Kick-start to Image Processing Algorithms on CUDA

(Pending Publication)

Siddharth Kumar and Nishant Basu

IIIT BHUBANESWAR

Abstract

The concept of parallel programming started back in the initial days of intel chips, when the DMA controllers were used to do the parallel task of data transfer, while CPU would be engaged in other processing work. But it was limited to data transfer tasks only. So, we saw the rise of GPUs, which were capable of performing parallelly and so is the evolution of CUDA. A framework to parallelly run tasks on the GPU, and reduce the time required for processing drastically. As it is a known fact, that image processing is a tricky task when we look into its complexity, so utilizing the huge power of GPU is a nice fit. The literature in terms of image processing on the CUDA framework is very scattered, and so our aim is to present a paper which is a one stop junction for all the major algorithms in image processing, performed parallelly with the help of CUDA framework. In this way, the comparison and study is easy. We will be focused on the software side of the parallel processing rather than focusing on the hardware. Our aim is to kickstart a reality based on image processing on CUDA and making it mainstream, as it reduces the time exponentially and suits the modern era of real-time applications.

1. Introduction

In the digital universe, the world of images is very simple. They are simply very large and small matrices and so, image processing is nothing less than an overload of operators on these matrices.

Then the question arises, is it so hard that we need to perform these operations parallelly? and the answer lies in what operation you perform, and what is the application of your operation. For instance, if we took into account an algorithm known as OILY STYLE NPR, which is basically generating a new image as an oil artistic view of the original image, and apply it to an image of size 2278 x 1712, then the CPU takes 94k ms(milliseconds), so if you want an artistic view of your portrait photograph, you need to have patience, but what if we do the same task on multiple threads on a GPU, we can safely say that it's good for non-patient people, because it takes only 1k ms. This is roughly a 98% decrease of time, which when seen in terms of real time applications, is groundbreaking (Saxena, Sharma & Sharma, 2016).

GPUs are in the core of the era of parallelism and fast processing. The NVIDIA GPUs have 700 to 4000 cores, this huge number indicates the degree in which hardware has developed to suit parallelism, so the only problem lies in the software environment to fit this, and the needs of real-time (Garland et al., 2008).

CUDA is a software environment and programming model to facilitate scalable parallel programming. It extends native language approach of C/C++. It suits the need of massive multithread GPUs and scalability across various hardware (Garland et al., 2008).

In this paper, we will try to ask following questions and hope that we find some answers in all the literature we are going to go through.

Q1- What is CUDA and how does it implement parallelism?

Q2- What are the different aspects of Image processing and why we require parallel processing for it?

Q3- What are the ways to measure these performances

Q4- How is it different from CPU Image processing?

Q5- What are the different categories of Algorithms for Image Processing?

Q6- How is CUDA trying to solve these problems efficiently?

The paper is structured in a way that in section [2] and [3], we will learn about CUDA and Image processing along with the ways to measure the performance efficiently. Then we will dive into section [4], which is the list of major categories of algorithms. In section [5], which is the Literature review section, we will look into the literatures related to CUDA implementation, and analysis of one of the algorithms of these categories, so that it can kickstart the implementation journey. Then all these will be comprehended in a tabular form in section [6], where we look into the methods and their results for better understanding. In section [7], we will analyses the results of section [6], and then we will conclude with what we gain in this journey in section [8].

2. CUDA (Compute Unified Device Architecture)

GPU is a collection of huge number of parallel processors, the NVIDIA G80 constitutes of 128 processors all of them support many active threads, it goes as much as 12,288 on a typical G80, but with this massive processing capability there must be a programming model that can support this tremendous parallelism. The focus must be on data parallelism and CUDA implements such a programming model (Kirk, 2007).

CUDA is a programming model that focuses on two major design goals:

1. Extending native programming languages like C/C++ with minimal abstraction so that it can support parallelism.

2. To develop highly scalable parallel code that can work on many concurrent threads and processor cores. (Garland et al., 2008)

Parallelism through CUDA (Kirk, 2007)**:** Different parallel portion of an application are executed on a GPU by the use of kernels, which is basically a function which runs on a GPU. Each kernel is executed once and an array of threads execute a CUDA kernel, these threads are identified by ids, which is used to make decisions at control time and to calculate memory addresses. These threads are very lightweight in CUDA. The switching is instant and very fast.

But these threads need to cooperate to share memory, results and synchronization and for that purpose only the execution of kernel in CUDA is taken care by a grid, which consists of thread blocks (Fig 1.). These add the advantage of synchronization in execution, and sharing of memory between them, so in that way property of cooperation is established between these blocks.

