Research Article

Optimized OpenCL™ kernels for frequency domain image high-boost filters using image vectorization technique

Ashutosh Satapathy[1] · L. M. Jenila Livingston[1]

Abstract
Image high boost filtering uses high-boost filters to enhance the quality of an image, which has also seen in remote sensing, satellite broadcasting, classroom monitoring, and many more real-time video processing applications and requires its faster implementation. OpenCL is a widely adapted parallel programming framework that provides core level data parallelism, and dedicated for heterogeneous parallel devices like from low cost DSP to high-end CPU, GPU and FPGA. In this article, we have considered mostly used Ideal, Gaussian, Butterworth, and Laplacian of Gaussian frequency domain high-boost filters and implemented channelized OpenCL kernels for their rapid execution. In addition to that, these kernels are modified using image vectorization technique to optimize their time utilization by reducing the execution time of these OpenCL kernels to half. At last, performance analysis is carried out for these two types of OpenCL kernel implementations to determine their effectiveness with regard to time consumption and accuracy. Here, different image performance evaluation metrics like entropy, standard deviation, mean absolute error, percentage fit error, SSIM, correlation, and peak signal to noise ratio are applied to measure rightness of the above high-boost filters. From the results, we have concluded that a vectorized Butterworth high-boost filter kernel is the suitable one to provide better results among those filters, which might be highly adaptable in time bound real-time applications using various embedded devices.

Keywords Image enhancement · OpenCL architecture · OpenCL execution model · OpenCL memory model · Image high-boosting · Performance metrics

1 Introduction

An image in the spatial domain is the pictorial description of various observations that explain thousands of meaningful information about a specific environment. Most of the real-world images recorded by sensing devices are affected by various circumstances from which blurring is one of the factors that causes image degradation due to bad weather, poor illumination, improper focusing, or image denoisification, which needs to be enhanced [1]. Image enrichment or image high boosting is one of the key aspects in the fields of image processing and computer vision that amplifies higher frequency components in an image by keeping lower frequency components as they are. Moreover, the real-time problems like remote sensing, disaster monitoring, satellite broadcasting, and traffic monitoring use image enhancement techniques to boost the detail information present inside videos and also require their quicker execution at the same time [2–4]. Recent research on Image retrieval, representation and classification have shown lots of interest on image enhancement at their preprocessing stage. Content based image retrieval (CBIR) system built on color histogram and DWT extracts color and texture features [5], whereas CBIR based on SIFT and SURF extracts scale, rotation and illumination invariant features [6]. Bag of Features (BoF) representation in image retrieval system lacks in spatial information, which is improved with the help of histogram of

✉ L. M. Jenila Livingston, jenila.lm@vit.ac.in; Ashutosh Satapathy, ashutosh.satapathy2013@vit.ac.in | [1]SCSE, Vellore Institute of Technology, Chennai, India.

Received: 20 May 2019 / Accepted: 7 October 2019 / Published online: 16 October 2019

triangular regions [7]. Apart from the information retrieval, Histogram based image representation of BoVW model requires proper encoding of spatial information and it was addressed by Zafar et al. [8]. Hybrid geometric image representation of BoF model depends upon circular, triangular, and rectangular region based histograms over an image [9]. Recent image classification researches such as makeup invariant face recognition [10], pose invariant 3D face recognition [11], variance based facial image registration and recognition [12] or impact of asymmetric left and right faces for accurate age estimation [13] uses convolutional neural network to improve their accuracy level. In all the cases, Image quality improvement of a degraded image tries to raise the pixel values where the change of intensities happens immediately by keeping other pixel values as constant, i.e., pixels lie on the lines, curves, corners, or boundaries are the target areas for this enhancement [14, 15]. So, the image high-boosting techniques utilize different high-pass filters to extract high-frequency images, which are added to their blurred images to produce the final outputs. An image from CalTech Face 1999 dataset [16] is smoothed by a Gaussian low pass filter and its reconstructed image by a Butterworth high boost filter with cutoff frequency 50 and order 2 are shown in Fig. 1.

Image high-boost filtering is one type of image enhancement techniques, which can be accomplished in the spatial domain or frequency domain [17]. Quality enhancement in the frequency domain uses high boost filters which are of two types: linear and nonlinear [17–20]. OpenCL is from the Khronos Group, a nonprofit organization responsible for the creation, distribution, and maintenance of various applications in the field of parallel programming, multimedia, graphics, signal, image, and video processing. OpenCL kernel allows each work item inside all the compute units to execute an instance of it in a synchronous fashion so that the integrity of information is repeatedly maintained throughout the program execution. Before explaining two different types of kernel approaches, it requires a prior understanding of multiple models made inside the OpenCL building block along with four different frequency domain high boost filters normally used to raise the level of information presented inside this multimedia content. In the end, OpenCL kernels of those high-boost filters are designed for the faster image enrichment, which are again optimized using image vectorization techniques and tested on parallel computing platforms from Intel and NVIDIA.

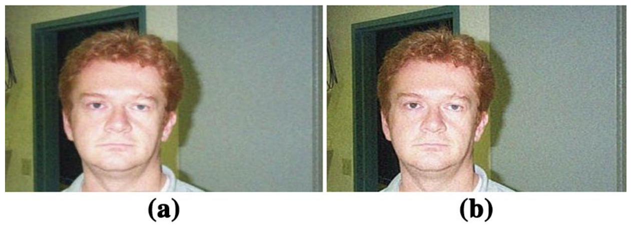


Fig. 1 An example of image high boosting: **a** blurred image; **b** its restored image. Image source: <http://www.vision.caltech.edu/html-files/archive.html>

2 Related work

Comparison between the various filters are done for removing the fractional Brownian noise in Brain MRI images whose produced outputs are low qualitative in nature and also consume large amount of time during their CPU implementation [21]. Shukla and Singh implemented frequency domain Gaussian high-boost filter using Matlab 7.8 for an image of size 128×128 pixels that consumes around 80 ms for high boosting of various noisy images [22]. Subsequently, Yano and Kuroki approximated the 2D Gaussian filter using multilayer convolution of multiple binomial filters enabled with basic shift and add operations for its faster implementation [23], while the Gaussian kernel of an edge preserving bilateral filter was approximated using raised cosines and MonteCarlo sampling takes around 17 s on a Intel 4-core machine for an image size of 512×512 pixels [24]. Nair and Sankaran presented a center surround filter to reduce the speed and memory requirement for color image dehazing in RGB, Lab and HSV color spaces, but its computation cost is still high for a small scaled image [25]. Preeti and vishvaksenan [26] have compared CPU and GPU implementation of Gaussian filter using OpenCV library packages. OpenCV enabled with CUDA implementation speed up the 2D filtration on a GPU than the CPU. Oza and Joshi also proposed a fast bilateral filter implemented using CUDA for medical image processing [27]. In both the cases filters are designed using CUDA, which makes them not suitable for other parallel platforms. Later Rakhshanfar and Amer created a cascaded 2D Gaussian filter to gain better image quality than the standardize Gaussian filter that consumes an significant amount of time for a 768×512 RGB image on a CPU, and GPU respectively and make it not suitable for a time bound real time application [28]. Due to the above limitations, Mukherjee and Mukhopadhyay come with two fast hardware architectures named Generalized Filter Architecture (GFA) and Separable Filter Architecture (SFA) for the 5×5 Gaussian filter and tested on a field programmable gateway array (FPGA) [29].

Enhancing a 480×320 image by the Butterworth high pass filter in frequency domain requires 0.702 s CPU time in Matlab [30], and Matlab 7.8 implementation of this filter requires roughly 90 ms to enhance a noisy 128×128 image on a 4 core device [22]. To enhance the quality of an image, Fan et al. worked on single-scale Retinex algorithm in HSV and RGB color spaces independently and the enhanced images from these color spaces are fused to compute the loss. Gaussian filter was replaced by the Butterworth filter in the Retinex algorithm for better image enhancement [31], but the complexity of the model made it slower than the brightness balancing method using the Butterworth filter, developed by Zhao et al. [32]. In 1987, Chen, Huertas and Medioni introduced the fast execution of a Laplacian of Gaussian (LoG) convolution mask of variance σ by decomposing into a Gaussian mask and a LoG mask of variance $\sigma_1 < \sigma$ [33], but Wu implemented the fast LoG convolution mask using the CUDA API targeted for only NVIDIA GPU devices [34]. CUDA implementation of Laplacian filter in the spatial domain speedup 200× on NVIDA GeForce GT 620 GPU than AMD Phenom II X4 810 CPU time [35]. Besides that, Bao and Sheng explained a computational expensive parameterized logarithmic method using LoG filter that divides an image into multiple equal size blocks for better edge and contrast enhancement [36]. Arif, Li and Cheng suggested a minutiae extraction algorithm (MEA) enabled with high boost filters for improved finger print recognition, while enhancement using Laplacian filter takes 0.354 s on a normalized finger print image from FVC2004 on a 4 threaded core i3 CPU [37]. Recently, Rafaela implemented the local Laplacian filter using Vulkan API and compared its performance speedup with respect to its OpenCL and OpenGL implementation for an 800×533 image [38].

3 Opencl architecture

Parallel processing allows a multiple compute units working synchronously or asynchronously to accomplish a particular task, which is impossible in a normal computing device having a few numbers of cores with a limited amount of memory and network bandwidth. Various parallel programming and API interfaces are available to make use of such heterogeneous parallel platforms, and some of them are OpenMP, MPI, OpenACC, CUDA, OpenCL, and Renderscripts, which deliver bit, instruction, data, and task level concurrency upon the heterogeneous parallel devices [39]. OpenCL is one of the open source programming standards for the heterogeneous parallel system like CPU, GPU, and FPGA from different vendors that allows programmers to develop efficient, reliable, and portable kernels, which can be switched from one device to another without any extra set of configurations. MAGMA, clAMDBLAS, clAMDFFT, BOLT C++, OpenCV, and JACKET like many libraries use the OpenCL programming interfaces for GPU acceleration as OpenCL built upon CUDA, CUBLAS, CUFFT, CUSPARSE, and Trust like a vast set of boosting libraries. Access to OpenCL native programming interface can also be done through Python, C/C++, and Java languages. The Khronos OpenCL programming framework is modularized into the platform, execution, memory, and programming models, and the detail explanation of those are presented in the upcoming subsection [40].

3.1 OpenCL platform model

OpenCL platform model includes multiple heterogeneous parallel devices like CPU, GPU, FPGA, or DSP from different brands, are connected to a single host machine for the shake off building a heterogeneous parallel environment. Every device is made of multiple computation units, which further consist of a bunch of processing elements or work items where the actual kernel execution happens simultaneously to make the whole operation faster. Data transmission speed must be fast enough between the host and device memories to mask the transmission bottleneck with the high computational capability of each streaming processor; in some cases, it is avoided by using shared memory communication between the host and OpenCL devices. Here, we have used Intel Xeon, HD Graphics P530, and NVIDIA GTX 1050 Ti devices for image high boosting, whose processing elements are segregated into 2, 6, and 4 computing units having 8192, 256, and 1024 work items respectively [41–43].

3.2 OpenCL execution model

OpenCL execution model mainly focuses on its two execution units, namely host and kernel program execution. Here, the kernel program is targeted for multiple OpenCL devices, while the host program runs on the host machine, but the calling of these kernel programs happen from the host program itself. Execution model projects all the processing elements present inside an OpenCL device into an N-dimensional index space, and the value of N can be varied between one and three based on the application requirement [40].

In image high boosting operation, the size of an index space is the total number of pixels constitute an input image and the index space creation happens during clEnqueueNDRange() function execution by the host program at the running time. Figure 2 shows, sixty-four work items are visualized in a 1D, 2D, and 3D index space for better understanding on the orientation of index space, and a single cube in all three index spaces represents a work unit, which carries three different coordinate values in these NDRange representations.

