PARALLEL OPTICAL FLOW DETECTION USING CUDA

Thesis submitted
By

Khalid Hossen

Student ID: 10101019

Hasan Mahmud

Student ID: 10101026

For the B. Sc in Computer Science and Engineering Degree, presented on 30th April 2014, in

BRAC University, Dhaka, Bangladesh

Supervisor – Md. Zahangir Alom

Co-supervisor – Risul Karim

Declaration

This is to certify that the thesis entitled "Parallel Optical Flow Detection Using CUDA", which is submitted by Khalid Hossen (ID: 10101019) and Hasan Mahmud (ID: 10101026) in partial fulfillment of the requirement for the award of degree of Bachelor of Science in Computer Science & Engineering to the Department of Computer Science & Engineering, BRAC University, 66 Mohakhali C/A, Dhaka, 1212, comprises only their original work and due acknowledgement has been made in the text to all other material used. The result of the thesis has not been submitted to any other University or Institute for the award of any degree or diploma.

Approved By

Md. Zahangir Alom(Supervisor)____________________________

Professor Md. Zahidur Rahman(Chairperson)__________________

Date of Approval

____

Acknowledgement

We would like to express our special appreciation and thanks to our Supervisor Md. ZahangirAlom and co-supervisor Risul Karim who have been a tremendous mentor for us. We would like to thank you for encouraging our research. We also want to thank Md. Shamsul Kaonine for allocating university resources for our use for this research.

Finally, we thank our family, friends, and all the teachers for their motivation, inspiration and support.

ABSTRACT

The intention of this thesis paper is to deploy a parallel implementation of the optical flow detection algorithm known as the Lucas-Kanade algorithm. As an important algorithm in the field of computer vision, it is believed that it holds much promise and shows much potential for benefiting from techniques used to enhance performance through parallel programming which can be executed with the use of CUDA.

Though more techniques of parallel programming exist that can be used to fasten the process, Lucas-Kanade has never been implemented in parallel programming before. The result of the research has shown both serial and parallel implementation of optical flow detection using deferent processing units (CPUs and GPUs). The parallel implementation have lessened 2 to 13 seconds of processing time (depending on the hardware configuration) for the same database compare to serial implementation.

TABLE OF CONTENTS

LIST OF FIGURES

LISTS OF TABLES

ABBREVIATIONS

CUDA - Compute Unified Device Architecture

GPU – Graphics Processing Unit

CPU – Central Processing Unit

SAD- Sum of Absolute Difference

SSD- Sum of Square Difference

Chapter 1
Introduction

1.1 Motivation

The main motivation was to learn parallel programming with one of the best topic. Optical flow detection is a very important subject to modern science for finding solution to many problems like traffic control, airport security, medical research, controlling robots and many others. So we had chosen optical flow detection and solved it with parallel programming algorithm using Matlab. Parallel programming is also useful for modern hardware implementation of software. The reason behind it is the high efficiency of modern GPUs which are designed to work with different problems simultaneously.

1.2 Methodology

The purpose of this paper is to propose a design to detect important flow changes between images solving with a parallel programming method which gives most of the times a better performance.

We begin by detecting optical flow between two images using a simple and not parallel algorithm. At first we run the codes in CPU only comparing two images. Then we run the codes on a video which is not more than a number of images. After that we run the algorithm parallel with the help of MatLab and its CUDA library on GPU. The parallel programming model helps a lot with boosting the performance with an increase of CUDA core of the GPU.

1.3 Tools Used

This simulation software utilizes Nvidia Compute Unified Device Architecture (CUDA) GPU Computing Platform in order to achieve the communication with the graphics device. The CUDA platform is a proprietary platform of Nvidia Corporations and is only supported by Nvidia Graphics Processors. The other alternative to this platform is the OpenCL (Open Computing Library) which is open-source and free to use and is supported by most every graphics processors with programmable pipelines. However, the CUDA platform provides some more benefits such as better support and more efficient communication between supported devices compared to OpenCL as well as better documentation and more efficient libraries.

The algorithm is programmed in MatLab and implemented on CUDA to improve performance with running the algorithm in parallel.

This method improves upon the time factor of the simulation significantly with respect to the CPU only implementation and allows for distributed computing using multiple CPU as well as multiple GPU giving this a very large scale implementation possibility.

We also needed to install visual studio as a requirement of installing CUDA. We used Windows 8 as our operating system.

Chapter 2
Optical Flow Detection and Implementation

2.1 Problem definition

Optic flow is defined as the change of structured light in the image, e.g. on the retina or the camera’s sensor, due to a relative motion between the eyeball or camera and the scene. Further definitions from the literature highlight different properties of optic flow.

In a bio-inspired context retinal flow is the change of structured patterns of light on the retina that lead to an impression of movement of the visual imagery projected onto the retina. Figure 2.1a depicts the production of retinal optic flow for the displacements of two exemplary visual features. In technical terms and in the context of computer vision, changes of the environment in the image are represented by a series of image frames. Figure 2.1b shows three frames of an image sequence, which can be obtained by a spatial and temporal sampling of the incoming light. Optic flow captures the change in these images through a vector field. Research emphasizes on the accurate, pixel-wise estimation of optic flow, which is a computationally demanding task. Nowadays, optic flow can be estimated in close to real-time for a reasonable image resolution.

