Accelerate Local Tone Mapping for High Dynamic Range Images Using OpenCL with GPU

Kuo-Feng Liao[1], Yarsun Hsu[2]

Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan 30013

[1]louis7921@gmail.com

[2]yshsu@ee.nthu.edu.tw

Abstract -- Tone mapping has been used to transfer HDR (high dynamic range) images to low dynamic range. This paper describes an algorithm to display high dynamic range images. Although local tone-mapping operator is better than global operator in reproducing images with better details and contrast, however, local tone mapping algorithm usually requires a huge amount of computation and it takes a long time to display an HDR image. We have designed a highly parallel method using Graphics Processing Unit (GPU) to accelerate the computation in order to achieve a real-time display. The algorithm can be highly parallelized. In order to run on different heterogeneous systems, we choose OpenCL, instead of CUDA, for our implementation. We have demonstrated the speed-up can be as high as 63 times for a 1280x960 image.

Keywords— **OpenCL, HDR, tone mapping, GPU, parallel programming**

I. INTRODUCTION

1.1 Motivation

Recently researchers have been trying to use GPU to accelerate the performance of many applications. There are two main frameworks proposed to utilize the powerful GPUs. CUDA proposed by Nvidia is popular for parallel computing with Nvidia's GPUs, and OpenCL developed by the Khronos Group is another common framework for heterogeneous systems.

Since CUDA can be only used by Nvidia's GPUs, it is hard to use it in heterogeneous computing environment consisting of different platforms. Unlike CUDA, OpenCL can easily utilize heterogeneous systems, and it can choose CPU or GPU as the compute unit. Therefore, we choose OpenCL as the main framework to implement our parallel algorithm in this work. The remaining question is how we can parallelize the algorithm with OpenCL for the application we are interested in in order to take advantage of the powerful GPUs.

In the real world, there are a wide range of color and luminance information in our visual system. Thus, representing High Dynamic Range (HDR) images will need more than 8 bits in each channel. With captures and storage methods for high dynamic range being improved quickly, the imaging technologies can be used in real world. In contrast, there is a limited dynamic range in the modern display devices. As a sequence, a number of tone mapping operators have been proposed and designed to display HDR images on these devices with low dynamic range (LDR). As we mention before, GPUs are powerful for general purpose computing. Before CUDA and OpenCL are proposed, there were some works [19, 20] trying to use graphics hardware to process tone mapping with limited success. In this work, we would like to take advantage of the powerful resources offered by modern GPUs to accelerate the speed of the local tone mapping operator.

There are two main categories for tone mapping algorithm: global tone mapping operators and local tone mapping operators. The global tone mapping operators apply the same designed mapping method to every pixel of the image. Therefore, global operators will reduce the contrast and detail of the image. On the other hand, local tone mapping operators use spatially varying mapping function for each pixel with its localized content.

Adaptive Local Histogram Adjustment (ALHA) based tone mapping operator has been presented by [3]. ALHA is one kind of local tone mapping operators. Although ALHA is designed to be a fast local operator, it still needs a huge amount of computation like other local operators. Fortunately, ALHA is a segmentation based tone mapping operator, we may be able to parallelize it on GPU to gain a tremendous improvement on performance. Qiyuan et al. had also implemented this operator on the GPU with CUDA [19]. However, we think the performance can still be improved. We use openCL to implement the algorithm onto GPU and compare the performance between their CUDA version and our OpenCL version.

1.2 Goals and Contributions

Adaptive Local Histogram Adjustment (ALHA) is designed to reproduce high dynamic range image with better contrast and details. We can separate an image into non-overlapping regular regions and apply Histogram Adjustment based Linear to Equalized Quantizer (HALEQ) in each block. This would extend the dynamic range for display. Each region can get larger dynamic range for display and thus provide us better visual experience.

For displaying HDR images, local tone mapping operators give better contrast and detail than global operators. However, the computing for image reproduction is very time-consuming with local operators and the speed to display an HDR image is usually slow. Therefore, we hope to accelerate the process,

and we think powerful GPUs can be adopted to achieve the goal. Once the speed of reproduction improves, displaying not only an image but also HDR video may be achieved in real-time.

