A CUDA BASED IMPLEMENTATION OF LOCALLY- AND FEATURE- ADAPTIVE DIFFUSION BASED IMAGE DENOISING ALGORITHM

Ali Pour Yazdanpanah, Ajay K.Mandava, Emma E. Regentova, Venkatesan Muthukumar,
Department of Electrical and Computer Engineering
University of Nevada, Las Vegas
Las Vegas, NV 89154, USA
Email: pouryazd@unlv.nevada.edu

George Bebis.
Department of Computer Science and Engineering,
University of Nevada, Reno,
NV 89557, USA
bebis@cse.unr.edu

***Abstract*—In this paper we introduce a parallel implementation of locally- and feature-adaptive diffusion based (LFAD) method for image denoising using NVIDIA CUDA framework and graphics processing units (GPUs). LFAD is a novel method for removing additive white Gaussian (AWG) noise in images reported to yield high quality denoised images [1]. It approaches each image region separately and uses different number of nonlinear anisotropic diffusion iterations for each region to attain best peak signal to noise ratio (PSNR). The inverse difference moment (IDM) feature is embedded into a modified diffusion function. As the method has attained highest performance in the class of advanced diffusion based methods and it is competitive with all the state-of-the-art methods, however computationally intensive when executed on the general purpose CPU. To improve the performance, we implemented using the CUDA computational framework. In order to minimize GPU kernel access to the global memory, we use shared memory and the texture memory per multiprocessor. The performance of the GPU implementation of the LFAD has been tested on the standard benchmark images. We demonstrate that with a single NVIDIA Tesla C2050 GPU we can expedite the sequential CPU implementation in most cases from 13 to 20 times.**

Keywords- LFAD, Image Denoising, CUDA Implementation, NVIDIA, GPU

I. INTRODUCTION

A certain level of noise can be present in images due to imperfection in formation, transmission, or recording processes and thus denoising methods are demanded to enhance image quality prior to visualization or analysis. However, none of the denoising methods can cope with the noise without degrading the original quality of the image. These degradations either affect edges, which are blurry or create artifacts, also introduce new "false" edges. If the denoising method is based on the block transformations, the inevitable a certain "blocking" effect is observed.

The locally- and feature- adaptive diffusion based image denoising (LFAD) method [1] has demonstrated highest performance in the class of advanced diffusion based methods and is competitive with all the state-of-the-art methods. However, the iterative LFAD process demands intensive computations which lower its efficiency for on-line applications.

Graphics processing units (GPUs) are efficient for highly parallel and computationally-intensive applications. GPUs architecture has evolved from application-specific architectures into general-purpose GPU architectures (GPGPUs) that can run any arbitrary computation. The Compute Unified Device Architectures (CUDA) platform by NVIDIA enables the GPU to solve complex computing problems. It also greatly simplifies the GPU programming [2]. CUDA has many unique properties that are very suitable for real time image processing applications.

In this paper we present a CUDA implementation for LFAD denoising algorithm and demonstrate that with a single NVIDIA Tesla C2050 GPU the speedup ranges from 13 to 20 when compared to the optimal sequential implementation.

The rest of the paper is organized as follows. Section 2, describes the phases of the LFAD image denoising method and associated algorithms. Section 3, presents the GPU hardware interface and programming model in the CUDA environment. Section 4, introduces the CUDA implementation and the GPU optimization. Section 5, presents results of the experiments. In Section 6, we draw the conclusions.

II. LFAD : LOCALLY- AND FEATURE- ADAPTIVE DIFFUSION

The LFAD denoising method is performed as follows:

a) Image is over-segmented into k approximately equally-sized patches(regions);

978-1-4799-3187-3/14 $31.00 © 2014 IEEE
DOI 10.1109/ITNG.2014.113

b) each patch is diffused individually until best PSNR is attained;

c) adjacent regions are merged based on a similarity metric;

d) diffusion repeats for merged regions until PSNR shows improvement. Subsections below discuss each of the above steps in more details.