There's another popular approach for parallelism which is GPGPU, which is GPU through graphics API, but CUDA has various advantages over it like shared memory and thread synchronization. In addition to these, it also has a very low learning graph, as it is an extension to previous programming languages like C and C++, and hence it is the most suitable approach for parallelism.

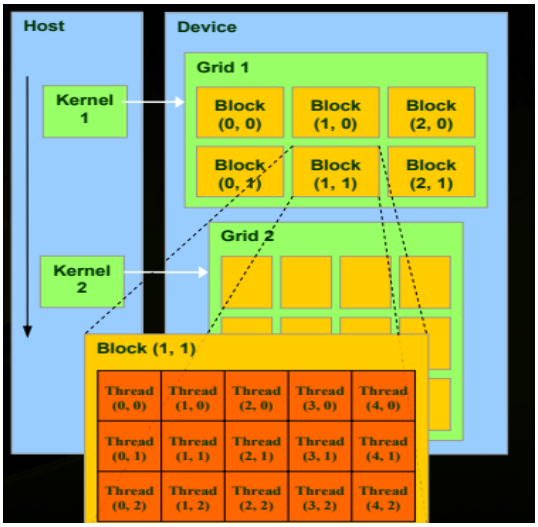

Fig 1. (Kirk, 2007)

Overall, the CUDA model provides various benefits:

- Thousands of lightweight threads with no switching.
- Shared memory and thread cooperation.
- Random access to global memory.

It's the most efficient use of current generation of Hardware and well suited for parallelism, hence in this section we got the answer for Q1.

3. Image Processing

In our terms, simply put, Image is a 2-D matrix of different pixel values and processing these pixels is called image processing. It's important for various purposes, because it has the widest variety of applications one can possibly think of, and it has become more and more important in this age of the internet.

The task of image processing as mentioned by (González, 2001) is generally done at these different levels:

A. Low level image processing

These are operated at the pixel level and input to these operators is an image, whereas output is an image or data.

B. Intermediate level image processing

These are operated at the abstraction level, which is derived from pixels of image (or low-level processing), so that it can help making other decisions about image. Example - region labelling.

C. High level image processing

These are operated on even higher abstraction levels i.e. the abstraction derived from intermediate levels; these are generally used to derive content of the image. The high-level image processing operations operate in order to generate higher abstractions. E.g. classification and object recognition. (Prajapati & Vij, 2011).

But not all of these levels require so much of processing. But the rise of real time applications needs the processing to be faster. Harshad says in (Prajapati & Vij, 2011) how in the near

future there can be searches based on image processing, there can be other widest sophisticated use of different image processing techniques in e-commerce like a 360 degree view, and all this is in a real time, and so the processing speed becomes a key factor in all these applications. But how do we measure the processing powers? and the answer to this is explained in (Saxena, Sharma & Sharma, 2016), where he gives various performance measures for quality of an algorithm in parallel environment. The brief introduction of these are given as follows which answers our Q3.

Parallel run time:

As mentioned by ("Rajaraman and Siva Ram Murthy, 2006") Time requirement of a n-processor parallel computer for execution of the program is parallel run time.

Throughput:

Throughput tells us how much data can be transferred from one location to another in a given amount of time.

Speedup (Hennessy & Patterson, 2011):

As defined by Amdahl's law in (Hennessy & Patterson, 2011) speedup is the enhancement in speed when the enhancement in the computer, mathematically it is:

Speedup = (Execution time for entire task without using the enhancement) / (Execution time for entire task using the enhancement when possible) (A)

Efficiency:

It measures efficiency of use of processors in a parallel program. It can be expressed as:

$E_p=S_p/p = Ts(n,1)/pT(n, p)$ (B)

where, here Ts(n,1) is time of best sequential algorithm, Ep denotes efficiency, Sp denotes speedup.

But we already had the power of the CPU, Why switch? The answer is beautifully explained in table 1 by (Saxena, Sharma & Sharma, 2016) here they compare the approximate time between GPUs and CPUs on some major algorithms.

