https://doi.org/10.2352/ISSN.2470-1173.2017.12.IQSP-220
© 2017, Society for Imaging Science and Technology

GPGPU based implementation of a high performing No Reference (NR) - IQA algorithm, BLIINDS-II

Aman Yadav*[1]*, Sohum Sohoni*[1]*, Damon Chandler*[2]*;* [1]*Arizona State University (USA) and* [2]*Shizuoka University (Japan)

Abstract

A relatively recent thrust in IQA research has focused on estimating the quality of a distorted image without access to the original (reference) image. Algorithms for this so-called no-reference IQA (NR IQA) have made great strides over the last several years, with some NR algorithms rivaling full-reference algorithms in terms of prediction accuracy. However, there still remains a large gap in terms of runtime performance; NR algorithms remain significantly slower than FR algorithms, owing largely to their reliance on natural-scene statistics and other ensemble-based computations. To address this issue, this paper presents a GPGPU implementation, using NVidia's CUDA platform, of the popular Blind Image Integrity Notator using DCT Statistics (BLIINDS-II) algorithm [8], a state of the art NR-IQA algorithm. We copied the image over to the GPU and performed the DCT and the statistical modeling using the GPU. These operations, for each 5x5 pixel window, are executed in parallel. We evaluated the implementation by using NVidia Visual Profiler, and we compared the implementation to a previously optimized CPU C++ implementation. By employing suitable optimizations on code, we were able to reduce the runtime for each 512x512 image from approximately 270 ms down to approximately 9 ms, which includes the time for all data transfers across PCIe bus. We discuss our unique implementation of BLIINDS-II designed specifically for use on the GPU, the insights gained from the runtime analyses, and how the GPGPU techniques developed here can be adapted for use in other NR IQA algorithms.

Introduction

Effective and efficient quality assessment of visual content finds application in a plenty of areas ranging from quality monitoring of video delivery systems, comparison of compression techniques to image reconstruction. Unfortunately, the benefits of recent advances in IQA and VQA have not carried over to real world systems owing largely to long execution time of these algorithms even for a single frame of image as has been pointed out in multiple publications [1][2][3][9] in the past. GPGPU based implementation for three different Full Reference IQA algorithms have been presented in [4], [5] and [6] with varying success. In time sensitive applications like quality of service monitoring in live broadcasting and video conferencing, a fast performing No Reference IQA is very essential. Addressing this strong need [7] for real time No Reference IQA, we apply GPGPU techniques to a high performing No Reference IQA algorithm, BLIINDS-II.

The objective of our project is to utilize the data parallelism in BLIINDS-II NR-IQA by implementing it using a GPGPU. We aim to study the compute resources and the memory bandwidth needed along with latency issues following the data access pattern of the algorithm and propose suitable optimization techniques.

Overview of BLIINDS-II algorithm

BLIINDS-II is a Non Distortion Specific Natural Scene Statistics (NSS) based NR-IQA. NSS models are the statistical models that represent undistorted images of natural scenes. The algorithm seeks to predict the quality score of a distorted image by estimating the deviation of a distorted image from NSS models. The algorithm first learns how the NSS model parameters vary with varying levels various types of image distortion. Later this learning is applied for the prediction of quality scores using the features extracted from the distorted image. It has been demonstrated to correlate well with human subjective image quality score and compares very well with other high performing FR IQA algorithms in literature.

Next we describe the overall framework of the BLIINDS-II algorithm as shown in Figure 1. First the 2-D DCT coefficients of the input image are computed. These DCT coefficients are computed for each 5x5 pixel block of the image, with an overlap of one pixel width between two successive 5x5 blocks.

The second step of the BLIINDS-II pipeline builds a parametric model of the extracted local DCT coefficients. Four parameters are computed for each 5x5 DCT block by applying a univariate generalized Gaussian density model to the non-DC coefficients of each block. These four parameters are described further below.

In the third step, a feature vector is populated from the DCT coefficient parameters obtained in the previous step. There are two features extracted for each parameter. The obtained parameter values across all the 5x5 DCT blocks are averaged over the top 10 percentiles and top 100 percentiles. These two averages are the two features for each parameter. At this point we have 8 features extracted at input image resolution (512x512).

The features are extracted across three spatial scales, so the input image is down sampled two times and steps number one to three are repeated to obtain a features vector of length 24, 8 for each spatial scale.