A. Image Over-segmentation

As stated above, we need to start with an over-segmented image. For this purpose, we use the superpixel segmentation method. It groups pixels into perceptually meaningful regions that can be used instead of the rigid structure of the pixel grid.

The superpixel segmentation method works with a parameter k which is a desired number of approximately equally-sized superpixels. The procedure begins with an initialization step in which k initial cluster centers C_i are sampled on a regular S- pixel grid space. To produce roughly equally sized superpixels, the grid interval, S is set: $S = \sqrt{N/k}$, where N is the total number of pixels, The centers are moved to seed locations corresponding to the lowest gradient position in a 3x3 neighborhood, and thus avoid centering a superpixel on an edge. This reduces the chance of seeding a superpixel with a noisy pixel. Next, in the assignment step, each pixel i is associated with the nearest cluster center whose search region overlaps its location. The distance measure D, determines the nearest cluster center for each pixel. Since the expected spatial extent of a superpixel is a region of an approximate size of $S \times S$, the search for similar pixels is carried in a region of size $2S \times 2S$ around the superpixel center. Once each pixel has been associated with the nearest cluster center, an update step adjusts the cluster centers to be the mean vector of all the pixels belonging to the cluster. The L2 norm is used to compute a residual error between center locations of the new and previous clusters. The assignment and update steps are repeated iteratively until convergence.

B. Region Merging

If image I is partitioned into regions (initially, superpixels) $R_1, R_2, \ldots, R_m$, the following properties must hold true:

1. $R_1 \cup R_2 \cup \ldots \cup R_n = I$;
2. R_i *is connected;*
3. $R_i \cap R_j$ *is empty.*

The regions are merged based on the similarity metric which is chosen to be the intensity variance. Let us denote a pair of adjacent regions $Ri \sim Rj$ and merged regions $Ri \cup Rj$. The region merging algorithm is performed as follows:

1. For $\forall\ Ri \sim Rj$, if $\sigma_j^2 \le \alpha * \sigma_i^2$ then $Rm = Ri \cup Rj$
2. If $Rm \ne I$, Increment α. Goto Step 1; otherwise Goto Step 3
3. Stop.

Here α is a constant multiplier, determined experimentally.

C. Diffusion

The equation below describes the diffusion process

$$\frac{\partial}{\partial t} I(x,y,t) = \nabla \bullet (c(x,y,t)\nabla I), \qquad (1)$$

Where I(x,y,t) is an image, t is the iteration step and c(x,y,t) is a monotonically decreasing diffusion function of the magnitude gradient.

The LFAD method uses the normalized inverse difference moment (IDM) feature in the equation (1) for $c(x,y,t)$.

$$c = \exp\left(-\left(\frac{IDM(I)}{\lambda}\right)^2\right) \qquad (2),$$

where λ is referred to as a diffusion constant [3], and IDM is calculated as in (3).

$$IDM = 1 - \sum_{i=0}^{G-1}\sum_{j=0}^{G-1} \frac{1}{1+(i-j)^2} P(i,j) \qquad (3)$$

A value of IDM equal to zero indicates a pixel being part of a homogenous neighborhood. A value close to one indicates that the pixel belongs to texture or an object boundary.

Consider MxN neighborhood containing G gray levels, let f(m,n) be the intensity at sample m, line n of the neighborhood. Then

$$P(i,j \mid \Delta x, \Delta y) = W \cdot Q(i,j \mid \Delta x, \Delta y) \qquad (4),$$

where

$$W = \frac{1}{(M-\Delta x)(N-\Delta y)};$$

$$Q(i,j \mid \Delta x, \Delta y) = \sum_{n=1}^{N-\Delta y}\sum_{m=1}^{M-\Delta x} A \qquad (5)$$

and

$$A = \begin{Bmatrix} 1, iff\ (m,n) = i \quad and \quad f(m+\Delta x, n+\Delta y) = j \\ 0,\ elsewhere. \end{Bmatrix} \qquad (6).$$

D. LFAD Algorithm code

Let us denote I as the input image, k the number of regions, m – number of merging steps, *Var* –intensity

variance and n is a number of diffusion steps. The method performs according to the following steps:

