Journal of
Electronic Imaging

SPIEDigitalLibrary.org/jei

Microarchitectural analysis of image quality assessment algorithms

Thien D. Phan
Siddharth K. Shah
Damon M. Chandler
Sohum Sohoni

Microarchitectural analysis of image quality assessment algorithms

Thien D. Phan,[a] Siddharth K. Shah,[b] Damon M. Chandler,[a,*] and Sohum Sohoni[c]
[a]Oklahoma State University, Laboratory of Computational Perception and Image Quality, School of Electrical and Computer Engineering, Stillwater, Oklahoma 74078
[b]Oklahoma State University, Computer Architecture Education Simulation and Research, School of Electrical and Computer Engineering, Stillwater, Oklahoma 74078
[c]Arizona State University, Department of Engineering and Computing Systems, Mesa, Arizona 85212

Abstract. Algorithms for image quality assessment (IQA) aim to predict the qualities of images in a manner that agrees with subjective quality ratings. Over the last several decades, the major impetus in IQA research has focused on improving predictive performance; very few studies have focused on analyzing and improving the runtime performance of IQA algorithms. This paper is the first to examine IQA algorithms from the perspective of their interaction with the underlying hardware and microarchitectural resources, and to perform a systematic performance analysis using state-of-the-art tools and techniques from other computing disciplines. We implemented four popular full-reference IQA algorithms (most apparent distortion, multiscale structural similarity, visual information fidelity, and visual signal-to-noise ratio) and two no-reference algorithms (blind image integrity notator using DCT statistics and blind/referenceless image spatial quality evaluator) in C++ based on the code provided by their respective authors. We then conducted a hotspot analysis to identify sections of code that were performance bottlenecks and performed microarchitectural analysis to identify the underlying causes for these bottlenecks. Despite the fact that all six algorithms share common algorithmic operations (e.g., filterbanks and statistical computations), our results revealed that different IQA algorithms overwhelm different microarchitectural resources and give rise to different types of bottlenecks. Based on these results, we propose microarchitectural-conscious coding techniques and custom hardware recommendations for performance improvement. *© 2014 SPIE and IS&T* [DOI: 10.1117/1.JEI.23.1.013030]

Keywords: image quality assessment; performance analysis; Vtune; microarchitectural analysis.

Paper 13561 received Oct. 4, 2013; revised manuscript received Jan. 7, 2014; accepted for publication Jan. 28, 2014; published online Feb. 26, 2014.

1 Introduction

Digital images are subject to various forms of processing as they are captured, transmitted, and ultimately displayed to consumers. Because such processing can change the image's appearance, it is critical to analyze the changes in order to determine the impact on the image's visual quality. To this end, numerous algorithms for image quality assessment (IQA) have been researched and developed over the last several decades. IQA algorithms aim to provide an automated means of gauging an image's visual quality in a manner that agrees with human judgments of quality. Today, IQA research has emerged as an active subdiscipline of image processing, and many of the resulting techniques and algorithms have begun to benefit a wide variety of applications ranging from image compression (e.g., Refs. 1 to 3), to denoising (e.g., Ref. 4), to gauging intelligibility in sign language video.[5]

Most IQA algorithms are so-called full-reference algorithms, which take as input a reference image and a processed (usually distorted) image, and yield as output either a scalar value denoting the overall visual quality or a spatial map denoting the local quality of each image region. More recently, researchers have begun to develop no-reference and reduced-reference algorithms, which attempt to yield the same quality estimates either by using only the processed/distorted image (no-reference IQA) or by using the processed/distorted image and only partial information about the reference image (reduced-reference IQA). See Refs. 6 to 9 for recent reviews.

All three types of IQA algorithms have been shown to perform quite well at gauging quality. Some of the best-performing full-reference algorithms, such as multiscale structural similarity (MS-SSIM),[10] visual information fidelity (VIF),[11] and most apparent distortion (MAD),[12] have been shown to generate estimates of quality that correlate highly with human ratings of quality, typically yielding Spearman and Pearson correlation coefficients in excess of 0.9. Research in no-reference and reduced-reference IQA is much less mature; however, recent methods, such as DIIVINE,[13] BLIINDS-II,[14] and BRISQUE,[15] can yield quality estimates that also correlate highly with human ratings of quality, sometimes yielding correlation coefficients that rival the full-reference methods.

Although a great deal of research on IQA has focused on improving prediction accuracy, much less research has addressed performance issues with respect to algorithmic, microarchitechtural efficiency, and program execution speed. As IQA algorithms move from the research environment into more mainstream applications, issues surrounding efficiency—such as execution speed and memory bandwidth requirements—begin to emerge as equally important

*Address all correspondence to: Damon M. Chandler, E-mail: damon.chandler@okstate.edu

0091-3286/2014/$25.00 © 2014 SPIE and IS&T

performance criteria. Many IQA algorithms that excel in terms of prediction accuracy fall short in terms of efficiency, often requiring relatively large memory footprints and runtimes on the order of seconds for even modest-sized images (e.g., <1 megapixel). As these algorithms are adapted to process frames of video (e.g., Refs. 16 and 17) or are used during optimization procedures (e.g., during research and develop optimization in a coding context), efficiency becomes of even greater importance.

From a signal-processing viewpoint, it would seem that the bulk of computation and runtime are likely to occur in two key stages, which are 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 first of these two stages can potentially require a considerable amount of computation and memory bandwidth, particularly when a large number of frequency bands are analyzed and when the decomposition must be applied to the image as a whole. The latter of these two stages would seem to require more computation, particularly when multiple statistical computations are computed for each local region of coefficients. For example, in MS-SSIM[10] an image is decomposed into different scales, and local image statistics are computed for each block of coefficients (via a sliding window). In VIF,[11] wavelet subband covariances can be computed via a block-based or overlapping block-based approach. In MAD,[12] variances, skewness, and kurtosis of log-Gabor coefficients are also computed for overlapping blocks in each subband. These approaches have been argued to mimic the cortical processing in the human visual system (HVS) in which the statistics of local responses of neurons in primary visual cortex (modeled as coefficients) are computed and compared in higher-level visual areas. Yet, unlike the HVS, most modern computing machines lack dedicated hardware for computing the coefficients and their local statistics.

Due to their extensive use in image compression and computer vision, a considerable amount of research has focused on accelerating two-dimensional (2-D) image transforms, which provide local frequency-based decompositions. For example, the discrete cosine transform (DCT) has been accelerated at the algorithm level by using variations of the same techniques used in the fast Fourier transform (FFT) (e.g., Ref. 18) and by exploiting various algebraic and structural properties of the transform, e.g., via recursion,[19] lifting,[20] matrix factorization,[21] cyclic convolution,[22] and many other techniques (see Ref. 23 for a review). Numerous techniques for hardware-based acceleration of the DCT have also been proposed using general-purpose computing on graphics processing units (GPGPU)-based and field-programmable gate array-based implementations (e.g., Refs. 24 to 27). Algorithm- and hardware-based acceleration has also been researched for the discrete wavelet transform (DWT) (e.g., Refs. 28 to 30) and Gabor transform (e.g., Refs. 31 to 34).

Techniques for accelerating the computation of local statistics in images has also been researched, though to a much lesser extent than the transforms. One technique, called integral images, which was originally developed in the context of computer graphics,[35] has emerged as a popular approach for computing block-based sums of any 2-D matrix of values (e.g., a matrix of pixels or coefficients). The integral image, also known as the summed area table, requires first computing a table that has the same dimensions as the input matrix, and in which each value in the table represents 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 for computing higher-order moments, such as the variance, skewness, and kurtosis (see, e.g., Refs. 36 and 37).

In Ref. 38, Chen and Bovik presented the fast SSIM and fast MS-SSIM algorithms, which are accelerated versions of SSIM and MS-SSIM, respectively. Three modifications were used for fast SSIM: (1) The luminance component of each block was computed by using an integral image. (2) The contrast and structure components of each block were computed based on 2×2 Roberts gradient operators. (3) The Gaussian-weighting window used in the contrast and structure components was replaced with an integer approximation. For fast MS-SSIM, a further algorithm-level modification of skipping the contrast and structure computations at the finest scale was proposed. By using these modifications, fast SSIM and fast MS-SSIM were shown to be, respectively, 2.7× and 10× faster than their original counterparts on 768×432 frames from videos of the LIVE Video Quality database.[39] Although algorithm-level modifications were used, the authors demonstrated that these modifications had negligible impact on predictive performance; testing on the LIVE Image Quality and Video Quality databases revealed effectively no impact on Spearman rank-order correlation coefficient, Pearson correlation coefficient, and root mean square error. By further implementing the calculations of the contrast and structure components via Intel SSE2 [single instruction multiple data (SIMD)] instructions, speedups of ∼5× for fast SSIM and 14× for fast MS-SSIM were reported. In addition, speedups of ∼17× for fast SSIM and 50× for fast MS-SSIM were reported by further employing parallelization via a multithreaded implementation.

In Ref. 40, Okarma and Mazurek presented GPGPU techniques for accelerating SSIM, MS-SSIM, and combined video quality metric (CVQM) (a video quality assessment algorithm developed previously by Okarma, which uses SSIM, MS-SSIM, and VIF to estimate quality). To accelerate the computation of both SSIM and MS-SSIM, the authors described a compute unified device architecture (CUDA)-based implementation in which separate GPU threads were used for computing SSIM or MS-SSIM on strategically sized fragments of the image. To overcome CUDA's memory-bandwidth limitations, the computed quality estimates for the fragments were stored in GPU registers and transferred only once to the system memory. Okarma and Mazurek reported that their GPGPU-based implementations resulted in 150× and 35× speedups of SSIM and MS-SSIM, respectively.

In Ref. 37, Phan et al. presented the results of a performance analysis and techniques for accelerating the MAD algorithm.[12] Although MAD is among the best in predictive performance, it is currently one of the slowest IQA algorithms when tested on several modern computers (Intel Core 2 and Xeon CPUs; see Ref. 37). A performance analysis revealed that the main bottleneck in MAD stemmed from its appearance-based stage, which accounted for 98% of the total runtime. Within this appearance-based stage, the computation of the local statistical differences accounted for most of the runtime, and computation of the log-Gabor

decomposition accounted for the bulk of the remainder. Phan et al. proposed and tested 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 of these modifications resulted in an ∼17× speedup over the original MAD implementation. The latter two resulted in an ∼47× speedup over the original MAD implementation.

Although these studies have successfully yielded more efficient versions of their respective algorithms, several larger questions remain unanswered, specially with respect to IQA algorithms: To what extent are the bottlenecks in IQA algorithms attributable to the decomposition and statistical computation stages versus more algorithm-specific auxiliary computations? To what extent are the bottlenecks attributable to computational complexity versus limitations in memory bandwidth? Are there generic implementation techniques or microarchitectural modifications that can be used to accelerate all or at least several IQA algorithms? The answers to these questions can 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. In this paper, we present the results of a performance analysis designed to examine, compare, and contrast the performances of four popular full-reference IQA algorithms (MS-SSIM[10] published in 2003, VIF[11] in 2006, visual signal-to-noise ratio (VSNR)[41] in 2007, and MAD[12] in 2010) and two no-reference IQA algorithms (BLIINDS-II[14] and BRISQUE[15] in 2012).

This work draws upon techniques that are standard in the field of performance analysis and software tuning. Listed below are some similar studies done in the area of multimedia applications. We take a similar approach in this paper.

In Ref. 42, Bhargava et al. evaluated the effectiveness of the ×86 multimedia extension (MMX) instructions for digital signal processing and multimedia applications using Intel's Vtune Amplifier XE profiler.[43] Their analysis showed that MMX assembly called within C programs is not an effective strategy to improve performance. They recommend comprehensive hand-coding and restructuring of programs to fully utilize MMX capabilities. They also conclude that parallel processing using the SIMD extensions puts a higher burden on the memory system. Their recommendations have guided developers and compiler writers as well as computer architects over the years.

In Ref. 44, Gordon et al. analyzed the performance of model-based video compression for a GPGPU implementation using CUDA and found that, surprisingly, the GPGPU implementation was slower than the native CPU implementation. The authors analyzed data with the help of Intel's Vtune Amplifier XE performance analyzer to gain insight into the specific reasons for the surprising results and found a high cache miss rate and heavy stalling of the load/store unit of the CPU in the GPGPU version. This led to the discovery that the GPGPU implementation was using the CPU's load store units to access system memory instead of using direct memory access.

In Ref. 45, Martinez et al. analyzed the performance of commercial multimedia workloads on Intel's Pentium 4, focusing on whether these applications make use of the 4-wide out of order superscalar pipeline. They found that the count for instruction per cycle (IPC) is very low, indicating that these applications do not utilize the underlying microarchitecture. They conclude that the low IPC was a result of branch mispredictions and data cache misses, and they recommend static code layout techniques that are aware of cache topology to maximize the utilization of data caches.

Although our examination is limited to six algorithms, we believe that this study is an important first step toward investigating broader performance-related issues in the design and application of IQA algorithms. The findings and recommendations presented in this paper apply broadly to all current-generation Intel IA-32 and Intel 64 based general-purpose computing platforms, whether laptops, servers, or desktops, even though the actual hotspot and bottleneck details might vary. Architectures that are radically different, with hardware accelerators, dedicated image processing cores (such as those found on some tablets and smart phones), and memory shared between GPUs and CPUs (such as AMD's Fusion APUs), are expected to show very different execution characteristics.

This paper is organized as follows. Section 2 provides a brief review of each of the six algorithms, including details of the code implementations and the results of the performance analysis for each algorithm. Sections 3 and 4 provide the analysis methodology and some architectural concepts. The performance analysis results are presented in Sec. 5. Section 6 compares and discusses the differences in performances of all six algorithms. General conclusions are provided in Sec. 7.

2 Algorithms

This section provides an introduction and a brief overview of all six algorithms. For each algorithm, the Basic port of the code to C++ subsection presents some techniques that we used when porting the code to C++, for example, computing FFT, calculating matrix's eigenvalues, and code optimizations. The algorithms are ordered here in terms of year of publication.

2.1 *Multiscale Structural Similarity*

The MS-SSIM was developed by Wang et al.[10] in 2003. MS-SSIM extends the original SSIM[46] algorithm by applying and combining SSIM for multiple scales, based on the argument that the correct scale depends on the viewing conditions. The SSIM algorithm is derived from a hypothesis that the HVS is highly adapted for extracting structural information. Therefore, measure of structural similarity between the reference and distorted images can be extended to estimate visual quality. The hypothesis also states that one could capture image quality with three aspects of information loss: luminance distortion, contrast distortion, and structural distortion.

2.1.1 *Overview of the algorithm's steps*

A block diagram of the MS-SSIM algorithm is shown in Fig. 1. The algorithm is implemented with five scales, in which the reference and distorted images serve as the first

scale. To obtain the other four scales, a low-pass filter, LPF_1, of size 2×2 pixels and a downsampling by a factor of two are applied repeatedly. For each scale, a low-pass filter, LPF_2, of size 11×11 is applied to prevent artifacts from the discontinuous truncation of the image. The luminance, contrast, and structure are computed and compared to yield a different map for each scale. This map is then combined across scales and collapsed to obtain the MS-SSIM quality index.