We focus on how to parallelize the algorithm. There are several parts that can be parallelized in the algorithm. The most time-consuming part would be in weighting function, and this part may be improved tremendously with GPU. In this study we have achieved a speedup of up to 63x in reproducing an image with 1280*960 resolution.

1.3 Organization

This paper is organized as following: Section 2 describes related works and background, including the OpenCL programming and tone mapping algorithm. In section 3, we describe the modification of Adaptive Local Histogram Adjustment (ALHA) algorithm. Then, section 4 will describe the design and implementation of the algorithm and also the parallelizable parts in detail. Finally, the experiment results and analysis will be stated in section 5, and section 6 will present the conclusion of this study.

2. Related Work and Background

2.1 Tone mapping operators

2.1.1 Global tone mapping

Global tone mapping operator was first developed by Tumblin and Rushmeier [4] and Ward [5]. Global operator proposed by Tumblin and Rushmeier was designed to match perceived brightness of displayed image with brightness of the original scene. And Ward's operator was designed to balance the contrast between the digital image and the original scene. Later, another global operator with computational model of visual adaptation was proposed by Ferwerda et al. [6]. Furthermore, there was a global operator proposed by Larson et al. [7] using a histogram adjustment method to map the luminance of our real world into limited dynamic range. This idea successfully gained a great effect compared to linear tone mapping operators. Larson et al. next proposed a more complex operator with more models of human visual system. And Drago et al. [8] designed a global operator using logarithmic compression of luminance to display HDR images with high quality. The effect of this kind of tone mapping operators is very similar to the human response to light. Recently, a new novel global tone mapping operator designed by Duan et al. [9] used learning-based technique to display HDR image.

2.1.2 Local tone mapping

Since images usually lose the details and contrast after global tone mapping processing, local tone mapping operators have been proposed to alleviate this problem. Some local tone mapping operators are listed below:

(1) Tumblin and Turk [10]: They used a layer-based method to display HDR images, and extended anisotropic diffusion which is the edges-preserving low-pass filter.
(2) Durand and Dorsey [11]: They also used layer-based method but simpler than Tumblin and Turk's. They filtered the image with a bilateral filter into a base layer and a detail layer.
(3) Li et al. [12]: They first decomposed images into two layers with bilateral filter, and adjusted the base layer using global tone mapping operator. They then used result obtained in base layer to enhance the detail layer.
(4) Li and Lavanya et al. [13]: They proposed a multi-scale image processing technique using a symmetrical analysis-synthesis filter bank and computing a smooth gain map for multi-scale subband images.
(5) Reinhard et al. [14]: They used a novel local tone mapping method, which was based on the well-known photographic practice of dodging-burning.
(6) Fattal et al. [15]: The method based on manipulation of gradient domain in logarithmic space first calculated the gradient domain in the logarithmic luminance domain and then detected the contrast magnitude in the corresponding position in the original luminance domain.
(7) Qiyuan et al. [3]: They proposed an algorithm which combined global tone mapping with local tone mapping. However it needs a lot of computation. In this work, we modify their algorithm to make it parallelizable so that it can run on GPU with OpenCL.

3. Adaptive Local Histogram Adjustment Algorithm

This section describes the algorithm used. Please refer to [1] for additional information.

3.1 Logarithmic mapping (global tone mapping)

Mapping luminance with the logarithmic function compresses the contrast and brightness for high luminance values but increases the low luminance values at the same time. As the initial step, the following logarithmic function is used to compress the luminance of an HDR image to a corresponding luminance D.

$$D(I) = (D_{max} - D_{min}) * \frac{\log(I+\tau)-\log(I_{min}+\tau)}{\log(I_{max}+\tau)-\log(I_{min}+\tau)} + D_{min} \quad (1)$$

τ is set to control the brightness of the whole image. Large τ will make the image darker while the small value makes the image brighter. Since the choice of τ is a trial and error process, the authors designed an approach to automatically set the parameter τ, which was inspired by the work [14].

