Analysis and Performance Optimization of a GPGPU Implementation of Image Quality Assessment (IQA) Algorithm VSNR

by

Ayush Gupta

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

Approved April 2017 by the
Graduate Supervisory Committee:

Sohum Sohoni, Chair
Ashish Amresh
Ajay Bansal

ARIZONA STATE UNIVERSITY

May 2017

ABSTRACT

Image processing has changed the way we store, view and share images. One important component of sharing images over the networks is image compression. Lossy image compression techniques compromise the quality of images to reduce their size. To ensure that the distortion of images due to image compression is not highly detectable by humans, the perceived quality of an image needs to be maintained over a certain threshold. Determining this threshold is best done using human subjects, but that is impractical in real-world scenarios. As a solution to this issue, image quality assessment (IQA) algorithms are used to automatically compute a fidelity score of an image. However, poor performance of IQA algorithms has been observed due to complex statistical computations involved. General Purpose Graphics Processing Unit (GPGPU) programming is one of the solutions proposed to optimize the performance of these algorithms.

This thesis presents a Compute Unified Device Architecture (CUDA) based optimized implementation of full reference IQA algorithm, Visual Signal to Noise Ratio (VSNR) that uses M-level 2D Discrete Wavelet Transform (DWT) with 9/7 biorthogonal filters among other statistical computations. The presented implementation is tested upon four different image quality databases containing images with multiple distortions and sizes ranging from 512 x 512 to 1600 x 1280. The CUDA implementation of VSNR shows a speedup of over 32x for 1600 x 1280 images. It is observed that the speedup scales with the increase in size of images. The results showed that the implementation is fast enough to use VSNR on high definition videos with a frame rate of 60 fps. This work presents the optimizations made due to the use of GPU's constant memory and reuse of allocated memory on the GPU. Also, it shows the performance improvement using profiler driven GPGPU development in CUDA. The presented implementation can be deployed in production combined with existing applications.

TABLE OF CONTENTS

Page

LIST OF TABLES V

LIST OF FIGURES VI

CHAPTER

1. INTRODUCTION 1

   Image Quality Assessment (IQA) Algorithms 2

   GPU Computing 3

   CUDA Programming 5

   Overview 6

2. RELATED WORK 8

3. VSNR ALGORITHM 11

   Algorithm Steps 16

   Performance of VSNR 20

4. CUDA IMPLEMENTATION OF VSNR 22

   Image Loading 24

   Luminance Vector 25

   Calculation of Mean 26

   Calculation of RMS Contrast 27

   2D Discrete Wavelet Transform 28

   Computation on Sub-Bands 30

CHAPTER Page

5. METHODOLOGY 33

6. RESULTS 36

Analysis of Image Loading 38

Analysis of Computation on Sub-Bands 39

Analysis of Computation of Luminance 40

7. DISCUSSION OF RESULTS 42

8. CONCLUSION 45

9. FUTURE WORK 46

REFERENCES 47

LIST OF TABLES

Table Page

1. Experiment Environment Details 33
2. GPU Specifications 33
3. Image Quality Database Details 35
4. Execution Time and Speed Up of VSNR on CPU and GPU 36
5. Execution Time Breakdown of Primary Functions 38
6. Time Consumed in Memory Transfer and Allocation 38
7. Execution Time Comparison of Two Approaches to Load Image to GPU 39
8. Execution Time Comparison of Two Approaches to Compute Sub-Bands 40
9. Execution Time Comparison of Two Approaches to Calculate Luminance 40

LIST OF FIGURES

Figure Page

1. Grid of Thread Blocks (CUDA C Programming Guide, 2015, p. 23) 5
2. Memory Hierarchy (CUDA C Programming Guide, 2015, p. 25) 6
3. Overview of VSNR (Phan, et al., 2014) 11
4. Detailed Flowchart of VSNR.. 16
5. 3-Level 2D DWT Sub-Bands.. 18
6. Execution Time Distribution on CPU (Phan, et al., 2014) 21
7. CPU-GPU Division of VSNR.. 23
8. Approach 1- Loading Image to GPU 24
9. Approach 2- Loading Image to GPU 24
10. Evaluation of Luminance Vector on GPU 25
11. Evaluation of Luminance Vector Using Lookup Table 26
12. Example of Reduction on GPU... 27
13. Evaluation of RMS on GPU 28
14. Example of 2D DWT on GPU 30
15. Evaluation of Standard Deviation of Sub-Bands on GPU 31
16. Computation on Sub-Bands on GPU 32
17. Execution Time of VSNR on CPU and GPU 37
18. Speedup Curve 37
19. Breakdown of Execution Time 41

Chapter 1. Introduction

Image processing technologies have changed the way we view, store and share images. Transmission of large images over the network has increased dramatically since the recent advancement of connectivity among devices (Chandler, 2013). The real-time image and video transmitting mechanisms have been possible because of the betterment in image processing techniques. One important factor for seamless transfer of multiple images or video over the network is image compression. In the case of lossy compression, the quality of images is compromised. For example, in JPEG compression, blurring and blocking are some of the significant artifacts generated that lead to degradation (Wang, Bovik, Sheikh, & Simoncelli, 2004).

To ensure that image degradation due to losses in image compression is not detectable by humans, we need a method to analyze the compressed image. Human assessment of an image could be a method to analyze the quality of images but it is inefficient and possibly varied across different people. Firstly, it requires large-scale studies with hundreds of human subjects per image. Secondly, the quality score can be varied across individuals (Yadav, 2016). Hence, it is not feasible to use humans for analyzing the huge number of images that are expected to be transferred in real time. Image quality assessment (IQA) algorithms are developed to automate the process of analyzing images.

The goal of IQA algorithms is to develop a quantitative measure that can automatically predict the quality of the image (Wang, et al., 2004). IQA has gradually become a major subfield of image processing and is attracting a lot of researchers because of its utility.

Image Quality Assessment (IQA) Algorithms

IQA and Video Quality Assessment (VQA) algorithms employ different approaches but they share two common algorithmic operations, a local frequency-based decomposition (filtering/filter banks or transforming) and block-based statistical comparisons between the frequency coefficients of the reference and distorted images/videos (statistical computation) (Phan, Sohoni, Chandler, & Larson, 2012). There are three types of IQA algorithms currently in research i.e. full reference, reduced reference and no reference algorithms.

Full reference algorithm takes an original image and a distorted image as input and returns a fidelity metric for the level of distortion. These algorithms are mostly used in image compression and television where the ideal image is available (Li, Wu, Chen, & Li, 2009). One example of full reference IQA algorithm is Visual Signal-to-Noise Ratio (VSNR), which first detects the distortions in the presence of a reference image. Based on the level of distortion, it operates based on the low-level visual property of contrast and mid-level visual property of global precedence to find the quality score (Chandler & Hemami, 2007).

Reduced reference algorithm assesses the quality using a distorted image and partial information about the original image. The partial information is the set of extracted features from the ideal image (Li, et al., 2009). Wang & Simoncelli (2005) developed a reduced reference IQA algorithm that uses the Kullback-Leiber distance between the marginal probability distributions of wavelength coefficients of the reference and distorted images as a measure of image distortion.

No reference algorithms assess the quality of images using a distorted image and without the availability of original image. These algorithms are used in photography, where the ideal image is not available (Li, et al., 2009). An example of no reference IQA algorithm is Blind, which uses natural scene statistics (NSS) model of discrete cosine transform (DCT) coefficient to predict the quality score (Saad, Bovik, & Charrier, 2012).

One of the most prominent problems in IQA algorithms is their performance which refrains us to use these algorithms for real-time applications. IQA algorithms require intensive computation power to attain high accuracies. As we start using IQA algorithms for high definition images (4K resolutions) and video processing, the runtime performance of the algorithms run short. As suggested by Chandler (2013), to use these algorithms in mainstream applications, we need to optimize the performance of these algorithms while maintaining the accuracy. We can optimize performance by either making algorithm level modification or accelerate the processing by using alternative processing methods.

One solution is to use modern CPUs but they are insufficient to process IQA algorithms in real time (Park, Singhal, Lee, Cho, & Kim, 2011). Other solution for accelerating performance could be the General Purpose Graphics Processing Unit (GPGPU). Graphics Processing Units (GPUs) have evolved into an extremely powerful resource, which is an attractive alternative to process images for assessing quality in real time.

GPU Computing

GPU computing has been the most emerging alternative for modern computation. Now, GPUs are used not just for graphical processing but also for arithmetic computation. The rapid increase in the performance of GPUs and improvement in its programmability has made them an attractive option for high computation tasks (Owens, et al., 2007). For example, the number of flops of NVIDIA Ge80 series has got 520G flops in late 2006 whereas Intel 64-bit dual core CPU has only 32 G flops (Yang, Zhu, & Pu, 2008). This shows the difference between the computation powers of CPU and GPU.

NVIDIA introduced a new programming model named Compute Unified Device Architecture (CUDA) that helps programmers write massively parallel code in C/C++. CUDA makes it simple to write code that can run on thousands of threads in parallel. CUDA uses the approach of General Purpose GPU (GPGPU) programming where the program is partially executed on CPU and partially on GPU. CPU is more generally used to transfer memory to GPU and initiate execution on GPU. High programmer acceptability makes CUDA an excellent programming model for massively parallel programming.

In image processing, substantial pixel data is processed and with the increase in the number of pixels, as in the case of high definition and post high definition images, the size of images increases significantly. Thus, CPU computation power does not suffice in some cases and GPGPU programming can provide highly data-parallel processing (Yang, Zhu, & Pu, 2008). GPGPU programming, being the Single Instruction Multiple Data (SIMD), can work well with huge matrices, like images (Yang, Zhu, & Pu, 2008). The implementation of histogram equalization using CUDA was reported with more than 40x speed and speed up increased with the size of the image significantly (Yang, et al., 2008).

CUDA Programming

CUDA is a programming model that consists of two sections, one is the device code that is executed on GPU and the other is the sequential C code that is executed on CPU. Device code is called using the kernel functions. Each instruction is executed by a thread, which is a smallest individually executable unit. Thread in CUDA has three dimensions, making it ideal to work with matrices. Threads combine to form a block and blocks combine to form a grid. Depending on a GPU, a single block can contain up to 2048 threads. Each thread execution can be forced to wait for other threads to reach the instruction to synchronize. Figure 1 demonstrates the structure of grid, blocks, and threads.

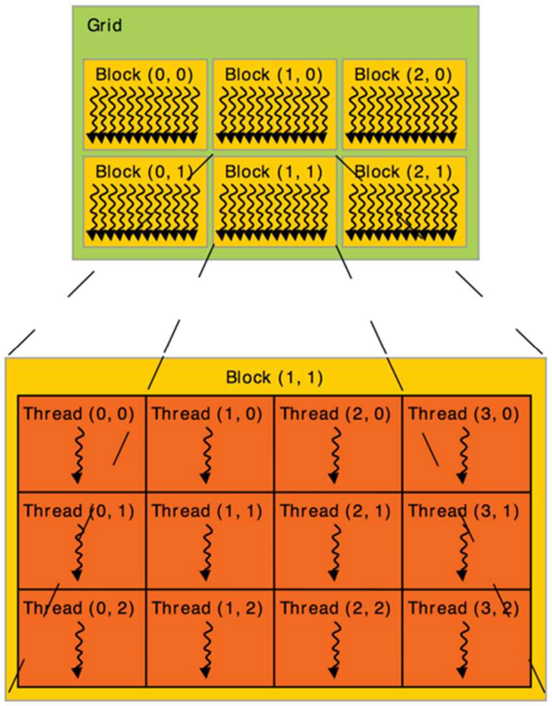

Figure 1 *Grid of Thread Blocks (CUDA C Programming Guide, 2015, p. 23)*

CUDA provides a functionality to allocate and deallocate memory from the host i.e. CPU. Based on the GPU, CUDA can provide support for different types of memories.