1. Initialize $m=0$, $\alpha = 1.1$, $\lambda = 10$. Segment image into k $(k \neq 1)$ regions.
2. Initialize $n=0$. Calculate PSNR for each region of initial partition, i.e., $[PSNR_k^{(0)}]_0$.
3. Iteration step: Diffuse image pixel $I_{i,j}$ using Eq.(7).

$$I_{i,j}^{n+1} = I_{i,j}^{n} + (\nabla t) \bullet \begin{bmatrix} c_N\left(\nabla_N I_{i,j}^{n}\right) \bullet \nabla_N I_{i,j}^{n} + c_S\left(\nabla_S I_{i,j}^{n}\right) \bullet \nabla_S I_{i,j}^{n} + \\ c_E\left(\nabla_E I_{i,j}^{n}\right) \bullet \nabla_E I_{i,j}^{n} + c_W\left(\nabla_W I_{i,j}^{n}\right) \bullet \nabla_W I_{i,j}^{n} \end{bmatrix} \quad (7)$$

Subscripts N, S, E, and W (North, South, East, and West) describe the direction of the local gradient, and the local gradient is calculated using nearest-neighbor differences as

$$\nabla_N I_{i,j} = I_{i-1,j} - I_{i,j};$$
$$\nabla_S I_{i,j} = I_{i+1,j} - I_{i,j} \quad \nabla_E I_{i,j} = I_{i,j+1} - I_{i,j};$$
$$\nabla_W I_{i,j} = I_{i,j-1} - I_{i,j} \quad (8)$$

4. For $\forall R_i$: *if* $[PSNR_k^{(n+1)}]_m > [PSNR_k^{(n)}]_m$, Goto Step 3; else Goto Step 6.
5. While $R_m \neq I$, *for* $\forall$ $Ri \sim Rj$, *if* $Var(R_j) \leq \alpha * Var(R_i)$, then $R_i \cup R_j$: $m=m+1$; update k;, Goto Step 2, else Repeat Step 5 with $\alpha = \alpha + 0.1$.
6. Stop.

Here, PSNR is calculated according to Eq. (9)

$$PSNR = 10 \log_{10}\left(\frac{I_{\max}^{2}}{MSE}\right) \quad (9)$$

where $I_{\max}$ is the maximum intensity of the image, and MSE is the mean square error:

$$MSE = E\left[\left(I_{Original} - I_{Denoised}\right)^2\right] \quad (10)$$

The denoising result of the LFAD method for additive white Gaussian (AWG) noise with σ =20 is shown in Fig. 1.

III. CUDA ARCHITECTURE

At the software level, the parallel codes, written in the CUDA environment, are divided into CPU part (host code) and GPU part (device code). At the beginning, a part of the code is executed on the CPU, then all necessary data are copied to the GPU and the data-parallel functions running on the GPU; finally, the results are copied back to CPU. Applications code is divided into independent tasks. These tasks are parallelized by scalar execution units called threads. A set of threads, called blocks, run on multiprocessor at a given time, and the threads within a block can share data. One can use several points of synchronization to control the execution flow of all the threads in each block. A set of blocks can be assigned to a single multiprocessor and their execution is time-shared. The collection of all blocks in a single execution is called a grid. Fig. 2 shows the CUDA architecture model as a collection of blocks running in parallel.

Figure 1. Left: "Lena" image with AWG noise, σ =20. Right: image results by the LFAD denoising method.

At the hardware level, the CUDA boards contain a set of single instruction multiple data (SIMD) stream multi-processors (SM), and each SM includes several stream processors (SP). GPUs have much longer RAM delay, smaller cache and poorer branch prediction than CPUs so if the parallel code doesn't have any high concurrency degree; the improvement is not significant [4].

