Rendering (TBDR) architecture. The core design principle of the TBDR architecture is to keep the system memory bandwidth requirements of the graphics hardware to a bare minimum.

As data transferred to and from system memory is the biggest cause of graphics hardware power consumption, any reduction made in this area will allow the hardware to operate at a lower power. Additionally, the reduction in system memory bandwidth use and the hardware optimisations associated with it, such as using on-chip buffers, enables an application to execute its render at a higher performance than other graphics architectures.

Due to the balance of low-power and high-performance, PowerVR graphics cores are dominant in the mobile and embedded devices market.

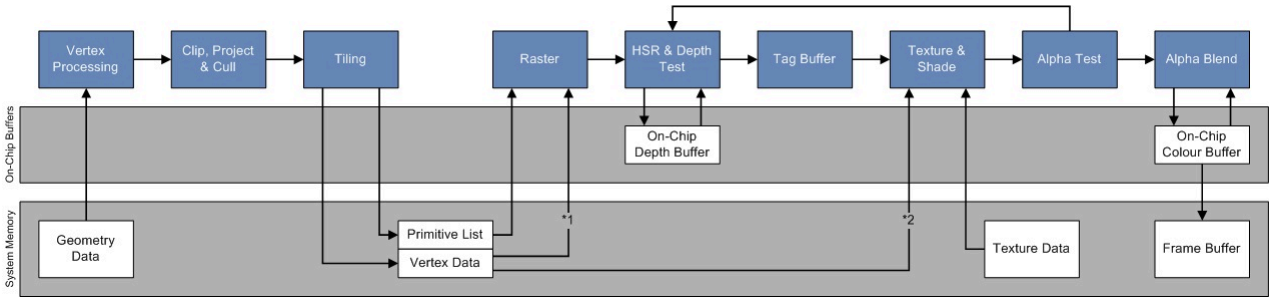
Tile-Based Deferred Rendering (TBDR)

The usual rendering technique on most GPUs is known as Immediate Mode Rendering (IMR) where geometry is sent to the GPU, and gets drawn straight away. This simple architecture is somewhat inefficient, resulting in wasted processing power and memory bandwidth. Pixels are often still rendered despite never being visible on the screen, such as when a tree is completely obscured by a closer building.

PowerVR's Tile-Based Deferred Rendering architecture works in a much smarter way. It captures the whole scene before starting to render, so occluded pixels can be identified and rejected before they are processed. The hardware starts splitting up the geometry data into small rectangular regions that will be processed as one image, which we call “tiles”. Each tile is rasterized and processed separately, and as the size of the render is so small, this allows all data to be kept on very fast chip memory.

Deferred rendering means that the architecture will defer all texturing and shading operations until all objects have been tested for visibility. The efficiency of PowerVR Hidden Surface Removal (HSR) is high enough to allow overdraw to be removed entirely for completely opaque renders. This significantly reduces system memory bandwidth requirements, which in turn increases performance and reduces power requirements. This is a critical advantage for phones, tablets, and other devices where battery life makes all the difference.

The diagram below illustrates the Tile-Based Deferred Rendering (TBDR) pipeline.

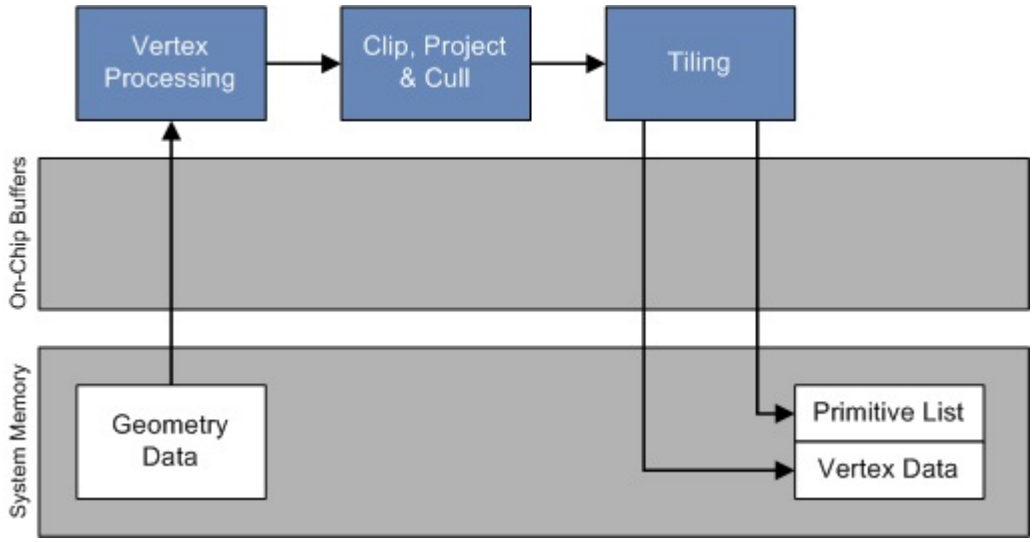


Generally, the parts up to and including "Tiling" are considered part of the Tiler, and the parts from "Raster" onwards are considered part of the Renderer. These are described in more detail on the following pages.

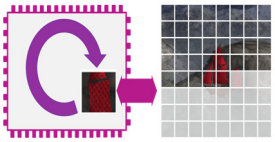
Vertex Processing (Tiler)

Every frame, the hardware processes submitted geometry data with the following steps:

1. The execution of application-defined transformations, such as vertex shaders (Vertex Processing).
2. The resulting data is then converted to screen-space (Clip, Project, and Cull).
3. The Tile Accelerator (TA) then determines which tiles contain each transformed primitive (Tiling).
4. Per-tile lists are then updated to track the primitives which fall within the bounds of each tile.



Each tile in the tile list contains primitive lists which contain pointers to the transformed vertex data. The tile list and the transformed vertex data are both stored in an intermediate store called the Parameter Buffer (PB). This store resides in system memory, and is mostly managed by the hardware. It contains all information needed to render the tiles.

