Low-latency Mixed Reality Headset

Tan Nguyen
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{tan.nqd}@berkeley.edu

***Abstract*—This work aims to build an open-source, low-latency hardware-accelerated headset for mixed (virtual or augmented) reality applications. A mixed reality headset is a complex system that encompasses many components, such as displays, IMU sensors, cameras, and processing units. Our motivation is that the commercial headsets are still not fast enough to prevent motion sickness caused by the lag between the display content and the movement of the headset user: the so-called motion-to-photon latency. Technique such as post-render warping (or timewarp) can be utilized to warp the images right before the display based on the tracking information, and is already implemented in many commercial products such as Oculus or Hololens. Nonetheless, the latency is still insufficient to combat motion sickness. In this project, we delve into understanding and reducing the motion-to-photon latency by leveraging specialized hardware platforms as well as computer vision algorithms to build a headset from scratch. Our headset is capable of running simple mixed reality demo applications (cubemap rendering for VR, 3D static object overlay for AR) with a motion-of-photon latency of 13.4 ms. More importantly, it is fully open-source.**

I. Introduction

The global market for Head Mount Display (HMD) of virtual and augmented reality technology is forecasted to grow to hundreds of billion dollars by 2025 [1]. Virtual reality renders a surrounding that immerses users in a virtual world, whereas augmented reality overlays artificial objects on top of the real world. The technology is increasingly being found in many applications ranging from entertainment, education, to healthcare, and manufacturing industry in the form of HMD/headset, which notable products include Facebook Oculus, Microsoft Hololens, or HTC Vive. A VR/AR headset is a complex computing embedded system involving tracking sensors, depth cameras, graphics processor, application processor, displays, or even a deep neural network accelerator. It is of no trivial engineering feat to fit all those blocks into a small form factor as normal ergonomic glasses while functioning as an energy-efficient computing platform.

However, to truly deliver an immersive experience to users, the motion-to-photon latency of a headset is the utmost important factor. The motion-to-photon latency is defined as the time between the motion captured by the headset sensor (IMU, or Inertial Measurement Unit), to the time when the corresponding image gets displayed on the headset lenses (photon). As an example, when a user turns his head to the left, the displayed content should move to the right, thus the motion-to-photon in this case can be understood as the length of time when the user starts turning his head until the displayed starts changing with respect to this movement.

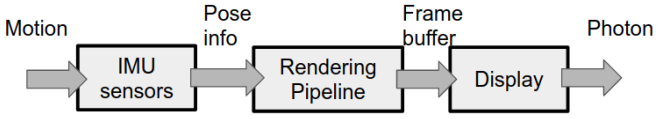

Figure 1: A typical VR/AR Pipeline

Previous work demonstrated that a slow motion-to-photon latency could lead to the mis-registration between artificial and real worlds, and hence causes various effects on users such as nausea or motion sickness. [2] suggests that a latency of 20 ms or less for VR, and 5 ms or less for AR is sufficient to have a decent experience without motion sickness. Nonetheless, it is challenging to meet such lofty latency targets, given that a VR/AR pipeline typically consists of many stages, as shown in Figure 1. First, the IMU sensors sample the motion data to give user's pose information (3D translation and orientation information). The information is passed to the Rendering engine to create a framebuffer of the scene based on the current pose. The framebuffer is then sent to the Display to emit the rendered image onto the headset's lenses. By the time the user observes the image, it is already tens or more miliseconds late in comparison to the current user's pose, depending on the latency of the rendering engine and the display technology. The time lag may be noticeable and create undesirable visual effects. To improve the latency, a classical approach typically utilized in many commercial headsets (or even gaming engines) is time-warp, or post-render warping. That is, the framebuffer is warped based on the latest pose information right before display, hence it completely bypasses the renderer [3]. The technique effectively decouples the display stage from the rendering stage, thereby they can operate at different frame-rates. One caveat is that the technique can only be applied to 2D transformations, as the framebuffer does not contain the depth information. Commercial headsets already implemented this technique in one form or another. Nevertheless, on personal accounts, the latency is still visible to users.

In addition to post-render warping, it is a general consensus that high frame-rate, low-latency display technology is also vital to cut down the latency as much as possible. Typical headsets utilize either OLED or LCD displays. Those displays do not go as far as 120Hz. Another excellent display candidate for VR/AR is Digital Light Projector (DLP) from Texas Instrument. The projector can run at 240Hz or higher, hence

greatly improves the user experience.

In this work, we build an open-source mixed reality headset (supporting both virtual and augmented reality applications) shooting for low latency by using the post-render warping technique and the 240Hz-projectors (DLP). We use the Xilinx UltraScale+ ZCU102 System-on-Chip platform to prototype the hardware architecture and system of our headset. In particulars, the post-render warping accelerators are implemented on the FPGA of the ZCU102. The FPGA also provides great I/O capability for driving the projectors with very high frame-rate via HDMI output. The ARM processor of the ZCU102 runs our primitive applications as well as interfaces with the IMU in a Linux-based environment. Our headset frame is based on the open-source project NorthStar [4]. However, we replaced their LCDs with the DLPs. To the best of our knowledge, this is the first academic work to build an open-source headset based on DLP technology at 240Hz. More importantly, we would like to emphasize the importance of having an open-source headset. We do not rely on any proprietary software stack as in many commercial headsets. On the contrary, Microsoft Hololens 2.0 can only run their own Microsoft RTOS and workloads [5]. By open-sourcing the architecture of the headset and hopefully setting a standard, we envision that this could be extremely useful for VR/AR researchers, developers, or even hardware architects as an established and stable testing platform for future novel applications.

In summary, our contributions are as follows.

1. We show how to build a hardware-accelerated, low-latency, inside-out tracking mixed reality headset by utilizing IMU sensor, FPGA, and DLP projectors running at 240Hz.

