International Journal of Electrical and Computer Engineering (IJECE)
Vol. 8, No. 6, December 2018, pp. 5457~5471
ISSN: 2088-8708, DOI: 10.11591/ijece.v8i6.pp5457-5471

Real-Time Implementation and Performance Optimization of Local Derivative Pattern Algorithm on GPUs

Nisha Chandran S, Durgaprasad Gangodkar, Ankush Mittal
Department of Computer Science and Engineering, Graphic Era University, India

Article Info

Article history:

Received Jan 21, 2018
Revised Mei 28, 2018
Accepted Jun 14, 2018

Keyword:

CBIR
CUDA
GPGPU
Local Derivative Pattern
Paralleization
Texture Descriptor

ABSTRACT

Pattern based texture descriptors are widely used in Content Based Image Retrieval (CBIR) for efficient retrieval of matching images. Local Derivative Pattern (LDP), a higher order local pattern operator, originally proposed for face recognition, encodes the distinctive spatial relationships contained in a local region of an image as the feature vector. LDP efficiently extracts finer details and provides efficient retrieval however, it was proposed for images of limited resolution. Over the period of time the development in the digital image sensors had paid way for capturing images at a very high resolution. LDP algorithm though very efficient in content-based image retrieval did not scale well when capturing features from such high-resolution images as it becomes computationally very expensive. This paper proposes how to efficiently extract parallelism from the LDP algorithm and strategies for optimally implementing it by exploiting some inherent General-Purpose Graphics Processing Unit (GPGPU) characteristics. By optimally configuring the GPGPU kernels, image retrieval was performed at a much faster rate. The LDP algorithm was ported on to Compute Unified Device Architecture (CUDA) supported GPGPU and a maximum speed up of around 240x was achieved as compared to its sequential counterpart.

Corresponding Author:

Nisha Chandran S.,
Departement of Computer Science and Engineering,
Graphic Era University,
Dehradun 248002, Uttarakhand, India.
Email: nisha.unnikrishnan@gmail.com

1. INTRODUCTION

In Content Based Image Retrieval (CBIR), images are indexed automatically by their own visual content instead of the text-based key words. The basic step involved in CBIR is extracting and indexing the visual content (low level features) from the images. Feature vector extraction is performed as a pre-processing step in most of the CBIR systems. An efficient image feature descriptor must have excellent discrimination property to discriminate various image classes and it must be computationally inexpensive. Different feature descriptors like color, shape and texture are used for efficient image retrieval in CBIR systems. Among the various visual feature descriptors, texture descriptor plays a significant role in feature extraction and it refers to the visual patterns that have homogeneity property. Texture descriptor represents structural arrangement of surfaces in relation to the surrounding environment. Pattern based texture features like Local Binary Pattern (LBP) [1] represents images based on the first order derivative micro patterns which apparently fail to extract detailed information contained in the images. Researchers therefore, further investigated the possibility of using higher order operators for capturing more detailed information.

LDP operator proposed by Zhang et al. [2] is a texture descriptor which encodes higher order derivative information. LDP represents the higher order pattern information of a local region by encoding the change in the neighbourhood derivative directions and therefore contains more detailed discriminative features than the first order derivatives used in LBP. However, LDP being a higher order operator produces a

very large feature vector which makes the algorithm computationally very expensive. Currently, with the advancements in the digital signal technology images are being captured at a very high resolution and extracting LDP features using the current general-purpose hardware from large databases of such high-resolution images becomes very challenging. This limits the applicable domains of this powerful algorithm with high discriminative property. LDP like most image processing algorithms perform the same computation on a number of pixels, which is a typical data parallel operation. Therefore, computing the LDP features from different regions of the high-resolution image in parallel will reduce the computation time of the algorithm thereby making it efficient for real time applications.

Currently, Graphics Processing Unit (GPU) has entered the General-Purpose Computing Domain (GPGPU). In comparison to the single processor CPU, GPGPUs have very high computation power. They are inexpensive and scalable than CPUs. The highly data parallel image processing algorithms exploits the Single Instruction Multiple Data (SIMD) architecture and can be efficiently parallelized on a GPGPU [3]. However, traditional GPGPU development is based on graphics function library like OpenGL and Direct 3D, which restricts its use by professionals, who are familiar with graphics API. The emergence of Compute Unified Device Architecture (CUDA), NVIDIA's parallel architecture implementation, removes these inconveniences as it provides APIs for programmers to develop parallel applications using the C programming language. GPUs are inexpensive, scalable and have very high computation power compared to single processor CPUs. GPUs have been widely used now for accelerating many efficient CBIR algorithms [4], [5] in the field of image vision, medical imaging [6], remote sensing [7] etc. This paper proposes a parallel implementation of the LDP algorithm on NVIDIA GPUs. The implementation exploits the features like the global memory, shared memory, L1 cache, CUDA streams and optimal use of registers. We have measured the GPU implementations on Fermi as well as the latest Kepler architecture. To the best of our knowledge this is the first reported effort to parallelize the LDP algorithm using GPUs. Main contributions of the paper are summarized as follows: (1) We present an efficient strategy for parallel calculation of the LDP feature vector of an image by optimally using the GPU global memory and the faster shared memory (2) We have further tuned the speedup of LDP computations by using the L1 data cache. (3) We have accelerated the GPU operations using the concept of CUDA streams. (4) We have exploited the asynchronous nature of the CUDA kernel calls by deploying calculation on host while the device executes the GPU kernels thereby extracting enhanced parallelism.

The rest of the paper is organized as follows. We begin with a discussion on the related work in section 2. Section 3 reviews the mathematical foundation of the LDP algorithm and describes the main concepts of programming GPUs using NVIDIA's CUDA model. In section 4, we present our approach for computing the LDP using the GPU. Finally, section 5 presents the experiments done and a detailed analysis of the results that demonstrates the efficiency of our GPU algorithm and section 6 draws some conclusions about the presented work and suggests some optimizations as future work.

2. RELATED WORK

The most computationally expensive part of the CBIR algorithms is the feature extraction. Color, texture or shape features or a combination of these are generally extracted from the images for efficient image matching and retrieval. GPGPUs are now being efficiently used for implementing this computationally expensive feature extraction task as well as the matching task in CBIR algorithms. It is a well-known fact that the GPUs perform much faster than CPUs and attain excellent speedup for image processing applications which are specifically tailored for parallel programming. Several notable works showing the advantage of using a GPU and the substantial amount of speed up obtained over the sequential version, for such applications are available in the literature. Basic image processing algorithms like histogram equalization, Discrete Cosine Transform (DCT) encode and decode, cloud removal etc. are efficiently parallelized using CUDA and is reported to have attained excellent speedup [8]. In Ref. [9] a parallel version of the fractal image encoding algorithm is built on the GPU and the results indicate the encoding was achieved in milliseconds time without compromising on the quality of the decoded image. Tek et al. [10] efficiently parallelized a face recognition framework using CUDA and reported a speedup of 29x. Gangodkar et al. in [11] implemented a parallel version of fast normalized cross correlation for real time template matching which attained a speedup of 17x compared to the sequential execution. In Ref. [12] a parallel algorithm for accelerating the image inpainting process is proposed. The intense computational task of processing the fourth order PDE equations is accelerated by 36x using the NVIDIA GEFORCE GTX 670 GPU card. A parallel implementation of content-based video search and retrieval of videos is proposed in Ref. [13]. The