Table 1 (Saxena, Sharma & Sharma, 2016)

Algorithms	**Environment**	**Image Size**	**Execution time (CPU) (ms)**	**Execution time (GPU) (ms)**
Image Brightening	GeForce 430 GT (96 cores), 1 GB RAM, Linux OS, AMD 3.2 GHz Quad-core	256×256	8.847117	0.19903
		512×512	36.12278	0.761405
		1024×1024	142.6773	2.995606
		1800×1400	342.4271	7.20223
		4000×3000	1610.854	33.97197
Darkening image transformation	Same as above	256×256	9.5512192	0.2016992
inverse sinusoidal contrast transformation	Same as above	256×256	11.26008	0.203088
		512×512	46.3111264	0.776816
		1024×1024	184.731943	3.057152
		1800×1400	448.714719	7.3071904
		4000×3000	2040.16223	34.6397216
Hyperbolic tangent contrast transformation	Same as above	256×256	5.681072	0.1872896
		512×512	22.7657601	0.7213312
		1024×1024	91.6525903	2.823536
		1800×1400	219.788937	6.8108832
		4000×3000	1054.94413	32.2680701

Sine contrast transformation execution times	Same as above	256×256 512×512 1024×1024 1800×1400	12.5076096 51.3341637 205.06483 500.958685	0.2077856 0.7901344 3.1164224 7.4421792
Linear feature extraction	CPU: Q9450 GPU: NVIDIA GeForce 9800 GTX(128 cores)	512×512 1024×768 1280×1024 1200×1800 2278×1712	109 422 610 1250 2375	54.77 166.63 250.42 471.88 1018.98
JPEG2000 encoding (DWT)	Same as above	512×512 1024×768 1280×1024 1200×1800 2278×1712	31.85 150.60 164.50 264.93 471.66	7.84 20.72 31.17 51.52 91.08
JPEG2000 encoding (Tier-1)	Same as above	512×512 1024×768 1280×1024 1200×1800 2278×1712 3024×2089	94 234 328 891 1640 1500	205 390 484 735 1468 2062
Cartoon style NPR	Same as above	512×512 1024×768	4594 14594	49.31 149.66

		1280×1024 1200×1800 2278×1712	18688 47688 93891	243.35 406.53 741.42
Oily style NPR	Same as above	512×512 1024×768 1280×1024 1200×1800 2278×1712	7172 15609 35313 49047 94406	87.77 226.22 334.83 589.78 1107.12
Multi View stereo matching	Same as above	Temple ring (47 Images)	18422	340
Corners and edge detection	GPU: GeForce GTX 280(240 CUDA cores) CPU: Dual Core 6600, 2.40 GHz	2048×2048	4006	1240
Content authentication	Intel xeon 5520(2.26GHz) Tesla C1060 CUDA 2.3	1024×1024	28877.66	903.83
Binarize	Intel core-i7 GeForce GTX470(448 cores)	2048×2048	106	0.34
Copy	Same as above	2048×2048	99	0.34
Transpose	Same as above	2048×2048	88	0.50

Blur	Same as above	2048×2048	208	0.74
Sobel	Same as above	2048×2048	230	1.21
Erode	Same as above	2048×2048	151	0.65
Dilate	Same as above	2048×2048	445	1.67
Gradient	Same as above	2048×2048	432	1.45
Sum	Same as above	2048×2048	37	0.28
Max	Same as above	2048×2048	41	0.57
Min	Same as above	2048×2048	41	0.56
Histogram	Same as above	2048×2048	213	0.47

Tools and Technology

(Prajapati & Vij, 2011) has compiled the tools already available for implementing the parallel processing paradigm which is shown in Table 2. Here we are focusing on software tools only, and we assume that the hardware requirements are already taken care of.

Table 2

Tool and Technology	**Details**	**Parallel/ Distributed Processing**

MPI	Communication Protocol specifications	Parallel processing+Distributed Computing (low latency)
OpenMP	Parallel applications API	Parallel processing
CUDA	Nvidia GPU engine	Parallel processing
Cilk	Programming Language for parallel programming	Parallel processing
TBB	C++ Library for parallel processing	Parallel processing
MATLAB's PCT with MDCS	MATLAB's support for parallel computing.	Parallel processing+Distributed Computing

Why parallelism?

As we even discussed in the introduction section that how an OILY STYLE NPR has 98% reduction in time, and now we came to know the difference between the approximate time requirement of all these algorithms on CPU and GPU. We can see how there is a significant decrease in time that we can achieve through GPU and parallel processing. As we conclude and as said by Eric and Frank in NVISION 08:

"Image processing is a natural fit for data parallel processing because"

- It's easy to appoint each pixel to a thread.
- There's a need to share data between pixels.

And we saw in the CUDA section how the CUDA programming model is the most state of the art architecture which elegantly implements all of these conditions and so it answers our Q2.

