PowerVR Series5

Architecture Guide for Developers

Filename : PowerVR Series5.Architecture Guide for Developers

Version : PowerVR SDK REL_17.2@4910709a External Issue

Issue Date : 30 Oct 2017

Author : Imagination Technologies Limited

Contents

List of Figures

1. Introduction

The purpose of this document is to provide graphics programmers with an overview of the PowerVR Series5 (SGX and SGX-MP) graphics hardware architecture, while also highlighting why some performance recommendations make such a significant difference to graphics rendering on PowerVR platforms. Furthermore, the information in this document outlines the purpose of each hardware module for which PVRTune provides performance counters.

2. PowerVR Series5 Overview

The PowerVR architecture consists of several core modules that convert application submitted 3D data into a rendered image, as shown in Figure 1. These modules are the Tile Accelerator (TA), Image Synthesis Processor (ISP) and the Texture & Shading Processor (TSP). The core modules make use of the Universal Scalable Shader Engine (USSE) and Parameter Buffer (PB) shared components to keep the design as flexible and scalable as possible.

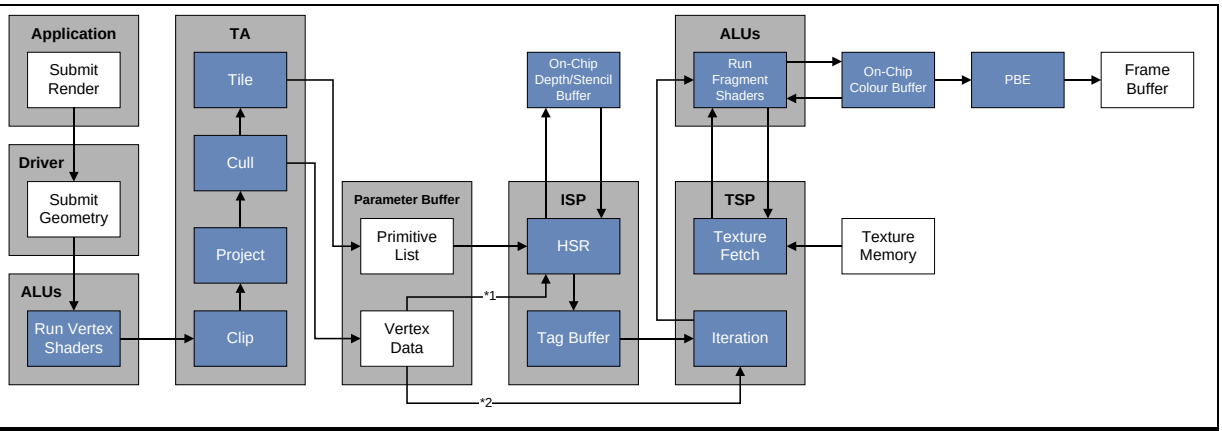

Figure 1. High level hardware overview

2.1. Universal Scalable Shader Engine (USSE)

The USSE is a flexible, multi-threaded processor capable of executing vertex, fragment and GP-Graphics Core instructions. As vertex and fragment processing tasks are completely decoupled (all geometry processing is done, then rasterization begins, as shown in Figure 2), the USSE's thread scheduler can automatically load balance a queue of tasks, which ensures idle time is kept to a minimum and latency is hidden as much as possible (Figure 3).