2. Our prototype headset is highly agile, simple, and mobile. The visor can move around with ease.

3. Our headset is fully open-source.

II. Related work

Perhaps as closest as in terms of motivation to our work is [6], [7]. They built an optical-see-through (OST) AR system to examine motion-to-photon latency. However, their setup involves a lot of complex mechanical parts and it is uncertain how to move around in that setting. Their system supports only two motions such as pitch and yaw using a shaft encoder. In contrast, our headset can go as far as 6 DoFs (Degrees of Freedom), including 3D translation and 3D orientation (roll, pitch, and yaw) by leveraging consumer-grade IMU sensor. Interestingly, they use DLP projector for their display, similar to our approach. However, they generate binary frames instead and implement a modulation scheme to vary the brightness of the output image. Thus, the latency of their display setup is very low, but it only supports gray-scale format. On the other hand, our headset supports RGB color format. Notably, we achieve stereo vision, while their system only works for single eye. In overall, our headset is closer to being a realistic VR/AR HMD in terms of form factor and the image quality it can deliver.

NVIDIA also built their prototype glasses using laser scan projectors for peripheral display [8]. However, it is unclear what the motion-to-photon latency of their glasses is, and their focus is not on optimizing latency, but rather on producing foveated image using gaze tracking. Therefore, their work is orthogonal to ours.

Recently, Leap Motion released an open-source AR project called NorthStar [4]. It offers a prototype platform including the mechanical components of the headset frame, the two LCDs, and a hand-tracking module. The original NorthStar does not have any built-in IMU or any processing unit. Our headset (or at least the mechanical framework) is based heavily on the NorthStar. However, we replaced the two existing LCDs (1440x1600 at 120Hz) with the DLPs to go with higher frame-rate.

Post-render warping is by no mean novel and is believed to be available in many commercial headsets [2], [9], [10]. Post-render warping can be implemented in commodity hardware such as CPU, GPU, or DSP. For example, by leveraging GPU preemption, the rendering is interrupted to start the post-render warping so that the generated framebuffer is up-to-date with the current tracking information from IMU sensors. In some headsets, a dedicated hardware block (ASIC) is used to accelerate post-render warping task. For instances, the Holographic Processing Unit inside Microsoft Hololens 2.0 has a specialized (and large, by looking at the photo of the chip floorplan) module named Late Stage Reprojection [5]. There is not much public information regarding the functionality of this block, however, one could as well understand that it plays a similar role to post-render warping, but perhaps with some additional support for 3D transformation. Regardless of the implementation, it is desirable that the post-render warping module is as close to the display as possible to minimize the latency. In our work, we design a post-render warping accelerator in the FPGA, since we also use the FPGA to drive the HDMI output to the DLPs. We do not buffer the warped image back to the main memory or the embedded memories on the FPGA as it would incur additional latency, but rather stream the pixels of the warped image directly to the HDMI output in raster order.

III. Background

A. IMU

IMU (Inertial Measurement Unit) is an electronic device that gives a body's tracking information in 3D world through a combination of accelerometer, gyroscope, and sometimes magnetometer. The tracking information is generally 6 DoFs, i.e. translation in (x, y, z) coordinates as well as orientation in roll, pitch, and yaw axes as shown in Figure 2. Thanks to these features, IMU sensors are typically found in smart phones, airplanes, ships, cars, or HMD, etc. as a means for navigation and tracking. In a VR/AR pipeline, IMU is hardly the bottleneck, since some IMU sensors has a very high sampling rate (1000Hz or higher). In this work, we incorporate the Intel Realsense T265 to our headset. It is a tracking camera leveraging Bosch BMI055 IMU sensor to sample 6DoF pose data at around 262Hz [11]. Through a combination with the fish-eye sensors, T265 is able to perform visual odometry to

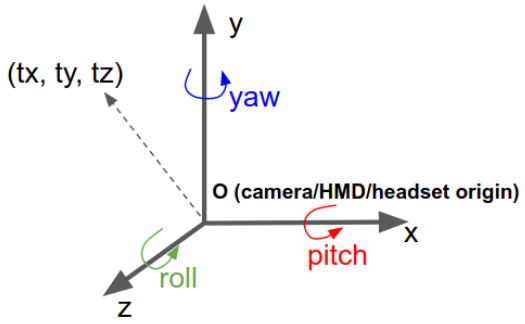

Figure 2: Translation and Orientation in camera's coordinates

mitigate drifting effect which is a common drawback of most IMUs.

B. FPGA

FPGA (Field Programmable Gate Array) is an electronic platform enabling rapid prototyping of hardware circuits by assembling heterogeneous primitive blocks such as Lookup Tables (LUTs), Flip Flops/Registers (FFs), Embedded block RAMs (BRAMs), and Digital Signal Processing blocks (DSPs). FPGA is reconfigurable: the current implemented circuits can be updated by downloading a new bitstream to FPGA. Therefore, it presents a great flexibility to hardware designers who may want to experiment with different circuit implementations. FPGA stands between general-purpose processors and ASICs in terms of performance and energy efficiency. For a project that is meant to build agile prototype to which could be quickly adapted new ideas, FPGA is a suitable platform. In our work, we leverage Xilinx UltraScale+ ZCU102 System-on-Chip [12]. This platform has a quad-core ARM processor coupling with the FPGA fabric via AXI buses. We use the ARM processor as a control processor, and implement computer vision accelerators as well as video processing logic on the FPGA. The platform also offers rich I/O capability to interface with peripheral devices like projectors. Since we need two HDMI output to drive two projectors, we use the two available FMC slots to plug the extension HDMI cards from AVNet [13] instead of the single HDMI output onboard.

C. DLP

DLP (Digital Light Projector) is a display technology from Texas Instrument that is typically found in digital cinemas. DLP projects images using DMD (Digital Micromirror Device) chips. The chips can be toggled to produce gray-scale images. DLP enables color images by using a color wheel. Therefore, the displayed image is not interleaved RGB frames (line-sequential), but rather color-sequential sub-frames (R sub-frame, G sub-frame, followed by B sub-frame). In this work, we use TI DLP Evaluation Kit 2010 as a projector for VR/AR display [14]. We flashed a new firmware (provided by TI) to the boards to upgrade the frame-rate to 240Hz.