Specifically, the luminance comparison between two scales is derived from the means of scales' pixel values. The contrast comparison is calculated from the variances, and the structure comparison is computed from both the variances and covariance of the two scales. The single-scale similarity between the original and distorted scales is then calculated from the product of these three comparisons.

Finally, the MS-SSIM index is combined from a weighted geometric mean of contrast and structure comparisons of all five scales and luminance comparison for the last scale. These weights are used to adjust the relative importance of different components.

2.1.2 *Basic port of the code to C++*

We implemented MS-SSIM in C++ by porting its MATLAB® implementation, which is publicly available from the authors of the algorithm. The input images are loaded into one-dimensional (1-D) double arrays and are accessed as 2-D matrices via a thin C++ wrapper class.[41] The filter LPF_2 was of size 11×11 in MATLAB® version; in our C++ version, we convolve the image twice with two length 11 1-D filters. By taking advantage of this separable convolution, we reduced the number of multiplications for one 512×512 image from $512 \times 512 \times 11 \times 11$ to $512 \times 512 \times 11 \times 2$.

2.2 *Visual Information Fidelity*

The VIF was developed by Sheikh and Bovik[11] in 2006. Using natural scene statistic models, VIF quantifies the loss of image information due to the distortion process by considering the relationship among image information, the amount of information shared between a reference and a distorted image, and visual quality. Specifically, VIF quantifies the information content of the reference image as being the mutual information between the input and output of a modeled HVS channel; this is the information that the brain could ideally extract. Using a similar modeled HVS channel, VIF measures information that the brain would ideally extract from the distorted image. These two information measures are then combined to form the VIF index that correlates with the visual quality.

2.2.1 *Overview of the algorithm's steps*

A block diagram of the VIF algorithm is shown in Fig. 2. First, VIF filters the input images using the Steerable Pyramid[47] to model the image information in wavelet domain. In this step, the Steerable Pyramid is employed with four scales and six orientations, but only eight subbands of interest are used later.

In the second step, the subbands of the reference image are modeled using a Gaussian scale mixtures model. Each subband is modeled as one random field (RF), which is a product of two independent RFs. The first RF is a positive scalar, and the second RF is a Gaussian vector with zero-mean and a covariance matrix. For the distorted image, the same idea is applied: signal attenuation by a deterministic scalar gain field, and a stationary additive zero-mean Gaussian noise RF in the same wavelet domain.

In the next step, in order to calculate the reference and distorted image information, VIF also models HVS noises for two channels as two stationary RFs with zero-means and same covariance, which are uncorrelated multivariate Gaussians with the same dimensionality as the reference image. The image information for each subband is calculated with this noise.

Finally, the image information is summed over eight subbands, and the VIF index is given by the ratio between the distorted image information and reference image information.

2.2.2 *Basic port of the code to C++*

We implemented a C++ version of the VIF algorithm by porting the code from its MATLAB® implementation, which is publicly available from the authors of VIF at Ref. 48. The input images are loaded into 1-D double arrays and are accessed as 2-D matrices via a thin C++ wrapper class.[41] The original VIF algorithm uses the Steerable Pyramid toolbox[47] in MATLAB®; when ported to C++, we used the Steerable Pyramid C library by the same author. Originally, the Steerable Pyramid library with six orientations and four

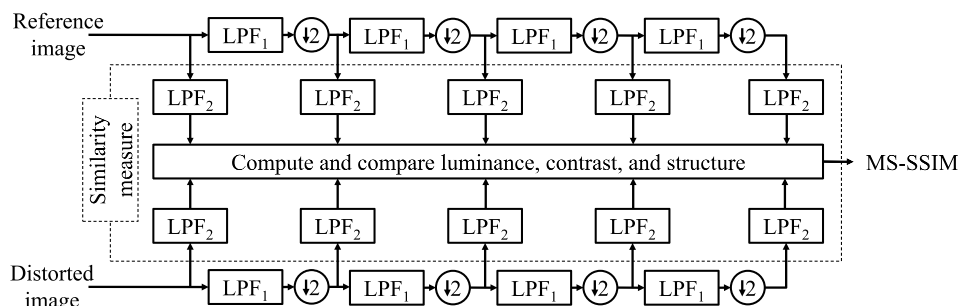

Fig. 1 Diagram of the multiscale structural similarity (MS-SSIM) algorithm. LPF_1 is a low-pass filter of size 2×2. $\downarrow 2$ is a downsampling by a factor of two. LPF_2 is a low-pass filter of size 11×11. The reference and distorted images serve as the first scale. The other four scales are obtained by applying LPF_1 and $\downarrow 2$ repeatedly. For each scale, the similarity between two images is measured by applying LPF_2 to prevent artifacts. Finally, the MS-SSIM index is formed via a combination of the luminance, contrast, and structure comparisons from different scales.

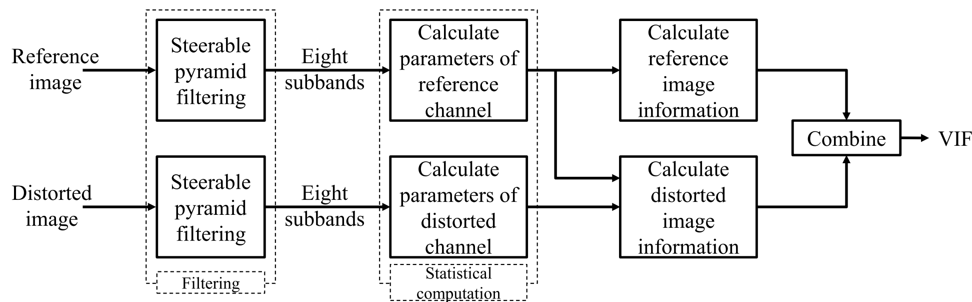

Fig. 2 The block diagram of our implementation of visual information fidelity (VIF) algorithm. First, two input images are filtered via a six-orientation and four-level Steerable Pyramid, which is modified to yield eight subbands for faster computation. The parameters of reference and distorted channels are calculated from the filtered images. Finally, the information of reference and distorted images are calculated and collapsed into a VIF index.

scales applied the filter 24 times. The VIF algorithm, however, needs only eight subbands; therefore, we modified the library to generate only these eight subbands. This modification reduced the number calls to the filtering function from 24 to 8. To calculate the eigenvalues of covariance matrix, we employed the Newmat C++ matrix library.[49]

2.3 *Visual Signal-to-Noise Ratio*

The VSNR was developed by Chandler and Hemami[41] in 2007. VSNR estimates the visual perception of distortions in natural images based on the root-mean-squared (RMS) contrast by computing the contrast thresholds for detection of distortions, the perceived contrast of the distortions, and the degree to which the distortions disrupt global precedence and, thereby, degrade the image's structure.

2.3.1 *Overview of the algorithm's steps*

A block diagram of the VSNR algorithm is shown in Fig. 3. VSNR is computed via three main steps: a DWT (via filtering with the 9/7 DWT filters), a statistical calculation to compute the perceived contrast, and a final computation for the disruption of global precedence.

In the first step, the reference image is filtered via a five-level 2-D DWT to generate 16 subbands. An error image, obtained by subtracting the two input images, is also passed through a 2-D DWT transform with the same number of levels. VSNR takes into account the viewing conditions by transforming images into luminance domain via a black-level offset, the pixel-value-to-voltage scaling factor, and a power to the gamma of the display monitor. A set of spatial

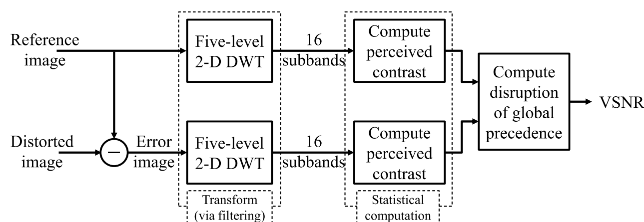

Fig. 3 The diagram of visual signal-to-noise ratio (VSNR) algorithm. First two input images are subtracted to generate an error image. The reference and error images are then filtered via a five-scale two-dimensional (2-D) discrete wavelet transform. Each set of filtered subbands is employed to calculate the perceived contrast. Finally, the VSNR is obtained by computing the disruption of global precedence.

frequencies, $\mathbf{f} = [f_1, f_2, \ldots, f_5]$, is used to describe the radial frequency content of visual stimuli expressed in units of cycles per degree of visual angle (cycles/deg), given by $f_m = 2^{-m} rv \tan(\pi/180)$ with $m = 1, 2, \ldots, 5$; r and v denote the resolution of the display and the viewing distance, respectively. This vector of spatial frequencies is also computed in this first step.

In the next step, the two sets of DWT subbands and the vector of spatial frequencies are employed to assess the detectability of the distortions. The perceived contrast of each image is computed and the contrast thresholds are calculated within each band centered at f_m to determine whether the distortions in the distorted image are visible. The contrasts at each level are calculated from the standard deviations of the three oriented subbands (LH, HL, and HH).

At this point, if the distortions are below the threshold of visual detection for all subbands, the distortions are not visible, and therefore, the distorted image is deemed to be of perfect visual quality. If the distortions are suprathreshold, the last step is performed. In this last step, the visual distortion is calculated from a weighted geometric mean of total RMS distortion contrast and a measure of the disruption of global precedence. Finally, the VSNR quality estimate is given as the log of the ratio of the RMS contrast of the reference image and the visual distortion.

2.3.2 *Basic port of the code to C++*

The original C++ code of VSNR was obtained from the author's website.[41] The reference image and distorted image are loaded into a 1-D flоad array, and it is accessed as 2-D matrix via a thin C++ wrapper class.[41] The five-level 2-D DWT decomposition step is implemented based on the lifting scheme (fast DWT[50]) using the default Cohen-Daubechies-Feauveau 9/7 wavelet. In the statistical computation step, in order to obtain the image perceived contrasts, we need to calculate the average luminance of the image, which requires calling power, multiplication, and addition operations for all pixels. We modified this part by using a look-up table technique to obtain a faster implementation that uses those operations only 256 times.

2.4 *Most Apparent Distortion*

The MAD algorithm was developed by Larson and Chandler[12] in 2010. MAD uses two strategies to estimate image quality. First, a detection-based strategy is used for

near-threshold distortions. In this case, the image is most apparent, and thus, the HVS attempts to look past the image and look for the distortions. Second, an appearance-based strategy is used for clearly visible distortions. In this case, the distortions are most apparent, and thus, the HVS attempts to look past the distortion and look for the image's subject matter.

2.4.1 *Overview of the algorithm's steps*

A block diagram of the MAD algorithm is shown in Fig. 4. From the input images, the MAD index is computed via two main stages: the detection-based stage and appearance-based stage. Each stage consists of two basic steps: filtering and statistical computations. These two stages yield two quantities indicating the quality of each stage: a detection-based index and an appearance-based index. These two indices are then combined to obtain the overall quality of the distorted image.

In the detection-based stage, a model of local visual masking, which takes into account the contrast sensitivity function (CSF), and luminance and contrast masking, is employed. The two input images are first passed through a contrast sensitivity function filter[51] in the frequency domain using FFT and inverse FFT. Two local contrast maps of the two filtered images are computed in an overlapping block-based fashion with blocks of size 16×16 and a 12-pixel overlap between neighboring blocks. From these two local contrast maps, the visible distortion map is calculated locally for the regions that are deemed to be visibly distorted. This map is then collapsed via mean squared error (MSE) measure to obtain a detection-based index.

In the appearance-based stage, the appearance-based difference map is computed from the difference of low-level statistics (mean, variance, skewness, and kurtosis) for all local blocks of the log-Gabor filtered images (subbands). First, the input images are filtered via a log-Gabor filter bank with five scales and four orientations to obtain 20 subbands. The steps to compute those subbands include computing the image's FFT, a product of this image's FFT with a set of 2-D frequency responses, and an inverse FFT. Each pair of two sets of 20 reference and distorted subbands is then divided into small blocks, each of size 16×16 (and 12 pixels of overlap between neighboring blocks). The standard deviation, skewness, and kurtosis of each block is calculated and compared to generate the statistical difference maps for each scale/orientation. The 20 statistical difference maps are then combined via a weighted mean across scales and collapsed via a 2-norm to obtain the appearance-based index.

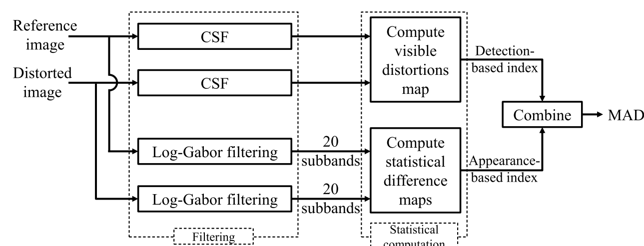

Fig. 4 The diagram of most apparent distortion (MAD) algorithm. For detection-based stage, reference and distorted images are first filtered using a contrast sensitivity function. The distortion map is then computed from filtered images and collapsed via a MSE measure to obtain a detection-based index. For the appearance-based stage, both images first are filtered using log-Gabor with five scales and four orientations. The statistical difference map is computed from the 20 filtered subbands and then collapsed into an appearance-based index. Finally, the MAD index is given by taking a weighted geometric mean of the appearance-based index and detection-based index.

Finally, the overall quality of the distorted image is computed by taking a weighted geometric mean of the detection-based index and the appearance-based index, where the weight is chosen based on the detection-based index.

2.4.2 *Basic port of the code to C++*

We implemented a C++ version of the MAD code by porting from its MATLAB® version, which is publicly available to download from the authors of MAD at Ref. 52. The input images are loaded into 1-D double arrays and are accessed as 2-D matrices via a thin C++ wrapper class.[41] In the detection-based stage, the images are transformed to the luminance domain by using a look-up table. The Ooura FFT library[53] is employed for calculating FFT and inverse FFT. This Ooura library is also used in the log-Gabor decomposition in the appearance-based stage. The log-Gabor filter was implemented based on Kovesi's work.[54] The statistical difference maps are calculated using integral images for higher orders; details of these modifications can be found in Ref. 37.

2.5 *Blind Image Integrity Notator using DCT Statistics*

The BLIINDS-II algorithm[14] was developed by Saad et al.[14] in 2012. BLIINDS-II is a no-reference IQA algorithm using DCT statistics. It inherits the idea from the BLIINDS-I algorithm[55] that the data histograms of specific-domain-transformed natural images share the same shape. One such domain is local DCT, which utilizes the generalized natural scene statistic based model. The model's parameters are transformed into features, and the generalized probabilistic model is then applied to these features to predict the visual quality of the input distorted image. The BLIINDS-II algorithm is trained using features derived directly from a generalized parametric statistical model of natural image DCT coefficients against various perceptual levels of image distortion.

2.5.1 *Overview of the algorithm's steps*

A block diagram of the BLIINDS-II algorithm is shown in Fig. 5. The algorithm is a multiscale algorithm, similar to the MS-SSIM algorithm. The input image, considered as the first scale, is low-pass filtered and downsampled twice to obtain two more scales. Each scale is then passed through the same procedure: first a block-based DCT, a generalized Gaussian modeling, and then extraction of features. Finally, all features from the three scales are combined to create a BLIINDS index indicating the quality of the distorted image.

In the first stage of the algorithm, all three scales of the input image are divided into small blocks of size 5×5 with two pixels of overlap between neighboring blocks. Each block is then subjected to a 2-D DCT.

