Received March 23, 2022, accepted April 11, 2022, date of publication April 14, 2022, date of current version April 25, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3167510

Real-Time Lens Distortion Algorithm on an Edge Device With GPU

YOUNG-WOO KIM, HYEON-SEOK YANG, AND DUKSU KIM
School of Computer Engineering, Korea University of Technology and Education (KOREATECH), Cheonan 31253, South Korea

Corresponding author: Duksu Kim (bluekdct@gmail.com)

This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) funded by the Korean Government [Ministry of Science and ICT (MSIT)] through the Morphable Haptic Controller for Manipulating Virtual Reality (VR)·Augmented Reality (AR) Contents 100%under Grant 2020-0-00594; and in part by the Education and Research Promotion Program of Korea University of Technology and Education (KOREATECH), in 2022.

ABSTRACT The lens distortion process is essential for displaying VR contents on a head-mounted display (HMD) with a distorted display surface. This paper proposes a novel lens distortion algorithm to achieve real-time performance on edge devices with an embedded GPU. We employ unified memory space to reduce the data transfer overhead based on an architectural characteristic: an integrated CPU and GPU memory system. The lens distortion kernel is based on the lookup table-based mapping algorithm whose performance is bounded by memory operations rather than computations. To improve the kernel's performance, we propose a compressed lookup table approach that reduces the memory transactions on the kernel while slightly increasing computation. We tested our method on three different edge devices and a desktop system while varying the image resolution from 720p (1,280×720) to 8K (7,680×4.320). Compared with prior GPU-based lookup table algorithms, our method achieved up to 1.72-times higher performance while consuming up to 28.93% less power. Also, our method demonstrates real-time performance for up to a 4K image with a low-end edge device (e.g., 56 FPS on Jetson Nano) and up to an 8K image with a mid-range device (e.g., 94 FPS on Jetson NX). These results demonstrate the benefits of our approach from the perspectives of both performance and energy.

INDEX TERMS Head mounted display, stereoscopic, distortion, GPU, edge device.

I. INTRODUCTION

Virtual reality (VR) is a technique that provides a simulated experience to users, which is widely being used in various applications, including training, education, prototyping, and entertainment [1]. One of the simplest ways to make VR content is converting an existing 2D image (or video) to 3D. Stereoscopy is a technique for creating the illusion of depth in a flat (2D) image using binocular disparity [2]. A stereoscopic image consists of two images for the left and right eyes, and displaying them to each eye is the easiest way to enhance depth perception in the viewer's brain. A head-mounted display (HMD) is the most widely employed device to achieve this effect.

Since the screens in HMDs are placed only a few inches from the user's eyes, an optical lens is placed between them. The optical lens system locates the images at a comfortable distance for the user [3], [4]. It also magnifies the image to provide a reasonable filed-of-view (FOV) [5]. Although the optical lens system has such benefits, it also causes non-linear distortion (e.g., radial distortion) on the image. To correct such distortion, a pre-distortion process on the image is required based on an optical model for HMDs [6], [7]. Since stereoscopic rendering with the pre-distortion process requires a large amount of computation, a separation strategy is needed in most current VR systems; it performs most computations on a PC (or laptop), and the HDM just gets the resulting images while displaying them. This is one of the obstacles in the popularization of VR and HMDs [5].

Edge devices have recently improved their processing power significantly while embedding multi-core CPUs and a GPU (Graphics Processing Unit). Current commodity edge devices have up to six CPU cores and a powerful GPU (e.g., 384 CUDA cores) within a credit card size [8]. With the edge devices' computing power expected to grow continually, designing an algorithm that can efficiently use the

The associate editor coordinating the review of this manuscript and approving it for publication was Massimo Cafaro.

heterogeneous architecture of edge devices will be critical to improving the convenience of using HMDs. It will also be a key to accelerating the popularization of VR systems.

In this work, we propose a novel lens distortion algorithm for edge devices with an embedded GPU. To reduce the data (i.e., input and output images) transfer overhead, we designed our system to employ unified memory space while taking advantage of the integrated CPU and GPU memory architecture of edge devices (Sec. IV-A). Our distortion kernel is based on a lookup table algorithm that significantly reduces the computation on a naive lens distortion method by using pre-computed (i.e., lookup table) values (Sec. IV-B). Since the main idea of the lookup table approach is using a pre-computed value instead of computing the value on the fly, the distortion kernel's performance is bounded by the memory operations while wasting the computational capability of the GPU. We propose a compressed lookup table method that decreases the memory transaction and improves the efficiency of the distortion kernel (Sec. IV-C).

To demonstrate the benefits of our method, we tested its performance on three different edge devices having an embedded GPU and a desktop system having an external GPU for various resolutions of images. We also implemented four alternative methods, including prior lookup table algorithms on the CPU and GPU, and compared their performance with our method (Sec. V). Overall, our method shows the best performance on all the devices independent of the image resolution. It achieved up to 1.72 times (1.35 times on average) higher performance than a prior lookup table algorithm on a GPU. Especially on edge devices, our method shows a meaningful performance improvement over prior lookup table methods on a GPU: about 43.24% on average. As a result, with our method, we can achieve real-time lens distortion performance on up to a 4K (3840×2160) image with a low-end edge device (e.g., 56 FPS with Jetson Nano) and up to an 8K (7680×4320) image with mid-range edge devices (e.g., 94 FPS with Jetson NX). We also compared the power consumption of our method with the prior GPU algorithm and found that ours consumes up to 24.86% less energy. These results demonstrate the efficiency and usefulness of our approach in both computational performance and energy cost.

II. RELATED WORK

Robinett and Rolland [7] pointed out the optical distortion problem and described an optical model for HMDs. Based on their model, they proposed a method of correcting the optical distortion in the HMD by applying an inverse function of the lens distortion to the image on the screen. The lens then restores the predistorted image, and the users see the correct image.