authors achieved around 80% accuracy in video search and retrieval and a speedup of 5x compared to the sequential version of the algorithm. In Ref. [14] a parallel implementation of the audio compression algorithm Apple Lossless Audio Codec (ALAC) is presented and a speedup of 80-90% is achieved. Heidari et al. [15] proposed a parallel implementation of color moments and texture feature and achieved a speed up of 144x over the sequential implementation. Zhu et al. [16] have parallelized the SURF algorithm and have achieved a speedup of 15x over its sequential counterpart. In [17] Priscariu and Reid have implemented Histogram of Oriented Gradients (HOG) in GPU and have achieved a speedup of 67x. In [18] Park et al. explored the design and implementation issues of image processing algorithms on GPUs with CUDA framework. They have selected multi view stereo matching, linear feature extraction, JPEG2000 image encoding non-photorealistic rendering as example applications and efficiently parallelized them on the GPU. In [19] the authors proposed an implementation of Ising simulation work using the CUDA framework. They have attained a performance improvement of 20-40 times compared to the conventional CPU implementation. In [20] the authors have implemented a parallel algorithm in GPU for value prediction and speculative execution. They have observed that software value prediction techniques did not attain much performance gain whereas the software speculation techniques attained considerable performance gain. However, optimizing a parallel GPU code is not a trivial task. To achieve good performance the programmer should fine tune the code taking into account the underlying architecture.

Several tuning strategies using different aspects of the GPU architectures have been discussed in the standard CUDA books [21]-[23]. In [24] authors give insights into the relationship between occupancy, thread block size and shape, cache hierarchy configuration and thread access pattern to global memory. By taking simple matrix multiplication as an example they relate the thread block size and shape to both cache misses and occupancy. Lobeiras et al. in [25], by taking Fast Fourier Transform (FFT) algorithm as an example, have analyzed different aspects of the GPU architectures, like the influence of memory access patterns and register pressure on performance. They have shown that the performance of register-based solutions is much better than the shared memory due to the higher register bandwidth. In [26] Liang et al. have fine tuned their 3D localization application in GPU by optimizing the number of threads per thread block and registers per thread. By experimenting with different values of the thread block size and the number of registers they have found an optimal setting which gives the best performance. Torres et al. in [27] have studied the relation between different thread block sizes and shapes with the global memory and the cache memory access patterns. In [28] Lee et al. have discussed optimization strategies for compute bound as well as memory bound algorithms. They have optimized the compute bound algorithms by reducing the number of registers and by increasing the thread block size whereas memory bound algorithms are optimized by storing the data in limited GPU resources like constant or shared memory in an optimized way.

3. LDP DESCRIPTOR AND USAGE FOR IMAGE RETRIEVAL

LDP algorithm contains very detailed discriminative features of an image as it encodes the higher order derivative information. For an image $I(Z)$, the first order derivatives along 0°, 45°, 90° and 135° directions are denoted as $I'_{\alpha}(Z)$ where α= 0°, 45°, 90° and 135°. Let us consider a pixel Z_0 in $I(Z)$ and its eight neighbors Z_i , where i=1… 8 as shown in Figure 1a. The four first order derivatives [2] for the pixel Z_0 along the four different directions are given as in (1) to (4).

$$I'_{0^{\circ}}(Z_0) = I(Z_0) - I(Z_4)\,; \tag{1}$$

$$I'_{45^{\circ}}(Z_0) = I(Z_0) - I(Z_3); \tag{2}$$

$$I'_{90^{\circ}}(Z_0) = I(Z_0) - I(Z_2); \tag{3}$$

$$I'_{135^{\circ}}(Z_0) = I(Z_0) - I(Z_1) \tag{4}$$

The directional differential channels for the four directions are as shown in Figure 1b.
The second order directional [2] LDP, for the pixel Z_0 $LDP^2_{\alpha}(Z_0)$, in α direction is given as in (5).

$$LDP^2_{\alpha}(Z_0) = \begin{Bmatrix} f(I'_{\alpha}(Z_0), (I'_{\alpha}(Z_1)), f(I'_{\alpha}(Z_0), (I'_{\alpha}(Z_2)) \\ \dots, f(I'_{\alpha}(Z_0), (I'_{\alpha}(Z_8)) \end{Bmatrix} \tag{5}$$

where $f(.,.)$ is a binary encoding function which determines the transitions in the local pattern and is given as in (6).

$$f\left(I'_\alpha(Z_0), I'_\alpha(Z_i)\right) = \begin{cases} 0, if\ I'_\alpha(Z_i).I'_\alpha(Z_0) > 0 \\ 1\ , if\ I'_\alpha(Z_i).I'_\alpha(Z_0) \leq 0 \end{cases} \quad (6)$$

i=1, 2 ...8.
where '.' represents multiplication operator. The third order local derivative pattern is computed using the second order derivatives. The third order directional LDP [2], for the pixel Z_0 $LDP^3_\alpha(Z_0)$, in α direction is given as in (7).

$$LDP^3_\alpha(Z_0) = \begin{Bmatrix} f(I''_\alpha(Z_0), (I''_\alpha(Z_1)), f(I''_\alpha(Z_0), (I''_\alpha(Z_2)) \\ \dots, f(I''_\alpha(Z_0), (''(Z_8)) \end{Bmatrix} \quad (7)$$

In general, the n[th] order LDP is computed using the (n-1) [th] order derivatives. The n[th] order directional LDP [2], for the pixel $Z_0, LDP^n_\alpha(Z_0)$ in α direction is given as in (8).

$$LDP^n_\alpha(Z_0) = \begin{Bmatrix} f(I^{n-1}_\alpha(Z_0), (I^{n-1}_\alpha(Z_1)), f(I^{n-1}_\alpha(Z_0), (I^{n-1}_\alpha(Z_2)) \\ \dots, f(I^{n-1}_\alpha(Z_0), (I^{n-1}_\alpha(Z_8)) \end{Bmatrix} \quad (8)$$

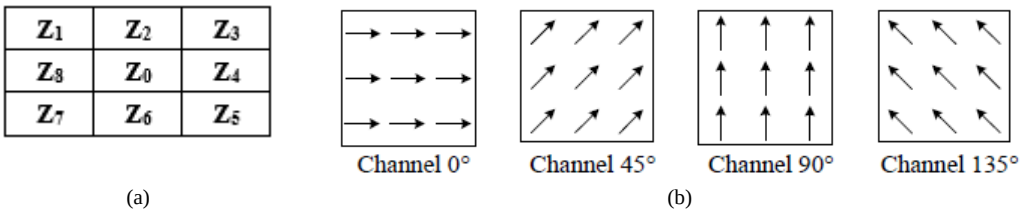
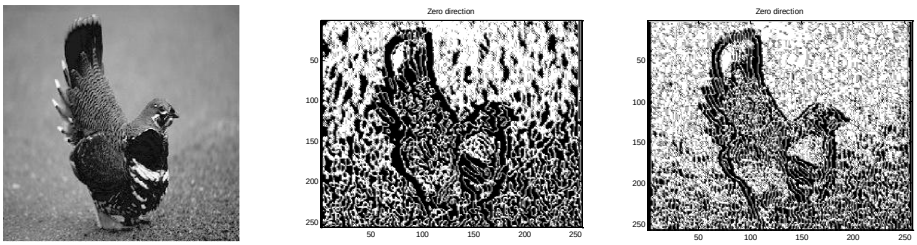


