Toward High Mobile GPU Performance Through Collaborative Workload Offloading

Chao Wu, *Member, IEEE*, Bowen Yang, Wenwu Zhu, *Fellow, IEEE*, and Yaoxue Zhang, *Member, IEEE*

Abstract—The ever increasing of display resolution on mobile devices raises high demand for GPU rendering details. However, the challenge of poor hardware support but fine-grained rendering details often makes user unsatisfied especially in calling for high frame rate scenarios, e.g., game. To resolve such issue, we propose ButterFly, a novel system which collaboratively utilizes mobile GPUs to process high-quality rendering details for on-the-go mobile users. In particular, ButterFly achieves three technical contributions for the collaborative design: (1) a mobile device can migrate GPU workloads in buffer queue to peers, (2) the collaborative rendering mechanism benefits user high quality details while significant power saving performance, and (3) unnecessary 3D texture rendering can be clipped for further optimization. All the techniques are compatible with the OpenGL ES standards. Furthermore, a 40-person survey perceives that ButterFly can provide excellent user experience of both rendering details and frame rate over Wi-Fi network. In addition, our comprehensive trace-driven experiments on Android prototype reveal the benefits of Butterfly have more superior performance over state-of-the-art systems, which achieves more than 28.3 percent power saving.

Index Terms—Mobile applications, distributed system, code offload, performance optimization

1 Introduction

Since the conception of "Retina Display" proposed by Apple Inc. in 2010 [2], techniques of display resolution on mobile devices have continued to rise, from 960*640 pixels of the iPhone 4 to 1920*1080 pixels (FHD) of Google Nexus 6, and recently to 2560*1440 pixels (2K) of Samsung Galaxy S5 LTE-A. Motivated by such widespread promotions from industry, today's on-the-go mobile user demands higher visual quality experience. As a result, devices, e.g., smartphones, are often suffering from battery running out and overheating caused by enthusiastic GPU power, which, however, cannot yet afford user either expected frame rate, e.g., at 60 FPS on 2K pixels, or high quality rendering experience in daily usage.

Meanwhile, the idea of crowdsourcing- or collaboration-based mobile code offloading/migration has been around for over a decade [3]. There have been renewed interests to make mobile devices last longer, e.g., [4], [5], [6], since year 2010, inspired in large part by the MAUI work [7]. Thus, there comes a succeeded proposal that paved a novel avenue for addressing the aforementioned problems by leveraging the server-side collaborative rendering mechanism [8].

However, today's single-deployed sever-side infrastructures, e.g., implemented by [8], cannot yet be afforded economically. In particular, by means of large-scale data-driven measurements in our early stage work, we observe that 80 percent users suffer network access latencies of more than 817 ms (Fig. 2) if they want to obtain collaborative rendering services from a test sever we deployed in California (Section 2.1). Such latencies limit users to low visual experience, i.e., rough rendering details and frame-drops. Moreover, wholesale infrastructure deployments also undoubtedly result in non-trivial economic cost [9].

In this paper, to provide high quality experience while aiming at an economical and power saving effect, we present *ButterFly*, a mobile rendering system based on collaborative GPU process, which overcomes the drawbacks of poor hardware constraint and potential high RTT latency caused by aforementioned factors. Thus, the main idea of ButterFly is to collaboratively render with peer devices. In particular, it relies on mobile GPUs to generate input-to-display workloads, which are processed by peers and then return fine-grained rendering details at expected frame rate.

The key insight behind ButterFly solution is that, with the poor hardware support on single mobile device, fine-grained details are prohibitively expensive for a mobile GPU to render at a high frame rate. Thus, it is highly non-trivial if we are able to provide high quality details in an energy-efficient style. To achieve such goal, we develop GPU workload migration technique for seamless and real-time detail rendering. Indeed, we intercept the physical graphics buffer to support an OS-level design, without requiring any changes or recompiling from any mobile apps or OSes. In particular, we treat the challenges of Android implementation as device sensing and GPU rendering, by which mobile devices with poor hardware might also migrate tasks or be assigned through their data traffic to peers, e.g., users' OpenGL ES compatible pads, APs and smartphones, with fine-grained detail feedback. Moreover, to avoid unnecessary texture render and reduce their energy overhead, we develop the visibility-driven texture simplification (VTS) technique to

• *The authors are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {chaowu, wwzhu, zyx}@mail.tsinghua.edu.cn, yangbw16@mails.tsinghua.edu.cn.*

Manuscript received 23 May 2017; revised 15 Sept. 2017; accepted 16 Sept. 2017. Date of publication 20 Sept. 2017; date of current version 12 Jan. 2018. (Corresponding author: Chao Wu.)
Recommended for acceptance by X. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2754482

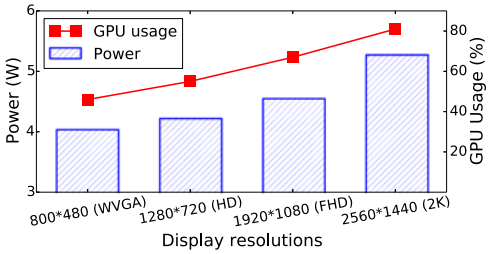


Fig. 1. System power and GPU utilization of demo rendering on Galaxy S5 LTE-A with different visual qualities.

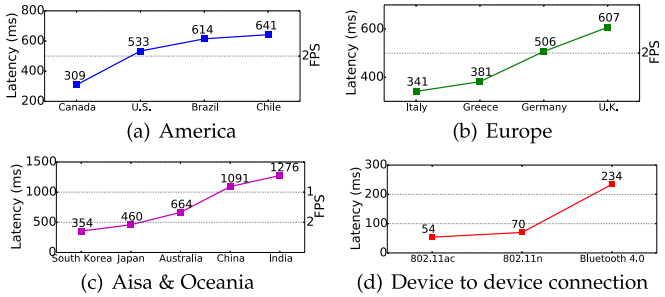


Fig. 2. 12,197 volunteers' average RTT latency distribution with accessing to server in San Mateo, CA, U.S. against connection under 802.11 network (via Cisco AIR-SAP2602I-C-K9 AP) and bluetooth.

leverage internal knowledge of the graphics display to make efficient energy optimization. Fundamentally, we employ ButterFly energy saving guided by user-specified policies in a way that mobile graphics rendering are collaboratively processed to save energy, as long as user-related viewing experience are also being maximized.

We have implemented a ButterFly prototype that works on today's commercial Android smartphones such as Galaxy S5 LTE-A and Google Nexus 6. We also conduct a 40-person study and comprehensive trace-driven experiments to evaluate our Android prototype. Results of the user study indicate that our system perceives good user experience with qualitative details (see Table 3) when compared with original approach. In addition, experimental evaluations have quantified that the energy per frame rendering can be reduced by 28.3 percent for 2K pixels and up to 45.1 percent when the display resolution is demisemi, i.e., WVGA. Specifically, in our *Final Fantasy* game (Fig. 3) process, ButterFly achieves a superior 8.5 percent power saving than state-of-the-art work (Section 5.3) while only has a goodput of 341 KB/s in the most consuming case, i.e., rendering details for 2K pixels at 60 FPS (Section 5.4.3). We also have applied the VTE technique to 10 most representative 3D graphics models on commercial smartphone for trials. It results in a significant energy (27.2 percent) and peer device saving benefits on average and has little compromised viewing experience, compared to the default applications.

To the best of our knowledge, we are the first to design and implement peer collaborative rendering system for mobile devices. We identify the main contributions of this paper as:

- We design a system-level collaborative rendering mechanism and propose technique within it for ButterFly's design and implementation, which is compatible with the OpenGL ES/EGL standards.
- We joint both energy and RTT delay considerations for a global optimization in ButterFly.
- We have implemented ButterFly prototype on nowaday Android smartphone which has few hardware constraints.
- We conduct comprehensive experiments and a user study to confirm the effectiveness of our system for optimizing user experience and power overheads.

We introduce ButterFly as: Section 2 presents the motivation and goals of our work. Then, Section 3 introduces the logic workflow and optimal considerations. Section 5 next evaluates our work with both qualitative and quantitive experiments. Section 7 reveals the related work, and Section 8 closes this paper with conclusion as well as future work.

2 Motivation, Goal and Primitives

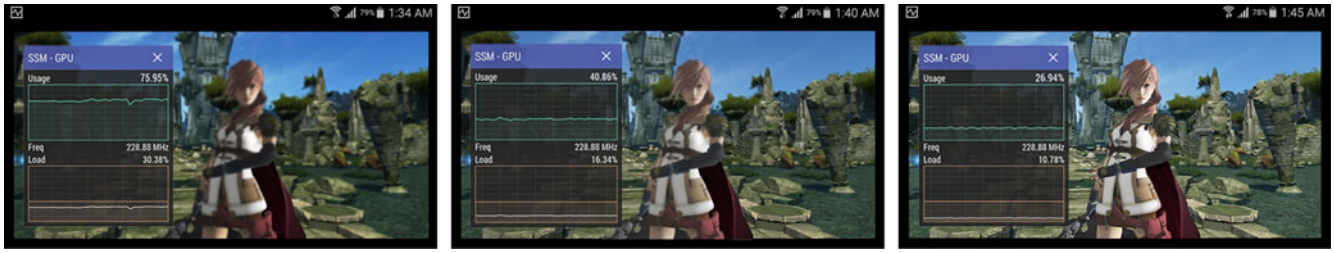
This section analyzes fundamental problems in high quality rendering process and existing proposals for mobile devices (Section 2.1), to further motivate our work and set our goals of this paper (Section 2.2). To better present the difficulties in collaborative GPU rendering, we close this section by exampling primitives for Android graphics (Section 2.3).

