PERFORMANCE AND MICROARCHITECUTRAL ANALYSIS FOR IMAGE QUALITY ASSESMENT ALGORITHMS

By

SIDDHARTH K. SHAH

Bachelor of Engineering

University of Pune

Pune, Maharashtra, India

2009

Submitted to the Faculty of the
GraduateCollege of the
OklahomaStateUniversity
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2012

PERFORMANCE AND MICROARCHITECURAL ANALYSIS FOR IMAGE QUALITY ASSESMENT ALGORITHMS

Thesis Approved:

Dr. Sohum Sohoni

ThesisAdvisor

Dr. Damon Chandler

Dr. John Acken

ACKNOWLEDGMENTS

I would like to thank my Adviser, Dr. Sohum Sohoni for his help and support in the work presented in this thesis. He has been a great advisor and guide. I would also like to thank my lab mates Randal Allison for proof reading my thesis document and Wira Mulia for making me understand some important concepts.

I would also like to thank Dr. Damon Chandler and Thien for their research on Image Quality Assesment without them this thesis would not have been possible.

I would like to thank Dr. John Acken for his comments and guidelines for this thesis document. A special thanks to the researchers and contributors on the Intel VTune Amplifier XE forum for patiently answering and commenting on my queries about the tool and performance analysis.

Lastly, I want to thank my parents, Reema and all my other friends who have always been a driving force in my life.

.

Name: SIDDHARTH K. SHAH

Date of Degree: DECEMBER, 2012

Title of Study: PERFORMANCE AND MICROARCHITECTURAL ANALYSIS OF IMAGE QUALITY ASSESSMENT ALGORITHMS

Major Field: ELECTRICAL ENGINEERING

This thesis presents performance analysis for five matured Image Quality Assessment algorithms: VSNR, MAD, MSSIM, BLIINDS, and VIF. The performance parameter considered is execution time. First, we conduct hotspot analysis to find the most time consuming sections for the five algorithms. Second, Microarchitecural analysis is conducted to analyze the behavior of the algorithms for Intel's sandy Bridge microarchitecture and find architectural bottlenecks. The current research for improving performance for IQA algorithms is based on advanced signal processing techniques. This research focuses on the behavior of IQA algorithms with underlying hardware and architecture. We study the behavior of these algorithms with the architectural resources and propose techniques to improve performance using coding techniques that exploit the hardware resources and consequently improve the execution time and computational performance. Along, with software tuning methods, we also propose a generic custom IQA hardware engine based on the microarchitectural analysis and the behavior of these 5 IQA algorithms with the underlying microarchitectural resources.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This research thesis is based on performance analysis of the Image Quality Assessment (IQA) algorithms. Image Quality Assessment is a technique to automatically judge visual quality of an image. IQA plays a very important role in numerous image processing applications. Currently, research in IQA spans two different domains. First, developing better IQA algorithms which agree more closely with the human visual system. Second, improving the computational performance and execution time for the current matured IQA algorithms. This thesis research focuses on the later research area.

This chapter introduces several aspects of the research work, namely a brief background discussion on the Image Quality Assessment (IQA) algorithms, current state of research in IQA, objectives and goals for the research. Section 1.1 presents a brief introduction to IQA and current state of research in its realm. Further, Section 1.2 discusses the current signal processing techniques and methodologies used by researcher to improve computational performance and execution time for the various IQA algorithms. Section 1.3 discusses the key issues the research addresses. Finally, section 1.4 gives an overview of the experimental setup and methodology for performance profiling.

1.1 Background on Image Quality Assessment Algorithms

The visual quality of a digital image can degrade when they are captured, stored (compressed), transmitted or processed. Such degradation can change the appearance of an image, thus it is important to judge an image's visual quality before it is displayed to or used by the consumer. Many algorithms have been developed since the origin of research on IQA. IQA algorithms aim to provide an automated means of judging image's visual quality which is concurrent with human judgments of quality. Currently, IQA research is an active sub discipline of image processing and benefits a wide variety of applications ranging from image compression (e.g., [1]–[3]), to denoising (e.g., [4]), to predicting intelligibility in sign language video [5].

There are currently two categories of IQA algorithms. The first is full reference IQA algorithms, where a reference image has a good/acceptable visual quality and a subject image (distorted version) whose visual quality needs to be judged is used. The output of such algorithms is either a scalar value denoting the overall visual quality or a spatial map estimating the local quality of each image region (e.g., [6]–[34]). More recently, researchers have begun to develop a second category of IQA algorithms. These are *no-reference* or *reduced-reference* algorithms. These algorithms estimate visual quality of an image without using a reference image or just use partial information about the reference image (no-reference IQA; e.g., [35]–[39]), (reduced-reference IQA; e.g., [40]–[45]).

With consistent research in IQA, all three types of IQA algorithms have been able to predict the visual quality of an image efficiently which agrees with human judgments of an image's visual quality. Some of the best-performing full-reference algorithms such as MS-SSIM [29], VIF [30], and MAD [34] estimate the visual quality of the image that highly correlates with human ratings of quality, typically yielding Spearman and Pearson correlation coefficients (measure of dependence between two variables reference and subject image in our case) in excess of 0.9. Research in no-reference and reduced-reference IQA is much less mature; but, recent methods such as DIIVINE [37], BRISQUE [38], and BLIINDS-II [39], yield quality estimates which also highly correlate with human ratings of quality.

1.2 Signal Processing based methodologies and techniques for improving computational performance

From a signal-processing viewpoint, it seems that the majority of computation and runtime are likely to occur in two key stages, employed by most IQA algorithms: (1) local frequency-based decompositions of the input image(s); and (2) local statistical computations on the frequency coefficients. The local frequency- based decomposition stage would require higher computation power and memory bandwidth as well. Particularly, when a large number of frequency bands are analyzed, and when the decomposition must be applied to the image as a whole. On the other hand, the statistical computation on frequency coefficients would seem to require higher computation power, mainly when multiple statistical computations are computed for each local region of coefficients. For example, in MS-SSIM [29] an image is decomposed into different scales, and for each block of coefficients local image statistics are computed (via a sliding window). In VIF [30], wavelet subband covariances can be computed via a block-based or overlapping block-based approach. In MAD [34], variances, skewnessness, and kurtoses of log-Gabor coefficients are also computed for overlapping blocks in each subband. These approaches mimic the cortical processing in the human visual system (HVS) for some aspects. The statistics of local responses of neurons in primary visual cortex (modeled as coefficients) are computed and compared in higher-level visual areas. But still unlike the HVS, most modern computing platforms lack dedicated hardware for computing the coefficients and their local statistics.

The IQA algorithms are extensive used in image compression and computer vision. Consequently, a considerable amount of research focus has been on accelerating two-dimensional image transforms which provide local frequency-based decompositions which is one of the major operations in IQA that requires high computation power and memory bandwidth. For example, using variations of the same techniques used in the Fast Fourier Transform (FFT) the discrete cosine transform (DCT) has been accelerated at the algorithm level (e.g., [48]). Also, there has been significant improvement in performance by exploiting various algebraic and structural properties of the transform, e.g., via recursion [49], lifting [50], matrix factorization [51], cyclic convolution [52], and many other techniques (see [53] for a review). Along with techniques at the algorithm level, many techniques for hardware-based acceleration of the DCT have also been proposed using GPGPU-based and FPGA-based implementations (e.g., [54]–[57]). Examples for algorithm and hardware-based acceleration research for the discrete wavelet transform can be found in ([58]–[60]) and for Gabor transform can be found in (e.g., [61]–[64]).

Furthermore, for computing the local statistics, research has been conducted to accelerate the computation of local statistics in images, but to a lesser extent in comparison with transforms. Techniques like *integral images*, originally developed in the context of computer graphics [65], is a popular approach for computing block-based sums of any two-dimensional matrix of values (e.g., a matrix of pixels or coefficients). The integral image, also known as the *summed area table*, computed a table which has the same dimensions as the input matrix; it stores the sum of all matrix values above and to the left of the current position. Thereafter, the sum of values within any block of the matrix can be rapidly computed via addition/subtraction of three values in the table. A similar technique can be used to compute higher-order moments such as the variance, skewness, and kurtosis (see, e.g., [66], [67]).

Along with research for improving computational performance for the above two common operations for IQA algorithms, there has been research on techniques for accelerating specific IQA algorithms. For example, in [68], Gordon *et al.* investigated the acceleration of PSNR by using general-purpose-GPU (GPGPU) implementations in both CUDA and OpenGL. An investigation on how the application and system performance gets affected by utilizing GPGPU acceleration of PSNR in a model-based coding application (the primary bottleneck in model-based coding stems from the optimization procedure used to determine the model parameters from the input image) was made. From the analysis, they concluded that a non-GPGPU version of PSNR runs faster.

Chen and Bovik present techniques to accelerate execution of SSIM and MS-SSIM in [69]. Techniques like integral images for calculating the luminance block have been employed. They also use an integer approximation for the Gaussian weighing window for SSIM. For Fast MS-SSIM, a further algorithm level modification of skipping the contrast and structure computations at the finest scale was proposed. Implementing the calculations of the contrast and structure components via Intel SSE2 (SIMD) instructions, they achieved speedups of approximately 5 times for Fast SSIM and 14 times for Fast MS-SSIM. In addition, speedups of approximately 17 times for Fast SSIM and 50 times for Fast MS-SSIM were reported employing parallelization via a multithreaded implementation. In [70], Okarma and

Mazurek presented GPGPU techniques for accelerating SSIM, MS-SSIM, and CVQM (a video quality assessment algorithm developed previously by Okarma, which uses SSIM, MS-SSIM, and VIF to estimate quality).

In [67], Phan *et al.* conduct performance analysis to find sections of code that consume maximum of the total execution time and present techniques for accelerating the MAD algorithm targeting these sections[34]. MAD has one of the best predictive performance but is also one of the slowest IQA algorithms, it requires over 55 seconds for a 512x512 image when tested on several modern computers (Intel Core 2 and Xeon CPUs; see [67]). The results for performance analysis show that the main bottleneck for MAD is the appearance-based stage, taking about 98% of the total runtime. Within the appearance-based stage, the computing the local statistical differences accounted for most of the runtime, and computation of the log-Gabor decomposition was the other major bottleneck. Phan *et al.* proposed four techniques of acceleration: (1) Using integral images for the local statistical computations; (2) using procedure expansion and strength reduction; (3) using a GPGPU implementation of the log-Gabor decomposition; and (4) precomputation and caching of the log-Gabor filters. The first two techniques lead to a boost 17x speedup over the original MAD implementation. The latter two resulted in an approximately 47x speedup over the original MAD implementation.

Thus, we see that there are been significant research on improving performance using advance signal processing techniques. Very little or no research exists on the behavior of IQA algorithms with the computing platform and optimize/tune algorithms based on the hardware resource utilization of corresponding algorithms. This is a basis of the research objectives presented in the next section.

1.3 Research Objectives

Although these studies have successfully yielded more efficient versions of their respective algorithms, several larger questions remain unanswered. We will present a performance analysis for 5 matured IQA algorithms VSNR, MAD, MS-SSIM, BLIINDS and VIF. This research is focused to answer the following questions and also present a framework for the mentioned 5 algorithms to further improve performance.

1.3.1 To what extent are the bottlenecks attributable to computational complexity vs. limitations in memory bandwidth?

As mentioned previously, there are two components to computational complexity with IQA algorithms. (1)Local frequency-based decompositions of the input image(s); and (2) local statistical computations on the frequency coefficients. Along with the computational complexity of the algorithms, Image processing inherently requires a higher memory bandwidth. Thus, we conduct microarchitectural analysis to investigate, how the algorithms behave on a general computing platform and study the memory requirements and bottlenecks.

1.3.2 Are there generic implementation techniques or microarchitectural modifications that can be used to accelerate all or at least several IQA algorithms?

We wish to compare and contrast the 5 IQA algorithms based on their performance profiling. This analysis can be would be a foundation to propose generic software optimization techniques, as well as hardware specific techniques to improve computational performance.

The research will provide important insights for (1) designing new IQA algorithms, which are likely to draw on multiple approaches used in several existing IQA algorithms; (2) efficiently implementing multiple IQA algorithms on a given hardware platform; (3) efficiently applying multiple IQA algorithms to specific applications; and (4) selecting and/or designing specific hardware which can efficiently execute multiple IQA algorithms.

1.4 Outline

The outline of the thesis is as follows:

- Chapter 2 provides a discussion on performance analysis and methodology along with a description of the profiling tool we use for the experiment: Intel Vtune Amplifier XE.
- Chapters 3, 4, 5, 6, 7 give the summary of each specific IQA, followed by results and discussion on hotspot analysis and microarchitectural analysis for VSNR, BLIINDS, MS-SSIM, VIF and MAD respectively.
- Chapter 8 discusses the results and answers the questions raised in section 1.3, and provides recommendations to improve performance. We also propose a hardware IQA engine framework which can be used to develop custom hardware for individual algorithms and a generic IQA engine.

CHAPTER II

ANALYSIS METHODOLOGY

For performance analysis and profiling we use the Intel Vtune Amplifier XE. The tool provides us with execution time of the algorithms along with execution time of the individual functions. With Vtune Amplifier XE we also conduct microarchitectural analysis, which provides details about the behavior of the code with the processor hardware.

This chapter provides an overview of the performance profiling tool, Intel VTune Amplifier XE [71]. Section 2.1 discusses the analysis procedure and methodlogy along with the specification of the hardware platform used. Section 2.2 provides a brief discussion about the tool in general. In section 2.3, we discuss hotspot analysis with reference to Intel VTune amplifier XE and discuss the way the Intel VTune Amplifier XE samples data to provide profiling results. Finally in section 2.4, we discuss microarchitectural analysis, specifically general exploration analysis type, the metrics used for analysis, and the hardware based event sampling utilized to provide results for microarchitectural analysis.

2.1 Experimental setup

To begin to address the goals described in the previous chapter, we define an experimental framework for performance analysis designed to examine, compare, and contrast the performances of five popular IQA algorithms (VSNR [33], MAD [34], MS-SSIM [29], BLIINDS-II [39] and VIF [30]) on a typical desktop computing platform. To provide a common codebase, we implemented each of the algorithms in C++ based on the original MATLAB code provided by the authors of their respective algorithms. An initial code-level profiling was performed using Intel Vtune to identify and correct obvious inefficiencies in the baseline implementations. Next, we execute multiple trials of each of the five algorithms on 7 different images varying in image content. We use 3 different distortion types AWGN, Blurring, and Jpeg compression with 2 levels of distortion, making a total of 42 Images from the CSIQ image database [34]. The details for the subject images are highlighted in Table1.

Number of Images varying in Content	7
Types of Distortions	3 (AWGN, Blurring, Jpeg compression)
Levels of Distortions	2 (Level 1 and Level 5)
Total subject Images	42
Image size	512x512
Frames (loops for the Algorithm)	30
Hotspot Analysis sample time [72]	1ms
Microarchitecture code name	Sandy Bridge [73]
Microarchitectural Analysis Type	General Exploration

Table 1: Details of subject Images for performance profiling and microarchitectural analysis.

2.2 Performance profiling methodology

After the experimental setup is fixed, with define the performance analysis flow and methodology. This section discusses an overview of performance profiling flow and the sequential steps for the analysis. First, we identify sections of program that should be targeted for improving the computation performance [74]. Such target sections of the program are called “hotspots” [75]. The process of identifying hotspots and improving performance is termed “tuning” [76]. Once the top hotspot functions are known, we conduct microarchitecural analysis to observe the interaction of the algorithm with the processor and other microarchitectural sub-systems. Microarchitectural analysis is used to inspect the processor resources used by the algorithm, related hardware bottlenecks and its reason. We use the Intel VTune Amplifier XE performance profiling tool to conduct our experiment and use the top down process suggested in the Intel optimization manual [77] for analysis. The suggested top down process in discussed below.

2.2.1: System Tuning

The first step for optimization/ tuning of an application is System Tuning [76]. Deciding the hardware that best suits the given algorithm is called system tuning. For example, if your program is multithreaded, a multicore multi-threaded processor is advisable. We chose the hardware specifications to match the modern general purpose PC. We use the 2nd generation Intel core i5-2430M processor [78-79] clocked at 2.4GHz and a system memory (RAM) of 4GB .The microarchitecture is Sandy Bridge. Further details about the caches and memory hierarchy can be found in Table 2

Processor	Intel® Core™ i5-2430M
Frequency	2.4 GHz
Microarchitecture	Sandy Bridge
System Memory (RAM)	4 GB
L1 Instruction Cache	32KB per core
L1 Data Cache	32KB per core
L2 Cache (Unified Instruction and Data Cache)	256KB per core
L3 Cache (Unified Instruction and Data Cache)	3MB shared

Table 2: Hardware specifications for the performance analysis experiement.

Since, the hardware specifications are fixed; we will focus our experiment on algorithmic and microarchitectural analysis.

2.2.2: Algorithmic Analysis

Once the hardware specifications are fixed, the next step for optimization/ tuning is Algorithmic/ Hotspot analysis and tuning the algorithm based on software inefficiencies. During the prototyping phase, the software techniques employed usually aim for more readable and easier to debug programs. Such implementations are not optimized for a particular metric. The compiler does a tremendous amount of code optimization, but there are typically two situations where it cannot perform optimizations. First, when the inefficiency becomes apparent and occurs for only certain inputs that are not known at compile time, and second, when the inefficiency is at the algorithmic level. For example, for a sorting algorithm bubble sort has a computational complexity of O (n^2) while quick sort has O (log (n)) computational complexity. Thus using quick sort will have a better computational performance when compared to bubble sort. We analyze the software for these inefficiencies that are beyond the scope of the compiler. While VTune cannot identify and recommend which sorting algorithm one must use, it does profile the code to point out sections of the code in which a majority of the execution time is spent. These sections of the code are the hotspots and can be targeted to improve performance. The algorithms can also be modified and a completely different implementation methodology can be used to improve performance [77]. During the Hotspot analysis phase of the experiment we note the total execution time for the algorithms, individual hotspots with their corresponding execution time and contribution to the total execution time.

2.2.3: Microarchitectural Analysis

For algorithmic/hotspot analysis, we do not take into consideration the hardware specifics like memory utilization, instruction stalls during execution for tuning or optimizing the code. Microarchitectural analysis inspects the program as it traverses through the different subsystems of a processor and provides information to find architectural bottlenecks. This method is applied after hotspot analysis.

For microarchitectural analysis, we analyze the results to study the instruction throughput of the code as a whole and the instruction throughput of individual hotspot functions. The metric that we use is retired pipeline slots [72]. The retired pipeline slots metric is the number of instructions that exit the processor successfully, divided by the total number of clock cycles the hotspot utilized. The metric is normalized from 0 to 1. The Intel i5 2430M processor can generate and execute up to four instructions every clock cycle [78-79]. The instructions which do useful work are called retired [80]. Therefore, the retired pipeline slots metric gives us a measure of the number of instructions that exit the processor doing useful work. The ideal value for this metric is 1. However, the observed value is rarely equal to the ideal or maximum value. This can be due to several factors: some pipeline slots cannot be filled with useful work, either because the fetch and decode units of the processor could not fetch or decode instructions in time or because the execution units were overwhelmed and cannot accept more operations of a certain kind. This is called a structural hazard. Even if pipeline slots contain useful work it is possible that the instructions may not retire due to bad speculation like mispredicting a branch. After considering the retired pipeline slots and determining the computing efficiency, we inspect hotspot functions for microarchitectural sub-system bottlenecks. A basic top down methodology for the complete tuning process is shown in Figure 1.

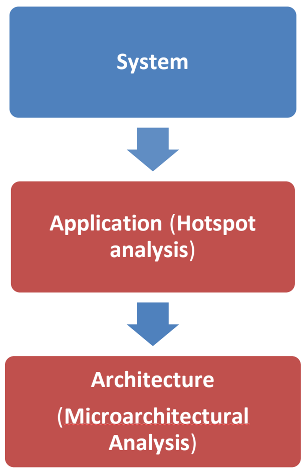


Figure 1: Top Down Tuning methodology for analysis and tuning

The goal of the research is to identify and point out sections of individual algorithms which are computationally intensive and take higher execution time.. The hotspot analysis type in the Vtune Amplifier XE gives a list of hotspot functions and corresponding relevant details. Using microarchitectural analysis, we analyze the behavior and interaction of such sections with the underlying hardware to find architectural bottlenecks. We conduct this microarchitectural analysis with the general exploration type of analysis in Vtune Amplifier XE. After both the hotspot analysis and microarchitectural analysis wemap the hotspot functionsto specific execution blocks of the respective algorithm. Advanced software techniques along with microarchitectural consideration can be used to further improve computational performance.

2.3 Intel Vtune Amplifier XE Overview

The Intel VTune Amplifier XE is a performance profiling tool which provides information about the application's execution on software basis as well as its interaction with different Intel hardware platforms [71]. Using the Intel VTune Amplifier XE, we can determine and locate hotspots, sections of code that have low instruction throughput and do not exploit hardware resources and related bottlenecks. For multithreaded applications the tool can also provide information about thread activity and transitions. In the following sections we describe a brief overview of hotspot analysis followed by microarchitectural analysis. Top features of the tool are highlighted in Table 3. More details about VTune Amplifier XE can be found at [71, 72, 76, and 77].

Hotspot Analysis	Located most time consuming sections of the code in a sorted list of functions.
Jump to source code	Double click on the hotspot function in the results takes us directly to the source code.
Lock and waits Analysis	We can analyze locks and waits for parallel applications. These are the major cause for slow multithreaded applications.
Low overhead/High resolution Hardware profiling	Lightweight hotspot analysis, using performance monitoring units (PMU).
Predefined Hardware Events and thresholds for bottlenecks	Advanced profiles like memory bandwidth analysis, memory access and branch mispredictions find tuning opportunities. Predefined thresholds for identifying bottlenecks.

Table 3: Features of the performance profiling tool Intel Vtune XE Amplifier [71].

2.3.1: Hotspot Analysis:

In this section, we first discuss hotspot analysis w.r.t Intel VTune Amplifier XE and then describe the associated result windows: summary window, bottom-up pane, top-down pane and the call stack pane along with sampling of data for hotspot analysis. As discussed previously, hotspots are the most time consuming units in the program. A generic methodology for hotspot analysis and tuning as described in [81] is shown below.

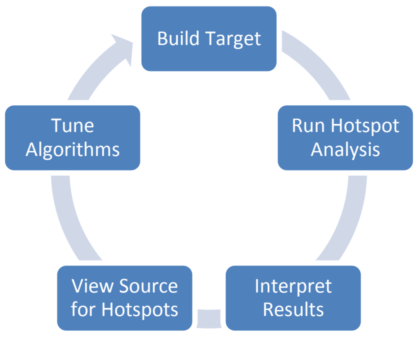
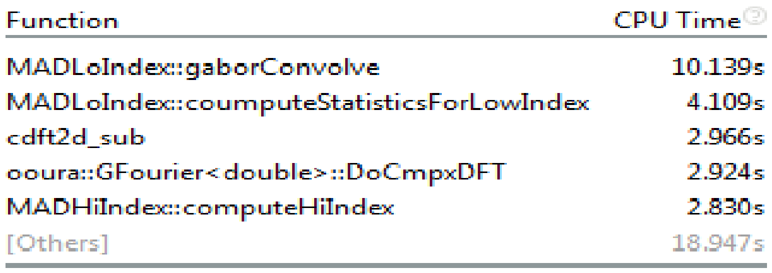


Figure 2: Hotspot analysis methodology and the tuning process.

