







Efficient Augmented Reality on Low-Power Embedded Systems

Alessandro Longobardi^(✉) , Franco Tecchia , Marcello Carrozzino ,
and Massimo Bergamasco 

Scuola Superiore Sant'Anna, Mechanical Intelligence Institute, San Giuliano Terme PI,
56127 Pisa, Italy

alessandro.longobardi@santannapisa.it

Abstract. In this paper we propose a development technique for low-power devices with limited computing capacity to obtain efficient, high-performance and non-CPU-invasive Augmented Reality (AR) applications. The paper will discuss how to exploit both the available hardware and software resources. Many boards on the market are equipped with CPUs with low computing power together with GPUs for 2D/3D graphics and multimedia. The paper analyses the strengths of these architectures and how to exploit them. The Operating System (O.S.) also provides features that allow greater control over the system (e.g., avoid wasting resources) and its performance. The techniques proposed are then used, as an example, in the development of an AR application for remote assistance.

Keywords: Augmented reality · Low-power · GPU · Embedded system · Wearable system

1 Introduction

Nowadays, there are a multitude of low-power, small, wearable boards on the market.

These boards are often equipped with single-core processors with limited computing capacity, making it burdensome use the CPU inefficiently or to perform heavy computationally tasks. For this reason, it is necessary to exploit all the resources made available by the hardware: there are hardware chips in the processor's System on Chip (SoC) that can perform specific tasks very efficiently, both in terms of performance and power consumption. Concerning, for example, the various multimedia tasks: video compression, image processing or 2D/3D graphics. Performing these tasks in software (SW) adds a prohibitive load for this type of system.

1.1 Multimedia

Many SoCs have a GPU capable of multimedia functionality:

- Video encoding and decoding: supports many hardware compression standards like h264/h265/mjpeg.

- Image encoding and decoding: support for jpg/png/bmp in hardware.
- Image processing: application of filters and effects.
- Image conversion: image format conversion (e.g., from YUYV to RGB and vice versa).
- Image resize: image downscaling and upscaling.

Exploiting the hardware (HW) features, from the developer's point of view, implies, use libraries/frameworks provided by the chip vendor that do not necessarily work on known standards. This adds constraints to the application design:

1. Availability only in certain languages.
2. Abstraction level.
3. Lack of documentation.

1.2 Bus and Memory

There are chips with special buses (e.g., mipi bus [11]) that allow certain peripherals, like a camera, to be connected directly to the GPU: avoiding unnecessary frame copy from GPU to CPU and vice versa, saving bandwidth.

In addition, many cards feature unified memory, which can be used to advantage by reducing memory bandwidth in graphics applications.

System memory that was previously owned only by the CPU is now shared between the CPU and GPU.

In a unified memory architecture with a unified addressing scheme, both devices share the same virtual address space. Therefore, instead of explicitly transferring data, the CPU can pass the pointer of input data to the GPU.

1.3 2d/3d Graphics

The graphics capabilities of these low-power cards are growing, with full support for vertex/fragment shaders, and higher fillrate.

However, care must be taken when updating textures or reading pixels, considering to take advantage of unified memory or other types of hardware acceleration to reduce memory bandwidth.

1.4 O.S. Feature

The O.S. of many embedded boards are based on the Linux kernel and therefore provide all the means for parallel and concurrent programming. In the design of AR applications, it is essential to exploit multi-threading. Design the software using periodic threads is a key factor for two reasons:

1. Logical: each thread has its own modular, limited and well-defined function.
2. Performance: periodicity is an excellent parameter to regulate the performance and system reactivity.

AR applications can be considered soft-real system, since if the operations are not executed within a deadline performance degrades. Many O.S. provide a series of tools that allow greater control over periodic threads in execution: in the Linux kernel, for example, there are a series of real-time schedulers. The objective of these schedulers is to reduce latencies and have a deterministic response time.

1.5 Applications

At the time of writing we are witnessing the COVID-19 pandemic which has changed the way of working in many sectors: remote assistance (, i.e. the possibility to supervise complex machines/systems remotely) has become increasingly essential.

Virtual reality applications for cultural heritage (e.g. remote museum tours) have proved to be a new way of disseminating culture.

Unfortunately, such applications require expensive and complex devices (e.g., htc vive, oculust rift, etc.).

The techniques described in this paper fit well with the applications described above and target low-cost devices, which are certainly more accessible to everyone.

In addition, with the spread of cloud computing, these techniques are even more fundamental, they concern the edge computing part (i.e. those tasks that must necessarily be performed on the edge device): many AR-cloud applications are based on streaming video to servers that will process the video and returning it with AR info, it is essential to adopt any mechanism that reduces the latency in processing the video on the end device.

