Energy-Efficient 360-Degree Video Rendering on FPGA via Algorithm-Architecture Co-Design

Qiuyue Sun
qsun15@u.rochester.edu

Amir Taherin
ataherin@ur.rochester.edu

Yawo Siatitse
asiatits@u.rochester.edu

Yuhao Zhu
yzhu@rochester.edu

University of Rochester
http://horizon-lab.org

Abstract

360° panoramic video provides an immersive Virtual Reality experience. However, rendering 360° videos consumes excessive energy on client devices. FPGA is an ideal offloading target to improve the energy-efficiency. However, a naive implementation of the processing algorithm would lead to an excessive memory footprint that offsets the energy benefit. In this paper, we propose an algorithm-architecture co-designed system that dramatically reduces the on-chip memory requirement of VR video processing to enable FPGA offloading. Evaluation shows that our system is able to achieve significant energy reduction with no loss of performance compared to today's off-the-shelf VR video rendering system.

CCS Concepts

• **Hardware** → **Hardware accelerators**; • **Computing methodologies** → *Virtual reality*; • **Computer systems organization** → Reconfigurable computing.

Keywords

Virtual Reality; Panoramic Video; Perspective Projection; Locality

ACM Reference Format:
Qiuyue Sun, Amir Taherin, Yawo Siatitse, and Yuhao Zhu. 2020. Energy-Efficient 360-Degree Video Rendering on FPGA via Algorithm-Architecture Co-Design. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3373087.3375317

1 Introduction

With the rapid development of panoramic cameras and popularity of Virtual Reality technologies, there is an explosion of 360° content in recent years. Content sharing websites such as YouTube, Facebook, and Instagram support editing, streaming, and sharing this new form of content, further accelerate the penetration of 360° videos. Fueled by next-generation cellular technologies such as millimeter wave that promise orders of magnitude higher bandwidth and lower latency, users soon will be able to create, share, and watch 360° videos just like any other media.

Although with huge opportunities, rendering 360° content is power-hungry. Previous work has shown that rendering a 720p 360° video in 30 frames per second (FPS) consumes over 4 W power [18, 19], exceeding the Thermal Design Point (TDP) of typical mobile devices [14]. The reason that rendering 360° is power-hungry is that today's rendering software translates 360° video rendering to a texture mapping problem that gets offloaded to the GPU [9]. Although using the GPU in off-the-shelf Systems-on-a-chip (SoCs) accelerates the adoption of 360° content, GPUs are power hungry.

We expect that next-generation mobile SoCs will soon integrate 360° content-specific Intellectual Property (IP) blocks to improve the rendering energy-efficiency. To facilitate this trend, we propose a new 360° video rendering accelerator. We choose to base the accelerator design on FPGA, which not only allows us to exploit the fine-grained, pixel-level parallelisms exist in the 360° content rendering algorithms, but also to retain flexibility to accommodate future developments in the rendering algorithms.

The key challenge of accelerating 360° rendering is the rendering algorithm's large memory footprint and irregular data access pattern, which not only introduce a high memory footprint that exceeds the on-chip memory of a mobile SoC and but also are not amenable to conventional memory optimizations such as line-buffering and prefetching. As a result, frequent, and random, DRAM accesses would have to be made, offsetting the energy benefit of hardware acceleration.

This paper proposes an algorithm-architecture co-designed system for efficient 360° content rendering. Our key observation is that the irregular memory accesses in today's rendering algorithm are fundamentally caused by the algorithm's data-flow that leads to arbitrary indexing of the input frame pixels. To tame the memory-inefficiencies, we propose a new rendering algorithm that, by design, enforces a different data-flow that guarantees a streaming memory access pattern. As a result, the rendering computation becomes a stream of stencil operations, each operating on a fixed-size window of pixels in a raster order.

The new rendering algorithm uniquely enables us to design a simple, yet efficient, hardware accelerator. The accelerator architecture exploits the pixel-level parallelisms by pipelining the rendering of different pixels, and hides the memory transfer latency with rendering computations. We judiciously apply a series of energy-oriented optimizations including trading-off the pipeline depth for the overall latency and tuning pixel data representations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA '20, February 23–25, 2020, Seaside, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7099-8/20/02...$15.00
https://doi.org/10.1145/3373087.3375317

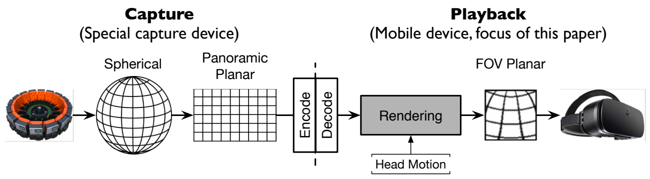


Fig. 1: **A typical** 360° **content delivery pipeline. This paper focuses on the client rendering on mobile devices.**

We implement our algorithm-architecture co-designed system on the Xilinx Zynq Ultrascale+ ZCU104 FPGA development board [7]. Comparing against an off-the-shelf baseline running on the Nvidia Jetson TX2 development board utilizing its mobile Pascal GPU [2], our system achieves 55% energy savings at the same frame rate.

