Computers & Graphics 61 (2016) 40–49

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

L-Bench: An Android benchmark set for low-power mobile GPUs

Jae-Ho Nah[a], Youngsun Suh[b], Yeongkyu Lim[a,*]

[a] LG Electronics, 56, Digital-ro 10-gil Geumcheon-gu Seoul, South Korea
[b] LG Electronics, 19 ,Yangjae-daero 11-gil Seocho-gu Seoul, South Korea

ARTICLE INFO

Article history:
Received 14 April 2016
Received in revised form
9 September 2016
Accepted 12 September 2016
Available online 28 September 2016

Keywords:
GPU benchmark
OpenGL ES

ABSTRACT

In recent years GPUs have become one of the most important components in mobile application processors (APs). Thus, performance measurement and analysis of mobile GPUs are crucial to mobile AP manufacturers, device manufacturers, graphics application programmers, and end users. However, it is hard to analyze mobile GPUs in depth via existing high-level (with frames per second) or low-level benchmarks (with a fill rate, ALU performance, etc.). To bridge the gap between the benchmarks, we present a novel Android benchmark set for low-power GPUs, called L-Bench. This benchmark set consists of mid-level micro-benchmarks implemented on OpenGL ES 3.1, which are carefully chosen for different workloads. By analyzing the results, this benchmark suite provides not only frames per second of each benchmark but also performance of each GPU subsystem (geometry units, ALUs, texture mapping units, raster operations pipelines, caches/memory units, and tessellators) and overall GPU performance. For experiments, we perform our benchmark suite on five representative mobile devices that have different mobile GPUs, after that, we describe comprehensive analysis of each GPU architecture.

1. Introduction

With increasing demands on high-quality graphics on mobile devices, GPUs are now indispensable units for mobile devices, such as smartphones and tablets. In particular, mobile devices with a high-resolution screen require more powerful GPUs. As a result, GPUs usually occupy a large die area of application processors (APs), and GPU performance is becoming one of the important measures for selecting mobile devices.

However, GPU performance analysis is not an easy task because a GPU consists of very complex components, such as geometry units, ALUs, texture mapping units, raster operations pipelines (ROPs), caches/memory units, and tessellators. Of course, GPU vendors provide peak performance of each unit (e.g, the triangle rate, the FLOPs, the texel rate, the fill rate, the peak memory bandwidth, etc.). However, these peak performance values may not be comparable between different GPU architectures because actual performance can be varied by various factors such as hardware organization, architectural features, scheduling policies, driver support, and so on. This is why proper GPU benchmarks are needed.

In contrast to desktop environments, very high-end graphics applications, which can stress out GPUs, are limited in mobile environments for the following reasons. First, mobile applications should exploit limited hardware resources in mobile devices, so programmers usually avoid implementing complex graphics effects to get sufficient frame rates. Second, the OpenGL ES API only recently supports advanced features in OpenGL, so it has been hard to directly port desktop/console programs to mobile applications. As a result, only a few mobile GPU benchmarks have been usually used to measure GPU performance of state-of-the-art mobile devices, instead of real-world applications.

There are two types of mobile GPU benchmarks: high-level benchmarks and low-level benchmarks. High-level benchmarks, such as GFXBench 4.0 [1], Basemark ES 3.1 [2], and 3DMark Sling Shot Benchmark [3], usually render game-like scenes and measure their frame rates (or calculate their scores) on a device. This type of benchmarks can be useful to know overall GPU performance. In contrast, low-level benchmarks aim at measuring specific features or performance of each component in a GPU. Low-level tests in GFXBench measure performance of tessellation, ALU, driver overhead, and texturing. DrawElements Quality Program (dEQP) analyzes feature conformance and performance through several thousand function-level tests, and dEQP is now included in Google Android Compatibility Test Suite (CTS).

A limitation of the existing benchmarks is the difficulty in analyzing the results in depth. For example, if same-grade GPUs show different frame rates in high-level benchmarks, it is hard to know the reason from the results themselves. Analysis of both the high- and low-level benchmark results may not be very useful because there is no direct relationship between them if the benchmarked

* Corresponding author.

http://dx.doi.org/10.1016/j.cag.2016.09.002
0097-8493/© 2016 Elsevier Ltd. All rights reserved.

GPUs have different hardware architectures. Another concern is a correlation with real-world scenarios; because the high-level benchmarks usually render only one or two scenes, some driver optimizations for specific benchmarks by GPU vendors can achieve huge speed ups in the benchmarks and these results may be different from the actual performance in real-world scenarios.

To overcome the above limitations, we present a novel benchmark set for low-power embedded GPUs, called L-Bench. In contrast to the existing high- or low-level benchmarks, we get benchmark results from a set of mid-level micro-benchmarks, similar to SPECviewperf [4]. Our micro-benchmarks visualize one or two well-known effects and have very different workloads, so we can extract performance of each GPU component from results of the micro-benchmarks. We also calculate overall GPU performance from the results. For experiments, we executed our benchmark app on five mobile devices that have different GPUs. The results of our experiments show that the calculated overall performance in L-Bench shows similar results to the existing high-level benchmarks even though the benchmarked GPUs show very different frame rates in each micro-benchmark.

Compared to prior benchmarks, L-Bench offers the following advantages.

- *In-depth results*: L-Bench provides not only frames per second (FPS) but also performance of both each subsystem in a GPU and each graphics effect. It can be a hint to analyze pros and cons of each GPU architecture.
- *Extensibility*: L-Bench can be easily extended to test new features included in recent APIs by adding additional micro-benchmarks. In fact, we have included an Android Extension Pack (AEP) benchmark in L-Bench by simply porting an existing OpenGL tessellation source code to OpenGL ES. Additionally, more micro-benchmarks will provide more accurate, abundant results.
- *Neutrality*: L-Bench consists of several micro-benchmarks with different scenes and does not use any vendor-specific optimizations. Therefore, the overall results from L-Bench can be less influenced by specific scene features, H/W features, or driver optimizations.

The rest of this paper is organized as follows. In Section 2, we briefly summarize prior mobile GPU benchmark methodologies. In Section 3, we introduce our design criteria and the benchmark implementation process. In Section 4, we describe the techniques, workloads, and implementation details of the micro-benchmarks. In Section 5, we show the results on five mobile devices and analyze each GPU architecture in the devices. Finally, we describe conclusions, limitations, and future work in Section 6.

2. Related work

There are a few papers about 3D graphics benchmark methodologies on mobile devices. GraalBench [5] is a 3D graphics benchmark suite for mobile devices. This benchmark suite consists of two games and four Viewperf/VRML scenes. By collecting trace data from the scenes, the benchmark suite provides detailed workload characterization of each scene. This tool is good for understanding the benchmark scenes but does not aim at performance evaluation of actual mobile devices.

TEAPOT [6] is a mobile GPU simulator to evaluate GPU architectures. This simulation infrastructure utilizes GPU traces from mobile applications similar to GraalBench and estimates performance and energy consumption of GPU architectures from the results.

