GPGPU based Implementation

of BLIINDS-II NR-IQA

by

Aman Yadav

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2016 by the
Graduate Supervisory Committee:

Sohum Sohoni, Chair
Daniel Aukes
Sangram Redkar

ARIZONA STATE UNIVERSITY

August 2016

ABSTRACT

The technological advances in the past few decades have made possible creation and consumption of digital visual content at an explosive rate. Consequently, there is a need for efficient quality monitoring systems to ensure minimal degradation of images and videos during various processing operations like compression, transmission, storage etc. Objective Image Quality Assessment (IQA) algorithms have been developed that predict quality scores which match well with human subjective quality assessment. However, a lot of research still remains to be done before IQA algorithms can be deployed in real world systems. Long runtimes for one frame of image is a major hurdle. Graphics Processing Units (GPUs), equipped with massive number of computational cores, provide an opportunity to accelerate IQA algorithms by performing computations in parallel. Indeed, General Purpose Graphics Processing Units (GPGPU) techniques have been applied to a few Full Reference IQA algorithms which fall under the. We present a GPGPU implementation of Blind Image Integrity Notator using DCT Statistics (BLIINDS-II), which falls under the No Reference IQA algorithm paradigm. We have been able to achieve a speedup of over 30x over the previous CPU version of this algorithm. We test our implementation using various distorted images from the CSIQ database and present the performance trends observed. We achieve a very consistent performance of around 9 milliseconds per distorted image, which made possible the execution of over 100 images per second (100 fps).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF NOMENCLATURES

Nomenclature	Definition
α	Normalizing Parameter in Gaussian Density Model
β	Scaling Parameter in Gaussian Density Model
γ	Shape Parameter in Gaussian Density Model
μ	Mean in Gaussian Density Model
$\hat{\boldsymbol{I}}_{in}$	Input Distorted Image as a Float Array of Size $n \times m$
$\hat{\boldsymbol{I}}_{pad}$	Padded Image of Size $(n+5) \times (m+5)$
$\widehat{\boldsymbol{C}}$	Local 5 × 5 Pixel Block Cropped from $\hat{\boldsymbol{I}}_{pad}$
$\hat{\boldsymbol{I}}_{crp}$	Array of all Blocks $\widehat{\boldsymbol{C}}$ with the Overlap Policy as shown in Figure 4
$\widehat{\boldsymbol{D}}$	2D 5 × 5 DCT Coefficients Corresponding to $\widehat{\boldsymbol{C}}$
$\hat{\boldsymbol{I}}_{dct}$	Array of Linearized Blocks of $\widehat{\boldsymbol{D}}$ Corresponding to each $\widehat{\boldsymbol{C}}$ in $\hat{\boldsymbol{I}}_{crp}$
$\hat{\boldsymbol{\gamma}}$	Array of γ for each $\widehat{\boldsymbol{D}}$ in $\hat{\boldsymbol{I}}_{dct}$
ζ	Coefficient of Frequency Variation Parameter of BLIINDS-II
$\hat{\boldsymbol{\zeta}}$	Array of ζ for each $\widehat{\boldsymbol{D}}$ in $\hat{\boldsymbol{I}}_{dct}$
SE	Energy Subband Ratio Parameter of BLIINDS-II
$\widehat{\boldsymbol{SE}}$	Array of SE for each $\widehat{\boldsymbol{D}}$ in $\hat{\boldsymbol{I}}_{dct}$
OR	Orientation Model Parameter of BLIINDS-II
$\widehat{\boldsymbol{OR}}$	Array of OR for each $\widehat{\boldsymbol{D}}$ in $\hat{\boldsymbol{I}}_{dct}$
$\hat{\boldsymbol{\gamma}}_{sorted}$	$\hat{\boldsymbol{\gamma}}$ Array Sorted in Descending Order
$\hat{\boldsymbol{\zeta}}_{sorted}$	$\hat{\boldsymbol{\zeta}}$ Array Sorted in Ascending Order
$\widehat{\boldsymbol{SE}}_{sorted}$	$\widehat{\boldsymbol{SE}}$ Array Sorted in Ascending Order
$\widehat{\boldsymbol{OR}}_{sorted}$	$\widehat{\boldsymbol{OR}}$ array sorted in ascending order
$\widehat{\boldsymbol{G}}$	3 × 3 Gaussian Filter
$\hat{\boldsymbol{I}}_{conv}$	Image Array obtained after Convolution of $\hat{\boldsymbol{I}}_{in}$ with $\widehat{\boldsymbol{G}}$
$\hat{\boldsymbol{X}}$	Array of Features Extracted from Input Image

Chapter 1. INTRODUCTION

Image Quality Assessment (IQA) refers to the quantification of the visual quality of an image subject to be viewed or used by a human. The first difficulty with that is the fact that image quality is a very subjective topic and the quality of the same image maybe perceived differently by different individuals. Surveys and studies have been conducted over large databases of images with representative human pools to draw out some empirical correlations between images and their quality as perceived by humans. The response of human subjects is averaged after certain adjustments and reported as Mean Opinion Scores (MOS) or Differential Mean Opinion Scores (DMOS). These scores provide the means to compare the perceived quality prediction accuracy of any IQA algorithm (Chandler 2013). A well performing IQA algorithm is expected to predict the correct MOS or the DMOS value of the distorted images in any image quality database. The performance of IQA algorithms are compared based on: (1) Prediction accuracy, (2) Prediction monotonicity and (3) Prediction consistency as recommended by Video Quality Experts Group (VQEG 2003).

Based upon the availability of a reference image, IQA algorithms can typically be classified into a) Full Reference (FR), b) No Reference (NR), and c) Reduced Reference (RR). Full reference IQA algorithms require both the undistorted and the distorted image as input. A simple and computationally easy FR-IQA algorithm calculates the mean squared error (MSE) between the two images using the error value at each pixel location. No reference IQA algorithms on the other hand do not require an undistorted source image at all, which makes it all the more difficult to design such an algorithm. But it is also closer to many real life applications where the undistorted image may not always be

available. Reduced Reference IQA algorithms lie somewhere in between in the sense that they do not use the entire undistorted image, but rather they require some features extracted from the original image and use that to make comparisons with the distorted image. Blind Image Integrity Notator using DCT coefficients, BLIINDS-II (Saad, Bovik and Charrier) is one such algorithm which falls under the no reference IQA paradigm and has been shown to correlate highly with subjective human judgment of image quality.

Several studies have quantified the relationship between the physical attributes of an image with their psychological and neurophysiological effects on humans (Ahumeda 1996, Ramos and Hemami 2001, Chandler and Hemami 2002, Wang and Simoncelli 2004, Chandler, Lim and Hemami 2006). They help us understand the Human Visual System (HVS) better and model IQA algorithms based on this knowledge. It has been shown (Chandler 2013) that better HVS modeling can lead to development of IQA algorithms with a higher quality prediction accuracy and greater robustness for changing visual signals.

One of the most important issues holding IQA algorithms back from practical widespread application is their poor computational performance. The best performing IQA algorithms (Phan et al. 2014) in literature require execution time of the order of seconds for a single image smaller than one megapixel. They have been designed with a focus on better quality score prediction, rather than their computational efficiency, which is also equally important for their real world application. Incorporating a better model of the HVS in new IQA algorithms will definitely result in an increase in the computational

complexity. So we have an even stronger case for analyzing and improving the computational efficiency of existing IQA algorithms.

Most image processing algorithms (including IQA algorithms) exhibit high data level parallelism, as they all perform many computational operations repeated over blocks of pixels. This property of IQA algorithms makes them ideal candidates for a General Purpose Graphics Processing Unit (GPGPU) implementation. A GPU with its massively parallel architecture is capable of performing the same computational operation on multiple data simultaneously (SIMD), thereby reducing the run time through clever management of compute and memory resources.

Chapter 2. RELATED WORK

In this chapter, we talk about the achievements made in recent years to accelerate the run time performance of IQA algorithms. Three possible ways to attain this are: a) algorithm modification, b) reducing computational complexity, and c) hardware resource conscious optimizations. We look into the various works related to IQA acceleration and discuss their technique used, results obtained and contributions made.

Yang (2008) have presented positive perform gains by implementing traditional image processing algorithms like histogram equalization, edge detection, DCT etc. by using NVidia's CUDA (Compute Unified Device Architecture) technology for GPGPU computations. The massive data parallel cores on a GPU showed great promise for image processing algorithms.

Gordon et al. (2010) have tried to accelerate the execution of PSNR algorithm by exploring GPGPU implementations using both CUDA and OpenGL.

Chen and Bovik (2013) presented Fast SSIM and Fast MS-SSIM and demonstrated impressive speed up in the run time for SSIM and MS-SSIM (Wang et al. 2003) by applying algorithm level modifications while preserving the predictive performance for both the algorithms. They reported seedups of 2.7x and 10x for SSIM and MS-SSIM just by implementing their modifications. They further reported speed ups of 5x and 14x using Intel SSE2 (SIMD) instructions to compute contrast and structure components. Adding parallelization via multithreading brought their final speed up to 17x and 50x for Fast SSIM and Fast MS-SSIM respectively.

A GPGPU based implementation of SSIM, MS-SSIM and CVQM (Okarma 2010) was presented by Okarma and Mazurek (2011) and reported an execution time reduction by 150x and 35x for SSIM and MS-SSIM respectively. They handled the GPU and CPU memory bandwidth bottleneck by utilizing GPU registers to store computed data locally on the GPU, thereby minimizing the number of data transfers back and forth.

In 2012 Phan et al. presented an optimization for Most Apparent Distortion (MAD) (Larson and Chandler 2013) algorithm. They proposed four techniques including the use of integral images for local statistical computations and the use of GPGPU for calculating the required forty log-Gabor filters. An acceleration of 47x was reported in the run time of MAD after all the four suggested optimizations were implemented.