The process of tuning takes multiple iteration. It starts with building and compiling the application, run hotspot analysis and note the hotspots and corresponding execution time. Analyze the results and the calling functions and the source code for the hotspot function, make optimizations and build/compile the application again. This process is repeated till the target for optimization is achieved.

When we run the hotspot analysis the Intel VTune Amplifier XE finalizes the results and opens up the hotspot view which has the summary of the hotspots and also has options to open the bottom-up, call stack pane and top-down tree windows. A screenshot for the summary window is shown below.

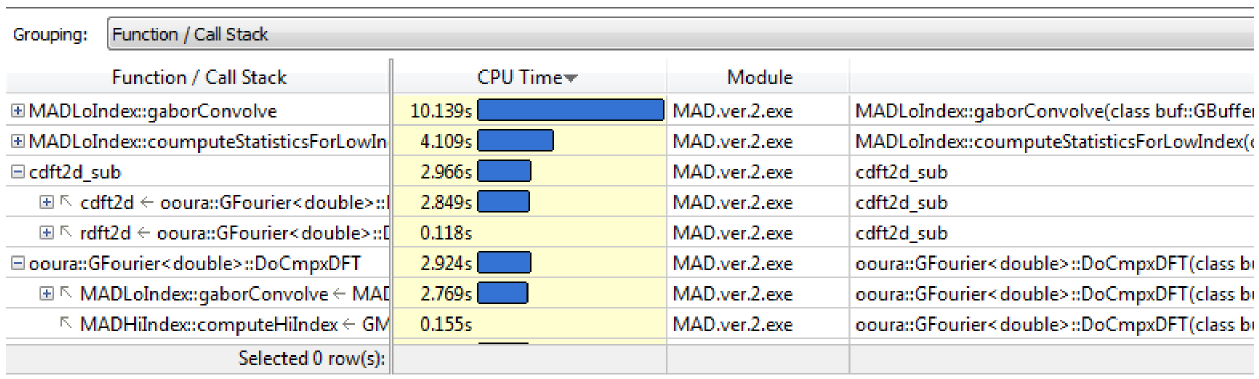


Function	CPU Time
MADLoIndex::gaborConvolve	10.139s
MADLoIndex::coumputeStatisticsForLowIndex	4.109s
cdft2d_sub	2.966s
ooura::GFourier<double>::DoCmpxDFT	2.924s
MADHiIndex::computeHiIndex	2.830s
[Others]	18.947s

Figure 3: Summary window hotspot analysis with top hotspot functions and corresponding execution time.

The summary window shows the total execution time of the program along with individual execution time for top hotspot functions. As seen from the figure, the top hotspot function is gaborConvolve which takes around 10.139s. The hotspot functions are arranged in a descending order in terms of their execution time. We can also navigate to bottom-up analysis or the top-down tree from the summary window. The bottom-up analysis next, then the top down tree and call stack pane.

The bottom-up pane shows the top hotspot as the first entry by default.It presents analysis specific data starting from the function up to their calling functions and hence bottom-up analysis [72]. If multiple functions call the hotspot function, the calling sequence for all the instances is displayed. Refer to the figure 4 below. We see that the function cdft_2d_sub is called by cdft2d as well as rdft2d.

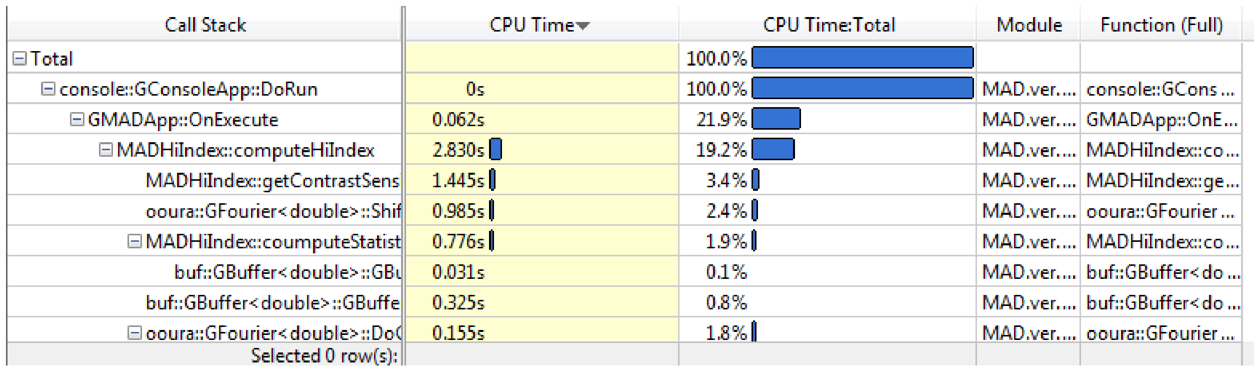


Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Module	
⊞ MADLoIndex::gaborConvolve	10.139s	MAD.ver.2.exe	MADLoIndex::gaborConvolve(class buf::GBuffe
⊞ MADLoIndex::coumputeStatisticsForLowIn	4.109s	MAD.ver.2.exe	MADLoIndex::coumputeStatisticsForLowIndex(
⊟ cdft2d_sub	2.966s	MAD.ver.2.exe	cdft2d_sub
⊞ ↖ cdft2d ← ooura::GFourier<double>::[	2.849s	MAD.ver.2.exe	cdft2d_sub
⊞ ↖ rdft2d ← ooura::GFourier<double>::D	0.118s	MAD.ver.2.exe	cdft2d_sub
⊟ ooura::GFourier<double>::DoCmpxDFT	2.924s	MAD.ver.2.exe	ooura::GFourier<double>::DoCmpxDFT(class b
⊞ ↖ MADLoIndex::gaborConvolve ← MAD	2.769s	MAD.ver.2.exe	ooura::GFourier<double>::DoCmpxDFT(class b
↖ MADHiIndex::computeHiIndex ← GM	0.155s	MAD.ver.2.exe	ooura::GFourier<double>::DoCmpxDFT(class b
Selected 0 row(s):			

Figure 4: Bottom up pane, Hotspot analysis with a list of hotspot functions in a sorted order along with their complete calling sequence.

The top-down tree pane presents analysis specific data starting from the application root (usually main () function) down to function callees [71]. It is used to explore the call sequence flow of the application and analyze the time spent in each function or program section and on its callees. The top down tree can be used to find the critical path to the hotspot function. Figure 5 shows the top-down tree pane.



Call Stack	CPU Time	CPU Time:Total	Module	Function (Full)
⊟ Total		100.0%		
⊟ console::GConsoleApp::DoRun	0s	100.0%	MAD.ver....	console::GCons ...
⊟ GMADApp::OnExecute	0.062s	21.9%	MAD.ver....	GMADApp::OnE...
⊟ MADHiIndex::computeHiIndex	2.830s	19.2%	MAD.ver....	MADHiIndex::co...
MADHiIndex::getContrastSens	1.445s	3.4%	MAD.ver....	MADHiIndex::ge...
ooura::GFourier<double>::Shif	0.985s	2.4%	MAD.ver....	ooura::GFourier ...
⊟ MADHiIndex::coumputeStatist	0.776s	1.9%	MAD.ver....	MADHiIndex::co...
buf::GBuffer<double>::GBu	0.031s	0.1%	MAD.ver....	buf::GBuffer<do ...
buf::GBuffer<double>::GBuffe	0.325s	0.8%	MAD.ver....	buf::GBuffer<do ...
⊟ ooura::GFourier<double>::Do(	0.155s	1.8%	MAD.ver....	ooura::GFourier ...
Selected 0 row(s):				

Figure 5 Top Down window for Hotspot analysis. It displays the traversal of the application along with the execution time of individual sections.

As seen from figure 5, the application starts with the DoRun function and followed by the OnExecute function and so on down to the end of execution. Thus, the top down tree presents the functions in a sequence with which they traverse during the run time of the application.

2.2.3: Microarchitectural Analysis

In this section, we discuss microarchitectural analysis, specifically general exploration type microarchitectural analysis,associated hardware metrics and discuss event based sampling for microarchitectural analysis. The microarchitectural analysis provides an insight on how the application interacts with the CPU and consequently provide information to find architectural bottlenecks. The Intel VTune Amplifier XE uses the processors performance monitoring units (PMU) to sample hardware events [72]. First, we discuss an abstract view to the processor design and categorize the instructions entering a processor into 4 categories. 1: Retired 2: cancelled 3: Front end bound and 4: back end bound. This categorization in based on Intel Vtune optimization guide [77].

Processors are divided into two sections: the front end which fetches instructions and decodes them into micro-operations (Uops) to feed them to the respective execution units. For the i5-2430M machine the front end generates upto four micro-operations maximum [78-79]. These micro operations are then fed to the respective execution unit which is called the back end of the processor. If the front end cannot fetch and decode then the instruction is called to be "Frontend bound" and if the back end is not able to accept operations then the instructions are said to be "Backend bound". If the micro-operations exit the pipeline with doing useful work then these operations are said to be "retired" or else these operations are said to be "cancelled" [77]. The operations can get cancelled due to many reasons like branch misprediction or any other speculation done by the processor. The Intel®Vtune™ XE Amplifier classifies performance issues based on the pipeline slot the micro-operations can be in. The four categories and the corresponding hierarchy for the categories are shown in the figure 6 below.

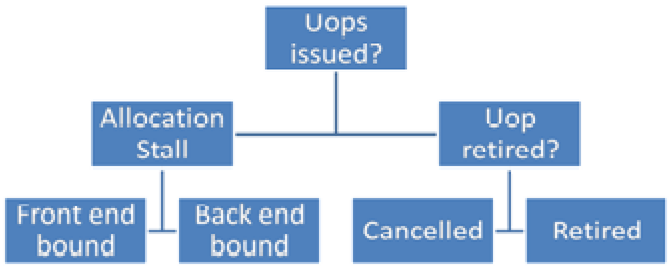


Figure 6: Categories of Pipeline slots an instruction entering the processor can be in.

For microacrhitecturalanalysis, we first narrow down the bottlenecks into the above four categories and then highlight specific microarchitectural sub-systems which are bottlenecks for individual hotspot functions.

The VTune Amplifier XE provides a wide variety of microarchitectural analysis like general Exploarion, Bandwidth analysis, Cycles and Uops etc. More details about the different analysis types within microarchitectural analysis can be found in the VTune Amplifier XE help manual [71]. We use the general exploration type of analysis. It is an analysis which has a set of hardware metrics predefined. All the metrics for the analysis in general Exploration are highlighted in table 4.

Pipeline Slot Category	**Specific Hardware Metric**
Frontend Bound	Icache Misses
Frontend Bound	ITLB Overhead
Frontend Bound	DSB to MITE switch overhead
Backend Bound	LLC load miss (memory latency)
Backend Bound	LLC miss (memory latency)
Backend Bound	LLC hit (memory latency)
Backend Bound	DTLB overhead (memory latency)
Backend Bound	Contested Accesses (memory latency)
Backend Bound	Data Sharing (memory latency)
Backend Bound	L1 replacement (memory replacement)
Backend Bound	L2 replacement (memory replacement)
Backend Bound	LLC replacement(memory replacement)
Backend Bound	Loads block (memory reissue)
Backend Bound	Split Loads(memory reissue)
Backend Bound	Split Stores(memory reissue)
Backend Bound	4k aliasing(memory reissue)
Backend Bound	DIV Active (Core bound)
Backend Bound	Merge flags(Core bound)
Backend Bound	Slow LEA overhead (Core bound)
Retired Pipeline slot	Assists
Cancelled Pipeline slot	Branch Mispredict
Cancelled Pipeline slot	Cancelled pipeline slots

Table 4: General Exploration Hardware metrics with Pipeline slot categories and sub categories.

More details about the specific metrics, the related hardware issues and designs are discussed in the microarchitectural analysis section for individual algorithms when they are found to be bottlenecks.

CHAPTER III

VSNR

This chapter provides a brief description of the VSNR algorithm in Section 3.1. Section 3.2 provides an insight into the implementation details of the code and specific methods used. The performance profiling and hotspot analysis is discussed in Section 3.3 and finally, microarchitectural analysis with corresponding hardware bottlenecks mapped to the specific algorithmic block presented in Section 3.4.

3.1. Overview

The VSNR algorithm [33] provides an estimation of the visual perception of distortions in natural images. The strategy for VSNR is to compute the root-meansquared (RMS) contrast by calculating the contrast thresholds for detection of distortions, the perceived contrast of the distortions and the extent to which the distortion affects global precedence, to degrade the image's structure. The block diagram for the algorithm is shown in figure 7.

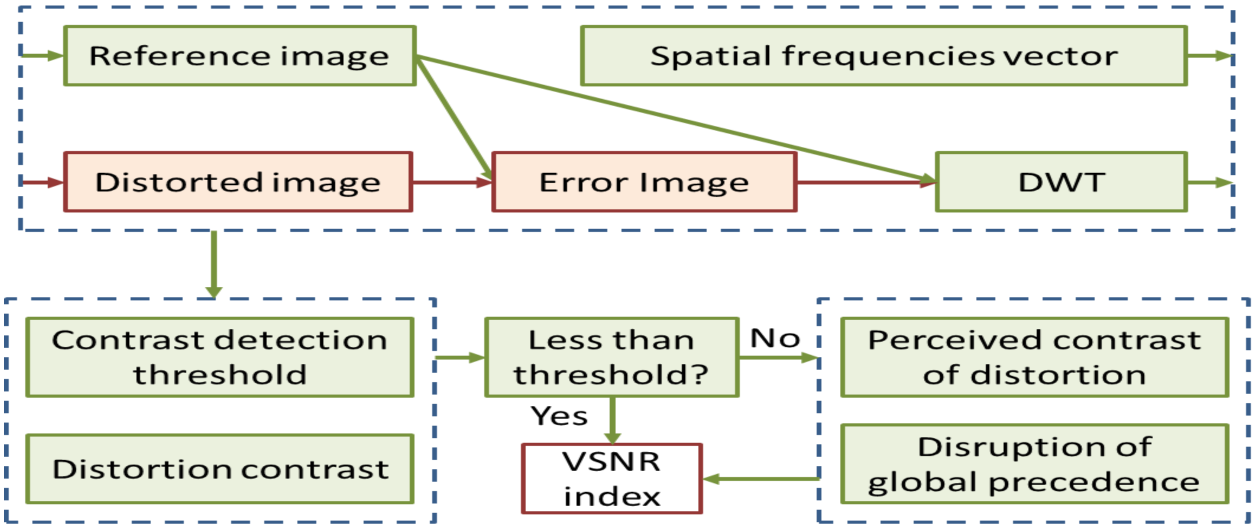

Reference image
Spatial frequencies vector
Distorted image
Error Image
DWT
Contrast detection threshold
Distortion contrast
Less than threshold?
No
Yes
VSNR index
Perceived contrast of distortion
Disruption of global precedence

Figure 7: Block Diagram for VSNR.

An efficient metric for quantifying the visual fidelity of natural images is the visual signal-to-noise ratio (VSNR). The metric is based on near-threshold and suprathreshold properties of human vision. The algorithm is divided into 2 stages. During the first stage, contrast thresholds for detection of distortions in the presence of natural images are calculated. These thresholds are computed via wavelet-based models of visual masking and visual summation. The thresholds determine if the distortions are directly visible. If the distortions are below the threshold of detection then the distortions are not visible and the distorted image is considered to be of perfect visual quality with VSNR = infinity and the algorithm halts without conducting any further estimation of visual quality. But if the distortions cross the threshold; further processing is done which operates based on the low-level visual property of perceived contrast, and the mid-level visual property of global precedence. These two properties are then modeled as Euclidean distances in distortion-contrast space of multiscale wavelet decomposition, and VSNR is computed based on sum of these Euclidean distances.

3.2 Implementation Details and specifics

The original C++ code of VSNR is obtained from author's website [82] and is optimized to remove obvious inefficiencies and have less execution time. As shown in Figure 11, the reference image and distorted image are loaded and stored in 2-D float arrays. The DWT decomposition step is implemented with five-level two-dimension wavelet transform based on lifting scheme using the default Cohen-Daubechies-Feauveau 9/7 wavelet [83]. In the second step, in order to have the contrast, we need to calculate the average luminance of the image. In this function, the original C++ code uses 512 × 512 power operation, multiplication and addition. We modify this part using look up table technique to have a faster implementation which uses these operations only 256 times. In step 3, the global-precedence-preserving contrast signal-to-noise ratio, is computed using bisection search with index of visibility in the range of [0, 1].

3.3 Hotspot Analysis

As described in section 2.1, the process of finding critical sections of the program which consume higher execution time is called hotspot analysis. For our experiment, we find the hotspots for all the algorithms based on their contribution to the total execution time.

 For the hotspot analysis for VNSR, we first find the total execution time for the algorithm. The average execution time for VSNR is 0.700s.After the finding the average execution time we observe the analysis results for hotspot functions to find the most time consuming sections of the code. The results for hotspot analysis are shown in table 5.

Function/block	**Average individual execution time(S)**	**% of total execution**
All	0.724	100.00%
1D DWT-Columns	0.236	32.61%
Variance	0.119	16.45%
1D DWT-Rows	0.099	13.68%
Others	0.270	37.28%

Table 5: Average execution time for the Hotspot function, also expressed as percentage of total execution time.

We find that the top hotspot or the most time consuming block/function for VSNR is 1D DWT-Columns. It takes about 32% of the total execution time an average of 236ms. As mentioned previously, the 1D DWT-Columns function computes a 1-D Discrete Wavelet Transform (DWT) across the columns of the original (distortion less) and the subject (Distorted image).

The next hotspot function/block is the variance block. It takes about 17% of the total execution time and 119 ms at an average. The variance function/block calculates variance for contrast and threshold detection. It is used for the image statistics.

Following the 1D DWT-Columns block the next hotspot function/block is1D DWT-Rows. The 1D DWT-Rows block consumes about 14% of the total execution time and 99 ms at an average. The 1D DWT-Rows function similar to 1D DWT-Columns calculates the DWT coefficients but across the rows on the reference as well as the subject image. Finally, all the other functions add up to 38% of the total execution time and an average of 270ms.The results are plotted in figure 8, figure 9, and figure 10.

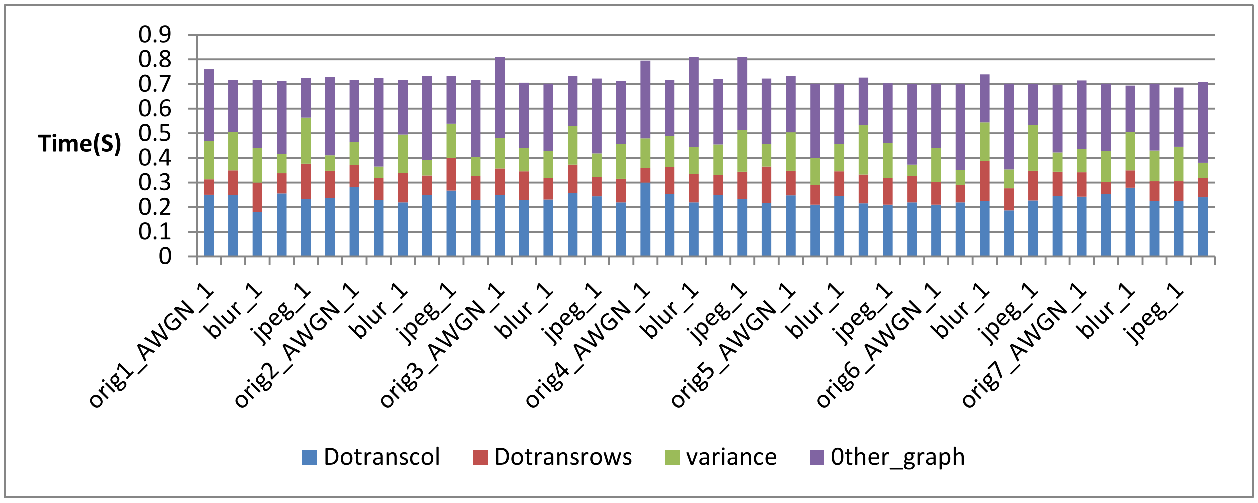


Figure 8: Individual execution time for all the hotspot functions and all the test images stacked to add up to the total execution time for a loop of 30 frames.

The figure shows that there is not a very high variation in the across the different test images with a few exceptions. To gather further insight into the variations, we plot the average execution of the different subject images across variation in distortions, figure 9 and across variation in image content, figure 10.

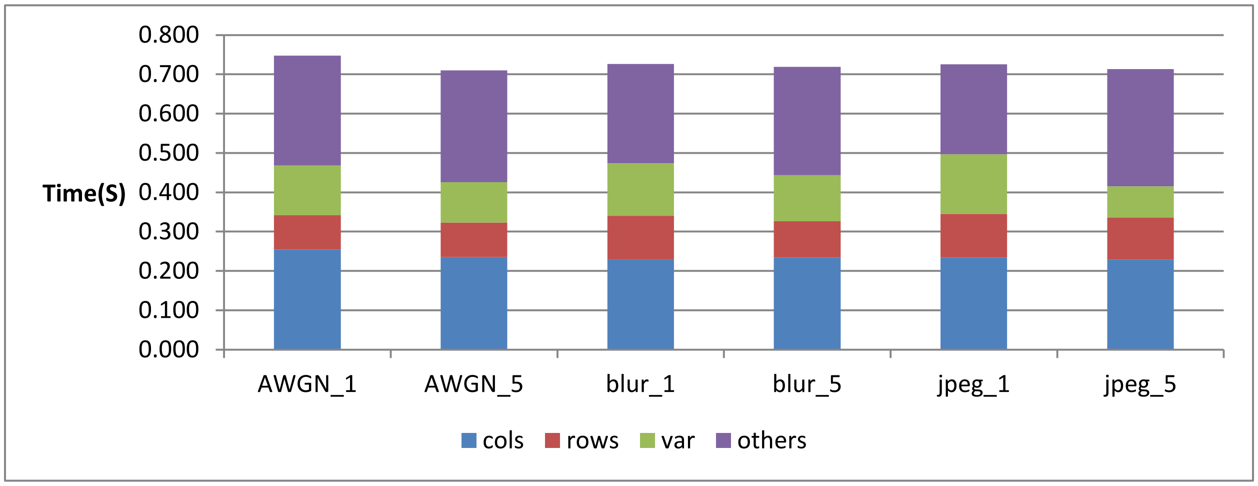


Figure 9: Average execution time of the hotspot functions varied across the 6 different distortions.

The plot shows the execution time of different images averaged across the different distortion levels. As seen from the graph, we find that there is no significant variation in the total execution time of the program as well across the different hotspot functions.