In contrast to GraalBench and TEAPOT, Ma et al. [7] measured the actual performance of two game benchmarks on three smartphone models. To analyze computation time and power consumption of each graphics pipeline stage, their approach disables some or all graphics pipeline stages.

Meanwhile, Johnsson et al. [8,9] mainly focused on power efficiency. Johnsson et al. [8] built a power measuring device, and using the device, they measured power and performance of six rendering/shadow algorithms on four discrete/integrated/mobile GPUs. Energy per pixel of each case was reported as the results. Johnsson et al. [9] presented a simpler, non-invasive measurement method. Because the beginnings of the frames can be detected in a semi-automatic way, the method can be used when the source code is not available.

Finally, we briefly compare our approach with SPECviewperf [4]. Both approaches include a set of mid-sized micro-benchmarks to measure graphics performance. SPECviewperf aims at measuring professional graphics performance on workstations, and the micro-benchmarks of SPECviewperf are from actual applications. In contrast, our approach aims at measuring mobile graphics performance based on OpenGL ES, and each micro-benchmark represents one or two effects and different workloads. Additionally, we extract the performance of each GPU subsystem from the results for understanding of GPUs. A more detailed description of our design criteria will be shown in the next section.

3. Design criteria and implementation process

Before we describe details of the micro-benchmarks, we introduce our design criteria and benchmark implementation process in this section. Since we started this benchmark project, we have made the following criteria for further implementation.

- The benchmark is implemented using OpenGL ES 3.1 which is the dominant API for modern mobile devices. Additionally, our benchmark suite is based on the Android platform. This is because Android is the most widely used mobile operating system now and we can compare various GPUs designed by different vendors on the Android platform. Additionally, we include a micro-benchmark to test AEP because the recent OpenGL ES 3.2 spec includes all features of AEP and all major mobile GPU IP vendors have announced AEP support at least in the near future.
- Proper micro-benchmark selection is very important to result in accurate GPU evaluation, so we first establish micro-benchmark selection criteria before actual benchmark implementation. According to the criteria, the benchmark suite should deal with (a) well-known traditional techniques widely used in both mobile and desktop platforms (e.g., shadow mapping, cube mapping, transparency, and skinning), (b) techniques that are common on desktops but forthcoming on mobile devices (e.g., deferred shading, screen-space ambient occlusion, occlusion culling, and tessellation), and (c) main new features introduced in OpenGL ES 3.1/AEP (e.g., multiple render targets, geometry instancing, occlusion queries, 3D textures, and compute/tessellation shaders).
- Micro-benchmarks should have different workloads as far as possible to analyze each GPU subsystem in a GPU (geometry units, texture mapping units, ALUs, ROPs, caches/memory units, and tessellators).
- To derive fair results from the benchmarks, we exclude the use of vendor-specific extensions even if that can achieve huge performance improvements.

- All textures are compressed by ETC2/EAC [10] to reduce memory traffic. ETC2 is the standard texture compression format in OpenGL ES 3.0.
- Because some mobile GPU architectures have different processing capacity between 16-bit and 32-bit ALUs, a proper floating-point/integer/texture precision selection in a shader program can reduce computational loads. To utilize this feature, we basically use the highp and mediump precision for vertex and pixel processing, respectively. After that, we additionally tune the precision of each variable according to its type; for example, normals are processed with the mediump precision, world-coordinate positions are processed with the highp precision, and mode flags are stored with the lowp precision. However, lower precision may cause render artifacts on some devices. Therefore, whenever we change the precision of each variable to mediump or lowp, we verify whether the lower precision does not make any visible artifacts on all test devices. If the precision change does not pass this test, we roll back the precision.
- To get on-screen results from various screen resolutions (720p, 1080p, and 1440p), we render an image to an off-screen render target and blit the image to the full screen. This is possible using the glBlitFramebuffer() function introduced in OpenGL ES 3.0. By doing the blit operation, we can get full-screen images even if the selected screen resolution in the benchmarks does not match the device's screen resolution.

Fig. 1 illustrates our benchmark implementation process. First, we implement micro-benchmark code on a laptop with Windows 7 using the OpenGL ES emulator in PowerVR SDK. Second, we port the source code to the Android environment though Android NDK. Third, we execute the benchmark app on several mobile devices and test whether the application runs properly on the devices. If the test does not make any problem, we then get the frame rates of each micro-benchmark from the devices. Fourth, we measure the workload of each micro-benchmark using four GPU profilers and four different GPUs: ARM DS-5 Streamline 5.22 with Mali T628MP4, nVIDIA Tegra Graphics Debugger 2.0 with mobile Kepler K20A, Qualcomm Snapdragon Profiler 1.5 with Adreno 418, and AMD GPUPerfStudio 3.2.18 with Radeon HD 7650M. Finally, we insert the score calculation code to the benchmark code by using the maximum FPS values and the calculated workloads of each micro-benchmark. The values are used to calculate relative performance of the fastest device (Fig. 2).

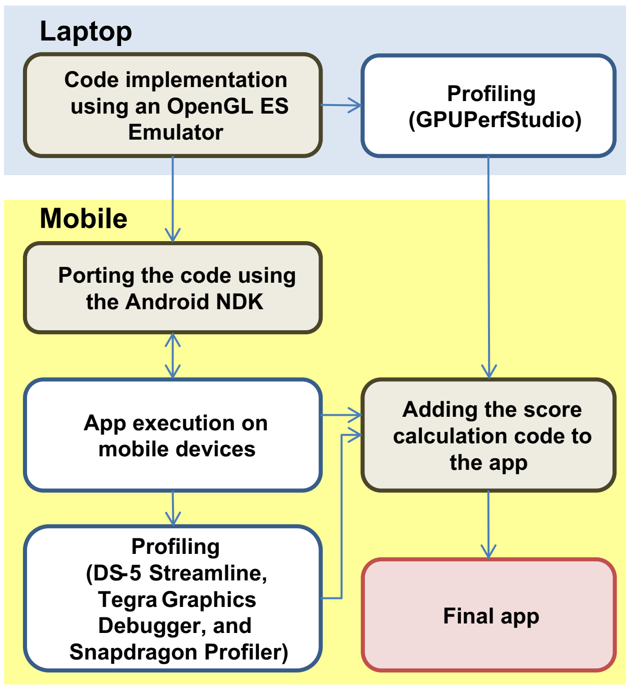
Laptop
Code implementation using an OpenGL ES Emulator
Profiling (GPUPerfStudio)
Mobile
Porting the code using the Android NDK
App execution on mobile devices
Adding the score calculation code to the app
Profiling (DS-5 Streamline, Tegra Graphics Debugger, and Snapdragon Profiler)
Final app

Fig. 1. The overall benchmark implementation process.

Our profiler choice criteria are described as follows. Among each mobile GPU vendor's profiler, we select profilers which can provide utilization of at least three of the six GPU subsystems. As a result, three mobile GPU profilers (DS-5 Streamline, Tegra Graphics Debugger, and Snapdragon Profiler) are used for our workload characterization. We additionally use AMD GPUPerfStudio with an OpenGL ES Emulator for more plentiful analysis because GPUPerfStudio provides all counter values required for our benchmark suite.

