A
DISSERTATION
On

"Acceleration of Key Frame Extraction and Tracking Algorithms through General Purpose GPU Computing Using OpenCL Programming Framework"

Thesis submitted in partial fulfillment of the requirement for the degree of

MASTER OF TECHNOLOGY
IN
VLSI DESIGN AND EMBEDDED SYSTEM

Submitted By: **-HITESH GARG**
University Roll No: (03/VLS/2k10)

Under the Guidance of

Prof. ASOK BHATTACHARYYA
Department of Electronics & Communication Engineering
Delhi Technological University, Delhi.
&
Dr. KAUSHIK SAHA
Principal Member Technical Staff
STMicroelectronics Pvt. Ltd. Greater Noida, India

DEPARTMENT OF ELECTRONICS &COMMUNICATION ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY, DELHI
(FORMERLY DELHI COLLEGE OF ENGINEERING)

2010 -2012

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGG
DELHI TECHNOLOGICAL UNIVERSITY, DELHI
(FORMERLY DELHI COLLEGE OF ENGINEERING)

CERTIFICATE

This is certified that the thesis work entitled **"Acceleration of Key Frame Extraction and Feature Tracking Algorithm Through General Purpose GPU Computing using OpenCL Programming Framework "** is bonafide work carried by **HITESH GARG (Roll No: 03/VLS/2k10)** in partial fulfillment for the award of degree of Master of Technology in VLSI Design and Embedded System of the Delhi Technological University, Delhi during the year 2010-2012. The project report has been approved as it satisfied the academic requirements in respect of thesis work prescribed for the Master of Technology Degree.

Prof. ASOK BHATTACHARYYA
Department of Electronics &
Communication Engineering
Delhi Technological University, Delhi.

Dr. KAUSHIK SAHA
Principal Member Technical Staff
STMicroelectronics Pvt. Ltd., Greater
Noida, India

ACKNOWLEDEMENT

I would like to dedicate my work to my parents and my brothers, for their constant prayers and encouragement throughout my life and showing me the silver lining in the dark clouds.

It has been rightly said, “*We are built on the shoulders of others*”. For everything I have achieved, the credit goes to my academic guide, Professor Asok Bhattacharyya, Department of Electronics & Communication Engineering, Delhi Technological University. His motivation, support and guidance during this thesis work have been invaluable.

I would like to sincerely thank my supervisor from industry Dr. Kaushik Saha (Principal Member Technical Staff, ST Microelectronics Pvt. Ltd. India) for providing me the opportunity to work on the project providing the facilities at ST Microelectronics Lab at Greater Noida, India. His constant guidance and invaluable suggestions throughout the project, specially at critical junctures has led to the successful completion of this project.

I am very thankful to Prof. Rajeev Kapoor (H.O.D. Department of Electronics & Communication, Delhi Technological University) and to Prof. S. Maji (Dean IRD, Delhi Technological University) who have allowed me to do project under the Guidance of Professor Asok Bhattacharyya in collaboration with ST Microelectronics India.

I wish to express my heartfelt thanks to my colleagues and friends who supported me in all endeavors I had during thesis work in ST Microelectronics. It has been very enlightening and enjoyable experience to work with all of them.

Last but not the least I express my gratitude to the almighty for keeping me in good health and spirits throughout my thesis work.

HITESH GARG
M.Tech (VLSI Design & Embedded System)
University Roll No: 03/VLS/2k10

ABSTRACT

Modern processor architectures have embraced parallelism as an important pathway to increased performance. Now, Central Processing Units (CPUs) improve performance resulted by adding multiple cores. Graphics Processing Units (GPUs) have also evolved from fixed function rendering devices into programmable parallel processors. As today's computer systems often include highly parallel CPUs, GPUs and other types of processors, to take full advantage of these heterogeneous processing platforms, OpenCL (Open Computing Language) provides the new way of computing. OpenCL plays a significant role in emerging interactive graphics applications which integrates general parallel computing algorithms with graphics rendering pipelines. Here GPU computing is applied on General Purpose applications that are Key Frame Extraction and Tracking Algorithms with the help of OpenCL.

In order to retrieve a particular piece of information in a video, of late, Video summarization, aimed at reducing the amount of data that must be examined and that also becomes an essential task in applications of video analysis and indexing. Generally, a video summary is a sequence of still or moving images, with or without audio. Our work is mainly based on acceleration of one such algorithm that utilizes visual summary using still images, called key frames, extracted from the video. Here advantages of still images is that it can summarize the video content in more rapid and compact way, so users can grasp the overall content more quickly from key frames than by watching a set of video sequences. In our case we optimized the pre-processing algorithms for image refinement using Frequency Selective Weighted Median Filter (FSWM) and feature extraction using histogram calculation to accelerate the Key Frame Extraction (KFE) algorithm. The optimization is done through general purpose GPU computing using OpenCL programming framework.

Other part of our work is related to the acceleration of the feature tracking algorithms. As the ability to reliably detect and track human motion is a useful tool for higher-level applications, such as image analyzer that rely on visual input, interacting with human activities are at the core of many problems in intelligent systems, such as human-computer interaction and robotics. Our work focuses on how to speed up the process of KLT (Kanade-Lucas-Tomasi) tracking and how to utilize advantages of FAST (Features from Accelerated Segmented Test) algorithm in the KLT tracking. The algorithm (FAST and KLT) selects the features that are optimal for tracking and keeps the track of these features.

ABBREVIATIONS

GP	General-Purpose
GPU	Graphic Processing Unit
OpenCL	Open Compute Language
CPU	Central Processing Unit
CU	Compute Unit
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
SPMD	Single Process Multiple Data
SM	Streaming Multiprocessor
PE	Processing Element
SM	Streaming Multiprocessor
API	Application Program Interface
NDRange	N- Dimensional Range
DSP	Digital Signal Processors
KFE	Key Frame Extraction
FSWM	Frequency Selective Weighted Median Filter
KLT	Kanade-Lucas-Tomasi
FAST	Features from Accelerated Segmented Test

Contents

LIST OF FIGURES

LIST OF TABLES

Chapter-1

INTRODUCTION

1.1 Objective

This work is intended to improve the applicability of KFE and KLT algorithms to real time scenarios using advantages of GPGPU computing. In this work we shall consider parallelizing the iterative components in the implementation of these algorithms over GPGPU. Real time applications are often served using embedded devices. The GPU architecture - unlike multiprocessing systems is expected to serve as good candidate for parallelization because of its availability in embedded devices.

1.2 Video Summarization

The growing interest of consumers in the acquisition of and access to visual information has created a demand for new technologies to index and retrieve multimedia data. Very large databases of images and videos require efficient algorithms that enable fast browsing and provide access to the information. In the case of videos, in particular, much of the visual data offered is simply redundant, and we must find a way to retain only the information strictly needed for functional browsing and querying [1].
Video summarization, aimed at reducing the amount of data that must be examined in order to retrieve a particular piece of information in a video, is an essential task in video analysis and indexing applications. Video summary is basically a sequence of still images (creation of the visual summary using still images, called key frames extracted from the video) or moving images (when video summarization is achieved using moving images usually called video skimming), with or without audio. Users can grasp the overall content more quickly from key frames than by watching a set of video sequences. In both the approaches images must preserve the overall contents of the video with a minimum of data.

One improvement regards the possibility to quickly understand the content of the video and select more efficiently what the user is seeking, without having the necessity of looking all the content.

1.3 Tracking algorithm

In principle, the stream of images produced by a moving camera allows the recovery of both the shape of the objects in the field of view, and the motion of the camera. The problem of computing the motion in an image is known as finding the optical flow of the image or feature tracking. A feature, or a point of interest, is a point or a set of points where an algorithm can look and follow the motion through frames. There are several ways to select the features: based on brightness and colors or based on corners and edges detection [2]. Depending on the algorithm that we choose, it will determinate in each way the features are selected.

To track features there are essentially two important steps. The first one is to decide which features to track, and the second one is the tracking in itself. There are a variety of well- under stood techniques for doing so, but the Kanade- Lucas- Tomasi method stands out for its simplicity and lack of assumptions about the under lying image [3].

1.4 Organization of Report

- ✓ Chapter 2: provides the information regarding need of parallel computing as well as use of GPGPU computing for Parallelism. This chapter also gives the introduction to OpenCL, the language used to program the GPU for General Purpose computing.

- ✓ Chapter 3: provides the basic theory required to understand the different algorithms, which we have to optimize. Such algorithms are KFE, KLT and FAST etc.

- ✓ Chapter 4: describes the methodology used for optimization of said algorithms. It provides the information of the hardware and software used for our work. It gives the information of profiling results applied to the code.

- ✓ Chapter 5: gives the results achieved during this work.

- ✓ Chapter 6: Concludes our work with the future scope of work that can be done for further performance improvements.

