Optimization of GPU-Based Surround View on TI's TDA2x SoC

Lucas Weaver, H. Hariyani, S. Sivasankaran   *Embedded Processing Automotive*

ABSTRACT

Automotive surround view is an evolving technology that provides drivers with a real-time 360° view of the area surrounding the vehicle. As an evolving technology, surround view systems pose increasing demands on existing system resources. This requires the development of new features, performance improvements, and optimizations. This application report describes optimizations for GPU-based surround view performed on TI's on TDA2x System-on-Chip (SoC). Specifically, this document describes optimizations to perform 3D mesh table generation on the DSP, region-based rendering, and the parallelization of fragment and vertex shader operations.

This document builds upon the explanation of TI's surround view camera system described in *Surround View Camera System for ADAS on TI's TDAx SoCs* [1].

Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 3D Mesh Table Generation Optimization

1.1 3D Mesh Table Generation

When the system of cameras are calibrated, a LUT (look-up table) is created, mapping the surround view output to the input camera images. The assumption is that the camera positions are fixed after calibration. The mesh table encodes information about the shape of the 3D world around the vehicle, and texture mapping. This enables the graphics processor to generate output rendering from various camera viewpoints.

Figure 1 shows a block diagram for the mesh table generation procedure. The mesh table consists of 3D world coordinates for locations surrounding the vehicle, and the associated input locations for texture mapping from adjacent cameras viewing the scene for a given location. The output is represented as a bowl, whose height varies as a function of the distance from the center of the vehicle. The collection of the mesh table, which includes output mesh and the associated texture mapping, is passed on to the graphics processor for further rendering. Along with the mesh table, a blending LUT is also generated, which encodes the weights for linear combination of image intensity information received at each location from adjacent cameras. This blending table, along with the mesh table, is generated once at start-up, and stored in memory to be re-used in each. The following section describes how these tables are used to generate 3D surround view rendering from various viewpoints using the SGX (graphics) core on the embedded device.
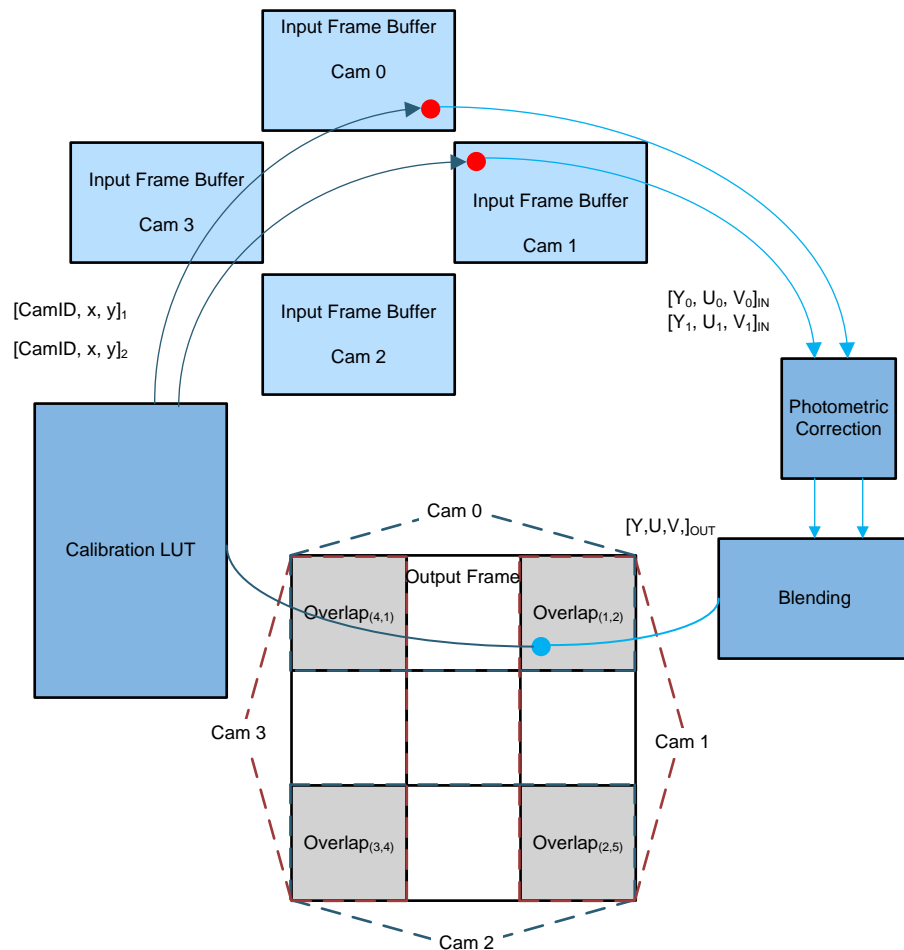