2.1 Motivation

High-Quality Details but "Wasted" Power Cost. To provide mobile users high quality rendering details in fine-grained applications such as gamings, VR and AR, we have seen rapid improvements in desktop/mobile GPU processing capabilities. Unfortunately, today's mobile devices cannot yet duplicate the sophisticated graphical detail provided by multimedia/game consoles and high-end desktop GPUs. The primary reason of such performance gap is that high quality detail presentation often comes with exhausted power cost, which reaches a scale of 150 W (WVGA) to 300 W (FHD) to 600 W (2K) on desktop side [10]. To further explore the power costs to mobile device, in Fig. 1, we apply *Final Fantasy* game rendering (see Fig. 3a) through Rajawali engine [11] on Samsung Galaxy S5 LTE-A smartphone. By tuning different display resolutions,[1] we observe that more fine-grained (2K) rendering details consume 131 percent power (5.4W [2]) than rough experience (e.g., WVGA). Moreover, the battery capacity of mobile devices is limited and growing slowly, and high power consumption requires sophisticated and bulky thermal dissipation systems that are incompatible with mobile form factors. In addition, today's smartphone often affords a power upper limit of 8.4 W and a rated power of 3.7 W, which demands an emerge and challenging technique to support mobile user a power-saving but high quality rendering details experience.

RTT Latency and Non-Trivial Economic Cost. The demand for high quality rendering technique is also constrained with poor frames per second (FPS) performance in the presence of mobile hardware. To relief the hardware resource limitation, Kahawai [8] exploits a system that provides high-quality experience on mobile devices, e.g., tablets and smartphones, by offloading a portion of the GPU computation to server-side infrastructure. However, this benefit comes at a non-trivial cost since it demands

1. This is executed under the same environment as Fig. 11.
2. Note that the battery capacity of Samsung S5 LTE-A is 10.78Wh.



(a) ButterFly with no worker (20 FPS on 2K) (b) ButterFly with 1 worker (25 FPS on 2K) (c) ButterFly with 3 workers (60 FPS on 2K)

Fig. 3. Screenshots of using ButterFly to render Final Fantasy game with Rajawali engine on Samsung S5 LTE-A.

abundant servers to support worldwide on-the-go mobile users if they want to have a good RTT latency, which includes both network access and GPU computation. More specifically, by measuring a demographical composition of 12,197 volunteers from 76 countries,[3] we explore their HTTP response latencies from accessing our test server in San Mateo, CA, U.S., and Fig. 2 reveals the latency distributions (in continent locations). We see that single severside design cannot afford user with rated 60 FPS experience on full display resolution [12]. On contrast, device to device connection brings a factor of 95.6 percent RTT reduction over 802.11 network. Given that server-side proposal mandates wholesale changes to the network infrastructure, it is natural to ask if this complexity is worthwhile. Specifically, we ask:

- Does migration provide significant benefits?
- If so, can we achieve the benefits in economical fashion within the scope of today's available techniques?

2.2 Goal

The aforementioned questions motivate us to design a mobile collaborative GPU rendering mechanism among devices. Because, in real life, users' devices are inactive in most cases [13]. This inspires us that the inactive hardware resources are available for our system design. Nevertheless, as resource limited in real life, we try to use a few peers for global power and latency optimization (Section 3.4). For instance, let us consider the *Final Fantasy* game rendering in Fig. 3 again. In the test program, each frame i's power cost P_i for rendering details has correlation as $P_i \propto resolution$, which, as Fig. 3 shows, only affords mobile user rough rendering experience but occupies over 76 percent GPU usage.[4] In contrast, by using ButterFly, the user has high quality detail experience while lower GPU usage, which decreases to 27 percent. Thus, in this paper, *our goal* is to migrate frames' rendering workload on mobile GPU to other devices for a collaborative process. As the difference of processing time plays a critical role on people's visual experience [8]. Indeed, it is of great problem of workload migrating decision to guarantee ButterFly's quality of experience. Note that, the RTT of whole process, which includes the network delay and input-to-display latency, should be no more than $\frac{1}{FPS}$ second for avoiding framedrops, as Section 5.4.3 elaborates.

3. We convene the volunteers from our Twitter followers.
4. Note that we measure the rough GPU performance by using Simple System Monitor tool [14]

2.3 Primitives for Android Graphics Rendering

Modern Android graphics rendering consists of a sequence of steps called graphics pipeline which has multiple steps via using OpenGL ES/EGL library. For a concise illustration, Fig. 4 summarizes the main steps in top-down style. Specifically, in OpenGL ES, complex geometry scenes are represented by basic vertexes and the relationships among them. For instance, four vertexes form a square face. The *vertex-processing* stage handles vertices, typically performing operations such as transformations and skinning [15]. In this case, the vertex-processing is done in a square, uniform coordinate system called normalized device coordinate space. After the vertex-processing stage, each vertex has its own normalized device coordinates. Then, the *rasterization* stage solves the relationships (i.e., pointsprite, lines and triangles) among these vertexes and maps these lines and triangles from the continuous device-coordinate space to a discrete window-pixel space. Note that, in OpenGL ES, this coordinate mapping is configured by calling the *glViewport ()* API which takes four parameters to determine the affine transformation. Last, the *pixel-processing* stage generates meta data for each pixel, e.g., colors and depths.

For a given frame, once the display resolution increased, there generates more pixels to process. As a result, the mobile GPU workload of both the two pixel-based stages of rasterization and pixel-processing increases accordingly. Consequently, more battery are consumed for suffering the additional workload in rendering high quality frames, or provide users low quality frames without charging any more GPU resources. Intuitively, it is very difficult to customize any sub-functions in OpenGL ES library. However, it gives us an insight to migrate frame rendering workload to peer devices for achieving power saving but high quality

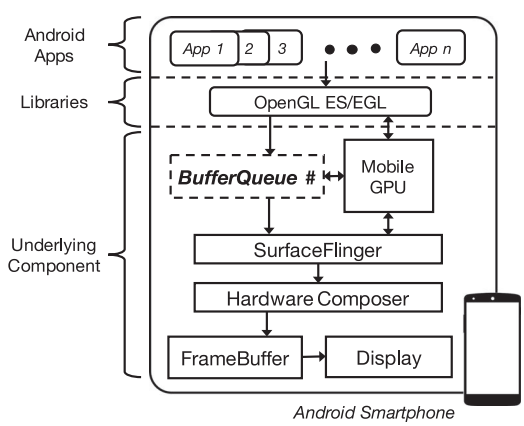


Fig. 4. The process of modern Andriod graphics rendering.

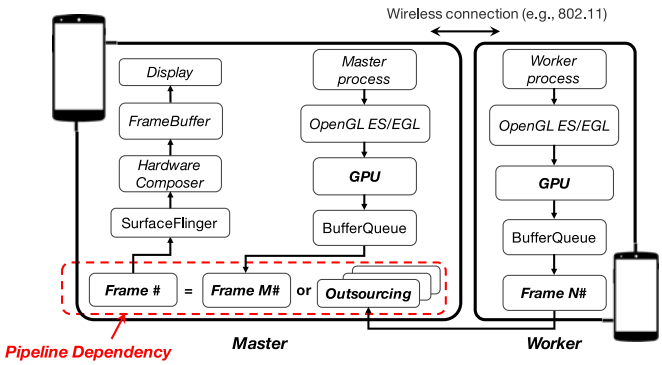


Fig. 5. Workflow of collaborative rendering in ButterFly.

experience goal. To provide such collaborative rendering, a simple but efficient approach is to separately migrate GPU workloads in the BufferQueue (as highlighted in Fig. 4). We leave details of designing this mechanism until Section 3.

3 ButterFly Design

ButterFly provides user collaborative GPU rendering supports. In this section, we propose design details of ButterFly within the scope of today's available techniques.

3.1 Overview

To better present the logic, Fig. 5 introduces the conceptual workflow of ButterFly from a high level. It depicts that ButterFly employs a *Master-Worker* pattern to achieve the mobile collaborative rendering. Specifically, it consists of three associated stages, i.e., master processing, worker processing, and pipeline dependency.

Preliminaries. As studied in Section 5.2, real-life user often owns less than 3 available devices to play as the worker, especially for devices that are in charging. Thus, we design ButterFly from a joint energy and delay optimization. To achieve this, ButterFly employs the aforementioned two considering models and a principle that using less than 3 workers to schedule tasks. Note that, in ButterFly, as the workers are in charged and Wi-Fi connected, the experimental metrics in Section 5 are conducted for the master.

Master Processing. A master in ButterFly is the rendering workload (task) requester who consumes the input-to-display rendering frames, makes control decision on whom to assign, and receives the results from requestee, i.e., worker. As illustrated in Fig. 5, different from traditional graphics rendering presented by Fig. 4, the workflow of master processing injects a pipeline dependency link before calling SurfaceFlinger interface for Android graphics rendering. In particular, given each frame rendering task T in master's BufferQueue, the master (M) intelligently chooses the matched workers (N) for collaborative process through wireless network or handle it locally. Once the result (frame $M\#$ or $N\#$) returned, the master passes it to the tail of buffer queue for further displaying, which can benefit user more fine-grained details since display details are well rendered by multiple GPUs.

Worker Processing. A worker is the requestee who supports master a collaborative frame rendering. Different from master, the worker does not display the rendering result on screen. Thus, as shown in Fig. 5, the workflow of worker

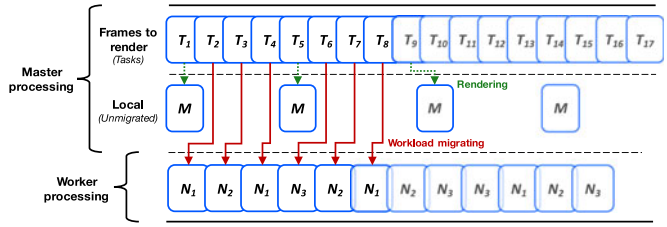


Fig. 6. Task assignment. The subscripts indicate the frame number. Green dashed arrows indicate local pipeline dependencies, and red arrows show examples of task migration.

processing finishes when result has been returned to the master. Thus, given a specific rendering task requested from master, the worker processes it, and then passes the result from its BufferQueue directly. Note that, in ButterFly design, the worker processes works in background service.

