STREAM PROCESSOR BASED REAL-TIME VISUAL TRACKING USING APPEREANCE BASED APPROACH

By

ALOK WHIG

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2009

To my parents, I dedicate the thesis for their perennial love, blessings and support

ACKNOWLEDGMENTS

I am deeply indebted to my advisor Dr. Jeffery Ho whose able guidance, moral support and useful discussions enabled the completion of the thesis. I thank Dr. Jorg Peters and Dr. Paul Fishwick for their generous consent to be on my committee. Next, I express my gratitude to the staff at UFEDGE at the University of Florida for providing congenial environment to carry out my research successfully. I add special mention of Dr Pamela Dickrell and Bob Mason at UFEDGE for their generous support and permission to use UFEDGE resources. Ronald Wisener for kindly arranging required hardware. I thank the administrational staff of my department for its help in smoothly carrying out the paper work for graduation. At last my gratitude goes to my seniors and friends at the Department of Computer Science and Engineering at University of Florida for their support and well wishes.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

GPGPU	General Purpose Graphical Processing Unit
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
PCA	Principal Component Analysis
HPC	High Performance Computing
PCI	Peripheral Component Interconnect
AGP	Accelerated Graphics Port
MP	Multi-processor
R-W	Read-Write
W-W	Write-Write
W-R	Write-Read
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
SISD	Single Instruction Single Data

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

STREAM PROCESSOR BASED REAL-TIME VISUAL TRACKING USING APPEREANCE BASED APPROACH

By

Alok Whig

December 2009

Chair: Jeffery Ho
Major: Computer Engineering

Graphics Processing Unit (GPU) has changed the scenario of desktop computing due to its extremely high parallel processing capability and availability at affordable cost. The GPU was originally designed to solve graphical problems that often involve time expensive mathematical operations applied to a large set of points. Recent upgrades to GPU architecture enabled research community to harness full potential of its processing capabilities to solve non-graphical problems without approximating to a graphical problem. Implementing non-graphical problems to run on GPU are filled with challenges and not straightforward. The thesis aims to study mapping challenges involved thereof and explore solutions to get best possible performance gain of an application.

Problem of real-time visual tracking is chosen as a test case to implement and run on GPU using CUDA – a parallel computing software tool. Real-time visual tracking was chosen because of its popularity and application in Computer Vision forming one of its core research areas. Several tracking algorithms are available but often suffer from speed issues due to demand of high degree of accuracy. The thesis revisits work in [18]

and implements the same in CUDA. Significant overall speedup is the major goal of the work presented.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

Visual Tracking is an important area of research under Computer Vision where a computer system is made to mimic a human visual system to track object(s) of interest(s). Real-time tracking finds its applications in many industries such as traffic control, automobile, defense and entertainment. In order to be deemed successful, a tracking algorithm must meet requirements of accurately tracking object of concern, irrespective of external environmental conditions, surrounding target, at real-time. Noise, occlusion, change in lighting intensity and thermal noise of video recorder are prominent challenges that prove obstructive to tracking algorithm in judging position, attitude and form of target. Intermediate stages in tracking are time consuming and tradeoffs to speed.

To overcome problem of speed drop a new parallel computing chip called GPU is employed to help tracking algorithm in expediting the process of recognition and associated complex calculations. The tracking problem is a non-graphical problem where it is not every time possible for operations to run in parallel. The task is to redesign sequential implementation to run on GPU for best possible speed gain of full application.

1.2 Motivation

The topic was chosen to study for factors promising significant speedup for problems exhibiting properties desirable to parallel computing. GPU was originally meant for solving large graphical problems operations mainly as matrix multiplication, vector transformation, arithmetic operations. The enormity of processing speed

attracted attention of researchers and scientists to harness its power to non-graphical problems as in Statistical analysis, Database transactions and manipulations, Simulation in physics, DNA modeling in Biotechnology and Image Processing to name a few, that often have sizeable portion needing expensive mathematical computations, to be solved at a real-time speed.

Another reason to implement on GPU is because of its low cost and high availability. A chip at affordable price with great processing power gave desktop computing a status at par to state-of-art multiprocessor systems.

Lastly, there exist many advanced tracking algorithms categorized by different approaches followed to solve problem, see Chapter 4. For example, probabilistic approach to predict position of target and involve expensive re-weighting calculations for updating object's state is used in [23] whereas the thesis refers to [18] to solve tracking problem that uses a different approach. The thesis is an attempt to aid future users to consider important strategies and results before proceeding to implementation.

1.3 Proposed Solution

Important concerns to real-time visual tracking are speed and accuracy that unfortunately are not easy to meet at the same time owing to time expensive operations intrinsic to intermediate stages before GPU. Converting an implementation at traditional processor to a parallel processor is tricky where performance gain may or may not be experienced. The work presented in the thesis does a detailed analysis of migration problems and useful strategies to overcome the same for significant speed gain. Issues that are bottleneck to overall application performance are discussed with possible solutions. Reducing time complexity by exploiting space complexity is the crux of presented solution. Exploiting space-complexity at GPU is affordable and possible

because of independent sets of processing units and memory with faster fetch rates, see Chapter 2 and Chapter 3.

Section on experimental results and conclusion (Chapter 6) presents results with varying parameters used for tracking and tries to determine optimal configuration of parameters at GPU.

CHAPTER 2
PARALLEL PROCESSING

2.1 Background

Ever since the innovation of transistor in 1970s lead to a breakthrough in digital computing, there has been a race to increase number of transistors on chip. Issues needed to be dealt with, by increasing transistors, were increasing die size and increasing power consumption resulting in heat generation as by-product. Researches helped to solve the problem of power consumption by preferring CMOS based chips to MOS chips. Nanotechnology helped in decreasing wire size and circuitry, leaving space for more circuits on same die size. According to Moore's law the number of transistors added doubled every two years. Manufacturers followed Moore's law to increase processing speed due to ever increasing demands imposed by evolving software applications.

High-end applications such as real-time simulation, statistical applications and graphics forced researches to concentrate on quality rather than quantity. Particularly, graphics applications developed by leaps and bounds in recent years. As a result, game development and its use became popular, requiring real-time processing with high degree of quality. Inspite of synthesizing high speed chips, performance of system often reached saturation owing to limits on physical capacities and capabilities of chip. For example, high-end applications requiring higher degree of precision had to contend with low precision depending on instruction set followed and chip architecture.

High precision calculations, requiring higher clock cycles, were avoided lest speed was compromised resulting game lovers to accept some degree of quality loss.

Demand for real-time processing and higher degree of accuracy lead to development of dedicated chips or co-processors with a view to concentrate on efficiently solving important modules intricate to many popular algorithms running applications.

With growing popularity of digital computing to solve large problems in seconds, expectations grew in direct proportion. It captured interests of scientists and analysts to transfer large problems requiring extensive and complex calculations, to be solved at real-time, to carry out their researches. This saw the advent of special chips, such as DSP (Digital Signal Processing) chips for signal modulation – solving Fourier Transforms, Audio-Video chips for Video Processing, Math co-processors to efficiently execute time expensive floating point operations, linear transformations, equation solver and most recently the graphics card that off-loaded main system processor of all expensive graphical computations. For very large non-graphical and scientific tasks, manufacturers developed special machines designed to solve special problems used by scientific community. This gave rise to high performance computing (HPC) that use, many processors tied together in a topological fashion and solves independent modules constituting a huge task. These machines were in a different league of systems owned by large organizations and not meant for home computing.

At home computing front, dedicated chips and cards helped system garner the beauty of high-precision especially game users realized the most where they now did not have to compromise on quality and speed requirement was met. Much of the chip development was in response to user requirement and market sales. Development in graphics and gaming industry grew the demand for system resources dedicated to meet

real-time speed challenge and efficient complex computations for preserving details. This is observed by a series of paradigm shifts in game development where in early ‘80s it was 8-bit third person game, changed to arcade in early ‘90s then to first person game with 3D effects and recently to multiplayer network games. Coordinating multiplayer data and maintaining high graphical details were difficult and needed channels with higher bandwidth in addition to traditional requirements of high precision and processing speed. Improvements in bus technology took care of channel bandwidth issue. For graphics, the communicating channel between main CPU and graphic card followed chronological improvements as AGP to PCI, PCI to PCI-X and finally from PCI-X to PCI-e. Similarly, there has been a paradigm shift in the way a problem is solved by processor.

A traditional processor follows Von-Neumann architecture and runs on control flow model where availability of next instruction governs action by processor. Operations are handled sequentially. The other paradigm to solve problems is data centric approach where a computation will not proceed till relevant operands are available. This gave rise to parallel computing where a single operator could be applied to multiple data requiring same operation.

2.2 Parallel Computing - Brief Overview

Parallel computing is defined as processing of multiple operations on multiple processors to reduce effective load of operations per processor and speed up the overall processing. Parallelism is achieved by dividing a big problem to multiple small but independent modules. As explained in previous section, parallel computing came as a result of over demand of real-time processing and high degree of precision in computation. Parallel computing can be realized in different ways. Prominently there are

three ways to realize parallelism: at hardware level, software level and at organizational level:

- Hardware Level: Where a system has multiple processors arranged. Each processor is independent but shares a common pool of memory for transacting results. Master processor controls the set of processors.

- Organizational Level: Cluster computing and Grid computing fall here. Here, parallelism is achieved by delegating a large task to different machines, physically separated. Tight coupling and loose coupling are associated with organizational level parallel computing. Loosely coupled systems are distributed systems and rely on network links for communication.

- Software Level: Consists of commands and compiler techniques that identify independent operations at programmer's end and break down to independent modules.

The stress of the thesis is to focus on hardware and software level to achieve parallel computing.

The hardware level parallelism is the most important to parallel computing.

2.3 Types of Parallelism in Computing

Hardware parallelism is implemented in many ways: -

- Bit Level Parallelism: First level of parallelism where more data was packed per word size. Parallelism is accounted by increasing word size. [5] gives detail.

- Instruction Level Parallelism: Data independent instructions are executed simultaneously. Data independent instructions are necessary for parallelism to avoid R-W, W-W and W-R hazards as mentioned in [17]

- Data Parallelism: Multiple computing nodes on data sets requiring same operations perform a single operation. Sec. 2-5-3 explains data parallelism with respect to GPU architecture and operation. Data parallelism is implemented depending on what architecture a chip follows. According to Flynn's classification a chip may fall into MIMD, MISD, SISD or SIMD. Traditional CPU that executes instructions in sequence follows SISD. Executing single instruction for single data set does not show parallelism. SIMD and MIMD show data parallelism. The GPU and other HPC devices fall into SIMD and MIMD types. MISD is of theoretical consideration and has no commercial chip following it. High performance multi-processor super-computers follow MIMD architecture. MIMD is not covered here. Data parallelism is implemented through thread parallelism where a single task is broken down to modules, each handled by an independent thread of same process.

- Task Parallelism: Task parallelism is an extension to data parallelism where multiple nodes may or may not perform same operations on different data sets. Different nodes execute different tasks at same time. GPU running CUDA programs implement task parallelism to increase throughput. Sec. 3-2 explains more about task parallelism realization.

2.4 Hardware Chip Evolution

A chip primarily contains CPU and on-chip memory. Its main functions are to execute instructions and control resources such as I/O devices and main memory. CPU chip forms the brain of any computer system. Early ‘80s saw computer hardware development majorly stressed on upgrading chip architecture. Changes to chip architecture included increasing register size, circuit design and instruction set.

Earlier advancements in chip architecture and hardware were governed by scientific demand and academical interests, leading to mainframes and supercomputers. Chip development started from SISD architecture where, as explained in Sec. 2-3, series of instructions were executed in sequence. SISD was followed by SIMD and MIMD along with increasing register size and type of instruction set, namely RISC and CISC, to meet challenges put forward by industry.

The other branch of hardware development was driven by desktop computing or home computing. Home computing gained prominence due to its popularity, with simultaneous hardware price reduction and user-friendly software applications. Since late 80s when Intel launched its x86 series of chips, home computing has come a long way in terms of software and hardware development. Desktop computing has governed bulk sales making fortunes of several vendors. The mid ‘90s saw unsymmetrical developments in software and hardware technologies with the former leading the latter by order of magnitude. Incessant improvements and developments in software

applications stressed hardware resources and demanded for high processing speed processors.

One solution, engineers came up with was to increase clock speed and resources. Increasing clock speed as solution could not stand long as there was danger of overheating. Bit level parallelism superseded by instruction level parallelism, Sec. 2-3, and super-scalar chips were launched. Later, vendors like AMD and Intel reduced circuit wire size to add more transistors. The Pentium chip series added more registers and special processing units for multiplicative and additive operations

Advent of multiprocessor systems came as a reason to speed up special applications that exhibit high degree of arithmetic intensity. Examples include computer graphics, imaging tools, heavy-duty database applications and server side web applications. A multiprocessor has more than one independent CPU that work in coordinated manner to solve a large problem. Multiprocessor was not successful for home-based computer applications owing to high cost and complex code required for gaining full efficiency. Server side applications were targeted owing to high bandwidth. Intel Xeon chip is a multiprocessor system.

Recently, multi-core systems are a success due to their lower cost and high speed of operation. As explained in Sec. 2-6, a multi-core system is a variant of a multi-processor system and suitable for home computing. Famous brands like AMD's Athlon, Intel's Core 2 Duo and Quad 4 chips made their marks and are still prevalent among buyers. Sec. 2-6 briefly describes major differences between a multi-core system and a multi-processor system.

Continuous software development generated demand for sophisticated chips with specialization. DSP (Digital Signal Processor) chips, Video Processing and Graphics chips acted as co-processors to main CPU. These chips were special in the sense of being fast and dedicated. Inclusion of extra processing elements for carrying out floating point operations and transcendental functions with high speed internal bus made them dedicated and of high accuracy.

The promising future of Gaming industry has seen a spurt growth in online gamers. This surely has put enough stress on traditional multi-core systems to perform at real time. Latest trend set by online multiplayer games demanded for high precision and computation devices, leading engineers to introduce many-core chips. A many-core chip is an extension to multi-core chip that works best for massively parallel operations. Sec. 2-6 highlights important differences between a multi-core and many-core chip. GPU and GPGPU are many-core chips.

The last part in chip development for home computing is an upgrade to many-core GPU to enable programmers to solve non-graphical problems without converting it to a graphical problem. General programming on GPU is called GPGPU. GPGPU is a logical concept of solving non-graphical problems on a GPU. It is not a hardware device but is used to emphasize use of GPU to solve non-graphical problem. In thesis, GPGPU at places is treated separate to GPU to highlight this aspect. A GPGPU uses GPU to solve non-graphical problems. It is to be noted that GPU architecture has undergone some changes, recently, at architecture that helps in realizing GPGPU smoothly. Sec. 2-7 describes more. NVIDIA’s Physx [33] technology on GPU is a good example of a

GPGPU that is dedicated to physics operations that include collision detection, cloth simulation and the like.

Sec. 2-8 briefly visits and lists other many-core architecture based chips that not GPU. A GPU is an invention of NVIDIA and a type of stream processor [Sec. 2-5]. Current GPUs and GPGPUs are not independent and need CPU for their initiation to start operating. Last two decades of chip development tremendously affected desktop computing. What started as response to academical and scientific interest, ended as a necessity to common end-user. The attempt to make GPGPU powerful and independent to CPU is narrowing the gap between scientific computing and home computing. It is not surprising to see several researches going on using GPU as computing device.

2.5 GPU and Stream Processing

2.5.1 Overview

Sec. 2-4 detailed on history of chip development and latest trend being followed. GPU is a many core parallel processor that has revolutionized visual computing. Its upgrade variant GPGPU has successfully made foray to desktop computing. GPU is special because of inclusion of multiple functional cores that independently compute same operation on independent data. The origin of GPU is attributed to computer gaming where real-time speed and accuracy are essential to enjoy. Mammoth number of data elements need to be operated on at same time to maintain real time speed and accuracy. For example in a 3D game, effects of explosions, wind, lighting, water simulation and object instancing require different sets of operations to be performed at huge data set, running into gigabyte. Such humungous task when delegated to CPU,

that has other applications running, becomes a tedious task to handle. Expensive calculations and low bandwidth prevented traditional CPU to output results at real-time.

To offset problems of low bandwidth and low throughput, special co-processors were dedicated to handle special type of problems. Examples include graphic cards for handling graphical operations that are mathematical oriented. Although earlier dedicated graphic cards, (e.g. Vodoo, AGP based cards, Intel), performed complex computational tasks at high speed, the mode of operation was still sequential. Limited memory storage was provided that made them appear as special functional units to CPU. Particular calculations included floating point additions, multiplication, vector processing for matrix transformations and transcendental computations. Soon the performance saturation point was reached with advanced software applications such as CAD, Interactive gaming. Serialized operations for each primitive proved to be a cumbersome task. The GPU exploited the independence between every pixel to parallelize different operations

Focusing at graphical applications, traditional CPU and dedicated graphic cards followed fixed functionality graphics pipeline as shown in Figure 2-1. Traditional graphics pipeline was fixed due to specific sequence of stages that a datum went through before getting displayed on screen.

GPU in this context proved to be a boon to graphics applications developers where they could control and change operations in between to generate variety of interesting of effects. The traditional graphics pipeline ([42], [43]) was a black box where programmers could not manipulate data intermittently and at intermediate stages. Only

feeding of input points were allowed, followed by their display as output on screen. Each point had to go through a series of stages for processing.