The detailed profiling and score calculation metrics are described as follows. First, the percentage of workloads is calculated from the profilers. Because the performance counters in each profiler are different, we align each profiler's counters to the utilizations of GPU subsystems. The detailed alignment is described in Table 1. If some utilization cannot be directly derived from counters in some profiler, we try to combine multiple values to get the value and carefully compare the calculated value to the related value from the other profilers. The sum of workload percentages of each GPU subsystem is normalized to 100% in order to give the same weight to each micro-benchmark. Having obtained the FPS of each micro-benchmark from the experiment on multiple devices, we calculate the benchmark scores according to the following equations:

$$score(dev, subsys) = \frac{100}{24} \sum_{res=1}^{3} \sum_{bench=1}^{8} \frac{FPS(dev, bench, res)}{FPS_{MAX}(bench, res)} workload(subsys, res) \tag{1}$$

Table 1
Alignment of each profiler's counters to our workload characterization.

Subsystems	Counters
ARM DS-5 Streamline	
Geometry	JS0_ACTIVE/GPU_ACTIVE
ALU	ARITH_WORDS/GPU_ACTIVE
Texture	TEX_ISSUES/GPU_ACTIVE
ROP	(FRAG_QUADS_EZS_TEST*4 + FRAG_THREADS_LZS_TEST + FRAG_CYCLES_NO_TILE)/GPU_ACTIVE
Memory	Max(L2_EXT_READ_BEATS + L2_EXT_R_BUF_FULL + L2_EXT_RD_BUF_FULL + L2_EXT_AR_STALL, L2_EXT_WRITE_BEATS + L2_EXT_W_BUF_FULL + L2_EXT_W_STALL)/GPU_ACTIVE
nVIDIA Tegra Graphics Debugger	
Geometry	Max(IA SOL, IA Bottleneck)
Texture	Max(TEX SOL. TEX Bottleneck)
ROP	Max(Rasterization SOL, Rasterization Bottleneck, ZCull SOL, ZCull Bottleneck, ROP SOL, ROP Bottleneck)
Memory	L2 Bottleneck
Tessellator	Tessellator Bottleneck
Qualcomm Snapdragon Profiler	
Geometry	% Vertex Fetch Stall
ALU	% Time ALUs Working
Memory	% Stalled on System Memory
AMD GPUPerfStudio	
Geometry	PrimitiveAssemblyBusy
ALU	Sum((VS/DS/HS/GS/PS)ALUBusy)
Texture	Sum((VS/DS/HS/GS/PS)TexBusy)
ROP	DepthStencilTestBusy
Memory	TexUnitBusy-Sum((VS/DS/HS/GS/PS)TexBusy)
Tessellator	TessellatorBusy

Table 2
Performance-independent statistics of our benchmark set. The values in this table were obtained from nVIDIA Tegra Graphics Debugger.

Micro-benchmark	Input primitives/frame	Setup primitives/frame		
		720p	1080p	1440p
Instancing	84,638	606,722	1,011,829	1,311,817
SSAO	100,016	36,013	40,809	43,206
Tiled shading	279,167	38,247	40,688	42,013
CHC++	570,967	64,977	86,432	105,177
Shadow	417,925	85,407	86,954	87,976
OIT	2,614,244	588,189	828,792	987,153
Cube mapping	336,010	114,914	126,425	132,422
BVH	240,082	92,382	111,754	120,118
Tessellation	1	141,497	145,873	147,577

Micro-benchmark	Shaded fragments / frame			Draw calls/frame
	720p	1080p	1440p	
Instancing	477,616	1,074,353	1,908,921	1
SSAO	1,564,025	3,519,817	6,258,140	3
Tiled shading	2,821,176	6,515,601	11,995,884	106
CHC++	1,494,585	2,935,801	5,058,123	1257
Shadow	1,090,964	2,454,591	4,364,315	113
OIT	1,664,881	3,746,105	6,659,751	4
Cube mapping	2,769,889	6,232,271	11,079,509	8
BVH	311,455	578,006	924,559	19
Tessellation	897,784	2,019,504	3,590,389	1

$$overall\_score(dev) = \sum_{subsys=1}^{5} score(dev, subsys) \cdot weight(subsys) \qquad (2)$$

where *dev*, *bench*, *subsys*, and *res* indicate each device, each micro-benchmark, each GPU subsystem, and each screen resolution, respectively. The maximum value of each score is 100 in our benchmark environments because all scores are normalized by the highest frame rates of each experiment. Of course, our benchmark app can result in scores higher than 100 when another test device shows higher frame rates than our test devices. Thus, the score 100 means a calibrated baseline using the five test APs launched in late 2014/early 2015; this score calculation metric is similar to that of Geekbench 3 [11], a well-known CPU benchmark. Note that the last micro-benchmark in the next section (Tessellation) only affects the tessellator performance and its result is not used for calculating the other scores because only AEP-capable GPUs support tessellation. Finally, when we calculate the overall score, we give the same weight (20%) to each subsystem because GPU vendors usually try to configure their GPU architectures to provide balanced performance.

4. Details of the micro-benchmarks

This section deals with details of the micro-benchmarks. We have made nine micro-benchmarks and analyzed each benchmark using nVIDIA Tegra Graphics Debugger. Tables 2 and 3 tabulate performance-independent statistics and the percentage of the instruction count of each shader, respectively.

As shown in the tables, characteristics of each benchmark vary considerably, as a result, each benchmark gives different workloads to GPU subsystems. The number of primitives reduces by culling back-facing, off-screen, or very small (<1 pixel) primitives, but some benchmarks (Instancing and Tessellation) amplify the number of primitives. The screen resolution also affects the GPU workload because higher resolution increases the number of shaded fragments, understandably. BVH update and Tessellation benchmarks additionally include compute and tessellation shader programs, respectively.

In each subsection, we first briefly summarize the used technique for the micro-benchmark, after that, we analyze workload characterization of the micro-benchmark. We also describe our implementation tips, which are how to port the existing base OpenGL/CUDA source code to our OpenGL-ES-based mobile platforms.

4.1. Geometry instancing with skinned animation

Geometry instancing facilitates reusing geometry data multiple times, and it is one of the new features in OpenGL ES 3.0. We implement a micro-benchmark to test the feature. We first use a simple geometry instancing code in OpenGL ES 3.0 Programming Guide [12] as the base code, after that, we additionally implement skinned animation and per-vertex lighting to the vertex shader code. We render 25 elephants, and each elephant object consists of 84K triangles. Thus, total 2.1M triangles are rendered in this scene. As a result, this implementation mainly concentrates on geometry processing.

4.2. Screen-space ambient occlusion

Screen-space ambient occlusion (SSAO) [13,14] is an approximation technique for ambient occlusion, and it is now commonly used in various desktop/console games. This technique approximates the ambient occlusion effect by using depth and normal values of adjacent pixels in the screen space, so it can be implemented using multiple render targets (MRTs) in OpenGL ES 3.0.