In the final step, a Bayesian inference approach is used to predict the image quality score from the extracted features. This involves computation of the posterior probability of each possible quality score given the extracted set of features using a multivariate generalized Gaussian density model trained on a subset of LIVE IQA image database.

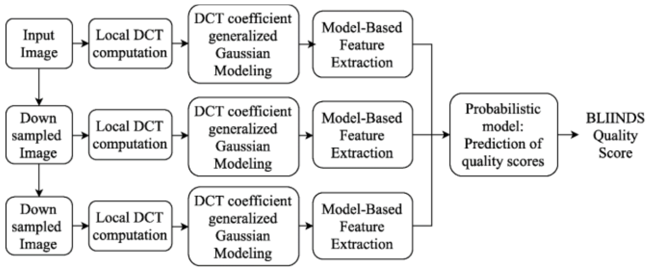
Input Image
Local DCT computation
DCT coefficient generalized Gaussian Modeling
Model-Based Feature Extraction
Down sampled Image
Local DCT computation
DCT coefficient generalized Gaussian Modeling
Model-Based Feature Extraction
Down sampled Image
Local DCT computation
DCT coefficient generalized Gaussian Modeling
Model-Based Feature Extraction
Probabilistic model: Prediction of quality scores
BLIINDS Quality Score

Figure 1. High-Level Overview of the BLIINDS-II Framework

Model based DCT Domain NSS Parameters

The univariate generalized Gaussian density model is applied to the non-DC coefficients of each 5x5 local DCT block for obtaining the model parameters. It is given by:

$$f(x|\alpha,\beta,\gamma) = \alpha e^{-(\beta|x-\mu|)^{\gamma}} \tag{1}$$

where μ is the mean, γ is the shape parameter, α and β are the normalizing and the scale parameters.

Generalized Gaussian Model Shape Parameter (γ)

The shape parameter γ is estimated using the following:

$$\frac{\Gamma(1/\gamma)\Gamma(3/\gamma)}{\Gamma^2(2/\gamma)} = \frac{\sigma_X^2}{\mu_{|X|}^2} \tag{2}$$

where Γ denotes the gamma function. We compute the right hand side of equation (2) and use a look up table to obtain the value of the shape parameter γ. The CPU function and the GPU kernel to do this is called gama_dct.

Coefficient of Frequency Variation (ζ)

The coefficient of frequency variation (ζ) is obtained using:

$$\zeta = \frac{\sigma_{|X|}}{\mu_{|X|}} \tag{3}$$

The CPU function and the GPU kernel to do this is called rho_dct.

Energy Subband Ratio Measure

Each 5x5 DCT block is grouped into three radial subbands. The energy contained within each is referred to as E_n where n can be 1, 2 or 3. The energy subband ratio is computed using:

$$R_n = \frac{|E_n - \frac{1}{n-1}\sum_{j<n} E_j|}{E_n + \frac{1}{n-1}\sum_{j<n} E_j}\ \sigma_n^2 \tag{4}$$

where n can be 2 or 3. The energy subband ratio measure is the mean of R_2 and R_3.

Orientation Model-Based Parameter

Similarly, each 5x5 DCT block is grouped into three orients. The ζ value for each orient is calculated and the variance of the three values is reported as Oriented Model-Based Parameter.

Experimental Methodology

Figure 1 shows a diagram of the flow of our CUDA implementation of BLIINDS-II algorithm. The distorted image is first read and cast as an array of floats, and then sent to the GPU across PCIe. Both the transform and the ensuing statistical operations in the algorithm are data parallel between 5x5 blocks of pixel. On the GPU, we execute all of these 5x5 blocks in parallel. After the features are extracted, the function that maps from feature space to quality score space is executed on the CPU. This CPU function is referred to as "Predict Score".

We implement the BLIINDS-II algorithm on an NVidia Tesla K40c with GK110 microarchitecture using CUDA/C++ for the GPU and the CPU codes. The performance analysis of our application was done using NVidia Visual Profiler (nvvp) and nvprof. Metrics related to compute utilization, achieved memory bandwidth and latency were analyzed to achieve better runtime performance across various images of natural scenes with various distortions and varying levels of each. For all time comparisons, between the C++ implementation and the CUDA implementation of the algorithm, we have used the results reported in [3] for C++ and our own observations for CUDA. Table 1 provides technical specifications of our test system.