In the second stage, a generalized Gaussian model is applied to each block of 25 DCT coefficients, as well as for specific partitions within each block. Here, the DCT histogram is fitted with a multivariate Gaussian model to extract parameters via a line search procedure.

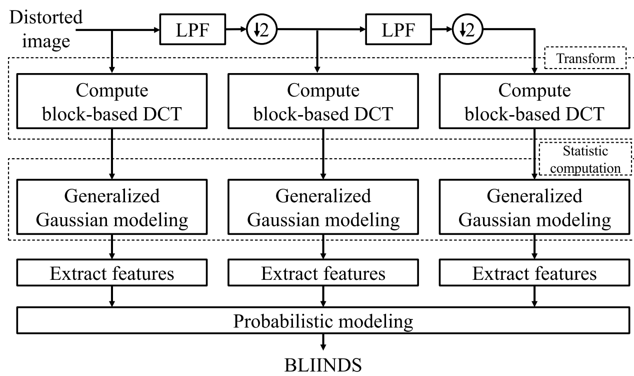

Fig. 5 BLIINDS-II algorithm. LPF is a low-pass filter of size 3×3. $\downarrow 2$ is a downsampling by a factor of two. The input image serves as the first scale. Two more scales are obtained by the low-pass filter and downsampling. Each of three scales is divided into blocks of size 5×5 so that the discrete cosine transform (DCT) can be applied for each block. The transformed coefficients are then modeled using generalized Gaussian to extract features. Finally, a probabilistic modeling is applied to yield the BLIINDS index.

Next, in the feature-extraction stage, eight features are derived from the model parameters of all blocks. All parameters are pooled in two ways: first, a percentile pooling that takes the average of the lowest (or highest) 10th percentile, and second, the ordinary sample mean, which is the average of the 100th percentile. Taking both the averages of 10th and 100th percentiles allows the algorithm to determine if the distortions are uniform over space.

The final step employs a probabilistic modeling, where the BLIINDS index (the score for the image quality) is computed by using a simple Bayesian model from a trained parameters set. This model is applied to the 24 features extracted from three scales.

2.5.2 *Basic port of the code to C++*

The MATLAB® code was obtained from the first author of BLIINDS-II via email in early 2012, and it has been confirmed to be the latest version available. We ported this MATLAB® code to C++. As shown in Fig. 5, the distorted image is loaded into a 1-D double array, and it is accessed as 2-D matrix via a thin C++ wrapper class.[41] The low-pass filter step was optimized by using a separable convolution; we convolve the image with two 1-D kernels separately, each of size 3×1. The 2-D DCT for blocks of size 5×5 was also optimized by using a look-up table for 25 values of the cosine function. In the generalized Gaussian modeling step, some functions are called repeatedly for each small block. In our C++ code, these functions are precalculated out of the main loop. This step includes a fitting process of the DCT data histogram to the model as a line search procedure over 9970 values in the range of [0–10]. In the feature-extraction step, the algorithm needs to sort values to determine 10th and 100th percentiles. This sorting procedure is performed via a quick sort algorithm.

2.6 *Blind/Referenceless Image Spatial Quality Evaluator*

The BRISQUE algorithm[15] was developed by Mittal et al. in 2012. In contrast to BLIINDS-II, which operates in the DCT domain, BRISQUE claims that in the spatial domain, natural images share the same properties. The mean subtracted contrast normalized (MSCN) coefficients of the image and the pairwise products of neighboring MSCN coefficients have the histograms of Gaussian-like appearances. These histograms distribute vary as a function of distortion. The histograms of natural images have the bell shape, while the histograms of distorted images could be of any shape, e.g., Laplacian distributions for blurred images and unusual tails for white-noise images. The generalized Gaussian distribution (GGD) and asymmetric generalized Gaussian distribution (AGGD) models are used for quantifying the features from shape, variance, left variance, and right variance of a histogram. From these features, the final quality score is given via a trained mapping by using a support vector machine (SVM) regressor.

2.6.1 *Overview of the algorithm's steps*

A block diagram of the BRISQUE algorithm is shown in Fig. 6. Similar to previous approaches, this algorithm utilizes two scales by first downsampling the input image to obtain the second scale. The following three stages are then applied.

In the first stage, the locally normalized luminances are computed via local mean subtraction and divisive normalization. This step mainly contains a filtering process of the image and its divisive normalization by a size 7×7 2-D circularly symmetric weighted Gaussian filter. The statistical relationships between neighboring pixels are also modeled in this stage. Specifically, four orientations, horizontal (H), vertical (V), main-diagonal (D1), and secondary-diagonal (D2) of MSCN coefficients are computed and multiplied point-by-point with the MSCN coefficients.

In the second stage, the GGD model is applied to calculate the shape and variance (feature 1 and feature 2 for first scale, feature 19 and 20 for second scale) from histograms of MSCN coefficients. The AGGD model is employed to calculate the shape, mean, left variance, and right variance from histograms of each of the four pairwise products (each scale has 16 features, four features for four products).

In the final stage, all 36 features (two scales, each scale has two features for GGD fitting and four features for AGGD fitting of each orientation) are collapsed into one index for

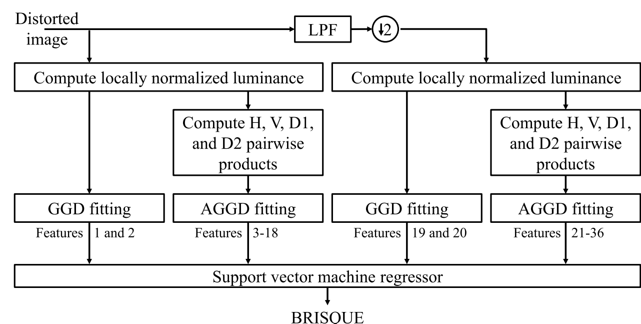

Fig. 6 BRISQUE algorithm. LPF is a low-pass filter and $\downarrow 2$ is a downsampling by a factor of two; they are utilized to obtain a smaller scale of the input image, which serves as the first scale. Each of two scales is employed to compute locally normalized luminance via local mean subtraction and divisive normalization. The luminances and their pairwise products of neighboring mean subtracted contrast normalized (MSCN) coefficients along four orientations (H, V, D1, and D2) are fitted with generalized Gaussian distribution (GGD) and asymmetric GGD (AGGD) models to extract 36 features. Finally, the support vector machine regressor is applied to yield the BRISQUE quality index.

the score of image quality via a regression module. Here, the BRISQUE algorithm is trained to use the SVM regressor to map the 36 features to a quality score.

2.6.2 *Basic port of the code to C++*

We implemented a C++ version of the BRISQUE code by porting from its MATLAB® version, which is publicly available from the authors' website.[39] The input image is loaded into 1-D double array and is accessed as a 2-D matrix via a thin C++ wrapper class.[41] In the first stage, the Gaussian filter was optimized by using a separable convolution; we convolve the image with two 1-D kernels separately, each of size 7×1. In the second stage, the *GGD fitting* and *AGGD fitting* functions contain a fitting process of data histograms to the model as a line search procedure over 9801 gamma values. In our program, the gamma values were precalculated and stored for faster speed. In the last stage, the SVM regression module, we used the same LIBSVM executable[56] as in the MATLAB® version.

3 Analysis Methodology

3.1 *Algorithms and Profiler*

We define an experimental framework for performance analysis designed to examine, compare, and contrast the performances of the six algorithms on a typical general-purpose computing platform. To provide a common codebase, we implemented five algorithms in C++ based on the original MATLAB® code provided by the authors. The C++ implementation for VSNR was directly available from its author and further optimized as described in Sec. 2.3.2. An initial code-level profiling was performed in both MATLAB® and Intel's Vtune Amplifier XE[43] to identify and correct obvious inefficiencies in the baseline implementations. For performance profiling, we first used Intel's Vtune Amplifier XE to identify segments of the program where most of the execution time was spent. Such sections of the program are called hotspots, and they should be targeted for improving the computation performance. After the top hotspot functions were identified, we conducted a microarchitectural analysis to observe the interactions between the hotspot functions and the processor and other microarchitectural subsystems. Our specific goal was to find architectural bottlenecks and map them to specific execution blocks of the respective algorithm.

3.2 *Sample Images*

To get the results shown in this paper, we executed 30 trials of each of the six algorithms on a set of 42 images. These images were taken from the CSIQ database,[12] including seven different original images (I1 to I7), each with three different distortion types, additive Gaussian white noise (AWGN), Gaussian blurring, and JPEG compression, with two levels of distortion each, level 1 for low-distorted images (AWGN1, BLUR1, and JPEG1) and level 5 for highly distorted images (AWGN5, BLUR5, and JPEG5). The original images span a variety of commonplace subject matters in five categories, animals, landscapes, people, plants, and urban. They are shown in Fig. 7. Nine of the highly distorted versions (AWGN5, BLUR5, and JPEG5) of I2, I3, and I7 are shown in Fig. 8. A summary of the experiment images is provided in Table 1.

3.3 *Analysis Platform*

For this study, we use the second-generation Intel Core i5-2430M processor clocked at 2.4 GHz and a system memory (RAM) of 4 GB. The microarchitecture was Sandy Bridge. Further details about the caches and memory hierarchy are provided in Table 2.

4 Architectural Concepts

This section provides a brief explanation of the architectural concepts, which apply to our results presented in Sec. 5.

4.1 *Virtual Memory*

Virtual memory is an abstraction provided by operating systems to the programmer, so that the programmers do not need

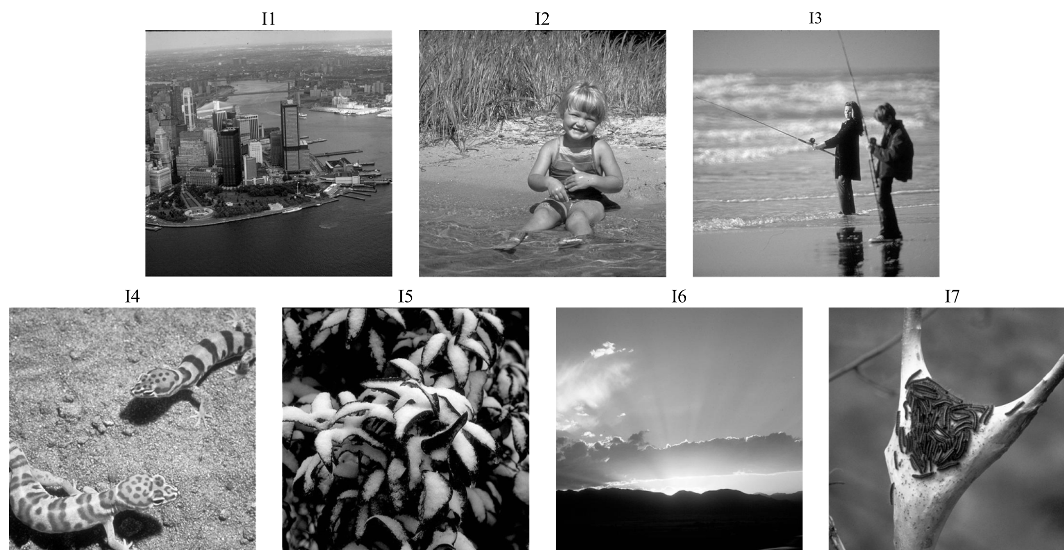

Fig. 7 Seven original images span a variety of commonplace subject matters in five categories: animals, landscapes, people, plants, and urban.

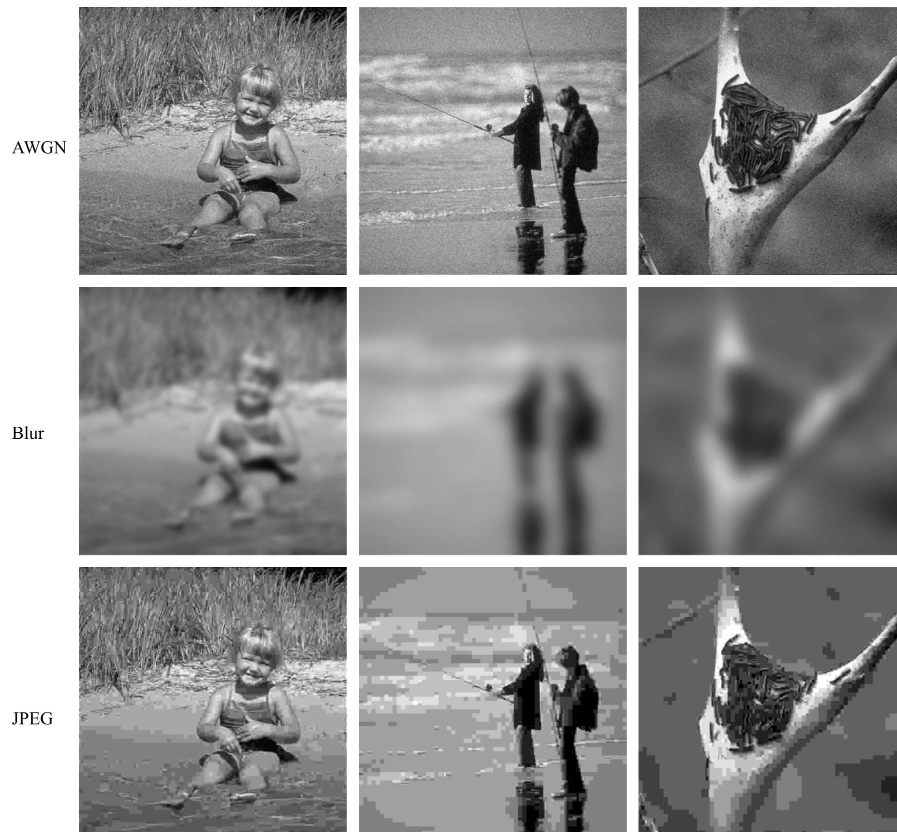

Fig. 8 Some distorted versions (AWGN5, BLUR5, and JPEG5) of I2, I3, and I7. Original images are shown in Fig. 7.

to worry about size constraints and the layout of blocks in the actual physical system memory. Without virtual addressing, the programmer would have 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. Thus, all addresses generated by the program are virtual addresses spanning the entire 32-bit or 64-bit address space and need to be translated to the actual physical locations held by the data. A mapping table called the page table handles this process. Pages are typically 4 KB in size, and they facilitate the virtual to physical mapping of addresses because a single entry in the mapping is required for all the addresses in the 4 KB range instead of individual mappings for each address. The page table is stored in the main system memory, which takes hundreds of clock cycles to access. To be able to translate faster so as to be more compatible with the speed of the processor, a cache (faster and smaller memory) is used to store those translations that are currently in use. This cache is called the translation look-aside buffer (TLB), and there is usually a separate TLB for the program instructions (ITLB) and data (DTLB).

Table 1 A summary of the experiment images.

Number of images varying in content	7
Types of distortions	Additive Gaussian white noise, blurring, JPEG compression
Levels of distortions	2 (level 1, level 5)
Total subject images	42
Image size	512×512

Table 2 Processor and system hardware specifications for the experiment.