2 Related Work

2.1 AR on Embedded GPU

The potential of GPUs in embedded systems has been discussed in recent times; Embedded devices can be used in various areas: AR, computer vision, IoT, etc..

Lopez et al. [3] identified the main HW. and SW. components required by AR systems. The processing architectures investigated are two:

- *Edge Systems*: all operations are performed on the device.
- *Cloud Systems*: operations are delegated to a server.

With the progress of Cloud technologies the second approach is interesting [6]. Unfortunately there are limitations due to latencies and bandwidth required [7] that make the Quality of Experience (QoE) [8] not satisfactory, making these technologies still not completely mature.

Therefore, it makes sense to continue investigating approaches based on edge computing.

Elteir et al. [4] investigated the potential of embedded GPUs in IoT boards as enablers for computationally intensive applications.

Due to the technological differences between the various embedded GPUs, it is difficult to develop high-level platform-independent frameworks. Such frameworks, for

example, must take into account whether the GPU is discrete (dGPU) or integrate (iGPU), as dGPU has its own private memory space for the system and data, introducing overhead for copying data between the CPU and GPU. In contrast, iGPUs share the same memory space with the CPU without the need to copy data.

Interesting is the work of Cameanu et al. [5] in developing a Component-based development (CBD) approach that is independent of the CPU-GPU used. The limitation of their approach is for computationally oriented applications since the framework they developed is based on OpenCL.

2.2 Real-Time Support

Real-time operating systems are widely used in the embedded world. Since AR systems can be considered as soft-real time systems, limitations and benefits of using real-time techniques/approaches are being evaluated. Interesting is the analysis carried out by Elliott and Anderson [1] that highlights the limitations and constraints imposed by current GPUs:

- *Isolation*: most of the drivers are closed-source, then system isolation must be provided to protect from unknown behaviors. Since even soft real-time systems require provable analysis, the uncertain behaviors of the driver force integration solutions to treat it as a black box.
- *Throughput oriented*: The GPU sw. and hw. are designed to be used by only one process at a time. Low latency of operations and the sharing of GPU among processes are only supported to a limited degree.
- *GPU Interrupt handling*: interrupts are difficult to manage in a real-time system. Interrupts may occur periodically, sporadically, or at entirely unpredictable moments, depending upon the application. Interrupts often cause disruptions in a real-time system since the CPU must temporarily halt the execution of the currently scheduled task. The GPU driver must be designed to minimize the interrupt duration.

The authors found that GPU resources in soft real-time systems can be managed through a real-time CPU scheduler.

The Linux kernel offers several real-time schedulers [10] and also a patch to make the kernel real-time [9], which mitigates the limitations written earlier (e.g., interrupts are handled with threads (thus scheduled)).

3 System Model

In order to make the presented methodology clearer, let's evaluate a use case: developing a wearable device with an AR application for remote assistance.

The wearable device is equipped with a camera and head-mounted display (HMD), the device streams the camera (i.e., the point of view of the field operator) to one or more remote people (the expert operator) who provide information/indications in AR rendered on the head mounted display (HMD).

The wearable system is composed of:

- embedded board small in size, low-power and equipped with a SoC combining CPU /GPU sharing memory and Wi-Fi module.
- camera.
- audio module (sound card which can be integrated into the embedded board or external).
- HMD.

The proposed technique starts by identifying which tasks can be performed in HW. and which in SW.

Starting from the application requirements and cross-referencing the datasheet, identifying functions/services that can be implemented taking full advantage of the hardware acceleration. In this case we need to:

1. Acquire frames from the camera.
2. Rendering on HMD the camera frames along with the AR info received.
3. Rendering in a framebuffer the camera frame with some processing/filtering.
4. Encoding/decoding the framebuffer into a video stream.

Point 1 is obtained by exploiting dedicated camera interface: many embedded boards are equipped with a special bus, like the MIPI-CSI-2, that allows the GPU to be connected “directly” to the camera (we will deal with this aspect in the next Section).

Points 2 and 3 obviously require GPUs capable of handling graphical contexts. The HMD is used to show the camera frames along with the AR information coming from the expert operator, this type of information could be simple indications/text or complex 3D figures. In any case reduce the memory bandwidth between CPU and GPU is primary: if the camera frame is used as a texture, it must be saved in the same memory portion that the GPU will looking for.

Regarding point 4, targeting applications that require encoding/decoding of video streams: it is worth checking whether there is a dedicated HW. in the GPU for its processing. Many commercial cards are equipped with HW. acceleration for h264/h265 video encoding. In order to program them, low-level standards (e.g., OpenMax) up to higher-level frameworks (e.g., libav/openh264) can be used, allowing to create/manage/integrate video codecs inside applications with different abstraction levels.