2.5.2 Traditional Graphics Pipeline Vs Flexible Graphics Pipeline

- As illustrated in Figure 2-1, the original pipeline applied a coordinate conversion from object space to world space. Prominent operations included local object transformations (translations and rotations) and per vertex lighting.
- Next, transformed object was converted to eye-space coordinate system, keeping viewpoint as point of reference.
- This was followed by primitive generation where a primitive was associated to set of input points. Example, point, line, triangle and others as described in [43].
- The viewing transformation stage then rejected points that lay outside view-port window dimensions and view-frustum. Clipping and culling operation were performed to discard vertices not falling inside their ranges.
- The last vertex operation stage projected processed point set to map them to screen space, where point' coordinates were defined with respect to view screen.
- The scan conversion stage or Rasterization stage generated pixels for every primitive using interpolation techniques. Rasterization filled in the region between vertices of a primitive.
- Per-pixel operations followed next where per-pixel lighting was performed. Just as pixels were generated through interpolation between vertices of a primitive, texels were generated using texture coordinates after accessing texture memory. Pixels or fragment shading was also performed at this stage.
- The advanced stage of pixel operations included scissoring, accumulating, blending operations for effects that gave smooth finish to final scene to be displayed.
- Input data after passing through above stages, was stored in frame buffer memory for final display.

GPU made vertex and pixel operations programmable by exposing vertex and fragment processing stages to programmer. The programmable units were called Shaders executing instructions written to by programmer for data manipulation at vertex

processing stage and similarly at fragment processing stage. The flexibility gave a chance to programmer to send additional data to carry out complex calculations resulting in variety of interesting effects such as toon shading, color bleeding, ray tracing to name a few. Chapter 3 discusses more on GPU languages and results that changed graphics programming forever.

Figure 2-2 shows stages affected by GPU. Shaded stages were made programmable. Each programmable stage added flexibility to the pipeline. Vertex shader handled vertex related operations. Now, programmer could control point transformation, shading and texture generation by writing a program called vertex shader. Chapter 3 briefly describes steps on shader compilation and linking. Geometry

2.5.3 GPU Hardware Architecture

A GPU houses more than one graphics pipeline that work in parallel to achieve high throughput. Figure 30-3 in [11] (on p.474) illustrates setup of NVIDIA GeForce 6 series GPU architecture. Careful observation clearly shows multiple graphics pipeline. There is more than one vertex processing unit and fragment processing unit. Each processing unit is delegated an independent stream of data. Here each processing unit acts as a kernel, as described earlier. Figure 30-3 in [11] (on p.474) agrees to Figure 2-2, where processing units are replicated at shaded stages. Input data is multiplexed to each vertex processor and de-multiplexed at output for Clipping and Rasterization stages to act on.

The NVIDIA GeForce6 series provides for multiple texture caches for operation speedup and higher capacity for storing textures at run time. Multiple texture units are also used for reading and writing texture data through images. Every texture unit is accessible to vertex and fragment shader. One thing to note about Figure 30-3 in [11]

(on p.474) is higher number of fragment processors than vertex processors. The reason is amount of generated pixels or fragments, differing to generated vertices and/or texel by a second order of magnitude or more. To maintain real-time operations higher fragment processors are necessary. Other reason of difference being, more information available at fragment processing stage than at vertex processing stage. Chapter 3 describes common GPU languages such as GLSL, Cg, HLSL that take advantage of GPU architecture to provide efficiency.

Figure 30-4 in [11] (on p. 475) shows the architecture of a vertex processor in GeForce 6 series GPU. With reference to Figure 2-2, the primitive assembly stage is incorporated inside every vertex processor in Figure 30-4 in [11] (on p. 475). Primitive assembly establishes connectivity between set of input vertices. Example, a triangle has three vertices and the edges establish connectivity among them. Every vertex processor has components specialized for specific function. Floating-point scalar unit deals with floating-point operations, floating-point vector unit is for efficient transformation of a column or matrix of points. Vertex fetch treats texture unit as cache and reads in points for processing and passing out to primitive assembly. Branching unit is for handling conditional statements. View-port processing stage is for defining constraints and restricting points with values falling out of range.

Figure 30-5 in [11] (on p.476) gives block diagram of a single fragment processor belonging to GeForce 6 series GPU. Internally, shader unit work in tandem with texture processor for interpolating texels during shading. Interpolation generates huge pixels and texels that are kept at texture caches for later use. Branch processor allows only those values to pass that meet constraints. Fog ALU is for fogging effect.

The main reason for presenting architectures above is to convey the high computational capability of each core working in tandem with others to produce high throughput. Independent data stream reduces cycle wastage. Figure 30-3 in [11] (on p.474) is an example of SIMD architecture. Post fragment processed operation such as depth comparison and blending are done simultaneously at multiple buffers. Again, multiple depth buffers indicate operations on independent data streams. As referred in traditional graphics pipeline setup, secondary operations that include blending, alpha value testing (for transparency), scissoring are done on each datum (final pixel/texel) before display at frame buffer. Frame buffer memory is also partitioned and interleaved to form memory banks.

2.6 CPU vs. GPU

A GPU is a many-core hardware device that accommodates more processing units, called as cores, than memory. In Figure 2-3 A and Figure 2-3 B, squares represent functional units that individually operate on different but highly independent data streams. Figure 2-3 A is a model diagram of a multi-core CPU that is latest in market. Figure 2-3 B has reduced space for memory and unlike in CPU, cache is no longer common to all functional units. Single controller is assigned to set of units indicating SIMD architecture. The MIMD feature is exhibited by having multiple SIMD units as indicated by multiple controllers, each controlling a set of function units. The functional units is described as ALU as a generic functional unit for the ease of understanding. Data streams requiring different operations but same operation for all its constituting elements are assigned to different controllers. Each stream element is operated at by one of many functional units. Operating on vector of elements is easier. Time complexity is reduced as a result of increase in space complexity.

Hardware level differences between CPU and GPU are also reflected at software level as programming constraints. Chapter 3 elaborates the effect of architectural constraints at programming level. Performance of CPU depends on fast cache access and functional units. Traditional, uni-processor CPU is control flow based where data elements are sequentially processed and is instruction centric. GPU on the contrary is data-flow based model where multiple data can be operated at same time. To a set of functional units, instruction is same; an operation is executed when relevant data elements are available. In other words, CPU is driven by instruction whereas, GPU is data driven.

Traditional CPU follows SISD architecture while GPU follows SIMD and sometimes MIMD (see Sec. 2-5). Threads running in thousands can be activated for the purpose of solving a large task, in GPU. For multi-core CPU, even though multiple threads can be generated, efficiency is restricted due to thread scaling, when inter thread communication takes a significant role. Race around conditions, scheduling and synchronization issues become dominant then. With GPU, above challenges are better handled. To be accurate GPU follows SIMT architecture [34] as parallel threads control an operation rather than parallel processes. Chapter 3 briefs on NVDIA GPU's handling of thread handling.

Lastly, CPU stresses more on reducing memory latencies, thereby providing cache layers at different levels. A GPU gives high priority to throughput and bandwidth despite experiencing high memory latencies. A GPU hides memory latencies by keeping functional units busy with surplus data.

2.7 General Purpose GPU

A CUDA enabled General Purpose Graphics Processing Unit is an upgrade to GPU architecture where besides having multiple functional units, called cores, has variety of memory models. Again, it is stressed that a GPGPU is a logical concept where a GPU is used to solve non-graphical problem without approximating it to a graphical problem to follow graphical pipeline. To realize the need of not converting a problem to its graphical equivalent, certain changes were done at GPU architecture to enable programmers to write programs much simpler than worrying about shading languages. CUDA is explained in Chapter 3.

Different memory models have different features and are covered in detail in Chapter 3. It is to be noted that at places in thesis, CUDA enabled GPU is interchangeably used as GPGPU. A CUDA enabled GPGPU handles its resources much better to GPU. Partial communication among multiple threads is possible in GPGPU. In GPU a thread is mapped to a single graphic pipeline and multiple threads are equivalent to multiple pipelines. As pipelines are independent, so are threads. Other major difference between the two is GPU's restriction from accessing frame buffer memory in between. GPGPU has a different architecture where programmer is allowed to load and store data at main memory. Thus, a GPGPU is a special type of GPU that is closer to CPU. A GPGPU has its own RAM called Global Memory. Other memory models include Texture memory, Constant memory, Shared memory and local memory. Figure 2-4 illustrates a GPU-CPU interaction for solving a task. The figure also shows different memory models in component form and their interaction to surrounding environment. Here, the environment defines the scope.

According to Figure 2-4, GPU is not self sufficient to start and carry out whole operation of completing a problem delegated to it. CPU intervention is required to provide with initial data set and trigger series of operations. Among different memory models in GPU, the global memory (or Video memory as indicated in the figure) and the local memory (cache memory associated with each controller) are readable and writable. Readable and Writable jobs are Gather and Scatter respectively in GPU terminology. A scatter is a read operation where a value is passed on to another variable or broadcast to a vector of variables. Gather operation is reverse to a scatter operation where values are collected from a set of variable or a single variable to continue further computation. GPU treats a stream of data as continuous chain of multiple graphics pipeline found at GPU.

It is to be noted that a CUDA enabled GPU may have a different architecture to GPU where fragment and vertex processors may be replaced by independent functional units arranged differently topologically. For example, GPUs of GeForce 7 Series and below support GPGPU but requires translating a non-graphical problem to a graphical problem. Here GPGPU is a logical concept where solutions to non-graphical problems are approached using GPU architecture, consisting of multiple graphic pipelines.

CUDA enabled GPUs belonging to GeForce 8 Series and above have multiple cores arranged as a 2D grid of multiprocessors. Each multiprocessor has multiple cores. In this fashion, latest GeForce series chips partially follow MIMD architecture and each multiprocessor follows SIMD architecture, illustration in Figure 2-5 below. Another difference between CUDA enabled GPU and traditional GPU is that former is more

integrated than later. Chapter 3 has details where it is useful to describe GPGPU architecture along with software level architecture.

In GeForce 8 series and above, programmable unified shader replaces traditional graphics pipeline stages. Unified shader is a collection of processing units each with different functions – vertex processing, geometry processing, fragment processing as mentioned at [24]. Advantage of programmable unified shader is direct access to GPU's main memory.

2.8 Other Alternatives to GPU

A stream is defined as an ordered set of independent data elements requiring same operation and a kernel is an operator or function that transforms an input stream. A stream processor is a hardware device with multiple functional units, acting as shaders, that processes streams. A GPU is a kind of stream processor meant for solving graphical problems. The shaders associated to are fragment and vertex shader. A kernel is a program written to run on one of the processor types. Architecture followed by a stream processor may fall into either of SIMD or MIMD or hybrid of both.

There are other alternatives to GPU that are stream processor. Apart from NVIDIA GPU GeForce Series that is home bound, NVIDIA Tesla and Quadro are tailored for scientific and database servers respectively. The difference between GeForce and Tesla and Quadro model lies in presence of extremely efficient functions chips, high-speed memory buses and more number of cores. The latest in line is Fermi that has even more cores. Apart from NVIDIA; there are other stream processors available, tailored to specific tasks.

Many-core chips: FGPA, ClearSpeed, Intel's Larabee, AMD Fusion, ATI Radeon and Cell B.E are other stream processors explored in [41]. Each available option has

advantages and drawbacks with respect to a GPU. For instance, FGPA has arrangement of functional units and unlike fixed design during manufacturing, programming, suiting a problem to solve, can change the design of FGPA. ClearSpeed slightly differs from GPU in availability of ECC, Error Correcting Codes in its memory; desirable for HPC (High performance computing). Intel's Larabee has multiple cores that run on x86 instruction set. Larabee claims to outperform NVIDIA's GPU series by efficiently dealing with memory latency issues. According to [41], Intel's Larabee will have the facility of inter-processor communication that is lacking in GPU. Presently, little is known about Larabee's architecture and is to be released by 2010 beginning.

Cell B.E. (Broad band Engine) is a many-core processor that has the capability to do stream processing and is a close cousin of GPU. Current studies have shown Cell B.E to be superior to GPU in terms of bandwidth and are better suited for same data set requiring complex operations at a time, whereas GPU executes a single operation on a data set at a time. We chose GPU despite of Cell B.E's edge due to former's wider market presence and availability. Another reason of choosing GPU over Cell B.E was latter's handling of memory differently. Unlike GPU architecture, where there is a separate cache per core, similar to a normal CPU, the Cell B.E uses DMA (Direct Memory Access) transfer control indicating a little programmer's intervention.

High popularity of GPUs demanded for programmer friendly software that could let a programmer to design a program without bothering about underlying GPU architecture. Computer Graphics vendors like NVIDIA and ATI came up with CUDA™ (Compute Unified Device Architecture) and CTM™ (Close To Metal) [now succeeded by AMD Stream SDK] respectively to support programmers' wishes. Not going much into

details, little changes made in GPU architecture and software architecture enabled parallel processing not to be restricted to graphics pipeline for realization. In other words, a problem requiring parallel processing need not be transformed to approximate a graphical problem.

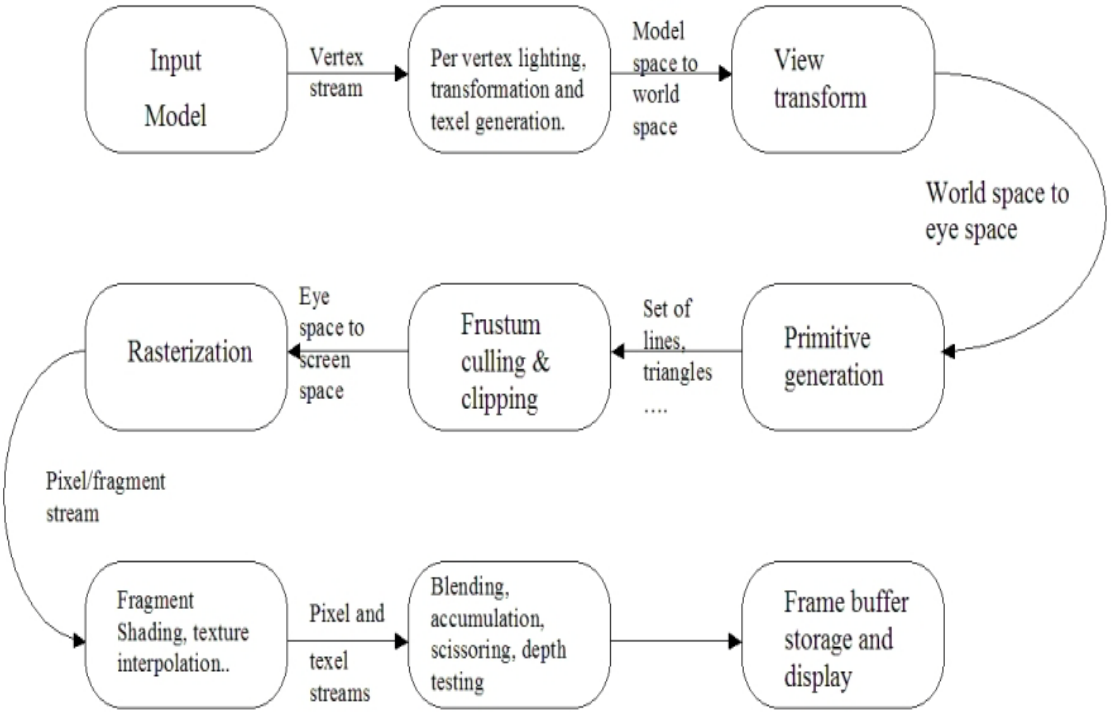

Figure 2-1. Traditional Graphics Pipeline Stages.

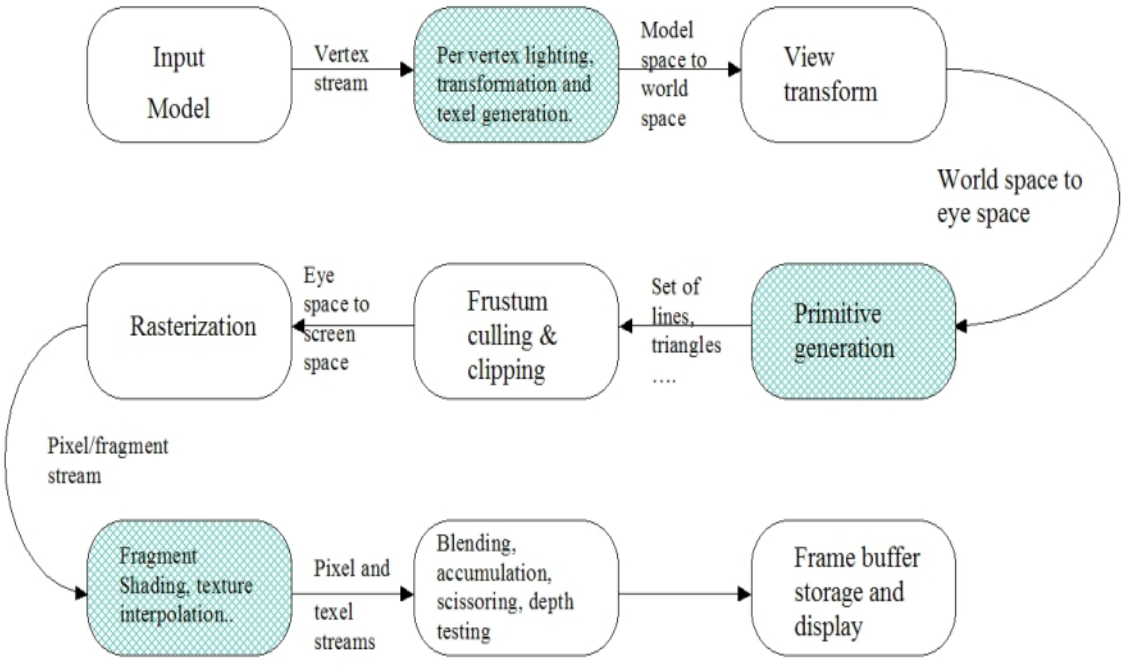

Figure 2-2. Stages in GPU graphics pipeline.