Figure 1. (a) Schematic showing the location of the pixel Z_0 and the neighboring pixels Z_1...Z_8 used in the LDP algorithm [2], (b) Schematic showing four directional differential channels

Higher order LDPs extract higher spatial frequencies that contain finer details of an image. This is very useful for image retrieval and pattern matching operations. However, if the image contains noise the higher order LDPs also enhances the noise, especially from the fourth order and above. LDP thus represents the higher order pattern information by labeling the pixels of an image by comparing two derivative directions at neighboring pixels and concatenating the results into a 32-bit binary sequence. Figures 2b and 2c show the visualization of the second order and third order LDP in zero direction respectively for an example image shown in Figure 2a. As shown in figures the third order LDP gives more detailed information than the second order. We have used the third order LDP in our experiments as it extracts higher spatial frequencies thereby extracting finer details of an image. As our preliminary work we have implemented the LDP algorithm for medium sized images using only the GPU global memory [29]. The pseudo code of the LDP algorithm is given in Table 1 and its implementation using the global memory space can be found in [29].

3.1. GPU and NVIDIA CUDA

An overview of the basic GPGPU architecture is presented in this section. GPGPU generally consists of hundreds of Streaming-Processors (SPs) called cores which are further grouped into Streaming Multi Processors (SMPs) [21]. The cores perform data processing in the Single Instruction Multiple Thread (SIMT) model. Mapping of the algorithm to the GPU is done by first identifying the highly parallel phases of the algorithm. Those parallel phases are then offloaded to the GPU and are then called the CUDA kernels. The rest of the phases are mapped onto the CPU. On completion of execution of a CUDA kernel the CPU collects the execution results and the GPU is then scheduled with a new kernel for execution. CUDA exploits data level parallelism by making every thread work on a different set of data. The light weight CUDA threads combine to form thread blocks and the thread blocks combine to form a computational grid. The grids and the threads can be arranged in one, two or three dimensions. Each thread and thread block can be identified by unique ids and the kernel is launched by a grid of thread blocks.



(a) (b) (c)

Figure 2. (a) Sample query image from Corel database, (b) Second order LDP of the query image in zero direction. Decimal equivalent of the binary pattern is shown, (c) Third Order LDP of the query image in zero direction. Decimal equivalent of the binary pattern is shown

Fermi [30] the world's first complete GPU architecture was a significant leap forward from the G80 chipset series. The SMPs in the Fermi architecture consists of 32 cores and each of these can execute one floating-point or integer instruction per clock. Kepler architecture launched in 2010 is the latest NVIDIA generation of GPU architectures and is built based on the foundation of Fermi. Each SMP in Kepler architecture has 192 single precision CUDA cores, each one with floating-point and integer arithmetic logic units. Tesla K40c launched in 2012 is NVIDIA's flagship compute GPU, based on the Kepler GK110 architecture [31]. With 5GB or 6GB of GDDR5 memory, they provide up to 3.95 TFLOPS single precision and 1.33TFLOPS double precision floating point performance. To hide the long off chip memory access latencies NVIDIA Fermi and Kepler GPUs include software managed caches, the shared memory and the hardware managed caches, the L1 data caches. Shared memory is on chip and the CUDA C compiler treats variables in the shared memory different from the typical variables. It creates a copy of the variable for each block that is launched on the GPU. Every thread in the block shares the memory and thus can communicate and collaborate on the computations done. Furthermore, shared memory and L1- D cache on a GPU utilize the same physical storage and their capacity can be configured at runtime [32]. Each SM in the Fermi and Kepler architecture has 64KB of on-chip memory, divided into shared memory and L1 cache. The programmers may choose between the two configurations: 48KB of shared memory and 16KB of L1 cache or 16KB of shared memory and 48KB of L1 cache. However, an increase in cache memory reduces the amount of shared memory to 16KB, which can lead to the reduction in occupancy of the SMs.

In addition to shared memory and L1 cache, CUDA streams [33], [34] also plays an important role in accelerating the computations done on the GPU. CUDA stream represents a queue of GPU operations that get executed in a specific order. Both kernels launch, and memory copies can be added into a stream. The order in which the operations are added to the stream specifies the order in which they will be executed. When using streams, instead of copying the input buffers entirely to the GPU, they are split into smaller chunks and the memory copies to and from the GPU as well as the kernel launches are performed on each chunk.

Table 1. The pseudo code for the LDP calculation [29]

Algorithm: Pseudo code for the LDP calculation
Input: Query Image; Output: retrieval result
Step 1: Divide the input image of $N \times N$ pixels into blocks of $k \times k$ sub- images. Each sub image has $\frac{N}{k} \times \frac{N}{k}$ pixels.
Step 2: For i= 1 to k^2 do steps 3 to 5
Step 3: For the image pixel $I(Z_0)$, the location of which is as shown in Fig.1, calculate the first order derivatives along 0^0, 45^0, 90^0 and 135^0 directions as given in eqs.1 to 4.
Step 4: Calculate binary encoding function as given in eq.6.
Step 5: Calculate the histograms of the binary patterns.
Step 6: Concatenate the histograms of all the sub images to construct the feature vector of the image.

4. OPTIMUM SIZING OF EES

The approach adapted to parallelize the LDP algorithm is to break the data level parallelism involved in the algorithm. The algorithm is divided into two parts: CPU (Host) and GPU (Device) processing as shown in Figure 3. The image is read by the host, copied to the device memory, spatial histograms of the image is computed and is copied back to the host memory. The images are divided into 16x16 sub blocks and the blocks are processed by the GPU. To show the computation of LDP a sample 7x7 block size is taken as shown in Figure 4. A block of 9 threads is required to calculate the third order LDP of the black square region in Figure 4. The LDP patterns for the block are calculated using the binary encoding functions given

in Equation 7. The binary encoding functions for the pixel Z_{16} , the position of which is shown in Figure 4, for the four different directions 0°,45° , 90° and $135^{\circ}$is given as in (9).

$$LDP_{\alpha}^{3}(Z_{16}) = \begin{Bmatrix} f\left(I_{\alpha}''(Z_{16}), I_{\alpha}''(Z_{8})\right), f\left(I_{\alpha}''(Z_{16}), I_{\alpha}''(Z_{9})\right) f\left(I_{\alpha}''(Z_{16}), I_{\alpha}''(Z_{10})\right) \\ f\left(''(Z_{16}), I_{\alpha}''(Z_{17})\right) f\left(I_{\alpha}''(Z_{16}), I_{\alpha}''(Z_{24})\right) f\left(I_{\alpha}''(Z_{16}), I_{\alpha}''(Z_{23})\right) \\ f\left(I_{\alpha}''(Z_{16}), I_{\alpha}''(Z_{22})\right) f\left(I_{\alpha}''(Z_{16}), I_{\alpha}''(Z_{15})\right) \\ \text{where } \alpha = 0^{\circ}, 45^{\circ}, 90^{\circ} \text{ and } 135^{\circ} \end{Bmatrix} \quad (9)$$

4.1. GPU and NVIDIA CUDA

This section describes how our implementation parallelizes the LDP algorithm. The algorithm is divided into two main steps the LDP histogram calculation and the histogram intersection. Out of the two the most critical step in total execution time is the LDP histogram calculation and therefore all optimizations are done to speedup this kernel. The GPU algorithm thus consists of two kernels to execute the operations of the two basic steps of the LDP algorithm.

