GBooster: Towards Acceleration of GPU-intensive Mobile Applications

Elliott Wen[1], Winston K.G. Seah[1], Bryan Ng[1], Xue Liu[2], Jiannong Cao[3] and Xuefeng Liu[4]

[1]Victoria University of Wellington
[2]McGill University
[3]The Hong Kong Polytechnic University
[4]Huazhong University of Science and Technology

Abstract—**The performance of GPUs on mobile devices is generally the bottleneck of multimedia mobile applications (e.g., 3D games and virtual reality). Previous attempts to tackle the issue mainly migrate GPU computation to servers residing in remote cloud centers. However, the costly network delay is especially undesirable for highly-interactive multimedia applications since a fast response time is critical for user experience. In this paper, we propose GBooster, a system that accelerates multimedia mobile applications by transparently offloading GPU tasks onto neighboring multimedia devices such as Smart TVs and Gaming Consoles. Specifically, GBooster intercepts and redirects system graphics calls by utilizing the Dynamic Linker Hooking technique, which requires no modification of the applications and the mobile systems. In addition, a major concern for offloading is the high energy consumption incurred by network transmissions. To address this concern, GBooster is designed to intelligently switch between the low-power Bluetooth and the high-throughput WiFi based on the traffic demand. We implement GBooster on the Android system and evaluate its performance. The results demonstrate that it can boost applications' frame rates by up to 85%. In terms of power consumption, GBooster can preserve up to 70% energy compared with local execution.**

I. Introduction

Multimedia applications such as 3D games and augmented reality are proliferating in mobile devices nowadays. As current devices are still facing constraints of processing capabilities and battery power, the resource-hungry multimedia applications are inevitably pushing the limit of the devices, causing the applications' running at a low frame rate and leading to short battery lifetime.

A great number of research works such as MAUI [1] and CloneCloud [2] attempt to alleviate these issues by offloading CPU computation to cloud. However, these systems are not effective for the multimedia applications as a majority of them are GPU-intensive (see Section. II). To bridge the gap, a small number of studies focusing on offloading GPU tasks have been carried out. For instance, OnLive [3] and G-Cluster [4] feature a remote-rendering architecture, in which, multimedia applications run in virtual machines residing in cloud servers and the screen-shots of the applications are delivered to users through the Internet. Meanwhile, the users' control inputs are transmitted and replayed in the servers. Recently, a more sophisticated component-based offloading architecture has been proposed in [5]. Instead of executing an entire application in cloud servers, it distributes the application's independent components to either cloud servers or local devices for execution, as determined by the devices' current workload and their network connectivities.

Despite the promising results obtained, these systems possess certain limitations. Specifically, they require an Internet connection with huge bandwidth, which is not always available for users. Moreover, as the cloud servers tend to be remotely located, the network delay incurred by transferring screen-shot frames in the remote-rendering architecture can be fairly costly. This is especially undesirable for some highly-interactive applications such as first person shooter games, where a fast response time is critical to the user experience. Though the component-based architecture may alleviate the latency issue by avoiding screen-shot transmissions, it causes unbearable burden to the developers as they are required to modify and recompile source codes for the legacy software.

To tackle these issues, we introduce GBooster, a system that accelerates GPU-intensive multimedia mobile applications by seamlessly leveraging ambient graphics processing capacities. Specifically, without modification of the applications and the mobile operating systems, GBooster offloads GPU tasks onto neighboring multimedia devices such as gaming consoles, personal computers, and Smart TVs. Since these devices are connected with high speed local area networks (e.g., WiFi) and located at close physical proximity to users, the connections have much higher bandwidth and smaller communication latency compared with the Internet. It thereby guarantees the user experience for highly-interactive applications.

Since the neighboring multimedia devices tend to have heterogeneous hardware architectures and operating systems, it is almost infeasible to adapt the existing solutions that are dedicatedly designed for cloud servers. In this paper, we propose a novel GPU-task offloading approach and take the following challenges into consideration. First of all, mobile applications may be implemented using different graphics engines or even different programming languages, it is considered to be challenging to propose a universal offloading technique that supports all the applications without modifying them. Besides, though offloading compute-intensive GPU tasks saves considerable amounts of power, it also incurs extra network transmissions, which may result in high energy consumption. How to reduce the energy overhead without degrading system performance remains a non-trivial issue. Finally, it is rather common that there exist multiple offloading destinations (i.e., multimedia devices) within a network, how to aggregate the distributed computation capacities to improve the system performance is not a simple task.

In this paper, we develop practical solutions to cope with the above challenges. To enable offloading for unmodified multimedia applications, we intercept system graphics calls and redirect them to nearby devices by utilizing the Dynamic Linker Hooking technique. To lower down the energy overhead of network transmission, GBooster is designed to intelligently switch between the high-throughput high-power WiFi and the low-throughput low-power Bluetooth based on the traffic volume. To harness the processing power from multiple offloading destinations, GBooster parallelizes and dispatches an application's GPU tasks to different devices as determined by their workload and capabilities.

We consolidate the above techniques and implement a prototype system in the Android platform. We evaluate the prototype in terms of application acceleration and energy saving by carrying out experiments on 6 popular mobile games. The results show that GBooster can successfully accelerate the applications' frame rates by up to 85% and save up to 70% energy compared with local execution.

To the best of our knowledge, GBooster is the first system that is able to accelerate all GPU-intensive mobile applications with the help of neighboring multimedia devices. The main contributions of this paper are as follows:

1) We exploit the graphics processing power of ambient multimedia devices via a novel GPU-computation offloading technique, which requires no modification of the existing applications.
2) We propose an approach to reduce energy overhead of offloading by intelligently switching between the high-throughput WiFi and low-power Bluetooth.
3) We conduct experiments to confirm the effectiveness of our prototype for application acceleration and power saving.

II. Motivations

Though a multimedia mobile application generally involves CPU and GPU computation, it is often the case that the GPU is the bottleneck of the application due to its limited processing power and high energy consumption.

