Programming manual

STM32MP157 GPU application programming manual

Introduction

The STMicroelectronics STM32MP157 line microprocessors embed a Vivante GPU (graphics processor unit) that uses the Khronos® Group OpenGL® ES (embedded system) and OpenVG™ standards. This document addresses a number of items that need to be considered when using the Vivante GPU to accelerate a graphics-based application. When used efficiently, the Vivante GPU accelerates the eye-catching visuals of an application while minimizing system resource (CPU, memory, bandwidth, power) loading. The end result is an optimized solution that maximizes the user experience. In the remaining sections it is assumed the reader understands the fundamentals of OpenGL® ES programming or another graphics API (application programming interface).

There are a few hints and tricks to optimize the application to take full advantage of the GPU hardware. The following sections show some general recommendations and best practices for OpenGL ES and graphics programming in general.

This document describes generic GPU features; see the related STM32MP157 documentation [1], [2] for more technical details.

This document contains proprietary copyright material disclosed with permission of Vivante Corporation.

Reference documents

[1]	*STM32MP157 advanced Arm®-based 32-bit MPUs* reference manual (RM0436)
[2]	*STM32MP157 GPU tool kit* user manual (UM2547)

1 General information

The following table presents a non-exhaustive list of the acronyms used in this document.

Table 1. List of acronyms

Acronym	Definition
AI	Artificial intelligence
API	Application programming interface
ES	Embedded systems
FPS	Frames per second
GUI	Graphical user interface
GPU	Graphics processing unit
MSAA	Multisample anti-aliasing
OpenGL	Open graphics library
SoC	System on chip
VBO	Vertex buffer object
VTK	Vivante tool kit/STM32MP157 line GPU tool kit

This document applies to the STM32MP157 line devices, which are Arm®-based devices.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 Introduction to OpenGL/OpenGL ES

OpenGL is a royalty-free, cross platform C-based API that is portable across a wide range of systems. The API is maintained by the Khronos Group and additional information can be found at *www.khronos.org*. OpenGL is a method of rendering graphics (3D / 2D) data onscreen by transforming graphics primitives (points, lines, triangles) into visual information that a user can see. For example, a developer can program and configure the Vivante 3D GPU pipeline, send data to the pipeline, and the GPU executes the graphics commands.

OpenGL ES (OpenGL for embedded systems) is a subset of OpenGL that removes redundant functionality and packages the API in a library that is targeted for mobile and embedded platforms. Even though this document focus on OpenGL ES 2.0 in the following sections, the same rules apply to OpenGL ES and other 3D rendering APIs.

3 Application programming recommendations

The recommendations listed below take a holistic approach centered on overall system level optimizations that balance graphics and system resources.

3.1 Understand the system configuration and target application

Detailed application and use-case knowledge allows developers to correctly utilize the hardware resources in an ideal access pattern. For example, an implementation for a 2D or 3D GUI could be rendered in a single pass instead of multiple passes if the draw call sequence is correctly ordered. In addition, knowing the most common graphics function calls allows developers to parallelize rendering to maximize performance.

The use of Vivante and vendor specific SoC profiling tools allow the user to determine bottlenecks in the GPU and CPU and to make changes as needed. For example, in a 3D game, most CPU cycles may be spent on audio processing, AI, and physics and less on rendering or scene setup for the GPU. In this case the application is CPU-bound and configurations dealing with non-graphics tasks need to be reviewed and modified. If the system is GPU-bound, then the profiler can point out where the GPU programming code bottlenecks are located and which sections to optimize to remove restrictions.

3.2 Optimize off-chip data transfer such as accessing off-chip DDR memory/mobile DDR memory

Any data transfer off-chip takes bandwidth and resources from other functional blocks in the SoC, increases power, and causes additional cycles of latency and delay as the GPU pipeline needs to wait for data to return from memory. Using on-chip cache and writing the application to better take advantage of cache locality and coherency increases performance. In addition, accessing the GPU frame buffer from the CPU (not recommended) causes the driver to flush all queued render commands in the command buffer, slowing down performance as the GPU has to wait since the command queue is partially empty (inefficient use of resources) and CPU-GPU synchronization is not parallelized.