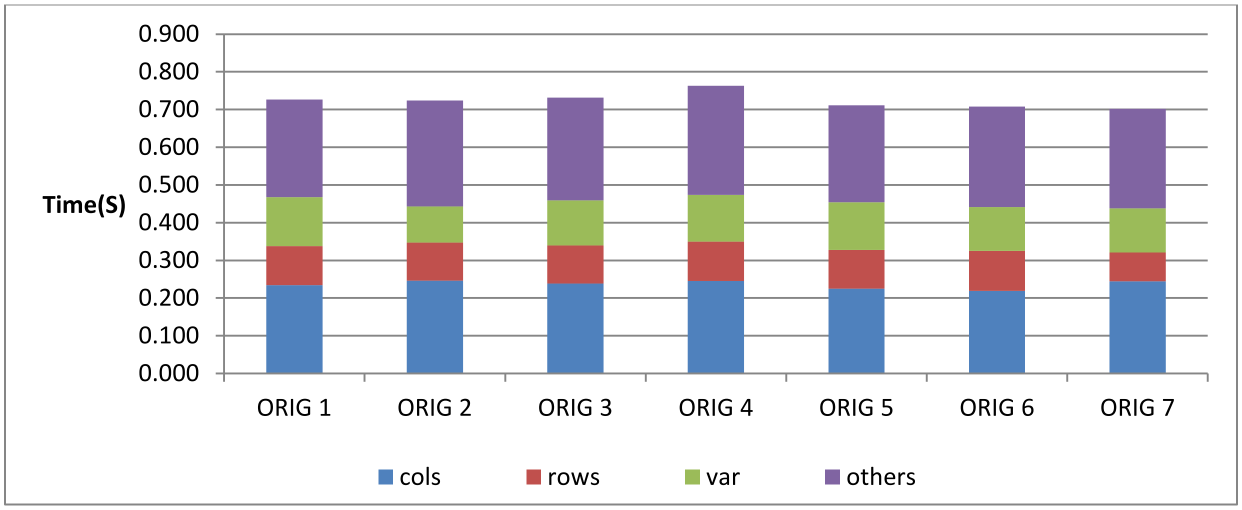


Figure 10: shows the average execution time of all the hotspot functions varied across 7 different image contents.

As seen from figure 10, we find that similar to variation across different distortions there is less variation in execution time of the complete program as well as individual hotspot functions with an except for orig 4 images. The orig 4 images have a higher execution time. Because the hotspot functions do not change ranks, the sequence/priority of the functions remains the same for optimization. As discussed in the methodology section, after the hotspot analysis we conduct microarchitectural analysis to analyze the behavior of the code with the underlying hardware. The results are discussed in the next section.

3.4 Microarchitectural Analysis:

After the hotspot analysis, we conduct microarchitectural analysis to inspect the utilization of architectural resources. The goal is to identify what microarchitectural resources the hotspot functions need, and to find microarchitectural bottlenecks. First, we analyze the results to study the instruction throughput of the code as a whole and the individual hotspot functions. The metric that we use is retired pipeline slots. We observe that there are bottlenecks related to the memory sub-systems and thus, briefly discuss the memory hierarchy design of the processor to create a platform for our discussions on specific bottlenecks, than associate these bottlenecks with the algorithmic blocks.

The main memory or the RAM is fairly slow and takes about 100 clock cycles to deliver operands to the processor. So, to deliver operands to the processor every clock cycle a fast and small cache memory is placed between the processor and system memory. The cache memory closest to the processor is Level 1 cache or L1 cache. It is the fastest and smallest cache in the hierarchy. The next level of the cache memory is bigger and slower called the L2 cache. Usually, the L1 and L2 cache have 2 categories, First, Instruction cache where the instructions of the code are stored. The instruction cache aids to deliver the instructions to the processor every clock cycle and second, Data cache where operands and resultants for the instruction are stored. The level 3 or L3 is the last level cache which is the biggest and slowest. It is called the Last Level cache (LLC). Now, after knowing about the memory hierarchy we associate these memory sub-systems to the hotspot functions/blocks. More details about cache memories and memory hierarchy can be found in [80].

The results for the microarchitectural analysis, the hardware bottlenecks and associated blocks are shown in table 6

Results Summary:

Function/Block	**Image Type**	**Pipeline Slot Category**	**Backend Slot Category**	**Retired Pipeline slot**	**Hardware Bottlenecks**
1D DWT-Columns	All	Backend	Memory	0.100	L1,L2 replacement and LLC hits
Variance	All	None	None	0.383	None
1D DWT-Rows	All except jpeg-5	Backend	Memory	0.350	4k aliasing
1D DWT-Rows	Jpeg-5	Backend	Memory	0.221	Machine clears

Table 6: Microarchitectural analysis for VSNR with hotspot functions and associated bottlenecks.

L1D- Level 1 Data Cache, L2D- Level 2 Data Cache, LLC – Last level Cache.

Observing the hotspot analysis in the previous section, we find that 1D DWT-Columns block/function is the top hotspot. As mentioned, it is employed to do take a DWT across the columns of the subject image and the reference image. Conducting microarchitectural analysis, we find that the retire pipeline slot metric for the block is around 0.1 for all the images. Typically, if the metric is above 0.6, the section of the program is considered to be efficient. [76]. Thus, 1D DWT-Columns is highly inefficient. Upon further investigation we find that 1D DWT-Columns is backend bound memory bound. The major penalty is due to LLC hits which mean the program has to access the last level cache frequently. The LLC takes about 26-31 clock cycles [78-79] for a single memory access, which is very costly. Consequently, 1D DWT-Columns is the top hotspot. Along with LLC accesses as a bottleneck, we find that there are penalties due to data replacement in L1D and L2D caches. Since accessing the LLC requires high number of clock cycles, optimization should be made so that the processor doesn't have to access the LLC frequently. This can be done by reducing the L1D and L2D replacement and the L1D and L2D misses. We discuss the techniques to improve the cache performance in the later part of this section.

Followed by 1D DWT-Columns the next hotspot function is the 1D DWT-Rows which is similar to 1D DWT-Columns except that it is used to calculate DWT coefficients for rows. We investigate the retired pipeline slot metric to determine the efficiency of the block. We find that the retired pipeline slot metric is 0.35 which means that the function is inefficient. After investigating the architectural reasons we find that the block is backend memory bound and specifically there are memory reissues because of 4k aliasing.

To understand 4k aliasing, we need to look at how modern processors handle loads and stores. Load instructions refer to the transfer of data from memory to a CPU register. Stores do the opposite, i.e. write to a memory location from a CPU register [80]. A typical hardware optimization is to give priority to loads over stores, because loads are typically followed by arithmetic-type instructions that use the value that was loaded, and if a load is delayed, it stalls the subsequent use. Out-of-order execution is another common feature of modern processors, where instructions are executed out of the program order to make efficient use of hardware resources, but are retired in the correct program order [80]. Both the optimizations described above can give rise to a situation where a store instruction writes to the same memory location that subsequent loads read from, but the load instruction is actually processed before the store. This would yield a stale or incorrect value to be loaded from that memory location. This requires checking the memory address that every load instruction gets its data from, with the target address for all pending store instructions. However, the pending stores might not have their effective addresses generated yet, and the load instruction could be delayed until at least all the store instructions have addresses that can be compared with the current load. This is not usually done, and the load is allowed to proceed. It is important to understand why modern architectures proceed with a speculative load in such a situation. As mentioned above, there could be instructions that depend on the load instruction. There are often several instructions that depend on a load, and delaying the load limits the options for the instruction scheduler for out-of-order processing. Furthermore, a load might miss in the first level cache, and might require several clock cycles to be serviced from the lower levels of the memory hierarchy (as seen above with the LLC accesses). Thus, it is important to issue the load without waiting for all the subsequent stores to have their addresses ready.

As this operation could require several comparisons, instead of comparing the entire 32-bit or 64-bit address, only the last 12 bits are typically compared. If these addresses are 4K bytes (2^{12}) or multiple of 4096 bytes apart then a false hazard is detected. This is called 4k aliasing and the load has to be reissued. Consequently, because of 4k aliasing the processor has to perform loads again and the throughput goes down.

A possible solution to this problem from the hardware side is to check all 32/64 bits of the address instead of just 12. On the other hand, if most applications do not show the 4k aliasing to be a bottleneck, hardware designers are likely to check only 12 bits to save on hardware complexity, chip space, and power consumption.

There are some interesting solutions on the software side. One can investigate why there are so many 4k aliasing problems for this algorithm, and align data to 32 Bytes. Change offsets between input and output buffers if possible. Use 16-Byte memory accesses on memory which is not 32-Byte aligned. More details on solving 4k aliasing can be found in Intel optimization manual [77].

Also, as seen for jpeg images there are micro operations that get cancelled due to machine clears. Machine clears are typically caused by memory ordering violations or loads to illegal memory range. Based on the previous finding of 4k aliasing, we can conclude that the machine clears here are due to memory ordering violations, and are in fact caused by false positives from the 12-bit comparisons. Load and store instructions do not exit the processor until it is validated that there is no memory ordering violation in the actual order of execution that was used. Stores do not get sent to memory until they are ready to be retired. If the processor detects a memory ordering violation, it discards all unretired operations (including the offending memory operation) and restarts execution at the oldest unretired instruction. This is because the load might be speculative as described above, and all the instructions that depend on that load are also marked as speculative. After the comparison with the stores, if there is not violation detected, then the instructions are deemed good and are retired. If a violation is detected (either because there was a violation or because of the false sharing), then the load has to be reissued, and all the dependent instructions are also cleared and reissued.

From the hotspot analysis we notice that even though the DWT functions do exactly the same computation 1D DWT-Columns calculates the DWT coefficients across the columns and while 1D DWT-Rows across the rows of the reference and the subject image, there is a significant difference in the execution time for both the functions. To investigate the reason behind this we describe the design of the caches and how the data is brought and replaced in a cache.

The caches are design on the principle of locality of reference. It is based on the fact that the programs usually access the same or nearby memory locations repeatedly and frequently. Usage of same memory location repeatedly falls is called temporal locality while access of nearby memory locations is spatial locality [80]. An example for temporal locality would be instructions executed in a loop. These instructions are executed again and again and the processor fetches the instructions from the same memory location. For spatial locality, arrays serve as a great example. An image is basically a 2 dimensional array of pixels. Usually if a pixel is accessed, the next pixel or a nearby pixel is also accessed. Based on the concept of locality, the cache controller brings a complete block of data and not just the operand that is required by the processor at that instant. The cache block for the Sandy bridge microarchitecture is 64bytes [78-79]. Thus, we can say the caches operate at a granularity of 64bytes. All the blocks brought into and removed from the cache are 64 bytes in size.

Now, we analyze how the image data is brought into the cache and correlate it with difference in execution time for calculating the DWT coefficients across columns and rows of an image. Figure 11 shows how image data is mapped to memory locations in an L1 cache.

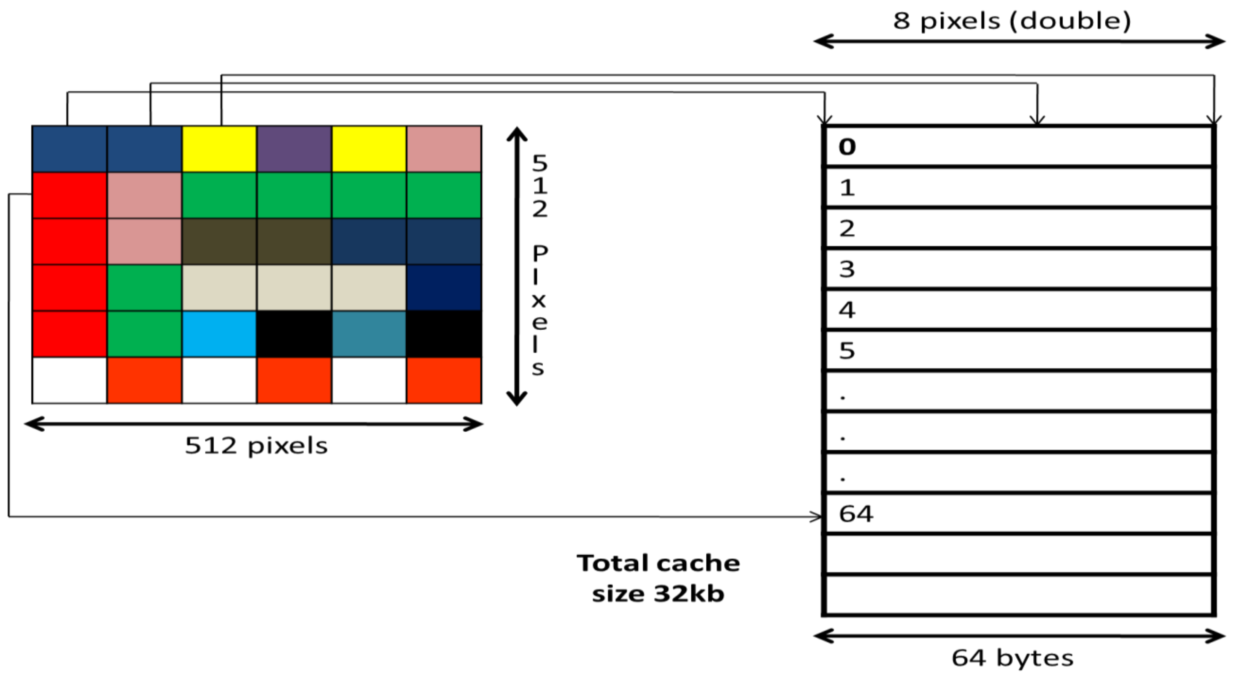


Figure 11: L1 cache with a cache line of 64bytes along with mapping of image data to memory locations

As seen from the figure the image pixels are brought in the cache row wise. A section of the image row is brought into the cache, when a block of data is stored in the cache. Specifically 8 pixels with double data type are brought into the cache. The calculations are shown below:

1 block of cache = 64bytes (cache line width)/ 8 bytes (double data type size) = 8 pixels.

Thus, for getting a complete row of pixels in an image we require 64 blocks. Thus, a new column pixel is resides in the cache after every 64 cache line blocks. Therefore, in terms of memory every column pixel resides after 64x64 bytes= 4096 bytes or 4kb. Hence for a total cache size of 32k, if the cache is assumed to be only filled with image pixels, there can be only 8 column pixels. Whereas the number of row pixels that can reside into the cache is 4088. Thus, it is seen that the number of row pixels brought into the cache is much higher than the number of column pixels brought into the cache. Consequently, the hit rate for row pixels is higher, the processor has to go to the next level of cache less frequently to fetch pixels as more pixels can reside and the execution time is less.

Also, the other way to look at this problem is that for every block brought inside the cache all memory locations for that particular block are utilized by the 1D DWT-Rows functions (spatial locality) but for every block brought inside the cache there can be only one column pixel. Consequently, more blocks have to be brought and more current blocks residing inside the cache have to be replaced to make room for newer blocks. For this reason, the 1D DWT-Columns function has a higher L1D and L2D replacement penalty and has higher execution time.

A basic solution to cache memory replacement and cache misses problem is to use a chipset with larger cache size. This will lead more data to fit in the cache and increase the hit rate to improve performance. Another software technique to improve cache performance is called Loop tiling or loop Blocking. Loop tiling or loop blocking. As the title suggests, loop tiling or blocking is a mechanism in which the loops are broken into smaller chunks, to ensure that the data used by the loop fits in the cache and does not cause a miss. When we partition loop iteration (large array into smaller blocks), the accessed array elements fit into cache size, enhancing cache reuse and eliminating cache size requirements. Loop blocking allows reuse of the arrays by transforming the loops such that the transformed loops manipulate array strips that fit into the cache. In effect, a blocked loop uses array elements in sections that are optimally sized to fit in the cache. An example for loop blocking and how it improves cache utilization is presented in section 8.1.

Followed by the wavelet block the next hotspot function is variance. The retired pipeline slot metric is averages to 0.4 which means that the function is executed inefficiently but has a throughput better than the DWT functions. From the microarchitectural analysis we find that there are no hardware bottlenecks. This means that the function has complex instructions which take multiple clock cycles to execute. As mentioned, the variance function is used to calculate the variance for threshold and contrast detection. Calculation of variance is a floating point operation. Operating on floating point numbers is inherently slow, thus the function takes more time to execute. The analysis doesn't show a bottleneck because none of the floating point execution unit is overwhelmed and causes any stalls in processor.

After analysis of the hotspot functions and highlighting their corresponding bottlenecks we map these hotspot functions to algorithmic blocks so that the bottlenecks can be associated with the algorithm rather than specific code implementations and functions. The red sections in figure 12 show the hotspot blocks.

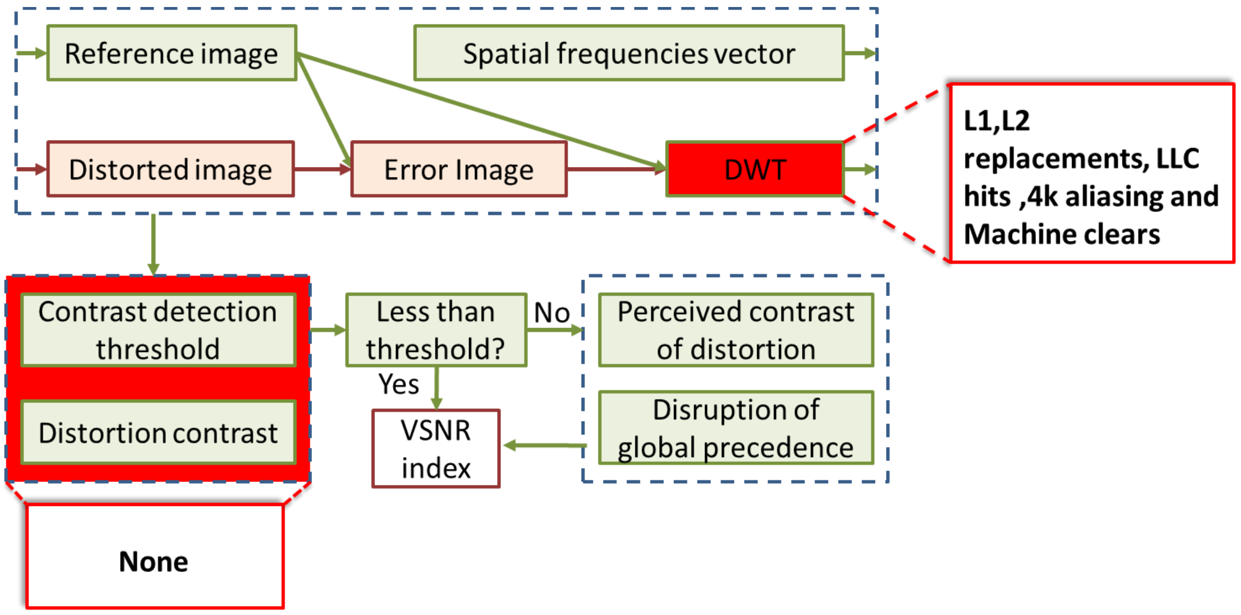

Reference image
Spatial frequencies vector
Distorted image
Error Image
DWT
L1,L2 replacements, LLC hits ,4k aliasing and Machine clears
Contrast detection threshold
Distortion contrast
Less than threshold?
No
Yes
Perceived contrast of distortion
Disruption of global precedence
VSNR index
None

Figure 12: Block diagram of VNSR with top hotspots mapped to corresponding algorithmic blocks and their corresponding bottlenecks.

The DWT-Rows and the DWT-cols functions belong to the DWT block with L1D, L2D replacements and LLC hits as bottlenecks for DWT cols and 4k aliasing a bottleneck for DWT rows. The variance function belongs to the contrast detection and distortion contrast block. It suffers from no hardware bottleneck.

CHAPTER IV

MAD

This chapter provides a brief description of the algorithm in section 4.1. Section 4.2 provides an insight into the implementation details of the code and specific methods used. The performance profiling and hotspot analysis is discussed in section 4.3 and finally, microarchitectural analysis and corresponding hardware bottlenecks mapped to the specific algorithmic blocks is presented in section 4.4.

4.1 Overview

MAD is one of the algorithms which has high estimation accuracy for visual quality of an image. Most of the IQA algorithms focus on the most dominating strategy used by the Human Visual System (HVS) while MAD uses multiple strategies to determine image quality of an image [34]. For images with near threshold distortions, the algorithm uses detection based strategy. While, the appearance-based strategy is used when the images containing clearly visible distortions. The diagram of MAD algorithm is shown in Figure 13. Using the input images: the reference image and the distorted/ subject image, the MAD index is computed via two main stages, the detection-based stage and appearance-based stage. Using the detection-based strategy, the detection-based difference map, the difference between the original and distorted images is computed. For the appearance-based strategy, the appearance-based difference map is computed using mean, variance, skewness and kurtosis for all local blocks of the log-Gabor filtered images. The detection-based difference and appearance-based difference maps combined to get the high-quality and low-quality index. Weighted geometric mean of the indexes is computed. The final MAD index is computed using the weighted mean and a specific weight determined based on the amount of distortion.

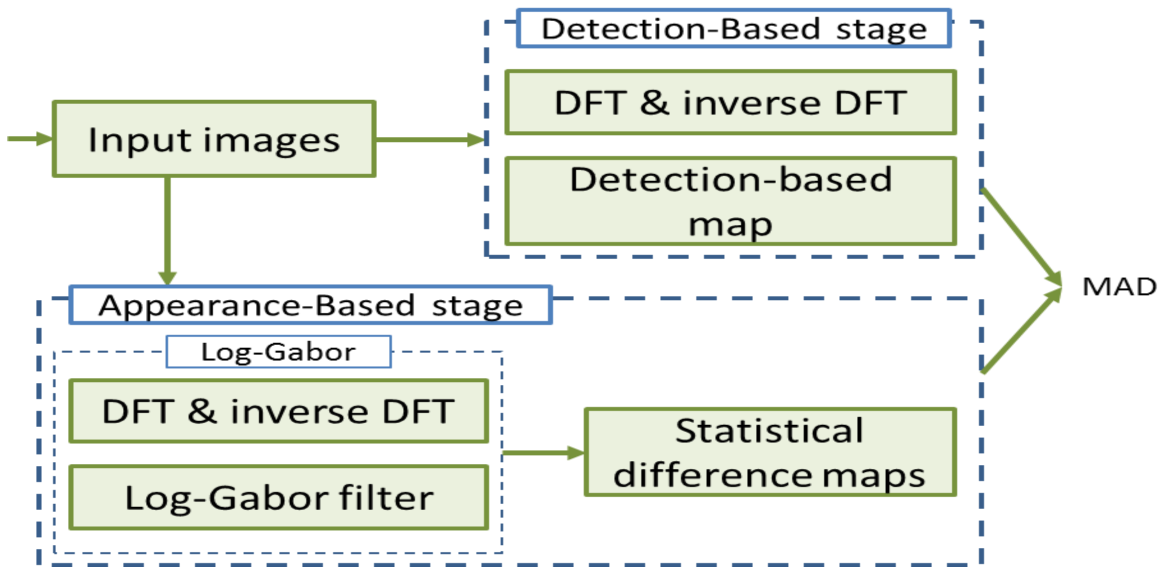


Figure 13: Block Diagram for MAD.

