Analysis of Implementing Mobile Heterogeneous Computing for Image Sequence Processing

Aram BAEK[1], Kangwoon LEE[1], Jae-Gon KIM[2], and Haechul CHOI[1*]
[1] The Department of Multimedia Engineering,
Hanbat National University, Daejeon, Korea
[e-mail: aram98123@naver.com, kkawoons@naver.com, and choihc@hanbat.ac.kr]
[2] School of Electronics, Telecommunication, and Computer Engineering,
Korea Aerospace University, Gyonggi-do, Korea
[e-mail: jgkim@kau.ac.kr]
*Corresponding author : Haechul CHOI

Received April 13, 2017; accepted June 17, 2017; published October 31, 2017

Abstract

On mobile devices, image sequences are widely used for multimedia applications such as computer vision, video enhancement, and augmented reality. However, the real-time processing of mobile devices is still a challenge because of constraints and demands for higher resolution images. Recently, heterogeneous computing methods that utilize both a central processing unit (CPU) and a graphics processing unit (GPU) have been researched to accelerate the image sequence processing. This paper deals with various optimizing techniques such as parallel processing by the CPU and GPU, distributed processing on the CPU, frame buffer object, and double buffering for parallel and/or distributed tasks. Using the optimizing techniques both individually and combined, several heterogeneous computing structures were implemented and their effectiveness were analyzed. The experimental results show that the heterogeneous computing facilitates executions up to 3.5 times faster than CPU-only processing.

Keywords: Image processing, mobile platform, heterogeneous computing, embedded GPU, GPGPU.

https://doi.org/10.3837/tiis.2017.10.014 **ISSN : 1976-7277**

1. Introduction

Mobile devices such as smartphones and tablets have been rapidly evolving with high performance built-in cameras and remarkable high-speed communication, giving rise to a variety of multimedia applications. Many of the multimedia applications such as computer vision, video enhancement, and augmented reality utilize image sequences processing technologies that refer to methods for the digital processing of a set of images. These technologies require significant amounts of computation because image sequences have a large amount of data. However, mobile platforms are still subject to constraints such as battery capacity and computational power. The battery supplies a limited power that remains a key factor in determination of computability. In addition, the mobile platform lacks floating point units and random access memory (RAM), which significantly reduces arithmetic accuracy and makes it difficult to handle high resolution video, respectively.

In an effort to enable better image sequence processing, general-purpose computation on graphics processing units (GPGPU) [1] on the mobile platform has been actively researched. A graphics processing unit (GPU) enables parallel processing with multiple cores, which leads to significant improvement in energy consumption efficiency and execution time speedup. In addition to the GPU, image sequence processing also needs a central processing unit (CPU), through which the GPGPU performs, the operating system runs, traditional serial tasks are conducted, and images to be stored or streamed are entered as input for processing. A system with two or more dissimilar processors such as CPU and GPU is referred to as a heterogeneous computing system. This approach further improves image sequence processing on the mobile platform.

To implement image sequence processing based on heterogeneous computing on the mobile platform, the following factors have to be considered: idle time between working schedules of the processors, format conversion to enable operable data on the other processor, available application memory supported by the mobile OS, data transmission between processors, and balancing of the differing computational power of the processors. On the basis of these considerations, this paper deals with several optimizing techniques including task distribution on the CPU side, double buffering, frame buffer object (FBO), and parallelization of the CPU and GPU. Applying the techniques both individually and combined, various structures: CPU-only (CO), CPU-GPU sequential (CGS), CPU-GPU parallel (CGP), distributed CPU-GPU sequential (DCGS), and distributed CPU-GPU parallel (DCGP) image sequence processing structures, are implemented and analyzed. In the CO structure, all jobs are conducted on the CPU. In the CGS structure, two kinds of tasks are distributed to each processor and are conducted sequentially for a frame. The CGP is an improvement on the CGS in which tasks are performed in parallel. For example, the n-th and (n+1)-th frames are processed on the GPU and the CPU, respectively. In the DCGS and the DCGP structures, the CPU task is divided into multiple modules and the modules are performed in parallel on the CPU. Experiments conducted, in which these

Table 1. Conventional mobile GPU based works for image processing and their performances relative to CPU-only computing.

Reference	Use Case	API	Source Type	Image Resolution	Device	Speedup
Ensor and Hall [7]	Canny edge detection	OpenGL ES 2.0	Image	640 × 480	Nexus One, iPhone 4, etc.	0.4x–2.4x
Singhal et al. [8]	SURF, Image processing	OpenGL ES 2.0	Image	800 × 480	ARM Cortex A8 (1 GHz) and PowerVR SGX 540 (200 MHz)	1.68x –6.98x
Wang et al. [9]	Visual object removal	OpenCL	Video	512 × 384	Qualcomm Snapdragon S4	-
Pulli et al. [10]	Image processing	OpenCV	Image	10,000 iterations on GPU	Tegra 3	2.4x–14x
Leskela et al. [12]	Image processing	OpenCL	Video	2016 × 1512	ARM Cortex A8 (550 MHz) and PowerVR SGX 530 (110 MHz)	3.5x
Lopez et al. [13]	Image recognition	OpenGL ES 2.0	Image	64 × 64, 512 × 512, 1024 × 1024	ARM Cortex A8 (770 MHz) and PowerVR SGX 535 (110 MHz)	2.5x
Cheng and Wang [14]	Face recognition	OpenGL ES 2.0	Image	128 × 128, 512 × 512, 1024 × 1024	Tegra 2	1.8x
Rister et al. [15]	SIFT	OpenGL ES 2.0	Image	320 × 280	Qualcomm Snapdragon S4, etc.	6.4x

various configurable structures were evaluated according to their structural differences, showed how the optimizing techniques affect the execution performance of various image sequence processing algorithms in the mobile environment.

The remainder of this paper is organized as follows. Section 2 presents conventional works related to image processing on mobile platforms. Sections 3 and 4 outlines considerations for mobile heterogeneous computing and the optimizing techniques used to accelerate image sequence processing with mobile heterogeneous computing, respectively. Section 5 describes the various heterogeneous computing implementation methods. Section 6 presents the experimental results obtained for the implementation methods. Finally, Section 7 concludes this paper.

2. Related Work

Image sequences with high resolution and quality have a large amount of data on which most image sequence processing algorithms carry out repetitive operations. These repetitive operations can be efficiently accelerated by a GPU, which has a parallel architecture consisting of multiple cores designed for handling multiple operations simultaneously. One of the key factors in the design is the use of properties such as unified shader, texture compression, and tiled architecture. The desktop platform has a variety of vendor-specific solutions and flexibility such as compute unified device architecture [2], Open Computing Language (OpenCL) [3], and OpenVIDIA [4] to facilitate use of the GPU for general-purpose computation. On the other hand, the mobile platform is limited to OpenCL and Open Graphics Library for Embedded Systems (OpenGL ES) [5]. OpenGL ES is an application programming interface (API) for hardware-accelerated graphics

rendering on mobile devices. OpenGL ES versions 1.0 and 1.1 are available for the implementation of simple GPGPU techniques using a fixed rendering pipeline. Version 2.0 supports complicated GPGPU operations with programmable pipeline via the OpenGL shader language (GLSL) [6]. Version 3.0 further expands the programmability of the pipeline. By using GLSL, the fixed-function vertex and fragment operations of the graphics pipeline are replaced with user-specified programs, also known as vertex shader and fragment shader. These shaders provide flexible programmability, and support various control-flow construct sets and complex data types.