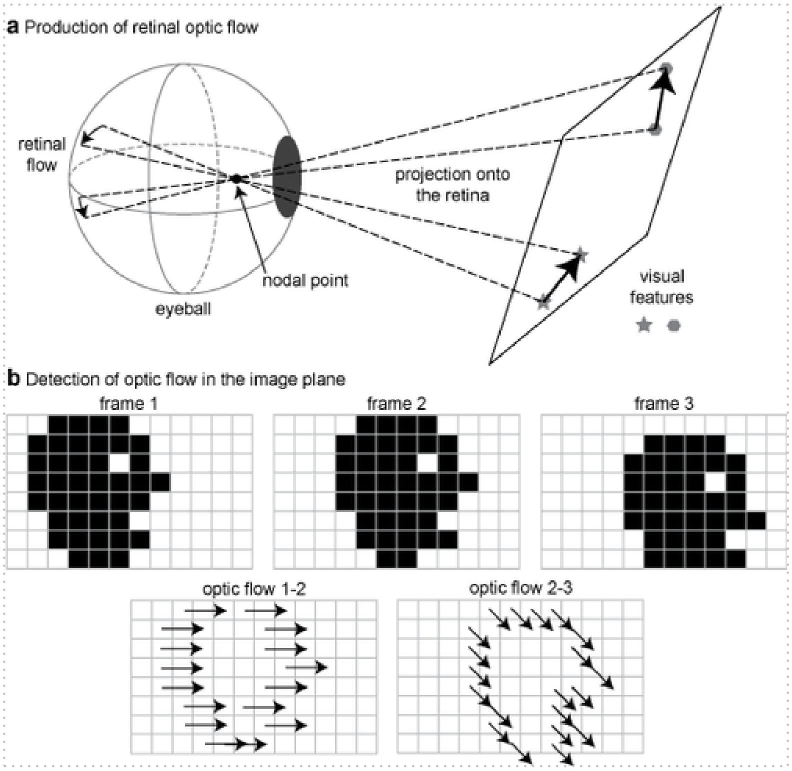
a Production of retinal optic flow
retinal flow
projection onto the retina
nodal point
eyeball
visual features
b Detection of optic flow in the image plane
frame 1
frame 2
frame 3
optic flow 1-2
optic flow 2-3

Figure 2.1 a and b

This figure 2.1 shows the production and detection of optic flow. a) Optic flow is generated on the retina by changes in the patterns of light. The example shows the shift of two visual features (star and hexagon) on a plane and their angular displacements on the surface of the eyeball. b) If the structured light is sampled spatially and temporally this results in an image sequence. The example shows three frames, which show the movement of the silhouette of a head. The optic flow is depicted as the correspondence of contour pixels between frame 1 and 2 as well as frame 2 and 3. For methods estimating flow, the challenge is to find the point correspondence for each pixel in the image, not only the contour pixels.

2.2 Background

2.2.1 Principle Component analysis

2.2.1.1 Taylor Series

In mathematics, a Taylor series is a representation of a function as an infinite sum of terms that are calculated from the values of the function's derivatives at a single point.

The concept of a Taylor series was discovered by the Scottish mathematician James Gregory and formally introduced by the English mathematician Brook Taylor in 1715. If the Taylor series is centered at zero, then that series is also called a Maclaurin series, named after the Scottish mathematician Colin Maclaurin, who made extensive use of this special case of Taylor series in the 18th century.

It is common practice to approximate a function by using a finite number of terms of its Taylor series. Taylor's theorem gives quantitative estimates on the error in this approximation. Any finite number of initial terms of the Taylor series of a function is called a Taylor polynomial. The Taylor series of a function is the limit of that function's Taylor polynomials, provided that the limit exists. A function may not be equal to its Taylor series, even if its Taylor series converges at every point. A function that is equal to its Taylor series in an open interval (or a disc in the complex plane) is known as an analytic function.

The Taylor series of a real or complex-valued function $f(x)$ that is infinitely differentiable at a real or complex number a is the power series,

$$f(a)+\frac{f'(a)}{1!}(x-a)+\frac{f''(a)}{2!}(x-a)^2+\frac{f'''(a)x^3}{3!}(x-a)^3$$

which can be written in the more compact sigma notation as,

$$\sum_{n=1}^{\infty}\frac{f^n(a)}{n!}(x-a)^n$$

where $n!$ denotes the factorial of n and $f^{(n)}(a)$ denotes the nth derivative of f evaluated at the point a. The derivative of order zero f is defined to be f itself and $(x-a)^0$ and 0! are both defined to be 1. When $a = 0$, the series is also called a Maclaurin series.

2.2.1.2 Transpose of Matrix

In linear algebra, the transpose of a matrix A is another matrix A^T (also written A′, A^{tr},${}^{t}A$ or A^t) created by any one of the following equivalent actions:

- reflect A over its main diagonal (which runs from top-left to bottom-right) to obtain A^T
- write the rows of A as the columns of A^T
- write the columns of A as the rows of A^T

Formally, the *i* th row, *j* th column element of A^T is the *j* th row, *i* th column element of A:

$$[A^T]_{ij} = [A]_{ji}$$

If A is an $m \times n$ matrix then A^T is an $n \times m$ matrix.

2.2.1.3 Partial Derivatives

In mathematics, a partial derivative of a function of several variables is its derivative with respect to one of those variables, with the others held constant (as opposed to the total derivative, in which all variables are allowed to vary). Partial derivatives are used in vector calculus and differential geometry.

The partial derivative of a function *f* with respect to the variable *x* is variously denoted by

$$f'_x, f_x, \partial_x f, \frac{\partial}{\partial_x} f, or\ \frac{\partial f}{\partial_x}$$

The partial-derivative symbol is ∂. One of the first known uses of the symbol in mathematics is by Marquis de Condorcet from 1770, who used it for partial differences. The modern partial derivative notation is by Adrien-Marie Legendre (1786), though he later abandoned it; Carl Gustav Jacob Jacobi re-introduced the symbol in 1841.