To accelerate the predistortion process, various parallel computing hardware has been employed. Some of the approaches utilize multi-core CPUs [9], but they do not meet real-time performance. Others employ specialized hardware (i.e., FPGA) and achieve real-time performance for VGA output resolution (640×480) [10], [11]. However, the most widely employed approach is using a GPU. Watson and Hodges [12] suggested using graphics hardware for predistorting images. They found that predistortion can be represented as a simple texture mapping onto a 3D polygon reflecting the distorted shape if the undistorted image is a texture. As a result, they achieved up to 10 frames/second (FPS) at a 640×480 resolution with high-end graphics hardware at that time: Silicon Graphics Onyx Reality Engine II. This texture mapping approach has been widely employed for correcting lens distortion in various fields using wide-angle lenses, such as HMDs, the medical domain, and surveillance [13], [14]. Traditionally, texture mapping is realized by shader languages like GLSL (OpenGL shader language), and there are two methods: pixel-based and mesh-based [15]. Pixel-based implementation generates a high-quality result since it computes the distortion coordinates of every pixel in the image. On the other hand, the mesh-based method transforms vertices on a plane mesh according to the distortion equation and fills the other region by interpolating the values for the vertices. Therefore, it can accelerate the distortion process by using a low-resolution mesh; however, its quality also decreases. Shuhua *et al.* [14] employed the pixel-based method and achieved up to 190 times higher performance than the CPU algorithm. We compared the performance of both pixel- and mesh-based texture mapping algorithms with our method in the supplementary report.

Since a lens and the distortion model in a system (e.g., HMD) are fixed, the distortion parameters for each pixel can be reused. Therefore, the texture mapping concept can be extended to build a lookup table (or map) and be used to improve distortion performance [16], [17]. Shehrzad Qureshi [18] introduced lookup table-based lens correction implementation with OpenCL. The introduced method pre-computes two tables for each x- and y-coordinate and generates distortion results by looking at the table. Although the two-table approach is simple to implement, it requires memory transactions twice for processing each pixel. We found that frequent memory access causes performance degradation of GPU algorithms, especially for embedded GPUs on edge devices. To solve this issue, we propose a compressed lookup table approach (Sec. IV).

Lee *et al.* [13] implemented a distortion correction algorithm in a General Purpose GPU (GPGPU) platform with CUDA. They exploited the GPU's hardware-accelerated interpolation ability and achieved a near-real-time performance for HD resolution images (e.g., 48 FPS for 1920×1080 on Nvidia GT555M). However, this method requires additional RGB splitting and mering steps to utilize hardware-accelerated interpolation. Van der Jeught *et al.* [19] also implemented a distortion correction algorithm using a GPU, and their method shows a near-real-time performance (e.g., 30 FPS) for 1024×768 images. Our method also exploits the parallel computing power of a GPU. However, we propose a real-time lens distortion algorithm running on an edge device with a GPU, not on an external GPU equipped on a desktop or laptop.

III. PRELIMINARIES

This section provides the preliminaries for understanding the proposed approach, including the lookup table-based distortion method and the characteristics of edge devices with an embedded GPU.

A. LOOKUP TABLE-BASED LENS DISTORTION ALGORITHM

$$\begin{bmatrix} x_d \\ y_d \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \end{bmatrix} + \begin{bmatrix} x_u - x_c \\ y_u - y_c \end{bmatrix} / (1 + k_1 r^2 + k_2 r^4) \qquad (1)$$

Eq. 1 is the radial distortion model for pre-distorting the input image, where (x_d, y_d) and (x_u, y_u) are the pixel coordinates in the distorted and undistorted images, (x_c, y_c) is the center coordinate of the image, and r is the distance between (x_d, y_d) and (x_c, y_c). k_1 and k_2 are the distortion coefficients, which depend on the target lens. This equation is used to find the matched pixel on the input image (x_u, y_u) for the pixel on the distorted image (x_d, y_d). Since pixels in an image have discrete coordinates, we can pre-compute the $(x_d, y_d) \rightarrow (x_u, y_u)$ pairs for every pixel of the distorted image. A lookup table is the set of all the pairs. Once we have the lookup table for the target lens, we can simplify the distortion process by reading the lookup table and copying the pixel value on the input image to the target position on the distorted image. Therefore, it becomes IO-bounded work, which means the memory operations bounded the performance of the work. Our distortion algorithm is designed based on this lookup table approach. However, the memory access pattern is optimized for the lookup table to improve the distortion performance on the embedded GPU.

B. UNIFIED MEMORY ACCESS IN EDGE DEVICE

In this work, we target edge systems with an embedded GPU. Compared with a general computing system (e.g., PC and workstation) with an external GPU connected over a system bus (e.g., PCIe), an edge system usually integrates a CPU and GPU into a chip while sharing the DRAM [20]. Although each processing unit (i.e., CPU and GPU) in an edge device has its own dedicated memory space on the shared DRAM, it can communicate more efficiently than using physically separated memories, such as an external GPU system. This is realized by specialized memory mapping methods like Nvidia's unified memory [21] and AMD's heterogeneous memory access (hUMA) [22]. The common ground of those methods is mapping the memory region of multiple processing units into unified memory space. Then, every processing unit can access the unified memory space without any explicit memory copy operation. The unified memory space is cached on each processing unit's memory region; therefore, we can avoid the expansive overhead for data copy between the host memory (i.e., system memory) and the device memory (i.e., GPU's DRAM). In our method, we employ unified memory to share input and output images between the CPU and GPU.

IV. LENS DISTORTION ALGORITHM ON AN EDGE DEVICE WITH GPU

A. SYSTEM OVERVIEW

Fig. 1 shows the overview of our lens distortion correction system. The system includes three components: CPU, GPU, and unified memory space. In the preprocessing step, the GPU gets the lens distortion parameters, including the image resolution and distortion coefficients (Eq. 1), from the CPU. Then, the GPU computes the lookup table and stores the result in its device memory. We represent the lookup table as a compressed form to optimize the memory access pattern while considering the characteristics of the distortion algorithm (Sec. IV-C). At runtime, the CPU acts as a manager that obtains input images from an external device (e.g., camera, disk) and orders distortion computation for the images to the GPU. The CPU puts an input image into the unified memory space and calls the GPU kernel (Sec. IV-B). Then, the GPU computes the distorted image of the input image based on the lookup table, and it returns the resulting image into the unified memory space. Finally, the CPU takes the distorted image and outputs it to the target display device (e.g., an HMD or a monitor). We repeat this runtime process until all the input images are processed.