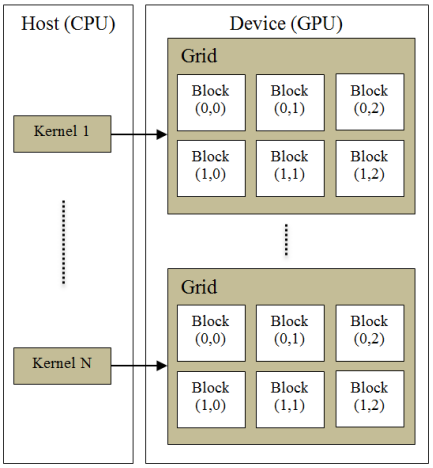

Figure 2. CUDA architecture model

The CUDA programming environment does not have any GPU memory restrictions, and thus the whole CUDA memory is available, however there will be different access times for different types of memory in GPU. The CUDA memory includes global (device) memory, shared memory, and Constant memory (Fig. 3). All threads can access global memory. For each block, shared memory is available for all threads within the block, while registers are the local storage for each SP. Register and shared memory are much faster than device memory that can be used to speed up the access. Constant memory and texture memory are read only memories accessible for all threads. Texture memory is a device memory which is cached for locality and constant memory is cached memory that can be written by the CPU and read by the GPU. Highest throughput can be achieved by

accessing consecutive memory locations by the threads simultaneously, that is with the memory access coalescing.

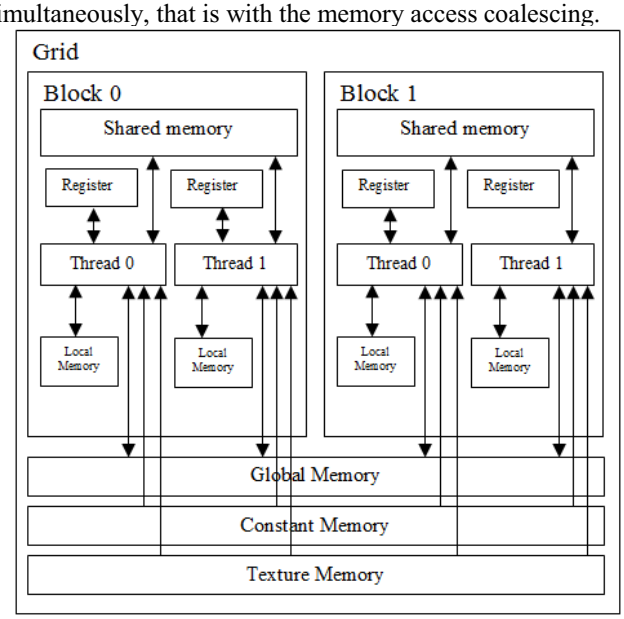

Figure 3. CUDA memory model.

IV. CUDA IMPLEMENTATION AND GPU OPTIMIZATION

In our implementation, we assign each thread process to a single pixel to have all pixels simultaneously processed. The GPU version was implemented using four CUDA kernels: (1) Superpixel Segmentation, (2) Diffusion, (3) PSNR Calculation (4) Region merging. The IDM is calculated on the CPU. The flow chart is shown in Fig. 4. Memory transfers are shown by wide arrows. The kernels running in GPU are called by host.

To increase the performance, we have used two general optimization techniques such as memory management overhead reduction and the memory transfer overhead reduction [5].

In CUDA, memory allocation (*cudaMalloc* and *cudaFree*) are more intensive operations than standard C functions (*malloc* and *free*). Therefore we have allocated the GPU memory just once at the beginning and then we accessed and changed that memory in any kernel calls, finally at the end we just brought the results back from the GPU memory to the host just one time. For memory transfer overhead reduction, we have to avoid unnecessary data transfers between GPU and CPU during the execution of the method. So we performed most of the computationally expensive procedures in GPU.

In the following subsections we describe the CUDA implementation of each stage of the LFAD algorithm.

A. IDM Feature Calculation

Since the IDM (Fig. 5) is calculated once only and it is based on the input image, we execute this stage on CPU and transfer the feature values to the GPU at the beginning of the algorithm. We load the IDM values as an one dimensional texture memory per thread.