4. List of Algorithms.

As mentioned on the source ("List of algorithms", 2020) the main classes of image processing algorithms are as follows:

- Contrast Improvement
- Connected Component Labeling
- Half toning and Dithering
- Difference map Algorithms
- Feature observation
- Richardson–Lucy algorithm
- Blind deconvolution
- Median Filter
- Seam carving
- Image segmentation

And this list brings answers to Q5. Now in the coming sections, we will be looking specifically into the available literatures, which tends to analyze few of these algorithms for mostly all the mentioned classes, on the CUDA parallel processing model. Eventually we will try to conclude what we gain, in a tabular form so that all the basics of image processing on parallel framework is comprehended easily.

5. Literature Review

In this section we will try to find the answer of Q6 by looking into how CUDA is implemented across different Image Processing Algorithms. In all these subsections, all the details are drawn from the research paper mentioned in the bracket.

A. Contrast Enhancement ([4])

Contrast enhancement algorithms are used to revamp image quality destroyed due to very low contrast and exposure. In this section, we will look into the paper ([4]), where we analyze the “Multi Scale Retinex” contrast improvement algorithm with the help of CUDA architecture.

MSR:

Some other famous contrast enhancement algorithms consist of histogram equalization, wavelet transform, homomorphic filter and contrast stretching. Most of them give results only when the images satisfy certain conditions, and hence are not versatile in every case. But the "Multi Scale Retinex" image improvement algorithm works best in majority cases.

Algorithm:

The retinex theory has an assumption, that color perception is consistent color wise, and does not depend on illumination and intensity in the environment. Based on this theory:

I (x, y) (Image) is described as:

"I(x,y) = E(x,y). R (x, y)" (A)

R(x,y) is a reflectance image and E(x,y) is an illumination image, for numerical simplification the algorithm is converted to a logarithmic domain.:

"log(I(x,y)) =log(E(x,y))+log(R(x,y))" (B)

The basic Single scale retinex is:

"$R_i(x,y)$ = log $I_i(x,y)$ - log[F(x,y) * $I_i(x,y)$]" (C)

$R_i(x,y)$: retinex output in *i*th spectral band , "*" : convolutional operator, F(x,y) is the surround function

MSR is basically weighted sum of SSRs:

$\mathbf{R_{MSRi}} = \sum_{n=1}^{N} wn\, Rcn^{i}$ **(**D)

N=number of scales, R_{cn}^{i} is the ith component of the nth scale.

Implementation:

MSR is naturally parallelizable, because it has already included different operations like difference in logarithmic domain and gaussian filtering etc.

In this paper GPU performs these operations by attaching each pixel a thread, and kernel functions are required to communicate the value processed to the next kernel function. In this way, it gets implemented.

Result:

With basic environmental requirements fulfilled, the experiment was conducted. A sharp speedup was recorded. The actual

data will be provided in the conclusive table of all the algorithms. (Table 3)

B. Connected component Labeling (Šťava & Beneš, 2011)

The Connected component Labeling algorithm puts tag on every region it finds connected in images or in general data. It has applications in computer vision, pattern recognition etc. It is usually solved by a sequential algorithm, but the time requirement is too high to be used for real time applications.

Two pass algorithm:

first pass: Equivalence of all neighboring elements which are connected is stored.

second pass: Assigns label to individual elements based on the analysis of equivalence.

Equivalence is stored in terms of forest data structure with several trees and these trees represents each connected component.

In the pseudo code mentioned in the paper the forest is stored in the form of a 2-D array, where root is held as a 1D array, and is the identifying element. So the equivalence is checked in terms of UNION, which checks and updates elements within its connection with elements in the neighborhood, which are already processed.

FINDROOT (), flatten the tree which makes every element equivalent to the root of the tree. Which ultimately becomes the label.

CUDA Algorithm and Implementation:

There are various complex functions and implementations that is explained in the paper and one can always refer to them, but overall idea of their algorithm is in two steps:

1- To start with, the Connected component Labeling issue is settled for a little subset of information in shared memory.

2- Local solutions are then recursively merged to get the global solution.

It is done by the help of 4 kernels, first kernel solves the Connected component Labeling problem in shared memory for small tiles. One thread block solves each tile.

The merging happens with the help of two kernels, one kernel takes equivalence trees from tiles and merges them, and the second one is used to modify tags at the end before merging. The fourth kernel processes and updates the labels.