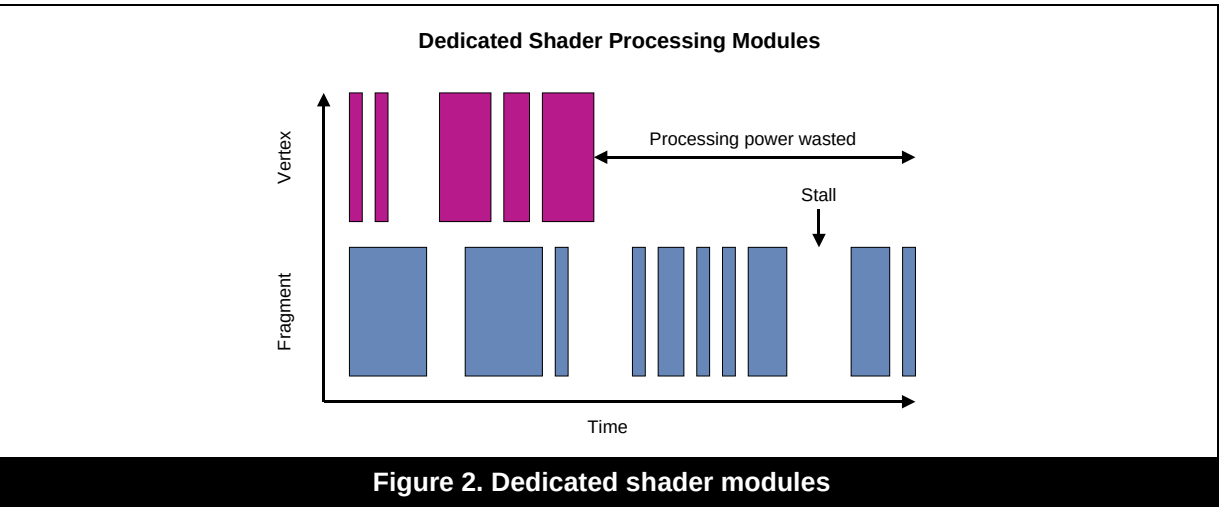

Figure 2. Dedicated shader modules

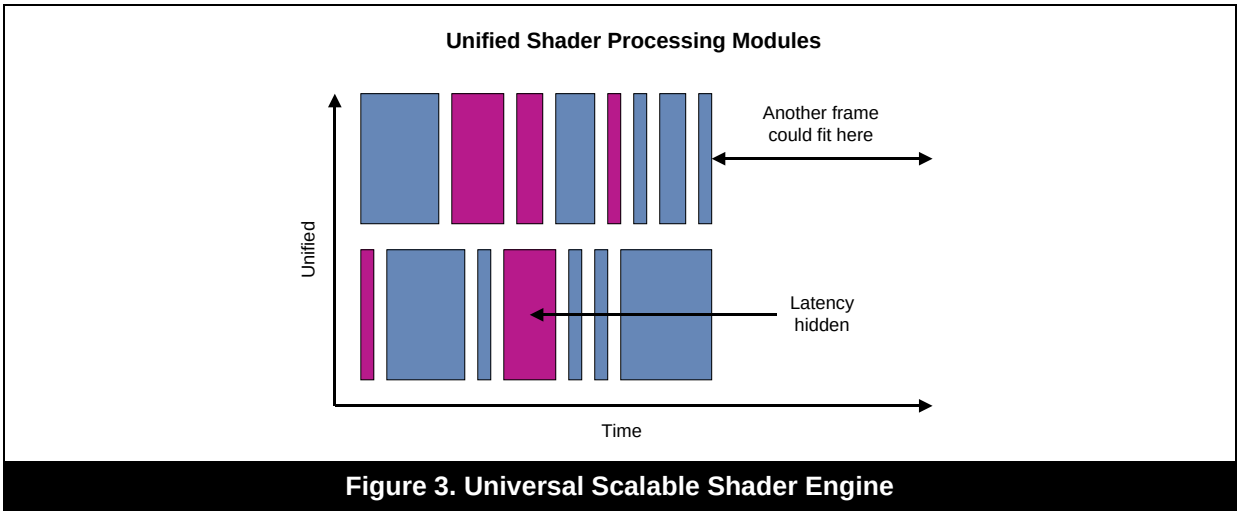

Figure 3. Universal Scalable Shader Engine

2.1.1. Coarse Grain Scheduler (CGS)

The CGS takes vertex, fragment and GP-Graphics Core jobs as input and breaks them down into tasks (smaller units of work) that can be submitted to available USSE execution units.

2.1.2. Thread Scheduling

Each USSE execution unit in a given SGX graphics core has its own thread scheduler. Each scheduler manages 16 threads, 4 of which can be active at a given time (Figure 4). The benefit of managing this many threads simultaneously is that when an active thread stalls, the scheduler can suspend it and schedule in a previously suspended thread with a zero cycle scheduling overhead. This approach keeps latency to a minimum as threads will still be processed while a stalled thread resolves. When an active thread completes a task, the thread scheduler will retrieve a new task from the CGS.

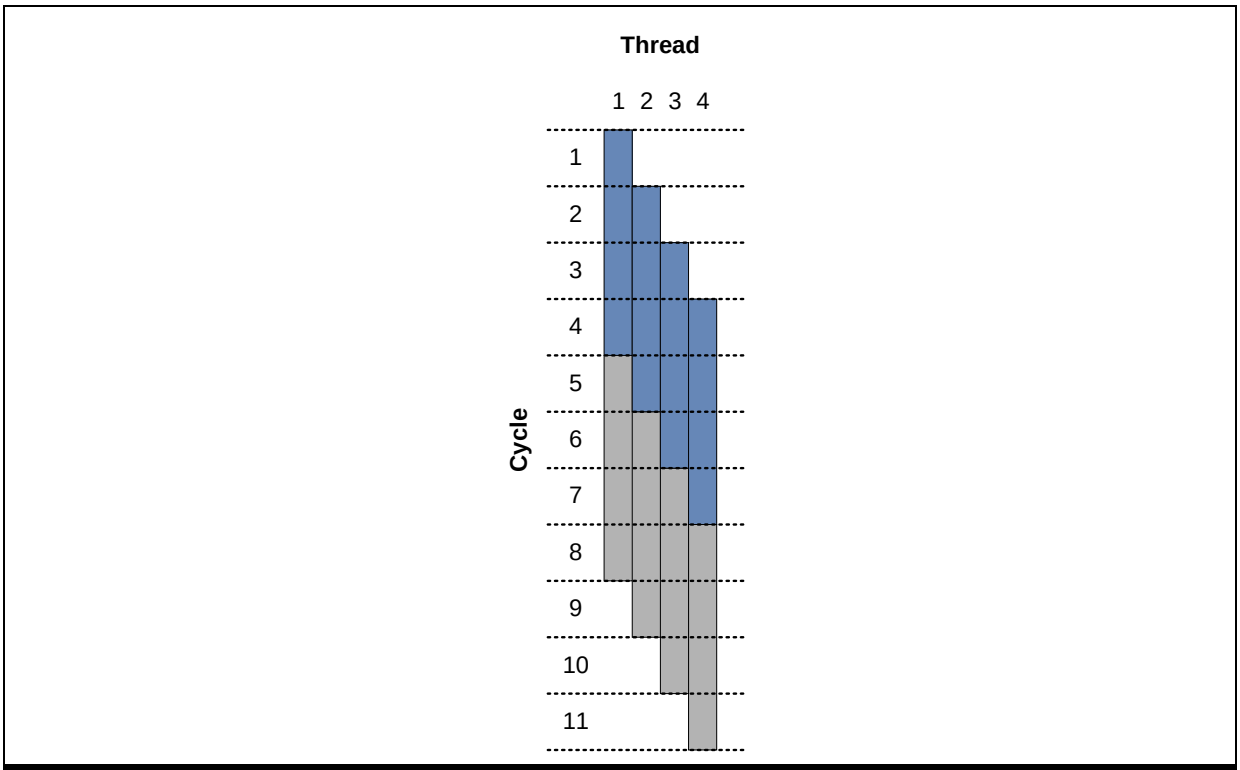

Figure 4. Dedicated shader modules

This efficient, hardware based, data driven thread scheduling can benefit applications by hiding the latency induced by any stalls, such as dependent texture reads and branching in shaders. Applications can take advantage of this by placing as much work as possible ahead of the point at which the shader is likely to stall, which will increase the number of instructions the hardware can use to mask latency induced by the stalls.

2.2. Parameter Buffer (PB)

The Parameter Buffer is an area of system memory that is used to store intermediate data (Figure 5), which allows the geometry processing and rasterization stages of rendering to be separated. The hardware and drivers handle this storage space entirely. The Parameter Buffer consists of Primitive Blocks and Display Lists.