Chapter-2

PARALLELISM

2.1 Introduction to Parallel Computing [4]

Traditionally, software has been written for serial computation:

- ✓ To be run on a single computer having a single Central Processing Unit (CPU).
- ✓ A problem is broken into a discrete series of instructions.
- ✓ Instructions are executed one after another.
- ✓ Only one instruction may execute at any moment in time.

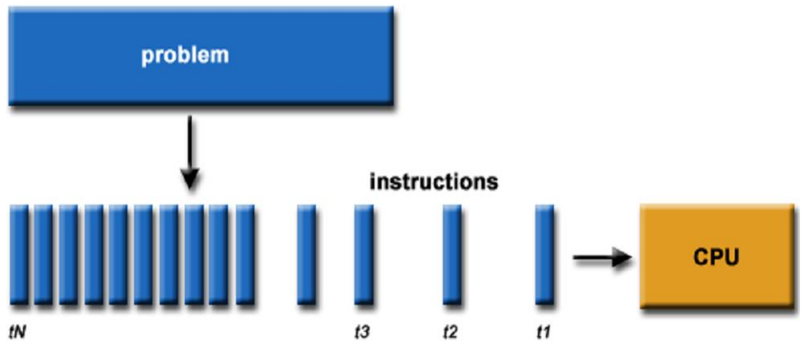


Figure 2.1: *Sequential execution of instruction on CPU*

In the simplest sense, *Parallel Computing* is the simultaneous use of multiple compute resources to solve a computational problem:

- ✓ To be run using multiple CPUs
- ✓ A problem is broken into discrete parts that can be solved concurrently
- ✓ Each part is further broken down to a series of instructions
- ✓ Instructions from each part execute simultaneously on different CPUs

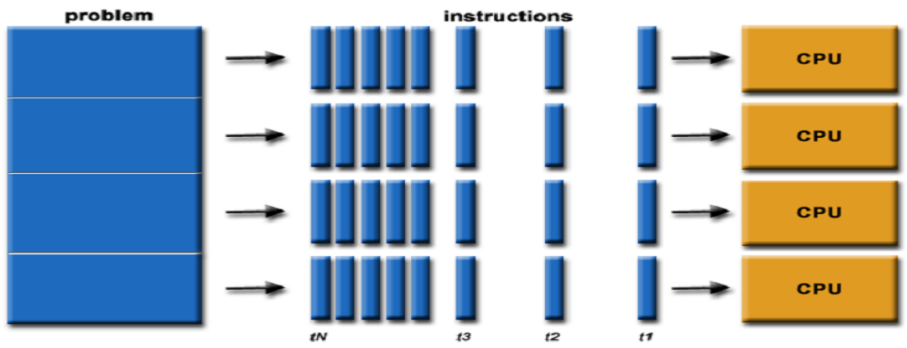


Figure 2.2: *Parallel execution of instructions on CPUs*

2.1.1 Advantages of parallel computing

- ✓ **Save time and/or money:** by using more resources at a task will shorten it's time to completion, with potential cost savings. Parallel computers can be built from cheap, commodity components.
- ✓ **Solve larger problems:** Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
- ✓ **Provide concurrency:** A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

Current computer architectures are increasingly relying upon hardware level parallelism to improve performance:

- ✓ Multiple execution units
- ✓ Pipelined instructions
- ✓ Multi-core

As we have seen that parallel computing is the present and future of computing. Future microprocessor development efforts will continue to concentrate on adding cores rather than increasing single-thread performance. So GPUs that are high performance multi-core processors can be used to accelerate a wide variety of applications using parallel computing, instead of CPUs.

2.2 Parallelism through GPUs

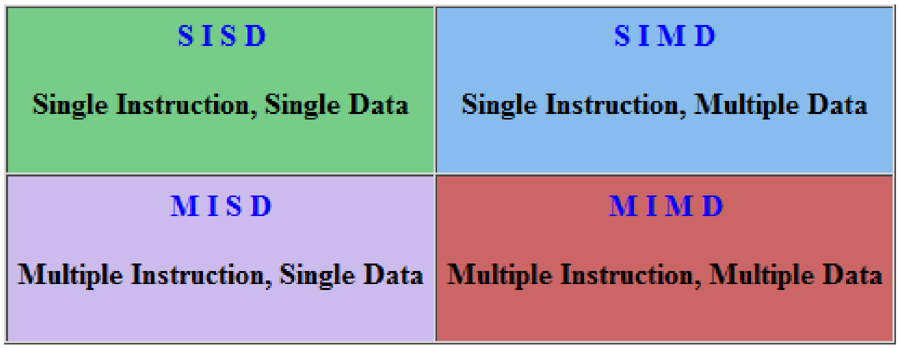
The highly parallel graphics processing unit (GPU) is rapidly gaining maturity as a powerful engine for computationally demanding applications such as- *Key Frame Extraction, Feature Selection and Tracking Algorithms.* The GPU's performance and potential offer a great deal of promise for today's computing systems, because of its different architecture and programming model than most other single-chip processors.
The GPU is designed for a particular class of applications with the following characteristics-
• **Computational requirements are large**: - A real-time application requires billions of pixels per second, and each pixel requires hundreds or more operations. GPUs must deliver an enormous amount of compute performance to satisfy the demand of complex real-time applications.

• **Parallelism is substantial**: - The graphics pipeline is well suited for parallelism.

2.2.1 Classification of Processor Architecture [4]

There are different ways to classify processor architecture. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.

Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of Instruction and Data. Each of these dimensions can have only one of two possible states: Single or Multiple.
The matrix below defines the 4 possible classifications according to Flynn:



SISD Single Instruction, Single Data	**SIMD** Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	**MIMD** Multiple Instruction, Multiple Data

Figure2.3: *Classification of Processor Architecture*

1. Single Instruction, Single Data (SISD):

A serial (non-parallel) computer, **Single Instruction**: Only one instruction stream is being acted on by the CPU during any one clock cycle. **Single Data:** Only one data stream is being used as input during any one clock cycle. It has Deterministic execution.

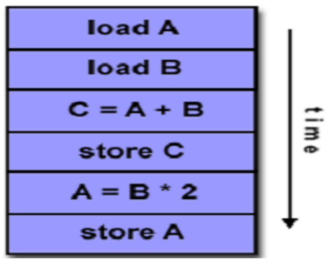


Figure2.4: *Example of SISD*

This is the oldest and even today, the most common type of computers. Examples: older generation mainframes, minicomputers and workstations; most modern day PCs.

2. Single Instruction, Multiple Data (SIMD):

A type of parallel computer **Single Instruction:** All processing units execute the same instruction at any given clock cycle, **Multiple Data**: Each processing unit can operate on a different data element.

It is best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing. Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

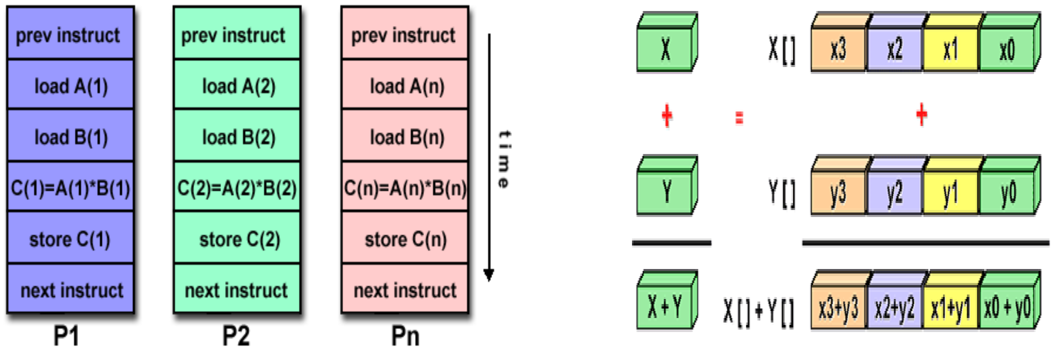


Figure2.5: *Example of SIMD*

3. Multiple Instruction, Single Data (MISD):

A type of parallel computer **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams. **Single Data:** A single data stream is fed into multiple processing units.

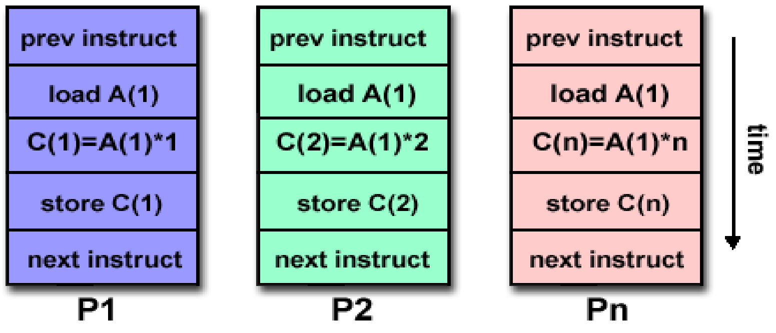


Figure2.6: *Example of MISD*

4. Multiple Instruction, Multiple Data (MIMD):

A type of parallel computer **Multiple Instruction:** Every processor may be executing a different instruction stream **Multiple Data:** Every processor may be working with a different data stream.