In the proposed model the best solution is always to use low level standards, like OpenMax: they guarantee full control over configurable parameters, less overhead and moreover there is no risk of using SW. solutions without the developer “*awareness*”: this occurs very frequently in high level frameworks when an implementation is not available for a specific HW., there is a fallback to SW. approaches which in embedded systems risk saturating the CPU.

4 Proposed Approach

4.1 Graphics Standard

The structure of the various standards for programming the GPU is often not clear: it is important to know the relationship between them and their interoperability.

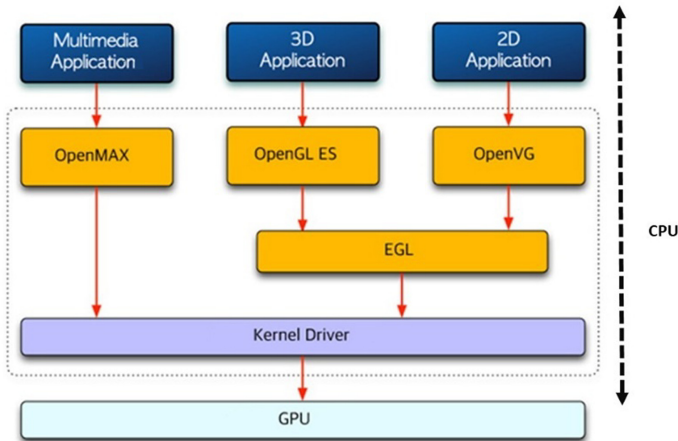


Fig. 1. GPU standards structure

The standards considered are:

- **OpenMax:** OpenMax is a unified abstraction layer that allows access to hardware that otherwise requires vendor specific APIs. OpenMax therefore allows a (sort of) portable implementation of software that utilizes such hardware. OpenMax provides an abstraction over hardware that is capable to perform operations with multi media (audio, images and video). Hardware can only be in one state at a time so it resembles a state machine. OpenMax maps to that state machine and provides an API to manipulate it.
- **OpenGL ES:** OpenGL ES is a cross-platform API for full-function 2D and 3D graphics on embedded systems, including consoles, phones, appliances, and vehicles. It consists of well-defined subsets of desktop OpenGL, creating a flexible and powerful low-level interface between software and graphic acceleration. OpenGL ES includes profiles for floating-point and fixed-point systems and the EGL specification for portably binding to native windowing systems. What distinguishes OpenGL ES from OpenGL is its emphasis on low-capability embedded systems. OpenGL ES is a simplified and tidied-up form of OpenGL but still has the same capabilities as OpenGL. The emphasis with this API is on 2D and 3D graphics, suitable for games programming and other high-demand graphics applications. It will work in cooperation with other windowing systems, such as the X Window System or Wayland, or where no other windowing system is running.

- **OpenVG:** OpenVG is a cross-platform API that provides a low-level hardware acceleration interface for vector graphics libraries such as Flash and SVG. OpenVG is targeted primarily at handheld devices that require portable acceleration of high-quality vector graphics for compelling user interfaces and text on small-screen devices, while enabling hardware acceleration to provide fluidly interactive performance at very low power levels. OpenVG is an alternative approach to graphics to OpenGL. While OpenGL renders onto textures, OpenVG is more concerned with drawing lines to form shapes and then rendering within those shapes.
- **EGL:** EGL is an interface between Khronos-rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system. It handles graphics context management, surface/buffer binding, and rendering synchronization and enables high-performance, accelerated, mixed-mode 2D and 3D rendering using other Khronos APIs. EGL is the “glue” layer between the higher-level APIs and the hardware. It isn’t used extensively by the application programmer, just enough to give the higher level the hooks into the lower level. Both OpenGL ES and OpenVG sit above EGL.

Many manufacturers develop specific features for their hardware, which are accessible via so-called standards extensions:

- **OpenGL extension:** the OpenGL standard allows individual vendors to provide additional functionality through extensions as new technology is created. Extensions may introduce new functions and new constants, may relax or remove restrictions on existing OpenGL functions. Proprietary pixel format can be defined: how pixels are mapped in texture memory, many chip vendors create their own internal conventions, opaque to the programmer, that allow proprietary technologies to be used to the advantage of performance and power consumption.
- **OpenMax tunneling:** it is possible to exchange information between multimedia components (e.g., by directly passing camera frames to a video encoder) without involving the CPU.
- **EGL extension:** it is possible, to use a camera frame or decoded video frame directly as a texture. The solutions adopted by chip vendors use proprietary technologies (e.g., proprietary pixel format or special memory mapping exploiting unified memory architecture) that are available to the programmer as extensions to EGL. One of the most useful is the *EGL image*: this extension defines a new EGL resource type that is suitable for sharing 2D arrays of image data between client APIs., Although the intended purpose is sharing 2D image data, the underlying interface makes no assumptions about the format or purpose of the resource being shared, leaving those decisions to the application and associated client APIs.