D. Post-Render Warping

Post-Render Warping is a technique that warps the image (or framebuffer) using homography transformation. It transforms an image captured at a specific camera pose to an image at a new camera pose. The new camera pose is identified by a homography matrix which defines how much translation and rotation of the camera in 3D world coordinates. The transformation is essentially a vector-matrix multiplication operation of the homography matrix and a particular pixel index; it maps the pixel index of source image $I(src_x, src_y)$ to a corresponding pixel index of the destination image $I(dst_x, dst_y)$. The homography matrix is obtained from the IMU tracking information. The T265 camera returns orientation/rotational data in the form of quaternion qw, qx, qy, qz which we convert to rotational matrix. The translation tx, ty, tz is optionally incorporated if the 3D information of the scene is available (we will discuss more in the demo application section).

$$a = qw; b = qx; c = qy; d = qz;$$

$$H = \begin{pmatrix} a^2+b^2-c^2-d^2 & 2bc-2ad & 2bd+2ac & tx \\ 2bc+2ad & a^2+c^2-b^2-d^2 & 2cd-2ab & ty \\ 2bd-2ac & 2cd+2ab & a^2+d^2-b^2-c^2 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$I(dst_x, dst_y, dst_z) = H \times I(src_x, src_y, src_z)$$

IV. Implementation

A. Headset Architecture

Figure 3 shows the architecture of our prototype headset. The head frame is 3D-printed and based on the NorthStar headset. The reflectors (or lenses) are also from NorthStar. We replace the rear LCD panels with the rear projection films [15]. On each side, the film reflects the light emitted from the projector to the reflector, then to user's eye. They can also enhance the brightness of the displayed image. We place the two DLPs on the sides on the far back and ensure that the placement of the DLPs gives a good throw ratio (i.e., the projected image fits well onto the film). We mount two 80/20 frames as handhelds for the headset. To make room for the user's head to wear the headset comfortably as well as not block the light from the DLPs, the handhelds are slanted apart instead of parallel. The T265 camera is mounted on top of the head mechanical frame without any occlusion so that it gives the best visual odometry performance, and also ensures that it moves in accordance to the headset movement. It connects to the ZCU102 by the USB3.0 interface. The two DLPs are hooked up to the ZCU102 via HDMI cables. Given the clunkiness of the ZCU102, it remains mostly stationary in our experiments, but within a reasonable distance to the visor. With this lightweight and agile architecture, the visor is able to grip the handhelds and move around with ease. In addition, this "open" architecture does not specifically tie to any hardware

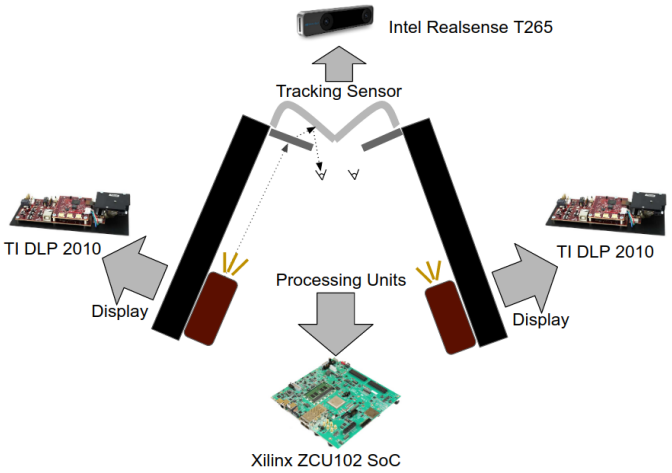

Figure 3: Headset Architecture

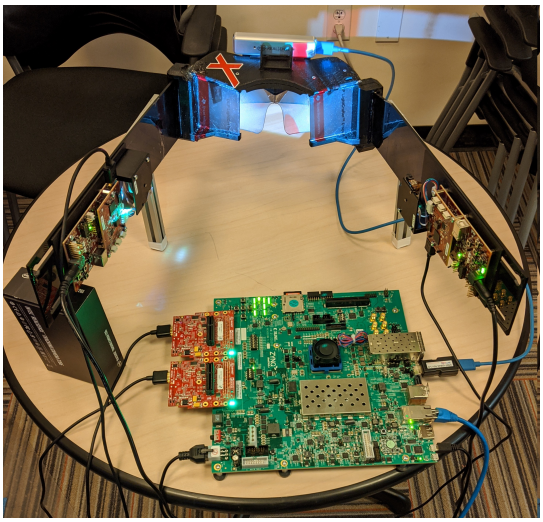

Figure 4: Headset: View from the top

platform. One could replace the T265 with a better IMU, or change the projectors. In future work, we could leverage more novel hardware platforms with small form factor to make a more portable headset.

Figure 4 gives a complete look of our headset.

B. Headset System

Figure 6 demonstrates an overview of our headset system from the perspective of the ZCU102 SoC, as it is the main processing unit of our headset. The ARM processor of the ZCU102 can boot a full-fledged Linux (Ubuntu) operating system. The Intel Realsense T265 camera is connected to the ZCU102 by USB3.0 interface. Intel also provides ARM-compatible Realsense device driver so that our applications are able to query pose information from the T265 camera using their API calls. Though the camera is said to have a sampling rate of 262Hz (3.82 ms), we found that the actual time it takes to read a pose data from the device is from 4.8 ms to

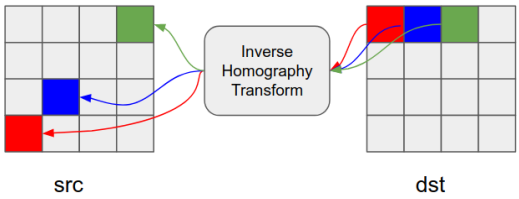

Figure 5: Inverse Homography transform causes non-sequential reads of source image

5.2 ms. We suspect that the USB3.0 interface adds additional I/O overhead. Moreover, the device occasionally returns NaN error after running for a while, forcing a reset.