If f is a function of more than one variable. For instance,

$$z = f(x, y) = x^2 + xy + y$$

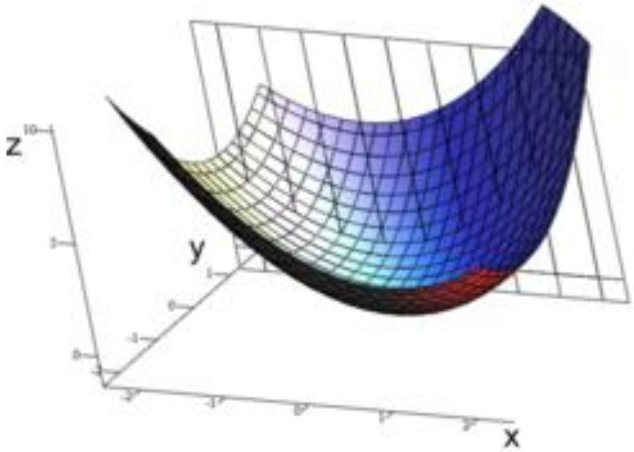

Figure 2.2.1.3.a Graph representation of z = x2 + xy + y2

A graph of $z = x^2 + xy + y^2$. For the partial derivative at (1, 1, 3) that leaves y constant, the corresponding tangent line is parallel to the xz-plane.

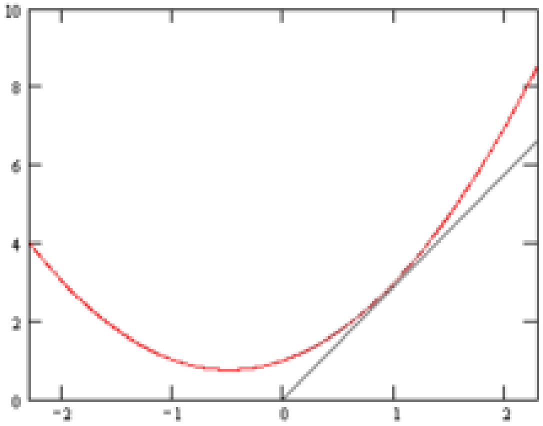

Figure 2.2.1.3.b Graph in the xz-plane at y= 1

A slice of the graph above showing the function in the *xz*-plane at *y*= 1

The graph of this function defines a surface in Euclidean space. To every point on this surface, there are an infinite number of tangent lines. Partial differentiation is the act of choosing one of these lines and finding its slope. Usually, the lines of most interest are those that are parallel to the *xz*-plane, and those that are parallel to the *yz*-plane (which result from holding either y or x constant, respectively.)

To find the slope of the line tangent to the function at P(1, 1, 3) that is parallel to the *xz*-plane, the *y* variable is treated as constant. The graph and this plane are shown on the right. On the graph below it, we see the way the function looks on the plane *y* = 1. By finding the derivative of the equation while assuming that *y* is a constant, the slope of *ƒ* at the point (*x*, *y*, *z*) is found to be:

$$\frac{\delta z}{\delta x} = 2x + y$$

So at (1, 1, 3), by substitution, the slope is 3. Therefore

$$\frac{\delta z}{\delta x} = 3$$

at the point (1, 1, 3). That is, the partial derivative of *z* with respect to *x* at (1, 1, 3) is 3.

2.2.2 Optical Flow Detection

2.2.2.1 Introduction

Optical flow or optic flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer (an eye or a camera) and the scene. The concept of optical flow was introduced by the American psychologist James J. Gibson in the 1940s to describe the visual stimulus provided to animals moving through the world. James Gibson stressed the importance of optic flow for affordance perception, the ability to discern possibilities for action within the environment. Followers of Gibson and his ecological approach to psychology have further demonstrated the role of the optical flow stimulus for the perception of movement by the observer in the world perception of the shape, distance and movement of objects in the world; and the control of locomotion. Recently the term optical flow has been co-opted by robot cists to incorporate related techniques from image processing and control of navigation, such as motion detection, object segmentation, time-to-contact information, and focus of expansion calculations, luminance, and motion compensated encoding, and stereo disparity measurement.

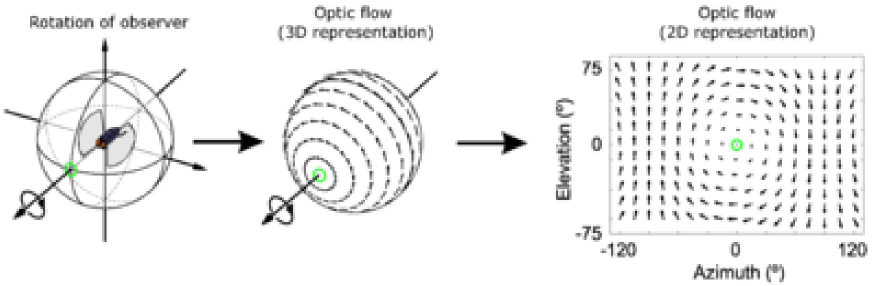

Figure: 2.2.2.1 Optical flow representations

2.2.2.2 Block Matching Method

This is the simplest way of optical flow detection from images. Considering a Region of Interest in first image (a block), the purpose is to find the displacement of this ROI in the next one. To this end, we compare the correlation scores between the original block and a family of candidates into a search area ΩROI, in the second frame. Among the correlation criteria of the most often used, one can find some well-known cost functions to minimize, as the sum of absolute differences (SAD) or the sum of squared differences (SSD),

$$\mathbf{SSD} = \sum_{\Omega \boldsymbol{ROI}} (\mathbf{I}(\mathbf{x}, \mathbf{y}, \mathbf{t}) - \mathbf{I}(\mathbf{x}+\mathbf{u}, \mathbf{y}+\mathbf{v}, \mathbf{t}\ +\mathbf{1})^{\mathbf{2}}).$$
$$\mathbf{SAD} = \sum_{\Omega \boldsymbol{ROI}} |\mathbf{I}(\mathbf{x}, \mathbf{y}, \mathbf{t}) - \mathbf{I}(\mathbf{x}+\mathbf{u}, \mathbf{y}+\mathbf{v}, \mathbf{t}\ +\mathbf{1})|.$$

To avoid exhaustive search in W and speed up the process, a lot of exploration algorithms have been developed. However, the main problem of such a method remains its pixel accuracy. Working on over-sampled images is a way to solve this this issue, but also increases the amount of computation.