In summary, this paper makes the following contributions:

- We provide a detailed analysis of the memory access patterns of today's 360° content rendering algorithm, and demonstrate the inefficiencies of conventional memory optimizations such as line-buffering. (§ 3).
- We propose a new 360° content rendering algorithm that improves the data locality of 360° content rendering, and thus enables efficient hardware acceleration (§ 4.1).
- We propose an accelerator architecture that is co-designed with the new algorithm to maximize its efficiency (§ 4.2).
- We prototype the co-designed system on an embedded FPGA and demonstrate significant energy savings (§ 5).

The rest of the paper is organized as follows. We first provide the background and related work of 360° content rendering (§ 2). We then analyze today's rendering algorithm (§ 3), focusing on its irregular memory accesses. We then introduce our new rendering algorithm and the co-designed hardware architecture (§ 4). We evaluate the our system (§ 5), followed by conclusion (§ 6).

2 Background

This section briefly introduces the necessary background and terminologies that are used throughout this paper. We refer interested readers to El-Ganainy and Hefeeda [12] for a comprehensive survey.

Panoramic Content 360° video is a form of Virtual Reality that has seen wide adoption recently in many areas such as news, movie, sports, and medical industry [21]. Fig. 1 shows an end-to-end 360° content delivery pipeline. It mainly consists of a capturing (creation) phase and a playback phase. 360° videos are typically created using special capture devices (e.g., an omnidirectional camera [8]) that capture every direction of the scene, which later are stitched together to form panoramic frames that present a 360° view of the scene to users. After creation, 360° videos are streamed to client VR device for playback, which is the focus of this paper.

Rendering Algorithm Once on the VR device, the playback software renders different regions of the frame according to the user's head movement. In the context of 360° video only rotational motion is captured, but not translational motion. The head motion can be characterized by the *polar* and the *azimuthal* angles in a spherical coordination system. The size of the displayed region depends on the field-of-view (FOV) of a particular device, which characterizes the vertical and horizontal angles of the viewing area.

Algorithm 1: Classic 360° video rendering algorithm.

```
Input: Input panoramic frame $I_{in}$; FOV size $\alpha$, $\beta$; Head
       orientation $\theta$, $\lambda$.
Result: Output FOV frame $I_{out}$.
$W_o, H_o = I_{Out}.res$ ;                    // Output resolution
// iterate over all coordinates in output frame
for $i = 0;\ i < H_o;\ i = i + 1$ do
    for $j = 0;\ j < W_o;\ j = j + 1$ do
        // rotation
        $<x, y, z> = \mathcal{R}(i, j, \alpha, \beta, \theta, \lambda)$;
        // projection
        $<u, v> = \mathcal{P}(x, y, z)$;
        // filtering
        if u and v are integer coordinates then
            $I_{out}(i, j) = I_{in}(u, v)$;
        else
            $I_{out}(i, j) = \mathcal{F}(I_{in}, u, v)$;
        end
    end
end
```

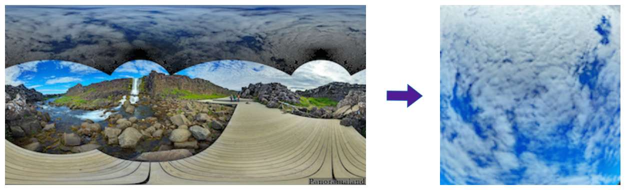
Conventional video frames, once decoded, can be directly rendered on the display. However for 360° videos, the client rendering software converts an input panoramic frame (decoded from the video streamed from the cloud) to a frame that contains only the user's viewing area based on the user's viewing angle and device's FOV. Prior work shows that the rendering algorithm contribute about 40% of the device power consumption [19].

Hardware Architecture Today's off-the-shelf mobile SoCs directly support rendering 360° videos. In particular, the video codec first decodes 360° videos into a set of panoramic planar frames; the Graphics Processing Unit (GPU) is then used to execute the rendering algorithm that converts the panoramic planar frames to FOV frames that are then sent to the display processor [9].

