Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU

Jose-Maria Arnau
Computer Architecture Department
Universitat Politecnica de Catalunya
jarnau@ac.upc.edu

Joan-Manuel Parcerisa
Computer Architecture Department
Universitat Politecnica de Catalunya
jmanel@ac.upc.edu

Polychronis Xekalakis
Intel Labs
Intel Corporation
polychronis.xekalakis@intel.com

***Abstract*—Perhaps one of the most important design aspects for smartphones and tablets is improving their energy efficiency. Unfortunately, rich media content applications typically put significant pressure to the GPU's memory subsystem. In this paper we propose a novel means of dramatically improving the energy efficiency of these devices, for this popular type of applications. The main hurdle in doing so is that GPUs require a significant amount of memory bandwidth in order to fetch all the necessary textures from memory. Although consecutive frames tend to operate on the same textures, their re-use distances are so big that to the caches fetching textures appears to be a streaming operation. Traditional designs improve the degree of multi-threading and the memory bandwidth, as a means of improving performance. In order to meet the energy efficiency standards required by the mobile market, we need a different approach. We thus propose a technique which we term Parallel Frame Rendering (PFR). Under PFR, we split the GPU into two clusters where two consecutive frames are rendered in parallel. PFR exploits the high degree of similarity between consecutive frames to save memory bandwidth by improving texture locality. Since the physics part of the rendering has to be computed sequentially for two consecutive frames, this naturally leads to an increase in the input delay latency for PFR compared with traditional systems. However we argue that this is rarely an issue, as the user interface in these devices is much slower than those of desktop systems. Moreover, we show that we can design reactive forms of PFR that allow us to bound the lag observed by the end user, thus maintaining the highest user experience when necessary. Overall we show that PFR can achieve 28% of memory bandwidth savings with only minimal loss in system responsiveness.**

***Keywords*—Mobile GPU, Memory bandwidth, Rendering**

I. Introduction

Increasing the energy efficiency of mobile GPUs has recently attracted a lot of attention from the architecture community [1]–[8]. Being battery-operated devices, the power/performance constraints for GPUs targeting tablets or smartphones are very different from those of conventional desktop GPUs [9]. Interestingly enough with each new tablet/smartphone generation, the need for visually compelling graphics increases. Today, mobile devices excel in providing very rich user interfaces with media content powered by specialized graphics hardware. Not surprisingly, previous studies have identified the GPU as one of the main battery consumers on a smartphone for many use cases [2], [10].

Focusing on the how to improve the GPU, recent work in [9] revealed that a large fraction of its power can be attributed to external off-chip memory accesses to system RAM. As noted by [11], [12], most of these accesses fetch textures. Figure 1 depicts the memory bandwidth usage on a mobile GPU, similar to ARM Mali [13], for a set of commercial Android games. Our numbers clearly support the prior claims and show that 62% of these memory accesses can be directly attributed to texture data. Ideally removing these accesses would result in significant energy savings, while also improving performance substantially.

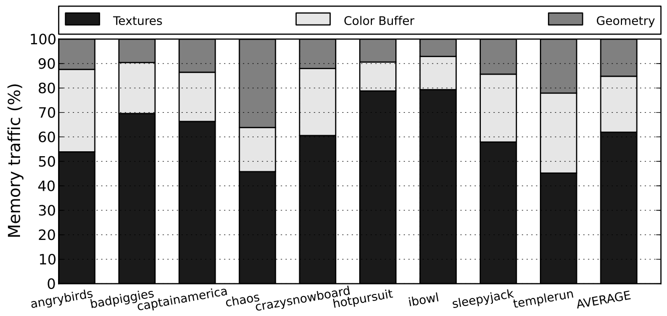


Fig. 1: Memory bandwidth usage on a mobile GPU for a set of commercial Android games. On average 62% of the bandwidth to system memory is employed for fetching textures.

Focusing on the texture data used by successive frames, we realized that there exists a large degree of reuse across frames. As shown in Figure 2 consecutive frames share 96% of the texture addresses requested on average for a set of Android games. Hence, the same textures are fetched frame after frame, but the GPU cannot exploit these frame-to-frame re-usages due to the huge size of the texture dataset. This is confirmed by inspecting the average re-use distance of the memory accesses (Figure 3). Contrary to common belief that GPUs need to deal with the streaming memory behavior of graphical applications, we argue it is perhaps more efficient to change the rendering model leading to this memory behavior.

In this paper we propose Parallel Frame Rendering (PFR), a novel technique for improving texture locality on a mobile GPU. In PFR the GPU is split in two clusters where two consecutive frames are rendered in parallel. By using this organization each texture is read from memory once and employed for rendering two successive frames. Therefore, textures are fetched from main memory just once every two frames instead of being fetched on a frame basis as in conventional GPUs.

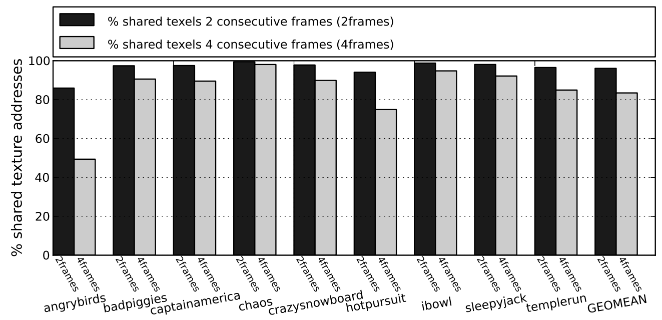


Fig. 2: Average percentage of shared texels (texture elements) between consecutive frames. Mostly the same texture dataset, 96% on average, is employed from frame to frame since consecutive frames tend to be very similar.

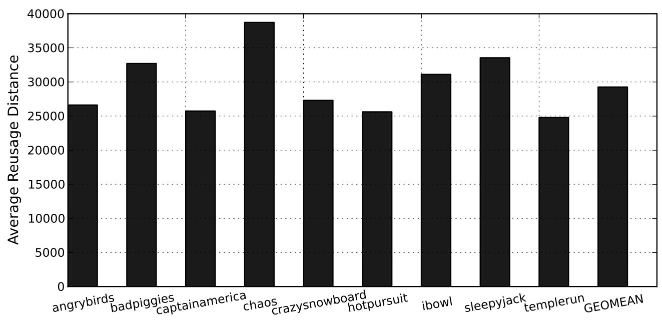


Fig. 3: Average block re-use distances in the L2 cache for several Android commercial games, computed as number of distinct block addresses referenced between two accesses to the same block.