The ZCU102 also features a small embedded Mali GPU for graphics programming. However, the existing kernel module only supports displaying the GPU-generated framebuffer to the Display Port (on-screen rendering). In this project, we do not use the Display Port of the ZCU102. We were able to get the GPU to perform off-screen rendering, i.e. storing the framebuffer to main memory. However, it wastes significant number of CPU cycles of copying the framebuffer to the memory region which is physically allocated for the FPGA to access. The overhead grows with the image resolution. To solve this problem, one could write a device driver to do a zero-copy using DMA engines without involving CPUs. Due to time constraint, we did not figure out how to do it properly, and this is subjected to future work. Therefore, in our current pipeline, we do not have any GPU renderer. For our applications, we assume that the rendered content is available and is stored in the DRAM. Thus, we only support static scenes (pre-rendered images) for now.

The programmable logic (PL, or FPGA) can directly access the DRAM main memory via AXI High Performance (HP) ports. There are 4 HP ports in total where each has a theoretical memory bandwidth of 2.4 GB/s. Thus, the aggregated memory bandwidth of the system is 9.6 GB/s [16]. However, we were not able to take full advantage of the bandwidth for our applications since our memory access pattern is random, as illustrated in Figure 5. Since we would like to keep the latency of our pipeline very low, we cannot afford to buffer the result image back to the main memory or the on-chip BRAM. Therefore, we perform an inverse-mapping operation of the homography transformation: for each destination pixel, we find the corresponding pixel of the source image by the inverse of the homography matrix. This allows us to stream the output pixels in raster order to the video controller implemented on the FPGA, which requires pixel scanlines for the display to behave properly. Due to this inverse-mapping operation, it is virtually impossible to do burst accesses or data caching to efficiently utilize the DRAM bandwidth. In addition, note that we are building a stereo headset, thus, the amount of memory traffic doubles to support both left and right channels.

As the ZCU102 only has one on-board HDMI output slot, we utilize the dual FMC slots to provide two HDMI outputs.

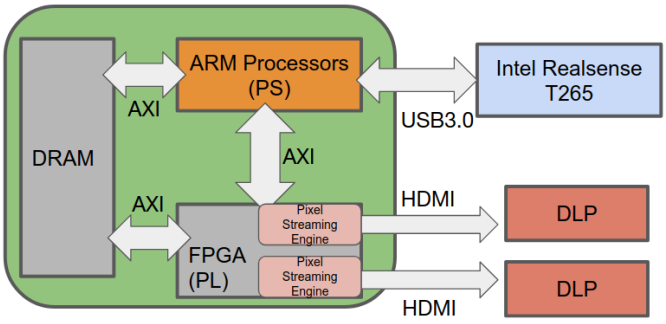

Figure 6: Headset System Overview

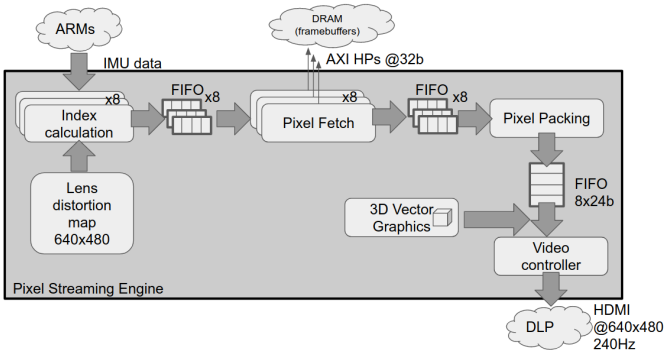

Figure 7: Pixel Streaming Engine Architecture

Table I: Pixel clock values of different resolutions

Resolution	Frame size	Pixel clock (ns)
640×480	800×585	9.92
800×600	1056×628	6.28
1280×720	1650×750	3.37
1920×1080	2200×1125	1.68

In particulars, we use the HDMI FMC cards from AVNet. The card has both HDMI In and Out ports. For our current study, we only need to use the Output port to stream the output image to the DLP. One advantage of using FPGA is that we can easily modify the pixel clock to increase the frame-rate of the HDMI Output. The pixel clock can be understood as the amount of time to generate and output a single pixel. Given a target frame-rate and resolution, we can calculate the pixel clock using the following formula.

$$pixel\_clock = 1/(frame\_rate \times frame\_width \times frame\_height)$$

As an example, with a target frame-rate of 240Hz, one could calculate the pixel clocks of different resolutions as in Table I.

Note that we use the frame size instead of resolution when calculating the pixel clock. The frame size includes the horizontal and vertical blanking periods which are the period of times the video controller waits before emitting next pixel scanline or next frame. For our FPGA circuit design, the pixel clock is used to clock the entire hardware circuit, so that new pixel is output every clock cycle. As can be seen from the table, the pixel clock becomes increasingly demanding as one goes to higher resolution, and it is more challenging for the CAD tool to do timing closure. In our current study, we only target a resolution of 640x480 to reduce timing closure effort (and another contributing factor is memory bandwidth). Exploring different design spaces with higher resolutions is definitely an important future work for our system.

C. Pixel Streaming Engine

Figure 7 details the architecture of our Pixel Streaming Engine implemented on the Programmable fabric. This module also performs Post-render warping (or inversed homography transformation) as mentioned in previous section. The ARM processor performs a AXI4-Lite bus transaction to write the IMU pose data to the Memory-mapped IO registers from which the module can read. The pose data includes the rotational information in the form of quaternion as well as translation information (optional). Besides, a lens distortion map lookup table is stored in the BRAM for fast access (one-cycle read) to do lens correction (which will be discussed in the subsequent section). The pixel index calculation is therefore a distortion map lookup operation followed by the inverse homography transform.

Once the source pixel index is computed, the Pixel fetch logic sends memory request to the memory controller for that pixel. As discussed earlier, the memory accesses are hardly sequential which lead to poor memory bandwidth utilization. Another issue is the long memory access latency to the external DRAM. In order to overlap the memory latency as well as improve the memory throughput, we instead instantiate multiple parallel Pixel fetch units, and in turn multiple Index calculation units. The Pixel fetch units issue parallel memory requests to the memory controller, and thus get multiple pixels in return; the Pixel packing module packs those pixels to a single wide pixel data that is sent to the video controller. Our video controller is configured to work with multiple pixels per clock, i.e., for each pixel clock, the video controller expects to receive a multiple of pixels, but it still outputs one pixel to HDMI display per clock cycle. Thus, some internal buffering is required. In our implementation, we find that 8-pixel per clock is optimal in terms of balancing the memory requests to main memory and minimizing the amount of on-chip buffering. Setting a smaller number of pixels per clock leads to under-utilize the memory bandwidth, thus the Pixel Streaming Engine takes longer to output each pixel. On the other hand, a higher number of pixels per clock overwhelms the AXI bus to the memory controller due to complex arbitration of the AXI crossbars. Therefore, it results to longer memory access latency and also wastes huge amount of FPGA resource due to large crossbars and internal buffers.