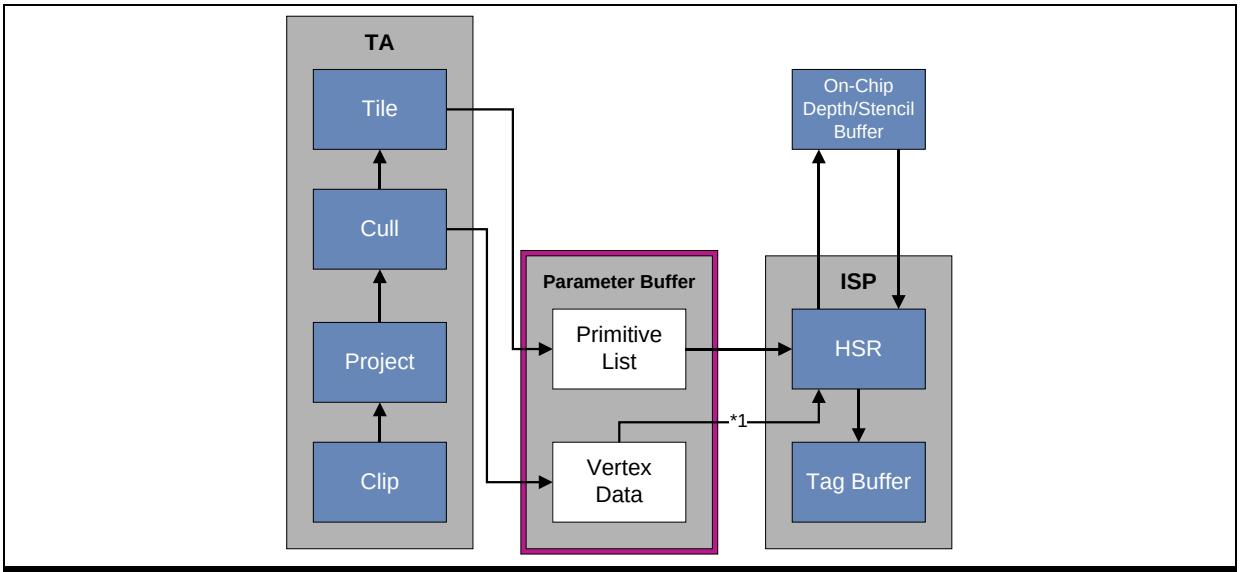

Figure 5. Parameter Buffer

2.2.1. Primitive Blocks

A Primitive Block is a list of primitives, where each primitive consists of its indices and the screen space x and y positions of its vertices. The data contained within a Primitive Block is used to reference transformed geometry data that has been clipped, projected and culled by the TA and written into Parameter Buffer memory. Display Lists are per-tile linked lists that reference the Primitive Blocks of objects that lie within the tile's boundaries.

2.2.2. Display Lists

To ensure that tiles minimise the amount of geometric data they have to fetch, Display Lists use primitive and vertex masks (e.g., for a triangle strip that goes beyond the boundaries of the tile, masks are used so the hardware will only fetch data for triangles that cover the tile).

2.3. Tile Accelerator (TA)

The TA takes geometric data that has been transformed by the USSE as input and clips, projects and culls it. In a shader based graphics API, such as OpenGL ES 2.0, the USSE executes vertex shaders on the geometry to transform and perform other per-vertex operations, such as lighting, before the resultant data is given to the TA. If polygons have been deemed visible after these tests, the TA updates all of the Display Lists of tiles that the object covers to reference it and writes out the transformed data into a Primitive Block (Figure 6).

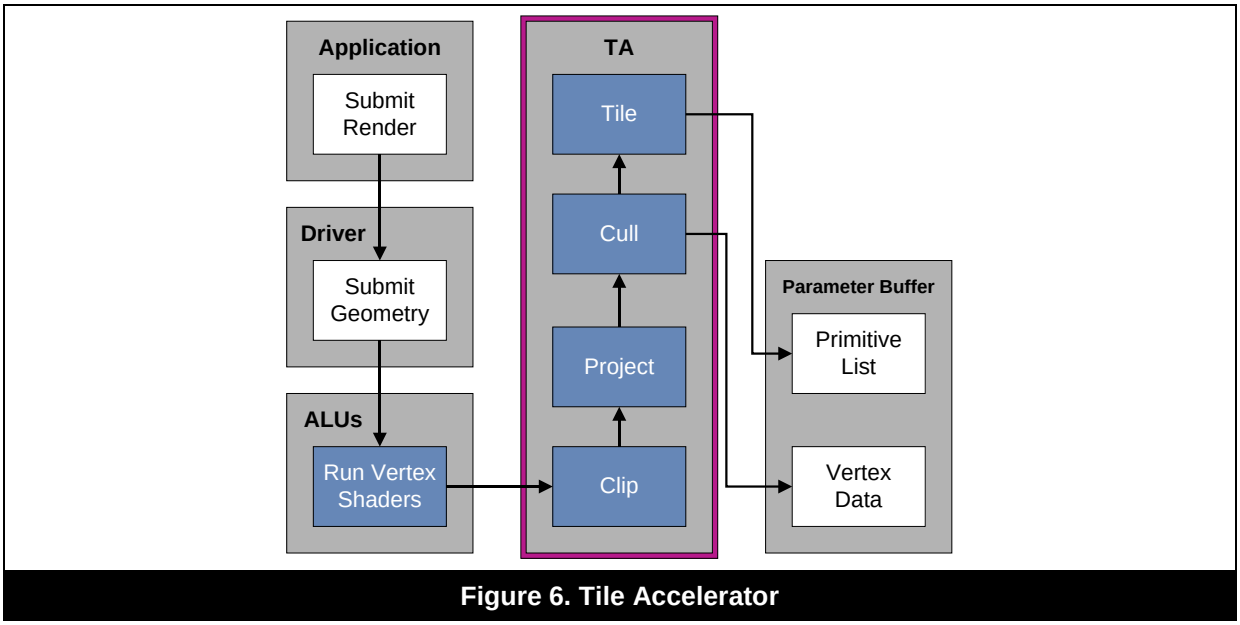

Figure 6. Tile Accelerator

The number of tiles required to complete the render is determined to be the resolution of the render pass divided by the tile size, where the tile size is a fixed value that is hardware specific (e.g., many SGX cores use a 16x16 pixel tile size). Larger tile sizes do improve performance (e.g., there will be fewer tiles to process, fewer Display Lists to update, single cycle scan-line depth tests can yield greater benefit, etc.), but they require the on-chip memory requirements of the graphics core to increase. The tile size used in a given graphics core is a balance between required performance and the cost of the additional resources.