By loading the IDM values to texture memory, we can take advantage of a special architecture of the GPU which provides an on-chip caching that is secure and more efficient off-chip memory access.

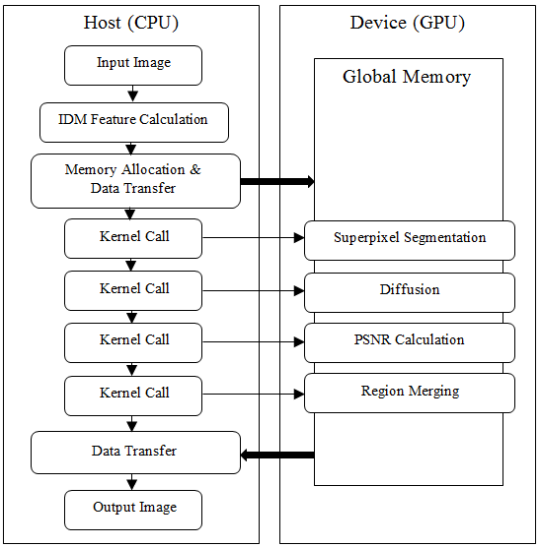

Figure 4. CUDA-based LFAD algorithm flow chart.

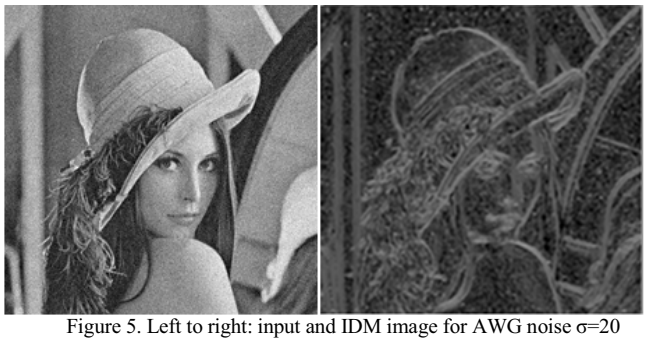

Figure 5. Left to right: input and IDM image for AWG noise σ=20

B. Superpixel Segmentation

In this stage the SLIC (Simple Linear Iterative Clustering) method is employed that efficiently groups pixels to nearly uniform superpixels [6,7]. In [8], authors introduced an optimized CUDA implementation of the SLIC superpixel method. They achieved speedups of 10x to 20x times. Due to simplicity and efficiency of this method we have adopted this implementation on CUDA for our algorithm. Figure 6, shows the result of the application of the method on Lena benchmark image.

C. Diffusion

Due to repetitive iterations in the LFAD diffusion process, we store neighborhood pixel values in four directions (N,S,W,E) in 2D texture memory. The diffusion procedure updates all the pixel values simultaneously due to processing of multiple threads in parallel. Listing 1, provides an example of 2D texture memory definition for the diffusion procedure.

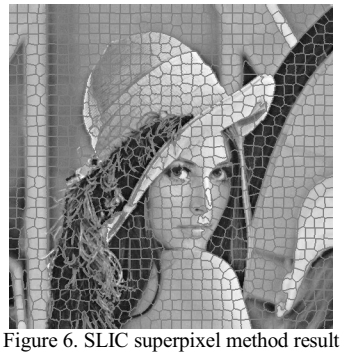

Figure 6. SLIC superpixel method result

The texturing hardware in CUDA has a boundary handling property for image processing kernels. Basically, when one defines a texture reference, the address mode is specified. This mode describes the behavior when textures are accessed out of bounds and can be set to clamp or repeat accesses.

```
texture<float, 2, cudaReadModeElementType> textImage;
float x=(float)ix+0.5f;      // ix = each threadIdx.x
float y=(float)iy+0.5f;      // iy = each threadIdx.y
float  t    = tex2D(textImage, x, y);
float  tN  = tex2D(textImage, x-1, y);
float  tW = tex2D(textImage, x, y-1);
float  tS   = tex2D(textImage, x+1, y);
float  tE  = tex2D(textImage, x, y+1);
```

Listing 1. 2D texture memory definition for diffusion process

The diffusion kernel is implemented on a grid of *W/32×H/32* thread blocks (*H & W* are the image dimensions). Before doing any arithmetic operation, data accessed by each thread block are read first from the global memory into the texture memory because being an on-chip memory the latter has lower than of the global memory latency and a much higher bandwidth. In the diffusion kernel, as illustrated by Fig.7 (a), in every iteration, for minimizing the repetitive access to off-chip memory, each thread reads neighborhood pixels from the four-connected neighborhood, N,S,W, E into texture caches and applies the corresponding diffusion equation.