The OpenCL execution unit uses certain terms like work-item, work-group, global-id, local-id, work-group-id, global-size, and local-size; those play significant roles

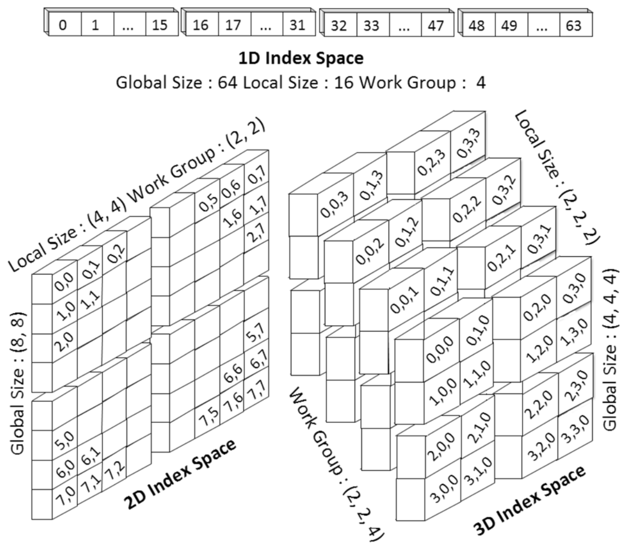


Fig. 2 1D, 2D, and 3D Index space of sixty-four work items

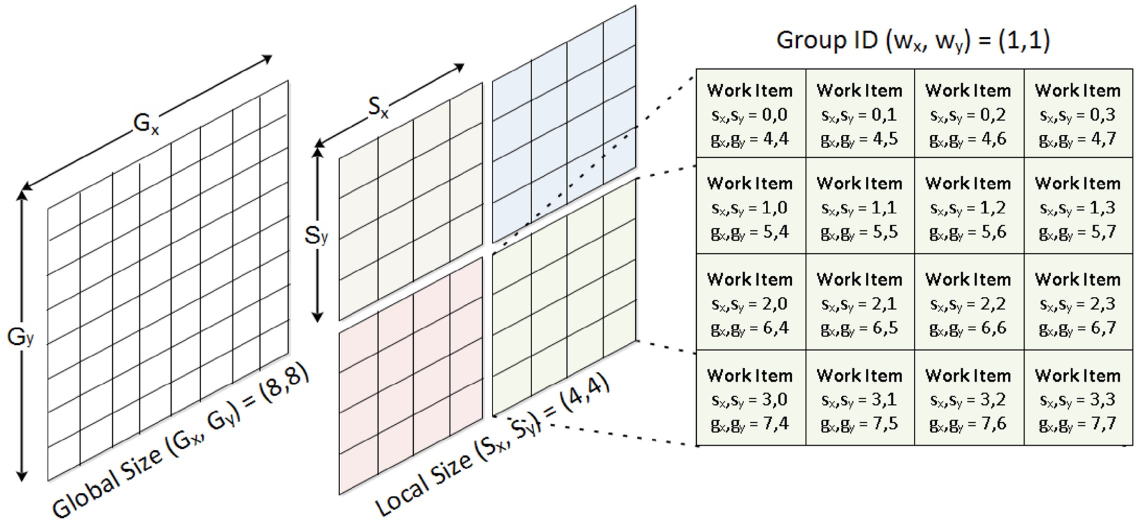
during kernel execution to deliver programming flexibility, reliability, robustness, and faster computation of instances on the index space. A work-item represents a processing element in an OpenCL device, which has a unique coordinate value in index space so-called its global-id. A cluster of work-items is dedicated to a distinct task based on a program requirement to form a work-group, and the coordinate of a work unit inside its work-group indicates its local-id. Like a work unit, a work-group has its work-group-id; a unique id denotes its place inside the NDRange, and its arrangement can happen in one, two, or three directions same as its index space dimension. Global-size of an index space is the total number of work-items span along its every dimension, whereas local-size stands for the number of work-items span along each direction of a work-group. These values are initialized and passed to every kernel module using clEnqueueNDRange() API in a host program, and for better understanding, these attributes are visualized in Fig. 3 using a two-dimensional index space comprised of 64 processing elements.

As shown in the above figure, 64 work items are organized in a 2D index space of its global-size (Gx = 8, Gy = 8), which is further divided into four work-groups of local-size ($S_x = 4$, $S_y = 4$) each, i.e., 16 work units span over a two-dimensional space to form a work-group. Here, (s_x, s_y) stands for local-id of a work-item in its work-group, whose global-id (g_x, g_y) calculation is completely dependent upon its work-group-id (w_x, w_y), work-group-size (S_x, S_y), and local-id (s_x, s_y) as given in Eq. 1.

$$\text{global\_id}\,(g_x,\ g_y) = \ (w_x \cdot S_x + s_x,\ w_y \cdot S_y + s_y) \tag{1}$$

3.3 OpenCL memory model

The OpenCL memory model divides the device memory into four regions based on their size, bandwidth, and their availability to different processing units [40]. Global memory is the largest memory region shared by all the processing units present inside different compute units, and any changes in the global memory by one work item make these pieces of information clearly visible to other work items too. The size of this memory is declared and passed as a kernel argument by host program where a variable is defined using the __global keyword inside the OpenCL kernel that catches the argument. The constant memory region is also shared by all the work items and initialized by the main program like global memory, but


Gx
Gy
Global Size (Gx, Gy) = (8,8)
Sx
Sy
Local Size (Sx, Sy) = (4,4)
Group ID (wx, wy) = (1,1)

Work Item s_x,s_y = 0,0 g_x,g_y = 4,4	Work Item s_x,s_y = 0,1 g_x,g_y = 4,5	Work Item s_x,s_y = 0,2 g_x,g_y = 4,6	Work Item s_x,s_y = 0,3 g_x,g_y = 4,7
Work Item s_x,s_y = 1,0 g_x,g_y = 5,4	Work Item s_x,s_y = 1,1 g_x,g_y = 5,5	Work Item s_x,s_y = 1,2 g_x,g_y = 5,6	Work Item s_x,s_y = 1,3 g_x,g_y = 5,7
Work Item s_x,s_y = 2,0 g_x,g_y = 6,4	Work Item s_x,s_y = 2,1 g_x,g_y = 6,5	Work Item s_x,s_y = 2,2 g_x,g_y = 6,6	Work Item s_x,s_y = 2,3 g_x,g_y = 6,7
Work Item s_x,s_y = 3,0 g_x,g_y = 7,4	Work Item s_x,s_y = 3,1 g_x,g_y = 7,5	Work Item s_x,s_y = 3,2 g_x,g_y = 7,6	Work Item s_x,s_y = 3,3 g_x,g_y = 7,7

Fig. 3 OpenCL execution model

the information present inside the constant memory is immutable throughout the kernels' execution. The __const keyword is used to capture the constant variable inside the kernel normally declared by the host program, whose CL_MEM_READ_ONLY flag is set to one for read-only access by various processing units. Apart from global and constant memories, each work-group is well linked with its own local memory. Each work-item presents inside a single work-group share its local memory, and its closeness to processing elements make it faster than the previous two memory regions, which is initialized by a keyword called __local or local. In addition to that, every processing element is associated with a set of registers called its private memory, and read or write to this memory by a work item is not visible to other work items in the same or from different work-group. In OpenCL memory model, speed of a region is reciprocal to the size of this region, i.e., the largest size of global memory makes it slower than other memory regions, while smaller and adjacency private memory is fastest among other memory regions, whose transmission speed is about 1 TB/s (Fig. 4).

Overall, OpenCL enables a programmer for proper division of information, while maintaining the synchronization between these memory regions, which is normally done through an explicit action so that information reaches to work items correctly at each stage of processing.

3.4 OpenCL programming model

An OpenCL programming model is responsible for the creation of program and kernel objects before the device execution and manages them, while their execution happens in parallel on different work items in a device. A host application can have more than one program objects dedicated for separate contexts, and each program objects can initiate multiple kernel objects committed for different functionalities in the applications. A declaration of a program object is made using the cl_program keyword, and it can be created from source code using clCreateProgramwithSource() function or from a binary file using clCreateProgramWithBinary() function. At last, the program is built by clBuildProgram() function where both compilation and linking happen; if the program object is created using the first API, otherwise only linking is performed in case of second API [44]. OpenCL programming model splits a multithread program into a set of threads and distributes them across the cores in an OpenCL device. So, a GPU having many cores always executes an application faster than a CPU or DSP with fewer numbers of cores.

4 Frequency domain image highboosting

Image high boosting is a branch of image enhancement operation used to improve high-frequency regions of an image while keeping lower frequency regions as they are, i.e., it is an arithmetic addition between an input image and its scaled enhanced image produced by a high-pass filter [45]. A frequency domain filtering works on a transformed image that multiplies with a high pass filter matrix to produce the frequency map in the transform domain which is then transformed back to yield the final one. In the frequency domain filtering, an MxN image is first transformed using various transformation techniques to create an array of MxN frequency coefficients, representing the rate of change in pixel intensities at each position on the input image [46]. An MxN high-pass filter matrix is multiplied with the MxN transformed image matrix, and the resulted matrix is

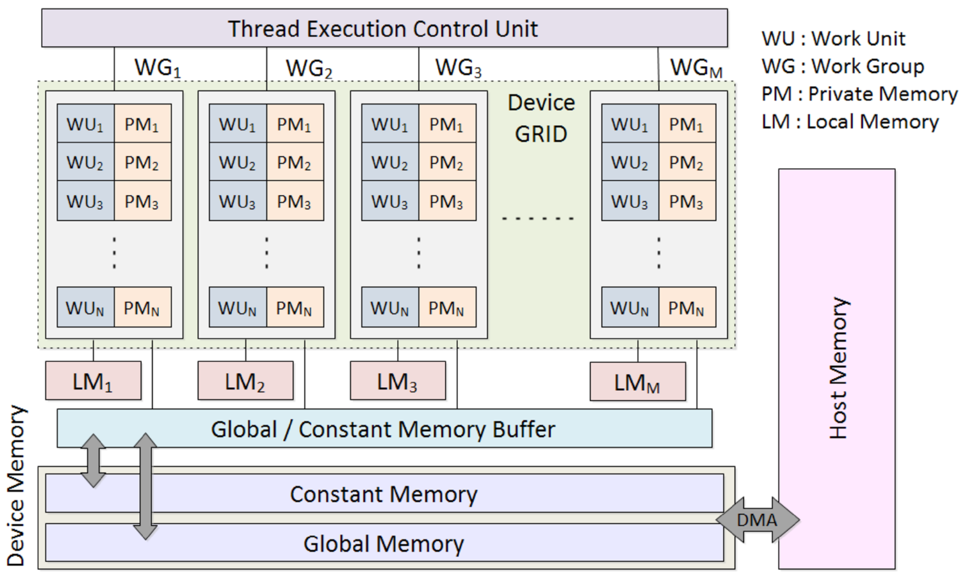


Fig. 4 OpenCL memory model

converted back using the inverse transformation technique to get an enhanced image in the spatial domain.

Mostly used DFT technique uses sinusoidal functions to describe clearly visible frequency values of an image that influence image quality in the spatial domain without capturing all frequency components in the image [47]. In high-pass filtering, selected lower frequency coefficients are excluded from the transformed image by equalizing them to zero, and selection of lower frequency is strictly dependent on the filter's cutoff frequency, which is computed by taking the distance between each coordinate and DC component in two-dimensional frequency space [48, 49]. As all the image high boost filtering operation includes high pass filters, mostly used four high pass filters and their transfer functions in frequency domain are narrated below.