If perfect texture overlapping between both GPU clusters is achieved then the amount of memory bandwidth required for fetching textures is reduced by 50%. Since textures represent 62% of memory bandwidth for our set of commercial Android games as illustrated in Figure 1, PFR can achieve up to 31% bandwidth savings.

Overall, we make the following contributions:

- We present PFR, a technique for improving texture locality on a mobile GPU that provides 28% memory bandwidth savings on average for a set of commercial Android games. This results in 19% energy savings and 10% speedup over a conventional mobile GPU.
- We show that for most applications PFR does not degrade the user experience, as the input lag is generally unaffected.
- For the few cases where the input lag is increased noticeably, we propose the use of a reactive approach which is able to adapt to the amount of input provided by the user.

The rest of the paper is organized as follows. The next section provides some background on mobile GPUs and describes the baseline GPU architecture assumed in this paper. Section III presents Parallel Frame Rendering. Section IV describes the evaluation methodology, whereas the experimental results are presented in Section V. Section VI contains a discussion of related work and, finally, we conclude in Section VII.

II. Background

This section presents the baseline GPU architecture assumed along this paper. As shown in Figure 4, it tries to track the microarchitecture of the ARM Mali 400-MP [13] mobile GPU. This is a processor with programmable vertex and fragment shaders, which uses Tile-Based Deferred Rendering (TBDR), so we first describe TBDR, and next we overview the main components of the GPU.

A. Tile-Based Deferred Rendering

In conventional rendering, a triangle is first transformed and then it is immediately sent down the graphics pipeline for further pixel processing. Most desktop GPUs, and also the mobile GPU inside the NVIDIA Tegra SoC [14], are based on conventional rendering. However, conventional rendering suffers from *overdraw* [15]: the colors of some pixels are written multiple times into memory because graphical objects that overlap are drawn over the top of one another. This problem can be completely avoided by using Tile-Based Deferred Rendering (TBDR) [16] instead of conventional rendering. Since *overdraw* results in a significant waste of memory bandwidth, hence in energy, it is not surprising that TBDR is becoming increasingly popular in the mobile GPU segment: the ARM Mali [13], the Imagination PowerVR [12] and the Qualcomm Adreno [17] are clear examples.

In a TBDR architecture the screen is divided in rectangular blocks of pixels or tiles —a typical tile size is 16 x 16 pixels— and transformed primitives (commonly triangles) are not immediately rasterized, but they are sorted into tiles and saved in memory. For each tile that a triangle overlaps, a pointer to that triangle is stored. After this has been done for all the geometry, each tile contains a list of triangles that overlap that particular tile, and the rendering starts tile by tile. The major advantage of TBDR is that tiles are small enough so all the pixels of a tile can be stored in local on-chip memory. When all the triangles for the tile have been rendered, the pixels are transferred just once to system memory, hence completely avoiding the *overdraw*. The downside is that it increases bandwidth requirements for the geometry because transformed triangles must be stored in memory and fetched back for rendering. Obviously, TBDR trades pixel for geometry memory traffic, but overall TBDR has proved to be effective in avoiding off-chip memory accesses on a mobile GPU [18].

B. Main Architectural Components

As shown in Figure 4, the baseline mobile GPU architecture assumed in this paper consists of 3 main components: the *Geometry Unit*, the *Tiling Engine* and the *Raster Unit*. The *Geometry Unit* converts the input 3D world-space triangles into a set of transformed and shaded 2D screen-space triangles. Next, the *Tiling Engine* sorts the transformed 2D triangles into screen tiles. Once all the geometry for the frame has been processed and sorted, the *Raster Unit* performs the rendering tile by tile. Let us explain them in more detail.

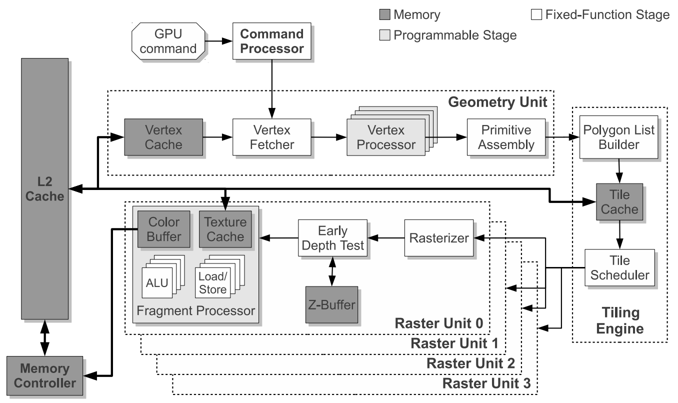


Fig. 4: Assumed baseline GPU architecture, similar to ARM Mali 400 [13].

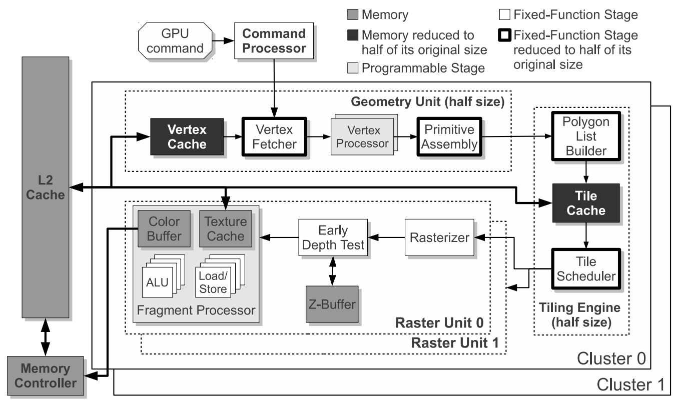


Fig. 5: GPU architecture employed for implementing Parallel Frame Rendering. The *Geometry Unit* and the *Tiling Engine* are reduced to half of their original size.

Regarding the *Geometry Unit*: in first place, the *Vertex Fetcher* reads the input vertices from memory, with the help of a *Vertex Cache*. Next, the vertices are transformed and shaded in the *Vertex Processors* based on the programmed by the user vertex program. Finally, the vertices are assembled into the corresponding triangles in the *Primitive Assembly* stage, where non-visible triangles are culled and partially visible triangles are clipped.

Regarding the *Tiling Engine*: in first place, the *Polygon List Builder* saves the 2D triangles to memory and sorts them into tiles. The region of main memory containing the transformed 2D triangles and the sorting of triangles into tiles is often referred in the literature as the Scene Buffer [19]. Once all the geometry has been sorted, the *Tile Scheduler* assigns tiles to *Raster Units*. All the triangles overlapping a tile are fetched from memory with the help of a *Tile Cache* and dispatched to the corresponding *Raster Unit* for rendering.