2.2.2.3 Lucas-Kanade Method

The Lucas–Kanade method assumes that the displacement of the image contents between two nearby instants (frames) is small and approximately constant within a neighborhood of the point *p* under consideration. Thus the optical flow equation can be assumed to hold for all pixels within a window centered at *p* namely, the local image flow (velocity) vector ($V_{x,}, V_y$) must satisfy

$$I_x(q_1)V_x + I_y(q_1)V_y = - I_t(q_1)$$

$$I_x(q_2)V_x + I_y(q_2)V_y = - I_t(q_2)$$

$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots$$

$$I_x(q_n)V_x + I_y(q_n)V_y = - I_t(q_n)$$

Where $q_1, q_{2,\dots\dots} q_n$ are the pixels inside the window, and $I_y(q_i), I_y(q_i), I_t(q_i)$ are the partial derivatives of the image *I* with respect to position *x*, *y* and time *t*, evaluated at the point q_i and at the current time.

These equations can be written in matrix form, $Av = b$, where,

$$A = \begin{bmatrix} Ix(q1) & Iy(q1) \\ Ix(q2) & Iy(q2) \\ \dots\dots. & \dots\dots\dots. \\ Ix(qn) & Iy(qn) \end{bmatrix}, v = \begin{bmatrix} Vx \\ Vy \end{bmatrix} \text{ and } b = \begin{bmatrix} -It(q1) \\ -It(q2) \\ -It(qn) \end{bmatrix}$$

This system has more equations than unknowns and thus it is usually over-determined. The Lucas–Kanade method obtains a compromise solution by the least squares principle. Namely, it solves the 2×2 system

$$A^T Av = A^T b \text{ or}$$

$$V = (A^T A)^{-1} A^T b$$

Where, A^T is the transpose of matrix A . That is, it computes

$$\begin{bmatrix} Vx \\ Vy \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i)\, I_y(q_i) \\ \sum_i I_y(q_i)\, I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i)\, I_t(q_i) \\ -\sum_i I_y(q_i)\, I_t(q_i) \end{bmatrix}$$

with the sums running from i=1 to n.

The matrix $A^T A$ is often called the structure tensor of the image at the point p.

The plain least squares solution above gives the same importance to all n pixels q_i in the window. In practice it is usually better to give more weight to the pixels that are closer to the central pixel p. For that, one uses the weighted version of the least squares equation,

$$A^T WAv = A^T Wb \text{ or}$$

$$V = (A^T WA)^{-1} A^b Wb$$

Where, W is an n×n diagonal matrix containing the weights $W_{ii}=w_i$ to be assigned to the equation of pixel q_i. That is, it computes

$$\begin{bmatrix} Vx \\ Vy \end{bmatrix} = \begin{bmatrix} \sum_i w_i I_x(q_i)^2 & \sum_i w_i I_x(q_i)\, I_y(q_i) \\ \sum_i w_i I_y(q_i)\, I_x(q_i) & \sum_i w_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i w_i I_x(q_i)\, I_t(q_i) \\ -\sum_i w_i I_y(q_i)\, I_t(q_i) \end{bmatrix}$$

The weight w_i is usually set to a Gaussian function of the distance between $q_i$and p.

2.2.3 Parallel Programming

2.2.3.1 Introduction

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multicore and multiprocessor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time after that instruction is finished, the next is executed.

Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

2.2.3.1 Amdahl's Law

Optimally, the speedup from parallelization would be linear doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speed-up. Most of them have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s. It states that a small portion of the program which cannot be parallelized will limit the overall speedup available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. If α is the fraction of running time a program spends on non-parallelizable parts, then

$$\lim_{p\to\infty} \frac{1}{\frac{1-\alpha}{P}+\alpha} = \frac{1}{\alpha}$$

is the maximum speed-up with parallelization of the program. If the sequential portion of a program accounts for 10% of the runtime$(\alpha = 0.1)$, we can get no more than a 10x speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned.

2.2.3.2 Gustafson's Law

Gustafson's law is another law in computing, closely related to Amdahl's law. It states that the speedup with P processors is

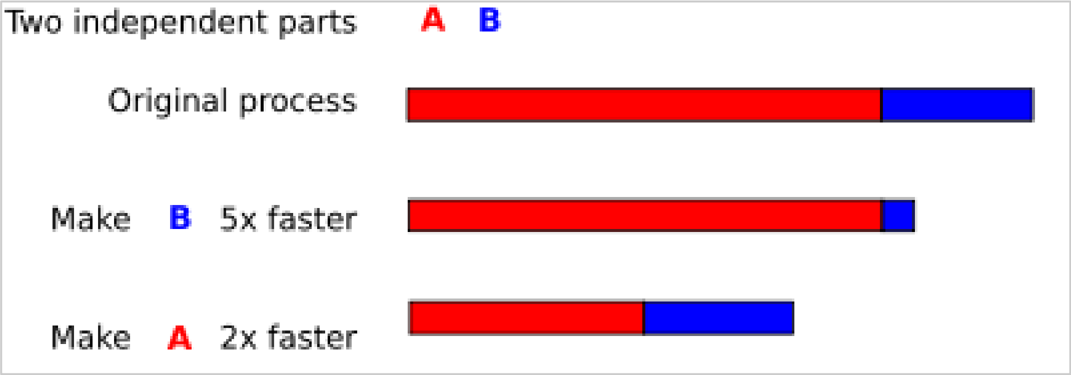

Figure: 2.2.3.2 Comparison of two process using Gustafson’s Law

Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. With effort, a programmer may be able to make this part five times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A twice as fast. This will make the computation much faster than by optimizing part B, even though B got a greater speed-up (5x versus 2x).

$$S(P) = P - \alpha(P-1) = \alpha + P(1-\alpha)$$

Both Amdahl's law and Gustafson's law assume that the running time of the sequential portion of the program is independent of the number of processors. Amdahl's law assumes that the entire problem is of fixed size so that the total amount of work to be done in parallel is also independent of the number of processors, whereas Gustafson's law assumes that the total amount of work to be done in parallel varies linearly with the number of processors.