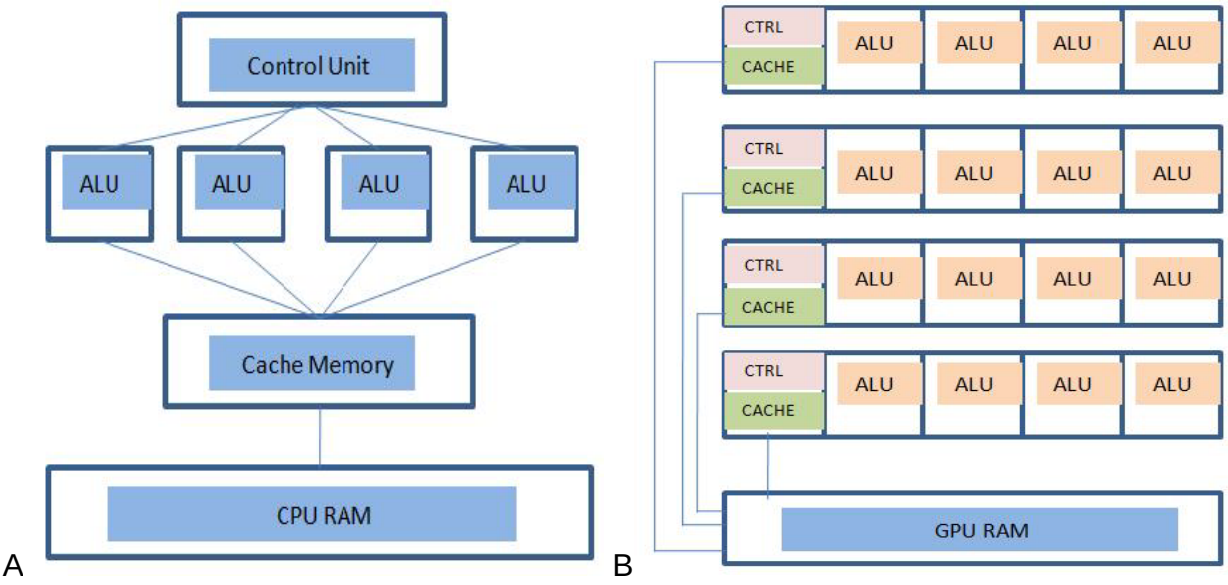

Figure 2-3. Architectural difference between a CPU and GPU. (Adapted version source: NVIDIA CUDA Programming Guide Version 2.2.1(May 2009), available at http://www.nvidia.com/object/cuda_get.html. Last accessed November 2009).

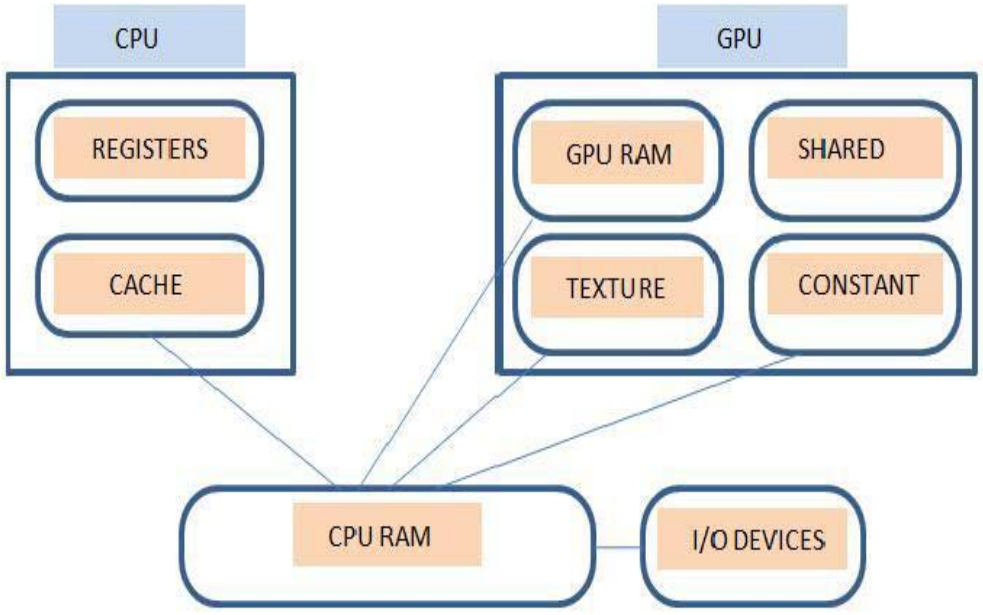

Figure 2-4. CPU-GPU Memory structure and their interactions. (Adapted from source: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter33.html. Last accessed on November 2009)

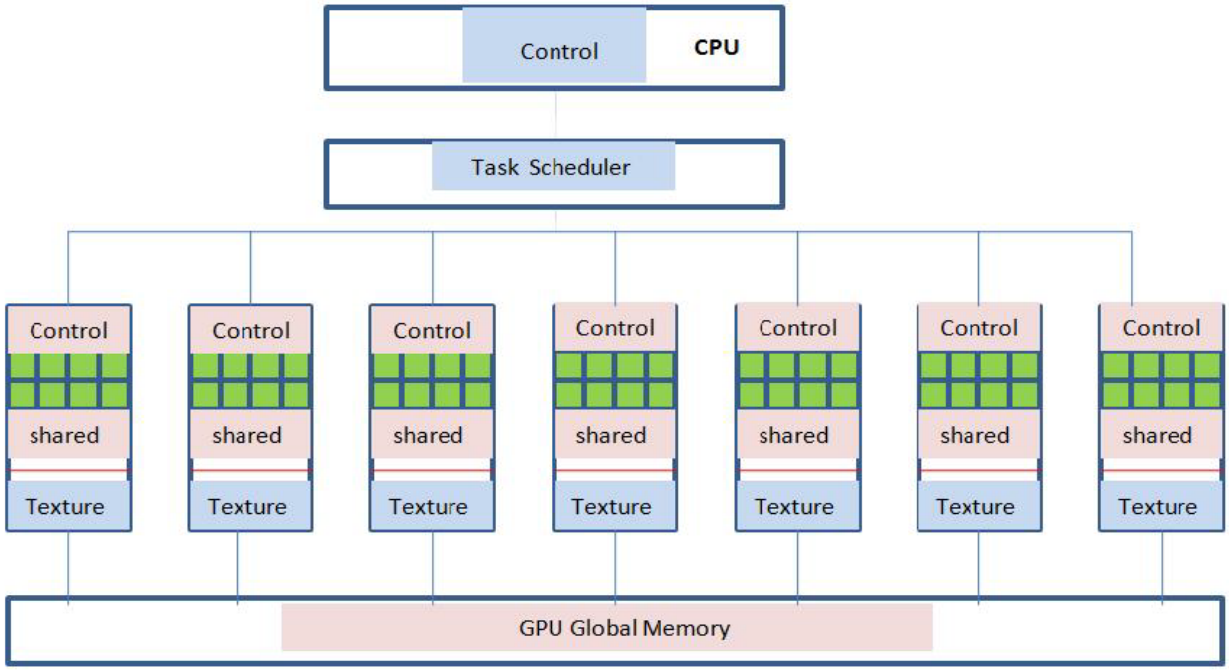

Figure 2-5. Schematic diagram of GPGPU. (Adapted from source: http://www.sbel.wisc.edu/Courses/ME964/2008/Documents/gpubook.pdf . Accessed on November 2009)

CHAPTER 3
A VISIT TO MANY-CORE COMPUTING LANGUAGES

3.1 GPU Languages

Chapter 2 gave an idea of characteristics of a graphical problem, namely high arithmetic intensity, and high degree of data independency. A graphical problem maps well to GPU. A non-graphical problem has low degree of parallelism where major chunk is sequential in nature and data dependency is present. Underlying device architecture directly influences programming limitation. Sections 2-5 to 2-8 described GPU architecture and differences to CPU. GPU architecture permits operations that use multiple graphic pipelines working in tandem on independent streams and finally pooling results to memory. No data is allowed to exchange between streams. For data within a stream, low degree of dependency is possible using external storage called cache. Programming constructs like control flow statements and feedback operations such as recursions were not possible in earliest models of GPU.

With popularity of GPU for visual computing, demand for improving programming flexibility increased. Visual computing language, also known as shader programming, received a boost in its development. Several vendors released packages for visual computing. Khronos Group launched GLSL as an extension to OpenGL programming language; Microsoft released HLSL as an extension to Direct3D language. NVIDIA jumped into the fray with Cg scripting tool that is similar to Direct3D and closer to C programming. GLSL is popular for being platform independent and can work on NVIDIA's GeForce Series and ATI's Stream processor. GLSL is compatible to Direct3D programming. Programmers were able to manipulate data at vertex and fragment generation stages for achieving customized effects.

3.1.1 GLSL Overview

GLSL was launched as an API (Application Programmer Interface) to openGL version 1.4 in 2005 and later became a part of OpenGL from version 2.0 and onwards. It is a high level programming language that exposes OpenGL graphics pipeline for manipulation, commonly known as shader programming where shaders are the vertex shader at vertex processing stage, fragment shader at fragment processing stage and geometry shader at primitive assembly stage before Rasterization.

GLSL written programs are not stand-alone and need to be attached to main program to achieve desired results. The idea of including a discussion on GLSL is to quote on important data structures and GPU concepts useful in parallel programming.

3.1.2 Special data structures and GPU concepts

A GPU is a hardware acceleration device that operates on several data elements in parallel. GLSL has built in functions that are optimized to run on GPU. The optimized built-in functions take advantage of GPU architecture discussed in Chapter 2. Following are prominent data structures and concepts used in GPU programming:

- Swizzle operator: Used for interchanging elements. Shader programming have modified data types where swizzling works best. For example, data type float4 is treated as a single entity with four float components. Such data types are obvious as a homogenous vertex or color datum has four components.
- Sampler: A sampler is an opaque 1D\2D\3D memory space in texture memory, of GPU, that is designed for caching data. Texture memory is fast and used for reading in input data. It is opaque to modification. Type of sampler (other than dimension) determines capacity to store data. For example, a 1D sampler of type RGBA would store 32 bit data as one texel.
- Frame Buffer Object (FBO): A logical structure of pointers, where texture units are attached to and used to write (render) to texture. FBO is used in Ping Pong reduction technique.

- Reduction: Technique used in computing extremes, sum, average of a large series in logarithmic time. Reduction uses divide and conquer strategy to solve a problem. At each step size of input array is reduced to half.
- Sorting Networks: Used in sorting elements. Methods such as Radix sort, Bitonic sort are implemented.

A good reference to above concepts is [13], [15]. Shader languages like GLSL and Cg retain most of the properties of high-level languages. Limitations follow according to capability of hardware - GPU. Recursion cannot be implemented. A multi-pass operation (for-loop) is broken down to single pass operation by executing same function for each core with a different value as shown in Figure 3-1. Here PE stands for Processing Element.

3.2 CUDA Architecture

CUDA is software and hardware architecture for realizing parallel computations at GPU. It can be viewed as architecture for hardware device when referring to hardware threads, memory models and processing core arrangement. As software, it is an extension to C language where special commands enable a GPU to be used to perform computations at GPU. CUDA act as a driver that plays a role of interface between software program and device. Currently, multiple languages are supported by CUDA such as C, C++ (excluding inheritance, class), DirectX, MATLAB, Java, FORTRAN, openGL. At software driver level, commands issued in native language are converted to assembly language forcing underlying device to execute instructions. Figure 3-2 illustrates software level architecture of CUDA.

At hardware level, CUDA deals with thread management and memory addressing. In comparison to traditional shader languages, CUDA allows for direct memory addressing of GPU for greater programming flexibility. Chapter 2 gave an overview of

GPU architecture as a many-core processor. Processing cores are grouped into as set of multiprocessors

CUDA works on NVIDIA GPUs. For a system with heterogeneous many-core processors – ATI GPUs (Radeon series, Mach series, FireGL, Rage, FireM, FireMPro) Intel's Larabee, NVIDIA GPUs (GeForce, Tesla, Quadro), multi-core CPU and other multiprocessor OpenCL [31] technology is used as an intermediate tool to interface between them.

3.3 CUDA Interaction with GPU

When a kernel is launched by CUDA, it allocates set of threads for execution. The set of threads are organized as block of threads in a grid as shown in Figure 3-3.

Each thread is indexed by three built-in variables: threadIdx, blockIdx and blockDim. Each of the variables points to three-dimensional arrangement of threads. At the highest hierarchy of thread organization is Grid – a 2D matrix of blocks. A block is constituted by set of threads. Threads inside a block can be arranged in 1D\2D\3D. For GeForce 8 series, maximum number of threads in a block can be 512 in total, irrespective of its dimensionality. Example: 512 threads can be arranged as (512, 1, 1) or (256, 2, 1) or (128, 2, 2) and similar permutations. Variable blockDim tells total length of block. Variable threadIdx is used to identify a thread, local to a block. Above variables are important from the point of view of removing need of a for-loop for iteration – saving time. Similarly, maximum value of grid dimensional value can be 65,536 in X and Y dimension and 64 in Z direction. Thus, million threads can be allocated for kernel execution by CUDA using kernel execution configuration.

Execution configuration is important to pre-allocate GPU resources for kernel execution. An execution configuration has four parameters besides function's parameters.

Foo<<<BLOCKS, THREADS, SHARED_MEM, STREAM_ID>>> (<params>)

BLOCKS describe the block of threads arrangement in 2D grid. Built-in variable blockDim tells the size of each block. THREADS inform number of threads in each block. SHARED_MEM is optional and tells the amount of shared memory allocated for carrying out calculations by kernel Foo (). STREAM_ID is optional and its value identifies the stream in use. STREAM_ID becomes important when dealing with asynchronous data transfer and computations.

3.4 Performance Metrics

A GPU is a co-processor to CPU that offloads parallel processing tasks from CPU, executes them and sends back results to CPU through a data link called PCIe (Peripheral Computer Interface Express).

Efficiency of parallel computing on GPU is measured by two metrics:

- Bandwidth measurement: Rate of bytes transferred, determining speed of operation.
- Processing throughput also called arithmetic intensity gives the rate of floating point operations.

Performance of CUDA enabled GPU is also determined by execution configurations. Tests have shown GPU works best when number of threads in block is a multiple of 32, typically 32, 64 and 128. The reason is based on GPU architecture. According to [32], [34], [35], GPU executes set of 32 threads in one transaction called warp. 32 threads constituting a wrap are executed without synchronization and fastest.

When executing a block, CUDA architecture divides a block to set of wraps, and schedules each wrap. A wrap waiting for an operand fetch is rescheduled into a queue. High number of blocks (as set by execution configuration), are assigned to different multiprocessor. Each multiprocessor queues blocks. High speed of GPU is attributed to allow only one block to be executed at a time. If current block waits for an operand, its state of execution is saved and en-queued. The GPU instead of waiting for operand fetch executes the next in line block. This ensures high occupancy for each core.

Block size as multiples of 32 – 64, 128 have shown to work better. At these configurations, GPU is optimally occupied. CUDA recommends number of blocks to be at least twice the number of multiprocessor in GPU. For NVIDIA GeForce 8800 GS, experiments agree to the recommendation above. At high configurations of 256 and 512, the performance slightly decreases due to scheduling. For our problem we tried with configuration of lower multiple of 32; 32, 64, 96 and 128.

GPU (also called as device) performance also depends on memory and instruction optimizations. Memory optimization depends on alignment of data fetched by GPU. Memory coalescing ([32], [34], [35]) is desirable to avoid wastage of extra memory cycles. Data is fetched as a set of 32-bit, 64 bit or 128 bit per memory cycle. Data of size from one of the three is well aligned. Misaligned data causes extra cycles to spend on fetching full data. It is not clear, but we guess, data element size of 32-bit, 64-bit or 128-bit may be the reason behind GPU's best performance when number of threads in a block is a multiple of 32. Chapter 3 discusses thread organization.

Each core processor in GPU processes fast but has to wait for next block to be transferred to registers from global memory. Global memory fetch has latency of 400 to

600 cycles and is time expensive with respect to processing time. This results a functional unit to be idle. Block size of higher multiples of 32:- 256 and 512 involve breaking it to smaller units. Subunits of a block are en-queued and then processed. Multiple large blocks are required to wait till all parts of first block in queue are finished. Throughput decreases when large sized blocks are processed due to limited queue size. Instruction stalls cannot be ruled out as an effect, decreasing throughput.