Processor	Intel core i5-2430M
Frequency	2.4 GHz
Microarchitecture	Sandy Bridge
System memory (RAM)	4 GB
L1 instruction cache	32 KB per core
L1 data cache	32 KB per core
L2 cache (unified instruction and data)	256 KB per core
L3 cache (unified instruction and data)	3 MB shared

As the TLB stores only a subset of the page table, there are times when the mapping is not found in the TLB. This is called a TLB miss. On a TLB miss, the page table entry has to be brought into the TLB from the main memory. This transfer can take hundreds of clock cycles, and thus, frequent TLB misses can lead to a performance loss.

4.2 *CPU Caches*

The main system memory (RAM) is fairly slow and takes hundreds of clock cycles to deliver operands to the processor. Therefore, 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 the level-1 (L1) cache; it is the fastest and smallest cache in the hierarchy. Usually, the L1 is split into an instruction cache (I-cache) to store the program's instructions and the data cache (D-cache) for the operands. The next level of the cache memory is bigger and slower, and is called the level-2 (L2) cache. The level-3 (L3) cache is the last-level cache (LLC) in the hardware architecture for our analysis platform. More details about cache memories and memory hierarchy can be found in Ref. 57.

4.3 *Address Calculation*

The load effective address (LEA) instruction is an assembly instruction that calculates the effective address of memory operand and places it in a CPU register. Modern superscalar processors have multiple dispatch ports to dispactch instructions to execution units. LEA instructions with two operands can be dispatched through port 1 and port 5 on the Intel Sandy Bridge microarchitecture. LEA instructions with three operands have a longer latency of three clock cycles and can only be dispatched through port 1. Thus, a large number of back-to-back three operan LEA instruction will cause a performance bottleneck. There are some other special situations in which the LEA instructions can take three clock cycles to execute even with two operands. Details about these cases can be found in Ref. 58.

4.4 *Speculative Loads*

Modern processors internally execute instructions out of the correct program, in order to use the hardware more efficiently, and then commit or retire the instructions in the correct program order. In this process, instructions that load data from the main memory to the CPU are often given a higher priority because the data that they load are to be used in subsequent instructions. The speculative load has to be compared with any pending instructions that might be storing data in the same address. Because this operation could require several comparisons, instead of comparing the entire 32-bit or 64-bit address, typically, only the last 12 bits are compared. If these addresses are 4 KB or multiple of 4096 bytes apart, a false hazard is detected. This is called 4K aliasing,[58] and the speculative load has to be reissued in its correct program order. This reduces the throughput of instructions and creates a performance slowdown. These false positives from 12-bit comparisons can also cause machine clears.

5 Results

In this section, we provide results of the microarchitectural analysis for each of the six algorithms. The algorithms' results are presented here in terms of alphabetical order for full-reference algorithms, followed by BLIINDS-II and BRISQUE.

5.1 *Performance Analysis of MAD*

To perform hotspot analysis, the MAD algorithm was applied for 30 iterations for each image. The results of the hotspot analysis are provided in Table 3. These results show that the average execution time for all 42 images for MAD is ∼41.97 s. The average execution time for top hotspot functions/blocks is provided in second column with the standard deviation. The hotspot functions are listed in descending order, with the function consuming the highest execution time listed first. The table shows the average execution time for individual functions calculated across the 42 images. We also plot the individual execution time for all 42 images in Fig. 9.

As shown in Table 3, the top hotspot functions contribute ∼48.62% of the total execution time, and the other functions add up to the remaining 52.38%. MAD has minimal variation of total execution time across different image content as well as different distortion types. Thus, any optimizations for MAD can be made without any specific consideration of image content or distortion.

Also shown in Table 3 are hardware bottlenecks (for each hotspot function) identified via the microarchitectural

Table 3 Analysis results of most apparent distortion. Average execution time for top hotspots functions/blocks of 42 images is presented with the standard deviation. The total execution for each hotspot is calculated from the average. The hardware bottlenecks are also provided.

Function/block	Execution time (s)	Total execution (%)	Hardware bottlenecks
Gabor convolution	10.19 ± 0.31	24.29	Data translation lookaside buffer (DTLB) overhead
Appearance-based statistical difference map computation	4.06 ± 0.25	9.67	DTLB overhead
Detection-based visible distortion map computation	2.83 ± 0.10	7.13	Last-level cache (LLC) misses
Fast Fourier transform	2.91 ± 0.19	6.52	L1D replacement, L2D replacement, LLC misses
Other	21.98 ± 0.44	52.38	N/A
All	41.97 ± 0.48	100	N/A

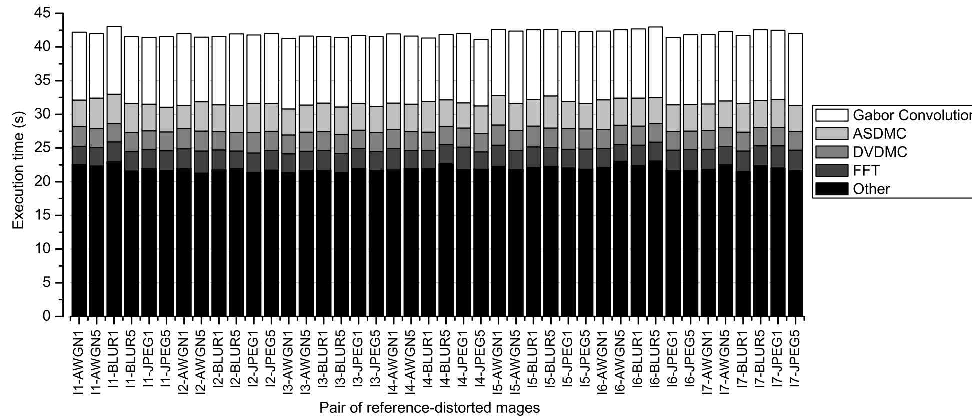

Fig. 9 The execution time of MAD for each pair of reference and distorted images. The contributions of hotspot functions are stacked together to form the total execution time.

analysis. The following subsections describe details of the observed results for each of the hotspot functions and include explanations of underlying computer architecture concepts whenever necessary.

5.1.1 *Gabor convolution*

The Gabor convolution function is used to decompose input images with five scales and four orientations to yield a set of 20 subbands. This function is called by the log-Gabor filtering block [as shown in Fig. 10(c)]. This block includes three main functions: FFT, Gabor convolution, and inverse FFT. The function takes ∼10.19 s, which is 24.29% of the total execution time.

By investigating the microarchitectural resources utilized by the Gabor convolution function, we find that the performance bottleneck is in the memory subsystem, specifically with DTLB overheads. The results show that the Gabor convolution function has a high DTLB overhead. The function generates a set of 40 filtered images (subbands), which is $40 \times 512 \times 512 \times 8$ bytes (80 MB). With a typical page size of 4 KB, this set spans over 20 thousand pages, with each page requiring its own entry in the TLB for translation. The hardware architecture of our analysis platform has 64 entries in the level-1 DTLB and 512 entries in the level-2 DTLB. Thus, the 20,000 translations required for the 40 subbands cause a large number of misses, each of which takes hundreds of clock cycles to service. One technique to overcome the problem of TLB overhead is to use superpages. Details on using superpages along with other techniques to reduce penalties due to TLB overhead are discussed in Sec. 5.

5.1.2 *Appearance-based statistical difference map computation*

The appearance-based statistical difference map computation (ASDMC) function calculates the statistical difference map using variance, skewness, and kurtosis of the subbands. This function takes two sets of 20 subbands as the inputs. For each pair of subbands of the same scale and orientation, it calculates the local standard deviation, skewness, and kurtosis

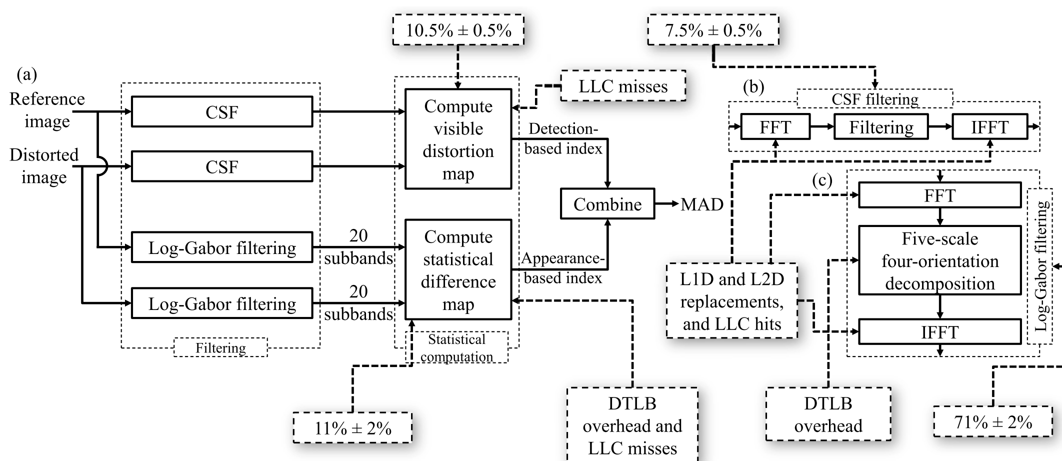

Fig. 10 Mapping between hotspots/hardware bottlenecks and the algorithmic blocks for MAD. (b) The detail of contrast sensitivity function block. (c) The detail of log-Gabor filtering block. Log-Gabor block suffers from L1D and L2D replacements, last-level cache (LLC) misses, and data translation lookaside buffer (DTLB) overhead. Fast Fourier transform (FFT) and inverse FFT blocks suffer from L1D and L2D replacement, and LLC misses; the appearance-based statistical difference map computation and detection-based visible distortion map computation functions are a part of statistical computation block, which suffer from DTLB overhead and LLC misses.

difference maps. The combined statistical difference map is then collapsed into an appearance-based index. This function takes 4.06 s, which is 9.67% of the total execution time.

The ASDMC function, similar to the Gabor convolution function, suffers from bottlenecks in the memory subsystem and faces penalties due to DTLB overhead. As with the Gabor convolution, the performance penalty is also due to the traversal of a large memory space, as the algorithm calculates statistics (standard deviation, skewness, and kurtosis) for all the 40 subbands.

5.1.3 *Detection-based visible distortion map computation*

The detection-based visible distortion map computation (DVDMC) function calculates the detection-based map and then collapses it into a detection-based index. 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 via the FFT to the reference and the error image. The function takes 2.83 s, which is 7.13% of the total execution time.

The DVDMC function suffers from LLC misses, which face a high performance penalty because they are serviced from the main memory. DVDMC processes input images, luminance images, and matrices in the Fourier domain. This huge data set cannot fit in the caches, and consequently, the function suffers from a large number of LLC misses. The data have to be fetched from the main memory, which causes a slowdown.

5.1.4 *Fast Fourier transform*

The FFT function converts the reference and distorted images into the Fourier domain. The FFT function takes 2.91 s, which is 6.52% of the total execution time.

From the microarchitectural analysis, we find that there are misses at the levels of cache, which requires that we investigate the working data set for the FFT function. The output of the FFT operation is a 512×512 complex matrix, including both real and imaginary parts. This matrix uses the data type double to represent floating-point numbers, and hence, each pixel is 8 bytes. The total data set for the function is $2 \times 512 \times 512 \times 8$ bytes (4 MB). As the total data set is larger than data caches' sizes, data need to be fetched from the main memory, causing performance degradation.

Whether caches are used effectively depends on spatial and temporal locality. Spatial locality involves accessing memory addresses that are close to each other, and temporal locality involves repeated accesses to the same data. A higher number of misses for L1D and L2D caches suggests that the access pattern lacks locality of reference. Techniques such as cache blocking or loop tiling can be used to improve locality of reference, thereby improving performance by reducing misses. A further discussion of cache blocking and other techniques to improve cache performance are discussed in Sec. 6.1.

5.1.5 *Mapping algorithmic blocks to hotspots/hardware bottlenecks*

Figure 10 shows the mapping between hotspots/hardware bottlenecks and the algorithmic blocks of MAD algorithm. The log-Gabor filtering block includes three main blocks: FFT (fast Fourier transform function), five-scale and four-orientation decomposition (Gabor convolution function), and IFFT (inverse FFT function) blocks. This block consumes ~71% of the execution time. It suffers from L1D and L2D replacements, LLC hits, and DTLB overhead. The CSF block endures L1D and L2D replacements, and LLC hits, and consumes ~7.5% of the execution time. The compute visible distortion map block (including DVDMC function as its main function) faces the LLC misses and consumes ~10.5% of the execution time. The compute statistical difference map block consumes ~11% and experiences DTLB overhead and LLC misses.

5.2 *Performance Analysis of MS-SSIM*

To perform hotspot analysis, the MS-SSIM algorithm was applied for 30 iterations for each image. The results of hotspot analysis are provided in Table 4. These results show that the average execution time for all 42 images is ~2.51 s. The algorithm spends ~56.95% of the total time in the top hotspot function and ~74.15 or 80.73% of the total time in the top two or three hotspot functions, while the remaining functions require only 19.27% of the execution time.

We also plot the individual execution time for all 42 images in Fig. 11. The figure shows that MS-SSIM has minimal variation of total execution time across different image content as well as different distortion types. Thus, similar to MAD, optimizations can be made without specific consideration of the image content or distortion.

Table 4 Analysis results of multiscale structural similarity. Average execution time for top hotspot functions/blocks of 42 images is presented with the standard deviation. The total execution for each hotspot is calculated from the average. The hardware bottlenecks are also provided.

Function/block	Execution time (s)	Total execution (%)	Hardware bottlenecks
Low-pass filter 11	1.43 ± 0.10	56.95	L1D, L2D replacement
Similarity measures	0.43 ± 0.06	17.20	L1D, L2D replacement, LLC miss. Assists
LCS average	0.17 ± 0.03	6.58	L2D replacement, LLC miss, DTLB overhead. Floating-point divide
Other	0.48 ± 0.09	19.27	N/A
All	2.51 ± 0.03	100	N/A

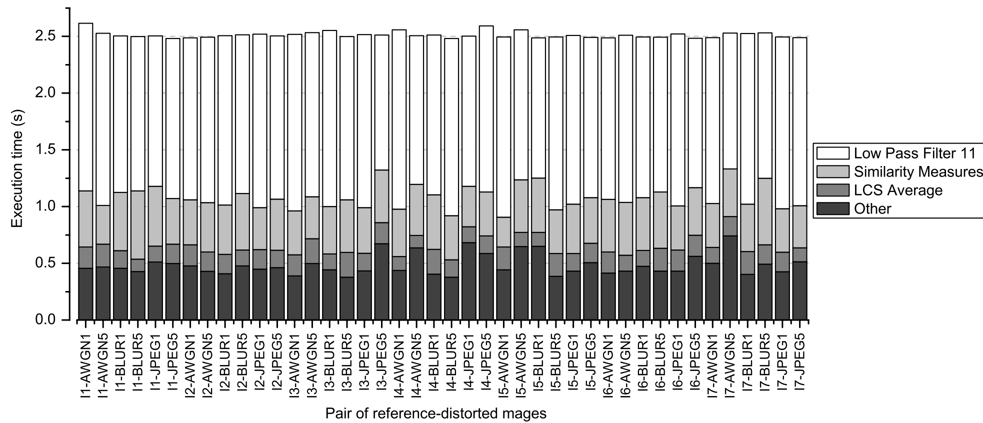

Fig. 11 The execution time of MS-SSIM for each pair of reference and distorted images. The contributions of hotspot functions are stacked together to form the total execution time.