3.3 Avoid random cache or memory accesses

Cache thrashing, misses, and the need to access data in external memory causes performance hits. An example would be random texture cache access since it is expensive when performing per-pixel texture reads if the texture units need to access the cache randomly and go off-chip if there is a cache miss.

3.4 Optimize the use of system memory

Memory is a valuable resource that needs to be shared between the GPU (frame buffer), CPU, system, and other applications. If the application allocates too much memory for your OpenGL ES application, less memory is available for the rest of the system, which may impact system performance. Claim enough memory as needed for the application then deallocate it as soon as the application no longer needs it. For example, the application can allocate a depth buffer only when needed or if the application only needs partial resources, it can load the necessary items initially and load the rest later.

3.5 Take advantage of fast memory transfers

Use compression to reduce image bandwidth and vertex/index buffers to increase memory transfer speeds. Compression reduces texture memory footprint and lower bus bandwidth. Mipmapping also reduces page-breaks and better utilize texture cache.

3.6 Target a fixed frame rate that is visibly smooth

Smooth frame rate is achieved from a combination of a constant FPS and the lowest FPS (frames per second) that is visually acceptable. There is a trade-off between power and frame rates since the graphics engine loading increases with higher FPS. If the application is smooth at 30 FPS and no visual differences for the application are perceived at 50 FPS, then the developer should cap the FPS at 30 since the extra 20 FPS do not make a visual difference. The FPS limit also guarantees an achievable frame rate at all times. The savings in FPS help to lower GPU and system power consumption.

3.7 Minimize GL state changes

Setting up state values between draw calls adds significant overhead to application performance so they must be minimized. Most of these call setups are redundant since you are saving / restoring states prior to drawing. Try to avoid setting up multiple state calls between draw calls or setting the same values for multiple calls. Sometimes when a specific texture is used, it is better to sort draw calls around that texture to avoid texture thrashing which inhibits performance. Application developers should also try to group state changes.

3.8 Batch primitives to minimize the number of draw calls

When the application submits primitives to be processed by OpenGL ES, the CPU spends time preparing commands for the GPU hardware to execute. If the application batches only draw calls into fewer calls, it reduces the CPU overhead and increases the draw call efficiency. Batch processing allows a group of draw calls to be quickly executed without any intervention from the CPU (driver or application) in a fire-and-forget method.

Some examples of batching primitives are:

- Branching in shaders may allow better batching since each branch can be grouped together for execution.
- For primitives like triangle strips, the developer can combine multiple strips that share the same state to save successive draw calls (and state changes) into a single batch call that uses the same state (single setup) for many triangles.
- Developers can also consolidate primitives that are drawn in close proximity to take advantage of spatial relationships. If the batched primitives are too far apart, it is more difficult for the application to effectively cull if they are not visible in the frame.

3.9 Perform calculations per vertex instead of per fragment/pixel

Since the number of vertices is usually much less than the number of fragments/pixels, it is cheaper to do per vertex calculations to save processing power.

3.10 Enable early-Z, hierarchical-Z and back face culling

Hardware support of depth testing to determine if objects are in the user’s field of view are used to save workload and processing on vertex and pixel processing. If the object is in view, then the vertices are sent down the pipeline for processing. If the object is hidden or not viewable, the triangles are culled and not sent to the pipeline. This improves graphics performance since computations are only spent on visible objects. If the application already knows details about the contents and relative position of objects in the scene or screen, the developer can use that information to automatically bound areas that never need to be touched (for example an automotive application that has multiple layers of dials where parts of the underlying dials are occluded can have the application avoid occluded areas from the beginning). Another optimization is to perform basic culling on the CPU since the CPU has first-hand information about the scene details and object positions so it knows what scene data to send to the GPU.

