Preprint of paper presented at IMQA 2016

ON THE COMPUTATIONAL PERFORMANCE OF SINGLE-GPU AND MULTI-GPU CUDA IMPLEMENTATIONS OF THE MAD IQA ALGORITHM

Joshua Holloway[1], *Vignesh Kannan*[2], *Damon M. Chandler*[3], *and Sohum Sohoni*[2]

[1]Oklahoma State University, USA; [2]Arizona State University, USA; [3]Shizuoka University, Japan

ABSTRACT

As modern image quality assessment (IQA) algorithms gain widespread adoption, it is important to achieve a balance between their computational efficiency and their quality prediction fidelity. One way to improve computational performance to meet real-time constraints is to use simplistic models of visual perception, but such an approach has a serious drawback in terms of poor quality predictions and limited robustness to changing distortions and viewing conditions. This paper presents a better alternative, in which computer engineers and image quality experts work together to implement a best-in-class IQA algorithm, Most Apparent Distortion (MAD), on graphics processing units (GPUs). The paper illustrates that an understanding of the GPU and CPU architectures, combined with detailed knowledge of the IQA algorithm, can lead to non-trivial speedups without compromising quality prediction. A single-GPU and a multi-GPU implementation showed a 24x and a 33x speedup, respectively, over the baseline CPU implementation.

1. INTRODUCTION

Algorithms for predicting the visual qualities of images and videos have proved crucial for the design and evaluation of numerous image and video processing systems. Over the last decade, the field of image and video quality assessment (QA) has seen enormous growth in research activity, particularly in the design of new QA algorithms for improved quality predictions. However, an equally important, though much lesser pursued topic of QA research is in regards to computational performance. The vast majority of existing QA algorithms are designed from the outset to sacrifice or even completely neglect the use of perceptual models in order to achieve runtime practicality. Yet, as highlighted in a recent survey [1] and special issue [2], improved QA techniques can be achieved via more extensive perceptual models, even if such models are not computationally practical. The objective of the present study is to facilitate the use of such perceptual models in forthcoming QA algorithms and applications by analyzing the performance gains and bottlenecks associated with a massively parallel implementation.

An intuitive method of achieving large performance gains, in relation to the runtime of the QA algorithm, would be to implement the algorithms on a graphics processing unit (GPU). Using GPUs for general purpose computational tasks has recently become an incredibly important topic in both academia and industry due to the many computational problems that can be solved up to hundreds of times more efficiently on GPUs than on CPUs. Generally, the types of problems that are well-suited for a GPU implementation are ones which can be decomposed into an algorithmic form that exposes a large amount of data-level parallelism. Furthermore, if a specific computational problem can be decomposed into an algorithmic structure that exposes both data- and task-level parallelism, then multiple GPUs can be used to gain even greater performance benefits as opposed to using a single GPU.

Perceptual QA algorithms, in particular, should be prime candidates for a GPU implementation due to the fact that their underlying models attempt to mimic arrays of visual neurons, which also operate in a massively parallel fashion. Furthermore, many perceptual models employ multiple independent stages, which are candidates for task-level parallelization. However, as demonstrated in previous studies [3, 4, 5, 6], the need to transfer data to and from the GPU's DRAM gives rise to performance degradation due to memory bandwidth bottlenecks. This memory bandwidth bottleneck is commonly the single largest limiting factor in the use of GPUs for image and video processing applications. The specific impacts of this bottleneck on a GPU/multi-GPU implementation of a perceptual QA algorithm has yet to be investigated.

To address this issue, in this paper, we implement and analyze the Most Apparent Distortion (MAD) [5] image QA (IQA) algorithm, which is highly representative of a modern perceptual IQA algorithm that can potentially gain massive computational performance improvements using GPUs. Although, MAD is one of the best- performing algorithms in terms of prediction accuracy, MAD employs relatively extensive perceptual modeling which imposes a large runtime that prohibits its widespread adoption into real-time applications. MAD exhibits both levels of parallelism which potentially allows for large performance gains from a multi-GPU implementation. We specifically analyze the performance gains between single- and multiple-GPU implementations of MAD using Nvidia's CUDA parallel computing platform.

As we will demonstrate, it is possible for MAD to achieve near-real-time performance using a single-GPU implementation that parallelizes the modeled neural

We thank Nvidia Corporation for their support of this work.

responses and neural interactions, in a manner not unlike the parallel structure of the primary visual cortex. We further demonstrate that by using three GPUs, and by placing the task-parallel sections of computation on separate GPUs, an even larger performance gain can be realized, though care must be taken to mask the latency in inter-GPU and GPU-CPU memory transfers (see [7, 8] for related masking strategies).