2.2.3.3 Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, and whether or not those instructions were using a single set or multiple sets of data.

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Table: 2.2.3.3 Comparison between single and multiple instruction

The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set. This is commonly done in signal processing applications. Multiple-instruction-single-data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs.

According to David A. Patterson and John L. Hennessy, "Some machines are hybrids of these categories, of course, but this classic model has survived because it is simple, easy to understand, and gives a good first approximation. It is also—perhaps because of its understandability—the most widely used scheme.

2.2.3.4 CUDA Parallel Programming Language

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).

With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA. Here are a few examples,

Identify hidden plaque in arteries: Heart attacks are the leading cause of death worldwide. Harvard Engineering, Harvard Medical School and Brigham & Women's Hospital have teamed up to use GPUs to simulate blood flow and identify hidden arterial plaque without invasive imaging techniques or exploratory surgery.

Analyze air traffic flow: The National Airspace System manages the nationwide coordination of air traffic flow. Computer models help identify new ways to alleviate congestion and keep airplane traffic moving efficiently. Using the computational power of GPUs, a team at NASA obtained a large performance gain, reducing analysis time from ten minutes to three seconds.

Visualize molecules: A molecular simulation called NAMD (Nano scale molecular dynamics) gets a large performance boost with GPUs. The speed-up is a result of the parallel architecture of GPUs, which enables NAMD developers to port compute-intensive portions of the application to the GPU using the CUDA Toolkit.

2.3 Real Time Flow Detection

2.3.1 Proposed Algorithm

We are using Lucas-Kanade Algorithm to find the optical flow between images for our thesis as Lucas-Kanade is very efficient and can be implemented in parallel programming model very easily.

2.3.2 Flow Colors Computation

The commonest form of color wheel shows the three subtractive primaries, red, yellow, and blue, with equal spacing.

The Munsell system of colors, which is one of the best color systems, uses a circle with ten hues; the five colors red, yellow, green, blue, and purple are equally spaced. Red, yellow, green, and blue are identified as the "psychological primaries". Color television and our computer monitor uses red, green and blue as additive primaries and color printing unlike color painting, terms the subtractive primaries magenta, yellow, and cyan.

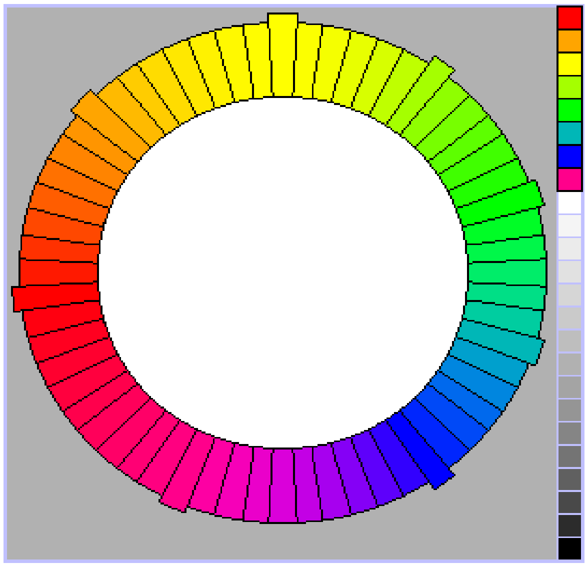

Figure: 2.3.2 color hue representation

Red is closest to yellow, blue is further away from red, since those two colors stand at opposite ends of the spectrum and blue is farthest from yellow, since green is perceived as almost a primary and the distance between blue and green equals the distance between green and yellow and both are less than the distance between red and yellow. A nice color circle meeting those conditions with 60 divisions can be produced by using the sequence of numbers 4-5-6.

Red to orange and orange to yellow are each 8 hues apart. Blue to purple and purple to red are each 10 hues apart. Yellow to green and green to blue are each 12 hues apart. Here's the color circle on that basis.

2.3.3 Flow Detection Steps

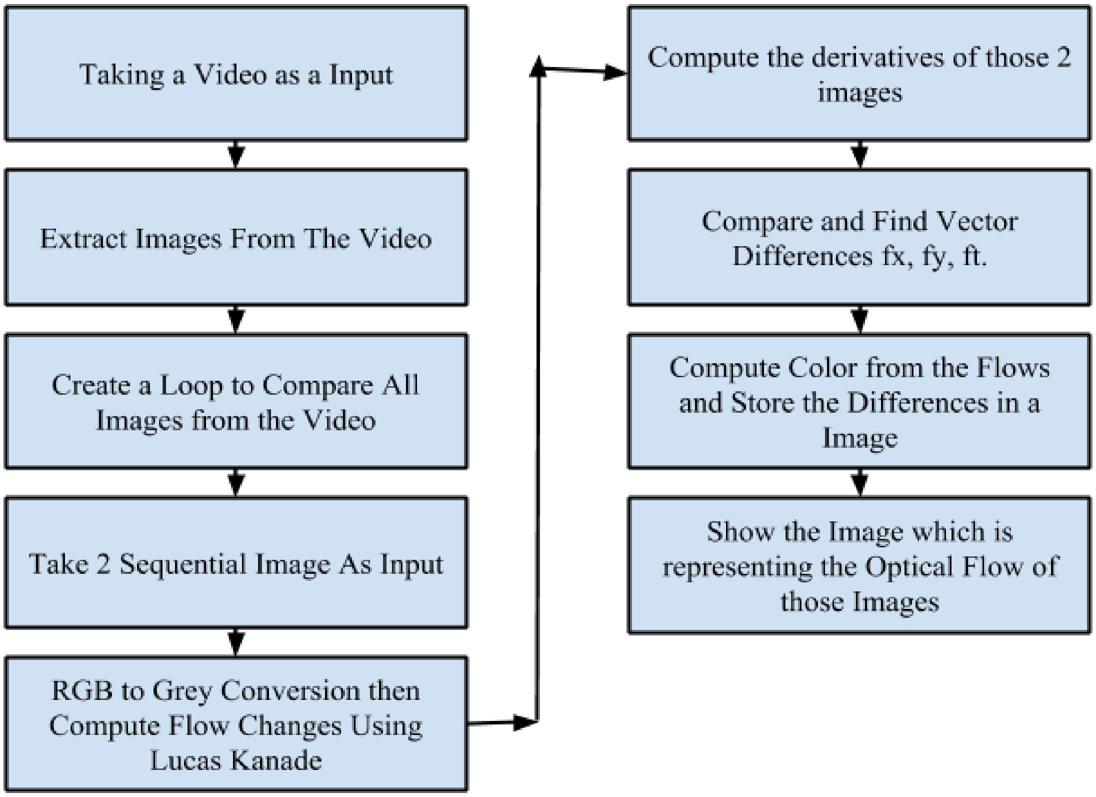
Taking a Video as a Input
Extract Images From The Video
Create a Loop to Compare All Images from the Video
Take 2 Sequential Image As Input
RGB to Grey Conversion then Compute Flow Changes Using Lucas Kanade
Compute the derivatives of those 2 images
Compare and Find Vector Differences fx, fy, ft.
Compute Color from the Flows and Store the Differences in a Image
Show the Image which is representing the Optical Flow of those Images