2.4. Image Synthesis Processor (ISP)

The ISP (Figure 7) is responsible for per-tile HSR (Hidden Surface Removal) to ensure that the only fragments processed by the TSP (Texture and Shading Processor – see section 2.5) are those that will affect the rendered image. To do this, the ISP processes all of the triangles referenced in the tile's Display List one by one, calculating the triangle equation and, if depth testing is enabled, projecting a ray at each position in each triangle to retrieve accurate depth information for each fragment that the primitive covers within the tile. The calculated depth information is then compared with the values in the tile's on-chip depth buffer to determine if the fragments are visible. The ISP uses a Tag Buffer to track all visible fragments and the primitives that they belong to. To minimize data fetch, the ISP only retrieves position information for geometry that is required to render the tile, with vertex and primitive masks ensuring the smallest possible data set is retrieved. When all of the primitives that the tile's Display List references have been processed, the ISP submits fragments to the TSP in groups of fragments that belong to the same primitive, rather than submitting visible fragments on a per scan line basis. Doing so allows the hardware to improve the efficiency cache access and fragment processing.

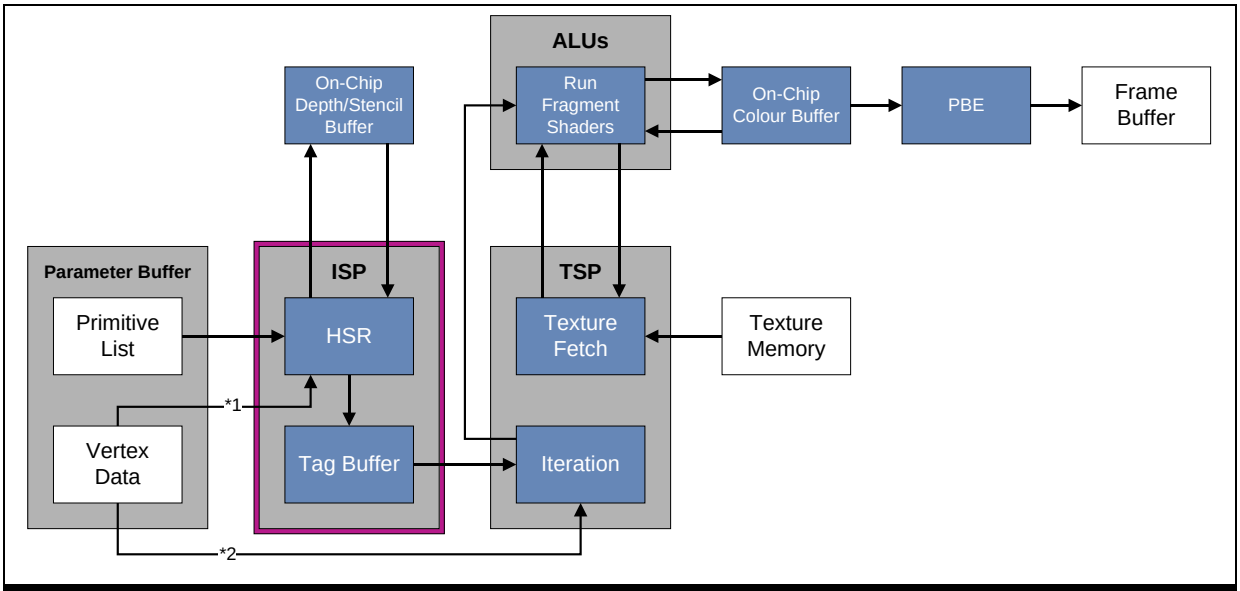

Figure 7. Image Synthesis Processor

Figure 8 and Figure 9 are two screenshots from Quake 3. The first screenshot is a normal screenshot from a scene in Quake 3 while the second renders everything translucently to get an idea of the overdraw in the scene. Although Quake 3 uses Binary Space Partitioning (BSP) and portal techniques to attempt to reduce overdraw, the scene complexity is such that overdraw is often quite high. As an example we measured overdraw of the Quake 3 demo001 at 3.39 (i.e., each pixel on the screen is drawn an average of 3.39 times). As the ISP can eliminate overdraw entirely for opaque pixels (i.e., those without any blending or alpha test/discard), TBDR hardware can reduce overdraw further than these BSP and portal techniques are capable of alone.

The benefit of the ISP, compared to Early-Z techniques in IMR and TBR architectures, is that overdraw in 3D applications can be significantly reduced independently of the submission order of draw calls, as the hardware has full visibility of the scene within the tile when tests are performed. In entirely opaque render passes, this technique can reduce overdraw to zero, meaning the number of fragments processed is the same as the resolution of the render's viewport. Unlike Early-Z techniques used in other architectures, this approach does not require applications to sort and submit geometry in front to back order to get the most out of the technique. Because of this, developers have no need for per-frame CPU sorting algorithms to further reduce overdraw and it also gives developers the freedom to submit their geometry in other ways that can benefit the render, such as batching draw calls by render state.