As the base source code, we use a sample code with the Dragon Scene (100K triangles) included in OpenGL SuperBible [15]. This code executes two passes, render and SSAO, and a floating-point MRT is used to transfer color data and combined depth/normal data from the render pass to the SSAO pass. However, floating-point render targets are not officially supported in OpenGL ES 3.1. To work around this obstacle, we instead use two-channel RG textures with a floating-point encoding/decoding technique in [16]; the data are dealt with as an 8+8-bit fixed-point representation.

We also add two further optimizations to the original code for higher performance. First, we separate depth and normal data for better texture cache utilization. Second, we use the same size of the render target texture as the initially selected resolution for the

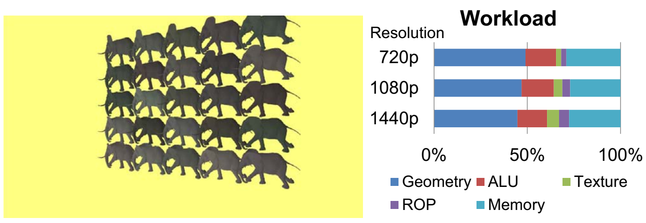

Fig. 2. Benchmark 1 – Geometry instancing with skinned animation.

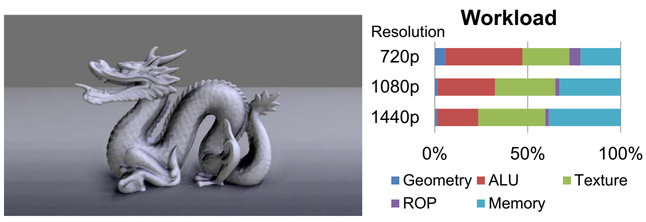

Fig. 3. Benchmark 2 – screen-space ambient occlusion.

Table 3
The percentage of the instruction count of a vertex shader (VS), pixel shader (PS), tessellation controller shader (TCS), tessellation evaluation shader (TES), and a compute shader (CS) in each micro-benchmark. The values in this table were obtained from nVIDIA Tegra Graphics Debugger.

Micro-benchmark	VS (%)			PS (%)			TCS + TES* or CS** (%)		
	720p	1080p	1440p	720p	1080p	1440p	720p	1080p	1440p
Instancing	96.9	94.8	92.6	3.1	5.2	7.4			
SSAO	6.0	2.0	0.7	94.0	98.0	99.3			
Tiled shading	33.5	12.2	11.4	66.5	87.8	88.6			
CHC++	77.7	69.2	61.1	22.3	30.8	38.9			
Shadow	55.8	41.8	31.0	44.2	58.2	69.0			
OIT	41.4	31.2	24.4	58.6	68.8	75.6			
Cube mapping	55.4	43.0	34.0	44.6	57.0	66.0			
BVH**	25.0	21.6	18.5	28.6	41.1	50.8	45.5	37.1	31.2
Tessellation*	0.7	0.6	0.5	20.0	31.9	42.6	79.3	67.6	56.9

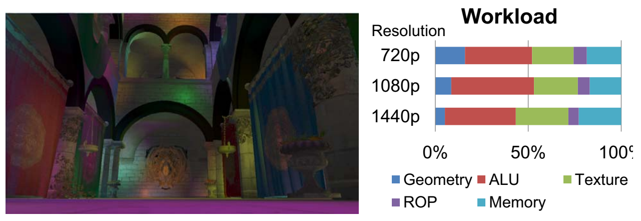

Fig. 4. Benchmark 3 – tiled deferred shading.

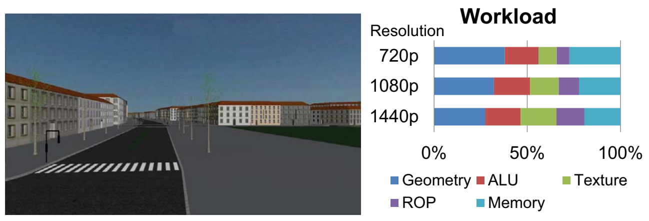

Fig. 5. Benchmark 4 – occlusion culling.

final off-screen render target (720p, 1080p, or 1440p) to reduce unnecessary computations caused by a resolution mismatch. This is possible by using a non-power-of-two texture introduced in OpenGL ES 3.0.

The SSAO shader code consists of somewhat complex ALU operations, e.g. 32 iterations for random sampling are performed per pixel. This random sampling makes an incoherent texture access pattern. Therefore, the SSAO benchmark is compute-intensive at low resolutions and memory-intensive at high screen resolutions as depicted in Fig. 3.

4.3. *Tiled deferred shading*

Tiled deferred shading [17] with many light sources is a representative application of MRTs. The aim of this technique is to cull unnecessary lighting operations. To achieve this, first the geometry is rendered into the fat G-buffers. Next, a screen-space grid is built on a CPU by dividing the screen into multiple tiles. The minimum and maximum depth values of each tile are first calculated (a min–max reduction), after that, all light sources in the scene are assigned to each affected 2.5D (2D + a depth range) grid cell. The final lighting operations are then performed using the G-buffers and the light grid.

For this benchmark, we start from the authors' OpenGL implementation. This code cannot be directly executed on mobile devices due to the two following reasons. First, the grid cell data constructed on a CPU are stored in uniform block objects, but the maximum uniform block size in mobile GPUs is usually from 16K to 64K. This size limit heavily restricts the maximum number of light sources. Second, as the SSAO benchmark, floating-point MRTs cannot be used, so transmission of floating-point depth data from the depth min–max reduction shader to the CPU grid construction is impossible with glReadPixels().

To resolve the above problems, we use the following approaches. First, we store the grid data to textures instead of uniform buffers. Second, we use the floatBitsToUint() function to handle the depth data as unsigned integer data. By decoding a float into an unsigned int, we can easily obtain the original floating point data from the integer data.

The rendered scene in Fig. 4 is CryTek Sponza with 262K triangles and 1024 point light sources. This benchmark is very compute-intensive because multiple render passes with complex fragment shaders are executed. Many texture operations and memory accesses are also required for G-buffer and texture accesses.

4.4. *Occlusion culling*

Occlusion queries are supported in OpenGL ES 3.0 and can be used for occlusion culling in large scenes. Coherent hierarchical culling revisited (CHC++) [18] is a famous technique for occlusion culling on desktop platforms, so we choose this technique to measure the occlusion query performance. The CHC++ algorithm traverses a bounding volume hierarchy (BVH) in a front-to-back order and checks visibility of nodes using the occlusion queries on the nodes. As a result, occluded objects are culled (Fig. 5).

The authors' source code uses fixed-function pipelines, so we need to convert that to a shader-based code for our benchmark because OpenGL ES 2.0 or later no longer supports fixed-function pipelines. We have unified the three render functions (sky rendering, object rendering, and box rendering) to one shader program with branches to reduce the number of shader program changes. We also try to reduce driver overhead by using vertex array objects and reducing unnecessary transfer of the uniform matrix variables when the render state is changed. Furthermore, we disable visible node skipping using temporal coherence in the original CHC++ algorithm because this skipping sometimes produces unfair results when benchmarking different devices; the use of temporal coherence can negatively affect the result in the low frame-rate case.