Pipeline Dependency. ButterFly schedules task assignment in parallel whenever possible to reduce framedrops and input-to-display delay (i.e., RTT latency of the whole ButterFly process). For example, Fig. 6 illustrates the pipeline dependency by assigning tasks (j) to three workers (N_i) or directly rendering on the master (M). In particular, it is executed through considering if any GPU resource are now available. Thus, each stage executes in a different thread and depends on the output of the previous stage. Specifically, in ButterFly we have the following asynchronous stages:

- Render (M): In this brief stage, ButterFly queries the whole rendering frames and periodically assign frames to render by itself, i.e., rendered by master's GPU.
- Render (N_i): In this stage, the consecutive frames, e.g., among T_2 and T_4, are assigned and rendered by workers' GPU.
- Deliver (N_i): In this stage, workers return the rendered results (frames) to the master via network.
- Display ($M||N_i$): After results received, this stage displays user well rendered frames (T_i) smoothly (at expected FPS).

Note that the overheads of each individual stage are highly variable, depending on the logical (e.g., physics) and graphical (e.g., polygon count) complexity of the frame being rendered.

3.2 Energy Consideration

In ButterFly, each master maintains its own worker list $\mathcal{N}$ in the *worker sensing* process. Specifically, when a collaborative rendering task arrives, as Fig. 7 illustrates, the *Scheduler* determines whether to migrate the task or render locally. If migrated, worker $n \in \mathcal{N}$ is assigned by considering both energy and RTT factors. Moreover, the network congestion avoidance should also be considered for improving the RTT responsiveness of our system (Section 5.4.3).

Indeed, the power consumption of the ButterFly collaborative rendering can be generalized into three categories: *picking workers*, *data transmission* among devices and *rendering* on mobile GPU. For modeling, we denote the power cost of these states as P_{pic}, P_{dat} and P_{gpu} respectively. The power cost of rendering migration can be modeled as next.

Given master's rendering task T_i arrives at time moment t_i with frame data size D_i, and the most prior task in the

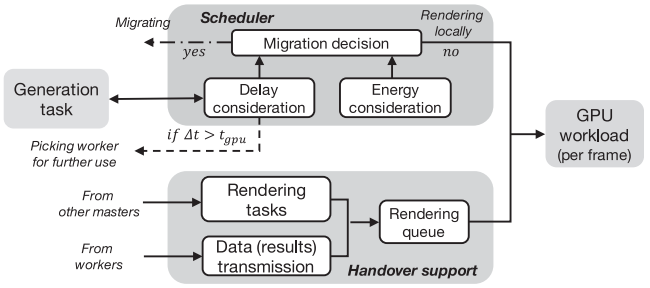


Fig. 7. Migration methodology of ButterFly.

mobile device's process is T_j. The data goodput at t_i is r_{dat}. The time interval between task T_i and T_j is $\Delta t = t_i - t_j - D_j/r_{dat}$. There are three cases to formulate T_i's power cost on the master side which is related to Δt. 1) If Δt is larger than the rendering timer t_{gpu}, i.e., the GPU resource is in *avaliable* state before T_i arrives, then T_i will consume extra picking energy for future assignment, and GPU process energy, besides the data transmission energy. 2) If Δt is smaller than t_{gpu} but bigger than 0, there is rendering energy but no more picking energy. 3) If Δt is smaller than 0, T_i will be overlapped with T_j for some time, and no more extra rendering energy. Thus, we have

$$E^i(T_j) = \begin{cases} P_{pic} \times t_{pic} + P_{dat} \times \frac{D_i}{r_{dat}} + P_{gpu} \times t_{gpu}, \\ \qquad\qquad \text{if } \Delta t > t_{gpu} \\ P_{dat} \times \frac{D_i}{r_{dat}} + P_{gpu} \times \Delta t, \\ \qquad\qquad \text{if } 0 < \Delta t \leq t_{gpu} \\ P_{dat} \times max\left\{\Delta t + \frac{D_i}{r_{dat}}, 0\right\}, \text{Otherwise} \end{cases} \tag{1}$$

If this task is rendered locally, the energy consumption E^i_{local} can be calculated using Equation (1). Otherwise, if this task is migrated to worker n, the data transmission energy should be the data delivery energy on n, and plus the additional energy of the system's network interface. Since it is hard to predict when the future tasks will be scheduled, thus we compute the total energy based on the most consuming case (i.e., has additional picking energy, as Equation (2) illustrates

$$\begin{aligned} E^i_{remote} = &\frac{D_i}{r_{sys}} \times P_{sys} + \frac{D_i}{r^n_{dat}} \times P^n_{dat} \\ &+ \frac{D_i}{\Gamma} \times \left(P^n_{gpu} \times t^n_{gpu} + P^n_{pic} \times t^n_{pic}\right), \end{aligned} \tag{2}$$

where Γ is data size threshold for scheduling remote tasks.

3.3 RTT Delay Consideration

The RTT delay to complete a task is also related to Δt. If Δt is smaller than t_{gpu}, the delay is only the data transmission and GPU processing time. Otherwise, the delay will include additional worker picking time, as Equation (3) formulates

$$d^i_{dat}(T_j) = \begin{cases} D_i/r_{dat} + t_{gpu} + t_{pic}, & \text{if } \Delta t > t_{gpu} \\ D_i/r_{dat} + t_{gpu}, & \text{Otherwise} \end{cases}. \tag{3}$$

If task T_i is rendered locally, the delay d^i_{local} can be computed using Equation (3). When migrated to worker n, the delay contains four parts: the time to render the first task in the remote queue, queue delay, the time to get T_i through wireless network and the time to deliver it back through the network interface. Similarly, there are also two cases to compute the delay of the first remote task. In case 1, the delay of the first task in the remote queue can be estimated by $1/\lambda^n_l$. In case 2, the primary task of the remote queue is to wait till the total data size of the remote queue is larger than $\frac{\Gamma - S^n_r}{\lambda^n_r \times D^n_r}$. The queue delay depends on the total remote task size S^n_r. The wireless network delay and the network access delay depend on D_i. Putting them together, the delay to render T_i at the worker n is

$$d^i_R = \frac{D_i + S^n_r}{r^n_{dat}} + \frac{D_i}{r_{sys}} + \max\left\{\frac{1}{\lambda^n_l}, \frac{\Gamma - S^n_r}{\lambda^n_l \times D^n_r}\right\}. \tag{4}$$

Where $\lambda_{l(r)}$ is data arriving rate of local (remote) task queue.

3.4 Joint Energy and Delay for Optimization

We now propose the mathematical model in our system by jointing energy and RTT delay for ButterFly system optimization. Similar to above derivation, we note that a graphics rendering pipeline is chopped into N frames, each of which is indexed by i (i.e., $i = 1, 2, \ldots$). Each frame lasts T seconds and has been pre-encoded into multiple versions using a set of rendering decision $\mathcal{R}$ and resolution pixels $\mathcal{B}$. To offload each frame i, ButterFly selects the master itself or some worker, thus we have $(r(n), b(n)) \in \mathcal{R} \times \mathcal{B}$.

Constraints. Given the quality of ith frame can be estimated by a Quality Controller (QC) function $Q(r(i), b(i))$. To provide high quality and energy saving rendering results, the first constraint of QC is to ensure the $Q(r(i), b(i))$ for each frame to be reasonably high. Indeed, as frames are then fetched into the BufferQueue (as we introduce in Fig. 4) of the master device before playback. The buffer size, i.e., the duration of the *downloaded-yet-unplayed* frame, evolves dynamically as frames being downloaded and played. At the ith step, the master downloads the whole frame i and the downloading time depends on the frame size and network throughput. Thus, the frame is being played and the buffer size is being reduced. After downloading frame i, the buffer size $B(i)$ becomes

$$B(i) = B(i-1) + T - \frac{T \cdot r(i)}{W(i)}, \tag{5}$$

where $T \cdot r(i)$ is the size of frame i and $W(i)$ is the average network throughput when downloading frame i.

To derive $W(i)$, we note ButterFly finishes the downloading of frame $i-1$ at time $t(i-1)$. When frame $i-1$ is completely fetched, the SurfaceFlinger immediately loads frame i at time $t(i)$. Thus, the average throughput $W(i)$ can be derived as

$$W(i) = \frac{\int_{t(i-1)}^{t(i)} W_t \, dt}{t(i) - t(i-1)}, \tag{6}$$

where W_t is the downlink throughput at time moment t.

To save energy, another important goal is to avoid graphics in excessive display density, where the GPU processing can drain the battery. To prevent this annoying effect,

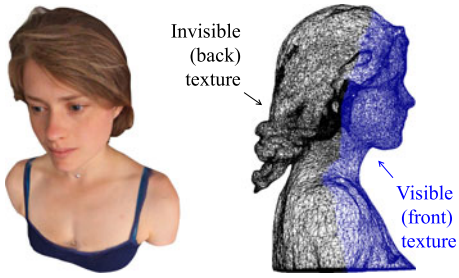


Fig. 8. A 3D model provided by Sketchfab [16]: We color visible face in blue and the invisible in black.

the second constraint of QC is to bound the buffer size at every step as next,

$$0 < B(i) \leq B_{max}(r(i)), \tag{7}$$

where $B_{max}(r(i))$ is the maximum buffer size determined by the storage capacity of resolution pixels $r(i)$. In this paper, we note $r(i) = 2560 * 1440$. In a rare case when buffer is full, ButterFly will sleep for a while before new frames arrive.

Objective. As we aim to minimize the sum of energy overhead of mobile GPU while improving viewing quality. To this end, we note the energy overhead of master device for rendering frames $1, 2, \ldots, i$ to provide basic viewing as

$$\mathcal{O}_{mas}(i) = E_{mas} \frac{T \cdot b(i)}{W(i)}, \tag{8}$$

where E_{mas} is the energy requirement for each frame. It is a constant in according to the display density, and can be manually tuned. In this paper, we adopt the 2k as baseline. In addition to achieve high quality viewing with worker devices, we employ extra frames rendering on workers as

$$\mathcal{O}_{extra}(i) = E_{extra}(r(i)) \cdot T, \tag{9}$$

where $E_{extra}(r(i))$ is the supplemental energy consumption in rendering frame i.