The *LDP_compute* kernel computes the LDP values of all the pixels in the 16x16 block of the image along all the four directions 0°, 45° , 90° and 135°. The sum of the decimal equivalent of the patterns is then used to form the LDP histogram. Threads are allocated for each pixel of the image block and each thread calculates and stores the LDP histogram values. The histogram values are then transferred to the host for normalization.

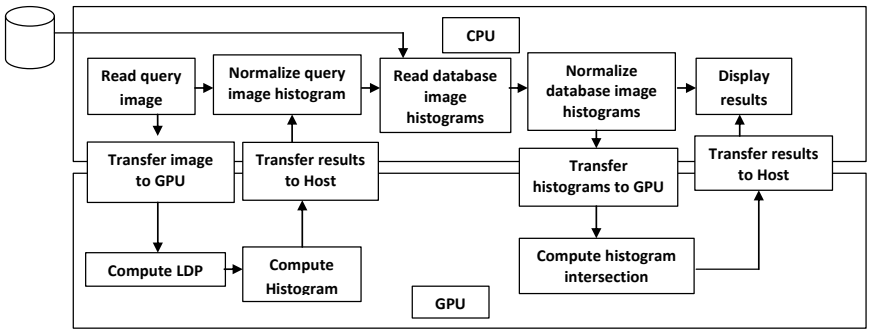


Figure 3. Steps in the computation of LDP

Z_0		Z_1	Z_2	Z_3	Z_4	Z_5	Z_6
Z_7		Z_8	Z_9	Z_{10}	Z_{11}	Z_{12}	Z_{13}
Z_{14}		Z_{15}	Z_{16}	Z_{17}	Z_{18}	Z_{19}	Z_{20}
Z_{21}		Z_{22}	Z_{23}	Z_{24}	Z_{25}	Z_{26}	Z_{27}
Z_{28}		Z_{29}	Z_{30}	Z_{31}	Z_{32}	Z_{33}	Z_{34}
Z_{35}		Z_{36}	Z_{37}	Z_{38}	Z_{39}	Z_{40}	Z_{41}
Z_{42}		Z_{43}	Z_{44}	Z_{45}	Z_{46}	Z_{47}	Z_{48}

Figure 4. Image block of size 7x7 to demonstrate the computation of a thread block.
LDP of the region in black rectangle is to be computed

The *Hist_intersect* kernel computes the histogram intersection of the normalized query image LDP histogram and the database image histogram. Histogram intersection [2] is used to measure the similarity between two histograms and is given by the (10).

$$S_{HI}(H,S) = \sum_{i=1}^{B} \min(H_i S_i) \quad (10)$$

where $S_{HI}(H,S)$ is the histogram intersection statistic. This measure calculates the common parts of two histograms. Histogram intersection thus finds the common parts of the query image histogram and the database image histograms. Depending upon the size of the histograms threads are allocated for comparing the elements in the query and the database image histograms.

4.2. Design Space Exploration and Optimization

In the following sections we discuss in detail the architecture related design space exploration and optimization focusing mainly on the CUDA thread block sizes, registers, memory spaces and CUDA streams.

4.2.1 Thread Block Size

The general methodology adapted when optimizing a GPU accelerated application is to increase the occupancy of the SMPs. It is essential to run as many threads as possible on the SMPs to hide the long memory latency. To increase occupancy either the number of registers per thread has to be decreased or the number of threads in the thread blocks has to be adjusted according to the number of registers per thread. The occupancy of a GPU of compute capability 2.1 for 42 registers per thread is more than 68% using 128,192 and 256 threads whereas with 64 threads it is 33%. We explore the optimum thread block size for different image sizes that gives the maximum occupancy as well as the least execution time.

4.2.2 Registers

Reducing the number of registers used by the threads is another option to increase occupancy. The CUDA nvcc compiler performs large scale register allocation to remove instruction dependencies and to maximize single thread performance [27], [35]. Registers being a limited resource for a block an incremental increase in register allocation will lead to running fewer thread blocks. Thus, there is a tradeoff between the single thread performance and the number of active threads at a time. In [35] the authors have used a metric RegRatio which is the ratio of the registers used per thread to the maximum allowed registers per thread. Using this metric, we have optimized the usage of the registers. We keep the RegRatio value between 0 and 1 and choose a high value of RegRatio so that there is a good tradeoff between the single thread performance and the number of simultaneously active threads. For a particular kernel the maximum number of registers per thread can be specified to the nvcc compiler at compile-time using the *maxrregcount* option. Theoretically, the actual register use might be less than the limit. We have also verified the performance gain achieved by RegRatio using a heuristic approach where for each image size, thread size and block size the number of registers is varied and the combination which gives the maximum occupancy and minimum kernel execution time is found.

4.2.3 Shared Memory and L1 Cache

To further improve the performance of the LDP algorithm, in the next step we make use of the faster on chip shared memory. The number of registers and the thread block size is kept at an optimum value obtained from the earlier study. Further we choose between the two configurations: 48KB of shared memory and 16KB of L1 cache or vice versa to study the performance improvement obtained by using L1 cache instead of shared memory.

4.2.4 CUDA Streams

Fermi and Kepler architecture-based GPUs support device overlap which is the capacity to simultaneously execute a CUDA kernel while performing a memory copy operation between device and host memory. This is facilitated using CUDA streams. The use of streams is very profitable in applications where input data instances are independent. A stream is a sequence of commands that is executed in order. By default, a CUDA program has only one stream called the default stream [22], [23]. When multiple streams are involved the order in which the operations are done becomes an important issue. There are two basic approaches used to order the operations, breadth first and depth first. In depth first approach all operations of a given stream are grouped together and in breadth first approach similar operations of different streams are grouped together. Correct results are obtained with both approaches though both perform differently on the specific generation of GPU [34]. In [35] the authors have shown that for a certain application which uses D input data instances and defines B blocks of threads for kernel execution the data instances can be broken into *nStream* streams. Therefore, each stream now works with D/*nStream* data instances and B/*nStream* blocks and the memory copy of one stream overlaps the kernel execution of the other stream achieving a performance improvement. In our application depending on the size of the image the value of nStream varies.

5. EXPERIMENTS AND ANALYSIS OF RESULTS

All experiments in this paper are done on two different machines. Preliminary experiments are done on Intel Core i3-2375M with 4GB RAM and Geforce720m (GPU 1) and are further repeated on Intel Xeon CPU E5-1650 with 64GB RAM and Tesla K40C (GPU 2). The authors in the original paper [2] have reported the CPU execution time for the sequential LDP algorithm as 0.18sec for an image of size 88 x 88 pixels on a PC with 3 GHz CPU and 2 GB RAM. In our experiments we have used our own sequential implementation of

the algorithm. Using our implementation, the time taken for 88x88 pixels on GPU 1 is 0.078sec and the time taken on GPU 2 is 0.036sec. We have used high resolution images ranging from sizes 256x256 to 4096x4096 for our experiments. The experiments for medium image sizes using the global memory on GeForce 720m is presented in [29]. In this paper the LDP algorithm is implemented on large sized images using the fast-shared memory and further improved using the concept of L1 data cache and CUDA streams. High resolution images ranging from size 256x256 to 4096x4096 pixels are considered in this paper.