The test scene in this benchmark is the Vienna scene with 2.5M triangles. This benchmark is bounded by draw operations because the number of draw calls per frame in the scene is around 1200. On the GPU side, various GPU subsystems are quite evenly utilized; the amount of geometry processing decreases by occlusion queries, and the shader program and other function calls are relatively simple.

4.5. Shadow mapping

Shadow mapping is a traditional technique to render shadows on raster-based GPUs. In this technique, depth values in the light's view are first stored in a shadow map. After that, the distance between the light and each point in the camera's view is compared to the distance stored in the shadow map; if the former is higher than the latter value, the point is in shadow because there is an occluder (or occluders) between the light and the point (Fig. 6).

Shadow mapping often suffers from aliasing artifacts, so cascaded shadow mapping [19,20] was proposed to reduce the artifacts. This method uses multi-resolution shadow maps; objects near the camera fetch a higher resolution map, and objects far from the camera fetch a lower resolution map. As the example code of cascaded mapping in the Adreno SDK, we store four shadow maps with different resolutions into a single 3D texture; the support of 3D textures in OpenGL ES 3.0 enables us to do that.

To render dynamic shadows created by both movable light sources and dynamic objects, a hand object (1.5K triangles) moves in the Sponza (66K triangles) scene and the position of a point light source in the scene is continuously changed. We use a large cascaded shadow map (up to 2K×2K), so this benchmark is useful on texturing performance measurement. Also, a lot of depth tests are performed for both the shadow map generation and render passes.

4.6. Order-independent transparency

Order-independent transparency (OIT) is one of the challenging topics on raster-based GPUs because simple alpha blending may result in different images according to the order of submitted geometry. Weighted blended OIT [21] is a lightweight algorithm introduced recently. This approach gives different weights to fragments according to their depth values when these fragments are blended. Only classic blending operations with bounded memory are required for the approach, but the blended images are plausible.

Our OpenGL ES implementation adds two additional pre-Z passes in order to get min/max depth values due to lack of floating-point render targets. The depth range obtained from the pre-Z passes is used in a linear depth weight function modified by us. The range of weighted sum values is set from 0.0 to 1.0 to use RGBA textures for blending. As a result, the workload of this benchmark varies according to the resolution. When we render the Dragon scene with 871K triangles (Fig. 7) at 720p resolution, it shows geometry-intensive feature due to the pre-Z passes; in contrast, rendering the same scene at 1440p shows a fragment-intensive feature due to the increasing number of depth tests, alpha blending, and fragment shading.

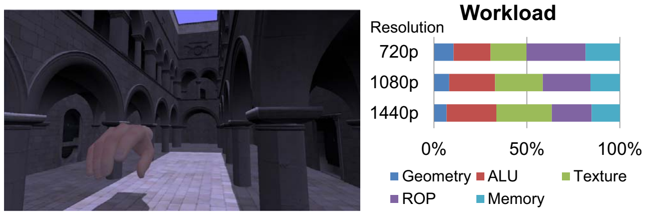

Fig. 6. Benchmark 5 – shadow mapping.

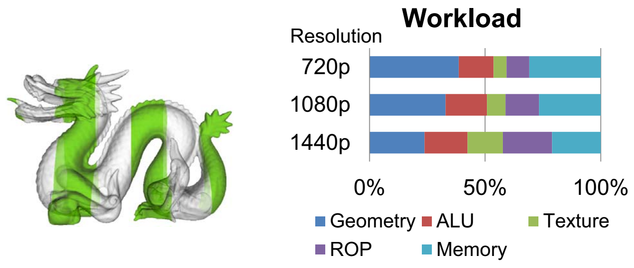

Fig. 7. Benchmark 6 – order-independent transparency.

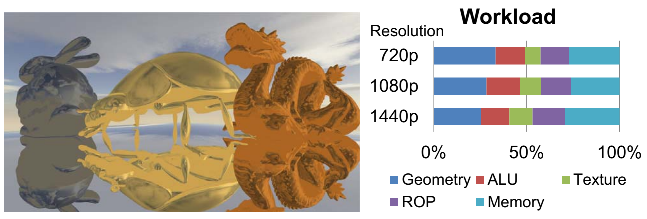

Fig. 8. Benchmark 7 – cube mapping and planar reflection.

4.7. Cube mapping and planar reflection

Cube mapping [22] is a widely used technique for reflection effects. In this method, an environment image is projected onto the six faces of a cube map, and the map is fetched by setting incident and normal vectors when reflective objects are rendered. Because dynamic cube map generation is costly, cube mapping is usually used for representing static environments, such as the sky and mountains. Therefore, in case of dynamic reflection in real-time rendering, planar reflections using the stencil buffer [23] are usually used. This technique renders a reflected image onto a planar object (e.g., a floor) with reflective materials, and restricts the reflected area by testing stencil values (Fig. 8).

We implement the above two techniques together. In the basic cube-mapping sample with a dragon statue (100K triangles) in OpenGL SuperBible, we add two more objects (a bunny statue with 40K triangles and a lady bug statue with 83K triangles) and a mirror floor below the three objects. We also implement a planar reflection code using the stencil buffer for the mirror floor. The resolution of the cube map is 1K×1K.

This mixed implementation is very common for reflections in traditional graphics applications. Thus, this micro-benchmark represents features of traditional feed-forward rendering in simple scenes: multi-pass geometry processing for reflections, simple shader processing, cube-map texturing, and depth/stencil tests.

4.8. BVH update and visualization

OpenGL ES 3.1 introduces compute shaders. The BVH update and visualization benchmark in this subsection is an example of how to integrate compute shaders into existing graphics applications. This benchmark updates the BVH of a scene on the fly and visualizes the BVH on the screen. The BVH is updated in parallel on the GPU in accordance with the level-by-level method described in gProximity [24]. This BVH update can be used for BVH-based collision detection [24], ray tracing [25], and occlusion culling [18] in dynamic scenes (Fig. 9).

This benchmark comprises four stages: the key-frame interpolation, BVH update, object render, and BVH visualization stages. Among the stages, the first and second stages are executed on compute shaders. We had originally unified those two stages, however, we decided to separate those stages to decrease the number of shader storage buffer objects (SSBOs) because some mobile GPUs (Mali and PowerVR) limit the maximum number of SSBOs to only four. The separation decreases the number of SSBOs

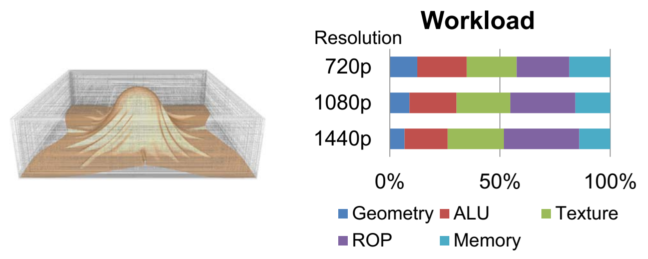

Fig. 9. Benchmark 8 – BVH update and visualization.

by one because the interpolated vertices are input to the BVH update program instead of vertices of two key frames. Even though this separation leads to a performance penalty due to increased memory read/write operations, the benchmark is now able to be executed if the maximum number of SSBOs is limited. In the BVH visualization stage, we perform alpha blending for better visibility; if alpha blending is disabled, the objects can be hidden behind lines of BVH nodes.