Extensive research has recently been conducted on a mobile programmable embedded GPU for image processing and computer vision. **Table 1** shows various conventional studies and their accelerations relative to the CPU-only structure. Ensor and Hall [7] implemented a real-time Canny edge detection shader and obtained 2.4x faster total execution time. Singhal et al [8] focused on optimized implementations of the cartoon-style non-photorealistic rendering, stereo matching, and speedup robust feature (SURF) algorithms using OpenGL ES 2.0. Their experimental results showed performance improvements in the range 5.2x through 6.9x for the image processing algorithms and 1.68x for the SURF algorithm. Wang et al [9] studied a mapping object removal algorithm using OpenCL. Their accelerations came from the partitioned GPU operations. Pulli et al [10] introduced a variety of acceleration methods with the GPU version of the open source computer vision library (OpenCV) [11]. Their results showed 2.4x to 14x faster total execution time than the CPU version of OpenCV.

As evidenced from these conventional works, most existing works [7-10] focused on mobile GPU acceleration. However, Leskela et al [12] proposed a heterogeneous computing method in which they analyzed the frame timing and introduced an optimization technique for work scheduling. Lopez et al [13] and Cheng et al [14] implemented image and face recognition systems on mobile platforms. The systems, which were implemented with CPU and GPU operations combined, exhibited 2x to 10x faster execution times for face recognition algorithms. Rister et al [15] introduced several techniques, such as partitioned heterogeneous computing and data compression for memory transfer, to speed up the scale-invariant feature transform detection (SIFT) algorithm.

This paper takes into account both the CPU and the GPU on the mobile platform and presents the factors that have to be considered when using both processors. On the basis of these consideration, various optimizing techniques are implemented and their performance are analyzed.

3. Considerations for Mobile Heterogeneous Computing

3.1 Mobile Heterogeneous Computing

Heterogeneous computing uses more than one kind of processor, such as a CPU and a GPU, to combine particular processing capabilities for high computational performance. Use of a GPU for general purposes can result in remarkable speed-ups on both mobile and desktop

platforms, especially with image processing and computer vision algorithms. In such cases, the CPU is employed for the OS, traditional serial tasks, and inputting image sequences stored in main memory and transmitted over networks.

Nowadays, mobile devices are equipped with a system-on-a-chip mobile application processor (AP). This mobile AP is typically configured with a CPU, GPU, image signal processor, multi-format video codec, and various interface IPs; that is, a typical heterogeneous computing system. The GPU and CPU are connected via the system bus, which makes it possible to share the memory between them both. Through this memory sharing, one processor can access data processed by the other processor using OpenGL ES on mobile OS frameworks [16].

Several factors have to be considered in heterogeneous computing on mobile platforms. The next subsections present these considerations in detail.

3.2 Idle Time

In a simple heterogeneous computing system, tasks are distributed to the CPU and GPU and performed in sequential order. In this sequential system, the CPU and GPU may have idle time when image sequences are processed because one processor can only start working after the other processor has finished its task [17]. Specifically, the GPU waits for an input image while the CPU is performing its task. Then, when the GPU is working on the input image from the CPU, the CPU falls into an idle state. The idle time between the tasks increases the total execution time in such a heterogeneous computing system.

3.3 Data Transmission and Format Conversion

In a mobile AP, the CPU and GPU share physical memory and the GPU obtains virtual memory from this shared memory. The CPU and GPU can transmit data to each other through the system bus. However, the system bus has a low memory bandwidth because of power supply and physical hardware size limitations on mobile devices. In addition, image sequences typically have a large amount of data and processing them is highly computation intensive. Thus, transmission may take a relatively long time on a heterogeneous computing.

Moreover, there is an additional process required to transmit data between the CPU and GPU. The CPU and GPU handle data in different forms—pixels and vectors, respectively. More specifically, the data format that can be processed by the GPU is texture, which provides the information for rendering an image onscreen. These different data formats make it impossible to provide direct access between CPU arrays and GPU textures. Thus, one data format has to be converted into the other data format. The conversion instructions are supported by the OpenGL ES framework.

Other format conversions may also be required for heterogeneous computing. Mobile OSs are limited to available data formats that support direct conversion from pixel to texture, called “texture conversion” in this paper. If data formats that do not support direct texture conversion are utilized in CPU processing, the data have to be changed to an intermediate format that provides direct texture conversion. This intermediate-format

Table 2. Format conversion and data transmission times on mobile CPU-GPU sequential structure.

Resolutions	Image sequence display using GPU			
	(a): Intermediate-format conversion time (s)	(b): Transmission + Texture conversion time (s)	(a) + (b): % of (c)	(c): Total execution time (s)
854 × 480	1.58	5.2	58.24%	11.65
1280 × 720	2.16	10.93	62.59%	20.92
1920 × 1080	3.0	23.87	71.78%	37.43

conversion is conducted as a CPU task, followed by texture conversion as a GPU task.

As described above, the use of GPGPU requires data transmission and format conversion. To determine the effect of these requirements, the times required were measured for a case in which image sequences were displayed through a GPU. The results are shown in **Table 2**. In this experiment, bitstreams coded using H.264/AVC [18] were given as inputs, and test sequences with three spatial resolutions, 500 frames, and 50 Hz were original or down-sampled versions of BascketballDrive—a common test sequence in the standardization of high efficiency video coding (HEVC/H.265) [19]. The mobile platform comprised an ARM v7-based dual-core CPU (1.3 GHz) and a PowerVR SGX 543MP3 triple-core GPU (325 MHz). As shown in the table, the transmission and conversion times are considerable and constitute higher percentages for higher image resolutions. Therefore, heterogeneous computing should be implemented in manner such that transmission frequency and the amount of data transmitted between the CPU and GPU are minimized.

3.4 Low Memory

The small size of the mobile AP necessitates physical constraints on the amount of memory relative to that of the desktop platform. In addition, mobile applications are allowed only very limited memory because a mobile OS has to manage multiple applications at the same time. When the amount of memory occupied by an application exceeds the maximally available capacity, the mobile OS forcibly terminates the application. Consequently, for high resolution images, memory usage should be optimized within an available memory capacity range.

3.5 Computational Power

Because the power of the mobile AP is supplied from a battery, there are physical limitations to the power efficiency of the processors and other hardware. Thus, the mobile platform has a smaller number of cores, a lower CPU clock frequency, and a smaller number of shader units in the GPU than the desktop platform. This is the reason why the CPU and GPU of a mobile platform perform significantly lower than their desktop counterparts.

Image sequence processing algorithms typically contain many operation commands for a large amount of data, which incurs a significant overhead on the mobile platform.