4.2 Implementation details and specifics

The MAD code is ported to C++ from the Matlab version, which is publically available to download [71]. The input images are stored in 2-D double precision arrays using GBuffer image library. In the detection-based stage, the images are taken to the luminance domain using look-up table. The Ooura's mathematical software packages [72] are employed for calculating Fast Fourier Transform and inverse FFT. This Ooura's library is also used in the log-Gabor decomposition in the appearance-based step. The statistical difference map is calculated using integral images for higher orders, detail can be found in our previous paper [67].

For each 512×512 image, MAD needs memory space for two input images, two luminance images, one for the error image, one for the CSF filter, four for the Ooura FFT computations, five for the masking and contrast maps, and around 80 images for the log-Gabor filter along with Ooura FFT, and statistical difference maps.

4.3 Hotspot Analysis

In this section, we discuss the top hotspot and their corresponding execution time for MAD. From the hotspot analysis we find that MAD has a very high execution time. The average total execution time and average execution time for individual hotspots, along with their % contribution to the total execution time is tabulated below.

Function/Block	**Average total execution time(s)**	**% of total execution time**
All	41.970	100%
Gabor Convolve	10.195	24.29%
Low index	4.061	9.67%
High index	2.827	6.73%
DFT	2.906	6.92%
Other	21.982	52.38%

Table 7: Average execution time for the Hotspot function, also expressed as percentage of total execution time.

As seen from the table, the top hotspot functions contribute about 50% of the total execution time and the other functions add up to the remaining 50%. So, to significantly remove performance or the total execution time all the top hotspot functions have to be targeted for optimizations. The top hotspot function is the Gaborconvolve function. It is used to calculate the appearance based difference map [34]. The function uses 5 scales and 4 different orientations to yield a total of 20 images for the original (reference) and the distorted (subject) image each. The function takes about 25% of the total execution time. Next is the Low Index function. It calculated the statistical difference map using variance, skew and kurtosis of the gabor filtered images [34]. The low index function takes about 10% of the total execution time. After the Lowindex function, the next hotspot is the High index function. It calculates the detection based map. The detection based map is calculated by finding the luminance of the reference and the distorted image, calculating the luminance error image and then applying a contrast sensitivity function to the DFT of the reference and the error image [34]. The final hotspot function is the DFT function; it converts the 2 input images into the Fourier domain. For the DFT function we use the Ooura's mathematical software packages [72]. The high index and DFT function take about 7% of the total execution time. We plot the individual execution time for all the subject images in figure 14.

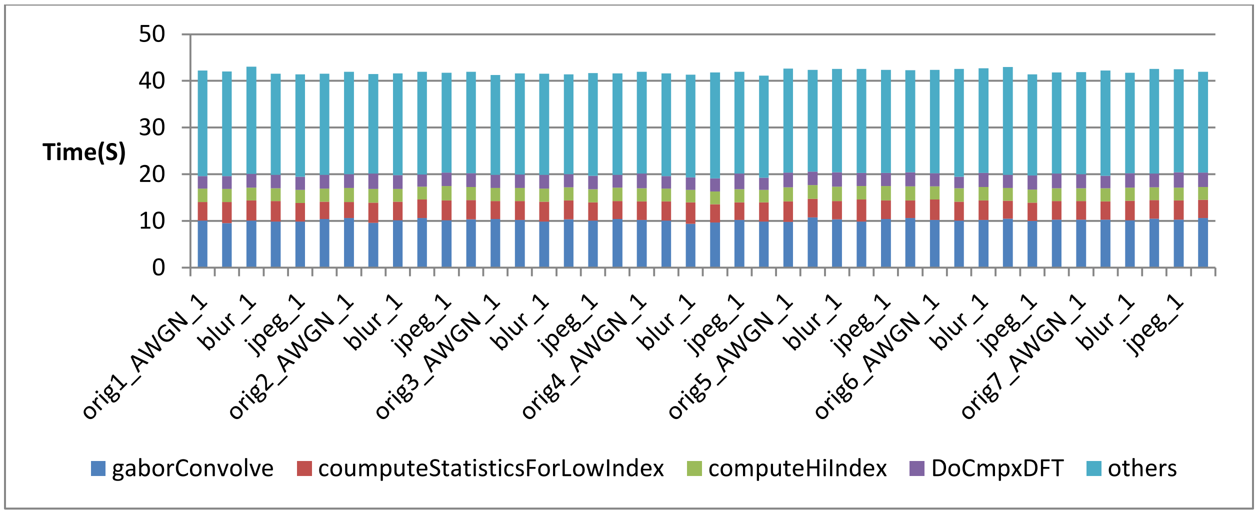


Figure 14: Individual execution time for all the hotspot functions and all the test images stacked to add up to the total execution time for a loop of 30 frames.

As seen from the figure MAD has very minimal variation in total execution time for the different subject images. Also, we analyze MAD to investigate if there are any variations in execution with change in Image content or the type of distortion of the subject Image. The results plotted and discussed below. Figure 15 shows the variation of average execution time of the hotspots w.r.t change in the Image contents. While, figure 16 shows the variation of average execution time for the hotspots w.r.t change in the type and level of distortion.

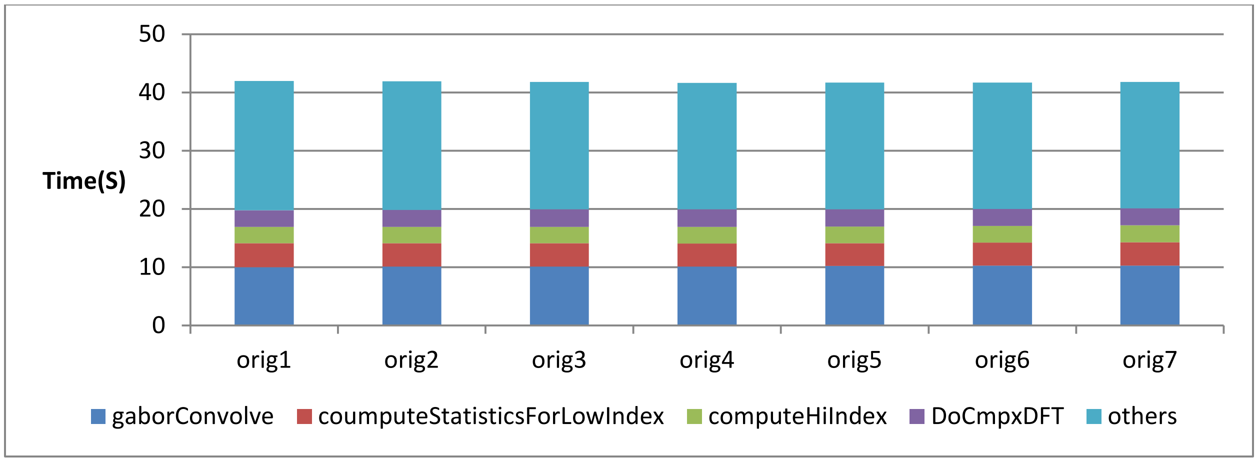


Figure 15: shows the average execution time of all the hotspot functions varied across 7 different image contents

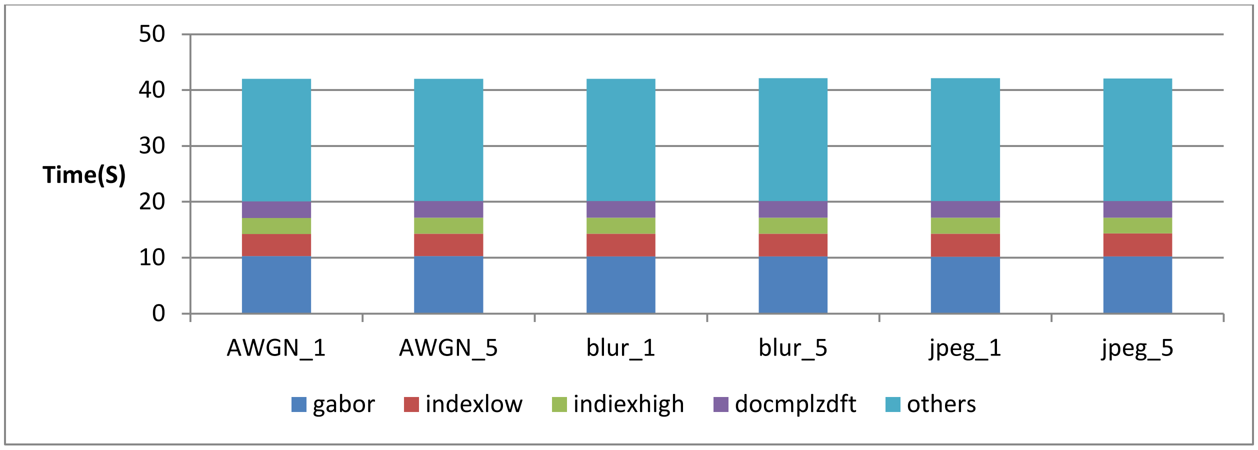


Figure 16: Average execution time of the hotspot functions varied across the 6 different distortions.

As seen from both the plot, we find the algorithm is stable across changes in the image content as well as across changes in the type of distortion. Thus, all the hotspots can be targeted directly for optimization, without any consideration of the image content or the type of distortion. No Image specific or distortion specific techniques or hotspot optimization are required for MAD. The optimizations made would be universal.

4.4 Microarchitectural Analysis

From the Hotspot analysis we find that the execution time for the execution time for MAD is very high. The execution time for individual hotspot functions is also high. We investigate the microarchitectural sub-systems to find the reasons and bottleneck for higher execution time and analyze the behavior of the algorithm and its interaction with the underlying hardware. The results are shown in table 8.

Results Summary:

Function/Block	**Image Type**	**Pipeline slot category**	**Backend slot category**	**Retired pipeline slot**	**Hardware bottlenecks**
Gabor Convolve	All	Backend	Memory	0.221	DTLB
Low index	All	Backend	Memory	0.475	DTLB
High index	All	Backend	Memory	0.374	L1D replacement, LLC Misses
DFT	All	Backend	Memory	0.225	L1D, L2D replacement, LLC misses

Table 8Microarchitectural Analysis for MAD, with hotspot functions and associated bottlenecks

The top hotspot function is gaborconvolve. As mentioned previously, it is used to calculate the appearance based difference map [34] and produces an output of a total of 40 images with double precision floating point number data type.   It has a retired pipeline slow metric of 0.221.The retired pipeline metric indicates that the block is inefficient and has a low instruction throughput. Investigating the microarchitectural resources utilized by the block, we find that the block is backend memory bound with DTLB overheads causing penalty and higher computational time. Along with major penalty from DTLB we find that there is penalty due to L1D, L2D replacement and LLC hits and misses.

To understand DTLB overhead, we first discuss a virtual memory system used in multiuser and multitasking environment and then discuss the use of DTLB in the virtual memory system [84].

Virtual memory separates the programmer's view of the main memory from the actual physical placement of blocks in the memory. This scheme creates an illusion that all virtually addressable memory is present in the physical memory and can be accessed by the program without any restrictions. Thus, the programmer can write his code at an abstract level without any consideration of memory size and knowledge of actual physical memory layout. Without virtual addressing, the programmer has to explicitly manage physical memory resources shared among multiple programs and multiple users. For example, a program would have to explicitly load and unload sections of the code that correspond to different phases of the program execution, because loading the entire program and the corresponding working data set would overwhelm the physical memory and consequently, other concurrent programs would starve for memory.

When the data is fetched from the main memory or stored back into the memory by the processor, it requires actual physical addresses. Thus, a translation from virtual address to physical address is required. This is done by a mapping table called the page table [80]. The page table is stored in the main memory. For the processor to get the physical address would take 100s of clock cycles because fetching data from the main memory is slow. So, to have a faster translation which is compatible with the speed of the processor, a cache is used. This cache is called the Translation Lookaside Buffer (TLB). A TLB stores a sub-set of the page table and provides mapping at a faster rate to the processor [80]. Since, the TLB stores only a subset to the page table, there might be cases where the mapping in not resident in the page table, this is called a TLB miss. When there is a miss in the TLB the mapping has to be brought in the TLB from the memory. This can take 100s of clock cycles. Frequent TLB misses can lead to a very high penalty and this is called the TLB overhead.

As TLB is very small in size, in order of a few bytes, and thus cannot store all the mappings. Now, as we see that the gaborconvolve function generates a total of 40 images, which is 512x512x4x40 bytes= 41.943MB. The TLB cannot store mapping for such a huge memory size, so there are high numbers of misses in the TLB. The penalty for a TLB miss is 7 clock cycles [77], assuming that the mapping is found in the next level of TLB. If the mapping is not found in the next level then the mapping has to be fetched in the main memory, which can be even more costly about 100s of clock cycles.

Followed by the gaborconvolve the next hotspot block is the Low index function which has a retired pipeline slot metric of 0.475. The low index function calculates the statistical difference using the variance, skew and kurtosis for the gabor filtered images [34]. The metric indicates that the block has an acceptable throughput but is not really efficient or close to the typical value. The computestatistic block like the gabor convolve block is also backend memory bound and has penalties due to DTLB overhead. As we see that the statistics are calculated for all the 40 gabor filtered images, the data set is still very huge and the mappings miss the TLB and thus cause a miss in the TLB and function suffers penalty.

To reduce the DTLB overhead, we recommend using “Super pages”. The Operating systems assign a virtual address space for a program at the granularity of a page size. Usually the page size by default is 4Kb. Super pages are pages with a page size larger than 4Kb. The intel sandy Bridge microarchitecture has support for page size of 2MB/4MB or 1GB [76]. Larger page sizes mean that a TLB cache of the same size can keep track of larger amounts of memory, which avoids the costly TLB misses. More details about the support for larger page size or super pages for intel architectures can be found in [76].

The high index function calculates the detection based map. The retired pipeline slot metric for the Highindex is 0.220. The block suffers from LLC misses which can cause very high penalties. As mentioned previously, the input data set for the function is input images, Luminance images and followed by Images in Fourier domain, all the levels of caches cannot fit all these images at one time and so the function suffers from LLC misses and the data or pixels have to be fetched from the main memory.

Finally, the DFT function has a retired pipeline slot metric of 0.4 and is also backend memory bound. We find that there are misses from all the levels of cache. The algorithm uses the DFT from the ooura mathematical library [72]. We take the DFT for the input images. The output of the DFT operation is a 1024x1024 pixel image (Imaginary part + real part). These images use the double precision floating point representation and hence each pixel is 64 bits. Total data set for the function is

$$2*512*512*8 \text{ bytes} = 4\text{MB}.$$

As we see, the total data set is larger than the size of all the data caches so the images do not fit into the cache and the memory controller has to go to the main memory too fetch the operand or pixels.

A brief description of the memory hierarchy and caches along with techniques like cache blocking that improves performance are provided in section 3.3.

Now, after studying the hotspot functions and the associated hardware bottlenecks, we map these functions to the algorithmic blocks in the block diagram. This gives a better idea to at an algorithm and generic level about the operations which are time consuming and can be a target for optimization. The mapping is shown in figure 17.

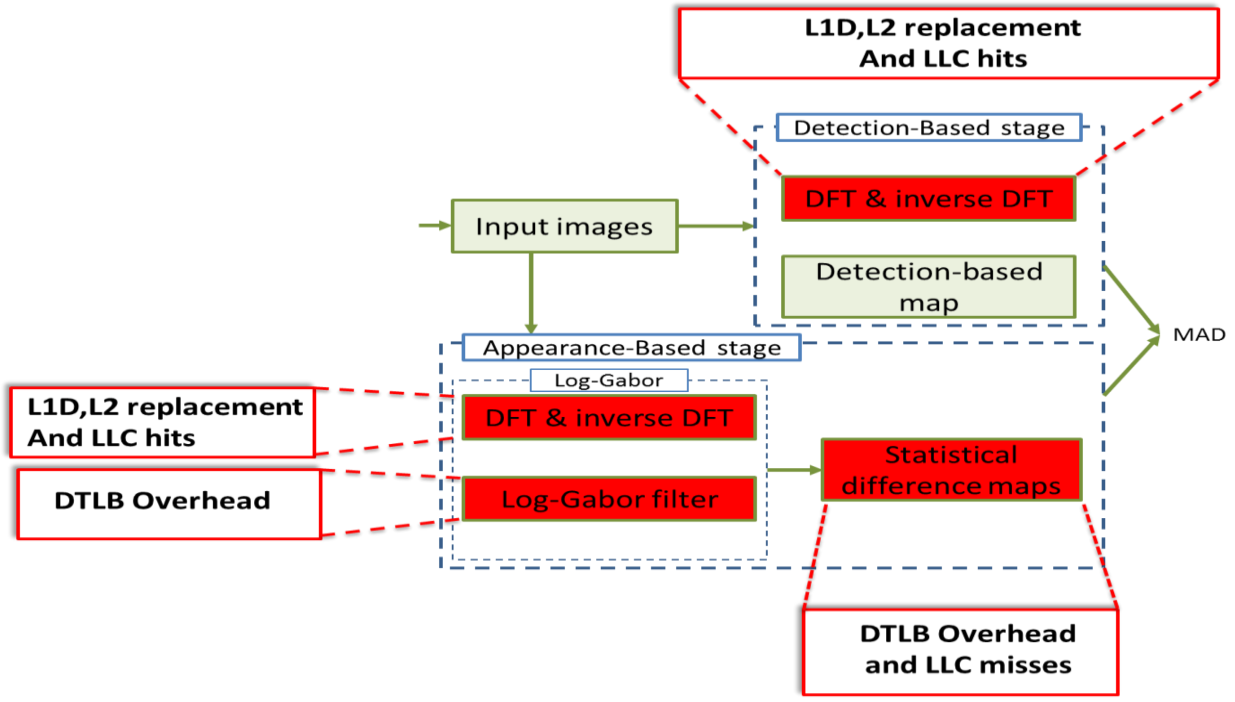


Figure 17: Hotspot function mapped with algorithmic blocks with Log-Gabor filter having the hotspot function gaborconvolve and DFT & IDFT block has DFT function from ooura library and low index function is a part of statistical difference maps block.

As we see from the figure, all the hotspot blocks are memory bound. The Gaborconvolve function belongs to the log Gabor block for appearance based stage. The DFT function is from the DFT & Inverser DFT block and the low index function performs the statistical analysis on the images and belongs to the statistical difference map block.

CHAPTER V

MS-SIM

This chapter provides a brief description of the algorithm in Section 5.1. Section 5.2 provides an insight into the implementation details of the code and specific methods used. The performance profiling and hotspot analysis is discussed in section 5.3 and finally, microarchitectural analysis and corresponding hardware bottlenecks mapped to the specific algorithmic block presented in section 5.4.

5.1 Overview

MS-SSIM [29] is based on the concept of SSIM [85] algorithm. The SSIM algorithm is based on a single scale while MS-SIM operates by applying and combining SSIM for multi-scales. The concept behind using multiple scales is the fact that the right scale depends on the viewing conditions.. SSIM is derived by considering hypothetically what constitutes a loss in signal structure. The algorithm is based on the hypothesis that distortions in an image due to variations in lighting, such as contrast or brightness changes, are nonstructural distortions, and should be considered and treated differently from structural ones, and that one could capture image quality with three aspects of information loss that are complementary to each other: correlation distortion, contrast distortion, and luminance distortion.

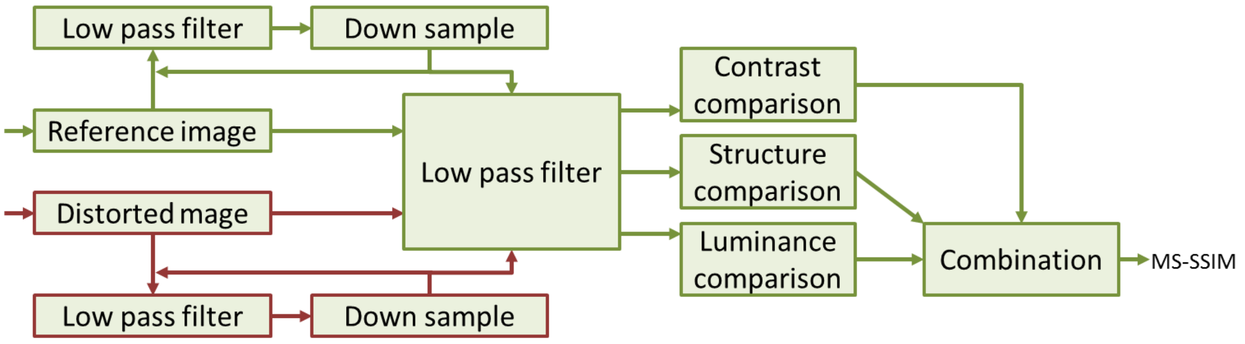


Figure 18: Block Diagram for MS-SIM.

5.2 Implementation Details and specifics

The C++ code implementation of the algorithm is basically ported from its Matlab version, which is publically available on LIVE website [77]. The input images are loaded to memory into 2-D double arrays using GBuffer library. In particular, the code uses five scales. To have smaller scales, a low-pass filter and down sampling operation are used. The image is filtered with the filter's size 2×2 and then down-sampled with a factor of two. For each scale, the SSIM is calculated starting with a Gaussian low-pass filter. The filter has the size 11×11 in Matlab version, now in C++ version we convolute the image twice with two one-dimension kernels. By convoluting separately, we reduce the number of callings to multiplication for one 512 × 512 image from 512×512×11×11 to 512×512×11×2. For the multi-scale, we need memory space for two input images and five scales of them. To store a 512×512 double precision image in memory, we need 2MB; thus, the algorithm needs approximately 50MB total in RAM.

5.3 Hotspot Analysis

Similar to the previous algorithms, we perform hotspot analysis to find the average total execution time and the most time consuming functions/blocks in the program. The top hotspot functions with their average execution time and their percentage contribution to the total execution time are shown in table 9.

Function/Block	**Average Execution time**	**% of total Execution**
All	2.513	100.00%
filter2_valid_11	1.431	56.95%
ssim_index	0.432	17.20%
operator_to_means	0.166	6.58%
Others	0.484	19.27%

Table 9: Average execution time for the Hotspot function, also expressed as percentage of total execution time.

As seen from the table, about 75% of the execution time is consumed by the top 2 hotspot functions filter2_valid_11 and the ssim_index. The filter function is an implementation of an 11x11 Gaussian low pass filter on reference and the subject image as well as their 5 scaled versions [29]. The average execution time is 1.431s which is about 56% of the total execution time. The next hotspot functions is the ssim_index_new function calculates the SSIM index for all the 5 scales using the luminance, contrast and the structure images [29]. Average execution time for SSIM_index_new function is about 0.432s which is 17% of the total execution time. Followed by these functions, the next hotspot is the operator_to_means finctions which calculates the luminance and the contrast images [29]. The block takes about 6% of the total execution time with 0.166s as the average execution time. We plot the individual total execution time along with the corresponding execution time for the hotspot functions for all the subject images in figure 19.