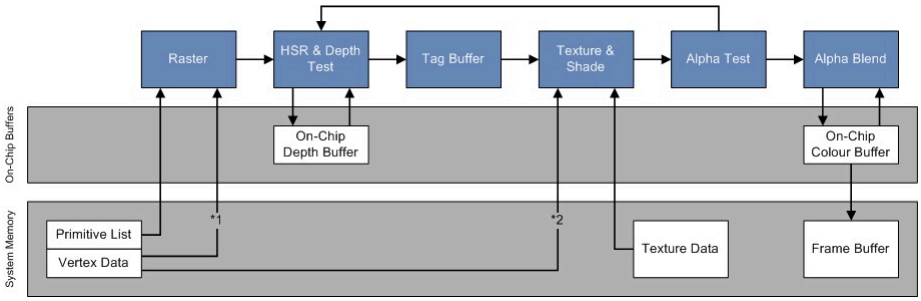


Per-Tile Rasterization (Renderer)

Rasterization and pixel colouring are performed on a per-tile basis with the following steps:

1. When a tile operation begins, the corresponding tile list is retrieved from the Parameter Buffer (PB) to identify the screen-space primitive data that needs to be fetched.
2. The Image Synthesis Processor (ISP) fetches the primitive data and performs *Hidden Surface Removal* (HSR), along with depth and stencil tests. The ISP only fetches screen-space position data for the geometry within the tile.
3. The Tag Buffer contains information about which triangle is on top for each pixel.
4. The Texture and Shading Processor (TSP) then applies colouring operations, like *fragment shaders*, to the visible pixels.
5. Alpha testing and subsequently alpha blending is then carried out.
6. Once the tile's render is complete, the colour data is written to the frame buffer in system memory.

This process is repeated until all tiles have been processed and the frame buffer is complete.



Further TBDR Details

On-chip buffers

Read-Modify-Write operations for the colour, depth and stencil buffers are performed using fast on-chip memory instead of relying on repeated system memory access, as traditional IMRs do. Attachments that the application has chosen to preserve, such as the colour buffer, will be written to system memory.

PowerVR shader engine

The PowerVR shader engine is based on a massively multi-threaded and multi-tasking approach. It is hardware-managed and load-balanced by using a data driven execution model to ensure the highest possible utilisation efficiency. This approach schedules tasks based on data availability, and enables switching between

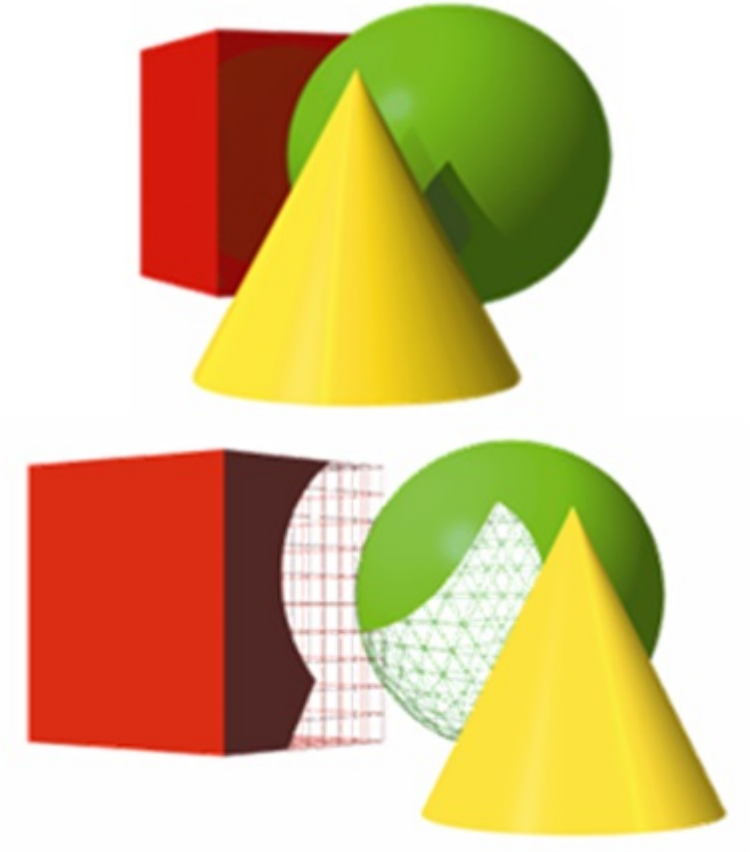
independent processing tasks to ensure that data dependency stalls are avoided at all costs.

Firmware

In many graphics architectures, hardware graphics events are handled on the CPU by the graphics driver. All PowerVR graphics cores are managed by *firmware*, enabling the graphics processor to handle the majority of high level graphics events internally. This approach keeps event handling latency to a minimum and reduces the graphics driver's CPU overhead.

Hidden Surface Removal Efficiency

Overdraw is where pixels are wastefully coloured when they will not contribute to the final image colour as they are overwritten by another object.

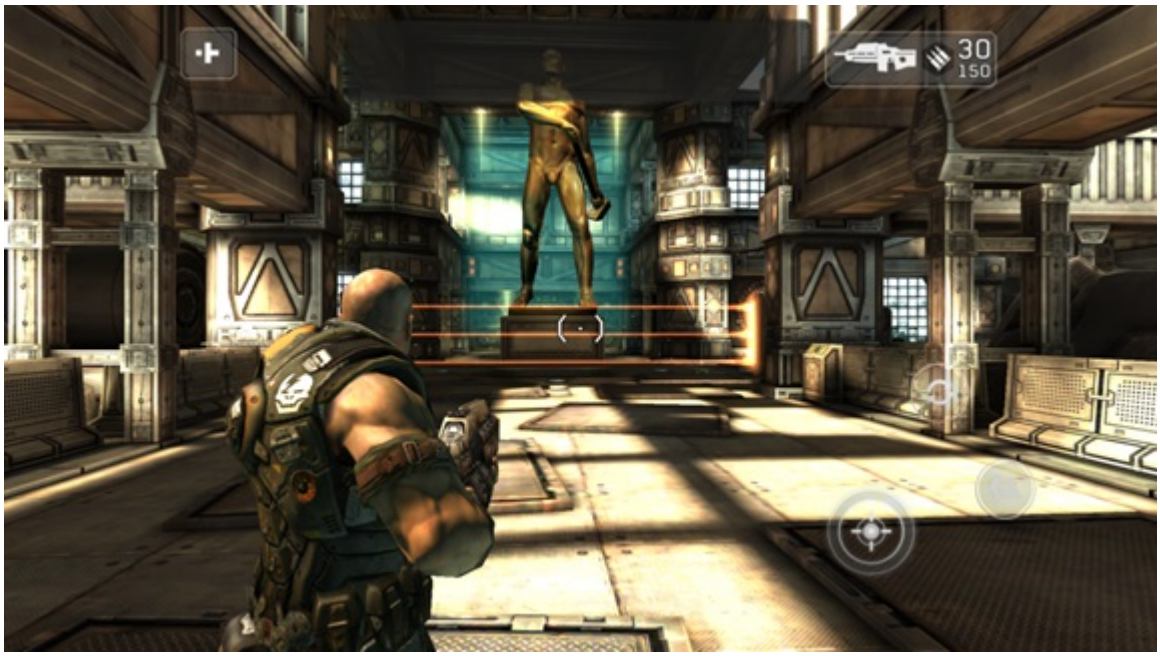


In a traditional IMR architecture, the scene shown above would cause green and red colours to be calculated for the sphere and cube respectively in the areas that are occluded by the yellow cone.