We can now summarize the objective of our optimization as minimizing the total energy overhead within the entire tasks (totally $\mathcal{I}$ frames), i.e.,

$$min \sum_{i=1}^{\mathcal{I}} \mathcal{O}_{mas}(i) + \mathcal{O}_{extra}(i). \tag{10}$$

Optimization. Therefore, we can formally define the ButterFly optimization problem as next,

$$\begin{aligned} & min_{\overrightarrow{r}, \overrightarrow{b}} \sum_{i=1}^{\mathcal{I}} \mathcal{O}_{mas}(i) + \mathcal{O}_{extra}(i) \\ & s.t. \quad Q(r(i), b(i)) \geq \delta \\ & \quad\; Q(r(i), b(i)) \geq \delta \\ & \quad\; 0 < B(i) \leq B_{max}(r(i)) \\ & \quad\; B(0) = B_{start} \\ & \quad\; r(i) \in \mathcal{R}, b(i) \in \mathcal{B} \\ & \quad\; Equation\ (5), (6), (8), (9), \forall i = 1, \ldots \mathcal{I}, \end{aligned} \tag{11}$$

where δ is the threshold of resolution pixels and B_{start} is the startup buffer size after which the playback starts.

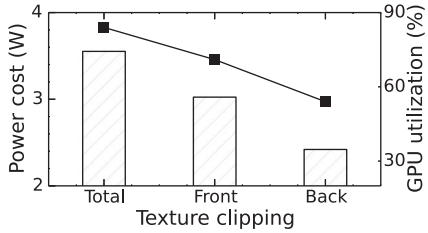


Fig. 9. System power overhead to different framerate rendering on S5 by using VTS technique.

The intuition of Equation (11) is to optimize the rendering quality across all $\mathcal{I}$ frames such that minimal energy overhead can be achieved. Thus, ButterFly is generically designed and it is also straightforward to formulate other types of high-quality GPU workloads.

4 Further Optimization for 3D Graphics

In this section, we further explore the key factor in 3D mobile graphics rendering. To this end, we study Rajawali, an open source graphics engine for Android, and identify the issue in existing computation-intensive mobile graphics, i.e., unnecessary texture rendering.

Unless otherwise specified, all the numbers presented in this section are average experiment results obtained on a Samsung S5 LTE-A (SM-G906K) smartphone with 7.7 W rated power. All the graphics data are loaded from inner flash (writes 2.2 mW, reads 23.1 mW) for repeatable experiments. The total system energy cost to the device are measured using a Monsoon power meter [17].

4.1 Unnecessary Texture Rendering

When rendering input-to-display 3D texture, OpenGL ES pipeline will first entry the vertex processing stage where typically performs operations such as transformations and skinning [18]. Next, the rasterization stage solves the relationships (i.e., the pointsprites, lines and triangles) among the vertexes and maps lines and triangles from continuous device-coordinate space to discrete window-pixel space. Last, the pixel processing stage generates meta data for each pixel, e.g., colors and depths. Afterward, OpenGL ES shader program reads the texture from each application's BufferQueue, renders it into an image and finally shows the image to user. This is the most basic but crucial mobile graphics processing because the OpenGL ES renderer will consume mobile GPU resource [15] and further make power consumption.

However, we have observed that plenty of texture rendering results are actually not displayed as they are out of user's field of view. To demonstrate this observation, we apply a female's 3D gaming model as the study case. Fig. 8 shows the 3D rendering and texture wireframe (consist of vertex and edge) of the model in the direction perpendicular (90 degree) to user's line of sight. Specifically, we color the user-visible (in the field of view) wireframe in blue, and invisible wireframe in black. Fig. 9 shows the system power consumed to render the 3D model on S5 LTE-A, at 2560x1440 (2K) resolution pixels and 60 FPS framerate. In contrast, by the texture elision technique we develop in Section 4.2, we separately render the front (visible) texture and back (invisible) texture. Indeed, it uses 84 percent of GPU

usage and 3,550 mW system power to render the whole 3D texture. However, rendering the front texture only costs 71 percent of GPU usage and 3,020 mW system power, thus leads to unnecessary energy consumption. Besides, in rendering the back texture, both GPU usage (54 percent) and system power (2,420 mW) are lower than before which means more energy now can be wasted by default renderer. What's worse, the renderer persistently keeps on loading and processing the unclipped graphics at every moment, whether ever the field of view has changed or not. The results can vary by graphics contexts, fields of view and directions to line of sight. However, such thoroughly rendering is not energy friendly, overly unnecessary for viewing, and should be re-considered to improve the energy efficiency of mobile 3D graphics processing.

4.2 Visibility-Driven Texture Simplification

As analyzed in Section 4.1, we find many graphics texture render (particularly for invisible or low visibility concavity regions) are not actually required. To this goal, we design and implement a visibility-driven texture simplification technique based on a classic graphics simplification [19] to clip the unnecessary energy. VTS first employs the built-in gyroscope sensor, which might use 30 ± 1.3 mW power [20], to detect user's current field of view by using the `SensorManager` class. Within such a field, VTS then combines the surface visibility measure with a well-known geometric error measure called the quadric measure [19], [21], which is defined for each edge (mesh object) in the OpenGL ES vertex processing stage. To better illustrate the principle, we note that **e** be the next edge to collapse into a point **v**, represented in homogeneous coordinates as $(x_0, y_0, z_0, 1)^T$, where T is all the triangles in M that are adjacent to at least one vertex of **e**, i.e., T is the union of the 1-ring neighborhoods of both vertices of edge **e**, allowing the triangles in both neighborhoods to be counted twice. Then, each triangle **t** has a plane equation

$$A_t x + B_t y + C_t z + D_t = 0. \tag{12}$$

As the quadric measure is then defined as

$$E_q(\mathbf{e}) = \sum_{\mathbf{t} \in T} (distance(\mathbf{v}, \mathbf{t}))^2. \tag{13}$$

Thus we have

$$E_q(\mathbf{e}) = \sum_{\mathbf{t} \in T} (A_t x_0 + B_t y_0 + C_t z_0 + D_t)^2 = \mathbf{v}^T \mathbf{Q} \mathbf{v}, \tag{14}$$

where

$$\mathbf{Q} = \sum_{\mathbf{t} \in T} \begin{pmatrix} A_t^2 & A_t B_t & A_t C_t & A_t D_t \\ A_t B_t & B_t^2 & B_t C_t & B_t D_t \\ A_t C_t & B_t C_t & C_t^2 & C_t D_t \\ A_t D_t & B_t D_t & C_t D_t & D_t^2 \end{pmatrix}. \tag{15}$$

To combine the visibility measure with the quadric measure, we note the quadric measure is the sum of the squared distance from a point to many planes. We construct a tessellation of user eyes (i.e., a "camera") space S by subdividing the faces of an octahedron three times and placing sample cameras at each vertex of the resulting mesh. We note a camera pose **v** sees a triangle **t** if and only if at least part of t is visible from **v**; $F(\mathbf{t}, \mathbf{v}) = 0$ if **t** is entirely invisible from **v**, and 1 otherwise; $N(\mathbf{t})$ is the normal of **t**, and $R(\mathbf{v})$ is the viewing direction of **v**. We note the tessellation 0.1 of the camera space is even. Thus, $area(\mathbf{v})$ is the same for all **v**

$$V(\mathbf{t}) = \frac{\sum_{\mathbf{v} \in S} F(\mathbf{t}, \mathbf{v}) * (R(\mathbf{v}) \cdot N(\mathbf{t})) * area(\mathbf{v})}{\sum_{\mathbf{v} \in S} (R(\mathbf{v}) N(\mathbf{t})) * area(\mathbf{v})}. \tag{16}$$

Further, if edge **e** is adjacent to some triangles with low visibility, the distance from **v** to this plane makes less visual impact than the distances from **v** to high visibility triangles if the geometric errors are the same. Thus, our visibility-driven error measure can be defined as

$$E_v(\mathbf{e}) = \sum_{\mathbf{t} \in T} (distance(\mathbf{v}, \mathbf{t}) V(\mathbf{t}))^2, \tag{17}$$

where $E_v(\mathbf{e})$ directs which edges are collapsed, i.e., this measure is used to order the priority queue.

By recalling the meaning of $V(\mathbf{t})$ as the weighted sum of dot products between a triangle's normal with incoming ray directions, our visibility-directed error measure for one triangle is the weighted average projected distance from all viewing directions. This means edges with higher geometric errors can be chosen for removal if they situate in extremely low visibility regions, such as interiors and creases. In our current implementation, we use the default quadric matrix to select the best new vertex location for the collapsed edge as demonstrated in [21]. Based on this, our measure can be derived as

$$E_v(\mathbf{e}) = \mathbf{v}^T \mathbf{Q}_\mathbf{v} \mathbf{v}, \tag{18}$$

where

$$\mathbf{Q}_\mathbf{v} = \sum_{\mathbf{t} \in T} \left[\begin{pmatrix} A_t^2 & A_t B_t & A_t C_t & A_t D_t \\ A_t B_t & B_t^2 & B_t C_t & B_t D_t \\ A_t C_t & B_t C_t & C_t^2 & C_t D_t \\ A_t D_t & B_t D_t & C_t D_t & D_t^2 \end{pmatrix} V^2(\mathbf{t}) \right]. \tag{19}$$

Afterward, unnecessary texture rendering are clipped out of OpenGL ES pipeline. In practice, by the current implementation of VTS on Android, we find it can cost up to 27.6 s to process a complex texture elision. To avoid passive effects, we now pre-execute VTS for several potential fields of view during graphics data loading thus can dynamically switch to and support on-demand 3D rendering. In the future, we will seek to improve this processing through code offloading to the cloud.

Comparison to State-of-the-Art Techniques. Although state-of-the-art GPU APIs already provides workload reduction by using classic simplification algorithms, e.g., back-face culling [22], they can usually bring unsatisfied viewing experience. To explore it, Fig. 10 shows an example of texture simplification with a 3D cup model by classic back-face culling and our proposed VTS approaches. We observe the back-face culling algorithm results in image loss within the cup (which we annotate with dotted line), while VTS has few viewing experience downgrade.

Taken together, VTS can provides both energy saving and high quality viewing experience in mobile rendering.

(a) Back-face culling (b) VTS clipping

Fig. 10. Viewing quality of rendering 3D cup model by using back-face culling algorithm and VTS.