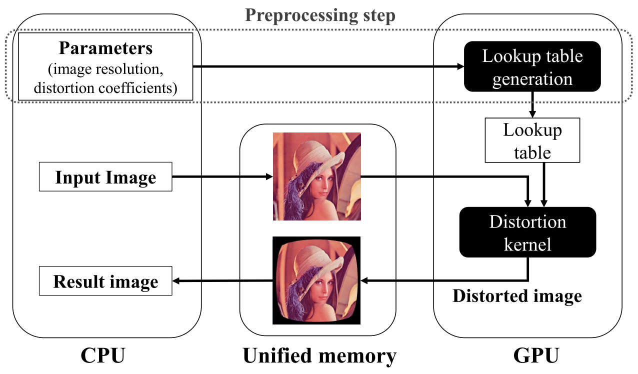


FIGURE 1. Overview of the proposed system.

B. DISTORTION KERNEL ON GPU

The work of the GPU kernel using the lookup table is simply reading the lookup table and copying a pixel from the input image to the output image. Since all pixels are independent of each other, the GPU kernel's simple but efficient thread layout is allocating a thread per pixel. The lookup table can be used for forward and backward mapping, which determines the thread layout of the GPU kernel. Forward mapping gets the target pixel index on a distorted image from the input (undistorted) pixel index (i.e., $(x_u, y_u) \rightarrow (x_d, y_d)$). In this case, each GPU thread handles a pixel on the input image. In the reverse direction, backward mapping obtains the pixel index on an input image from the pixel index of an output image (i.e., $(x_d, y_d) \rightarrow (x_u, y_u)$). For this backward direction, the GPU kernel allocates a thread per pixel on the output image.

We found that backward mapping achieves about 13.91% higher performance on average than forward mapping. The forward mapping kernel requires memory copies equivalent to the number of pixels on the input image. On the other hand, backward mapping requires fewer memory copies (e.g., 22.21% for 1920 × 1080 resolution) than forward mapping because some regions on the distorted image are blank. Also, forward mapping leads to more serialized memory access than backward mapping because the pixel index is discrete, and multiple pixels on the input image can be mapped to the same pixel on the output image. Therefore, we use the backward mapping approach on the GPU distortion kernel.

C. OPTIMIZING MEMORY ACCESS WITH A COMPRESSED LOOKUP TABLE

A straightforward form for the lookup table is a two-dimensional array of a structure consisting of two integer values for the x- and y-coordinates. However, such an array of structure (AoS) leads to uncoalesced memory access, which means threads in a warp read (or write to) a non-contiguous memory region [21]. Uncoalesced memory access requires more memory transactions than coalesced memory access. To improve lookup table access efficiency, Shehrzad Qureshi [18] employed an SoA (Structure of Array)-style lookup table that consists of two separate arrays for the x- and y-coordinates.

To improve memory access efficiency even further than Shehrzad Qureshi [18]'s approach, we propose a compressed lookup table that halves memory transactions for reading the lookup table. Instead of using two separate arrays, we encode x- and y-coordinates into an integer value. Eq. 2 is our encoding equation where (x, y) is the input coordinates, $c_{x,y}$ is the compressed value, and $(width, height)$ is the image resolution.

$$c_{x,y} = \begin{cases} x * width + y, & if\ width \geq height \\ y * height + x, & otherwise \end{cases} \tag{2}$$

The key to our compression method is the image resolution. Since the coordinate for the shorter axis cannot exceed the resolution of the longer axis, we can decode (or decompress) the $c_{x,y}$ into (x, y) coordinates with one division and one modular operation, as shown in Eq. 3.

$$(x, y) = \begin{cases} (c_{x,y}/width,\ \ c_{x,y}\ mod\ width), & \\ & if\ width \geq height \\ (c_{x,y}\ mod\ height,\ \ c_{x,y}/height), & otherwise \end{cases} \tag{3}$$

Our compressed lookup table can obtain the target coordinate with one read operation and threads in a warp access continuous memory region. As a result, we can halve the memory transaction while accessing the memory efficiently by coalesced memory access. Even though it requires additional computation to decode the value (i.e., $c_{x,y}$) instead of memory transaction, it is beneficial to the performance of the distortion process. GPU cores can handle other threads during the memory transaction latency of a set of threads (e.g., warp) without penalty based on the GPU's zero context-switching overhead property [23]. Therefore, replacing some memory operations of the I/O-bounded kernel with computational tasks improves the utilization efficiency of resources in the GPU. With the two separated (uncompressed) lookup tables for the x- and y-coordinates, the main job of the distortion kernel is memory access. It is an I/O-bounded kernel, and the computing units (i.e., GPU cores) are idle most of the time; therefore, the compressed lookup table improves the balance between memory transactions and computation tasks, improving the distortion kernel's performance. By profiling the reasons for stalling threads during kernel processing with Nvidia Nsight Compute [24], we found that the ratio of stalls by memory latency is reduced up to 31.12% (29.57% on average) with the compressed lookup table. It also improves the kernel performance up to 1.72 times (1.35 times on average) more than using an uncompressed lookup table.

V. RESULTS AND ANALYSIS

We implemented our lens distortion algorithm on four systems, including three edge devices having different computing powers and a desktop system having an external GPU (Table 1). We used CUDA 11.0 for the desktop and 10.2 for edge devices. We implemented two versions of our method to discern the effects of the mapping direction (i.e., forward and backward).

- $Ours_B$ is an implementation of our algorithm with backward mapping that uses the compressed lookup table (Sec. IV-C). The distortion kernel allocates a thread per pixel on the output image.
- $Ours_F$ is the forward mapping version of our algorithm. In this algorithm, each GPU thread handles a pixel on the input image.

To analyze the benefits of our approach, we also implemented four alternative algorithms based on prior approaches. There are two categories of alternative methods. The first group includes two CPU-based algorithms.

- CPU_{naive} is an implementation of the lens distortion process without pre-computation (i.e., lookup table). This method computes the target pixel according to the distortion equation (Eq. 1) for each pixel. We used the same number of threads as the number of CPU cores in the system (e.g., four-thread for Jetson Nano and six threads for Jetson NX).
- CPU_{LUT} is a CPU implementation that employs the lookup table approach. This method uses a pre-computed lookup table for backward mapping. At runtime, it finds the target pixel by using the lookup table instead of calculating it on the fly. This algorithm also used the same number of CPU threads with the number of cores in the system.

The second group of alternative algorithms are prior GPGPU algorithms.