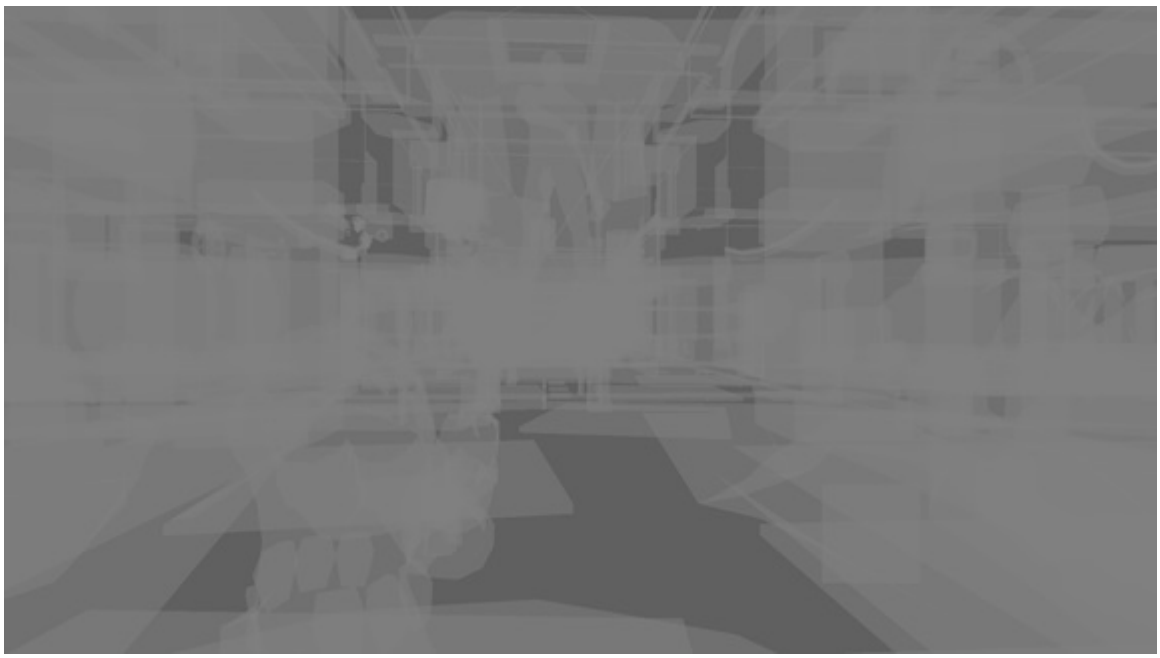
In architectures that include *early-Z* testing, an application can avoid some overdraw by submitting draw calls from front to back. Submitting in this order builds up the depth buffer so occluded *fragments* further from the camera can be rejected early. However, this creates additional burden for the application, as draws must be sorted every time the camera or objects within the scene move. It also does not remove all overdraw as sorting per-draw is very coarse – for instance, it cannot account for overdraw caused by object intersections. It also prevents the application from sorting draw calls to keep graphics API state changes to a minimum.

With PowerVR TBDR, Hidden Surface Removal (HSR) will completely remove overdraw regardless of draw call submission order.

The screenshot below is a capture from MadFinger Game's Shadowgun.



The image below highlights the amount of overdraw in the same scene, ignoring *Early-Z* or HSR optimisations that may be applied by a graphics core. The closer to white a pixel is, the more overdraw is present.



In this frame, 4.7 fragments are coloured on average per screen pixel.

The final image below shows the amount of "PowerVR overdraw" (post-HSR) for the same captured frame. On a PowerVR device, 1.2 fragments are coloured on average per screen pixel, which is 75% fewer fragments than the application submitted.

The render cannot achieve a 1:1 ratio between coloured fragments per screen pixel as the scene isn't completely opaque, because blended UI elements are contributing to the average.

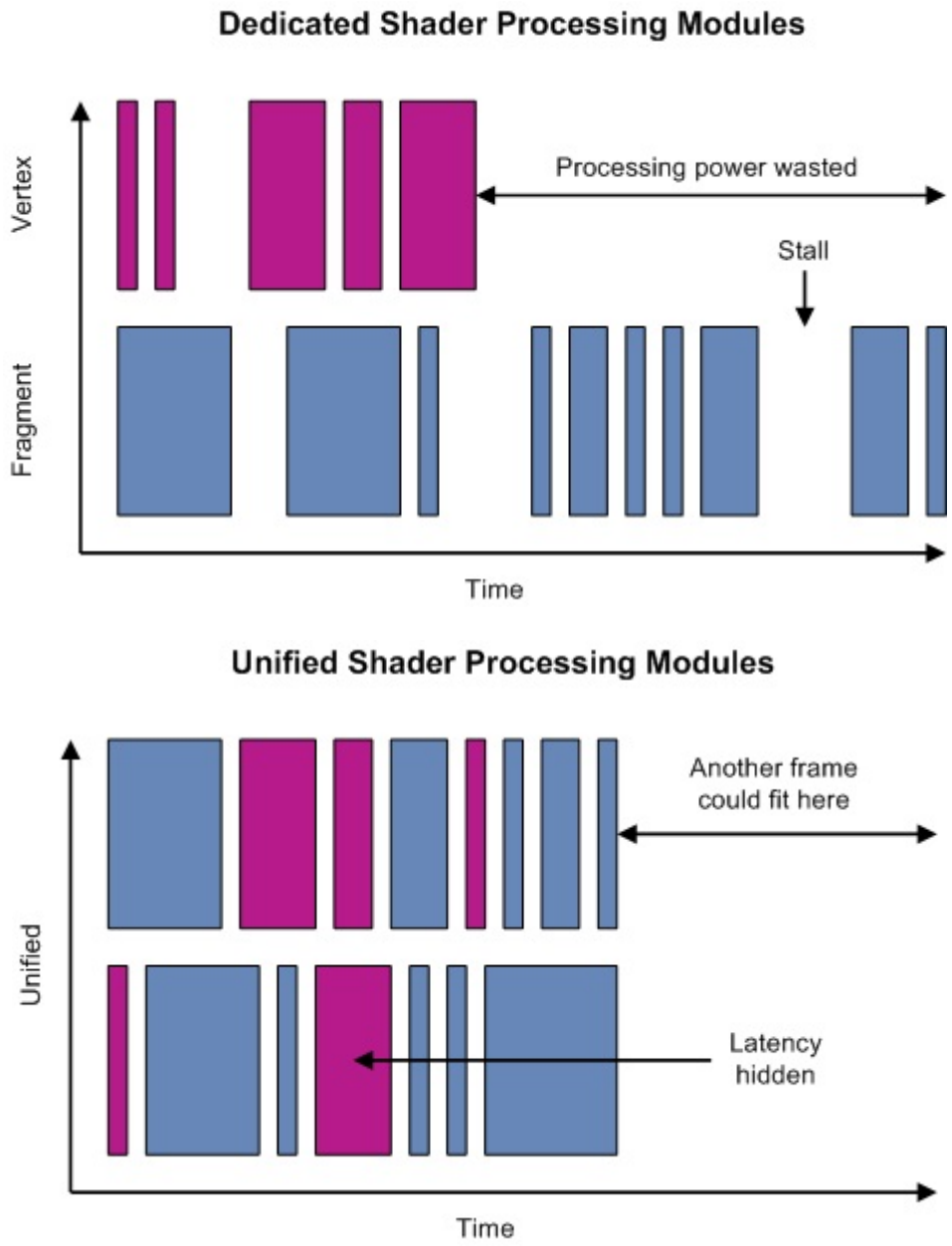
Unified and non-unified shader architectures