Regarding the *Raster Unit*: it computes the colors of the pixels within a tile. First, the *Rasterizer* converts the triangles into fragments, where a fragment is all the state that is necessary to compute the color of a pixel (screen coordinates, texture coordinates...). Next, occluded fragments are discarded in the *Early Depth Test* stage. Finally, the color of each visible fragment is computed in a *Fragment Processor*, based on the programmed by the user fragment program. The *Fragment Processors* include specialized texture sampling units for processing texture fetching instructions, which access a *Texture Cache*.

Regarding the memory hierarchy: several first level caches are employed for storing geometry (*Vertex* and *Tile Caches*) and textures (*Texture Cache*), which are connected through a shared bus to the L2 cache. The Color Buffer is the region of main memory that stores the colors of the screen pixels, whereas the Z-Buffer stores a depth value for each pixel, which is used for resolving visibility. In TBDR, the GPU employs local on-chip memories for storing the portion of the Color Buffer and the Z-Buffer corresponding to a tile. The local on-chip Z-Buffer does not need to be written to main memory, whereas the local on-chip Color Buffer does because the entire Color Buffer has to be generated and sent to the display. The local on-chip Color Buffer is directly transferred to system memory when all the triangles for the tile have been rendered, so each pixel is written just once in main memory, which completely avoids the *overdraw*.

Two of the GPU components, the *Vertex Processors* and the *Fragment Processors*, are fully-programmable simple in-order vectorial processors whereas the rest are fixed-function stages. A form of SMT is employed, where the processor interleaves the execution of multiple threads to avoid stalls due to long-latency memory operations.

III. TRADING RESPONSIVENESS FOR ENERGY

Traditionally GPUs have high memory bandwidth requirements as they need to fetch from memory a very large dataset (textures), which typically thrashes the cache. As shown earlier, texture data locality exists, however it can only be exploited across frames. In fact, we have observed that 96% of the texture data is shared between consecutive frames. Most of these accesses miss in cache due to the huge working set involved in rendering every single frame, i. e. they are capacity misses.

Fetching from memory is one of the main sources of energy inefficiency for mobile devices [9], [11], [12], and as such improving the inter-frame locality will likely lead to significant improvements in energy efficiency.

A. Parallel Frame Rendering

The idea of PFR consists on rendering two consecutive frames in parallel. If the two rendering tasks are well synchronized then they will hopefully perform similar operations in a similar order, so they will access the same textures within a small reuse distance. If that distance is small enough, the texture access of the first frame will perhaps miss in cache but will result in a cache hit when accessed by the second frame. Ideally, if both frames perform all the same accesses, and data is effectively kept in cache until reused by the second frame, we may expect a texture miss ratio reduction by one half that translates to a large reduction in memory traffic and energy waste.

To render two frames in parallel, we split the baseline GPU into two separate clusters, cluster 0 and cluster 1, each

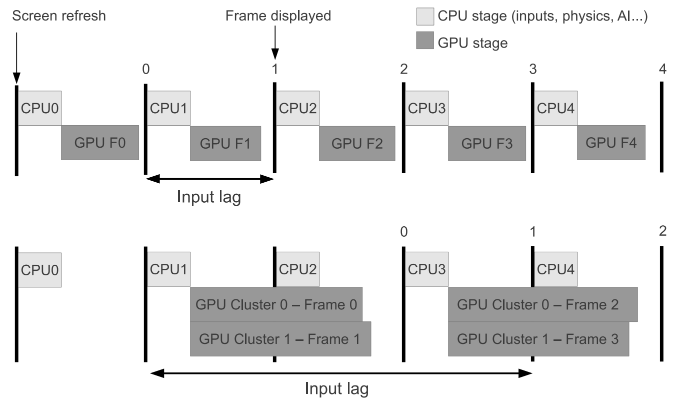


Fig. 6: Conventional rendering (top) vs Parallel Frame Rendering (bottom). Rendering a frame in a GPU cluster requires twice of the time since each cluster has half of the resources of the conventional GPU.

including one half of the resources, as illustrated in Figure 5. Even frames are then rendered by cluster 0, odd frames by cluster 1. Fragment Processors have private first level caches, but the second level cache is shared among all processors of both clusters, so we expect that one cluster prefetches data for the other cluster as discussed above. To further reduce the re-use distance between texture accesses of parallel rendered frames, the two clusters are synchronized by processing tiles in lockstep, since the same screen tile usually employs a very similar texture dataset in consecutive frames.

Figure 6 illustrates the difference between conventional rendering and PFR. To simplify the discussion, let us assume that conventional rendering fits in one screen refresh interval, then PFR spans two refresh intervals since each frame is rendered with half the number of processors. Notice that two frames are rendered in parallel, so the frame rate is still the same. With PFR, two consecutive frames are processed simultaneously, so two framebuffers are required instead of one. To avoid flicker or tearing, double buffering is usually employed in conventional rendering, which requires a separate frontbuffer for displaying the rendered image to the screen while the next frame is being drawn to the backbuffer. With PFR, double buffering requires two frontbuffers and two backbuffers. Hence, PFR increases the memory footprint. For a typical smartphone screen resolution of 800x480 (WVGA) and 32 bits per pixel (RGBA), conventional rendering requires 2.92 MBytes of main memory for storing the front and back Color Buffers, then PFR requires 5.84 MBytes. Since current smartphones usually feature 1GB of main memory, the memory footprint increment can be easily assumed. Notice that the memory bandwidth employed for transferring the Color Buffer is not increased despite two images are generated in parallel. In PFR two different Color Buffers are written into memory during two screen refreshes, whereas in conventional rendering the same Color Buffer is written two times during two refresh intervals. Therefore, the same amount of pixels are transferred in the same amount of time in both cases.

In conventional Tile-Based Deferred Rendering (TBDR), described in Section II-A, an entire frame has to be captured and the rendering is deferred until all the drawing commands for one frame have been issued by the application. In PFR the GPU driver has to be slightly modified, since two frames have to be buffered instead of one and the rendering is triggered when all the drawing commands for the second frame have been issued by the application and buffered by the GPU driver. Once the rendering of the two frames starts, the graphics hardware reads commands from both frames in parallel. The GPU Command Processor dispatches commands from the first frame to cluster 0 and commands from the second frame to cluster 1. Each GPU cluster behaves as an independent GPU, rendering the frame by using conventional TBDR as described in Section II.