Limited GPU Capacities: Table. I demonstrates the recommended CPU/GPU requirements of the most demanding games in recent years including *Modern Combat 5: Blackout* [6] in 2014, *GTA San Andreas* [7] in 2015, and *The Walking Dead: Michonne* [8] in 2016. The table also shows the CPU/GPU capabilities of the mainstream smartphones in those years including *Samsung Galaxy S5* (2014), *LG G4* (2015), and *LG G5* (2016) respectively. Note that the CPU and GPU capabilities are demonstrated in terms of CPU clock rate and GPU fillrate (GPixel/s) respectively. It can be seen that the devices' CPU capacities are commonly beyond the requirements of the most demanding games. On the other hand, the games are pushing the limit of the devices' GPUs, which become the performance bottleneck.

	2014	2015	2016
CPU/GPU requirement	1.5 GHz 3.6 GP/s	1 GHz 4.8 GP/s	1.2 GHz 2-Core 6.7 GP/S
CPU/GPU capability	2.5 GHz 4-Core 3.6 GP/s	1.8 GHz 6-Core 4.8 GP/s	2.15 GHz 4-Core 6.7 GP/s

TABLE I: Game Requirement versus Smartphone Capability. The recommended requirements represent the capabilities needed for running those games in the highest graphics settings and achieving a frame rate of at least 30 frames per second.

To make matters worse, the performance is usually downgraded due to the overheating issue. Nowadays, GPUs tend to create abundant heat energy when they are heavily utilized. To prevent overheating, mobile device systems have to reduce the GPU's operating frequency and suppress its performance when its temperature exceeds certain thresholds. Fig. 1 demonstrates how the GPU frequency and temperature changes with time when the device LG G4 is running the game *GTA San Andreas*. The GPU frequency initial reaches 600Mhz and remains steady for the first 10 minutes. After that, the GPU temperature meets the threshold and the operating frequency drops drastically to 100Mhz. As a consequence, the application's performance is significantly downgraded, resulting in an unacceptable user experience.

High Energy Consumption: Heavy GPU utilization also leads to swift battery drain. To demonstrate this, we run a test program [9] that renders a static triangle at a default frame rate of the Android system, which is 60 frames per second (FPS) on the three mobile devices mentioned above. We then measure the energy consumption incurred by CPU and GPU using approaches introduced in [10] and [11]. The results show that the power usage for each GPU is approximately 3 W, almost 5 times higher than the energy used by the CPU. As the program only performs elementary GPU computation, the power consumption of complex multimedia applications can be far more than this result, which can significantly shorten the battery lifetime.

Fig. 1: GPU frequency trace.

To alleviate these issues, conventional solutions typically offload GPU tasks to remote cloud servers. However, these cloud-based solutions usually require an Internet connection with huge bandwidth, which is not always available for users. Moreover, the long physical proximity of the cloud centers usually leads to high network latency, which is undesirable for highly-interactive multimedia applications.

In light of the above issues, we introduce GBooster, a novel GPU-task offloading system that aims to meet the following objectives.

High FPS, Low Latency, and No Requirement of the Internet. GBooster enables smart devices to run a GPU-intensive multimedia application at a high frame rate and low latency without an Internet connection.

Extend Battery Life. GBooster reduces energy consumption of multimedia applications and extends the battery lifetime.

GBooster achieves these goals by exploiting the processing power of neighboring multimedia devices including game consoles, smart TVs, and PCs. These devices possess two essential advantages compared with cloud servers. First, they are prevalent and equipped with abundant processing power. According to [12] and [13], 80% of U.S households own a game console and half of them own a smart TV. All these devices are usually equipped with powerful GPU chipsets. For instance, the game console *Nvidia Shield* [14], is equipped with a GPU with a fillrate up to 16 GP/s, making it an ideal offloading destination. Besides, traditional PCs could be another sound option as modern computers generally possess GPUs that are 10 times more powerful than mobile devices' [15]. Second, these devices are typically connected with a local area network (e.g., in-home WiFi), which provides significantly larger bandwidth and smaller latency compared with an Internet connection.

GBooster works on every commercial Android device and supports all multimedia applications without changing or recompiling any source codes of the applications and the Android operating system. Note that GPU is not only heavily used by high-end 3D applications like games, but also widely used for rendering 2D user interfaces for various non-gaming applications. In this paper, we mainly focus on gaming applications, but we will show that non-gaming applications can also benefit from our system.

Fig. 2: System Architecture

III. System Overview

Fig. 2 depicts the system architecture, which contains the essential procedures to migrate graphics computation from a user mobile device to an offloading destination. In the following sections, we refer to a user's mobile device as a **User Device**. Besides, we refer to an offloading destination as a **Service Device**.

As shown in Fig. 2, GBooster first dynamically inserts one wrapper layer to the user device while a multimedia application starts running. The wrapper enables the system to intercept all graphics commands from the application and redirect them to a remote service device. Based on the received commands, the service device conducts the graphics computation using its own GPU. Once the computation is done, the rendered results will be encoded and delivered to the user device. Finally, the user device decodes and displays the images on the device's screen. We will describe the whole process in details in Section IV.

In this architecture, the network communication between the user device and service device plays an essential role. In order to improve network performance and reduce energy consumption, we propose an approach that eliminates redundant data and intelligently switches among multiple wireless interfaces based on the traffic volume. The details will be elaborated in Section V.

Note that Fig 2 only depicts the scenario with one user device and one service device for better demonstration purposes. In Section VI, we extend the system such that it can aggregate distributed processing capabilities from multiple service devices to obtain further performance improvement.

IV. Enable GPU Task Offloading

The Android system provides high performance graphics processing support for multimedia applications with the help of Open Graphics Library named OpenGL ES [16]. OpenGL ES is a cross-platform graphics API that specifies a standard interface for GPU and the Android applications could invoke the APIs to directly interact with the GPU. OpenGL ES features a client-server model as shown in Fig. 3. The application that invokes OpenGL ES APIs behaves as a client. It keeps generating a series of graphics commands to the server component. The server, which is typically executed in the GPU, interprets the commands and performs the actual graphic computation.

Based on this model, we propose an offloading approach that we intercept the command streams from a OpenGL client and redirect them to a OpenGL ES server residing in a remote machine. This approach contains two key advantages. First, it works universally for every multimedia application regardless of its implementation details (e.g., programming languages or graphics engines), since all of them internally invoke OpenGL ES calls. Besides, it requires no source code modification of the legacy applications.