Shader architectures can be unified or non-unified. PowerVR has a unified shader architecture.

- A unified shader architecture executes shader programs, such as fragment and vertex shaders, on the same processing modules.
- A non-unified architecture uses separate dedicated processing modules for vertex and fragment processing.

Unified architectures can save power and increase performance compared to a non-unified architecture.

Unified architectures also scale much more easily to a given application, whether it is fragment or vertex shader bound, as the unified processors will be used accordingly.


Dedicated Shader Processing Modules
Vertex
Fragment
Processing power wasted
Stall
Time
Unified Shader Processing Modules
Unified
Another frame
could fit here
Latency
hidden
Time

5. Optimising for PowerVR

This section covers key principles to be followed to avoid critical performance flaws when developing graphics applications. These recommendations come from the combined experience of the PowerVR Developer Technology Support team and the developers they work with, through profiling and optimising their applications and games.

1. ***Do Understand the Target Device***
   - Seek to learn as much information about the target platforms as possible in order to understand different graphics architectures, to use the device in the most efficient manner possible.
2. ***Do Profile the Application***
   - Identify the bottlenecks in the application and determine whether there are opportunities for improvement.
3. ***Do Not Use Alpha Blend Unnecessarily***
   - Be sure Alpha Blending is used only when required to make the most of deferred architectures and to save bandwidth.
4. ***Do Perform Clear***
   - Perform a clear on a framebuffer's contents to avoid fetching the previous frame's data on tile-based graphics architectures, which reduces memory bandwidth.
5. ***Do Not Update Data Buffers Mid-Frame***
   - Avoid touching any buffer when a frame is mid-flight to reduce stalls and temporary buffer stores.
6. ***Do Use Texture Compression***
   - Reduce the memory footprint and bandwidth cost of texture assets.
7. ***Do Use Mipmapping***
   - This increases texture cache efficiency, which reduces bandwidth and increases performance.
8. ***Do Not Use Discard***
   - Avoid forcing depth-test processing in the texture stage as this will decrease performance in the early depth rejection architectures.
9. ***Do Not Force Unnecessary Synchronisation***
   - Avoid API functionality that could stall the graphics pipeline and do not access any hardware buffer directly.
10. ***Do Move Calculations 'Up the Chain'***
    - Reduce the overall number of calculations by moving them earlier in the pipeline, where there are fewer instances to process.

There are also a few other more minor *considerations* that can be found in the next section.

Do Understand the Target Device

Seek to learn as much information about the target platforms as possible in order to understand different graphics architectures, to use the device in the most efficient manner possible.

Manufacturers' websites for devices are a good place to look for specifications and they may also provide other helpful developer community resources. The PowerVR Graphics SDK provides public architecture and performance recommendation documents for reference:

- *PowerVR Architecture Overview*
- *PowerVR Series5 Architecture Guide for Developers*
- *PowerVR Performance Recommendations*
- *PowerVR Low Level GLSL Optimisations*
- *PowerVR Instruction Set Reference*

Note: Further PowerVR architecture documentation is *available from us* under a non-disclosure agreement.

Even after the graphics architecture is thoroughly understood, it is important to remember that other factors such as variations in CPU processing power, memory bandwidth, and thermal load will also impact an application's performance.

Do Profile the Application

Identify the bottlenecks in the application and determine whether there are opportunities for improvement.

It is important to understand where performance is bottlenecked before attempting to optimise an application. This ensures effort is not wasted, or visual quality is not sacrificed for minimal gains. If an optimisation is inappropriately applied to an area that is not bottlenecking performance, there may be no performance improvement. In some cases, an incorrectly applied optimisation may lead to worse performance.

From the PowerVR Developer Technology team's experience, we have derived the following list of common bottlenecks generally found in applications that have not been optimised, as ordered from most to least common:

- CPU usage
- Bandwidth usage
- CPU/graphics core synchronisation
- Fragment shader instructions
- Geometry upload
- Texture upload
- Vertex shader instructions
- Geometry complexity.

Profiling tools are vital in this process for developers to understand what is happening in their application, the hardware it is running on, and how and where bottlenecks are occurring. The PowerVR SDK includes the profiling tools *PVRTrace/PVRCarbon* and *PVRTune* to aid development on platforms powered by PowerVR hardware.

Do Not Use Alpha Blend Unnecessarily

Be sure alpha blending is used only when required to make the most of deferred architectures and to save bandwidth.

Disable alpha blending wherever possible. If transparent objects are required, keep the number of transparent objects to a minimum. The reasoning behind this is that deferred renderers, such as PowerVR graphics cores, calculate the visibility of fragments before the corresponding fragment shader is invoked to process it. This prevents invisible fragments in the output image being processed unnecessarily.

If alpha blending is enabled, then the hardware used to determine a fragment's visibility cannot be used. This is because the occluded (alpha-blended) fragment may impact the final rendered image. Due to this behaviour, enabling alpha blending eliminates the benefits of deferred rendering graphics architectures. This means the hardware is no longer able to make decisions about a fragment's visibility and drop it from the pipeline. This will likely result in overdraw which is where fragments are being processed that are not actually visible in the final image. Overdraw can negatively impact the application's performance, particularly if the application is already limited by rendering.

Do Perform Clear

Perform a clear on a frame buffer's contents to avoid fetching the previous frame's data on tile-based graphics architectures, which reduces memory bandwidth.

System memory accesses use more bandwidth and power than any other graphics operation. Keeping memory accesses to a minimum will reduce the chances of an application being memory bandwidth bound, and will also reduce the power consumption of an application.

Most applications need to generate a colour image at the end of the render, but have no need to preserve depth and stencil data between frames. Therefore, if frame buffer attachments do not need to be preserved at the end of a render, the appropriate frame buffer attachments can be invalidated to prevent them being written out to system memory.