Phan et al. (2014) further performed a micro architectural analysis of four Full Reference IQA algorithms (MAD, MS-SSIM, VIF and VSNR) and two No Reference IQA algorithms (BLIINDS and BRISQUE) on an Intel based general purpose computing platform. All the algorithms follow similar main stages of filtering (transforming) and further statistical operations. They reported the various kinds of memory and computational bottlenecks encountered for the different algorithms for various kinds of distortions of an image and proposed various platform specific code optimizations to overcome the observed bottlenecks. Based on their analysis they also proposed a custom IQA engine framework to efficiently handle the common IQA algorithm bottlenecks.

Josh (2015) presented a GPGPU implementation and its analysis for MAD. He reported a 24.6x speedup over the CPU implementation. Additionally, he presented an implementation with multiple GPUs by executing independent task sequences on three

different GPUs. The multi GPU approach fetched another 1.74x speedup over the single GPU implementation, which was reported to be lesser than expected. Splitting the task over multiple GPUs necessitated transfer of data between the GPUs and the CPU across PCIe bus, which resulted in a lower than expected speedup.

Chapter 3. BLIINDS-II ALGORITHM

The BLIINDS algorithm takes in only the distorted image as input and processes it to arrive at a final quality score corresponding to it. This is according to the philosophy of No Reference Image Quality Assessment algorithm (NR-IQA) to process only a distorted image without 'referring' to any pure, clean version of it. Figure 1 gives a broad overview of the flow of BLIINDS-II algorithm.

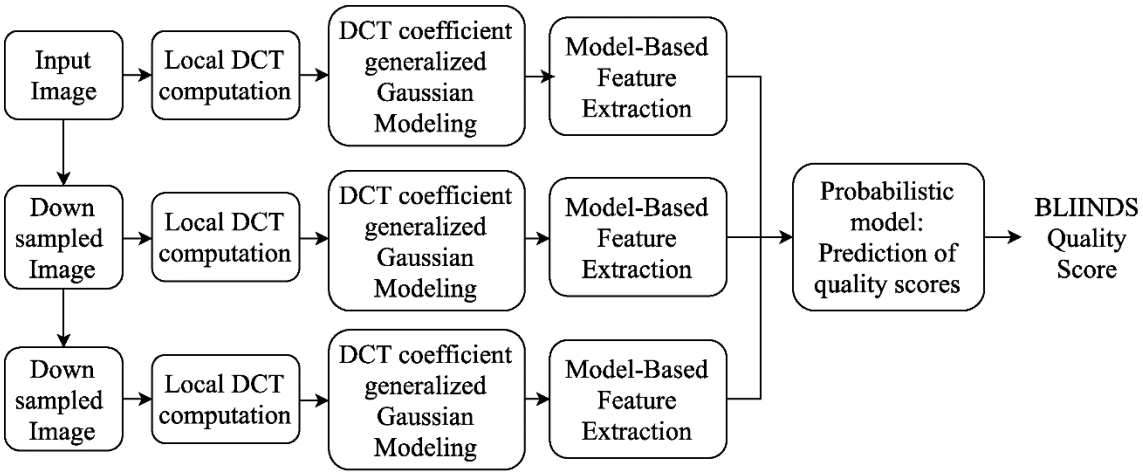

Figure 1. High-Level Overview of the BLIINDS-II Framework.

The algorithm runs on three different spatial scales of the image, in accordance with the HVS property of spatial local decomposition of visual stimulus as has been shown in (Blake and Sekuler 2006). The flow of BLIINDS-II can be classified into four distinct steps.

First, the input image is padded with zeros and then partitioned into equally sized blocks of 5x5 pixels, called local image patches. Local 2-D DCT is then computed for each of these local patches to find block DCT coefficients in order to mimic the Human Visual System's (HVS) property of local spatial visual processing.

The second step of the BLIINDS pipeline applies a univariate generalized Gaussian density model to the non-DC coefficients of each block. The generalized Gaussian density model is given by

$$f(x|\alpha,\beta,\gamma) = \alpha e^{-(\beta|x-\mu|)^{\gamma}} \tag{3.1}$$

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

$$\alpha = \frac{\beta\gamma}{2\Gamma(1/\gamma)} \tag{3.2}$$

$$\beta = \frac{1}{\sigma}\sqrt{\frac{\Gamma(3/\gamma)}{\Gamma(1/\gamma)}} \tag{3.3}$$

Where σ is the standard deviation and Γ denotes the gamma function given by

$$\Gamma(z) = \int_0^{\infty} t^{z-1}\, e^{-t} dz \tag{3.4}$$

This Gaussian model is also applied to specific partitions within each block of local DCT coefficients across different orientations. These orientations are discussed in more detail later. A generalized Gaussian fit is obtained for each of the orientations and radial sub regions.

The third step of the algorithm consists of extracting the required features from the Gaussian model parameters developed in the previous step. Eight features are extracted in this way from the input distorted image at the original scale. Further, sixteen more features are extracted from the image by down sampling the image once and pushing it

through the BLIINDS-II pipeline to extract eight more features and down sampling again to extract another eight features.

In the final step of the algorithm, a simple Bayesian inference approach is used to predict the image quality score from the extracted features. The Bayesian approach essentially maximizes the probability that the distorted image has a certain quality score given the model-based features. This step is discussed in more detail in a later section.

The entire run of the algorithm is represented below step by step. Understanding these steps is crucial to understand our GPU implementation details, its analyses and optimizations. For a better understanding of the underlying theoretical concepts and experimental results of the BLIINDS-II algorithm, the reader should refer to (Saad 2012). The focus of our work is on the acceleration of this high performing NR-IQA algorithm by studying its interaction with the implementing hardware to aid the deployment of IQA techniques in real-world time-sensitive applications.

In the next sections, we provide the working details of the BLIINDS-II algorithm and then move on to discuss operations/routines in the algorithm that are deserve a deeper discussion.

3.1 BLIINDS-II algorithm steps

Here, we provide a description of the computational and mathematical operations in each stage of the BLIINDS-II algorithm. The algorithm proceeds as follows-

1) Read the input image as $\widehat{\boldsymbol{I}}_{in}$ and let the size of it be $n \times m$.

2) Pad image with zeros as shown in Figure 2.

The size of the padded image is $(n + 5) \times (m + 5)$. The 5×5 pixel blocks used in the algorithm have been designed to have 1-pixel width of overlap on each side. We refer to this width as $overlap\ width$. As can be seen in Figure 2, the pad width on the left and the top of the image is equal to the $overlap\ width$ in 5×5 pixel block. The crop window is explained in the next step. On the right and the bottom of the image, the pad width is $(5 - overlap\ width)$. It should be noted that 5 is equal to the width or height of 5×5 pixel block.

$$\hat{\boldsymbol{I}}_{in} \rightarrow \hat{\boldsymbol{I}}_{pad}$$

3) Crop the padded image in blocks of 5×5 with an overlap of one pixel on each side. A cropped block is represented by $\widehat{\boldsymbol{C}}$. The 5×5 crop window proceeds in the manner shown in Figure 4. Let $\hat{\boldsymbol{I}}_{crp}$ represent a vector of all such cropped blocks.

Therefore,

$$\hat{\boldsymbol{I}}_{crp} = \left[\widehat{\boldsymbol{C}}_1, \widehat{\boldsymbol{C}}_2, \widehat{\boldsymbol{C}}_3, \dots, \widehat{\boldsymbol{C}}_{\left(\frac{n}{3}+1\right)\times\left(\frac{m}{3}+1\right)}\right]$$

4) Compute 2D 5×5 DCT of each cropped block as shown in Figure 5. Let a 5×5 DCT block of coefficients be represented by $\widehat{\boldsymbol{D}}$ and the vector containing all such DCT blocks be $\hat{\boldsymbol{I}}_{dct}$. Therefore,

$$\hat{\boldsymbol{I}}_{dct} = \left[\widehat{\boldsymbol{D}}_1, \widehat{\boldsymbol{D}}_2, \widehat{\boldsymbol{D}}_3, \dots, \widehat{\boldsymbol{D}}_{\left(\frac{n}{3}+1\right)\times\left(\frac{m}{3}+1\right)}\right]$$

5) Apply univariate generalized Gaussian density model to the obtained DCT coefficients of each block. The Gaussian density model is also applied to two specific partition schemes within each block. This partitioning is aimed at capturing the

orientation data in the image. A total of four model parameters are obtained for each block of DCT coefficients. The model parameters are represented by,

GGD Model Shape Parameter (γ),

$$\widehat{\boldsymbol{\gamma}} = \left[\gamma_1, \gamma_2, \gamma_3, \dots, \gamma_{\left(\frac{n}{3}+1\right)\times\left(\frac{m}{3}+1\right)}\right]$$

Coefficient of Frequency Variation Parameter (ζ),

$$\widehat{\boldsymbol{\zeta}} = \left[\zeta_1, \zeta_2, \zeta_3, \dots, \zeta_{\left(\frac{n}{3}+1\right)\times\left(\frac{m}{3}+1\right)}\right]$$

Energy Subband Ratio Parameter (SE),

$$\widehat{\boldsymbol{SE}} = \left[SE_1, SE_2, SE_3, \dots, SE_{\left(\frac{n}{3}+1\right)\times\left(\frac{m}{3}+1\right)}\right]$$

Orientation Model based Parameter (OR),

$$\widehat{\boldsymbol{OR}} = \left[OR_1, OR_2, OR_3, \dots, OR_{\left(\frac{n}{3}+1\right)\times\left(\frac{m}{3}+1\right)}\right]$$

A more detailed explanation for calculating each of the four parameters will be provided later.