Figure 8. Quake 3 arena screenshot

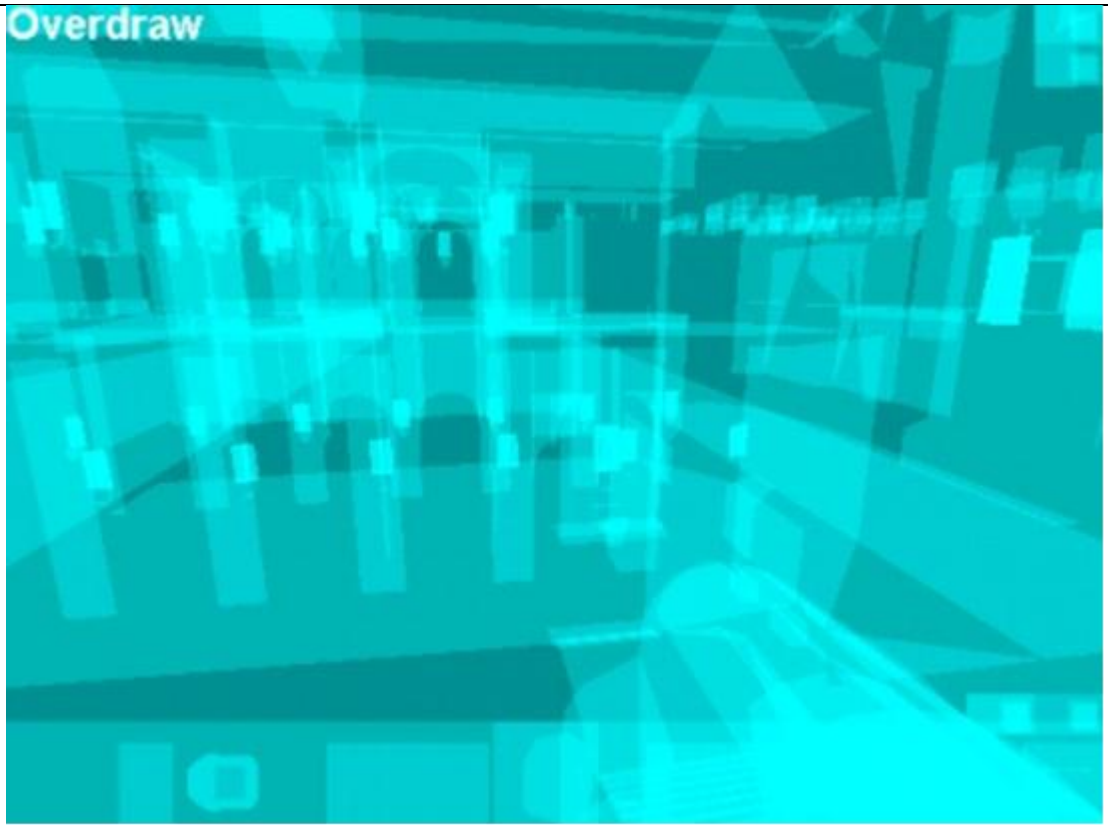

Figure 9. Quake 3 overdraw

2.5. Texture and Shading Processor (TSP)

Rather than texturing and shading fragments itself, the job of the TSP is to schedule fragment processing tasks (performed by the USSE), iterations, and texture data pre-fetch (Figure 10). If texture coordinates are calculated during iteration (e.g., texture coordinate varyings used in GLSL shaders), they can be used to pre-fetch texture data, which ensures the data is available when the USSE comes to process the fragments. Once the iteration has been performed, the TSP can submit groups of visible fragments to the USSE via the CGS to be textured and shaded (visible fragments are grouped by the ISP during the HSR process). When using a shader based graphics API, such as OpenGL ES 2.0, fragment shaders are executed at this stage to calculate fragment colour.

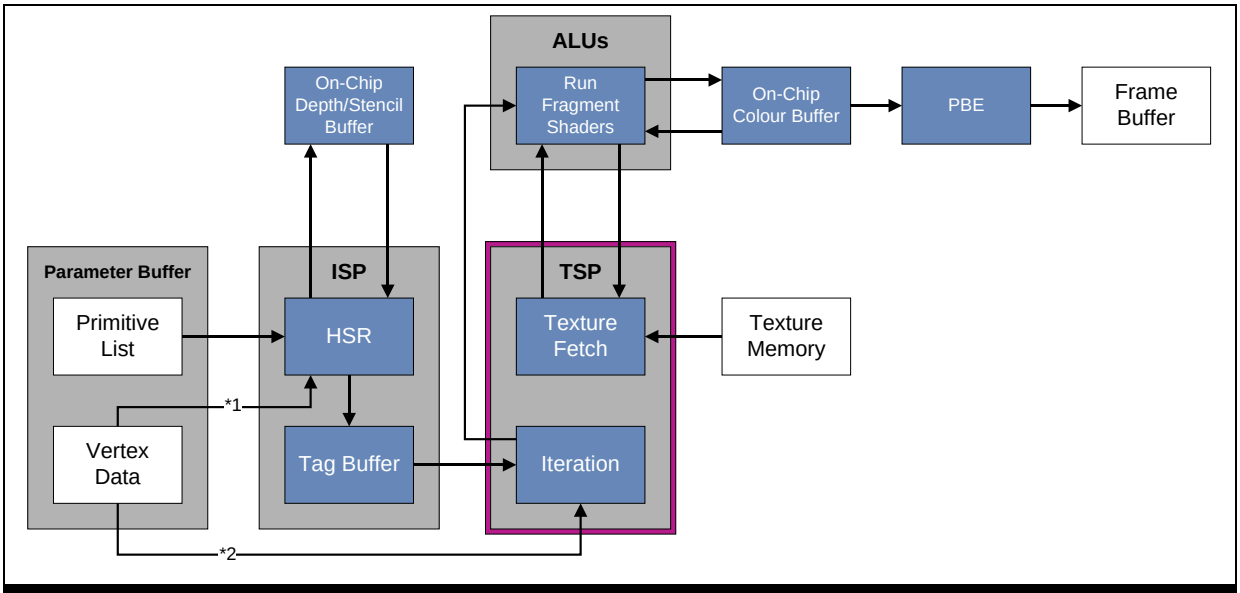

Figure 10. Texture and Shading Processor

2.6. SGX Micro Kernel

The Micro Kernel is specialised software that the hardware runs on the available USSE general purpose execution units. Unlike many other graphics architectures, the Micro Kernel enables the Graphics Core to handle internal interrupt events generated by the Graphics Core entirely on the Graphics Core. By utilising the Micro Kernel in this way, the graphics core has minimal impact on CPU load, performance is kept as high as possible (increased parallelism between the CPU and Graphics Core) and synchronisation issues between the CPU and Graphics Core are decreased, as highlighted in Figure 11.