For block sizes at 64 bit, 96 bit and 128 bit, queue is full and no stalling is involved, occupying the processing elements with data. Thus the memory latency is compensated by functional units by being busy in processing. After processing current block, next in line block is available instantly, giving optimal performance.

Another factor that affects GPU's performance is its compute capability. Compute capability defines restrictions beyond which a device performs less efficiently. For example GPU of compute capability 1.1 cannot work with double precision values. GeForce 8800GS is compute capability 1.1 where as GTX280 is 1.3 and can work at ease with double precision values. For compute capable 1.1 chips to work with double precision, a double value is recast as single precision value. [34] covers more on compute capability.

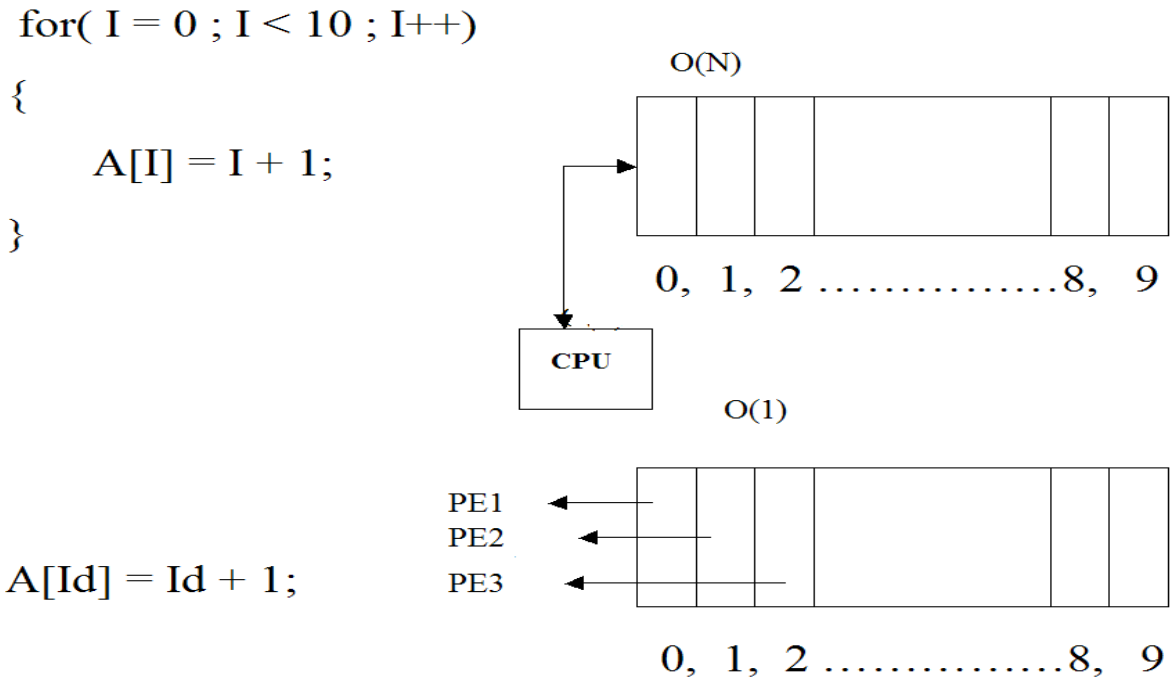

Figure 3-1. Multi pass function to single pass at GPU.

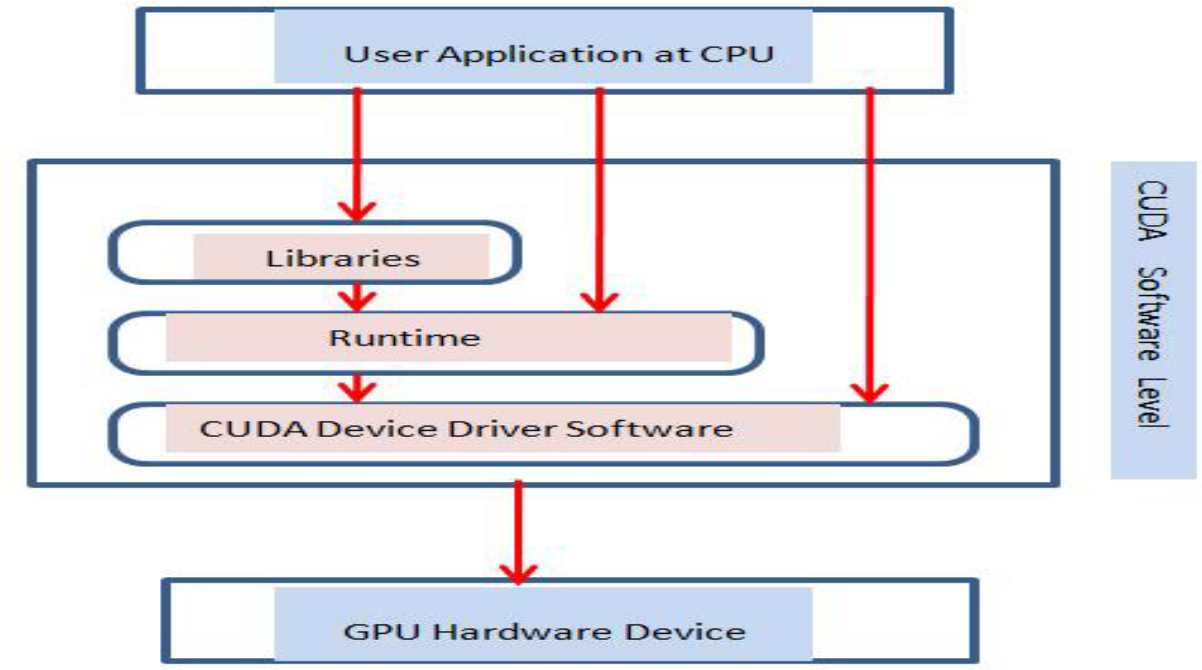

Figure 3-2. CUDA Software Level Architecture. (Adapted from source: NVIDIA CUDA Programming Guide 1.1, available at http://www.nvidia.com/object/cuda_get.html. Last accessed November 2009).
CUDA as hardware architecture

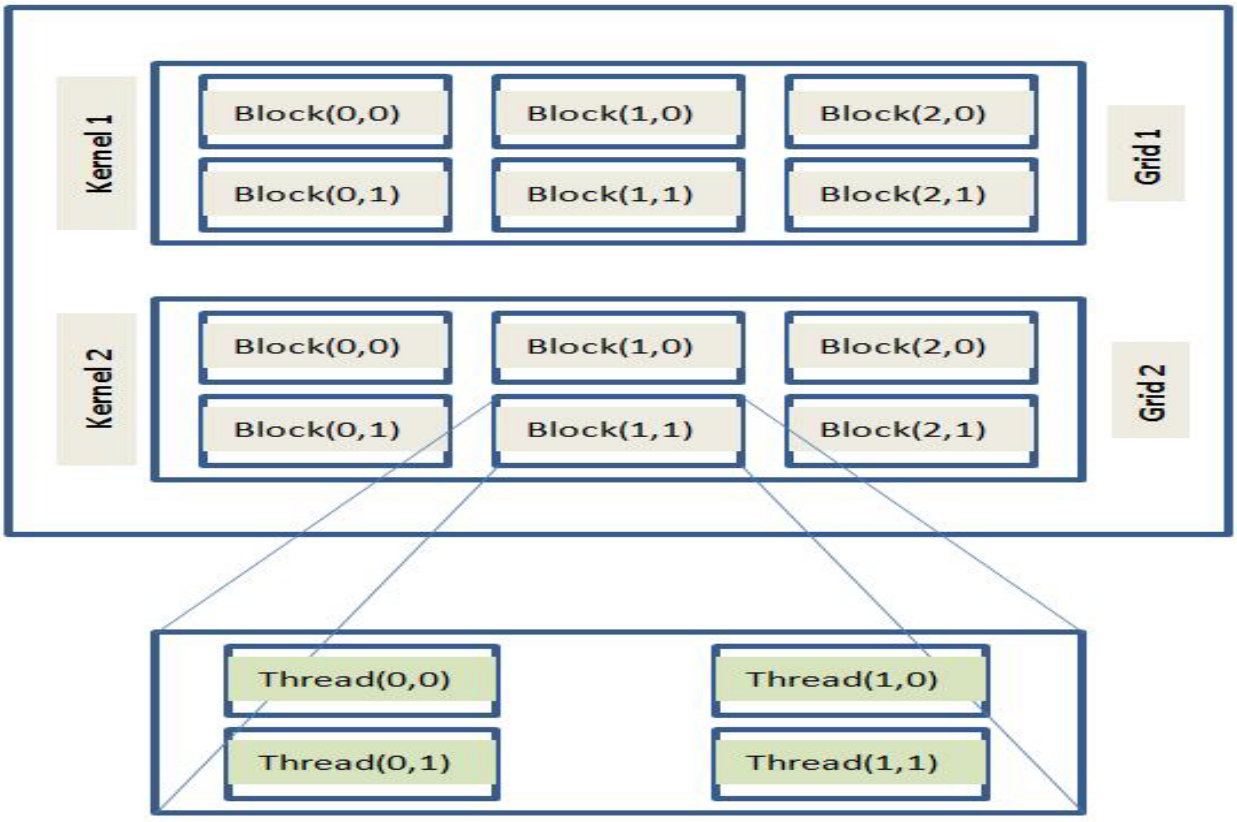

Figure 3-3. CUDA thread organization. (Source: NVIDIA CUDA Programming Guide 1.1, available at http://www.nvidia.com/object/cuda_get.html. Last accessed October 2009). CUDA as hardware architecture

CHAPTER 4
REAL TIME VISUAL TRACKING ON GPU

4.1 Overview of Visual Tracking Concepts

Visual tracking is an important problem in Computer Vision where an artificial computer system is made to mimic human vision system, to detect and track objects of concern. A vital part of visual tracking is to maintain invariance to environmental changes surrounding an object through adaptation. Prominent environmental factors include change of illumination, occlusion and orientation. The final image as received at viewpoint can be further deteriorated by thermal noise generated by recording instrument.

The last decade has seen intense research in developing visual tracking algorithms. A challenging part of tracking is to correctly predict occurrence and state of concerned object on the basis of current and previous observations of its state. The challenge becomes intense when varying environmental factors are included. Significant gains and robust techniques have evolved to meet above challenges but still not to perfection. The limitation is imposed by the design of algorithm and availability of hardware chip for deployment.

Real-time tracking anticipates two factors to be met efficiently in order to it be successful. First, to accurately track object and second, to track with speed, avoiding interruptions. A break-even point for terming tracking algorithm as real-time is if procedures of detection and recognition are done at frame speed near to twenty-four frames per second. To be accurate a tracking algorithm tries to reduce the impact of external environmental factors.

4.2 Different Approaches to Solve Tracking Problem

There are several approaches to solve tracking problems and are categorized majorly into five groups:

- Probabilistic Based Tracking: Notable works using this approach are by [7]. Here parameters of object to track are estimated using probabilistic techniques such as EM algorithm, particle filtering [1]. Degree of accuracy is controlled by error tolerance parameter. [23] is a sparse template based matching tracker that uses particle filtering for estimation.

- Pixel Statistic Based Tracking: Pixel attributes (such as intensity and color values) are used to define an appearance of target object. Sequences of images are used to compute the attributes assuming a model of distribution, normally Gaussian model. Exact distribution model is estimated on frame-by-frame basis till a good result is achieved from where it is easier to predict attributes. CONDENSATION technique [19] is widely used for this purpose. [4] used CONDENSATION technique. [28] uses histogram technique to track objects.

- Appearance Based Tracking: Models appearance of target object using a set of bases in image subspace. The subspace is synthesized by PCA (Principal Component Analysis) to a collection of frames. The technique is effective as demonstrated in [40]. Notable work done in [16], [20].

- Contour Based Tracking: The approach is used to track deformable templates using active contours. Here the silhouette of object to track is used as a changing curve with time. Object contour is represented as specially designed energy function [21] that encompasses thin-plate function, point function and edge function.

- Blob Tracking: The aim is to identify invariant features used as key points for tracking. Applying certain mathematical operations such as gradient and differentiation to get maximal or minima values identifies key points. Nocontributions are by [3], [22], [25], [27].

4.3 Visual Tracking - An Example of Non-Graphical Application

Object tracking involves procedures that demand for complex computations and high memory storage for feedback operations. Object tracking (Figure 4-1) is summarized below:

Marking the target object manually initializes tracking operation. Initialization records target's starting location and information is used for next frame. The next frame goes through image enhancement and noise removal for detection procedure to detect edges to ascertain any object in scene. For example, a scene may have faces, books and other arbitrary objects. The information of each detected object is passed to recognition stage where one of the four approaches is used to recognize target object.

An approximate location and pose of object is determined and recorded. The approximate location is used to predict motion and position in next frame. Error is detected at prediction stage when actual location and predicted location differs. Position recorded previously is updated. Update information is passed on to recognition stage for robustness and next frame in queue is waited for.

Recognition stage is computationally expensive and often recursive. Recursion and complex computation are purely sequential in nature as high data dependency and memory demand increases according to level of complexity. Neural network methods are often applied to recognize objects and improve accuracy of the whole operation. Out-of-order execution and complexity is the mainstay of such procedures and challenges can me successfully met by CPU implementation. For image smoothening and enhancement operations, filters are needed that indicate to sequential nature of the whole process. Conditions of high arithmetic throughput and independence are not exhibited by above operations.

4.4 Code Mapping – CPU to GPU

Mapping from CPU to GPU is challenging and boils down to transforming a sequential version of a function to parallel version. Not every sequential procedure can be completely parallelized but portions of it can be done. For example Kalman filter

based tracker for motion tracking, is sequential and involve complex computations. However, researchers have successfully ported it to GPU to get huge speedups as reported in [2]. [37] successfully ports neural networks based recognition engine to GPU for good speed gains. [39] implemented genetic algorithm on GPU.

The essence to work with GPU is to either port a portion of a sequentially complex algorithm or to re-design it completely at GPU. The latter task is difficult but worth doing. There is no criterion to decide on to port or redesign a code. The only saving tips are to scan code and identify components where maximum gain can be made. It is wise to choose portions of sequential code that can be parallelized and make major chunks of the CPU processing time. Chapter 3 already explained points to optimize algorithm.

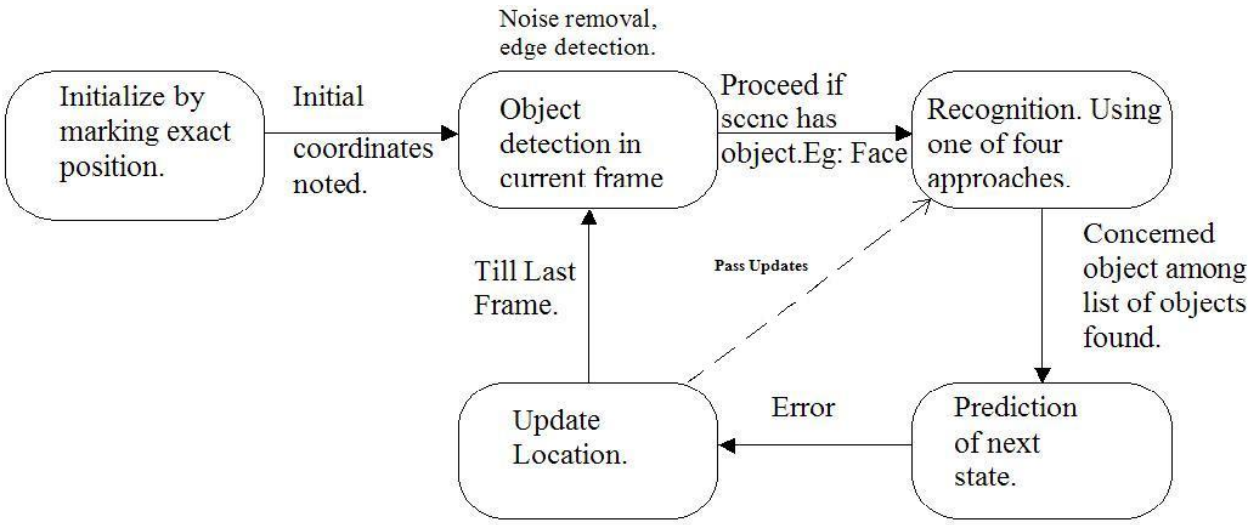

Figure 4-1. Stages in object tracking.

CHAPTER 5
LINEAR SUBSPACE BASED REAL-TIME TRACKING

5.1 Algorithm

This section revisits [18] for the original algorithm and briefly discusses operations involved there at. A tracking algorithm has three prominent stages: Object detection, Update and Position tracking (refer Figure 4-1). Detection is responsible for object identification at a given frame. The algorithm starts to operate by manual tracking of object at first frame, to initialize detection and aid in tracking by remaining stages. Initial location of object is recorded and a vector represents object. The vector encodes object's pose and orientation, collectively called as its appearance.