Since the latency of external memory access is unpre-

dictable, we instantiate FIFOs among the units to ensure that each unit is able to operate in its own rate and there is no stall of execution in the system. The FIFO to the video controller is particular important, as it needs to always have available data so that the display remains steady and stable without flickering. In our implementation, we set the size of that FIFO to be 16,348 elements of eight-wide 24-bit (R, G, B – each 8-bit) pixel data. We also set a fill time – the time that the video controller waits until the FIFO is filled with enough data before it begins streaming pixels to ensure steady display. The fill time is 16,000 cycle where each cycle is 9.92ns, hence it is around 0.16 ms which is negligible to our pipeline. Note that this latency is a one-time cost at the initialization of the system. After the wait time, the video controller expects to retrieve the pixel data from the FIFO every cycle. In addition, the blanking periods also help to refill the FIFO. To produce a 240 frames per second, our Pixel Streaming Engine must finishes streaming a full frame in about 4.2 ms. Since this is a stereo system, we have two Pixel Streaming Engines, one for each display. A complication of such system is that the demand for memory bandwidth as well as the fabric resource will double.

D. 3D Vector Graphics

As noted in the previous section, the post-render warping only works with 2D images, and we did not have time to make the Mali GPU works optimally with our system. To demonstrate that our headset can do some 3D tracking, rendering, and display, we implement a simple 3D vector graphics core on the FPGA. It is meant to build for the AR demo application. The module can rasterize some basic polygon shapes, such as quadrilateral or triangle. For each incoming pixel from the previous stage of the pipeline, we check if it lies within the polygon by doing line testing [17]. If yes, we color the pixel and no otherwise. We do not perform any interpolation scheme or shading. Therefore, the vector graphics is very compact and it is fully pipelined with a few pipeline stages (two or three). Hence, it has a minuscule impact on the latency of our headset system. We use the ARM processor to do the 3D-to-2D projection of the objects. We found that it is more efficient to do the projection on the processor since it runs at a higher clock frequency and has a better floating-point capability than the FPGA. Once the image coordinates of the objects are computed, we send them to the Pixel Streaming Engine, and then to the 3D Vector Graphics Engine to draw the object. The new coordinates are sampled every full frame.

E. Lens correction and Stereo

One of the biggest issues of a headset system is tackling the lens distortion problem. As our headset uses elliptical lenses, the reflected image get distorted. Worse yet, the left and right reflectors distort the image in the opposite ways, thus render it impossible to achieve stereo vision if one does not address this problem beforehand. Figure 8 shows one example of a badly distorted image on the lenses. The original image contains

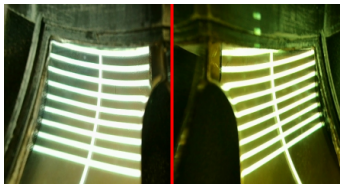

Figure 8: An example of lens distorted images

multiple horizontal white bars along with a single vertical white bar. The distorted images bent the bars differently for left and right channels.

To do lens correction, we inversely distort the input image so that the reflected image gets distorted again, thus become undistorted. As an example, some VR headsets present a barrel distortion on their lenses. The VR pipeline fixes this issue by applying pincushion distortion on the displayed image, so that when looking at the lenses, the user sees a rectified, corrected image. In our work, we need to figure out what kind of distortion effect is for our reflectors. Since we use the lenses from project NorthStar, we simply follow their lens calibration process. NorthStar performs a novel calibration process using ray tracing technique [18]. The outcome of the process is the distortion maps of the lenses. Due to time constraint, we did not rerun the entire laborious calibration process as did by NorthStar, but instead reuse their calibration file to generate the distortion maps for our lenses. We anticipate that it should as well work in our system since our headset frame is not so different from the original NorthStar structurally. As a future work, we will carry out our own calibration process to achieve better visual result.

Figure 10 shows an overview of the lens correction process. The distortion maps (in (u, v) coordinates) are used to perform lens correction. Each is interpolated to 640x480 to match the size of the input image. The input image is treated as a texture to be mapped onto the distortion map (surface). Since texture mapping operation requires GPU to run efficiently, we instead use a lookup-based approach instead. We convert the distortion maps to two lookup tables per map (one for column indices, LUT_x, one for row indices, LUT_y) stored in the BRAMs of the FPGA. This only requires one-cycle read latency from the BRAMs with the trade-off of additional memory storage. Lens correction is essentially a remap operation from the destination image I_{dst} to the source image I_{src}.

$$I_{src}(LUT_x(x), LUT_y(y)) = I_{dst}(x, y)$$

We did not apply the input image on the full maps obtained from the Northstar calibration file, as the remapped image still had some lens-distortion artifacts. We speculate that there might be some slight variation between our headset and the Northstar used for the calibration, so that the remapped results are not perfect. We made some adjustment to the maps manually to fine-tune the re-mapped image until it appears rectified on the lens. Basically, we attempt to find the region of the maps to land the input image to obtain the

Figure 9: An example of lens corrected images

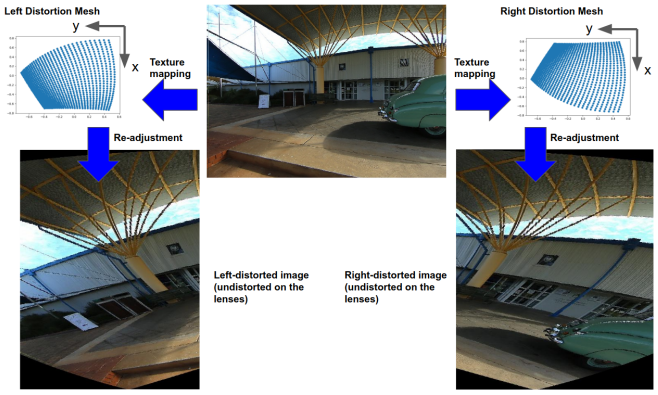

Figure 10: Lens correction process

best lens-correction output. However, by doing so, it results to a reduction in field-of-view (FoV) our our headset. A FoV defines the horizontal and vertical angles covered by each headset's lens. The original image in Figure 10 is generated with a horizontal FoV of 90°and a vertical FoV of 113°. We can see that the horizontal FoV of the remapped image is somewhat around 45°or less.