3.11 Use branching carefully

Static branches perform well since states are known but they tend to use many general purpose registers. An example is a long shader that combines multiple shaders into a single, large shader that reduces state changes and batch draw calls. Dynamic branching has non-constant overhead since it processes multiple pixels as one and everything executes whether a branch is taken or not. In other words, dynamic branching goes through different permutations/branches in parallel to reach the correct results. If all pixels take the same path, then performance is good. The more pixels processed translates to higher overhead and lower performance. For dynamic branching, smaller pixel sizes/groups are optimal for throughput. Developers need to be aware of branching in their code to make sure excessive calculations and branches are efficient. Profiling tools can help determine if certain parts of code are optimized or not.

3.12 Use VBO for vertex data instead of static or stack data

A vertex buffer object (VBO) is a buffer object that provides the benefits of vertex array and display list and allows a substantial performance gain for uploading data (vertex position, color, normals, and texture coordinates) to the GPU. VBOs create buffer objects in memory and allow the GPU to directly access memory without CPU intervention (DMA). The memory manager can optimize buffer placement using feedback from the application. VBOs can also handle static and dynamic data sets and are managed by the Vivante driver. The benefits of each are:

- A vertex array reduces the number of function calls and allows redundant data to be shared between related vertices, instead of re-sending all the data each time. Access to data can be referenced by the array index.
- The display list allows commands to be stored for later execution and can be used repeatedly over multiple frames without re-transmitting data, thus minimizing CPU cycles to transfer data. The display list can also be shared by multiple OpenGL / OpenGL ES clients so they can access the same buffer with the corresponding identifier. If computationally expensive operations (such as lighting or material calculations) are put inside the display lists, then these computations are processed once when the list is created and the final result can be re-used multiple times without needing to re-calculate again.

If the application combines the benefits of both vertex array and display list by using VBO, performance is increased over static or stack data sets.

3.13 Use dynamic VBO if data is changing frame by frame

Locking a static vertex buffer while the GPU is using it can create a performance penalty since the GPU needs to finish reading the vertex data from the buffer before it can return to the calling application. Locking and rendering from a static buffer many times per frame also prevents the GPU buffering render commands since it mush finish commands before returning the lock pointer. Without buffered commands the GPU remains idle until the application finishes filling the vertex buffer and issues the draw commands.

If the scene data never changes from frame to frame then a static buffer may be sufficient. With newer applications (such as games or maps) that have dynamic viewports where vertex data changes multiple times per frame or frame-to-frame, then a dynamic VBO is required to ensure performance is still met. If the current buffer is being used by the GPU when a lock is called, a pointer to a new buffer location is returned to the application to ensure updated data is written to the new buffer. The GPU can still access the old data (current buffer) while the application puts updated data into the new buffer. The Vivante memory management unit and driver automatically take care of allocating, re-allocating or destroying buffers.

The application can implement dynamic VBO depending on their preference, but the recommendation is to allocate a 1 Mbyte dynamic VBO block and upload data to using different offsets for each dynamic buffer. If the buffer overflows, the application can loop back and use location offset 0 again.

3.14 Tesselate data for a better hierarchical Z (HZ) performance

Open GL and Open GL ES handle tesselation in a different way as described below.

OpenGL only renders simple convex polygons (edges only intersect at vertices with no duplicate vertices and only two edges meet at any vertex), in addition to points, lines, and triangles. If the application requires concave polygons (polygons with holes or intersecting edges), those polygons need to be subdivided into simple convex polygons, which is called tessellation (subdividing a polygon mesh into a bunch of smaller meshes). Once the application has all the meshes in place, the HZ hardware can automatically cull hidden polygons to efficiently process the frame, effectively breaking the frame into smaller chunks that can be processed very fast.

OpenGL ES only renders triangles, lines, and points. The same concepts apply as in OpenGL, which is to avoid very large polygons by breaking them down into smaller polygons where our internal GPU scheduler can distribute them into multiple threads to fully parallelize the process and remove hidden polygons.

3.15 Use dynamic textures as a texture cache (texture atlas)

The main reason for using dynamic textures as a cache is the application developer can create one larger texture that is subdivided into different regions (texture atlas). The application can upload data into each region and use an application side texture atlas to access the data. Each dynamic texture and sub-region can be locked, written to, and unlocked each frame, as needed. This method of allocating once is more efficient than using multiple smaller textures that need to be allocated, generated, and then destroyed each time.