4.1 Ideal filter

An ideal high-pass filter rejects all the frequency components below its cutoff frequency by equating them to zero without altering high frequencies in an image. The rectangular frequency response of this filter prevents any existence of a transition zone between its stop bands and pass band regions, unlike other practical high pass filters [50]. The transfer function H(k, l) value is suddenly raised to one after its cutoff frequency, and a strict removal of frequency components below the cutoff frequency brings some sort of distortion in the texture of the output image. The Point Spread Function (PSF) of an Ideal high-pass filter is given in Eq. 2.

$$\mathrm{H(k,\ l)} = \begin{cases} 0 & \mathrm{D(k,l)} < \mathrm{D_f} \\ 1 & \mathrm{D(k,l)} \geq \mathrm{D_f} \end{cases} \tag{2}$$

4.2 Gaussian filter

The behavior of a Gaussian high-pass filter is like a bell-shaped curve, whose response increases gently even after the cutoff frequency to avoid such distortion in the case of an Ideal filter. Unlike the ideal filter, the transition region between its pass band and stop band is improved gradually until its value equals to one, and length of its transition region reduces with the increase of its cutoff frequencies [51, 52]. The transfer function of a Gaussian high-pass filter is given in Eq. 3.

$$\mathrm{H(k,\ l)} = 1 - \mathrm{e}^{\frac{-\mathrm{D^2(k,l)}}{2\mathrm{D_f^2}}} \tag{3}$$

4.3 Butterworth filter

A Butterworth high-pass filter allows frequencies above its cutoff frequency in such a way that steady growth in the response from fractional to uniform between its stop band and pass band. As the cutoff frequency increases, It brings its performance closer to an Ideal filter that obeys strict rejection of frequency components below its cutoff frequency [51, 52].

$$\mathrm{H(k,\ l)} = \frac{1}{1 + \left[\frac{\mathrm{D_f}}{\mathrm{D(k,l)}}\right]^{2n}} \tag{4}$$

In the first order Butterworth filter, the roll-off rate from pass band to stop band is 6 dB/octave (20 dB/decade); it increases to 12 dB/octave (40 dB/decade) for a second order and even improves to 24 dB/octave (80 dB/decade) for a fourth order filter.

4.4 Laplacian of Gaussian filter

Laplacian of Gaussian (LoG) filter's frequency response is the second order derivative of Gaussian function that highlights the sections where the change of intensity levels happened frequentlyWhen a change in the pixel intensities arises on the input, its response is +ve on the darker side and –ve on the lighter side that creates a thin edge between these two sides [53]. The PSF of a LoG high-pass filter having cutoff frequency $\mathrm{D_f}$ is mathematically expressed in Eq. 5.

$$\mathrm{H(k,\ l)} = \left[\frac{D^2(k,l)}{2D_f^2} - 1\right] e^{\frac{-D^2(k,l)}{2D_f^2}} \tag{5}$$

In this filtering operation, an image is first filtered using a Gaussian filter to clear any unwanted noises; those can be sensitive for finding Laplacian zero crossings in the image. Smooth, localize, and separable nature of the Gaussian filter removes false edges and minimizes error while maintaining the computational efficiency.

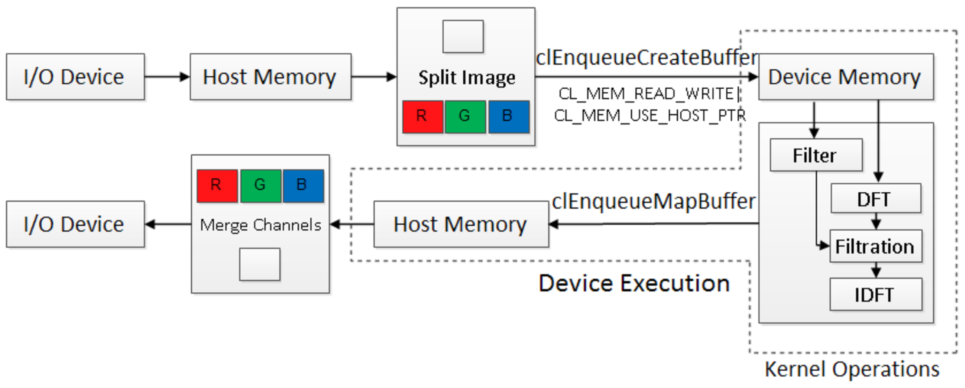
5 Opencl kernel approaches

OpenCL kernel objects are initialized using a program object with the help of the __kernel qualifier, created using clCreateKernel() function and passed as an argument using clSetKernelArg() API by the host program [44]. After successfully passing all the device buffers and kernel name as arguments, clEnqueueNDRangeKernel() API executes such kernel by creating a single instance per work unit in the NDRange. All the kernels written in C languages are targeted for C99 compiler, whose datatypes are inspired directly from C basic data types and some vector data types like float2, int3, and char4, which are obtained by combining multiple basic data types in a single container [54]. As kernels are compiled and built during host program execution; writing a kernel requires clear attention because a minor fault causes difficult to identify errors exist in it.

Figure 5 shows, input images stored on the host memory are coming from various input devices like external media drives, visual sensors, or any real-time applications. In a channelized OpenCL kernel operation, a single RGB image is first captured using an OpenCV cv::mat() container and then given to cv::split() library function, which extracts red, green, and blue channels of a colored image and saves them in different cv::mat() variables. Later, three independent OpenCL cl::clEnqueueCreateBuffer() functions transfer these channels' information from host memory to device memory, whose parameters CL_MEM_USE_HOST_PTR and CL_MEM_READ_WRITE create three device buffers and maps the information from host buffers to the corresponding device buffers [55]. Multiple channelized OpenCL kernels are executed for the creation of frequency domain image matrices from the spatial image matrices using Discrete Fourier Transformation (DFT), filtration, and Inverse Discrete Fourier Transform (IDFT) for transforming them back to the spatial domain after the filtration operation. Apart from that, there is a separate OpenCL kernel for the creation of the filter matrix, which will be applied independently to the transformed channels' data after the OpenCL DFT kernel execution. This filter creation kernel runs separately from the image filtration process to reduce the time consumption, which is done at an early stage of DFT kernel execution. After the DFT transformation, filter matrix is multiplied with transformed matrices to boost the high-frequency components of these channels without changing their lower frequency components, and IDFT Transformation is taken on those restored channels to convert them back into spatial domain image matrices. After those successive kernel execution, all the channels are copied to the host buffer using cl::clEnqueueMapBuffer() function, and the enhanced red, green, and blue channels are merged using cv::merge() function to generate the final enhanced RGB image, which is then sent to the output device for displaying purposes [55]. Here, the device execution time incorporates the total amount of time taken between transferring the channels' information from host memory to device memory and sending back the modified channels' information to host memory after the image enhancement, whereas the whole image high boosting operation processing time contains the time span between an RGB image's split() and merge() operations.

Unlike the channelized OpenCL kernel operation, an RGB image saved on the host memory is moved to an OpenCV cv::mat() container, which is then given to cv::cvtColor() function to generate RGBA image and store them in a separate mat() variable. OpenCL cl::clCreateImage() library function with parameters CL_MEM_READ_WRITE and CL_MEM_USE_HOST_PTR initiates an image buffer and maps information from host buffer to it (Fig. 6). Like channelized OpenCL kernels, here also multiple kernels are used for filter matrix creation, DFT, and IDFT conversion of the image matrix and filtration of the image matrix, but a single kernel is run for each RGBA image instead of a distinct kernel for each channel of an image. OpenCL read_only image and write_only image data types are used to capture input and output images, whose image_format having image_channel_order is set to CL_RGBA [56]. The OpenCL DFT kernel generates frequency domain transformed image matrix from its spatial matrix, and its output is propagated to filtration kernel that also receives filter matrix as one of its inputs. The OpenCL filter kernel runs simultaneously with the DFT kernel to create the filter matrix, whose output is later supplied as an input to filtration kernel. After multiplying the image matrix with a transformed image matrix, IDFT kernel converts the transformed image matrix to its spatial domain. All OpenCL kernels use read_imagef() function to capture channels' pixel values in a float4 vector data type variable at a particular position. Unlike channelized OpenCL kernel, all the operations are performed on RGBA channels' values at a particular instance using a float4 variable, whose make the kernel execution time faster than three simple

Fig. 5 Dataflow diagram of channelized OpenCL kernels' device execution



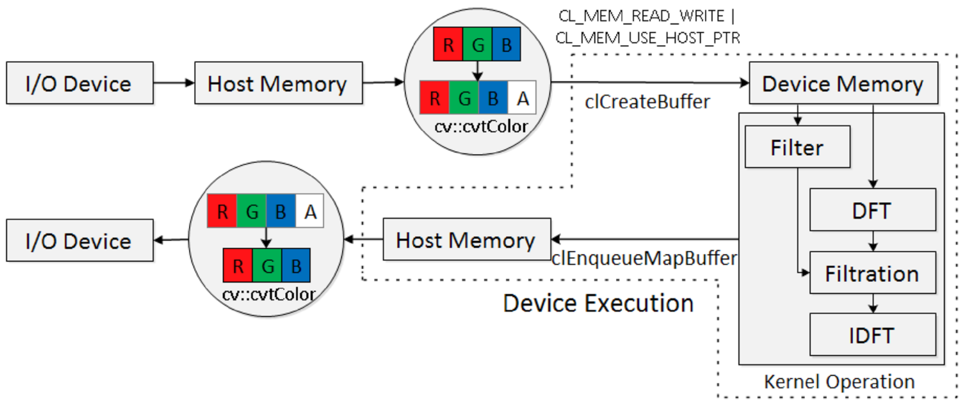


Fig. 6 Dataflow diagram of vectored OpenCL kernels' device execution

OpenCL kernels for three channels respectively. After all successive kernels executions, the enhanced image information is mapped back to the host buffer using clEnqueueMapBuffer(), and it is then provided as an input to cv::cvtColor() for converting it back from RGBA to RGB image [55]. At last, the final image is propagated to the output device to store or exhibit purposes. Like previous kernel executions, here the total execution time of an image high boosting is the time gap between two cv::cvtColor() functions, which includes device execution time starts by transferring image data from host buffer to device buffer and ends after transfer it back to host buffer. Overall, as the image vectored OpenCL kernel operates on all the channel values at the same time, it not only optimizes program execution but also reduces time complexity without hampering the image quality.

Tables 1 and 2 present the OpenCL kernel implementation for Ideal, Gaussian, Butterworth, and Laplacian of Gaussian high-boost filters using channelization and image vectorization techniques. As we have seen, the two-dimensional float2* data pointer variable in a channelized kernel is used to hold the real and imaginary part of a single channel in an image, whereas the float4 type ipixelValueR and ipixelValueI capture and process the real and imaginary part of four channels in an RGBA image, which reduce separate invocation of the channelized OpenCL kernel for each channel of a colored image to one. So, the computational cost involved with the host program while three times passing the kernel parameters and queuing the kernels to index space during channelized kernel implementation are successfully eliminated by the employment of the image vectorized kernels. In the end, the __read_only image2d_t and __write_only image2d_t

Table 1 Channelized OpenCL kernel of the above high-boost filters

Ideal	__kernel void ideal_kernel (__global float2* data,int height,int width,int CUTOFF)
	{ uint index = get_global_id(0); int U = index / width; int V = index % width;
	  float D = pow(height/2-abs(U-height/2),2.0)+pow(width/2-abs(V-width/2), 2.0);
	  float H = 1.0+((sqrt(D)>CUTOFF)?1.0:0.0); data[index].x = data[index].x * H;
	data[index].y = data[index].y * H; }

Gaussian	__kernel void gauss_kernel (__global float2* data,int height,int width,int CUTOFF)
	{ uint index = get_global_id(0); int U = index / width;int V = index % width;
	  float D = pow(height/2-abs(U-height/2),2.0)+pow(width/2-abs(V-width/2),2.0);
	  float H = 2.0 - pow(2.72, (-1.0 * D / (2.0 * pow(CUTOFF, 2.0))));
	  data[index].x = data[index].x * H; data[index].y = data[index].y * H;}