Fig. 3: The Client/Server Model of OpenGL ES.

A. *Intercepting and Rewriting OpenGL ES Functions*

Although this approach seems straightforward, implementing it entails a challenging issue; the OpenGL implementation in Android OS is closed-source and thus we cannot revise it to add functionalities for interception and redirection. To address this issue, we adopt a technique named Dynamic Linker Hooking [17].

Specifically, hooking is the process of intercepting a program's execution at a specific point, typically entries of functions, in order to alter or augment the program's behavior. The dynamic linker hooking technique enables hooking in the runtime by forcing a program to load shared libraries specified by the user instead of the original ones provided by operating systems. In our case, rather than the genuine OpenGL ES library provided by the Android multimedia framework, we instruct the applications to load a wrapper library, which intercepts all the graphics command calls.

In detail, we notice that an application could invoke the OpenGL ES graphics APIs in three different ways:

1) An application may link to an OpenGL ES library so that it can directly call the OpenGL ES APIs.

2) An application may utilize the *eglGetProcAddress* function to get pointers to the OpenGL ES APIs.
3) Less likely, an application uses system calls *dlopen* and *dlsym* to dynamically load the OpenGL ES APIs.

Therefore, we have to intercept the OpenGL ES APIs in all these situations. For the first case, we simply implement wrapper functions for all the OpenGL ES APIs in our wrapper library. We then force the application to use the wrapper library by applying the Dynamic Linker Hooking technique. It is worth to note that the hooking can be easily done by setting the application's LD_PRELOAD environment variable in the Android system. Regarding the second case, we intercept and rewrite the *eglGetProcAddress* function such that it directly returns the pointers pointing to our wrapper functions. Similarly, we handle the third case by rewriting the *dlopen* and *dlsym* functions so that they load our wrapper library in preference of the original OpenGL ES library.

B. Forwarding Graphics Commands

We are now able to capture the OpenGL commands and ready to forward them to a remote service device. To facilitate network transmissions, we first need to serialize the commands' parameters.

OpenGL ES commands contain two types of parameters; one is the basic data types (e.g., integer and string) and the other one is the pointer type. It is straightforward to handle basic data types as we can easily calculate the length of the data. On the other hand, the situation becomes complicated when dealing with pointers. A pointer parameter typically refers to a sequence of data stored in RAM. Generally, the length of the sequence is either provided as a parameter or could be calculated with prior knowledge of its data structure layout. However, a heavily-invoked function *glVertexAttribPointer* contains a pointer parameter whose size could not be determined at the moment we intercept the function. Instead, the actual length is only revealed in consecutive drawing commands (e.g., *glDrawElements*) which render geometries using the pointer. To enable correct serialization, our system defers the transmission of the *glVertexAttribPointer* command until the pointer size is obtained in the later calls. We found that the reorder does not influence the final results so long as *glVertexAttribPointer* appears before the drawing calls.

Once the serialization is done, we could start transmitting the data over a network connection. Since the graphics commands must be delivered to a remote service device in a reliable and in-order manner, we may select TCP as the transmission protocol. However, due to its complex retransmission mechanism, TCP possesses an inherent delay, which is approximately 40 ms in general settings [18] and could be significantly higher under a poor network condition. To alleviate the delay, instead of TCP, we select the UDP transportation protocol to provide fast delivery of the graphics commands. To prevent packet loss and out-of-order delivery, we implement a light-weight and reliable transmission mechanism in the application layer [19].

C. Executing Commands and Retrieving Results

Upon receiving the graphics commands, the service device delivers them to its local GPU for execution. Since GPUs in the majority of multimedia devices provide native support for OpenGL ES, the service device simply acts as a relay and feeds the commands into the GPUs directly. Regarding a small number of devices such as Mac OS X that lack support for OpenGL ES, we could still bypass the restriction by utilizing OpenGL ES emulators [20] that translate OpenGL ES API calls to other natively graphics API calls.

When the computation is completed, the rendered images are transmitted back to the user device for display purposes. The display system of Android adopts a double-buffering mechanism to reduce image flicker and tearing [21]. As a sequence, when an application decides to redraw the screen, it has to invoke a graphic API named *SwapBuffer*. The API will notify the Android system to retrieve rendered images from the GPU and draw them on the screen. However, in our case, the rendered images are obtained from the network rather than the local GPU. To tackle this issue, our system intercepts and changes the behavior of the *SwapBuffer* command; upon intercepting the command, our system directly forwards an image received from network to the Android system for display.

V. Energy-Saving Network Transmission

By offloading GPU tasks, GBooster reduces the power consumption of high-power GPUs. However, it comes at the energy expense of network transmissions, which may negatively impact the battery life. Considering that the energy cost of a WiFi interface is nearly proportional to the traffic load [22], we propose several approaches to reduce traffic volume. Besides, we reveal the potential of the low-power low-throughput BlueTooth interfaces to further suppress energy consumption.

A. Eliminating Redundancy of Network Traffic

GBooster transmitting unoptimized traffic data consumes enormous bandwidth (approximately 200 Mbps) even with a low-quality graphics setting (i.e., a resolution of 600×480 with 25 FPS). We investigate this issue and notice that the traffic data including graphics commands and rendered images contains vast redundancy.

First, the sequences of graphics commands to generate consecutive frames tend to contain huge similarities. For example, an application might draw a same object with two different rotation angles, in which, the corresponding sequences may only differ slightly in the rotation command. We eliminate the redundancy by applying the LRU caching algorithm; the system caches the latest and frequent commands on the user device and the service device. Thereby, the user device can skip transmitting the commands which are cached. Besides, we further reduce the redundancy by using a light-weight general stream compression algorithm named *LZ4* [23], which achieves a compression ratio of 70% while barely incurs extra CPU workload.