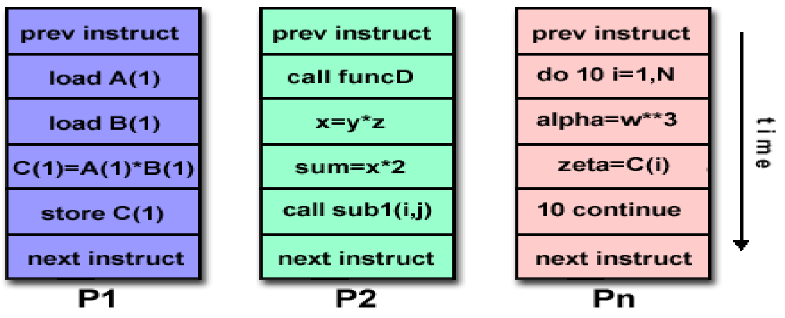


Figure2.7: *Example of MIMD*

Execution can be synchronous or asynchronous, deterministic or non-deterministic. Currently the most common type of parallel computer - most modern supercomputers fall into this category.
Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
Note: many MIMD architectures also include SIMD execution sub-components

2.2.2 GPU Computing [5]

Now that we have seen the hardware architecture of the GPU, we turn to its programming model.

A. The GPU Programming Model

The programmable units of the GPU follow a single instruction multiple-data (SIMD) programming model. For high performance, the GPU processes many elements (threads) in parallel using the same program (as discussed in section 2.2.1). Each element (threads) is independent from the other elements (threads), and elements cannot communicate with each other. Each element can operate on 32-bit integer or floating-point data with a reasonably complete general-purpose instruction set. Elements can read data from a shared global memory and can write data to shared global memory.
Most of programming model is well suited to sequential programs, as many elements can be processed sequential manner. Code written in this manner is single instruction, single data (SISD). As for GPU programming model, programs have become more complex, it allows different elements to take different paths through the same program, leading to the more general SIMD model.
For development of GPU as a General-Purpose (GP) computing engine it is important, the advancement of the programming model and programming tools. The perfect balance between Low-level access to the hardware, for getting good performance and High-level programming languages and tools, for flexibility and productivity is a big challenge, for GPU vendors.

B. General-Purpose Computing on the GPU

General Purpose calculations on Graphics Processing Units (GPGPU) is a term that refers to using the graphics processing unit (GPU) for general purpose calculations instead of graphics rendering. Today GPUs are very powerful devices for faster computations. The time has passed when they were only usable for graphics purposes. Many efforts to use that calculation power are centered on the term “GPGPU” or “GPU Computing”.

GPGPU computing applications are structured in the following way.

1. The programmer directly defines the computation domain of interest as a structured grid of threads.
2. An SPMD (SIMD) general-purpose program computes the value of each thread.

3. The value for each thread is computed by a combination of different math operations and both read accesses from and write accesses to global memory. Here same buffer can be used for both reading and writing, to allow more flexible algorithms.
4. The resulting buffer in global memory can then be used as an input in future computation, which decreases the CPU-GPU memory transfer.

OpenCL and CUDA are the mostly used for writing general purpose programs that execute on GPUs without the need to map their algorithms onto a 3D graphics API such as OpenGL or DirectX.
We have used OpenCL for writing general purpose programs to port on GPUs OpenCL is an open industry standard for programming a heterogeneous environment of CPUs, GPUs and other computing devices organized into a single platform.
When Apple and Khronos Group made OpenCL as a multi platform standard they had one very big and strong rival - CUDA from nVidia corp. Some differences between CUDA and OpenCL are listed below.

Table2.1: *Differences between CUDA and OpenCL* [6]

1. Based on Terminology

OpenCL Terminology	CUDA Terminology
Work-item	Thread
Work-group	Thread block
Local memory	Shared memory
Private memory	Local memory, Register
Compute Unit (CU)	Streaming Multiprocessor (SM)
Processing Element (PE)	Streaming Processor (SP)
barrier()	__syncthreads()
cl_kernel	CUfunction
cl_program	CUmodule

2. Based on Features

Feature	OpenCL	CUDA
Compilation Method	Online + Offline	Offline only
Mathematical Precision	Well Defined	Undefined
Math Libraries	Defined Standard	Proprietary
CPU Support	OpenCL CPU Device	No CPU Support
Native Task Support	Task parallel compute model	No native thread support
Extension Mechanism	Defined Mechanism	Proprietary
Vendor Support	Industry- Wide support	NVIDIA only

2.3 OpenCL – Portable Parallelism

The OpenCL is an open and royalty-free parallel computing API that allows GPU's and other coprocessors to work with the CPUs to provide additional raw computing power.

Thus OpenCL is an open industry standard for programming a heterogeneous environment of CPUs, GPUs and other computing devices organized into a single platform. Due to its broad industry support, OpenCL has the potential to become the de facto software for portable multi-core and many-threaded applications.

The OpenCL standard was suggested by Apple and created by non-commercial Khronos Group, which has created own standards.

2.3.1 Heterogeneous computing

The use of various types of computational units is called heterogeneous computing. A computational unit can be a CPU or a GPU or a special purpose processing unit (such as DSPs). As given in the definition of OpenCL it provides heterogeneous environment.

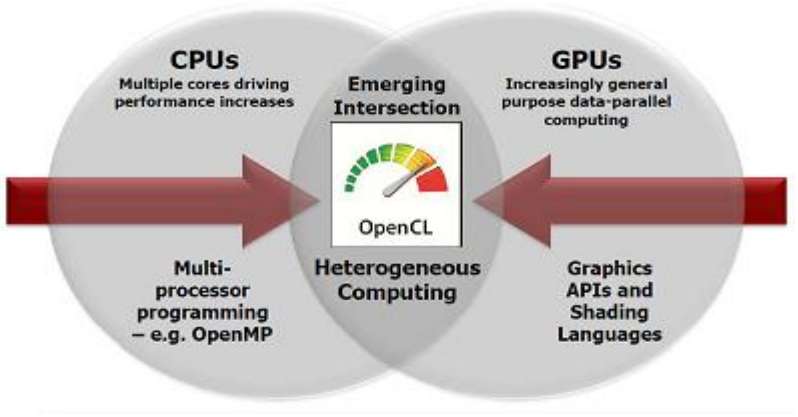

CPUs
Multiple cores driving performance increases
Emerging Intersection
GPUs
Increasingly general purpose data-parallel computing
OpenCL
Multi-processor programming – e.g. OpenMP
Heterogeneous Computing
Graphics APIs and Shading Languages

Figure 2.8 *Heterogeneous computing through OpenCL*

2.3.2 The OpenCL Framework [7]

The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system.

The framework contains the following components:

- ✓ **OpenCL Platform layer:** The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts, through device query.
- ✓ **OpenCL Runtime:** The runtime allows the host program to manipulate contexts once they have been created.
- ✓ **OpenCL Compiler:** The OpenCL compiler creates program executables that contain OpenCL kernels. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism. OpenCL has well defined IEEE 754 numerical accuracy for all floating point operations and a rich set of built-in functions.

2.3.3 The Thought behind OpenCL

The big idea behind OpenCL is a portable execution model that allows a kernel to execute at each point in a problem domain. A kernel is a function or a part of the program that runs on GPU. It is identified by the __kernel qualifier applied to any function defined in a program. Kernels can operate in either a data-parallel or task-parallel fashion.

To describe the core ideas behind OpenCL, we will use a hierarchy of models:

- ✓ Platform Model

- ✓ Memory Model
- ✓ Execution Model
- ✓ Programming Model

1. Platform Model

The Platform model for OpenCL shown in figure 2.9 consists of a host connected to one or more OpenCL devices. An OpenCL device is a collection of one or more compute units (CUs) which are further composed of one or more processing elements (PEs). Computations on a device occur within the processing elements. The OpenCL application submits commands from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units or as SPMD units. SPMD instructions are mainly executed on CPUs while SIMD instructions executed on Vector processors such as a GPU or vector unit in a CPU.

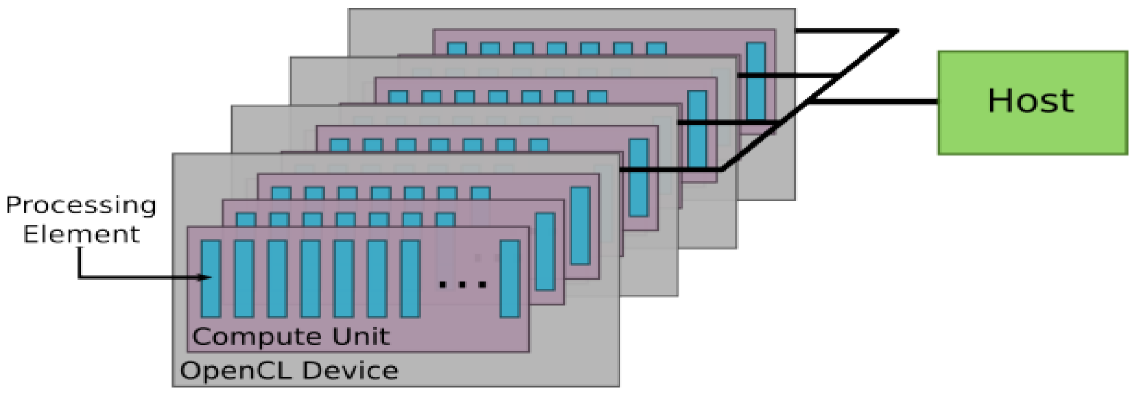


Figure 2.9 *Platform Model*

2. Execution Model

Execution of an OpenCL program occurs in two parts:

- ✓ **kernels** that execute on one or more OpenCL devices and
- ✓ **A host program** that executes on the host.