Figure 9 shows how the reflected image of the previous example looks like after applying lens correction. In addition to lens correction, the distortion maps give us stereo vision as well, i.e. the images of left and right lenses converged, since the maps were generated in a stereo NorthStar setup, though it is not entirely perfect and the user sometimes has to adjust the handhelds to see a better stereo image.

V. Evaluation

We evaluate our mixed reality headset with some simple VR and AR applications. For all the experiments, our target resolution is 640×480, frame-rate is 240Hz, and color channel is RGB 24-bit.

A. Demo Applications

1) Cubemap Rendering: A cubemap consists of six images/texture that renders the surrounding of an object. The six images define the top, bottom, left, right, up, down faces of the virtual environment. For our application, we use the following cubemap shown in Figure 11 where each RGB image has a dimension of 1000×1000. The cubemap is stored in the DRAM with an offset of 1000×1000 between each face. Thus, the amount of storage is around 23 MB (we treat each pixel as 32-bit or 4B data type). The cubemap is pre-loaded to the DRAM at a known physical address once the application

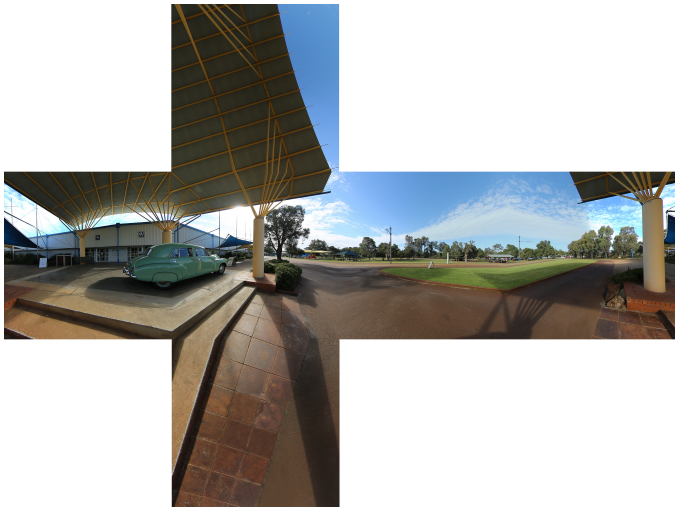

Figure 11: Cubemap

starts running. The FPGA is informed of the physical address of the cubemap for accessing.

By providing a complete surrounding, the cubemap application functions as if the user sees a 360-degree image when he rotates his head. Note that there is no translation involved in this application; only the head orientation data is taken into account. The orientation data forms a rotation matrix which transforms the current displayed image based on the head's orientation. The transformed pixels are fetched by the Pixel Streaming Engines to stream to the display. In addition to the post-render warping transformation and lens correction, there is additional task to determine the correct cube face to render. As a supplemental material, the video of our demo can be found at the following URL: https://drive.google.com/file/d/1UkzO0RdaWEVSbzQ4hBjFIvm6jndyUH3g/view?usp=sharing.

2) Static AR object overlay: Thanks to the 3D vector graphics engine, we can draw some basic 3D objects such as boxes to the displays. Our application first allows the user to define the location and shape of a box as well as the number of boxes in the environment (camera/headset coordinates). When the visor moves around, the coordinates of the objects are updated based on the latest pose information. Then, the processor performs a 3D-to-2D projection to map the objects to image coordinates. The 2D coordinates are then streamed to the Pixel Streaming Engines to rasterize the objects. By taking advantage of the Floating-point Unit and the Neon SIMD unit on the ARM processor, the projection task can run very fast, under sub-milisecond (depending on the number of objects in the scene). This application supports both translation and rotational movement. The visor should expect to see static objects when walking around. However, as the time of submitting the report, the demo does not work quite well. The AR objects appear to be drifted away after the visor moves for a while. This could be attributed to the error accumulation of the Realsense T265. In addition, object scaling is not very realistic when

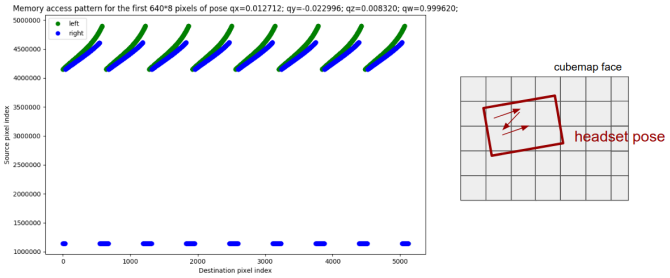

Figure 12: Memory access pattern of pose 1

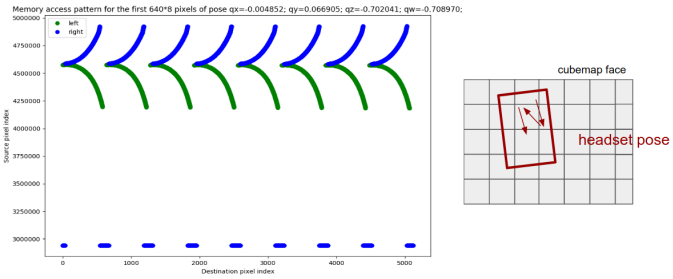

Figure 13: Memory access pattern of pose 2

the visor moves farther or closer to the virtual object. Further investigation and calibration are necessary to make the application work properly. The video of our demo can be found at the following URL: https://drive.google.com/file/d/1UlA3nw_gSl9GVk3BWwzBNQu1g09GrzN_/view?usp=sharing. In this demo, three virtual boxes with different shapes are overlayed in the environment. The boxes should ideally remain stationary regardless of user's movement.