For this benchmark, we render the Cloth Simulation scene (92K triangles) with key-frame interpolation. This benchmark is not only compute-intensive (for BVH update) but also ROP-intensive (for BVH visualization) because multiple lines can be blended over each other and a lot of depth tests are also required between lines and the object in the BVH visualization stage. Even though textures are not used in this scene, texturing is also heavily performed on Mali GPUs to handle SSBOs, in contrast to AMD GPUs.

For workload characterization of this benchmark, profiling results from the Tegra Graphics Debugger are not used; our characterization requires GPU subsystem utilization, but the debugger does not show the utilization when compute shaders are processed because it does not operate in the OpenGL mode but operate in the OpenCL/CUDA mode for the case.

4.9. *Displacement mapping with tessellation*

The tessellation shader is one of the main features of AEP. Between the tessellation control and evaluation shaders, a hardware tessellator is located, and we test its performance in this displacement mapping benchmark. The source code and scene for this benchmark are from OpenGL SuperBible; a terrain scene (Fig. 10) is rendered by using displacement mapping and tessellation. The process is briefly described as follows. First, a 64×64 patch is input. After that, the tessellation levels are calculated in the tessellation control shader at once. Next, the result from the tessellator is then output to the tessellation evaluation shader, and displacement mapping is performed in the shader using a height map. Finally, texture mapping is performed in the fragment shader.

We slightly change the setting of the original code and scene. First, to stress out the tessellation workload, we increase the tessellation factor by a factor of eight. Second, we compress the height map and the color map in the scene using EAC and ETC2, respectively. This texture compression results in decreased memory traffic and slightly increased performance. For the workload characterization of this micro-benchmark, only nVIDIA and AMD's profilers, which have tessellator utilization counters, are used.

5. Experiments and results

5.1. *Experimental setup*

For our experiments, we chose five different devices including different APs launched in late 2014 to early 2015 in Table 4; LG G3 Screen is a mid-end smartphone, LG G4 and Samsung Galaxy Note 4 are flagship smartphones, ThinkWare iNaviTab XD10 Duo is a dual boot tablet (Android and Windows 10), and nVIDIA Shield Tablet is a gaming tablet. Each AP has a GPU from different vendors, so we can compare different GPU architectures from the chosen devices. We have updated the Android version of each device to the latest version, so all the devices now support OpenGL ES 3.1. Among them, Adreno 418, Mali T760, HD Graphics, and Kepler GK20A support AEP as well.

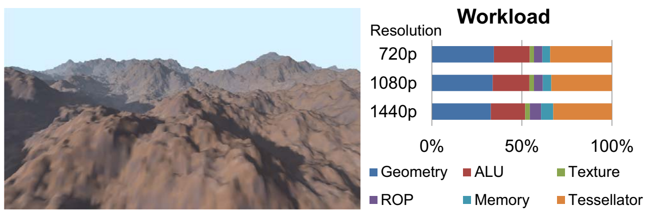

Fig. 10. Benchmark 9 – displacement mapping with tessellation.

Before we introduce the benchmark results in the next subsection, we briefly compare the GPU architectures. First, we compare the architectures in terms of the rendering approach. PowerVR is based on Imagination Technologies' unique tile-based deferred rendering architecture (TBDR). This architecture is basically the same as the tile-based rendering architecture (TBR), which includes local tile memory to reduce external memory accesses for per-fragment operations. However, in contrast to TBR, TBDR defers fragment shading operations until the visibility of the fragments is known for effective overdraw reduction. Mali is based on TBR, and its forward pixel kill technique reduces the amount of operations for further rejected pixels. Adreno includes Flexrender which is a technique for automatic switching between TBR and immediate-mode rendering (IMR); TBR is basically used for normal scenes to decrease memory traffic and IMR is used for scenes with low-depth complexity or high polygon counts to decrease tiling overhead. The mode is automatically chosen by the driver. Intel HD Graphics (with 12 execution units) is the lowest version of Intel's 8th generation HD Graphics series for low-power ultra-mobile devices (smartphones and tablets). Thus, this GPU architecture inherits the feature of integrated HD graphics series based on IMR. nVIDIA Kepler GK20A was also originated from the desktop Kepler architecture based on IMR, and some optimizations were added in order to increase power efficiency. To reduce the required memory bandwidth, depth compression units and hierarchical depth buffers are included in GK20A in common with the desktop Kepler architecture.

Second, we compare the ALUs in the GPU architectures. The single instruction, multiple threads (SIMT) architecture with scalar ALUs is used in PowerVR (later 6 series), Adreno (later 300 series), and Kepler. In contrast, Mali is based on the VLIW architecture with a mix of vector and scalar ALUs. Scalar processing is robust for the width of the data, but branch divergence can decrease ALU utilization. Vector processing usually achieves high ALU utilization in traditional graphics applications and have less penalty from branch divergence. However, a lot of scalar data can decrease ALU utilization because the ALU efficiency heavily depends on the compiler in that case. Interestingly, Intel HD Graphics supports both modes; vertices are arranged in the array of structures (AOS) form for SIMD4 or SIMD4x2 modes as vector processing, and pixels and compute data are arranged in the structure of arrays (SOA) form for SIMD8 or SIMD16 modes as scalar processing.

Third, Adreno 418, HD Graphics, and Kepler GK20A include hardware tessellators to support AEP. In contrast, ARM Mali T760 supports tessellation via compute shaders, and new AEP-support drivers (r7p0 or higher) have been recently released for this

Table 4
Benchmarking devices. The GFXBench results (unit: FPS) were obtained from 1080p off-screen tests.

Device	LG G3 Screen	LG G4	Samsung Galaxy Note 4	Thinkware iNaviTab XD10 Duo	nVIDIA Shield Tablet
AP	LG Nuclun LG7111	Qualcomm Snapdragon 808	Samsung Exynos 5433	Intel Atom X5-Z8300	nVIDIA Tegra K1
Process	28 nm	20 nm	20 nm	14 nm	28 nm
Memory	32-bit dual-channel 756 MHz LPDDR3	32-bit dual-channel 933 MHz LPDDR3	32-bit dual-channel 825 MHz LPDDR3	64-bit single-channel 800 MHz DDR3L RS	32-bit dual-channel 933 MHz LPDDR3
GPU	Imagination Technologies PowerVR G6430	Qualcomm Adreno 418	ARM Mali T760MP6	Intel HD Graphics (12 EUs)	nVIDIA Kepler GK20A
Clock frequency	460 MHz	600 MHz	700 MHz	500 MHz	852 MHz
Driver version	1.4	136.0	r7p0	4.51.85-R	361.0
GFXBench Manhattan 3.1	4.8	11.4	11.5	11.4	23.1
GFXBench Car Chase	N/A	6.7	7.1	7.4	14.9

Table 5
Benchmark results (unit: FPS). The bold fonts represent the highest values in each benchmark.