D. PSNR calculation

To calculate the patch-based PSNR values in parallel, we have used the shared memory on the GPU. We have assigned a region per block and pixel values to the threads inside the blocks. For the MSE calculation, because the result needs to be the sum squared of all pairwise subtractions, each thread keeps a running sum of the pairs it has subtracted.

We have declared two buffers of the shared memory, one used to store each thread's running subtraction and the other one for comparison and the maximum value of the intensities. In CUDA, the variables in shared memory are handled different from ordinary variables. CUDA generates a copy of a variable for each block launched on the board. In this case, we can declare a shared memory array using the *__device__ __shared__ qualifiers* in CUDA. Since the shared memory variables reside physically on the GPU, the latency to access this memory is considerably lower than ordinary variables.

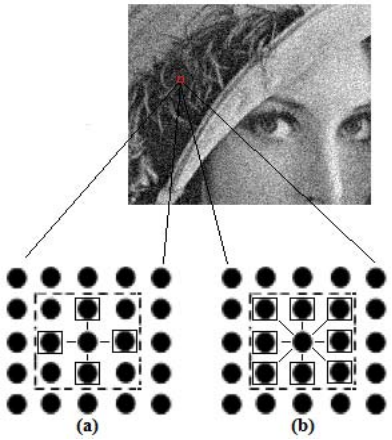

Figure 7. 2D texture memory in a) diffusion and b) region merging stages.

E. Region Merging

The region merging kernel is also implemented on a grid of *H/32×W/32* thread blocks. Based on LFAD method, we are merging the adjacent region that hold this condition: $Var(R_j) \leq \alpha^* Var(R_i)$. First, in order to find the adjacency map for the image, we used the approach used in the diffusion stage, but here each thread reads values from the eight- neighborhood (Fig.7 (b)) into the texture caches. The class labels in the adjacency map are updated in parallel.

V. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of the LFAD algorithm on both CPU and GPU. As for the CPU implementation, the algorithm is tested using single-threaded implementation in Matlab. Both CPU and CUDA implementations of the algorithm have been performed on a workstation whose characteristics are as in Table I. In Table II we provide the properties of NVIDIA Tesla C2050 used in this work.

TABLE I. TEST SYSTEM MAIN CONFIGURATION

Parameters	***Values***
CPU	Intel core 2 Duo E8400 (3.00GHz)
RAM	4 GB
GPU architechture	NVIDIA Tesla C2050
OS	Microsoft Windows 7
CUDA	V4.0

We compare the execution times of the CPU and CUDA versions of the LFAD method in Fig. 8 to illustrate the accelerating performance of CUDA. Both execution times of CPU and CUDA versions are average values of execution

over multiple runs. We can see that the execution time of CPU version grows rapidly as the noise level (σ) increases, while the CUDA version grows slowly. The speedups achieved with the proposed CUDA implementation range from 13x to 20x, as illustrated in Fig. 9.

According to the experimental results in Fig. 8 and 9, we observe that: (a) compared to the serial single-threaded algorithm running on the CPU, the parallel algorithm on the GPU through the CUDA environment improved the performance significantly. For the input image of 512 x 512 pixels, the speed up is up to 20 times.