Even fewer applications have a genuine need to upload the contents of the colour buffer's previous contents at the start of a new frame. Therefore, if the contents previously written to a frame buffer are not required, the driver can be informed not to load them from system memory to on-chip tile memory through a *clear* operation at the start of the render.

The net result of performing a clear and invalidating frame buffers will be a massive reduction in system memory bandwidth usage, and reduced power consumption.

In OpenGL ES® a clear can be performed by calling the `glClear` function at the beginning of a render. Additionally, the `glDiscardFramebufferEXT` or `glInvalidateFramebuffer` functions can be used to invalidate a frame buffer at the end of a render.

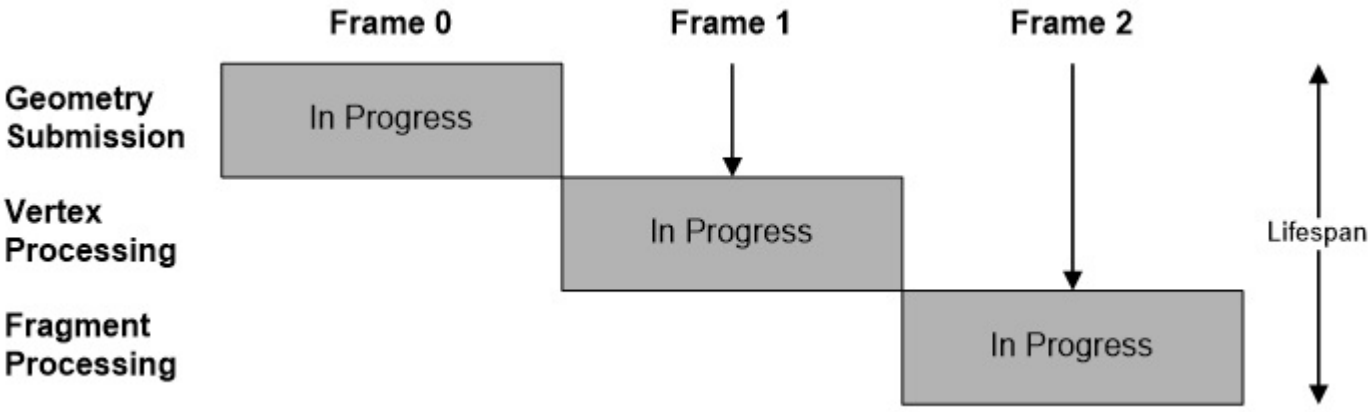
In Vulkan, the API gives explicit control over load and store operations on frame buffer attachments. When creating a frame buffer, set the load operation to either `VK_ATTACHMENT_LOAD_OP_DONT_CARE` or `VK_ATTACHMENT_LOAD_OP_CLEAR`. The store operation should preferably be set to `VK_ATTACHMENT_STORE_OP_DONT_CARE` unless the data requires preserving.

Do Not Update Data Buffers Mid-Frame

Avoid touching any buffer when a frame is mid-flight to reduce stalls and temporary buffer stores.

Modifying in-flight resources currently in use by the GPU such as vertex buffers and textures has a significant cost. Graphics processors tend to have at least one frame of latency to ensure that the hardware is always well-occupied with work. Therefore, altering a resource required by an outstanding render will usually result in one of the following actions being taken:

1. Stall in the buffer modifying API call until the outstanding render completes.
2. A new temporary buffer allocated for the new data, so the buffer modifying API call can complete without stalling the CPU thread.



As textures are generally accessed during fragment shading much later in the *graphics pipeline* than vertex attributes, the cost of a graphics driver stalling a texture modification is higher than modifying a vertex buffer. The driver may choose to avoid a stall entirely by creating temporary buffer stores (ghosting) which is good for performance, but it may not be desirable for applications that are already running out of buffer storage space.

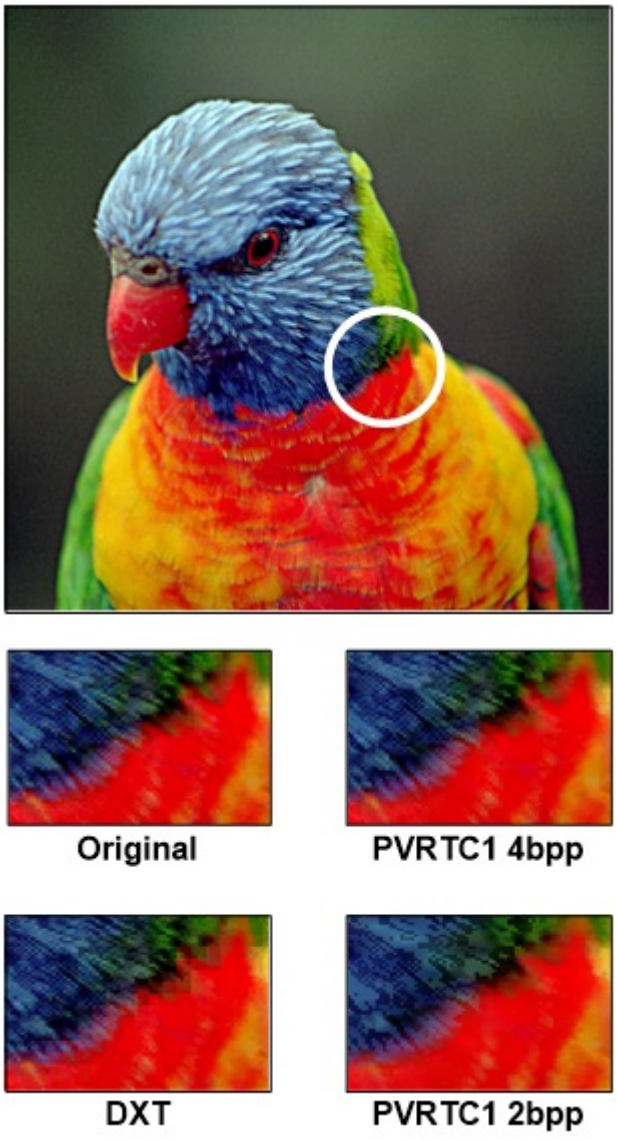
The stalling and ghosting behaviour of graphics processors varies between different GPUs and driver versions. For optimal performance, only modify vertex buffers and textures when absolutely necessary. If buffers must be modified, use application-side circular buffering so that the graphics processor can read from one buffer object while the application's CPU thread writes to another. This prevents the stalling and ghosting behaviours.

If the application is using the Vulkan graphics API, then it is the responsibility of the application developer to synchronise with the graphics processor. The appropriate mechanisms such as fences and semaphores must be put in place, to ensure that the application does not access a resource while the graphics processor is using it. This gives much more control over how and when resources are accessed, but comes at the cost of a more complex application as the driver will not safeguard against accessing data currently in use by the graphics processor.