Results:

The approximate speedup of the CUDA bases from the tested CPU algorithm is around 10x-15x, the exact data about the comparisons will be comprehended in the last concluding table. (Table 3)

C. Dithering and Half-toning (Zhang et al., 2011)

Dithering refers to randomization of color values, intensity. It is intentionally applying noise to simulate more tone than available, or creating color banding type patterns in images, essentially it fools the eyes in believing the presence of more colors than actual. An example would be black-white band type symbols that appear to be different tones of grey, black and white.

Half-Toning is an effect where a fixed pattern of dots is varied in amplitude and size to create an illusion of continuous tones. It is similar to dithering.

In this paper we see one of the famous Half-Toning algorithms called Error diffusion.

Pinwheel Error Diffusion Algorithm:

It works in two basic steps:

Step 1: Two groups of images are formed in a chess-board fashion, let's say in grey and white groups. Then the algorithm processes parallelly all blocks in the same group, with grey boxes being done first.

Step 2: Each block is processed in a spiral path to scan each pixel. The scan path is inward for white blocks and outwards for grey blocks.

In this way it diffuses errors from grey block to white block, across block

boundaries, this is basically diffusing quantization error to process each pixel by setting up a threshold and then weighing it to an error-weighing matrix.

Implementation:

The paper implements each image block to one CUDA thread, so many images are attached to each block which is generally of size 64, which has enough registers, as well as it has a good number of threads occupied. The paper uses three types of weighing matrices, this can be quite complex, so the paper uses a program to eliminate look-ups.

Results:

The algorithm time was almost as linear for large images, for some images the speedup is as high as 90x, the actual data will be comprehended in the last conclusive table. (Table 3)

D. Feature Detection (Yuancheng Luo & Duraiswami, 2008)

Feature detection consists of many subtasks, like image retrieval, object detection, smoothing, filtering etc.

It takes local decisions at image point and computes abstract information about the image, it results in a subset of image domain.

Canny Edge Detection:

It's an edge detector algorithm, it optimally finds a group of conditions that maximizes chances of true edge detection. It is generally used in pre-processing step, it has a lot of sub steps to detect the true and false edges including smoothing, filtering, Non-maximum suppression and connected component analysis. It is widely used in separating objects from their background.

Algorithm:

Step 1: Use gaussian filter for image smoothening.

Step 2: Convolve the image with a 2-D first derivative operator which detects the

changes in intensity. Gradient magnitude and directions are calculated, and the maxima of these gives the ridge pixels which are set of possible edges.

Step 3: A hysteresis technique is performed on this set to determine the final set of edges.

Implementation:

There were total of 5 operations which when sequentially performed resulted in the edge detection namely:

- Smoothing with separable filters.
- Gradient by 1-D Sobel.
- Non-maximum suppression
- Hysteresis comparison
- Gradient magnitude
- Gradient direction

All of them instead of computed sequentially are computed parallelly with different pixels appointed to different threads and in some cases with an apron, as required by the algorithm.

Results:

In most cases the speedup was around 3x the actual data will be comprehended in the concluding table. (Table 3)

E. Blind deconvolution (Mazanec, Hermanek, & Kamenicky, 2010)

The naïve model for image processing algorithms is usually the **"Single Instruction Multiple Data (SIMD) model"**, where the execution is easy on normal hardware. The paper **implements "blind image deconvolution algorithm"** using GPU with the help of SIMD addons. Then comparison is done to the naive CPU algorithm.

Algorithm and Implementation:

Blind Deconvolution aims to restore the image distorted due to blur effects and movement. If we assume linearity and go with the stochastic approach, the degraded image can be said to be convolution of unknown system h(x).

The original image is u(x).

Noise is n(x).

"$Z_k(x) = (h_k * u)(x) + n_k(x)$"

Where k lies from 1 to K

K = number of channels.

The algorithm divides the task of deconvolution into subtasks and then applies the CUDA framework.

The tasks are:

- construction of image derivative
- gradients enumeration
- energy function enumeration
- Image state is updated according to previous results.

Results:

The result indicates, that even a complex algorithm can be optimized by the use of CUDA and GPUs. The speedup was less than expected and the reason for this was given to the sequential parts of the algorithm which cannot be parallelized. (Table 3).

F. Median Filtering (Perrot, Domas, & Couturier, 2013)

Median filtering has two tasks:

- It minimizes the blurring of edges.
- It reduces the effects of noise.

It is a very simple process; it simply replaces the gray-scale value of each pixel with the median values of its neighbors around a window of n=k*k with center at this pixel.