This paper is organized as follows: Section 2 provides an overview of MAD along with details of the implementations, images, and test system. Section 3 provides results of the single-GPU implementation. Section 4 provides results of the multi-GPU implementation. Conclusions are provided in Section 5.

2. METHODS

2.1. MAD Algorithm Description

MAD takes as input a distorted image and its undistorted version. There are two different stages in the algorithm, called the detection stage and the appearance stage, as shown in Figure 1, which are independent of each other until the final calculation of the quality score after both stages are complete. The detection stage analyzes near-threshold distortions; the appearance stage analyzes suprathreshold distortions.

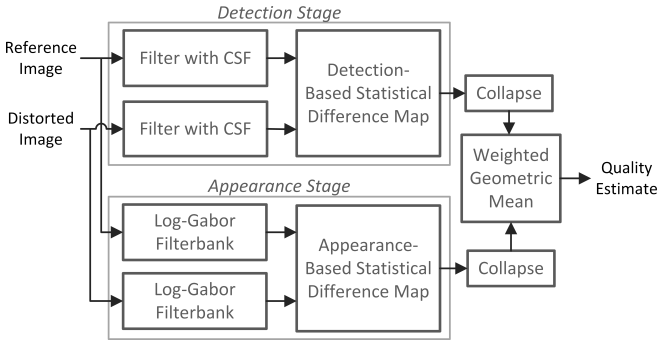

Fig. 1. Block diagram of the MAD IQA algorithm.

The detection stage, represented in the top portion in Figure 1, first performs a series of preprocessing steps on both images. The images are then each filtered with a contrast sensitivity function (CSF) filter kernel. After filtering the images, local statistics are extracted and compared to create a visibility difference map between the two processed images.

In the appearance stage, represented in the lower portion of Figure 1, each image is first decomposed into 20 log-Gabor subbands (5 scales $\times$ 4 orientations) via a filterbank. The local statistics are then extracted from each individual subband.

The detection and appearance difference maps are each collapsed into a scalar quantity through the use of a Euclidean 2-norm. The two resulting scalar values are then combined into a final quality score via a weighted geometric mean.

2.2. Common CUDA Implementation

In this section, we briefly describe the implementation details that are common to both the single- and multi-GPU implementations. Other details of the individual implementations are described in Sections 3 and 4.

The images are read into the computer's memory and are first converted into linearized single precision arrays. The arrays are then copied into the GPU DRAM by transferring the data across the PCIexpress (PCIe) bus. As previously described, and as demonstrated in similar studies [4, 6], transferring data to and from the GPU memory creates a performance bottleneck which often might results in a latency that can obviate the performance gains.

All fully data-parallel operations in our CUDA implementation are performed by individual kernel functions, which are executed such that each CUDA thread performs the kernel-defined operation on a separate array element (pixel or subband coefficient value). For example, in the preprocessing step, before the CSF filter is applied, the images are converted from the pixel domain to the relative lightness domain via a pointwise operation. Because all pixel computations in this operation are independent of each other, we use a separate thread for each pixel. The CSF and log-Gabor filters were applied via frequency-domain filtering. We utilized the cuFFT library to perform all forward and inverse 2D FFTs.

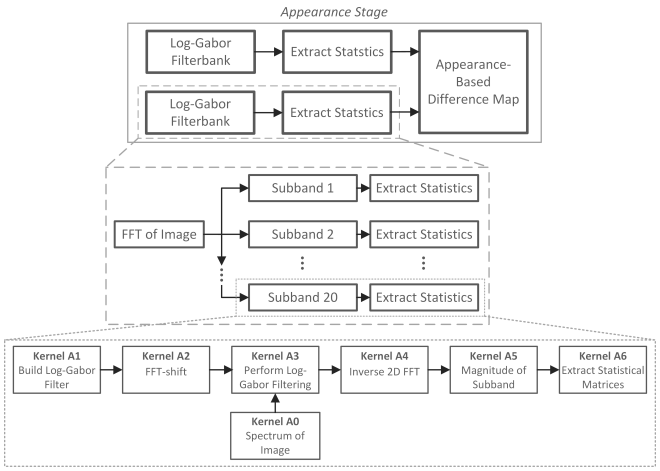

Fig. 2. Diagram of MAD's appearance stage (upper portion) and its GPU implementation (lower cluster).

Figure 2 shows the specific breakdown of how the appearance stage was implemented on the GPU via CUDA kernels. Each log-Gabor filter and statistical computation section is composed of the following CUDA kernels (Kernel A1-A6):

- **Kernel A1** builds the frequency response of the log-Gabor filter.
- **Kernel A2** shifts the filter to accommodate for the DC component lying on the edge of each quadrant.
- **Kernel A3** pointwise multiplies the filter's and image's spectra (cached in GPU memory).