Figure: 2.3.3 flow detection steps.

Chapter 3
Results and Discussion

3.1 Database

We have taken a video of resolution 160*120(width * height), with 15 frame rate(fps) named “viptraffic” from Matlab example. The length of the video is 8 second and total extracted images are 120.

3.2 Runtime Environment

We have used 2 computers of different specifications for optical flow detection. The specifications of these computers are discussed below.

No. 1 PC Specification

- Intel Core i7 4770 3.40GHz.
- Number of cores in CPU: 8 cores.
- Ram 16 GB.
- GPU: NVidia GTX 660
- Multiprocessor Count for GPU: 6.
- Number of CUDA core in GPU: 960.
- GPU memory size: 1GB.
- GPU memory Interface width: 192-bit.
-

No. 2 PC Specification

- Intel Core i7 4770 3.40GHz.
- Number of cores in CPU: 8 cores.
- Ram 16 GB.
- GPU: NVidia GTX 780
- Multiprocessor Count for GPU: 12.
- Number of CUDA core in GPU: 2304.
- GPU memory size: 3GB.
- GPU memory Interface width: 384-bit.

3.3 Experimental Results

Experiment with No1 PC

Core Used	Total Time Taken(in Seconds)	Time Taken per Frame(in Seconds)
Serial	338.591614	2.8215
1	339.614317	2.8301
2	337.232934	2.8102
3	336.202633	2.8016
4	384.223449	3.2019
5	437.585529	3.6465
6	459.758039	3.8313

Table 3.3.a

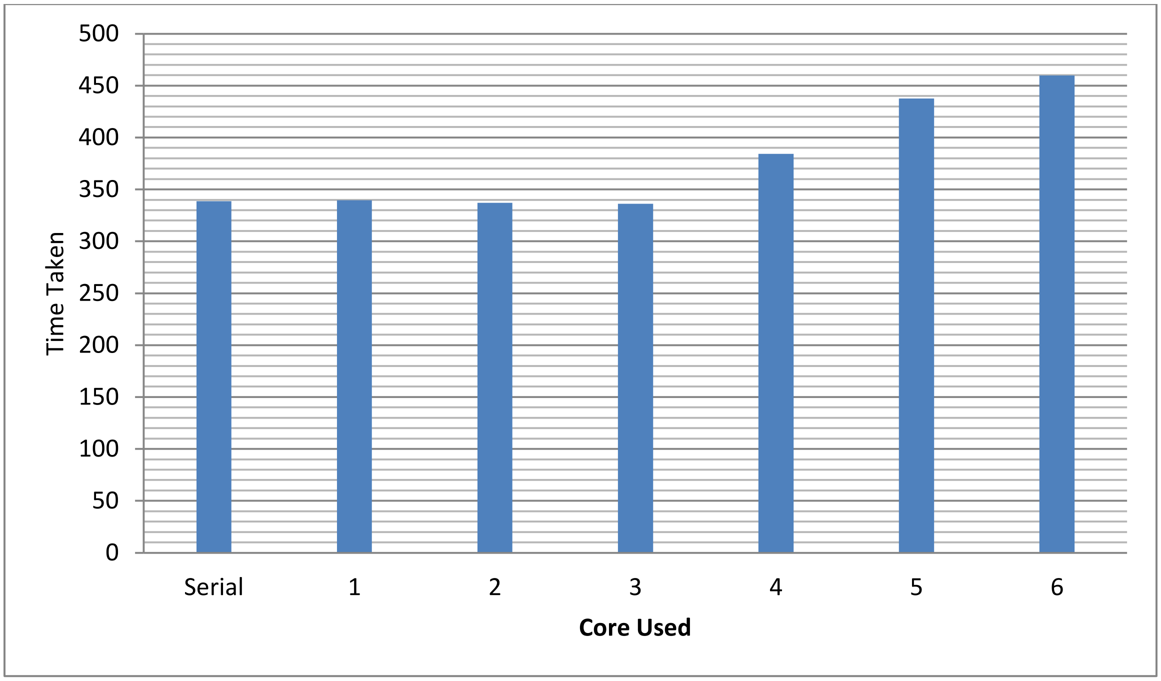

Figure: 3.3.a Time taken for used cores (GTX660)

Experiment with No2 PC

Core Used	Total Time Taken(in Seconds)	Time Taken per Frame(in Seconds)
Serial	332.100913	2.7650
1	335.343075	2.7945
2	327.005117	2.7250
3	319.783982	2.6648
4	364.374475	3.0364
5	427.749014	3.5645
6	444.144528	3.7012
7	515.940868	4.2995
8	596.794590	4.9732
9	673.641076	5.6136
10	754.603825	6.2883
11	832.345944	6.9362
12	941.466541	7.8455