First, global and constant memories, which are shared among all threads executed on the GPU. Next, shared memory, which is faster than the global memory and is shared among the threads in a block. Last, local memory, which is the fastest and smallest memory of all, and is local to a single thread. Figure 2 shows the overview of memory distribution on a GPU.

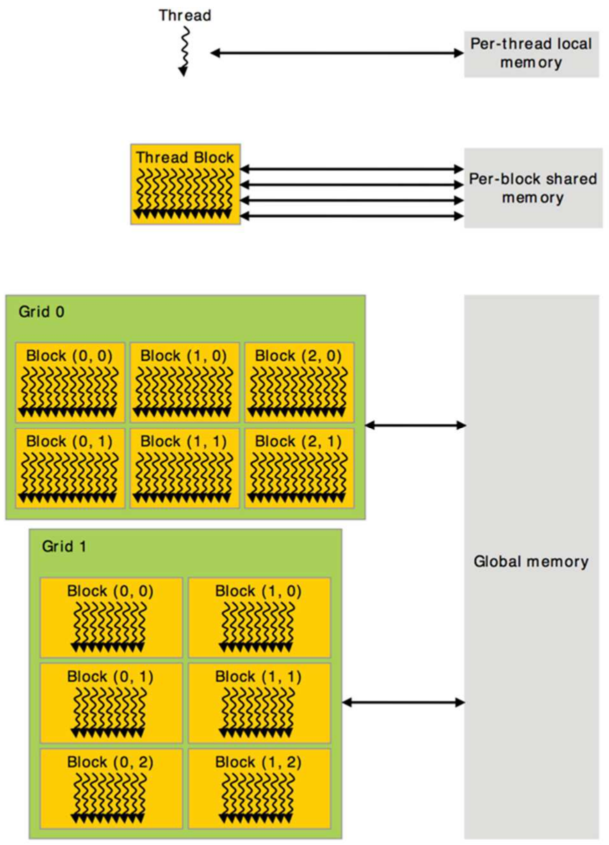

Figure 2 *Memory Hierarchy (CUDA C Programming Guide, 2015, p. 25)*

Overview

In this thesis, an effort to optimize a wavelength based full reference IQA algorithm, VSNR using CUDA is presented. Chapter 2 will discuss the previous work on microarchitectural analysis and optimization of IQA algorithms. Chapter 3 will introduce VSNR algorithm and its performance review in detail. Chapter 4 will discuss the implementation of VSNR using CUDA. Chapter 5 will specify the details of the experiment. Chapter 6 will analyze the results of the experiments. Chapter 6 will include the conclusion of the experiments. Finally, Chapter 7 will discuss the future works related to VSNR optimizations.

Chapter 2. Related Work

In this chapter, we will discuss the other publications related to IQA algorithm and other efforts to optimize IQA algorithms. As mentioned in the previous section, IQA Algorithms require a lot of statistical computation to assess a quality of image accurately and therefore raises the issue of slow execution. As suggested by Chandler (2013), to use these algorithms in mainstream applications, we need to optimize execution time of these algorithms while maintaining the accuracy. We can optimize performance by either making algorithm level modification or accelerate the processing by either using GPUs or multi-core CPUs.

One of the efforts to improve performance was presented by Chen & Bovik (2011) called Fast Structural Similarity index (SSIM) and Fast Multi-Scale-SSIM (MS-SSIM), which were the algorithm level modifications to optimize SSIM and MS-SSIM respectively. Firstly, Fast SSIM modified the calculation of luminance term using 8 x 8 square window and an integral image technique. Secondly, calculation of variance was replaced by gradient value. Lastly, integer approximation was used in place of the Gaussian weighting window. These modifications resulted in the speedup of 2.68x and approximately 10x for Fast SSIM and Fast MS-SSIM respectively.

Another effort was made by Okarma & Mazurek (2011) to optimize SSIM and MS-SSIM using CUDA based implementation, which resulted in the speedup of 150x and 55x for the SSIM and MS-SSIM respectively.

Gordon, Sohoni, & Chandler (2010) reported the degradation in performance of PSNR when implemented using CUDA. The degradation was reported due to the overhead of repeated memory transfers between CPU and GPU which overruled statistical computation.

Phan, et al. (2012) performed analysis of the IQA algorithm Most Apparent Distortion (MAD) (Larson & Chandler, 2010), to accelerate expensive stages of the algorithm, i.e. local frequency-based decomposition and statistical comparisons between the frequency coefficients of the reference and distorted images. The author proposed four methods of acceleration, generalized integral images, inline expansion, GPGPU and code optimizations.

Phan, et al. (2014) further conducted a microarchitectural analysis of IQA algorithms. It included four full reference IQA algorithms, MAD (Larson & Chandler, 2010), MS-SSIM (Wang, Simoncelli, & Bovik, 2003), VIF (Sheikh & Bovik, 2006) and VSNR (Chandler & Hemami, 2007), and two no reference IQA algorithms, Blinds (Saad, Bovik, & Charrier, 2012) and Brisque (Mittal, Moorthy, & Bovik, 2012). To perform microarchitectural analysis, the code of these algorithms was first ported to C++ for uniformity, and Intel's Vtune Amplifier XE (Intel, 2017) was used for microarchitectural analysis and to identify the bottleneck segments. Phan, et al. (2014) reported bottlenecks in two central categories, memory bottlenecks, and core/computational bottlenecks. They proposed to analyze characteristics of execution on different architectures, such as image processing cores and GPUs.

Holloway (2015) made an effort to optimize MAD (Larson & Chandler, 2010) using GPGPU, as proposed by Phan, et al. (2012). He reported speed up of 24.76x speedup over the CPU implementation for single GPU implementation and 43x for multi-GPU (3 GPUs used in the experiment) implementation. The reported multi-GPU implementation was approximately 1.74x faster than single GPU implementation, which is less than expected. The reduced speed up was revealed to be due to the overhead of transferring data across PCIe buses.

Kannan, Holloway, Sohoni, & Chandler (2017) conducted a microarchitectural analysis of CUDA implementation of MAD (Holloway, 2015) using the Nvidia Visual Profiler (Nvidia Visual Profiler, 2017). He reported the memory bandwidth, i.e. rate of reading and writing data to memory, as the major hotspot of the implementation.

Yadav, Sohoni, & Chandler (2017) presented a GPU implementation of Blinds (Saad, et al., 2012) to improve performance. He reported a speed up of 30x over the CPU implementation for thirty iterations of a 512 x 512 image. In the implementation, the bottleneck was sorting operations on image vector, which took 44% of the total execution time.

Previous research (Phan, et al., 2014; Chandler, 2013) has suggested the use of GPUs for performance optimization of IQA algorithms. While the degradation of performance was observed in the GPU based CUDA implementation of Peak Signal to Noise Ratio (PSNR) (Gordon, et al., 2010), tremendous speedups were observed from GPU based CUDA implementation of SSIM (Okarma & Mazurek, 2011), MS-SSIM (Okarma & Mazurek, 2011), MAD (Holloway, 2015) and Blinds (Yadav, 2016). The above examples show that CUDA has a potential to optimize an algorithm significantly considering the memory transfers do not overrule the computation. Hence, an effort to optimize VSNR using CUDA has the potential to show positive results.

Chapter 3. VSNR Algorithm

VSNR is an IQA algorithm developed by Chandler & Hemami (2007). It provides a metric for quantifying visual fidelity of natural images based on near threshold and suprathreshold properties of human vision.

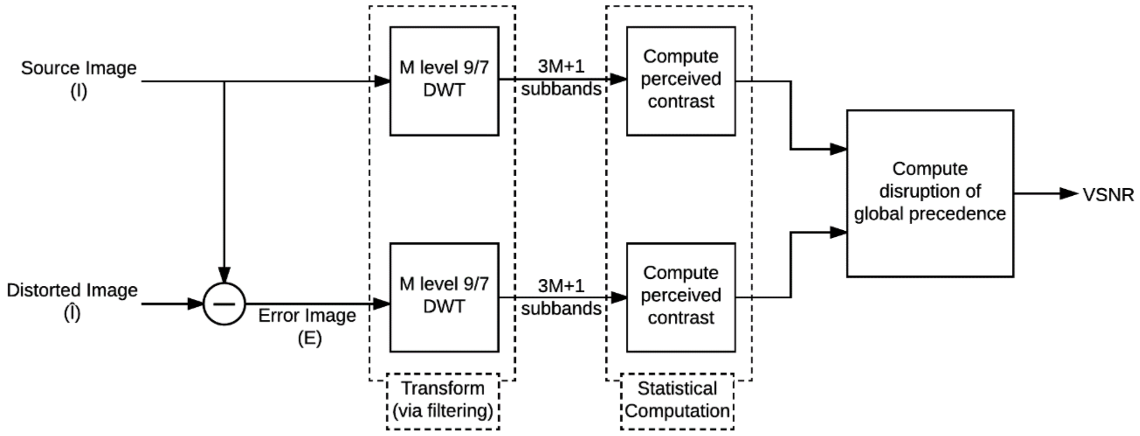

Figure 3 *Overview of VSNR (Phan, et al., 2014)*

The algorithm first operates on low-level Human Visual System (HVS) properties of contrast sensitivity and visual masking. If the distortions are below the threshold, the low-level HVS properties obtained from the previous step and mid-level properties of global precedence are used to measure the structural degradation (Chandler & Hemami, 2007). VSNR uses following to measure the visual fidelity (Chandler & Hemami, 2007);

1. contrast threshold for detection of distortions.
2. measure of the perceived contrast of the distortion
3. a measure of the degree to which the distortions disrupt global precedence and degrade the image structure.

VSNR uses two images, original image (I) and distorted image (Î). From the I and Î, an error image is computed, given by $E = \hat{I} - I$, which denotes the distortions in Î. Both I and E are preprocessed using M-level 2D separable Discrete Wavelet Transform (DWT) with 9/7 biorthogonal filters. M-level DWT results in 3M+1 sub-bands (Chandler & Hemami, 2007).

Next both images are modeled, according to the viewing conditions, using the pixel-value-to-luminance response characteristic,

$$L(P) = (b + kP)^{\gamma} \tag{1}$$

where *b* represents the black-level offset, *k* the pixel-value-to-voltage scaling factor, and $\gamma$the gamma of the display monitor (Chandler & Hemami, 2007). In addition to this, a vector of octave-spaced frequencies *f*, in cycles/degree, is computed based on the viewing distance and the resolution of display using

$$f_m = 2^{-m} r v \tan\left(\frac{\pi}{180}\right) \tag{2}$$

where *m* denotes level from DWT, *r* denotes the resolution of the display in pixel per unit distance, and *v* is the viewing distance expressed in the corresponding units of distance (Chandler & Hemami, 2007).

As specified earlier, VSNR works in two stages: first, it computes the contrast threshold and second, if the distortions have exceeded the threshold, i.e. suprathreshold, visual fidelity is estimated using perceived contrast and the extent to which the global precedence is disrupted by distortion (Chandler & Hemami, 2007).