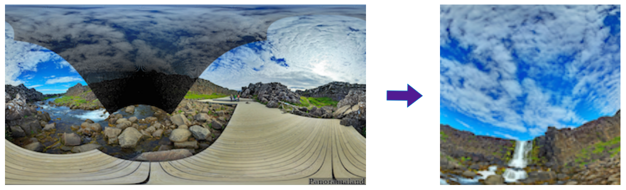
The reason that the GPU is tasked with the rendering algorithm is that the latter can be viewed as a texture mapping problem [15], where the planar panoramic frame is treated as a texture map that is mapped to a particular region on a sphere. The spherical region's size is the same as the device's FOV and its location is determined by the user's viewing angle. Modern GPUs can effectively execute texture mapping through the specialized Texture Mapping Unit (TMU) and the texture cache [13]. The TMU accelerates the computation of texture mapping, and the texture cache captures the irregular data access pattern to the texture map, i.e., the input panoramic frame in the case of 360° video rendering.

3 Rendering Algorithm Analysis

This section first presents the algorithm used in today's 360° rendering software and discusses its computation pattern that is suitable for hardware acceleration (§ 3.1). We then particularly focus on the irregular memory access patterns of the algorithm (§ 3.2), from which we motivate the need for a new algorithm-architecture co-designed strategy.



(a) Head orientation at (45°, 90°).



(b) Head orientation at (45°, 45°).

Fig. 2: 360° **frame rendering at two different head orientations. The left figures are the input panoramic frame, and the right figures are the output FOV frame**[1]**. The black pixels in the left figures are the pixels that are referenced during the rendering process.**

3.1 Algorithm and Its Computation Patterns

The goal of the rendering algorithm is to generate an output frame I_{out} from the input panoramic frame I_{in}. Algo. 1 shows the pseudocode. Specifically, the rendering algorithm calculates each output frame pixel ($<i, j>$) by mapping it to a pixel in the input frame ($<u, v>$), effectively sampling the input frame. The mapping is done by raster scanning all the points in the output frame and iteratively applying two operations, *rotation* ($\mathcal{R}$) and *projection* ($\mathcal{P}$), on each $<i, j>$ point to obtain its corresponding $<u, v>$ coordinates. $\mathcal{R}$ and $\mathcal{P}$ are matrix multiplications and cartesian-spherical conversions to support perspective rotation and projection [20]. The renderer then uses the $<u, v>$ coordinates to look up the input frame, and returns the corresponding input pixel as $I_{out}<i, j>$. If $<u, v>$ are not integer coordinates, the renderer applies a so-called *filtering* function ($\mathcal{F}$), such as nearest neighbor or bilinear filtering [15], to return a "best approximation" of the pixel value at $I_{out}<i, j>$.

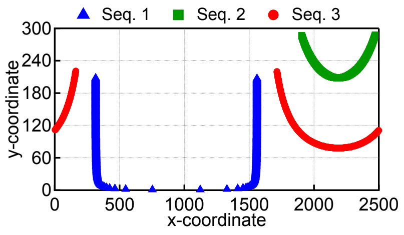
The rendering algorithm is highly parallel. In particular, the rendering of every output pixel is completely independent of each other. Under a particular head orientation, an output pixel's value $I_{out}(i, j)$ depends only on its coordinates $<i, j>$. In addition, the computation involved in $\mathcal{R}$, $\mathcal{P}$, and $\mathcal{F}$ are mostly affine transformations that are suitable for efficient hardware implementations. Overall, the computation patten is ideal for hardware acceleration.

3.2 Memory Access Patterns

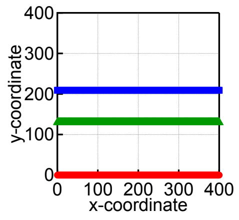
In stark contrast to the computation pattern, the memory access pattern of the rendering algorithm is far from ideal for an efficient accelerator design, especially on FPGAs.

Large Footprint The rendering algorithm accesses the memory in the filtering step, which uses the $<u, v>$ coordinates generated from the projection step to index into the input frame (I_{in}), and sequentially writes to the output frame (I_{out}) in the raster order. The output frame is small in size; its accesses are sequential, and thus could be buffered on-chip and efficiently streamed to the DRAM in the end [22]. However, the input frame can not be fully captured by a typical on-chip memory. For instance, a 1080 and 4K frame would require over 5.9 MB and 23.7 MB memory, respectively.

[1]All figures here are down-sampled to reduce their sizes.



(a) Input accesses.



(b) Output accesses.