Geforce 720m is a card of compute capability 2.1 and a thread block can contain a maximum of 1024 threads and every SMP can process a maximum of 1536 threads whereas Tesla K40C is a card of compute capability 3.5, in which a thread block can contain a maximum of 1024 threads and every SMP can process a maximum of 2048 threads. The test images are taken from Corel database [36] and Essex face database [37]. Corel database consists of various categories like animals, cars, distractions and transport. For our experiments we have collected 1000 images from various categories of the Corel database to form the dataset. The Essex face database consists of 20 images each of 153 persons. We have taken 10 images of each 153 persons for our experiments. The approach followed in our parallel implementation is to achieve almost the same retrieval results as that of the sequential implementation. The images are divided into non-overlapping image blocks of size 16x16. The experiment is repeated for different image sizes and the performance gain of the kernel is analyzed. CUDA Visual Profiler is used to analyze the CUDA kernel performance [38]. We have developed two kernels for image retrieval, one for LDP histogram computation and another for the similarity matching using histogram intersection. For the implementation the query image is copied to the GPU global memory. The image is divided into 16x16 blocks and the LDP histogram values of each block are calculated. We have chosen a block size of 256 threads so that every SMP in GPU1 with compute capability 2.1 can schedule six blocks whereas GPU2 with compute capability 3.5 can schedule eight blocks by optimum resource utilization. The kernel is launched with the necessary threads to calculate the LDP histogram values. The computed LDP histogram is then transferred back to the CPU for normalization. The matching is then done by the second kernel using the histogram intersection method.

To avoid the CPU from being idle when the GPU is calculating, we efficiently divide the task of histogram intersection between the two in such a way that in minimum time the calculation is completed. This is done by launching the kernel asynchronously, i.e. whenever the kernel is executing in the GPU, the CPU is not blocked and is free to perform its own computations. Our strategy for balancing the work load between CPU and GPU is given below. Let R be the ratio of the time taken by the CPU to perform the task to the time taken by the GPU to perform the task. If T is the total number of images to be processed then images to the GPU is given by $\frac{T*R}{1+R}$ and images to the CPU is given by $\frac{T}{1+R}$ where R= $\frac{time\ taken\ by\ CPU}{time\ taken\ by\ GPU}$.

Figure 5 shows the percentage of allocation of the images to the CPU and GPU. The ratios *CPU/T=1/1+R* and *GPU/T = R/1+R* are plotted against the logarithmic values of the speedup. From Figure 5 it can be seen that if the execution time taken by the CPU and the GPU is almost the same then the images can be equally divided between the CPU and the GPU. In our experiments we allotted 50 images of size 256x256 pixels to the CPU and 950 images to the GPU. For images of size 4096x4096 we allotted 30 images to the CPU and 970 to the GPU. As GPU2, the Tesla machine is more powerful and more suitable for scientific calculations we have done all the computations in the GPU. In the following section we discuss the performance improvement achieved by our design space exploration.

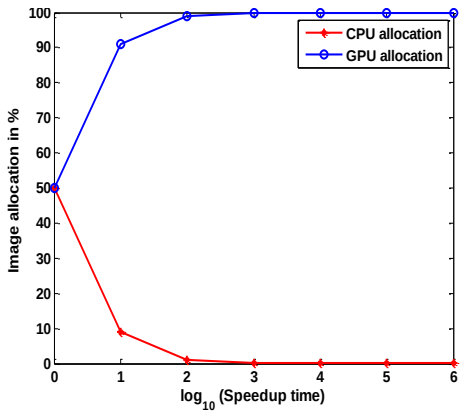
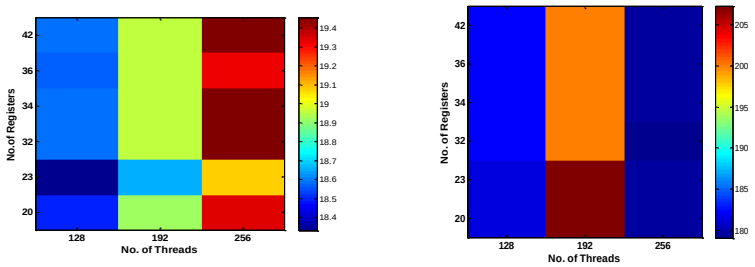


Figure 5. Optimal work load balance between CPU and GPU

5.1. Optimizing the Number of Threads and Registers

We have found optimized values for registers using the method discussed in section 4.1.2. We consider threads per thread block (N) and register allocation per thread (R) as input variables. The default register use (without register limit) of our LDP implementation is 42. The maximum value for N is 768 due to the constraint of register use. We explore various combinations of N and R values and analyze the performance improvement. Only a subset of the results is displayed here. As shown in Figures 6a and 6b the design space exhibits high variation in terms of execution time. The overall performance critically depends on both N and R. For example, for an image of size 256x256 the optimal setting is N=128 and R=23, whereas for an image of size 1024x1024 the optimal setting is at N=256 R=32. We have done an optimal choice by taking into consideration the maximum occupancy and minimum time. With the default number of registers, 42 in our case, the maximum number of active threads per multiprocessor is limited to 768 in a device of compute capability 2.1. By decreasing the number of registers to the optimal value i.e. 23 and a thread block size of 128, the active number of threads per multiprocessor is increased to 1024. In a device of compute capability 3.5 the number of threads is increased from 1280 to 2048 with the optimal choice of the number of registers per thread and the number of threads per thread block.

Combining the aforementioned various renewable energy resources with energy storage systems and diesel engine generator in an interconnected system, the generated electric energy can be effectively distributed and controlled to meet the energy requirement of the connected loads.



(a) (b)

Figure 6. (a) Optimal no. of registers and thread block size for an image of size 256x256, (b) Optimal no. of registers and thread block size for an image of size 1024x1024

5.2. Implementation using Global Memory

The first version of the algorithm was implemented using the global memory space. The detailed implementation procedure of the LDP algorithm using the global memory and the time comparison between the sequential and parallel version for medium image sizes on GPU1 was presented in [29]. Optimized number of registers and threads were considered for each image size. Figure 7 shows the time comparison of LDP_compute on GPU2 for image sizes 256x256, 512x512, 1024x1024, 2048x2048 and 4096x4096. As shown in Figure 7 the CUDA kernel gives more performance gain for higher size images than smaller images. This is because the host memory to device memory copy takes considerable amount of time and the speedup obtained must payoff for this copy overhead. In case of smaller sized images such as 256x256 pixels or 512x512 pixels, the speedup obtained is not enough to payoff for this copy overhead. However, as the image size increases to 2046x2046 pixels and 4096x4096 pixels this overhead is paid off and the performance gain increases.

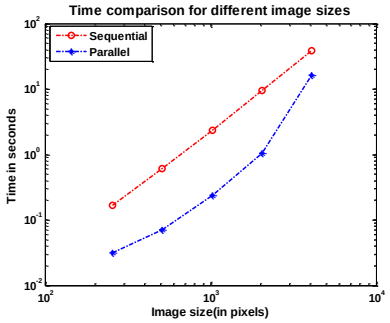


Figure 7. Time comparison between sequential version and parallel version in Tesla K40c when global memory is used. Logarithmic scales are used for plotting

This shows that the GPUs give excellent performance for large workloads which are highly parallel and throughput oriented. However, the large number of registers required by the thread will limit the number of active threads per multiprocessor. By reducing the number of registers per thread and by selecting an ideal thread block size the number of active threads per multiprocessor can be increased which in turn will increase the overall performance.