This paper works on Parallel Register-only Median filtering and the coding is limited to 8- or 16-bit gray level pictures with height and with being multiples of 512, this is done for better concision.

Implementing a fast-median filter:

Filter H (i, j) is defined as a square window of size n=k*k centered at each pixel I (i,j).

For data transfer we will allocate non-pageable memory, which is better in terms of classic allocation and so provides efficiency.

It also uses registers for storing temporary data of kernels. To use it

sparingly, it uses the forgetful selection algorithm.

Algorithm:

A list of pixel values from the n=k*k window is stored in a list of Rn. Then identification and elimination of elements with maximum and minimum value takes place. One value of the original list which was left is taken, and this process is repeated till the values are exhausted. The elements remaining in the list is the global median value.

Rn is the minimum number of registers required for this algorithm and is defined as Rn = [n/2] +1.

Results:

The major asset is the rise in transfers, which is as high as 75% than the naïve one. It matters because the data transfer between CPU and GPU is 13% of the runtime for 8-bit images, but 82% for 16-bit images so overall it makes it faster, the actual data will be comprehended in the last conclusive table. (Table 3)

G. Seam carving (*Jacob Stultz)*

There hasn't been any paper published by jacob stultz but in a project, he proposed this technique of seam carving, which is essentially image resizing in a content aware fashion, or without changing the aspect ratio. The computational complexity of the algorithm makes it perfectly fit for CUDA implementation.

Algorithm:

Step 1: Depending on the direction of resizing (left to right or top to bottom), a seam is identified. A seam is the least contiguous path from top to bottom or left to right. It contains only one pixel per row/column and each pixel must be directly or diagonally adjacent to the next.

Step 2: This one-pixel wide seam is removed.

Step 3: Iteratively the least important seam is found and removed until the image is of desired size.

An energy function is used to determine the least important seam and a dynamic programming algorithm is used to find the minimum energy path.

Implementation:

Energy calculation and Minimum Energy Path calculation are two parallelized tasks in the implementation. For the Energy map calculation, image data is split across processors with individual processors calculating the energy of each pixel. Similarly, the Path is also calculated by spreading out tasks to individual processors but with communication between different blocks. The details can be found out in the paper.

Results:

The large images gained a speed up of approximately double the sequential processing. The actual data will be comprehended in the concluding table. (Table 3)

H. Image Segmentation (Zhuge, Cao, & Miller, 2009)

Segmentation refers to dividing an image into different segments or set of pixels also known as image object. Segmentation aims to convert images into something more meaningful to analyze but they are excessively expensive in terms of computation. In this paper the effort is to introduce a parallelly implemented fuzzy connected segmentation algorithm with the help of CUDA which reduces its time and makes it suitable to be used in real-time and in medical domains, as it is widely used there.

FUZZY-CONNECTEDNESS

PRINCIPLES:

Basic Information needed for algorithm:

Cuboidal volume elements are basically a point defining a unit(voxel) of graphic information in 3D space. They constitute a rectangular array C and each of these units have a scene intensity function f.

So, the volume image also knows as scene is defined by C= (C, f)

“o” belongs to C

and fuzzy affinity is k.

the output is C_{k0} which is k-connectivity

scene or = (C,f_{k0})

CUDA implementation of algorithm:

1-Track-kernel

Track-kernel (CKo, Cm1, Cm2, μκ)

"Begin:

If f_{m1}(c)then

Iterate for each voxel and check $\mu\kappa(c,e)>0$

Do

set $f_{min} = \min\{f_{Ko}(c), \mu_\kappa(c, e)\}$

if $f_{min} > f_{Ko}(e)$ then

set $f_{Ko}(e) = f_{min}$

set $f_{m2}(e) = 1$

End".

2- Algorithm

Begin

ALL voxels of C_{ko} = 0 and o=1

ALL voxels of C_{m1}= 0 and o=1

Iterate while ALL C_{m1} is not 0

Do

ALL voxels of C_{m1}= 0 and o=1

For every voxel in parallel

Call Track-kernel (CKo, Cm1 , Cm2 , μκ) on grid

C_{m2} is copied to C_{m1}

End

Results:

The speedup observed was over 7-14x with this CUDA implementation over CPU. The actual data will be comprehended in the last concluding table. (Table 3)

Table 3

Class of Algorithm	Algorithm	Reference	System and Environment used	Methodology	Image Size	Performance comparison	
						Time(ms)	Speedup