Also shown in Table 4 are hardware bottlenecks (for each hotspot function) identified via the microarchitechtural analysis. The following subsections describe details of the observed results for each of the hotspot functions and include explanations of underlying architectural concepts whenever necessary.

5.2.1 *Low-pass filter 11*

The low-pass filter 11 function is an implementation of an 11×11 Gaussian low-pass filter over the reference image, the distorted image, and their five different scaled versions. The average execution time is ∼1.43 s, which is 56.95% of the total execution time.

The results show that the associated hardware bottlenecks are in the memory subsystem because of the penalties due to L1D and L2D replacements, similar to the functions in MAD. The filter is initially applied over both the reference and distorted images, and then the filtered images are downsampled to calculate the next scale, after which these downsampled images are again filtered. This process is repeated to achieve total five different scales. We can infer that the filter function demonstrates the temporal locality: the filtered image is used to further downsample the images and filter them again. Thus, once the block of data is brought into the cache, it is accessed repeatedly before it is evicted. Although the function has temporal locality, there are replacements in L1D and L2D cache due to the large working data set, which is $\sim 4 \times 512 \times 512 \times 8$ bytes (8 MB). This large working data set leads to cache replacements, which cause a performance penalty and, thus, higher execution time.

5.2.2 *Similarity measures*

The similarity measures function calculates the SSIM index for all five scales by using the luminance, contrast, and structure maps. The average execution time for similarity measures function is ∼0.43 s, which is 17.20% of the total execution time.

The microarchitectural analysis indicates that there is a penalty due to L1D replacements, L2D replacements, and LLC misses due to the large working data set. Along with bottlenecks in the memory subsystem, the function also suffers from hardware bottlenecks due to assists.[59] There are instructions in the block that cannot be directly executed by the processor. These instructions are converted into a stream of microcode that can be executed by the processor. Each such instruction can generate microcode, which can be hundreds of instructions long. Therefore, executing these functions creates a high latency. Calculation of the SSIM index for each scale requires floating-point operations. Although the IEEE 754 standard is used for implementation of floating-point operations, if the floating-point numbers are very small (denormals), they cannot be directly executed by the processor. Thus, these floating-point operations are converted in a stream of microcode and then inserted in the pipeline of the processor. This microcode is hundreds of instructions long, causing performance degradation. One solution to this problem is to write assembly code directly to set denormals to zero.

5.2.3 *LCS average*

The LCS average function calculates the luminance and contrast maps. The function takes ∼0.17 s, which is 6.58% of the total execution time.

Observing the microarchitectural analysis results, we find that the bottlenecks fall into two categories: memory subsystem and core subsystem. The bottlenecks within the memory subsystem are due to L2D replacements, LLC misses, and DTLB overhead.

The bottleneck within the core subsystem is the floating-point divide unit. The calculation of luminance and contrast requires floating-point operations, which are inherently long-latency operations. Because of the continuous feed of floating-point operations for every pixel and a total of 10 images, the floating-point divide unit is overwhelmed. One solution to improve the performance is to use single-precision floating point instead of double precision.

5.2.4 *Mapping algorithmic blocks to hotspots/hardware bottlenecks*

Figure 12 shows the mapping between hotspots/hardware bottlenecks and the algorithmic blocks of MS-SSIM algorithm. The hotspot functions belong to two blocks: LPF_2 and the block corresponding to computation and comparison of luminance, contrast, and structure. The low-pass filter 11 function belongs to the LPF_2 block (55.5% of the execution

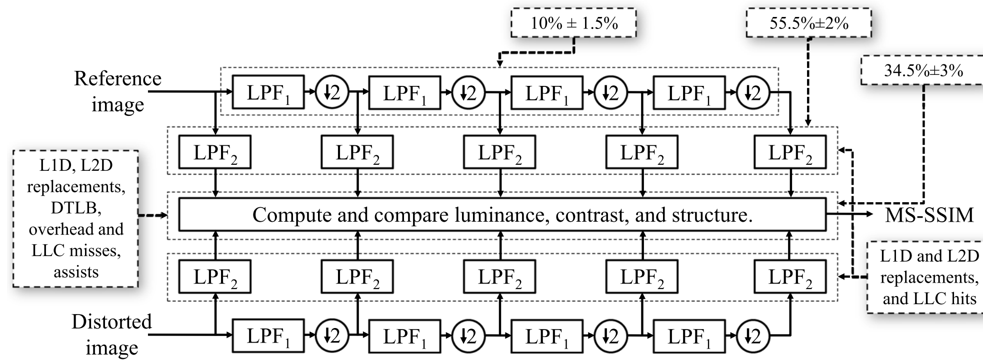

Fig. 12 Mapping between hotspots/hardware bottlenecks and the algorithmic blocks for MS-SSIM. The LPF_2 block suffers from L1D and L2D replacements, and LLC misses. Similarity measures and LCS average functions belong to the block corresponding to computation and comparison of luminance, contrast, and structure, which suffers from L1D replacement, L2D replacement, DTLB overhead, assists, and floating-point divide unit.

time) with L1D and L2D replacements, and LLC misses as the performance bottlenecks. The remaining functions, the similarity measures and the LCS average function, belong to the block corresponding to computation and comparison of luminance, contrast, and structure, which suffers from the memory bottlenecks along with a core bottleneck of floating-point divide unit.

5.3 *Performance Analysis of VIF*

To perform hotspot analysis, the VIF algorithm was applied for 30 iterations for each image. The results of the hotspot analysis are shown in Table 5. These results show that the average execution time for all 42 images is ∼12.12 s. We also plot the individual execution time for all 42 images in Fig. 13. The results show that the top two hotspot functions contribute ∼53.89% of the total execution time. There is minimal variation of the total execution time across different image content as well as different distortion types for VIF, similar to MAD and MS-SSIM. Therefore, neither the image content or distortion need to be considered when making the optimizations for VIF.

5.3.1 *Pyramid filtering*

The pyramid filtering function is the main function of the Steerable Pyramid. This function is employed to compute the reference image's subbands, which are used later to compute parameters of the reference channel. It consumes ∼3.69 s, which is 30.47% of the total execution time. The results of the analysis also show that pyramid filtering has a bottleneck in the core subsystem with stalls due to LEA instructions.

5.3.2 *Pyramid step filtering*

The pyramid step filtering function is similar to pyramid filtering function. This function takes the subsampling according to the START, STEP, and STOP parameters. This function is employed to compute the distorted image's subbands, which are used later to compute parameters of the channel. It takes ∼2.84 s to execute, which is 23.42% of the total execution time.

The microarchitectural analysis for the blocks shows that the pyramid step filtering suffers from bottlenecks in the memory subsystem specifically due to L1D replacements. The results of the analysis also show a bottleneck in the core subsystem with stalls due to LEA instructions.

5.3.3 *Parameters calculation*

The parameters calculation function computes the parameters of channels from the filtered subbands. This function takes ∼2.15 s to execute, which is 17.71% of the total execution time. The function also suffers from memory bottlenecks caused by LLC hits and LLC misses. The processor

Table 5 Analysis results of visual information fidelity. Average execution time for top hotspot functions/blocks of 42 images is presented with the standard deviation. The total execution for each hotspot is calculated from the average. The hardware bottlenecks are also provided.

Function/block	Execution time (s)	Total execution (%)	Hardware bottlenecks
Pyramid filtering	3.69 ± 0.10	30.47	Slow load effective address (LEA) stalls
Pyramid step filtering	2.84 ± 0.08	23.42	L1D replacement. Slow LEA stalls
Parameters calculation	2.15 ± 0.08	17.71	L1D, L2D replacement, LLC hit, LLC miss, DTLB overhead.
Other	3.44 ± 0.14	28.41	N/A
All	12.12 ± 0.07	100	N/A

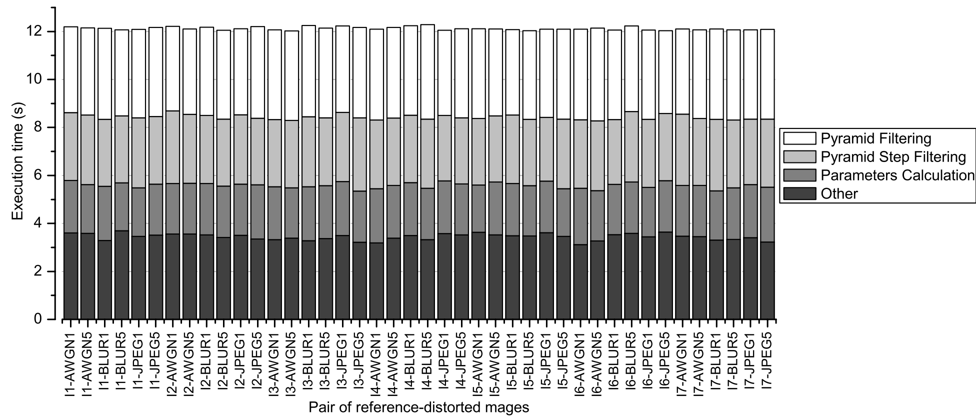

Fig. 13 The execution time of VIF for each pair of reference and distorted images. The contributions of hotspot functions are stacked together to form the total execution time.

has to fetch data from the LLC or the RAM. The penalty for accessing the LLC is ∼26 to 31 clock cycles, while that for accessing the main memory is hundreds of clock cycles. Consequently, parameters calculation function is one of the top hotspots.

5.3.4 *Mapping algorithmic blocks to hotspots/hardware bottlenecks*

Figure 14 shows the mapping between hotspots/hardware bottlenecks and the algorithmic blocks of VIF algorithm. The Steerable Pyramid filtering block, which includes pyramid filtering and pyramid step filtering functions, consumes ∼60% and suffers from generation of slow LEA instruction stalls generated by the compiler. The statistical computation block (parameters calculation is the main function of this block) consumes ∼28.5% and suffers from the major bottleneck in the memory subsystem with L1D and L2D replacements, LLC misses, and DTLB overheads.

5.4 *Performance Analysis of VSNR*

To perform hotspot analysis, the VSNR algorithm was applied for 30 iterations for each image. The results of hotspot for VSNR are provided in Table 6. These results show that the average execution time for all 42 images for MAD is ∼0.72 s. We also plot the individual execution time for all 42 images in Fig. 15. The results show that VSNR has minimal variation of total execution time across different image content as well as different distortion types. Thus, any optimizations for VSNR can be made without any specific consideration of image content or distortion, similar to MAD, MS-SSIM, and VIF.

As shown in Table 6, the top two hotspot functions contribute ∼49.06% of the total execution time, while all others account for the remaining 50.94%. Also shown in this table are hardware bottlenecks (for each hotspot function) identified via the microarchitectural analysis. The following subsections describe details of the observed results for each of the hotspot functions and include explanations of

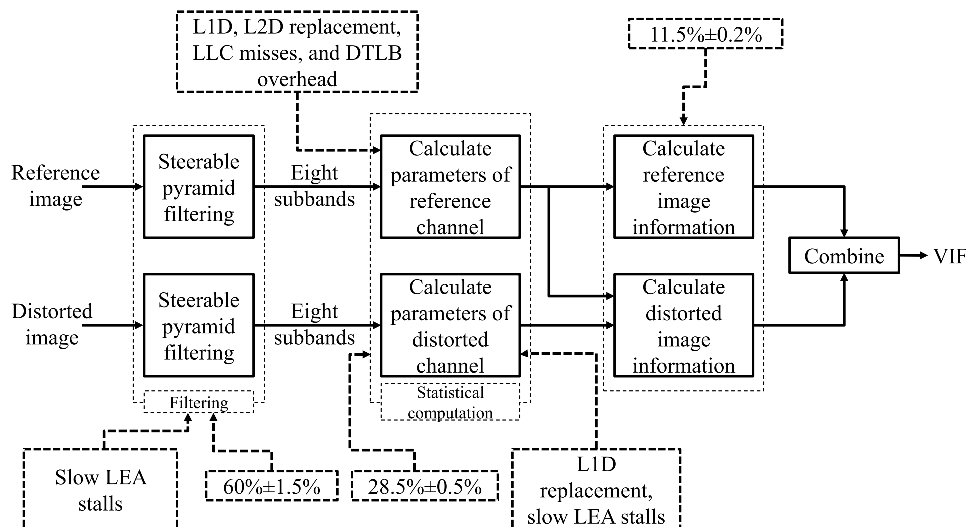

Fig. 14 Mapping between hotspots/hardware bottlenecks and the algorithmic blocks for VIF. The Steerable Pyramid filtering block suffers from generation of slow load effective address (LEA) instruction. The statistical computation block suffers from memory bottlenecks and generation of LEA instructions.

Table 6 Analysis results of visual signal-to-noise ratio. Average execution time for top hotspot functions/blocks of 42 images is presented with the standard deviation. The total execution for each hotspot is calculated from the average. The hardware bottlenecks are also provided.

Function/block	Execution time (s)	Total execution (%)	Hardware bottlenecks
One-dimensional (1-D) discrete wavelet transform (DWT) columns	0.24±0.02	32.61	L1, L2 replacement, and LLC hits
Variance	0.12±0.04	16.45	None
1-D DWT rows	0.10±0.02	13.68	4K aliasing except JPEG5, machine clears for JPEG5
Others	0.27±0.05	37.28	N/A
All	0.72±0.03	100	N/A

the underlying computer architecture concepts whenever necessary.

5.4.1 *1-D DWT-columns*

The 1-D DWT-columns function computes a 1-D DWT across the columns of the reference and distorted images. The function takes ~0.24 s, which is 32.61% of the total execution time.

Investigating the microarchitectural resources utilized by the function, we find that the major penalty for the DWT-columns function is due to LLC hits, which means that the function is accessing LLC frequently. The LLC takes ~26 to 31 clock cycles for a single memory access, which is expensive. Consequently, 1-D 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. Performance can be improved by reducing the L1D and L2D misses, which will automatically reduce LLC accesses.

5.4.2 *Variance*

The variance function takes ~0.12 s, which is 16.45% of the total execution time. The variance function is employed to calculate the RMS contrast in the statistical computation block (Fig. 3). From the microarchitectural analysis, we find that there are no hardware bottlenecks. This means that the function has complex instructions with floating-point numbers that take multiple clock cycles to execute. The analysis does not show a bottleneck because none of the floating-point execution units are overwhelmed, and thus, none of the units cause stalls in processor.

5.4.3 *1-D DWT-rows*

The 1-D DWT-rows function consumes ~0.10 s, which is 13.68% of the total execution time. The 1-D DWT-rows function, similar to 1-D DWT-columns, calculates the DWT coefficients, but across the rows instead of the columns of the reference and distorted images.

The microarchitecture analysis shows that the 1-D DWT-rows function has performance bottlenecks in the memory subsystem. There are memory reissues because of 4K aliasing in this function. In addition, for JPEG5 images, our results show that there are also micro-operations that get cancelled due to machine clears.