Figure 1. 3D Mesh Table Generation Flow Diagram

1.2 3D Mesh Table Generation Optimization on DSP

The 3D mesh table generation algorithm has been optimized on the C66x at 750 MHz, refreshing every frame at 30 fps for a 4×4 sub-sampled 1080 × 1080 bowl, to take approximately 21 milliseconds. The DSP core simulation of the algorithm (not considering memory latencies) shows a best case performance of 7.14 milliseconds. Although this value is unrealistic due to memory latencies, DMA-based memory access could improve bowl generation time to be closer to the best case scenario. One of the big advantages of heterogeneous systems such as TI's TDA2x with TI C66x DSP is that the mesh table can be generated real-time independent of GPU and ARM bandwidth. This is achieved by taking advantage of TI's SoC and DSP architecture, and because the DSP is better at performing pixel remapping operations than the GPU. This approach offloads the GPU and results in about an 8x improvement in performance on shaders.

2 OpenGL Region-Based Rendering

2.1 GPU Rendering

When the 3D mesh table is generated, it is passed to the OpenGLES part of the application for GPU rendering. The GPU used here is Imagination Technologies PowerVR SGX544 (MP2 or MP1, depending on the device).

The mesh table is read as a set of vertex attributes – vertex coordinates (x, y, and z) and texture coordinates for each of the two cameras contributing to a point on 3D surround view. A separate blend table assigns weights to pixel contributions from each camera.

Camera images are passed on the GPU using GL_OES_EGL_image_external extension that allows YUV images to be passed on to the GPU as textures.
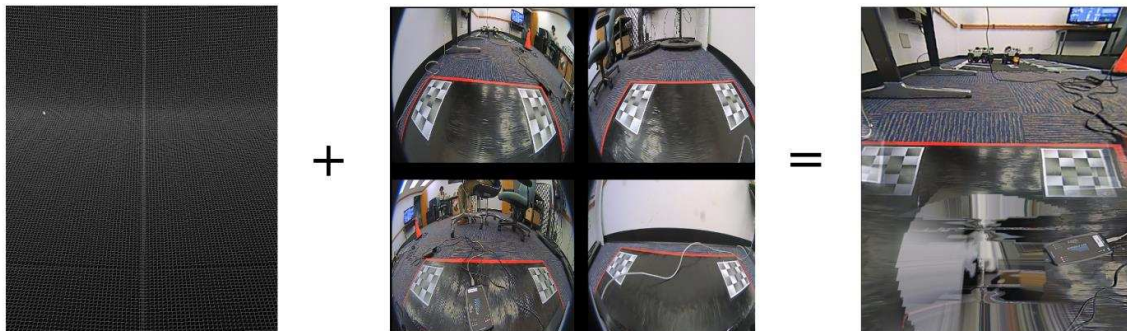

Figure 2. 3D Mesh, Input Texture Images, and Final Rendered Output

2.2 Region-Based Rendering

A normal straightforward approach is to treat the entire bowl as a single mesh. The downside of this approach is that because each vertex has two cameras contributing, the contributing cameras must be found in the shader code. This is inefficient, as it requires branches in shader code. Considering a large number of vertices, this inefficiency results in decreased frame rate and increased latency.

In a region-based approach, the rendering is split into multiple overlapping regions, such that each region has vertices corresponding to only two cameras. The regions are overlapping, because in general, vertices do not correspond to pixels and can change when the user zooms in or out, resulting in big jumps between two regions.

Moreover, for a particular region, the camera identities remain the same. A separate blend table is used to assign weights between 0 and 1 for each camera. In sub-regions where pixel contributions come from both cameras, the blend table provides factors of pixel contributions to be blended from each camera image. For sub-regions where pixel contribution comes from a single camera, weights are either 0 or 1, indicating the camera that contributes to the pixel value.