Object's appearance is a template of object used in successive frames for matching. Estimation of target location in successive frame is determined by generating window samples. Every window sample is rectangular and sample parameters are randomly generated following Gaussian distribution, in the vicinity of target object's location at previous frame. Generated *S* window samples, $[x_0, x_1, \ldots, x_{S-1}]$, are of different dimensions and orientation. A window sample's signature is determined by a quintuple set [x, y, w, h, θ], where location is given by coordinates (x, y), dimensions by (w, h) and orientation by θ. Each window sample is rectified to a fixed size k by k. Digital image is discrete and each rectified window sample can be treated as a vector of pixels of size k * k. Vector of size k * k is represented by a point in high dimensional space of R^K, K = k * k.

R^K is a high dimensional vector space representing images as points and called Image Space. Window samples are represented as points $[x_i]$, $i \in [0..S-1]$. At each frame, appearance model is represented by linear subspace L of image space R^K. L is

constructed by collection of N vector representations of window samples corresponding to previous frames when object was detected successfully. Before constructing L, location of target object is approximated by matching different, generated, window sample vectors to vector representation of window sample of previous frame containing the target. Window samples are generated with positional parameters near to most recent object location. The reason for choosing near-by locations is based on assumption that speed of object does not vary drastically in real-time. The second part of the algorithm is to maintain consistency in wake of changing external factors such as variation in illumination, pose change and occlusion. Consistency provides robustness to successful tracking.

Linear subspace L is constructed using the window sample W_i whose corresponding image point vector x_i is least distant to sample vector of correct target location from previous frame. At initial stage, linear space is represented by current window sample point correctly determining object's position. For successive frames of observations, mean of observations is used of dimension D = N/k where, N is number of previous observations corresponding to previous frames when object's location was detected correctly. N is retained by tracker memory. k is batch size and N > k. Every frame gives y_i (candidate sample least distant to template) and for every batch (of size k each) of frames, a mean vector is used to construct and update L. Once L is formed, sample points of each successive frame in a batch are matched to this mean vector. All boils to computing distances between candidate window samples to bases vectors spanning L.

The second matching is the basis of robustness against variations. The difference between sample point x_i and y_i is worthy to note. Former is a window sample belonging to a frame while the latter is the window sample x_i that is least distant to vector represented window sample in previous sample containing correct position of target object. W_i is used as image point in second step of distance computation when comparing to mean vector in L for subspace update and consistency against external changes.

Distance Metric uses L^2 distance norm to locally determine target's position among different samples W_i. Next from a given set of N observations, $[y_1, y_2, \ldots y_{N-1}]$ subspace, L, construction uses L^∞ norm (uniform norm) to search vectors spanning L. According to, [18], an observation means local window sample candidate that has least L^2 norm among randomly generated samples at a frame. In other words, tracking object's location is done locally at frame first and robustness to external changes is done as second step. Both step use different norms for different reasons. [18] uses L^∞ norm (Uniform Norm) in L^∞ (L , $[y_1, y_2, \ldots y_{N-1}]$) < δ. to compute distance value for every observed sample y_i from current frame to subspace L. The inequality gives non-unique solution and corresponding many subspaces but the main advantage is that only one of many solutions can give a subspace to be used for estimation leading to an inexpensive subspace update algorithm.

Subspace L is a collection of mean observation from a set of recent past k frames that successfully determined target object's location, $[m_0, m_1, \ldots, m_{D-1}]$, where D is subspace dimension determined as N/k. Following LIFO order where a new mean observation from a new batch is inserted and first observed mean observation is

discarded carries out the update. The new set becomes [m_1, $m_{2,}$..., m_D]. Thus, the new subspace L' is now spanned by [m_1, $m_{2,}$..., m_D]. The process keeps on repeating till last batch of frames in source. An intermediate step to update procedure is calculation of set D orthonormal vectors from [m_i, $m_{i+1,}$..., m_{D-i}] using Gramm-Schmidt's method. It makes calculations easy to perform using orthonormal vectors especially during vector comparison.

As explained earlier, subspace update emulates template matching where variance between candidate observation and mean observation allow for tolerance to external changes mainly occlusion, pose and illumination. [18] tackles these challenges by using information from computed subspace L. Subspace is decomposed to constituents as $L = L_1 + L_2$, where L_1 is formed by radial vectors representing mean vectors of a batch and accounts for illumination changes. L_2 represents affine space and models affine space generated by mean vectors.

5.2 Modified Algorithm for GPGPU

The section discusses on converting sequential implementation of tracking algorithm steps to stream processor based implementation. There are two aspects to parallel implementation of a sequential code. First is to redesign original code to take advantage of underlying device architecture, and second is to port portions of original code to gain significant speed up.

The appearance based approach to track object is simplest and robust to variations owing to explicit control on its parameters. Detection is done by comparing a number of portions of images, called samples, occurring near to target object and selecting the sample with closest resemblance to target object using uniform norm as metric. The generation of samples and their comparisons to window containing target

object can be time expensive as the number is scaled. Here GPU was employed to save time. C with CUDA (Compute Unified Device Architecture) was used for implementation. CUDA removes the constraint of converting a general problem to a graphical problem. In other words, implementation does not require mapping to graphics pipeline for execution.

The tracking algorithm is analyzed for possible components of tracking function that can be mapped well to GPU to perform operations in parallel.

The algorithm can be summarized as follows:

1. Sample windows, where attributes of every sample are taken from a Gaussian distribution.
2. Rectify every sample produced in step 1 as a k-by-k matrix and rasterize it to form a vector in R^K image space.
3. Compare each rectified window sample to window of the previous frame containing tracked object. Find L^2 norm square for the same.
4. Choose first half of the window samples of the current frame sorted according to their distance measures as stated in step 3.
5. From the set of window samples selected in 4, evaluate their distances to the subspace using orthonormal bases. This step rejects half of the selected window samples and increases speed of tracking.
6. The window sample with the least distance is closest to tracked object window of the previous frame and hence is the approximate position of tracked object for the current frame.
7. Update the subspace by retaining estimated tracked object position for recent frames.
8. Repeat above steps till the last frame sequence.

Steps 1 to 4 predominantly determine tracking and steps 6 and 7 are useful for providing robustness against pose, illumination variation and occlusion. Among the steps listed, we proceed to identify steps compatible to GPU on the basis of two factors: (i) Data size involved in each step and (ii) Data independence in each step.

On analysis we find steps 1, 2, 3 and 5 qualify. Step 4 that sorts values is possible to implement at GPU but is not efficient except for network sorting. [6] implemented Quick sort on GPU and claim to get good speedups better than Network sorts but not include data setup and transfer time. [18] used Quick sort for step 4 that is not as efficient even if tried to implement its iterative version at GPU. Radix sort is better but slightly complex for floating point values that forms the data and so we delegate sorting to CPU. Figure 5-1 shows schematic state diagram of plan of action.

Another major concern to gain speed from GPU is barrier provided by transfer time and nature of computations. The best strategy is then to combine functions that are called separately in sequence exhibiting parallel nature, prompting to combine steps 1,2,3 and 5 as one function. By doing so I/O based transfer operations were reduced. Figure 5-2 illustrates the idea. The modified tracking algorithm works as follows:

1. CPU Stage: At the first frame, tracking window is initialized and its parameters are noted down. The parameter list includes location <x, y>, dimensions <width, height> and orientation <θ>. This step forms the initialization part of tracking.
2. CPU Stage: In the next frame, S samples are generated using Gaussian distribution with mean as the values of parameters of tracked frame previously, as explained earlier. The stage could be implemented at GPU but we left it to CPU due to less number of elements and low amount of data (running in KB). As an example, for 4000 window samples per frame, data generated would be 4000 * 4 * 5 = 80,000 bytes or approximately 80KB, assuming 1 KB = 1000 bytes.
3. GPU Stage: The generated samples are rectified with respect to target window identified in previous frame. This step is done so as to be fair with each sample (x_i) comparison to identified target window in previous frame. Sample rectification is very important to be done at GPU and is primarily responsible for huge speed gain, owing to high data quantity.
4. GPU Stage: Compute L^2 distance between each sample x_i and local mean vector m as explained in [18]. Here m is vectorized version of window, of previous frame, containing tracked object.

5. CPU Stage: Sort samples x_i, according to their, above computed, distances from local mean. We apply Quick sort for sorting as the best case time is O(NlogN) and worst is $O(N^2)$. Quick sort can be achieved on GPU but code is not simple.

6. GPU Stage: Choose, first half of such x_i, from above sorted list and compute their distances to orthonormal bases U of the subspace L. The orthonormal bases for a vector space are computed using Gram Schmidt's method that is implemented on CPU.

7. CPU Stage: Window W_i corresponding to x_i with minimum distance to subspace L, is the new estimate of target frame and is used for next frame in sequence. This way, subspace L is learnt by incorporating newly estimated information of target frame. Finding minimal of a sequence can be implemented at GPU but then transfer time would be associated and it makes less sense to implement a portion of a function at GPU, then transfer back results to CPU for executing remaining part of the function.

8. CPU Stage: Subspace L is updated by implementing Subspace Update Algorithm to get new set of orthonormal bases and set of batch means.

The Subspace update algorithm was let to remain on CPU as data evaluations were not independent and could be better dealt sequentially - a trademark for CPU.

We again point that copying processed data back to CPU from GPU is time expensive as it employs memory transfer operations, thus wasting precious CPU cycles. Despite being time consuming, speed was achieved as many operations exhibiting data independence were solved in linear time O (1) against O (N) in linear loop. For L^2 distance square measure calculation, we arranged samples as batches where set of batches represented blocks making a 2D grid of blocks and batch size as number of threads within each block. Processing speed of GPU was tested by performing computations at global memory and then at texture memory to find significant speed up of operations. The same is reported in Chapter 6.

As shown in Figure 5-1, the control flows through CPU and GPU at alternate stages. Data alternating between CPU and GPU can prove costly due to in-bound I/O operations. To overcome the problem, some stages requiring GPU computation were

merged. Figure 5-2 illustrates the idea. Sample rectification, L^2 norm computation and partial computation required for computing candidate sample distance from subspace bases were combined as one GPU stage. Schematic flow in Figure 5-2 is more compact than first and gave better results.

The sample rectification step was the most expensive, requiring arithmetic computation. The L^2 computation involved less calculation per rectified sample but expensive in terms of data size involved. Thus, it was logical to combine rectification and L^2 computation. Chapter 6 gives analysis and experimental results of tests conducted.

5.3 Implemented Kernels

Three major functions of original tracking functions were implemented at GPU:

- L^2 Norm Square Calculation.
- Window Sample Rectification.
- Function that reconstructs subspace L.

Tests proved CPU spent more than ninety percent of time to execute above three functions. Figure 5-3 and Figure 5-4 compare implementation of L^2 norm computation at CPU and GPU respectively.

The CPU version function was simple and easy to understand. Time complexity was $O(N^2)$. The GPU version showed O(1) time complexity but space complexity increased. It was affordable to exploit space complexity due to multiple processing cores. Each core gets same copy of instruction. For-loop in GPU version was reduced to a single element computation with thread index controlled by special variables: threadId, blockId and blockDim (see Chapter 3). GPU version of L^2 was slightly complex but worth implementing.

Figure 5-5 summarizes the setup of our new modified algorithm where functions are re-arranged. Figure 5-5 can be treated as work flow view of Figure 5-2.

There are some limitations faced by our algorithm. The limitations are inherited from original algorithm as described in [18]. The thesis does not add or reduce any function from the original list of functions but modifies steps to achieve its goal with speed. The limitations are as follows:

- Drastic illumination change may result in target loss.
- More than two-third occlusion of subject fails the algorithm to track.
- High noise hampers performance.
- Sudden pose change may fail in tracking.

A notable difference between Figure 5-1 and Figure 5-2 is that in former, tracking stages are ported to GPU where as for latter, code was redesigned such that maximum work was done at GPU. Results presented pertain to Figure 5-2.

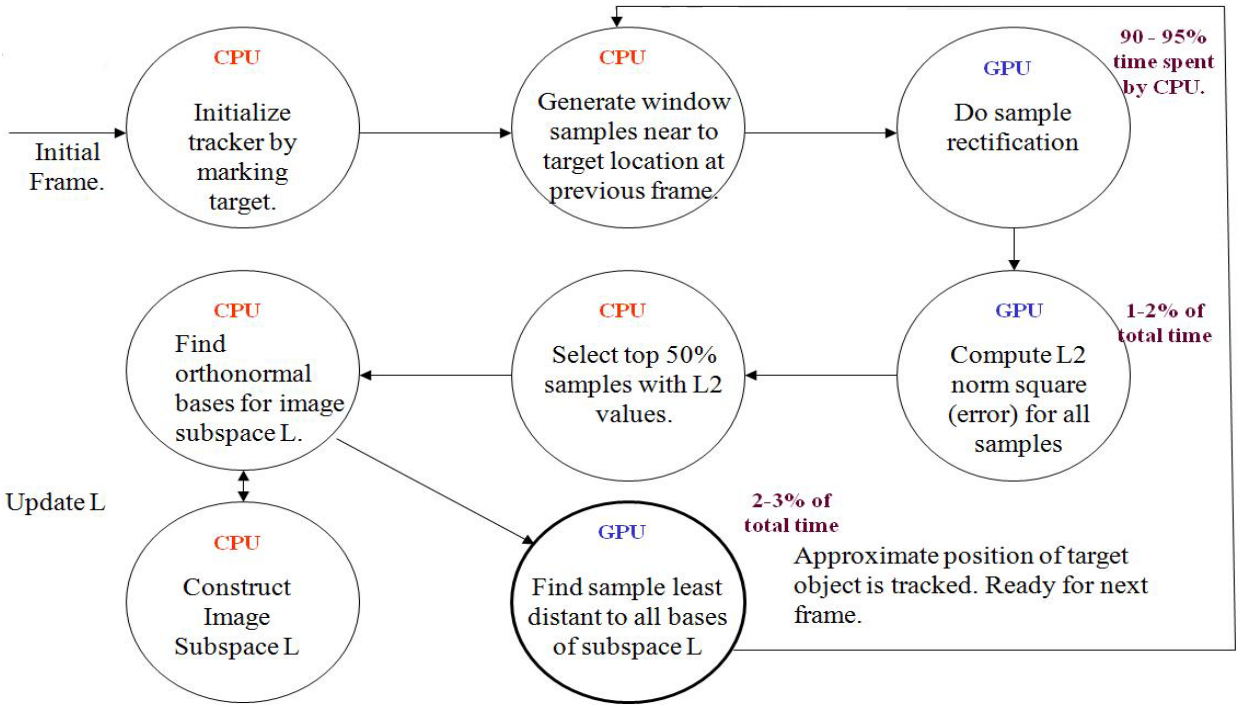

Figure 5-1. State diagram showing tracking algorithm stages executed at CPU and GPU.