- **Kernel A4** performs an inverse FFT on the filtered image.
- **Kernel A5** takes the magnitude of each complex valued entry in the filtered image array.
- **Kernel A6** extracts three statistical matrices from each subband, corresponding to the standard deviation, kurtosis, and skewness of each 16x16 sub-block, each with four pixels of overlap between neighboring blocks, in each subband.

The set of statistical matrices extracted from each log-Gabor subband corresponding to each pair of distorted and reference image are collected into a single kernel which computes the difference map.

2.3. Apparatus and Images

To collect data for the experiments, we ran the single- and multi-GPU implementations of MAD using images from the CSIQ database [9]. As documented in [6], the semantic category of the reference image did not have a significant effect on MAD's runtime; however, a minor effect was observed for different distortion types and amounts. Thus, we used two reference images, all six distortion types, and two levels of distortion (high and low) for each type.

The test system consisted of a modern desktop computer with an Intel Core i7 processor and three Nvidia GPUs (two GeForce Titans and one Tesla K40; all use the Kepler GK110 GPU architecture). The details of the test system are shown in Table 1. For the single GPU experiment, we used one of the Titans (labeled GPU 2 in Table 1). All performance timings for the GPU-based implementations were performed by using Nvidia's NSIGHT visual profiler.

Table 1. Details of the test system.

CPU	Intel Core i7-4790K CPU @ 4GHz (Haswell) Cores: 4 cores (8 threads) Cache: L1: 256 KB, L2: 1 MB, L3: 8 MB
RAM	16 GB GDDR3 @ 667 MHz (dual channel)
OS	Windows 7 64-bit
Compiler	Visual Studio 2013 64-bit; CUDA 7.5
GPU 1	Nvidia Tesla K40c (PCIe 2.0)
GPU 2	Nvidia GeForce Titan Black (PCIe 3.0)
GPU 3	Nvidia GeForce Titan Black (PCIe 3.0)

3. EXPERIMENT 1: SINGLE GPU

In order to minimize latency introduced by PCIe memory traffic, we transferred the image data from the computer's RAM (the host memory) to the GPU memory only a single time during the beginning of the program to get the image data into the GPU memory.

The results of the performance analysis of the single-GPU implementation are shown in Table 2. Also shown are results for the baseline C++ implementation from [6] measured on our test system.

Overall, going from a CPU-based implementation to a GPU-based implementation results in a significant speedup. The C++ version required approximately 960 ms, whereas the CUDA version required approximately 40 ms, which is a 24x speedup. A runtime of 40 ms would allow MAD to operate at 25 frames/sec in a video application, which can be considered near-real-time performance.

The data in Table 2 indicate that the next major bottleneck in the GPU implementation is located in the statistical computations of MAD's appearance-based stage. We are currently investigating the specific software-hardware interactions.

Table 2. Results of the single-GPU implementation.

	C++		CUDA	
MAD Stage	**Time (ms)**	**% time**	**Time (ms)**	**% time**
Entire Program	958.7	100%	40.2	100%
CSF Filter	77.4	8.1%	0.6	1.4%
Detection Stats.	87.5	9.1%	5.8	14.4%
Log-Gabor Filtering	499.6	52.1%	8.5	21.0%
Appearance Stats.	280.1	29.2%	23.4	58.1%

4. EXPERIMENT 2: THREE GPUS

The most straightforward method of utilizing multiple GPUs is to divide the task-independent portions of an algorithm among the different GPUs. MAD's detection and appearance-based stages do not share any information until the final computation of the quality score; thus, these two stages are easily performed on separate GPUs. The appearance stage can be further decomposed into a largely task-parallel structure by observing that the subbands produced from filtering the reference and distorted images with the log-Gabor filters are not needed together until computing the difference map. Together, the filterbank and the statistics extracted from each subband consume the majority of the overall runtime. Therefore, implementing each of the two filterbanks, and subsequent statistics extractions, on separate GPUs in parallel has the potential to significantly reduce overall runtime.

Thus, we performed the detection stage on one GPU, and the two filterbanks were placed on the remaining two GPUs. We chose to perform the detection stage on GPU 1 because this stage only requires three memory transfers: two 1 MB transfers from the host to copy the image data onto the GPU, and one transfer to copy the detection-based statistical difference map back to the host upon completion of the detection stage. Because GPU 1 used a PCIe 2.0 bus it allowed for the three memory transfers to all occur in parallel with computation on the other two GPUs.

We placed each of the two filterbanks and each subsection's statistical extraction on neighboring GPUs. The 20 individual sections of the filterbank and their subsequent statistical extraction for each image were performed in parallel on GPU 2 and GPU 3, respectively. We then collected all of the statistical matrices

Table 3. Results of the multi-GPU implementation.