5.4.4 *Mapping algorithmic blocks to hotspots/hardware bottlenecks*

Figure 16 shows the mapping between hotspots/hardware bottlenecks and the algorithmic blocks of VSNR algorithm. The five-level 2-D DWT block, which includes 1-D DWT-columns and 1-D DWT-rows functions, consumes ~62% of the execution time, and it suffers from memory bottlenecks ranging from cache misses to memory violations. Those bottlenecks, however, are nowhere to be found in the statistical

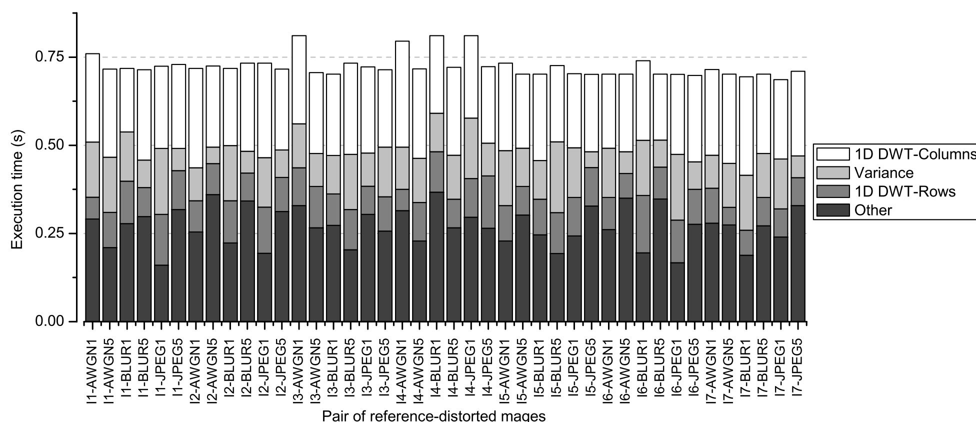

Fig. 15 The execution time of VSNR for each pair of reference and distorted images. The contributions of hotspot functions are stacked together to form the total execution time.

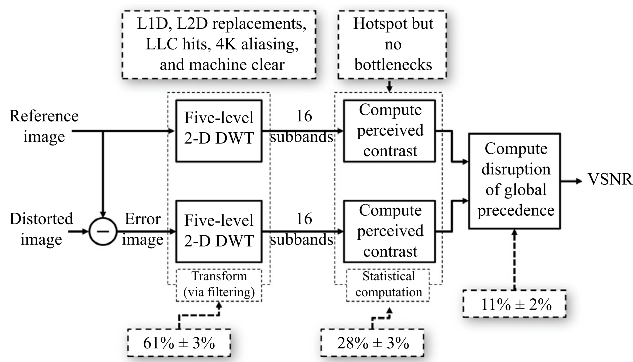

Fig. 16 Mapping between hotspots/hardware bottlenecks and the algorithmic blocks for VSNR. The five-level 2-D discrete wavelet transform block (61% of the execution time) suffers from cache replacements, LLC misses, 4K aliasing, and machine clears. The statistical computation block is a hotspot with ∼28% of the execution time, but there are no bottlenecks.

computation block (variance and some other functions), which is also a hotspot with ∼28% of the execution time.

5.5 *Performance Analysis of BLIINDS-II*

To perform hotspot analysis, the BLIINDS-II algorithm was applied for 30 iterations for each image. The results of the hotspot analysis are provided in Table 7. These results show that the average execution time for all 42 images for BLIINDS-II is ∼8.03 s. The top hotspot function, fast-DCT2, contributes ∼47.44% of the total execution time, while the second one consumes ∼23.44%. The other functions add up to the remaining 29.12%.

Figure 17 shows the individual execution time for all 42 images. This figure shows that the execution time of BLIINDS-II for JPEG5 images is considerably higher than that for the other distortion types. The bottlenecks for JPEG5 images and all the other images are discussed in the later subsections.

5.5.1 *Fast-DCT2*

The fast-DCT2 function calculates the 5×5 DCT of the image at each of the three scales. The function takes ∼3.81 s, which is 47.44% of the total execution time.

The microarchitectural analysis show that fast-DCT2 function does not have any bottlenecks. Therefore, to gain further insight, we calculate the throughput of the function using the retired pipeline slot metric[59] to investigate if the function traverses the pipeline efficiently. The retired pipeline slot metric for the fast-DCT2 function is 0.65, which is greater than the acceptable value of 0.6. The hardware resources are being used optimally in this situation. As mentioned in Sec. 2.5.2, the fast-DCT2 function uses a look-up table to store the cosine values. In addition, we use single loops for first row and column and a nested loop for the remaining pixels. Because the cosine operation is eliminated using the look-up table and all other functions are not relatively inexpensive, the throughput is acceptable, having few stalls and no hardware bottlenecks.

5.5.2 *Gamma*

The gamma function performs a fitting process of the DCT data histogram to the Gaussian model as a line search procedure over 9970 values. This function takes ∼1.88 s to operate, which is 23.44% of the total execution time.

The microarachitectural analysis shows that the bottleneck for this function occurs only for JPEG5 images. For these images, the gamma function becomes the top hotspot. The hardware bottleneck is caused by the data being replaced in the L1D cache, which means the processor fetches data from the L2 cache, which has higher latency. During the line search process, the function traverses an array of 9970 values. If there is a match, the traversal stops. We observe that for JPEG5 images, the function has to traverse to the end of the array. This 9970-value array needs $\sim 9970 \times 8$ bytes (80 KB). However, the L1 cache is 32 KB and cannot hold all of the data. Therefore, some of the values in the array are stored in the next level of cache, and for JPEG5 images, the function has to fetch data from the next level of cache (L2), resulting in higher latency and higher execution time. To improve the performance for the gamma function, one suggestion would be to traverse the array based on the input image's profile. For JPEG5 images, traversing the array from the end would match the value in fewer iterations and improve performance.

5.5.3 *Rho*

The rho function is a sorting function used for feature extraction, which takes the 10th percentile of the sorted array. Because it is employed for multiple features, it is called

Table 7 Analysis results of BLIINDS-II. Average execution time for top hotspot functions/blocks of 42 images is presented with the standard deviation. The total execution for each hotspot is calculated from the average. The hardware bottlenecks are also provided.

Function/block	Execution time (s)	Total execution (%)	Hardware bottlenecks
Fast discrete cosine transform 2	3.81 ± 0.23	47.44	None
Gamma	1.88 ± 1.70	23.44	None/L1D replacement for JPEG5
Rho	0.30 ± 0.11	3.69	None
Convolution	0.29 ± 0.02	3.64	L1D, L2D replacement, LLC hit
Others	1.75 ± 0.22	21.79	N/A
All	8.03 ± 1.69	100	N/A

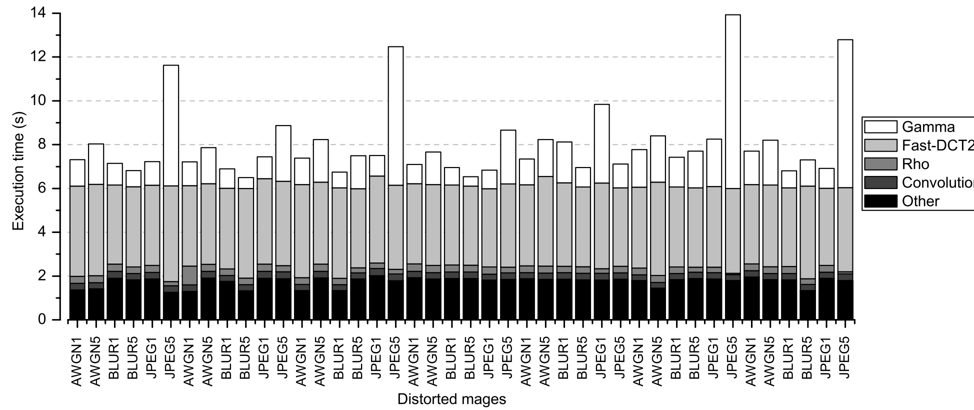

Fig. 17 The execution time of BLIINDS-II for all distorted images. The contributions of hotspot functions are stacked together to form the total execution time.

multiple times in the code. The results of analysis show that the execution time for rho function is ~0.30 s, which is 3.69% of the total execution time.

No bottlenecks were shown in the microarchitectural analysis; we, therefore, computed retired pipeline slot metric to find the throughput of the function. The retired pipeline slot metric is 0.2, which is much lower than the acceptable minimum of 0.6. This finding suggests that the rho function inherently has complex computations, which require higher number of clock cycles, but the Vtune Amplifier XE is unable to identify the exact cause.

5.5.4 *Convolution*

The convolution function performs convolution across the image and is employed to perform low-pass filtering. Its average execution time is close to the rho function's, at ~0.29 s, which is 3.64% of the total execution time. From the microarchitectural analysis, we find that there are L1D as well as L2D replacement penalties because the function accesses the LLC to fetch its operands.

5.5.5 *Mapping algorithmic blocks to hotspots/hardware bottlenecks*

Figure 18 shows the mapping between hotspots/hardware bottlenecks and the algorithmic blocks of BLIINDS-II algorithm. The block-based DCT block is a hotspot with ~66% of the execution time. However, this block has no bottlenecks with the optimized fast-DCT2 function. The gamma function is the main function of generalized Gaussian modeling block (24% of the execution time) and has no bottlenecks except L1D replacements for JPEG5 images. The convolution function, one of the LPF block's functions, suffers from memory bottlenecks, L1D and L2D replacements, and LLC hits.

5.6 *Performance Analysis of BRISQUE*

To perform hotspot analysis, the BRISQUE algorithm was applied for 30 iterations for each image. The results of the hotspot analysis are provided in Table 8. These results show that the average execution time for all 42 images for BRISQUE is ~2.65 s. The top hotspot function, circularly shifting, contributes ~23.02% of the total execution time. The second hotspot consumes ~20.75%. The others add up to the remaining 56.23%.

Figure 19 shows the individual execution time for all 42 images. This figure shows that the execution time of BRISQUE for JPEG5 images is faster than that for the other distortion types. Among the top hotspots, the GGD fitting and AGGD fitting blocks exhibit the most variation; this variation is discussed in Sec. 5.6.4.

5.6.1 *Circularly shifting*

The circularly shifting function circularly shifts the MSCN coefficients one pixel to four orientations to obtain horizontal (H), vertical (V), main-diagonal (D1), and secondary-diagonal (D2) orientated versions. This function is employed to take into account the statistical relationships between neighboring pixels. This block takes ~0.61 s to operate, which is 23.02% of the total execution time.

The circularly shifting function clearly shows a loss of performance due to the CPU caches. The data caches at both the L1 and L2 caches, and the combined LLC are all overwhelmed by this function. This is because computing the pairwise products of the MSCN coefficients with their four circularly shifted versions requires large amounts of memory.

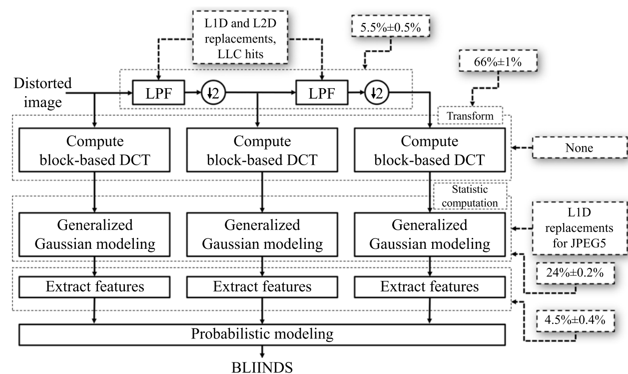

Fig. 18 Mapping between hotspots/hardware bottlenecks and the algorithmic blocks for BLIINDS-II. The block-based DCT is a hotspot with ~66% of the execution time. However, this block has no bottlenecks. The *Gamma* function is the main function of generalized Gaussian modeling block and has no bottlenecks except L1D replacements for JPEG5 images. The *Convolution*, one of the LPF block's functions, suffers from memory bottlenecks, L1D and L2D replacements, and LLC hits.

Table 8 Analysis results of BRISQUE. Average execution time for top hotspot functions/blocks of 42 images is presented with the standard deviation. The total execution for each hotspot is calculated from the average. The hardware bottlenecks are also provided.

Function/block	Execution time (s)	Total execution (%)	Hardware bottlenecks
Circularly shifting	0.61 ± 0.01	23.02	LLC misses, and L1D, L2D replacements
Low-pass filter 7	0.55 ± 0.01	20.75	L1 cache misses, L2D replacements, and LLC misses
SVM regressor	0.47 ± 0.03	17.74	Branch mispredict, front end (I-cache)
Generalized Gaussian distribution (GGD) and asymmetric GGD fitting	0.46 ± 0.07	17.36	LLC misses, L1D, L2D replacements, and DTLB overhead
Others	0.56 ± 0.01	21.13	N/A
All	2.65 ± 0.08	100	N/A

5.6.2 *Low-pass filter 7*

The low-pass filter 7 function is an implementation of a 7×7 circularly symmetric Gaussian filter. This function is called four times totally for two scales. The average execution time is ∼0.55 s, which is 20.75% of the total execution time.

Observing the microarchitectural analysis results, we find that the bottlenecks are again purely in the memory subsystem. Specifically, the bottlenecks are caused due to L1 cache misses, L2D replacements, and LLC misses. The low-pass filter 7 function was optimized to work with two 7×1 windows instead of one 7×7 window (see Sec. 2.6.2), and thus, there is no core bottlenecks found here. It needs memory for the MSCN coefficients, squared MSCN coefficients, and Gaussian filtered images. Consequently, this function also has a large memory footprint causing bottlenecks at all levels of caches in the memory subsystem.

5.6.3 *SVM regressor*

The SVM regressor block is basically a function call to the LIBSVM executable via the system function, similar to the MATLAB® version. It collapses 36 features into one single quality index. The time for this function is ∼0.47 s, which is 17.74% of the total execution time.

The code currently makes calls to the executable file to collapse the previously calculated 36 features into a single BRISQUE index. Since, this executable is not compiled as a part of the C++ implementation, there is no prefetching of the instructions (not fetched into the I-cache), which causes the I-cache misses and branch mispredictions. Similar issues when linking to external files have been previously reported (see Ref. 60).

5.6.4 *GGD and AGGD fitting*

The GGD fitting and AGGD fitting blocks perform a fitting of the MSCN coefficients histogram to the generalized Gaussian model, and four pairwise products with an asymmetric generalized Gaussian model as a search procedure over 9801 values. Similar to the gamma function in BLIINDS-II, GGD and AGGD fitting employs a line search to fit shape, mean, left variance, and right variance to the values of a gamma function. However, in contrast to BLIINDS-II, where the fitting process is applied for every small 5×5 DCT block, for BRISQUE, the fitting process is employed only 10 times. During the line search process, there could be an early stop when a match is found. Therefore, the GGD and AGGD fitting block's execution times have a larger variation.