Moreover, the CPU and GPU have differences in computability and processing time. These differences are likely to cause idle states in the processors when multiple frames are processed. Thus, an understanding of the overall performance of the mobile AP and maintaining a balance between the task loads of the processors in the implementation of image sequence processing algorithms are essential.

4. Mobile Heterogeneous Computing for Image Sequence Processing

On the basis of the considerations discussed in the previous section, this section presents several optimizing techniques for efficient implementation of mobile heterogeneous computing.

4.1 Parallelization of CPU and GPU

The CPU and GPU of a mobile platform can execute operations in parallel. In the case of the CGS structure, the GPU starts accepting the n-th frame after it is processed on the CPU. Thus, the GPU may fall into an idle state while the CPU is processing a frame. On the other hand, in the case of the CGP structure, the GPU can process the n-th frame while the CPU is processing the (n+1)-th frame. This kind of parallelization can improve the operational efficiency of both the CPU and the GPU because they can simultaneously perform their instructions for image frames. Thus, the parallelism can decrease the waiting time and reduce significantly the total execution time.

For parallelization of the CPU and GPU, the CPU-GPU pipeline structure should be retained because the CPU and GPU are usually not allowed to handle the same frame at the same time, and the CPU and GPU tasks should be independently processable on each processor. The parallelization can be implemented as the following workflow using thread level parallelism (TLP). The execution of a mobile application starts on the main thread by the CPU because a mobile OS gains access to the application stored in the main memory via CPU processing; the main thread then calls the sub-thread for the GPU task. In this parallelization technique, independent instruction streams of the CPU and GPU tasks are processed simultaneously on the main thread and sub-threads, respectively.

The CPU task itself can also be processed in parallel, for which the CPU task is divided into modules that are distributed to threads—called task distribution. The CPU task typically includes independently processable modules such as the decoding and the intermediate-format conversion when a coded bitstream of an image sequence is given from the main memory as input. These modules can be distributed into separate threads in such a manner that the decoding operation is on the main thread and the intermediate-format conversion is on another sub-thread. These threads can operate in parallel using TLP. In this case, the decoding and the conversion modules should also maintain a pipeline structure, as in CPU-GPU parallelization.

4.2 Double Buffering

As described in Section 3, the processing times of CPU and GPU tasks differ according to

Table 3. Data loss and meaningless repetitions in parallel processing of the CPU and GPU tasks—Gray-scaling and Sobel edge detection, respectively.

Resolutions	Heterogeneous Structure Types	Gray-scaling on CPU + Sobel edge detection on GPU	
		Number of GPU processed frames	Total execution time (s)
854 × 480	Sequential	500	15.0
	Parallel	411	8.2
1280 × 720	Sequential	500	24.8
	Parallel	374	14.9
1920 × 1080	Sequential	500	47.1
	Parallel	542	35.5

the processing powers of the processors and the amount of data contained in each frame. These differences can cause critical problems such as data loss and meaningless repetitions of the same processing if the tasks are operating in parallel [20]. For example, if the processing time of a CPU task is considerably less than that of the GPU task, the cumulative time difference can cause the CPU processed data to be dropped, much like the effect of buffer overflow. Meaningless repetitions can occur in the opposite case, buffer underflow. If the GPU task is finished much earlier than the CPU task, the same frame can be fed again to the GPU task. These problems were observed in experiments conducted, the results of which are shown in **Table 3**; the experimental conditions were the same as in **Table 2**, except for the image processing algorithms implemented. In these experiments, all test sequences had 500 frames and CPU and GPU tasks included Gray-scaling and Sobel edge detection, respectively. The sequential structure processed exactly 500 frames on the GPU without data loss or repetition problems. However, the parallel structure had fewer frames processed on the GPU for test sequences with 854×480 and 1280×720 pixels, which shows that data loss occurred. For test sequences with 1920×1080 pixels, the GPU task processed 43 more than the original frames. This reveals that 43 meaningless repetitions happened because the CPU task took more time for the relatively larger amount of image data. These problems, caused by the parallelism, reduce the accuracy and efficiency of the image sequence processing.

These problems can be resolved by inserting a buffer between the CPU and the GPU and checking the buffer state. However, as described in Section 3.3, the data transmission time associated with buffer read and write is considerable on a mobile platform. When one processor is transmitting data to the buffer, the other should wait to access the buffer during the transmission time. A suitable solution to all of these problems is double buffering method. **Fig. 1** shows double buffering control using two First-In First-Out buffers, which enables one thread to occupy one buffer when the other thread is accessing the other buffer. Thus, the parallelized tasks can read or write image data alternately from or to the buffers. In this simultaneous access manner, the double buffering reduces the waiting time to access the buffer and solves parallelization problems such as the data loss and meaningless

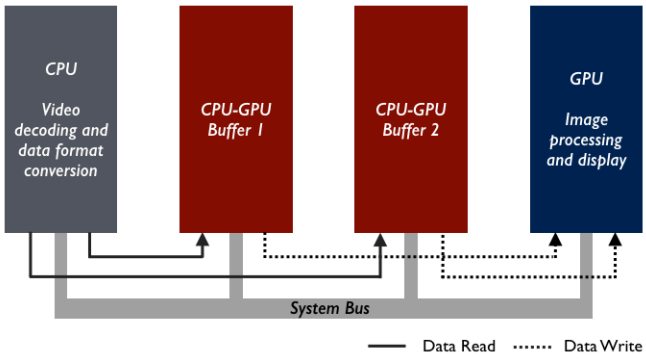

Fig. 1. Double buffer for parallel processing.

repetitions.

Another consideration for the buffer is a lack of RAM on the mobile platform. As described in Section 3.4, the mobile OS may forcibly terminate any application exceeding the memory capacity limit. Thus, in the implementation of mobile heterogeneous computing, the buffer size should be carefully determined considering the available memory capacity and image resolution.

4.3 Data Transmission Minimization

As described in Section 3.3, the use of a mobile GPU for image sequence acceleration requires data transmission between the CPU and GPU, which is a major cause of performance degradation in mobile heterogeneous computing. The transmission time increases proportionally with the size of image data, such as spatial and temporal resolutions of image sequences. Thus, it could be reduced by methods that compress image data [10, 15, 21].

The transmission time also increases, especially in multiple serial processing, where operations are applied to the result of the previous operation. This sort of multiple serial processing is included in many image sequence processing algorithms such as Sobel and Canny edge operators. In these algorithms, the results of a GPU task need to be transmitted to a CPU task and then returned as input for the next operation. This procedure requires textures that include many instructions to access the CPU image array, which makes transmission occur frequently. To minimize this kind of transmission, a competitive solution is the ping-pong technique using FBO [22]. The mobile GPU transmits the image data processed by GLSL operations to a frame buffer for output on a device screen. At this point, the processed image data in the frame buffer can be copied and stored in the FBO of the GPU memory. When this is done, the data in the FBO can be reused for the next processing stage on the GPU without being displayed onscreen by employing the Off-Screen Rendering command of OpenGL ES. Consequently, FBO enables reuse of GPU results, preventing the results from being passed again through the CPU, which minimizes transmission time.