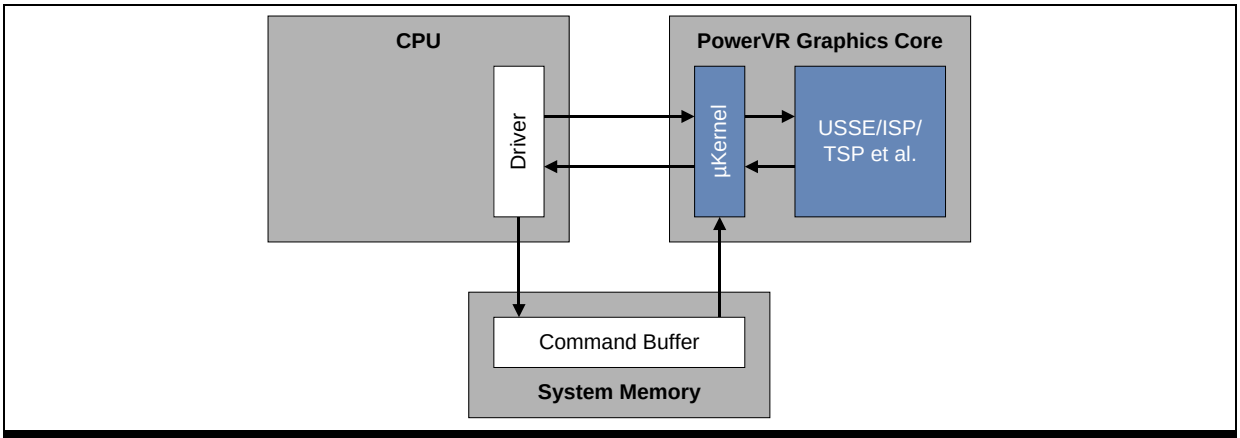

Figure 11. SGX and SGX-MP Micro Kernel

2.7. SGX-MP

The SGX-MP architecture is designed to scale as close to linearly as possible as more graphics cores are used. In typical real-world performance conditions (running well known game engines, graphics applications, benchmarks, etc.), each additional core runs at 95%+ the efficiency of a single core. Additionally, adding another core to the system only increases the overall memory bandwidth for a frame by <1%.

In SGX-MP devices, the hardware and drivers have total control over the multi-core logic. For this reason, applications that already run on SGX devices will be able to run without modification on SGX-MP devices.

When processing geometry, the hardware splits objects into chunks of work that can be distributed evenly between all available graphics cores. By doing this, the hardware can ensure that idle time in the cores is kept to a minimum.

During the rasterization, texturing and shading process, each additional graphics core allows another tile to be processed in parallel, as shown in Figure 12. To enable this, the hardware manages a queue of all tiles that need to be processed and each graphics core fetches and processes a tile from this queue. By taking this approach, the hardware keeps idle time to a minimum and avoids creating hotspots that could occur with static tile distribution (e.g., one core may have more work than others, which would cause the other cores to go idle). A tile task submitted to a graphics core includes HSR, texturing and shading.

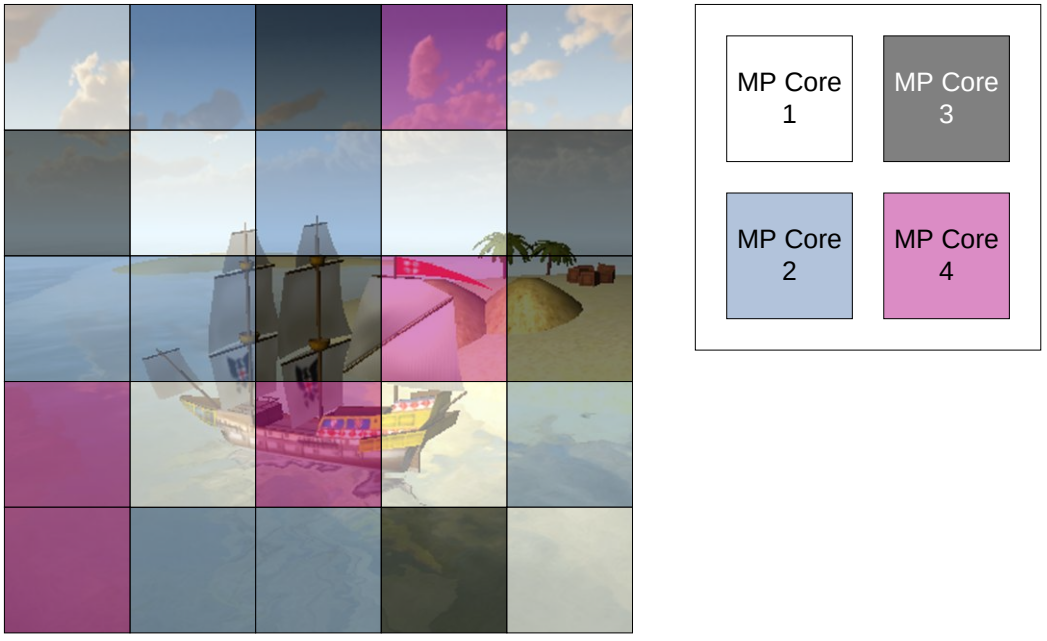

Figure 12. SGX-MP example tile distribution

3. SGX Hardware Schematic

The first stage of the processing is for the driver to submit geometry data from system memory to the graphics core's Vertex Data Master. This module takes the stream of vertex data and submits vertex processing jobs to the CGS, which in turn breaks down the jobs into tasks (smaller units of work) that can be distributed to the Thread Schedulers of the available USSE Execution Units. The USSE Execution Units then process the vertex tasks that have been given to it and the resultant transformed data is given to the TA. The TA applies clipping, projection and culling to the geometric data and, if it passes these tests, the data will be written into a Primitive Block in Parameter Buffer memory. The TA also updates the Display Lists of all tiles the object covers to reference the object's Primitive Block. As the Display Lists are updated, the TA creates vertex and primitive masks to restrict the amount of geometry data that will be fetched when per-tile operations are performed. Figure 13 illustrates these interactions.