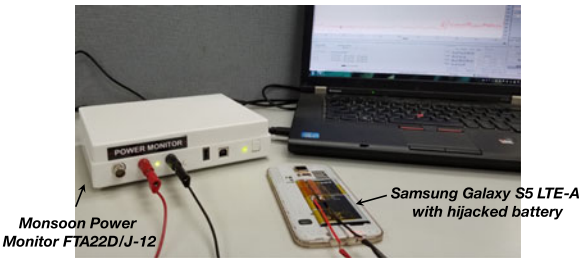


Fig. 11. The hardware power meter used for energy measurements with Android smartphone.

Thus, in this paper, we employ such technique for 3D graphics optimization in using ButterFly.

5 Evaluation

In this section, we conduct user study and trace-driven evaluation over our Android implementation to investigate the performance of ButterFly.

5.1 Implementation and Experiment Setup

Android Implementation. We have implemented ButterFly prototype system on the basis of Android 4.2+ with OpenGL ES 3.0 by revising an open source graphics engine called Rajawali version 1.0.325 forked from Github project [23], the latest stable version of Rajawali when we conducted the work of this paper. We evaluate ButterFly by running it on SamsungGalaxy S5 LTE-A, S4, Google Nexus 5 and Nexus 6 smartphones. Similar to homogeneous smartphones, such OSes and OpenGL ES libraries are not fully open source. Thus, we cannot modify the source code and re-build the original ROM. Instead, we utilize binary-rewriting techniques to intercept the system API calls, e.g., *glViewport()*. We also implemented the ButterFly prototype on Android.

Power Meter Setup. As our goal is to provide users with high quality rendering experience but charge them with little consumptions, the metric of ButterFly energy is extremely significant. However, today's smartphones do not offer a way to obtain fine-grained energy measurements of an application. Instead, they simply show users with a toast that "how much battery (in %) is left", which is very coarse-grained and often unreliable [7], [24]. Therefore, to better quantify a device's energy consumption, we attach a hardware power meter [17] to the smartphone's hijacked battery (as shown in Fig. 11). Note that this power meter provides fine-grained energy measurements: it samples the current drawn from the battery with a frequency of 5 KHz and a mean error which is less than 3 percent [7], [17].

TABLE 1
Information of 40 Participants in our User Study

Enrollment	Knowledge	Enrollment	Knowledge
10	IxD	8	CS
6	EE	5	finance
5	machinery	6	others

TABLE 2
Participants' Hardware Support

802.11 pattern	Uplink/downlink speed	Average GPU driver overhead	Average available workers
ac	7.12/17.14 Mbps	26.7 FPS	$\lceil 2.4 \rceil = 3$
bgn mixed	7.73/15.47 Mbps		

Qualitive and Quantitive Evaluations. To comprehensively explore the system performance, we begin by breaking down the potential benefits of ButterFly into two categories. The first class of qualitative benefits with *users studies*, i.e., visual quality and framedrop rating, whose results are need to be coupled with on-the-go mobile users. The second class of quantitative and objective benefits stem from the ability to reduce smartphone's battery overheads by *trace-driven experiments* against with state-of-the-art work, e.g., Kahawai-like server-side approach.

5.2 User Study on ButterFly

Study Design. We first evaluate the qualitative benefits of ButterFly. To better understand how collaborative rendering affects users' daily experience, we conducted a user study in which 40 participants (15 females and 25 males) were asked to run several game rendering: *Final Fantasy*, *Alien Shooter* and *Call of Duty* on the top of Ranawali rendering engine. Each participant is asked to use both ButterFly and original approach at least 20 minutes. As Table 1 depicts, the 40 participants are equipped with various knowledge to ensure the pertinent ranking. All devices are connected to an 802.11ac/bgn mixed campus Wi-Fi. In addition, we refer to the configuration where volunteers experienced unmodified rendering details as well as ButterFly's on the same devices. The study was performed between May 10th and 20th, 2016 within the range of Tsinghua campus in Beijing, China.

We asked all participants to complete a survey (for both original and ButterFly experience). The survey contained a consent form, asked for basic demographic information, and assessed their concerns in and use of mobile rendering. The survey also requires participants to provide their hardware resource support information as summarized in Table 2. Specifically, participants were asked to rate their experience of ButterFly against the original approach on a scale of one ("Poor") to two ("Fair") to three ("Good") to four ("Excellent") to five ("Identical") in according to questions as: *"I consider it has a high rendering quality (looked good) experience."*, *"I consider it has few framedrops (ran smoothly) during rendering process."* and *"How many devices around me would be the worker?"* etc. In particular, we used these ratings to evaluate how ButterFly benefits users with qualitative visual improvement.

Result and Analysis. Of the 40 participants, 20 are with 7.12/17.14 Mbps (802.11ac) uplink/downlink speed, while

TABLE 3
Mean Opinion Scores and Values of ButterFly Performance Against with Original Experience

Mean Opinion Score	Quality	Framedrop
1 (Poor)	-	2%
2 (Fair)	-	5%
3 (Good)	15%	40%
4 (Excellent)	75%	50%
5 (Identical)	10%	3%

the rest 20 are with 7.73/15.47 Mbps (802.11bgn mixed). In our survey, the average GPU driver overhead performance is 26.7 FPS. Note that, in normal use, the best rendering performance of a certain app is usually less than the ability of GPU driver overhead (often around 20 to 22 FPS) at full resolution which comforts to our test in Fig. 3a. In according to our statistics, more than 65 percent participants think they would simultaneously have at most *3 workers* to use ButterFly, which inspires us that our system implementation should have superior performance with no more than three workers, as we elaborated on later. We also summarize user ratings of ButterFly in Table 3, we see that, as for the visual quality and framedrop experience, 85 percent participants rate ButterFly as *"good, I like the quality of rendering!"*, and 93 percent participants think *"there are no framedrops when I use it."*

In summary, our user study indicates that ButterFly can benefit users with high quality rendering experience, which again demonstrates the vantage points of our system.

5.3 Comparison to Other Systems

Our user study indicates that ButterFly's collaborative rendering provides good visual experiences. In this section, we then use trace-driven inputs to our Android prototype to characterize the visual quality of the collaborative rendering (FPS) and power consumption. Note that we use those same metrics to compare directly with three benchmarks as next.

1) Original approach, i.e., rendering on smartphone locally.
2) Server-side approach, i.e., using server-side rendering without considering the latency issue (we implement a server in LAN).
3) Server-side in practice, i.e., using server-side rendering and take latency in consideration (we use the server on San Mateo, CA, U.S.).
4) ButterFly approach, i.e., using ButterFly with at most three workers (determined by the aforementioned study).

FPS Performance. We comprehensively compare the FPS performance under 4 display resolutions[5] on Samsung S5 LTE-A phone, respectively. The results are shown in Fig. 12. We see that the increase of display resolution results in weaker FPS performance for all the four metrics. In addition, for a specific resolution, e.g., full 2560*1440 (2K) pixels on S5 LTE-A smartphone, there appears a significant performance gap among all the metrics. In particular, ButterFly benefits users a balanced experience of 57 FPS with only 3

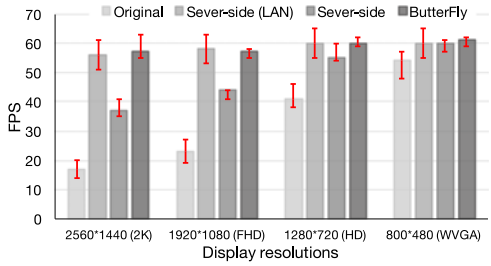


Fig. 12. Quality experience achievement of original, server-side approach, server-side implementation with latency consideration and ButterFly to different scenarios.

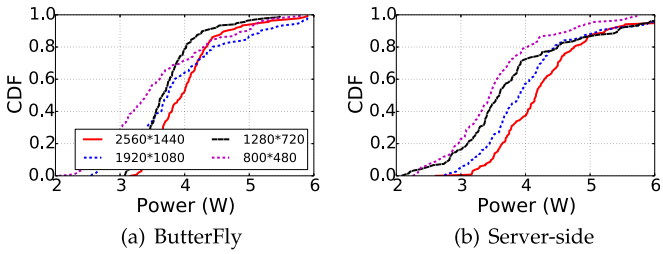


Fig. 13. Power comparison of ButterFly and server-side.

workers communicated with 802.11 Wi-Fi, on average. Note that, by migrating GPU workload to the three workers, the performance reaches a peak of 60 FPS while a valley of 51 FPS. It also comforts to Samsung S5 LTE-A's GPU driver overhead in 2K pixels scenario [12]. As benchmarks, the original approach only provides users with 17 FPS on average, server-side approach benefit users with a high FPS quality of 37 FPS (55 FPS in LAN). It demonstrates the efficiency of ButterFly solution. We shall emphasis that, by adding more workers, ButterFly can stabilize at 60 FPS experience as the rest bars (e.g., HD pixels case) in Fig. 12.[6]

Power Overhead. We then compare the power consumption caused by using both ButterFly (Fig. 13a) and server-side (Fig. 13b) approaches. Fig. 13 depicts the cumulative probability distribution (CDF) of the power consumption for all the collaborative rendering tests. We observe that more fine-grained quality rendering would be significantly power exhausted on smartphone's battery. For instance, in 800*480 (WVGA) pixels case, ButterFly costs user an average power of 3.5 W. Nevertheless, it climbs to 4.0 W (3.7 W, 3.9 W) for collaborative rendering in 2K (HD, FHD) pixels scenario. Upon comparison, the server-side approach consumes user powers of 3.5 and 4.3 W for collaborative rendering in WVGA and 2K pixels, respectively. On the other hand, we find that ButterFly has a better power-saving performance of 8.5 percent (0.3 W, which is 6.8 percent of the rated power) when compared with server-side approach on S5 LTE-A smartphone. It again demonstrates the significance of our work. Moreover, with more workers, ButterFly can further reduce the power (energy) consumption but improve the rendering quality as comparing Figs. 3b with 3c. In addition, in according to our user study results in Section 5.2, ButterFly can benefit users with smooth experience, i.e., user can perceive fewer or even none framedrops.