In stage 1, initially, to compute threshold of contrast detection, the threshold contrast SNRs centered at spatial frequency are computed using following model for each level of DWT

$$CSNR_{f_m}^{thr} = a_0 f_m^{a_2 \ln(f_m) + a_1} \tag{3}$$

where f_m denotes the spatial frequency for level m and $a_0 = 58.9$, $a_2 = -0.1258$, $a_1 = -0.1087$ are the constants based on the average threshold (Chandler & Hemami, 2007). Now, values obtained from Equation 3 is used to compute contrast detection threshold using

$$CT(E|I) = \left\| \frac{C(I_{f_m})}{CSNR_{f_m}^{thr}} \right\| \tag{4}$$

where $C(I_{f_m})$ is

$$C(I_{f_m}) \approx \frac{k\gamma}{2^m \mu_{L(I)} \left(b + k_{\mu_I}\right)^{(1-\gamma)}} \sqrt{\sigma^2\left[s_{I(m,LH)}\right] + \sigma^2\left[s_{I(m,HL)}\right] + \sigma^2\left[s_{I(m,HH)}\right]} \tag{5}$$

where σ is the standard deviation, *LH*, *HL,* and *HH* are the sub-bands from DWT, $\mu_{L(I)}$ is the mean of pixel-value-to-luminance of the original image, and μ_I is the mean of the image (Chandler & Hemami, 2007). Secondly, $C(E)$ is computed, which is the RMS contrast defined by

$$C(E) = \frac{1}{\mu_{L(I)}} \left(\frac{1}{N} \sum_{i=1}^{N} \left[L(E_i + \mu_I) - \mu_{L(E+\mu_I)} \right]^2 \right)^{\frac{1}{2}} \tag{6}$$

Finally, if $C(E) < CT(E/I\,)$, the distortions are below threshold and Î is visually indistinguishable from I. Otherwise stage 2 is executed to compute the fidelity score (Chandler & Hemami, 2007).

In stage 2, at first, measure of the perceived contrast of distortions, denoted by d_{pc} (Chandler & Hemami, 2007) is computed by

$$d_{pc} = C(E) \tag{7}$$

Secondly, measure of the extent to which global precedence has been disrupted, denoted by d_{gp} (Chandler & Hemami, 2007) is computed by

$$d_{gp} = \left\| \left(\sum_{m=1}^{M} \left[C^{*}\left(E_{f_m}\right) - C\left(E_{f_m}\right) \right]^{2} \right)^{\frac{1}{2}} \right\| \tag{8}$$

where $C(E_{f_m})$ is defined by equation 5 for the error image and $C^{*}(E_{f_m})$ indicates comparison of contrast of the distortions within each band centered at $f_m$and the corresponding global precedence preserving contrast and is computed by

$$C^{*}(E_{f_m}) = \frac{C(I_{f_m})}{CSNR^{*}_{f_m}(E)} \tag{9}$$

where $CSNR^{*}_{f_m}(E)$ is, global precedence preserving contrast (Chandler & Hemami, 2007), and is computed by

$$CSNR^{*}_{f_m}(E) = \ b_0(E) f^{b_2(E)\ln(f) + b_1(E)} \tag{10}$$

where $b_0(E)$, $b_1(E)$, and $b_2(E)$ are computed by

$$b_0(E) = -a_0 v(E) + a_0 \tag{11}$$

$$b_1(E) = (1.0 - a_1) v(E) + a_1 \tag{12}$$

$$b_2(E) = (-1.0 - a_0) v(E) + a_2 \tag{13}$$

where $v(E) \in [0, 1]$ represents an index of visibility chosen such that total RMS contrast of the distortion is $C(E)$ (Chandler & Hemami, 2007). Next visual distortion (VD) is computed using

$$VD = \alpha d_{pc} + (1-\alpha)\frac{d_{gp}}{\sqrt{2}} \tag{14}$$

where $\alpha = 0.04$ (Chandler & Hemami, 2007). Lastly, visual fidelity metric VSNR is computed using following:

$$VSNR = 10\log_{10}\left(\frac{C(I)}{ad_{pc} + (1-\alpha)\frac{d_{gp}}{\sqrt{2}}}\right) \tag{15}$$

where $C(I)$ denotes the RMS contrast of the original image (Chandler & Hemami, 2007). Next section specifies the algorithm steps that will help in understanding the complexity and discrete steps of VSNR. Figure 4 demonstrates the algorithm steps of VSNR in a flow chart.

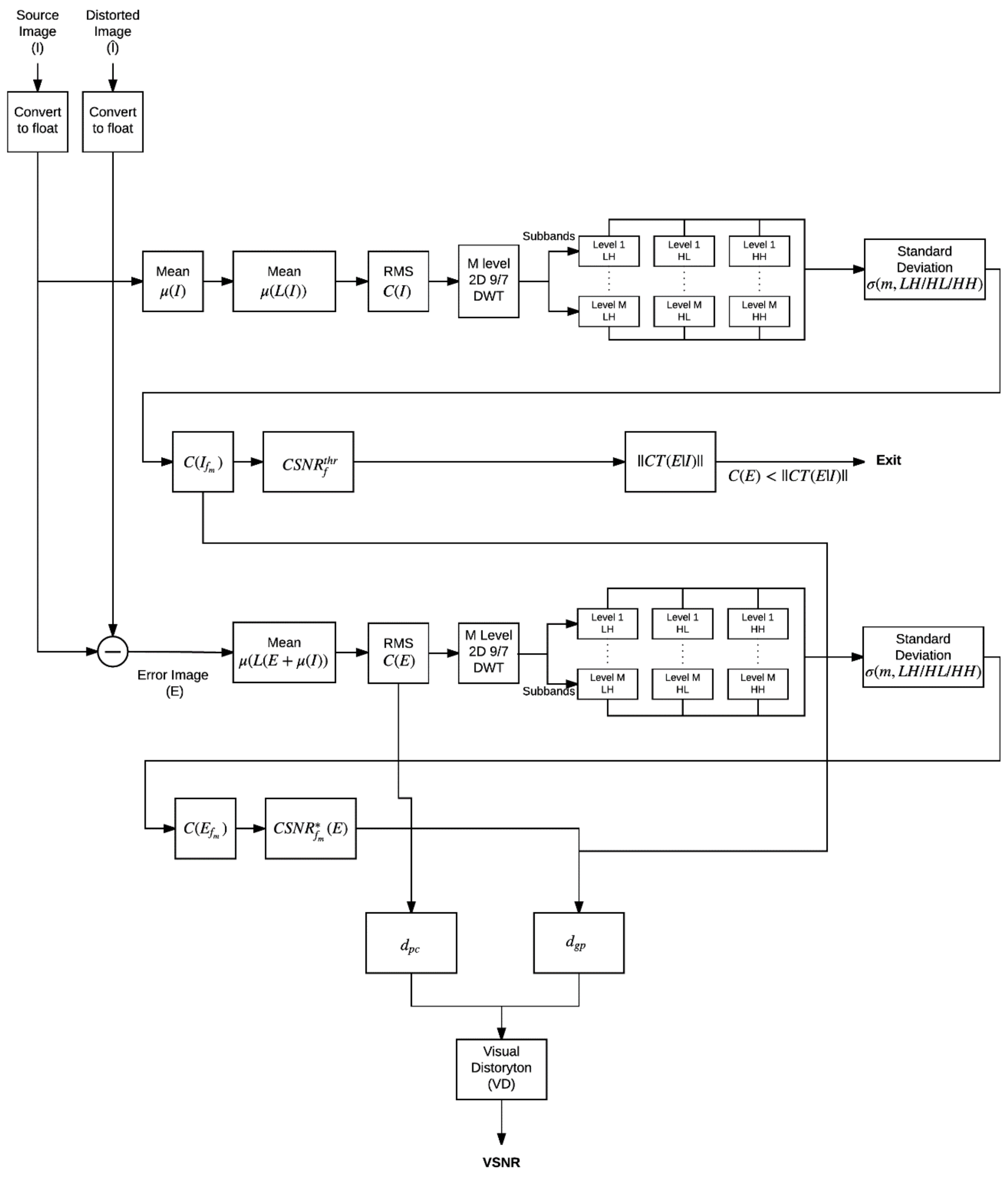

Figure 4 *Detailed Flowchart of VSNR*

Algorithm Steps

Read images. The initial step for executing the algorithm is to read source (I) and distorted (Î) images of size $m \times n$ into the memory, where m represents the number of rows and n represents the number of columns. Once images are read into memory, each pixel is converted from unsigned char to float that allows arithmetic computation on the pixel values. The process of reading and converting them takes $2 \times m \times n$ time.

Next function in the algorithm is to analyze source image from memory.

1. Compute the mean of the source image.
2. Convert the source image into a vector of pixel-to-luminance value. The conversion creates a new vector of size $m \times n$.
3. Compute the mean of the pixel-to-luminance value vector.
4. Calculate the RMS, denoted by $C(I)$, of the source using vectors obtained from steps 2 and 3.
5. Process source image with M-level 2D separable discrete wavelet transform (DWT) with 9/7 biorthogonal filters. DWT will result in 3M+1 vectors as specified in the image below. Each level of DWT divides both dimensions of image by a factor of 2 and results in 4 sub-bands where LL is used to decompose the image further. Figure 5 shows the sub-bands obtained from a three Level 2D DWT (though by default there are five levels in VSNR for DWT but for demonstration purposes three levels have been used). This process would take another memory of $m \times n$ to store the sub-bands.

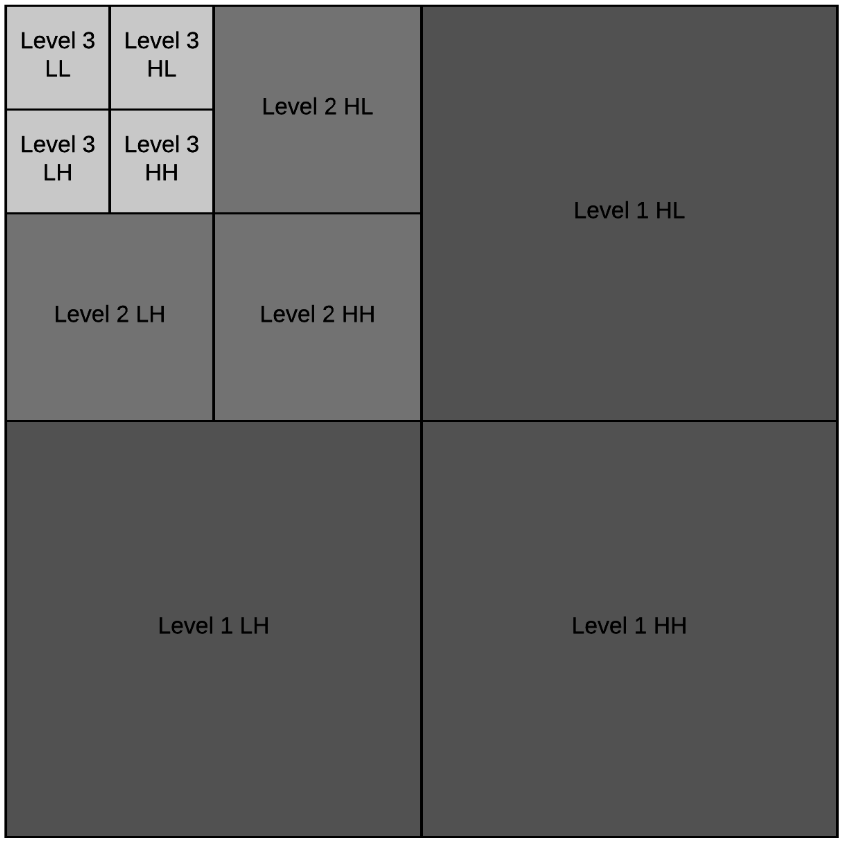

Figure 5 *3-Level 2D DWT Sub-Bands*

6. For each of the 3M+1 bands obtained from DWT, calculate the standard deviation and compute $CSNR_{f_m}^{thr}$ and $C(I_{f_m})$.
7. From the vectors obtained from step 6, calculate $CT(E|I)$ by using L2-norm.

Analyze Distorted Image. Next function in the algorithm is to analyze source image from memory.

1. Compute the error vector (E) of size $m \times n$ from the distorted image computed using following

$$E = \hat{I} - I + \mu_I \tag{16}$$

2. Calculate the mean of the error vector.

3. Convert the error vector into a vector of pixel-to-luminance value. Instead of converting in place, conversion creates a new vector of size $m \times n$.
4. Compute the mean of the pixel to luminance value vector.
5. Calculate the RMS, denoted by $C(E)$, of the source using vectors obtained from step 3 and 4.
6. Check if $C(E) < CT(E/I)$ then exit otherwise continue to step 7.
7. Process error vector with M-level 2D separable discrete wavelet transform (DWT) with 9/7 biorthogonal filters. DWT will result in 3M+1 vectors as specified earlier. This process would take another memory of size $m \times n$ for storing the sub-bands.
8. For each of the bands obtained from DWT calculate the standard deviation and compute $C(E_{f_m})$.
9. Return $C(E_{f_m})$.