B. Memory performance analysis

Since the AR application does not access external memory, we focus on the VR application (cubemap rendering) for memory access analysis. Figure 12, 13, 14 shows the access patterns of different poses for the first 8 scanlines of the output image when the application is running. The access pattern in pose 1 exhibits a decent amount of spatial and temporal locality. This is because this pose does not have much rotation in any axis, therefore, the output image aligns well with a corresponding cubemap face. Pose 2 and 3 are obtained when the headset is rotated 90°and 180°in the z-axis (the user "rolls" his head) respectively, thus access patterns change drastically and do not utilize cache memory well. In the three cases, there is little overlap between the left and right channel. Hence, it is challenging to design an efficient memory reuse strategy for such application.

C. FPGA Resource consumption

Table II provides the resource consumption of our FPGA hardware circuit implementation. The memory interconnect consumes most of LUT and FF resources. We have a high BRAM utilization due to internal buffering in order to mitigate the unpredictable external memory read latency so that the

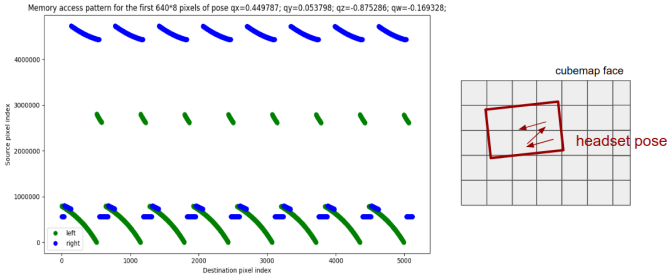

Figure 14: Memory access pattern of pose 3

Table II: ZCU102 Resource Utilization

LUT	55.84%
FF	32.70%
BRAM	87.28%
DSP	71.03%

video streaming remains stable. The DSP blocks are mainly used for pixel index calculation as well as the vector graphics core. The DSP resource grows as we increase the number of AR objects that the vector graphics core can draw.

D. Pixel Streaming Engine Performance

To justify the decision of putting the Pixel Streaming Engines in the Programmable fabric, we also did a software version that runs in the ARM processor. The software essentially performs index calculation by using homography transformation and distortion remap operation, similar to the hardware counterpart. However, the result image is stored back to the main memory. The video controller in the FPGA then streams the image to the display. In this setting, we only use the FPGA to implement the video controller for interfacing with HDMI output. Table III shows the performance of the SW and HW implementations of the Pixel Streaming Engine. The runtime of the SW version varies depending on the sampled pose. Drastic pose change also causes cache misses, thus degrades the performance. In overall, the HW version outperforms the SW counterpart by 19.7x on average. More importantly, it remains steady during the course of execution regardless of pose change which is very critical to ensure stable video streaming. Note that the SW version actually runs two Pixel Streaming kernels, because we have left and right channel, and we did not attempt multi-threading for further improvement. Nonetheless, even with parallel execution of left and right channel on the ARM processor, it still falls short to meet the timing demand of our application (4.2 ms, or 240 FPS).

Table III: Pixel Streaming Engine Performance: SW vs. HW

SW (ms)	**HW (ms)**	**Speedup (avg.)**
78-86	4.2	19.5

Table IV: System Performance

Stage	Platform	Latency (ms)
IMU Sampling	ARM	4.8-5.2
3D-to-2D projection (3 AR objects)	ARM	0.013
Pixel Streaming Engines	FPGA	4.2
Display	DLP	4.2
End-to-end latency (avg.)		13.4

E. System Performance

In this section, we evaluate the end-to-end latency of our pipeline. Table IV gives the latency of each stage of our system. The latency results are measured internally using a timer in Linux. We ignore the memory initialization time (e.g., loading cubemap to DRAM) because the cubemap does not change over the course of our application. We also ignore the FIFO fill time and the vector graphics pipeline since they incur very minimal latency.

Our end-to-end pipeline, from motion to photon, latency is 13.4 ms on average. Interestingly, the DLP has an internal latency of a frame (we verified it by using the high-speed camera – detail is omitted). We suspect that because DLP works as a color-sequential display, whereas the HDMI input is an interleaved RGB frame, it needs to buffer the frame internally first, then performs frame separation to different color sub-frames, as described in [2].

VI. Conclusions

We have shown what it takes, at minimum, to build an open-source, low-latency, mixed reality headset from scratch that is fully functional with simple VR and AR applications. Our headset achieves a motion-to-photon latency of 13.4 ms which meets VR target but fails to satisfy AR goal. Our prototype headset is fully open-source and does not tie to any specific hardware platform or proprietary software stack; other researchers could attempt to build the same architecture or leverage ours to develop novel applications or features. Though our accomplishment might seem humble, we believe this is a baby step forward to make a great stride in democratizing VR/AR headset and encourage future academic researches in this direction.

VII. Future work

As the time of writing this report, we have only finished half of what we set out to do. We have shown that the headset can be built with a low motion-to-photon latency. However, we did not have time (and resource) to evaluate the motion-to-photon latency of different headsets available in the market. For our own system, we measure the latency internally using the get-time function in Linux. However, to truly evaluate the motion-to-photon latency of different devices, we need a more systematic and cross-platform approach. One potential approach is using a high-speed camera that captures both the motion and the headset display, then measure the length of time between them. Another method is conducting user experience study. For example, paid volunteers might be asked to use different headsets for a period of time, and report on their feelings. This study might give an insight to how much latency a person can tolerate before suffering from motion sickness.

From the system perspective, many aspects could be improved in the future. Our system only supports a resolution of 640x480 at 24-bit RGB color – clearly way below modern VR/AR standard. Due to time constraint, we did not have enough time to experiment with higher resolutions. Higher resolutions at high frame-rate is definitely a challenging problem to solve, as it entails clever circuit design techniques to meet the timing and the available on-board resource. For our current work, we did not pay attention to best hardware design practices; we used High-level Synthesis to quickly prototype the required Computer Vision algorithms and ensured that the entire circuit met target clock frequency. Therefore, our FPGA circuit might not be optimal from resource stand-point.