Micro-benchmark	720p resolution					1080p resolution					1440p resolution				
	PVR G6430	Adreno 418	Mali T760MP6	HD Graphics	Kepler GK20A	PVR G6430	Adreno 418	Mali T760MP6	HD Graphics	Kepler GK20A	PVR G6430	Adreno 418	Mali T760MP6	HD Graphics	Kepler GK20A
Instancing	5.0	16.5	21.3	20.5	**22.8**	4.8	13.8	21.0	19.1	**22.5**	5.0	12.3	20.6	17.8	**23.2**
SSAO	11.3	35.1	28.6	27.9	**66.0**	6.1	13.5	14.0	12.8	**25.8**	3.9	5.8	6.5	6.8	**10.5**
Tiled shading	5.6	10.6	6.4	11.3	**20.4**	3.8	7.7	4.2	6.7	**10.1**	2.6	3.6	3.2	4.0	**6.8**
CHC++	4.3	6.7	6.2	22.6	**25.0**	3.7	6.4	5.7	21.2	**23.5**	3.1	5.4	5.4	18.7	**22.0**
Shadow	8.3	27.4	36.1	41.0	**96.2**	6.5	22.0	26.8	29.2	**75.7**	4.7	17.3	19.9	21.4	**59.0**
OIT	10.2	22.6	27.9	25.2	**44.6**	7.9	18.2	25.2	17.8	**36.2**	6.5	14.1	20.3	13.0	**26.4**
Cube mapping	32.7	69.4	73.2	51.8	**73.6**	24.6	47.7	64.7	35.6	**69.1**	19.5	34.6	**56.3**	26.5	47.9
BVH	22.6	45.6	31.5	48.8	**66.3**	16.4	31.2	20.1	34.2	**49.4**	12.8	24.4	13.8	26.6	**37.7**
Tessellation	N/A	21.3	15.0	34.4	**63.6**	N/A	19.6	14.6	29.3	**60.3**	N/A	17.9	14.8	25.4	**56.9**

tessellation support. The comparison of these different approaches will be described in the next section.

5.2. Results and analysis

Table 5 and Fig. 11 summarize the results of our experiments. First, Table 5 includes the measured frame rates obtained from the devices in Table 4. We skip half of eglSwapBuffers calls if the frame rates are higher than 60 FPS. According to Eqs. (1) and (2) in Section 3, the scores in Fig. 11 were calculated from the results in Table 5.

The overall scores show that the results from L-Bench is similar to the results of GFXBench 4.0 based on OpenGL ES 3.1/AEP in Table 4. Both results from GFXBench and L-Bench indicate that PowerVR G6430 shows the lowest performance among the test GPUs, Adreno 418, Mali T760MP6, and Intel HD Graphics are similar grade GPUs, and Kepler GK20A provides the highest performance. It means that our micro-benchmark choice and score calculation metric are reasonable. Note that a higher score does not mean a better GPU architecture because the target platforms (smartphones vs tablets), the process technologies (14–28 nm), and memory clock frequencies are different between the test devices as described in Table 4. Generally, the tablet form factor enables the possibility to run at higher clock speeds for longer periods of time than compared to a more thermally constrained

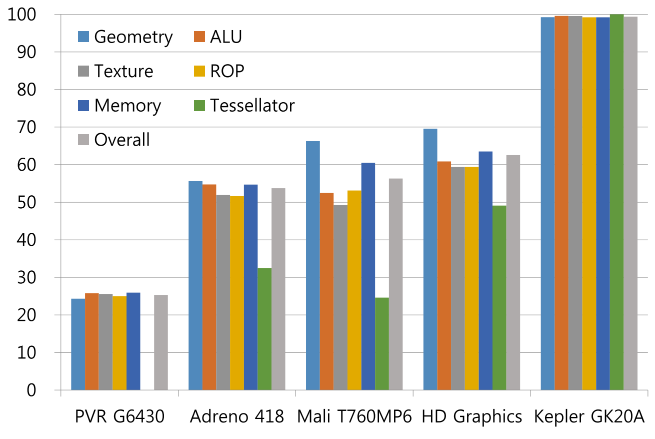

Fig. 11. Benchmark scores.

phone form factor, so tablets usually have performance advantages compared to smartphones. To prevent thermal mitigation situations on our experiments, our benchmark has a limited total execution time.

We now analyze the results of our experiments. First of all, Kepler GK20A achieves the highest frame rates in most cases. Therefore, this GPU gets the highest scores in all sections as depicted in Fig. 11, and this result is the reference point for further analysis of other GPUs.

PowerVR G6430 shows approximately a quarter of Kepler GK20A's performance. Its TBDR architecture does not result in very distinguished speedup in our benchmarks; TBDR is beneficial for traditional forward shading in scenes with high depth complexity and no alpha blending, but a number of our micro-benchmarks use deferred shading with manual pre-Z stages, alpha blending, or occlusion culling.

Adreno 418 shows approximately half of the performance compared to Kepler GK20A. This GPU provides balanced performance between subsystems, and is generally advantageous at low resolution compared to Mali T760MP6. Additionally, relative tessellation performance of Adreno 418 is 32% compared to Kepler GK20A.

Mali T760MP6 achieves high performance when triangle counts are high (Geometry instancing and Weighted OIT) or scenes are simple (Cube mapping). On the other hand, this GPU can suffer from low ALU utilization when the shader has many scalar ALU operations as the BVH update benchmark. Additionally, its tessellation performance is lower than that of Adreno 418 and HD Graphics; this result shows that compute-shader-based S/W tessellation can be area-efficient but is likely to be hard to provide high tessellation performance.

Intel HD Graphics gets relatively higher overall scores (6–8 points) than Adreno 418 and Mali T760MP6 in our benchmark. Its hybrid vector/scalar processing results in high frame rates in both VS-oriented (Geometry instancing) and CS-oriented (BVH update) benchmarks. Additionally, its driver handles the CHC++ benchmark very well, as a result, the GPU gets a considerable overall score gain from the micro-benchmark. This is an interesting result; even though occlusion queries are supported from OpenGL ES 3.0, tile-based mobile GPU architectures (PowerVR, Adreno, and Mali) usually cannot effectively perform occlusion queries due to a difficulty in batching multiple draw calls for a tiling process until the query result becomes available. In contrast, HD Graphics and Kepler, derived from IMR-based desktop GPU architectures, do not carry heavy performance penalty when occlusion queries are made. Thus, we feel the necessity of a new occlusion culling algorithm dedicated for tile-based GPUs to more effectively use occlusion queries, but it is out of scope of this paper.

6. Conclusions, limitations, and future work

We have presented a benchmark set based on OpenGL ES 3.1/AEP, called L-Bench. This benchmark set provides a hint to analyze GPU performance by combining results of mid-sized micro-benchmarks. We have also tested five mobile devices with L-Bench.

We believe L-Bench can be beneficial to various types of developers and users. First, GPU architects can understand bottlenecks of their GPUs and improve their GPU architectures. Second, AP designers can exploit L-Bench to evaluate performance of GPU IP. Third, mobile graphics programmers may utilize the OpenGL ES implementation tips when they try to include the graphical effect in Section 4 in their applications. Finally, end-users can compare GPU performance of mobile devices before their final purchase decision.