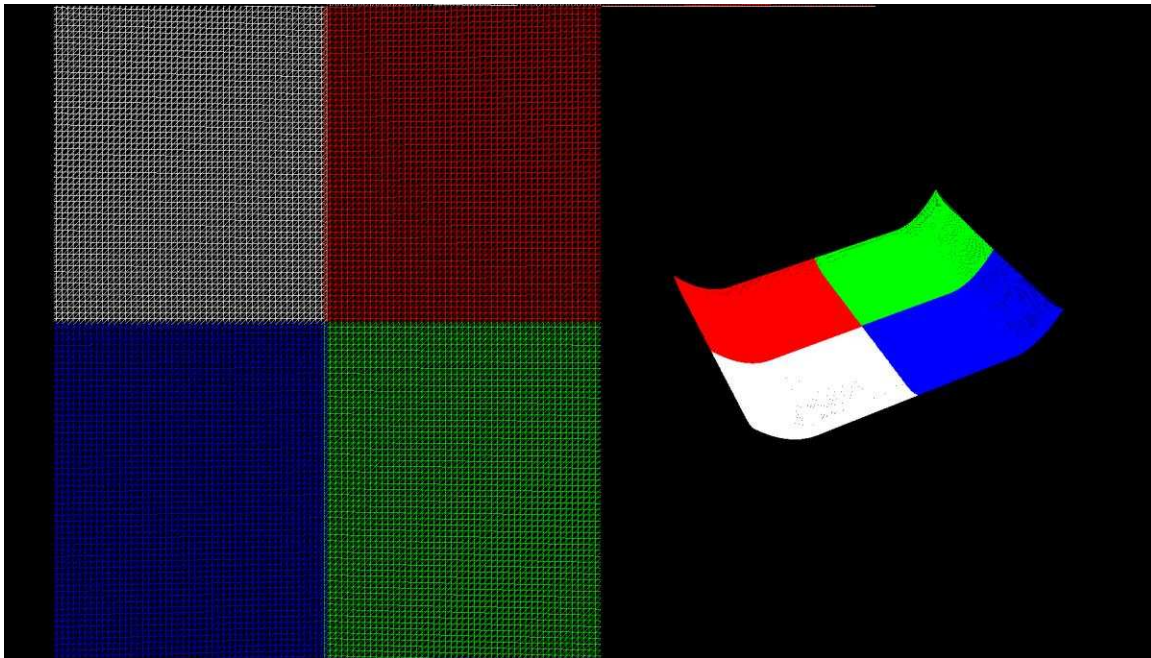With four cameras, the regions are shown as quadrants (see Figure 3).

Figure 3. Overlapping Quadrants (regions)

3 OpenGL Region-Based Rendering

GPU usage can be maximized by parallelizing fragment and vertex shader operations. The framework or application should support this by implementing a multi-thread approach for render job submission and wait-sync. Figure 4 shows the two task model. Where the process task receives the input video frames, prepares the render job, and submits the same to GPU; eglWaitGL is not implemented on this process task. EglWaitGL is implemented on the second thread: sync task. With this approach, multiple render jobs can be submitted to the GPU without waiting for eglWaitGL to overcome the blocking nature of the render process. The second thread gets triggered by eglWaitGL, then sends out the rendered output buffers to the next process module, usually the display, and frees up the input video frames.
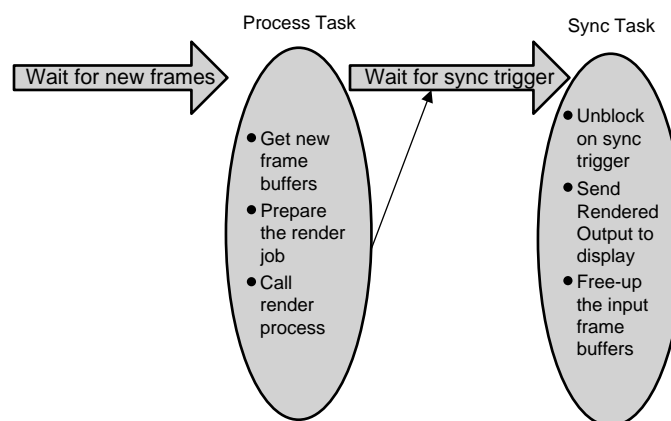
Figure 4. Two Task Approach to Parallelizing Fragment and Vertex Shader

Figure 5 shows how parallelism can be achieved between the fragment and vertex shader operations. F1, F2.., Fn are the frames captured by camera in a frame to frame manner. Fragment shader operation of the n'th frame is done in parallel with vertex shader operation of the (n+1)th frame.

F1 . . . Fn are frames in chronological order
T1 . . . Tn time in progressive manner
Color coding
Vertex Shader
Fragment Shader

T1	T2	T3	T4	T5	T6
F1					
	F1				
	F2				
		F2			
		F3			
			F3		
			F4		
				F4	
				F5	

Figure 5. Parallelizing Fragment and Vertex Shader operations

4   References

1. Surround view camera system for ADAS on TI's TDAx SoCs