From the user experience, besides low resolution, our headset suffers from jaggy/pixelated effect since we only use nearest-neighbor filtering. To achive higher-quality output image, some advanced interpolation scheme is definitely inevitable. One could apply a bilinear interpolation which fetches four nearest pixels per pixel index calculation. However, as discussed before, memory bandwidth is the bottleneck of our current implementation. We were not successful in implementing bilinear interpolation in our circuit design, since the FIFO to the video controller drains quickly due to high memory request traffic. We attempted to configure the Quality-of-Service block of the DDR memory controller, but failed to get any substantial improvement. As a future work, we need to think of a better strategy to organize memory accesses, perhaps by bank allocation or custom memory controller. We also have to carefully examine the memory systems of ZCU102 to see how far we can push the limit. Our feeling is that we have not yet exhausted its capability.

In this work, we only build the headset from the system perspective. We have yet to pay attention to the programming side. How to truly make a headset that is easy to port mixed reality applications from the upper layer of the stack is an open question. The industry is trying solve the VR/AR fragmentation problem by introducing the OpenXR standard [19] to cover application-space and runtime of mixed reality devices. We could try to support it in future work.

A mixed reality headset can be viewed as an edge device that functions in a networked computing environment. We are working towards providing high frame-rate video streaming input to the headset. The video could be coming from a desktop with powerful rendering GPU to the network via high-bandwidth, low-latency network, such as 5G/mmWave. In addition, enabling multiple headsets to work collaboratively is also an interesting problem.

To enhance the accuracy of AR object tracking, we could also leverage outside-in tracking along with the inside-out tracking of the headset. For example, Optitrack sensors could be placed in the surrounding environment to perform localization of the visor.

VIII. Acknowledgement

This is an on-going collaborative research project among many groups at CMU and UC Berkeley. Prof. Anthony Rowe (CMU) initiates the ambitious idea of building an open-source VR/AR headset using high frame-rate projectors. The author would like to especially thank Vinamra Benara (work done at CMU, now PhD student at UC Berkeley) for his early contribution for cubemap application which helps shape the course of the project. The author would like to particularly thank John Miller (CMU) for numerous advice drawing from his brilliant technical expertise, especially on lens correction and IMU filtering. Special thanks to Joshua Corona (CMU) and Nathan Seraphin (CMU) for prototyping as well as assembling the mechanical components of the headset. The author thanks Jonathan Nee (CMU) and Marie Nguyen (CMU) for their helps related to the FMC setup for HDMI display. Thanks to all the advisors: Prof. John Wawrzynek (UC Berkeley), Prof. James Hoe (CMU), and Prof. Anthony Rowe (CMU) for their advice and feedback. The author thanks the instructors of EE290T and CS262A, Prof. Avideh Zakhor and Prof. John Kubiatowicz, for their feedback and permission to the author to pursue this as course projects (since this seems to be an odd choice to both course topics).

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Program (SRC) sponsored by DARPA.

References

[1] https://www.marketwatch.com/press-release/augmented-reality-and-virtual-reality-market-737-cagr-2018-2025-overview-share-scope-revenue-trade-analysis-trends-2019-11-20.

[2] https://medium.com/@DAQRI/motion-to-photon-latency-in-mobile-ar-and-vr-99f82c480926.

[3] https://www.roadtovr.com/exclusive-how-nvidia-research-is-reinventing-the-display-pipeline-for-the-future-of-vr-part-1/.

[4] https://developer.leapmotion.com/northstar.

[5] https://www.anandtech.com/show/14775/hot-chips-31-live-blogs-microsoft-hololens-20-silicon.

[6] P. Lincoln, A. Blate, M. Singh, T. Whitted, A. State, A. Lastra, and H. Fuchs, "From motion to photons in 80 microseconds: Towards minimal latency for virtual and augmented reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 4, pp. 1367–1376, April 2016.

[7] P. Lincoln, A. Blate, M. Singh, A. State, M. C. Whitton, T. Whitted, and H. Fuchs, "Scene-adaptive high dynamic range display for low latency augmented reality," in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '17. New York, NY, USA: ACM, 2017, pp. 15:1–15:7. [Online]. Available: http://doi.acm.org/10.1145/3023368.3023379

[8] J. Kim, Y. Jeong, M. Stengel, K. Akşit, R. Albert, B. Boudaoud, T. Greer, J. Kim, W. Lopes, Z. Majercik, P. Shirley, J. Spjut, M. McGuire, and D. Luebke, "Foveated ar: Dynamically-foveated augmented reality display," *ACM Trans. Graph.*, vol. 38, no. 4, pp. 99:1–99:15, Jul. 2019. [Online]. Available: http://doi.acm.org/10.1145/3306346.3322987

[9] J. M. P. van Waveren, "The asynchronous time warp for virtual reality on consumer hardware," in *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, ser. VRST '16. New York, NY, USA: ACM, 2016, pp. 37–46. [Online]. Available: http://doi.acm.org/10.1145/2993369.2993375

[10] D. Evangelakos and M. Mara, "Extended timewarp latency compensation for virtual reality," in *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '16. New York, NY, USA: ACM, 2016, pp. 193–194. [Online]. Available: http://doi.acm.org/10.1145/2856400.2876015

[11] https://www.intelrealsense.com/how-to-getting-imu-data-from-d435i-and-t265/.

[12] https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[13] https://www.avnet.com/shop/us/products/avnet-engineering-services/aes-fmc-hdmi-cam-g-3074457345635221625/.

[14] http://www.ti.com/tool/DLPDLCR2010EVM.

[15] https://www.ssidisplays.com/rear-projection-film.

[16] https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[17] https://cs184.eecs.berkeley.edu/sp19/lecture/2/rasterization.

[18] http://blog.leapmotion.com/bending-reality-north-stars-calibration-system/.

[19] https://www.khronos.org/openxr/.