- GPU_{naive} is a CUDA-based version of CPU_{naive}. This method launches as many threads as the output image resolution. We tested various thread layouts and found that the (16 × 16) thread block generally performs better

TABLE 1. System specification of three edge devices and a desktop PC used in our experiments. The asterisk(*) means the system shared memory. The bold font in the power row marks the corresponding power mode we used for the experiments.

	Jetson Nano	Jetson NX	Jetson AGX Xavier	RTX 2060s
CPU architecture	ARM	ARM	ARM	AMD Ryzen 5
# of cores (Clock)	4 (1.43 GHz)	6 (1.4 GHz)	8 (2.26 GHz)	6 (3.6 GHz)
GPU architecture	Nvidia Maxwell	Nvidia Volta	Nvidia Volta	Nvidia Turing
# of GPU cores	128	384	512	2176
Memory size	4 GB*	8 GB*	32 GB*	8 GB
Memory type	LPDDR4	LPDDR4x	LPDDR4x	GDDR6
Memory bandwidth	25.6 GB/s	51.2 GB/s	137 GB/s	448 GB/s
Power	5W / **10W**	**10W** / 15W	10W / 15W / **30W**	**175W**
Size	70×45 mm	103×90.5 mm	105×105 mm	225×128 mm (GPU only)

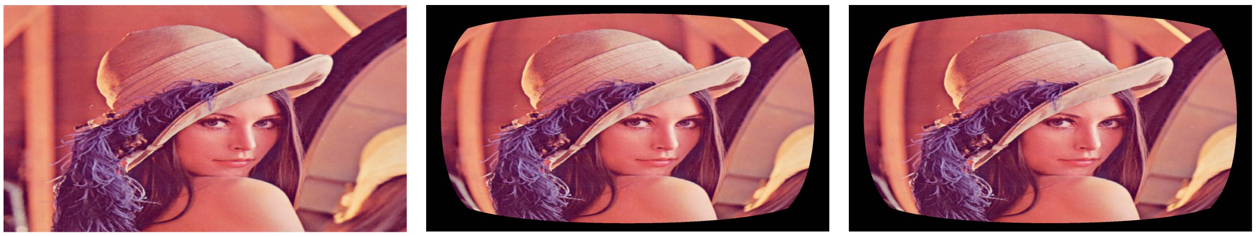


FIGURE 2. This figure shows the input image and the distortion results by the naive algorithm and ours.

than other layouts. Therefore, we used this thread block size in our experiments.

- *GPU_{LUT}* is an implementation of the lookup table distortion algorithm. Following Shehrzad Qureshi [18], we built two separate tables of x- and y-coordinates for backward mapping in the device memory. The thread layout and the block size are the same as GPU_{naive}.

We employed the unified memory approach for implementing all the GPU algorithms, including GPU_{navie}, GPU_{LUT}, $Ours_F$, and $Ours_B$. However, for the external GPU system (i.e., RTX 2060s in Table 1), we used explicit memory copy between the host and device memories instead of unified memory. We found that the explicit version shows about 79.01 times (70.47 times on average) better performance for the RTX 2060s system.

We used the Lenna (Fig. 2) image and varied the image resolution from 1280 × 720 (i.e., 720p) to 7680 × 4320 (i.e., 8K) and checked the performance of the six algorithms. We performed the distortion process 100 times for each resolution and reported the average processing time. It should be noted that the lens distortion performance is affected by the image resolution, not by the contents (e.g., colors) on the image.

A. RESULTS

Table 2 shows the processing time of the six different algorithms for lens distortion computation on different devices and various resolutions. Since GPU algorithms on edge devices use unified memory space, there is no explicit memory copy between the host and device memories. Therefore, the kernel processing time includes the memory transaction time. Unlike edge devices, an external GPU system requires explicit data copy between the host and device memories for an external GPU (i.e., RTX 2060s system). We measured all memory operation times and computation times. The parentheses for RTX 2060s in Table 2 show the time only for the kernel.

The lookup table algorithms on both CPU and GPU show better performance than the naive approach. CPU_{LUT} and GPU_{LUT} achieved up to 8.82 and 67.16 times higher performance than CPU_{navie} and GPU_{navie}, respectively. Also, GPU_{LUT} shows up to 10.51 times (4.60 times on average) higher performance than CPU_{LUT}. These results validate the benefit of the lookup table algorithm and that the lookup table approach is appropriate for GPU architecture.

Both our mapping algorithms with a compressed lookup table achieved the best performance in all cases. Fig. 3 compares the performance of three lookup table-based GPU algorithms, including GPU_{LUT}, $Ours_F$, and $Ours_B$. Between our two methods, $Ours_B$ generally shows better performance (e.g., 13.91% on average), as discussed in Sec. IV-B. $Ours_F$ achieved up to 1.10, 1.40, 1.27, and 1.22 times (1.08, 1.37, 1.23, and 1.09 times on average) over GPU_{LUT}, and $Ours_B$ improved the lens distortion performance up to 1.26, 1.71, 1.72, and 1.22 times (1.20, 1.53, 1.57, and 1.10 times on average) over GPU_{LUT}. This performance improvement of $Ours_B$ over GPU_{LUT} validates the benefits of our compressed lookup table approach (Sec. V-B). An interesting result is that our methods achieved better performance on edge devices

TABLE 2. This table shows the average processing time (milliseconds) of lens distortion per frame. For RTX 2060s, the parentheses show the time only for the kernel. The bold font marks the best performance among the eight algorithms.