Butterworth	__kernel void bw_kernel (__global float2* data, int height, int width,
	                         int CUTOFF, float Ord)
	{ uint index = get_global_id(0); int U = index / width; int V = index % width;
	  float D = pow(height/2-abs(U-height/2),2.0)+pow(width/2-abs(V-width/2),2.0);
	  float H = 1.0 + 1.0 / (1 + pow (CUTOFF / sqrt(D), 2 * Ord));
	  data[index].x = data[index].x * H; data[index].y = data[index].y * H; }

LoG	__kernel void LoG_kernel (__global float2* data,int height,int width,int CUTOFF)
	{ uint index = get_global_id(0); int U = index / width; int V = index % width;
	  float Freq = pow(CUTOFF, 2.0);
	  float D = pow(height/2-abs(U-height/2),2.0)+pow(width/2-abs(V-width/2),2.0);
	  float H = 2.0 - (1.0 - D / Freq) * pow(2.72, -1.0 * D / (2.0 * Freq));
	  data[index].x = data[index].x * H; data[index].y = data[index].y * H;}

Table 2 Vectored OpenCL kernel of the above high-boost filters

Ideal

```
__kernel void ideal_kernel (__read_only image2d_t orimage,__read_only image2d_t
oiimage,__write_only image2d_t trimage,__write_only image2d_t tiimage, int height,
int width, int CUTOFF) { uint index = get_global_id(0);int u = index / width;
 int v = index % width; float4 ipixelValueR, ipixelValueI;
 float D = pow(height/2-abs(u-height/2),2.0) + pow(width/2-abs(v-width/2),2.0);
 float H = 1.0 + ((sqrt(D) > CUTOFF)? 1.0 : 0.0);
 ipixelValueR = read_imagef(orimage, image_sampler, (int2)(u, v));
 ipixelValueI = read_imagef(oiimage, image_sampler, (int2)(u, v));
 write_imagef(trimage, (int2)(u, v), ipixelValueR * H);
 write_imagef(tiimage, (int2)(u, v), ipixelValueI * H);}
```

Gaussian

```
#define EXP 2.72
__kernel void gaussian_kernel (__read_only image2d_t orimage, __read_only
image2d_t oiimage,__write_only image2d_t trimage,__write_only image2d_t tiimage,
int height,int width, int CUTOFF)
{ uint index = get_global_id(0); int u = index / width;int v = index % width;
  float4 ipixelValueR, ipixelValueI;
  float D = pow(height/2-abs(u-height/2),2.0)+pow(width/2-abs(v-width/2),2.0);
  float H = 2.0 - pow(EXP, (-1.0 * D / (2.0 * pow(CUTOFF, 2.0))));
  ipixelValueR = read_imagef(orimage, image_sampler, (int2)(u, v));
  ipixelValueI = read_imagef(oiimage, image_sampler, (int2)(u, v));
  write_imagef(trimage, (int2)(u, v), ipixelValueR * H);
  write_imagef(tiimage, (int2)(u, v), ipixelValueI * H); }
```

Butterworth

```
__kernel void butterworth_kernel (__read_only image2d_t orimage,__read_only
image2d_t oiimage,__write_only image2d_t trimage,__write_only image2d_t tiimage,
int height,int width, int CUTOFF,float Ord)
{ uint index = get_global_id(0); int u = index / width;int v = index % width;
  float4 ipixelValueR, ipixelValueI;
  float D = pow(height/2-abs(u-height/2),2.0)+pow(width/2-abs(v-width/2),2.0);
  float H = 1.0 + 1.0 / (1 + pow (CUTOFF / sqrt(D), 2 * Ord));
  ipixelValueR = read_imagef(orimage, image_sampler, (int2)(u, v));
  ipixelValueI = read_imagef(oiimage, image_sampler, (int2)(u, v));
  write_imagef(trimage, (int2)(u, v), ipixelValueR * H);
  write_imagef(tiimage, (int2)(u, v), ipixelValueI * H);}
```

LoG

```
#define EXP 2.72
__kernel void LoG_kernel (__read_only image2d_t orimage, __read_only image2d_t
oiimage,__write_only image2d_t trimage,__write_only image2d_t tiimage,int height,
int width, int CUTOFF)
{ uint index = get_global_id(0); int u = index / width; int v = index % width;
  float Freq = pow(CUTOFF, 2.0); float4 ipixelValueR, ipixelValueI;
  float D = pow(height/2-abs(u-height/2),2.0)+ pow(width/2-abs(v-width/2), 2.0);
  float H = 2.0 - (1.0 - D / Freq) * pow(EXP, -1.0 * D / (2.0 * Freq));
  ipixelValueR = read_imagef(orimage, image_sampler, (int2)(u, v));
  ipixelValueI = read_imagef(oiimage, image_sampler, (int2)(u, v));
  write_imagef(trimage, (int2)(u, v), ipixelValueR * H);
  write_imagef(tiimage, (int2)(u, v), ipixelValueI * H);}
```

inside the filter kernels in Table 2 are the data types for the input and output images that map the pixels in two-dimensional index space.

6 Results

In Image high boosting, a filter is applied one time on the transformed data irrespective of its cutoff frequency and degree, which not only reduces the overall computational complexity but also makes the operation quite faster than the spatial domain image filtration. All those four filters were first implemented using C and OpenCV on Xeon E3 1225 V5, and later, the channelized and vectored OpenCL kernels were created and run on Xeon E3 1225 V5, HD Graphics P530, and GTX 1050 Ti devices while maintaining the accuracy level in all those implementations. The dedicated framework on which all the high boost operations are carried out listed in Table 3, and various blurred images were used during the enhancement operation to produce their enhanced images displayed in Fig. 7.

In Fig. 7a–e images are the distorted images, which are first added with Poisson, Gamma, Exponential, Uniform, and impulse noises separately, then these noisy images are filtered by Ideal, Gaussian and, Butterworth low-pass filters. The detail information regarding the parameters of these noises and the corresponding applied filters are presented below of each image. The motto behind the use of

Table 3 Framework specification

Components	Specification
Processor	Intel Xeon E3-1225 v5
Memory	DDR4 8 GB 2133 MHz
Graphics processor (integrated)	Intel HD Graphics P530
Graphics processor (dedicated)	NVIDIA GTX 1050 Ti
Operating system	Windows Server 2012 R2
Packages	C, OpenCV, OpenCL
Image	Size = 256 × 256, Bit Depth = 24.

these distorted smooth images to check the accuracy level of those above high-boost filters with respect to outputs and their respective originals. For OpenCL DFT, IDFT, and filtering operations, global_size of the index space is the total number of pixels present in the input image, but the local_ size reduces to the total pixels along the horizontal and vertical direction of the workgroup. These global_size and local_size values set to 65,536 and 256 for a 256 × 256 image in the case of vectored filter kernel, but the local_ size sets to 1024 for Xeon E3 1225 V5 and GTX 1050 Ti in the case of channelized filter kernel implementation. For our evaluation, the following sample codes inside the host program set the size of the index space and each work-group present inside it.

`size_t global_size = Image.rows * Image.cols, size_t local_size = 256; /*DFT & IDFT*/`

size_t global_size = `Image.rows * Image.cols, size_t local_size = 256; /*Filtering*/`

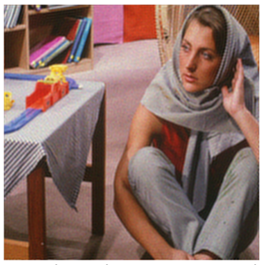
clStatus = `clEnqueueNDRangeKernel(command_queue_highboost, kerne, 1, NULL, &global, &local, 0, NULL, NULL);`

Figure 7f–j are the sample output images from the Ideal, Gaussian, Butterworth or LoG high-boost filters at different cutoff frequencies. Various image performance evaluation metrics were used to estimate correctness of the produced images with the help of the above four filters with regard to their inputs; higher values of signal-to-noise ratio (SNR), peak signal-to-noise ratio (PSNR), Entropy, Correlation, and Structural Similarity Index (SSIM) and lower mean absolute error (MAE), standard deviation (SD), and percentage fit error (PFE) values indicate better image enhancement [57, 58]. Table 4 carries the metric values, computed over those five distorted smoothed images by applying an Ideal filter at cutoff frequencies 30, 40 and 50 i.e., it represents the qualitative nature of an Ideal filter on various noisy environments. Table 5 lists out, the computational cost of various implementations of an Ideal filter at the same cutoff frequencies on heterogeneous platforms. It indicates the total time taken by an Ideal filter for a particular image at a given cutoff frequency on different platforms. The OpenCV-C implementation of an Ideal filter is dedicated to CPU only, whereas the non-vectorization (channelization) and vectorization implementation of this filter are targeted to CPU and GPU from different vendors. As we have moved to better platforms, the ratio between the channelized and vectorized kernel's execution time increases at a significant amount. Similarly, Tables 6, 8 and, 10 explain performance metrics of the Gaussian, Butterworth and, LoG filters, while Tables 7, 9 and, 11 describe the evaluation time with regard to each and every images. At last, Table 12 sum up the mean metric values of all the five images from those above filters.

As explained earlier, Tables 4, 6, 8 and 10 convey the performance metrics of output images from the five images by the Ideal, Gaussian, Butterworth, and LoG filters having cutoff frequencies at 30, 40, and 50, whereas Tables 5, 7, 9 and 11 exhibit the time consumption by the channelized and vectored kernels of those filters at 30, 40, and 50 for those five images. Table 12 shows the overall Correlation, SSIM, SNR, and PSNR values increase with an increase of the cutoff frequency of an Ideal filter, but PFE, SD, MSE, RMSE, MAE, and Entropy values decrease at the same time. The rigid frequency response of an Ideal high-boost filter brings some sorts of artifacts in the final image, which is not present in the original one as shown in Fig. 7f. The higher standard deviation at cutoff frequency 30 indicates better detail enhancement than the cutoff frequency at 50, i.e., as the cutoff frequency reduces, the relationship between the pixels is loosely maintained in the enhanced image than its source image. The entropy of an image specifies the amount of information is needed to successfully encode an image and its value 0.75792 at (D_f = 30) implies more information is required than 0.75434 at (D_f = 50) in case of the Ideal Filter.

A LoG filter sharpens edges and curves present inside an image, which ultimately boosts the overall image quality, but the similarity between the input and output has slightly destroyed due to the excessive improvement of high-frequency regions. As given in Table 12, the source images are quietly preserved during high boosting operation by the Gaussian high-boost filter contradict to other filters. As we have seen in Table 12, SSIM and PSNR values (0.5999, 0.6108, 0.6197) and (16.34, 16.54, 16.64) respectively are strong enough compare to other. In contrast to that, LoG's Entropy and SD values are (7.5936, 7.59646, 7.59042) and (57.18, 55.29, 53.84) demonstrate better contrast enhancement. At the same time, SSIM and PSNR values are (0.5597, 0.5668, 0.5759) and (15.65, 15.85, 16.04) signify the low restoration of the original image quality due to high contrast enhancement. A Butterworth filter tries to make a tradeoff between the output image quality from a Gaussian filter and enrichment of high-frequency regions by a LoG filter (Fig. 8). In the end, the first order

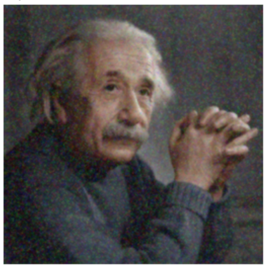
Fig. 7 Enhanced images **f**–**j** generated from their distorted smoothed images **a**–**e** with the help of Ideal, Gaussian, Butterworth and LoG high boost filters


(a) Image dependent Poisson noise having scale factor = e^{12} filtered by Ideal low pass filter at D_f = 90.

(b) Gamma noise having range [2, 5] and scale factor = 0.1 filtered by Gaussian low pass filter at D_f = 70.

(c) Exponential noise having A=1 and scale factor = 0.2 filtered by Gaussian low pass filter at D_f = 70.