Table 4. Data transmission speedup using the frame buffer object on the CPU-GPU sequential image processing structure for Canny edge detection algorithm.

Algorithms	Resolutions	Total execution time (s)		Speedup
		Structure without FBO	Structure with FBO	
Canny Edge Detection	854 × 480	39.84	16.84	x2.37
	1280 × 720	79.71	32.68	x2.44
	1920 × 1080	165.03	69.32	x2.38

Table 5. Performance comparison of mobile CPU and mobile GPU for various image processing algorithms.

Algorithms	Resolutions	Number of processed frames (fps)		Speedup
		on CPU	on GPU	
Gray-scaling	854 × 480	67	33	x0.49
	1280 × 720	34	21	x0.60
	1920 × 1080	17	14	x0.80
Gaussian blurring	854 × 480	12	33	x2.75
	1280 × 720	6	21	x3.50
	1920 × 1080	3	14	x4.60
Sobel edge detection	854 × 480	16	33	x2.00
	1280 × 720	8	21	x2.60
	1920 × 1080	4	14	x3.50
Dilation	854 × 480	13	33	x2.50
	1280 × 720	6	21	x3.50
	1920 × 1080	3	14	x4.60
Bilateral filter	854 × 480	20	33	x1.65
	1280 × 720	9	21	x2.30
	1920 × 1080	4	14	x3.50

To ascertain how much transmission time can be minimized using FBO, experiments were conducted for structures with or without FBO; the structure without FBO repeatedly passes intermediate results through the CPU, whereas the structure with FBO does not. In the experiments, the experimental conditions were the same as those in **Table 2**. As shown in **Table 4**, the structure with FBO achieved a higher acceleration with average 2.4x speedup over that without FBO. This acceleration resulted from lower transmission time and higher GPU usage efficiency.

4.4 Task Assignment

It has been reported that GPGPU is suitable for image processing. To verify this claim, various image processing algorithms were simulated on the mobile CPU and GPU, respectively, under
the same experimental conditions as those in **Table 2**. **Table 5** shows that most algorithms

are appropriate for GPU operations, but Gray-scaling is more beneficial for CPU operations. These results confirm that image processing algorithms do not always have to be implemented on the GPU. This is because data transmission is a key factor in deciding whether a workload is suitable for CPU or GPU. In general, compute-intensive algorithms can benefit more from the GPU, while memory intensive applications are better suited to the CPU. Thus, when implementing image sequence processing algorithms for mobile heterogeneous computing, the algorithms should be modularized considering their characteristics and the modules should be distributed appropriately to each processor.

5. Implementation Structures of Heterogeneous Computing System

To analyze the effectiveness of various heterogeneous computing structures, we implemented four structures, CGS, CGP, DCGS, and DCGP, while taking into account the considerations discussed in Section 3, and applying selectively the methods described in Section 4. The CPU-only structure was also implemented as an anchor for comparisons. In those implementations, all heterogeneous computing structures had FBO to minimize the data transmission time. To accept the inputs of image sequences from H.264/AVC coded bitstreams stored in memory, all of the structures contained an H.264 decoding module that performed on the CPU.

5.1 CPU-only Processing (CO)

The CO structure is an easy and common implementation on the mobile platform. As shown in **Fig. 2**(a), all modules perform the image processing sequentially as a CPU task, and the GPU is used only for displaying the output on a screen. To obtain images from compressed bitstreams, this structure performs H.264 decoding. Intermediate-format conversion is not always required in implementations of image sequence processing, but is necessary when using image processing libraries such as OpenCV. The CO structure could be suitable in cases where image sequence processing algorithms have a small number of commands.

5.2 CPU-GPU Sequential Processing (CGS)

CGS is a base structure that employs a GPU to accelerate image sequence processing. As shown in **Fig. 2**(b), the CPU tasks include decoding and intermediate-format conversion, and the GPU tasks include texture conversion, image processing, and display. More specifically, on the CPU, decoding is first performed, followed by intermediate-format conversion from pixel data to texture convertible format. On the GPU, texture data are obtained by texture conversion, and then image processing is performed.

In the CGS structure, the operations concentrated in the CPU in the CO structure are distributed between the CPU and the GPU. This distribution can significantly reduce the image sequence processing time, especially for complex image processing algorithms. On the other hand, the idle time between the tasks still occupies a significant proportion of the execution time in the sequential structure.