Fig. 3: **Memory access sequences of input frame (left) and output frame (right) as scatter plots. Each marker $<x, y>$ indicates that the pixel at position $<x, y>$ is referenced. The three sequences correspond to when the rendering algorithm scans three different rows (i in Algo. 1) of the output frame. Different markers in the same sequence correspond to different columns (j in Algo. 1) on the same row. The head orientation is $(45^\circ, 90^\circ)$, same as in Fig. 2a.**

Irregularity The pixel access pattern of the input frame is non-sequential, which severely hurts the efficiency of hardware acceleration. Fig. 2a shows a rendering example where the input frame on the left is rendered to the output frame on the right. In this example, the FOV size is $110^\circ \times 110^\circ$, and the head orientation is $(45^\circ, 90^\circ)$. The black pixels at the top of the input frame refer to all the pixels that are accessed by the rendering algorithm. To better illustrate the memory access pattern, Fig. 3a plots the input pixels that are accessed as the rendering algorithm iterates over three output frame rows, which are shown in Fig. 3b. Each $<x, y>$ marker in the figures indicates that the pixel at position $<x, y>$ is referenced.

Although the output frame pixels are accessed sequentially in a streaming fashion, the input frame pixels are referenced irregularly as is evident in the three access traces in Fig. 3a. Irregular memory accesses are known to hurt efficiency for two reasons. First, random DRAM accesses consume much more energy than sequential DRAM accesses [6, 17]. Second, irregular memory accesses require explicit control logic to manually coordinate the traffic between DRAM and on-chip memory [10]. This is particularly an issue for FPGAs, which implement control flow logic rather inefficiently. Ideally, FPGAs prefer streaming data accesses, which exhibit strong locality and can be effectively captured by memory optimizations such as line-buffering [16]. The non-raster access order of the input pixels indicates that line-buffer would be inefficient.

To quantify the ineffectiveness of using a line-buffer, Fig. 4 shows how the line-buffer efficiency (left y-axis) and hit rate (right y-axis) vary with the line-buffer capacity when rendering the frame in Fig. 2a. The efficiency is defined as the percentage of pixels that are brought into the line-buffer and that are actually referenced; the miss rate is defined as the number of memory references that are not found in the line-buffer and thereby cause pipeline stall; the line-buffer capacity is characterized by the number of lines in the input frame the line-buffer can hold. The rendering algorithm requires about 512 lines, which roughly equate almost 4 MB of

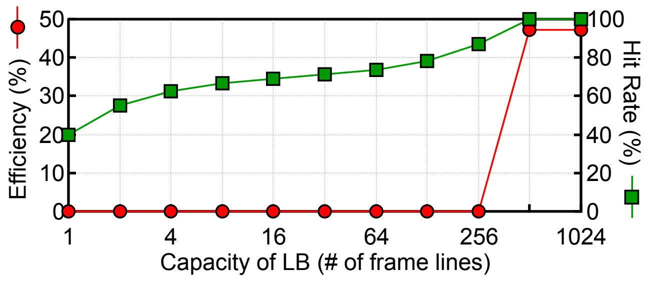


Fig. 4: **The efficiency (left) and miss rate (right) of using a line-buffer vary with the line-buffer capacity.**

line-buffer size, to reach 100% hit rate. Even under such a large line-buffer size, over 50% the fetched pixels would never be referenced, leading to significant bandwidth waste and energy-efficiencies.

Variation The irregular memory access pattern varies both spatially and temporally, making static optimizations ineffective.

On one hand, the rendering algorithm exhibits different input access patterns when iterating over different output rows as shown in Fig. 3a, exhibiting spatial variance. On the other hand, although the input access pattern is deterministic given a particular head orientation, the access pattern changes across different head orientations as users move, exhibiting temporal variance. Fig. 2b illustrates the memory accesses for a different head orientation at (45°, 90°), which has a different pattern from that of (45°, 45°) shown in Fig. 2a. Since the head orientation is not known until runtime, pre-computing and memoizing the access streams for all possible head orientations would lead to prohibitive memory overhead.

4 Algorithm-Hardware Co-Design

We propose a new 360° content rendering algorithm, which streamlines the memory accesses while retaining the pixel-level parallelism, enabling practical FPGA acceleration. We first describe the algorithm (§ 4.1), and then describe the co-designed the hardware architecture and the implementation details (§ 4.2).

4.1 Algorithm

Overview Fundamentally, the root-cause of the irregular memory accesses in the original rendering algorithm is inherent in the algorithm's data-flow. In particular, the rendering algorithm calculates each output frame pixel by mapping it to a pixel in the input frame. Since the input pixel indexing is arbitrary, memory accesses to the input frame are irregular. Our idea is to *invert* the rendering algorithm such that it scans the input frame in the raster order, and maps each input pixel to one pixel in the output frame. In this way, the input frame is accessed in a streaming fashion, enabling efficient line-buffer optimizations. The trade-off is that output frame is now accessed in an arbitrary order. However, this is an acceptable trade-off because the output frame is small in size and could be buffered on-chip before streaming out.