The OpenCL execution model is based on the parallel execution of a computational *kernel* over a 1-D, 2-D, or 3-D grid, or *NDRange* (“N-Dimensional Range”).The host program defines the context for the kernels and manages their execution.

Following are some core OpenCL terms:

Devices: OpenCL device to execute kernels

Work item: Kernel instances, called Work-Item. In CUDA these are known as threads. This enables to parallelize the execution of the kernels.

Kernel: the code for a work item, that execute on one or more OpenCL devices.

Program: A collection of kernels and other functions.

Context: The environment within which work items executes, which includes devices and their memories and command queues.

Work Group: Work items organized into work groups.

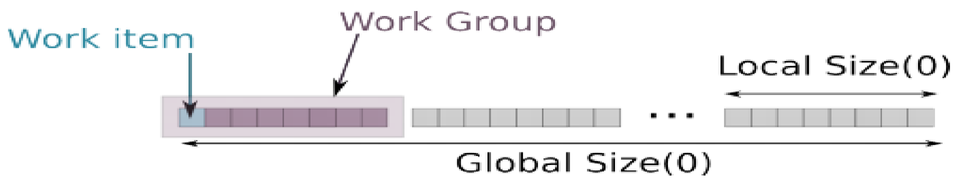


Figure2.10: *1D Index Space*

The core of the OpenCL execution model is defined by how the kernels execute. When a kernel is submitted for execution by the host, an index space is defined as shown in figure 2.10. Work-item executes for each point in this index space (1-D or N-D as shown in figure 2.11) and is identified by its point in the index space, which provides a global ID for the work-item.

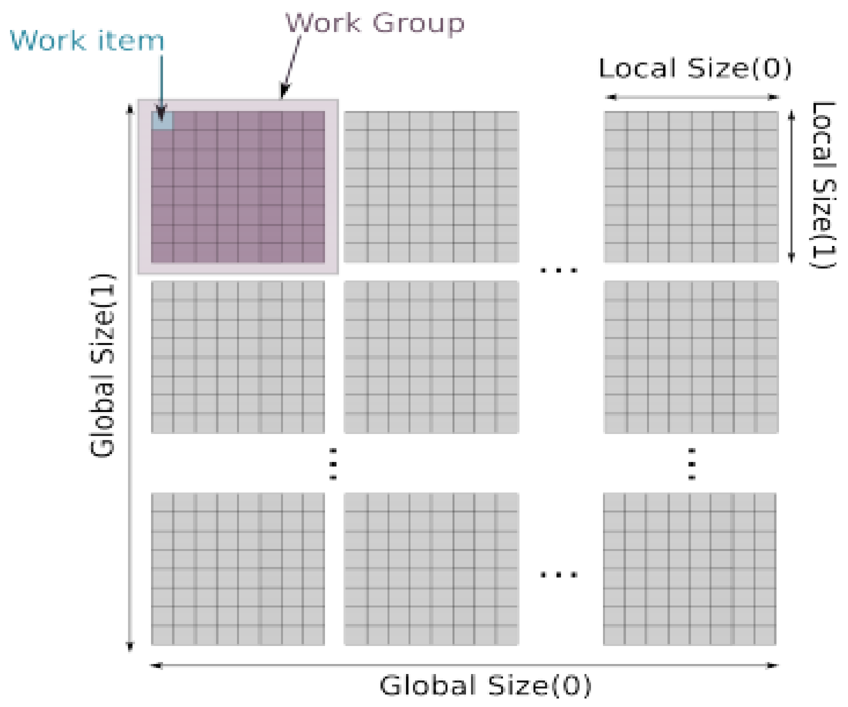


Figure2.11: *2D Index Space*

Example: processing a 1024 x 1024 image:
Global Size (0) = Global Size (1) = 1024
1 kernel execution per pixel =>1,048,576 total kernel executions

Each work-item executes the same code but the specific execution pathway through the code and the data can be different for each work-item. Work-items are organized into work-groups. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items.
Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID.
Inside the kernel, global coordinates are found by calling get_global_id (index), where index is 0, 1, or 2 depending on the dimensionality of the grid. Coordinates local to the work-group are found via get_local_id (index). The number of dimensions in use is found with get_work_dim ().

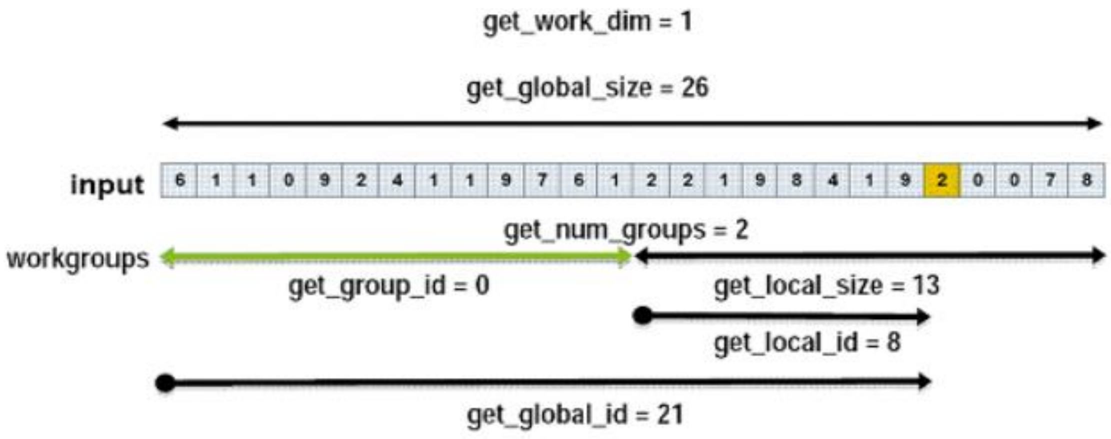


Figure2.12: *Identification of work group and work-item*

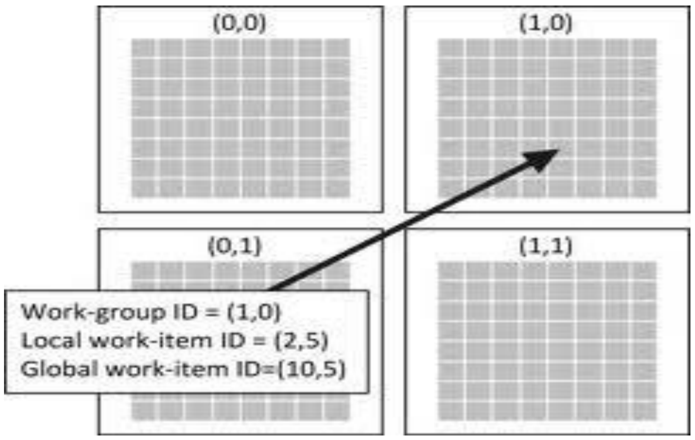


Figure2.13: *2D view of NDRange showing work group and work-item IDs*

An NDRange is defined by an integer array of length N specifying the extent of the index space in each dimension. Each work-item's global ID and local ID are N-dimensional tuples. The global ID components are values in the range from zero to the number of elements in that dimension minus one. A complete 3D view of NDRange is shown in figure 2.14