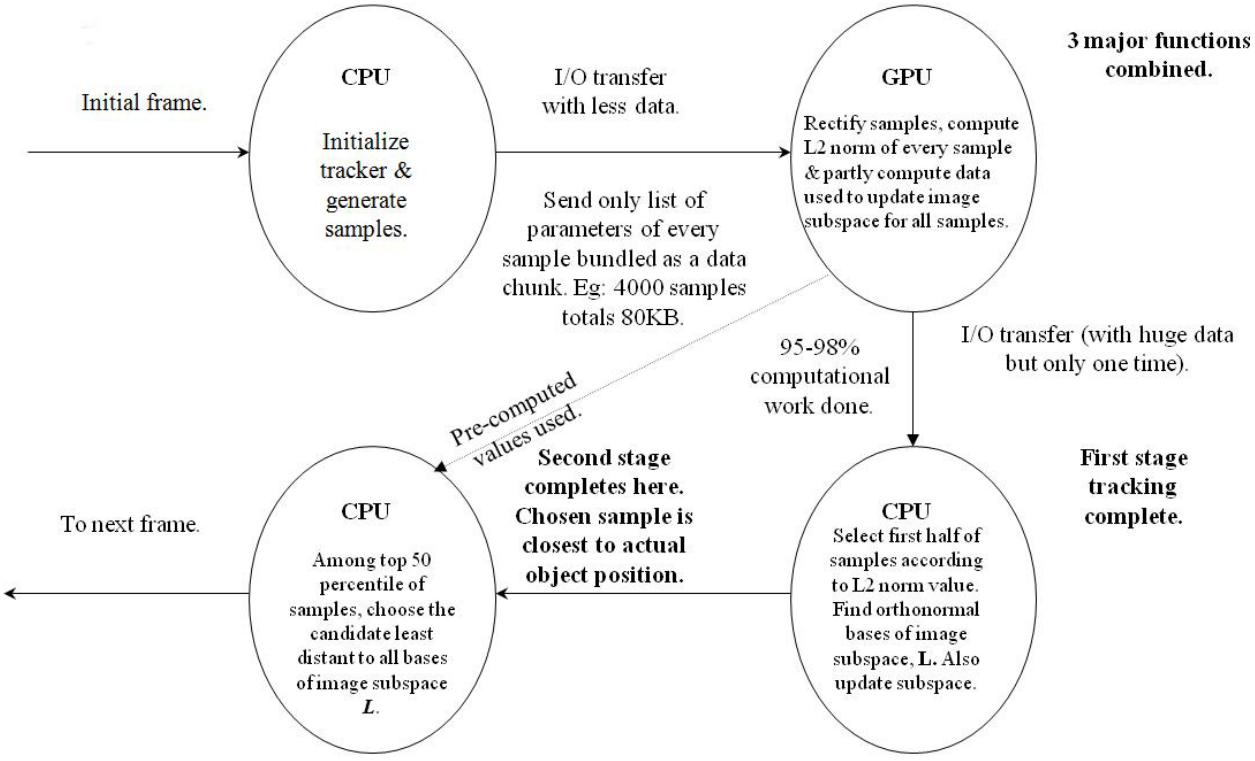

Figure 5-2. Modified state diagram showing reduced number of stages in Figure 5-1.

```
float L2Dis(float *A, float *B, int D)
{
        int i;
        float Result=0.0;
        for( i = 0; i < D; i++ )
                Result += (A[i] - B[i])*(A[i] - B[i]);
        return Result;
}

 for(i=0; i<NumSamples; i++)
 {
    D[i] = L2Dis(SampleImages[i], MeanVec, VecLen);
 }
```

Figure 5-3. L^2 norm computation for CPU.

```
__global__ void L2Dis(float* d_Input , float* d_L2Norm , int NumSamples , int SampleLen)
{
  int N = NumSamples * SampleLen;
  int id = threadIdx.x + blockIdx.x * blockDim.x;
  __shared__ float s[3000];
  __shared__ float temp[361];

  //initialize shared var
  s[blockIdx.x] = 0.0;
  temp[threadIdx.x] = 0.0;
  __syncthreads();
  if(id < N) {
          for(int i = 0 ; i < blockDim.x ; i++) {//copy data from input sample to shared variable
          temp[i] = d_Input[i + blockIdx.x * blockDim.x ]; __syncthreads();
        s[blockIdx.x] += temp[i] * temp[i]; __syncthreads();
      }
   }
   d_L2Norm[blockIdx.x] = s[blockIdx.x]; //copy data to global output variable
}
```

Figure 5-4. L^2 Norm computation for GPU

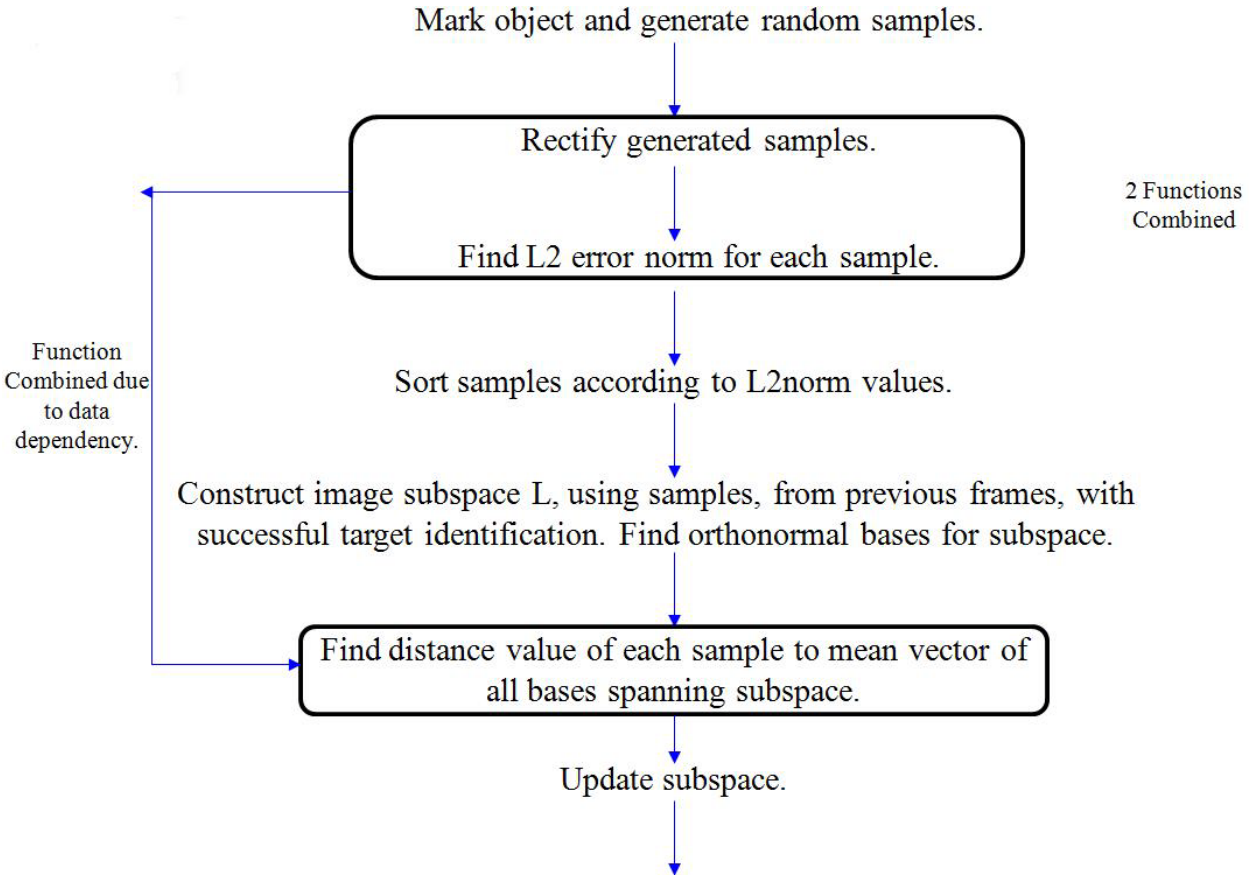

Figure 5-5. Work flow view of modified GPU based tracking algorithm.

CHAPTER 6
RESULTS AND FUTURE WORK

6.1 Experimental Approach

Tests were conducted using NVIDIA GeForce 8800GS as GPU and compared against AMD Athlon Dual Core as CPU. The NVIDIA GeForce 8800GS contains 96 stream processors, arranged as a set of eight cores per multiprocessor, 550MHz as core clock speed, 384MB RAM and 192 bit memory bus width with peak memory bandwidth of 38.4GB/s. Full specification in [17]. The AMD Athlon has 2.01GHz clock speed and 2GB RAM. We implemented two versions of the algorithm at GPU. The code was implemented in CUDA 2.2 with C in Visual Studio 2008 under Microsoft Windows environment. The versions differed in use of different memory models, explained earlier, and modes of transferring data. The CPU code was implemented in Visual C++ 6.0 with MFC as interface and OpenCV used for handling AVI input files.

As discussed in Chapter 3, our implementation used performance metrics to improve performance. In both versions, instruction level optimizations were used. Loop unrolling, faster variants of mathematical operations for trigonometric functions and pre-computing loop invariants were majorly used for instruction level optimization. The first version of our code used global memory for instruction and operand fetches and store operations. The second version used texture memory for reading input frame image data and global memory for store operations.

Shared memory was used sparingly for temporary calculations. Shared memory and constant memories were used for small chunk of data. 24-bit uncompressed AVI files were used as source inputs for image frames to our tracker. Too much use of

shared memory added constraint on GPU due to synchronization requirement for consistency.

Presented below is a detailed performance analysis done on one source file (Figure 6-2), that shows a clip from popular serial Friends. The file was chosen due to its near to homogeneous background composition. This helped our tracker to concentrate on moving object and was not disturbed by external factors. Results for other tested source samples shall be presented next. The file has 100 frames and in AVI file format. Each frame was true color and measured 320 by 240 pixels in dimensions.

6.6.1 Unoptimized Version

Operations were performed at global memory that is slowest of all available memory models refer [32], [34]. A global memory fetch takes between 400 - 600 cycles, hundred times more than a fetch from shared, texture or constant memory models. Global memory was chosen due to its easy access and demanded less code modification. Constant memory was used for storing mean vector representing image subspace. Mean vector was computed as mean of image subspace bases. Table 6-1 shows local speed gain achieved for calculating L^2 norm function, in original algorithm, when ported to GPU. Speed gain is defined as execution ratio between CPU to GPU under same conditions.

The conditions controlling execution of algorithm were:

- Window sample size.
- Number of frames retained by tracker N.
- Frame batch size k.
- CPU-GPU I/O data transfer.
- Image frame size.
- Error tolerance.

The GPU added extra control parameter called number of threads per block under execution configuration. GPU execution configuration was discussed in Chapter 3.

Table 6-1 shows results for local speed gain obtained at porting L^2 norm computation to GPU. At 200 samples, for 19 by 19 sample length, the speed is almost 2.5 times more than that at CPU. The performance gap widens to 4.48 before stabilizing within range of 6 to 7 times. Speedup variations were attributed to random execution time by GPU. Each observation is an average of at least three runs for same parameter values.

Table 6-2, Table 6-3 and Table 6-4 give an idea of overall speed-up of modified algorithm with respect to sequential counterpart on CPU for different window sizes. According to Figure 6-1 and Table 6-2, the overall speed gain at 200 samples per frame neared 6. The ratio increased with increasing number of samples. For 500 samples and above, the CPU showed linear increase in time, and a steep rise in speed gain. The O(N*M) time complexity by CPU is attributed due to linear increase in number of samples per frame (N) and constant length, M, window sample size for L^2 norm computation and sample rectification. The GPU however showed a constant time increase O(1). The O(1) time is attributed to space-complexity exploitation of computing resources inside GPU.

As described in Chapter 3, a GPU is a many-core processor that breaks a for-loop as a single pass execution on each processing core. The speed ratio remained within 20-25 times range. Peak ratio was achieved at 3000 - 3500 samples per frame before a slight dip to 23.36 at 4000. The decline in performance ratio is attributed to constant

increase in I/O bound CPU-GPU data transfer. The result was based on 32-samples per block as GPU execution configuration. Results also show speedup when CPU-GPU I/O operation is discarded. The last column under speedup shows true processing speed achieved for portion of code parallelized. Results in second graph used 32 samples per block as GPU execution configuration. Figure 6-2 presents graph when I/O transfer time is not included.

Table 6-5 shows results on source file with varying background intensity as referred to by Figure 6-6. Tracking parameters were determined through experiments. Best parameter configuration at which tracking was successful was as follows:

Number of previous frames was kept at 20 due to varying intensity values. Rate of updating frame was kept at 5. Batch size of frames was put as 10. Tolerance value was kept at 0.1. Deviation in angular movement of subject was set at 10. As shown in Table 6-6, speed gain varies. The CPU time was recorded multiple times to ensure consistency. GPU results were consistent and showed linear increment. Highest speed gain was achieved for 1000 samples per frame.

Test was also conducted on Football clip (240 frames) (Figure 6-5) where change of illumination and occlusion are involved. As the source involved occlusion with change of illumination, tests were conducted at several configurations. For 32 bit block size, 19 by 19 sample overall speed up (including I/O transfer) was 21. For 500 samples per frame and same parameter configuration, speed up was 10.74. Increasing number of previous frame recorded to 20, 19 by 19 window sample size and 32 bit block, for 200 samples, overall speed gain was 11.25. Football clip was chosen to test GPU's

performance when magnitude of calculations was increased for ensuring robustness to external factors.

For ice hockey clip (351 frames) (Figure 6-6), at 128 block size, 1000 window samples per frame, 21 by 21 window sample size and remaining parameters unchanged, overall speed gain (including time spent on sequential code, execution time on parallelized functions and I/O transfer) reached 9.6 times. Table 6-5 and Table 6-6 presents tracking player's head in ice hockey clip (Figure 6-6). The input file proved to be milder to the football clip (Figure 6-5) as only light intensity changes in background.

6.1.2 Optimized Version

To reduce memory fetch latencies, we used texture memory for storing in-coming frame. Constant memory was used to store mean vector of the bases of image subspace L. Secondly, page-locked memory or pinned memory was used to cut time at CPU-GPU data transfer. We present two tables where texture memory and asynchronous data were incorporated separately for clear analysis. It is obvious to get significant overall speed gain when the two factors are simultaneously incorporated.

Table 6-7 shows rough estimates of time saved by asynchronously transferring data between the two devices. Pinned memory asynchronous transfer reduced transfer time by sixty to seventy percent. For higher number of samples, implying high data, time reduction was significant. Speed gain was improved further. The transfer time difference was tested on NVIDIA GeForce 8800GS at 128 samples per block, as execution configuration. Results, in Tables 6-7, were for data with biggest size in the application; the collective window sample data.

Table 6-8 shows nominal gains made when texture memory was put to use for storing input image frames. Texture memory is fast as data are coalesced and low data-

fetch latency. Maximum speed improvement of 1 ms was seen for sizable data between 3000 - 4000 samples. Texture memory was not used for storing data generated from intermediate computation due to rapidly changing data in each new frame.

The GPU was run using 32 samples per block. Each sample is 361 units long. CPU testing was conducted with tracker application running. To warm up CPU and GPU, the first test result was not considered due to device initialization.
Testing was done at other source input files. Source file showing a football clip (in Figure 6-5) has 240 frames and each frame is 240 by 320, true color. The input file has illumination changes and partial occlusion. The average speedup varied between 10 and 20.

Source file (missile tracking – 227 frames) from high speed camera records was also tested. To track high speed object such as given by Figure 6-3, parameters were changed. Angle parameter was set to 0, number of previous frame to record was set to 3. Mean batch size was kept at 1. Update frame rate was also kept at 1. The reason was due to linear displacement of object. Drastic background changes due to high speed of flying projectile were another reason to change parameter configuration. At 30 by 4 sample size and 32 bit block size, frame rate touched 400 frames per second. Window size was chosen to be 30 by 4 units because object to track is elongated.

6.2 Comparison to other works

Some work has been done to implement tracking on GPU. To the best of our knowledge, we have not come across a visual tracker totally designed at GPU. [23] ports re-weighting stage based on particle filtering and reports a overall speedup between 3 and 4 for single face tracking. The particle filtered weighting stage is reported to gain speed 8 – 10 times at GPU. [23] reports to achieve 40 frames per second n

single face tracking, where as our method reaches close to 400 fps as top speed for 200 samples, 19 by 19 sample size and previous frames at 7. At higher number of samples per frame we also indirectly improved tracking accuracy.

[51] is another KLT feature tracking algorithm where reported top speed is 260fps and 216fps when gain estimation is set. They report their data on GeForce 8800 Ultra. Our implementation on the other hand shows achievable speed touching 400fps for 200 samples for 19 by 19 window sample as presented in tables earlier.

[44] proposes novel way of implementing KLT and SIFT feature extraction for motion tracking and reports 10 – 12 times speedup but for SIFT feature extraction only. [26] is a model based three dimensional tracking system and reports speed of 30fps using GeForce 7800GTX.

6.3 Strategies to follow

Following level of strategies must be kept in mind in trying to make code efficient and get good speed gain at GPU.

1. Macro Level
   - 1-1 Coalesce stages, in an algorithm, where inter dependency is high. Aim should be to delegate expensive and majority portion of tasks to GPU. The bottleneck of I/O transfer between host (CPU) and device (GPU) is compensated by high processing speed of the latter.
   - 1-2 Implement those stages that are required to run at GPU. If possible combine functions to make a single kernel. This reduces chances of transfer operations with GPU memory.

2. Micro Level
   - 2-1 Instruction optimization: [34], [35] and [36] give a detailed discussion on optimized variants of mathematical function that are fastest. The references describe new variant of functions that take less GPU clock cycle. Loop unrolling is advisable for cases where for loop is moderate. Pre-computing loop invariants can save GPU from redundant work. Swizzling is advantageous for instant data swapping and/ or replication. Having sufficient number of thread blocks can appreciable affect performance of GPU.

2-2 Memory optimization: Switching to faster memory models (shared, constant, texture) from global memory improves GPU's performance by an order of magnitude. With availability faster memory models restrictions do apply. For instance shared memory is limited in size and cannot be used for exchanging data between blocks. Unnecessary use of shared memory also demands for synchronization that forces GPU operations to be serialized.

2-3 I/O optimization: Use pinned memory for faster transfer. The only drawback associated with its excessive use is code complexity and less availability of space to CPU. As pinned memory locks a memory page, relocation is not possible by host resulting in less flexibility, especially during fragmentation. Pinned memory transfer is asynchronous and a DMA operation.

6.4 Conclusion and Future Work

The work demonstrates a CPU based tracking algorithm gaining significant speed at GPU by exploiting space-complexity of the problem. The trick is to identify components that work on huge data exhibiting high degree of independence. Modified algorithm is implemented on different memory models. Care is to be taken to minimize data transfer between GPU and CPU. Operations where CPU is as fast as GPU should remain at CPU for it adds unnecessary I/O transfer between CPU and GPU. I/O transfer using asynchronous data transfer should be used sparingly, as DMA (Direct Memory Access) transfer can impede host system's performance due to cycle stealing.

Parallel programming is manipulation of space-complexity to reduce time complexity of a problem. The tracking approach used by [18] is though not state-of-the-art, can holistically track an object. Comparing a candidate window sample to bases of linear subspace provides little degree of robustness. The accuracy of algorithm can be improved further by incorporating advanced detection methods like Steerable filters for edge detection and methods for prediction probability. The L^2 distance matching is a crude version of probabilistic state prediction. We did not test the speed of our modification to multiple face tracking.

Table 6-1. Speed Ratio between CPU and GPU for L2 Norm computation for Figure 6-5.