5. Note that the resolution parameters are set in according to the quality ranking reported by Wikipedia [25].

6. Note that low display resolution requires few GPU workload, which can be totally processed by the 3 workers.

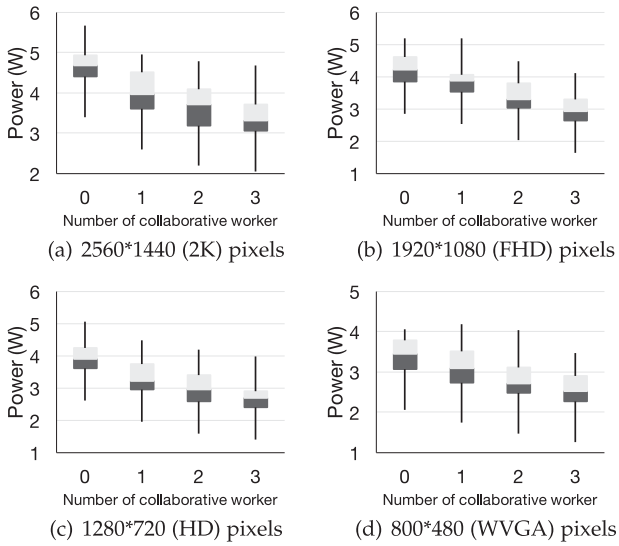


Fig. 14. Power consumption variation to different collaborative workloads and rendering qualities.

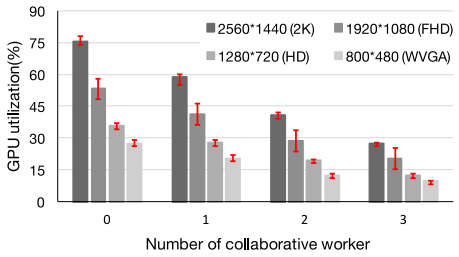


Fig. 15. GPU utilization variation of ButterFly.

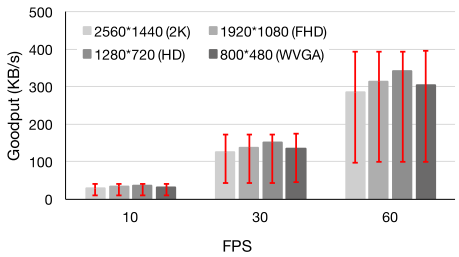


Fig. 16. Goodput requirements with different scenarios.

5.4 Microbenchmarks

5.4.1 Processing Efficiency

As we aim to achieve high framerate during video playing, it is essential to evaluate system processing efficiency within ButterFly system. To this end, Fig. 18 measures the average per frame rendering (PFR) time consumption of GPU rendering and total (1,000 frames) system wall-clock time in rendering the Final Fantasy gaming (as Fig. 3 shows) with ButterFly. We see that by adding more worker devices, ButterFly can provide a significant wall-clock time saving benefits with 73 percent reduction (from 3.7 to 0.9 s). However, the PFR time are all around 57 ms. This is because ButterFly is designed to offload amount of GPU rendering workloads to potential worker devices while a single task is fine-grained rendered on a master's or worker's GPU. Thus, ButterFly can provide high framerate and excellent viewing.

5.4.2 Overheads of ButterFly to Different Worker Amount

The benchmark evaluation illustrates the superior performance of ButterFly. Moreover, it also is important to explore the power consumption under different amounts of workers and GPU utilization. In this section, we next use trace-driven inputs to evaluate the mentioned baselines.

Power Cost to Different Amounts of Worker. To investigate the power consumption variation with different amount of workers, we measure the 0 to 3 workers' power performance under 4 resolutions respectively. As Fig. 14 summarizes, we observe that with adding more workers in the collaborative rendering process, each mobile device's power consumption for frames rendering tends to be decreased. For example, in the 2K resolution case, as Fig. 14a depicts, when there are none (0) worker, i.e., rendering locally, the power reaches to an average value of 4.68 W. It varies on a scale of 3.97 W (1 worker) to 3.7 W (2 workers) to 3.3 W (3 workers). We see that with 3 workers, ButterFly can benefit user 28.3 percent power saving. In addition, by comparing with different pixels, we also find that the value of power saving increased significantly on a scale of 39.5 percent (FHD), 41.2 percent (HD), and 45.1 percent (WVGA). This is because, for lower resolution rendering, device finish more workloads in per unit time. Then, by migrating each workload to workers, the master device will has fewer work locally. It inspires us that ButterFly works well with both high resolution and low resolution, and achieves over 28.3 percent power reduction.

GPU Utilization Variation. We also measure GPU utilization by SSM tools in according to different number of workers. Fig. 15 outlines the variation in four different resolution scenarios. We see that, by using ButterFly, a room of 48 percent (33, 15, 11 percent) GPU utilization is cleared under 2K (FHD, HD, WVGA) pixels scenario. Note that the decreasing trend of clear room demonstrates the heavy workload caused by high quality rendering, e.g., on full 2K rendering, which also leads to power-hungry scenarios.

5.4.3 RTT Responsiveness to 802.11 Environments

As discussed in Section 3.3, a major concern of ButterFly is the impact on goodput and RTT latency, and most importantly how that affects system's efficiency.

Goodput of a Single Device. We first investigate the adaptability of ButterFly to dynamic channel conditions and in 802.11 environments. For every experiment, 100 tests are conducted with persistent collaborative rendering process. Fig. 16 reveals that the peak goodput of ButterFly arrives in 398 KB/s when every 2K (WVGA, HD and FHD) frame detail is rendered at 60 FPS. We also observe that the balanced goodput of per frame rendering has nonlinear correlation with the increasing of display resolution. For example, the goodput of HD case occupies the maximal value (e.g., 341, 172 and 53 KB/s at 60, 30, 10 FPS, respectively) than any other baselines. What's more, the goodput of ButterFly tends to be lower than the peak value in other frame rate cases. In summary, ButterFly can best use every correctly-received bit with each rendered detail.

To explore whether ButterFly provides good frame rates with modest RTT latency overhead above either the network access or the time to render a high quality details, we

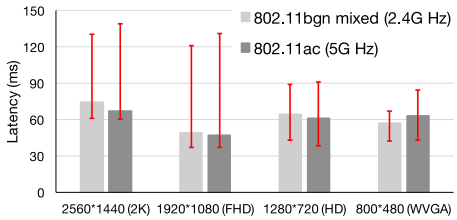


Fig. 17. Latency constraint with different scenarios.

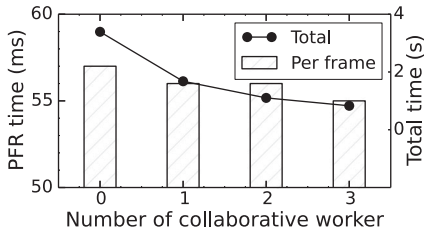


Fig. 18. Average system time consumption of per frame rendering (PTR) and total workloads in using ButterFly.

perform the following experiment. Fig. 17 shows that when the network RTT is high, the input-to-display latency reaches 139 ms. These latency values are high enough that users can perceive a loss in quality. This is consistent with participants' rating the visual quality and smoothness of the mobile devices as better than any of the ButterFly prototype in our user study. Nonetheless, Fig. 17 also suggests that collaborative rendering also did not impact user's rendering performance with balanced delay less than 90 ms in any pixels or over any 802.11 network conditions.

5.5 3D Rendering Performance with VTS

We next evaluate how exactly the VTS technique can benefit user to save energy consumption in 3D-specific OpenGL ES rendering on mobile GPU. To this end, we first explore the efficiency of texture simplification within VTS. On this basis, we then evaluate the benefits for ButterFly master device by jointing collaborative graphics rendering with VTS to maximally achieve GPU optimization.

5.5.1 Energy Optimization by Using VTS

Here, we receive the general energy saving benefits from ButterFly by different texture simplification strength. For comprehensive user preference coverage, we note four energy saving strength in our evaluations as: (1) the baseline, i.e., default 3D texture rendering without any energy optimization; (2) a high energy saving strength that 3D texture rendering are from a specific view of field and 60 percent of the texture are clipped; (3) a median energy saving strength that 3D texture rendering are from a specific view of field and 40 percent of the texture are clipped; and (4) a low energy saving strength that 3D texture rendering are from a specific view of field and only 20 percent of the texture are clipped.

We mainly conduct experiments on a Samsung Galaxy S5 LTE-A smartphone that uses a 2560x1440 default display resolution pixels and a quad-core Qualcomm Snapdragon 805 SoC and with 600 MHz core speed GPU. We use Monsoon Power Monitor tool to measure the total system energy

TABLE 4
Normalized Power per Frame (PPF) of Default Rendering and ButterFly by Different Strength

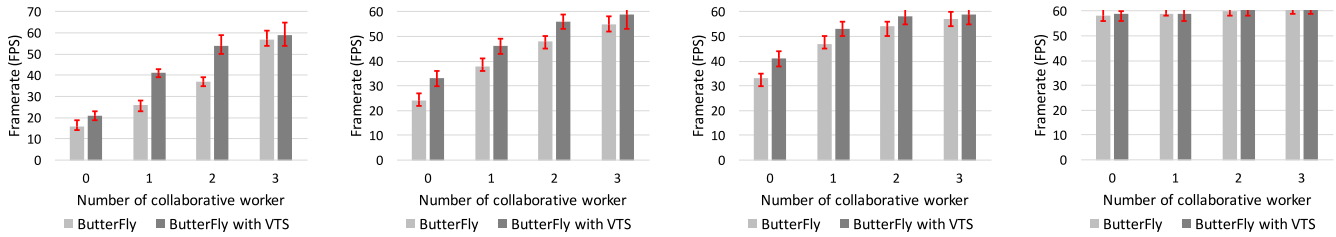
Graphics (Scene) Name and Architecture	Default (0% save)	Energy Saving Strength		
		High	Median	Low
Energy Cost in OpenGL ES Rendering Process				
Dinosaur (*v*:2k‡; *p*:4k*)	3,886 mW	23.2%	19.0%	15.3%
Rocker (*v*:9k‡; *p*:18k*)	3,964 mW	23.1%	19.3%	14.2%
FanDisk (*v*:9.4k‡; *p*:19k*)	3,879 mW	20.4%	16.7%	12.1%
Sphere (*v*:10k‡; *p*:20k*)	3,826 mW	27.2%	20.5%	17.0%
Arma (*v*:23k‡; *p*:45k*)	4,123 mW	16.9%	13.2%	11.2%
Kitten (*v*:24k‡; *p*:49k*)	4,014 mW	24.5%	18.0%	12.9%
Bunny (*v*:35k‡; *p*:70k*)	4,185 mW	21.5%	16.2%	11.6%
Horse (*v*:48k‡; *p*:96k*)	3,919 mW	19.7%	16.2%	13.4%
Budda(*v*:61k‡; *p*:1233k*)	4,215 mW	26.4%	19.2%	16.2%
Dragon (*v*:104k‡; *p*:209k*)	4,230 mW	25.1%	17.6%	14.5%
Average Benefits	4,024 mW	22.8%	17.5%	13.8%

‡*Vertex number;* **Polygon numbers.*

consumed to smartphone. To minimize measurement noise, we only use graphics data on ButterFly but not mobile apps for power and time measurement, remove all unnecessary applications and background services, disable irrelevant hardware components such as camera, GPS and other sensors, and turn off all wireless network interfaces. The screen brightness is set to the lowest level.