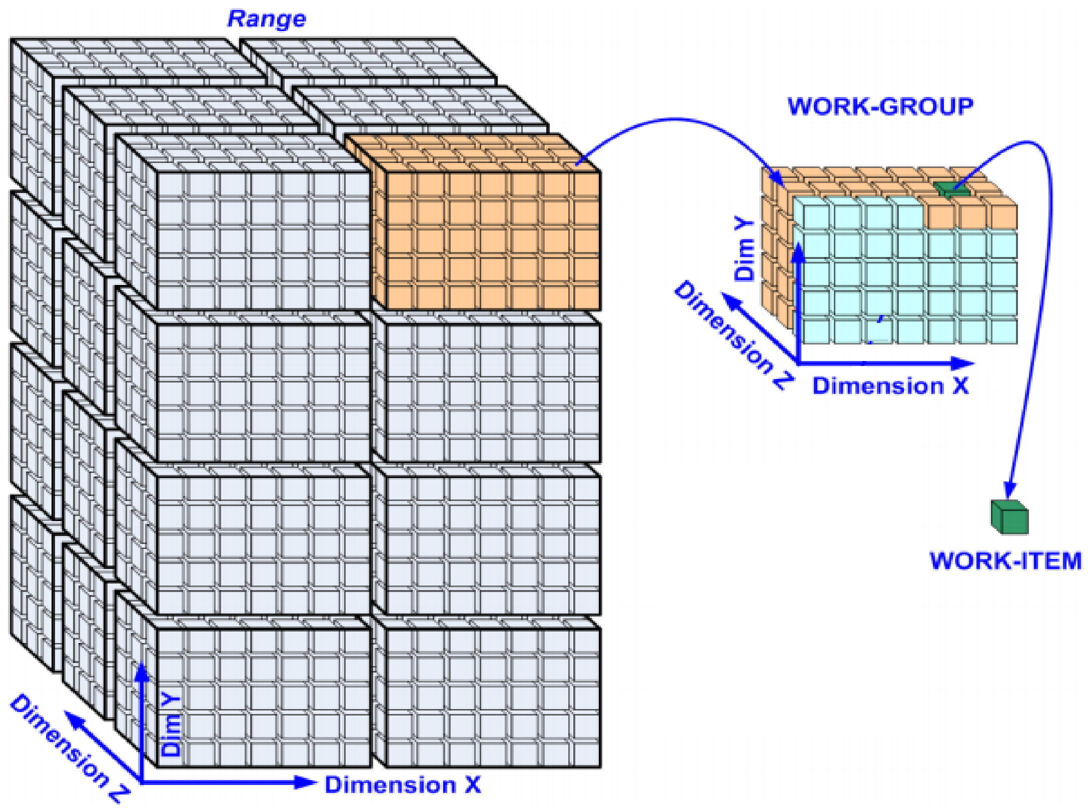


Figure2.14: *3D view of NDRange showing work group and work-item*

3. Memory Model

Work-item(s) executing a kernel have access to four distinct memory regions:

Global Memory: Accessible to all work items and the host. This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.

Constant Memory: Visible to all workgroups, read-only. A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

Local Memory: A memory region local (shared) to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device.

Private Memory: A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

Host Memory: Host-accessible

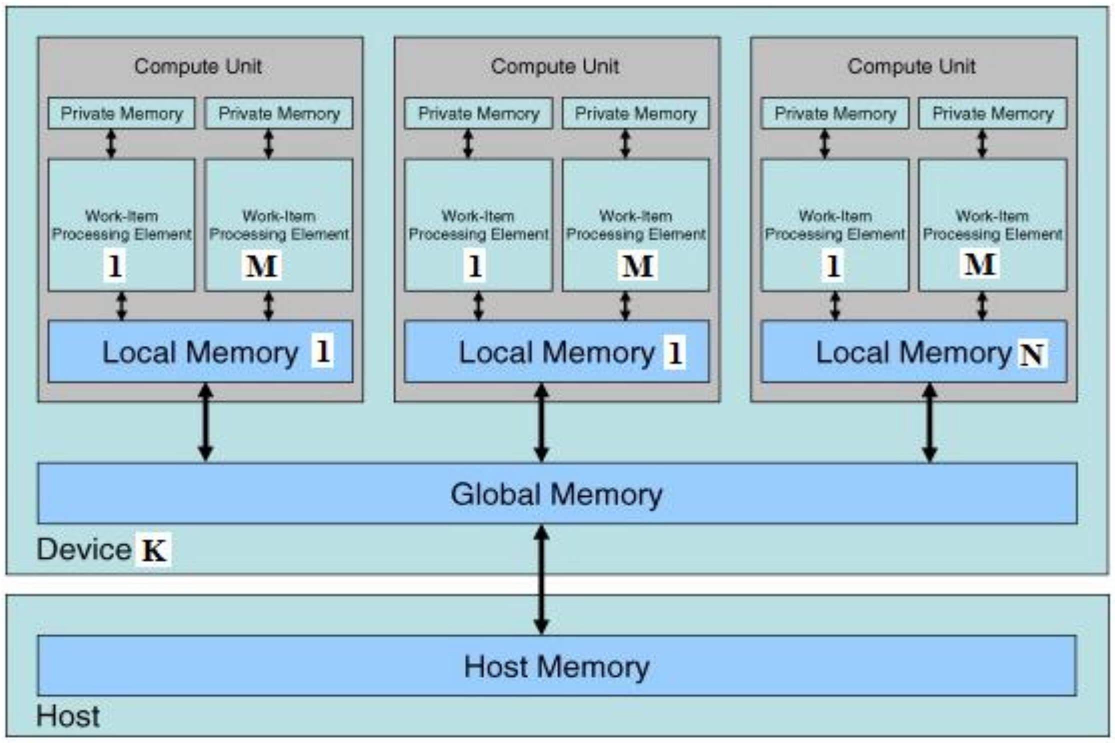


Figure2.15: *Memory Model of OpenCL*

4. Programming Model

OpenCL supports data-parallel and task-parallel programming models, as well as hybrids of these models. Of the two, the primary one is the data-parallel model.

1. Data-Parallel Programming Model

In the data parallel programming model, a computation is defined in terms of a sequence of instructions that executes at each point in an N-dimensional index space. Or can say same independent operations on lots of data. The OpenCL data-parallel programming model is hierarchical. The hierarchical subdivision can be specified in two ways:

• **Explicitly** - the developer defines the total number of work-items to execute in parallel, as well as the division of work-items into specific work-groups.
• **Implicitly** - the developer specifies the total number of work-items to execute in parallel, and OpenCL manages the division into work-groups.

Examples:

- ✓ Modify every pixel in an image with the same filters
- ✓ Update every point in a grid using the same formula

2. Task-Parallel Programming Model

In this model, independent threads can process separate functions. This is equivalent to executing a kernel on a compute device with a work-group and NDRange containing a single work-item. Parallelism is expressed using vector data types implemented by the device, enqueuing multiple tasks, and/or enqueuing native kernels developed using a programming model orthogonal to OpenCL.

2.3.4 Structure of OpenCL Programming [7]

A key point to note is that in OpenCL the compiler is built into the runtime, which provides exceptional flexibility and portability as OpenCL applications can select and use different OpenCL devices in the system at runtime. It is even possible to create OpenCL application executables today that can use - without modification - devices that have not even been invented yet! There is two part of OpenCL Programming:

1. **OpenCL Host Program**
2. **OpenCL GPU Program *(Kernel*)**

2.3.4.1 OpenCL Host Program Flow

Before, writing the kernel (i.e. the GPU code) we require writing the host program (i.e. the CPU code) to use and control the GPU. Host program will initialize the GPU; it will send data and the kernel code to the GPU. Afterwards it also instructs the GPU to start execution and when the results are ready, it read back the results from the GPU. OpenCL makes it easy for multiple implementations of OpenCL to co-exist on the same machine.