Number of Samples (N)	Total CPU execution time for full tracking function	Percentage of CPU code parallelized	CPU time spent in executing L2 norm computation (in ms)[1]	Frame Rate (In fps)	GPU time spent in executing L2 norm computation (in ms)[1]	Frame Rate (In fps)	Local speed gain (S)
200	35.4965	2.57	0.9149	1092.99	0.3669	2725.39	2.42
500	229.4860	0.99	2.2807	438.44	0.4667	2142.49	4.48
1000	355.5232	1.29	4.5946	217.64	0.6783	1474.26	6.77
1500	467.9995	1.42	6.8714	145.52	1.1839	844.59	5.80
2000	763.90	1.20	9.1927	108.78	1.4951	668.84	6.14
2500	917.17	1.34	12.3625	80.88	1.8062	553.61	6.84
3000	1100.00	1.25	13.8282	72.31	2.1250	470.57	6.50
3500	1307.74	1.26	16.5545	60.40	2.3648	422.85	7.00
4000	1362.90	1.39	19.0607	52.46	2.5115	398.16	7.58

[1] Sample size is 19 by 19. Execution configuration for GPU was 32 samples per block. Source file has 100 frames with each frame of dimension 320 by 240.

Table 6-2. Full application speed ratio between CPU and GPU for window sample size 19 by 19 for Figure 6-5.

Number of Samples (N)	Total CPU execution time for full tracking function (in ms).[1]	CPU execution time spent on code not parallelized (sequential code)[2] (in ms)	GPU execution time for portion of code parallelized. (in ms)[3]	GPU – CPU data transfer time. (in ms). (Average reading)	Overall application speedup when I/O data transfer is considered. S_1[4]	Overall application speedup when data transfer is not considered. S_2[5]	Application speedup only at portion of original code parallelized. S_3[6]
200	35.4965	0.4867	2.8039	2.5306	6.098	10.78	12.48
500	229.4860	1.2271	5.010	3.9320	22.56	36.79	45.55
1000	355.5232	2.5257	9.3450	4.5711	21.62	29.94	37.77
1500	467.9995	3.7946	14.1936	5.1000	20.26	26.01	31.06
2000	763.9066	5.1460	20.5408	5.5200	24.47	29.73	36.93
2500	917.17207	9.2500	23.8608	6.1636	23.35	27.69	38.18
3000	1100.0035	7.8619	28.1200	6.6688	25.79	30.57	38.83
3500	1307.7404	9.3463	33.3864	7.2063	26.18	30.60	38.88
4000	1362.9084	11.5018	39.1189	7.7048	23.36	26.92	32.86

[1] Each observation recorded is average of three runs of execution. Source file has 100 frames.

[2] Sequential code is constituted by quick sort, sample parameter generation function and function for finding orthonormal bases and image subspace update.

[3] GPU execution configuration used 32 samples per block.

[4] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion + GPU-CPU data transfer time.

[5] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion.

[6] Speed ratio is calculated by taking total CPU time on parallelized portion to total GPU execution time on parallelized portion.

Table 6-3. Full application speed ratio between CPU and GPU for window sample size 21 by 21for Figure 6-5.

Number of Samples (N)	Total CPU execution time for full tracking function (in ms).[1]	CPU execution time spent on code not parallelized (sequential code)[2] (in ms)	GPU execution time for portion of code parallelized. (in ms)[3]	GPU – CPU data transfer time. (in ms). (Average reading)	Overall application speedup when I/O data transfer is considered. S_1[4]	Overall application speedup when data transfer is not considered. S_2[5]	Application speedup only at portion of original code parallelized. S_3[6]
200	90.5070	0.4827	3.6292	2.6083	13.46	22.01	24.80
500	232.8503	1.2293	6.4959	4.5802	18.92	30.14	35.84
1000	352.97650	2.5075	12.5265	5.6924	17.03	23.47	28.17
1500	572.0246	3.8079	19.0825	6.3115	19.58	24.98	29.97
2000	794.0643	5.1505	27.2064	6.3570	20.51	24.54	29.18
2500	931.7751	6.8094	31.6413	7.0343	20.48	24.23	29.44
3000	1071.7841	9.0131	37.9725	7.7089	19.59	22.81	28.22
3500	1268.6380	9.5200	43.5696	8.4120	20.62	23.89	29.11
4000	1398.8064	10.6979	51.6788	9.1014	19.56	22.42	27.06

[1] Each observation recorded is average of three runs of execution. Source file has 100 frames.

[2] Sequential code is constituted by quick sort, sample parameter generation function and function for finding orthonormal bases and image subspace update.

[3] GPU execution configuration used 32 samples per block.

[4] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion + GPU-CPU data transfer time.

[5] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion.

[6] Speed ratio is calculated by taking total CPU time on parallelized portion to total GPU execution time on parallelized portion.

Table 6-4. Full application speed ratio between CPU and GPU for window sample size 24 by 24 for Figure 6-5.

Number of Samples (N)	Total CPU execution time for full tracking function (in ms).[1]	CPU execution time spent on code not parallelized (sequential code)[2] (in ms)	GPU execution time for portion of code parallelized. (in ms)[3]	GPU – CPU data transfer time. (in ms). (Average reading)	Overall application speedup when I/O data transfer is considered. S_1[4]	Overall application speedup when data transfer is not considered. S_2[5]	Application speedup only at portion of original code parallelized. S_3[6]
200	107.1707	0.4929	4.5046	2.6705	13.97	21.44	23.68
500	250.0508	1.2292	6.5447	4.5355	20.31	32.16	36.15
1000	406.6268	2.5217	13.8699	5.4368	18.62	24.80	29.13
1500	680.0704	3.9253	19.7705	6.3275	22.65	28.69	34.19
2000	833.8979	5.2900	28.9216	7.1714	20.15	24.37	28.83
2500	1043.6181	6.7103	32.8125	8.0625	21.93	26.40	31.60
3000	1136.1170	8.0684	39.9645	9.0936	19.88	23.65	28.22
3500	1396.7756	9.3086	43.7366	9.9097	22.18	26.33	31.75
4000	1542.6705	10.6297	53.3908	10.7453	20.63	24.09	28.69

[1] Each observation recorded is average of three runs of execution. Source file has 100 frames.

[2] Sequential code is constituted by quick sort, sample parameter generation function and function for finding orthonormal bases and image subspace update.

[3] GPU execution configuration used 32 samples per block.

[4] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion + GPU-CPU data transfer time.

[5] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion.

[6] Speed ratio is calculated by taking total CPU time on parallelized portion to total GPU execution time on parallelized portion.

Table 6-5. Full application speed ratio between CPU and GPU for window sample size 21 by 21 for Ice Hockey clip (Figure 6-4)

Number of Samples (N)	Total CPU execution time for full tracking function (in ms).[1]	CPU execution time spent on code not parallelized (sequential code)[2] (in ms)	GPU execution time for portion of code parallelized. (in ms)[3]	GPU – CPU data transfer time. (in ms). (Average reading)	Overall application speedup when I/O data transfer is considered. S_1[4]	Overall application speedup when data transfer is not considered. S_2[5]	Application speedup only at portion of original code parallelized. S_3[6]
200	32.17285	0.6199	3.7712	2.3733	04.756	07.32	08.53
500	55.88387	1.3577	6.9047	4.1771	04.492	06.76	08.09
1000	194.4942	2.6279	12.9842	4.6407	09.603	12.45	14.97
2000	214.6808	5.2372	25.5169	5.9373	05.850	06.98	08.41
3000	356.9619	7.8909	41.7109	7.2794	06.275	07.19	08.55

[1] Each observation recorded is average of three runs of execution. Source file has 351 frames.

[2] Sequential code is constituted by quick sort, sample parameter generation function and function for finding orthonormal bases and image subspace update.

[3] GPU execution configuration used 128 samples per block.

[4] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion + GPU-CPU data transfer time.

[5] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion.

[6] Speed ratio is calculated by taking total CPU time on parallelized portion to total GPU execution time on parallelized portion.

Table 6-6. Full application speed ratio between CPU and GPU for window sample size 19 by 19 for Ice Hockey clip

Number of Samples (N)	Total CPU execution time for full tracking function (in ms).[1]	CPU execution time spent on code not parallelized (sequential code)[2] (in ms)	GPU execution time for portion of code parallelized. (in ms)[3]	GPU – CPU data transfer time. (in ms). (Average reading)	Overall application speedup when I/O data transfer is considered. S_1[4]	Overall application speedup when data transfer is not considered. S_2[5]	Application speedup only at portion of original code parallelized. S_3[6]
200	29.20456	0.6044	03.0024	2.2827	04.958	08.097	09.72
500	050.06479	1.3698	05.0825	3.8824	04.844	07.759	09.58
1000	171.97055	2.5580	09.7466	4.4680	10.253	13.976	17.64
2000	203.3582	5.2970	20.9178	5.4871	06.414	07.757	09.72
3000	252.0958	8.0038	28.9434	6.6273	05.785	06.823	08.70

[1] Each observation recorded is average of three runs of execution. Source file has 351.

[2] Sequential code is constituted by quick sort, sample parameter generation function and function for finding orthonormal bases and image subspace update.

[3] GPU execution configuration used 32 samples per block.

[4] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion + GPU-CPU data transfer time.

[5] Speed ratio is calculated by taking total CPU time to total GPU execution time on parallelized code + CPU time on non-parallelized portion.

[6] Speed ratio is calculated by taking total CPU time on parallelized portion to total GPU execution time on parallelized portion.

Table 6-7. Estimate of time saved using pinned memory asynchronous transfer

Samples per frame	Data size (in MB)	Normal memory allocation synchronous transfer time (in ms)	Pinned memory (page-locked) asynchronous Transfer Time (in ms)	Transfer speedup
200	0.2888	0.4639	0.0988	4.6937
500	0.7220	1.4730	0.2344	6.2826
1000	1.444	2.5795	0.4619	5.5838
1500	2.166	2.8396	0.6881	4.1261
2000	2.888	4.0262	0.9142	4.4040
2500	3.610	4.4456	1.1394	3.9014
3000	4.332	6.0020	1.3665	4.3922
3500	5.054	6.1810	1.5934	3.8789
4000	5.776	7.6517	1.8214	4.2009

Table 6-8. Execution time when texture memory is used.

Samples per frame	GPU execution time using texture memory (in ms)	GPU execution time using global memory (in ms)	Time reduced (in ms)
200	2.5971	2.8039	0.2067
500	4.6303	5.0103	0.3799
1000	8.9282	9.3450	0.4168
1500	13.8334	14.1936	0.3062
2000	20.1594	20.5408	0.3814
2500	23.2683	23.8608	0.5924
3000	27.2325	28.1200	0.8874
3500	32.2636	33.3864	1.1227
4000	38.1535	39.1189	0.9654

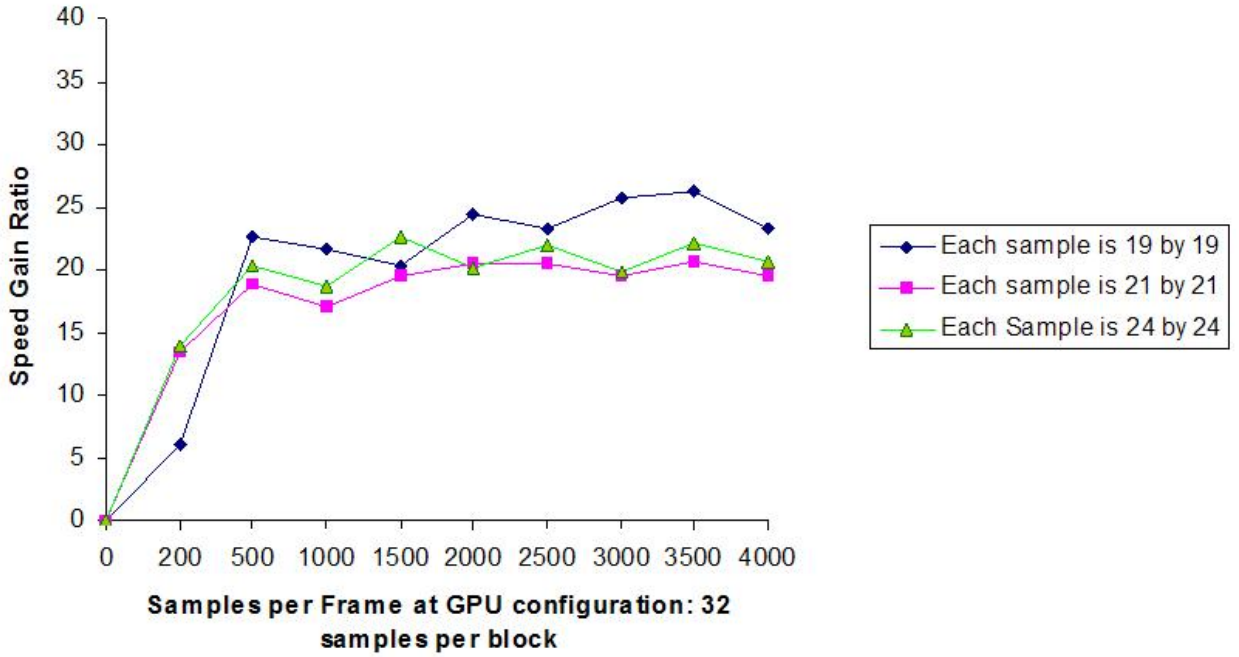

Figure 6-1. Application speedup when CPU-GPU data transfer is included.

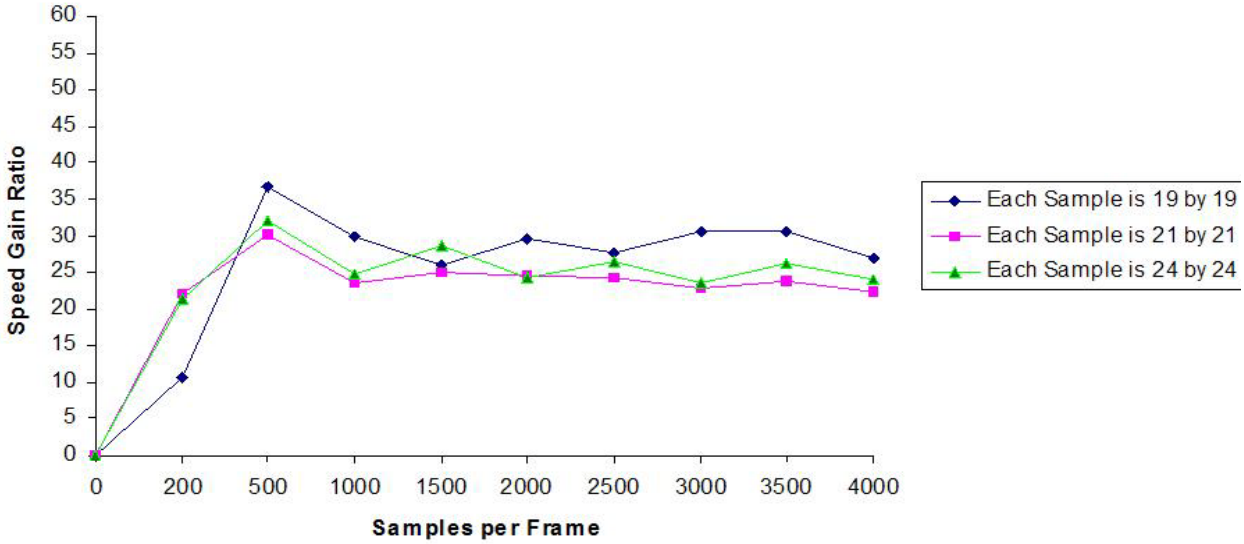

Figure 6-2. Application speedup when CPU-GPU data transfer is not considered.

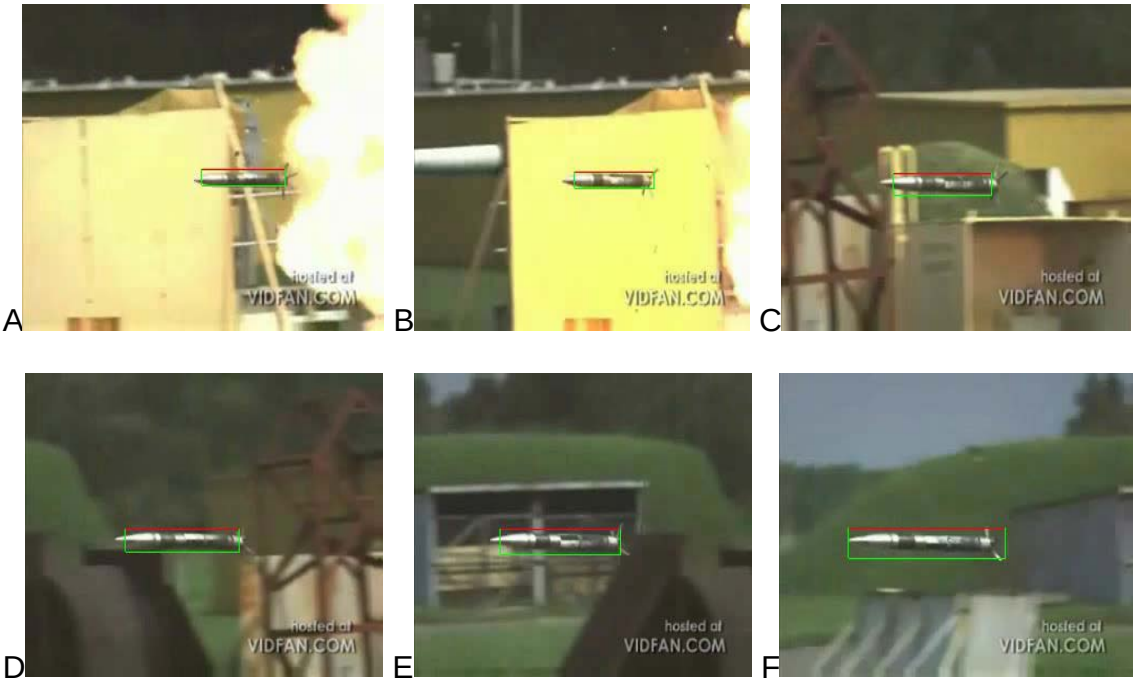

Figure 6-3. Some tracked frames from a high speed camera missile tracking sequence on GPU. Source www.youtube.com. Accessed on November 2009.