Resolution	720p (1280x720)				1080p (1920x1080)			
Device	Nano	NX	AGX	RTX 2060s	Nano	NX	AGX	RTX 2060s
CPU_{naive}	23.48	9.13	18.55	12.73	66.56	10.60	24.92	14.65
CPU_{LUT}	5.89	5.14	3.61	4.76	8.27	7.59	4.59	6.82
GPU_{naive}	71.06	32.20	28.81	0.75 (0.32)	159.23	74.43	64.99	1.66 (0.69)
GPU_{LUT}	2.55	0.49	0.47	0.6 (0.08)	5.46	1.11	1.09	1.13 (0.16)
$Ours_F$	2.31	**0.35**	0.37	**0.49 (0.05)**	5.20	0.82	0.90	1.07 (0.10)
$Ours_B$	**2.15**	0.38	**0.34**	0.49 (0.06)	**4.79**	**0.77**	**0.70**	**1.07 (0.10)**
Resolution	**4K (3840x2160)**				**8K (7680x4320)**			
Device	**Nano**	**NX**	**AGX**	**RTX 2060s**	**Nano**	**NX**	**AGX**	**RTX 2060s**
CPU_{naive}	192.23	28.52	26.10	22.26	1034.55	86.15	81.12	377.61
CPU_{LUT}	34.45	21.56	21.30	12.08	117.33	57.41	55.79	88.90
GPU_{naive}	636.14	290.18	260.18	5.91 (2.08)	2560.14	1147.75	1023.52	23.82 (8.57)
GPU_{LUT}	21.57	4.59	4.50	4.36 (0.53)	89.36	18.12	17.66	17.44 (2.19)
$Ours_F$	20.30	3.40	3.78	4.18 (0.35)	81.98	13.16	14.22	16.71 (1.47)
$Ours_B$	**17.89**	**2.76**	**2.73**	**4.15 (0.33)**	**71.10**	**10.60**	**10.28**	**16.59 (1.35)**

TABLE 3. This table shows the top seven reasons and related operations for stalling a warp. The last four columns show the number of the stalled cycles by each reason for GPU_{LUT} and $Ours_B$. We used an external GPU system (i.e., RTX 2060s) for this experiment. The reported data is the average value measured for every resolution. A detailed explanation of the reasons for each stall is available in Chapter 4 of the kernel profiling guide [25].

Reason for stall	Related operations	GPU_{LUT}	Ratio	$Ours_B$	Ratio
Long Scoreboard	L1TEX(local, global, surface, tex) data access	24.58	0.7381	20.73	0.8820
Short Scoreboard	MIO(memory input/output) operation	1.8	0.0551	1.19	0.0504
LG Throttle	Local and global memory operation	5.65	0.1697	1.20	0.0509
IMC Miss	Immediate constant cache miss	0.13	0.0038	0.12	0.0051
MIO Throttle	MIO(memory input/output) operation	0.87	0.0262	0.06	0.0027
Drain	All memory operations	0.10	0.0031	0.04	0.0017
Math Pipe Throttle	Arithmetic operation	0.14	0.0041	0.17	0.0071
Total		33.30	1.0000	23.51	1.0000

(except the low-end one, Jetson Nano) than on the external GPU (i.e., the RTX 2060s system) with a higher computational capability. We found that the kernel processing time is much faster on the external GPU than on edge devices. However, the data communication between host and device memories on the external GPU system takes more time than the kernel processing time on edge devices, which use unified memory (Sec. V-C). Overall, with our method, we could achieve real-time lens distortion performance on up to a 4K image with a low-end edge device (e.g., 56 FPS on Nano) and up to an 8K image (e.g., 94 FPS on NX) on mid-range devices.

B. BENEFIT OF COMPRESSED LOOKUP TABLE

To ascertain how our compressed lookup table approach ($Ours_B$) improves the performance over GPU_{LUT}, we profiled the kernels of the two methods by using the Nvidia Nsight Compute [24]. We investigated the reasons for stalling a warp on each kernel, and Table 3 summarizes the results. The math pipe throttle occurs when all the math pipes are busy, which means the computational task causes this stall. On the other hand, the other six reasons are related to memory operation. As shown in Table 3, memory operations dominate the performance of both kernels. This result is consistent with the characteristics of the mapping kernel, on which the main work is reading the lookup table and copying pixels from an input image to an output image.

We found that our compressed lookup table approach ($Ours_B$) reduces the total stalls by about 30% compared with GPU_{LUT}. This is because our method needs just one read operation to the compressed lookup table, unlike GPU_{LUT}, which requires two memory reads for the x- and y-coordinates of tables. Although the stall by math pipe throttle is slightly increased (about 1.2 times) due to the decompression process, it still takes less than 1% of total stalls. Therefore, the benefit of using a compressed lookup table (i.e., reducing stalls by memory operations) is overwhelming in the increment of computational overhead. These results demonstrate the efficiency of our compressed lookup table approach.