6) Sort, in ascending order, each of the vectors containing the model parameters for the input image.

$$\widehat{\boldsymbol{\gamma}} \xrightarrow{sort} \widehat{\boldsymbol{\gamma}}_{sorted}$$

$$\widehat{\boldsymbol{\zeta}} \xrightarrow{sort} \widehat{\boldsymbol{\zeta}}_{sorted}$$

$$\widehat{\boldsymbol{SE}} \xrightarrow{sort} \widehat{\boldsymbol{SE}}_{sorted}$$

$$\widehat{\boldsymbol{OR}} \xrightarrow{sort} \widehat{\boldsymbol{OR}}_{sorted}$$

7) Calculate the mean of first 10% of all the values in the sorted parameter vectors. Also calculate the mean of all the values for each parameter vector. Store all these eight

values as features ($\widehat{\boldsymbol{X}}$) extracted from the image.

$$\widehat{\boldsymbol{X}} = [\gamma_{m\_10}\ \gamma_{m\_100}\ \zeta_{m\_10}\ \zeta_{m\_10}\ SE_{m\_10}\ SE_{m\_100}\ OR_{m\_10}\ OR_{m\_100}]$$

where, the subscript m_10 denotes the mean of top 10 percentile values and m_100 denotes the mean of the top 100 percentile of the vector.

8) Apply the 3×3 Gaussian kernel ($\widehat{\boldsymbol{G}}$), to the input image $\hat{\boldsymbol{I}}_{in}$.

$$\widehat{\boldsymbol{G}} = \begin{bmatrix} 0.0113 & 0.0838 & 0.0113 \\ 0.0838 & 0.6193 & 0.0838 \\ 0.0113 & 0.0838 & 0.0113 \end{bmatrix}$$

$$\hat{\boldsymbol{I}}_{conv} = \hat{\boldsymbol{I}}_{in} \times \widehat{\boldsymbol{G}}$$

9) Subsample the convoluted image $\hat{\boldsymbol{I}}_{conv}$ by a factor of 2 to obtain a downsampled image of size $(n/2) \times (n/2)$.

10) Repeat steps 2-6 again for the downsampled image. Add eight more features to the original features vector for the input image at this scale.

11) Subsample the downsampled image and perform step 10 for it. We now obtain a features vector of size 24.

$$\widehat{\boldsymbol{X}} = \begin{bmatrix} \gamma_{m10_1} & \gamma_{m100_1} & \zeta_{m10_1} & \zeta_{m100_1} & SE_{m10_1} & SE_{m100_1} & OR_{m10_1} & OR_{m100_1} \\ \gamma_{m10_2} & \gamma_{m100_2} & \zeta_{m10_2} & \zeta_{m100_2} & SE_{m10_2} & SE_{m100_2} & OR_{m10_2} & OR_{m100_2} \\ \gamma_{m10_3} & \gamma_{m100_3} & \zeta_{m10_3} & \zeta_{m100_3} & SE_{m10_3} & SE_{m100_3} & OR_{m10_3} & OR_{m100_3} \end{bmatrix}$$

12) Compute the posterior probability given the set of derived features for all quality scores from 0 to 100 in increments of 0.5. This step is explained further in the section on the Prediction Model.

13) Find the quality score with the maximum posterior probability and return that as the BLIINDS quality score.

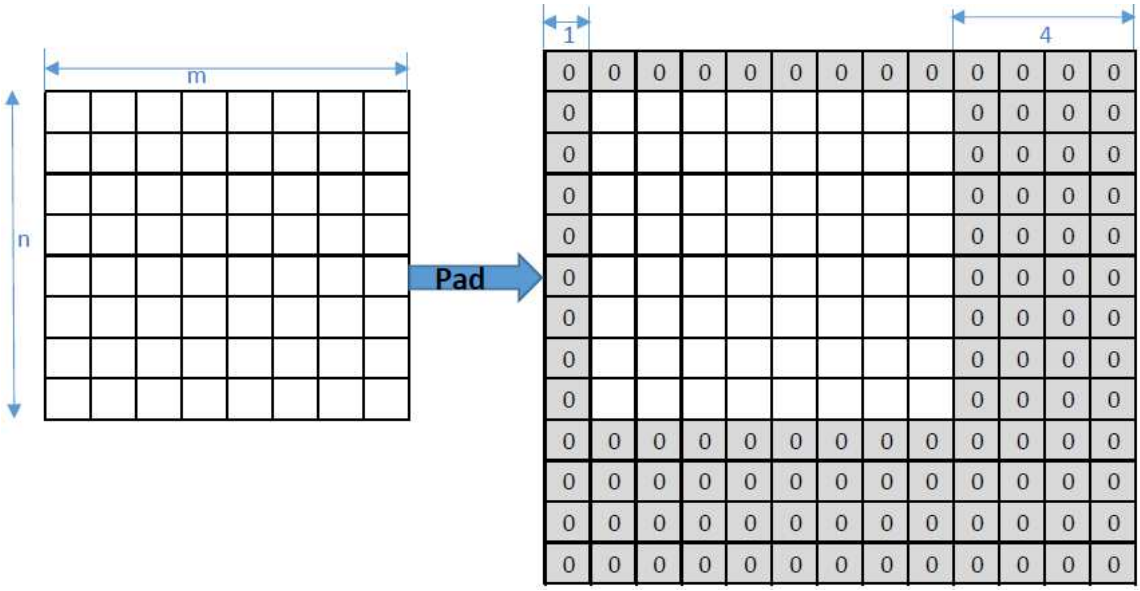

Figure 2. Padded Image.

3.2 *Statistical operations on DCT coefficients*

Now we discuss each of the four statistical operations on each 5×5 block of DCT coefficients for obtaining the model parameters is required for understanding the issues associated with their GPU implementation. For statistical modeling of the DCT coefficients, the random variable used is represented by X. The values of the random variables are the 24 non-DC values from the DCT coefficients.

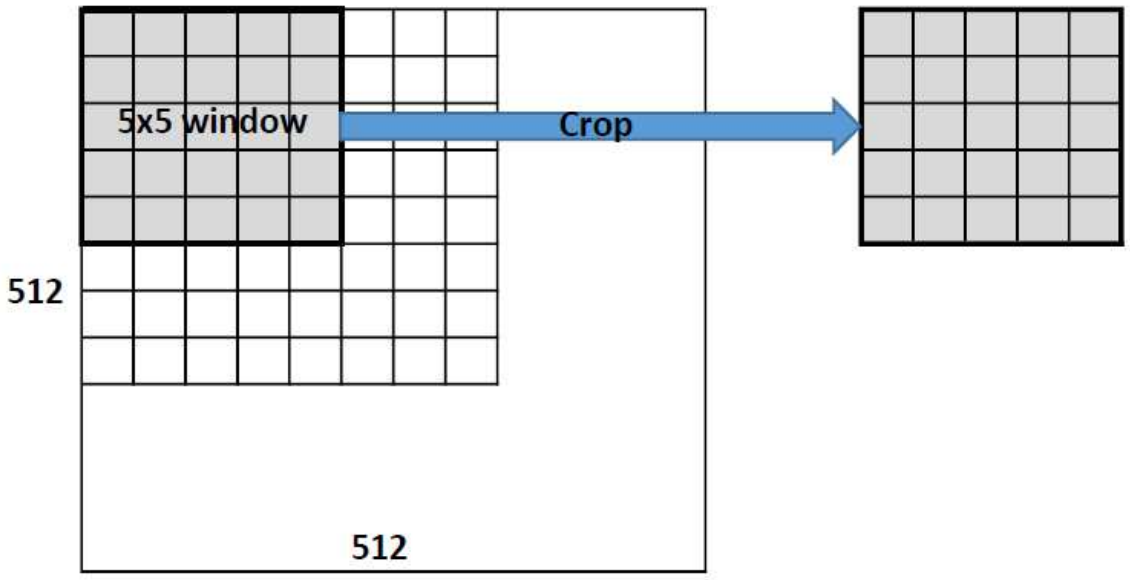

Figure 3. Local 5×5 Block of Pixels.

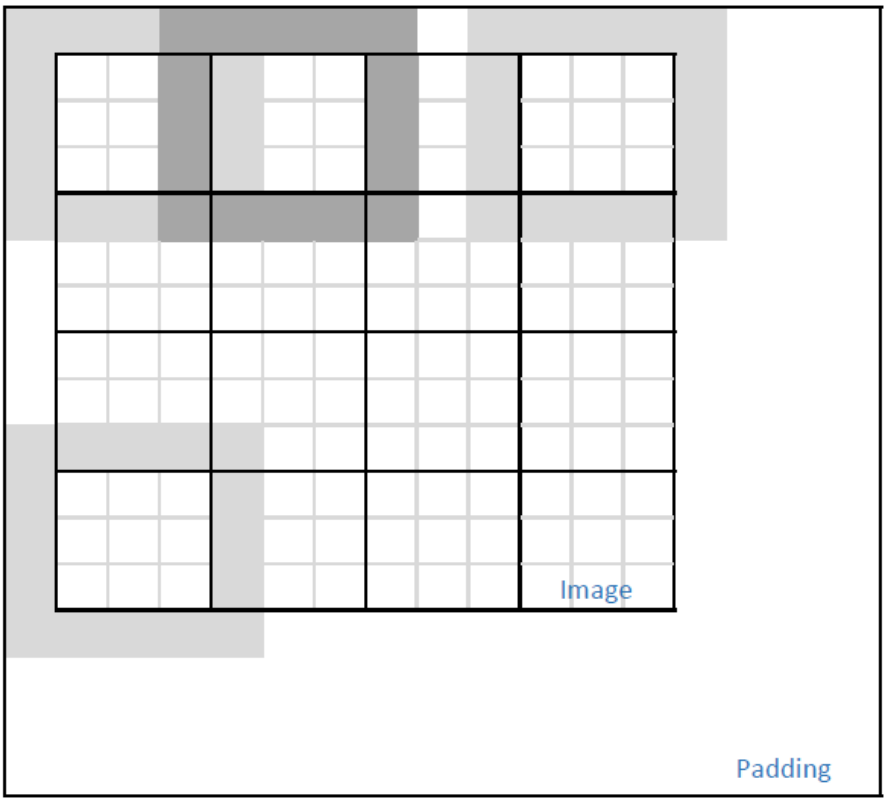

Figure 4. Overlapping Local 5x5 Blocks.