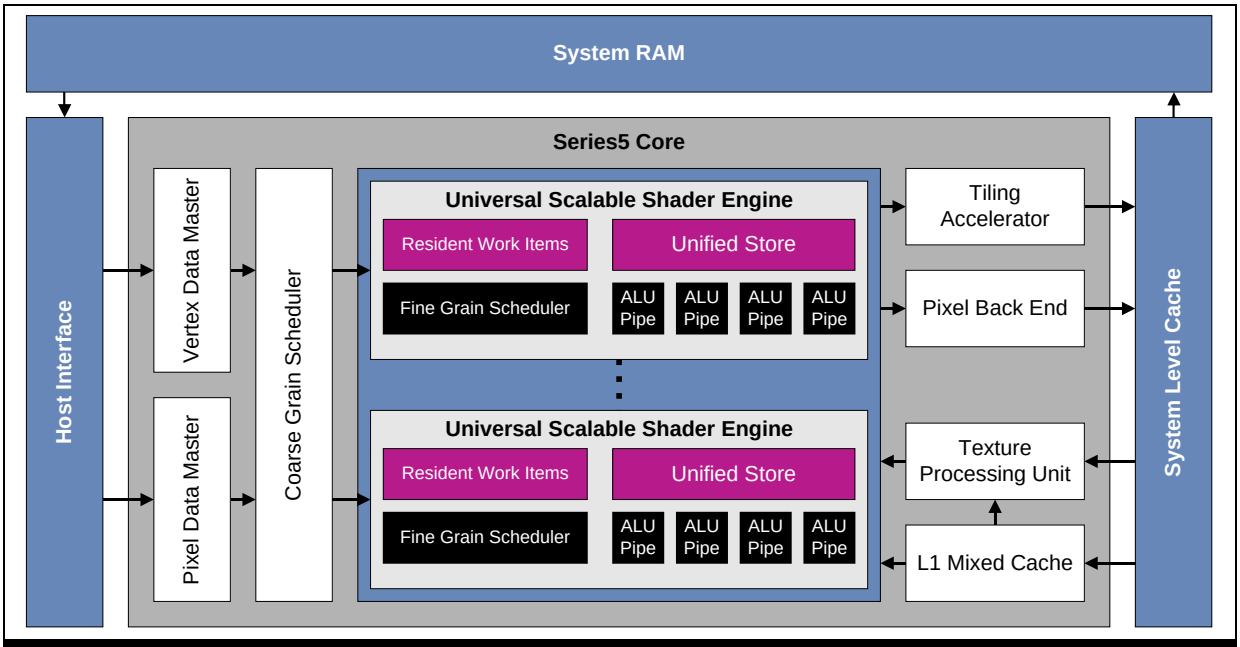

Figure 13. SGX hardware schematic

Once all of the scene's geometry has been processed by the TA, per-tile rasterization, texturing and shading can begin. A tile task is first given to the ISP, where HSR, depth and stencil tests are performed. Once this is done, iteration is performed for visible primitives within the tile and a fragment processing job is submitted to the CGS for each group of visible fragments, where the grouping has been done by the ISP during the HSR process – see Section 2.4 for more information. When iteration is performed for texture coordinates, the resultant values are sent to the Texturing Co-Processor to pre-fetch texture data.

The CGS breaks each fragment processing job into tasks that can be distributed to the Thread Schedulers of available USSE Execution Units. When dependent texture reads are performed by the USSE Execution Units, the thread will be suspended and a texture fetch task will be issued to the Texturing Co-Processor. When the tile's fragment processing is complete, the PBE (Pixel Back End) accesses the tile's buffers and applies final operations to the rendered image before it is flushed to the frame buffer in system memory, e.g., applying dither patterns when required, combining MSAA sub-sampled, etc.

4. Other Considerations

4.1. Alpha Test/Fragment Discard

Performing alpha test/discard in an application removes some of the advantage of overdraw reducing techniques such as HSR and Early-Z. The reason for this is that fragment visibility is not known until per fragment operations are performed, which means the hardware has to assume that all fragments for that object may contribute to the frame buffer colour and process them anyway. As more objects using discard overlap a given pixel, more overdraw will be introduced (all layers using discard have to be processed as the hardware does not know which, if any, of the fragments will contribute to the frame buffer colour).

For an object using discard, the ISP assumes all fragments will be visible and submits them all to the TSP. The TSP then submits all of the fragments to the USSE to be processed and, potentially, discarded. As fragments are discarded, the USSE sends fragment visibility information back to the ISP so that depth and stencil buffers can be updated accordingly (as highlighted in Figure 14). To improve the performance of discard, the hardware utilises a number of optimizations to reduce the workload as early as possible in the pipeline.

Applications should avoid the use of alpha test/discard where possible as objects using this feature cannot fully benefit from the HSR process. When an application needs to use discard, all opaque objects should be rendered before objects using discard so that obscured fragments can be removed from the render, e.g., if a wall of a building is partially obscuring a tree object using discard, rendering the opaque building first will ensure the hardware only has to process the area of the tree that is not obscured.