(d) Uniform noise having range [0, 1] and scale factor = 0.3 filtered by Butterworth low pass filter at D_f = 50 and n = 1.


(e) Impulse noise 20% of total pixel having range [0, 255] filtered by Butterworth low pass filter at D_f = 90 and n = 2.

(f) Ideal high boost filter, D_f = 30.

(g) Gaussian high boost filter, D_f = 40.

(h) Butterworth high boost filter, D_f = 30, n = 1.

(i) Butterworth high boost filter, D_f = 50, n = 2.

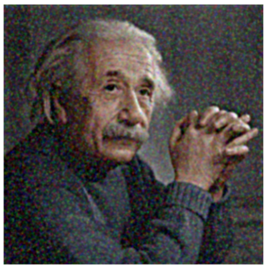

(j) LoG high boost filter, D_f = 30

Table 4 Performance metrics of an Ideal high boost filter with respect to frequency $D_f = 30$, 40 and 50

C. Freq.	Images	Entropy	MAE	MSE	RMSE	SD	SNR	PSNR	PFE	SSIM	COR
($D_f = 30$)	7a	7.7983	15.340	450.139	21.216	56.41	15.38	21.60	3.435	0.6959	0.9269
	7b	7.4957	63.283	4.35e03	65.989	47.90	04.33	11.74	73.37	0.6494	0.9436
	7c	7.7872	51.620	3.29e03	57.332	62.89	06.24	12.96	51.16	0.5897	0.9406
	7d	7.7709	50.677	3.04e03	55.108	60.24	06.65	13.30	50.52	0.5747	0.9692
	7e	7.0439	17.265	513.203	22.654	35.03	09.57	21.03	25.66	0.4175	0.9037
($D_f = 40$)	7a	7.7656	14.089	390.330	19.757	54.83	16.00	22.22	3.198	0.7053	0.9341
	7b	7.4758	63.374	4.32e03	65.771	46.86	04.36	11.77	73.46	0.6566	0.9504
	7c	7.8003	51.661	3.23e03	56.868	61.61	06.31	13.03	55.29	0.5977	0.9476
	7d	7.7677	50.724	2.98e03	54.636	59.14	06.72	13.38	50.59	0.5834	0.9756
	7e	7.0027	17.114	493.308	22.210	34.34	9.748	21.20	25.64	0.4293	0.9119
($D_f = 50$)	7a	7.7379	13.139	352.755	18.782	53.86	16.44	22.66	3.135	0.7144	0.9392
	7b	7.4589	63.418	4.30e03	65.603	46.15	04.38	11.79	73.49	0.6635	0.9556
	7c	7.7966	51.664	3.19e03	56.470	60.73	06.37	13.09	55.32	0.6052	0.9531
	7d	7.7620	50.752	2.95e03	54.341	58.52	06.77	13.42	50.62	0.5923	0.9798
	7e	6.9713	17.028	475.972	21.817	33.89	09.90	21.35	25.63	0.4430	0.9193

Table 5 Computation time of an Ideal high boost filter in seconds with respect to $D_f = 30$, 40 and 50

C. Freq.	Images	OpenCV-C Xeon E3- 1225	OpenCL-C Xeon E3-1225 Channelization	OpenCL-C Xeon E3-1225 Vectorization	OpenCL-C HD Graphics P530 Channelization	OpenCL-C HD Graphics P530 Vectorization	OpenCL-C GTX 1050 Ti Channelization	OpenCL-C GTX 1050 Ti Vectorization
($D_f = 30$)	7a	0.154	0.126	0.082	0.056	0.031	0.019	0.009
	7b	0.158	0.126	0.085	0.055	0.031	0.018	0.009
	7c	0.157	0.127	0.084	0.055	0.031	0.019	0.009
	7d	0.161	0.126	0.085	0.056	0.031	0.018	0.010
	7e	0.159	0.127	0.083	0.055	0.030	0.018	0.009
($D_f = 40$)	7a	0.152	0.124	0.082	0.055	0.030	0.018	0.009
	7b	0.154	0.125	0.083	0.055	0.031	0.018	0.009
	7c	0.153	0.125	0.083	0.054	0.031	0.018	0.009
	7d	0.155	0.124	0.084	0.055	0.031	0.017	0.009
	7e	0.154	0.125	0.083	0.054	0.030	0.018	0.009
($D_f = 50$)	7a	0.150	0.124	0.081	0.053	0.030	0.018	0.008
	7b	0.149	0.123	0.082	0.054	0.030	0.017	0.009
	7c	0.148	0.125	0.082	0.054	0.031	0.017	0.008
	7d	0.151	0.124	0.082	0.054	0.031	0.017	0.009
	7e	0.148	0.124	0.081	0.053	0.030	0.017	0.008

Butterworth filter performs a little better than its second order and much better than its higher-order at a given D_f.

OpenCV implementations of all those above filters consume significant amounts of times on Xeon E3 1225, which reduce with the increase of its cutoff frequency D_f except for the LoG filter. In case of a LoG high-boost filter, there is an increase in the computational cost with regard to the cutoff frequency, while high-boosting using a higher-order Butterworth filter runs a bit longer than its lower order as shown in Fig. 9. Here, a first-order Butterworth filter having a cutoff frequency at ($D_f = 30$, $D_f = 40$) consumes (0.1446 s, 0.1318 s) on CPU, but it again increases to 0.1384 s for this filter of order two and cutoff frequency at 50. The Channelized OpenCL implementations of these filters approximately reduce the overall computational costs by 20 percentages on the CPU, which diminish further by factors of 3 and 9 on Intel HD Graphics P530 and NVIDIA GTX 1050 Ti respectively. As the channelized kernel implementations are called separately for Red, Green, and Blue channels by the host program, context switching between the host and device create a bottleneck on their execution time,

Table 6 Performance metrics of a Gaussian high boost filter with respect to frequency $D_f = 30$, 40 and 50

C. Freq.	Images	Entropy	MAE	MSE	RMSE	SD	SNR	PSNR	PFE	SSIM	COR
($D_f = 30$)	7a	7.7881	13.794	383.599	19.586	56.55	16.08	22.29	3.160	0.7164	0.9381
	7b	7.4665	63.325	4.32e03	65.708	48.04	04.37	11.77	73.42	0.6583	0.9543
	7c	7.7857	51.608	3.22e03	56.785	63.15	06.32	13.04	63.14	0.6002	0.9516
	7d	7.7733	50.676	3.00e03	54.777	60.69	6.704	13.36	50.57	0.5897	0.9775
	7e	7.0256	16.783	480.418	21.918	35.01	9.863	21.31	25.61	0.4351	0.9155
($D_f = 40$)	7a	7.7521	12.708	337.615	18.374	54.99	16.63	22.85	3.122	0.7281	0.9432
	7b	7.4502	63.398	4.29e03	65.535	46.98	04.39	11.80	73.48	0.6670	0.9592
	7c	7.7908	51.645	3.17e03	56.333	61.72	06.39	13.11	55.31	0.6099	0.9571
	7d	7.7691	50.733	2.96e03	54.403	59.56	06.76	13.41	50.62	0.5993	0.9815
	7e	6.9862	16.777	465.161	21.567	34.33	10.00	21.45	25.61	0.4500	0.9224
($D_f = 50$)	7a	7.7256	11.966	310.032	17.608	54.03	17.00	23.22	3.104	0.7371	0.9467
	7b	7.4359	63.430	4.28e03	65.417	46.29	04.41	11.82	73.50	0.6733	0.9626
	7c	7.7843	51.662	3.14e03	56.009	60.77	06.44	13.16	55.34	0.6173	0.9611
	7d	7.7628	50.763	2.93e03	54.168	58.88	06.80	13.45	50.63	0.6071	0.9841
	7e	6.9581	16.791	453.077	21.569	33.89	10.11	21.57	25.61	0.4640	0.9280

Table 7 Computation time of a Gaussian high boost filter in seconds with respect to $D_f = 30$, 40 and 50

C. Freq.	Images	OpenCV-C Xeon E3- 1225	OpenCL-C Xeon E3-1225 Channelization	OpenCL-C Xeon E3-1225 With Vectorization	OpenCL-C HD Graphics P530 Channelization	OpenCL-C HD Graphics P530 With Vectorization	OpenCL-C GTX 1050 Ti Channelization	OpenCL-C GTX 1050 Ti With Vectorization
($D_f = 30$)	7a	0.158	0.126	0.085	0.056	0.032	0.019	0.010
	7b	0.158	0.127	0.082	0.055	0.031	0.020	0.009
	7c	0.159	0.127	0.083	0.056	0.030	0.019	0.010
	7d	0.160	0.126	0.082	0.055	0.029	0.020	0.010
	7e	0.157	0.126	0.085	0.055	0.031	0.019	0.010
($D_f = 40$)	7a	0.151	0.125	0.084	0.055	0.031	0.018	0.010
	7b	0.152	0.127	0.082	0.054	0.031	0.019	0.009
	7c	0.154	0.126	0.083	0.055	0.030	0.019	0.009
	7d	0.151	0.125	0.082	0.055	0.029	0.019	0.009
	7e	0.150	0.125	0.084	0.054	0.030	0.018	0.010
($D_f = 50$)	7a	0.147	0.124	0.083	0.055	0.031	0.018	0.009
	7b	0.146	0.126	0.082	0.054	0.030	0.018	0.009
	7c	0.147	0.124	0.081	0.055	0.030	0.018	0.009
	7d	0.148	0.125	0.082	0.054	0.028	0.019	0.009
	7e	0.144	0.124	0.083	0.054	0.030	0.018	0.009

which will be further optimized by processing all channels by a single kernel using the image vectorization technique.

In the image vectorization technique, at first, an RGB image is converted to an RGBA image, which is given to a filtration kernel for further processing. As the context switching happens one time between the host and device before image high boosting, it minimizes the overall computational time of the channelized kernel to half on GTX 1050 Ti GPU. Not only that, it has shown a significant rise in the kernels' speed by a factor of 1.33 and 1.6 on the Intel CPU and GPU respectively. Figure 10 exhibits, the Ideal and Gaussian kernels processing times deplete with respect to cutoff frequency D_f, but the time taken by a LoG kernel improves as the cutoff frequency increases. Like OpenCV implementation, both channelized and vectored image kernels execution costs are proportional to the order of a Butterworth filter at a given D_f, but these values are inversely related to the cutoff frequency of this filter at a constant order. Irrespective of the cutoff frequency,

Table 8 Performance metrics of a Butterworth high boost filter with respect to $D_f=30$, 40 and 50

C. Freq.	Images	Entropy	MAE	MSE	RMSE	SD	SNR	PSNR	PFE	SSIM	COR
($D_f=30$, n = 1)	7a	7.8070	14.148	396.406	19.910	57.67	15.93	22.15	3.197	0.7170	0.9384
	7b	7.4650	63.256	4.32e03	65.719	48.77	04.37	11.78	73.36	0.6565	0.9544
	7c	7.7628	51.544	3.23e03	56.814	64.04	06.32	13.04	55.17	0.5998	0.9521
	7d	7.7714	50.630	3.02e03	54.962	61.71	06.67	13.33	50.55	0.5915	0.9771
	7e	7.0401	16.697	474.162	21.775	35.42	09.92	21.37	25.59	0.4393	0.9172
($D_f=40$, n = 1)	7a	7.7728	13.092	352.349	18.771	56.00	16.45	22.66	3.131	0.7270	0.9423
	7b	7.4530	63.350	4.30e03	65.571	47.63	04.39	11.80	73.44	0.6645	0.9584
	7c	7.7810	51.606	3.18e03	56.428	62.55	06.38	13.10	55.27	0.6081	0.9565
	7d	7.7720	50.697	2.98e03	54.580	60.39	6.735	13.39	50.60	0.5986	0.9804
	7e	7.0028	16.700	462.586	21.507	34.69	10.02	21.48	25.60	0.4512	0.9226
($D_f=50$, n = 2)	7a	7.7298	12.422	326.655	18.073	53.92	16.77	22.99	3.116	0.7291	0.9438
	7b	7.4495	63.429	4.29e03	65.491	46.22	04.40	11.81	73.50	0.6701	0.9599
	7c	7.7897	51.664	3.16e03	56.199	60.72	06.41	13.13	55.33	0.6127	0.9578
	7d	7.7624	50.758	2.94e03	54.225	58.65	06.79	13.44	50.63	0.6009	0.9823
	7e	6.9650	16.892	464.803	21.559	33.88	10.00	21.46	25.62	0.4537	0.9236