5.3. Imlementation using Shared Memory and L1 Cache

The next optimization done is to use the software managed faster on chip shared memory instead of the global memory. Shared memory enables inter thread data reuse as the threads in the thread block share the data stored in the shared memory. In our implementation we have carefully divided the images in such a way that the 48KB per block constraint of shared memory is never violated and also we use the maximum shared memory available for processing. The image is divided into 16x16 tiles and seven such tiles are copied to the shared memory for LDP computation. The sums of the LDP values in the four directions are then written to the shared memory for further use. Total amount of shared memory used per block is 7KB. We have taken the thread block size as 128 in GPU 1 so that 6 blocks are active in one SMP and the total shared memory used per SMP is 42KB. This falls within the 48KB limit at the same time schedules the maximum number of threads possible. The speedup obtained for in GPU1 when shared memory is used is shown in Table 2a. In GPU2 the thread block size is taken as 256 and 6 blocks are active in one SMP and the total shared memory used per SMP is 42KB. The speedup obtained in GPU2 when shared memory is used is shown in Table 2b.

The time taken for the computation when shared memory is used is much less compared to the parallel version which uses the global memory. This is because as the shared memory is on-chip it is much faster thanthe global memory. However, to ensure correct results there must be synchronization between the threads which share the memory. CUDA's simple barrier synchronization primitive *__syncthreads ()* is used for this.

This avoids the race condition as the thread's execution proceed past a *__syncthreads ()* barrier only after all the threads in the block have executed the barrier synchronization.

We have further studied the option of using a larger L1 cache in place of shared memory and its effect on improving the performance of the LDP_compute kernel. Default L1 size is 16KB but in cases when register spilling is problematic increasing the size of L1 to 48KB improves the performance. The *cudaDeviceSetCacheConfig* function is used to set preference between the shared memory and L1 cache. The slight improvement in speedup obtained by using L1 cache when compared to shared memory for the two GPUs is as shown in Tables 3a and 3b. The L1 cache version shows only a slight performance improvement when compared to the shared memory version.

5.4. Implementation using Streams

LDP_compute kernel is further optimized using streams. To allocate operations to different streams the input image is divided into small chunks and the required operations are performed on each chunk. The three main operations in the LDP computation is memory copy from host to device, execution of the kernels and copying the result back from device to host. For an input image of size say N= 256x256 size, the staged execution of *nStreams*, transfers chunks of N/*nStreams* size i.e. 128x128 pixels. If B blocks are used for the computation of the whole image in the non-staged version then in the staged execution of *nStreams*, B/*nStreams* blocks are used for computation. Also, while the first chunk is being computed by the B/*nStreams* blocks, the second chunk is being transferred thereby giving an improvement in performance. Figure 8a shows such a concurrency between transfer and computation with *nStream* set as 4. The host memories are allocated as page locked. We have assumed that the kernel execution and memory copies take roughly the same execution time. The execution time line is as shown in Figure 8b. The inter engine dependencies are highlighted with arrows. On a GeForce 720m there is only a single copy engine and single kernel engine. Depth first approach does not give any speedup whereas breadth first search gives a speedup as shown in Table 4a. In K40c there are two copy engines one for host to device transfer and another for device to host transfer and a single kernel engine. Both depths first and breadth first achieve the same speedup. The operations are queued across the streams in such a way that the operation of one stream does not block the operation of the other streams. This allows the GPU to execute copies and kernels in parallel, allowing the application to run faster as shown in Table 4b.

The image retrieval results using the parallel implementation of LDP is given in Table 5. The performance evaluation was done by calculating two parameters- true positive rate (TPR) and false positive rate (FPR) given by $TPR = TP/(TP + FN)$ and $FPR = FP/(FP + TN)$ where TP is True Positive, FN is False Negative, FP is False Positive and TN is True Negative. The four parameters TP, FN, FP, TN are evaluated as follows: Firstly, the Ground Truth (GT) for all the images in the database is recorded as TRUE or FALSE depending on whether the image belongs to the TRUE class or the FALSE class. In the second

step the histogram intersection of the LDP of the query image and the database images are calculated using the Equation 10. The histogram intersection values fall within the range 0 and 1. If the value is above a pre-determined threshold then the Test Outcome (TO) is deemed to be TRUE else the TO is FALSE. If both the TO and the GT are TRUE, then the TP count is incremented by 1. If both the TO and the GT are false, then the TN count is incremented by 1. If the TO is true and the GT is FALSE, then the FP count is incremented by 1. If TO is FALSE and GT is TRUE, then the FN count is incremented by 1. The TPR and FPR values for the two databases are calculated and the Receiver Operating Characteristic (ROC) curves obtained using these values are shown in Figures 9a and 9b. The threshold of the point that is closest to the top leftmost point (0, 1) is chosen as the one which gives the maximum TPR and minimum FPR (shown in circles in Figures 9a and 9b). These optimum threshold values are used to obtain the results shown in Table 5. The representative images from the two databases are shown in Figures 10a and 10b. Example retrieval results for a given query image from the two databases Corel and Essex using LDP is shown in Figures 11 and 12 respectively.

Table 2. (a) The time comparison for different image sizes when shared memory is used (GeForce 720m)

Image size	CPU time (sec)	GPU time (sec)	Speedup
256x256	0.209	0.003318	62.98x
512x512	0.834	0.013184	63.25x
1024x1024	3.574	0.05238	68.23x
2048x2048	14.988	0.209413	71.57x
4096x4096	61.459	0.838844	73.25x

Table 2. (b) The time comparison for different image sizes when shared memory is used (TeslaK40c)

Image size	CPU time (sec)	GPU time (sec)	Speedup
256x256	0.092	0.00048	190.95x
512x512	0.35	0.00178	195.75x
1024x1024	1.4	0.00696	201.17x
2048x2048	5.6781	0.02759	205.76x
4096x4096	22.944	0.11004	208.51x

Table 3. (a) Time comparison for different image sizes when L1 cache is used (GeForce 720m)

Image size	CPU time (sec)	GPU time (sec)	Speedup
256x256	0.209	0.003233	64.64x
512x512	0.834	0.012733	65.49x
1024x1024	3.574	0.05	70.46x
2048x2048	14.988	0.202673	73.957x
4096x4096	61.459	0.810159	75.86x

Table 3. (b) Time comparison for different image sizes when L1 cache is used (TeslaK40c)