Moreover, the GPU driver has to allocate two front buffers, two back buffers if double buffering is employed, and two scene buffers for sorting triangles into tiles (see Section II-B). As previously mentioned, the memory footprint is augmented, but the memory bandwidth requirements are not increased, because rendering each frame takes twice of the time.

B. *Reactive Parallel Frame Rendering*

A valid concern for the PFR approach is that since it devotes half the amount of resources to render each frame, the time required to render a frame is longer. This is an unfortunate side-effect as it ultimately leads to a less responsive system from the end-user's perspective. It is important to point out at this point that the frame rate is not reduced, since only the time between input and display increases (see Figure 6).

Assuming a typical frame rate of 60 FPS, conventional rendering exhibits an input lag of 16 ms (1 screen refresh) whereas in PFR it is increased to 48 ms (3 refresh intervals). Overall due to the nature of the user interface in mobile devices (touch screens are quite slow), we argue that this will mostly not be noticed by the end-user. In fact most applications tend to require very little input from the user for this exact reason (i.e., Angry Birds is a prime example of this behavior). In fact, this behavior is very common in mobile graphical applications, since they are designed in such a way that the user provides some inputs and then it just observes how the simulation evolves during the following frames in response to the user interaction (Figure 7). For these kind of applications PFR can be widely employed without hurting responsiveness during phases of the application where no user input is provided.

Nevertheless, there exist applications for which high responsiveness is required. As pointed out in [20], lags bigger than 15 ms can be noticeable and produce errors in interaction for applications demanding high responsiveness. In order to maintain the same levels of responsiveness with conventional rendering, we propose to build a system reactive to user inputs. Our basic premise is that for phases of the application in which the user provides inputs, we can disable PFR and the system reverts to conventional rendering, i. e. the two GPU clusters are employed to render just one frame. On the contrary, during phases in which no user input is provided PFR is enabled and two frames are rendered in parallel to save memory bandwidth. We refer to this system as Reactive-PFR (R-PFR). Figure 8 shows that R-PFR can be very effective since no input is provided most of the time for the majority of the mobile graphical applications we examined, so we can employ PFR extensively.

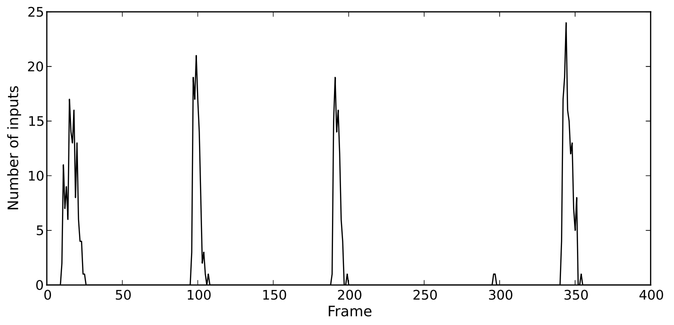


Fig. 7: Number of inputs provided by the user for 400 frames of Angry Birds. User inputs are heavily clustered, the game exhibits phases that require high responsiveness and phases where no user input is provided.

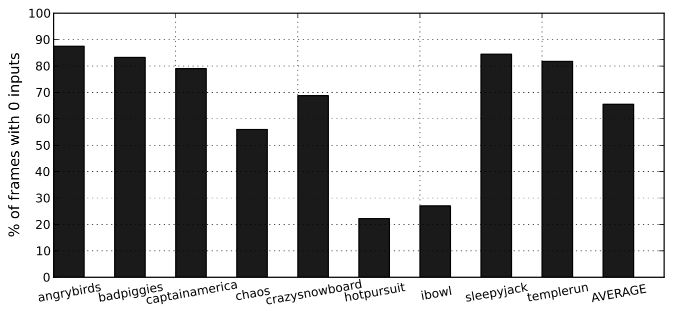


Fig. 8: Percent of frames where no user inputs are provided for several Android games. Two of the games, hotpursuit and ibowl, require high responsiveness whereas for the rest no user input is provided most of the time.

Switching from conventional rendering to PFR produces a very short screen stall. This is illustrated in Figure 9, where two frames have to be buffered and rendered in parallel before the screen is updated again. When switching from PFR to conventional rendering the two frames processed in parallel are discarded and a new frame is generated using both GPU clusters, so the same responsiveness than in conventional rendering is achieved. Hence, two frames are skipped when the system reverts to normal rendering, reducing the smoothness of the animation. Nevertheless, these perturbations happen in a very short time span of two refresh intervals, so an isolated switch is hardly noticeable by the user. Of course, excessive switching between rendering modes can still produce micro stuttering effects [21].

In R-PFR, the number of inputs received from the user are exposed to the GPU driver on a frame basis. The driver decides which rendering mode to employ every frame by monitoring user activity. The GPU maintains a counter with the number of frames without inputs since the last user interaction. Initially, conventional rendering is employed. When the counter of frames without inputs becomes bigger than a given threshold, the system switches to PFR. Finally, if the user provides some input the system immediately reverts to conventional rendering and resets the counter.

The threshold plays an important role in reducing the excessive number of switches between rendering modes and avoiding micro stuttering, as illustrated in Figure 10. If a big value for the threshold is employed, the number of switches is significantly reduced but the opportunities for applying PFR are severely constrained. A big threshold means that an important number of frames without user inputs have to be detected in order to switch to PFR, and all these frames are processed sequentially using conventional rendering. On the contrary, using a small value for the threshold increases the percentage of time PFR can be enabled, providing bigger energy savings, but it also increases the frequency of switches between rendering modes. We have selected 5 as the value for the threshold, since it provides a good trade-off for our set of commercial Android games (see Figure 10). By using a value of 5, the switches between rendering modes represent just 5% of the frames, so the small perturbations illustrated in Figure 9 are avoided 95% of the time. Even so, PFR can still be applied 42% of the time on average.