Table 3.3.b

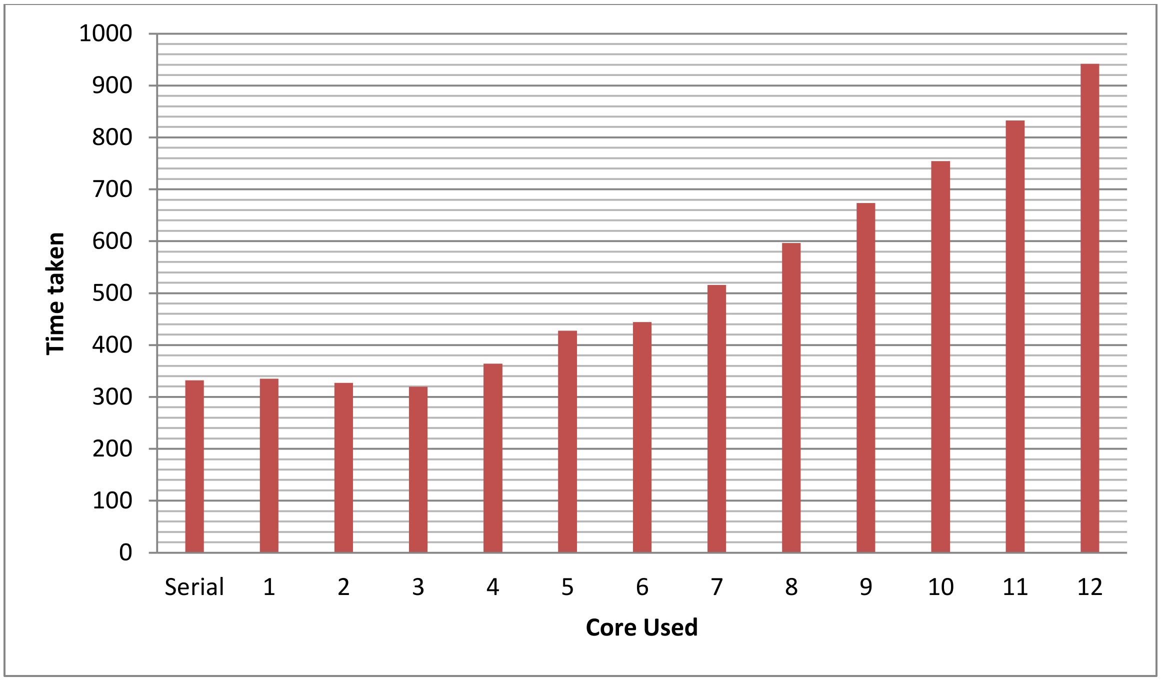

Figure: 3.3.b Time taken for used cores (GTX780)

Example of flow detection:

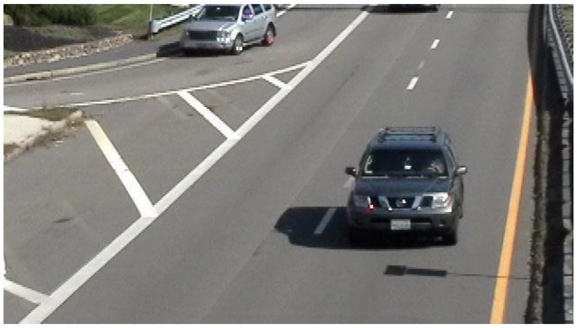

Image 1

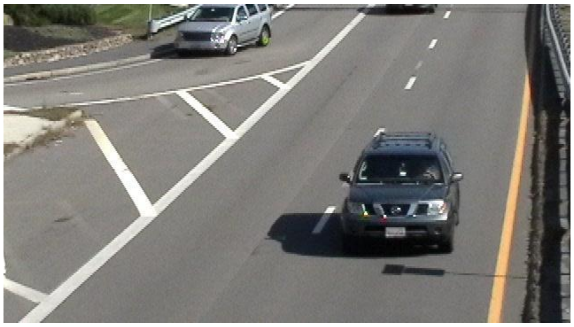

Image 2

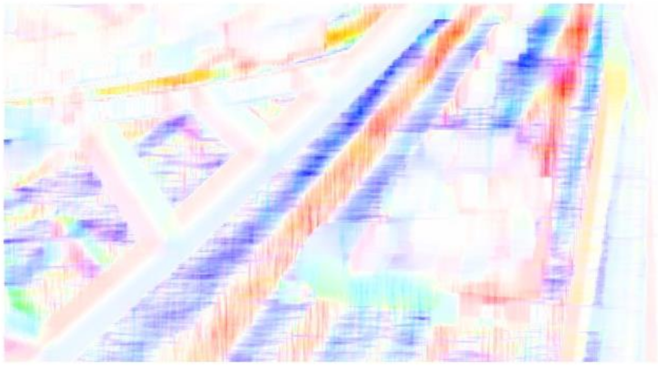

Figure 3.3.c: Flow difference between image 1 and image 2

3.4 Discussions

In No1 PC the GTX 660 GPU has multiprocessor count of 6. So, at best 6 cores can be used. Now when the code was serially implemented the time taken for the whole video which had 120 frames needed 338 seconds. When the code was implemented parallel in using 1 core the time to find flow was increased by one second. Though both had ran in 1 core but to make the code parallel the needed call of functions caused the extra second. When more cores are utilized the needed time reduced to 337 for 2 cores and 336 for 3 cores.

In No2 PC the GTX 780 GPU has multiprocessor count of 6. So, at best 12 cores can be used. Now when the code was serially implemented the time taken was 332 seconds. When the code was implemented parallel in using 1 core the time to find flow was increased by three seconds. When more cores are utilized the needed time reduced to 327 for 2 cores and 319 for 3 cores. So no 2 PC is detected flow much faster because it has more CUDA cores. So for this experimental video GTX 780 was 17 seconds faster for finding the optical flow.