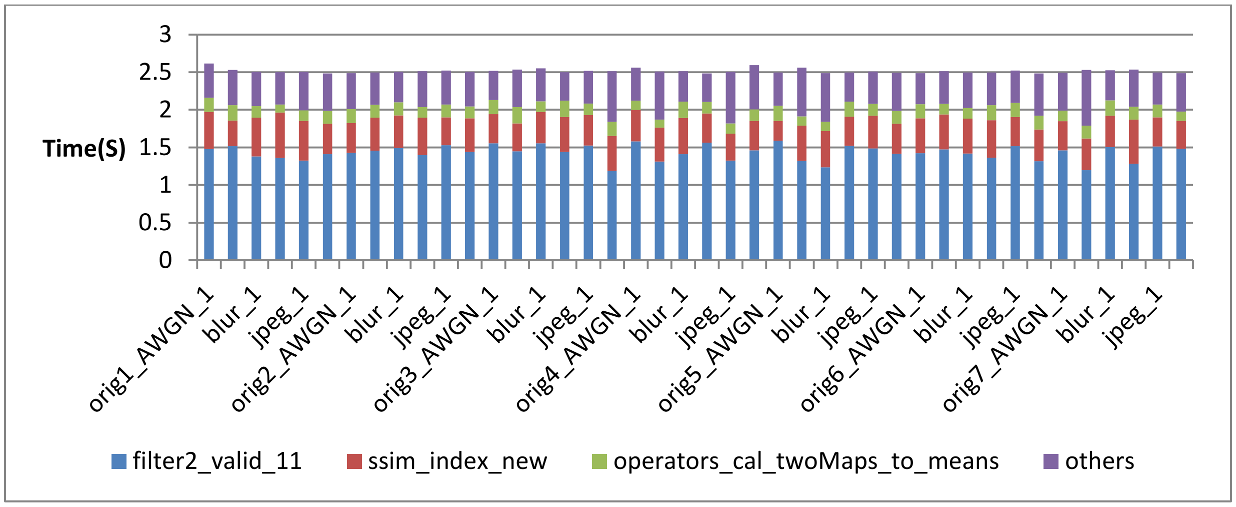


Figure 19: Individual execution time for all the hotspot functions and all the test images stacked to add up to the total execution time for a loop of 30 frames.

From the plot, we find that the algorithm does not have a very high variation in the total execution time for all the test images. We observe that the average execution for all the hotspot functions across the image content and different distortions is also stable. The graph for average execution time for the Hotspot functions across different Image contents is plotted in figure 20 and the plot across different Image distortion is shown in figure 21.

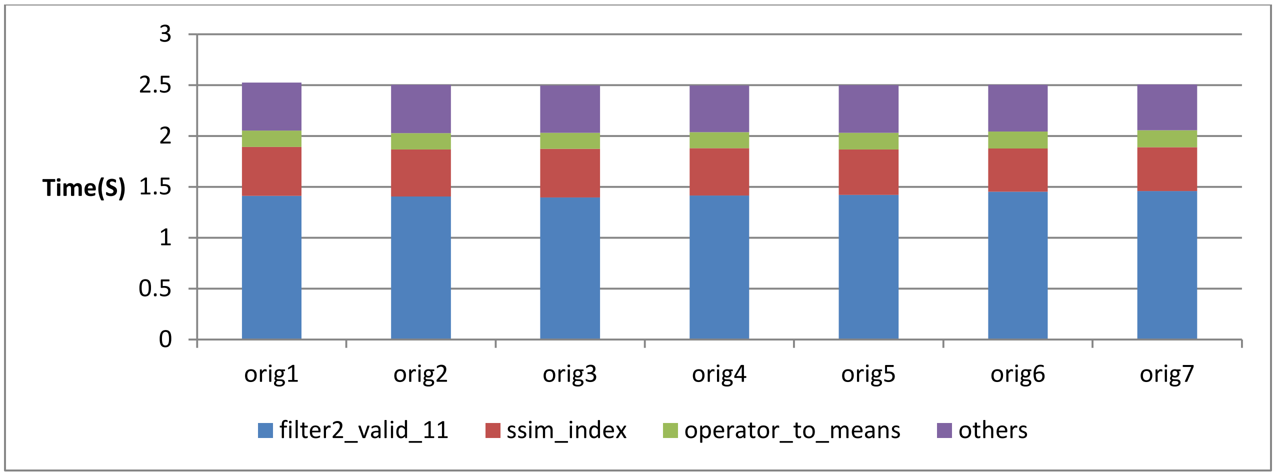


Figure 20: shows the average execution time of all the hotspot functions varied across 7 different image contents

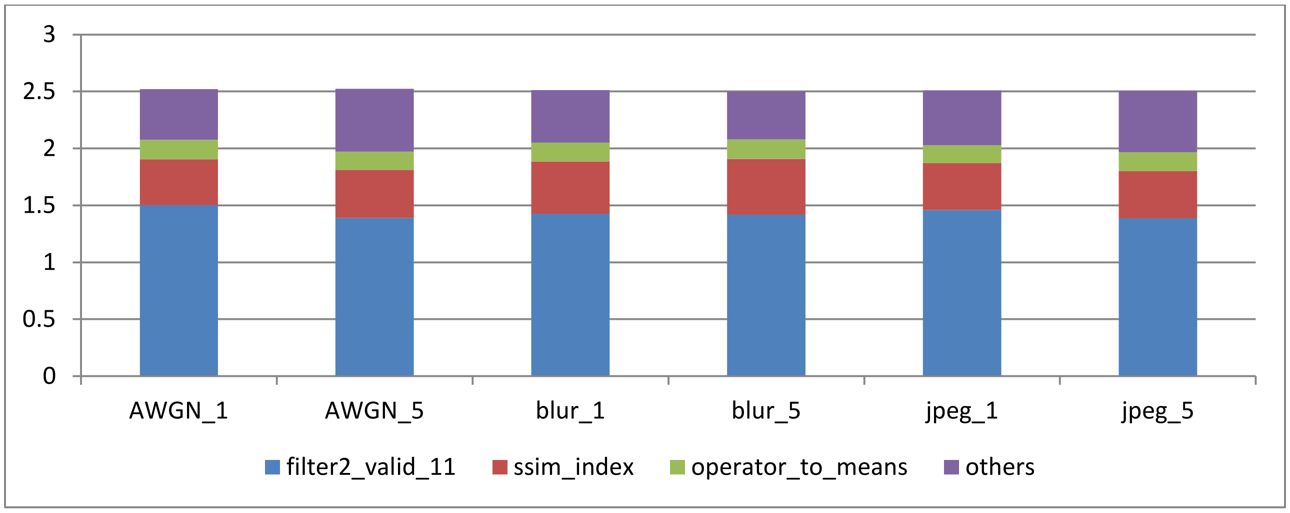


Figure 21: Average execution time of the hotspot functions varied across the 6 different distortions.

From the plots, we find that the algorithm is stable in terms of execution time for different Image contents as well as for different Image distortions. Thus, the hotspot functions can be directly targeted for optimization without any consideration of Image content or the type of distortion. The optimization can be universal, with the filter function as the first target for optimization.

5.4 Microarchitectural Analysis

After knowing the blocks/functions of the algorithm which take higher execution time we perform microarchitectural analysis to gain an insight into the hardware bottlenecks and see how the instructions traverse through the processor. The results are shown in table 10.

Results Summary:

Function/block	**Image type**	**Pipeline Slot category**	**Backend Bound slot category**	**Retired Pipeline slot**	**Hardware Bottlenecks**
Filter2_valid_11	All	Backend bound	Memory	0.620	L1D,L2D replacement
Ssim_index_new	All	Backend bound and Retired	Memory and core	0.350	L1D, L2D replacement, LLC miss. Assists (Retired)
Operator_tomeans	All	Backend bound	Memory and core bound	0.277	L2D replacement, LLC miss, DTLB overhead. Floating point Divide

Table 10: Microarchitectural analysis for MS-SSIM. DTLB – Data Translation look-aside Buffer.

The Filter2_valid_11 function is an implementation of a Gaussian low pass filter. This filter is applied to 5 scales of the reference image and the distorted image [29]. From the microarchitectural analysis, we find that the retired pipeline slot metric for the filter functions is 0.62. The metric shows that the function has a decent throughput but still can be targeted to improve performance. The associated bottlenecks are backend memory bound [72] because of the penalties due to L1D and L2D replacements [80].

As mentioned, the filter is applied to 5 different scales, which means that the input data set for the filter function is big and all the data cannot fit into the caches at a time. The filter is applied and then the filtered image is downsampled to calculate the next scale and then the down sampled image is again filter. We see that the result is utilized for the filter operation. Thus, from the hardware or caches point of view, once the block of data is brought in the cache it is utilized efficiently before it has to be evicted. So because of the larger data set we have replacements in L1D and L2D cache but once the data is brought into the cache all the required processing is done on it. So it is in the streaming mode for a certain period and then when a pixel is not found in the cache a miss occurs and data is brought in the cache from the next level. This replacement causes penalty and higher execution time and the utilization of a block already resident in the cache leads to an acceptable throughput. We present the technique to optimize cache utilization later in this section.

Following the filter block, the next hotspot block is Ssim_index_new. The SSIM index function calculates the SSIM index for each scale of the reference as well as the subject image [29]. As mentioned previously, a total of 5 scales are used. The input data set for each scale is a set of 3 512x512 double precision floating point images. Luminance image, contrast image and structure image. The retired pipeline slot value for the Ssim_index_new function is 0.35. The instruction throughput is low. The microarchitectural analysis indicates that the architectural bottlenecks for the blocks are in two categories backend bound and retired pipeline slots [tutorial]. First, the block is backend bound due to memory issues. We find that there is penalty due to L1D replacement, L2D replacement and LLC miss. As we see that the input dataset for the calculation of the SSIM index is very high 10 images. This big data set cannot fit into the caches. Thus, there are misses in all levels of cache. The penalty to satisfy LLC miss is very high and is typically 100s of clock cycle because the data is fetched off chip from the system memory. Thus, the block suffers many clocks cycles just with memory accesses and has a low instruction throughput.

To improve cache performance we suggest using cache blocking. Details of cache blocking are provided in section 3.3 and other techniques to improve cache utilization can be found in [86]

The second category of the bottlenecks for the SSIM function is the pipeline slot is retired instructions [72]. As mentioned previously, retired slots instructions are the ones which do useful work but the number of clock cycle to perform a task can be significantly high. There are instructions in the block which cannot be directly executed by the processor. Thus, these instructions are converted to a stream of microcode that can be executed by the processor. This micro code can be 100s of instructions long. Thus, the latency of executing these instructions can turn out to be very high. Since this micro code assists operations which cannot be directly handled by the processor, this code is called micro assist code. [76]

For calculating the SSIM index, luminance, contrast and structure are used to calculate the SSIM index [29, 85] for all the 5 scales. The formula to calculate the SSIM index is:

$$SSIM(x,y) = l(x,y)^{\alpha} * c(x,y)^{\beta} * S(x,y)^{\gamma}$$

Calculation of the SSIM index for each scale requires floating point operations. The IEEE 754 [87] standard is used for implementation for floating point operations but if the floating point numbers are very small and don't fall in the IEEE 754 standard, they cannot be directly executed by the processor. Thus, these floating point operations are converted in a stream of micro code and then inserted in the pipeline of the processor. This micro code has about 100s of instructions as mentioned. Thus, micro assists have a higher latency and are a bottleneck.

To remove micro assist overhead, we suggest to use single precision floating point numbers in place of double precision numbers wherever possible. Single precision computation is faster than double precision computation. Also, wherever possible use integers. Fast float to int routines can be used. We can use instructions in SSE2 or SSE3 [77] instructions. One another optimization method would be to use the intel compiler [88] which can generate instructions which exploit data level parallelism.

The next hotspot function is operator_tomeans. It is used to calculate the luminance and contrast for reference and the subject image and all the corresponding 5 scales. The function has a retired pipeline slot value of 0.277. The block is backend bound and bottlenecks are both the memory bound and core bound. For the memory bound the bottlenecks are the L2D replacement and LLC misses. Since, the input data set is 10 images with double precision floating point representation of pixels, it cannot fit in the cache and consequently the pixels or images are not resident in the cache and causes misses in all levels of cache. As discussed previously the LLC misses have a very high penalty on the instruction. Also, there is some penalty due to DTLB overhead. As described in section 4.3 the DTLB is used to store a subset of virtual address to physical address mapping page store in the main memory (RAM) to provide faster translation of virtual to physical address. If the virtual address is not found in the DTLB, then like the cache hierarchy the processor accesses the next level of TLB called the Second level Data translation look-aside buffer (STLB) [77] which has higher latency to provide physical address.

To improve performance and cache utilization, a cache blocking can be used. We have discussed cache blocking in section 3.3. For improving performance relevant to DTLB, using super pages would be helpful. Using super pages can get cover sparsely spaced memory, which is usually the cause for DTLB overhead.

For the core bound category, the hardware bottleneck is the floating point divide unit. As already mentioned, the function calculates the luminance and the contrast. The luminance is calculated using the mean while contrast is calculated using the variance. More details about the luminance and the contrast function can be found in [29]. The calculation of luminance and contrast is a complex floating point operation. The floating point operations inherently are long latency operations and because of the continuous feed of floating point operations for every pixel and a total of 10 images, we find that the floating point divide unit is overwhelmed and is a bottleneck and we have a structural hazard [80].

For improving the Floating point Divide overhead we recommend to use the logarithmic number systems to do operation. The Floating point divide would turn in a subtract instruction. The subtract instruction would take less number of clock cycles than floating point divide operation. The only overhead is the conversion of the numbers into a logarithmic number system and then takes an anti-log. Since we are doing a very high number of floating point operations the overhead is acceptable. Also, If possible floating point operations should be converted into integer operations to inherently make the operations faster.

Now, after studying the hotspot functions and the associated hardware bottlenecks, we map these functions to the algorithmic blocks in the block diagram. This gives a better idea to at an algorithm and generic level about the operations which are time consuming and can be a target for optimization. The mapping is shown in figure 22.

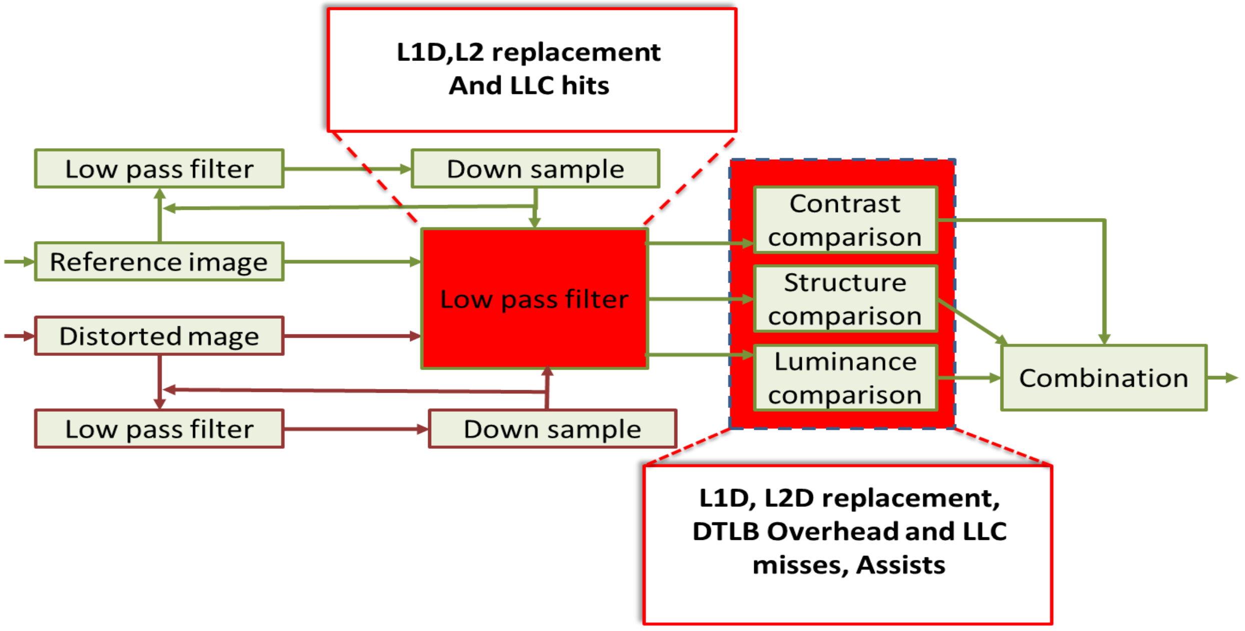


Figure 22: MS-SSIM block diagram with hotspot functions mapped to algorithmic blocks. The filter2_11 function belongs to Low pass filter block, the SSIM index and the operator to_means function belong to the comparison block.

The hotspot functions belong to 2 blocks. The filter functions belong to the low pass filter block and the remaining functions the SSIM_index and the operator_to_means function belong to the comparison blocks. The blocks suffer mainly due the memory bottlenecks.

CHAPTER VI

BLIINDS

This chapter provides a brief description of the algorithm in section 6.1. Section 6.2 provides an insight into the implementation details of the code and specific methods used. The performance profiling and hotspot analysis is discussed in section 6.3 and finally, microarchitectural analysis and corresponding hardware bottlenecks mapped to the specific algorithmic block presented in section 6.4.

6.1 Overview

The BLIIND-II algorithm [39] is a no reference algorithm. It uses the idea that the DCT statistics are symmetrically distributed but the distribution features are neither Gaussian nor laplacian. The generalized natural scene statistic based mode of local DCT coefficients is used and parameters are transformed into features. The block diagram of the algorithm is show in Figure below. The input image, is the first scale, it is low-pass filtered followed by a two times downsampling process to generate two more scales. During the first stage, all three scales (input image, 2 downsampled versions) are operated by taking a 2-D DCT-transform with the block size 5×5 and two pixels overlapping within two neighboring blocks. In the second stage, a generalized Gaussian density model is applied to each block of DCT coefficients, and specific partitions within each DCT block. In the third stage, which is the feature extraction process, the features are derived from these model parameters.Finally, BLIIND index is calculated by applying a simple Bayesian model, but the parameters need to be trained. The block diagram for Bliinds is shown in figure 23.

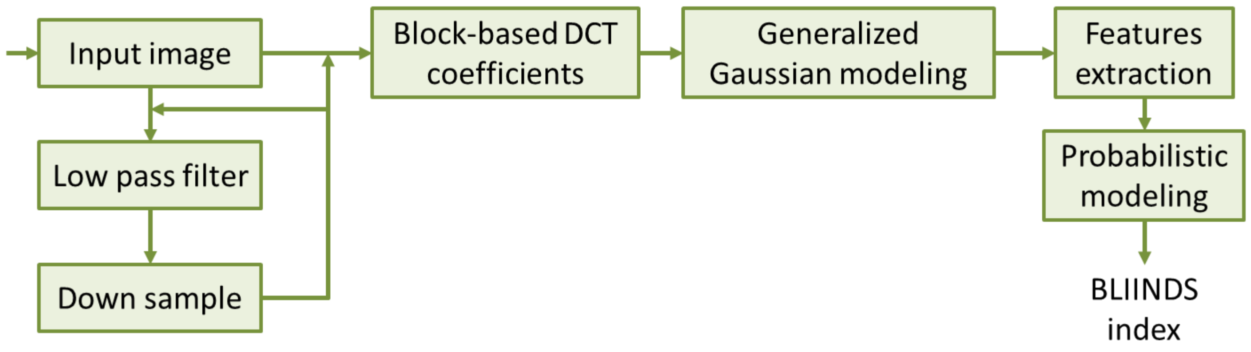


Figure 23: Block Diagram for Bliinds.

6.2 Implementation Details and specifics

The Matlab code was obtained from first author of BLIINDS-II via email in early 2012, and is confirmed to be the newest version available. The C++ code is ported from Matlab version with some optimizations. As shown in Figure (3), the input image is loaded to memory as a 2-D float precision array using GBuffer library. In the low-pass filter and down sample step of Matlab code, the image is convoluted with a Gaussian kernel size 3×3, and then down-sampled by factor of two. In the C++ version, we optimize this step by using separable convolution. We convolute the image with two 1-D kernels separately, each of the size 3×1 instead of 3×3. After this step, we have three scales. With each scale, the 2-D DCT coefficients are calculated locally for the blocks of the size 5×5 with 2 pixels overlapping between neighboring blocks. This step is also optimized using look up table for 25 values of cosine function.

In the Generalize Gaussian modeling step of Matlab version, function is calculated over and over for each block. In our C++ code, this function is pre-calculated once out of the loop. In this step, fitting process of DCT data histogram in each frequency bands to the model (12) is implemented as a line search procedure in 9970 values of in the range of $[0 - 10]$. In the features extraction step, the algorithm needs to sort algorithm to take 10^{th} and 100th percentile. Finally, the BLIINDS index is then calculated from the extracted features by some simple point-point multiplications. The first C++ code needs about the same running time as the Matlab code. By combining these above optimizations, we have a version that is 300 times faster. The new code needs less than 0.3 second to work with a 512 image while the original code needs more than 100 seconds. We need 1MB in memory to store a 512 image in float precision. The C++ version needs to store seven arrays of gamma, frequency, energy and rho features for three orientations with each scale, thus makes approximately 14MB of memory total.

6.3 Hotspot Analysis

We conduct hotspot analysis to find the total average execution time for Bliinds. Followed by calculation of the average execution time we find the top hotspots in the algorithm, along with their individual execution time and their percentage contribution to the total execution time. The results are shown in table 11.

Function/block	**Average Execution Time**	**% of total Execution time**
All	8.0327	100%
dct2_55_fast	3.811	47.44%
gama_dct	1.882	23.44%
rho_sorted	0.297	3.69%
convolve	0.292	3.64%
Other	1.750	21.79%

Table 11: Average execution time for the Hotspot function, also expressed as percentage of total execution time.

As seen from table 11, the average total execution time for BLIINDS is about 8s.The most time consuming function/block is the dct block with approximately 48% of the total execution time. The block has an average execution time of 3.81s. As mentioned previously, the block calculates the DCT coefficients of the subject image and the 2 different scale images (down sampled).The window size of the DCT transform is 5x5. Followed by the DCT block is the gama_dct block. The block basically maps the rho values to Gaussian. It is basically Gaussian curve fitting function. As seen from the table, the block consumes about 23% of the total execution time and at an average the execution time for the block is 1.88s. Consequently, we find that the DCT and gama DCT block/function takes about 70% of the total execution time.

Following the DCT blocks, the next hotspot function is the rho_sorted function/block. The function is basically a sorting function and after sorting the array it takes the 10th percentile of the array. The rho_sorted function is used for feature extraction. It is employed for multiple features and thus is called multiple times in the code. More details can be found in [29]. From the table we find that the execution time for the block is 0.296s at an average, which is about 4% of the total execution time. Also, the convolve function/block has the close average execution time to rho_sorted about 0.292s and consumes about 4% of the total execution time. The block performs convolution across the subject image and is employed as a low pass filter. Finally, all the other functions add up to 22% of the total execution time and have an average time of about 1.75s.The individual execution time for all the subject images in plotted in figure 24.