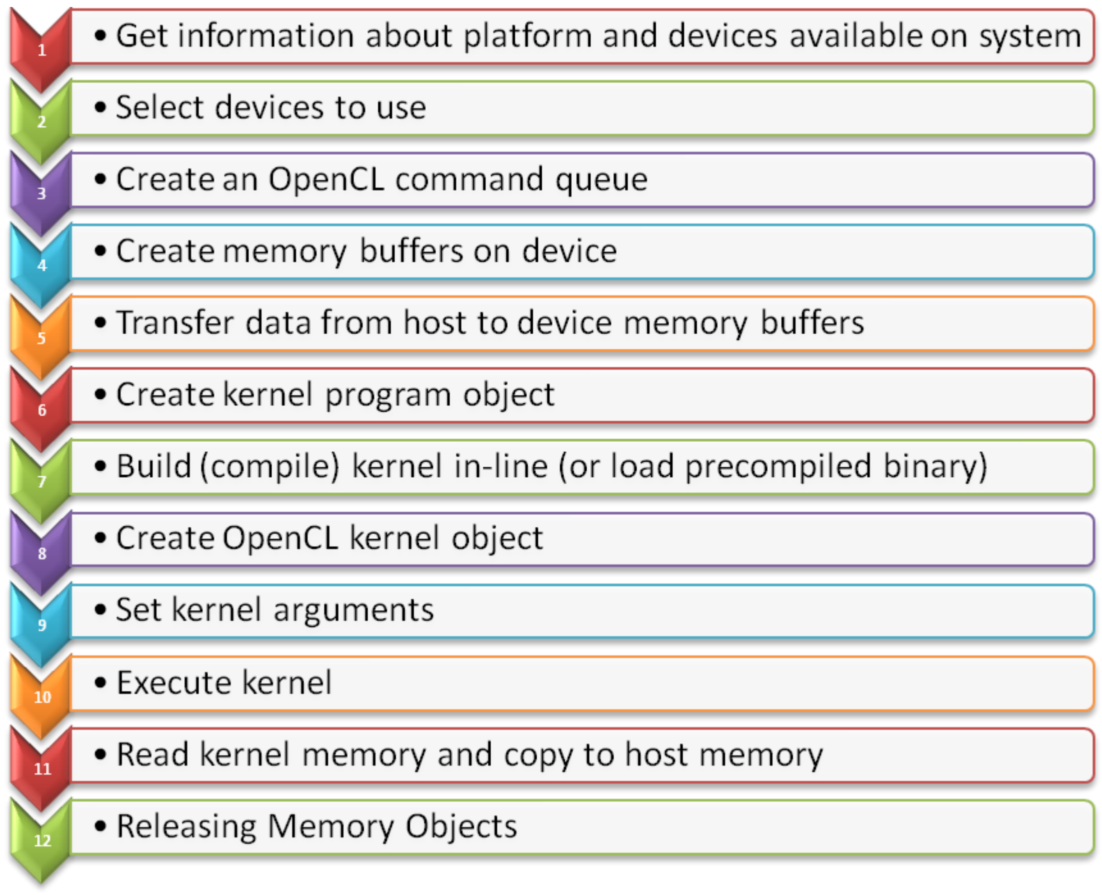

1
• Get information about platform and devices available on system
2
• Select devices to use
3
• Create an OpenCL command queue
4
• Create memory buffers on device
5
• Transfer data from host to device memory buffers
6
• Create kernel program object
7
• Build (compile) kernel in-line (or load precompiled binary)
8
• Create OpenCL kernel object
9
• Set kernel arguments
10
• Execute kernel
11
• Read kernel memory and copy to host memory
12
• Releasing Memory Objects

Figure2.16: *OpenCL Host Program flow*

A Brief description of above process [7]

1. **Platform IDs: (Platform layer)** First Step in any OpenCL application

   cl_int err = clGetPlatfromIDs(

   1, // the number of entries that can added to platforms

   &platforms, // list of OpenCL found

   &num_platforms // the number of OpenCL platforms available

   );

2. **OpenCL Device: (Platform layer)** Search for OpenCL compute devices in system

   cl_int err = clGetDeviceIDs(

   platform_id, // the platform_id retrieved from clGetPlatformIDs

```
                    CL_DEVICE_TYPE_GPU,    // the device type to search for
                    1,                     // the number of ids to add to device_id list
                    &device_id,            // the list of device ids
                    &num_of_devices        // the number of compute devices found
                    );
```

3. **Creating Context:** Manage command queues, program objects, kernel objects, memory object

```
context = clCreateContext(
                    properties,      // list of context properties
                    1,               // num of devices in the device_id list
                    &device_id,  // the device id list
                    NULL, // pointer to the error callback function (if required)
                    NULL,  // the argument data to pass to the callback function
                    &err             // the return code
                    );
```

4. **Creating Command Queue:** Allows kernel commands to be sent to Compute devices

```
command_queue = clCreateCommandQueue(
                    context,         // a valid context
                    device_id,  // a valid device associated with the context
                    0,               // properties for the queue (not used here)
                    &err              // the return code
                    );
```

5. **Create Program:**

```
program = clCreateProgramWithSource(
                    context,  // a valid context
```

```
                1,  // the number strings in the next parameter

                (const char **) &ProgramSource,  // the array of strings

                NULL,  // the length of each string or can be NULL terminated

                &err   // the error return code

                );
```

6. **Building Program Executables: (Compiler)** Compile and link program object created from **step5**.

```
   err = clBuildProgram(

                program,  // a valid program object

                0,  // number of devices in the device list

                NULL,  // device list –NULL means for all devices

                NULL,  // a string of build options

                NULL,  // callback function when executable has been built

                NULL  // data arguments for the callback function

                );
```

7. **Creating Buffer: (Runtime layer)**

```
cl_mem input;

input = clCreateBuffer(

        context, // a valid context

         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, // bit- field flag to
                                                       specify the usage of memory

        sizeof(float) * DATA_SIZE,   // size in bytes of the buffer to allocated

        inputsrc,  // pointer to buffer data to be copied from host

       &err  // returned error code

       );
```

8. **Reading/Writing Buffer Objects: (Runtime Layer)**

cl_int clEnqueueReadBuffer (

command_queue, // valid command queue

input, // memory buffer to write to

CL_TRUE, // indicate blocking write

0, // the offset in the buffer object to write from

sizeof(float) *DATA_SIZE, // size in bytes of data being read

host_ptr, // pointer to buffer in host mem to read data from

0, // number of event in the event list

NULL, // list of events that needs to complete before this executes

NULL // event object to return on completion

);

Similarly we can use Writing buffer objects for Read back results**.**

9. **Creating Kernel:** Kernel object encapsulates specified **__**kernel function along with the arguments. Kernel object is what get sent to command queue for execution. **(Runtime layer)**

cl_kernel kernel;

kernel = clCreateKernel(

program, // a valid program object that has been successfully built

"program_name", // the name of the kernel declared with __kernel

&err // error return code

);

10. **Setting Kernel Arguments:** Specify arguments that are associated with the **__**kernel function. **(Runtime layer)**

err = clSetKernelArg(

kernel, // valid kernel object

0, // the specific argument index of a kernel

sizeof(cl_mem), // the size of the argument data

&input_data // a pointer of data used as the argument

);

Example Kernel function declaration

__kernel Op_square (__global float *a, __global float *result)

11. **Executing Kernel: (Runtime layer)**

err = clEnqueueNDRangeKernel(

command_queue, // valid command queue

kernel, // valid kernel object

1, // the work problem dimensions

NULL, // reserved for future revision - must be NULL

&global, // work-items for each dimension

NULL, // work-group size for each dimension

0, // number of event in the event list

NULL, // list of events that needs to complete before this executes

NULL // event object to return on completion

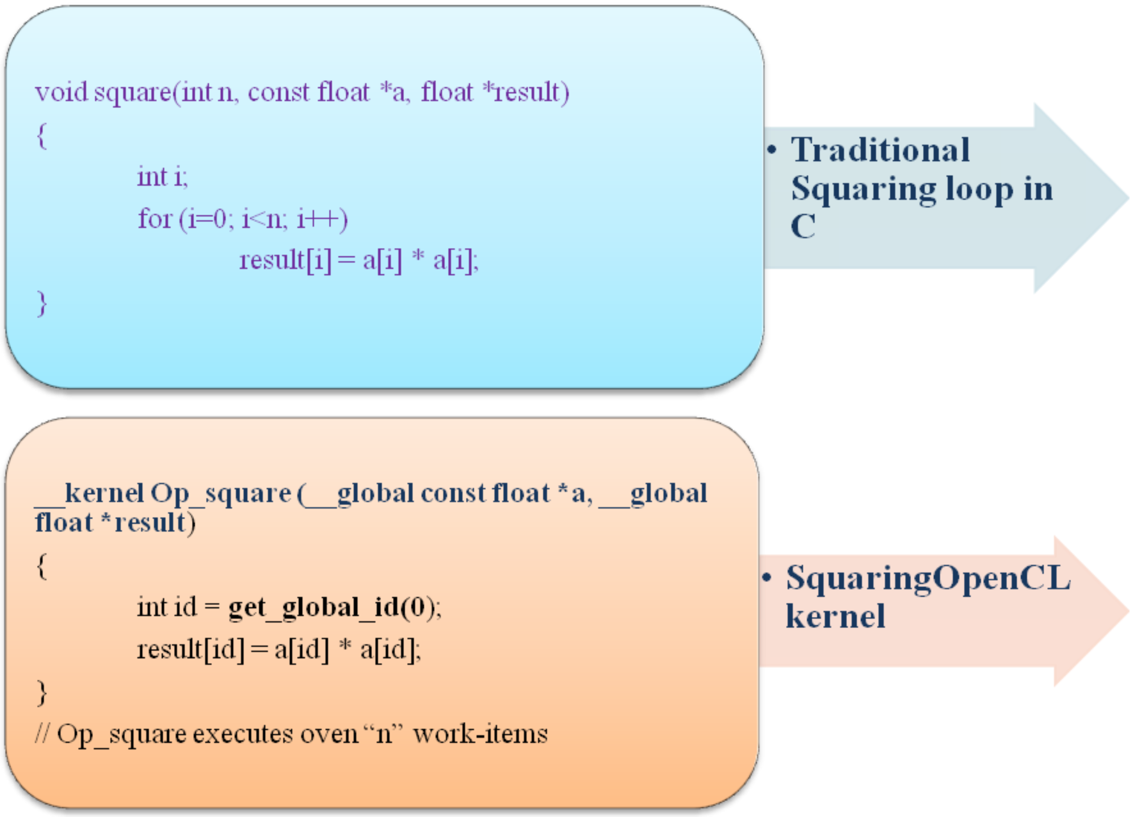
);

12. **Releasing Memory Objects**

```
clReleaseKernel(vector_add_k);
clReleaseCommandQueue(queue);
clReleaseContext(context);
clReleaseMemObject(src_a_d);
```

2.3.4.2 OpenCL GPU(kernel) Program

Main Idea of OpenCL is to replace loops with data-parallel functions (kernels) that execute at each point in a problem domain



```
void square(int n, const float *a, float *result)
{
        int i;
        for (i=0; i<n; i++)
                result[i] = a[i] * a[i];
}
```

```
__kernel Op_square (__global const float *a, __global
float *result)
{
        int id = get_global_id(0);
        result[id] = a[id] * a[id];
}
// Op_square executes oven "n" work-items
```

Figure 2.17: *Example of a Basic OpenCL Kernel*

Code comparison - note differences

- ✓ Loop over N elements) N kernel instances execute in parallel
- ✓ Qualifiers: kernel, global
- ✓ Each kernel instance has a global identification number (gid)
- ✓ An argument with __global keyword defines the global memory element of OpenCL.