Table 9 Computation time of a Butterworth high boost filter in sec. with respect to $D_f=30$, 40 and 50

C. Freq.	Images	OpenCV-C Xeon E3- 1225	OpenCL-C Xeon E3-1225 Channelization	OpenCL-C Xeon E3-1225 With Vectorization	OpenCL-C HD Graphics P530 Channelization	OpenCL-C HD Graphics P530 With Vectorization	OpenCL-C GTX 1050 Ti Channelization	OpenCL-C GTX 1050 Ti With Vectorization
($D_f=30$, n = 1) 1)	7a	0.144	0.127	0.084	0.056	0.031	0.020	0.010
	7b	0.146	0.126	0.084	0.056	0.031	0.020	0.009
	7c	0.145	0.128	0.083	0.055	0.031	0.019	0.009
	7d	0.143	0.126	0.083	0.055	0.031	0.020	0.010
	7.e	0.145	0.126	0.084	0.056	0.031	0.019	0.009
($D_f=40$, n = 1)	7a	0.132	0.125	0.083	0.056	0.030	0.019	0.008
	7b	0.131	0.125	0.082	0.055	0.030	0.018	0.009
	7c	0.133	0.126	0.082	0.053	0.031	0.018	0.009
	7d	0.133	0.125	0.082	0.054	0.029	0.019	0.008
	7e	0.130	0.124	0.083	0.055	0.030	0.018	0.009
($D_f=50$, n = 2)	7a	0.140	0.125	0.082	0.055	0.030	0.020	0.008
	7b	0.141	0.126	0.083	0.056	0.031	0.019	0.010
	7c	0.137	0.127	0.082	0.055	0.030	0.019	0.009
	7d	0.136	0.127	0.082	0.055	0.030	0.019	0.009
	7e	0.138	0.125	0.084	0.054	0.031	0.019	0.009

the computational cost of a vectorized Gaussian kernel over a 256 × 256 RGB image is slightly lower than its Matlab implementation over a 128 × 128 gray image [22] and sufficiently lesser than its Matlab implementation over a 512 × 512 RGB image on a 4 core CPU machine [24]. Surround filter using Gaussian weights dehazes an 267 × 188 image in HSV, Lab and RGB color spaces in (0.299 s, 0.689 s, 0.610 s) [25]; CPU and CUDA implementation of 5x5 Gaussian filter for single channel image on a core i5 and 384 NVIDIA GPU consumes 13 s and 7 ms respectively [26]. In contrast to that, CUDA enabled bilateral filter enhances a 256 × 256 gray image in 1.8 ms, but its CPU implementation takes 0.083 s, similar to the vectorized Gaussian kernel execution on a 256 × 256 RGB image. [27].GFA and SFA for a 2D 5x5 Gaussian filter on FPGA Virtex 6 takes (0.371 ms s, 0.394 ms) for a 256x356 gray image, which are quite faster than our vectorized Gaussian kernel, but their unique implementations are limited to FPGA that make

Table 10 Performance metrics of a LoG high boost filter with respect to frequency $D_f=30$, 40 and 50

C. Freq.	Images	Entropy	MAE	MSE	RMSE	SD	SNR	PSNR	PFE	SSIM	COR
($D_f=30$)	7a	7.8427	18.370	634.235	25.184	62.51	13.89	20.11	4.718	0.6668	0.9166
	7b	7.5154	62.906	4.43e03	66.541	52.16	04.26	11.67	72.98	0.6242	0.9312
	7c	7.7039	51.676	3.45e03	58.718	68.30	06.03	12.75	54.61	0.5648	0.9274
	7d	7.7547	50.645	3.19e03	56.486	65.13	06.44	13.09	50.21	0.5537	0.9588
	7e	7.1513	16.676	558.504	23.633	37.84	09.21	20.66	25.69	0.3894	0.8885
($D_f=40$)	7a	7.8357	16.825	543.943	23.327	59.97	14.56	20.77	4.083	0.6773	0.9214
	7b	7.5056	63.111	4.39e03	66.304	50.44	04.29	11.70	73.21	0.6327	0.9374
	7c	7.7583	51.674	3.39e03	58.248	66.34	06.10	12.82	54.94	0.5712	0.9330
	7d	7.7686	50.620	3.11e03	55.812	63.02	06.54	13.20	50.39	0.5593	0.9647
	7e	7.1141	17.313	541.369	23.267	36.69	09.34	20.79	25.64	0.3939	0.8931
($D_f=50$)	7a	7.8190	15.558	471.473	21.713	58.10	15.18	21.39	3.654	0.6893	0.9274
	7b	7.4932	63.243	4.37e03	66.087	49.22	04.32	11.73	73.34	0.6414	0.9434
	7c	7.7869	51.663	3.34e03	57.773	64.85	06.17	12.89	55.13	0.5794	0.9389
	7d	7.7732	50.641	3.06e03	55.321	61.64	06.62	13.27	50.50	0.5680	0.9647
	7e	7.0798	17.064	525.033	22.913	35.40	09.48	20.93	25.63	0.4015	0.8987

Table 11 Computation time of a LoG high boost filter in seconds with respect to $D_f=30$, 40 and 50

C. Freq.	Images	OpenCV-C Xeon E3- 1225	OpenCL-C Xeon E3-1225 Channelization	OpenCL-C Xeon E3-1225 With Vectorization	OpenCL-C HD Graphics P530 Channelization	OpenCL-C HD Graphics P530 With Vectorization	OpenCL-C GTX 1050 Ti Channelization	OpenCL-C GTX 1050 Ti With Vectorization
($D_f=30$)	7a	0.165	0.124	0.082	0.054	0.029	0.018	0.009
	7b	0.167	0.123	0.081	0.054	0.029	0.019	0.009
	7c	0.163	0.124	0.083	0.054	0.030	0.018	0.009
	7d	0.168	0.125	0.082	0.054	0.030	0.018	0.009
	7e	0.165	0.124	0.082	0.055	0.030	0.017	0.008
($D_f=40$)	7a	0.172	0.125	0.083	0.054	0.030	0.018	0.009
	7b	0.175	0.125	0.082	0.055	0.030	0.019	0.010
	7c	0.174	0.125	0.084	0.054	0.030	0.019	0.009
	7d	0.177	0.125	0.083	0.054	0.030	0.019	0.009
	7e	0.173	0.124	0.082	0.055	0.030	0.018	0.008
($D_f=50$)	7a	0.180	0.126	0.084	0.055	0.030	0.019	0.010
	7b	0.183	0.127	0.083	0.055	0.030	0.021	0.010
	7c	0.187	0.125	0.084	0.055	0.030	0.019	0.009
	7d	0.189	0.126	0.083	0.054	0.031	0.020	0.009
	7e	0.184	0.125	0.083	0.056	0.031	0.019	0.009

them unsuitable for other parallel devices [29]. 2D Butterworth filter enhances a 480x320 gray image 0.827 s [30] and various noisy 128 × 128 gray images, whose execution time varies from 0.059 s to 0.130 s [22], unlike the vectorized Butterworth kernel's execution stick around 0.083 s for the 256 × 256 size noisy colored images. On the other hand, CUDA implementation of the Butterworth filter requires 0.4 s for a 12 KB small scaled image [32] and the Retinex algorithm using Butterwoth filter needs 0.310 s to 0.648 s for enrichment of the medium scale images [31]. A parameterized logarithmic enhancement based on LoG filter enhances an 440 × 440 gray image in 14.98 s [36], whereas LoG filter takes 0.3535 s on a normalized image from FVC database 2004 using 4 core intel CPU [37]. Porting the local Laplacian filter into Vulkan API needs 10 s to boost an 800 × 533 colored image on a 1536 core NVIDIA GPU [38], which is not feasible for any time bound applications and made it possible by OpenCL implementation of image vectorized LoG kernel. Here, the total consumption cost of a filter includes the processing time between

Table 12 Overall performance metrics of Ideal, Gaussian, Butterworth and LoG high-boost filters at frequency 30, 40 and 50

Filter	D_f, n	COR	SSIM	PFE (e2)	PSNR (e2)	SNR (e1)	SD (e2)	RMSE (e2)	MSE (c4)	MAE (e2)	Entropy (e1)
Ideal	30	0.9358	0.5854	0.4082	0.1612	0.8434	0.5249	0.444598	0.23286684	0.39637	0.75792
	40	0.9439	0.5944	0.41635	0.163	0.8627	0.5135	0.438484	0.22827276	0.39392	0.756242
	50	0.9494	0.6036	0.4163	0.1642	0.8772	0.5063	0.434026	0.22537454	0.39200	0.75434
Gaussian	30	0.9474	0.5999	0.4318	0.1634	0.8667	0.5268	0.437548	0.22808034	0.39237	0.756784
	40	0.9525	0.6108	0.41628	0.1654	0.8834	0.5151	0.432424	0.22445552	0.39052	0.754968
	50	0.9565	0.6197	0.41636	0.1664	0.8952	0.5072	0.429542	0.22226218	0.38922	0.753334
Butterworth	30, 1	0.9478	0.6008	0.41573	0.1633	0.8642	0.5352	0.43836	0.22881136	0.39255	0.756926
	40, 1	0.9520	0.6098	0.41608	0.1648	0.8795	0.5225	0.433714	0.2254987	0.39089	0.755632
	50, 2	0.9534	0.6133	0.41639	0.1656	0.8874	0.5067	0.431094	0.22362916	0.39033	0.753928
LoG	30	0.9245	0.5597	0.41641	0.1565	0.7966	0.5718	0.461124	0.24525478	0.40054	0.75936
	40	0.9299	0.5668	0.41652	0.1585	0.8166	0.5529	0.453916	0.23950624	0.39908	0.759646
	50	0.9346	0.5759	0.41650	0.1604	0.8354	0.5384	0.447614	0.23533012	0.39633	0.759042

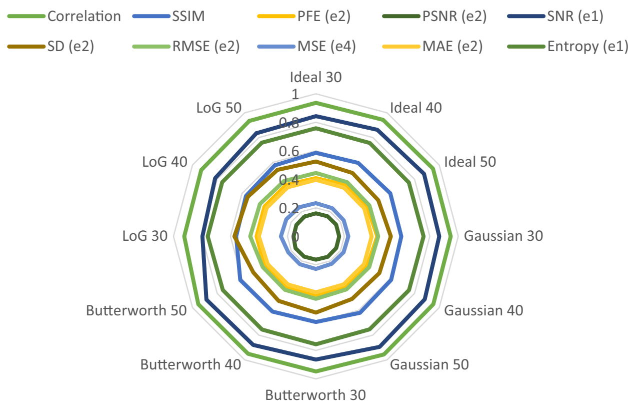
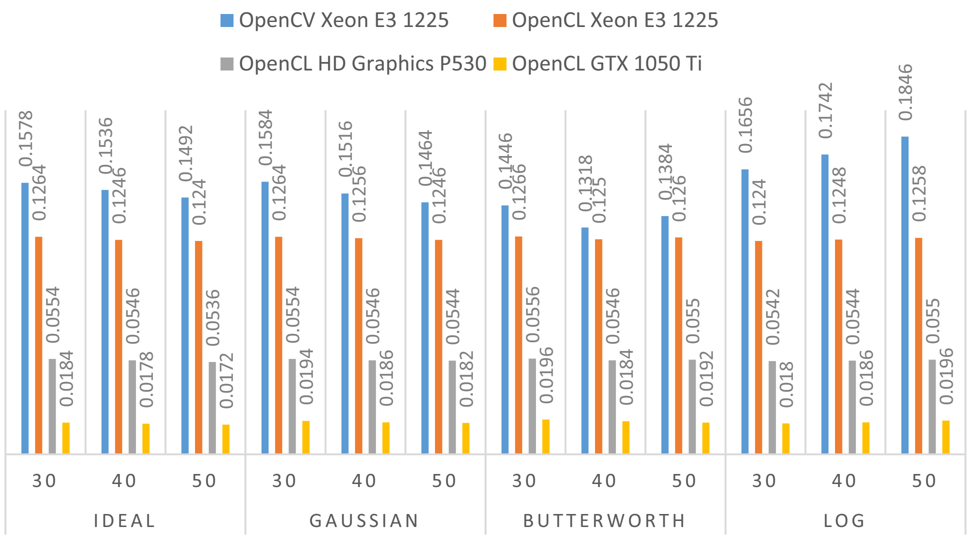


Fig. 8 Graphical representation of overall performance metrics of an Ideal, Gaussian, Butterworth and Laplacian of Gaussian filters at cutoff frequency 30, 40 and 50

the DFT transformation of an input image to generate its frequency map and inverse DFT transformation of its enhanced frequency map to produce the final one in the spatial domain.