Table 1. Test System Specifications.

CPU	Intel® Xeon® Processor E5-1620 @ 3.70 GHz Cores: 4 cores (8 threads) Cache: L1: 256KB, L2: 1 MB, L3: 10 MB
RAM	24GB DDR3@1866 MHz(dual channel)
OS	Windows 7 Enterprise 64-bit
Compiler	Visual Studio 2013 64-bit; CUDA 7.5
GPU1	NVidia Tesla K40(PCIe 3.0)
GPU2	NVidia NVS 310 (PCIe 3.0)

Results and Discussion

The previous C++ implementation of the BLIINDS-II algorithm was reported to take almost 8.0 seconds for 30 iterations for one 512x512 image (approx.. 270 ms for one iteration of one 512x512 image). This average run time was reported across images of various different natural scenes and across multiple distortion types with varying levels of distortion. These images are contained in the CSIQ database.

As seen in the Table 2, from the performance comparison of C++ version of BLIINDS-II, the two most time consuming functions are local 5x5 DCT computation and gamma_dct. Our CUDA implementation also has similar trends but both of these functions have been modified in their flow to better suit the GPU architecture, while preserving their output.

By employing suitable optimizations on code, we were able to reduce the runtime for each 512x512 image from approximately 270 ms down to approximately 9 ms, which is well within the rate for real-time performance (assumed to be a 30 fps video rate). The reported times for CUDA version include the time spent on transferring data across the PCIe bus. The following subsections provide details of our preliminary results.

Performance Comparison of C++ and CUDA Implementations

Table 2 shows the timing comparison of the C++ implementation and the CUDA implementation on the test system. Please note that these times are the total times for 30 runs of the algorithm. The C++ implementation required 8 seconds for the 30 runs, whereas the CUDA implementation required 0.27 seconds for the 30 runs. For the C++ implementation, the DCT stage required approximately half of the runtime. For the CUDA implementation, the Sort, took up half of the application run time. We have used sort from thrust library, included in CUDA toolkit.

Table 2. Performance Comparison of C++ and CUDA implementations of BLIINDS-II for 30 iterations.

Implementation	C++		CUDA	
	Time (ms)	% time	Time (ms)	% time
All	8032	100	266.31	100
DCT	3811	47.44	8.15	3.06
Gama_dct	1882	23.44	6.53	2.45
Rho_dct	297	3.69	4.38	1.64
Convolve	292	3.64	1.97	0.74
Sort	265	3.31	117.72	44.21
Other	1485	18.48	127.56	47.90

Individual Execution Times of Hardware Transactions

Table 3 shows the timings of each individual hardware-transaction stage of the CUDA implementation. The GPU kernels required nearly 60% of the runtime, and the CPU cores require approximately the remaining 40%. Due to the minimal amount of the data that needs to be transferred between the CPU and GPU, the PCI Express memory transfer requires a negligible runtime. Similarly, only a very small percentage of the runtime is required for Predict Score, the CPU function which maps from feature space to quality score space.

Table 3. Performance Evaluation of CUDA implementation of BLIINDS-II for 30 iterations

		Execution time (ms)	% of total time
Program time		266.31	100
GPU kernel execution		156.79	58.87
CPU execution	Total	109.52	39.38
	Predict Score	8.17	3.07
PCIe memory transfer		4.65	1.75

Performance Comparison of gama_dct Implementations

Table 4 shows the runtime spent on the function gama_dct both in the original C++ implementation and in our CUDA implementation. We compare the execution time between three versions of the function we implemented in CUDA. The first version traverses a lookup table stored in global memory, while the second version copies data into shared memory. In the third version, we split gama_dct into three smaller functions, the first split performs computation, the second a sort. Only the third split of the function traverses the lookup table, which we stored in constant memory of the GPU, thereby we could specialize each of the split functions.

The first version, we tried to read from a look up table stored in global memory. The table comprised of 9970 float values, and needed to be looked up for each 5x5 DCT block computed from the distorted image. For a 512x512 image, that results in 29241 DCT blocks, with a worst case of linear traversal of the entire table. Not surprisingly, this version takes up longer than the CPU version of gama_dct function.