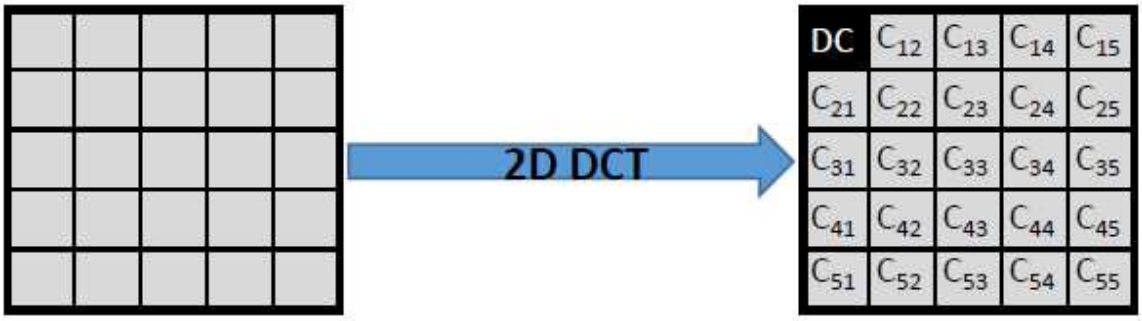

Figure 5. Apply Discrete Cosine Transform to Obtain Block of DCT Coefficients.

3.2.1 Gaussian Shape Parameter (γ)

In equation 5.1, the generalized Gaussian density model can be parametrized by β (scale parameter), μ (mean) and γ (shape parameter). The relationship used to calculate γ is given by

$$\frac{\Gamma(1/\gamma)\Gamma(3/\gamma)}{\Gamma^2(2/\gamma)} = \frac{\sigma_X^2}{\mu_{|X|}^2} \tag{3.5}$$

We compute the quantity on the right and then estimate the shape parameter, γ.

3.2.2 Coefficient of frequency variation (ζ)

This quantity is calculated from the following relation for each of the 5×5 DCT coefficient block.

$$\zeta = \frac{\sigma_{|X|}}{\mu_{|X|}} \tag{3.6}$$

Where $\sigma_{|X|}$ is the standard deviation of the absolute values of the random variable X and $\mu_{|X|}$ is the mean of the absolute values of the random variable X.

3.2.3 Subband energy ratio (SE)

The DCT coefficients are divided into three radial bands as shown in Figure 6. This statistical operation measures the relative energy stored in each of the radial band. Let Ω_n represent the set of DCT coefficients in band n, where $n = 1, 2, 3$. The energy E_n in Ω_n is given by

$$E_n = \sigma_n^2 \tag{3.7}$$

Where σ_n is the standard deviation of the values in Ω_n.

Next we compute the quantity R_n, which has been defined as the ratio of difference between energy contained in the radial frequency band Ω_n and the average energy contained in radial frequency bands up to Ω_n to the sum of these two quantities. This ratio is given by

$$R_n = \frac{|E_n - \frac{1}{n-1}\sum_{j<n} E_j|}{E_n + \frac{1}{n-1}\sum_{j<n} E_j}\ \sigma_n^2 \tag{3.8}$$

As can be seen from the equation, the quantity R_n is defined only for $n = 2, 3$. The values R_2 and R_3 are calculated and their mean is recorded as the subband energy ratio parameter of the statistical model for each 5×5 block of DCT coefficients.

DC	C_{12}	C_{13}	C_{14}	C_{15}
C_{21}	C_{22}	C_{23}	C_{24}	C_{25}
C_{31}	C_{32}	C_{33}	C_{34}	C_{35}
C_{41}	C_{42}	C_{43}	C_{44}	C_{45}
C_{51}	C_{52}	C_{53}	C_{54}	C_{55}

Figure 6. DCT Coefficients Grouped into Radial Subbands of Energy.

3.2.4 Oriented model parameter (OR)

The coefficient of frequency variation, ζ is calculated for each of the oriented set as shown in Figure 7. The variance of these three ζ values is stored as oriented model parameter for each 5×5 block of DCT coefficients.

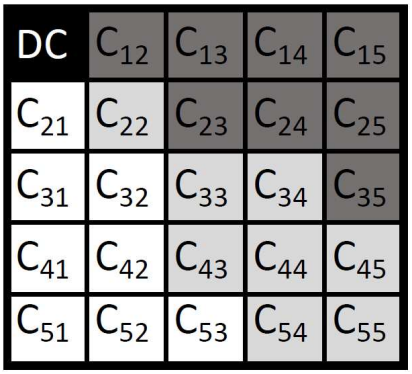

Figure 7. DCT Coefficients Grouped along three Orientations.

3.3 Prediction Model:

The prediction model maps from the feature space to image quality score space. Therefore, before computing the BLIINDS-II score, each of the four statistical model parameters discussed in the previous section is pooled in two ways: a) lowest 10^{th} percentile average, and b) 100^{th} percentile average or mean. Performing this operation at each spatial scale gives us 24 features for the distorted image.

Let $\widehat{\boldsymbol{X}} = [x_1, x_2, \ldots, x_{24}]$ be the vector of extracted features. We next calculate the probability the quality score is s corresponding to the features vector $\widehat{\boldsymbol{X}}$. Mathematically, this can be represented by $P(s|\boldsymbol{X})$ and read as the probability of quality score being s, given the features vector $\boldsymbol{X}$. We calculate this probability for each possible quality score s, ranging from 0 to 100 in increments of 0.5. The quality score S, in this range from 0 to

100, with the highest computed probability is returned as the BLIINDS image quality score.

Next, we will like to discuss the computation of the quantity $P(s|\boldsymbol{X})$. For this, we append s to the vector $\widehat{\boldsymbol{X}}$ as its 25th value, such that $\widehat{\boldsymbol{X}} = [x_1, x_2, \dots, x_{24}, s]$. Further, this new vector is plugged in to equation 3.9 and the obtained number is the required probability $P(s|\boldsymbol{X})$. The quantities α, β, γ, μ and Σ have been obtained by Support Vector Regression (SVR).

$$f(x|\alpha,\beta,\gamma) = \alpha e^{-\left(\beta(\mathbf{x}-\mu)^{\mathbf{T}}\Sigma^{-1}(\mathbf{x}-\mu)\right)^{\gamma}} \tag{3.9}$$

Chapter 4. CUDA Implementation of BLIINDS-II

4.1 Introduction

In this section we describe the specific approach we took to implement each step of the BLIINDS-II pipeline. Each of the intermediate step is implemented as either a CPU function or a GPU kernel. We have designed our CUDA implementation such that the distorted image is first cast as a linear array of floats on the CPU and then copied across to the GPU via PCIe bus. The GPU then performs the DCT model based feature extraction and copies the features array back to the CPU. It should be noted that the extracted features are an array of 24 floats. From there on the CPU computes the BLIINDS quality score from the obtained features.

We also present a comparative description on different versions of each GPU kernel we implemented. Each of these versions were analyzed for their run time performance.

4.2 CUDA implementation

A detailed flowchart describing our CUDA implementation of BLIINDS-II has been shown in Figure 9. The flow is divided into three main parts:

a) CPU: Distorted image is read and cast as float.
b) GPU: The DCT and statistical modeling is performed across the image on three spatial scales.
c) CPU: Using the features array populated by the GPU, the BLIINDS quality score is calculated.

As can be seen, we have tried to minimize the number of transactions across the PCIe bus, which is a known source of potential bottleneck.

In the following subsections, we describe each of these in detail. The description of the GPU part of the code is split in two subsections; the first subsection describes the kernels for computing the model based features at a particular spatial scale, the second subsection describes the implementation of the downsample kernels on the GPU.

GPGPU Implementation details:

4.2.1 *Image read on CPU*

The input distorted image is first read on the CPU as a 2D array of 8-bit unsigned char using OpenCV and then is converted into a 1D array (linearization) of floats (typecasting from unsigned char to float). This 1D array of floats is then transferred to the GPU's DRAM across the PCIe bus as can be seen in Figure 7. For a single channel input image size of 512x512, this transfer across the PCIe bus is 1MB.

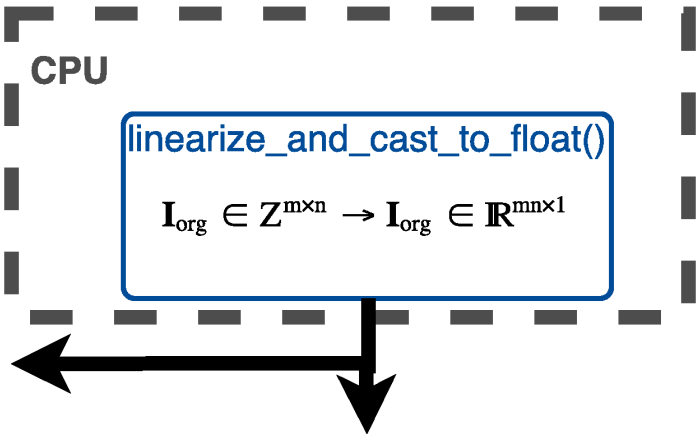

Figure 8. Preprocessing the Distorted Image on the CPU.