Do Use Texture Compression

Reduce the memory footprint and bandwidth cost of the texture assets.

In some instances, it is worth considering the balance between texture size and texture compression. It may be possible to use a larger texture and a low-bitrate compression scheme and achieve a better balance of bandwidth savings and acceptable image quality.



Texture compression, not to be confused with image file compression, minimises the runtime memory footprint of textures. This provides several performance benefits, but primarily reduces the amount of system memory bandwidth consumed sending data to the graphics core.

PVRTC and PVRTCII are PowerVR specific compression technologies and will achieve best performance on the hardware, consuming as little as 2 bits per pixel. These textures are also very texture cache efficient as the lower pixel size allows more pixels to fit in the limited amount of cache memory available to the texture units.

Depending on the PowerVR generation and graphics API targeted, additional compressed texture formats may be supported, such as ASTC.

Do Use Mipmapping

This increases texture cache efficiency, which reduces bandwidth and increases performance.

Mipmaps are smaller, pre-filtered variants of a texture image, representing different levels of detail of a texture. By using a minification filter mode that uses mipmaps, the graphics core can be set up to automatically calculate which level of detail comes closest to mapping the texels of a mipmap to pixels in the render target. This means it can then use the right mipmap for texturing.

Using mipmaps has two important advantages:

1. It increases graphics rendering performance by massively improving texture cache efficiency, especially in cases of strong minification - the texture data is more likely to fit inside tile memory.
2. It improves image quality by reducing 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 be avoided. For example:

- Where filtering cannot be applied sensibly, such as for textures that contain non-image data such as indices or depth textures.
- Textures that are never minified, such as UI elements where texels are always mapped one-to-one to pixels.

Ideally mipmaps should be created offline using a tool like *PVRTexTool*, which is available as part of the *PowerVR Graphics Tools and 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, and developers must generate mipmaps manually.

Generation of mipmaps online will not work with compressed textures such as PVRTC, 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

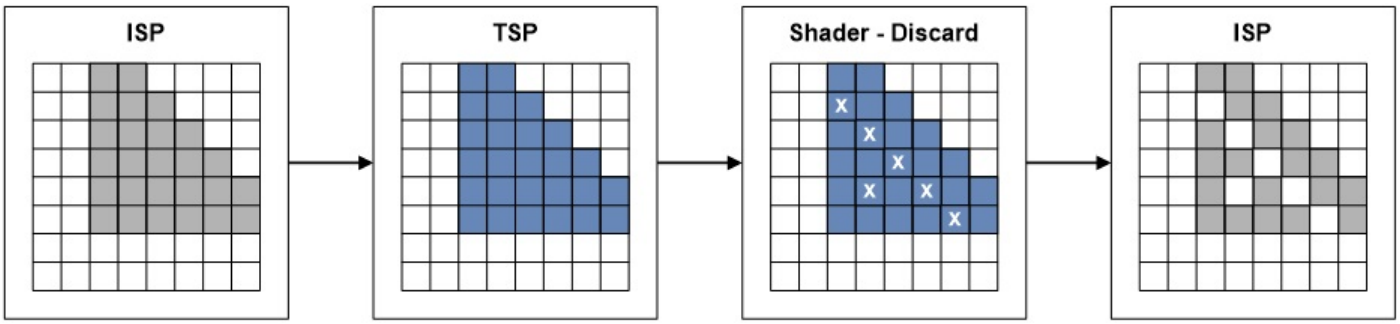
the runtime cost (and increased code complexity in the case of Vulkan) of generating mipmaps at runtime.

Do Not Use Discard

Avoid forcing depth-test processing in the texture stage as this will decrease performance in the early depth rejection architectures.

Applications should avoid the use of the discard operation in the fragment shader as using it will not improve performance. Most mobile graphics cores use a form of tile based deferred rending (TBDR) and using discard negates some of the benefits of this type of architecture. If possible, an application should prefer alpha blending over discarding.

Applications should also avoid alpha testing. When an alpha-tested primitive is submitted, early depth testing, such as PowerVR's Hidden Surface Removal (HSR), can discard fragments that are occluded by other fragments closer to the camera. Unlike opaque primitives which would also perform depth writes at this pipeline stage, alpha-tested primitives cannot write data to the depth buffer until the fragment shader has executed and fragment visibility is known. These deferred depth writes can impact performance, as subsequent primitives cannot be processed until the depth buffers are updated with the alpha tested primitive's values.



For optimal performance, consider alpha blending instead of alpha test to avoid costly deferred depth write operations. To ensure HSR removes as much *overdraw* as possible, submit draws in the following order:

1. Opaque
2. Alpha-tested
3. Blended.

Do Not Force Unnecessary Synchronisation

Avoid API functionality that could stall the graphics pipeline, and do not access any hardware buffer directly.

Graphics applications achieve the best performance when the CPU and graphics core tasks run in parallel. Graphics cores also operate most efficiently when the vertex-processing tasks of one frame are processed in parallel to the fragment-colouring tasks of previous frames. When an application issues a command that causes the CPU to interrupt the graphics core, it can significantly reduce performance.

The most efficient way for the hardware to schedule tasks is vertex processing executing in parallel to fragment tasks. To achieve this, the application should aim to remove functions which cause synchronisation between the CPU and graphics core wherever possible.

- In OpenGL ES - synchronisation functions such as `glReadPixels` , `glFinish` , `eglClientWaitSync` and `glWaitSync`.
- In Vulkan® - there is much finer control over synchronisation between resources, as any synchronisation between the graphics processor and CPU is defined by the developer.

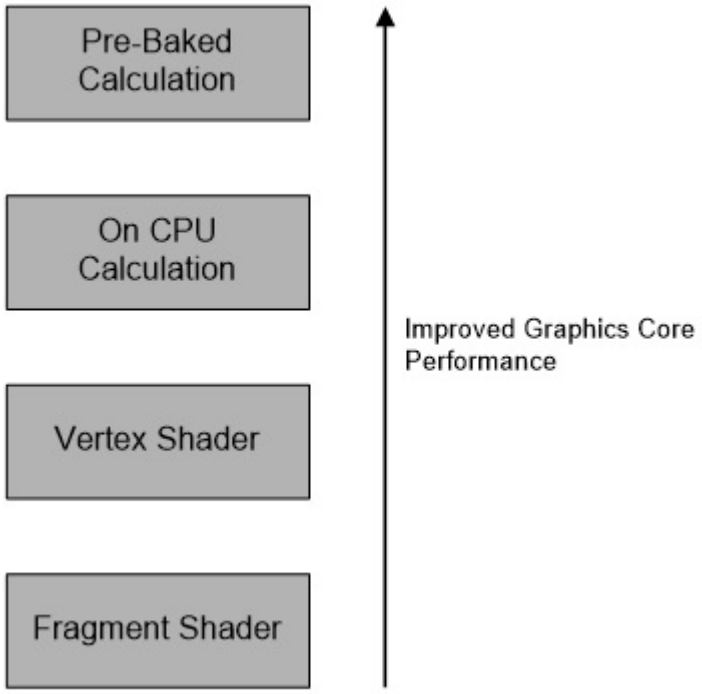
One of the most common causes of poor application performance is when the application accesses the contents of a frame buffer from the CPU. When such an operation is issued, the calling application's CPU thread must stall until the graphics core has finished rendering into the frame buffer attachment. Once the render is complete, the CPU can begin reading data from the attachment. During this time, the graphics core will not have write access to that attachment, which can cause the graphics core to stall subsequent renders to that frame buffer.