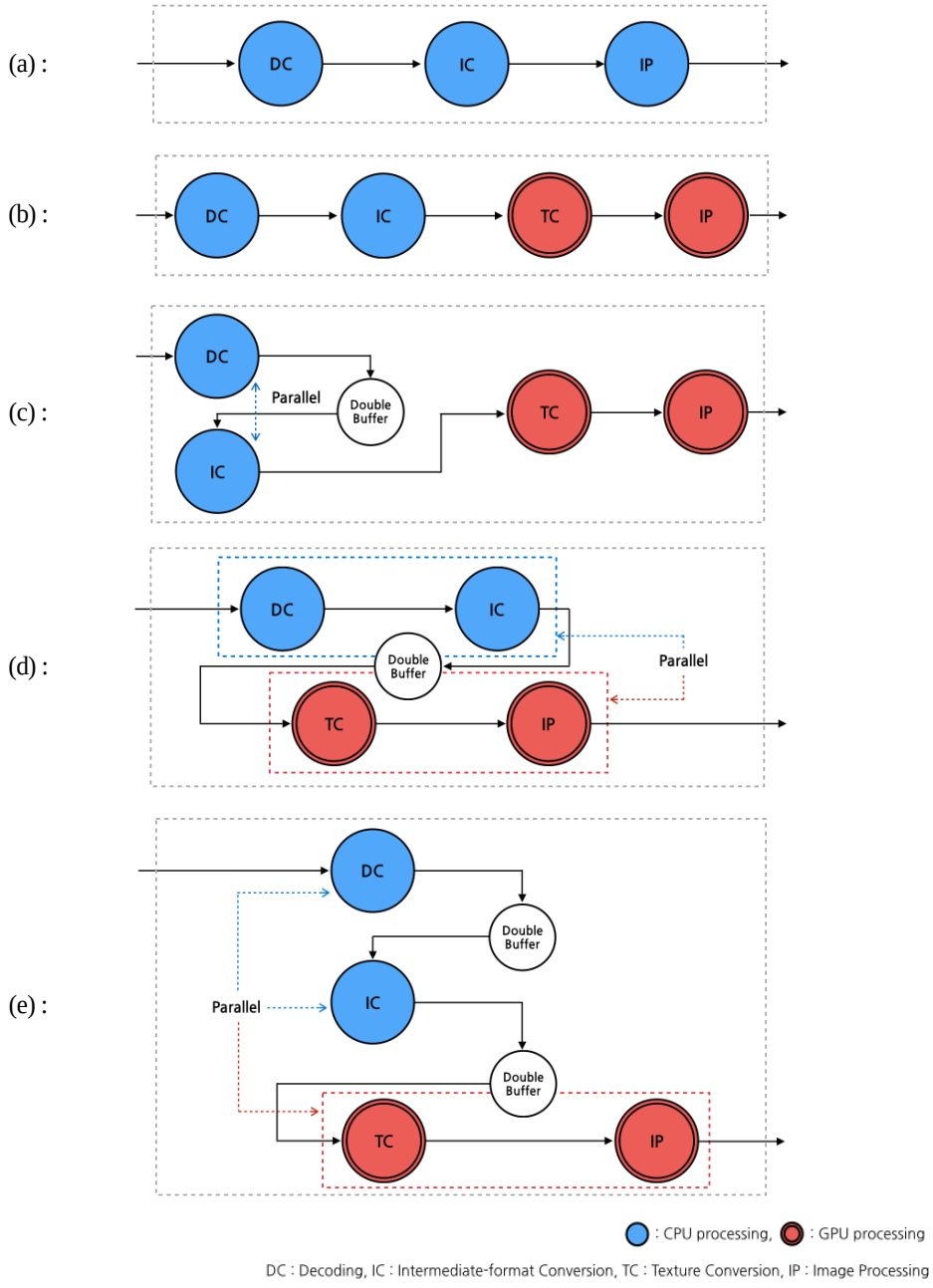

Fig. 2. Outlines of image sequence processing structures: (a) CPU-Only Processing, (b) CPU-GPU Sequential Processing, (c) Distributed CPU-GPU Sequential Processing, (d) CPU-GPU Parallel Processing, and (e) Distributed CPU-GPU Parallel Processing.

5.3 Distributed CPU-GPU Sequential Processing

In the CGS structure, the CPU burden rises significantly as the image resolution is higher. To reduce the CPU burden, the DCGS structure distributes and parallelizes the decoding and the intermediate-format conversion modules of the CPU task using a multithreaded pipeline, as shown in **Fig. 2**(c). This parallelization of the pipelined modules may generate data loss, meaningless repetitions for the same frame, and waiting time to access the buffer. To solve these problems, the CGS structure employs double buffering between a main thread for decoding and a sub-thread for intermediate-format conversion.

5.4 CPU-GPU Parallel Processing

The idle time between the CPU and the GPU can be reduced via processor parallelization of their tasks and double buffering between them. As shown in **Fig. 2**(d), the parallelization is implemented by utilizing a main thread for the CPU task and a sub-thread for the GPU task. On the CPU, the main thread performs decoding and intermediate-format conversion sequentially, and then stores the results into the double buffer. On the GPU, the sub-thread converts the texture convertible frame data to texture, performs image processing, and then displays the output.

5.5 Distributed CPU-GPU Parallel Processing

The DCGP structure applies task distribution, parallelization, and double buffering to the CPU tasks of the CGP structure to improve the sequential processing on the CPU of the CGP. Thus, all of the decoding of the CPU tasks, the intermediate-format conversion of the CPU tasks, and the GPU tasks are performed in parallel by using three threads, as shown in **Fig. 2**(e). More specifically, a main thread first performs the decoding and stores the decoded frames in the double-buffer located between the decoding and the intermediate-format conversion. Next, a sub-thread converts the decoded frames in the double-buffer to texture convertible format, and then stores the results in the other double-buffer located between the CPU and the GPU tasks. Third, a GPU thread conducts texture conversion and image processing. Finally, the output is displayed.

6. Experimental Results

To analyze the benefits of the parallel and distributed mobile heterogeneous computing system, we implemented the five image processing structures described in Section 5: CO, CGS, DCGS, CGP, and DCGP. The CGS, DCGS, CGP, and DCGP structures are based on heterogeneous computing, whereas CO utilizes only the CPU. On top of the structures, we also implemented a variety of commonly used image processing algorithms including Gray-scaling, Gaussian blurring, Bilateral filter, Sobel edge detection, Canny edge detection, Dilation, and Erosion. The mobile platform comprised an ARM v7-based 1.3 GHz Dual-core CPU, a PowerVR SGX 543MP3 triple-core GPU, and 1 GB RAM. To employ the GPGPU, OpenGL ES 2.0 was used. Bitstreams coded using H.264/AVC were given as inputs, and test sequences with three spatial resolutions, 500 frames, and 50 Hz

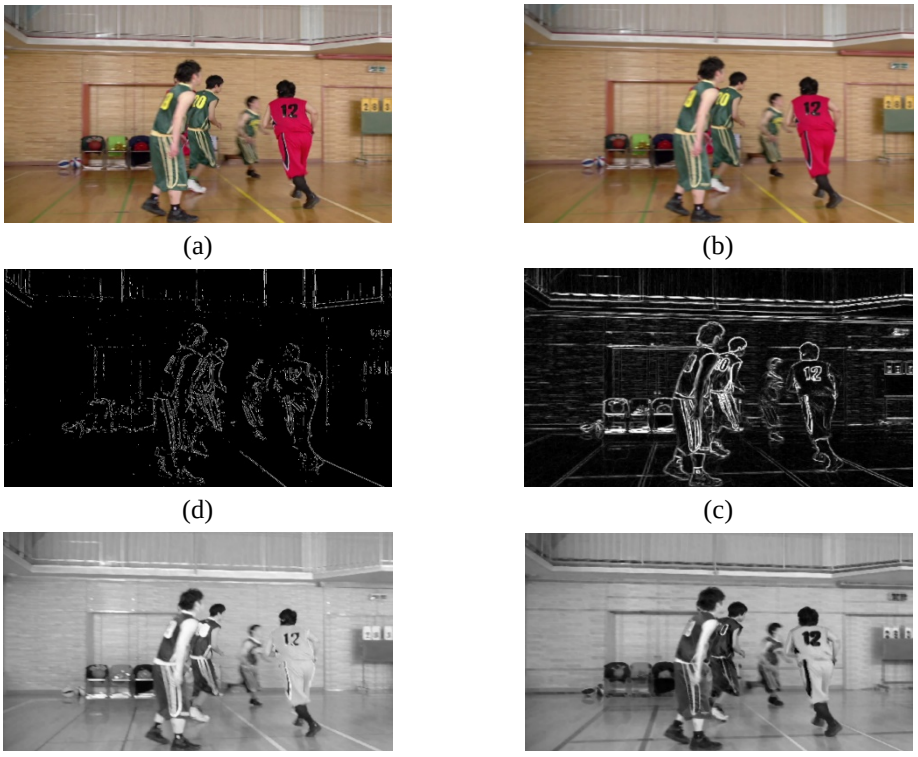

Fig. 3. Examples of image processing algorithms. (a) original image, (b) Gaussian blurring, (c) Sobel edge detection, (d) Canny edge detection, (e) Gray-scaling and Dilation, and (f) Gray-scaling and Erosion.

were original or down-sampled versions of BascketballDrive. **Fig. 3** shows examples of each image processing result on the mobile device. The performances of all the structures are listed in **Tables 6 and 7**, where the speedup is the ratio of the CO structure to a particular heterogeneous structure in terms of the total execution time. The speedup implies how much faster the particular heterogeneous computing structure is than the CO structure. Moreover, the running times are presented separately according to the CPU and the GPU tasks.