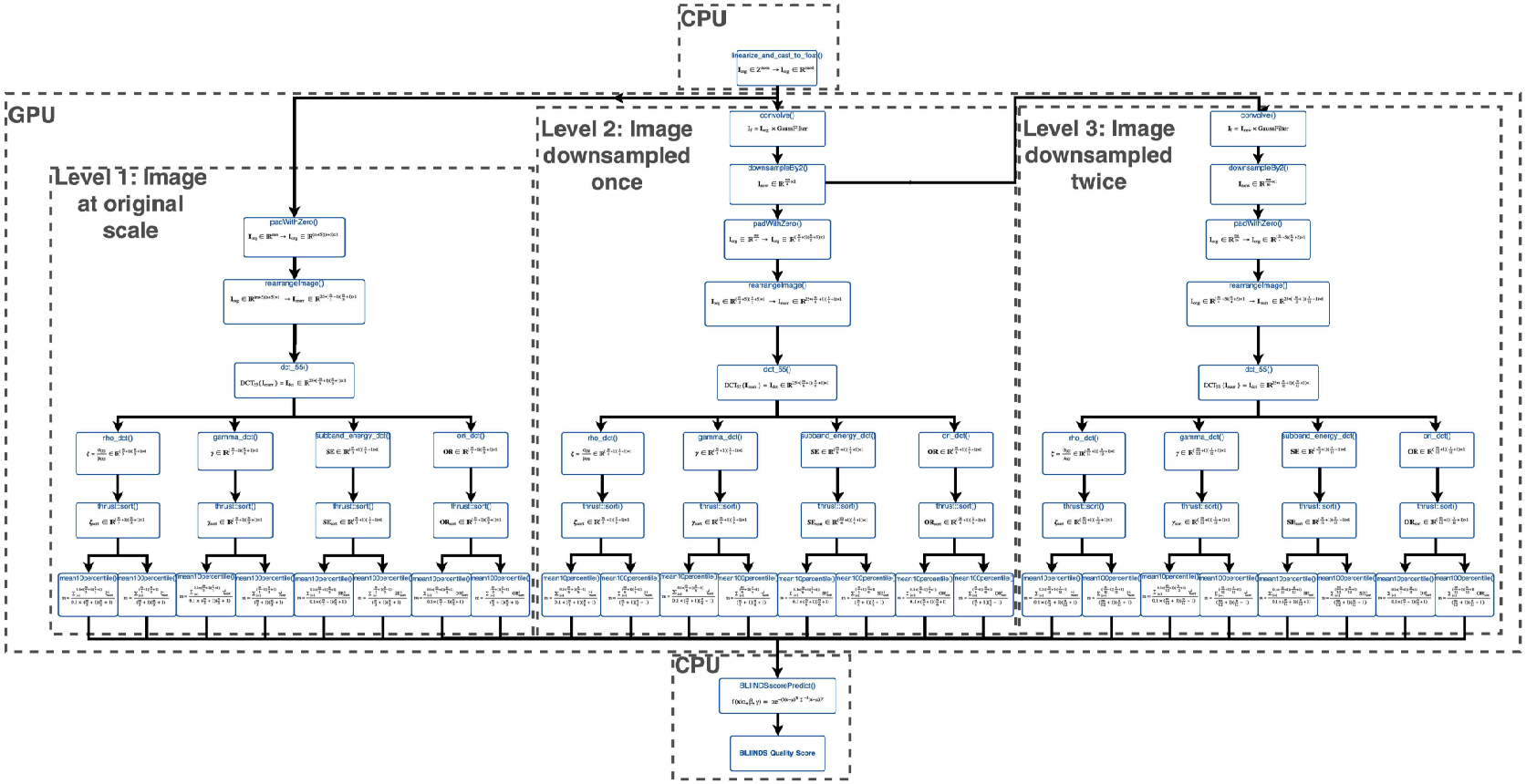

Figure 9. Flowchart of the GPGPU Implementation.

4.2.2 *Feature extraction at original spatial scale*

The computations on the GPU are divided into three sections, one for each scale level of the image. Figure 8 above shows the flow and size of data in various kernels.

4.2.2.1 padWithZero()

This kernel pads the input image with zeros as shown in Figure 2. The image array adds a width and a height of 5 elements each. This kernel is called with a grid size of $n + 5$ and a block size of $m + 5$, where $n \times m$ is the size of image. We add 5 because it is equal to the size of local square DCT blocks used in the algorithm. The kernel employs a thread index based condition to determine if it copies data from the input image or copies a 0 for the padding. This if statement is expected to lend some amount of thread divergence in the kernel.

4.2.2.2 rearrangeImage()

This kernel rearranges the linear array of image in order to place elements within the same 5x5 block to fall in contiguous memory locations. In other words, the kernel first forms a linear array of 25 elements within a 5x5 block, and then stores all such 25 element sequentially in global memory.

The image was rearranged according to 5x5 blocks of pixels. This was done to coalesce global memory reads for each thread in a warp as described in NVidia blog page (Harris 2013). Both the transform and the statistical operations on the distorted image are confined to local 5x5 blocks, so this new rearranged image array could be used by the ensuing kernels. It should be noted that this results in duplication of image data in the

rearranged, but aligns global memory reads within a warp perfectly. This duplication of data increases the size of image array (512x512) by a factor of 115.

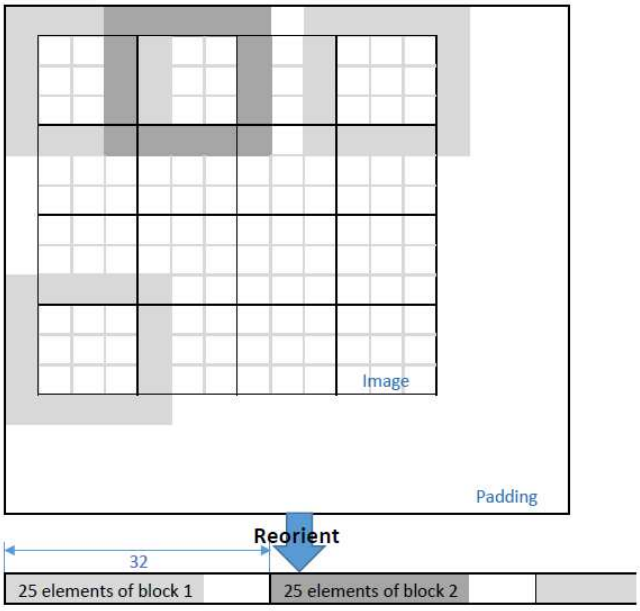

Figure 10. Reorient the Image Array to align Memory Accesses by each Warp on the GPU.

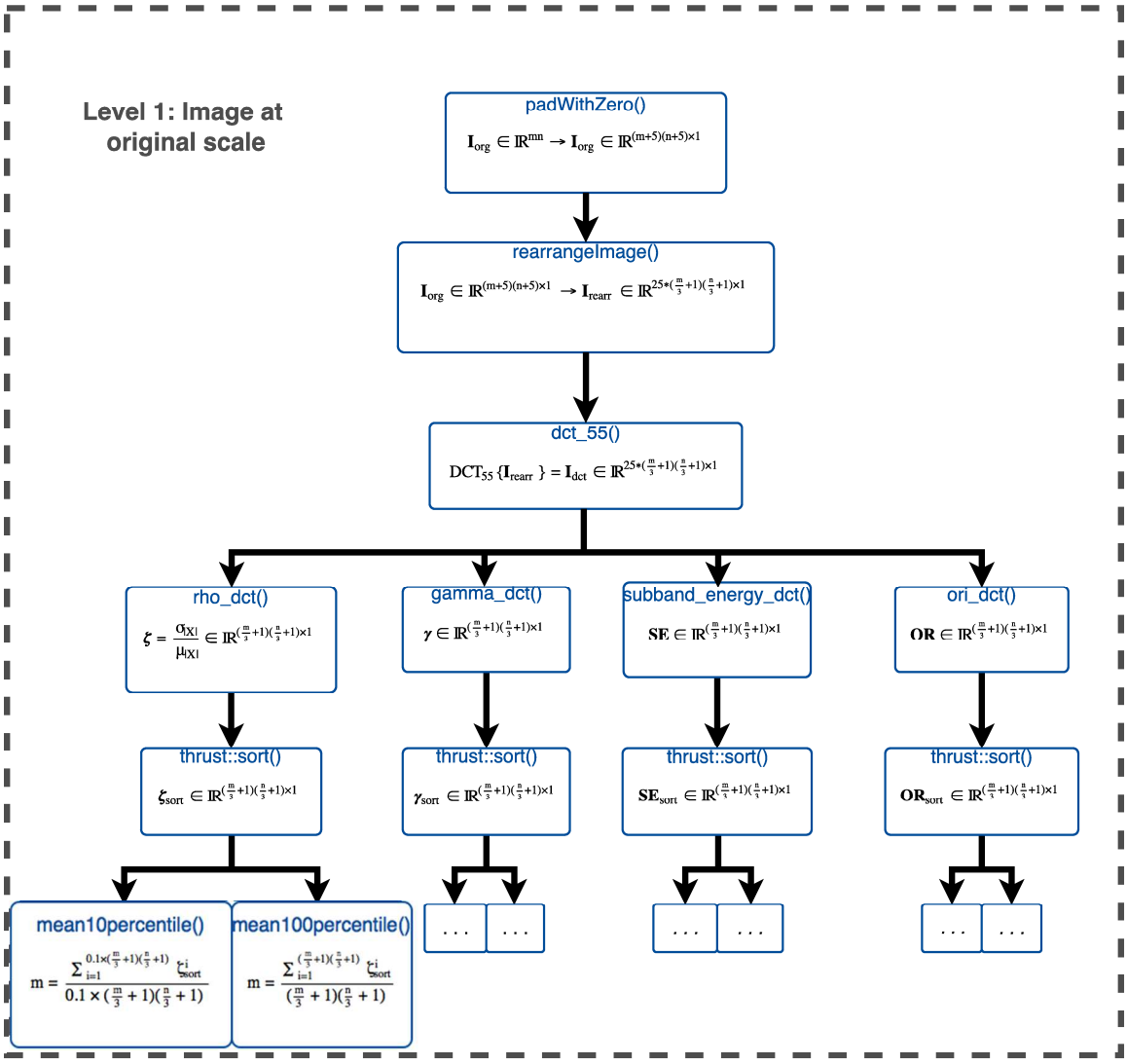

Figure 11. DCT and Statistical Operations at Original Scale (level 1) of Image.

4.2.2.3 Dct_55()

This kernel applies 2D 5x5 DCT to each of the individual 5x5 blocks, stored in contiguous memory by the previous kernel. We implemented two versions of local DCT computation, first using cuFFT library and the second using DCT matrix multiplication.