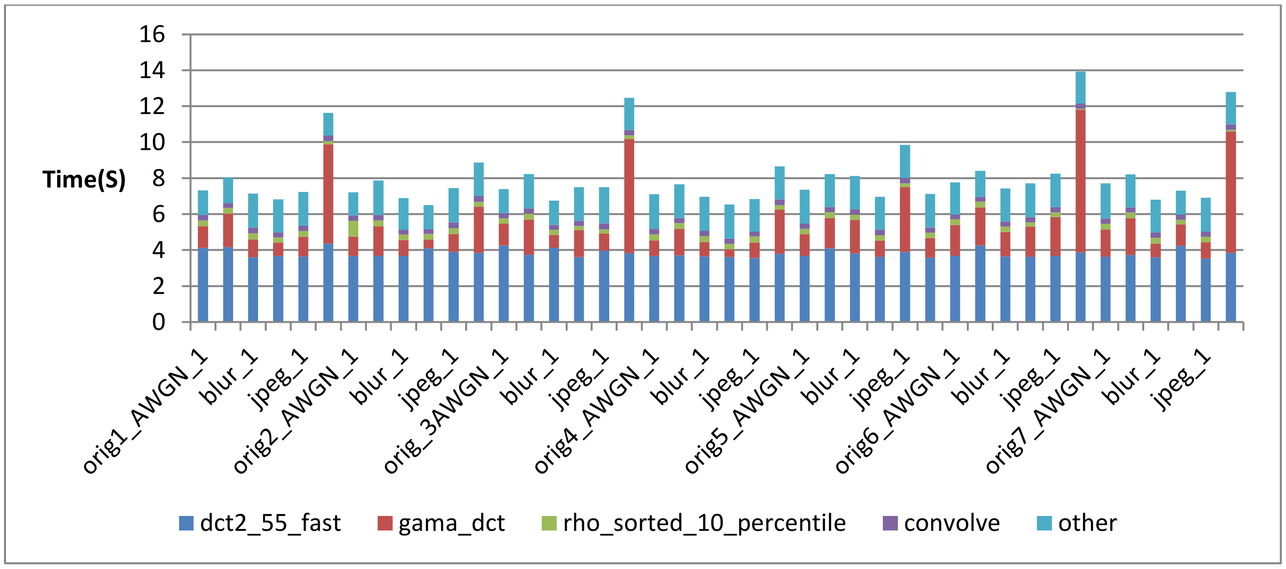


Figure 24: Individual execution time for all the hotspot functions and all the test images stacked to add up to the total execution time for a loop of 30 frames.

As seen from the figure 24, we see that the individual execution time varies from Image to image and for some images the variation is high. Thus, to study the variation we plot the average execution time for the hotspot functions varying across different distortions and across image content. These graphs are shown in figure 25 and figure 26.

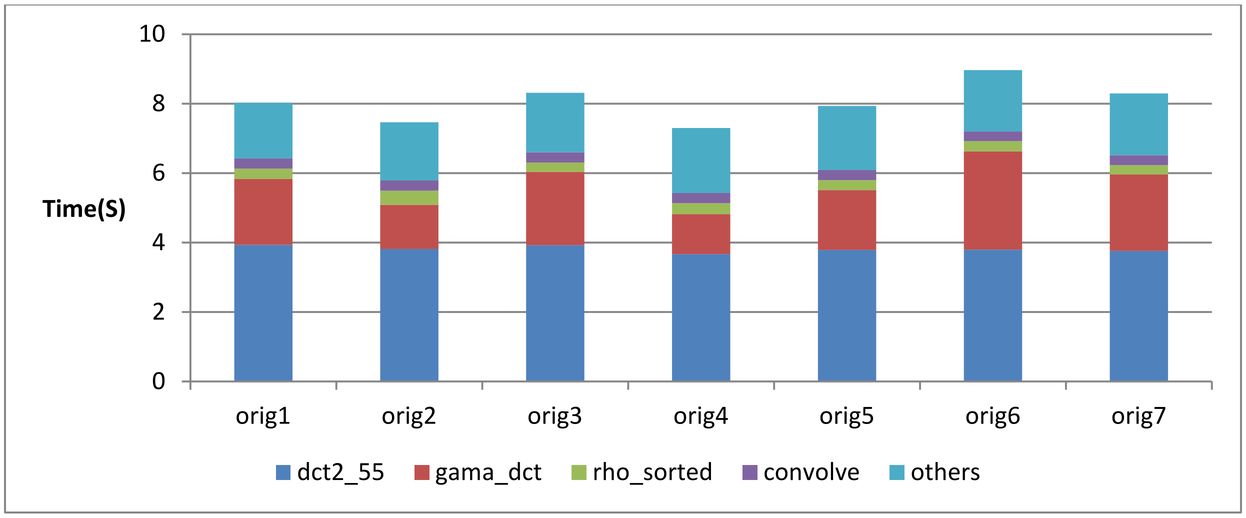


Figure 25: shows the average execution time of all the hotspot functions varied across 7 different image contents

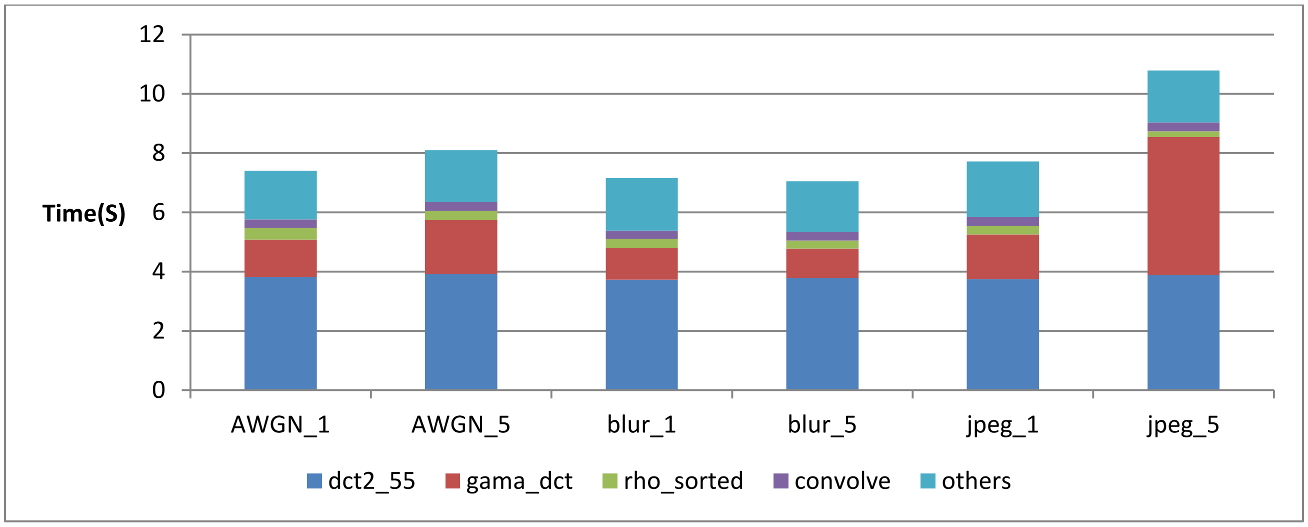


Figure 26: Average execution time of the hotspot functions varied across the 6 different distortions.

As seen from the plot, we find that there is significant variation in the total execution time based on the distortion of the subject image. As observed the images with distortion jpeg-5, have a higher execution time. We find that the gama dct block in the jpeg-5 images takes more time to execute and this is the reason for higher total execution time of the program. Similarly, the gama dct function takes higher execution time for AWGN-5 and jpeg-1 images. We find that the variation/increase for the gama dct function for the AWGN-5 and Jpeg-1 is not as large as jpeg-5 images. Thus, the execution time doesn't increase significantly as compared to jpeg-5 images. From the plots we infer that, if the subject Image is a jpeg image or an image with high white Gaussian noise the initial target for optimization should be the gama_dct block followed by the dct2_55 block.

6.3 Microarchitectural Analysis

After the hotspot analysis and knowing the blocks which take higher execution time we conduct microachitectural analysis to have a deeper insight into the architectural bottlenecks. The algorithmic blocks along with the corresponding hardware bottlenecks are listed in table 12.

Results Summary:

Function/block	**Image type**	**Pipeline Slot category**	**Backend Bound slot category**	**Retired pipeline slots**	**Hardware Bottlenecks**
Dct2_55	All	Backend bound	None	0.65	None
Gama_dct	All except jpeg-5	Backend bound	None	0.70	None
Gama_dct	Jpeg-5	Backend bound	Memory	0.50	L1D replacement
Convolution	All	Backend bound	Memory	0.487	L1D,L2D replacement, LLC Hit
Rho_percentile	All	None	None	0.244	None

Table 12: Results for BLIINDS**.**

Conducting the hotspot analysis we find that the top hotspot function is Dct2_55 function/block. For microarchitectural analysis, we use the retired pipeline slot metric to measure the throughput of a particular function and investigate if the function traverses the pipeline efficiently. Table3 shows that the retired pipeline slot metric value for the dct block which calculates the DCT coefficients averages to 0.65, which means that the hardware resources are being used optimally (0.6 is the typical value that is considered to be good and acceptable). When we investigate the behavior of the block with reference to microarchitectural sub-systems, we find that there are no architectural bottlenecks.

We describe in details the implementation of the DCT function to an insight about the throughput of the function. As mentioned in the implementation specifics section, the DCT function/block uses a look up table to store the cosine values. Since Cosine functions are expensive, precalculation of the cosine function saves computation. Along with using the lookup table, we use single loops for first row and column and a nested loop for the remaining pixels. As mentioned, the cosine operation is expensive but it is eliminated using the look up table, all the other functions are not very expensive so the throughput is acceptable. Consequently, number of stalls is not very high and the block/function has no hardware bottlenecks.

The next hotspot function is gama_dct function/block. It also has an acceptable retired pipeline slot value of 0.70 for all the images except jpeg-5. The function is a Gaussian curve fitting function. It maps rho values to Gaussian. [39].From the microarachitectural analysis we find that there are no hardware bottlenecks for the function except for jpeg-5 images. For Jpeg-5 images the gama_dct block has a retired pipeline slot value of 0.5 and takes maximum execution time to become the top hotspot. The associated hardware bottleneck is backend memory bound. Specifically the data is replaced in the L1D cache which means the processor fetches data from L2 cache. The L2 cache has higher latency.

As mentioned in the implementation section, the gama_dct function maps rho values to Gaussian. During the process it traverses an array of 9970 values. If there is a match the traversal stops. We observe that for Jpeg-5 images, the function has to traverse more indexes to map the rho values. The array contains floating point double precision values.

Array dataset= 9970*8= 79.76 Kb.

From table 1, the Level 1 data cache is just 32Kb and cannot hold all the data. Therefore, some of the values in the array are stored in the next level of cache. Consequently, for jpeg-5 images the functions has to go to the next level of cache to fetch the data. This causes higher latency and higher execution time for gama_dct function. To improve the performance for the gama_dct function, we suggest traversing the array based on the input image’s profile. For jpeg-5 images, traversing the array from the end would map the Gaussian value in fewer indexes and improve performance.

Followed by the DCT blocks the next hotspot is the convolution/function block. From the microarchitecutral analysis, the convolution block has a retired pipeline slot value of 0.5 at an average and is backend memory bound. We find that there are L1D as well as L2D replacement penalties and the processor also has to access the LLC for some operands. As mentioned, the LLC has a large latency of about 26-31 clock cycles and is the cause for the function to be slow [77]. We discuss the memory hierarchy sub-system section 3.4 and in [80].

Finally, the rho_percentile block has a retired pipeline slot value of 0.2 but the microarchitecural analysis does not show bottleneck which means that the block inherently has complex computation which take higher clock cycles.

Along with the top hotspot functions, we find some common bottlenecks for all the images for other functions. The floating point divide unit is a bottleneck for bloc_procexpnd and calculating the square root for statistical analysis.

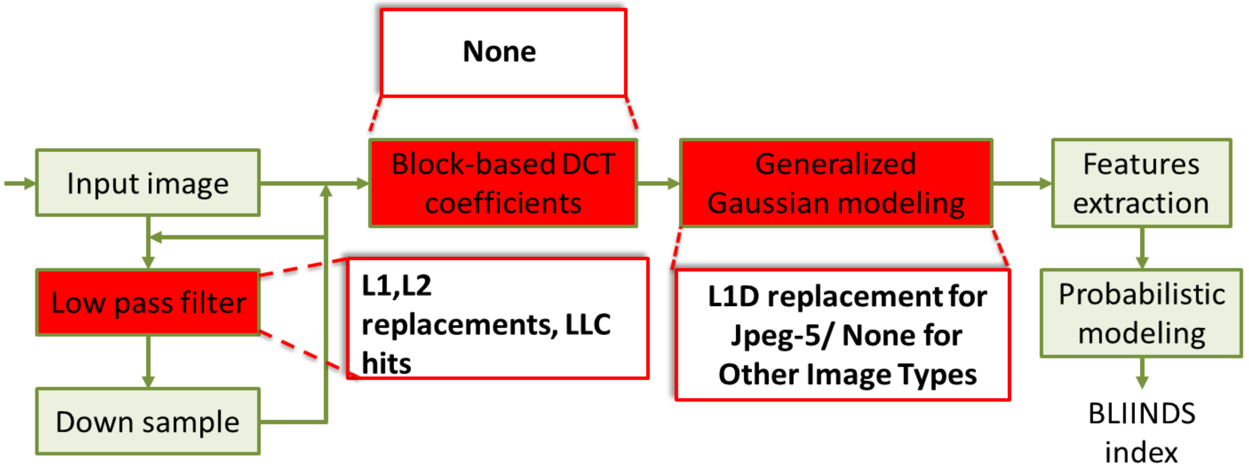

None
Input image
Block-based DCT coefficients
Generalized Gaussian modeling
Features extraction
Low pass filter
L1,L2 replacements, LLC hits
L1D replacement for Jpeg-5/ None for Other Image Types
Probabilistic modeling
Down sample
BLIINDS index

Figure 27 : Block diagram for BLIINDS with hotspots functions mapped to algorithmic blocks. The DCT function maps to Block-based DCT coefficient, Gama-DCT function maps to generalized Gaussian modeling block and filter function maps to the Low pass filter block.

CHAPTER VII

VIF

This chapter provides a brief description of the algorithm in section 7.1. Section 7.2 provides an insight into the implementation details of the code and specific methods used. The performance profiling and hotspot analysis is discussed in section 7.3 and finally, microarchitectural analysis and corresponding hardware bottlenecks mapped to the specific algorithmic block presented in section 7.4.

7.1 Overview

VIF [30] is a full reference IQA algorithm. The algorithm uses the Natural scene statistics based on information fidelity. VIF considers distortion of an image as loss of information. It explores the relationship between image information and the amount of information shared between a reference and a distorted image and its impact on visual quality. Along with this, VIF quantifies the information content of the reference image as being the mutual information between the input and output of the HVS channel. The information gained is the amount of information the human brain could extract from the reference image. Similar processing is done for the distorted image to yield the information that the brain could extract from the distorted image. These two information measures one from the reference image and the other from the distorted image are then combined to form VIF index which gives the measure of the visual quality.

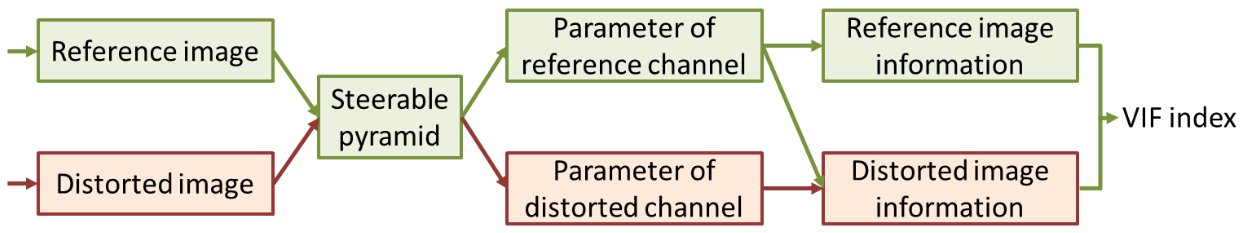

Reference image
Distorted image
Steerable pyramid
Parameter of reference channel
Parameter of distorted channel
Reference image information
Distorted image information
VIF index

Figure 28: Block diagram for VIF.

7.2 Implementation details and specifics

The C++ version of VIF algorithm is ported from Matlab code which is publically available in LIVE's website [75]. The VIF algorithm uses Steerable Pyramid toolbox [74] in Matlab, when ported to C++, it uses the Steerable Pyramid C library of the same author. The block diagram is shown in Figure 28. First, both reference and distorted image are loaded to 2-D double arrays using GBuffer library, then both of them are decomposed via Steerable Pyramid with four levels. In the Matlab version, the algorithm use Steerable Pyramid with six orientations for each level, thus 24 times calling to the filter, but VIF uses two orientations only. In C++ version, we modified the Steerable Pyramid library to filter only those orientations which VIF uses latter. This modification reduces the number callings to the filter from 24 to 8. From the reference subbands, the parameters of the reference channel are calculated, then the reference image information. Both distorted and reference subbands are used to compute the parameters of distorted channel, and then, from the parameters of both reference and distorted channel are evolved to compute the distorted image information, as in equations (8) and (9). To calculate the eigenvalues of CU, we employ the Newmat C++ matrix library [76]. There is the conversion between data type here, but the speed does not change much because the CU matrix is small, 9×9.

From the image information of both reference and distorted image, the VIF index of two input images is the rate of distorted image information over reference image information. The more VIF index closes to one, the better distorted image comparing with the reference image.

For input images with size 512×512, we need 2MB for each image in double precision. The code needs eight pairs of filtered subbands, one high band and one low band for Steerable Pyramid, eight images for G, V , S with different sizes and eight images for CU with 9×9, that makes approximately 20MB in memory.

7.3 Hotspot Analysis

Following the same procedure, we find the total average execution time for the algorithm, the hotspot functions and their average contribution to the total execution time. The results are shown in table 13.

Functions/block	**Individual Execution time**	**% of total Execution**
All	12.121	100.00%
internal_filter	3.693	30.47%
internal_reduce	2.839	23.42%
refparams_vecgsm	2.147	17.72%
other	3.444	28.41%

Table 13: Average execution time for the Hotspot function, also expressed as percentage of total execution time.

From the table, we see that the top hotspot functions contribute about 70% to the total execution time, with the top hotspot internal_filter taking about 30% of the total execution time at an average. The internal_filter function.The individual execution time for all the subject images and the hotspots is shown in figure 29. We find that there is not a very high variation in total execution time for Images. To investigate the variation of hotspots with change in Image content and types of distortions we plot the average execution time for all the hotspot functions varying across the Image content as well as types of distortion. The results are plotted in figure 30 and figure 31.

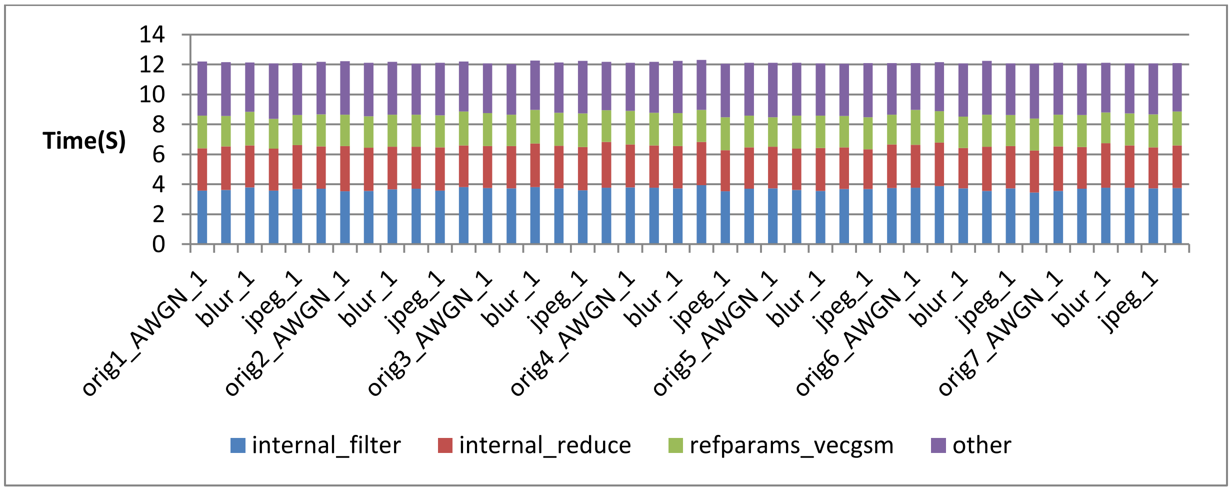


Figure 29: Individual execution time for all the hotspot functions and all the test images stacked to add up to the total execution time for a loop of 30 frames.

Now, we investigate the variation for the execution time with change in Image contents and types of distortion.

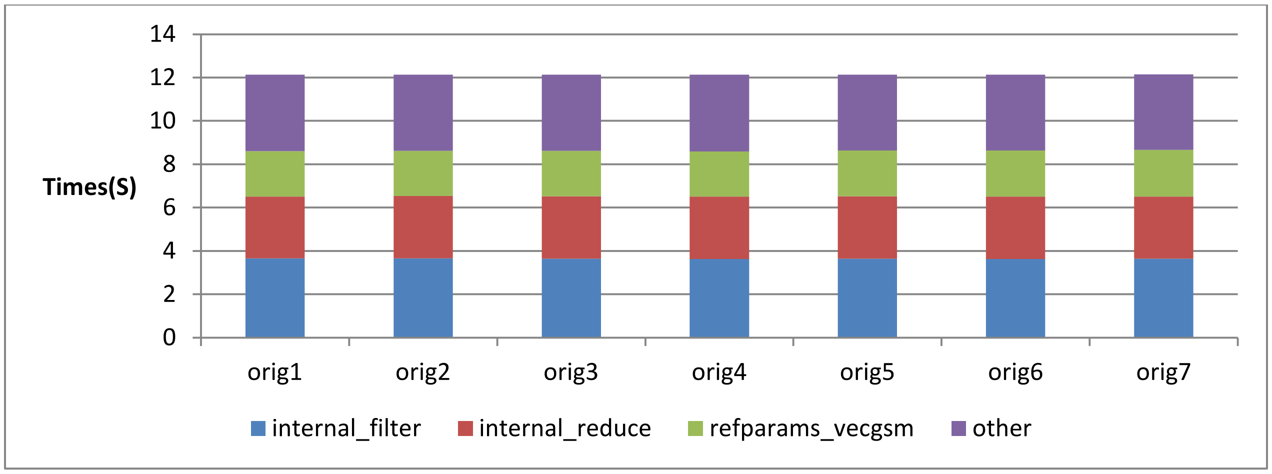


Figure 30: shows the average execution time of all the hotspot functions varied across 7 different image contents

As seen from the plot, the algorithm and all the hotspot functions are very stable with change in image content. The variation is minimal across the image contents. We now investigate the variation of the execution time with change in types of distortions for the subject Image. The results are plotted in figure 31.