Inverting the original algorithm is possible because the rotation function ($\mathcal{R}$) and projection function ($\mathcal{P}$) are *unique* and thus are naturally invertible. The filtering function ($\mathcal{F}$) is not invertible because it is not unique. Consider the simple nearest-neighbor filtering; multiple input points could be mapped to the same output

Algorithm 2: Proposed 360° video rendering algorithm.

```
Input: Input panoramic frame $I_{in}$; FOV size $\alpha$, $\beta$; Head
       orientation $\theta$, $\lambda$.
Result: Output FOV frame $I_{out}$.
$W_i, H_i = I_{In}.res$ ;                    // Input resolution
/* iterate over all output boundary coordinates
   */
for <i, j> coordinates on the $I_{out}$ boundary do
    <u, v> = $\mathcal{P}(\mathcal{R}(i, j, \alpha, \beta, \theta, \lambda))$;
    Add <u, v> to $\mathbb{B}$ ; // $\mathbb{B}$ is the input boundary set
end
/* iterate over all input pixels                    */
for $u = 0;\ u < H_i;\ u = u + 1$ do
    for $v = 0;\ v < W_i;\ v = v + 1$ do
        if <u, v> within the boundary $\mathbb{B}$ then
            <i, j> = $\mathcal{R}^{-1}(\mathcal{P}^{-1}(u, v), \alpha, \beta, \theta, \lambda)$
            $I_{out}(i, j) = I_{in}(u, v)$;
        end
    end
end
/* apply filtering to all output pixels             */
foreach <i, j> in $I_{out}$ do
    $\mathbb{F}'$(<i, j>);
end
```

point that is the nearest neighbor to both input points. However, since filtering is inherently an approximation, we could approximate the filtering step without loss of visual quality as we will demonstrate later.

Reduce Redundancies Naively inverting the rendering algorithm, however, introduces lots of redundant computation. This is because only a small fraction of the pixels in the input frames is actually referenced during the rendering process. For instance, only 17.1% and 16.5% of the input pixels are referenced in Fig. 2a and Fig. 2b, respectively. In other words, the vast majority of the input pixels will not be needed to generate any output frame pixels, and therefore inversely mapping them would waste computation.

To reduce the redundant computations, our idea is to determine the boundary of the input region that contains the pixels that are needed for rendering. In this way, we are able to apply the inverse mapping only to the input pixels that are within the boundary. Input boundary calculation can be easily achieved by applying the original rendering algorithm to the output boundary coordinates. Since boundary pixels are only a very small portion of the entire frame, boundary calculation has low overhead as we show later.

Algo. 2 shows the pseudocode of the new algorithm. It first applies the original rotation and projection functions $\mathcal{R}$ and $\mathcal{P}$ to generate a boundary set $\mathbb{B}$. It then iterates over all the input pixels, and apply inverse functions $\mathcal{R}^{-1}$ and $\mathcal{P}^{-1}$ to pixels that are within the boundary delineated by $\mathbb{B}$. In the end, it applies a filtering step of the entire output image. Note that this filtering function $\mathcal{F}'$ is not, and needs not to be, the same as the original filtering function $\mathcal{F}$ or its inverse $\mathcal{F}^{-1}$ due to the approximate nature of filtering.

Output Quality The output of our new rendering algorithm is not pixel-accurate compared to the original algorithm because the filtering function is non-invertible. We verify that the difference is acceptable, both objectively and subjectively.

Objectively, we use two metrics to quantify the difference between the outputs generated by our rendering algorithm and the original algorithm: the Peak Signal to Noise Ratio (PSNR) and the Normalized Root Mean Square Error (NRMSE). The PSNRs across three representative viewing angles, (0°, 0°), (45°, 45°), and (45°, 90°), are 40.4, 57.1, and 42.6, respectively, and the NRMSE is below 0.01, confirming the high precision of the new algorithm. Subjectively, we also conduct subjective user study and find that the difference is visually indistinguishable.

4.2 Architecture Co-design

We co-design the hardware architecture to maximize the efficiency of the proposed rendering algorithm. Fig. 5a shows the overall execution model. The boundary calculation is serialized with the rest of the processing because it provides the boundary set for testing input pixels. The input frame is streamed from the DRAM, which is overlapped with pixel rendering. We exploit the pixel-level parallelism of the new algorithm by pipelining the processing of different pixels. The pipeline has an initiation internal of one. That is, a new pixel starts execution every cycle. During pixel rendering, the output frame is buffered on-chip and is streamed out in the end.