However, though the whole brightness of the image can be set properly, the details and contrast are lost due to the Eq. (1). Since the range of the visualization devices is usually divided into equal length with 256 intervals, pixels located in the same interval will be assigned to the same integer display level. This causes the loss of the detail and contrast, and cannot fully utilize the display levels. Another method, histogram

equalization is applied to distribute the pixels into each interval equally. But the method only considers the pixel

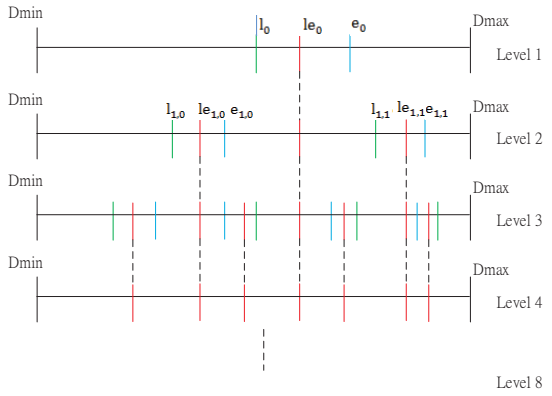

Fig. 1 HALEQ implementation with recursive binary cut method. This process is repeated with Eq. (2) until the number of intervals equal to 256 (display level).

advantage of linear quantization and histogram equalization, the authors presented a tone mapping operator called histogram adjustment based linear to equalized quantizer (HALEQ).

3.2 Histogram Adjustment based Linear to Equalized Quantizer (HALEQ)

The cuts can be found by the following function.

$$le_n = l_n + \beta(e_n - l_n) \quad (2)$$

where β is the controlling parameter in the range between 0 and 1. l_n is the cut for linear mapping, while e_n is the cut for histogram equalization. If $\beta = 0$, the quantization is linear, while $\beta = 1$, the quantization is histogram equalized.

As figure 1 shows, HALEQ starts by finding the cut l_0 which divides the range of $[D_{min}, D_{max}]$ into equal length intervals. Then the cut e_0 is found so that pixels in two intervals are equal to half of total pixel number. After the above two cuts are found, we can use Eq. (2) to find le_0. And using le_0 as the cut point to divide the pixels into two new groups $[D_{min}, le_0]$ and $[le_0, D_{max}]$.The process is repeated 8 times to generate 256 intervals and pixels in the same interval will map to the same display level.

For uniform areas, a common parameter β to improve local contrast may cause noise artifacts. In order to solve the problem, decreasing β in the uniform areas can easily decrease the degree of contrast enhancement. Therefore, the main problem is to define which region is a uniform area.

After determining the uniform areas in the image, the authors proposed the equation (3) to smoothly decrease the parameter β in these areas to reduce the contrast enhancement depending on the degree of uniformity.

$$\beta = 0.6 \times \left[1 - e^{-(SD_{max} - SD_n)}\right] \quad (3)$$

3.3 Weighting Function

When we divide the image into non-overlapping regular blocks and apply HALEQ to each block, two main artifacts will be generated. One is boundary artifact, and the other one is halo artifact. With HALEQ applied to each block, the sharp jumps show up among different blocks, and the result is called boundary artifact which is unacceptable for the image. And each block with different β will cause another artifact called halo effect. Halo artifacts occur in those blocks which are closed to or nearby the uniform areas. For each artifact, we add a weighting function to remove or decrease its effect.

3.4 Removing artifacts

In order to solve the halo artifact problem, a bilateral weighting scheme as shown in the following equation is proposed.

$$d(x,y) = \frac{\sum_{n=1}^{n=N} HALEQ_n[D(x,y)] \times w_{d(n)} \times w_s(n)}{\sum_{n=1}^{n=N} w_{d(n)} \times w_s(n)} \quad (4)$$

where $w_s(n) = e^{-(S_n/\sigma_s)}$, $S_n = \frac{|D(x,y) - Dmean_n|}{D_{max}}$, and $w_d(n) = e^{-(d_n/\sigma_d)}$

d_n is the Euclidean distance between D(x,y) and the center of each block, and σ_d is a parameter which controls the smoothness of the image. With a larger value of σ_d, the influence of d_n will be reduced when calculating w_d, which means that the locality will be less obvious.