C. N-Frames Reactive Parallel Frame Rendering

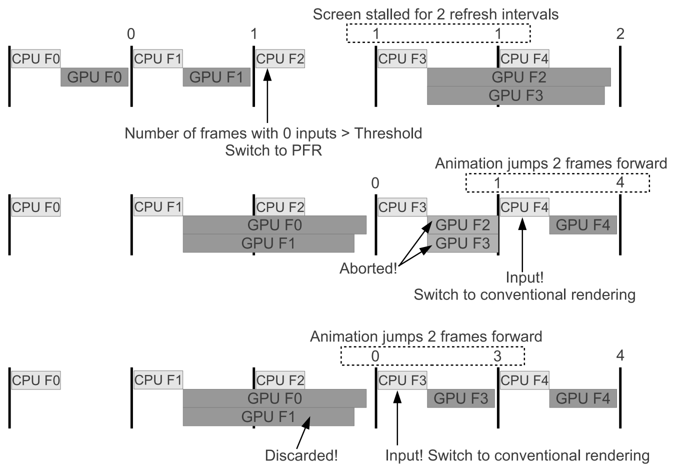
R-PFR achieves the same responsiveness as conventional rendering, but at the cost of smaller potential energy savings since PFR is applied just to a fraction of the total execution time. In an effort to recuperate the energy saving opportunities lost during phases of the application with user inputs, the system can apply parallel rendering more aggressively during phases without user inputs. Since input lag is not a problem during these phases the GPU can render 4 frames at a time instead of 2 to achieve bigger memory bandwidth savings. In that case, the GPU is split in 4 clusters that can render 1, 2 or 4 frames in parallel, each cluster having 25% of the resources of the baseline GPU. Initially, conventional rendering is employed so the 4 GPU clusters are used to render one frame. When the number of frames without inputs is bigger than a given threshold, T_1, the GPU driver switches to PFR so 2 GPU clusters are used to render odd frames and the other 2 clusters process even frames. If the number of frames without inputs becomes bigger than another threshold, T_2, then each GPU cluster renders a different frame so a total of 4 frames are processed in parallel. Finally, as in R-PFR, if the user provides some input the system immediately reverts to conventional rendering, so responsiveness is not reduced. We refer to this technique as N-Frames Reactive PFR (NR-PFR).

The thresholds T_1 and T_2 are set to 5 and 10 respectively, since we found these values provide a good trade-off between the number of switches and the percentage of time PFR is applied for our set of Android applications.

Note that 4 consecutive frames still exhibit a high degree of texture similarity, as shown in Figure 2. Ideally, if the 4 frames processed in parallel perform all the same texture accesses within a short time span, up to 75% memory bandwidth savings for texture fetching can be achieved.

D. Delay Randomly Parallel Frame Rendering

PFR delays user inputs in all the frames to achieve big memory bandwidth savings. R-PFR and NR-PFR do not delay any of the user inputs to achieve high responsiveness, at the cost of smaller bandwidth savings. Between these two extremes, a system can delay just a given percentage of the user inputs in order to trade responsiveness for memory bandwidth



Screen stalled for 2 refresh intervals
Number of frames with 0 inputs > Threshold
Switch to PFR
Animation jumps 2 frames forward
Aborted!
Input!
Switch to conventional rendering
Discarded!
Input! Switch to conventional rendering
CPU F0
GPU F0
CPU F1
GPU F1
CPU F2
GPU F2
CPU F3
GPU F3
CPU F4
GPU F4

Fig. 9: Switching from conventional rendering to PFR (top) and switching from PFR to conventional rendering for an even frame (middle) and for an odd frame (bottom).

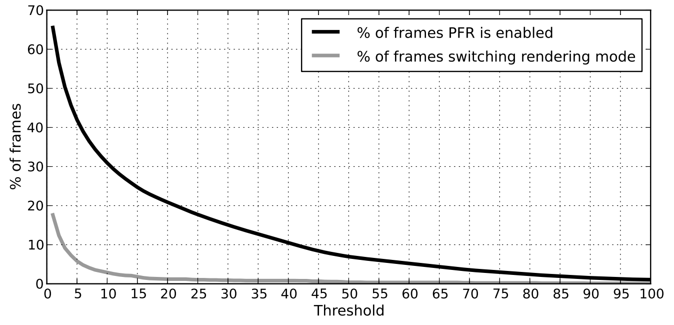


Fig. 10: Percentage of frames where PFR is applied and percentage of frames where a switch from PFR to normal rendering or vice versa takes place, vs the threshold for R-PFR. Increasing the threshold reduces the frequency of switches but it also reduces the percent of time for PFR.

savings. We refer to this system as Delay Randomly PFR (DRPFR).

The behavior of DR-PFR is very similar to R-PFR, but when the GPU driver detects user inputs the system does not always revert immediately to conventional rendering. Instead, a random number between 0 and 100 is generated and the system only switches to normal rendering if the random number is bigger than P, where P is the maximum percentage of frames where user inputs are allowed to be delayed. The bigger the value of P, the bigger the memory bandwidth savings but the smaller the responsiveness. The value of P can be set, for instance, depending on user preferences. It can also be set depending on the battery level, so it is dynamically increased as the battery level decreases in order to extend the use-time per battery charge. The frames where the inputs are delayed are randomly distributed to try to reduce the impact on user interaction, since random distortions are usually harder to perceive by the user than distortions that are very localized.

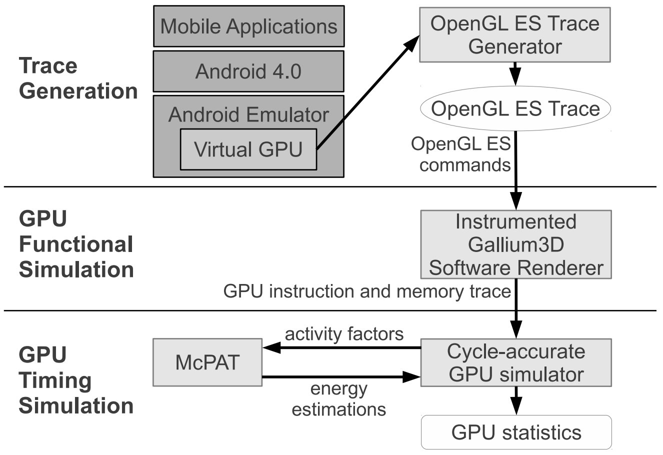


Fig. 11: Mobile GPU Simulation Infrastructure.

IV. EVALUATION METHODOLOGY

We have developed a mobile GPU simulation infrastructure [22], illustrated in Figure 11, which is able to run and profile Android applications. The infrastructure consists of three main components: the OpenGL ES trace generator, the GPU functional emulator and the cycle-accurate timing simulator. Regarding the trace generator, the Android Emulator available in the Android SDK is used to run the Android OS in a desktop machine. Furthermore the mobile software, i. e. the Android games, are executed on top. The GPU virtualization [23] feature of the Android emulator is employed, so the OpenGL ES commands issued by the mobile applications are redirected to the GPU driver of the desktop machine, providing hardware accelerated graphics for the Android OS running inside the emulator. We have developed an *OpenGL ES trace generator* that consists on a library interposed between the emulator and the desktop GPU driver. This library captures all the OpenGL ES commands issued by the Android applications, saves the per-command information in a trace file and redirects the commands to the appropriate GPU driver.