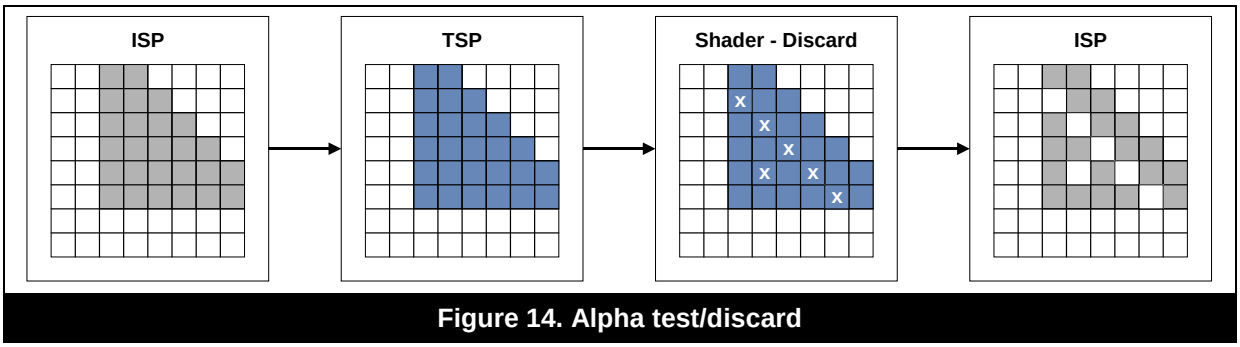

Figure 14. Alpha test/discard

4.2. Blending

To render blended objects correctly, the hardware has to process each object individually as they may all contribute to the frame buffer's colour. Similar to discard operations, the hardware utilises a number of optimizations to reduce the workload as early as possible in the pipeline.

All opaque objects should be drawn before blended objects to achieve the maximum benefit from the HSR process. When an application needs to use discard and blending, opaque objects should be submitted first, then discard objects, then blended objects.

As all blending operations are performed using on-chip memory, they can be executed very quickly and, unlike many other architectures, do not waste system memory bandwidth.

4.3. Parameter Buffer Management

Although the TBDR architecture provides a very efficient solution for processing 3D graphics, where overdraw and memory bandwidth use can be significantly reduced compared to other architectures, it does require system memory to be reserved for intermediate data stored in the Parameter Buffer (PB). As a finite amount of memory is allocated for the PB the hardware has to cope when this memory has been filled but the render is still incomplete. The TBDR solution to this is to flush the render when the buffer fills. This allows the hardware to free memory in the PB associated with objects that are rendered during the flush, the freed space can then be used to render other objects in the scene allowing the hardware to complete the render. The downside of this is that objects rendered

during the flush will not benefit from HSR performed on objects later in the render, which means overdraw may be introduced.

Additionally, the hardware has to flush all buffers associated with the render, which means depth and stencil buffers will have to be flushed in addition to the colour buffer. This needs to be done so that successive renders will execute correctly. Although there is some cost associated with this mode of operation, the technique has been used for years in PowerVR graphics hardware and, as such, has benefitted from much optimization that allows it to have a minimal effect on performance.

The amount of memory allocated for the PB on a given device is done in such a way that the memory footprint is kept as low as possible, but the majority of graphics applications will never encounter this mode of operation (only extremely complex scenes may induce it). On closed platforms using PowerVR graphics hardware, the OEM may choose to expose the ability for an application to choose the amount of memory that is allocated for the PB. By doing so, application developers can opt to allocate more or less memory for the PB than the default for the device, which allows them to raise the threshold before PB management is encountered.

5. Notable Features

5.1. Internal True Colour™

Internal True Colour™ is a term that refers to the hardware's ability to perform all blending operations at 32 bits colour precision. This is possible because a 32 bit colour buffer is always used when processing each tile, regardless of the target frame buffer precision. In traditional IMR architectures, image quality is directly related to the frame buffer colour depth. When a 16 bit colour buffer is used in these architectures, multiple reads and writes of the frame buffer will result in a loss of precision. This inaccuracy tends to create "banding" artefacts in the rendered image. While dithering is often used to reduce the effect of banding, it can potentially create worse results again, as the dither pattern applied to multiple blended layers accumulates and strong dither pattern artefacts are introduced, which gives the image a "grainy" appearance. On PowerVR hardware, all blending is performed at 32 bits precision on-chip and each pixel is only written into the frame buffer once, which results in a much higher quality image.

5.2. Full Screen MSAA

Another benefit of the SGX and SGX-MP architecture is the ability to perform efficient 4x Multi-Sample Anti-Aliasing (MSAA). MSAA is performed entirely on-chip, which keeps performance high without introducing a system memory bandwidth overhead (as would be seen when performing anti-aliasing in some other architectures). To achieve this, the tile size is effectively quartered and 4 sample positions are taken for each fragment (e.g., if the tile size is 16x16, an 8x8 tile will be processed when MSAA is enabled). The reduction in tile size ensures the hardware has sufficient memory to process and store colour, depth and stencil data for all of the sample positions. When the ISP operates on each tile, HSR and depth tests are performed for all sample positions. Additionally, the ISP uses a 1 bit flag to indicate if a fragment contains an edge. This flag is used to optimize blending operations later in the render.

When the subsamples are submitted to the TSP, texturing and shading operations are executed on a per-fragment basis, and the resultant colour is set for all visible subsamples. This means that the fragment workload will only slightly increase when MSAA is enabled, as the subsamples within a given fragment may be coloured by different primitives when the fragment contains an edge. When performing blending, the edge flag set by the ISP indicates if the standard blend path needs to be taken, or if the optimized path can be used. If the destination fragment contains an edge, then the blend needs to be performed individually for each visible subsample to give the correct resultant colour (standard blend). If the destination fragment does not contain an edge, then the blend operation is performed once and the colour is set for all visible subsamples (optimized blend).

Once a tile has been rendered, the Pixel Back End (PBE) combines the subsample colours for each fragment into a single colour value that can be written to the frame buffer in system memory. As this combination is done on the hardware before the colour data is sent, the system memory bandwidth required for the tile flush is identical to the amount that would be required when MSAA is not enabled.

5.3. PVRTC Texture Compression

PVRTC is the PowerVR texture compression scheme. PVRTC boasts very high image quality for competitive compression ratios: 4 bits per pixel (PVRTC 4bpp) and 2 bits per pixel (PVRTC 2bpp). These represent savings in memory footprint of 8:1 (PVRTC 4bpp) and 16:1 (PVRTC 2bpp) compared to 32 bit per pixel textures. By using PVRTC compressed textures, an application can significantly reduce its system memory bandwidth overhead without drastically reducing the quality of the texture. More information can be found in the "PVRTC & Texture Compression Usage Guide".

6. 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