This block suffers from memory bottlenecks. Specifically, the bottlenecks are LLC misses, and L1D and L2D replacements. The block operates on a total of 10 images (eight are generated from the circularly shifting function and two MSCN coefficients). Because the block operates separately

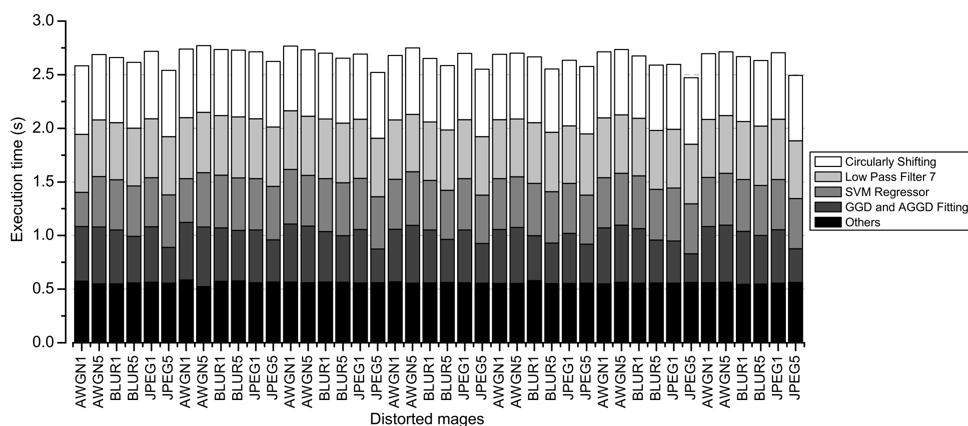

Fig. 19 The execution time of BRISQUE for all distorted images. The contributions of hotspot functions are stacked together to form the total execution time.

on two different data sets for two scales, there are replacements in the L1 and L2 cache and misses in the LLC. Since the block operates on two different data sets, the mapping of the virtual to physical addresses for both the data sets cannot fit into the small DTLB, which causes the DTLB overhead.

5.6.5 *Mapping algorithmic blocks to hotspots/hardware bottlenecks*

Figure 20 shows the mapping between hotspots/hardware bottlenecks and the algorithmic blocks of the BRISQUE algorithm. The compute locally normalized luminance block contains mainly the low-pass filter 7 function. It is a hotspot with ∼26.32% of the execution time and suffers from L1 cache misses, L2D replacements, and LLC misses. In the next stage, the compute H, V, D1, and D2 pairwise products block contain the circularly shifting function with point-by-point multiplications between the MSCN coefficients and their circularly shifted versions. This block takes ∼0.65 s to operate, which is 24.44% of the total execution time. The GGD fitting and AGGD fitting blocks consume 17.5% of the running time and suffer from LLC misses, L1D and L2D replacements, and DTLB overhead bottlenecks.

6 Discussion

In this section, we discuss the bottlenecks and provide an insight into them across all the six analyzed algorithms. We first discuss the most common memory bottlenecks and microarchitectural conscious coding techniques to gain better performance. Following the memory bottlenecks, we discuss the core bottlenecks and the techniques to boost performance. We also propose a custom hardware framework, which can be used as a platform to design engines for IQA algorithms and image processing algorithms in general.

6.1 *Memory Bottlenecks*

The results of the microarchitectural analysis show that all of the IQA algorithms have a backend-bound memory bottleneck, but the amount of performance degradation due to these bottlenecks varies greatly for individual algorithms. A memory bottleneck essentially means that the hotspot functions spend a significant amount of time accessing image data. This is usually because they have read/write access patterns that result in misses in the CPU caches. These cache misses have to be serviced from lower levels of the memory hierarchy that are slower to access, which is why we see the algorithms spending more time in these functions and causing them to be hotspots. The large number of cache misses is due to the large working data set for these algorithms. All of the algorithms produce intermediate matrices and process them to assess the image quality. Accessing these multidimensional arrays multiple times causes most of the performance bottlenecks for all of the IQA algorithms tested here.

Even though all the IQA algorithms experience memory bottlenecks, the root cause of the bottleneck as well as the extent of performance degradation due to these bottlenecks varies across algorithms. For example, all the hotspot functions in MAD are backend memory bound, whereas VIF has a memory bottleneck for just the parameters calculation function. It should be noted that the parameters calculation function is the last hotspot for VIF, which means that it is not the major cause for slowdown of the algorithm; its impact on the performance of the algorithm is not high as other functions, and these other functions are not memory bound. Thus, it is important to note that even if all the algorithms at some point have a memory bottleneck, the impact of this memory bottleneck on the speed or throughput of 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 is different for different algorithms. For example, the top two hotspot functions in MAD have poor performance because of the DTLB overhead, while the top two hotspots for MS-SSIM and BRISQUE have a higher execution time because of the L1D replacements, L2D replacements, and LLC misses. Different microarchitectural resources are overwhelmed by different functions and algorithms.

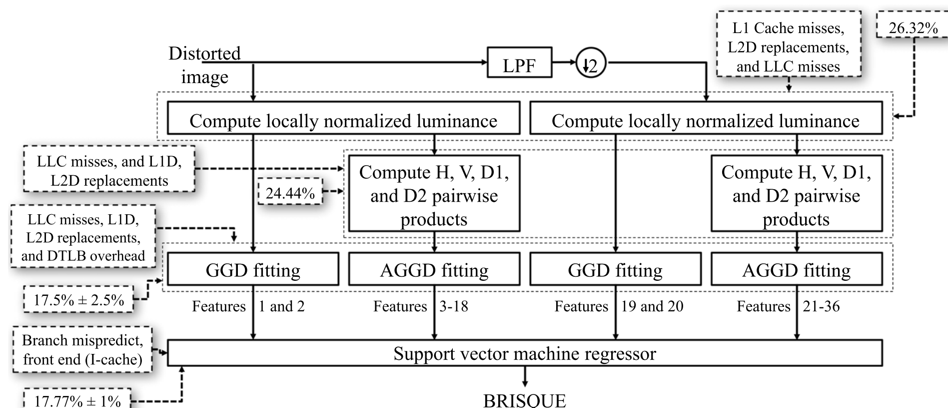

Fig. 20 Mapping between hotspots/hardware bottlenecks and the algorithmic blocks for BRISQUE. The *Compute locally normalized luminance* is a hotspot with ∼26.32% of the execution time and suffers from L1 cache misses, L2D replacements, and LLC misses. In the next stage, the *Compute H, V, D1, and D2 pairwise products* block contain the *Circularly Shifting* function with point-by-point multiplications between the MSCN coefficients and their circularly shifted versions. This block takes ∼0.65 s to operate, which is 24.44% of the total execution time. The *GGD fitting* and *AGGD fitting* blocks consume 17.5% of the running time and suffer from LLC misses, L1D, L2D replacements, and DTLB overhead bottlenecks.

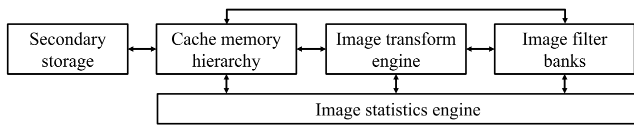

Fig. 21 Blocks for custom image quality assessment (IQA) engine framework. It consists of three basic computational blocks generally used in IQA algorithms: image transform engine, the filter banks, and the image statistics engine.

Apart from issues with the usual suspects in the memory hierarchy (CPU caches at different levels), there are hotspot functions that show penalties associated with machine clears and 4K aliasing (as in case of VSNR). Thus, our analysis has revealed some interesting performance bottlenecks that would otherwise have gone undetected.

One obvious way to reduce cache misses is to change the hardware platform to a processor with larger caches. This will allow more data to reside in the caches, and thereby reduce the number of cache misses, and will, consequently, improve performance. However, there is a limit to how big each level of cache in the memory hierarchy can be, and the L1 caches are usually kept under 64 KB for fast access. Thus, it is important that the implementation of an IQA algorithm respects the principle of locality of reference to maximize temporal and spatial locality. Locality of reference is the tendency of a program to access the same (temporal locality) or nearby (spatial locality) memory locations repeatedly and frequently. Thus, caching these memory locations can reduce misses. Coding an IQA algorithm with more/better locality can significantly improve performance.

Another technique for improving locality, which is commonly used especially in image processing, is called cache blocking. It works by dividing larger data chunks into smaller ones that fit in the cache and making sure that once the small block is brought into the cache, all the operations to be performed are performed before moving on to the next block. Thus, rather than traversing a whole image to perform one operation and then reading in the whole image to perform another operation (thus, effectively wiping out the cache repeatedly), parts of the image are read and processed at a time while that part is still in the cache.

Accesses to the main memory are very expensive, usually hundreds of clock cycles. For algorithms, such as MAD, MS-SSIM, VIF, and BRISQUE, where the bottlenecks are due to LLC misses/replacements, using software instructions that can prefetch the data into the caches from the main memory effectively masks the memory latency and can, thus, increase perfomance. Such prefetch instructions are suitable when memory accesses are predictable and when there are CPU stalls for data being unavailable for processing.

The final memory bottleneck is the DTLB overhead. The TLB is a small cache, which stores a section of the page table (a page table stores a mapping between virtual and physical addresses). When the working data set is large, as in the case of MAD, the TLB is not able to cache all the active mappings and this causes DTLB misses. One solution to this problem is to use larger page sizes or superpages. With a larger page size, a TLB of the same size can keep track of larger amounts of memory, which avoids the costly TLB misses, reducing the pressure on the TLB.

As an example of applying these techniques, we made slight modifications to the FFT and Gabor convolution functions for MAD. We chose these functions because they are in the top hotspot list. Based on our analysis, we knew that the architectural bottleneck was related to the memory hierarchy, as the DTLB, LLC misses, and L1D and L2D replacement feature prominently for MAD. We modified the nested loop in the FFT function to improve the locality and data reuse by removing one layer of nesting. This had the effect of accessing a cache block multiple times, as each cache block brought in eight elements of the array assuming a 64-byte block size, and they were processed sequentially resulting in seven hits for every miss in the cache for that loop. The original nested loop processed only one array element per cache block, which resulted in a 100% cache miss rate for that particular nested loop. The second modification we made to the Gabor convolution function was to eliminate the memory required for additional matrices of the same size of the input image. Essentially, the optimization was equivalent to changing the statement $C = A + B$ to $A = A + B$, thus reusing the memory allocated to A. These two modifications resulted in a 9% improvement in terms of running time.

While these examples seem somewhat obvious in retrospect, most codecs are written with many such opportunities overlooked because the focus may not have been on efficient use of the underlying hardware. It is also not possible to anticipate the exact hardware bottlenecks without conducting the kind of analysis illustrated in this paper. The analysis clearly helps pinpoint the functions that take up the largest share of the execution time and highlights the architectural resource being stressed. Thus, a programmer knows where to look and what changes to make.

6.2 *Core Bottlenecks*

The next category of bottlenecks is the core bottlenecks or the bottlenecks caused due to manipulation of data. First, we discuss the performance degradation caused due to floating-point operations because they have the most significant impact on performance in the category of core bottlenecks.

The floating-point operations inherently have a longer latency and, thus, have a large impact on performance. The floating-point unit is a bottleneck for MS-SSIM for the LCS average function. We discuss some generic guidelines to improve performance for floating-point units.

Operations carried on single-precision floating-point numbers execute faster than double precision numbers and consume less memory. For example, the LCS average function calculates the luminance and contrast of the reference and distorted images for multiple scales. The mean of pixel values is used to calculate luminance, while the variance is used to calculate contrast. All of the algorithms currently use an 8-bit gray-scale image, which implies that the resultant luminance and contrast would never exceed the range supported by 32-bit single-precision floating-point numbers. Since calculation for luminance and contrast is calculated for multiple scales, using single-precision floating-point numbers to calculate and represent luminance and contrast can significantly improve performance. On our test platform, single-precision floating point can be set through the precision control field in the ×87 floating-point unit.

Another simple solution is to use integers if possible. For example, if a particular range varies from 0 to 1, the programmer can estimate the degree of precision required and

choose to express the range from 0 to 10, 0 to 100, 0 to 1000, and so on (depending on whether one-, two-, or three-digit precision is required).

The developer of an IQA algorithm should keep in mind that the operations should remain in range, i.e., ensure that there are no overflows, underflows, or denormals (extremely small values) for the results. Denormal values and underflow can cause high penalties as they require microcode assists. To improve performance for such a situation, one solution is to write the code in such a way that denormals are not generated, which means that the developer should keep track of the range of the results produced during the floating-point operations. In addition, by enabling the flush-to-zero (FTZ) and denormals-are-zero (DAZ) modes,[61] there can be significant improvement in performance. Note that the FTZ and DAZ modes are applicable only for streaming SIMD extension (SSE) instructions.

The next bottleneck is caused due to generation of slow LEA instructions. LEA instruction is an X86 assembly instruction generated during compilation. On an Intel processor, one solution is to use an Intel compiler, which would produce assembly optimized for Intel's microarchitectures. For example, in a loop, the LEA instructions are generated for the index access. If the index access is reduced, it will decrease the number of LEA instructions to be executed and, thus, reduce the overhead.

6.3 *Summary and Recommendations for a Framework for Custom Image Quality Assessment Hardware*

From the microarchitectural analysis, we found that there are two categories of bottlenecks in the IQA algorithms tested here: execution/core bottlenecks and memory bottlenecks. From the analysis, we found that the majority of the algorithms show performance degradation because of the memory bottlenecks. Other studies also show that memory is usually a major bottleneck for image processing algorithms. Within the memory bottlenecks, the most common issue was due to L1D and L2D replacements and LLC misses. To improve performance in such cases, having larger caches is recommended. A suitable size for the caches and its configuration along with defining a memory hierarchy can be decided by performing cache simulations with various cache sizes and configurations. A combination of a particular cache size and configuration that surpasses a predefined threshold for hit/miss rates should be used. Creating models for a cache configuration that best suits the performance, cost, and other requirements of IQA and related algorithms are topics of future research.

The next common memory bottleneck was the DTLB overhead. MAD, VIF, and MS-SSIM show performance degradation due to DTLB overhead. A DTLB is a special cache that stores a subset of translations from virtual memory to physical memory. For a custom IQA engine, there is no requirement of a virtual memory system. Hence, this eliminates the requirement of using a TLB.

Another bottleneck was 4K aliasing. VSNR is the only algorithm with 4K aliasing bottleneck. This problem is caused due to out-of-order execution of memory instructions in the processor. If the custom hardware engine design is an in-order machine, we eliminate the possibility of 4K aliasing.

The most common core bottleneck was the overwhelming of the floating-point unit. All the image transform, image filtering, and statistic calculation require floating-point operations and, consequently, a floating-point execution unit. Operating on a logarithmic number system would improve the performance. The overhead for converting to log domain and then transforming back is negligible if the number of floating-point operations is very large. In a logarithmic system, the multiply and divide operations change to add and subtract operations, respectively, which are less expensive and can save many clock cycles. A custom IQA engine would have multiple floating-point units to exploit parallelism. In addition, the unit will be pipelined to hide/overlap the latency of the instructions for getting data to the IQA execution engine.

The next core bottleneck was due to the slow LEA instructions. These instructions are an outcome of the complex addressing mode of the CISC Intel architecture. Therefore, if the memory control hardware is designed as a load store machine, the performance degradation is automatically eliminated. In addition, the proposed custom engine is hard-coded, which eliminates issues due to generation of such instructions by the compiler.

The final core bottleneck was the generation of micro assists. Floating-point micro assists occurred because the operands or results of an operation were denormals. If the precision can be traded-off, these denormals are 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.