4.2 Multithreading

Many embedded O.S. have support for multithreading, which is a powerful tool for structuring application tasks in a modular and logical way. In graphic/multimedia applications, the use of periodic threads is a versatile design choice: performance can be adjusted to dynamically scale according to the quality required or the CPU load.

The structure of periodic thread is similar the following:

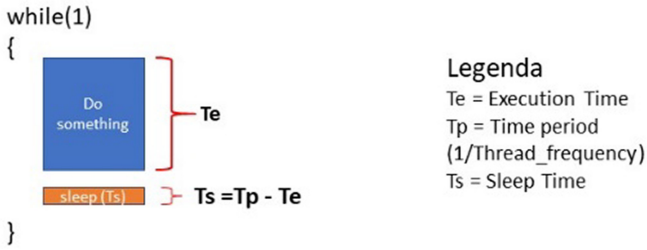


Fig. 2. Periodic thread structure

Threads along with the possibility of interconnecting them using the technologies/standards described in the previous paragraphs can be modeled as a series of high-level blocks that provide versatility in building applications on embedded devices (Fig. 3).

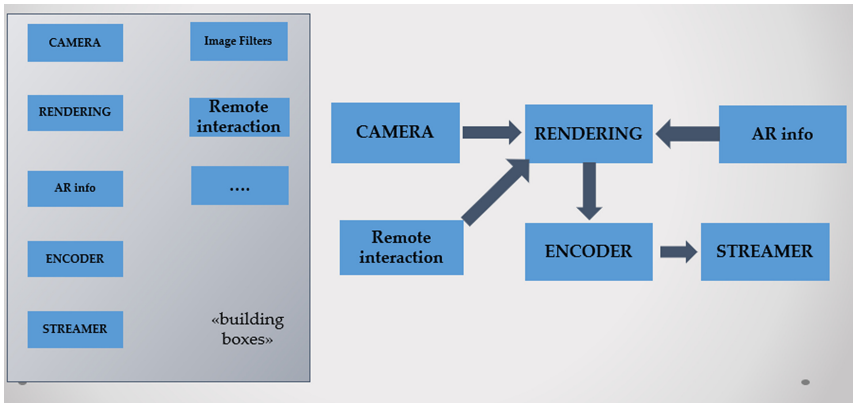


Fig. 3. The tasks implemented as threads can be modeled as high-level box.

4.3 Scheduling

Since AR systems can be considered soft real-time system (i.e., if the deadline is missed there is a performance degradation), it makes sense to investigate the real time schedulers made available by a multitude of O.S.

These allow greater control over the threads that are running.

Real-time systems require that tasks execution must follow a severe priority order. This requires that only the K highest-priority tasks be running at any given instant in time, where K is the number of CPUs. A variation to this requirement could be strict priority-ordered scheduling in a given subset of CPUs or scheduling domains. In both cases, when

a task is runnable, the scheduler must ensure that it be put on a run-queue on which it can be run immediately—that is, the real-time scheduler has to ensure system-wide strict real-time priority scheduling. Unlike non-real-time systems where the scheduler needs to look only at its run-queue of tasks to make scheduling decisions, a real-time scheduler makes global scheduling decisions, considering all the tasks in the system at any given point. A real-time task scheduler would trade off throughput in favor of correctness, but at the same time, it must ensure minimal task ping-ponging.

Linux kernel offers the following schedulers type:

- **SCHED_OTHER**: each process is assigned a maximum time slice and a dynamic priority given by the sum of a base value and a value that decreases as the CPU time used increases. Dynamic priority update: when all the time of the ready processes is exhausted, the priorities of all processes are recalculated. It aims to ensure a fair distribution of CPU between all processes, and to provide good response times for interactive processes.
- **REAL_TIME SCHEDULERS**:
 - **SCHED_FIFO**: it implements a first-in, first-out scheduling algorithm. When a **SCHED_FIFO** task starts running, it continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task. All other tasks of lower priority will not be scheduled until it leaves the CPU. Two equal-priority **SCHED_FIFO** tasks do not preempt each other.
 - **SCHED_RR**: is similar to **SCHED_FIFO**, except that such tasks are allocated time slices based on their priority and run until they finish their time slice. The default linux time slice is 100 ms.

Real-time schedulers allow to prioritize threads, thus having greater determinism.