4.2.2.3.1 DCT version 1

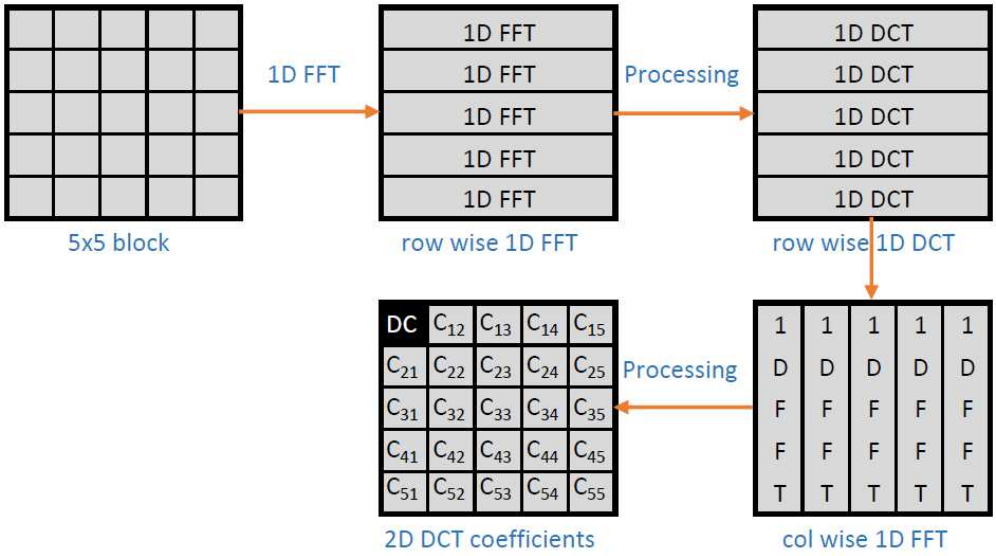

Figure 12. Computation of 2D DCT using two 1D FFTs.

The first method makes use of DFT to compute a DCT. We chose this path because CUDA provides a bundled library in the form of cuFFT to fast computation of DFT, while a tailor made API for DCT is not available. To compute a 2D DCT we proceed in the following manner:

1) Calculate row wise FFT in the 5 × 5 block. The row array is mirrored to produce a 10 element array. A 10 point FFT is performed over this modified row array.
2) Pick only the first 5 elements from the 10 point FFT and multiply the appropriate twiddle factor to map the FFT coefficient to DCT.
3) Repeat the steps 1 and 2 column wise instead of row wise.

4.2.2.3.1 DCT version 2

For the second method, we decided to obtain the DCT coefficient matrix by using matrix multiplication method. We obtained the DCT matrix from MATLAB using the command dctmtx() as shown below. It should be pointed out that using double precision for the

matrix multiplication is essential. Using single precision in matrix multiplication leads to inaccurate results.

Table 1. DCT Matrix for Matrix Multiplication Method.

0.4472	0.4472	0.4472	0.4472	0.4472
0.6015	0.3717	0	-0.3717	-0.6015
0.5117	-0.1954	-0.6325	-0.1954	0.5117
0.3717	-0.6015	0	0.6015	-0.3717
0.1954	-0.5117	0.6325	-0.5117	0.1954

4.2.2.4 Rho_dct()

This kernel models each of set of 5x5 DCT coefficients to univariate generalized Gaussian density given in equation (3.1). The kernel computes the quantity $\frac{\sigma_{|X|}}{\mu_{|X|}}$ (coefficient of frequency variation) for each 5x5 DCT coefficient block and X represents all the non-DC coefficients. The ζ value computed for each block of DCT coefficient is stored sequentially in an array represented by $\boldsymbol{\zeta}$.

We have implemented this kernel to use shared memory to store local DCT coefficients. For the launch of the kernel, one warp is dedicated to each 5×5 block of DCT coefficients. This ensures that the coefficients are always copied in parallel from the global memory. Further summation of the values for calculating $\mu_{|X|}$ and $\sigma_{|X|}$ is implemented as sequential summation in a for loop. We have used the compiler directive `#pragma unroll` to unroll the loop which has to run only 24 iterations. We made sure to use only floating point multiplications and divisions, as there are three times more the

number of the single precision cores as compared to the double precision cores on our test GPU.

4.2.2.5 Gamma_dct()

The gama_dct function in baseline C++ implementation proceeds in the following manner:

- Calculates rho = mean / (variance + 0.000001)
- Linearly searches through a lookup table containing values of rho
- Returns the gama value from another lookup table corresponding to the rho value obtained in previous search.
- Both the lookup tables are 1D arrays of 9971 floats each.

CUDA implementations:

4.2.2.5.1 gama_dct using only global memory

The first naïve implementation of this kernel read each value from the look up table stored in global memory.

4.2.2.5.2 gama_dct using shared memory for lookup

The second implementation made use of shared memory by copying the look up table for rho in it. The low latency of shared memory reads resulted in a significant speed up over the previous version.

4.2.2.5.3 gama_dct using constant memory

In this version, we split gama_dct kernel into three smaller steps. The first kernel calculates the rho value corresponding to each block of DCT coefficients and stores them as an array of floats in global memory. We then sort this array containing rho values. The final kernel reads in the sorted rho array and employs binary search while traversing the look up table for rho.

4.2.2.6 Subband_energy_dct()

It computes ratios of energy stored in radial frequency bands. The block DCT coefficients are divided into three bands of radial frequencies (low, middle and high) as shown in Figure 9. The kernel stores the mean of R_2 and R_3 (explained in section 3.2.3) for each block of DCT coefficients in a contiguous array of floats, $\boldsymbol{SE}$.

We design this kernel such that one warp of threads is dedicated to each 5×5 block of DCT coefficients. All the computations of E_n for each of the three radial bands are done sequentially.

4.2.2.7 Ori_dct: It computes the orientation model parameter, OR for each DCT coefficient block as described in section 3.2.4. The computed values for each block are stored in an array of floats, $\boldsymbol{OR}$.

We implement two different versions of this kernel

4.2.2.7.1 Individual kernel for each orient

We call three different kernels, one each for each orient of the DCT coefficients block as shown in Figure 7. For each orient we calculate the ζ value just like it was calculated in section 4.2.2.4 above. We write these values in three different arrays stored in global

memory. These three kernels are followed by the ori_dct_final() kernel. This fourth kernel calculates the variance of the ζ value for each orient.

4.2.2.7.2 Merge as one kernel

We merge the four kernels in to one kernel. This obviates the need for writing the intermediate results to global memory. First we calculate the ζ value for each orient and then the variance in computed.

4.2.2.8 Thrust::sort(): Thrust library sort function is used to sort each array of model parameters ($\boldsymbol{\zeta, \gamma, SE, OR}$). It sorts the respective arrays using radix sort, and writes them back to the same memory location.

4.2.2.9 Mean10percentile() and mean100percentile(): Both of these functions are implemented using thrust::reduce(). After the arrays are sorted, thrust::reduce on 10% of the array length returns the sum of the lowest 10% values in the array. This number is then divided by 10% of the array size. Similarly, the mean of 100 percentiles is also calculated.

4.2.3 *Down sample image to lower spatial scales*

Further, the implementation proceeds with calculations at the second spatial scale, that is with a downdsampled image.

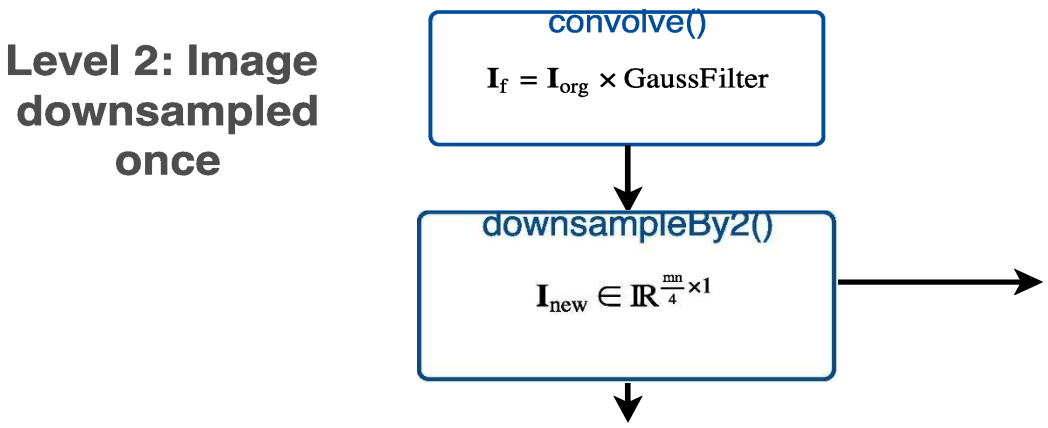

Figure 13. Convolution with GaussFilter Followed by downsampling for Level 2.

Convolve(): The input image at original scale is convolved with a 3x3 Gaussian filter. Since the Gaussian filter is separable, it has been implemented as two one dimensional filters of length 3. This had to be separated into two kernels, one for row-wise convolution and the other for column-wise convolution because the second convolution can start only after the first one completes, but there is no way to achieve a device wide synchronization. Both the row-wise convolution and the column-wise convolution are gather operations.

downSampleBy2(): This kernel copies over the filtered image as shown in Figure 12.

Further all the computations described in level 1 are repeated for this new downsampled image to obtain next eight features at the new spatial scale. Another downsampling and feature extraction operation is performed for level 3 computations.