Computation of Best CSNR. The goal of this function is to find the $v(E) \in [0, 1]$, such that the resulting $CSNR^*_{f_1}(E)$, $CSNR^*_{f_2}(E)$…, $CSNR^*_{f_m}(E)$ gives rise to a total distortion contrast of $C(E)$.

1. Initialize $v_{lo} = 0$ and $v_{hi} = 1$.
2. Compute $v = \frac{1}{2}(v_{lo} + v_{hi})$
3. Calculate $CSNR^*_{f_1}(E)$, $CSNR^*_{f_2}(E)$…, $CSNR^*_{f_m}(E)$ using equation 10.
4. Compute $\hat{C} = \left(\sum_{m=1}^{M} \left[\frac{C(I_{fm})}{CSNR^*_{fm}} \right]^2 \right)^{1/2}$
5. If $|\hat{C} - C(E)|$ is sufficiently small, then exit.

6. If $\left|\hat{C} - C(E)\right| > 0$ (i.e. v is too large), then let $v_{hi} = v$, and go to step 2
7. If $\left|\hat{C} - C(E)\right| < 0$ (i.e. v is too small), then let $v_{lo} = v$, and go to step 2
8. Iterate steps 2-7 until the function converges.

Computation of VSNR. This is the final function that uses the values computed in the previous functions to measure the visual fidelity metric VSNR.

1. Assign d_{pc} to $C(E)$, where $C(E)$ was obtained in step 5 of Analyze Distorted Image function.
2. Compute the best CSNR ($CSNR_{best}$) from the previous function.
3. Convert the best CSNR to contrast using $C(I_{f_m})/CSNR_{best}$.
4. Compute the actual CSNR ($CSNR_{act}$) using Analyze Distorted Image function.
5. Convert the actual CSNR to contrast using $C(I_{f_m})/CSNR_{act}$.
6. Compute d_{gp} by computing L2-norm of the contrast obtained in steps 3 and 5.
7. Compute visual distortion (VD) using equation 14.
8. Finally, compute VSNR using equation 15.

Performance of VSNR

The performance analysis of VSNR was conducted on a set of seven original images and six different distortions of each of them by Phan, et al. (2014). The platform used for analysis was second-generation Intel Core i5- 2430M processor clocked at 2.4 GHz and a system memory (RAM) of 4 GB. For thirty trials of each image, the average execution time was reported as ~0.72 seconds. Further analysis showed that DWT took ~61% of the total execution time and rest of the functions accounted for the remaining time. Among the remainder of the functions, statistical computation took ~28% of the time. Therefore, the optimization of DWT and variance on GPU could potentially improve the performance of the algorithm. Figure 6 below shows the execution time distribution of each component.

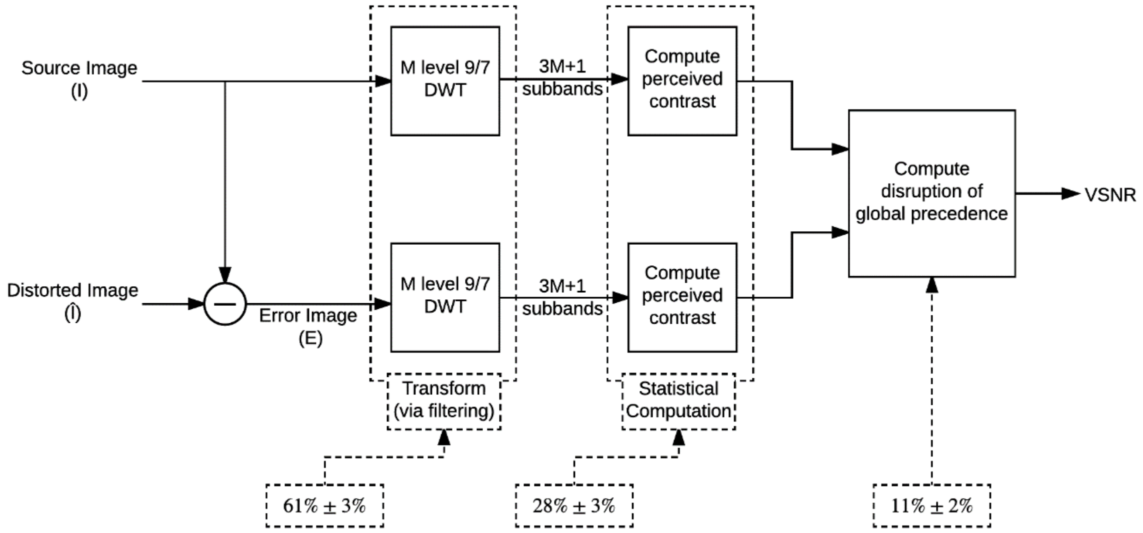

Figure 6 *Execution Time Distribution on CPU (Phan, et al., 2014)*

Chapter 4. CUDA Implementation of VSNR

CUDA implementation is based on the GPGPU as described earlier. The goal of the CUDA implementation is to optimize the VSNR algorithm while minimizing the memory overhead of GPGPU. As reported by Gordon, et al. (2010), the memory transfer from CPU to GPU could become a burden and deteriorate the performance.

Based on the performance analysis from the previous section, this implementation focuses on optimizing the functions that use large vectors with reduced branching and dependency using small vectors, like vectors of size 5, with branching and interdependency.

Figure 7 below distinguish the parts of VSNR algorithm that are computed on GPU and CPU.

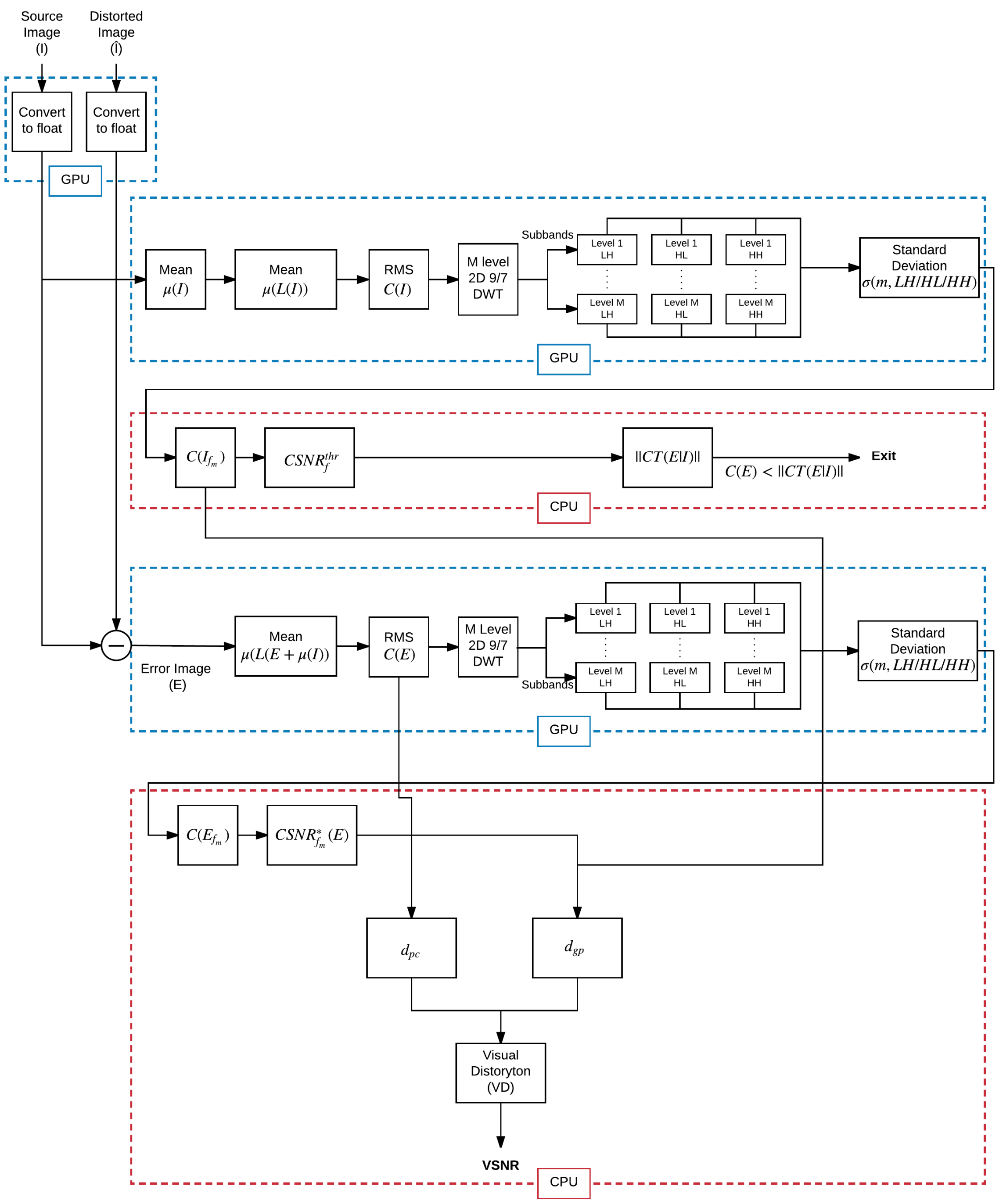

Figure 7 *CPU-GPU Division of VSNR*

Following section includes the detailed implementation of each function using CUDA

Image Loading

Considering the size of the image, loading images to GPU could be an overhead when optimizing algorithms. There are two ways to load the image on GPU.

The first approach is to load the image on CPU as an unsigned character, which is 1 byte per pixel, and then linearize and convert the image to float on CPU, which will result in consuming 4 bytes (size of float) per pixel. This process will take $m \times n$ time. Once converted to float, the image can be loaded to GPU. Figure 8 shows the approach mentioned above.

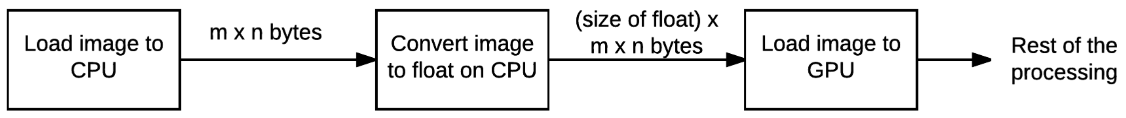

Figure 8 *Approach 1- Loading Image to GPU*

The second approach is to load the image on CPU as an unsigned character and then load the image to GPU. Once the image is loaded onto GPU, the image can be converted to float. Conversion to float process is executed in parallel as the conversion is independent of each pixel. As the image vector is loaded as an unsigned character instead of float on GPU, this approach reduces the memory overhead by the factor of 4 (size of float). Figure 9 shows the second approach of loading image onto GPU.

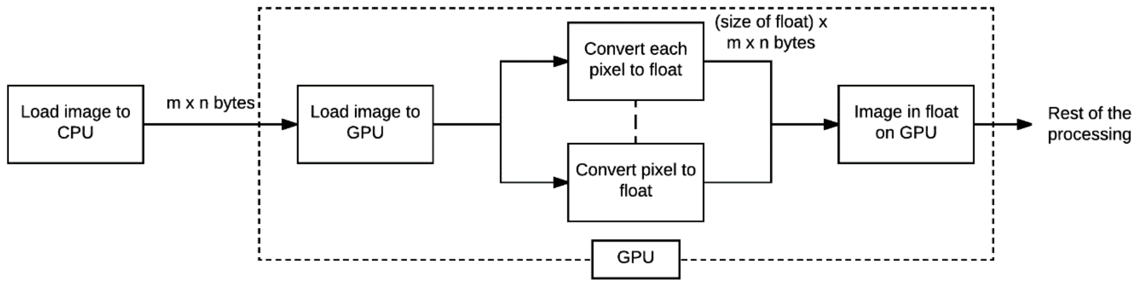

Figure 9 *Approach 2- Loading Image to GPU*

Luminance Vector

Once the images are loaded on the GPU, the images can be used for further computation without transferring them back and forth between CPU and GPU. First computation is the evaluation of pixel-to-luminance vector for original and error image. Additional space of 4 (size of float) $\times\ m\ \times\ n$ is allocated to store the luminance vector for each image. Computation of luminance is independent of each pixel, making it ideal for parallel implementation. By unrolling the iterations, the luminance vector can be computed using equation 1. Figure 10 shows the computation of luminance using arithmetic operations.