Not all tasks are equal: there are some that are more important than others, so it is fair to give them higher priority. In a typical AR application, it may be preferable to prioritize the camera frame capture thread over the video-rendering thread. This can be useful in CPU overloading situations: sometimes a real-time application running on the target computer does not have enough time to complete processing before the next time step, an overload happens every time an execution step is triggered while the previous one is running. On low-power single-core device, this phenomenon can occur for a variety of reasons, whether related to the design of the applications themselves (optimistic system design, based on average rather than worst-case behavior) or to the kernel (interrupt bursts, kernel exceptions, etc.). Faced with this situation, establishing an order of what to maintain with stable performance and what to degrade is essential.

5 Experimental Results

The test setup of the techniques shown is a wearable system for remote assistance.

The application use case is shown in Fig. 4.

The wearable device is equipped with a camera and viewers, this device streams what the operator in the field (called the field operator) sees to one or more remote

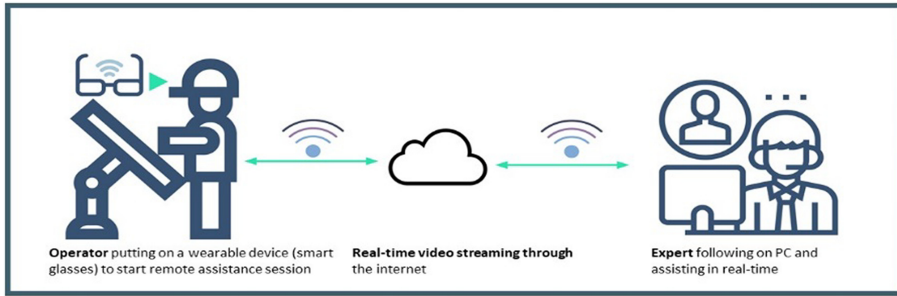


Fig. 4. Remote assistance application example

persons (the expert operator) who provide information/indications in AR rendered on the viewers together with the video of the camera itself.

5.1 Hw. Architecture

The system is composed by:

- Embedded board: custom low power board is used, it features a Broadcom BCM2835 system on a chip, which includes a 700 MHz ARM1176JZF-S processor, VideoCore IV Graphics Processing Unit (GPU), and 1Gb of shared memory. It has 16 KB Level 1 cache and 128 KB Level 2 cache, that is used primarily by the GPU.
- Camera: the camera chosen uses 5MP Omnivision 5647 sensor, capable of 1080@30 fps, 720@60fps, 480p@90fps. The communication interface is based on a 15-pin MIPI Camera Serial Interface that directly communicates to the GPU.
- HMD: 1080p@60fps HMD is used.
- Audio: audio card featuring microphone and headphones
- Connectivity: 2.4 Ghz interface
- O.S.: Linux 4.9
- Battery

5.2 Sw. Architecture

The application has the following pipeline:

Regarding the application design, the following scheme is used:

- Camera and Video Splitter (OpenMax domain): connected through the tunnelling mechanism. The video splitter takes the camera signal as input and forwards it to two output ports as *egl image*.
- Video rendering (OpenGL domain): the camera frames along with AR info are rendered, the *egl image* is used directly as texture.
- Framebuffer rendering (OpenGL domain): this rendering stage is used to prepare the frame to transmit, effects and filters are applied to the frame (e.g., increase contrast, use edge detector shaders, etc.). Again, the *egl image* is used directly as a texture.

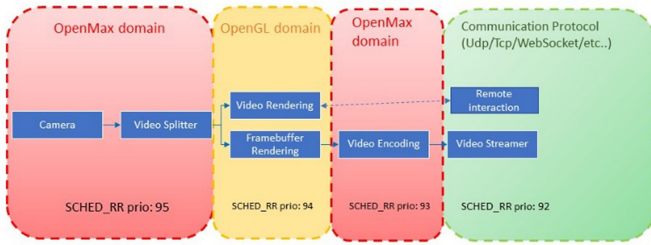


Fig. 5. Software pipeline

- Video encoding (OpenMax domain): the encoder takes the entire framebuffer object (as an *egl image*) as input and outputs an encoded video stream (e.g., h264/h265).
- Video streamer and the remote interaction: rely on classic communication protocols (e.g. udp/tcp/websocket, etc...).