$w_s(n)$ is the similarity weighting function, D_{max} is the maximum value in the D(x,y), S_n is the normalized difference between current pixel value and the average pixel value ($Dmean_n$) of block n. The similarity weighting function $w_s(n)$ increases the probability that pixels in the uniform areas are mapped to the similar value even though the pixels belong to different blocks. The value of σ_s will influence the degree of the elimination of haloes and local contrast. With a smaller σ_s, the elimination of haloes will be obvious but the local contrast will be lost.

4. Design and Implementation

4.1 Logarithm mapping

Although Eq. (1) is easy to parallelize for the GPU, we need to compute the average luminance and the parameter τ before we perform the logarithm mapping. Therefore, how to compute the sum of the luminance of the whole image efficiently is important. In traditional method, we can use for-loop to compute the sum, but it takes lots of time when the resolution of the image increases. Since we are going to use GPU to do logarithm mapping, we can also use GPU to

compute the sum. Here, we use a method like what the integral image does but only compute the sum of each column.

Each thread will add the value of each column from top to bottom which is more efficient than computing by rows. Since the data is arranged in 1-D array, when accessing the global memory, the locality of data in the cache can be accessed for each thread without repeatedly caching data for each thread. This method reduces more than 90% iterations comparing to the traditional for-loop. After we get the average luminance, we can continue to compute the parameter τ. This part is done on the HOST with the Newton method, which can compute the value of τ within 50 iterations. However, when we compute the τ, we need to handle the divergence situation. In the program, if the situation occurs, the value of τ will be NAN (not a number). Hence, we need to set another starting position to compute again.

If we successfully get the proper τ value, we can do the logarithm mapping. In the logarithm mapping, we can easily assign each pixel to a corresponding thread.

4.2 HALEQ in local region

Before we compute HALEQ for each block, we need to set up the number of blocks and parameters first. The more blocks we have, the more resources of the powerful GPU we can utilize. However, the size of an image is fixed, with more blocks, the size of each block is getting smaller. When using small blocks, the HALEQ might not have impact on the contrast and details. Thus, using the proper number of blocks can have a huge impact on the performance and the effect we want. Some parameter which is used in the weighting function such as D_{max}, and D_{mean} can also be computed in bincount kernel. This can increase a little bit computation for each thread. In the bincount kernel and HALEQ kernel, each thread will be assigned one block to compute.

As we have mentioned before, the number of blocks has impact on block size. The more blocks we have, the fewer information for each thread to compute. After SD_n and SD_{max} are computed, we can move on to the HALEQ step. In the HALEQ step, the first thing is to determine the parameter β.

In this step, we set the threshold to be 68.3% of the SD_{max}, since we consider the SD_n as a normal distribution. Although we only take the region of one standard deviation as the threshold, the value of β won't change a lot. According to Eq. (4), only when SD_n is very close to SD_{max}, the β will drop quickly. As a consequence, most of β computed by Eq. (4) are still close to 0.6. Next, we can compute the most important part of HALEQ. In order to generate new 256 intervals, the process will be repeated 8 times with each time producing 2^i $(i = 1,2,3...8)$ intervals. The computation of half pixel population is the most time-consuming part, since we need to compute it for each interval every time when the process repeat. Therefore, the block size will have impact on the performance.

4.3 Weighting Function

This part is the most time-consuming in the whole process. As Eq. (4) shows, when we want to compute the new display level for a pixel, we need to take the HALEQ of other blocks, distance(d_n), and normalized difference(S_n) in the computation. The distance d_n between the pixel and the center of each block can be computed easily. S_n is also easy to compute since we have computed the parameter it needs in the bnicount kernel. In order to find the display level of current pixel in other blocks' HALEQ, we can use the binary search to find the corresponding display level. Since the weighting function $w_d(n)$ and $w_s(n)$ are controlled by the σ_d and σ_n, the curve of the two weighting function are shown in figure 3.

The parameter σ_d and σ_n will control the curvature of the curve. The maximum values of $w_d(n)$ and $w_s(n)$ are 1, and then drop quickly. When d_n and S_n are larger than σ_d and σ_n, the value of $w_d(n)$ and $w_s(n)$ are less than 0.4. And the multiplication of the two weighting functions will be less than 0.2. Therefore, if a block is far from the current block, the influence of the block can be ignored.

Thus, we can decide how many blocks should be taken into computation. In order to speed up the performance, the fewer blocks we have in the computation, the less time we consume in this step.