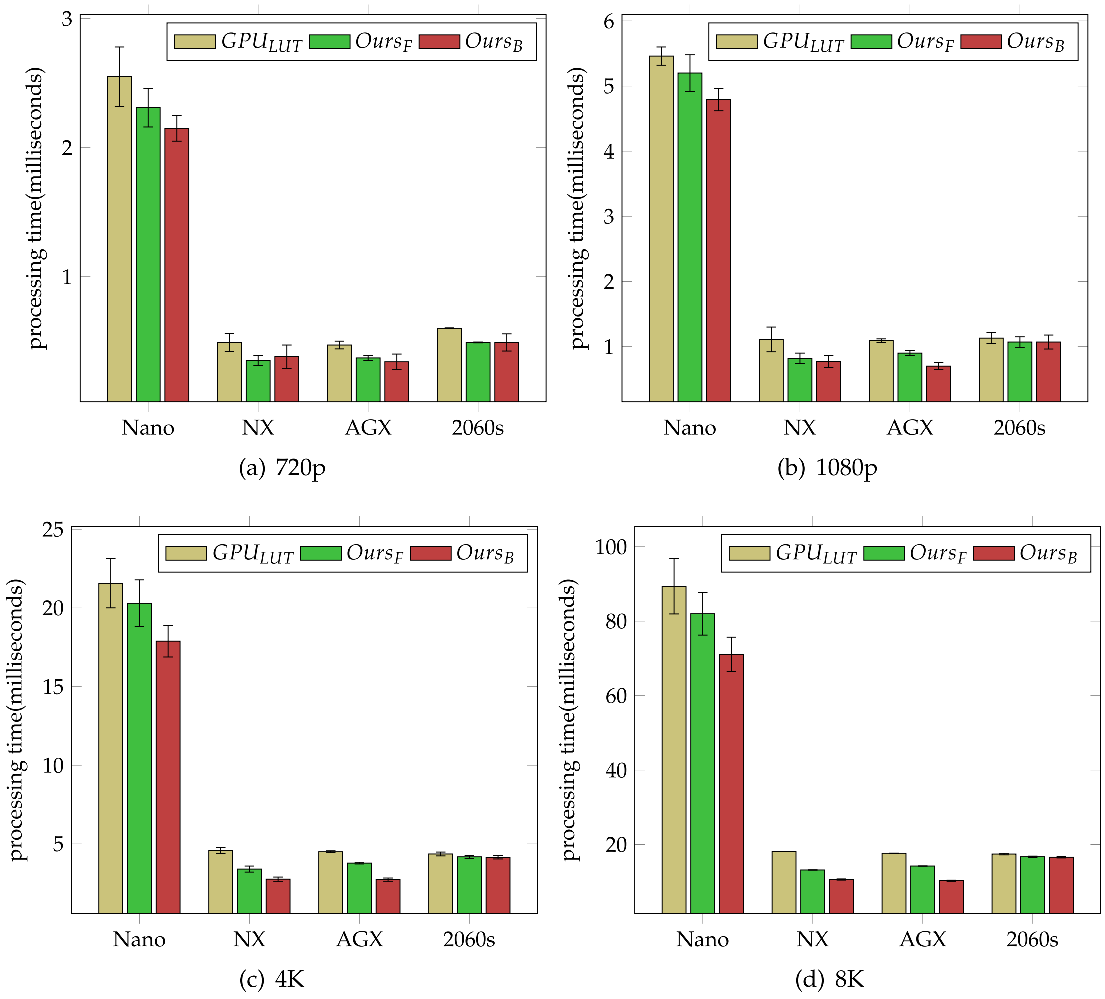


FIGURE 3. These graphs show the average processing time with a standard deviation of three GPU algorithms that use the lookup table approach.

TABLE 4. This table shows the processing times for each task on $Ours_B$ and $OursExp_B$. Copy(H→D) and Copy(D→H) denote the explicit data copy between host and device memories. Since $Ours_B$ uses unified memory space, there is no explicit data transfer while the kernel time includes all the data communication time.

Resolution		1080p				4K				8K			
Device		Nano	NX	AGX	2060s	Nano	NX	AGX	2060s	Nano	NX	AGX	2060s
$Ours_B$	Kernel	**4.79**	**0.77**	**0.70**	**78.15**	**17.89**	**2.76**	**2.73**	**274.66**	**71.10**	**10.60**	**10.28**	**1053.69**
$OursExp_B$	Copy(H→D)	0.71	0.30	0.38	0.49	3.03	1.08	1.34	1.91	16.86	4.13	5.25	7.62
	Kernel	4.73	0.61	0.68	0.10	17.56	2.23	2.61	0.33	69.63	8.89	10.20	1.35
	Copy(D→H)	0.71	0.30	0.38	0.49	3.03	1.08	1.34	1.91	16.86	4.13	5.25	7.62
	Total	**6.14**	**1.22**	**1.43**	**1.07**	**23.62**	**4.40**	**5.29**	**4.15**	**103.35**	**17.15**	**20.70**	**16.59**

C. EFFECT OF UNIFIED MEMORY

To check the effect of using unified memory on edge devices, we implemented an alternative version of our method ($OursExp_B$) that explicitly transfers the input and output images between the host and device memories with memory copy APIs (e.g., *cudaMemcpyAsync()*). We optimized the data transfer with pinned-memory (or page-locked memory) [21] for $OursExp_B$. Table 4 shows the processing time of the two versions of our method. The kernel of $OursExp_B$ takes slightly less time than the kernel processing time for $Ours_B$ because all the required data is in the dedicated device memory for $OursExp_B$, while $Ours_B$ requires access to the unified memory space. However, $OursExp_B$ requires separate data copy time between the host and device memories (e.g., Copy(H→D) and Copy(D→H) in Table 4) while kernel processing time of $Ours_B$ already includes all the

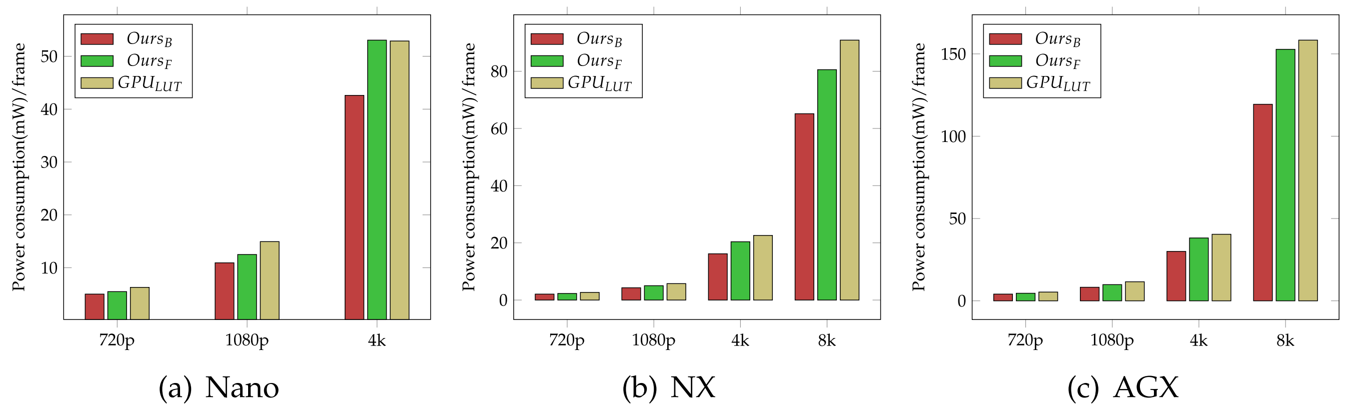


FIGURE 4. These graphs compare the power consumption of three GPU algorithms on each edge devices for various resolution.

data communication time. Overall, $Ours_B$ shows up to 1.45, 1.62, and 2.37 times (1.34, 1.60, and 2.09 times on average) higher performance than $OursExp_B$. This result validates the advantage of using unified memory instead of explicit data transfer on edge devices.

Unlike on edge devices having shared memory architecture, for a desktop environment with an external GPU (i.e., RTX 2060s system), using unified memory ($Ours_B$) leads to greatly reduced performance compared to performing explicit memory copy ($OursExp_B$). Since host memory and device memory are physically separated on an external GPU system, it should use a PCI bus for every access to the unified memory space. The bandwidth of PCI bus (e.g., 16GB/s for PCIe 3.0 x16) is much slower than the bandwidth of device memory (e.g., 448GB/s), and it is hard to exploit the peak bandwidth with frequent small-size data transfers.