Due to the severe cost, these operations should only be used when absolutely necessary - for example to capture a screenshot of a game when a player requests one.

Do Move Calculations 'Up the Chain'

Reduce the overall number of calculations by moving them earlier in the pipeline where there are fewer instances to process.

By performing calculations earlier in the pipeline, the overall number of operations can be reduced, and therefore the workload can also be substantially reduced. Generally in a scene there are far fewer vertices than fragments that need to be processed. This means processing per vertex, instead of per fragment, would greatly reduce the number of calculations. One use case, for example, could be to perform per vertex lighting instead of per pixel lighting.



It is also possible to consider moving calculations off the graphics core altogether. Although the graphics core may be able to perform operations far more rapidly than the CPU can, it would be even faster for the CPU to perform an operation just once instead of the allowing the operation to be performed for many vertices on the graphics core.

To take the concept even further, consider performing calculations offline by baking values into the scene, effectively replacing expensive run-time calculations with a simple lookup. For example, replacing real-time lighting with light maps for static objects in a scene, such as terrain, buildings and trees can be a particularly effective compromise. This substantially improves performance, and in many cases provides higher quality lighting than would be possible to calculate at run-time.

Other Considerations When Optimising for PowerVR

Here are some other points which should be considered to improve application performance:

- ***Do Group per Material***

  Grouping geometry and texture data can improve application performance.

- ***Do Not Use Depth Pre-pass***

  Depth pre-pass is redundant on deferred rendering architectures.

- ***Do Prefer Explicit APIs***

  Graphical application made using explicit APIs tend to run more efficiently, if set up correctly.

- ***Do Prefer Lower Data Precision***

  Lower precision shader variables should be used, where appropriate, to improve performance.

- ***Do Use All CPU Cores***

  Using multi-threading in applications is key to efficient CPU use.

- ***Do Use Indexed Lists***

  Indexed lists can reduce mesh storage requirements by eliminating redundant vertices.

- ***Do Use Level of Detail (LoD)***

  Accounting for Level of Detail allows an application to improve performance while maintaining perceived graphical quality.

- ***Do Use On-chip Memory Efficiently for Deferred Rendering***

  Making greater use of on-chip memory reduces overall system memory bandwidth usage.

Do Group Per Material

Grouping geometry and texture data can improve application performance.

Modifying the GL state machine incurs CPU overhead in the graphics driver, as changes need to be interpreted and converted into tasks that can be issued to the graphics core. To reduce this overhead, minimise the number of API calls and state changes made by the application.

For geometry data, combine as many meshes as possible into a single draw call. Here is an example use case:

> **Grouping train seat meshes**
>
> Meshes for seats on a train use the same render and have static position and orientation relative to one another. The seats and the train could all be combined into a single mesh. To draw the train interior, several draw calls have merged into a single call.

PowerVR makes grouping easier

With the Hidden Surface Removal (HSR) feature on PowerVR hardware it is not necessary to submit geometry in depth-order to reduce overdraw. By freeing applications from this restriction they can focus on sorting draws by render state, ensuring state changes are minimised.

Textures can also be grouped

Similar to geometry data, it is possible to combine several textures into a single bindable object by using texture atlases or texture arrays where available. Textures can then be applied per object with the appropriate shader uniforms.

Think carefully about how to group objects to achieve the best performance

As discussed in *Do Not Update Data Buffers Mid-Frame*, modifying buffer data may stall the *graphics pipeline* or increase the amount of memory allocated by the graphics driver. When batching draws together, it is important to consider the update frequency of buffers. For example, batch spatially coherent objects with static vertex data into one vertex buffer, and objects with dynamic data, such as soft body objects like cloth, into another.

Do Not Use Depth Pre-pass

Depth pre-pass is redundant on deferred rendering architectures.

On graphics hardware that employs a deferred rendering architecture such as PowerVR, an application should not perform a depth pre-pass as there is no performance benefit. Performing this operation would be a waste of clock cycles and memory bandwidth. This is because the hardware will detect and remove occluded (opaque) geometry from the pipeline automatically during *rasterization*, before fragment processing begins.

Do Prefer Explicit APIs

Graphical applications made using explicit APIs tend to run more efficiently, if set up correctly.

Vulkan® is a new generation graphics and compute API. It is highly efficient, streamlined, and modern, and designed to take advantage of current and future device architectures. Vulkan works on a wide variety of platforms such as desktop PCs, consoles, mobile devices, and embedded devices.

Vulkan makes full use of modern CPUs

Vulkan is designed from the ground up to take advantage of modern CPU architectures such as multi-core and multi-threaded systems, and rendering work can be spread over many logical threads. The Vulkan “Gnome Horde” demo in the *PowerVR SDK* shows this aspect of the API very nicely.

Vulkan does require more work upfront

Vulkan is designed to have minimal driver overhead, but this comes at the cost of a more complex programming paradigm – explicit. In Vulkan, it is up to the application developer to handle low level details such as memory allocation for buffers and explicit synchronisation between resources. However, once the API is mastered, a Vulkan graphics application is likely to run much more efficiently and more predictably across various devices compared to legacy graphics APIs.

PowerVR and Vulkan

Our *PowerVR SDK* includes a Framework for developers targeting PowerVR platforms. This Framework reduces the need for boilerplate code, provides helpers, and much more, making Vulkan development much easier.

Do Prefer Lower Data Precision

Lower precision shader variables should be used, where appropriate, to improve performance.

Variables in shaders declared with the `mediump` modifier are represented as 16-bit floating point (FP16) values. Applications should use FP16 wherever appropriate, as it typically offers a significant performance improvement by theoretically doubling the floating point throughput over FP32 (`highp`). It should be considered wherever FP32 would normally be used, provided the precision is sufficient and the maximum and minimum values will not overflow, as visual artefacts may be introduced.

Do Use All CPU Cores

Using multi-threading in applications is key to efficient CPU use.