Here, the parallel implementation seems to lessen a very little amount of time from serial implementation, which is not really that small because here we used a very small video (of resolution and length) but if we use big videos with higher resolution and length then the amount of time we have lessened for computation will be bigger.

Now here when we use more cores than 3 the needed time to detect the flow is increasing with gradual increase of cores. It is known that to implement parallel programs every core needs to communicate with other cores which need a significant amount of time. Excess use of processors can create overhead between the cores such as partitioning process, information passing, and coordination of processes. With the increase of cores the chance of overhead occurring increases which here resulted bigger time consumption of processing with more than 3 cores.

Considering that the optical flow requires the sequential input of images from one frame to the next frame in order to calculate the flow detection of objects, it can be said that there are very few prospects of being implemented in parallel. This is because without consecutive inputs of frames of the video calculating the direction of the objects is fairly hard, since the derivatives must be calculated in conjunction with each other. Therefore it is mandatory to have the frames extracted and processed in order to each other. This makes parallel inputs of frames unfeasible. As a whole considering the most important and computation intensive steps themselves are inherently sequential, it can be said that much of the program is unable to utilize the benefits of parallel programming.

Chapter 4
Conclusion and Future Work

4.1 Conclusion

The aim of the research was to implement the Lucas-Kanade in parallel programming model and decrease the required time of detecting optical flow. The serial implementation of Lucas-Kanade algorithm is really efficient but implementing it with parallel programming also gives it a small edge as it has been done. Our parallel implementation has lessened 2 to 13 seconds of processing time (depending on the hardware configuration) for the same database compare to serial implementation. Though more techniques of parallel programming exist that can be used to fasten the process, Lucas-Kanade has never been implemented in parallel programming before.

4.2 Future Works

Today optical flow detection is a very important topic as it helps artificial intelligence with finding path, detecting objects. The medical use of it is also very hopeful as it can help doctors with surgery. It also can be used to navigate vehicles movements and more.

The plan for future is real time implementation like path detection for robots. This research can be implemented using better parallel techniques like using pipelining with better hardware. OpenCV is one of the most customizable languages for parallel implementation in which we are planning to implement this research in the future.

References

[1]. Evaluating Optical-Flow Algorithms on a Parallel Machine, M. Fleury, A. F.Clark and A. C. Downton

[2]. Parallel Computing Experiences with CUDA, Michael Garland, Scott Le Grand, John Nicholls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, Vasily Volkov

[3]. Real-Time Dense and Accurate Parallel Optical Flow Using CUDA, Julien Marzat, Yann Dumortier, Andre Ducrot

[4]. Optical Flow estimation using insect vision based parallel processing, Mickael Quenlin

[5]. FPGA based High performance Optical Flow Computation using Parallel Architecture, N. Devi, V.nagarajan

[6]. Overtaking Vehicle Detection method and its Implementation using IMAPCAR Highly Parallel Image Processor, Kazuyuki Sakurai, Shorin Kyo and Shin'ichiro Okazaki

[7] L. Fausett, "Fundamentals of neural networks architectures, algorithm, and applications", Prentice Hall, New Jersey, USA, 1997.

[8] T. Harada, "Real-time rigid body simulation on GPU (chapter 29) in GPU Gems 3", Addison Wesley, 2008.

[9] S. Haykin, "Neural networks a comprehensive approach", Prentice Hall, New Jersey, USA, 1999.

[10] H. Jang, A. Park and K. Jung, "Neural network implementation using CUDA and OpenMP", DICTA '08: Proceedings of the 2008 Digital Image Computing: Techniques and Applications, Washington, DC, USA, 2008.

[11] A. Lefohn, J. Kniss and J. Owens, "Implementing efficient parallel data structures on GPUs (chapter 32) in GPU Gems 2", Addison Wesley, 2006.

[12] D. L. Ly, V. Paprotski and D. Yen, "Neural Networks on GPUs: Restricted Boltzmann Machines", Technical Report, Department of Electrical and Computer Engineering, University of Toronto, 2008.

[13] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau and A. Veidenbaum, "Efficient simulation of large-scale spiking neu-ral networks using CUDA graphics processors", IJCNN'09: Proceedings of the 2009 international joint conference on Neural Networks, Atlanta, Georgia, USA, 2009.

[14] S. Oh and K. Jung, "View-Point Insensitive Human Pose Recognition using Neural Network and CUDA", World Academy of Science, Engineering and Technology 60,2009.

[15] G. Poli and J. H. Saito, "Parallel Face Recognition Processing using Neocognitron Neural Network and GPU with CUDA High Performance Architecture", Face Recognition, Book edited by: Milos Oravec, ISBN: 978-953-307-060-5, INTECH, 2010.

[16] L. Prechelt, "PROBEN1 - a set of neural networks benchmark problems and benchmarking rules", 1994.

[17] D. E. Rumelhart and J. L. McClelland, “Parallel distributed processing: Explorations in the microstructure of cognition”, MIT Press, Cambrige, 1986.
[18] U. Seiffert, “Artificial neural networks on massively parallel computer hardware”, European Symposium on Artificial Neural Networks, Bruges, Belgium, April 2002.
[19] S. Suresh, S. N. Omkar and V. Mani, “Parallel implementation of back-propagation algorithm in networks of workstations”, IEEE Transactions on Parallel and Distributed Systems, 16 (1), pp. 24-34, 2005.
[20] R. K. Thulasiram, R. M. Rahman and P. Thulasiraman, “Neural Network Training Algorithms on Parallel Architectures for Finance Applications”, International Conference on Parallel Processing Workshops, Kaohsiung, Taiwan, October 2003.
[21] R. Uetz and S. Behnke, “Large-scale Object Recognition with CUDA-accelerated Hierarchical Neural Networks”, In Proceedings of the 1st IEEE International Conference on Intelligent Computing and Intelligent Systems, 2009.