In general, all tested IQA algorithms contain the same operation at an abstract level: an image transform or filtering and a statistical computing. For example, MS-SSIM and BRISQUE use low-pass filters, whereas VIF, VSNR, MAD, and BLIINDS-II transform the image into frequency domains via Steerable Pyramid, DWT, FFT, and DCT, respectively. These algorithms also perform a statistical computation. For example, MS-SSIM needs the mean and variance to calculate the structure similarity; MAD calculates the standard deviation, skewness, and kurtosis to form the statistical difference maps. Therefore, a generic IQA engine would have a transform engine, a filtering engine, and a statistical computation engine.

A block diagram for a general IQA hardware engine is shown in Fig. 21. We have a secondary storage to store the input images. These images 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 performed by the specific IQA algorithm. In addition, there are instances where an operation is performed multiple times. Therefore, we propose a fully connected internetwork. Such an interconnected network helps to feed data directly to the respective engine, which leads to reuse of the existing hardware and saves chip area and cost.

The different execution engines are designed as follows:

1. The transform block can be a general-purpose floating-point unit or a transform-specific custom design, such as 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, such as a log-Gabor filter unit in MAD.

3. The image statistics block can be implemented as a general-purpose engine or a custom engine. However, if we observe the algorithms, they comprise multiple statistical computations. Therefore, it is not recommended to create an engine for all, but rather to utilize a general-purpose floating-point unit. In order to define which operations needs to be performed, control signals can be generated. It is suggested to pipeline these engines. Pipelining would hide some latency for accessing memory and also improve the throughput of the hardware. Further details about pipelining, trade-offs for a pipeline, and designing a pipelined hardware can be found in Ref. 57.

7 Conclusions

This paper presented performance analyses of six popular IQA algorithms. Even though the approaches to the six IQA algorithms are different, the algorithms shared the same main stages: a filtering (transforming) and a statistical computation. Our results revealed that different IQA algorithms overwhelm different microarchitectural resources and give rise to different types of bottlenecks in two main categories: memory bottlenecks and core/computational bottlenecks. Specific microarchitectural bottlenecks for each function/block of each algorithm were pointed out. We also proposed the hardware/microarchitectural conscious coding techniques for optimization and performance improvement. The findings and recommendations presented in this paper apply broadly to all current-generation Intel IA-32 and Intel 64 based general-purpose computing platforms, whether laptops, servers, or desktops, even though the actual hotspot and bottleneck details might vary. Architectures that are radically different, with hardware accelerators, dedicated image processing cores (such as those found on some tablets and smart phones), and memory shared between GPUs and CPUs (such as AMD's Fusion APUs), are expected to show very different execution characteristics. Further studies using a similar methodology are recommended to analyze the performances of IQA algorithms on these specialized architectures.

Acknowledgments

This material is based upon work supported by the National Science Foundation Awards 0917014 and 1054612, and by the U.S. Army Research Laboratory (USARL) and the U.S. Army Research Office (USARO) under contract/Grant No. W911NF-10-1-0015.

References

1. C. J. B. 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*, Atlanta, Georgia, pp. 2291–2294 (1996).
2. Z. Wang, A. C. Bovik, and L. Lu, "Wavelet-based foveated image quality measurement for region of interest image coding," in *Proc. IEEE Int. Conf. on Image Processing*, Thessaloniki, Greece, pp. 89–92 (2001).
3. K. Yang and H. Jiang, "Optimized-SSIM based quantization in optical remote sensing image compression," in *Proc. of the 2011 Sixth Int. Conf. on Image and Graphics*, pp. 117–122, IEEE Computer Society, Washington, DC (2011).
4. A. Rehman et al., "SSIM-inspired image restoration using sparse representation," *EURASIP J. Adv. Signal Process.* **2012**(1), 16 (2012).
5. F. Ciaramello et al., "Predicting intelligibility of compressed American sign language video with objective quality metrics," in *Proc. Int. Workshop on Video Processing and Quality Metrics for Consumer Electronics* (2006).
6. K. Seshadrinathan and A. C. Bovik, "Automatic prediction of perceptual quality of multimedia signals—a survey," *Multimedia Tools Appl.* **51**(1), 163–186 (2011).
7. W. Lin and C. C. Jay Kuo, "Perceptual visual quality metrics: a survey," *J. Vis. Commun. Image Represent.* **22**(4), 297–312 (2011).
8. A. C. Bovik, "Automatic prediction of perceptual image and video quality," *Proc. IEEE* **101**(9), 2008–2024 (2012).
9. D. M. Chandler, "Seven challenges in image quality assessment: past, present, and future research," *ISRN Signal Process.* **2013**, 905685 (2013).
10. Z. Wang, E. Simoncelli, and A. Bovik, "Multiscale structural similarity for image quality assessment," in *Conf. Record of the Thirty-Seventh Asilomar Conf. on Signals, Systems and Computers*, Pacific Grove, California, Vol. 2, pp. 1398–1402 (2003).
11. H. Sheikh and A. Bovik, "Image information and visual quality," *IEEE Trans. Image Process.* **15**(2), 430–444 (2006).
12. E. C. Larson and D. M. Chandler, "Most apparent distortion: full-reference image quality assessment and the role of strategy," *J. Electron. Imaging* **19**(1), 011006 (2010).
13. A. K. Moorthy and A. C. Bovik, "Blind image quality assessment: from natural scene statistics to perceptual quality," *IEEE Trans. Image Process.* **20**(12), 3350–3364 (2011).
14. M. Saad, A. Bovik, and C. Charrier, "Blind image quality assessment: a natural scene statistics approach in the DCT domain," *IEEE Trans. Image Process.* **21**(8), 3339–3352 (2012).
15. A. Mittal, A. Moorthy, and A. Bovik, "No-reference image quality assessment in the spatial domain,"*IEEE Trans. Image Process.* **21**(12), 4695–4708 (2012).
16. K. Seshadrinathan and A. Bovik, "Motion tuned spatio-temporal quality assessment of natural videos," *IEEE Trans. Image Process.* **19**(2), 335–350 (2010).
17. P. Vu, C. Vu, and D. Chandler, "A spatiotemporal most-apparent-distortion model for video quality assessment," in *18th IEEE Int. Conf. on Image Processing*, Brussels, Belgium, pp. 2505–2508 (2011).
18. W.-H. Chen, C. Smith, and S. Fralick, "A fast computational algorithm for the discrete cosine transform," *IEEE Trans. Commun.* **25**(9), 1004–1009 (1977).
19. H. Hou, "A fast recursive algorithm for computing the discrete cosine transform," *IEEE Trans. Acoust. Speech Signal Process.* **35**(10), 1455–1461 (1987).
20. J. Liang and T. D. Tran, "Fast multiplierless approximation of the DCT with the lifting scheme," *IEEE Trans. Signal Process.* **49**(12), 3032–3044 (2000).
21. W. Yuan, P. Hao, and C. Xu, "Matrix factorization for fast DCT algorithms," in *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, Toulouse, France, Vol. 3, p. III (2006).
22. C. Cheng and K. Parhi, "Hardware efficient fast DCT based on novel cyclic convolution structures," *IEEE Trans. Signal Process.* **54**(11), 4419–4434 (2006).
23. V. Britanak, P. Yip, and K. Rao, *Discrete Cosine and Sine Transforms: General Properties, Fast Algorithms and Integer Approximations*, Academic, Oxford, UK (2007).
24. D. Trainor, J. Heron, and R. Woods, "Implementation of the 2D DCT using a Xilinx XC6264 FPGA," in *IEEE Workshop on Signal Processing Systems*, Leicester, UK, pp. 541–550 (1997).
25. G. Kiryukhin and M. Celenk, "Implementation of 2D-DCT on XC4000 series FPGA using DFT-based DSFG and DA architectures," in *Int. Conf. on Image Processing*, Thessaloniki, Greece, Vol. 3, pp. 302–305 (2001).
26. B. Fang et al., "Techniques for efficient DCT/iDCT implementation on generic GPU," in *IEEE Int. Symp. on Circuits and Systems*, Vol. 2, pp. 1126–1129 (2005).
27. S. Tokdemir and S. Belkasim, "Parallel processing of DCT on GPU," in *Data Compression Conf.*, Snowbird, Utah, p. 479 (2011).
28. T.-T. Wong et al., "Discrete wavelet transform on consumer-level graphics hardware," *IEEE Trans. Multimedia* **9**(3), 668–673 (2007).
29. C. Tenllado et al., "Parallel implementation of the 2D discrete wavelet transform on graphics processing units: filter bank versus lifting," *IEEE Trans. Parallel Distrib. Syst.* **19**(3), 299–310 (2008).
30. J. Franco et al., "A parallel implementation of the 2D wavelet transform using CUDA," in *17th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing*, Weimar, Germany, pp. 111–118 (2009).
31. M. Unser, "Fast Gabor-like windowed Fourier and continuous wavelet transforms," *IEEE Signal Process. Lett.* **1**(5), 76–79 (1994).
32. L. Tao and H. K. Kwan, "Fast parallel approach for 2-D DHT-based real-valued discrete Gabor transform," *IEEE Trans. Image Process.* **18**(12), 2790–2796 (2009).
33. X. Wang and B. Shi, "GPU implementation of fast Gabor filters," in *Proc. of 2010 IEEE Int. Symp. on Circuits and Systems*, Parris, France, pp. 373–376 (2010).
34. L. Tao and H. K. Kwan, "Multirate-based fast parallel algorithms for 2-D DHT-based real-valued discrete Gabor transform," *IEEE Trans. Image Process.* **21**(7), 3306–3311 (2012).

35. F. C. Crow, "Summed-area tables for texture mapping," *SIGGRAPH Comput. Graph.* **18**(3), 207–212 (1984).
36. F. Shafait, D. Keysers, and T. M. Breuel, "Efficient implementation of local adaptive thresholding techniques using integral images," *Proc. SPIE* **6815**, 681510 (2008).
37. T. Phan et al., "Performance-analysis-based acceleration of image quality assessment," in *IEEE Southwest Symp. on Image Analysis and Interpretation*, Santa Fe, New Mexico, pp. 81–84 (2012).
38. M.-J. Chen and A. C. Bovik, "Fast structural similarity index algorithm," *J. Real-Time Image Process.* **6**(4), 281–287 (2011).
39. H. R. Sheikh et al., "Image and video quality assessment research at LIVE," http://live.ece.utexas.edu/research/quality/ (4 February 2014).
40. K. Okarma and P. Mazurek," GPGPU based estimation of the combined video quality metric," in *Image Processing and Communications Challenges 3*, R. Choras, Ed., pp. 285–292, Springer, Berlin, Heidelberg (2011).
41. D. M. Chandler and S. S. Hemami, "VSNR: a wavelet-based visual signal-to-noise ratio for natural images," *IEEE Trans. Image Process.* **16**(9), 2284–2298 (2007).
42. R. Bhargava et al., "Evaluating MMX technology using DSP and multimedia applications," in *Proc. of the 31st Annual ACM/IEEE Int. Symp. on Microarchitecture*, pp. 37–46, IEEE Computer Society Press, Los Alamitos, California (1998).
43. Intel, "Intel's Vtune Amplifier XE," 2013, http://developer.intel.com/design/perftool/vtcd/ (28 December 2013).
44. B. Gordon, S. Sohoni, and D. Chandler, "Data handling inefficiencies between CUDA, 3D rendering, and system memory," in *IEEE Int. Symp. on Workload Characterization*, Atlanta, Georgia, pp. 1–10 (2010).
45. C. Martinez, M. Pinnamaneni, and E. B. John, "Performance of commercial multimedia workloads on the Intel Pentium 4: a case study," *Computers Electr. Eng.* **35**(1), 18–32 (2009).
46. Z. Wang et al., "Image quality assessment: from error visibility to structural similarity," *IEEE Trans. Image Process.* **13**(4), 600–612 (2004).
47. E. P. Simoncelli, "The Steerable Pyramid," 1995, http://www.cns.nyu.edu/~eero/steerpyr/ (4 February 2014).
48. H. R. Sheikh and A. C. Bovik, "VIF download," http://live.ece.utexas.edu/research/quality/VIF.htm (4 February 2014).
49. R. B. Davies, "Newmat C++ matrix library," 2006, http://www.robertnz.net/nm_intro.htm (4 February 2014).
50. J. Villasenor, B. Belzer, and J. Liao, "Wavelet filter evaluation for image compression," *IEEE Trans. Image Process.* **4**(8), 1053–1060 (1995).
51. J. L. Mannos and D. J. Sakrison, "The effects of a visual fidelity criterion on the encoding of image," *IEEE Trans. Info. Theory* **20**(4), 525–535 (1974).
52. E. C. Larson and D. M. Chandler, "MAD online," 2010, http://vision.okstate.edu/MAD (4 February 2014).
53. T. Ooura, "Mathematical Software Packages," 2006, http://www.kurims.kyoto-u.ac.jp/~ooura/ (4 February 2014).
54. P. D. Kovesi, "MATLAB and octave functions for computer vision and image processing," 2000, http://www.csse.uwa.edu.au/~pk/research/matlabfns/ (4 February 2014).
55. M. Saad, A. Bovik, and C. Charrier, "A DCT statistics-based blind image quality index," *IEEE Signal Process. Lett.* **17**(6), 583–586 (2010).
56. C.-C. Chang and C.-J. Lin, "LIBSVM: a library for support vector machines," 2001, http://www.csie.ntu.edu.tw/~cjlin/libsvm (4 February 2014).
57. J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, Vol. 2, McGraw-Hill Higher Education, New York, NY (2005).
58. Intel, "Architectures Optimization Reference Manual," 2012, http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf (4 February 2014).
59. Intel, "Software Documentation Library," 2013, http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/amplifierxe/win/win_ug/index.htm (4 February 2014).
60. S. Vlaovic, E. S. Davidson, and G. S. Tyson, "Improving BTB performance in the presence of DLLs," in *Proc. of the 33rd Annual ACM/IEEE Int. Symp. on Microarchitecture*, pp. 77–86, ACM, New York, NY (2000).
61. 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/.

Thien D. Phan received his BS degree in information technology from Hanoi University of Technology, Hanoi, Vietnam, in 2008. He is currently working toward his PhD degree in the School of Electrical and Computer Engineering, Oklahoma State University, Stillwater. His research interests include image processing, computer vision, data compression, machine learning, and optimization for multidimensional signal-processing applications.

Siddharth K. Shah received his BE degree in electronics and telecommunication from the University of Pune in 2009 and his MS in electrical engineering from Oklahoma State University in 2012. He is currently working with Advanced Micro Devices as a design engineer. His research interests include computer architecture, performance analysis, and design verification.

Damon M. Chandler received his BS degree in biomedical engineering from Johns Hopkins University, Baltimore, Maryland, in 1998 and his MEng, MS, and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 2000, 2003, and 2005, respectively. From 2005 to 2006, he was a postdoctoral research associate in the Department of Psychology at Cornell. He is currently an associate professor in the School of Electrical and Computer Engineering at Oklahoma State University, Stillwater, where he heads the Laboratory of Computational Perception and Image Quality. His research interests include image processing, data compression, computational vision, natural scene statistics, and visual perception.

Sohum Sohoni is an assistant professor in engineering and computing systems at Arizona State University (ASU). Prior to joining ASU, he was an assistant professor at Oklahoma State University. His research interests are broadly in the areas of computer architecture and performance analysis, and in engineering and computing education. His work appears in *ACM SIGMETRICS*, *IEEE Transactions on Computers*, *International Journal of Engineering Education*, and *Advances in Engineering Education*.