For the weighting function kernel, we separate the data up to four segments, as we have done for the logarithm mapping. Since the computation overhead for each thread is much heavier than previous kernels, we can use the stream method to overlap computation-communication time. Most of the data are left on the device, so we don't need to do any memory copy before we launch the weighting function kernel.

5. EVALUATION

5.1 Experiment Environment

The hardware configurations are listed in table I. The Nvidia Geforce GT 750M is the main GPU we use in the evaluation. Although we cannot use CUDA on the OSX, OpenCL is fully supported on the MAC.

Table I: Experiment Environment

Hardware specifications	
CPU	Intel Core i7-4580HQ @2.3GHz 4C/8T
RAM	8GB DDR3-1600MHz X 2
GPU	Nvidia Geforce GT 750M (384 cores@967MHz)
Storage	Apple SSD SM0512F (512GB)
Software specifications	
OS	OSX 10.10.5
File system	HSF+
OpenCL	1.2
OpenCV	3.0

5.2 Overall performance

At the beginning of the experiment, we first run the program with sequential version and GPU version under different image size and different number of blocks. Figure 2 shows the results of the above two versions under different image sizes, and the speed-up. This experiment uses 32*32 blocks and only computes one neighbor block around each block in the weighting function. And the performance time includes the input and output time.

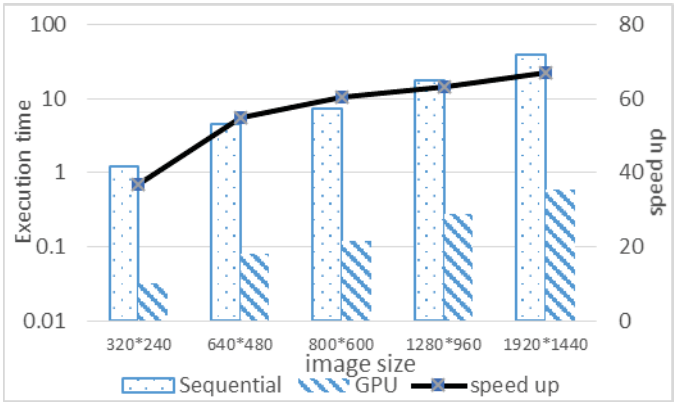

Fig.2 Performance and speed up of different image size.

As we can see from figure 2, the computation time for both versions increases when the image size increases, especially for sequential version. The most time-consuming part is in weighting function. We will show the performance time of the three main parts mentioned in section 4 in the next section. For GPU version, the performance time only increases a little which means that the program can be well parallelized. And the speed-up for the larger image size is more obvious, since the difference of two versions is becoming larger and larger.

Figure 3 shows the speed-up and performance time under different number of blocks. In figure 3, the image size we used is 1280*960, and it also computes one neighbor block around each block in weighting function. As we can see from figure 3, when the number of blocks is small, the advantage of GPU version is less obvious.

However, when the number of blocks increases, performance time for sequential version rises quickly due to heavy computation in the weighting function. Meanwhile, GPU version takes its advantage of powerful GPU with many cores and their execution times do not change very significantly as compared to the sequential version.

GPU version can assign each block to one thread, thus the execution time can be reduced largely. Therefore, the performance time of the GPU version does not change significantly when the number of blocks increases. Since the difference of two versions is getting larger and larger when the number of blocks increases, the speed-up from GPU also improves as the number of blocks increases.