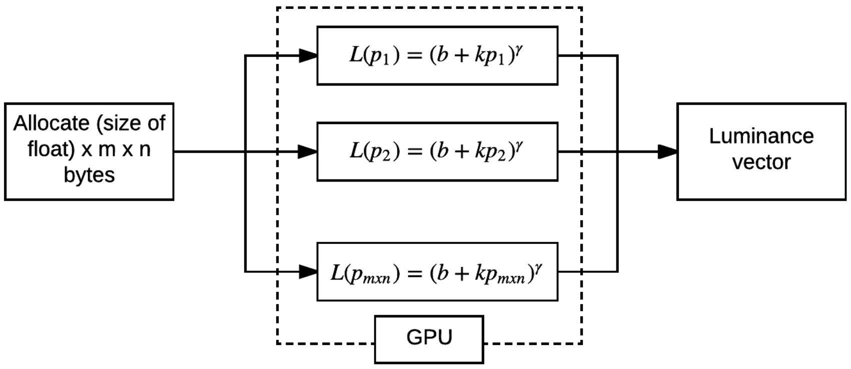
Allocate (size of float) x m x n bytes
$L(p_1) = (b + kp_1)^\gamma$
$L(p_2) = (b + kp_2)^\gamma$
$L(p_{mxn}) = (b + kp_{mxn})^\gamma$
GPU
Luminance vector

Figure 10 *Evaluation of Luminance Vector on GPU*

Another approach to computing the luminance vector is to use the lookup table. Since the pixel value of the image is in the range of 0 to 255, the values from equation 1 will be same for a particular pixel value. Therefore, look up table could optimize the algorithm further. Figure 11 demonstrates the computation of luminance using the lookup table.

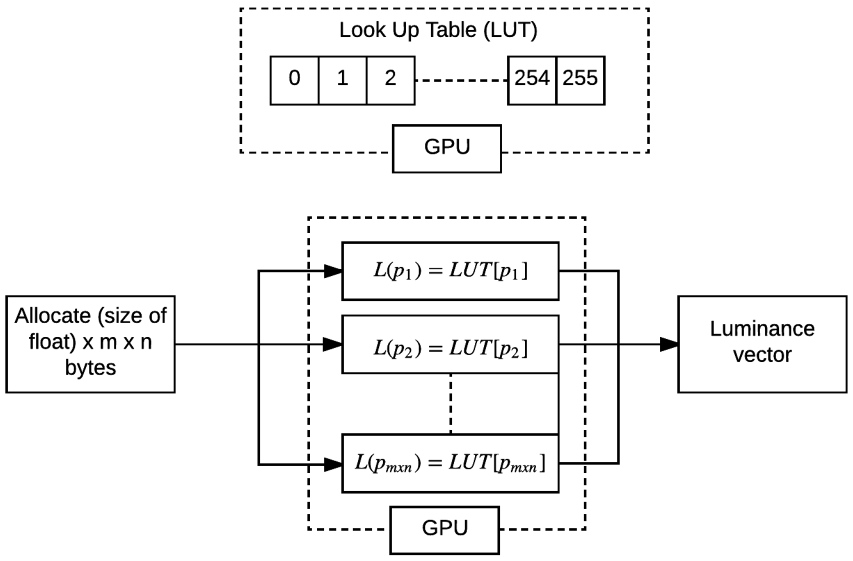

Figure 11 *Evaluation of Luminance Vector Using Lookup Table*

Calculation of Mean

In VSNR, mean needs to be calculated three times. The source image requires the calculation of means of image and luminance vector. And the mean of luminance vector is needed for the error image. Computation of mean takes $m \times n$ time on CPU. Though the sum of pixel values is dependent on values of other pixels, calculation of mean can still be optimized using the reduction technique. ArrayFire library (Yalamanchili, et al., 2015) has been used to calculate the mean of images, which provides a function to calculate mean on GPUs using reduction. Figure 12 demonstrates the sum by reduction for a vector of 16.

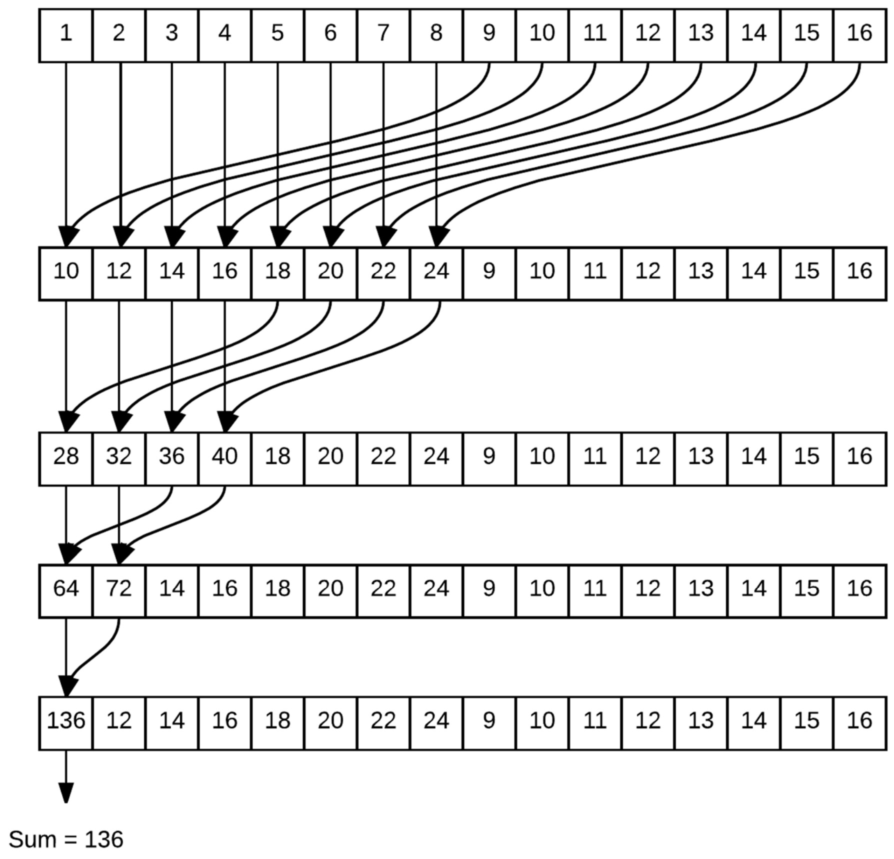

Figure 12 *Example of Reduction on GPU*

Calculation of RMS Contrast

RMS is the standard deviation of a vector as specified in equation 6. Computation of RMS is required for both source and error image and uses mean of luminance from previous procedure to reuse the computation.

In this CUDA implementation, a kernel is first called to compute the value $(L(p) - \mu_L)^2$ and stores the values into the intermediate vector of size $m \times n$. Once the intermediate vector is obtained, the mean function of ArrayFire (Yalamanchili, et al., 2015) is called to compute the mean. The square root of mean and division by mean of luminance gives the standard deviation.

This method allows reusing the mean of luminance without transferring significant memory between host and device. Figure 13 shows the calculation of RMS on GPU.

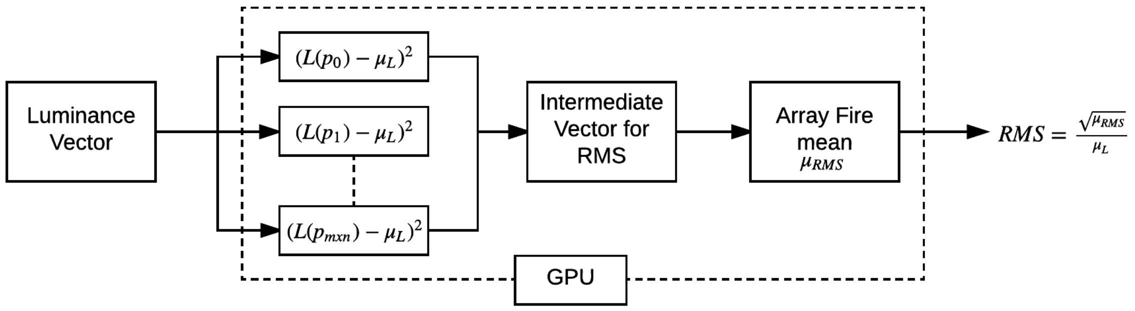
Luminance Vector
$(L(p_0) - \mu_L)^2$
$(L(p_1) - \mu_L)^2$
$(L(p_{mxn}) - \mu_L)^2$
Intermediate Vector for RMS
Array Fire mean μ_{RMS}
$RMS = \frac{\sqrt{\mu_{RMS}}}{\mu_L}$
GPU

Figure 13 *Evaluation of RMS on GPU*

2D Discrete Wavelet Transform

Based on the performance evaluation, 2D DWT is the component that takes ~61% (Phan, et al., 2014) of the total execution time. In DWT, computation on pixels is not completely independent but depends on the value of its neighbors. For each level, two passes are performed, one for the horizontal subsampling and the other for the vertical subsampling.

In horizontal subsampling, the value of pixels at even indexed columns is computed by the neighboring pixels in the corresponding columns using the biorthogonal low and high filters. The pixels at odd indexed columns are left out in resultant vectors and hence reduces the number of columns by two. It results in two sub-bands for both low and high filters of size $(m \times n/2)$.

In vertical subsampling, vectors from horizontal subsampling are used to obtain the value of the pixels at even indexed rows which are computed by the neighboring pixel in the corresponding rows for using the low and high filters. The pixels at odd numbered rows are left out in resultant vectors and thus reduces the number of rows by two. This result in the four sub-bands of size $(m/2 \times n/2)$ i.e. LL, HL, LH, and HH.

The number of neighbors to be selected is determined by the length of the filter. In each pass two filters are used, one is low, and the other is the high filter. For the next level, LL sub-band is used for further decomposition of the image. The algorithm employed in DWT implementation has the influence from Paleo (2015). Figure 14 demonstrates one level of DWT on 8×8 image and a filter of size four on GPU.