We found that the RTX 2060s shows much higher performance (e.g., up to 54.22 times) than edge devices only for kernel processing time. However, the data communication time is much larger than the kernel processing time and the processing time of $Ours_B$ on edge devices except for the low-end Jetson Nano device. Therefore, employing unified memory is not suitable for external GPU systems.

As a result, we achieved better lens distortion performance with edge devices (e.g., Jetson NX and AGX Xavier) than using a powerful external GPU with the $Ours_B$ algorithm. These results demonstrate the efficiency and suitability of our method for edge devices.

D. POWER CONSUMPTION

For a wireless system like wireless HMD, power consumption is one of the critical factors since it determines the maximum usage time. To check the advantage of our approach in energy efficiency, we measured the power consumption of three GPU algorithms on each edge device. Jetson boards include the INA3221 power monitor module(s), and we can read the monitoring information via Linux sysfs [26]. We implemented a power-consumption-measuring software based on the NVIDIA Jetson board support package (BSP) and read the information on INA3221 every 15 milliseconds. We ran the power measuring thread concurrently with the thread handling the distortion process and synchronized at

TABLE 5. This table shows the power consumption of each algorithm on different edge devices. We measured total power consumption for processing 5,000 times, and the reported values are the average power consumption per frame. Since Jetson NX does not provide CPU and GPU power consumption separately, we report a single value for the device.

Device	Res.	Algorithm	Power consumption(mW)		
			CPU	GPU	Total
Nano	720p	$Ours_B$	0.65	4.36	5.01
		$Ours_F$	0.73	4.75	5.48
		GPU_{LUT}	0.90	5.38	6.28
	1080p	$Ours_B$	1.21	9.72	10.93
		$Ours_F$	1.43	11.06	12.50
		GPU_{LUT}	2.96	11.97	14.94
	4k	$Ours_B$	5.76	36.83	42.60
		$Ours_F$	7.40	45.62	53.05
		GPU_{LUT}	6.22	45.94	52.89
NX	720p	$Ours_B$	1.24		2.07
		$Ours_F$	1.42		2.29
		GPU_{LUT}	1.61		2.66
	1080p	$Ours_B$	2.71		4.29
		$Ours_F$	3.20		5.01
		GPU_{LUT}	3.56		5.74
	4k	$Ours_B$	10.35		16.17
		$Ours_F$	13.24		20.38
		GPU_{LUT}	14.19		22.58
	8k	$Ours_B$	41.83		65.15
		$Ours_F$	52.72		80.55
		GPU_{LUT}	57.32		90.88
AGX	720p	$Ours_B$	0.43	1.52	4.08
		$Ours_F$	0.49	1.78	4.59
		GPU_{LUT}	0.87	1.95	5.32
	1080p	$Ours_B$	0.80	3.44	8.23
		$Ours_F$	0.81	4.25	9.84
		GPU_{LUT}	1.64	4.59	11.58
	4k	$Ours_B$	1.58	13.71	30.01
		$Ours_F$	1.87	17.89	38.19
		GPU_{LUT}	2.11	18.41	40.42
	8k	$Ours_B$	3.99	56.36	119.38
		$Ours_F$	6.29	73.18	152.77
		GPU_{LUT}	8.10	69.15	158.35

every execution. We ran each algorithm 5,000 times to have sufficient time for measuring power consumption accurately, and Table 5 reports the average power consumption for handling an image (or frame).

Fig. 4 compares the power consumption of three GPU algorithms on edge devices. $Ours_F$ consumes up to 16.32% (10.37% on average) less power than GPU_{LUT}. Our compressed lookup table approach halves the read operations for accessing the lookup table than GPU_{LUT} (Sec. IV-C), and it reduces not only the processing time but also power consumption. Also, $Ours_B$ achieved up to 1.28 times better power efficiency than $Ours_F$. As we inspired in Sec. IV-B, the backward mapping ($Ours_B$) approach requires fewer (e.g., 22.21%) memory copies than the forward mapping ($Ours_F$). Therefore, $Ours_B$ achieved much higher power efficiency than $Ours_F$. Overall, $Ours_B$ reduces power usage up to 28.93% (24.86% on average) than GPU_{LUT}. These results demonstrate the advantage of our method over the prior method on power efficiency.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented an efficient lens distortion algorithm for edge devices having an embedded GPU. We employed unified memory space to take advantage of the edge device's integrated memory architecture and optimized data transfer between CPU and GPU. We designed our distortion kernel based on backward mapping with a pre-computed lookup table. We then discovered that the performance of the lookup table method is bounded by memory operations while wasting the computational capability of the GPU. To improve the efficiency of the distortion kernel, we proposed the compressed lookup table approach. This approach balances the workload of memory operations and arithmetic computation, and it improves the kernel's performance. As a result, our method shows up to 1.72-times higher performance compared with a prior GPU-based lookup table algorithm (i.e., GPU_{LUT}). Also, our method demonstrates real-time performance for high-resolution images using low-end and mid-range edge devices. Furthermore, we found that our approach consumes less power (e.g., up to 28.93%) than GPU_{LUT}. These results demonstrate the benefits of our method from the perspective of both processing performance and energy efficiency, and it validates its suitability for edge devices.

Although our method achieved real-time performance for up to 4K resolution with low-end edge devices, future HMD will require a much higher resolution for a more immersive experience in the virtual world. As future work, we would like to make our algorithm meet real-time performance for ultra-high-resolution contents (e.g., 8K) even with low-end edge devices. To achieve this, we plan to design a heterogeneous parallel lens distortion algorithm, which fully utilizes both multi-core CPUs and GPU.

REFERENCES

[1] W. R. Sherman and A. B. Craig, *Understanding Virtual Reality*. San Francisco, CA, USA: Morgan Kaufmann, 2003.

[2] M. Mon-Williams, J. P. Warm, and S. Rushton, "Binocular vision in a virtual world: Visual deficits following the wearing of a head-mounted display," *Ophthalmic Physiol. Opt.*, vol. 13, no. 4, pp. 387–391, 1993.

[3] I. E. Sutherland, "A head-mounted three dimensional display," in *Proc. Fall Joint Comput. Conf., I*, 1968, pp. 757–764.