L-Bench is not a perfect tool for GPU benchmarking, and we would like to resolve its limitations as future work. First, the current version of L-Bench only measures performance, so it is hard to know power efficiency from L-Bench alone. Additionally, measurement of GPU power efficiency is difficult because temperatures and power consumption on devices are affected by various factors (e.g., CPUs, GPUs, displays, thermal management policies, etc.). If L-Bench is extended with existing energy measurement frameworks [7,9], we think the results from the benchmark suite can be more fruitful. Second, there is some room for improvement in terms of reliability. For example, profiling and measurement at a draw-call level will provide more accurate results than the current frame-level analysis. Additionally, if our benchmark suite can collect data from each GPU vendor's profiling tool on the fly, we will be able to get useful data for additional analysis such as potential bottlenecks, memory traffic, etc. Third, as mentioned in Section 1, our current implementation excludes the use of vendor-specific extensions even though some apps can use those extensions for better performance on specific devices. As future studies, we are interested in how much representative OpenGL ES extensions affect each micro-benchmark (e.g., the use of pixel local storage [26] for MRT-based benchmarks). Finally, our benchmark is implemented on OpenGL ES and Android platforms. If we extend the current version of L-Bench to a cross-platform benchmark suite, we can execute the app on other platforms, such as Apple iOS. We are also interested in porting our OpenGL ES code to newer graphics APIs, such as DirectX 12, Vulkan, and Metal.

Acknowledgments

We would like thank Seong-Wook Min, Sunho Ki, Youngduke Seo, Hanseng Choi, and Jinhong Park for supporting the test devices. Byeongjun Choi and Jin-Woo Kim helped us with the video recording. The used scenes are courtesy of Doug L. James and Christopher D. Twigg (Elephant); Frank Meinl (Crytek Sponza); Marko Dabrovic (Sponza); Roel Reijerse (the cube map); the Stanford 3D Scanning Repository (Bunny and Dragon); UNC Dynamic Scene Benchmarks (Cloth-ball simulation); and the Utah 3D Animation Repository (Hand). The Ladybug and terrain scenes are included in OpenGL Superbible, and the Vienna scene is included in the CHC++ source code.

Appendix A. Supplementary data

We attach the benchmark apk file as a supplementary material. To properly execute the benchmark app, both Android 5.0+ and OpenGL ES 3.1+ are required.

Supplementary data associated with this article can be found in the online version at http://dx.doi.org/10.1016/j.cag.2016.09.002.

References

[1] Kishonti Informatics. GFXBench 4.0. ⟨https://gfxbench.com/⟩; 2016.
[2] Basemark Ltd. Basemark ES 3.1. ⟨https://www.basemark.com/product-catalog/basemark-es-3-1/⟩; 2015.
[3] Futuremark Corporation. 3DMark Sling Shot Benchmark. ⟨http://www.futuremark.com/benchmarks/3dmark/android⟩; 2015.
[4] Standard Performance Evaluation Corporation. SPECviewperf 12. ⟨https://www.spec.org/gwpg/gpc.static/vp12info.html⟩; 2013.
[5] Antochi I, Juurlink B, Vassiliadis S, Liuha P. GraalBench: a 3D graphics benchmark suite for mobile phones. In: ACM SIGPLAN notices - LCTES '04; vol. 39. 2004, p. 1–9.
[6] Arnau JM, Parcerisa JM, Xekalakis P. TEAPOT: a toolset for evaluating performance, power and image quality on mobile graphics systems. In: Proceedings of international conference on supercomputing (ICS '13), 2013. p. 37–46.
[7] Ma X, Deng Z, Dong M, Zhong L. Characterizing the performance and power consumption of 3D mobile games. Computer 2013;4:76–82.
[8] Johnsson B, Ganestam P, Doggett M, Akenine-Möller T. Power efficiency for software algorithms running on graphics processors. In: Proceedings of high-performance graphics (HPG '12), 2012. p. 67–75.
[9] Johnsson B, Akenine-Möller T, Sathe R, Foley T, Salvi M, Andersson M, et al. Measuring per-frame energy consumption of real-time graphics applications. J Comput Graph Technol 2014;3:1.
[10] Ström J, Pettersson M. ETC 2: texture compression using invalid combinations. In: Proceedings of graphics hardware, 2007. p. 49–54.
[11] Primate Labs. Geekbench 3. ⟨https://www.primatelabs.com/geekbench/⟩; 2013.

[12] Ginsburg D, Purnomo B, Shreiner D, Munshi A. OpenGL ES 3.0 programming guide. 2nd ed. Addison-Wesley Professional; 2014 p. 169-172.
[13] Mittring M. Finding next gen: Cryengine 2. In: ACM SIGGRAPH 2007 courses, 2007. p. 97–121.
[14] Shanmugam P, Arikan O. Hardware accelerated ambient occlusion techniques on GPUs. In: Proceedings of the 2007 symposium on interactive 3D graphics and games. ACM; 2007. p. 73–80.
[15] Sellers G, Wright RS, Haemel N. OpenGL SuperBible: comprehensive tutorial and reference. 7th ed. Addison-Wesley; 2015, 624-663.
[16] Gerasimov P. Omnidirectional shadow mapping. In: GPU Gems: programming techniques, tips, and tricks for real-time graphics. Oxford University Press; 2004. p. 193–204 [chap. 12].
[17] Olsson O, Assarsson U. Tiled shading. J Graph GPU Game Tools 2011;15 (4):235–51.
[18] Mattausch O, Bittner J, Wimmer M. CHC++: coherent hierarchical culling revisited. Comput Graph Forum (EUROGRAPHICS 2008) 2008;27(2):221–30.
[19] Engel W. Cascaded shadow maps. In: ShaderX5: advanced rendering techniques. ShaderX series. Charles River Media, Inc; 2007. p. 197–206 [chap. 4].
[20] Dimitrov R. Cascaded shadow maps. Developer Documentation. NVIDIA Corp; 2007, http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf.
[21] McGuire M, Bavoil L. Weighted blended order-independent transparency. J Comput Graph Technol 2013;2(2):122–41.
[22] Greene N. Environment mapping and other applications of world projections. IEEE Comput Graph Appl 1986;6(11):21–9.
[23] McReynolds T, Blythe D, Fowle C, Grantham B, Hui S, Womack P. Programming with OpenGL: Advanced rendering. In: SIGGRAPH, vol. 97, 1997. p. 144–153.
[24] Lauterbach C, Mo Q, Manocha D. gProximity: hierarchical GPU-based operations for collision and distance queries. Comput Graph Forum (EUROGRAPHICS 2010); 2010;29(2):419–28.
[25] Nah JH, Kim JW, Park J, Lee WJ, Park JS, Jung SY, et al. HART: a hybrid architecture for ray tracing animated scenes. IEEE Trans Vis Comput Graph 2015;21(3):389–401.
[26] Bjørge M, Martin S, Kakarlapudi S, Fredriksen JH. Efficient rendering with tile local storage. In: ACM SIGGRAPH 2014 talks, 2014. p. 51:1.