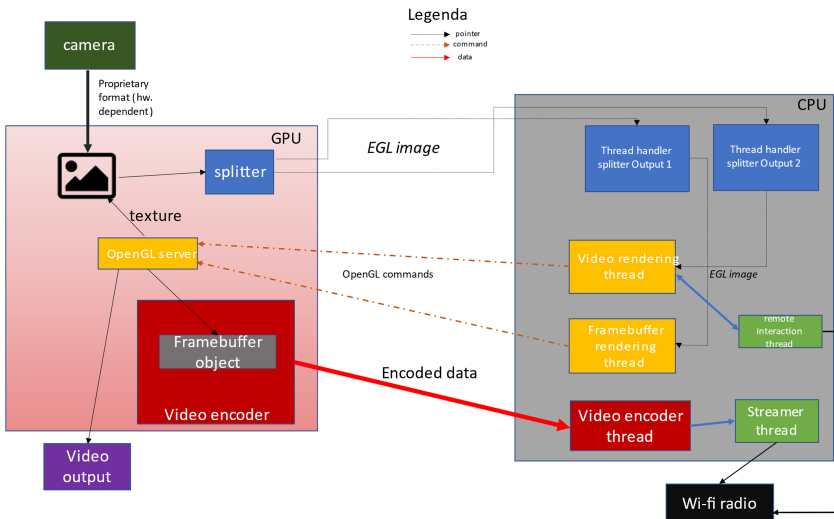


Fig. 6. CPU-GPU interaction

From an operational point of view, the CPU and GPU exchange the following information (Fig. 6):

4. GPU-> CPU: The *egl image* containing the camera frame. From developer’s point of view is a simple pointer indicating the GPU memory area containing the camera frame.
5. CPU-> GPU: OpenGL commands are sent to the server that resides on the GPU.
6. GPU-> CPU: what is rendered in the framebuffer is encoded as a streaming video. The video encoder generates video packets transferred to the CPU. This is the only

operation that requires a minimum memory bandwidth in the application, the video packets size is proportional to the encoding level, resolution and framerate.

5.3 Benchmark

A comparison is made using both real-time and classic priorities, simulating overloading situations (i.e., load average above 1 by introducing threads which performs some computation).

Real-time priorities have been given to prevent blocking of camera frame acquisition and any form of deadlock in the block diagram pipeline (Fig, 5): output splitter threads that handle *egl image* must have the highest priority, we must assure their execution as soon as possible to render the newest frame.

The real-time scheduler used is SCHED_RR (with default time slice 100 ms), priority ranges from 1(low) to 99(high) is given according to the following scheme:

1. Splitter output threads: priority 95.
2. Rendering threads: priority 94.
3. Video encoder thread: priority 93.
4. Remote interaction & Video streamer: priority 92.

The threads period is given by the working frequency of the associated component (e.g., if the video rendering is at 60 fps, then the video rendering thread on CPU is running at 60 Hz).

The tests are performed setting high resolution and fps value, pushing the HW. to its limit.

The experiment number 1 (Table 1) has the following parameters:

- Camera: 1280x960 with target 30 fps.
- Video-Rendering: 1280x960 with target 60 fps.
- Video-Encoding: 1280x960 with target 30 fps.

The measured/real fps are lower because are near the GPU and camera limits.

Using SCHED_RR there is an increment on measured average fps, confirming the goodness of the proposed approach. The major benefit is notable under overloading conditions, since given the strict priority order the scheduling algorithm tries to reach the target fps.

Same considerations are valid for experiment 2 and 3.

Figure 7, 8 and 9 show for each component and experiment the measured FPS. SCHED_RR in overloading conditions has a huge improvement compared to overloading in SCHED_OTHER.

Table 1. Experiment 1: camera 1280x960@30fps, video rendering 1280x960@60fps, encoding 1280x960@30fps

Simulate overloading	Camera resolution	Camera fps (target/measured)	Rendering resolution	Rendering fps	Encoding resolution	Encoding fps	Scheduling
no	1280 × 960	30/23	1280 × 960	60/20	1280 × 960	30/13	SCHED_OTHER
no	1280 × 960	30/25	1280 × 960	60/25	1280 × 960	30/13	SCHED_RR
yes	1280 × 960	30/15	1280 × 960	60/12	1280 × 960	30/10	SCHED_OTHER
yes	1280 × 960	30/21	1280 × 960	60/21	1280 × 960	30/10	SCHED_RR

Table 2. Experiment 2: camera 640x480@60fps, video rendering 1280x960@60fps, encoding 640x480@30fps

Simulate overloading	Camera resolution	Camera fps (target/measured)	Rendering resolution	Rendering fps	Encoding resolution	Encoding fps	Scheduling
No	640 × 480	60/47	1280 × 960	60/41	640 × 480	30/22	SCHED_OTHER
No	640 × 480	60/48	1280 × 960	60/42	640 × 480	30/22	SCHED_RR
yes	640 × 480	60/21	1280 × 960	60/15	640 × 480	30/10	SCHED_OTHER
yes	640 × 480	60/45	1280 × 960	60/40	640 × 480	30/21	SCHED_RR