3.16 Stitch together small triangle strips

It is better to combine several small, spatially related triangle strips together into a larger triangle strip to minimize overhead and increase performance. For each triangle strip, there are overhead and start up costs that are required by the CPU and GPU, including state loads. If there are too many small triangle strips that need to be loaded, performance is lowered. An application developer can combine multiple triangle strips by adding two dummy vertices to join the strips together. The overhead to restart multiple new strips is much higher than adding the two dummy vertices.

3.17 Specify EGL configuration attributes precisely

To obtain a 16 bit/pixel window buffer for rendering, the EGL config attributes need to be specified precisely according to the EGL spec. Specifying inaccurate EGL attributes may result in getting a 32-bit bit/pixel window buffer which doubles the bandwidth requirement for rendering which in turn leads to lower performance.

3.18 Use power-of-two aligned texture/render buffers

Most GPUs can process/render power-of-two aligned texture/render buffers more efficiently as the GPU's internal bus and cache are power-of-two aligned.

3.19 Disable MSAA rendering unless high quality is needed

Although MSAA rendering can achieve higher image quality with smoother lines and triangle edges, it requires much higher (4x, 8x) bandwidth because it has to rendering a single pixel 4x/8x times. So, if high rendering quality is not required, MSAA should be disabled.

3.20 Avoid partial clears

Most GPUs have special hardware logic to do a fast clear of an entire buffer. So it is better to utilize the fast clear function to clear the entire buffer then render graphics again, instead of doing a partial clear to preserve a graphics region.

3.21 Avoid mask operations

Do not use mask unless the mask is 0 (other than when you need a specific render quality). Clearing a surface with mask (color /depth stencil mask) could have a performance penalty. Pixel mask operations are normally quite expensive on some GPUs, as the mask operation has to be done on every single pixel.

3.22 Use MIPMAP textures

MIPMAP textures enable the application to sample a lower resolution texture image (1/2, 1/4, 1/8, 1/16, ... size of the original texture image) when the triangle is rendering further away from the view point. Thus, the bandwidth required to read the texture image is reduced which leads to better performance.

3.23 Use compressed textures if possible

Compressed textures normally are only a fraction (up to 1/8) of the original texture size. So the bandwidth required to upload and sample the compressed texture is much less than for an uncompressed texture.

3.24 Draw objects from near to far if possible

Drawing objects from near to far normally has better performance because the objects in the near foreground can block entire or partial objects in the background. Most GPUs have early Z rejection logic to reject the pixels that will fail a Z compare. The GPU can skip fragment shader computations on these rejected pixels.

3.25 Use indexed triangle strips; most optimal are pairs of six triangles

Index triangle strips can maximize the vertex cache utilization as each set of vertex data can be used in two triangles. Pairs of six triangles can fit exactly into the vertex cache.

3.26 Vertex attribute stride should not be larger than 256 bytes

Most Vivante GPUs provide native support for a 256 byte vertex attribute stride. If the vertex attribute stride is larger than 256 bytes, then the driver has to copy the vertex data around.

3.27 Avoid binding buffers to mixed index/vertex array

Most of Vivante GPUs do not natively support mixed index/vertex arrays. So the Vivante driver must copy the index and vertex data around to form separate vertex data streams for the GPU. Avoid mixing index and vertex data so the driver does not have to incur a performance hit while performing this task.

3.28 Avoid using CPU to update texture/buffer contexts during render

Do not use the CPU to update texture/buffer contexts in the middle of rendering. Using the CPU to update texture/buffer causes the rendering pipeline to flush and stall, so that CPU can safely update the buffer contents. The pipeline flush/stall/resume causes significant performance impact.

3.29 Avoid frequent context switching

Context switch is an inherently expensive operation as many GPU states need to be reset to start a new rendering context. Thus, frequent context switching will have a negative impact on application performance.

3.30 Optimize resources within a shader

Most GPUs have optimal support for a limited amount of resources (as uniforms or varying). Using resources beyond the optimal working set causes the GPU to fetch/store resources from a lower performance memory pool and shader performance is negatively impacted.

