UCI SAMSUNG PROJECT

Phase 2 Deliverable:
Report on Microbenchmarks

Period of Activity: 03/15/2015 - 06/14/2015
Due Date: 06/30/2015

University of California, Irvine
Principal Investigator: Nikil Dutt, dutt@ics.uci.edu
Graduate Student Researcher: Jurngyu Park, jurngyup@ics.uci.edu
# CONTENTS

I  **Workload Characterization and Micro-benchmarks**  3  
   I-A Workload Characterization ........................................... 3  
   I-B Design of Micro-benchmarks ........................................ 4  

II  **Experimental Setup and Methodology**  5  
   II-A Experimental Platform ............................................. 5  
   II-B ARM DS-5 Streamline Profiler ..................................... 5  
   II-C A Case Study: Developing and Profiling of mb-VerM ........... 6  

References  7
Phase 2: Micro-benchmark Development

In this phase, we will develop custom micro-benchmarks that are designed to stress different components of the abstracted mobile GPU hardware pipeline (e.g., Vertex Fetching, Vertex Shading, Texture Fetching and Fragment Shading unit) separately for graphics workload characterization. In particular, we will develop micro-benchmarks from the perspectives of GPU memory-, GPU computation- and CPU computation-bound workloads.

I. Workload Characterization and Micro-benchmarks

We now present the design of our micro-benchmarks that can be used to analyze the correlation between performance, power, and energy efficiency of the mobile platform by stressing different components of the mobile graphics pipeline stages. Accordingly, we first categorize our micro-benchmarks from the perspectives of GPU memory-, GPU computation- and CPU computation-bound workloads.

A. Workload Characterization

GPU memory: Here we target GPU vertex memory fetch and texture memory fetch. Vertex memory contains vertex attributes such as position information. Vertices are the foundation of graphics objects hence they are necessary in all graphics rendering. On the other hand, texture memory is typically used in games/applications that require rich graphic details in objects and scenes. External memory access is very expensive to GPU performance due to longer memory latencies. Additionally, the GPU consumes significantly more power during the memory access. Thus, the GPU memory is an important component for characterization.

GPU computation: Vertex shaders and fragment shaders are considered in this category. Starting from OpenGL ES 2.0, programmable shaders are supported in mobile platforms. The shader programs allow developers to control how the graphics objects are rendered. Since there may be a variety of shaders, we need to address their influence on performance and power.

CPU computation: Mobile applications can access hardware-accelerated graphics through OpenGL ES commands. These commands are generated by algorithms run within applications to determine what to display in the next frame. Graphics rendering includes the CPU as a whole, however, a complex programming model or complicated algorithms might reduce the FPS (resulting in quality loss) if excessive CPU time is spent on computation of OpenGL ES commands. Therefore, we also need to analyze the workload executed on CPU to understand its effect on graphics rendering.
B. Design of Micro-benchmarks

TABLE I: Micro-benchmarks and their Pipelined Workloads

<table>
<thead>
<tr>
<th>MBs</th>
<th>Workload Factor</th>
<th>CPU</th>
<th>Vertex Fetch</th>
<th>Vertex Shaders</th>
<th>Texture Fetch</th>
<th>Fragment Shaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>mb-VerM</td>
<td>Load/Store pipe (or Ext. R/W) cnt.</td>
<td>Only GL API</td>
<td>Num. of Ver.</td>
<td>Minimized (or None)</td>
<td>Minimized (or None)</td>
<td>Fixed</td>
</tr>
<tr>
<td>mb-TexM</td>
<td>Texture pipe (or Ext. R/W) cnt.</td>
<td>Only GL API</td>
<td>Fixed</td>
<td>Minimized</td>
<td>Texture Img. Size</td>
<td>Minimized</td>
</tr>
<tr>
<td>mb-VerSh</td>
<td>Arithmetic pipe cnt.</td>
<td>Only GL API</td>
<td>Fixed</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>mb-FragSh</td>
<td>Arithmetic pipe cnt.</td>
<td>Only GL API</td>
<td>Fixed</td>
<td>Minimized</td>
<td>None</td>
<td>Frag. Sh. Prog.</td>
</tr>
<tr>
<td>mb-App</td>
<td>CPU Exec. Time</td>
<td>GL API + Loop</td>
<td>Fixed</td>
<td>Minimized</td>
<td>None</td>
<td>Minimized</td>
</tr>
</tbody>
</table>

TABLE I summarizes the micro-benchmarks, the workload factor (i.e., which aspect of the mobile GPU pipeline is stressed), and the corresponding effects on the mobile GPU stages (i.e., Vertex Fetch, Vertex Shaders, Texture Fetch and Fragment Shaders). We briefly describe each micro-benchmark below:

*mb-VerM*: stresses the vertex fetcher (start of the GPU pipeline) by giving different number of vertices to vary the amount of vertex data read from main memory. The workload of following pipeline stages might be affected due to different amounts of vertex data. Therefore, in order to minimize additional workload in the following stages, a simple vertex shader program is used into the later pipeline stages where the vertices will be discarded by the clipping test.

*mb-TexM*: targets the amount of texture memory read from main memory. Both vertex and fragment shaders should support texture mapping in order to do texture fetch. The amount of texture memory read is varied by applying different sizes of texture images. Since we aim to evaluate only the amount of texture memory fetch, we use only fixed vertex data with fixed texture coordinates (instead of diverse texture mapping techniques).