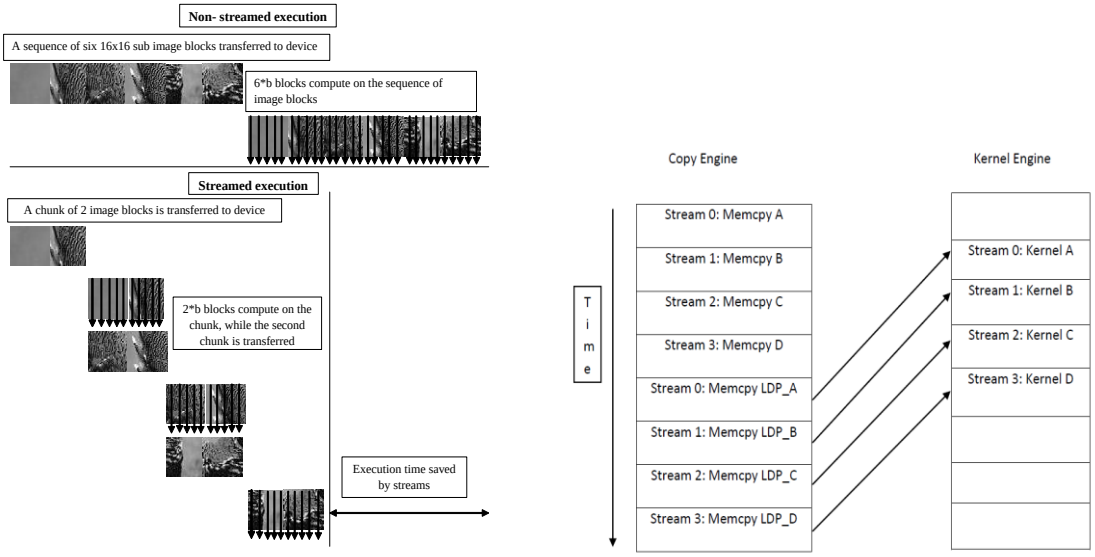
Image size	CPU time (sec)	GPU time (sec)	Speedup
256x256	0.092	0.00048	192x
512x512	0.35	0.001648	212.378x
1024x1024	1.4	0.006275	223.107x
2048x2048	5.6781	0.024649	230.358x
4096x4096	22.944	0.098124	233.826x

Table 4. (a) Time comparison for different image sizes when streams is used (GeForce 720m)

Image size	CPU time (sec)	GPU time (sec)	Speedup
256x256	0.209	0.003175	65.836x
512x512	0.834	0.012572	66.337x
1024x1024	3.574	0.049687	71.93x
2048x2048	14.988	0.19974	75.0375x
4096x4096	61.459	0.7973	77.0839x

Table 4. (b) Time comparison for different image sizes when streams is used (TeslaK40c)

Image size	CPU time (sec)	GPU time (sec)	Speedup
256x256	0.092	0.000479	192.067x
512x512	0.350	0.001772	197.52x
1024x1024	1.4	0.006927	202.11x
2048x2048	5.6781	0.02734	207.684x
4096x4096	22.944	0.1082	211.052x

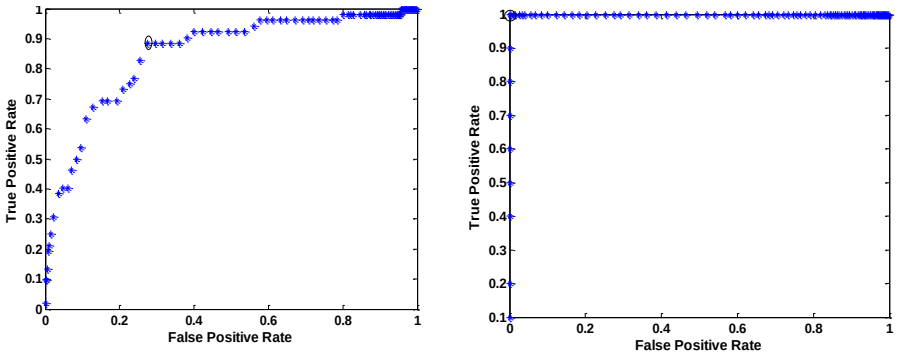


(a) (b)

Figure 8. (a) Computation on six image blocks for non-streamed and streamed execution (b) Execution timeline using multiple CUDA streams with arrows indicating inter engine dependencies.

Table 5. Retrieval results for the two databases

Databases	TP	FN	FP	TN	TPR	FPR
Corel	41	11	322	626	0.79	0.34
Essex	10	0	0	1520	1	0



(a) (b)

Figure 9. (a) ROC curves for the Corel database; the point with the optimum threshold is circled, (b) ROC curves for the Essex face database; the point with the optimum threshold is circled

6. POWER SYSTEM MODELLING

In this paper, we presented the design and implementation of a texture based CBIR algorithm called Local Derivative Pattern in GPGPU. We have implemented a parallel version of the LDP algorithm for high resolution images and showed the performance gain achieved. By using the fast on chip shared memory a speedup of around 240x times is achieved without compromising on the performance of the LDP algorithm. The parallel version of the algorithm is further revised to enable the use of additional parallelism in an attempt to improve the performance. We have achieved a slight improvement in the speedup when the hardware managed L1 data cache is used instead of the software managed cache. The synchronous parallel version was further revised to make use of the asynchronous programming using CUDA streams and a significant performance improvement was achieved over the sequential version. Our future work includes further further optimization of the algorithm using one image in one SMP. We will also be focusing on an in depth study into the tradeoffs between the shared memory and L1 cache.

(a) (b)

Figure 10. (a) Representative images from Corel database,
(b) Representative images from Essex face database

Figure 11. Top 5 images retrieved from Corel database using LDP, top leftmost is the query image

Figure 12. Top 5 images retrieved from Essex face database using LDP, top leftmost is the query image.

REFERENCES

[1] T. Ahonen, A. Hadid, and M. Pietikainen, "Face description with local binary patterns: Application to face recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28(12), pp. 2037-2041, Dec 2006.

[2] Zhang, B., Gao,Y., Zhao S. and Liu, J., "Local derivative pattern versus local binary pattern: face recognition with high-order local pattern descriptor," *IEEE Trans. Image Processing*, vol. 19(2), pp.533-544, 2010.

[3] "NVIDIA CUDA C Programming Guide," *NVIDIA Corporation*, vol.120 ,2011.

[4] Kusamura, Y., Amagasa, T., Kitagawa, H. and Kozawa, Y., "Efficient Content-based Image Retrieval for Position Estimation on GPU," *Proc. 15th ACM Int. Conf. on Advances in Mobile Computing & Multimedia*, Dec 04-06, Salzburg, Austria, pp. 58-66, 2017.

[5] Kusamura, Y., Kozawa, Y., Amagasa, T. and Kitagawa, H., "GPU Acceleration of Content-Based Image Retrieval Based on SIFT Descriptors", *Proc. 19th IEEE Int. Conf. on Network-Based Information Systems (NBiS)*, Sep 07-09, 2016, Ostrava, Czech Republic, pp. 342-347.

[6] Khor, H.L., Liew, S.C. and Zain, J.M., "A review on parallel medical image processing on GPU," *Proc. 4th IEEE Int. Conf. on Software Engineering and Computer Systems (ICSECS)*, Malaysia, pp. 45-48, Aug 2015.

[7] Sevilla, J., Jiménez, L.I. and Plaza, A., "Sparse unmixing-based content retrieval of hyperspectral images on graphics processing units," *IEEE Geoscience and Remote Sensing Letters*, vol.12(12), pp. 2443-2447, Oct 2015.

[8] Yang, Z., Zhu, Y., and Pu, Y., "Parallel image processing based on CUDA," in Proc. *IEEE Int Conf on Computer Science and Software Engineering(CSSE)*, Dec 12-14, Wuhen, Hubei, China, pp.198-201, 2008.

[9] Guo, H. and He, J., "A fast fractal image compression algorithm combined with graphic processor unit," *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol.13(3), pp.1089-1096, 2015.

[10] Tek, S.C. and Gökmen, M., "CUDA Accelerated Face Recognition Using Local Binary Patterns," *Threshold*, vol. 10101001(2), pp. 169, 2012.