Table 3. Experiment 3: camera 1920x1080@30fps, video rendering 1920x1080@60fps, encoding 1280x720@30fps

Simulate overloading	Camera resolution	Camera fps (target/measured)	Rendering resolution	Rendering fps	Encoding resolution	Encoding fps	Scheduling
No	1920 × 1080	30/15	1920 × 1080	60/15	1280 × 720	30/9	SCHED_OTHER
No	1920 × 1080	30/15	1920 × 1080	60/15	1280 × 720	30/12	SCHED_RR
yes	1920 × 1080	30/13	1920 × 1080	60/10	1280 × 720	30/8	SCHED_OTHER
yes	1920 × 1080	30/15	× 1080	60/15	1280 × 720	30/10	SCHED_RR

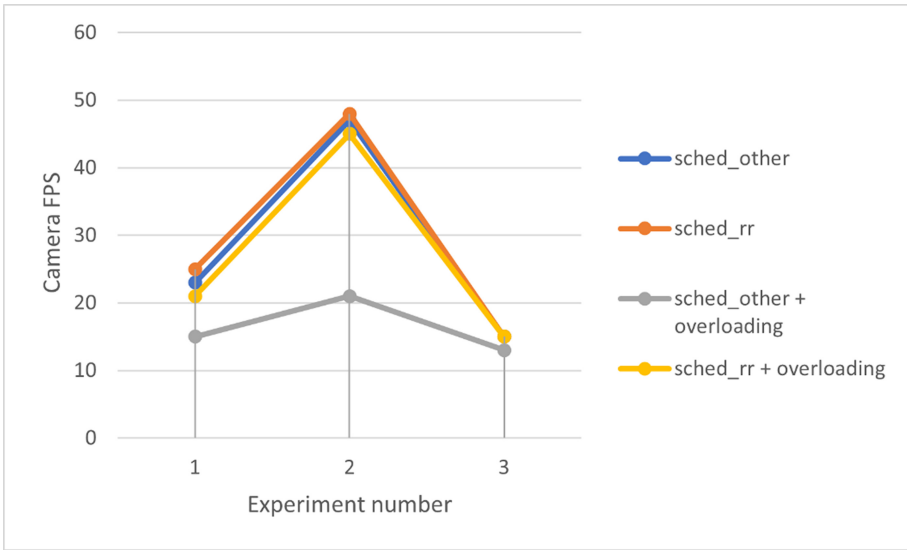


Fig. 7. Camera FPS variation considering experiment number and scheduling policy.

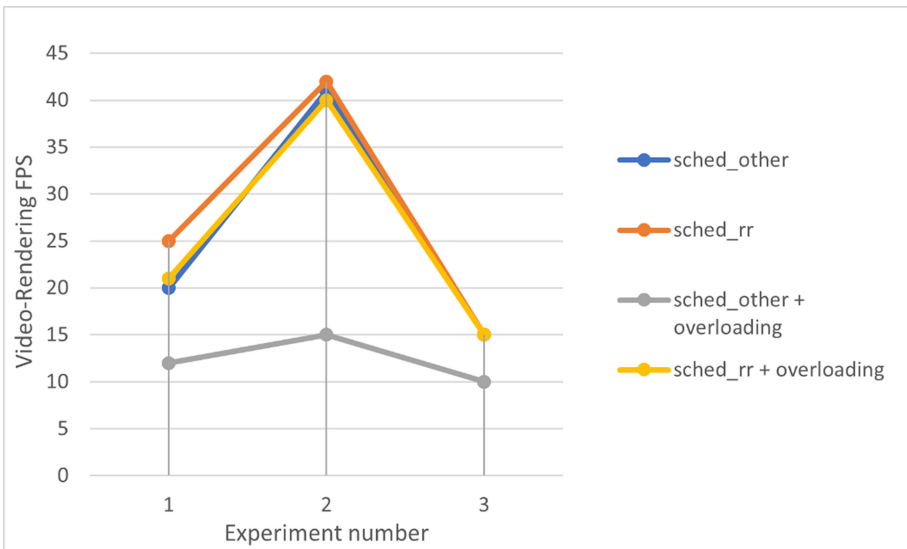


Fig. 8. Video-rendering FPS considering experiment number and scheduling policy.