The computation time listed in Figs. 9 and 10 include overall cost of the filter matrix creation, DFT and IDFT transformations, and filtration time, which are normally done by the separate OpenCL kernels. The computational costs of the DFT and IDFT transformations of a 256 × 256 image and the creation time filters' matrices for various filters are shown in Figs. 11 and 12. All those filters' matrices creation times consume a lot of time on NVIDIA GTX 1050 Ti than Xeon E3 1225 V5 and HD Graphics P530, whereas these values are minimum on HD Graphics P530. Eventually, Involvement of exponential function makes the Gaussian and LoG filters to run more than two times of

Fig. 9 Overall time consumption in seconds by OpenCV and channelized OpenCL implementations of high-boost filters in regards to cutoff frequencies at 30, 40 and 50



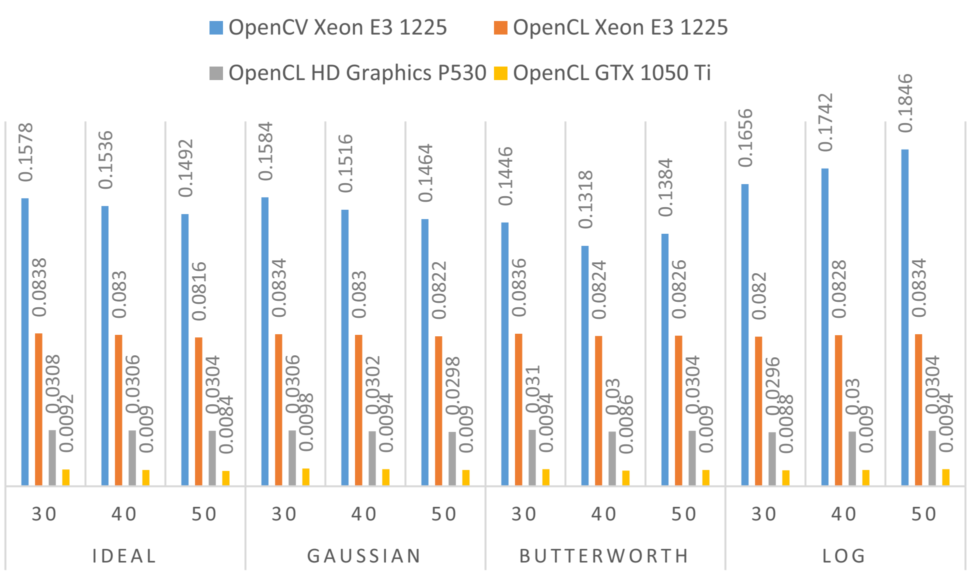


Fig. 10 Overall time consumption in seconds by high-boost filters with respect to cutoff frequencies at 30, 40 and 50, implemented using OpenCV and OpenCL with image vectorization

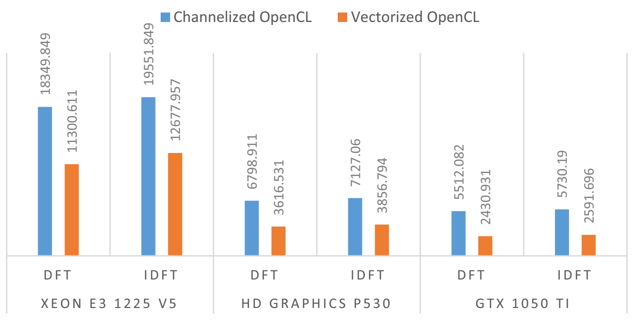


Fig. 11 Computational costs of OpenCL DFT and IDFT kernels in microseconds, implemented on heterogeneous devices using channelization and image vectorization techniques

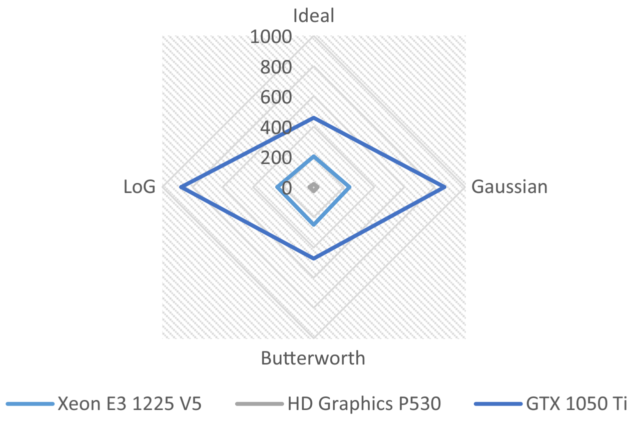


Fig. 12 Average creation time of filters' matrices in microseconds on Intel Xeon E3 1225 V5, Intel HD Graphics P530 and NVIDIA GTX 1050 Ti computing devices

the Ideal and Butterworth filter. As the filter's matrix creation happens only one time before the high-boosting operation; the overall execution time of image filtration will be minimized further in a real-time application. Again in future, by replacing 2D-DFT and 2D-IDFT by an 8×8 or 16×16 FFT and IFFT will bring down the execution time of these OpenCL filtration kernels by a significant amount [59].

As the channelized OpenCL kernels' implementations run independently for Red, Green, and Blue channels, the total execution times of the DFT and IDFT of a 256×256 image are 18.3498 ms and 19.5518 ms respectively on the Intel CPU. However, converting all these three channels to a vector and processing this input vector using OpenCL diminishes the processing cost to 11.3006 ms and 12.6779 ms on the four core CPU which is quiet similar to pyNUFFT on a 8 core CPU [60], but lower than 2D FFT implementation on 32 core GPU [61]. There are also remarkable reduction in the computation time for this kernel's GPU implementations. As the number of cores increases in a device, the values are gone down dramatically as reflected in Fig. 11. As shown in the figure, the vectorized DFT and IDFT kernels consume (3.6 ms, 3.8 ms) and (2.43 ms, 2.59 ms) on 192 cores Intel and 768 cores NVIDIA GPUs, which are comparatively lesser than the 2D FFT implementation using CUFFT library on 5120 cores NVIDIA GPU or on a FPGA XC7K410T at a clock frequency 40 MHz. [62, 63]. Figure 12 exhibits the average creation times of filters' matrices are too high for Gaussian and LoG filters, i.e., the expected filtration times are low compared to Ideal and Butterworth filters. In the end, we have concluded that the utilization of image vectored kernels instead of channelized kernels make the image high boosting operation efficient and reliable for various real-time applications. In addition to that, the creation of the filter's matrix at the beginning and replacement of the 2D-DFT and 2D-IDFT transformations by an 8x8 or 16x16 FFT and IFFT transformations with memory optimization using shared and texture memory of a parallel device will boost the filter kernel's speed [64].

7 Conclusion

Image high-boosting is one of the image enhancement methods, uses sharpening techniques to highlight the high-frequency details that become blur due to various factors like bad weather, poor illumination, improper focusing, and image denoisification. In this article, we have not only explained frequency domain filtering and mostly used high-pass linear and non-linear filters for image high-boosting but also discussed the OpenCL architecture for their parallel implementation. OpenCL kernels of these high-boost filters are designed for their fast implementation, which can be used in various real-world applications such as remote sensing, satellite broadcasting, classroom monitoring, object detection, recognition, and many more video processing applications. In addition to that, the vectorized implementation of the high-boost filters' kernels again minimizes the operation cost that makes them highly adaptable in low-cost real-time applications using various embedded devices. From the result analysis, we have found that a Gaussian filter's correlation, SSIM and PSNR values at three different cutoff frequencies are significantly higher than the other filters, while the Entropy, SD, RMSE, MAE of an output image from the LoG filter are notably larger than the rest of the filters. So, we have come to an end that a Gaussian filter OpenCL kernel tries to protect the original image quality, but a LoG filter kernel focuses on improving frequency components in an image during a high-boosting operation. The channelized implementation of a Gaussian and LoG filters consume 0.0187 s, whereas their vectorized OpenCL implementation require 0.0094 s and 0.00906 s to enhance an 256×256 image on a 768 cores GPU; which are quite lower than the computation time in most of the traditional CPU and GPU implementations of a Gaussian and LoG filters, as we have discussed in Sect. 2. Apart from that, these channelized filters' kernels take approximately 0.125 s to improve an image quality on a 4 core CPU, while they need around 0.0548 s on an Intel 192 cores GPU for the same image high boosting. On the other hand, the vectorized kernels minimize the overall computation time to 0.0828 s and 0.03 s on the 4 cores Intel CPU and Intel GPU respectively.

In spite of these, a Butterworth high-boost filter provides better correlation, SSIM and PSNR values than LoG filter, but slightly lower than the Gaussian filter; on the other hand, the entropy, SD, RMSE and MAE values are little smaller than the LoG filter. So, the Butterworth filter kernel is the suitable one to provide better performance tradeoff between preserving the original image quality and improving the higher frequency components. Regardless of different cutoff frequencies and orders of a Butterworth filter, the channelized and vectorized implementations of this filter requires on an average 0.0190 s and 0.009 s respectively on a 768 cores GPU to improve the quality of an 256×256 image. Although the channelized implementation of this filter enhances a 256×256 RGB image in 0.1254 s and 0.0549 s on a 4 cores Xeon CPU and 192 cores Intel GPU, these values are significantly minimized to 0.0828 s and 0.03 s by the OpenCL vectorized implementation of this filter. In most of the cases, our image vectorized Butterworth filter kernels outperforms the previous implementation of this filter in term of computational time and portability, highlighted under Sect. 2. Thus, a Butterworth high-boost filter is the suitable one to provide better results than other filters with regard to time and accuracy. Here, the accuracy specifies image quality preservation while improving details information present inside the image by highlighting high-frequency regions inside an image. So, the image vectorized Butterworth high-boost filter kernel is the worthy one to provide better results among those filters, which might be highly adaptable in time bound real-time applications using various embedded devices. Ultimately to meet the needs related to the current demands, more research has to be carried out later to improve the degree of precision and minimize the time utilization.