[4] E. M. Howlett, "Wide-angle orthostereo," in *Stereoscopic Displays and Applications*, vol. 1256, Int. Soc. Opt. Photon., 1990, pp. 210–223.

[5] C. Anthes, R. J. García-Hernández, M. Wiedemann, and D. Kranzlmüller, "State of the art of virtual reality technology," in *Proc. IEEE Aerosp. Conf.*, Mar. 2016, pp. 1–19.

[6] E. M. Howlett, "Wide angle color photography method and system," U.S. Patent 4 406 532, Sep. 27, 1983.

[7] W. Robinett and J. P. Rolland, "A computational model for the stereoscopic optics of a head-mounted display," *Presence, Teleoperators Virtual Environ.*, vol. 1, no. 1, pp. 45–62, Jan. 1992.

[8] NVIDIA. *Jetson Xavier NX for Embedded & Edge Systems*. Accessed: Oct. 24, 2021. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/

[9] K. Daloukas, C. D. Antonopoulos, N. Bellas, and S. M. Chai, "Fisheye lens distortion correction on multicore and hardware accelerator platforms," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2010, pp. 1–10.

[10] K. Gribbon, C. Johnston, and D. G. Bailey, "A real-time FPGA implementation of a barrel distortion correction algorithm with bilinear interpolation," in *Proc. Image Vis. Comput. New Zealand*, 2003, pp. 408–413.

[11] N. Bellas, S. M. Chai, M. Dwyer, and D. Linzmeier, "Real-time fisheye lens distortion correction using automatically generated streaming accelerators," in *Proc. 17th IEEE Symp. Field Program. Custom Comput. Mach.*, Apr. 2009, pp. 149–156.

[12] B. A. Watson and L. F. Hodges, "Using texture maps to correct for optical distortion in head-mounted displays," in *Proc. Virtual Reality Annu. Int. Symp.*, 1995, pp. 172–178.

[13] T.-Y. Lee, C.-H. Wei, S.-H. Lai, and R.-R. Lee, "Real-time correction of wide-angle lens distortion for images with GPU computing," in *Proc. IEEE Asia Pacific Conf. Circuits Syst.*, Dec. 2012, pp. 456–459.

[14] L. Shuhua and Z. Jun, "GPU-based barrel distortion correction for acceleration," in *Proc. IEEE 10th Int. Conf. High Perform. Comput. Commun.; IEEE Int. Conf. Embedded Ubiquitous Comput.*, Nov. 2013, pp. 845–848.

[15] C. Pötzsch. (2016). *Speeding Up GPU Barrel Distortion Correction in Mobile VR*. Accessed: Jul. 8, 2021. [Online]. Available: https://www.imaginationtech.com/blog/speeding-up-gpu-barrel-distortion-correction-in-mobile-vr

[16] J. P. Helferty, C. Zhang, G. McLennan, and W. E. Higgins, "Videoendoscopic distortion correction and its application to virtual guidance of endoscopy," *IEEE Trans. Med. Imag.*, vol. 20, no. 7, pp. 605–617, Jul. 2001.

[17] R. Shahidi, M. R. Bax, C. R. Maurer, J. A. Johnson, E. P. Wilkinson, B. Wang, J. B. West, M. J. Citardi, K. H. Manwaring, and R. Khadem, "Implementation, calibration and accuracy testing of an image-enhanced endoscopy system," *IEEE Trans. Med. Imag.*, vol. 21, no. 12, pp. 1524–1535, Dec. 2002.

[18] S. Qureshi. (Jun. 2013). Computer vision acceleration using GPUs. AMD Developer Central. Accessed: Jul. 8, 2021. [Online]. Available: http://developer.amd.com/wordpress/media/2013/06/2162_final.pdf

[19] S. Van der Jeught, "Real-time geometric lens distortion correction using a graphics processing unit," *Opt. Eng.*, vol. 51, no. 2, Feb. 2012, Art. no. 027002.

[20] NVIDIA. *CUDA for Tegra: CUDA Toolkit Documentation*. Accessed: Jul. 12, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

[21] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. Hoboken, NJ, USA: Wiley, 2014.

[22] H. Chu, "AMD heterogeneous uniform memory access," in *Proc. APU 13th Developer Summit*. Santa Clara, CF, U.S., 2013, pp. 11–13.

[23] *CUDA Programming Guide 9.2*, NVIDIA, Santa Clara, CA, USA, 2018.

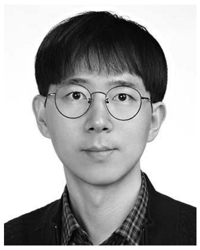
[24] NVIDIA. *NVIDIA Nsight Compute*. Accessed: Oct. 22, 2021. [Online]. Available: https://developer.nvidia.com/nsight-compute

[25] NVIDIA. *NVIDIA Kernel Profiling Guide*. Accessed: Oct. 22, 2021. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#sampling

[26] NVIDIA. *NVIDIA Jetson Linux Driver Package Software Features*. Accessed: Feb. 18, 2022. [Online]. Available: https://docs.nvidia.com/jetson/l4t/index.html

YOUNG-WOO KIM received the B.S. degree from Korea University of Technology and Education (KOREATECH), South Korea, in 2020, where he is currently pursuing the master's degree with the HPC Laboratory, Institute of Computer Science and Engineering. His research interests include VR, 3D reconstruction, and parallel computing.

HYEON-SEOK YANG received the B.S. degree from Korea University of Technology and Education (KOREATECH), South Korea, in 2020. His research interests include parallel computing and ML.



DUKSU KIM received the B.S. degree from Sungkyunkwan University, in 2008, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2014. He is currently an Assistant Professor with the School of Computer Engineering, Korea University of Technology and Education (KOREATECH). He spent several years as a Senior Researcher with the KISTI National Supercomputing Center. His research interests include designing heterogeneous parallel computing algorithms for various applications, including proximity computation, scientific visualization, and machine learning. Some of his work received the Distinguished Paper Award at Pacific Graphics, in 2009, and an ACM Student Research Competition Award, in 2009, and was selected as the Spotlight Paper for the September Issue of IEEE Transactions on Visualization and Computer Graphics, in 2013. He is a Young Professional Member of IEEE and a Professional Member of ACM.