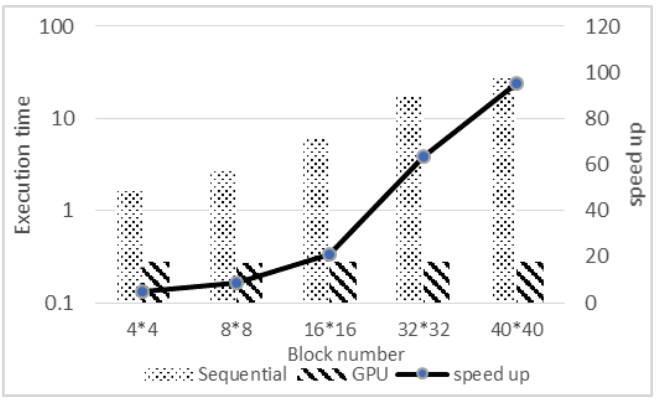

Fig.3 Performance and speed up of different block number

5.3 GPU improvement

Figure 4 shows the performance time of the two versions in each part. In this experiment, we use the image size of 1280*960 and 32*32 blocks. The weighting function also computes only one neighbour block around each block. And figure 8 shows the speed-up of the three parts. As we can see from figure 4, execution time of logarithm mapping and HALEQ step is very short for both versions, but execution time of weighting function exhibits a huge difference between two versions. Table II shows the percentage of each part. The system part includes I/O and other system time which are not parallelized. From figure 4 and table II, we can see the most time-consuming part is in the weighting function of the sequential version. However, the GPU version can highly parallelize the blocks in computation, so it can reduce the execution time very significantly. Figure 5 shows the separated speed-up in the three parts, and most of the acceleration comes from the weighting function. Table II shows that the large proportion of execution time changes from the weighting function in sequential version into the system part in GPU version. Since the system part is not parallelized, the performance will be limited by this part.

Qiuyan et al. had also implemented their ALHA algorithm on the GPU with CUDA [19]. Comparing with their study, our execution time is less than their execution time by 0.08s. The mapping function of ours is about 7.4x faster than theirs, and the weighting process of ours is about 9.11x faster than theirs.

Table II : Percentage of each part

	logarithm mapping	HALEQ	weighting function	system
Sequential	0.5%	0.3%	98.5%	0.7%
GPU	8.4%	1.8%	4.1%	85.7%

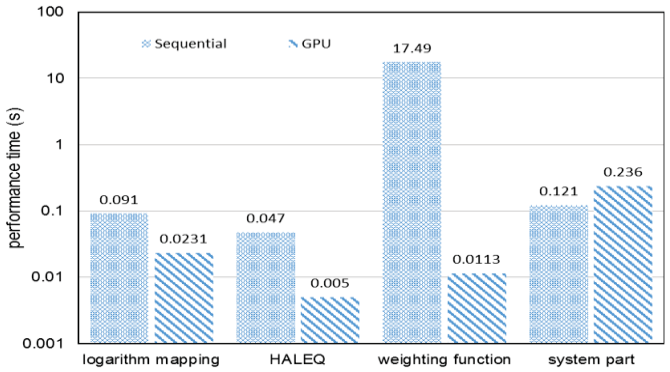

Fig.4 Execution time of each part

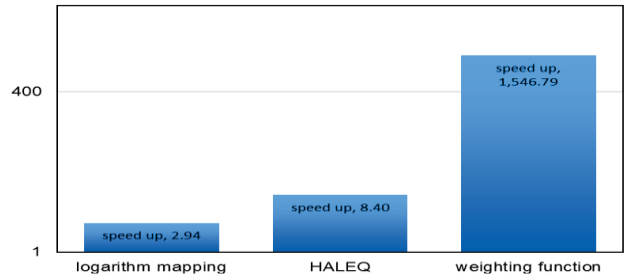

Fig. 5 speed-up of three parts

6. Conclusion

The algorithm of Adaptive Local Histogram Adjustment (ALHA) has been modified and implemented on GPU with OpenCL. We separate the algorithm into three steps: logarithm mapping, HALEQ and weighting function. We then use the new innovative methods to take advantage of the powerful resources offered by GPU.

In logarithm mapping step, we assign each pixel to a thread to utilize the full resources of the GPU. And in HALEQ step, each block can be computed by a thread. The more blocks we have, the larger speed-up we can gain in this step. However, in the weighting function step, we have observed that the required computation time is much longer than the previous two steps when running sequential programs on a single processor is used. For example, it takes about 17.49 seconds to process a 1280*960 image with 32*32 blocks when running this step with a sequential version on an Intel i7 processor. Therefore, this step becomes the bottleneck for the entire program. However, after we parallelize the weighting function and run it on the GPU, the processing time for this step can be reduced to about 0.012 seconds, and the total speed-up for the entire program is up to 63 times.