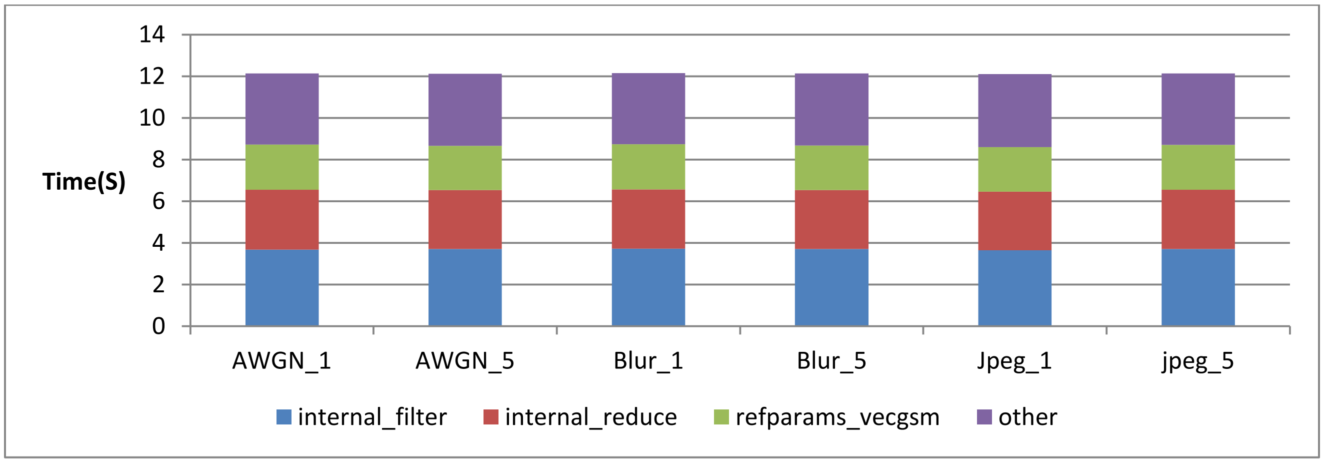


Figure 31: Average execution time of the hotspot functions varied across the 6 different distortions.

Similar to the changes in Image content plot, we find that the variation of the execution time for the hotspot functions with types of distortions is also stable. Thus, from all the plots we find that the algorithm doesn't have any variation with reference to different distortion levels as well as Images contents. Thus, the hotspots can be target for optimizations directly without any consideration of any parameters.

7.4 Microarchitectural Analysis

As per the procedure for the above 3 algorithms, after hotspot analysis we conduct microarchitectural analysis, study the retired pipeline metric determine efficiency and throughput of the block. Once the throughput is determined, we find the associated architectural bottlenecks. The function blocks and associated bottlenecks are shown in table 14.

Results Summary:

Function/block	**Image type**	**Pipeline Slot category**	**Backend Bound slot category**	**CPI**	**Hardware Bottlenecks**
Internal_filter	All	Backend bound	Core		Slow LEA Stalls
Internal_reduce	All	Backend bound	Memory and Core bound		L1D replacement. Slow LEA Stalls
Refparams_vcgsm	All	Backend bound	Memory		L1D, L2D replacement, LLC hit, LLC miss, DTLB overhead.

Table 14: Microarchitectural analysis for VIF. LEA- Load effective Address

As mentioned in the hotspot analysis, the top hotspot is the Internal_filter block. The block operates as a filter function on the image as mentioned in the previous section. We study the microarchitectural analysis to find that the block has a retired pipeline slot value which is 0.632. This implies that the block is close to the typical value of the metric and has a good throughput. The block is backend core bound. We study architectural resources utilized by the block to find that there is a minor penalty due to slow LEA instructions which have higher latency.

The Load effective address or the LEA instruction is an assembly instruction. As the name suggests calculates the effective address and places it in a register. These instructions can be very useful when performing an operation using the same address again and again. These instructions prevent the calculation of the address again.

Example:

Add.w $1, ($2, $3), A0;

This is an add instruction where the effective address for an operand is values $1+$2+$3. So if the operation is being used multiple times then a better way to do this operation is

Lea $1, ($2, $3), A5;

Add.w (A5), A0;

Thus, as seen Lea instruction is a powerful tool to prevent the calculation of an address again and again. For the Intel Sandy Bridge microarchitecture, these instructions are fed to the execution core via port 1 and port 5 [77]. But this is applicable only if the operands for the lea instructions are just one or two. For lea instructions with three operands as the one in the above example, these instructions can only be dispatched or fed to the execution core through port 1 only [77]. These instructions have a longer latency of 3 clock cycles [77].

Also, there are some other special cases, where the Lea instructions can take 3 clock cycles to execute even with 2 operands. The details about these cases can be found in [77]. We find that the LEA instruction where generated by the compiler for the index access variable in the loop.

Next, we analyze the second hotspot function, Internal_reduce. The block has a retired pipeline slot metric value of 0.62. This implies that the block has a good throughput similar to the above internal_filter block. The microarachitectural analysis for the blocks shows that the block is backend memory bound as well as core bound. There are minor penalties with L1D replacement and a few stalls due to LEA instructions.

The top third hotspot after the filter block is the refparams_vecgsm block. Using the microarchitectural analysis we find that the block has a low retired pipeline slot metric of 0.22. Consequently, the throughput for the block is low. Further investigating, we find that the block is backend memory bound and suffers major penalty due to LLC miss and LLC hits. The processor has to fetch data from the LLC or the DRAM (main memory). As mentioned previously, the penalty for accessing the LLC is about 26-31 clock cycles.

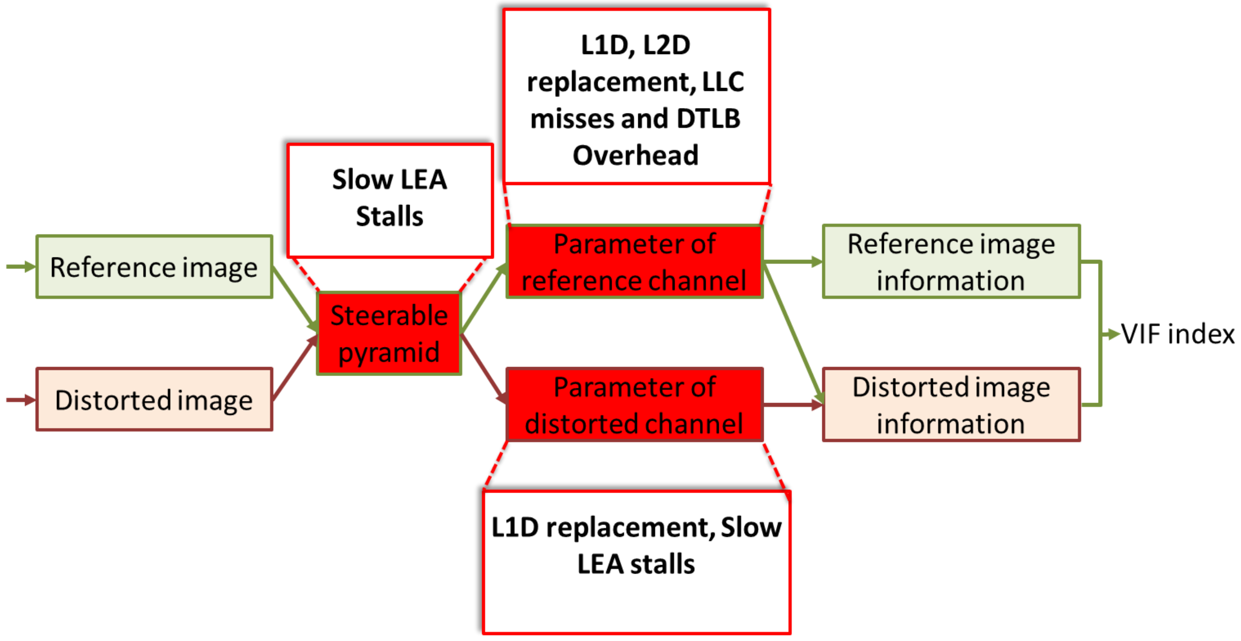

L1D, L2D replacement, LLC misses and DTLB Overhead
Slow LEA Stalls
Reference image
Distorted image
Steerable pyramid
Parameter of reference channel
Parameter of distorted channel
Reference image information
Distorted image information
VIF index
L1D replacement, Slow LEA stalls

Figure 32: Block diagram for VIF with hotspots functions mapped to algorithmic blocks.

As seen from the figure, the major bottleneck for the algorithm is the generation of the LEA instructions by the compiler. We also find memory bottlenecks due to data replacement in the caches and the DTLB overhead.

CHAPTER VIII

DISCUSSIONS AND RECOMMENDATIONS

This chapter we discuss the outcome of the analysis as a whole. Along with a discussion on the bottlenecks for various algorithms we recommendation techniques to reduce penalty from these bottlenecks and improve performance. Section 8.1 discusses the memory bottlenecks and various techniques to improve performance by optimizing cache usage and reducing DTLB overhead. Section 8.2 provides a brief discussion about the tool in general. In section 8.3, we discuss the bottlenecks in the processor core and finally in section 8.4 we proposed a custom hardware engine for IQA algorithms.

8.1 Memory Bottlenecks:

From the microarchitectural analysis, we find that all the algorithms have a backend bound memory bottleneck but the penalty suffered by individual algorithms varies highly. When there is a memory bottleneck mostly, what this means is that the hotspot functions are trying to read from or write to memory locations, and the access patterns of these reads and writes are such that there are misses in the CPU caches. These cache misses have to be serviced from lower levels of the memory hierarchy, which are slower to access, which is why we see the programs spending more time in these functions. Thus, the large amount of data generated while transforming these images into other domains (insert algorithms that do this), creating extra images (MAD and add any other algorithms that do that) and stepping through these multidimensional arrays many times causes most of the performance bottlenecks for IQA algorithms. If we observe the results, the penalty suffered varies from algorithm to algorithm. For example, all the hotspot functions in MAD suffer penalty from being backend memory bound while VIF has a memory bottleneck for the refparam_vcgsm function. It should be noted that the refparams_vcgsm function is the last hotspot for VIF. Thus, its impact on the performance of the algorithm is not high as other functions which are not memory bound. Thus, from the discussion we infer that even if all the algorithms at some point have a memory bottleneck its impact and severity of the penalty on the complete algorithm depends on the rank of the hotspot, which consequently would decide the priority for optimization.

Another observation from the analysis is that even though at an abstract level all the algorithms show memory as a bottleneck, the actual physical microarchitectural bottleneck are different for different algorithms. For example, the top 2 hotspot functions in MAD have poor performance because of the DTLB overhead, while the top 2 hotspot for MS-SSIM have a higher execution time because of the L1D, L2D replacement and LLC misses.

Apart from issues with the usual suspects in the memory hierarchy (CPU caches at different levels), there are hotspot functions which show penalties associated with shared memory issues and violations which cause machine clears and 4k aliasing as in case of VSNR. Thus, our analysis reveals some interesting performance bottlenecks that would otherwise have gone undetected. One of the contributions of this paper is that it brings these issues to light, and discusses ways to resolve them through software techniques and through microarchitectural and hardware techniques.

Thus, we can infer that even if in broader sense, memory is an issue due to which the algorithms suffer a loss of performance, the actual bottlenecks as well as the impact on the performance is specific to each particular algorithm. Different microarchitecutral resources are overwhelmed by different functions and algorithms.

Let us first look at the most common aspect of performance loss due to the backend memory: data replacement in the cache (cache misses). Thus, we first discuss techniques to reduce cache misses and improve cache utilization.

1: System Tuning: As mentioned in section 2.2.1, choosing the best hardware which suits the application is called system tuning. Thus, if it is possible to change the hardware platform for IQA we suggest using a processor with a larger cache. This will reduce the number of cache misses, replacements and consequently improve performance.

2: Exploit locality: Locality of reference is the tendency of the programs of access same (temporal locality) or nearby (spatial locality) memory locations repeatedly and frequently [80]. Thus, caching these memory locations can reduce misses. Writing a code which exploits locality can significantly improve performance.

To explain how exploiting locality and improve performance we take an example of the following codes.

SAMPLE CODE 1

```
float sum_array ( int a[M][N])
{
            Int I, j, sum =0;
            for (i = 0; i<M; i++)
            {
                    for (j=0; j<N; j++)
                         {
                                Sum += a[i][j];
                         }
                 }
             return sum;
}
```

SAMPLE CODE 2

```
float sum_array ( int a[M][N])
{
            Int I, j, sum =0;
            for (j = 0; j<N; j++)
            {
                    for (i=0; i<M; i++)
                         {
                                Sum += a[i][j];
                         }
                 }
             return sum;
}
```

Table 15: Sample codes to understand the concept of locality. Code 1 is a basic nested loop while code 2 is optimized to exploit caches.

Now, we walk through the process in which data is brought in the cache for CODE 1. Initially, the cache is empty so we have a cache miss for a[0][0]. When data is brought in the cache it is brought at the granularity at the size of a block. Assuming that the block size is 32 bytes, along with a[0][0] a[0][1], a[0][2], a[0][3] are also brought in the cache (Data is brought in the cache row wise). So until j=3 in the inner loop we experience a cache hit. This process repeats and we can say the miss rate is ¼.

Doing similar analysis for the CODE 2, Initially there is a miss for a[0][0]. So along with a[0][0] a[0][1], a[0][2] and a[0][3] are brought into the cache. But according to the loop the processor requires a[1][0] so the cache suffers a miss and a[1][0], a[1][1], a[1][2], a[1][3] are brought into the cache. During the second iteration of the inner loop i.e i=2, j=0, the processor requires a[2][0], It is not in the cache and there is a miss again. Doing similar analysis we can conclude that if the array cannot fit completely in the cache all the accesses to a[i][j] will cause a miss and the data is to be fetched from the next level of cache which has higher latency.

Thus, from the example we can see how exploiting locality, in the above case spatial locality improve performance.

3: Cache blocking: It is a technique that can help improve the temporal locality of the code [89]. The idea behind blocking is to divide the data structure in our case the image array in smaller blocks in a way that the complete block/chunk of the image array can fit into the cache and once it is brought in the cache all the processing, reading writing from/to the locations is done and then the next block is brought to the cache. When we partition loop iteration (large array into smaller blocks), the accessed array elements fit into cache size, enhancing cache reuse and eliminating cache size requirements. Loop blocking allows reuse of the arrays by transforming the loops such that the transformed loops manipulate array strips that fit into the cache. In effect, a blocked loop uses array elements in sections that are optimally sized to fit in the cache. To understand how the tiling process can help we take an example of accessing an image or a 2-d array we compare the basic nested method for accessing the image and the loop blocking technique. We also present the access pattern of the array elements.

Example code for loop blocking

Array Access without Blocking	**Array access with Blocking**

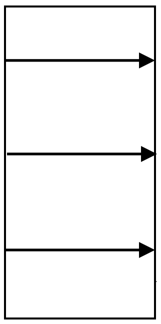
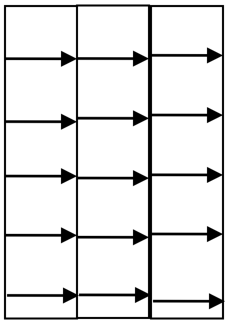
for (i=0; I < columns; i++) {
    for(j=0;j<rows; j++) {
        process array
}
    }
Access Pattern 
	for (i=0; I < columns i+=block_size) {
	    for(j=0; j< rows; j++) {
	        for(k=0; k<block_size; k++) {
	            process array
	}
	    }
	        }
	Access Pattern 

Figure 33: Code example for loop blocking and the array access patern for normal loops and loop blocking

The original loop iteration space is *columns* by *rows*. When columns and rows are too large and the cache size of the machine is too small, the accessed array elements in one loop iteration (for example, i = 1, j = 1 to rows) may cross cache lines, causing cache misses. But with blocking technique the large row is broken into a smaller chunk and can fit in the cache preventing misses. The block_size is decided based on the granularity with which the cache operates (cache line).

4: Software Pre-fetching for accesses to main memory: As describes earlier, access to the main memory is very expensive about 100s of clock cycles. So for instances where the bottlenecks are due to LLC miss it is recommended to use software instructions that can pre-fetch the data into the caches from the main memory for processing. Pre-fetching the data effectively would mask the memory latency to utilize memory bandwidth at the maximum [77]. The pre-fetch instructions are just a hint to the special prefetch hardware indicating to fetch data in advance [77]. It is recommended to use pre-fetch instructions in the following cases.

- Memory accesses are predictable.
- Time consuming inner loops.

- When there are CPU stalls for data being unavailable for processing.

More details about hardware and software prefetchers along with optimizing cache usage can be found in the Intel optimization manual [77]. Some other compiler optimization techniques like loop fusion, data alignment and data transposition can also be integrated in the code to improve cache performance.

Now, we discuss the techniques to improve DTLB overhead. The TLB is a small cache which stores a section of the page table. When the working data set is high as in case of MAD, the TLB is not able to fit all the mappings and there is a DTLB miss. To solve this problem we recommend using a larger page sizes. With a larger page size a TLB cache of the same size can keep track of larger amounts of memory, which avoids the costly TLB misses, reducing the pressure on the TLB cache.

Now, we discuss techniques to improve 4k aliasing. We have already discussed 4k aliasing in section 3.x.

4k aliasing memory aliasing occurs when the code stores to one memory location and shortly after that it loads from a different memory location with a 4-KByte offset between them.

The Intel optimization manual [77] recommends the following methods to reduce 4k aliasing.

- Align data to 32 Bytes.
- Change offsets between input and output buffers if possible.
- Use 16-Byte memory accesses on memory which is not 32-Byte aligned.

8.2 Core Bottlenecks

Now we discuss the bottlenecks that are core bound. Combing all the algorithms there are 3 different core bound bottlenecks.

1: The Floating Point unit.

2: Higher latency due to Assists.

3: Slow LEA instructions.

First, we discuss the floating point unit bottleneck and some techniques to improve performance of the floating point unit. As mentioned previously floating point operations inherently have a longer latency.

Some generic guidelines to improve performance for floating point units are:

1. Use Single precision floating point numbers in place of double precision whenever possible. Also if possible use integers. Using the single precision floating point number can be done by setting the precision control (PC) field in the x87 FPU control word to single precision [77]. This allows single precision (32-bit) computation to complete faster on some operations (for example, divides would take less clock cycle to execute).

2. The operations should remain in range. Denormal values and underflow can cause very high penalties.

3. Use fast data type conversion instructions (SSE2 or SSE3 instructions). The streaming mode SIMD instructions can save many uops. These instructions avoid changing the rounding mode of the floating point numbers. It is recommended that the data to be operated is aligned because the SIMD instructions operate on 16 byte aligned data [77].

4. Also, use of logarithmic number system can improve performance. Example: A floating point divide operation would change to a subtraction operation. Using the logarithmic number system can be found in [90]

5. Remove data dependencies.

   ```
   X=A+C;
   Y=X+D;
   Z=G+H;
   ```
   This sequence of code can be changed to

```
X=A+C;
Z=G+H;
Y=X+D;
```

Just switching the dependent instruction can eliminate stalls and improve performance. For floating point operations the improvement can be significant because the latency for individual operations is high.

6. The division operation takes the maximum number of clock cycles and from the analysis we find that there are instances where the divide unit is a bottleneck. If dividing by a constant, it is recommended to replace the divide by a product of the inverse of the constant. Also, if the denominator is an integer it is recommended to test if a right shift operation can be useful.

Now, we discuss the techniques to remove the generation slow LEA instructions and improve performance. As mentioned in section 7.4, LEA instruction is an assembly instruction. The assembly is generated during the compile time. Since, the compiler generates these instructions; one solution is to use an Intel compiler. Since the LEA instructions were generated for the index access in the loop, if the index access is reduced it will decrease the number of LEA instructions to be executed and consequently, reduce the overhead.

Now we discuss the techniques to resolve overheads due to generations of micro assists. Assists are a stream of instructions introduced in the processor to execute a function/instruction which cannot be directly executed by the processor. From the analysis we find that the bottleneck is due to the floating point assists. The code generates results which are denormals. These cannot be executed by the processor directly and therefore they are executed by inserting a stream of instructions. The execution of this stream takes more clock cycles and thus there is a bottleneck. To improve performance for such a situation, it is recommended to write the code in such a way that denormals are not generated. Another way to improve performance is to improve performance is to enable the Flush –TO-Zero (FTZ) and Denormals are Zero (DAZ) mode [91]. Since denormals are values that are very close to zero these modes approximate denormal values to zero. There is a significant improvement in performance when FTZ and DAZ modes are enabled for codes that generate denormals. Note that the FTZ and DAZ modes are only applicable for SSE instructions [91].

FTZ mode should be enabled when there is an underflow. It means that whenever the result of an operation is denormal floating point number, FTZ mode is enabled. DAZ mode is enabled when the source operands for an operation are denormals.

These modes can be enabled by masking the bits of the MXSCR register [78-79, 91]. The bits required to be masked and the corresponding modes are shown in table 16.

FTZ mode (Bit 15)	**Underflow Mask (Bit 11)**	**Operation/ Effect**
1	1	Output zero, Precision and underflow flags are set
0	1	Output precise, precision and underflow flags set.
1	0	Software Exception, Underflow flag set
0	0	Software Exception, underflow flag set
DAZ Mode (bit 6)	**Denormal mask (Bit 8)**	**Operation/ Effect**
1	1	Output zero, No flags set
0	1	Output Precise, Denormal flag set
1	0	Output Zero, No flags Set
0	0	Software Exception, Denormal flag set.

Table 16: Mask bits for FTZ and DAZ modes along with the effects or the operation that is performed by individual mask combination

Along with SSIM instructions, denormals can be handled in the X87 instructions also. The details of handling the denormals in the X87 Floating point assists along with sample code to set the FTZ and DAZ modes for both the x87 Floating point assists and the SSE floating point assists are given [91].

8.3: Custom Image Quality Assessment Hardware Engine

First we list all the major bottlenecks and discuss their characteristics to help define a generic IQA hardware engine. We start with the memory bottlenecks the most common bottleneck category and then the core/ execution unit bottlenecks. If we take an insight into these 2 categories of bottlenecks, it is said that the memory operations are "supporting operation", the memory operations only bring data into the processor for a particular operation. On the other side the core operation can be considered as "Real" operation. These are the operations where the data is actually processed.