Besides, the raw rendered images contain enormous redundancy, since the consecutive frames are typically similar to each other, especially when the images are static or barely vary. One straightforward solution is to encode the images into a video stream using the video encoder *x264* [24], which is considered the most efficient one. However, because the majority of multimedia devices other than PCs are equipped with ARM-based CPUs that the encoder is not optimized for, the encoding process is unacceptably slow. The normal speed is only around 1 MegaPixels/sec, far less than the speed of 7 MegaPixel/sec in which the application generates raw frames. Clearly, this approach fails to meet the requirement of real-time encoding. Rather than using a video encoder, we adopt a lightweight image encoding algorithm named Turbo [25]. The image encoder eliminates the redundant data by only transmitting incremental updates between consecutive frames and utilizing the JPEG image compression algorithm. It can provide a much more rapid encoding speed (up to 90MegaPixel/sec) and a high compression ratio (up to 25:1) without incurring heavy CPU load.

B. Enabling Transmission via Low-Power Interfaces

Nowadays, mobile devices are typically equipped with Bluetooth and WiFi. Wi-Fi interfaces offer a high-bandwidth data-link (up to 450 Mbps) while at the cost of high energy consumption (around 2 W when transmitting at the highest rate) [22]. On the other hand, Bluetooth is an order of magnitude more power efficient (less then 0.1 W) than WiFi, but with an order of magnitude lower bandwidth (approximately 21 Mbps) [26]. This presents us a chance to reduce power consumption by leveraging Bluetooth for network transmission on the premise of meeting demand of network traffic. In our system, we implement a mechanism that dynamically switches between the Bluetooth and the WiFi to meet the traffic demand while to preserve energy as much as possible.

However, implementing it entails a challenging issue resulted from the latency of switching the state of the WiFi interfaces [27]. Our preliminary experiments show that it takes at least 100 ms to wake up a disabled WiFi interface. More frequently, the interface has to re-associate with its access point after being in sleep mode awhile, making the wakeup time much longer (more than 500 ms). Consider a scenario that a system is transmitting data via its Bluetooth interface. If the increasing traffic load exceeds the throughput of the Bluetooth interface, the system has to enable the WiFi interface immediately in order to meet the demand. As the WiFi interface can not be fully functional instantly, the exceeding packets may be lost and retransmitting them will result in high network latency and frame jitter.

We address this issue by applying time-series analysis techniques, which enable us to foresee the escalating traffic trend and to turn on the WiFi interface beforehand. In other word, our objective is to predict traffic volume y_{T+h} given the information available at time T for $h > 0$. Mathematically speaking, we would like to obtain a forecast:

$$y_{T+h|T} = E(y_{T+h}|y_1, ..., y_T), \tag{1}$$

such that $y_{T+h|T}$ has minimum mean square forecast error (MSFE). To achieve this purpose, we first attempt to model the traffic volume with the widely-used Auto Regressive Moving Average (ARMA) model [28]. Specifically, $ARMA(p, q)$ with p autoregressive terms and q moving average terms can be described as follows:

$$y_t = \epsilon_t + \sum_{i=1}^{p} \varphi_i y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}, \tag{2}$$

where ϵ_t are white noise terms and $\epsilon_t \overset{iid}{\sim} Normal(0, \sigma_w^2)$, φ_i and θ_i are parameters for this model.

We conduct preliminary experiments to measure the prediction performance including False Negative (FN) rate and False Positive (FP) rate. The FNs refer to the scenarios that the model fails to predict a soaring traffic demand that exceeds BlueTooth throughput. Conversely, FPs describe the cases that the model wrongly forecasts a traffic demand overpassing the Bluetooth throughput. Clearly, a small FN rate is more important to the system compared with a small FP rate, because a FN case results in elevated network latency while a FP scenario just causes slight increase in energy consumption. Our experiments show that the ARMA model provides a FP rate of 23.7% and a FN rate of 35.1%.

We notice that the FN rate is rather high and negatively impacts the system performance. We investigate the cause and realize that ARMA attempts to recognize and fit the time series pattern solely based on historic traffic data. However, the pattern beneath the traffic demand of our system is also affected by other exogenous factors. For instance, burst touching events from users may lead to drastic changes in game scenes and transmitting the varying scenes may escalate the network traffic. However, this abrupt change caused by external factors may not be modeled by the ARMA instantly, resulting in a FN scenario.

To tackle this issue, we adopt the Auto Regressive Moving Average with Exogenous Inputs model (ARMAX). Specifically, the $ARMAX(p, q, b)$ with extra b exogenous input terms can be formulated as:

$$X_t = \epsilon_t + \sum_{i=1}^{p} \varphi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \sum_{i=1}^{b} \eta_i d_{t-i}, \tag{3}$$

where $\eta_1, \ldots, \eta_b$ are the parameters of the exogenous input d_t. The model enables us to model deterministic and stochastic parts of the system independently. Thereby, we now can take some external inputs of the system into consideration and achieve better prediction performance.

To fit the traffic data in the ARMAX model, we first need to identify the effective exogenous input attributes for our system. We have examined the following potential attributes:

1) Touchstroke frequency: As we mention above, the touchstroke information may be informative. We could obtain the touchstroke information from a system file which is located in /proc/interrupts.
2) Length of graphics command sequences for each frame. A frame composed by a large number of commands

likely has a complicated scene. It generates more traffic when delivering the frame.

3) Number of textures used in each frame. A frame filled out with a great number of textures tends to have a complicated scene. Transmitting it may consume more bandwidth.
4) Number of different graphics commands between two consecutive frames. If the command sequences composing two consecutive frames have immense difference, the scenes of the frames tend to vary significantly. Transmitting them may request more bandwidth.

We evaluate the qualities of the models consisted of different combinations of attributes by accessing the Raw Akaike Information Criteria (AIC) [29]. The results show that the best approximating model for the traffic is the one with the attribute 1 and 3.

In our implementation, we apply a recursive algorithm [30] for online estimating and updating the order (i.e., p, q, and b) and the corresponding parameters (i.e., φ_i, θ_i, and η_i) of the model. We forecast traffic demand for 500 ms and the experiment results show that the model could achieve a FP rate of 23% and a FN rate of 17%, outweighing the conventional ARMA model. When a soaring traffic trend that will exceed the Bluetooth throughput is predicted, our system turns on the WiFi interface and then configures the default route to direct the traffic through the interface. This process is performed smoothly and barely incurs packet loss.