6.1 CO vs. Heterogeneous Computing

First, the use of the GPGPU is evaluated. As shown in **Tables 6 and 7**, the structures using GPGPU, such as CGS, DCGS, CGP, and DCGP, achieved average CPU execution time reductions of 75.9%, 81.2%, 70.3%, and 70.2% compared to CO, respectively. These substantially reduced CPU burdens affecting directly the total execution times, which were also reduced on average by 58%, 64%, 70%, and 70%, respectively. The total execution time of the heterogeneous implementations contains the data transmission time and the texture conversion time, whereas these times are not required in the CO structure.

Table 6. Execution time and speedup ratio comparison of image processing algorithms on the homogeneous computing implementations (OC, CGS, and DCGS).

Algorithms	Spatial resolution	Execution Time (s)								
		OC	CGS				DCGS			
		CPU	CPU	GPU	Total	Speedup	CPU	GPU	Total	Speedup
Gray-scaling	854 x 480	10.21	7.58	4.17	11.75	0.87x	6.39	4.93	11.33	0.9x
	1280 x 720	19.96	15.74	5.12	20.86	0.96x	12.36	7.94	20.3	0.98x
	1920 x 1080	35.95	34.67	8.95	43.61	0.82x	24.39	11.52	35.91	1.0x
Gaussian blurring	854 x 480	19.94	7.23	7.88	15.12	1.32x	6.40	6.27	12.67	1.57x
	1280 x 720	29.91	15.73	9.33	25.06	1.19x	12.24	8.19	20.43	1.46x
	1920 x 1080	58.16	34.62	13.76	48.38	1.2x	24.19	12.64	36.82	1.58x
Bilateral filter	854 x 480	30.83	7.24	8.06	15.30	2.01x	6.18	6.98	13.16	2.34x
	1280 x 720	62.89	15.94	9.04	24.97	2.52x	12.46	7.92	20.38	3.09x
	1920 x 1080	122.38	34.47	10.48	44.95	2.72x	24.16	11.37	35.53	3.44x
Sobel edge detection	854 x 480	19.98	7.32	7.70	15.02	1.33x	6.54	5.80	12.33	1.62x
	1280 x 720	39.94	15.71	9.30	25.01	1.60x	12.10	8.12	20.21	1.98x
	1920 x 1080	79.66	34.71	12.72	47.43	1.68x	24.02	12.29	36.31	2.19x
Canny edge detection	854 x 480	54.03	7.28	9.56	16.84	3.21x	5.75	9.44	15.19	3.56x
	1280 x 720	103.94	15.63	17.05	32.68	3.18x	12.00	17.06	29.06	3.58x
	1920 x 1080	211.25	35.14	34.18	69.32	3.05x	26.76	34.41	61.18	3.45x
Dilation	854 x 480	49.91	7.25	8.05	15.30	3.26x	6.05	8.50	14.56	3.43x
	1280 x 720	99.59	15.78	12.55	28.33	3.51x	12.26	11.82	24.07	4.14x
	1920 x 1080	207.59	34.84	24.13	58.97	3.52x	26.26	23.23	49.49	4.19x
Erosion	854 x 480	50.01	7.21	8.00	15.21	3.29x	6.03	8.32	14.36	3.48x
	1280 x 720	100.34	15.78	12.57	28.34	3.54x	12.23	11.91	24.14	4.16x
	1920 x 1080	211.52	35.04	24.14	59.18	3.57x	26.17	23.19	49.37	4.28x
Average	854 x 480	33.56	7.30	7.63	14.93	2.25x	6.19	7.18	13.37	2.51x
	1280 x 720	65.22	15.76	10.71	26.47	2.46x	12.23	10.42	22.66	2.88x
	1920 x 1080	132.36	34.78	18.34	53.12	2.49x	25.14	18.38	43.51	3.04x

Table 7. Execution time and speedup ratio comparison of image processing algorithms on the heterogeneous computing implementations (CGP and DCGP).

Algorithms	Spatial resolution	Execution Time (s)							
		CGP				DCGP			
		CPU	GPU	Total	Speedup	CPU	GPU	Total	Speedup
Gray-scaling	854 x 480	9.85	10.10	10.10	1.01x	9.89	10.12	10.12	1.01x
	1280 x 720	19.47	19.94	19.94	1.0x	19.41	19.84	19.84	1.01x
	1920 x 1080	33.81	33.82	33.82	1.06x	33.26	33.26	33.26	1.08x
Gaussian blurring	854 x 480	9.90	10.12	10.12	1.97x	9.90	10.12	10.12	1.97x
	1280 x 720	19.36	19.88	19.88	1.5x	19.45	19.90	19.90	1.50x
	1920 x 1080	34.14	34.19	34.19	1.7x	33.81	34.49	34.49	1.69x
Bilateral filter	854 x 480	9.88	10.13	10.13	3.04x	9.94	10.14	10.14	3.04x
	1280 x 720	19.45	19.96	19.96	3.15x	19.27	19.74	19.74	3.19x
	1920 x 1080	33.69	33.70	33.70	3.63x	33.43	33.42	33.42	3.66x
Sobel edge detection	854 x 480	9.91	10.14	10.14	1.97x	10.11	10.32	10.32	1.94x
	1280 x 720	19.39	19.92	19.92	2.01x	19.37	19.82	19.82	2.02x
	1920 x 1080	34.14	34.17	34.17	2.33x	33.63	34.27	34.27	2.32x
Canny edge detection	854 x 480	12.15	12.53	12.53	4.31x	12.11	12.48	12.48	4.33x
	1280 x 720	21.88	22.60	22.60	4.6x	22.12	22.78	22.78	4.56x
	1920 x 1080	46.00	47.63	47.63	4.43x	46.31	47.85	47.85	4.41x
Dilation	854 x 480	9.99	10.28	10.28	4.85x	10.19	10.45	10.45	4.78x
	1280 x 720	19.42	19.98	19.98	4.98x	19.72	20.18	20.18	4.94x
	1920 x 1080	39.03	40.16	40.16	5.17x	39.56	40.72	40.72	5.10x
Erosion	854 x 480	12.02	10.28	10.28	4.86x	10.13	10.40	10.40	4.81x
	1280 x 720	19.41	19.94	19.94	5.03x	19.60	20.10	20.10	4.99x
	1920 x 1080	39.21	40.40	40.40	5.24x	39.77	40.84	40.84	5.18x
Average	854 x 480	10.24	10.51	10.51	3.19x	10.33	10.58	10.58	3.17x
	1280 x 720	19.77	20.32	20.32	3.21x	19.85	20.34	20.34	3.21x
	1920 x 1080	37.15	37.73	37.73	3.51x	37.11	37.84	37.84	3.50x

Nevertheless, the heterogeneous computing structures achieved significant speedups. It can also be seen that the compute-intensive algorithms Canny, Dilation, and Erosion had higher speedups of 2.0x through 5.2x than the simple algorithms Gray-scaling and Gaussian Blurring, which had only 1.3x through 1.9x. In the same fashion, test sequences with higher spatial resolutions obtained higher acceleration ratios. Another observation is that the Gray-scaling algorithm was accelerated in the CGP and DCGP structures. As described in Section 4.4, this algorithm is suitable for CPU operations. Thus, the CGS and DCGS structures have longer execution times than the CO structure. However, as the CGP and DCGP structures perform image processing in parallel with other modules such as decoding and intermediate-format conversion, they achieved an overall total execution time reduction.