In the second version, we copied the lookup table into shared memory, so as to reduce the access time for read. The size of a lookup table was around 40kB, and able to fit in a shared memory of 48 kB. Though there would still be some conflict because we have only one shared memory of 48 kB on a streaming multiprocessor (SM), but we have multiple threadblocks that map on to each SM. Each threadblock will try to put its own copy of the lookup table in shared memory and the shared memory won't fit that. Still, we see a huge gain in performance over the first version of the CUDA gama_dct kernel.

In the third version, we split the gama_dct kernel into three parts. The first split performed all the computations related to each 5x5 DCT block up until the lookup table was to be accessed. Instead of accessing the look up table, it stored its result for each DCT block in a global memory array. The lookup table was stored in constant cache. The reads to this memory are fast if an entire half warp accesses the same address in it. To suit this, the second split sorted the recently populated global memory array with the intermediate results. Finally the third split performed the look up for each entry in the global array. This lookup was executed as a binary search instead of linear search as in the previous versions. The time comparisons of each version is shown in Table 4.

Table 4. Performance Comparison of gama_dct computation methods for 30 iterations.

gama_dct	C++	CUDA Method 1 (Global Memory)	CUDA Method 2 (Shared Memory)	CUDA Method 3 (Split in three kernels)
Run time (ms)	1882	9215.73	781.84	6.53

Rearrange Image

Like many other IQA algorithms, there is local block based processing of image data in BLIINDS-II. Each of these local 5x5 blocks was laid out in memory sequentially, to create a data parallel path of execution of the algorithm. This rearrangement causes some pixel data duplication, owing to the overlap in local

DCT blocks. The new image array size turned to 171*171*32 elements as opposed to 512*512 elements in the original image, an increase in memory requirement by a factor of 3.6.

Compute Local 5x5 DCT

Table 5 shows a comparison of execution time of various implementations of local DCT computation. In our DCT method 1 for GPU, we used the cuFFT library to compute 1D Fourier transforms and used them to arrive at 2D cosine transforms. This method required two invocations of cuFFT 1D transform and two invocations of our custom kernel for converting 1D FFT to 1D DCT.

In DCT method 2, we compute DCT by matrix multiplication method on each local 5x5 pixel block. The DCT matrix was obtained from Matlab dctmtx function.

Table 5. Performance Comparison of DCT Computation using cuFFT (Method 1) and DCT Matrix Multiplication (Method 2).

	CPU Matrix Multiply DCT	DCT method 1	DCT method 2
Time (ms)	3811	25.36	7.62

Merge kernels to minimize global memory load store

We merged some kernels together so as to avoid having to write intermediate results back to GPU global memory. Tables 6 and 7 contain the timing results from our efforts at merging kernels. The results from merging rearrange image and compute local DCT are shown in Table 6. Compute DCT is the logical next step after rearrange image and directly uses the results of the latter.

Oriented_dct feature to be captured from the image for BLIINDS-II processes the local DCT blocks in three different orients. In the earlier C++ and original Matlab implementation of BLIINDS-II algorithm, there were three different function calls, each of which processed the DCT block separately. Method 1 (for individual orient) in Table 7 followed the same structure by executing different kernels for each orient. Method 2 on the other hand sought to merge them together and reported significant gains in performance.

Table 6. Performance Comparison of RearrangeAndDCT.

	Rearrange kernel	DCT kernel	RearrangeAndDCT
Run time (ms)	3.42	7.62	8.15

Table 7. Performance Comparison of oriented_dct kernels.

oriented_dct	Method 1 (For individual orient)	Method 2 (Merged kernel)
Run time (ms)	8.59	3.72

Conclusions

In this paper, we present a significantly accelerated version of the popular BLIINDS-II no-reference (NR) IQA algorithm by using the GPU. To the best of our knowledge, this is the first work to use GPGPU for NR IQA based on an analysis of the interaction of BLIINDS-II with the underlying hardware. Another contribution of this work is a unique implementation of BLIINDS-II's statistical operations designed specifically for use on the GPU. As a result, we are able to present to a version of BLIINDS-II that is capable of real-time performance, significantly improving upon the previously reported runtime after CPU optimizations in C++ [3]. A high speed of execution is critical in application of IQA to video delivery systems, where minimizing the lag is very important.

The analysis of our CUDA implementation shows the top two bottleneck functions to be thrust::sort (44.2%) and thrust::reduce (3.8%). Both of these are Nvidia Thrust library functions bundled with the CUDA toolkit. It is interesting to note that even though the bottleneck functions have changed in the CUDA implementation as compared to the C++ implementation, each kernel individually is faster than the corresponding function in the C++ implementation. Even thrust::sort, the bottleneck kernel for CUDA performs 2.3x faster than the sort in C++ implementation.