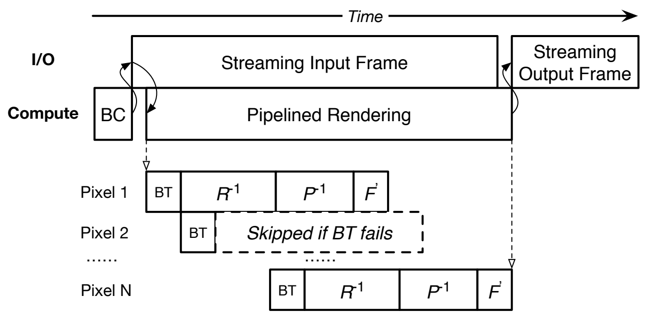
The architecture block diagram is illustrated in Fig. 5b. The boundary calculation and the rendering module both use a set of multiply-accumulate (MAC) units and trigonometric function hardware to support the perspective rotation, cartesian-spherical conversion, and filtering operations. To support the streaming I/O, we use the simple AXI4-Stream interconnect design, which has efficient IP implementation on FPGA [22].

Optimizations We apply a series of optimizations to improve the performance and reduce resource utilization. First, the boundary test is on the critical path and thus impacts the overall performance. Testing precisely whether a pixel is within the boundary requires storing all the boundary pixels and many comparisons. To reduce the boundary test, we approximate the boundary by its smallest bounding box (a rectangle), which requires us to store only four parameters and only four comparisons for boundary test. The trade-off is that the rendering algorithm now has to process more pixels that are not in the boundary. We find that this is a desirable trade-off because the benefits of reducing the per-pixel latency out-weights the overhead of pipelining more pixels.

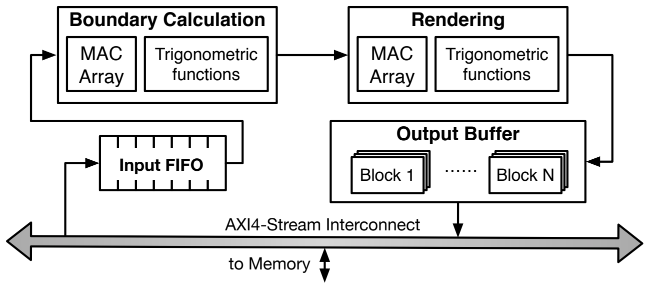
In addition, we choose to use a fixed-point representation for computation to improve the resource utilization and speed. We empirically find that a 28-bit representation with 14 bits for the integer part is sufficient to guarantee negligible loss of visual quality.

5 Evaluation Results

This section first introduces the experimental setup (§ 5.1). We then evaluate on a set of micro-benchmarks using different resolutions and head orientations to understand the efficiency of the co-designed system (§ 5.2). Finally, we present the evaluation results on a 360° dataset using real user head orientations (§ 5.3).



(a) The execution model. "BC" stands for "Boundary Calculation", i.e., the first loop in Algo. 2; "Pipelined Rendering" is the second and third loop in Algo. 2; "BT" stands for "Boundary Test", i.e., the test condition of the second loop in Algo. 2. Execution times are not to scale.



(b) Architecture block diagram.

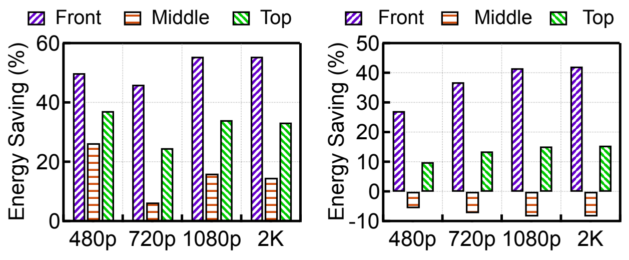
Fig. 5: **Architectural support for the proposed rendering algorithm.**

5.1 Experimental Setup

We implement our architecture on the Xilinx Zynq UltraScale+ MPSoC ZCU104 development board [7], which is specifically designed for embedded visual applications such as Augmented Reality and Virtual Reality. It has a programmable logic with 1.38 MB on-chip BRAM. We synthesize, place, and route the design using the Xilinx Vivado tool chain, and obtain the post-layout power consumption. The design is clocked at 100 MHz, which meets the 30 FPS real-time target for all resolutions. The ZCU104 development board uses a 2 GB, 64-bit wide DDR4 memory system [4]. We estimate the DRAM power using the Micron DDR4 power calculator [1, 5] based on the application's memory access traces.