VI. Harnessing Capacities from Multiple Service Devices

As it is fairly common that there exist multiple service devices within a network, we may naturally raise a question: whether it is feasible to harness capacities from a cluster of service devices? Specifically, whether it is possible to parallelize and distribute GPU tasks to multiple service devices such that we can obtain a further speedup (i.e., a higher FPS)? The answer turns out to be positive.

Fig. 4: Distributed Computation of GPU Tasks.

A. Parallelizing GPU Computation

The key of parallelization lies in the OpenGL ES internal mechanism for handling rendering requests. A rendering request is defined as a sequence of graphics commands for rendering a frame and will be executed in a non-preemptive way according to the modern GPU architecture [31]. Generally, OpenGL ES handles rendering requests in an asynchronous manner. In other words, when an application issues a request to initiate rendering, it is not guaranteed that the rendering request is delivered to GPU and executed right away. Instead, the request may be buffered by the OpenGL ES client to optimize system performance, since it avoids frequent time-consuming input/output operations between CPU and GPU. We take advantage of this mechanism and achieve distributed computation as shown in Fig. 4; whenever there are multiple pending requests in the internal buffer, we distribute and simultaneously execute them in different machines so that we could obtain a higher FPS.

However, in reality an application, after submitting a rendering request, tends to issue a *SwapBuffer* command. The command halts the application and waits for the results from the GPU. In this way, the application forces the GPU to execute its requests immediately and a new rendering request will be issued only if the preceding one is finished. As a consequence, there is at most one request in the internal buffer, rendering the parallelization infeasible.

We overcome this problem by altering the behavior of the *SwapBuffer* command. Invoking the modified command returns immediately and does not halt the application. In this manner, the application will generate rendering requests at its quickest rate and multiple requests could be buffered.

B. Maintaining State Consistency among Devices

To enable distributed computation, we have to overcome another technique hitch due to the stateful nature of OpenGL ES APIs. All OpenGL ES calls are implicitly associated with an OpenGL context parameter, which is essentially a state machine that stores all data related to the rendering process such as the cached textures and vertex programs. Invoking an OpenGL API call on different contexts likely generates different results or even leads to unexpected errors. Since each service device possesses its own OpenGL context, simply distributing requests to them does not guarantee the correctness of the distributed computation.

To ensure the correctness, we have to maintain the consistency of the states among different service devices. We achieve this by first identifying the graphics commands which may alter the OpenGL states. Upon intercepting such commands, we replicate and deliver them to all service devices such that the states are consistent among all the devices. As we need to transmit duplicated data to multiple devices, a unicast connection is not an optimal option since it could result in waste of network bandwidth and limited system scalability. Instead, we take advantage of the multi-cast capability of UDP, which allows a stream of data to be sent to multiple destinations with a single transmission operation to reduce network traffic.

C. Assigning Requests to Devices

The last question left is which service device a request should be assigned to. Clearly, our objective is to assign each request to a service device that can deliver the result in the least time. Mathematically speaking, considering there exist N service devices, we dispatch each request to a node n that satisfies the following criterion:

$$n = \arg\min_{j}(w^j + r)/c^j + l^j, \text{ for } j \in [1, ..., N], \quad (4)$$

where the workload of the request is denoted as r and the computation capability of the service device n is denoted as c^n. Besides, we let l^n denote the round-trip delay time between the user device and the service device n and let w^n be the

	Genre	Package Size
G1: GTA San Andreas [7]	Action	2.41 GB
G2: Modern Combat [6]	Action	0.89 GB
G3: Star Wars [32]	Role playing	2.4 GB
G4: Final Fantasy [33]	Role playing	3.05 GB
G5: Candy Crush [34]	Puzzle	0.17 GB
G6: Cut the Rope [35]	Puzzle	0.12 GB

TABLE II: Games for experiments and their package size.

workload of preceding tasks in its service queue. Note that the workload of each graphics command is profiled using the approach in [31] beforehand.

As this mechanism does not guarantee that a preceding request is finished earlier than a subsequent request, our system keeps track of the sequence numbers of the requests, such that we can display their results in a proper order.

VII. System Evaluation

In this section, we provide the detailed performance evaluation on GBooster.

A. Sample Games and Devices

Applications: We select six popular mobile games spanning three major game genres as shown in Table. II. The majority of them have a large installation package size (above 500 MB) and a high requirement for graphics processing power.

User Devices: We run these applications on a set of smartphones including LG Nexus 5 (2013) and LG G5 (2016), which correspond to old-generation and latest device models respectively. These smartphones are installed with different versions of Android, demonstrating that our system has good system compatibility.

Service Devices: We deploy and test our prototype on different types of multimedia appliances including a game console (Nvidia Shield), a smart TV box (Minix Neo Ui), a ladtop (Dell M4600), and desktop computers (Dell Optiplex 9010 with Nvidia GTX 750 Ti GPUs). All these service devices and smartphones are fully connected via a TP-Link WR802 router providing a 150 Mbps 802.11n WiFi network. To simplify the evaluation process, only one service device (the game console) is utilized in the following experiments for application acceleration and power saving. The PCs are only used in the evaluation for the scenarios with multiple service devices.

B. Application Acceleration

We first evaluate the effectiveness of application acceleration. In our experiment, we choose two FPS metrics that are widely used for measuring user experience of gameplay [36]. The first one is **median FPS** that represents the commonest frame rate experienced in the game and broadly correlates to what the player observes as graphical smoothness. A key advantage of using the median FPS that it naturally omits fringe results, for instance, 0 FPS or 60 FPS which commonly occur during a game's loading screens and menus. Besides, we are also interested in the **FPS stability** which is defined as how much of a game session is played within a 20 percent range of median FPS. If the stability is low, it can serve as an indicator that gameplay is prone to frequent occurrence of FPS jitters, which typically lead to poor gaming experience. Apart from the FPS metrics, another essential factor that affects the gaming experience is the **average response time**. This metric t_r represents the average timespan between the moment a rendering request is issued and the time its result is displayed on its screen. Clearly, when the application is executed locally, this metric is equal to the reciprocal of the FPS (i.e., $t_r = 1000/FPS$). If the computation is offloaded, the metric also includes the time t_p spent on the offloading intermediate steps such as network transmissions and image encoding. In other words, this metric can be represented as:

$$t_r = 1000/FPS + t_p. \tag{5}$$

We conduct the experiments in controlled conditions that we play a game for 15 minutes with the same graphical settings (where configurable) and on the same levels (where there is a choice), meanwhile shutting down other applications on the phone. For comparison purposes, the experiments are conducted twice; one is with our system enabled while the other is not. The results are demonstrated in Fig. 5.