We have demonstrated two instances where merging a few smaller kernels together led to better performance. This is seen in the case of merging rearrange image with compute DCT and merging the three oriented_dct kernels together.

In another instance, we have observed a speedup as we split gama_dct into smaller kernels performing a part of the original kernel. Such situations are likely to be observed in case of traversal through lookup tables.

Future Work

The strategies used here should be extended and applied to other implement other IQA algorithms. The performance gains achieved over here could be used to develop and implement IQA algorithms for colored images and/or higher resolution images. To further improve the performance, one must look into better performing alternatives to Thrust library kernels for both sort and reduce. A few options can be cub::DeviceRadixSort and ModernGpu MergeSort. Another avenue to look into for performance improvement is more opportunities for merging smaller kernels together.

The current CUDA implementation of BLIINDS-II should be applied to a real video delivery system, like a live video broadcast or video conference, to learn about the image quality degradations along the delivery channel. Such data can be used to better optimize existing delivery systems.

References

[1] Wang, Zhou. "Objective Image Quality Assessment: Facing The Real-World Challenges." Electronic Imaging 2016.13 (2016): 1-6.

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

[3] Phan, T., Sohoni, S., Chandler, D. M. and Larson, E. C. "Performance analysis-based acceleration of image quality assessment," in Proceedings of the IEEE Southwest Symposium on Image Analysis and Interpretation, April 2012.

[4] B. Gordon, S. Sohoni, and D. Chandler, "Data handling inefficiencies between CUDA, 3D rendering, and system memory," Workload

Characterization (IISWC), 2010 IEEE International Symposium on, pp.1–10, IEEE, 2010.

[5] Okarma, K., and Mazurek, P. "GPGPU based estimation of the combined video quality metric." Image Processing and Communications Challenges 3. Springer Berlin Heidelberg, 2011. 285-292.

[6] J. Holloway, V. Kannan, D. M. Chandler, and S. Sohoni, "On the Computational Performance of Single-GPU and Multi-GPU CUDA Implementations of the MAD IQA Algorithm", International Workshop on Image Media Quality (IMQA) 2016.

[7] Manap, Redzuan Abdul, and Ling Shao. "Non-distortion-specific no-reference image quality assessment: A survey." Information Sciences 301 (2015): 141-160.

[8] Saad, Michele A., Alan C. Bovik, and Christophe Charrier. "Blind image quality assessment: A natural scene statistics approach in the DCT domain." IEEE Transactions on Image Processing 21.8 (2012): 3339-3352.

[9] Phan, Thien D., et al. "Microarchitectural analysis of image quality assessment algorithms." *Journal of Electronic Imaging* 23.1 (2014): 013030-013030.

Author Biography

Aman Yadav received his B. Tech degree in Mechanical Engineering from Indian Institute of Technology (BHU), Varanasi (2014) and an MS degree in Engineering from Arizona State University (2016). He currently works as a Software Engineer at AMD in Radeon Technology Group. He works to capture GPU performance metrics in graphics applications and enhance the execution speed.

Sohum Sohoni received the B.E. degree in Electrical Engineering from Pune University, India, in 1998 and a PhD in Computer Engineering from the University of Cincinnati, Cincinnati, Ohio, in 2004. He is currently an Assistant Professor in The Polytechnic School in the Ira A. Fulton Schools of Engineering at Arizona State University. Prior to joining ASU, he was an Assistant Professor at Oklahoma State University. His research interests are broadly in the areas of engineering and computer science education, and computer architecture. He has published in the International Journal of Engineering Education, Advances in Engineering Education, and in ACM SIGMETRICS and IEEE Transactions on Computers.

Damon M. Chandler received the B.S. in Biomedical Engineering from The Johns Hopkins University (1998); and the M.Eng., M.S., and Ph.D. in Electrical Engineering from Cornell University (2000, 2004, 2005). From 2005-2006, he was a postdoc in the Department of Psychology at Cornell. From 2006-2015, he was on the faculty at Oklahoma State University. He is currently an Associate Professor at Shizuoka University, where his research focuses on modeling properties of human vision. He is as an Associate Editor for the IEEE TIP and the Journal of Electronic Imaging.