Baselines We compare against two baselines. First, we compare against a baseline that implements the original rendering algorithm (Algo. 1) on the mobile Pascal GPU available on the Nvidia Jetson TX2 development board [2]. TX2 is used in many off-the-shelf VR devices such as the ones from Magic Leap [3]. This baseline is representative of how 360° video rendering is performed in today's VR devices as discussed in § 2. GPU power is measured using TX2's built-in TI INA 3221 voltage monitor IC, from which we retrieve power consumptions through the I2C interface.

Second, we compare against an FPGA baseline that implements the original rendering algorithm on the ZCU104 FPGA. Comparing against this baseline normalizes the effect of FPGA acceleration and thus highlights the benefits of algorithm-architecture co-design.



(a) **GPU baseline comparison.** (b) **FPGA baseline comparison.**

Fig. 6: **Microbenchmarking energy savings over the baselines across different resolutions and head orientations.**

5.2 Microbenchmark Results

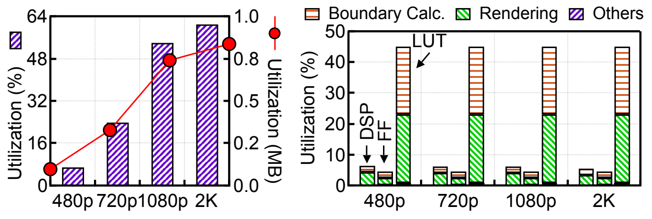
We evaluate four different resolutions, including 480p, 720p, 1080p, and 2K, to represent different rendering requirements. Since different head orientations affect how many input pixels are processed, we evaluate three different head orientations, (0°, 0°), (45°, 45°), and (45°, 90°), which mimic users watching the front (around the Equator), middle, and top (around the North Pole) region of a video.

Energy Savings Our co-designed system achieves significant amount of energy savings compared to the GPU baseline. Fig. 6a shows the energy savings per frame across different resolutions under the three viewing angles. Under the 480p and 2K resolutions and the front viewing angle, our system is able to save close to 75% and 55% of the energy compared to the GPU baseline, respectively. On average, under a 2K resolution, our system consumes about 1.4 W of power, of which about 65% is the dynamic power.

Our co-design system also out-performs the FPGA baseline with the original rendering algorithm in most cases, as shown in Fig. 6b. The only exception is under the middle viewing angle. This is inherent to our new rendering algorithm, which processes more pixels when the viewing angle is near the (45°, 45°) region of the sphere. Recall from § 4.2 that we first calculate a bounding box and then process all the pixels that are encapsulated by the box. The bounding box contains more pixels when the viewing angle is near the middle than near the Equator and the Poles.

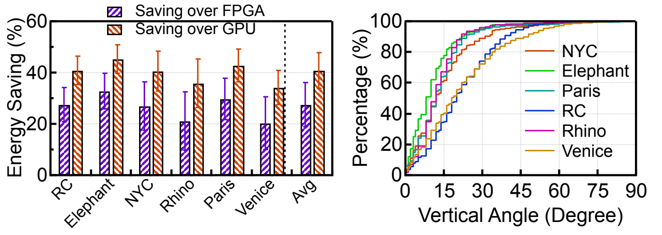
Latency Breakdown We find that each frame's execution time is consistently dominated by pixel rendering. We break down the frame latency into three main phases: boundary calculation, pixel rendering (which includes input streaming, which overlaps with pixel rendering), and output streaming. Regardless of the resolution, the pixel rendering time contributes to over 90% of the total frame latency. The boundary calculation has negligible execution time ($< 0.2\%$), indicating that although it is on the critical path of frame latency, it is far from being a bottleneck.

Resource Utilization Finally, we show that our proposed system has low resource utilization. Fig. 7a shows the BRAM utilization across different resolutions. The BRAM usage increases from 0.1 MB at 480p resolution to 0.84 MB at 2K resolution, but is still well under the budget of the mobile-grade Ultrascale+ FPGA. Fig. 7b shows the utilizations of other FPGA resources including DSP, FF, and LUT. Their utilizations do not change across resolutions because the underlying data-path is exactly the same for different resolutions. Although boundary calculation contribute little to the frame latency, it occupies significant amount of FPGA resources. This is a typical resource-performance trade-off that accelerators make.



(a) **BRAM utilization.** (b) **DSP, FF, LUT utilization.**

Fig. 7: **Resource utilization at the (0°, 0°) viewing angle.**



(a) **Energy savings.** (b) **CDF of vertical viewing angles.**

Fig. 8: **Evaluation results on real user viewing traces.**

5.3 Real User Trace Evaluation

We evaluate on a recently released 360° dataset [11], which includes the per-frame head orientations of 59 real users watching six different YouTube 360° videos. Fig. 8a shows the average energy saving of our system over the FPGA and the GPU baselines. The error bars indicate one standard deviation across all the users. On average, we achieve 26.4% and 40.0% energy savings over the GPU and the FPGA baselines in all five benchmarked videos across all users.