4.2.4 *Compute BLIINDS quality score*

All the 24 features are sent across the PCIe over to the CPU's RAM. This has been

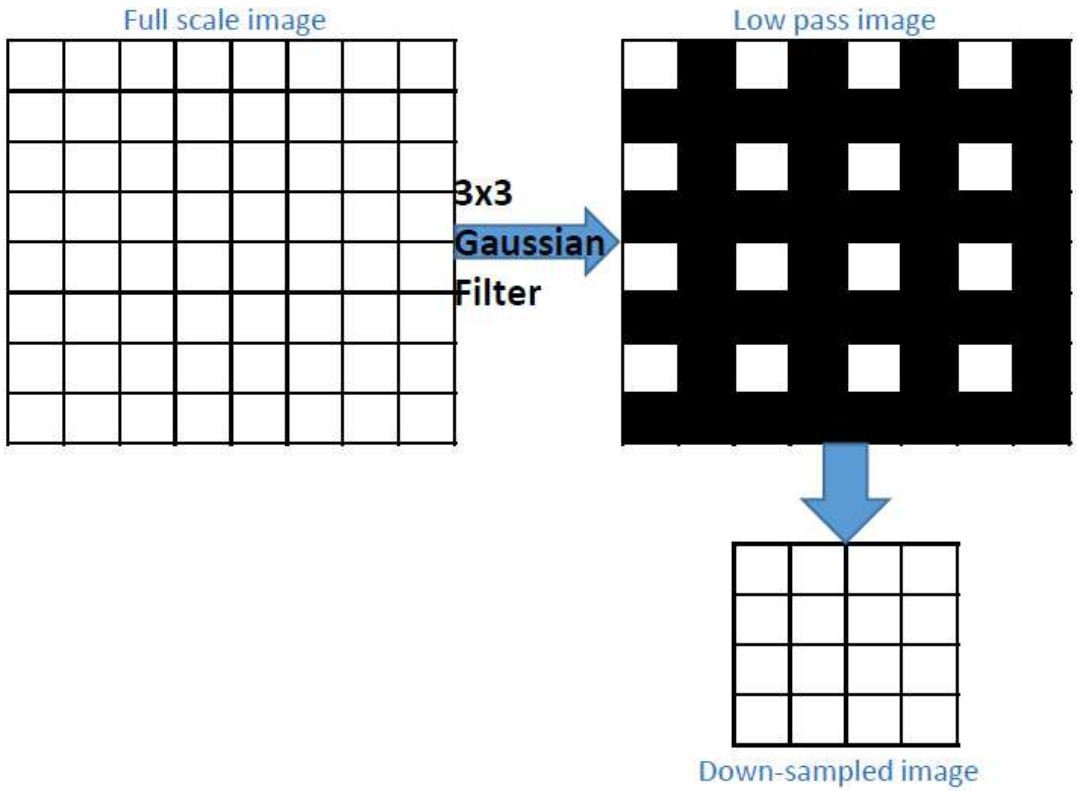

Figure 8. Image Convolution and Downsampling.

designed so because thrust::reduce() returns the result of reduction on GPU arrays to CPU. Further BLIINDSscorePredict() function is called to arrive at BLIINDS quality score from the extracted set of features.

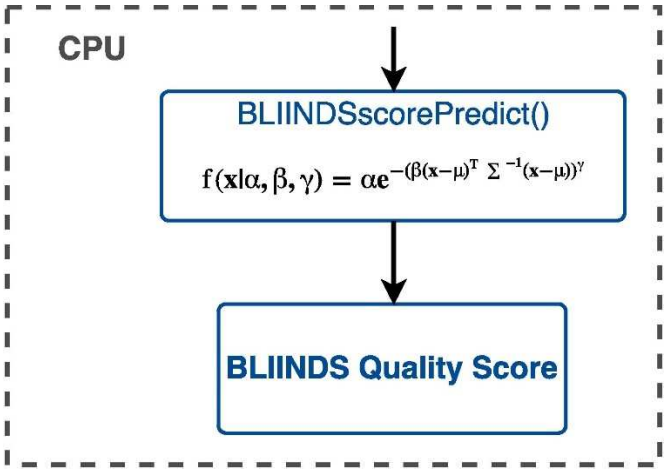

Figure 15. BLIINDS-II Quality Score Prediction Function on the CPU.

BLIINDSscorePredict(): This CPU function calculates the posterior probability of the quality score being a certain value given the set of extracted features. This probability is calculated for each quality score from 0 to 100 in steps of 0.5. To do this, the quality score is appended to the end of the array $\boldsymbol{X}$ and plugged into equation (3.9). The value of the parameters α, β, μ and Σ are pre-determined from training. The score corresponding to the maximum posterior probability is returned as the BLIINDS-II image quality score.

4.2.5 *Miscellaneous implementation details*

We make use of the compiler flags -use_fast_math which uses a fewer number of instructions for multiply, divide and other more complex math operations like sqrt(), sin(), log() etc. In turn we sacrifice some of the precision in the single precision computations. This flag affects only single precision calculations and not the double precision units.

We also make use of the compiler option below:

```
-arch=compute_35,sm_35
```

This produces CUDA binaries specifically tuned the architecture NVidia GPUs with compute capability 3.5, while the default is set for compute capability 2.0 in a new CUDA project.

Chapter 5. EXPERIMENTAL SETUP

We used an NVidia Tesla K40 GPU with Kepler GK110 microarchitecture. The relevant technical specifications about this GPU is mentioned in Table 2. The overall system used for our experiments had a configuration as described in Table 3.

For our tests, we picked distorted images of two different scenes from CSIQ database (Larson and Chandler 2010). We choose images with three kinds of distortions (AWGN, JPEG and BLUR) at two levels (high and low). Our choice is based on the choices made by Phan et al. (2014) which presents an efficient C++ implementation of BLIINDS-II algorithm and provides us with a baseline for performance comparison of our implementation. The colored distorted images in the database were converted to greyscale and then processed by the BLIINDS quality assessment function.

We use CUDA 7.5 for our implementation and use NVidia profiler (nvprof), the command line profiler and NVidia Visual Profiler (nvvp) for performance evaluations and collecting various performance metrics.

Table 2. Kepler GK110 Specifications.

Kepler GK110 Base Core Clock-Rate	889 MHz
Number of Cores	2,880
Computational Throughput	4290 GFLOPS Single Precision 1430 GFLOPS Double Precision
Memory Clock (Transfer) Rate	7000 Gbps
Memory Bus Support	PCIe 3.0 x16
Memory Bus Width	384 bits
Memory Size	12 GB
Memory Interface	GDDR5
Global Device DRAM Bandwidth	336 GB/s
Memory Controllers	64-bit (Quantity 6)
Streaming Multiprocessors (SM)	15 (192 Cores per SM)
L1-Cache (per SM)	64 KB

L2-Cache	1.5 MB
32-bit Registers (per SM)	65,536
Maximum Registers per Thread	255
Threads per Warp	32
Maximum Warps (per SM)	64
Maximum Thread Blocks (per SM)	16
Maximum Threads (per SM)	2,048
Maximum Threads per Thread Block	1,024
Pixel Fill-Rate	42.7 $(GP/s)^2$
Texture Fill-Rate	213 $(GT/s)^2$

Table 3. Test System Specifications.

CPU	Intel Xeon E5 1620 v2 @3.70 GHz
Microarchitecture	Ivy Bridge
No of Cores	4
No of Threads	8
L1 Cache	64 KB per core
L2 Cache	256 KB per core
L3 Cache	10 MB shared
Host RAM	24 GB GDDR3
Operating System	Windows 7 (64-bit)
Compiler	Visual Studio 2013, CUDA 7.5
GPU	NVidia Tesla K40

Chapter 6. PERFORMANCE EVALUATION

The previous C++ implementation of the BLIINDS-II algorithm was reported to take almost 8.0 seconds for 30 iterations 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.

The performance of the GPU version of the algorithm as captured by NVidia Visual Profiler can be seen in Figure 12. The timeline has been zoomed in so as to focus on the execution on one iteration of the input distorted image.

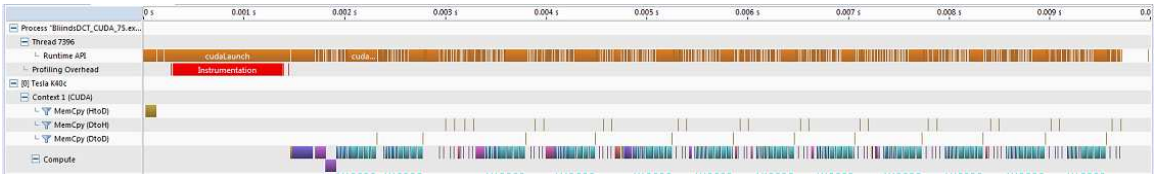

Figure 16. Kernel Execution Timeline as Captured by NVidia Visual Profiler (nvvp).

As seen in the Table 4 from the performance evaluation 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 to some extent, while preserving their output.

Table 4. Performance Comparison of C++ and CUDA Implementations of the BLIINDS-II Algorithm.

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

Table 5. Performance Evaluation of CUDA Implementation of BLIINDS-II.

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

Next we proceed to present other kernel specific optimization strategies and their results.

6.1 *CuFFT vs matrix multiplication*

For the local 2D DCT computation, we compared the performance of two proposed methods as described below:

The timing comparison between the two DCT methods can be seen in Table 4.

Table 6. 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	8.15

This difference in execution time can clearly be explained by the following:

1. More number of global synchronization points (each arrow in Figure 15) causing data flow to and fro from global memory. Transactions to and from the global memory are the most expensive ones on a GPU.

2. Method 2 uses just one kernel for matrix multiplication as opposed to four kernel calls in method 1.

6.2 *Merge Rearrange and DCT kernels*

Table 7. Performance Comparison of RearrangeAndDCT.

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

As can be seen from the timings, the combined kernel doesn't take an execution time equivalent to the sum of the individual Rearrange and DCT kernel runtimes. This was an expected result because it avoids writing the rearranged image values to global memory first and then copying those same values back to CUDA cores for the DCT kernel. We assemble the image in rearranged form in shared memory and perform DCT matrix multiplication.

6.3 *gama_dct*

Table 8. Performance Comparison of gama_dct Computation Methods.

CUDA gama_dct	Method 1 (Global Memory)	Method 2 (Shared Memory)	Method 3 (Split in three kernels)
Run time (ms)	9215.73	781.84	14.67

Discussion on method 3 and use of constant memory:

We use constant memory to store our look up table for rho. Constant memory is only 48KB in size and is useful when –

1. stored data will not change during the execution.
2. threads in a warp access the same element in constant memory. The constant memory can serve 4B of data to one warp in one cycle. Accesses to different elements from one warp will serialize the operation.