*mb-VerSh*: targets vertex shaders that perform coordinate transformation of vertex data. *mb-VerSh* stresses vertex shaders by changing the number of vertex instructions with a vertex shader program. The program computes the lighting effect on vertex color and repeats itself redundantly with a different number of iterations to increase the workload of vertex shader. *mb-VerSh* also adopts the same strategy as *mb-VerM* to disable the following pipeline stages.

*mb-FragSh*: stresses the Fragment shaders that manipulate data in each fragment such as color and texture. As with Micro-benchmark *mb-VerSh*, we apply a fragment shader program that supports lighting effect on fragment color, and increase the number of fragment instructions by repeated execution of this program to stress the fragment shader.

*mb-App*: simulates the higher application workload and the OpenGL ES API calls at the application level. The workload is stressed by modifying the time spent by CPU with a configurable code stub. In order to overcome varying execution times (due to effects of the CPU frequency governor and load balancer in Linux), we apply a fixed CPU frequency to emulate a consistent execution time for the same *mb-App* configuration.
II. EXPERIMENTAL SETUP AND METHODOLOGY

A. Experimental Platform

Table II summarizes our hardware and software platform configurations used in our experiments. We used Odroid- XU3 [1] development board installed with Android 4.4.2 and Linux 3.10.9. As shown in Figure 1, the board is connected with a touch screen and a host PC where we collect the experimental data and profiling results. The board is equipped with four TI INA231 power sensors measuring the power consumption of big CPU cluster(CPU-bc), little CPU cluster(CPU-lc), GPU and memory respectively.

**Table II: Platform Configuration**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>ODROID-XU3</td>
</tr>
<tr>
<td>SoC</td>
<td>Samsung Exynos5422</td>
</tr>
<tr>
<td>CPU</td>
<td>Cortex-A15 2.0Ghz quad core and Cortex-A7 quad core CPUs,</td>
</tr>
<tr>
<td>GPU</td>
<td>Mali-T628 MP6, 543MHz</td>
</tr>
<tr>
<td>System RAM</td>
<td>2Gbyte LPDDR3 RAM at 933MHz</td>
</tr>
<tr>
<td>Mem. Bandwidth</td>
<td>up to 14.9GB/s</td>
</tr>
<tr>
<td>OS/Platform</td>
<td>Android 4.4.2</td>
</tr>
<tr>
<td>Linux Kernel</td>
<td>3.10.9</td>
</tr>
</tbody>
</table>

![Fig. 1: Experimental Setup.](image)

As shown in Table III, CPU supports cluster-based DVFS at nine frequency levels in the big cluster and seven frequency levels in the little cluster, and GPU supports operating at six frequency levels by default.

**Table III: Frequency Configuration**

<table>
<thead>
<tr>
<th>Component</th>
<th>levels</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-bc (Ghz)</td>
<td>9</td>
<td>1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0</td>
</tr>
<tr>
<td>CPU-lc (Ghz)</td>
<td>7</td>
<td>1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6</td>
</tr>
<tr>
<td>GPU (Mhz)</td>
<td>6</td>
<td>177, 266, 325, 420, 480, 543</td>
</tr>
</tbody>
</table>

B. ARM DS-5 Streamline Profiler

ARM Streamline Performance Analyzer [2] is a system-wide visualizer and profiler for systems running ARM Linux or Android native applications and libraries. Combining an ARM Linux kernel driver, target daemon, and a graphical user interface, it transforms system trace and sampling data into reports that present the data in both visual and statistical forms. Streamline leverages hardware performance counters with kernel metrics to provide an accurate representation of system resources.

As shown in Figure 3, we can see at-a-glance how our system is performing with the Timeline view. With charts showing the values of each counter. By default, Streamline will collect a predefined set of hardware performance counters, but we are free to add or remove counters to suit our needs.
C. A Case Study: Developing and Profiling of mb-VerM

In this part, we introduce a case study of developing and profiling of mb-VerM, and this kind of methodology will be applicable for all micro-benchmarks. Firstly, we developed the mb-VerM micro-benchmark according to the design of mb-VerM as shown in Figure 2. We stressed the vertex fetcher by giving different number of vertices (i.e., 240 vs. 24,000) to vary the amount of vertex data from main memory.

Secondly, we calculated memory throughput [3] using Equation (1) after profiling mb-VerM Low and mb-VerM High using ARM DS-5 Streamline profiler. In order to calculate memory throughput, multiplication of bus size (i.e., 16 byte) and the number of external read (i.e., 5,356,320 in mb-VerM Low and 59,830,347 in mb-VerM High) and write beats (i.e., 1,731,652 in mb-VerM Low and 2,700,932 in mb-VerM High) was used, as shown in Figure 3.

\[ \text{Bandwidth Throughput} = \frac{(\text{number of ext. read beats} + \text{number of ext. write beats}) \times \text{bus size}}{\text{second}} \]  

Finally, Figure 4 shows the results of mb-Verm Low and mb-VerM High. When we give different number of vertices of 240 and 24,000, the results were FPS 60 and 22.7, Power 868 mW and 2042 mW, and memory throughput 113 MB/s and 1000 MB/s in mb-VerM Low and mb-VerM High respectively. In summary, we can clearly observe the workload variation by changing number of vertices on mb-VerM through this case study.
REFERENCES