Effectiveness on Old-Generation Devices: One first observation from Fig. 5 (a) and Fig. 5 (b) is that the system boosts the median FPS and increases the FPS stability for each game on the old-generation device Nexus 5. In particular, the performance improvement for the action game G1 and G2 is rather significant. The median FPS drastically rises from 23 and 22 to 37 and 40 respectively. Generally speaking, a minimum standard for good playability is a median frame rate of 24 FPS, as this means that most of the game is played at a frame rate similar to a standard animation or film. However, action games such as shooting games tend to have a slightly higher FPS requirement (usually above 30 FPS) in order to display smooth motion of onscreen objects and maintain the illusion of being real for players [37]. Our results indicate that with the help of our system, the players now can enjoy decent playability of the two action games. We also notice that although the remaining games receive performance improvement as well, it is somewhat less significant than the action games. Specifically, the median FPS of the puzzle game G5 merely improves from 50 FPS to 52 FPS. It may be due to that the puzzle games, which contain only a small amount of animation, are less GPU-intensive than the action games. Thus, the local GPU can handle the computation efficiently and the benefits of remote execution are less obvious.

Regarding the FPS stability, we can spot similar patterns as the median FPS. The system improves the FPS stability for all the games. In particular, the FPS stability for the two action games soars form 60% and 55% to 75% and 74% respectively. This phenomenon can be explained by the better overheating-proof design of the service device. As described in the Section. II, one major reason that results in unstable FPS is the GPU overheating. Since the GPU in the service device is usually equipped with cooling fans, it is less prone

Fig. 5: Median FPS and FPS stability.

to overheating issue and can deliver stable processing power for the applications.

We now turn our attention to the average response time. It can be seen in Fig. 5 (c) that the response time for all the games is below 36 ms. As the average response time for human being is generally above 100 ms [38], the result indicates that the players can barely perceive any response lag when running an application with our system enabled. Another interesting observation is that the impact on response time varies according to the genres of the games. The response time for the action games drops approximately 10 ms, while the time only decreases around 2 ms for the role-playing games. Conversely, the response time for the puzzle games increases 4 ms. The phenomenon can be explained by the trade-off between the gain of FPS and the extra time t_p for the offloading intermediate steps as described in Equation. 5. For the action games, the gain of FPS is significant, thus the drop of response time largely outweighs the t_p. Regarding the role-playing games, the FPS gain seems to neutralize the time t_p, barely causing any change in the response time. In contrast, there is bare FPS gain for the puzzle games, in which case, the t_p is largely attributed to the increase of the response time.

Effectiveness on New-Generation Devices: It can be seen in Fig. 5 (d) and Fig. 5 (e) that our prototype barely benefits the two metrics when running in the new-generation device LG G5. It is mainly due to that the device now possesses an powerful GPU and can efficiently handle all the computation tasks locally. Even for the GPU-intensive action games, the device can achieve a considerable frame rate of 40 FPS, which is approximately 2 times higher than that on the Nexus 5. Therefore, there is little room for performance improvement via remote execution. The tiny gain of FPS then results in the increase of the response time for all the games.

C. Power Saving

We evaluate how much power can be saved with the help of our prototype. Specifically, we run the sample games on the two smartphones mentioned above and measure the system power using a tool introduced in [10]. To obtain accurate results, we first turn the phone into airplane mode, reduce the backlight brightness to 50%, and shutdown other background activities. We also cool down the phones before each test to make sure that the GPU can keep working at a stable frequency during the experiment. We select a specific repeatable scene as the test case and each is repeated for five times. In order to conveniently demonstrate the effectiveness of energy saving for different games, we normalize the results to the case of local execution.

Fig. 6 (a) shows the experiment results. One major observation is that the prototype reduces the power consumption for all the games and smartphones. It is expected since offloading avoids the heavy utilization of high-power GPUs. Clearly, the more intensive the GPU tasks are, the more benefits we can obtain from the offloading. It explains the phenomenon that the GPU-intensive action game G2 could achieve a normalized energy saving of around 70%, while the energy-saving for the puzzle game G6 is less effective, which is approximately 30%.

In addition, to demonstrate the effectiveness of the energy-saving interface switching mechanism, we also measure the power consumption with the optimization disabled. As shown in Fig. 6 (b), the overall system power significantly increases. In particular, the power consumption of the G1 soars from around 40% to 65%. It indicates that the system could preserve considerable amounts of energy by leveraging the low-power Bluetooth interface for network transmissions.

Fig. 6: Normalized Energy Consumption for Different Games.

D. Multiple Devices

We now evaluate the system performance when multiple service devices are available. Specifically, we measure the FPS performance metrics of the action game G1 on the Nexus 5 Meanwhile, we gradually increase the number of service devices.

Fig. 7 demonstrates the experiment results. When the device number is zero, the game is executed locally. As the device number changes to one, the game obtains the most FPS improvement owning to the offloading. When two more devices are available, the FPS gains a significant increment from 40 to 51 by taking advantage of the distributed computation. However, the FPS barely increases and remains stable when more than 3 devices are available.

We examine the cause and notice that the internal buffer possesses at most 3 requests most of the time. Therefore, having more than 3 devices barely benefits the performance. The limited number of requests is possibly due to two reasons. First of all, most of graphics engines have a mechanism to ensure that the FPS does not exceed the device's maximum frame rate (60FPS). Thus, the speed of generating rendering requests may be limited. Besides, generating the requests consumes CPU resources and the number may also be constrained by the CPU.

In terms of FPS stability, it shows a similar pattern as the FPS metrics; the stability increases steadily as the device number is less than 3, and remains stable after that.