1. L1D, L2D replacement and LLC misses: As per our discussions in the microarchitectural analysis and in the previous sections in this chapter the data set for the algorithms causing the L1D, L2D replacement and LLC miss is high. Thus, having caches or fast memories with larger size is recommended. It is also recommended to have a larger register file to save the local operands directly ready for processing. A suitable size for the caches and its configuration along with defining a memory hierarchy can be decided by performing a cache simulation with various sizes and configurations. A configuration that surpasses a predefined threshold for hit/miss rates should be developed. Creating models for a cache configuration that best suits the performance, cost and other requirements are beyond the scope of this document.
2. DTLB Overhead: As mentioned in the previous sections, DTLB is specific cache which stores a sub-set of translations from virtual memory to physical memory. For a specific IQA engine there is no requirement of a virtual memory system. Hence this eliminates the requirement of using a TLB.
3. 4k aliasing: If we take an insight into 4k aliasing, it is caused due to out of order execution of memory instructions in the processor. If we define the Hardware engine which is In-order, we eliminate the possibility of 4k aliasing.
4. Machine clears: The cause of machine clears is memory violations. These are caused because of the shared memory between processes and processors. Since, the engine is specific for IQA and not a general purpose computing platform we eliminate the performance degradation due to machine clears also.

Now we analyze the execution or core bottlenecks. These are the bottlenecks related to actual processing. If we observe the IQA algorithms in general we find that there are 3 common operations that are carried out. First, an image transform. For example, the image is transformed to wavelet domain in VSNR. Second, the filter operation. Most of the algorithms use a low pass filter and the final common function that these algorithms perform is the calculation of image statistics. So, a generic IQA engine would have an Image transform engine, a filter engine and an engine to calculate image statistic. Now we discuss the core bottlenecks to understand how they can be integrated into the custom hardware to achieve an optimal design.

1. Floating Point Unit: If we see the above three operations i.e. the Image transform, filtering, calculation of image statistics, all these operations require floating point operations and consequently a floating point execution unit. Thus we can infer that the floating point unit is a critical hardware sub-system. Since all the operations are floating point and it is a critical bottleneck, it is advisable to design a hardware which operates on a logarithmic number system. The overhead for converting to log domain and then transforming back is negligible. As mentioned, the multiply and divide operations change to add and subtract operations. The add and subtract operations are less expensive and can save many clock cycles. If cost, chip area and power can be comprised to an extent, it is advisable to have multiple floating point units to exploit parallelism. Also, it is advisable to pipeline the unit to hide/overlap the latency of the supporting instructions for getting data to the IQA execution engine.
2. Slow LEA instructions: These instructions are an outcome of the complex addressing mode of the CISC Intel architecture [78-79]. So, if the memory control hardware is design simple to be like the RISC architecture with just a load store model, the performance degradation due to these instructions is automatically eliminated. Also, the custom engine proposed is hardcoded which eliminates issues due to generation of such instructions by the compiler.
3. Micro Assists: Floating point micro assists occur because the operands or results of an operation are/is a denormal. If precision can be compromised these denormals can be directly converted to zero and if precision is required, a custom hardware just to process denormals

can be designed. A special port can be designated to dispatch the denormals to this unit. If there are no denormals this unit can work as a normal Floating point unit.

A block diagram for a general IQA hardware engine is shown in figure 34.

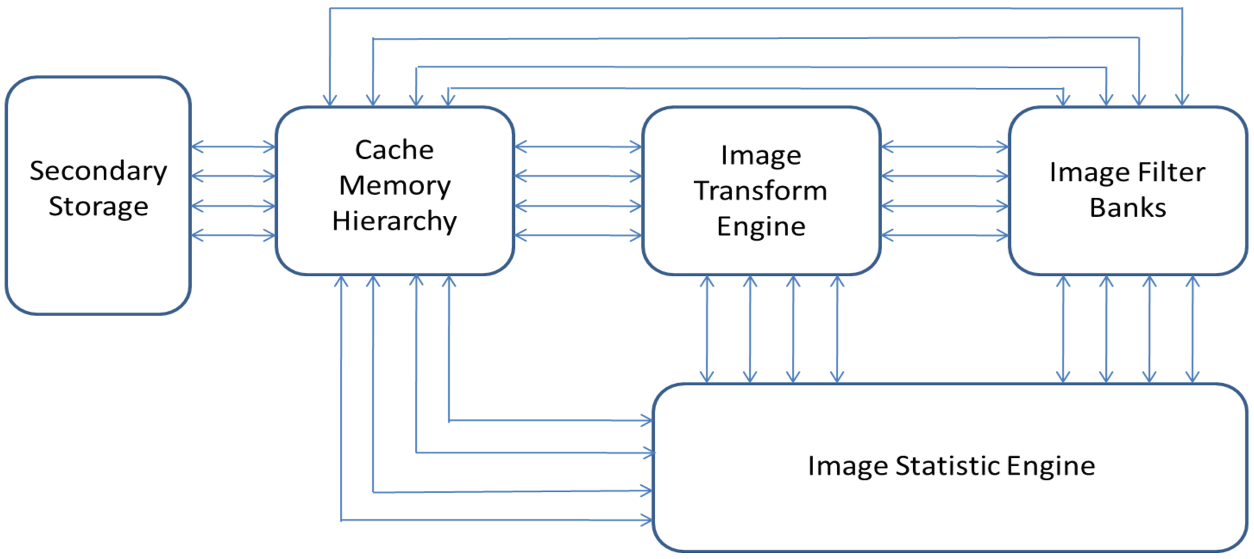


Figure 34: Blocks for custom IQA engine. It has 3 basic blocks the Image transform Engine, the filter banks and the image statistics engine.

As seen from the figure, we have a secondary storage to save a data base for the image. The images to be assessed along with the reference image (for a reference based algorithm) are brought into the fast memory the caches. From the caches the images act as operands to one or more of the three engines depending on the sequence of operations to be performed by the specific IQA algorithm. Also, there are instances where an operation is performed multiple times. Such an interconnection network helps to feed data directly to the respective engine. This also leads to reuse of the existing hardware and save chip area and cost.

The different executions engines are design as follows:

1: The transform block can be general purpose floating point unit or a transform specific custom design like a DWT unit in VSNR.

2: The filter blocks can be implemented as a general purpose filter if multiple filter banks are used or can be a specific implementation. For example, a log gabor filter unit in MAD.

3: The image statistics block similarly can be implemented as a general purpose engine or a custom engine. But if we observe the algorithms, they comprise of multiple statistical computations. Thus, creating an engine for all is not recommended. Rather a general purpose floating point unit can be used. Control signals can be generated to define which operation needs to be performs.

It is suggested to pipeline these engines. Pipelining would hide some latency for accessing memory. Also, it would improve the throughput of the hardware. More details about pipelining, tradeoffs for a pipeline and designing a pipelined hardware can be found in [80].

REFERENCES:

[1] C. J. van den Branden Lambrecht, “A working spatio-temporal model of the human visual system for image representation and quality assessment applications,” in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, May 1996, pp. 2291–2294.

[2] Z. Wang, A. C. Bovik, and L. Lu, “Wavelet-based foveated image quality measurement for region of interest image coding,” *Proc. IEEE Int. Conf. on Image Processing*, vol. 2001.

[3] Kai Yang and Hongxu Jiang, “Optimized-ssim based quantization in optical remote sensing image compression,” in *Proceedings of the 2011 Sixth International Conference on Image and Graphics*, Washington, DC, USA, 2011, ICIG ’11, pp. 117–122, IEEE Computer Society.

[4] Abdul Rehman, Mohammad Rostami, Zhou Wang, Dominique Brunet, and Edward R. Vrscay, “Ssim-inspired image restoration using sparse representation,” *EURASIP J. Adv. Sig. Proc.*, vol. 2012, pp. 16, 2012.

[5] F. Ciaramello, A. Cavender, S. Hemami, E. Riskin, and R. Ladner, “Predicting intelligibility of compressed american sign language video with objective quality metrics,” in *2006 International Workshop on Video Processing and Quality Metrics for Consumer Electronics*, 2006.

[6] J. L. Mannos and D. J. Sakrison, “The effects of a visual fidelity criterion on the encoding of image,” *IEEE Trans. Info. Theory*, vol. 20, pp. 525–535, 1974.

[7] F. Lukas and Z. Budrikis, “Picture Quality Prediction Based on a Visual Model,” *IEEE Transactions on Communications*, vol. 30, no. 7, pp. 1679–1692, 1982.

[8] N. Nill, “A Visual Model Weighted Cosine Transform for Image Compression and Quality Assessment,” *IEEE Transactions on Communications*, vol. 33, no. 6, pp. 551–557, 1985.

[9] S. Daly, “Visible differences predictor: an algorithm for the assessment of image fidelity,” in *Digital Images and Human Vision*, A. B. Watson, Ed., 1993, pp. 179–206.

[10] P. C. Teo and D. J. Heeger, “Perceptual image distortion,” *Proc. SPIE,* vol. 2179, pp. 127–141, 1994.

[11] S. J. P. Westen, R. L. Lagendijk, and J. Biemond, “Perceptual image quality based on a multiple channel HVS model,” *Intl. Conf. Acoustics, Speech, and Signal Processing*, vol. 4, pp. 2351–2354, 1995.

[12] J. Lubin, “A visual discrimination model for imaging system design and evaluation,” in *Vision Models for Target Detection and Recognition*, E. Peli, Ed., pp. 245–283. World Scientific, 1995.

[13] C. J. van den Branden Lambrecht, “A working spatio-temporal model of the human visual system for image representation and quality assessment applications,” in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing,* May 1996, pp. 2291–2294.

[14] A. B. Watson, M. Taylor, and R. Borthwick, “Image quality and entropy masking,” *Human Vision, Visual Processing, and Digital Display VIII, Proc. SPIE,* vol. 3016, pp. 2–12, 1997.

[15] Y. Lai and C. J. Kuo, “Image quality measurement using the haar wavelet,” *Proc. SPIE: Wavelet Applications in Signal and Image Processing V,* 1997.

[16] M. Miyahara, K. Kotani, and V. R. Algazi, “Objective picture quality scale (PQS) for image coding,” *IEEE Transactions on*

Communications, vol. 46, no. 9, pp. 1215–1226, 1998.

[17] W. Osberger, N. Bergmann, and A. Maeder, “An automatic image quality assessment technique incorporating higher level perceptual factors,” *Proc. IEEE Int. Conf. on Image Processing*, vol. 3, pp. 414–418, 1998.

[18] S. Winkler, “A perceptual distortion metric for digital color images,” *Proc. IEEE Int. Conf. on Image Processing*, vol. 3, pp. 399–403, 1998.

[19] A. Bradley, “A wavelet visible difference predictor,” *IEEE Trans. Image Process.*, vol. 8, pp. 717–730, May 1999.

[20] S. Winkler, “Visual quality assessment using a contrast gain control model,” in *IEEE Signal Processing Society Workshop on Multimedia Signal Processing*, September 1999, pp. 527–532.

[21] J. Lubin et al., “Method and apparatus for assessing the visibility of differences between two image sequences,” 1999, US Patent 5,974,159.

[22] P. LeCallet, A. Saadane, and D. Barba, “Frequency and spatial pooling of visual differences for still image quality assessment,” *Proc. SPIE Human Vision and Electronic Imaging V*, vol. 3959, pp. 595–603, 2000.

[23] “Jndmetrix technology,” http://www.sarnoff.com/, Sarnoff Corporation.

[24] T. N. Pappas, T. A. Michel, and R. O. Hinds, “Supra-threshold perceptual image coding,” *Proc. ICIP*, pp. 237–240, 1996.

[25] M. P. Eckert and A. P. Bradley, “Perceptual quality metrics applied to still image compression,” *Signal Processing*, vol. 70, pp. 177–200, 1998.

[26] N. Damera-Venkata, T. D. Kite, W. S. Geisler, B. L. Evans, and A. C. Bovik, “Image quality assessment based on a degradation model,” *IEEE Trans. Image Process.*, vol. 9, 2000.

[27] M. Carnec, P. Le Callet, and D. Barba, “An image quality assessment method based on perception of structural information,” in *ICIP 2003*, 2003, vol. 2, pp. 185–188.

[28] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, “Image quality assessment: From error visibility to structural similarity,” *IEEE Trans. Image Process.*, vol. 13, pp. 600–612, 2004.

[29] Z. Wang, E.P. Simoncelli, and A.C. Bovik, “Multiscale structural similarity for image quality assessment,” in *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, Nov 2003, vol. 2, pp. 1398 – 1402 Vol.2.

[30] H.R. Sheikh and A.C. Bovik, “Image information and visual quality,” *Image Processing, IEEE Transactions on*, vol. 15, no. 2, pp. 430 –444, feb. 2006.

[31] G. Zhai, W. Zhang, X. Yang, and Y. Xu, “Image quality assessment metrics based on multi-scale edge presentation,” *IEEE Workshop on Signal Processing Systems Design and Implementation*, pp. 331–336, 2005.

[32] A. Shnayderman, A. Gusev, and A. M. Eskicioglu, “An SVD-based grayscale image quality measure for local and global assessment,” *IEEE Trans. Image Processing*, vol. 15, no. 2, pp. 422–429, 2006.

[33] D. M. Chandler and S. S. Hemai, “Vsnr: A wavelet-based visual signal-to-noise ratio for natural images,” *IEEE Transactions on Image Processing*, vol. 16, no. 9, pp. 2284–2298, 2007.

[34] Eric C. Larson and Damon M. Chandler, “Most apparent distortion: full-reference image quality assessment and the role of strategy,” *Journal of Electronic Imaging*, vol. 19, no. 1, pp. 011006, 2010.

[35] H. Tang, N. Joshi, and A. Kapoor, “Learning a blind measure of perceptual image quality,” in *in International Conference on Computer Vision and Pattern Recognition*, 2011.

[36] P.Ye and D.Doermann, “No-reference image quality assessment using visual codebook,” in *in International Conference on Image Processing*, 2011.

[37] Anush Krishna Moorthy and A. C. Bovik, “Blind image quality assessment: From natural scene statistics to perceptual quality.,” *IEEE Transactions on Image Processing*, vol. 20, no. 12, pp. 3350–3364, 2011.

[38] Anish Mittal, Gautam S. Muralidhar, Joydeep Ghosh, and A. C. Bovik, "Blind image quality assessment without human training using latent quality factors," *IEEE Signal Processing Letters,* vol. 19, no. 2, pp. 75–78, 2012.

[39] Anush Krishna Moorthy and Alan Conrad Bovik, "Model-based blind image quality assessment using natural DCT statistics," *IEEE Transactions on Image Processing*, 2012, (to appear).

[40] Zhou Wang and Eero P. Simoncelli, "Reduced-reference image quality assessment using a wavelet-domain natural image statistic model," *Proc. Of SPIE Human Vision and Electronic Imaging*, vol. 5666, January 2005.

[41] Alireza Nasiri Avanki, Shahnam Sodagari, and Abolfazl Diyanat, "Reduced-reference image quality assessment metric using optimized parameterized wavelet watermarking," October 2008, pp. 26–29.

[42] K. Chono, Yao-Chung Lin, David P. Varodayan, Y. Miyamoto, and Bernd Girod, "Reduced-reference image quality assessment using distributed source coding," in *IEEE International Conference on Multimedia and Expo,* 2008.

[43] Qiang Li and Zhou Wang, "Reduced-reference image quality assessment using divisive normalization-based image representation," *IEEE Journal of Selected Topics in Signal Processing,* , no. 2, April 2009.

[44] Wu feng Xue and Xuan qin Mou, "Reduced-reference image quality assessment based on weibull statistics," *International Workshop on Quality of Multimedia Experience (QoMEX),* June 2008.

[45] Songnan Li Lin Ma, Fan Zhang, and King Ngi Ngan, "Reduced-reference image quality assessment using reorganized DCT-based image representation," *IEEE Trans. on Multimedia,* vol. 13, no. 14, August 2011.

[46] K. Seshadrinathan and A.C. Bovik, “Motion tuned spatio-temporal quality assessment of natural videos,” *IEEE Transactions on Image Processing*, vol. 19, no. 2, pp. 335 –350, Feb 2010.

[47] P.V. Vu, C.T. Vu, and D.M. Chandler, “A spatiotemporal most-apparent-distortion model for video quality assessment,” in *Image Processing (ICIP), 2011 18th IEEE International Conference on*, sept. 2011, pp. 2505 –2508.

[48] Wen-Hsiung Chen, C. Smith, and S. Fralick, “A fast computational algorithm for the discrete cosine transform,” *Communications, IEEE Transactions on*, vol. 25, no. 9, pp. 1004 – 1009, sep 1977.

[49] Hsieh Hou, “A fast recursive algorithm for computing the discrete cosine transform,” *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 35, no. 10, pp. 1455 – 1461, oct 1987.

[50] Jie Liang and Trac D. Tran, “Fast multiplierless approximation of the dct with the lifting scheme,” *IEEE Trans. on Signal Processing*, vol. 49, pp. 3032–3044, 2000.

[51] Wenjia Yuan, Pengwei Hao, and Chao Xu, “Matrix factorization for fast dct algorithms,” in *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, may 2006, vol. 3, p. III.

[52] Chao Cheng and K.K. Parhi, “Hardware efficient fast dct based on novel cyclic convolution structures,” *Signal Processing, IEEE Transactions on*, vol. 54, no. 11, pp. 4419 –4434, nov. 2006.

[53] V. Britanak, P.C. Yip, and K.R. Rao, *Discrete Cosine and Sine Transforms: General Properties, Fast Algorithms and Integer Approximations*, Academic, 2007.

[54] D.W. Trainor, J.P. Heron, and R.F. Woods, “Implementation of the 2d dct using a xilinx xc6264 fpga,” in *Signal Processing Systems, 1997. SIPS 97 - Design and Implementation., 1997 IEEE Workshop on*, nov 1997, pp. 541 –550.

[55] G. Kiryukhin and M. Celenk, “Implementation of 2d-dct on xc4000 series fpga using dft-based dsfg and da architectures,” in *Image Processing, 2001. Proceedings. 2001 International Conference on*, 2001, vol. 3, pp. 302 –305 vol.3.

[56] Bo Fang, Guobin Shen, Shipeng Li, and Huifang Chen, “Techniques for efficient dct/idct implementation on generic gpu,” in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, may 2005, pp. 1126 – 1129 Vol. 2.

[57] S. Tokdemir and S. Belkasim, “Parallel processing of dct on gpu,” in *Data Compression Conference (DCC), 2011*, march 2011, p. 479.

[58] Tien-Tsin Wong, Chi-Sing Leung, Pheng-Ann Heng, and Jianqing Wang, “Discrete wavelet transform on consumer-level graphics hardware,” *Multimedia, IEEE Transactions on*, vol. 9, no. 3, pp. 668 –673, april 2007.

[59] Christian Tenllado, Javier Setoain, Manuel Prieto, Luis Pinuel, and Francisco Tirado, “Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 299–310, 2008.

[60] J. Franco, G. Bernabe, J. Fernandez, and M.E. Acacio, “A parallel implementation of the 2d wavelet transform using cuda,” in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, feb. 2009, pp. 111 –118.

[61] M. Unser, “Fast gabor-like windowed fourier and continuous wavelet transforms,” *Signal Processing Letters, IEEE*, vol. 1, no. 5, pp. 76 –79, may 1994.

[62] Liang Tao and Hon Keung Kwan, “Fast parallel approach for 2-d dht-based real-valued discrete gabor transform,” *Image Processing, IEEE Transactions on*, vol. 18, no. 12, pp. 2790 –2796, dec. 2009.

[63] XinXin Wang and B.E. Shi, "Gpu implemention of fast gabor filters," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 30 2010-june 2 2010, pp. 373 –376.

[64] Liang Tao and Hon Keung Kwan, "Multirate-based fast parallel algorithms for 2-d dht-based real-valued discrete gabor transform," *Image Processing, IEEE Transactions on*, vol. 21, no. 7, pp. 3306 –3311, july 2012.

[65] Franklin C. Crow, "Summed-area tables for texture mapping," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 207–212, Jan. 1984.

[66] F. Shafait, D. Keysers, and T. M. Breuel, "Efficient implementation of local adaptive thresholding techniques using integral images," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, Jan. 2008, vol. 6815 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*.

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

[68] B. Gordon, S. Sohoni, and D. Chandler, "Data handling inefficiencies between cuda, 3d rendering, and system memory," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, dec. 2010, pp. 1 –10.

[69] Ming-Jun Chen and Alan C. Bovik, "Fast structural similarity index algorithm," *J. Real-Time Image Process.*, vol. 6, no. 4, pp. 281–287, Dec. 2011.

[70] Krzysztof Okarma and Przemyslaw Mazurek, "Gpgpu based estimation of the combined video quality metric," in *Image Processing and Communications Challenges 3*, Ryszard Choras, Ed., vol. 102 of *Advances in Intelligent and Soft Computing*, pp. 285–292. Springer Berlin / Heidelberg, 2011.

[71] http://software.intel.com/en-us/intel-vtune-amplifier-xe/

[72] Intel VTune Amplifier XE 2012 Help.

[73] http://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed/2

[74] John paul shen, Mikko H. Lipasti, "Modern Processor design Fundamentals of superscalar Processors", McGrawHill Higher Education, pp 10-33.

[75] http://software.intel.com/sites/default/files/article/157041/core-i7-vtune-sw-opt-guide-1.03.pdf

[76] Using Intel VTune Amplifier XE on 2nd generation Intel core family 1.01.pdf

[77] http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[78] http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html

[79] http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-2-datasheet.html

[80] John paul shen, Mikko H. Lipasti, "Modern Processor design Fundamentals of superscalar Processors", McGrawHill Higher Education, pp 206-209.

[81] Intel VTune Amplifier XE 2011 getting started guide.pdf

[82] D. M. Chandler and S. S. Hemami, "Availble online," http://foulard.ece.cornell.edu/dmc27/vsnr/vsnr.html.

[83] J. Villasenor, B. Belzer, and J. Liao, "Wavelet filter evaluation for image compression," *IEEE Trans. Image Process.*, vol. 4, pp.1053–1060, 1995.

[84] Microprocessors memory management units, Milan Milenkovic, IBM Corporation, Micro IEEE, 1990

[85] Z. Wang, E.P. Simoncelli, and A.C. Bovik, “Multiscale structural similarity for image quality assessment,” in *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, Nov 2003, vol. 2, pp. 1398 – 1402 Vol.2.

[86]http://techpubs.sgi.com/library/dynaweb_docs/0640/SGI_Developer/books/OrOn2_PfTune/sgi_html/ch06.html

[87] 754-2008 IEEE standard Binary floating point Arithmetic.

[88] http://software.intel.com/en-us/intel-compilers/

[89] Computer Systems: A programmer’s perspective, 2nd edition, Randal E. Bryant, David R. O’hallaron.

[90] Digital Arithmetic (The Morgan Kaufmann Series in Computer Architecture and Design),Milos D. Ercegovac , Tomás Lang

[91] http://software.intel.com/en-us/articles/x87-and-sse-floating-point-assists-in-ia-32-flush-to-zero-ftz-and-denormals-are-zero-daz/

Candidate for the Degree of

Master of Science

Thesis: PERFORMANCE AND MICROARCHITECTURAL ANALYSIS OF IMAGE QUALITY ASSESSMENT ALGORITHMS

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December,2012.

Completed the requirements for the Bachelor of Engineering in Electronics and Telecommunication at University of Pune, Pune, India in 2009.