We evaluate our techniques using the top 10 popular graphic models according to modern computer graphics related studies in the past two decades [19], [21], [26], [27]. As Table 4 shows, the 10 models include diverse diverse complex graphics architecture and can be obtained from three popular public repositories: Trimble 3D Warehouse, Yobi3D and Sketchfab. As the time consumed to render graphics data can be impacted by its vertex and polygon number. We execute texture simplification along loading graphics data by the current field of view (depends on gyroscope data). Unless otherwise specified, all results presented in this section are averaged through 30 repeatable experiments.

Table 4 also measures the normalized power per frame (PPF) of default rendering and using the revised ButterFly, i.e., enabling VTS feature. We observe that by adding more strict energy saving strength, the revised ButterFly can provide a significant energy saving benefits comparing to the baseline. For the 10 test cases, to render graphics data on default Rajawali, the system power reaches to an average value of 4,024 mW, which occupies almost 53 percent of device's rated power (7.7 Watt). By using our revised ButterFly under three energy saving strength for all the 10 test cases, we see that there extra brings an average 13.8 percent power reduction in using the lowest energy saving case, comparing to the default ButterFly. When we opt to a median texture simplification strength, on average for the 10 test cases, we can reduce the PPF by 17.5 percent, ranging from 13.2 percent (in Arma) to 20.5 percent (in Sphere). Furthermore, if we increase the energy saving to the highest strength, on average for the 10 test cases, we can now reduce the PPF by 22.8 percent, ranging from 16.9 percent (in Arma) to 27.2 percent (in Sphere). The experiment results can be impact by the architecture of graphics model,



(a) 2560*1440 (2K) pixels (b) 1920*1080 (FHD) pixels (c) 1280*720 (HD) pixels (d) 800*480 (WVGA) pixels

Fig. 19. Power consumption variation to different collaborative workloads and rendering qualities.

however, it demonstrates that using VTS for ButterFly can provide further benefits in both energy saving and workload decreasing.

Indeed, the strength of 3D texture simplification in our experiments can be further refined, widely used on more intricate GPU workloads (i.e., more fine-grained and complex models) and free of impact on users' viewing experience [19], which we leave as the future work.

5.5.2 Benefits of VTS in Collaborative Rendering

We now explore how VTS contribute to ButterFly in collaborative GPU rendering. Specifically, we carry out the evaluation under the same experiment setup as Section 5.3) on S5 LTE-A smartphones. We summarize the results in Fig. 19 and reveal that, ButterFly with VTS demands fewer worker devices to achieve the same rendering performance as default ButterFly. For example, as Fig. 19a shows, to obtain 2K display resolution and 30 FPS framerate viewing experience, the revised ButterFly needs only one extra worker device while the default ButterFly requires two worker devices. We further find that such benefit tends to be more significant in heavier GPU workload. For instance, both default ButterFly and revised ButterFly provide over 60 FPS framerate and WVGA (800*480) viewing quality performance with only one extra worker devices. Nevertheless, in the case of with one extra worker device rendering for FHD (1920*1080) viewing resolution rendering, the default ButterFly provides 38 FPS framerate while ButterFly with VTS can reach 47 FPS. This is because the VTS enables GPU to process less workload, and thus be more computation resource for receiving and processing following tasks.

In practice, for some mobile graphics, enabling only one of the techniques may slightly increase the system energy cost. For example, in rendering the Dragon test case, when the VTS feature is heavy invoked, i.e., user change their 3D field of view frequently, the energy cost can range from 1,434 to 3267 mW[7] when there is one time to three time per viewing changes. In the future, we can offload the VTS process to cloud for further benefits.

5.5.3 Overheads of Using VTS

We last evaluate the system overhead in using the revised ButterFly system. Specifically, we measure the processing time (latency), memory usage and energy cost per second for VTS. To minimize measurement noise, we use the most intricate 3D graphics model (the Dragon) as study case, and then average the results for power and time measurement. Similar to previous experiment setup, we also remove all unnecessary applications and background services, disable irrelevant hardware components such as camera, GPS and other sensors, turn off all wireless network interfaces and set screen brightness to the lowest level. We summarize the system overheads in Table 5.

System Latency. As illustrated in Section 4.2, once user change his/her 3D view of filed, the related modules such as gyroscope sensor and VTS program need to notify the ButterFly system, then suspend and restart the OpenGL ES pipeline for a while due to reload the resolution or framerate parameters to the `glViewport()` API and communication latency. To quantify such delay, we manually generate a large number of view of field changing signals into our system when rendering the Dragon model in Table 4. We first measure the total system latency of local VTS processing. To this end, we run the test for 10 minutes. In the test, our system (excluding VTS) costs almost 40 ms for inferring with built-in sensor communication and background ButterFly service running. However, the VTS technique costs average 27.6 s to finish the whole processing even we just execute one time texture simplification within per graphic data loading. It might make user wait for a long time. To address such issue, we can clip potential textures beforehand and load them on-demand instead of default data.

Memory Overhead. Our system allocates extra memory for handling 3D texture simplification and other ButterFly features. To quantify the memory overhead, we measure the extra memory of each technique used in all the 10 test cases in Table 4. Averagely, the VTS technique will consume the most memory space up to 100 MB. In addition, the rest components, e.g., gyroscope sensor, in ButterFly will use around 10 MB memory. Considering that today's mainstream smartphone usually owns 2 to 6 GB RAM space [28]. Thus, all the memory overhead of ButterFly are negligible.

Energy Overhead. The extra operations we used for improving energy efficiency in mobile viewing also bring extra energy cost. Indeed, the power consumption can be broken down into smartphone side and extra hardware side. We use the Monsoon power meter to measure each technique's power consumption along the system latency experiments. The average energy cost of ButterFly system

TABLE 5
System Overhead in Applying VTS on S5 LTE-A

Category	Latency	Memory	Power
System overheads	$\leq$ 27.6 s	100 MB	1.7 W

7. As space is limited, we omit the details of this measurement while instead present the results.

in the 10 minutes' testing is only 35 mW. We shall also emphasis that the VTS can be a bit power consuming as it costs up to 1.7 W in simplifying unnecessary 3D textures. Thus, we again declare the significance to offload the VTS to the cloud, which we leave as another future work.

6 Discussions

With ButterFly, existing graphics-intensive mobile applications can use the developed technique to improve their energy efficiency and image quality. In this section, we discuss potential limitations and future work of ButterFly.

Limitations. Our current system implementation is limited in several aspects. First, it is now only employed by Rajawali for energy and quality optimization. While for the most of the time, graphics-intensive mobile apps such as gamings and VR typically run atop Unity 3D and Google Daydream engines. However, as such engines are closed source yet. It is not sure whether our system can achieve the same or even better performance on them. Fortunately, by just tuning the graphics detail in open source Unity based gaming, i.e., Angry Bird and Call of Duty, we roughly verified the feasibility of ButterFly in these instance. Due to space limited, we omit their experiment details.

ButterFly in COTS Applications. To employ ButterFly in COTS mobile applications, a major source of non-determinism is the pseudo random number generator. In this paper, we did not need to intercept these calls because correctly replaying keyboard and mouse inputs ensures that random-number generation is deterministic. Also, another key challenge for handling user inputs (i.e., game interactions) can be viewed by looking at the rendering loop shown in the previous section. Similar to Kahawai [8], the thread handling this loop sequentially processes buffered inputs, updates the game state, and renders the scene. When master's loop is executing the rendering and screen refresh calls for frame N, the system is buffering input events for frame $N+1$. To keep the workers and master in sync, ButterFly must ensure that when any worker processes an input event at frame N, the master must also process that event at the exact same frame number N. To accomplish this, the worker's render thread timestamps inputs with the appropriate frame number before sending them to the master, which pauses until it receives inputs for next frame.

In fact, one naive approach to input handling in ButterFly can treat the input sampling as part of the rendering loop, while it also creates performance problems. Given the one-way delay from a worker to the master is 13 ms. After the worker finishes rendering frame N, it must send all buffered inputs for frame $N+1$ to the master. The master cannot render the next frame $N+1$ until at least 13 ms after the worker has finished rendering frame N and sampled the input for frame $N+1$. Thus, the application's framerate becomes inversely proportional to the network RTT latency between the worker and master, which is unacceptable for gaming display. ButterFly solves this problem with pipelining by decoupling input processing from the main game loop. Thus, many of ButterFly's tasks are run asynchronously in a pipelined fashion. This pipeline allows ButterFly to asynchronously process and refresh frames rather than waiting on the network to make progress.

Future Work. We plan to extend our current implementation to support more flexible mobile graphic applications, and port our implementation onto more platforms, including Android, iOS and Windows, for more testing.

7 Related Work

In this section, we review prior work in GPU power optimization, collaborative rendering and overdrawn graphics issues in mobile applications. Specifically, we highlight their key differences from ButterFly.

GPU Power Optimization. There are a lot of works on both modeling and further optimizing the GPU power consumption for desktop/server [7], [24], [29], [30] today, that focus on dynamic frequency scaling. [31] and [32] investigate the desktop platform gaming experience by tuning the display resolution and refresh rate, respectively. Results report that lower frame rates may lead to system power reduction. These techniques are generally applicable to smartphones. However, the GPU frequency scaling is controlled in the vendor-provided drivers, which is very difficult to be configured by mobile apps even by calling the OS level interface. However, in ButterFly, we present specific techniques that achieve high quality rendering on mobile devices without modifying the ROM.