E. Performance on non-gaming apps

Although we mainly focus on GPU-intensive mobile games in this work, we also evaluate what non-gaming applications can benefit from our prototype. We measure the effectiveness of application acceleration and power saving of three popular non-gaming applications including *Ebook Reader* [39], *Yahoo*

Fig. 7: FPS Metrics with Multiple Service Devices.

Application Name	FPS Boost	Energy Consumption
Ebook Reader	0	92.1%
Yahoo Weather	0	93.6%
Tumblr	0	93.3%

TABLE III: FPS Boost and Normalized Energy Consumption for Non-gaming Applications.

Weather [40], and *Tumblr* [41]. We utilize *MonkeyRunner* [42] to generate same sets of touch events for repeatable tests including reading an article, viewing weather information, and browsing a post for ten times.

Table. III demonstrates the experiment results. It can be seen that our prototype provides tiny energy saving (7% on average) and no FPS boost for the applications. It is expected since these applications generate much less GPU workload, compared to the games. However, The power saving is still valuable, considering that battery resource is rather scarce on smartphones.

F. Comparison with Cloud-based Solutions

For comparison purposes, we also evaluate the performance of the most popular cloud-based solution *OnLive* [3]. Specifically, we measure the median FPS and response time by adopting the measurement method in [43]. It is worth to note that unlike our system that universally supports all mobile multimedia applications, the platform offers a limited number of application choices. We conduct our tests on ten games and report the average results: with an Internet connection of 10 Mbps bandwidth, the platform can stream games at a resolution of 1280 $\times$ 720 with a frame rate of 30 FPS and average response time of approximately 150 ms. We notice that the FPS is capped at 30 FPS because of the setting of the video encoder used by the platform. Moreover, the average response time is almost 5 times longer than our prototype's due to the long proximity to the cloud server.

G. System Overhead

Memory Overhead. Our system allocates extra memory in user devices. To quantify the memory overhead, we measure the extra memory consumption in the games shown in Table. II. The experiment results show that the average memory footprint is fairly small, which is 47.8 MB. Considering typical smart devices are equipped with gigabytes of memory space, the memory overhead is almost negligible.

CPU Overhead. Our system consumes extra CPU resources for intermediate procedures of offloading such as data compression and image decoding. We measure the extra CPU usages on the Nexus 5 phone. The results show that when running locally, the most compute-intensive application G1 accounts for an average CPU usage of 68%. When the offloading is enabled, the CPU load increases to 79%. Clearly, the device's CPU is still underutilized and the tiny increment of CPU usage barely impacts the system performance.

VIII. Further Discussions

GBooster has made some advances towards acceleration of GPU-intensive mobile applications. Still, it bears several limitations that needs further improvement.

Scenarios without Available Devices Although GBooster outweighs existing cloud-based solutions in terms of response time and frame rate, it does not imply that GBooster can take place of the cloud-based solutions. Under some rare circumstances where there is no available multimedia device nearby, the cloud-based platforms could still provide service to users.

Different Mobile Operating Systems. Our current prototype is only implemented on the Android operating systems. We are investigating how to enable GBooster in other mobile platforms such as iOS and Windows Phone. Since the iOS utilizes OpenGL ES as the Android does, we may be able to directly port GBooster to iOS. Although Windows Phone uses a different graphics API named *Direct X* [44], we could still utilize the same API hooking technique and implement the corresponding wrapper library to support it.

Towards Multiple Users. The prototype is designed to serve multiple users simultaneously. All the service devices maintain a queue buffering the incoming requests and submit them to GPU for execution in a First-Come-First-Served (FCFS) manner. However, it takes no consideration of the tasks' priorities, which could be problematic for time-critical applications. For instance, when an fast-paced shooting game and a chess game that requests thoughtful consideration for each movement are running simultaneously, requests from the shooting game should receive higher processing priorities in order to provide the player with a fast response time. We plan to purpose sophisticated scheduling algorithms to meet requirement from multiple users.

IX. Related Work

Though last decade have witnessed tremendous performance improvement on mobile devices, various novel applications such as 3D mobile games and augmented reality are still overshadowed by the devices' hardware constraints of processing capabilities and battery power.

A considerable number of useful systems attempt to alleviate these issues by offloading CPU computation to cloud servers. Specifically, they first partition a mobile application into various sub-components. Then the compute-intensive components will be migrated and executed in remote cloud servers. These systems can be divided into different categories based on the granularity of the components such as method-based systems [1], class-based systems [45], thread-based systems [2], and process-based systems [46].

However, despite the promising results obtained, these systems may not fully address the issues of mobile multimedia applications due to lack of support for GPU task offloading. To bridge the gap, several research works such as OnLive [3] and Gaikai [47] propose a remote rendering architecture. Specifically, the multimedia applications are executed in cloud servers and the video frames are transmitted to users through the Internet. At the same time, the players' inputs are delivered and relayed in the corresponding server. This approach enables the players to run sophisticated applications regardless of their restricted hardware. However, transmitting a huge volume of video could consume a great amount of network bandwidth and lead to high network latency. Though a great number of works such as [48] and [49] attempt to alleviate the latency issue by optimizing the parameters of video encoding and data compression, the intrinsic delay imposed by the long-range network connections is still non-negligible.

Recently, a component-based solution has been proposed in [5]. This approach partitions a game engine into several independent components and dispatches the selected components from cloud to players' devices as determined by devices' workload and their network connectivity. Though the architecture may alleviate the latency issue by avoiding video transmission, it causes unbearable burden to game developers who are now required to modify the source codes of the legacy applications.

X. Conclusion

In this paper, we propose GBooster, a system that accelerates GPU-intensive mobile applications by transparently offloading GPU computation onto ambient multimedia devices such as SmartTV and Gaming Consoles. We implement GBooster on the Android platform and demonstrate that the prototype can significantly increase applications' frame rates and reduce their energy consumption.

Acknowledgement

The work of Elliott Wen, Winston K.G. Seah and Bryan Ng was supported in part by Victoria University's Huawei NZ Research Programme, Software-Defined Green Internet of Things project #E2881.