The OpenGL ES traces are fed to an instrumented version of Gallium3D [24]. Gallium3D is an infrastructure for building GPU drivers and it includes a complete software renderer, named *softpipe*. This software renderer executes the OpenGL ES commands on the CPU, providing GPU functional simulation and generating the GPU instruction and memory traces. Note that a software renderer is different from real hardware, so special care is taken to only trace the instructions that would be executed in the real GPU, i. e. the instructions in the vertex and fragment programs, and the memory requests that would be issued in the graphics hardware, i. e. memory accesses to read/write geometry, textures and the framebuffers.

Finally, the GPU instruction and memory trace is fed to our custom cycle-accurate GPU simulator, that models the mobile GPU architecture illustrated in Figure 4. Furthermore, it also models the clustered GPU shown in Figure 5, and we have implemented all the variations of PFR described in Section III. Regarding the power model, the McPAT [25] framework provides energy estimations. The parameters employed during the simulations are summarized in Table 1.

TABLE I: GPU simulator parameters. All the configurations include the same amount of resources: 1 big GPU cluster, 2 clusters with half the resources for each cluster or 4 clusters with 25% of the resources per-cluster.

Global Parameters		**Memory Hierarchy**	
Technology	32 nm	L2 Cache	128 KB, 8-way, 12 cycles
Frequency	300 MHz	Tile Cache	32 KB, 4-way, 4 cycles
Tile Size	16 x 16	Texture Caches	8 KB, 2-way, 1 cycle
Screen resolution	800 x 480 (WVGA)	Main memory	1 GB, 8 bytes/cycle (dual channel), 100 cycles

	Conventional rendering	**PFR, R-PFR, DR-PFR**	**NR-PFR**
Number of clusters	1	2	4
Raster units per cluster	4	2	1
Vertex Processors per cluster	4	2	1
Vertex Cache	8 KB, 2-way	4 KB, 2-way	2 KB, 2-way
Vertex Fetcher	16 in-flight vertices	8 in-flight vertices	4 in-flight vertices
Primitive Assembly	4 triangles/cycle	2 triangles/cycle	1 triangle/cycle
Polygon List Builder	4 in-flight triangles	2 in-flight triangles	1 in-flight triangle
Tile Fetcher	4 in-flight tiles	2 in-flight tiles	1 in-flight tile

Our set of workloads include 9 commercial Android games that are representative of the mobile graphical applications since they employ most of the features available in the OpenGL ES 1.1/2.0 API. We have included 2D games (`angrybirds` and `badpiggies`), since they are still quite common in the mobile segment. Furthermore, we have included simple 3D games (`crazysnowboard`, `ibowl` and `templerun`), with small fragment programs and simple 3D models. Finally, we have selected more complex 3D games (`captainamerica`, `chaos`, `hotpursuit` and `sleepyjack`) that exhibit a plethora of advanced 3D effects. Regarding the length of the simulations, we have generated traces of 400 frames for each game.

V. EXPERIMENTAL RESULTS

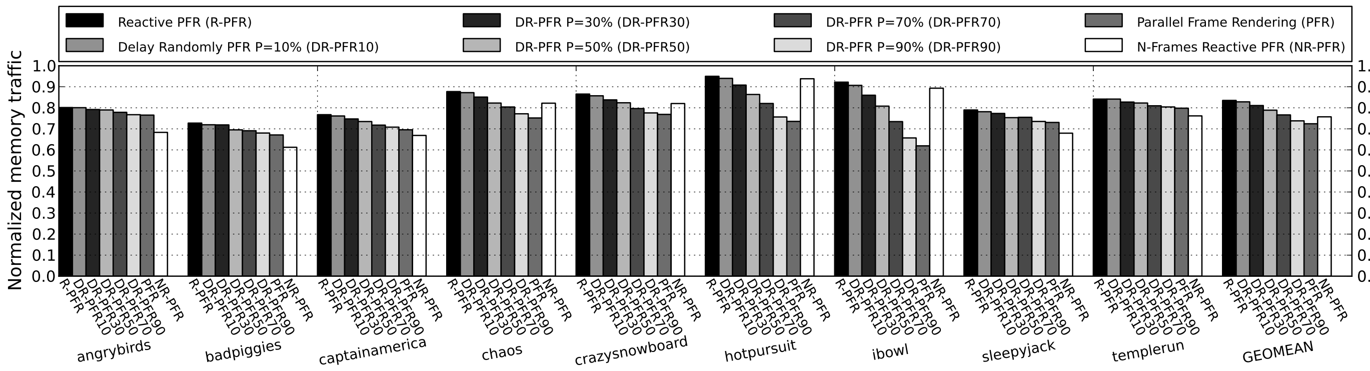
In this section we evaluate normalized off-chip memory traffic, speedups and normalized energy of the PFR variants described in Section III. The baseline does conventional rendering with a single big cluster. In first place, memory traffic is shown at the top of Figure 12. The basic version of PFR consistently provides important memory traffic reductions in all the applications, achieving 28% bandwidth savings on average. The reason is that PFR exploits the high degree of texture overlapping between two consecutive frames by processing them in parallel, so that most texture data fetched by one cluster is shortly reused by the other cluster before it is evicted from the shared cache, which converts almost half of the capacity misses produced in conventional rendering into hits.

Regarding R-PFR, we have set the threshold to 5, so the GPU driver switches from conventional rendering to PFR after 5 consecutive frames without user inputs. R-PFR achieves more modest memory bandwidth savings, 16% on average, because PFR is only enabled during phases of the application without user inputs, which limits the opportunities for processing frames in parallel but completely avoids any loss of responsiveness. R-PFR achieves its best results for games with a small number of user inputs, such as `angrybirds` or `badpiggies` (see Figure 8), where parallel processing is enabled most of the time. Conversely, it achieves its worst results for games with intensive user interaction, such as `hotpursuit` or `ibowl`, that force sequential frame processing most of the time.