In conclusion, this paper designs a highly parallelizable algorithm to display high dynamic range images. Although local tone-mapping operator is better than global operator in reproducing images with better details and contrast, local tone mapping algorithm usually requires a huge amount of computation and therefore it takes a long time to display an HDR image. We have demonstrated a highly parallel method using Graphics Processing Unit (GPU) to accelerate computation in order to achieve a real-time display. In order to run on different heterogeneous systems, we choose OpenCL, instead of CUDA, for our implementation. We have demonstrated the speed-up can be as high as 63 times for a 1280x960 image, and thus the performance can be improved significantly with GPU.

7. References

[1] N. Corporation, "Nvidia kepler gk110 architecture whitepaper," 2012

[2] I. Advanced Micro Devices, \Amd graphics cores next (gcn) architecture whitepaper,"2012

[3] Duan, Jiang, et al. "Tone-mapping high dynamic range images by novel histogram adjustment." Pattern Recognition 43.5 (2010): 1847-1862.

[4] Tumblin, Jack, and Holly Rushmeier. "Tone reproduction for realistic images."Computer Graphics and Applications, IEEE 13.6 (1993): 42-48.

[5] Ward, Greg. "A contrast-based scalefactor for luminance display." Graphics gems IV (1994): 415-421.

[6] Ferwerda, James A., et al. "A model of visual adaptation for realistic image synthesis." Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. ACM, 1996.

[7] Larson, Gregory Ward, Holly Rushmeier, and Christine Piatko. "A visibility matching tone reproduction operator for high dynamic range scenes."Visualization and Computer Graphics, IEEE Transactions on 3.4 (1997): 291-306.

[8] Drago, Frédéric, et al. "Adaptive logarithmic mapping for displaying high contrast scenes." Computer Graphics Forum. Vol. 22. No. 3. Blackwell Publishing, Inc, 2003.

[9] Qiu, Guoping, Jiang Duan, and Graham D. Finlayson. "Learning to display high dynamic range images." Pattern recognition 40.10 (2007): 2641-2655.

[10] Tumblin, Jack, and Greg Turk. "LCIS: A boundary hierarchy for detail-preserving contrast reduction." Proceedings of the 26th annual conference on Computer graphics and interactive techniques. ACM Press/Addison-Wesley Publishing Co., 1999.

[11] Durand, Frédo, and Julie Dorsey. "Fast bilateral filtering for the display of high-dynamic-range images." ACM transactions on graphics (TOG) 21.3 (2002): 257-266.

[12] Li, Xiaoguang, Kin Man Lam, and Lansun Shen. "An adaptive algorithm for the display of high-dynamic range images." Journal of Visual Communication and Image Representation 18.5 (2007): 397-405.

[13] Li, Yuanzhen, Lavanya Sharan, and Edward H. Adelson. "Compressing and companding high dynamic range images with subband architectures." ACM transactions on graphics (TOG) 24.3 (2005): 836-844.

[14] Reinhard, Erik, et al. "Photographic tone reproduction for digital images." ACM Transactions on Graphics (TOG). Vol. 21. No. 3. ACM, 2002.
[15] Fattal, Raanan, Dani Lischinski, and Michael Werman. "Gradient domain high dynamic range compression." ACM Transactions on Graphics (TOG). Vol. 21. No. 3. ACM, 2002.
[16] Lischinski, Dani, et al. "Interactive local adjustment of tonal values." ACM Transactions on Graphics (TOG) 25.3 (2006): 646-653.
[17] Reinhard, Erik. "Parameter estimation for photographic tone reproduction."Journal of graphics tools 7.1 (2002): 45-51.
[18] Phil Rogers. "THE PROGRAMMER'S GUIDE TO THE APU GALAXY." AFDS keynote
[19] Tian, Qiyuan, Jiang Duan, and Guoping Qiu. "Gpu-accelerated local tone-mapping for high dynamic range images." Image Processing (ICIP), 2012 19th IEEE International Conference on. IEEE, 2012.
[20] Roch, Benjamin, et al. "Interactive local tone mapping operator with the support of graphics hardware." Proceedings of the 23rd Spring Conference on Computer Graphics. ACM, 2007.

8. ACKNOWLEDGEMENT

The authors would like to thank the support from Ministry of Science and Technology under grant MOST 105-2221-E-007-113.