Modern mobile devices usually have more than a single CPU core. To achieve the best performance possible on modern CPU architectures, it is crucial that applications use multi-threading wherever possible.

For example, consider having graphics updates on the main thread, while having physics updates running on a separate worker thread. Splitting large chunks of work such as physics, animations, and file I/O over multiple threads enables the application to use the CPU more efficiently. This usually results in a smoother end-user experience.

If the application is targeting the Vulkan graphics API, it may be possible to split preparation of draw commands (building command buffers) over several threads.

Do Use Indexed Lists

Indexed lists can reduce mesh storage requirements by eliminating redundant vertices.

Vertex buffers enable the graphics driver to cache vertex data attributes, such as texture coordinates for mapping 2D images to the mesh, and model/space position. For static objects which have vertex attributes that change infrequently if at all, vertex buffers improve performance as the cached data can be reused to render many frames.

Index buffers allow vertex re-use for triangles that share an edge

In the example above, an index buffer is used in conjunction with a vertex buffer. Index buffers define the order in which elements of a vertex buffer should be accessed to represent the triangles in a mesh. Vertex attributes are written into the vertex buffer once, then referenced as many times as required to represent the triangles surrounding that vertex position. This means that index buffers improve performance and reduce the storage space requirements of complex mesh data.

PowerVR hardware is optimised for indexed triangle lists

For finely-tuned performance, vertex and index buffers should be sorted. This improves cache efficiency when the data is accessed by the GPU. Our 3D scene

exporter and converter tool, *PVRGeoPOD*, automatically applies sorting to mesh data when generating POD (PowerVR Object Data) files.

Do Use Level of Detail (LoD)

Accounting for Level of Detail allows an application to improve performance while maintaining perceived graphical quality.

Level of Detail is an important consideration for an application, the concept of 'good enough' should be employed here. Application developers must consider the usage of expensive graphics effects and high quality assets against the impact on performance.

Mipmapping is one form of LoD, which was discussed in *Mipmapping*. A second consideration for LoD is geometry complexity. An appropriate level of geometry complexity should be used for each object or portion of an object.

Inadequate consideration of LoD leads to wasted resources

The following are examples of a waste of compute and memory resources:

- Using a large number of polygons for an object that will never cover more than a small area of the screen, like a distant background object.
- Using polygons for detail that will never be seen due to camera angle, or culling – such as objects outside of the view frustum.
- Using large numbers of primitives for objects that can be drawn with much fewer numbers, with minimal to no loss in visual fidelity. As an example - using many hundreds of polygons to render a single quad.

Consider using shader techniques to reduce geometry complexity

Bump mapping can be used to minimise geometry complexity, but still maintain a high level of perceived detail. This is especially true for techniques such as reflection passes, where higher amounts of geometry may not be visible.

Do Use On-chip Memory Efficiently for Deferred Rendering

Making greater use of on-chip memory reduces overall system memory bandwidth usage.

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, still consumes a large amount of system memory bandwidth, which comes at a premium on mobile devices.

APIs have methods which allow efficient use of on-chip memory

Both OpenGL ES (3.x) and Vulkan graphics APIs provide a method to enable communication between fragment shader invocations which cover the same pixel location – through intermediate on-chip buffers. This buffer can only be read from and written to by shader invocations at the same pixel coordinate.

The GLES extension *shader_pixel_local_storage(2)* and Vulkan transient attachments enable applications to store the intermediate per-pixel data in on-chip tile memory. While each method has its own implementation details, they both provide similar functionally and both bring the same benefits. For example the "G-Buffer" attachments in a deferred lighting pass that are only needed once can be stored in tile memory, and then completely discarded when drawing is complete.

These features can potentially reduce the amount of system memory bandwidth used by deferred rendering

Both of the API features described above are extremely beneficial for tile-based renderers such as PowerVR graphics cores. The intermediate frame buffer attachments are never allocated or written out to system memory - they only exist in on-chip tile memory. This is extremely beneficial for mobile and embedded systems where memory bandwidth is at a premium.

Using these features correctly will result in a significant reduction in system memory bandwidth usage. Additionally, 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 these API features.

6. Glossary

A description of relevant graphical terms

Term	Meaning
ALU	Arithmetic Logic Unit. Responsible for processing shader instructions.
Early-Z	An umbrella term for a collection of optimisations commonly used by graphics cores. Early-Z techniques reduce overdraw by performing depth tests early in the graphics pipeline.
Firmware	A dedicated program running on the graphics core that handles hardware events. For example: a tile processing operation completing.
Fragment	The data necessary to calculate a pixel colour. Multiple fragments may contribute to the colour of a pixel. For example: when a transparent object is drawn in front of an opaque object.
Graphics pipeline	The sequence of processing stages within a graphics core that must be executed to render an image.
HSR	Hidden Surface Removal.
IMR	Immediate Mode Renderer.
ISP	Image Synthesis Processor.
Overdraw	The term “overdraw” refers to wastefully colouring pixels that do not contribute to the final image colour.
SIMD	Single Instruction, Multiple Data. Concurrent execution of a single instruction across multiple ALUs, where each ALU has unique input and output.
Scalar [shader architecture]	A shader architecture in which an ALU processes a single value at a time.
Pixel	The smallest addressable area of a frame buffer.
Rasterization	The process of determining which pixels a given primitive touches.
Render	The process of converting application-submitted data into coloured pixels that can be stored in the frame buffer.
Renderer	The tile processing stage of a TBDR pipeline. This includes rasterization and fragment shading.
TA	Tile Accelerator.
TBR	Tile Based Renderer.
TBDR	Tile Based Deferred Renderer.
Tile	A rectangular group of pixels. In TBR and TBDR architectures, the frame buffer is broken into many tiles. The tile size of each PowerVR graphics core is decided during hardware design, typically 32x32 pixels.
Tiler	The vertex shading, clipping, projection, culling, and tiling stages of a TBDR pipeline.
TSP	Texture and Shading Processor.

Term	Meaning
Vector [shader architecture]	A shader architecture in which an ALU processes multiple values simultaneously. Vector architectures commonly have a width of 4, allowing the ALU to calculate values for the ‘x’, ‘y’, ‘z’ and ‘w’ components of a vector data type.

7. Further Information

Over the years, there have been many generations of the PowerVR hardware family. All modern PowerVR generations are based on the Tile Based Deferred Rendering architecture outlined in this documentation. These are commercially available and actively targeted by 3D graphics developers.

For more information regarding the PowerVR hardware family, refer to the Imagination website:

https://www.imgtec.com/graphics-processors/

For more detailed information regarding the PowerVR hardware architecture, you may find what you need *here* or you can *contact us*. Some PowerVR architecture information is only available under NDA.

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

For general enquiries, please visit our website:

http://imgtec.com/corporate/contactus.asp