6.2 OC vs. CGS

CGS executes CPU and GPU tasks sequentially for a frame, and the image processing algorithms are executed on the GPU, as shown in **Fig. 4**(a). As listed in **Table 6**, for all the image processing algorithms except Gray-scaling, CGS showed faster execution times than CO as a result of high GPU throughput. These experimental results verify that use of the GPU for general purposes is feasible, especially for complex image processing algorithms. On the other hand, CGS ran slower by 2 to 18% than CO for Gray-scaling. Because Gray-scaling has simple and light computations, the acceleration achieved using the GPU is not enough to overcome the overhead caused by data transmission between the processors and format conversion that are necessary to use the GPU.

6.3 CGS vs. DCGS

As shown in **Table 6**, DCGS obtained on average 2.51x, 2.88x, and 3.04x speedup of the total execution time relative to CO for test sequences with 854×480, 1280×720, and 1920×1080 pixels, respectively. The performance of DCGS is slightly higher than that of CGS, which achieved on average 2.25x, 2.46x, and 2.49x speedup. It can be seen that the performance improvement of DCGS relative to CGS is higher for test sequences with higher spatial resolution. As a larger amount of computing is required, DCGS can reduce the execution time more significantly. The reason for the acceleration includes the fact that the processing time for decoding and intermediate-format conversion on the CPU can be reduced via distribution and parallelization, as illustrated in **Fig. 4**(b). As shown in the figure, Thread-0 for the decoding and Thread-1 for the intermediate-formant conversion perform in parallel. Thread-0 has no idle time when the buffer is not full. However, Thread-1 has latency until the Thread-2 has finished the GPU task because the CPU and GPU tasks operate sequentially. Therefore, as shown in **Table 6**, the GPU execution time of DCGS remained approximately unchanged relative to that of CGS.

6.4 CGS vs. CGP

The main difference between CGP and CGS is the parallelization of the CPU and GPU tasks. This parallelization can have Thread-0 conduct its task incessantly, as in **Fig. 4**(c).

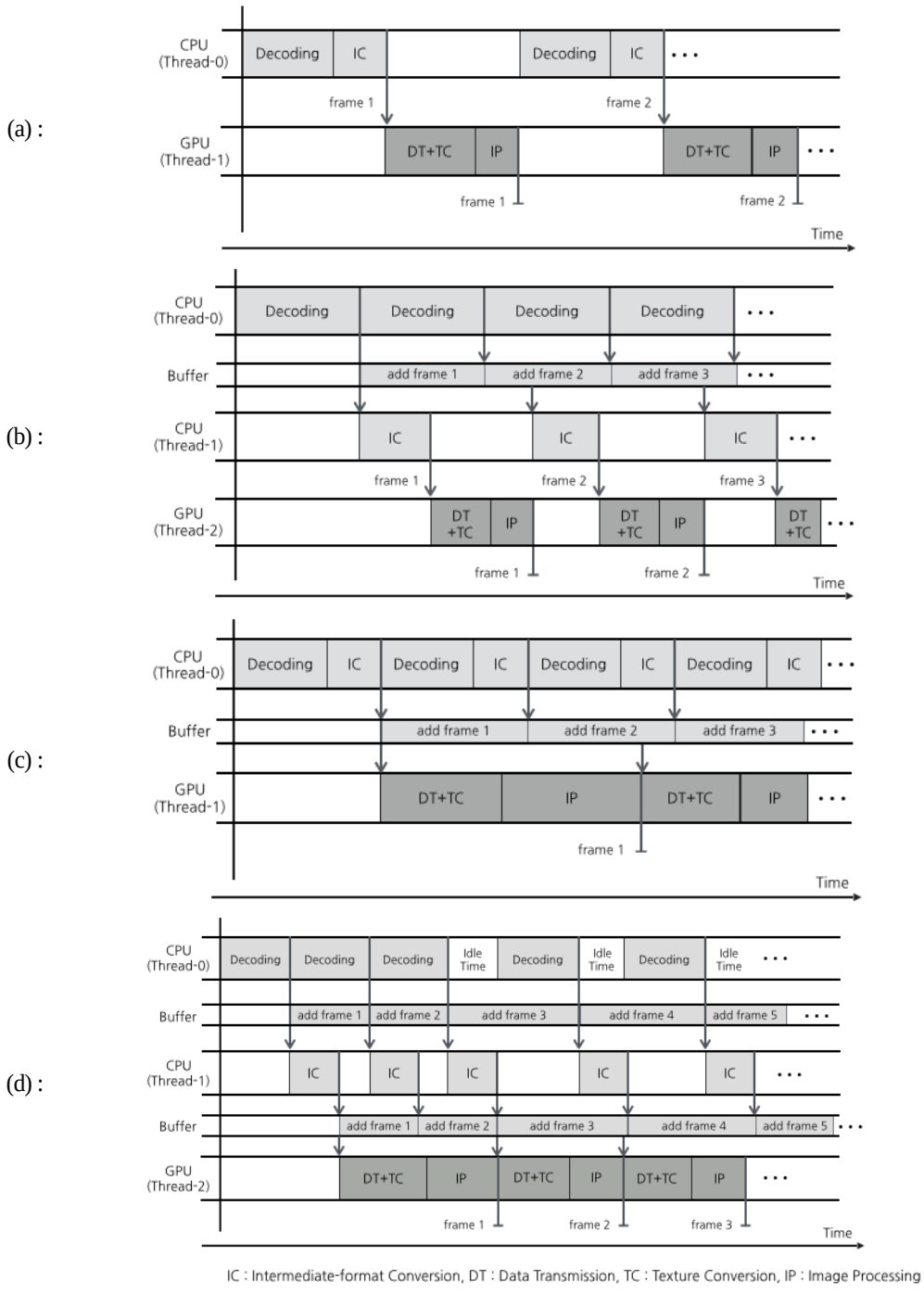

Fig. 4. Examples of processing procedure of each implemented structure (a) the CGS structure, (b) the DCGS structure, (c) the CGP structure, and (d) the DCGP structure.

As expected, CGP achieved an average reduction of more than 12% in the running time compared to CGS, as shown in **Tables 6 and 7**. The result proves that CGP works well with reduced idle time between the CPU and GPU as a result of parallelization. To decrease further the idle time, CGP should have similar processing times between the CPU and GPU tasks. The times can be brought closer by using task distribution and load balancing between the tasks.