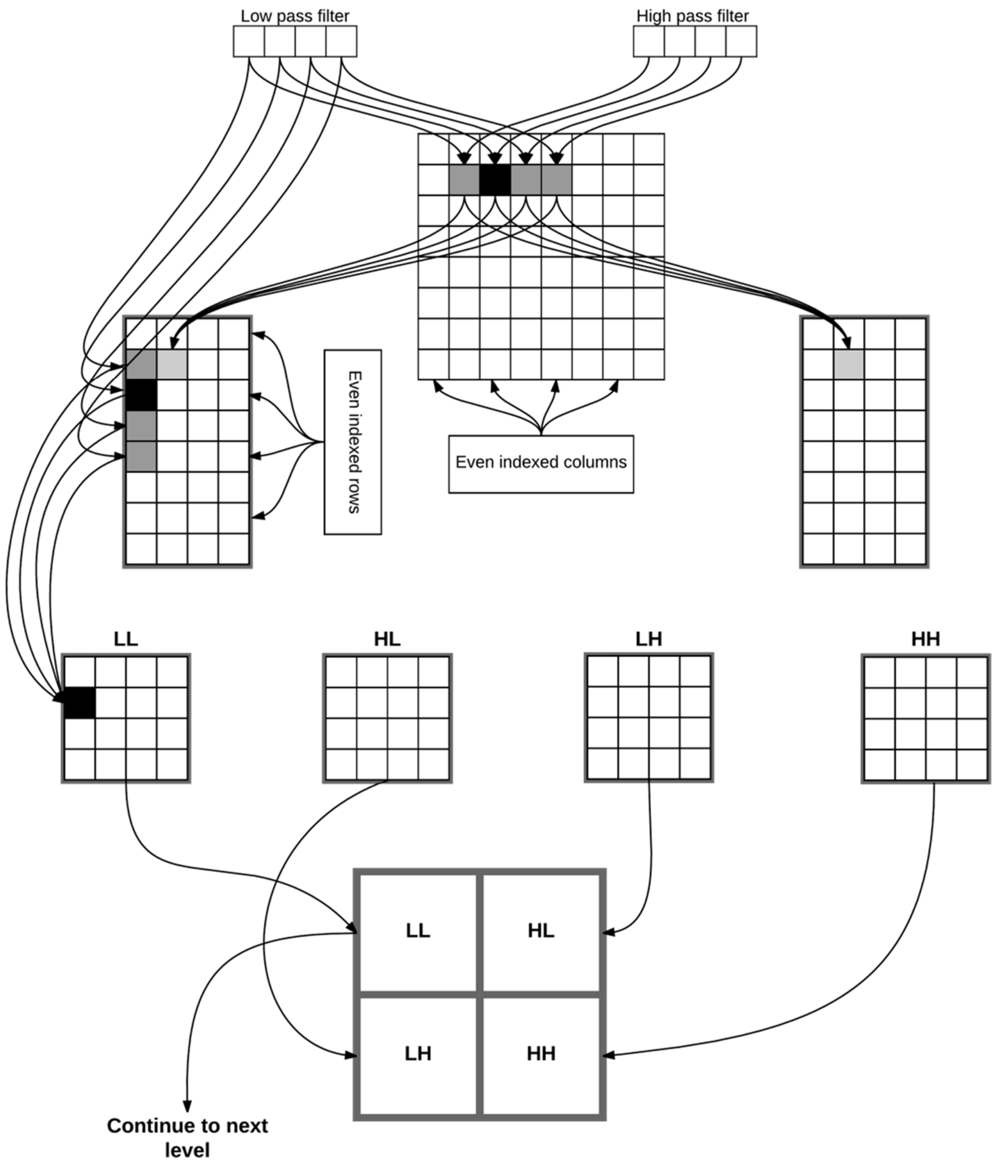

Figure 14 *Example of 2D DWT on GPU*

Computation on Sub-Bands

For the computation of $C(I_f)$ and $C(E_f)$, calculating the standard deviation of sub-bands is needed, which could take significant computation as the size of sub-bands could be large as demonstrated by the Figure 16. There are two approaches to obtain the standard deviation of sub-bands. The first approach is to use ArrayFire (Yalamanchili, et al., 2015) standard deviation function. The second approach is first to calculate mean of the sub-band using ArrayFire (Yalamanchili, et al., 2015) and then modify the sub-band vector to obtain an intermediate vector. Finally, the square root of the mean of the intermediate vector is computed. Following equation denotes the second approach to computing standard deviation.

$$\sigma(V) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(V(i) - \mu_V)^2} \tag{17}$$

where V denotes the sub-band vector.

Due to lack of performance of ArrayFire's (Yalamanchili, et al., 2015) standard deviation function, the second method has been used to perform computation on sub-bands. Figure 15 shows the evaluation of standard deviation of sub-bands using the second method on GPU.

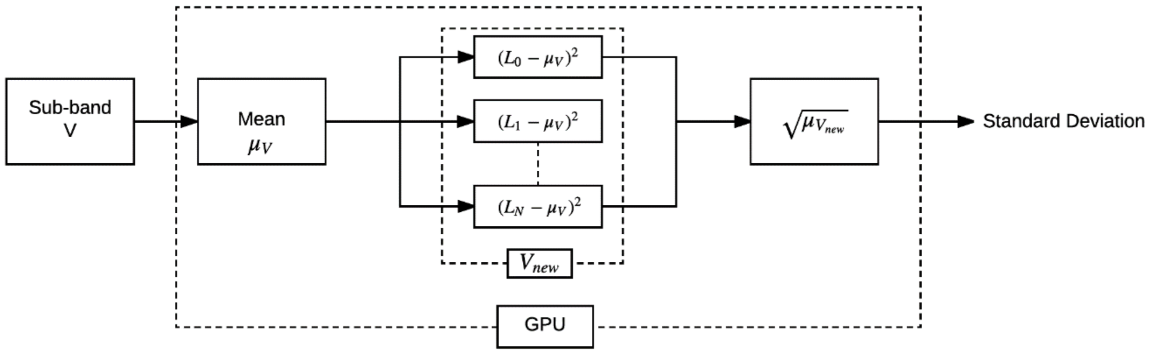

Figure 15 *Evaluation of Standard Deviation of Sub-Bands on GPU*

Figure 16 demonstrates the computation on sub-bands.

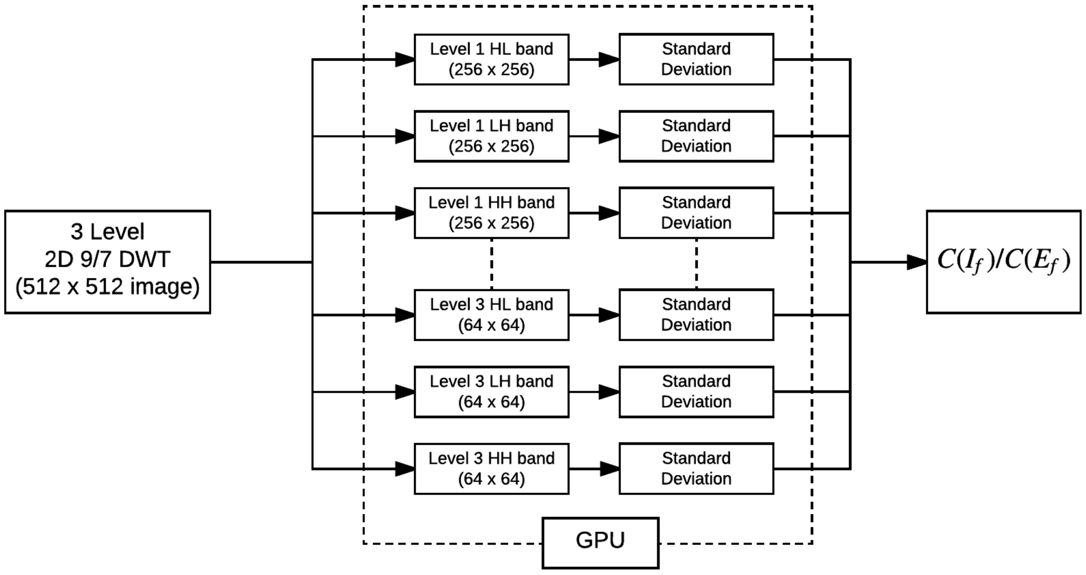
3 Level
2D 9/7 DWT
(512 x 512 image)
Level 1 HL band (256 x 256)
Level 1 LH band (256 x 256)
Level 1 HH band (256 x 256)
Level 3 HL band (64 x 64)
Level 3 LH band (64 x 64)
Level 3 HH band (64 x 64)
Standard Deviation
GPU
$C(I_f)/C(E_f)$

Figure 16 *Computation on Sub-Bands on GPU*

Chapter 5. Methodology

All the tests use a system with the i7 processor and Nvidia Tesla K40 with Kepler GK110 microarchitecture with CUDA 8. Table 1 and Table 2 provides more details about the system and the GPU configuration.

The development and testing have been conducted using Microsoft Visual Studio 2015. Nvidia Visual Profiler (Nvidia Visual Profiler, 2017) has been used to find bottlenecks of the implementation that helped to optimize the algorithm and to evaluate performance.

CPU	Intel Xeon E5 1620 v2 @3.70 GHz
Microarchitecture	Ivy Bridge
No of Cores	4
No of Threads	8
Host RAM	24 GB GDDR3
Operating System	Windows 7 (64-bit)
IDE	Visual Studio 2015

Table 1 *Experiment Environment Details*

Device Name	Tesla K40c
CUDA Driver Version	8.0
CUDA Capability	3.5
Number of multiprocessors	15
Number of CUDA cores per multiprocessor	192
Total number of CUDA cores	2880
GPU max clock rate	745 MHz (0.75 GHz)
Memory clock rate	3004 Mhz
Memory bus width	384-bit
Global Memory Size	11423 MB
Constant memory size	64 KB
Shared Memory size per block	48 KB
Number of registers per block	65536
Warp size	32
Maximum number of threads per block	1024
Maximum number of threads per multiprocessor	2048

Table 2 *GPU Specifications*

The goal of the experiments is to find the correct speed up of CUDA implementation of VSNR. To achieve accurate results, tests use four databases containing images of different sizes and distortions. Each image from the database has first been converted to grayscale to make it compatible with VSNR.

1. Seven original images of size 512 x 512 have been used from CSIQ database (Larson & Chandler, 2010). Each image is associated with three different distortions, Additive Gaussian White Noise (AWGN), Gaussian blurring, and JPEG compression, with two levels of distortions for each image. Level 1 is for low-distorted images (AWGN1, BLUR1, and JPEG1) and level 5 for highly distorted images (AWGN5, BLUR5, and JPEG5), which is aligned with the previous experiments reported by Phan, et al. (2014).
2. Fourteen original images of size 512 x 768 have been used from IRCCyN/IVC database (Tourancheau, Autrusseau, Sazzad, & Horita, 2008). Each image has two different distortions, JPEG and JPEG2000, and two levels, 0.24 and 0.79, for each distortion.
3. Six original images of size 1920 x 1080 have been used from JPEG-HDR database (Narwaria, Da Silva, Le Callet, & Pepion, 2013). Each original image has distortion of type JPEG with level 1 and level 5. Images have been resized to 1920 x 1024 to ensure compatibility with the CPU version of VSNR.
4. Six original images of size 1280 x 1600 are used from JPEG XR database (De Simone, Goldmann, Baroncini, & Ebrahimi, 2009). Each image is with distortions created by JPEG, JPEG2000 and JPEG XR, with levels 0.25 and 0.75.

Table 3 summarizes the details of image quality databases used for the experiment.

Database	**Image Size**	**Distortion Types**	**Distortion Levels**	**No. of Original Images**	**Number of distorted images**
CSIQ	512 x 512	Blur, AWGN, JPEG	1 and 5	7	42
IRCCyN/IVC	512 x 768	JPEG, JPEG2000	0.24 and 0.79	14	56
JPEG-HDR	1920 x 1080	JPEG	0.22 and 0.70	6	12
JPEG XR	1280 x 1600	JPEG, JPEG2000, JPEG XR	0.25 and 0.75	6	36

Table 3 *Image Quality Database Details*

Each combination of the original and the distorted image has been executed on CPU and GPU as a new process for thirty iterations. Running each iteration as a new process ensures that the reported time is close to the processing time of the algorithm in real-time applications. Average of execution time of images of same size has been taken to obtain the approximate execution times for different sizes of images.

In addition to this, an image of size 1920 x 1024 from JPEG-HDR database (Narwaria, et al., 2013) has been utilized to evaluate the performance of individual functions on both C++ and CUDA implementations of VSNR.

Chapter 6. Results

This section presents the result from the experiments introduced in the previous chapter. As expected, the execution of the CUDA implementation of VSNR on GPU showed performance optimization over its C++ implementation, executed on CPU while matching the final fidelity score of VSNR.

Table 4 shows the execution times of CUDA implementation and C++ implementation of VSNR for different image sizes and the bar graph in Figure 17 plots the execution time of VSNR on CPU and GPU on the logarithmic scale. The last column of the table shows the speed up received by CUDA implementation as compared to C++ implementation.