GPU and GPGPU Offloading. Collaborative rendering relies on low quality data representations to reduce network usage, but combines locally-computed low quality data with information from a server to give mobile clients access to high quality data. Thus, how to overcome the resource constraints of mobile devices by partitioning programs between a device and more powerful server infrastructure is challenging. Chroma [33] provides programmers to specify program partitions and conditions under which to adopt those partitions, while MAUI [7] and CloneCloud [4] leverage features of managed-language runtimes to automatically partition a program depending on runtime prediction of the partition's energy and performance benefits. However, all of these systems are designed for general-purpose CPU workloads and are inappropriate for applications like fast-action games that heavily rely on a desktop/mobile GPU. To relief this problem, Kahawai [8] adopt a server-side support. Nevertheless, it results in a non-trivial cost of large-scale deployment. Along another direction, in this paper, we utilize crowdsourced mobile GPU to achieve a high quality rendering. With the proliferation of general purpose GPU (GPGPU) computing on mobile platforms, the GPGPU was clear that the low computational capabilities and the limited battery capacity were obviously a bottleneck to users. To this end, researchers have proposed different solutions, e.g., [34], [35], to alleviate the burden of the low-powered devices by migrating the heavy computations from low-powered devices to more powerful machines. Along the same line, ButterFly offers a remarkable solution to the problem of high quality mobile GPGPU computation which can be applied in most COTS Android devices.

Overdrawn Graphics Issues. Much work has characterized mobile GPUs. For example, GraalBench [36] is a 3D benchmark suite for low-end mobile devices and Ma et al. [37] has characterized the power consumption of mobile games. While their methods are inspiring, none has studied the

inefficiency of OpenGL ES pipeline. Minimizing waste of mobile GPU resources has been a hot topic. LPD [38] reduces the memory and display interface traffic by only compositing the recently changed overdrawn UI regions. Android comes with various tools to bust GPU overdraw problems [39]. The goal of ButterFly is orthogonal to them as it avoids overdrawn texture by clipping GPU workloads.

8 Conclusion

We have presented ButterFly and a new technique of it to implement a collaborative system for high quality GPU rendering among mobile devices. ButterFly exploits the peers' resource for migrating GPU workload intelligently. Experiments with our ButterFly prototype demonstrate its potential to provide high visual quality with modest bandwidth and energy overheads. As future work, we will consider extending our system with a comprehensive implementation to support more cellular networks and other OS platforms.

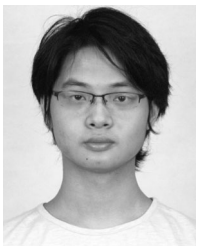
Acknowledgments

We thank the anonymous reviewers for their valuable and insightful comments. This work is supported by Tsinghua University Initiative Scientific Research Program under Grants No. 20161080066, National Key R&D Program of China under Grants No. 2017YFB1003003 and NSF China under Grants No. 61572281. Preliminary results of this paper has been presented in [1].

References

[1] C. Wu, et al., "Butterfly: Mobile collaborative rendering over GPU workload migration," in *Proc. 36th Annu. IEEE Int. Conf. Comput. Commun.*, 2017, pp. 1–9.
[2] Apple Presents iPhone 4, 2010. [Online]. Available: http://www.apple.com/pr/library/2010/06/07Apple-Presents-iPhone-4.html
[3] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Netw. Appl.*, vol. 18, no. 1, pp. 129–140, 2013.
[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.
[5] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 93–106.
[6] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? The bandwidth and energy costs of mobile cloud computing," in *Proc. IEEE INFOCOM*, 2013, pp. 1285–1293.
[7] E. Cuervo, et al., "MAUI: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst. Appl. Services*, 2010, pp. 49–62.
[8] E. Cuervo, et al., "Kahawai: High-quality mobile gaming using GPU offload," in *Proc. 13th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2015, pp. 121–135.
[9] T. L. Saaty and L. G. Vargas, *Decision Making with the Analytic Network Process: Economic, Political, Social and Technological Applications with Benefits, Opportunities, Costs and Risks*. Berlin, Germany: Springer, 2013.
[10] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving GPU energy efficiency," *ACM Comput. Surveys*, vol. 47, no. 2, 2015, Art. no. 19.
[11] Rajawali on Github, 2016. [Online]. Available: https://github.com/Rajawali/Rajawali
[12] Samsung Galaxy S5, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Samsung_Galaxy_S5
[13] G. P. Perrucci, F. H. Fitzek, and J. Widmer, "Survey on energy consumption entities on the smartphone platform," in *Proc. IEEE 73rd Veh. Technol. Conf. Spring*, 2011, pp. 1–6.
[14] Simple System Monitor, 2016. [Online]. Available: https://play.google.com/store/apps/details?id=com.dp.sysmonitor.app
[15] A. Munshi, D. Ginsburg, and D. Shreiner, *OpenGL ES 2.0 Programming Guide*. London, U.K.: Pearson Education, 2008.
[16] Sketchfab repository, 2017. [Online]. Available: https://sketchfab.com
[17] M. Solutions, "Monsoon power monitor," 2016.
[18] D. Ginsburg, B. Purnomo, D. Shreiner, and A. Munshi, *OpenGL ES 3.0 Programming Guide*. Reading, MA, USA: Addison-Wesley, 2014.
[19] E. Zhang and G. Turk, "Visibility-guided simplification," in *Proc. IEEE Vis.*, 2002, pp. 267–274.
[20] A. Carroll and G. Heiser, "The systems hacker's guide to the galaxy energy usage in a modern smartphone," in *Proc. 4th Asia-Pacific Workshop Syst.*, 2013, Art. no. 5.
[21] M. Garland and P. S. Heckbert, "Surface simplification using quadric error metrics," in *Proc. 24th Annu. Conf. Comput. Graph. Interactive Techn.*, 1997, pp. 209–216.
[22] J. F. Blinn, "Backface culling snags (rendering algorithm)," *IEEE Comput. Graph. Appl.*, vol. 13, no. 6, pp. 94–97, Nov. 1993.
[23] Rajawali: Android OpenGL ES 2.0/3.0 Engine, 2017. [Online]. Available: https://github.com/Rajawali/Rajawali
[24] L. Zhang, et al., "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2010, pp. 105–114.
[25] Display resolution, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Display_resolution
[26] K. Zhou, Z.-G. Pan, and J.-Y. Shi, "Mesh simplification algorithm based on triangle collapse," *Chin. J. Comput. Chin. Edition*, vol. 21, pp. 506–513, 1998.
[27] D. Brodsky and B. Watson, "Model simplification through refinement," in *Proc. Graph. Interface*, 2000, pp. 221–228.
[28] E. Confalonieri, P. Amato, D. Balluchi, D. Caraccio, and M. Dallabora, "Mobile memory systems," in *Proc. Mobile Syst. Technol. Workshop*, 2015, pp. 1–7.
[29] S. Hong and H. Kim, "An integrated GPU power and performance model," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 280–289, 2010.
[30] S. He, Y. Liu, and H. Zhou, "Optimizing smartphone power consumption through dynamic resolution scaling," in *Proc. 21st Annu. Int. Conf. Mobile Comput. Netw.*, 2015, pp. 27–39.
[31] J. D. McCarthy, M. A. Sasse, and D. Miras, "Sharp or smooth?: Comparing the effects of quantization versus frame rate for streamed video," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2004, pp. 535–542.
[32] M. Claypool and K. Claypool, "Perspectives, frame rates and resolutions: It's all in the game," in *Proc. 4th Int. Conf. Found. Digit. Games*, 2009, pp. 42–49.
[33] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb, "Simplifying cyber foraging for mobile devices," in *Proc. 5th Int. Conf. Mobile Syst. Appl. Services*, 2007, pp. 272–285.
[34] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2010, pp. 224–231.
[35] R. Montella, C. Ferraro, S. Kosta, V. Pelliccia, and G. Giunta, "Enabling android-based devices to high-end GPGPUs," in *Algorithms and Architectures for Parallel Processing*. Berlin, Germany: Springer, 2016, pp. 118–125.
[36] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha, "GraalBench: A 3D graphics benchmark suite for mobile phones," *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 1–9, 2004.
[37] X. Ma, Z. Deng, M. Dong, and L. Zhong, "Characterizing the performance and power consumption of 3D mobile games," *Comput.*, vol. 46, no. 4, pp. 76–82, 2013.
[38] M. Ham, I. Dae, and C. Choi, "LPD: Low power display mechanism for mobile and wearable devices," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 587–598.
[39] G. Romain, "Android performance case study," 2014, http://www.curious-creature.com/docs/android-performance-case-study-1.html

Chao Wu received the BEng degree in software engineering from Southeast University, Nanjing, China, in 2012 and the PhD degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2017. He currently takes a postdoc position in Tsinghua University. His research interests include mobile computing systems and networked systems with a focus on operating system, wireless communication, and applications. He is a member of the IEEE.



Bowen Yang received the BEng degree in computer science from Tsinghua University, Beijing, China, in 2016, and is currently working toward the MSe degree in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include mobile computing systems, mobile graphics optimization, and applications.

Wenwu Zhu received the PhD degree from the New York University Polytechnic School of Engineering, New York, NY, in 1996. He is currently a professor with the Computer Science Department, Tsinghua University, Beijing, China. His current research interests include multimedia cloud computing, social media computing, multimedia big data, and multimedia communications and networking. He has been serving as EiC for the *IEEE Transactions on Multimedia* since Jan. 1, 2017. He is a fellow of the IEEE.

Yaoxue Zhang received the BS degree from Northwest Institute of Telecommunication Engineering, China and the PhD degree in computer networking from Tohoku University, Japan, in 1989. He is currently a professor with the Computer Science Department, Tsinghua University, Beijing, China. He was a visiting professor of Massachusetts Institute of Technology and the University of Aizu, in 1995 and 1998. His major research interests include computer networking, operating systems, and ubiquitous computing. He is currently a fellow of the Chinese Academy of Engineering. He is a member of the IEEE.