						CPU	CUDA	
Contrast Enhancement	Multi-Scale Retinex	[2]	Xeon E5430 Nvidia Quadro FX3700 CUDA toolkit and SDK 2.0.	CUDA Parallel Programming model	256x256	158.4	4.27	37.19
					512x512	525.12	12.86	40.84
					768x768	1312.31	27.05	48.51
					1024xl024	3724.13	43.47	85.67
					2048x2048	15926.94	157.22	101.30
						Average speedup		
Connected Component Labeling	Two pass Connected component Labeling algorithm.	(Šťava & Beneš, 2011)	CPU: 3 GHz GPU: Nvidia GeForce GTX 480	CUDA Parallel Programming model	512x512	20.3x		
					1024xl024	9.9x		
					2048x2048	11.9x		
					4096x4096	13.6x		
						Speedup vs CPU (8x8 image block)		

Dithering and Half-toning	Pinwheel Error Diffusion Algorithm	(Zhang et al., 2011)	2.4 GHz Intel Core 2 Nvidia GTX 460 CUDA 3.2	CUDA Parallel Programming model	256x256 512x512 1024xl024 2048x2048 4096x4096	5x 17x 25x 33x 33x

						Time(ms)		**Speedup**
						Open CV	CUDA	
Feature Detection	Canny Edge Detection	(Yuancheng Luo & Duraiswami, 2008)	Intel Core2 2.40 GHz NV IDIA GeForce 8800 GTX	CUDA Parallel Programming model	256x256 512x512 1024x1024 2048x2048 3936x3936	3.06 8.28 28.92 105.55 374.36	1.82 3.40 10.92 31.46 96.62	1.681 2.43 2.64 3.35 3.87

						Time(ms)	
						CUDA	**CPU**
	Blind deconvolution with graphics processor	(Mazanec, Hermanek,	Not Mentioned.	CUDA Parallel	1 Mpix 2 Mpix	1.3s 2.0s	2.6s 4.6s

Blind deconvolution	as the SIMD computing resource	& Kamenicky, 2010))		Programming model	4 Mpix 8 Mpix		4.0s 13.8s	9.7s 55.6s
Median Filtering	GPU based median filtering	(“Perrot, Domas, & Couturier, 2013”)	C2070 GPU card Xeon E56202 40GHz CUDA v4.0	CUDA Parallel Programming model	512x512 1024x1024 2048x2048 4096x4096	**Throughput (8-and 16-bit coded image)**		
						T_8	**T_{16}**	
						3.06 8.28 28.92 105.55	1.82 3.40 10.62 31.46	
Image Resizing	Seam carving	(*Jacob Stultz)*	Not Mentioned.	CUDA Parallel Programming model	0.6Mpixel 2.5Mpixel 10Mpixel 40Mpixel	**Time(ms)**		
						Serial	**Parallel**	
						10 70 190 770	10 40 110 480	
Image Segmentation	Fuzzy Image Segmentation	(Zhuge, Cao, & Miller, 2009)	quad core 2.66GHz Intel Xeon NVIDIA GTX280	CUDA Parallel Programming model	Small Dataset Medium Dataset	**Time(s)**		**Speedup**
						GPU	**CPU**	
						0.86 1.84	6.17 13.36	7.2 7.3

			CUDA SDK 2.0		Large Dataset	1.94	27.88	14.4

8. Result Analysis

The Table 3 analyses all the major Image Processing Algorithms Categories. We see the performance of at least one algorithm from each category, both on the GPUs parallel processing architecture supported by CUDA, as well as on the naïve CPU approach. We learned different performance measuring criteria in section 3 so we can clearly spot the performance difference in the table. Keeping aside the irregularity in the environment of all the experiments (as most of them are standard conditions), a marginal increase in performance can be seen in both, at the performance levels, from the speedup and at the data transfer level, from the throughput. In some cases, the speedup was as great as 100x and time required was as less than 1.82 ms. These results are just on standard algorithms of image processing and with the current and past hardware resources available, the trend shows that the performance is bound to increase, as the hardware increases in capability. Even more complex algorithms can be easily optimized in time and data transfer.

9. Conclusion and Future scope

As it can be noticed that there are major 10 categories of Image Processing Algorithm as mentioned in the section 4. We discussed only 8 of them, and the reason for that is the scarcity of work in these two major categories namely, Difference-Map Algorithm and Richardson–Lucy algorithm, on CUDA model. So, there's scope for work in these two major categories. There's also scope for more work and more effective ways to utilize

the power of CUDA in other categories as well.