Database	Image Size	Time (ms)	Platform	Speed Up (CPU/GPU)
CSIQ	512 x 512	57	CPU	8.14x
		7	GPU	
IRCCyN/IVC JPEG HDR	512 x 768	86	CPU	10.75x
		8	GPU	
JPEGXR IRCCyN/IVC	1024 x 1920	484	CPU	32.27x
		15	GPU	
JPEG-HDR	1600 x 1280	493	CPU	32.87x
		15	GPU	

Table 4 *Execution Time and Speed Up of VSNR on CPU and GPU*

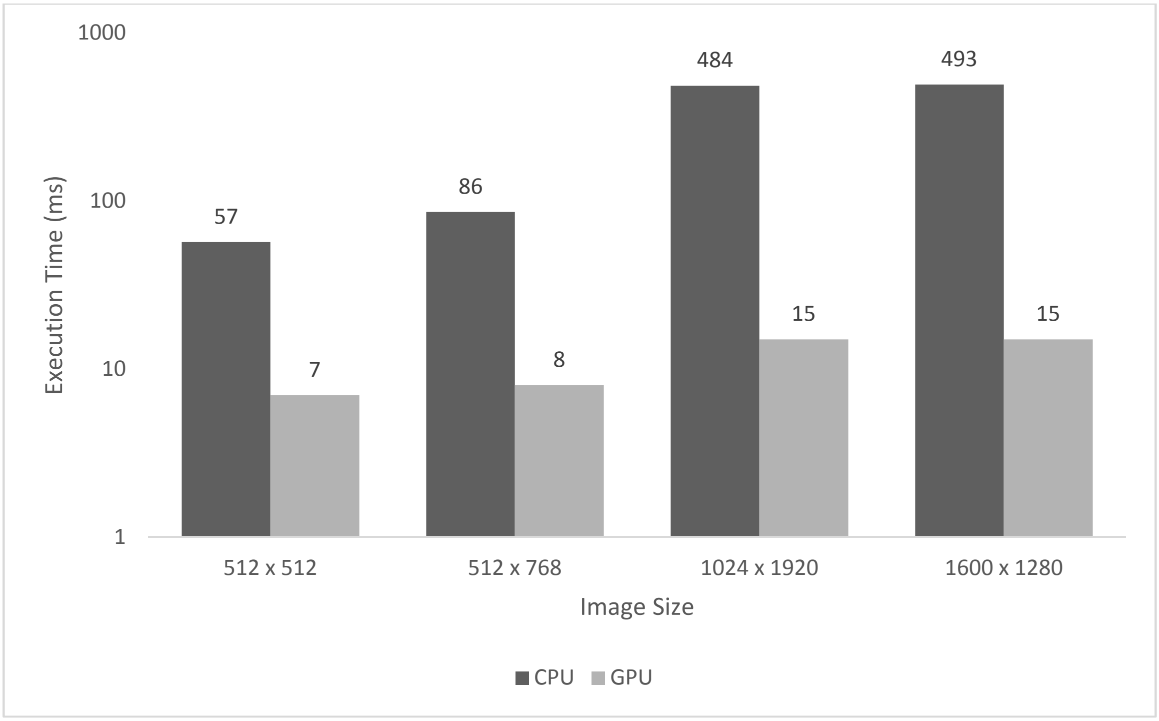

Figure 17 *Execution Time of VSNR on CPU and GPU*

The graph in Figure 18 shows the speedup vs. image size. It indicates that speed up is monotonically increasing with the increase in the scale of the image.

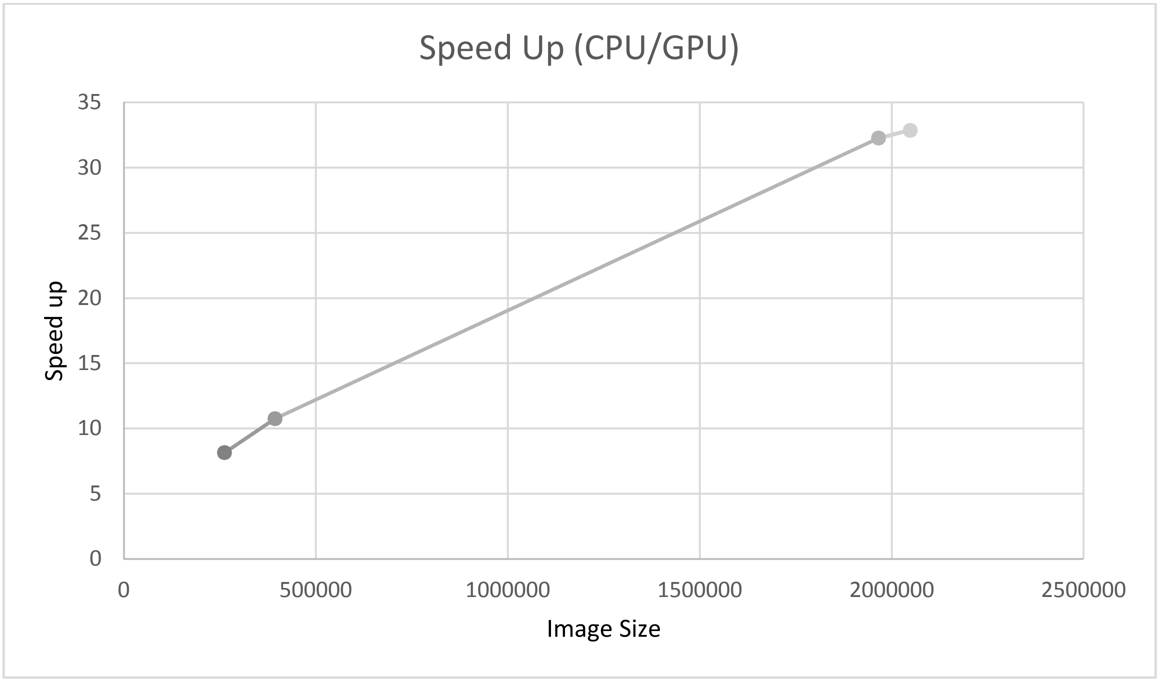

Figure 18 *Speedup Curve*

Table 5 shows the performance of functions that were executed on GPU, which were computed using Nvidia Visual Profiler (Nvidia Visual Profiler, 2017).

Function	GPU		CPU		Speed Up
	Absolute time (ms)	Percentage of time	Absolute time (ms)	Percentage of time	
DWT	5.333	35.55	314.116	64.9	58.9
RMS	0.98	6.53	35.816	7.4	36.55
Computation on sub-bands	3.55	23.67	31.46	6.5	7.43
Mean of Original Image	0.063	0.42	8.712	1.8	138.29

Table 5 *Execution Time Breakdown of Primary Functions*

Significant time is spent on the allocation of memory and copying data across the device and between the device and the host. Table 6 shows the time consumed on memory allocation and copying.

Method Type	Execution Time (ms)
Memory copy from host to device	0.51
Memory copy from device to device	0.086
Memory copy from device to host	0.14
Memory allocation on device	1.6
Total	2.336

Table 6 *Time Consumed in Memory Transfer and Allocation*

During the memory allocation on the device, the majority of the time was spent on the allocation of sub-bands, which are 3M+1, and took approximately 1ms time.

Analysis of Image Loading

As discussed in the Chapter 4, the original and the error image can be loaded using two methods. The first method converts the image on CPU and then copies it to GPU. The second method copies the image to GPU and then converts it into a float vector. The latter method is the most efficient and was adopted in the final version of the implementation. Table 7 shows the comparison of execution times between the two approaches.

Method	Method Execution Time (ms)	Total Execution Time (ms)
Convert on CPU	174	189
Convert on GPU	1.09	15

Table 7 *Execution Time Comparison of Two Approaches to Load Image to GPU*

The significant difference in the execution times of the two methods is due to the slow and sequential processing on CPU. The computation of error image itself takes 143ms due to the complicated type casting and arithmetic operations. Another factor affecting the performance is the cudaMemcpy function which copies image vector to GPU of size four times larger as compared to that of the vector when converting image directly on GPU.

Analysis of Computation on Sub-Bands

Table 8 compares the results obtained from computing the sub-bands using the both approaches mentioned in the Chapter 4. The first method uses ArrayFire's standard deviation function, and the second method first calculates the mean, then gets the intermediate vector and finally computes the standard deviation by taking the square root of mean. The prior approach was found slower due to the overhead of CUDA runtime, cudaMemCpy, and cudaFree functions where were controlled by adopting the second approach and reducing the redundancy of cudaFree calls.

Method	Method Execution Time (ms)	Total Execution Time (ms)
ArrayFire’s Standard Deviation Function	99.47	112
Custom Standard Deviation Function	3.55	15

Table 8 *Execution Time Comparison of Two Approaches to Compute Sub-Bands*

Analysis of Computation of Luminance

The computation of RMS of images requires the calculation of luminance. Two methods were discussed in Chapter 4 for computing the luminance. One method is to use a lookup table, and another method is to compute the luminance vector using arithmetic computation. Table 9 shows the comparison between execution times of the two approaches.

Method	Method Execution Time (ms)	Total Execution Time (ms)
Arithmetic computation	0.632	16
Look up table	0.363	15

Table 9 *Execution Time Comparison of Two Approaches to Calculate Luminance*

Figure 19 shows the breakdown of execution time on a GPU.

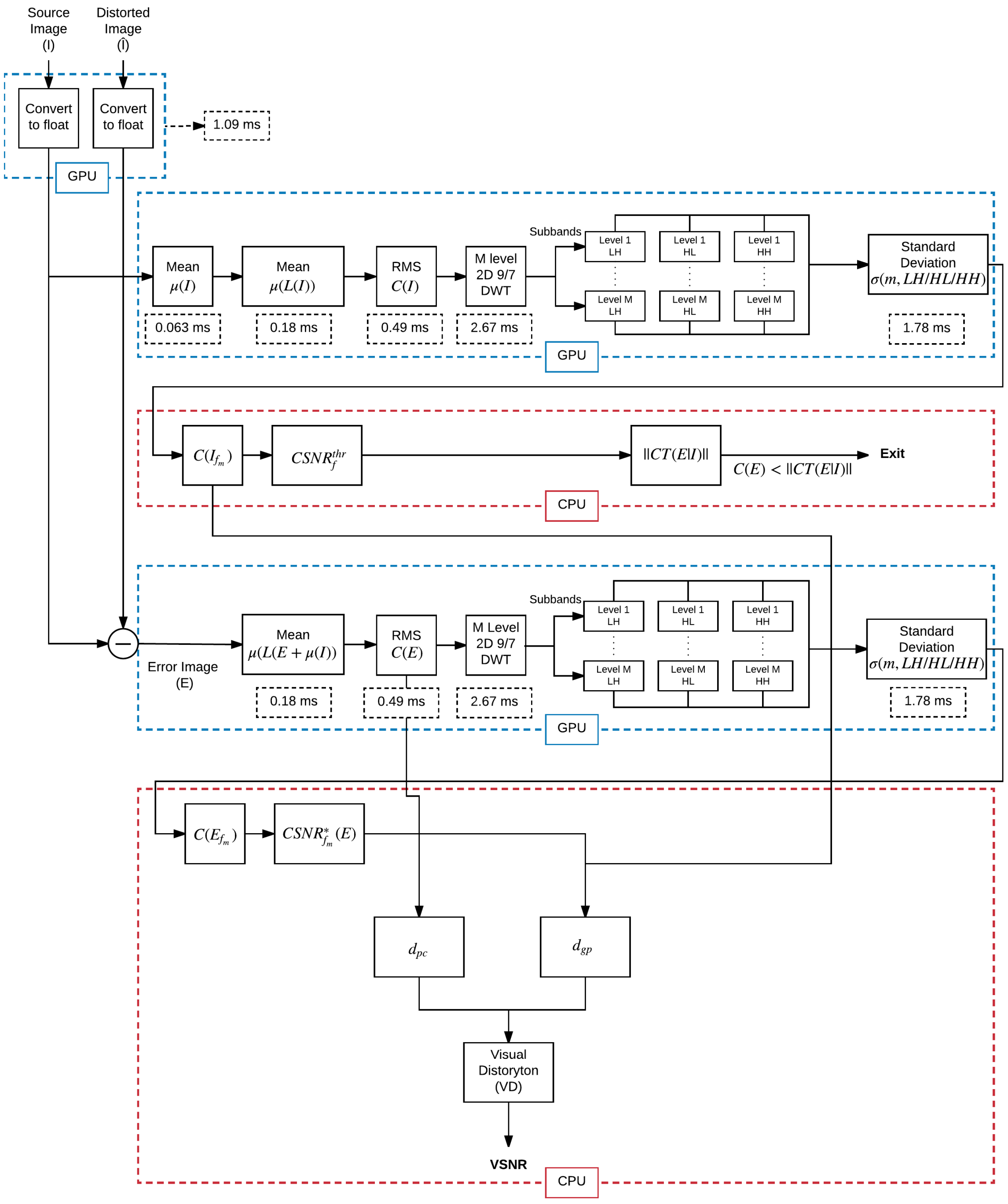

Figure 19 *Breakdown of Execution Time*

Chapter 7. Discussion of Results

The results as presented in previous chapter, show that the performance of CUDA implementation improved when the size of the images increased. The speed up monotonically increases with the increase in image size and varies between 8x and 32x for image sizes 512 x 512 and 1600 x 1280 respectively. With the increase in the size of images, the performance of the algorithm is affected more in the case of C++ implementation as compared to the CUDA implementation. Due to parallelism, CUDA implementation is expected to show greater speed up with image sizes larger than 1600 x 1280.