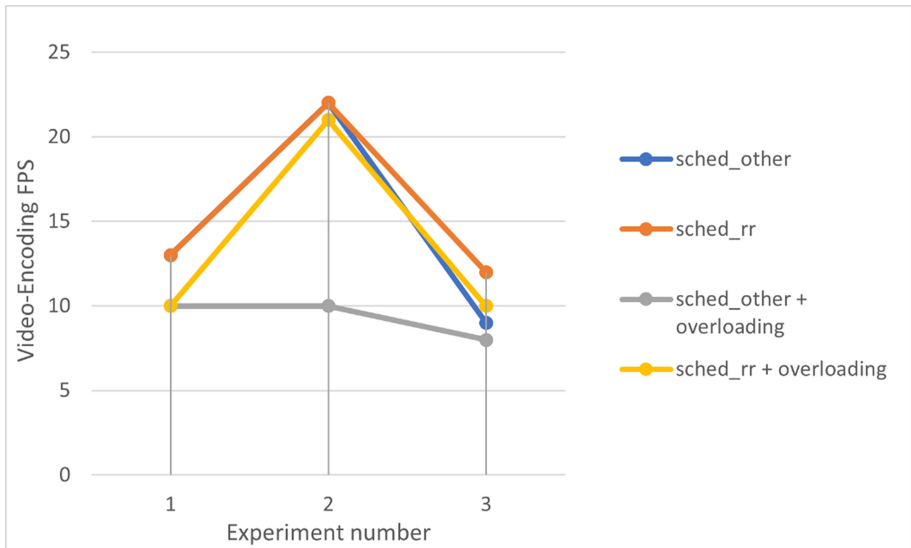


Fig. 9. Video-encoding FPS considering experiment number and scheduling policy.

6 Conclusions

The techniques shown provide generic guidelines on how to program embedded boards for augmented reality applications. It is shown how to take advantage of the unified memory and hardware acceleration available.

Structuring the application using periodic threads is a key-point, it allows to associate to each thread a specific and limited task, the periodicity is a good switch to adjust the performance.

Finally, it has been shown how the use of real-time scheduling policies allows a greater determinism of the application behavior, providing greater control in overloading situations.

As a use case an AR telemetry application for a low-power wearable system has been developed, validating the described approaches.

In the future it is interesting to evaluate the described approaches, especially related to the part of scheduling, on a full real-time O.S. as Linux PREEMPT-RT, which guarantee:

- Interrupts are handled with threads (thus scheduled).
- Spin locks replaced with mutexes.
- Priority inheritance is extended to the kernel.

References

1. Elliott, G.A., Anderson, J.H.: Real-world constraints of GPUs in real-time Systems. In: 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 48–54 (2011). <https://doi.org/10.1109/RTCSA.2011.46>

2. Schneider, M., Rambach, J., Stricker, D.: Augmented reality based on edge computing using the example of remote live support. In: 2017 IEEE International Conference on Industrial Technology (ICIT), pp. 1277–1282 (2017). <https://doi.org/10.1109/ICIT.2017.7915547>
3. López, H., Navarro, A., Relaño, J.: An analysis of augmented reality systems. In: 2010 Fifth International Multi-conference on Computing in the Global Information Technology, pp. 245–250 (2010). <https://doi.org/10.1109/ICCGI.2010.24>.
4. Elteir, M.K., Lazem, S., Azab, M.: Unleashing the hidden powers of low-cost IoT boards: GPU-based edutainment case study. *J. King Saud Univ. Comput. Inf. Sci.* 1319–1578 (2020)
5. Campeanu, G., Carlson, J., Sentilles, S.: Developing CPU-GPU embedded systems using platform-agnostic components. In: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 176–180 (2017). <https://doi.org/10.1109/SEAA.2017.20>.
6. Hou, X., Lu, Y., Dey, S.: Wireless VR/AR with Edge/Cloud Computing. In: 2017 26th International Conference on Computer Communication and Networks (ICCCN), pp. 1–8 (2017). <https://doi.org/10.1109/ICCCN.2017.8038375>
7. Maheshwari, S., Raychaudhuri, D., Seskar, I., Bronzino, F.: Scalability and performance evaluation of edge cloud systems for latency constrained applications. *IEEE/ACM Symp. Edge Comput. (SEC)* **2018**, 286–299 (2018). <https://doi.org/10.1109/SEC.2018.00028>
8. Zhang, W., Han, B., Hui, P.: On the networking challenges of mobile augmented reality. In: Proceedings of the Workshop on Virtual Reality and Augmented Reality Network (VR/AR Network 2017), pp. 24–29. Association for Computing Machinery, New York (2017). <https://doi.org/10.1145/3097895.3097900>
9. Reghenzani, F., Massari, G., Fornaciari, W.: The real-time linux kernel: a survey on PREEMPT_RT. *ACM Comput. Surv.* **52**(1), 1–36 (2019). <https://doi.org/10.1145/3297714>
10. Scordino, C., Lipari, G.: Linux and real-time: current approaches and future opportunities (2006)
11. <https://www.mipi.org/specifications/csi-2>