We find that when watching 360° videos users tend to focus on the scenes in front of them, under which circumstances our system is able to significantly out-perform the baselines as quantified before using microbenchmarks (Fig. 6). Fig. 8b shows the cumulative distribution function of the absolute vertical angles of all users across all videos. Each $< x, y >$ point in Fig. 8b reads as: users' vertical viewing angles are between -y° and y° for x% of time. While the vertical viewing angle theoretically span between −90° and 90°, over 80% of the time users focus on regions that are between −30° and 30° vertically. Users rare look at the 45° vertical angle, in which case our rendering algorithm introduces redundant pixel processing. In essence, our co-design system optimizes for the common case to achieve significant overall energy savings.

6 Conclusions

We expect that a significant amount of video traffic in the near future will be 360° panoramic videos. Mobile system designers will soon face the challenge of guaranteeing desirable user experience while rendering 360° content under severe energy constraints. This paper takes a promising first step in energy-efficient 360° content rendering through a specialized accelerator design. We demonstrate that the key is to tame the irregular memory accesses by co-designing the rendering algorithm with the architecture.

References

[1] [n. d.]. DDR4 Power Calculator 4.0 - Micron Technology, Inc. https://www.micron.com/~/media/documents/products/power-calculator/ddr4_power_calc.xlsm.
[2] [n. d.]. Jetson TX2 Module. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html.
[3] [n. d.]. Magic Leap One Powered by Nvidia Tegra TX2, Available Summer. https://www.tomshardware.com/news/magic-leap-tegra-specs-release,37443.html.
[4] [n. d.]. Micron MT40A256M16GE-083E DDR4 datasheet. https://www.datasheets360.com/pdf/-3972376754899166939.
[5] [n. d.]. TN-40-07: Calculating Memory Power for DDR4 SDRAM. https://www.micron.com/-/media/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf.
[6] [n. d.]. TN-41-01: Calculating Memory System Power for DDR3. https://www.micron.com/-/media/Documents/Products/Technical%20Note/DRAM/TN41_01DDR3_Power.pdf.
[7] [n. d.]. ZCU104 Evaluation Board User Guide. https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.
[8] Robert Anderson, David Gallup, Jonathan T Barron, Janne Kontkanen, Noah Snavely, Carlos Hernández, Sameer Agarwal, and Steven M Seitz. 2016. Jump: virtual reality video. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 198.
[9] Panagiotis Christopoulos Charitos, Hans-Kristian Arntzen, Alberto Duenas, and Daniele Di Donato. 2017. 360-Degree Video Rendering: Using Arm Technology to Implement 360-Degree Video Efficientl. (2017).
[10] Tao Chen and G Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proc. of MICRO*.
[11] Xavier Corbillon, Francesca De Simone, and Gwendal Simon. 2017. 360-degree video head movement dataset. In *Proc. of MMSys*.
[12] Tarek El-Ganainy and Mohamed Hefeeda. 2016. Streaming virtual reality content. *arXiv preprint arXiv:1612.08350* (2016).
[13] Ziyad S Hakura and Anoop Gupta. 1997. The design and analysis of a cache architecture for texture mapping. In *Proc. of ISCA*.
[14] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. 2016. Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction. In *Proc. of HPCA*.
[15] Paul S Heckbert. 1986. Survey of texture mapping. *IEEE computer graphics and applications* 6, 11 (1986), 56–67.
[16] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. In *Proc. of SIGGRAPH*.
[17] Bruce Jacob, Spencer Ng, and David Wang. 2010. *Memory systems: cache, DRAM, disk.* Morgan Kaufmann.
[18] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. 2018. Semantic-Aware Virtual Reality Video Streaming. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*. ACM, 21.
[19] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. 2019. Energy-Efficient Video Processing for Virtual Reality. In *ISCA*.
[20] William M Newman and Robert F Sproull. 1979. *Principles of interactive computer graphics*. McGraw-Hill, Inc.
[21] Benjamin O'Sullivan, Fahad Alam, and Clyde Matava. 2018. Creating Low-Cost 360-Degree Virtual Reality Videos for Hospitals: A Technical Paper on the Dos and Don'ts. *Journal of medical Internet research* 20, 7 (2018).
[22] Xilinx. 2017. AXI4-Stream Interconnect v1.1 LogiCORE IP Product Guide (PG035). (2017).