6.5 CGP vs. DCGP

In DCGP, the decoding, intermediate-format conversion, and GPU tasks operate in parallel using three threads and two of the double-buffers, as shown in **Fig. 4**(d). As shown in the aforementioned experimental results, the distribution of the CPU task in DCGS reduced the idle time between the CPU modules compared to CGS, and the parallelization of the CPU and GPU tasks in CGP also reduced the idle time between the two processors compared to CGS. Therefore, DCGP could reasonably be expected to have a fast execution time equal to the sum of the time gains obtained in the DCGS and CGP structures. However, DCGP had approximately the same performance as CGP without the execution time gain of the distributed and parallel processing of the CPU modules, as shown in **Table 7**. This unexpected result occurred for the following reasons: data transmission between the CPU and GPU of the n-th frame takes a longer time than CPU processing of Thread-0 and Thread-1 for the (n+1)-th frame, which results in idle time on the CPU side when the buffer for decoded frames is full. Therefore, in the DCGP structure, if the data transmission is faster than the CPU task processing, DCGP can surpass CGP.

7. Conclusion

Image sequence processing on a mobile platform can be accelerated by increasing the AP utilization efficiency via heterogeneous computing. For high utilization efficiency, this paper dealt with optimization techniques including task distribution on the CPU side, double buffering, FBO, and parallelization of the CPU and GPU. The experimental results obtained verified that the optimization techniques resulted in significantly improved image sequence processing times in the various heterogeneous computing structures. This kind of acceleration can facilitate real-time multimedia applications such as video enhancement, video coding, augmented reality, and computer vision on mobile devices.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (B0126-15-1013, Development of generation and consumption of jigsaw-liked ultra-wide viewing spacial Media).

References

[1] General Purpose GPU Programming (GPGPU). http://www.gpgpu.org/

[2] NVIDIA Corporation, Compute Unified Device Arhictecture (CUDA). https://developer.nvidia.com/cuda-zone

[3] Khronos Group, Open Computing Language. https://www.khronos.org/opencl/

[4] J. Fung, S. Mann, and C. Aimone, "OpenVIDIA : Parallel GPU computer vision," in *Proc. of ACM International Conference on Multimedia*, pp. 849-852, November, 2005. Article (CrossRef Link).

[5] Khronos Group, Open Graphics Library for Embedded Systems. https://www.khronos.org/opengles/

[6] R. J. Rost, OpenGL Shading Language, 2ed, Addision-Wesley Professional, 2006.

[7] A. Ensor and S. Hall, "GPU-based image analysis on mobile devices," in *Proc. of International Conference on Image and Vision Computing*, New Zealand (IVCNZ), 2011. Article (CrossRef Link).

[8] N. Singhal, J. W. Yoo, H. Y. Choi and I. K. Park, "Implementation and optimization of image processing algorithms on embedded GPU," *IEICE Transactions on Information and Systems*, vol. 95, no. 5, pp. 1475-1484, 2012. Article (CrossRef Link).

[9] G. Wang et al., "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU–A case study," in *Proc. of IEEE international Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 2629-2634, 2013. Article (CrossRef Link).

[10] K. Pulli et al., "Real-time computer vision with OpenCV," Commun. ACM, vol. 55, no. 6, pp. 61-69, June 2012. Article (CrossRef Link).

[11] Open Source Computer Vision (OpenCV). http://opencv.org/

[12] J. Leskela, J. Nikula, and M. Salmela, "OpenCL embedded profile prototype in mobile device," in *Proc. of IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 279-284, 2009. Article (CrossRef Link).

[13] M. B. Lopez et al, "Accelerating image recognition on mobile devices using GPGPU," in *Proc. of SPIE, Parallel Processing for Image Applications*, vol. 7872, pp. 78720R-78720R-10, 2011. Article (CrossRef Link).

[14] K. Cheng and Y. Wang. "Using mobile GPU for general-purpose computing—A case study of face recognition on smartphones," in *Proc. of International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1-4, 2011. Article (CrossRef Link).

[15] B. Rister et al, "A fast and efficient SIFT detector using the mobile GPU," in *Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 2674-2679, 2013. Article (CrossRef Link).

[16] Apple Inc, iOS Developer Library, OpenGL ES Programming Guide for iOS, 2016.

[17] A. Baek, K. Lee, and H. Choi, "CPU and GPU parallel processing for mobile augmented reality," in *Proc. of IEEE International Congress on Image and Signal Processing (CISP)*, vol. 01, pp. 133-137, 2013. Article (CrossRef Link).

[18] ITU-T and ISO/IEC JTC 1, Advanced Video Coding for Generic Audiovisual Services, ITU-T Rec. H.264 and ISO/IEC 14496-10 (AVC), version 1, 2003, version 2, 2004, versions 3, 4, 2005, versions 5, 6, 2006, versions 7, 8, 2007, versions 9, 10, 11, 2009, versions 12, 13, 2010, versions 14, 15, 2011, version 16, 2012.

[19] Joint Collaborative Team on Video Coding (JCT-VC), "High Efficiency Video Coding (HEVC) text specification draft 10(for FDIS & Consent)," *JCTVC-L1003*, Geneva, January 2013.

[20] A. Baek, K. Lee, and H. Choi, "Speed-up image processing on mobile CPU and GPU," in *Proc. of IEEE Asia Pacific Conference on Multimedia and Broadcasting (APMediaCast)*, pp. 79-81, April, 2015. Article (CrossRef Link).
[21] T. Akenine-Moller and J. Strom, "Graphics processing units for handhelds," in *Proc. of the IEEE*, vol. 96, Issue. 5, pp. 779-789, May 2008. Article (CrossRef Link).
[22] Dominik Göddeke: GPGPU Basic Math Tutorial, Fachbereich Mathematik, Universität Dortmund, Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300, November, 2005. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html

Aram Baek received B.S. and M.S. degrees from Hanbat National University, Daejeon, Korea, in 2012 and 2014, respectively, from the department of multimedia engineering. Currently, he is working toward Ph.D. degree in the department of multimedia engineering in Hanbat National University. His research interests include video coding, parallel processing, and multimedia processing.

Kangwoon Lee received B.S. and M.S. degrees from Hanbat National University, Daejeon, Korea, in 2012 and 2015, respectively, in the department of multimedia engineering. His research interests include mobile computing, GPU computing, and multimedia network.

Jae-Gon Kim received his B.S. in electronics engineering from Kyungpook National University, Daegu, Rep. of Korea, in 1990, and his M.S. and Ph.D. in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Rep. of Korea, in 1992 and 2005, respectively. From 1992 to 2007, he was with ETRI, Daejeon, Rep. of Korea, where he was involved in the development of digital broadcasting media services, MPEG-4/7/21 standards and related applications, and convergence media technologies. From 2001 to 2002, he was a staff associate in the Department of Electrical Engineering at Columbia University, New York, NY, USA. He is currently a professor in the School of Electronics and Information Engineering at Korea Aerospace University, Goyang, Gyeonggi-do, Rep. of Korea. His research interests include video signal processing, video coding, and digital broadcasting media.

Haechul Choi received his B.S. in electronics engineering from Kyungpook National University, Daegu, Korea, in 1997, and his M.S. and Ph.D. in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1999 and 2004, respectively. He is an associate professor in the Information and Communication Engineering at Hanbat National University, Daejeon, Korea. From 2004 to 2010, he was a Senior Member of the Research Staff in the Broadcasting Media Research Group of the Electronics and Telecommunications Research Institute (ETRI). His current research interests include image processing, video coding, and video transmission.