[11] Gangodkar, D., Gupta, S., Gill, G.S., Kumar, P. and Mittal, A., "Efficient variable size template matching using fast normalized cross correlation on multicore processors," in *Advanced Computing, Networking and Security(ADCONS)*, Springer, vol. 7135, pp. 218-227, 2012.

[12] Prananta, E., Pranowo, P. and Budianto, D., "GPU CUDA accelerated image inpainting using fourth order PDE equation," *TELKOMNIKA (Telecommun. Computing Electronics and Control)*, vol.14(3), pp. 1009-1015, 2016.

[13] Nasreen, A. and Shobha, G., "Parallelizing Multi-featured Content Based Search and Retrieval of Videos through High Performance Computing," *Indonesian Journal of Electrical Engineering and Computer Science*, vol.5(1), pp.214-219, 2017.
[14] Ahmed, R., Islam, M.S. and Uddin, J., "Optimizing Apple lossless audio codec algorithm using NVIDIA CUDA architecture," *International Journal of Electrical and Computer Engineering (IJECE)*, vol.8(1), pp.70-75, 2018.
[15] Heidari, H., Chalechale, A., and Mohammadabadi, A.A.,"Accelerating of color moments and texture features extraction using GPU based parallel computing," in *Proc. 8th Iranian Conf. on Machine Vision and Image Processing (MVIP)*, Zanjan, Iran, pp.430-435, 2013.
[16] Zhu, F., Chen, P., Yang, D., Zhang, W., Chen, H. and Zang, B., "A GPU-based high-throughput image retrieval algorithm," in *Proc. 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* , ACM London, pp.30-37, Aug 2012.
[17] Prisacariu, V. and Reid, I., "fastHOG-a real-time GPU implementation of HOG," *Technical Report*, vol. 2310, Department of Engineering Science, Oxford Univ, Jul 2009.
[18] Park, I.K., Singhal, N., Lee, M.H., Cho, S.,& Kim, C. W., "Design and performance evaluation of image processing algorithms on GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, pp. 91-104, 2011.
[19] Hawick, K.A.,Leist, A.,and Playne, D.P.,"Regular Lattice and small-world spin model simulations using CUDA and GPUs," *Int. Journal of Parallel Programming*, vol.39(2), pp.183-201, 2011.
[20] Liu, S., Eisenbeis, C., and Gaudiot, J.L., "Value prediction and speculative execution on GPU," *Int. Journal of Parallel Programming*, vol.39(5), pp.533-552, 2011.
[21] Sanders, J. and Kandrot, E., "CUDA by example: an introduction to general-purpose GPU programming," *Addison-Wesley Professional*, 2010.
[22] Wen-Mei, W.H., "*GPU Computing Gems Emerald Edition*," Elsevier, 2011.
[23] Kirk, D.B. and WenMei, W.H., "Programming massively parallel processors: a hands-on approach," *Newnes*, 2012.
[24] Cui, X., Chen, Y., Zhang, C., and Mei, H., "Auto-tuning dense matrix multiplication for GPGPU with cache," in *Proc. IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, China, pp. 237-242, Dec 2010.
[25] Lobeiras, J., Amor, M., and Doallo, R., "Performance evaluation of GPU memory hierarchy using the FFT. ",in *Proc. 11th Int. conf. on Computational and Mathematical Methods in Science and Engineering (CMMSE 2011)*, Benidorm, Alicante, Spain, pp.750-761, Jun 2011.
[26] Liang, Y.,Cui, Z., Zhao, S., Rupnow, K., Zhang,Y., Jones, D.L. and Chen. D., "Real-time implementation and performance optimization of 3D sound localization on GPUs," in *Proc. Design, Automation&Test in Europe Conference & Exhibition (DATE)*, Dresden, pp. 832-835 , Mar 2012.
[27] Torres.Y., Gonzalez-Escribano, A., and Llanos, D.R., "Understanding the impact of CUDA tuning techniques for Fermi," in *Proc.IEEE Int.Conf. on High Performance Computing and Simulation*, Istanbul, pp.631-639, Jul 2011.
[28] Lee,D.,Dinov,I., Dong, B.,Gutman,B.,Yanovsky,I., and Toga,A.W.,"CUDA optimization strategies for compute and memory bound Neuroimaging algorithms," *Computer methods and programs in biomedicine*, vol.106(3), pp.175-187, 2012.
[29] Chandran, S.N., Gangodkar, D. and Mittal, A., "Parallel Implementation of Local Derivative Pattern Algorithm for Fast Image Retrieval," in *Proc. IEEE Int. Conf. on Computing, Communication & Automation (ICCCA)*, Greater Noida, India, pp.1132-1137, May 2015.
[30] Wittenbrink, C.M., Kilgariff, E., and Prabhu, A., "Fermi GF100 GPU architecture", *IEEE Micro*, vol.2, pp. 50-59, 2011.
[31] "NVIDIA Corporation: Whitepaper NVIDIAs Next Generation CUDA Compute Architecture: Kepler G*K110", [Online] Available: www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-G110-Architecture-Whitepaper.pdf.*
[32] Li,C.,Yang, Y.,Dai, H.,Yan, S.,Mueller,F., and Zhou, H., "Undertsnding the tradeoffs between software-managed vs hardware-managed caches in GPUs," in *Proc. IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, pp. 231-242, Mar 2014.
[33] Gomez-Luna,J.,Gonzalez-Linares,J.M., Benavides, J.I., Guil,N., "Performance models for CUDA streams on NVIDIA GeForce series," *Parallel and Distributed Computing* , vol.72(9), pp.1117-1126, 2012.
[34] "NVIDIA Parallel Forall," *[Online] Available: https://devblogs.nvidia.com/parallelforall/*
[35] Lian, Y., Cui, Z., Rupnow, K., and Chen D.,"Register and thread structure optimization for GPUs," in *Proc 18th Asia and South Pacific Design and Automation Conference (ASP-DAC)*, Yokohama, pp.461-466, Jan 2013.
[36] Corel 1000 and Corel 10000 image databases [Online]Available: http://wang.ist.psu.edu/docs/related.html.
[37] Face database [Online] Available: http://cswwwessex.ac.uk/mv/allfaces/faces94.html.
[38] CUDA Visual Profiler [Online] Available: http://developer.nvidia.com/nvidia-visual-profiler.

BIBLIOGRAPIES OF AUTHORS

Nisha Chandran S. received her M.Tech degree in Computer Science and Engineering from Graphic Era University. She is currently working towards the Ph.D. degree with the Department of Computer Science and Engineering, Graphic Era University, Dehradun, India. Her research interests include image and video processing, content-based retrieval, data mining and high performance computing.

Durgaprasad Gangodkar received the Ph.D. degree from Indian Institute of Technology (IIT), Roorkee, India. For around six years, he was with Siemens Ltd., Mumbai, India, where he worked on many prestigious projects related to industrial automation. He is currently with the Department of Computer Science and Engineering, Graphic Era University, Dehradun, India. His research interests include high-performance computing, computer vision, video analytics, and mobile agents.

Ankush Mittal received the Ph.D. degree in electrical and computer engineering from the National University of Singapore, Singapore, in 2001. He was a Faculty Member with National University of Singapore and IIT, Roorkee, India. He is currently the Director of Research with Graphic Era University, Dehradun, India. He has been the author or a coauthor of more than 240 research papers. His research interests include image and video processing, content-based retrieval, e-learning, and bioinformatics.