Execution time can be affected by multiple memory allocation on device as shown in the results section. Computation of sub-bands of the error image reused the memory allocated for sub-bands of the original image instead of allocating new memory. This method reduced the time spent on memory allocation for sub-bands by half. Though allocating memory is inevitable, number of allocation can be minimized by reusing the allocated memory for different functions and hence improves the performance.

As the results and previous research (Gordon, et al., 2010) suggest, copying memory from CPU to GPU could be an overhead. A method to reduce this overhead, i.e. copying image as character, was introduced which led to the reduction in execution time by more than half. It can be suggested that the least amount of memory should be transferred between CPU and GPU, and statistical computations on matrices should be maximized on GPUs in parallel.

As reported by Phan, et al. (2014), 2D DWT was the hotspot of CPU computation. After implementing it on CUDA, 2D DWT remains the hotspot as compared to other functions but the execution time is relatively minute. 2D DWT works well on GPU because of massive parallelism and contiguous memory usage among threads.

The results show that the better approach to load image onto GPU is by loading the image to GPU as character and then converting the image to float. The parallel casting of image along with reduction of memory size that needs to be copied to GPU improves the performances. Thus, the execution time can be reduced by decreasing the memory transfer between the host and the device as in the case of loading image onto GPU.

The computation of sub-bands was optimized by avoiding a standard deviation library function which had side effects that degraded the performance. Using library functions are convenient but it is important to carefully observe if these functions are appropriate for the execution of an algorithm and do not create any side effects or overhead.

GPU has a constant memory which can be used to avoid computation. In the case of computation of luminance, the vector was using the constant value for all images. Thus, look up table outperformed runtime computation. The constant memory on GPU is relatively smaller. Therefore, it is necessary to use look up table for smaller constant vectors. Otherwise, constant vector would be fetched from the global memory, which in turn degrades performance. It is also important to make sure that the statistical operation takes more time than it would take GPU to fetch from the constant memory.

Reduction used for calculating mean improved performance tremendously. The performance of obtaining the mean of original image improved by 138x as compared to

CPU performance. Computation of mean was not just used for the original image but also for the calculation of luminance and sub-bands. In both computations speed up has been observed. Though, in the computation of mean, pixel values are dependent on each other, reduction method optimize the performance effectively.

Based on the results, it can be interpreted that an algorithm can be broken down into smaller parts to optimize performance. The blend of library functions and user defined functions with profiling conscious implementation can lead to an efficient implementation.

Chapter 8. Conclusion

In this thesis, the implementation of VSNR in CUDA is presented. The CUDA implementation runs 32x faster as compared to the C++ implementation of VSNR for large images. The speedup was observed due to memory optimized execution of arithmetic operations on pixels in parallel. The CUDA implementation can process videos with the frame size of 1600 x 1280 and frame rate of 60 frames per second whereas it can process videos with the smaller frame size of 515 x 512 and frame rate of over 140 frames per second.

The experiments were conducted as an extension to work by Phan, et al (2014) to find the change in the performance of the C++ implementation of VSNR with the increase in the size of images. As reported in Chapter 7, the hotspot of the C++ implementation remained same as the size of the image increased but the execution time increased significantly with size.

Results in Chapter 6 revealed important observations regarding GPGPU programming for image processing algorithms. Firstly, loading of images on GPU as an unsigned character and casting image from unsigned char to float on GPU resulted in performance optimization due to the reduced size of memory transfer between CPU and GPU. Secondly, profiler conscious implementation helped in the detection of runtime overhead due to the standard library function. Lastly, storing lookup table in GPU constant memory to replace redundant arithmetic computation, resulted in optimization. The listed methods can help in reducing overheads and optimizing performance of other CUDA based implementation of image processing algorithms.

Chapter 9. Future Work

The work presented in the thesis allows us to analyze quality of the high definition images and videos in real time. Though this implementation will work for the high definition of videos with frame rate of 60fps, more optimizations can be made specific for the video quality assessments (VQA). Firstly, since the scope of this work is focus on image quality assessment the results reported in this thesis are for a single image but more speed up can be expected reusing the allocated memory on the GPU, in the case of videos. Secondly, the subsequent frames in videos are not completely different and the algorithm might be modified to take benefit of this fact. Lastly, the continuous frames in the video could allow us to use multiple GPU for processing each frame in parallel. This could be another prospect for optimization.

Apart from this, similar performance analysis of VSNR for IQA or VQA can be performed using cloud based techniques like MapReduce or customized image processing hardware.

Another work effort can be made to optimize the time taken by the cudaMalloc statements in similar implementation. For a single image, the memory allocation can take significant time. An effort to reduce the memory allocation using CPU threads could be useful.

Combining this CUDA implementation with compression algorithms would provide the more insights into performance based on a specific utility of this algorithm.

References

Chandler, D. M. (2013). Seven challenges in image quality assessment: past, present, and future research. *ISRN Signal Processing. 2013.* Hindawi Publishing Corporation.

Chandler, D. M., & Hemami, S. S. (2007). VSNR: A wavelet-based visual signal-to-noise ratio for natural images. *16*, pp. 2284-2298. IEEE.

Chen, M.-J., & Bovik, A. C. (2011). Fast structural similarity index algorithm. *Journal of Real-Time Image Processing, 6*(4), 281-287.

CUDA C Programming Guide. (2015). Nvidia Corporation.

De Simone, F., Goldmann, L., Baroncini, V., & Ebrahimi, T. (2009). Subjective evaluation of JPEG XR image compression. *Proc. SPIE, 7443*, 74430L.

Gordon, B., Sohoni, S., & Chandler, D. M. (2010). Data handling inefficiencies between CUDA, 3D rendering, and system memory. *Workload Characterization (IISWC), 2010 IEEE International Symposium on* (pp. 1-10). IEEE.

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

Intel. (2017, February). Intel's VTune Amplifier 2016. Retrieved 2017, from Intel: https://software.intel.com/en-us/intel-vtune-amplifier-xe

Kannan, V., Holloway, J., Sohoni, S., & Chandler, D. M. (2017). Microarchitectural analysis of a GPU implementation of the Most Apparent Distortion image quality assessment algorithm. *Electronic Imaging, 2017*(12), 36-41.

Larson, E. C., & Chandler, D. M. (2010). Most apparent distortion: full-reference image quality assessment and the role of strategy. *Journal of Electronic Imaging, 19*(1), 011006-011006.

Li, C., Wu, H., Chen, S., & Li, X. (2009). Efficient implementation for MD5-RC4 encryption using GPU with CUDA. *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on* (pp. 167-170). IEEE.

Mittal, A., Moorthy, A. K., & Bovik, A. C. (2012). No-reference image quality assessment in the spatial domain. *IEEE Transactions on Image Processing, 21*(12), 4695--4708.

Narwaria, M., Da Silva, M. P., Le Callet, P., & Pepion, R. (2013). one mapping-based high-dynamic-range image compression: study of optimization criterion and perceptual quality. *Optical Engineering, 52*(10), 102008--102008.

Nvidia Visual Profiler. (2017). Retrieved from Nvidia Corporation: https://developer.nvidia.com/nvidia-visual-profiler

Okarma, K., & Mazurek, P. (2011). GPGPU based estimation of the combined video quality metric. *Image Processing and Communications Challenges 3*, 285-292.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., & Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer graphics forum, 26*(1), 80-113.

Paleo, P. (2015, December 23). *PDWT*. Retrieved February 2017, from Github: https://github.com/pierrepaleo/PDWT

Park, I. K., Singhal, N., Lee, M. H., Cho, S., & Kim, C. (2011). Design and performance evaluation of image processing algorithms on GPUs. *IEEE Transactions on Parallel and Distributed Systems, 22*(1), 91-104.

Phan, T. D., Shah, S. K., Chandler, D. M., & Sohoni, S. (2014). Microarchitectural analysis of image quality assessment algorithms. *Journal of Electronic Imaging, 23*(1), 013030-013030.

Phan, T., Sohoni, S., Chandler, D. M., & Larson, E. C. (2012). Performance-analysis-based acceleration of image quality assessment. *Image Analysis and Interpretation (SSIAI), 2012 IEEE Southwest Symposium on*, 81-84.

Saad, M. A., Bovik, A. C., & Charrier, C. (2012). Blind image quality assessment: A natural scene statistics approach in the DCT domain. *IEEE Transactions on Image Processing, 21*(8), 3339-3352.

Sheikh, H. R., & Bovik, A. C. (2006). Image information and visual quality. *IEEE Transactions on image processing, 15*(2), 430-444.

Tourancheau, S., Autrusseau, F., Sazzad, Z. P., & Horita, Y. (2008). Impact of subjective dataset on the performance of image quality metrics. *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference o* (pp. 365-368). IEEE.

Wang, Z., & Simoncelli, E. P. (2005). Reduced-reference image quality assessment using a wavelet-domain natural image statistic model. *Electronic Imaging 2005*, 149-159.

Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing, 13*(4), 600-612.

Wang, Z., Simoncelli, E. P., & Bovik, A. C. (2003). Multiscale structural similarity for image quality assessment. *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on. 2*, pp. 1398--1402. IEEE.

Yadav, A. (2016). *GPGPU based Implementation of BLIINDS-II NR-IQA.* Arizona State University. (Unpublished master's thesis).

Yadav, A., Sohoni, S., & Chandler, D. (2017). GPGPU based implementation of a high performing No Reference (NR)-IQA algorithm, BLIINDS-II. *Electronic Imaging, 2017*(12), 21-25.

Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P. a., Kloppenborg, B., Malcolm, J., & Melonakos, J. (2015). *ArrayFire - A high performance software library for parallel computing with an easy-to-use API.* (AccelerEyes, Producer) Retrieved from ArrayFire: https://github.com/arrayfire/arrayfire

Yang, Z., Zhu, Y., & Pu, Y. (2008). Parallel image processing based on CUDA. *Computer Science and Software Engineering, 2008 International Conference on. 3*, pp. 198-201. IEEE.

Yang, Z., Zhu, Y., & Pu, Y. (2008). Parallel image processing based on CUDA. *Computer Science and Software Engineering, 2008 International Conference on. 3*, pp. 198-201. IEEE.