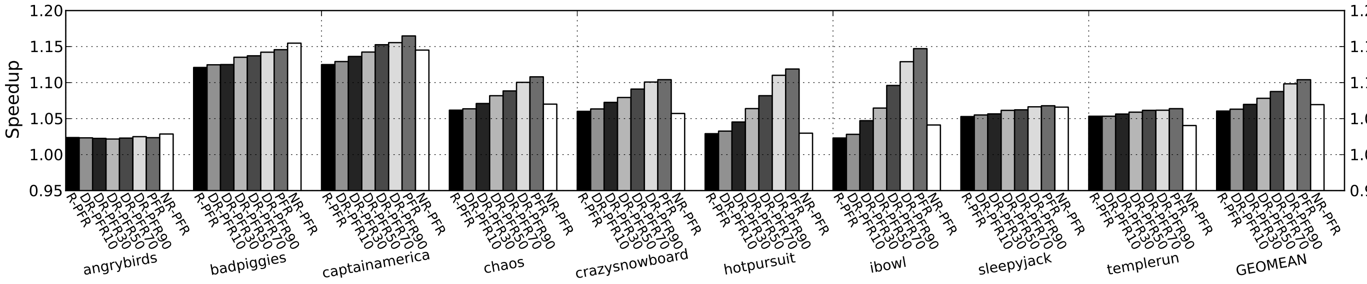
Regarding NR-PFR, we have set thresholds T_1 and T_2 to 5 and 10 respectively, so after 5 consecutive frames without user inputs the GPU driver starts processing 2 frames in parallel and after 10 it switches to 4 frames in parallel. For 5 games (`angrybirds`, `badpiggies`, `captainamerica`, `sleepyjack` and `templerun`), NR-PFR achieves even bigger memory bandwidth savings than PFR, while maintaining the same responsiveness than conventional rendering. Note that these games are the ones with more frames with no user inputs (see Figure 8), so the GPU driver can apply parallel rendering very aggressively most of the time. As in R-PFR, the savings are more modest for games with intensive user interaction (`hotpursuit` and `ibowl`). On average, NR-PFR achieves 24% off-chip memory traffic savings.

Regarding DR-PFR, we evaluated configurations with the parameter P set at 10, 30, 50, 70 and 90. P specifies the maximum percentage of frames where input lag is tolerated, providing fine-grained control to trade responsiveness for memory bandwidth savings. The results show that the biggest the value of P the biggest the bandwidth savings, but the smallest the responsiveness since more user inputs are delayed.

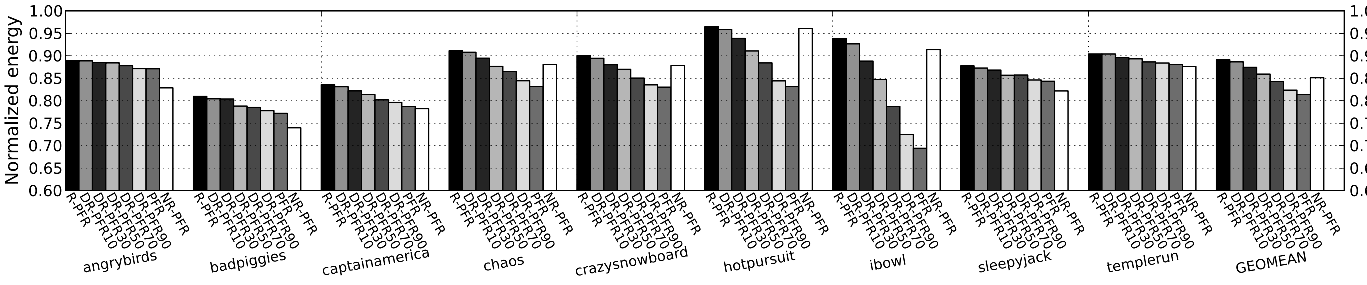
In second place, speedups are shown at the middle of Figure 12. It is worth noting that none of the configurations produce slowdowns in any of the games. On the contrary, PFR provides 10% speedup on average. Although the main objective of PFR is to save energy, performance is also



(a) Normalized memory traffic



(b) Speedups



(c) Normalized energy

Fig. 12: Normalized memory traffic (top), speedups (middle) and normalized energy (bottom). The baseline is a mobile GPU with one big cluster processing one frame at a time (configuration tagged as ”Conventional rendering” in Table 1).

increased as a side effect of reducing the memory traffic in a bandwidth bound system.

In third place, energy consumption is shown at the bottom of Figure 12. Energy results include the static and dynamic energy consumed by both the GPU and system memory. PFR achieves 19% energy savings on average that come from two sources. First, the dynamic energy is reduced because 28% of the off-chip memory accesses are avoided. Second, the 10% speedup achieved produces a reduction in static energy. Note that we do not assign any static energy consumption during long idle periods because we assume that the GPU could drastically reduce it by entering a deep low power state. Regarding R-PFR and NR-PFR, they achieve more modest energy savings since parallel rendering is enabled just a fraction of the total execution time in order to maintain responsiveness. Finally, DR-PFR provides bigger energy saving as the value of P is increased, at the cost of reducing responsiveness.

Figure 13 plots, for all configurations, the average normalized energy consumption versus loss of responsiveness, measured as a percentage of frames where user inputs are delayed. At one end, PFR provides the biggest energy reduction, 19% on average, but at the cost of delaying all the user inputs. At the other end, R-PFR does not lose responsiveness at all, since user inputs are never delayed, but achieves smaller energy savings, just 11% on average. NR-PFR also maintains full responsiveness, but achieves energy savings closer to PFR, 15% on average. Finally, DR-PFR allows fine-grained control over the energy savings and the responsiveness. The GPU driver can augment the energy savings by increasing the value of P, but then more user inputs are delayed. Conversely, the GPU driver can increase responsiveness by reducing the value of P, but losing part of the energy savings. By varying P, DR-PFR may achieve any intermediate energy versus responsiveness trade-off between PFR and R-PFR.

VI. RELATED WORK

Although it might appear that PFR is similar in concept to the NVIDIA AFR paradigm [26], our approach clearly

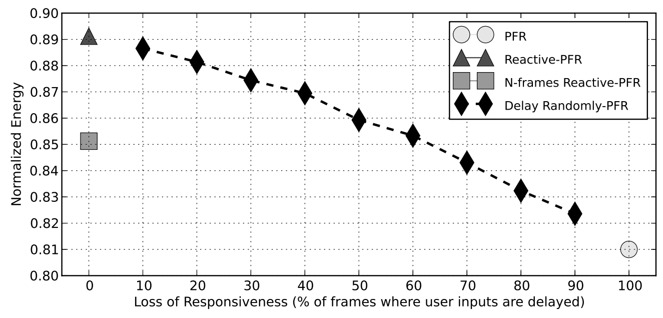


Fig. 13: Normalized energy vs. loss of responsiveness (percentage of frames where user inputs are delayed).

differs from the AFR, both in its goals and its methods: while AFR aims at increasing performance and frame rate by coordinating multiple independent GPUs with separate address spaces to maximize bandwidth, PFR pursues reducing energy by exploiting inter-frame temporal locality through splitting the computation resources of a single GPU with a shared last level cache and synchronizing memory accesses of consecutive frames.