Chapter-3

PROJECT BACKGROUND

3.1 Key Frame Extraction

In order to extract valid information from video, process video data efficiently, and reduce the transfer stress of network, more and more attention is being paid to the video processing technology. The amount of data in video processing is significantly reduced by using video segmentation and key-frame extraction. Experimental results show that the extracted key frames can summarize the salient content of the video and the method is of good feasibility, high efficiency, and high robustness.

Key Frame Extraction is a technology that allows summarizing a video in the most significant images, improving the organization and discovery of multimedia elements in large repositories. The key frame is the frame which can represent the salient content and information of the shot. The key frames extracted must summarize the characteristics of the video, and the image characteristics of a video can be tracked by all the key frames in time sequence. In recent years, many algorithms of key frame extraction focused on original video stream. It can introduce processing inefficiency and computational complexity when decompression is required before video processing. Furthermore, the content of the video can be recognized. A basic rule of key frame extraction is that key frame extraction would rather be wrong than not enough. So it is necessary to discard the frames with repetitive or redundant information during the extraction.

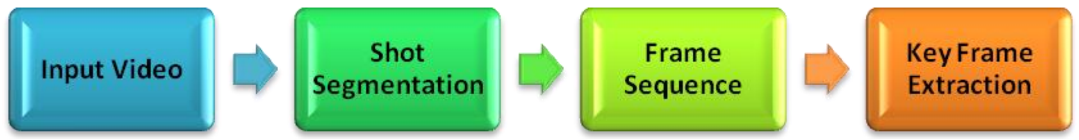


Figure 3.1: *The basic KFE algorithm from MPEG video stream*

The Key Frame Extraction analyzes the images of the video in order to extract information about the most relevant images. The analysis is performed by complex image analysis that could be improved through optimizations of the most intensive algorithms.

3.1.1 Pre-Processing Algorithms for KFE Algorithm

1. **Image Refinement using Frequency Selective Weighted Median Filter (FSWM)**

The finest detail in the image only is visible if the image is sharply focused. An image is in focus when the maximum detail is visible, focus criteria are often based upon the assessment of the difference between the intensity of adjacent pixels. The image is in focus when each part of it reaches a local maximum or minimum in intensity. Born and Wolf (1980) show the distribution of energy around the focus and it is clear that the intensity is a maximum at the focus [8].

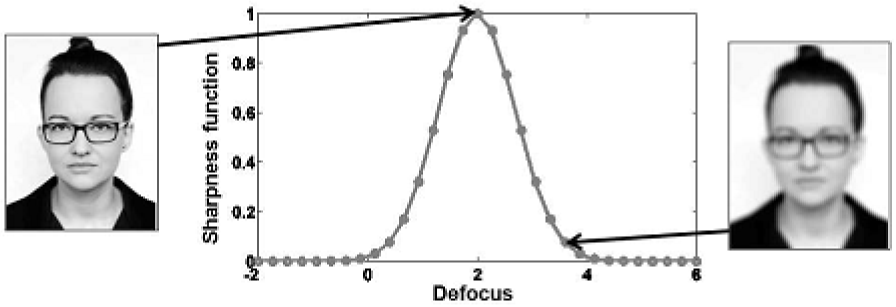


Figure 3.2: *Sharpness function reaches its optimum at the in-focus image. The goal of the autofocus procedure is to find the value of the defocus [9].*

In general focused images have high frequency components than defocused images of the scene. Goal of this preprocessing unit for KFE is to find out the high frequency components (focus value) of an image by adjusting the Defocus (focusing lens) of the camera. A relation between the focus value and the defocus is shown in the *figure 3.2*.

Here Frequency Selective Weighted Median Filter (FSWM) based focus measure are used in the presence of Impulsive noise (IN) as impulsive noise produced by image sensors or communication channels, corrupts images in many practical applications. This noise may cause miscalculation of sharpness values which, in turn, introduce considerable errors in an image. Experimental results show that FSWM based focus measure can provide better performance than other focus measures [10].
This process necessitates an analytic focus measure which can be used to evaluate the sharpness of focus in a part of the image.

A sharpness criterion should:

- ✓ Respond to high-frequency variations in image intensity.
- ✓ Be independent from the image content.
- ✓ Be computationally efficient for the real-time implementations.

1.1 Frequency selective weighted median filter (FSWM):

The FSWM based focus measure not only responds to high frequency components of the images, but also eliminates the effects of impulsive noise. The high frequency content of an image can be measured by a gradient estimator because it is inherently a high-pass filter. Therefore, it can measure the sharpness of an image more precisely.

The characteristics of the high-pass filter can be improved by a nonlinear weighted median (WM) filter. The WM filter can be represented with <*W*; *F*>, where $W = [w_1, w_2 \dots w_m]$ and $F = [f_1, f_2 \dots f_m]$ are the weight vector and the discrete time continuous valued input vector of a WM filter respectively.

The output of the WM filter is computed by repeating each sample f_i to the number of the corresponding weight w_i followed by sorting the resulting array. Then, the median value from the expanded vector is chosen.

For example, *median* {*f* (*i*-3), 2 ◊ *f* (*i*-2), 3 ◊ *f* (*i*), 2 ◊ *f* (*i*+1), f (*i*+4)}, where ◊ is the duplicating operator and *w* ◊ *f* represents that *f* is repeated *w* times. WM filters can be linearly combined to form an FSWM filter that can be defined as:

$$H = \sum_i a_i < W_i, F_i > \qquad \dots\dots\dots\dots\dots \textbf{\textit{(3.1)}}$$

For getting good sharpness values following high pass filter can be applied using Median Filter:

$$\begin{aligned} H &= med\{f(n-1), f(n), F(n+1)\} \\ &- \frac{1}{2} med\{f(n-3), f(n-2), f(n-1)\} \qquad \text{------------ } \textbf{\textit{(3.2)}} \\ &- \frac{1}{2} med\{f(n+1), f(n+2), f(n+3)\} \end{aligned}$$

Here H_h and H_v, respectively be FSWM filtering results that are obtained by applying the filter using (3.2) to an image along the horizontal and vertical directions. The sharpness measure can be defined as:

$$FSWM = \sum_i \sum_i \left[H_h^2 + H_v^2 \right] \qquad \text{-----------------} \textbf{\textit{(3.3)}}$$

1.2 Median Filter (MF)

As we have seen from above section that FSWM filtering works in the presence of Impulsive noise (IP), so it uses a Median Filter to remove these noise components . A Median Filter is high pass filter used in equation (3.2)
In median filtering, the neighboring pixels are ranked according to brightness (intensity) and the median value becomes the new value for the central pixel or for the pixel under evaluation. Median filters can do an excellent job of rejecting certain types of noise, in particular “shot” or impulse noise in which some individual pixels have extreme values. The median filtering is more robust “average” than the mean, as it is not affected by extreme values. Since the output pixel value is one of the neighboring values only, “unrealistic” values are not created near edges. Since edges are minimally degraded, median filters can be applied repeatedly, if necessary.

Median filters offer three advantages as compare to the smoothing filters.

- ✓ No reduction in contrast across steps, since output values available consist only of those present in the neighborhood (no averages).
- ✓ Median filtering does not shift boundaries, as can happen with conventional smoothing filters (a contrast dependent problem).
- ✓ Since the median is less sensitive than the mean to extreme values (outliers), those extreme values are more effectively removed.