8 Summary

This article discusses about mostly used Ideal, Gaussian, Butterworth, and Laplacian of Gaussian frequency domain high-boost filters and implemented channelized OpenCL kernels for their rapid execution. In addition to that, these kernels are modified using image vectorization technique to optimize their time utilization by reducing the execution time of these OpenCL kernels to half. At last, performance analysis is carried out for these two types of OpenCL kernel implementations to determine their effectiveness with respect of time consumption and accuracy. From the result analysis, we have found that a vectorized Butterworth high-boost filter kernel is the suitable one to provide better results than other filters with regard to time and accuracy, which might be highly adaptable in low-cost real-time applications using various embedded devices.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Jain R, Tyagi V (2014) Spatial and frequency domain filters for restoration of noisy images. IETE J Educ 54(2):108–116
2. Gonzalez RC, Woods RE (2008) Introduction. In: Horton MJ (ed) Digital image processing. Prentice-Hall, Upper Saddle River, pp 1–33
3. Jayaraman S, Esakkirajan S, Veerakumar T (2015) Introduction to image processing system. In: Jha S (ed) Digital image processing. Tata McGraw Hill Education, New Delhi, pp 1–46
4. Bovik AC (2009) Introduction to digital image processing. In: Bovik A (ed) The essential guide to image processing. Elsevier, Burlington, pp 1–21
5. Nazir A et al (2018). Content based image retrieval system by using HSV color histogram, discrete wavelet transform and edge histogram descriptor. In: International conference on computing, mathematics and engineering technologies. Sukkur, Pakistan
6. Ali N et al (2016) A novel image retrieval based on visual words integration of SIFT and SURF. PLoS ONE 11(6):e0157428. https://doi.org/10.1371/journal.pone.0157428
7. Ali N et al (2016) Image retrieval by addition of spatial information based on histograms of triangular regions. Comput Electr Eng 54:539–550
8. Zafar et al (2018) A novel discriminating and relative global spatial image representation with applications in CBIR. Appl Sci 8(11):1–23
9. Ali N (2018) A hybrid geometric spatial image presentation for scene classification. PLoS ONE 13(9):e0203339. https://doi.org/10.1371/journal.pone.0203339
10. Sajid M et al (2018) Data augmentation-assisted makeup-invariant face recognition. Math Probl Eng 2018:1–11. https://doi.org/10.1155/2018/2850632
11. Ratyal N et al (2019) Deeply learned pose invariant image analysis and applications in 3d face recognition. Math Probl Eng 2019:1–22. https://doi.org/10.1155/2019/3547416
12. Ratyal NI et al (2019) Three-dimensional face recognition using variance-based registration and subject-specific descriptors. Int J Adv Robot Syst 16(3):1–16
13. Sajid M, Ratyal NI, Ali N, Zafar B, Hanif Dar S, Mahmood MT, Joo YB (2019) The impact of asymmetric left and asymmetric right face images on accurate age estimation. Math Prob Eng 2019:1–10
14. Williams D, Burns PD (2008) Measuring and managing digital image sharpening. In: Archiving 2008 final program and proceedings. Society for Imaging Science and Technology, pp 89–93
15. Clark JL et al (2018) Effect of image sharpening on radiographic image quality. J Prosthet Dent 120(6):927–933
16. CalTech Face 1999 dataset web site: http://www.vision.caltech.edu/archive.html
17. Jayaraman S, Esakkirajan S, Veerakumar T (2015) Image enhancement. In: Jha S (ed) Digital image processing. Tata McGraw Hill Education, New Delhi, pp 243–323
18. Annadurai S, Shanmugalakshmi R (2007) Image enhancement. In: Fundamental of digital image processing. Pearson Education India, New Delhi, pp 73–130
19. Bovik AC, Acton ST (2009) Basic linear filtering with application to image enhancement. In: Bovik A (ed) The essential guide to image processing. Elsevier, Burlington, pp 225–239
20. Arce GR, Bacca J, Paredes JL (2009) Nonlinear filtering for image analysis and enhancement. In: Bovik A (ed) The essential guide to image processing. Elsevier, Burlington, pp 263–291
21. Chinnasamy G, Vanitha S (2015) Implementation and comparison of various filters for the removal of fractional brownian motion noise in brain MRI images. Int J Trends Eng Technol 3(3):29–33
22. Shukla A, Singh RK (2015) Performance analysis of frequency domain filters for noise reduction. E-J Sci Technol 5(9):167–178
23. Yano T, Kuroki Y (2016) Fast implementation of Gaussian filter by parallel processing of binomial filter. In: International symposium on intelligent signal processing and communication Systems, Phuket, Thailand. https://doi.org/10.1109/ispacs.2016.7824738
24. Ghosh S, Chaudhury KN (2016) Fast bilateral filtering of vector-valued images. In: IEEE international conference on image processing (ICIP), Phoenix, AZ, pp 1823–1827
25. Nair D, Sankaran P (2017) Color image dehazing using surround filter and dark channel prior. J Vis Commun Image Represent 105:98–105
26. Preeti K, Vishvaksenan KS (2018) Gaussian filtering implementation and performance analysis on GPU. In: International conference on inventive research in computing applications. Coimbatore, pp 936–939
27. Oza S, Joshi KR (2018) CUDA based fast bilateral filter for medical imaging. In: Fifth international conference on signal processing and integrated networks, Noida, pp 930–935
28. Rakhshanfar M, Amer MA (2019) Efficient cascading of multi-domain image noise filters. J Real Time Image Proc. https://doi.org/10.1007/s11554-019-00868-9
29. Mukherjee D, Mukhopadhyay S (2019) Fast hardware architecture for fixed point 2D Gaussian filter. AEU Int J Electron Commun 105:98–105. https://doi.org/10.1016/j.aeue.2019.03.020
30. Dyre S, Sumathi CP (2014). Hybrid approach to enhancing fingerprint images using filters in the frequency domain. In: International conference on computational intelligence and computing research, Coimbatore, India. https://doi.org/10.1109/iccic.2014.7238306
31. Fan T et al (2017) An improved single image defogging method based on Retinex. In: 2nd International conference on image, vision and computing, Chengdu, China, pp 410–413
32. Zhao et al (2016) An improved brightness balancing method and its GPU acceleration for digital images. J Appl Sci Eng 19(4):505–514
33. Chen JS, Huertas A, Medioni G (1987) Fast convolution with Laplacian-of-Gaussian masks. IEEE Trans Pattern Anal Mach Intell 9(4):584–590
34. Wu W (2016) Paralleled Laplacian of Gaussian (LoG) edge detection algorithm by using GPU. In: Eighth international conference on digital image processing, Chengdu, China, pp 1–5
35. Almazrooie M et al (2014) Parallel Laplacian filter using CUDA on GP-GPU. In: International conference of information technology and multimedia, Putrajaya, Malaysia, pp 60–65
36. Bao C, Sheng C (2013) A parametrized logarithmic image processing method based on Laplacian of Gaussian filtering for lung nodules enhancement in chest radiographs. In: Second international symposium on instrumentation and measurement, sensor network and automation, Toronto, ON, pp 649–652
37. Arif A, Li T, Cheng C (2017) Blurred fingerprint image enhancement: algorithm analysis and performance evaluation. SIViP 12(4):767–774
38. Rafaela GM (2019) Porting the Laplacian filtering application to the Vulkan API using OpenCL and OpenGL programming

models (Diploma Thesis, University of Thessaly, Volos, Greece). http://ir.lib.uth.gr/bitstream/handle/11615/49481/18311.pdf?sequence=1
39. Banger R, Bhattacharyya K (2013) Hello OpenCL. In: D'souza W, Pandey K, Colaco K (eds) OpenCL programming by example. Packt Publishing, Birmingham, pp 1–34
40. Banger R, Bhattacharyya K (2013) OpenCL architecture. In: D'souza W, Pandey K, Colaco K (eds) OpenCL programming by example. Packt Publishing, Birmingham, pp 35–58
41. Mann P (2016) Review: EVGA GeForce GTX 1050 Ti SC gaming. http://hexus.net/tech/reviews/graphics/98329-evga-geforce-gtx-1050-ti-sc-gaming/. Retrieved 17 June 2017
42. Intel (2016) Intel Xeon processor-based platforms for Internet of Things (IoT) solutions [pdf]. https://www.intel.com/content/dam/www/public/us/en/documents/platf-orm-briefs/xeon-processor-e3-1200-v5-workstation-platform-brief.pdf. Retrieved 26 Feb 2019
43. Kirsch N (2015). Intel HD Graphics 530 has 24 execution units—Intel gen9 graphics architecture detailed. https://www.legitreviews.com/intel-hd-graphics-530-has-24-execution-units-intel-gen9-graphics-architecture_170869. Retrieved 27 Feb 2019
44. Banger R, Bhattacharyya K (2013) OpenCL program and kernel objects. In: D'souza W, Pandey K, Colaco K (eds) OpenCL programming by example. Packt Publishing, Birmingham, pp 109–136
45. Gonzalez RC, Woods RE (2008) Image enhancement in the frequency domain. In: Horton MJ (ed) Digital image processing. Prentice-Hall, Upper Saddle River, pp 75–146
46. Das A (2015) Interpretation and processing of image in frequency domain. In: Wheeler W (ed) Guide to signals and patterns in image processing. Springer, Cham, pp 93–147
47. Burger W, Burge MJ (2008) The Discrete Fourier transform in 2D. In: Gries D, Schneider FB (eds) Digital image processing: an algorithmic introduction using java. Springer, London, pp 343–366
48. Marques O (2011) Frequency domain filtering. In: Practical image and video processing using MATLAB. Wiley, Hoboken, pp 235–264
49. Shaikh MS, Choudhry A, Wadhwani R (2016) Analysis of digital image filters in frequency domain. Int J Comput Appl 140(6):12–19
50. Grami A (2016) Signals, systems and spectral analysis. Introduction to digital communications. Elsevier, Waltham, pp 41–150
51. Dorga A, Bhalla P (2014) Image sharpening by Gaussian and Butterworth high pass filter. Biomed Pharmacol J 7(2):707–713
52. Zawaideh FH, Yousef QM, Zawaideh FH (2017) IJCSNS Int J Netw Secur 17(7):113–117
53. Kong H, Akakin HC, Sarma SE (2013) A generalized Laplacian of Gaussian filter for blob detection and its applications. IEEE Trans Cybern 43(6):1719–1733
54. Banger R, Bhattacharyya K (2013) OpenCL C programming. In: D'souza W, Pandey K, Colaco K (eds) OpenCL programming by example. Packt Publishing, Birmingham, pp 155–178
55. Banger R, Bhattacharyya K (2013) OpenCL buffer objects. In: D'souza W, Pandey K, Colaco K (eds) OpenCL programming by example. Packt Publishing, Birmingham, pp 59–86
56. Banger R, Bhattacharyya K (2013) OpenCL Images. In: D'souza W, Pandey K, Colaco K (eds) OpenCL programming by example. Packt Publishing, Birmingham, pp 59–86
57. Naidu VPS, Raol JR (2008) Pixel-level image fusion using wavelets and principal component analysis. Def Sci J 58(3):338–352
58. Seshadrinathan K et al (2009) Image quality assessment. In: Bovik A (ed) The essential guide to image processing. Elsevier, Burlington, pp 553–595
59. Al-Ani MS (2017) Fast two dimensional digital filter design based on fast Fourier transform. J Theoret Applied Inf Technol 95(23):6678–6689
60. Lin L (2018) Python non-uniform fast Fourier transform (pyNUFFT): an accelerated non-Cartesian MRI package on a heterogeneous platform (CPU/GPU). J Imaging 4(3):1–22
61. Shen F et al (2015) Research on the fast Fourier transform of image based on GPU. arXiv:1505.08019 [cs.MS]. Retrieved 21 Jan 2019
62. Cheng X et al (2018) Accelerating 2D FFT: exploit GPU tensor cores through mixed precision. In: The international conference of high performance computing, networking, storage, and analysis. Dallas, TX
63. Li M, Wyrwicz AM (2018) Parallel 2D FFT implementation on FPGA suitable for real-time MR image processing. Rev Sci Instrum 89(9):1–9
64. Zhang, F. et al. (2017). A GPU based memory optimized parallel method for FFT implementation. arXiv:1707.07263 [cs.DC]. Retrieved 15 Dec 2018