3.31 Avoid using glScissor Clear for small regions

glScissor Clear for small regions (less than 16x8 aligned window) falls back to CPU so the performance is not optimal.

3.32 Avoid using VBO and non-VBO for vertex attribute inputs within a draw call

Within one draw call, when one vertex attribute input is in a conventional vertex array (in application system memory, instead of in a vertex buffer object (VBO)) or a vertex attribute input is not from an array, but from a current vertex attribute value set by the API function glVertexAttrib* (not in a VBO), this mixed usage requires an added scan of the index buffer. If the index buffer is in GPU video memory, this causes a performance lag. Avoiding a scan of the index buffer is better. However, if a scan must be performed, it is better to have the index buffer in a conventional vertex array (in system memory) to reduce the performance hit.

The driver scans the index buffer to get the maximum and minimal index value. It uses these values to decide how much video memory to allocate to accommodate the data which is copied from application system memory or current vertex attribute value. Scanning the index buffer object if it resides in GPU video memory is slower. The best performance for a draw call is to put all vertex attributes in a VBO and not reference the current vertex attribute value.

3.33 Do not call glFinish unnecessarily

glFinish should be called only if the previous actions must be completed before the next command uses the result.

3.34 Schedule a dependable amount of work per frame to obtain a steady frame rate

Trying to render 10x the pixels in one frame relative to other frames results in a slower frame rate.

3.35 Select the number of pixels being rendered to correspond to the desired performance

The number of pixels being rendered is directly proportional to the performance. This also includes offline rendering.

3.36 Consider using front-to-back drawing order

Draw order is important. Consider the Z-buffer and use front-to-back drawing order instead of back-to-front.

3.37 Use eglWaitSync when compositing from multiple threads

When eglWaitSync is used with multiple threaded programs, this ensures the composition thread does not render before the child surface is finished rendering. eglWaitSync does not help with single threaded programs.

4 Recommendations summary

4.1 Best practices

The following operations optimize the application to take full advantage of the GPU hardware:

- Use profiling tools to identify bottlenecks and optimize resources
- Use on-chip cache and optimize use of system memory where possible
- Use buffer objects, especially vertex buffer objects (VBOs)
- Minimize GL state changes
- Batch primitives
- Use texture atlas caches
- Calculate per vertex instead of per pixel/fragment
- Make use of the hardware functions to control Z
- Select a fixed frame rate that is visibly smooth
- Specify EGL configuration attributes precisely
- Use power-of-two alignments for texture and render buffers
- Disable MMAA rendering unless high quality is needed
- Use MIPMAP textures
- Use compressed textures if possible
- Draw objects from near to far if possible
- Use indexed triangle strips; most optimal are pairs of six triangles
- Stitch triangle strips together with dummy vertices if practical
- Put all vertex attributes in a vertex buffer object (VBO) for a draw call
- Schedule a dependable amount of work per frame to obtain a steady frame rate
- Select the number of pixels being rendered to correspond to the desired performance
- Consider using front-to-back drawing order
- Use eglWaitSync when compositing from multiple threads

4.2 Bad practices

The following operations usually have a bandwidth/performance impact. Some are Vivante-specific restrictions which optimize performance on Vivante's GPU or driver architectures:

- Do not make the vertex attribute stride larger than 256 bytes
- Do not bind a buffer to mixed index/vertex arrays
- Do not use CPU to update texture/buffer contexts in the middle of a render
- Do not switch contexts too often
- Do not use too many resources in one shader (uniforms or varying)
- Do not use glScissor for clear for small regions
- Avoid partial clears
- Avoid mask operations
- Do not use glClear if the frame does not need to be cleared
- Do not call glFinish unnecessarily. glFinish should only be called if the previous actions must be completed before the next command utilizes the result.

Revision history

Table 2. **Document revision history**

Date	Revision	Changes
15-Feb-2019	1	Initial release.
11-Mar-2019	2	Update document's publication scope.
28-Mar-2019	3	Updated document's publication scope. Updated Introduction on cover page. Deleted a section that was appearing twice in the document ("Use power-of-two aligned texture/render buffers").

Contents