On the other hand, Hasselgren et al. [11] propose a bandwidth saving technique for systems with stereoscopic displays. Since multiple views of the same frame have to be generated for a 3D display, the authors propose to compute the different views concurrently to improve texture cache hit ratio by using a novel rasterization architecture. Although the objective is the same than in our technique, i. e. saving memory bandwidth, the implementation is significantly different since PFR splits the graphics hardware in multiple clusters, but each cluster is still a conventional mobile GPU based on TBDR [16]. Furthermore, PFR is not limited to 3D displays and it achieves bandwidth savings on smartphones with 2D screens. Note that both techniques can be combined in stereoscopic systems: PFR can exploit inter-frame texture similarity by processing consecutive frames in parallel, whereas each GPU cluster can employ the technique described in [11] to maximize intra-frame texture locality.

Our approach is orthogonal to a wide range of memory bandwidth saving techniques for mobile GPUs, such as texture compression [8], texture caching [27], color buffer compression [7], tile-based deferred rendering [19] or depth buffer compression [4].

VII. Conclusions

In this paper we argue that the memory bandwidth required for mobile GPUs can be significantly reduced by processing multiple frames in parallel. By exploiting the similarity of the textures across frames, we can overlap the execution of consecutive frames in an effort to reduce the number of times we bring the same texture data from memory. We term this technique Parallel Frame Rendering (PFR).

This, however, comes at a cost in the responsiveness of the system, as the input lag is increased. We argue that in practice this is not really important for most of the applications, as for mobile systems touch screens tend to be slow and thus applications tend to require small interaction with the user. Nevertheless, we show that adaptive forms of PFR can be employed, that trade responsiveness for energy efficiency. We present three variants of these adaptive schemes and show that we can achieve 24% reduction in memory bandwidth without any noticeable responsiveness hit.

Acknowledgments

This work has been supported by the Generalitat de Catalunya under grant 2009SGR-1250, the Spanish Ministry of Economy and Competitiveness under grant TIN 2010-18368, and Intel Corporation. Jose-Maria Arnau is supported by an FI-Research grant.

References

[1] T. Akenine-Moller and J. Strom, "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones," *ACM Transactions on Graphics*, vol. 22, pp. 801–808, 2003.

[2] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Boosting Mobile GPU Performance with a Decoupled Access/Execute Fragment Processor," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. IEEE Computer Society, 2012, pp. 84–93.

[3] S.-L. Chu, C.-C. Hsiao, and C.-C. Hsieh, "An Energy-Efficient Unified Register File for Mobile GPUs," in *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, 2011, pp. 166–173.

[4] J. Hasselgren and T. Akenine-Möller, "Efficient Depth Buffer Compression," in *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ser. GH '06. New York, NY, USA: ACM, 2006, pp. 103–110.

[5] B. Mochocki, K. Lahiri, and S. Cadambi, "Power Analysis of Mobile 3D Graphics," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, ser. DATE '06. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 502–507.

[6] V. Moya, C. González, J. Roca, A. Fernández, and R. Espasa, "A Single (Unified) Shader GPU Microarchitecture for Embedded Systems," in *Proceedings of the First international conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 286–301.

[7] J. Rasmusson, J. Hasselgren, and T. Akenine-Möller, "Exact and Error-Bounded Approximate Color Buffer Compression and Decompression," in *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ser. GH '07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 41–48.

[8] J. Ström and T. Akenine-Möller, "iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '05. New York, NY, USA: ACM, 2005, pp. 63–70.

[9] T. Akenine-Moller and J. Strom, "Graphics Processing Units for Handhelds," *Proc. of the IEEE*, vol. 96, no. 5, pp. 779–789, 2008.

[10] A. Carroll and G. Heiser, "An Analysis of Power Consumption in a Smartphone," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 21–21.

[11] J. Hasselgren and T. Akenine-Möller, "An Efficient Multi-View Rasterization Architecture," in *Proceedings of the 17th Eurographics conference on Rendering Techniques*, ser. EGSR'06. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 61–72.

[12] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson, "Adaptive Scalable Texture Compression," in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, ser. EGGH-HPG'12. Eurographics Association, 2012, pp. 105–114.

[13] "Mali-400 MP: A Scalable GPU for Mobile Devices," http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_ARM.pdf.

[14] "Bringing High-End Graphics to Handheld Devices," http://www.nvidia.com/content/PDF/tegra_white_papers/Bringing_High-End_Graphics_to_Handheld_Devices.pdf.

[15] "Mali GPU Application Optimization Guide," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0555a/CHDIAHCC.html.

[16] "Tiled Rendering," http://en.wikipedia.org/wiki/Tiled_rendering.

[17] "How to Engineer Fast into Smartphone Apps," https://www.uplinq.com/2011/sites/default/files/slides/How-to-Engineer-Fast-into-Smartphone-Apps.pdf.

[18] I. Antochi, "Suitability of Tile-Based Rendering for Low-Power 3D Graphics Accelerators," Ph.D. dissertation, 2007.

[19] I. Antochi, B. H. H. Juurlink, S. Vassiliadis, and P. Liuha, "Memory Bandwidth Requirements of Tile-Based Rendering," in *SAMOS*, 2004, pp. 323–332.

[20] M. J. P. Regan, G. S. P. Miller, S. M. Rubin, and C. Kogelnik, "A Real-Time Low-Latency Hardware Light-Field Renderer," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 287–290.

[21] "Micro Stuttering," http://en.wikipedia.org/wiki/Micro_stuttering.

[22] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 37–46.

[23] "GPU Virtualization in the Android Emulator," http://developer.android.com/tools/devices/emulator.html#acceleration.

[24] "Gallium3D," http://en.wikipedia.org/wiki/Gallium3D/.

[25] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480.

[26] "SLI Best Practices," http://developer.download.nvidia.com/whitepapers/2011/SLI_Best_Practices_2011_Feb.pdf.

[27] Z. S. Hakura and A. Gupta, "The Design and Analysis of a Cache Architecture for Texture Mapping," in *Proceedings of the 24th annual international symposium on Computer architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 108–120.