With the world deeply immersed in the real-time era, when everything happens in a blink, Image processing stands at the core. It has uses across all domains, medical, military e-commerce, security and many more. As the result section points out marginal increase in performance with the CUDA parallel programming model, we can safely say that a new trend is on the bank.

It's the need of hour to speed up the processing time and CUDA plays an important role in this, as it is the most versatile and simple to use model for parallel processing. Its scalability power is huge and it's just an extension of our previous programming models. In this paper we tried to compile all the major subjects related to image processing and CUDA framework, our goal was to kick start this journey so that more work can be done in this direction.

10. Reference

[1] Kirk, D. (2007). NVIDIA cuda software and gpu parallel computing architecture. *Proceedings Of The 6Th International Symposium On Memory Management - ISMM '07*. doi: 10.1145/1296907.1296909

[2] Saxena, S., Sharma, S., & Sharma, N. (2016). Parallel Image Processing Techniques, Benefits and Limitations. *Research Journal Of Applied Sciences, Engineering And Technology*, *12*(2), 223-238. doi: 10.19026/rjaset.12.2324

[3] List of algorithms. (2020). Retrieved 3 May 2020, from https://en.wikipedia.org/wiki/List_of_algorithms#Image_processing

[4] Wang Zheng-ning, Liu Changzhong, Lu Yu, Wu Min, & Zhang Ping. (2010). The implementation of Multi-Scale Retinex image enhancement algorithm based on GPU via CUDA. *2010 International Symposium On Intelligent Signal Processing And Communication Systems*. doi: 10.1109/ispacs.2010.5704755

[5] Šťava, O., & Beneš, B. (2011). Connected Component Labeling in CUDA. *GPU Computing Gems Emerald Edition*, 569-581. doi: 10.1016/b978-0-12-384988-5.00035-8

[6] Zhang, Y., Recker, J., Ulichney, R., Beretta, G., Tastl, I., Lin, I., & Owens, J. (2011). A parallel error diffusion implementation on a GPU. *Parallel Processing For Imaging Applications*. doi: 10.1117/12.872616

[7] Yuancheng Luo, & Duraiswami, R. (2008). Canny edge detection on NVIDIA CUDA. *2008 IEEE Computer Society Conference On Computer Vision And Pattern Recognition Workshops*. doi: 10.1109/cvprw.2008.4563088

[8] Yazdanpanah, A., Mandava, A., Regentova, E., Muthukumar, V., & Bebis, G. (2014). A CUDA Based Implementation of Locally-and Feature-Adaptive Diffusion Based Image Denoising Algorithm. *2014 11Th International Conference On Information Technology: New Generations*. doi: 10.1109/itng.2014.113

[9] Yonglong, Z., Kuizhi, M., Xiang, J., & Peixiang, D. (2013). Parallelization and Optimization of SIFT on GPU Using CUDA. *2013 IEEE 10Th International Conference On High Performance Computing And Communications & 2013 IEEE International Conference On Embedded And Ubiquitous Computing*. doi: 10.1109/hpcc.and.euc.2013.192

[10] Accelerating Convolution Operations by GPU (CUDA), Part 1: Fundamentals with Example Code Using Only Global Memory - Qiita. (2020). Retrieved 3 May 2020, from https://qiita.com/naoyuki_ichimura/items/8c80e67a10d99c2fb53c

[11]Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., & Morton, S. et al. (2008). Parallel Computing Experiences with CUDA. *IEEE Micro*, *28*(4), 13-27. doi: 10.1109/mm.2008.57

[12]González, R. (2001). *Digital image processing*. Reading, Mass.: Addison-Wesley.

[13] Prajapati, H., & Vij, S. (2011). Analytical study of parallel and distributed image processing. *2011 International Conference On Image Information Processing*. doi: 10.1109/iciip.2011.6108870

[14] Mazanec, T., Hermanek, A., & Kamenicky, J. (2010). Blind image deconvolution algorithm on NVIDIA CUDA platform. *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. doi: 10.1109/ddecs.2010.5491803

[15] Perrot, G., Domas, S., & Couturier, R. (2013). Fine-tuned High-speed Implementation of a GPU-based Median Filter. *Journal of Signal Processing Systems, 75*(3), 185–190. doi: 10.1007/s11265-013-0799-2

[16] Zhuge, Y., Cao, Y., & Miller, R. (2009). GPU accelerated fuzzy connected image segmentation by using CUDA. *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. doi: 10.1109/iembs.2009.5333158

[17] Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: A quantitative approach*. Elsevier.