Our implementation is conscious of both these conditions. The look up table for rho is constant and never changes. The array of rho values was sorted before accessing the look up table to ensure minimize the number of reads from different location in constant memory.

We also noted that the look up table for rho is arranged in a decreasing order. This enabled us to use binary search for look up instead of linear search, thereby reducing our look up time further.

As a consequence of sorting the array of rho values, the array of gamma value for each block is obtained as a sorted array. This fact obviates the need for sorting to compute the mean of lowest 10 percentiles. Hence, while reporting the execution time for method 3 for gama_dct, we have not included the time spent in sorting the intermediate array because we would anyway have to sort the final array of gamma values to compute the mean of lowest 10 percentiles.

6.4 *Oriented_dct*

Table 9. Performance Comparison of oriented_dct kernels.

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

Merging the individual orient kernels into one results in a decrease in the runtime, just as expected, because the data transfer from the global memory of the GPU is minimized. This result is very similar to what we saw in section 6.2.

Chapter 7. DISCUSSIONS AND CONCLUSION

We have demonstrated a promising GPGPU implementation of BLIINDS-II NR-IQA algorithm using NVidia Tesla K40 that is capable of processing a real time feed of video. The average processing time for one 512x512 image is estimated to be around 9 ms, which means the implementation can handle a video feed of more than 100 fps.

That said, the performance analysis showed that even though the implementation is capable of running at video speeds, its spends most of its execution time in sorting the statistical model parameters. The second biggest hotspot function was reduce operation (sum) used to compute mean of lowest 10 percentiles and 100 percentiles. Both of these are functions from the NVidia Thrust library. It is noteworthy that the bottlenecks functions have changed in the CUDA code, but still each individual function's performance on the GPU is faster than the corresponding performance on the CPU version of the algorithm. The slowest GPU kernel of sort is still 2.3x faster than sort on CPU.

We implemented GPU architecture conscious code optimizations and illustrated how they compare with naive versions. In general, for the GPU, data reuse in kernels is highly likely to result in performance gains. The core idea behind this approach is to minimize data load from and store to the off chip global memory by continuing to perform the next step in BLIINDS-II pipeline on the same cores that computed the previous step. These cores have the access to the required data either in shared memory or in registers and can start executing the next step computations on the data right away. This result can be seen as an extension of the idea of minimizing data transfers across PCIe bus between the

CPU and the GPU, only that now we are arguing to minimize the data transfer across off chip memory on the GPU.

The advantage of the approach described is clearly seen in the performance evaluation of merged Rearrange and DCT kernels and the merged kernel for orientation parameter. We further argue that merging the entire flow of the algorithm into as few kernels as possible will result in further performance acceleration. Merging the flow can of BLIINDS-II can be realized by extending the RearrangeAndDCT kernel to perform the functions of the statistical modeling. Though in the case of gamma_dct kernel, we observed that breaking down the kernel into smaller units helped us attain performance gains. We can conclude that look up tables should mostly be avoided on GPUs, and with the high number of compute resources, we should be more inclined towards compute.

In order to efficiently exploit the parallelism in the BLIINDS-II algorithm, we rearranged the input distorted image which led to an increase in the image size from 1 MB to approximately 115 MB (for a 512×512 image). This rearrangement of the image array laid out the each of the 25 elements in a 5×5 pixel block in contiguous memory. This in turn allowed us to launch further kernels for statistical modeling of each 5×5 pixel block in parallel. Any other approach to implement this will result in serialization of the statistical modeling step. Although, if the merging technique, as describe in the previous paragraph, is implemented, the need to store the rearranged image array may be eliminated. Such an implementation would directly proceed to compute the statistical model for each 5×5 pixel block on the same warp that first computed the DCT coefficients of the pixel block. Therefore, another benefit of merging kernels is found in

the fact that it will lead to a reduction in the memory signature required by the CUDA BLIINDS-II implementation.

Gamma_dct() function posed a major challenge in the GPGPU implementation because it uses a look up table to arrive at the gamma model parameter. In general using look up tables on the GPU is not preferred. We have managed to present gamma_dct on the GPU which is comparable in run time with the other statistical modeling kernels. This high performance for gamma_dct was achieved by breaking it into three separate kernels as described in the implementation section. The look up table itself was stored in constant memory, which is located off chip. The second kernel in this sequence for gamma_dct was to sort the array obtained after the kernel. This was done to attain maximum data access speed in constant memory and has been described in the chapter on implementation details. It should also be pointed out that the constant memory size on NVidia Tesla K40 is 64 KB and our look up table size is just short of 40 KB because it is stored as an array of floats instead of as an array of doubles. If it was stored as doubles, constant memory would have been insufficient to store this table. We have ensured that the accuracy of our GPU implementation is intact even with the use of single precision. We have also employed binary search for traversal of the look up table as against the linear search employed by the C++ and Matlab version of BLIINDS-II. This is an example of reducing the computational complexity in a part of the code and is a significant optimization as this traversal through the look up table was initially the most time consuming part of the CUDA implementation.

Another important conclusion we have that in GPGPU applications involving matrix multiplication of floats, it is likely that one may arrive at inaccurate results if double precision is not used for these operations. One must test the accuracy rigorously in case a matrix multiplication is implemented with single precision. We observed that both the convolve and the DCT matrix multiplication required use of double precision cores.

Chapter 8. FUTURE WORK

For further optimizations in the code, the most obvious first step is to look at the alternative sort functions one could employ. ArrayFire and CUB libraries have been reported in literature for having a better performance for sort as compared to Thrust. Sort kernels from these libraries should be implemented and compared with the existing CUDA version of BLIINDS-II.

Another area to be looked into is adding CPU multithreading to the existing BLIINDS-II CUDA application. Currently we have the code set up so that the CPU handles the task of casting the image from unsigned character to float, and then predicting the BLIINDS quality score after it receives the features array from the GPU. The GPU remains idle while the CPU is busy in these tasks. This situation can be set up so that these CPU tasks are handled by one CPU thread while another CPU thread controls the kernel launches on the GPU. This way, some of the latency between the CPU and the GPU can be hidden.

The possibility of merging smaller kernels should be explored further. However, not much significant gain is expected from this unless the high amount of time spent on sort is dealt with.

BLIINDS-II VQA algorithm should be considered next for a CUDA implementation, and that might provide a high performing real time No Reference video quality assessment. Development of such a system will be a huge boost to quality in video streaming applications.

References

Ahumeda, A. “Simplified vision models for image-quality assessment,” in *SID International Symposium Digest of Technical Papers*, vol. 27, pp. 397–402, Society for Information Display, 1996.

Blake, R. and Sekuler, R. *Perception*, 5th ed. New York: McGraw Hill, 2006.

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

Chandler D. M. and Hemami, S. S. “Additivity models for suprathreshold distortion in quantized wavelet-coded images,” in *Human Vision and Electronic Imaging VII*, B. E. Rogowitz and T. N. Pappas, Eds., Proceedings of SPIE, pp. 105–118, San Jose, Calif, USA, January 2002.

Chandler, D. M., Lim, K. H. and Hemami, S. S. “Effects of spatial correlations and global precedence on the visual fidelity of distorted images,” in *Human Vision and Electronic Imaging XI*, B. E. Rogowitz, T. N. Pappas, and S. Daly, Eds., Proceedings of SPIE, San Jose, Calif, USA, January 2006.

Chen, M. J. and Bovik, A. C. “Fast structural similarity index algorithm,” *Journal of Real-Time Image Processing,* vol. 6, no. 4, pp. 281–287, 2011.

Gordon, B., Sohoni, S. and Chandler, D. “*Data handling inefficiencies between CUDA, 3D rendering, and system memory*,” in Proceedings of the IEEE International Symposium on Workload Characterization (0IISWC '10), December 2010.

Harris, Mark. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. 7 January 2013. NVidia. <https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/>.

Holloway, Joshua K. *An Analysis of the Computational Performance of a Massively Parallel Perceptual Image Quality Assessment Algorithm Implemented on Multiple Graphics Processing Units.* Thesis. Oklahoma State University, 2015.

Larson, Eric C., Chandler, Damon M. "Most apparent distortion: full-reference image quality assessment and the role of strategy." *Journal of Electronic Imaging* 19.1 (2010): 011006-011006.

Larson, Eric C., and D. M. Chandler. "Categorical image quality (CSIQ) database." *Online, http://vision. okstate. edu/csiq* (2010).

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

NVIDIA cuFFT Library Users Guide, NVIDIA (2015).

NVIDIA Kepler GK110 Whitepaper, NVIDIA (2012).

Okarma, K. "Video quality assessment using the combined full-reference approach." *Image Processing and Communications Challenges 2*. Springer Berlin Heidelberg, 2010. 51-58.

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.

Ramos, M. G. and Hemami, S. S. "Suprathreshold wavelet coefficient quantization in complex stimuli: Psychophysical evaluation and analysis," *Journal of the Optical Society of America A*, vol. 18, no. 10, pp. 2385–2397, 2001.

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

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.

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

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.

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

VQEG, *Final Report From the Video Quality Experts Group on the Validation of Objective Models of Video Quality Assessment, Phase II*, 2003, http://www.vqeg.org.

Wang Z. and Simoncelli, E. P. "Stimulus synthesis for efficient evaluation and refinement of perceptual image quality metrics," in *Human Vision and Electronic Imaging IX*, vol. 5292 of *Proceedings of SPIE*, pp. 99–108, January 2004.

Wang, Zhang, Simoncelli, Eero P. and Bovik. Alan C. "Multiscale structural similarity for image quality assessment." *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*. Vol. 2. IEEE, 2003.

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

Yang, Zhiyi, Yating Zhu, and Yong Pu. "Parallel image processing based on CUDA." *Computer Science and Software Engineering, 2008 International Conference on*. Vol. 3. IEEE, 2008.