	Single GPU	Three GPUs
Entire Program	40.2 ms	28.9 ms
Total PCIe Memory Transfers	4	126
Latency from PCIe Transfers	0.4 ms	0.9 ms
GPU Overlap	n/a	19.5 ms
GPU Activity	38.9 ms	20.6 ms

onto GPU 2, and performed the difference map computation. Due to the need to collect all the statistical data from both filterbanks together to compute the difference map, we had to perform a memory transfers between GPU 2 and GPU 3 by using the host as an intermediary. Upon completion of the appearance stage and detection stage, we copied their difference maps back to the host memory to collapse the maps and calculate the final quality score on the CPU.

The requirement of needing to use the host memory as a buffer caused us to need to perform a very larger number of memory transfers across the PCIe bus. In the single GPU implementation, we only used two 1 MB data transfers at the beginning of the program to get the image data to the GPU, and two 1 MB transfers back to the CPU to collapse and combine the statistical difference maps. Decomposing the filterbank into task parallel sections, as previously described, causes the number of data transfers across the PCIe bus to increase to a total of 126 memory transfers.

The results of the performance analysis of the multi-GPU implementation, with corresponding measurements from the single-GPU implementation, are shown in Table 3. Going from a single GPU to multi-GPU resulted in an overall speedup of approximately 1.4x, which is noteworthy, yet significantly less than expected. Because the cost of transferring data across the PCIe buses is so expensive, we tried to determine a method of overlapping the computation with data transfers. By performing these memory copies asynchronously with respect to computation on both the CPU and GPU, we were able to produce a pipelined structure of our computation and memory transfers, which allowed the multi-GPU implementation to run 1.4x faster than the single GPU implementation. Nonetheless, more optimal strategies of overcoming the memory issue is clearly an area for further investigation; we are currently researching this issue.

5. CONCLUSIONS AND ONGOING WORK

Obtaining performance gains through general-purpose GPU solutions is an attractive area of research and development. To obtain performance gains from utilizing multiple GPUs is a more challenging problem with higher potential performance gains. It requires that the computations be decomposed into a largely task-parallel algorithmic form. If data must be communicated between the tasks placed on separate GPUs, the memory bandwidth bottleneck can prohibit potential gains. To accommodate for the limited PCIe memory bandwidth, an appropriate strategy must be formulated and tailored to the problem at hand and the particular microarchitectural resources of the GPUs.

One of the broader conclusions from this work is that obtaining performance gains through GPU solutions, especially when utilizing multiple GPUs, requires an understanding of the underlying architecture of the hardware, including the CPU, GPU, and their associated memory hierarchies and bus bandwidths.

The second conclusion that can be drawn from this work is that a large number of computational problems can hide the latency introduced by asynchronous memory transfers in the manner proposed in this work. The strategy we found to be best suited for a problem that requires many memory transfers, was to overlap computation with asynchronous memory transfers. Thus, we highly recommend considering this technique for image and video quality assessment algorithms.

6. REFERENCES

[1] D. M. Chandler, "Seven challenges in image quality assessment: Past, present, and future research," *ISRN Signal Processing*, vol. 2013, no. 905685, 2013.

[2] D. M. Chandler, U. Engelke, Y. Horita, and K. Seshadrinathan, "Recent advances in vision modeling for image and video processing," *Signal Processing: Image Communication*, vol. 39, no. B, pp. 1 – 15, 2015, Special Issue Editorial (in press).

[3] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pp. 73–82, 2008.

[4] B. Gordon, S. Sohoni, and D. M. Chandler, "Data handling inefficiencies between CUDA, 3D rendering, and system memory," in *Workload Characterization (IISWC), 2010 IEEE Intl. Symp. on*, pp. 1–10, 2010.

[5] L. Duo and F. X. Ya, "Parallel program design for JPEG compression encoding," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*, pp. 2502–2506, 2012.

[6] T. D. Phan, S. K. Shah, D. M. Chandler, and S. Sohoni, "Microarchitectural analysis of image quality assessment algorithms," *Journal of Electronic Imaging*, vol. 23, no. 1, pp. 013030, 2014.

[7] R. Suda, T. Aoki, S. Hirasawa, A. Nukada, H. Honda, and S. Matsuoka, "Aspects of GPU for general purpose high performance computing," in *Proc. 2009 Asia and South Pacific Design Automation Conference (ASP-DAC)*, Piscataway, NJ, USA, pp. 216–223, 2009.

[8] K. Okarma and P. Mazurek, *GPGPU based estimation of the combined video quality metric*, vol. 102 of *Advances in Intelligent and Soft Computing*, pp. 285–292, Springer Berlin / Heidelberg, 2011.

[9] E. C. Larson and D. M. Chandler, "Most apparent distortion: full-reference image quality assessment and the role of strategy," *Journal of Electronic Imaging*, vol. 19, no. 1, pp. 011006, 2010.