References

[1] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[3] "OnLive." [Online]. Available: http://www.onlive.com

[4] M. Frauenfelder, "G-cluster makes games to go," *The Feature: It's All About the Mobile Internet*, vol. 3, 2001. [Online]. Available: http://www.thefeaturearchives.com/13267.html

[5] W. Cai, C. Zhou, V. C. Leung, and M. Chen, "A cognitive platform for mobile cloud gaming," in *Proceedings of the IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2013, pp. 72–79.

[6] "Modern Combat 5: Blackout." [Online]. Available: https://goo.gl/gsYtsz

[7] "GTA San Andreas." [Online]. Available: https://goo.gl/SzkfbR

[8] "The Walking Dead: Michonne." [Online]. Available: https://goo.gl/XOzgN1

[9] "OpenGL ES example program." [Online]. Available: https://goo.gl/Zrnez4

[10] M. Solutions, "Power monitor," 2016.

[11] L. Ben-Zur, "Developer tool spotlight-using trepn profiler for power-efficient apps," 2011.

[12] "Million Americans play video games and 4 out of 5 households own a gaming device." [Online]. Available: https://goo.gl/0jMAhS

[13] "One in two households with internet access has tv hooked up for streaming." [Online]. Available: https://goo.gl/5btqVW

[14] "Specification of nvidia shield." [Online]. Available: https://goo.gl/vejzCK

[15] "List of nvidia graphics processing units." [Online]. Available: goo.gl/qcJgqJ

[16] A. Munshi, D. Ginsburg, and D. Shreiner, *OpenGL ES 2.0 programming guide*. Pearson Education, 2008.

[17] "Dynamic linker hooking techniques." [Online]. Available: http://man7.org/linux/man-pages/man8/ld.so.8.html

[18] "How do I control TCP delayed ACK and delayed sending?" [Online]. Available: https://access.redhat.com/solutions/407743

[19] Y. Gu and R. L. Grossman, "Udt: Udp-based data transfer for high-speed wide area networks," *Computer Networks*, vol. 51, no. 7, pp. 1777–1799, 2007.

[20] "ARM OpenGL ES emulator." [Online]. Available: http://malideveloper.arm.com/resources/tools/opengl-es-emulator/

[21] "Opengl default framebuffer." [Online]. Available: https://www.opengl.org/wiki/default-framebuffer

[22] D. Halperin, B. Greenstein, A. Sheth, and D. Wetherall, "Demystifying 802.11 n power consumption," in *Proceedings of the 2010 international conference on Power aware computing and systems*, 2010, p. 1.

[23] Y. Collet. (2013) Lz4: Extremely fast compression algorithm. [Online]. Available: http://lz4.github.io/lz4/

[24] L. Aimar, L. Merritt, E. Petit, M. Chen, J. Clay, M. Rullgrd, C. Heine, and A. Izvorski, "x264-a free h264/avc encoder," 2005.

[25] "From tight to turbo and back again: designing a better encoding method for turbovnc." [Online]. Available: http://www.virtualgl.org/pmwiki/uploads/About/tighttoturbo.pdf

[26] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, "Modeling, profiling, and debugging the energy consumption of mobile devices," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 39, 2016.

[27] S. Tang, H. Yomo, Y. Kondo, and S. Obana, "Wake-up receiver for radio-on-demand wireless lans," *EURASIP Journal on Wireless Communications and Networking*, vol. 2012, no. 1, pp. 1–13, 2012.

[28] J. D. Hamilton, *Time series analysis*. Princeton university press Princeton, 1994, vol. 2.

[29] H. Akaike, "Akaike's information criterion," in *International Encyclopedia of Statistical Science*. Springer, 2011, pp. 25–25.

[30] W. Luo and S. Billings, "Adaptive model selection and estimation for nonlinear systems using a sliding data window," *Signal Processing*, vol. 46, no. 2, pp. 179–202, 1995.

[31] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," 2011.

[32] "Star wars^{TM}: Kotor." [Online]. Available: https://goo.gl/25BScA

[33] "Final fantasy." [Online]. Available: https://goo.gl/Ekn6ec

[34] "Candy crush." [Online]. Available: http://candycrushsaga.com/en/

[35] "Cut the rope." [Online]. Available: http://www.cuttherope.net

[36] S. Wang and S. Dey, "Modeling and characterizing user experience in a cloud server based mobile gaming approach," in *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*. IEEE, 2009, pp. 1–7.

[37] "Understanding and optimizing video game frame rates." [Online]. Available: https://www.lifewire.com/optimizing-video-game-frame-rates-811784

[38] "Human being response time benchmark." [Online]. Available: https://goo.gl/9TxS6J

[39] "Ebook reader." [Online]. Available: https://goo.gl/5tVbvX

[40] "Yahoo weather." [Online]. Available: https://goo.gl/N9mLVY

[41] "Tumblr." [Online]. Available: https://goo.gl/rLr5HJ

[42] A. Developers, "Monkeyrunner," 2015.

[43] S.-W. Chen, Y. Chang, P. Tseng, C. Hang, and C. Lei, "Cloud Gaming Latency Analysis: OnLive and StreamMyGame Delay Measurement," in *Proceedings of the 19th ACM International Conference on Multimedia*, 2014, pp. 1269–1272.

[44] F. Luna, *Introduction to 3D game programming with DirectX 10*. Jones & Bartlett Publishers, 2008.

[45] E. Abebe and C. Ryan, "A hybrid granularity graph for improving adaptive application partitioning efficacy in mobile computing environments," in *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications (NCA)*, 2011, pp. 59–66.

[46] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao, "Energy-optimal software partitioning in heterogeneous multi-

processor embedded systems," in *Proceedings of the 45th annual design automation conference*. ACM, 2008, pp. 191–196.
[47] "Gaikai website." [Online]. Available: https://www.gaikai.com/
[48] S. Wang and S. Dey, "Rendering adaptation to address communication and computation constraints in cloud mobile gaming," in *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*. IEEE, 2010, pp. 1–6.
[49] ——, "Addressing response time and video quality in remote server based internet mobile gaming," in *2010 IEEE Wireless Communication and Networking Conference*, April 2010, pp. 1–6.