Figure 6-4. Some tracked frames from series Friends. Source www.youtube.com

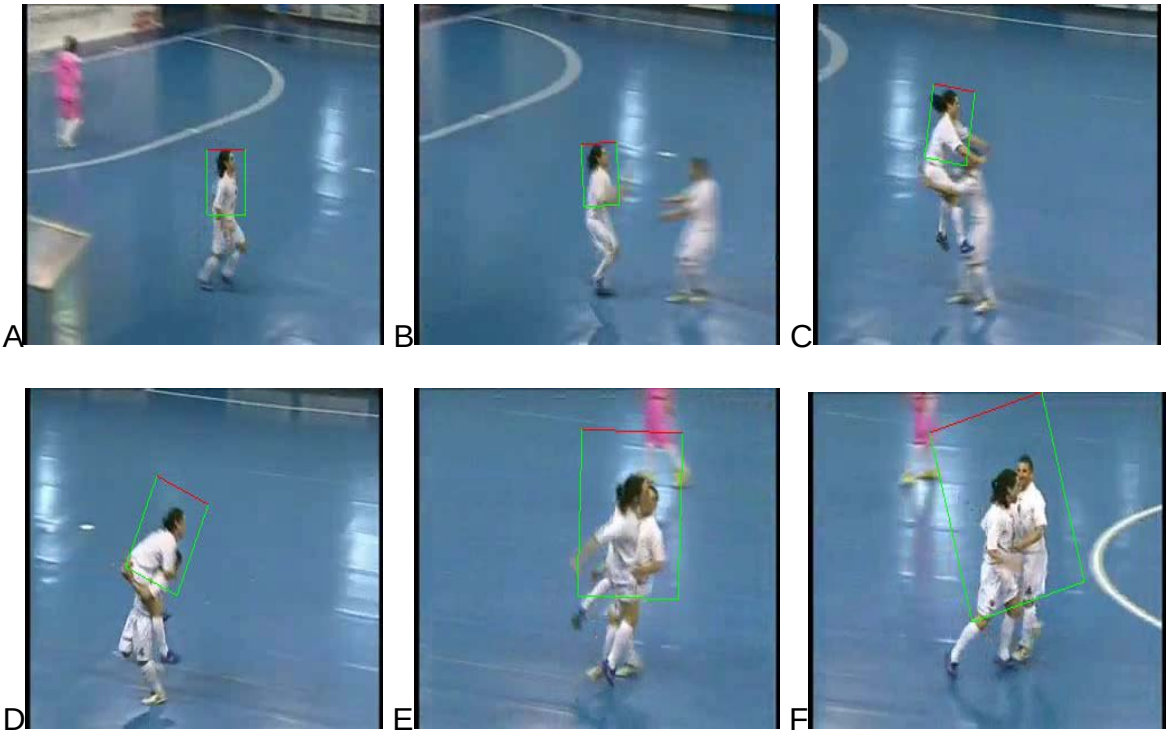

Figure 6-5. Some tracked frames from a football clip. Source www.youtube.com. Accessed on November 2009.

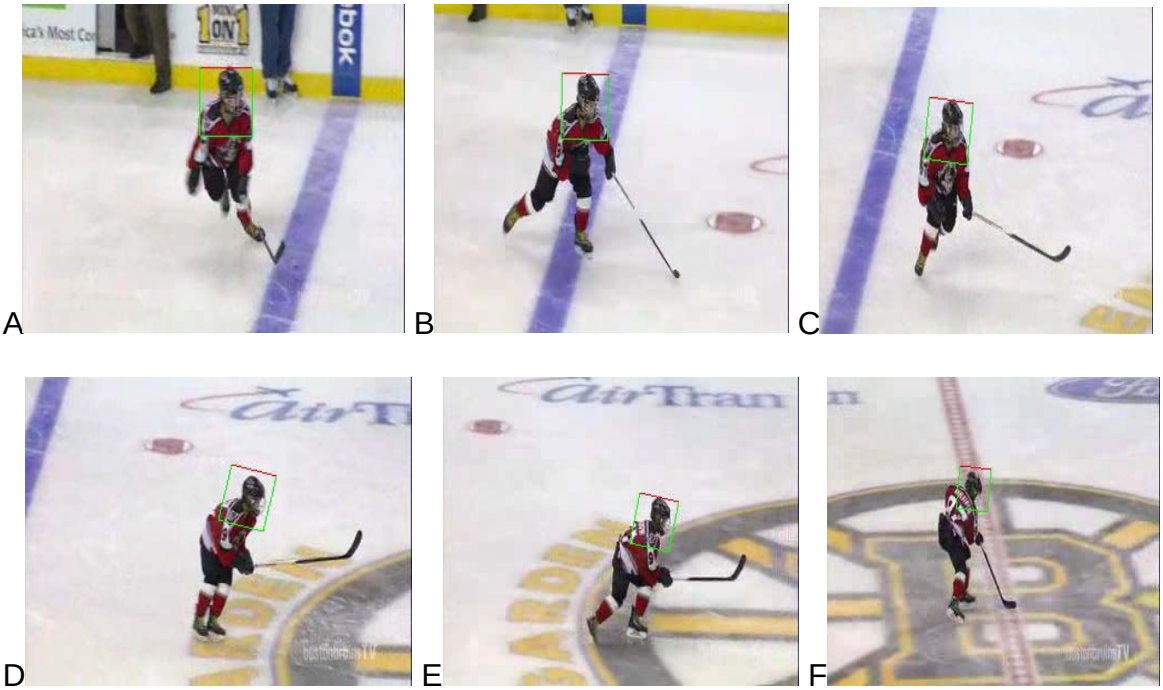

Figure 6-6. Some tracked frames of ice hockey player. Source www.youtube.com Accessed on November 2009.

REFERENCES

1. Arulampalam, S., Maskell, S. &, Gordon, N & Clapp T. - A tutorial on Particle Filters for on-line non-linear/non-gaussian bayesian tracking. *In IEEE Transactions on Signal Processing*, 2001, volume 50, pages 174 – 188.

2. Bach.M et al. (2008) – Porting a Kalman filter based track fit on NVIDIA. GSI Scientific Computing Report.

3. Bay, H, Tuytelaars, T & Gool, L (2006). "SURF: Speeded Up Robust Features. *Proceedings of the 9th European Conference on Computer Vision*, Springer LNCS volume 3951, part 1. Pages 404—417.

4. Birchfield, S. & Rangarajan, S - Spatiograms versus Histograms. In proceedings *IEEE Conf. of Computer Vision and Pattern Recognition* 2005, volume 2 pages 1158-1163.

5. Briggs F.A & Hwang K (1993). *Advanced Computer Architecture and Parallel Processing* .McGraw Hill.

6. Cederman, D. & Tsigas, P. (2008) – A Parallel Quicksort Algorithm for Graphic Processors. A technical report at Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden. Pages 246-268. doi:10.1007/978-3-540-87744-8.

7. Comaniciu, D., Ramesh, V. and Meer P, (2000) - Real-time tracking of non-rigid objects using mean shift. In proceedings *IEEE Conf. of Computer Vision and Pattern Recognition* 2000, volume 2 pages 142 – 149.

8. Fatahalian, K. & Houston, M.(2008) A closer Look at GPUs. Communications of the ACM, doi: 10.1145/1400181.1400197.

9. Feldman, M. (2009) NVIDIA Takes GPU Computing to the Next Level. HPC Wire Resource document. http://www.hpcwire.com/features/62800147.html. Accessed October 2, 2009.

10. Fernando, R. (2004). *GPU Gems Programming Techniques, Tips and Tricks for Real-Time Graphics.* Addison-Wesley. ISBN-10:0321228324.

11. Fernando, R. & Pharr, M. (2005). *GPU Gems 2 Programming Techniques.* for High Performance Graphics and General Purpose Computation. Addison-Wesley ISBN-10: 0-321-33559-7.

12. Forsyth & Ponce (2005). *Compute Vision Modern Approach.* Pearson Education.

13. General Purpose Computing Using Graphics Hardware. Resource document. http://www.gpgpu.org . Accessed November 18, 2009.

14. Giles, M. (2008) High Performance Computing for Finance. Resource document. http://people.maths.ox.ac.uk/~gilesm/hpc/. Accessed November 01, 2009.

15. Göddeke, D. (2009) GPGPU tutorials. Resource document http://www.mathematik.tu-dortmund.de/~goeddeke/gpgpu/index.html. Accessed November 01, 2009.

16. Harding, S. & Banzahf, W.- Fast genetic programming on GPUs. *In Proceedings of the 10th European Conference on Genetic Programming* 2007. Volume 4445, pages 90 – 101.

17. Hennessy, J.L. & Patterson, D. (2007). *Computer Architecture: A Quantitative Approach* 4th Ed. Morgan Kaufmann Publishers.

18. Ho, J, Lee, K.C., Yang, M.H. & Kriegman D- Visual Tracking Using Learned Linear Subspaces. In proceedings *IEEE Conf. of Computer Vision and Pattern Recognition* 2004, volume 2 pages 33-42.

19. Isard, M. & Blake, A. (1998) - CONDENSATION - conditional density propagation for visual tracking. *International Journal of Computer Vision*.

20. Jepson, A.D. &.Fleet, D.J and El-Maraghi, T.F – Robust Online Appearance Models for Visual Tracking. In *IEEE Transactions of Pattern Analysis and Machine Intelligence* 2003, volume 25 number 10 pages 1296 – 1311.

21. Kass, M, Witkin, A. & Terzopoulos, D (1987) - Active Contours. *International Journal of Compute Vision*. Volume 1(4) pages 321 – 331.

22. Lowe, D.G, (2004). Distinctive Image Features from Scale-Invariant Key points. *International Journal of Computer Vision* 60 (2): pages 91–110. doi:10.1023/B:VISI.0000029664.99615.94.

23. Lozano, O.M. & Otsuka, K. (2008). Real-time Visual Tracker by Stream Processing, Simultaneous and Fast 3D Tracking of Multiple Faces in Video Sequences by Using a Particle Filter. *Journal of Signal Processing System.*

24. Luebke, D & Humphreys, G. (2007). How GPUs Work. *IEEE computer society press*. (Vol. 40(2) pp. 96-100).

25. Matas, J., Chum, O., Urban, M. & Pajdla, T. (2002). "Robust wide baseline stereo from maximally sextremum regions". British Machine Vision Conference. pp 384-393.

26. Michel, P., Chestnut, J., Kagami, S., Nishiwaki, K., Kuffner, J., Kanade, T. (2007). *International Conference on Intelligent Robots and Systems.* Pages 463-469, doi:10.1109/IROS.2007.4399104.

27. Mikolajczyk, K. & Schmid, C. (2004). Scale and affine invariant interest point detectors. *International Journal of Computer Vision* 60 (1): pp 63–86. doi:10.1023/B:VISI.0000027790.02288.f2.

28. Nejhum, S.M.S, Ho,J. & Yang, M.H. -Visual Tracking with Histograms and Articulating Bocks. In proceedings *IEEE Conf. of Computer Vision and Pattern Recognition* 2008, pages 1-8.

29. Nguyen, H. (2007). *GPU Gems 3.* Addison-Wesley. ISBN: 0321515269.

30. NVIDIA Inc. *CUDA Zone* web forum. Resource document. http://forums.nvidia.com/index.php . Accessed on November 01, 2009.

31. NVIDIA Inc. (2009) OpenCL Programming for the CUDA Architecture. Resource document. http://www.nvidia.com/object/cuda_develop.html. Accessed November 01, 2009.

32. NVIDIA Inc. (2009) NVIDIA CUDA Documentation. Resource document. http://www.nvidia.com/object/cuda_develop.html. Accessed November 01, 2009.

33. NVIDIA Inc. *Physx*. Resource document. http://www.nvidia.com/object/physx_new.html . Accessed on November 01, 2009.

34. NVIDIA Inc. (2009) NVIDIA CUDA Programming Guide. Resource document. http://www.nvidia.com/object/cuda_develop.html. Accessed November 01, 2009.

35. NVIDIA Inc. (2009) NVIDIA CUDA Reference Manual. Resource document. http://www.nvidia.com/object/cuda_develop.html. Accessed November 01, 2009.

36. OpenGL® - The Industry's Foundation for High Performance Graphics. Opengl.org web forum. Resource document. http://www.opengl.org/. Accessed October 28, 2009.

37. Oh, K.S.K. & Jung, K. (2004) – GPU implementation of neural networks. At *Pattern Recognition Society.* Volume 37, issue 6, pages 1311-1314. doi:10.1016/j.patcog.2004.01.013.

38. Pande lab Stanford University. Folding Proteins at home – distributed computing. Resource document. http://folding.stanford.edu/. Accessed November 01, 2009.

39. Qizhi, Y., Chen, C.& Pan, Z (2005) – Parallel Genetic Algorithms on Programmable Graphics Hardware. Advances in Natural Computation. doi:10.1007/11539902.

40. Raytchev, B. & Murase, H. – Unsupervised Face Recognition from Image Sequences Based on Clustering with Attraction and Repulsion. In proceedings *IEEE Conf. of Computer Vision and Pattern Recognition* 2001, volume 2 pages 25-30.

41. Richardson, A. & Gray, A. (2008) *Utilization of the GPU architecture for HPC*. The HPCx technical report. EPCC, The University of Edinburgh, James Clerk Maxwell Building, Edinburg, U.K. Resource document http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0808.pdf. Accessed on October 02. 2009.

42. Rost, R.J. (2006) *OpenGL® Shading Language* Edition 2. Addison-Wesley Professional.

43. Shreiner, D., Woo M., Neider, J. & Davis, T. (2005) *The OpenGL® Programming Guide- The official guide to learning OpenGL® Version 2.0.* Fifth Edition. Addison-Wesley Professional.

44. Sinha, S.N., Frahm, J.M., Pollefeys, M. & Genc, Y (2006). *GPU-based Video Feature Tracking and Matching.* A technical report - In *Workshop on Edge Computing Using New Commodity Architectures*. Department of Computer Science, UNC Chapel Hill. Resource document. http://cs.unc.edu/~ssinha/pubs/Sinha06TechReport.pdf. Accessed November 01, 2009.

45. Viola, P. & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Proc. of the IEEE computer society conference on computer vision and pattern recognition* (Vol. 1, pp. 511-518).

46. Viola, P. & Jones, M. (2004). Robust real-time face detection. *International Journal of Computer Vision*, 57(2), 137-154.

47. Wolfe, M. (2008) Compilers and More: A GPU and Accelerator Programming Model. HPC wire. Resource document. http://www.hpcwire.com/features/Compilers_and_More_GPU_Architecture_and_Applications.html. Accessed October 2, 2009.

48. Wolfe, M. (2008) Compilers and More: GPU Architecture and Applications. HPC wire. Resource document. http://www.hpcwire.com/features/Compilers_and_More_GPU_Architecture_and_Applications.html. Accessed October 2, 2009.

49. Wolfe, M. (2008) Compilers and More: Accelerating High Performance. HPC wire. Resource document. http://www.hpcwire.com/features/Compilers_and_More_Accelerating_High_Performance.html. Accessed November 01, 2009.

50. Wolfe, M. (2008) How Should We Program GPGPUs. Linux Journal. Resource document. http://www.linuxjournal.com/article/10216. Accessed October 2, 2009.

51. Zach, C., Gallup, D. & Frahm, J.M. (2008) – Fast gain-adaptive KLT Tracking on the GPU. In proceedings *IEEE Conf. of Computer Vision and Pattern Recognition* pages 1-7.doi:10.1109/CVPRW.2008.4563089.

BIOGRAPHICAL SKETCH

Alok Whig received his B.Tech. (Honors) degree in Computer Engineering from Maharishi Dayanand University Rohtak (Haryana) India in 2002. Before coming to the University of Florida, he worked as a software engineer at Newgen Software Technologies Limited, Delhi(India). Alok completed his Master of Science in 2009. His research interests are Image Processing with special emphasis on Document Processing Computer Vision, Computer Graphics and High Performance Computing. He is writing a journal for Journal of Signal Processing titled Stream Processor Based Real-Time Visual Tracking Using Object Appearance Approach.

Alok is currently seeking employment in his area and plans to pursue his doctoral studies later on.