Because GPU has high parallel architecture, multiple input data blocks can be processed simultaneously. (b) The speed up is increases with the increase in noise level. When the noise level is low, the computation load of CUDA threads is low. When noise level is high, the threads computation load is sufficient and context switch cost is reduced. Although there is an improvement of the speed for the method, the iterative nature of the diffusion process is creating a bottleneck for parallel implementation. The future research will be conducted on finding modifications of the method and algorithms for better runtime on the given architecture without or with a minimum degradation of the original image quality.

TABLE II. PROPERTIES OF NVIDIA TESLA C2050

Parameters	*Values*
Number of CUDA cores	448
Memory Speed	1.5GHz
Dedicated Memory	3GB
Clock Rate	1.15 GHz
Memory Bandwidth	144 Gb/sec

VI. CONCLUSIONS

In this paper, we have presented a novel parallel implementations of locally- and feature-adaptive diffusion based LFAD method for image denoising using NVIDIA CUDA framework and graphics processing units. The final CUDA implementation of this method provides a speedup of 13x to 20x over the single-threaded Matlab implementation. By assuming perfectly linear scaling, for matching the performance of this CUDA implementation, would require about 14 - 20 CPU cores equivalent to the cores used in our experiment. The speedup is achieved by different optimization techniques such as memory management overhead reduction, memory transfer overhead reduction and by taking advantage of the CUDA memory patterns , such as shared and texture memories.

ACKNOWLEDGMENT

This material is based upon work supported by NASA EPSCoR under Cooperative Agreement No. NNX10AR89A.

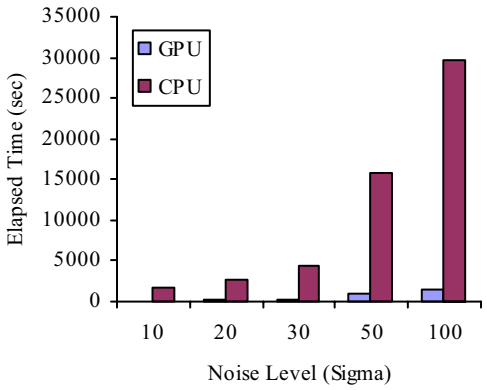

Figure 8. Execution times for Lena image using CUDA and Matlab under different noise levels.

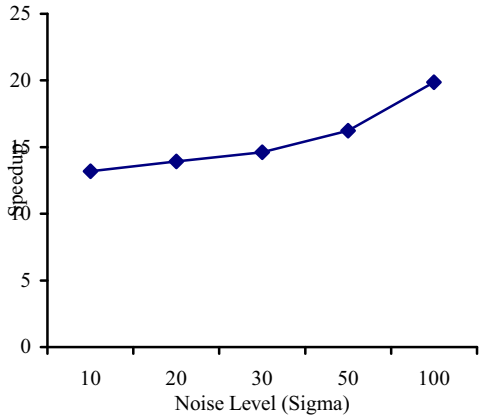

Figure 9. Speedup by CUDA vs the image noise level.

REFERENCES

[1] A. K. Mandava, E. E. Regentova, G.Bebis, "LFAD: Locally- and Feature-Adaptive Diffusion Based Image Denoising," Applied Mathematics & Information Sciences, No. 1, 1-12 (2014).

[2] CUDA Programming Guide, 2013. http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[3] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 12, no. 7, 1990.pp. 629–639

[4] Bo Shan, Jianjun Qi,and WeiLiu, "A CUDA-Based Algorithm for Constructing Concept Lattices", RSCTC 2012, pp. 297–302, 2012.

[5] Boyer M., Tarjan, D., Acton, S.T., Skadron, K, "Accelerating leukocyte tracking using CUDA: A casestudy in levera ging manycore coprocessors," IEEE International Symposium on Parallel & Distributed Processing, pp. 1–12, 2009

[6] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk,SLIC Superpixels Compared to State-of-the-art Superpixel Methods, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 34, num. 11, p. 2274 - 2282, May 2012.

[7] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk,SLIC Superpixels, EPFL Technical Report no. 149300, June 2010.

[8] Carl Yuheng Ren and Ian Reid, gSLIC: a real-time implementation of SLIC superpixel segmentation, University of Oxford, Department of Engineering Science, 2011.