1.2.1 How it works

A template of size 3x3, 5x5, 7x7… etc is applies to each pixel of the image. The values within this template are sorted and the middle of the sorted list is used to replace the templates central pixel:

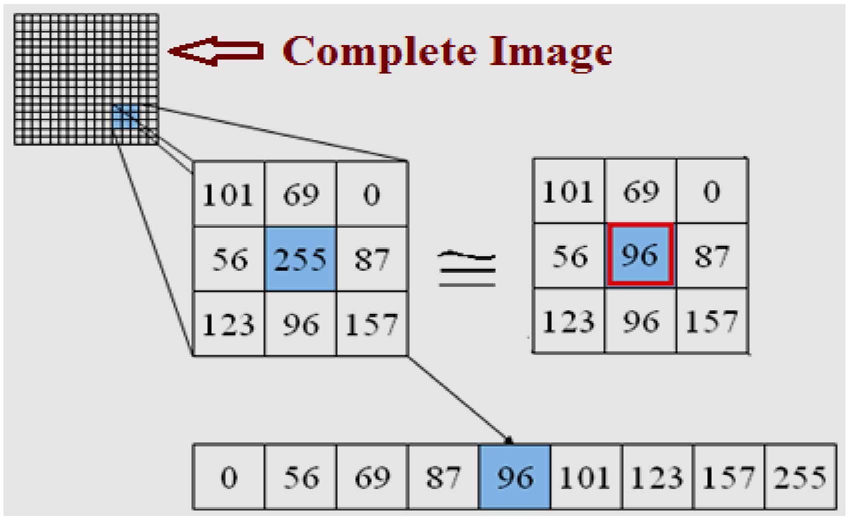
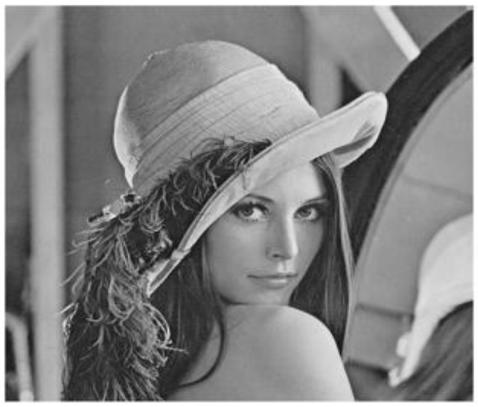
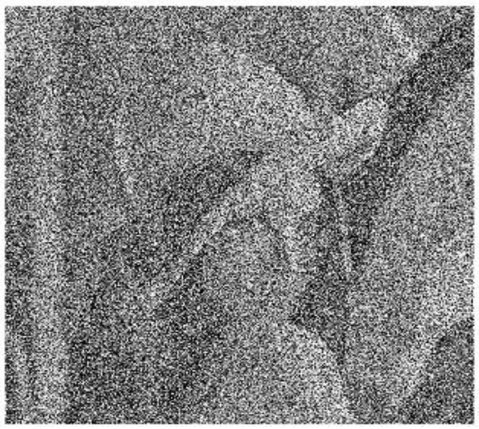


Figure 3.3: *Median Filtering with template size* 3x3





(a) *(b)*

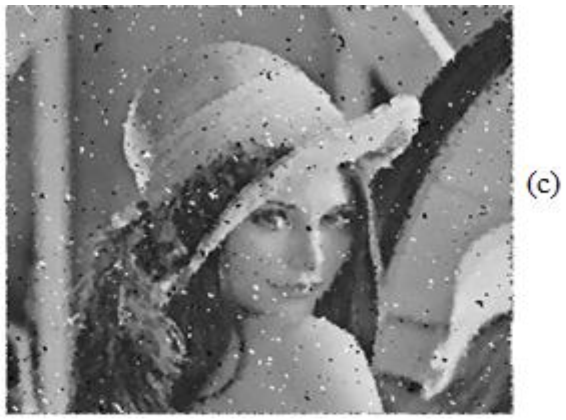


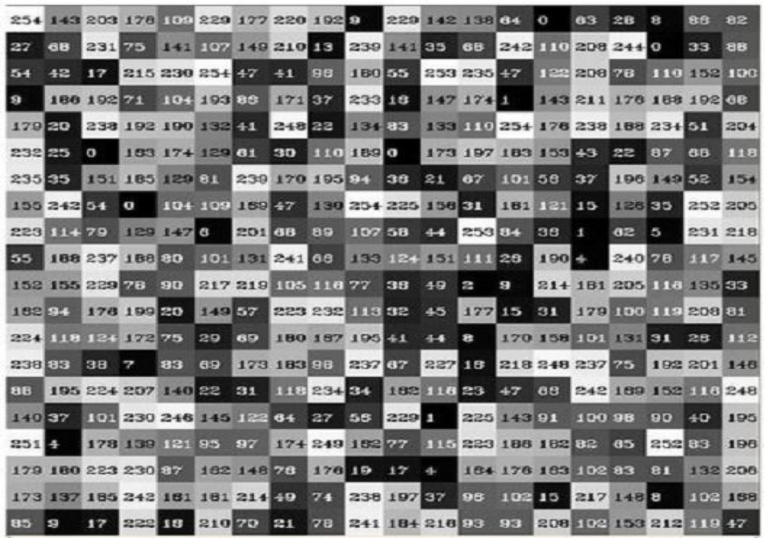
Figure 3.4: *Median Filtering Example (a) original image (b) corrupted with 60% noise (c) output from median filter*

2. Histogram Calculation for KFE Algorithm

2.1 What are histograms?

- ✓ Histograms are collected counts of data organized into a set of predefined bins. The Histogram of the image represents the distribution of the pixels in the image over the available gray level scale. When the gray level values of the pixels are too close together, modification of the image histogram enhances its contrast.
- ✓ When we say data we are not restricting it to be intensity. The data collected can be whatever feature we find useful to describe our image.

For an example, here is a Matrix that contains information of an image (i.e. intensity in the range 0- 255):



254	143	203	178	109	229	177	220	192	9	229	142	138	64	0	63	28	8	88	82
27	68	231	75	141	107	149	210	13	239	141	35	65	242	110	208	244	0	33	88
54	42	17	215	230	254	47	41	88	180	55	253	235	47	122	208	78	110	152	100
9	186	192	71	104	193	88	171	37	233	18	147	174	1	143	211	176	188	192	68
179	20	238	192	190	132	41	248	22	134	83	133	110	254	176	238	188	234	51	204
232	25	0	183	174	129	61	30	110	189	0	173	197	183	153	43	22	87	66	118
235	35	151	185	129	81	239	170	195	94	38	21	67	101	58	37	196	149	52	154
155	242	54	0	104	109	169	47	130	254	225	156	31	181	121	15	126	35	252	205
223	114	79	129	147	6	201	68	89	107	58	44	253	84	38	1	62	5	231	218
55	188	237	188	80	101	131	241	66	133	124	151	111	28	190	4	240	78	117	145
152	155	229	78	90	217	219	105	116	77	38	49	2	9	214	181	205	116	135	33
182	94	178	199	20	149	57	223	232	113	32	45	177	15	31	179	100	119	208	81
224	118	124	172	75	29	69	180	187	195	41	44	8	170	158	101	131	31	28	112
238	83	38	7	83	69	173	183	88	237	67	227	18	218	248	237	75	192	201	146
88	195	224	207	140	22	31	118	234	34	182	116	23	47	68	242	189	152	118	248
140	37	101	230	246	145	122	64	27	56	229	1	225	143	91	100	98	90	40	195
251	4	178	139	121	95	97	174	249	182	77	115	223	188	182	82	65	252	83	198
179	180	223	230	87	182	148	78	176	19	17	4	184	176	183	102	83	81	132	206
173	137	185	242	181	181	214	49	74	238	197	37	98	102	15	217	148	8	102	168
85	9	17	222	18	210	70	21	78	241	184	218	93	93	208	102	153	212	119	47

Figure 3.5: *An Image with its pixel values* [11]

Since the range of information value for this case is 256 values, to count this information in an organized way, we can segment our range in subparts (called bins) like:

[0; 255] = [0; 15] U [16; 31] U………………U [240; 255]

Range = bin_1 U bin_2 U………………. U $\text{bin}_{n=16}$

Thus we can keep count of the number of pixels that fall in the range of each bin_i. By applying this to the example shown in above *figure 3.5* we can get the image below (axis x represents the bins and axis y the number of pixels in each of them).

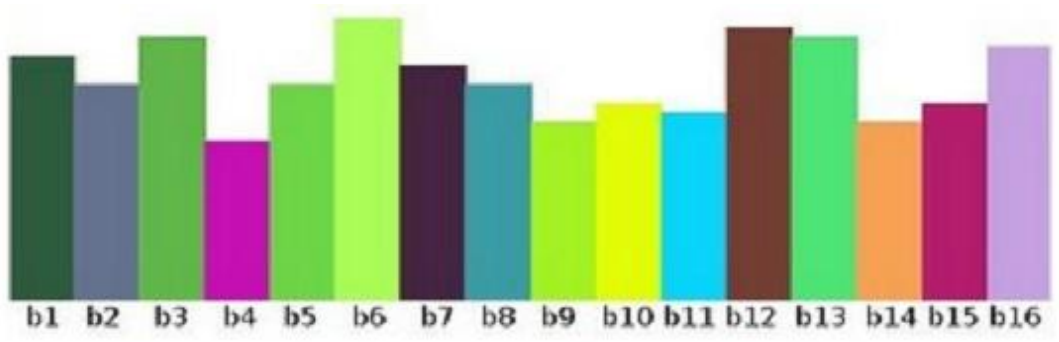


Figure 3.6: *Histogram of an Image*

This was just a simple example of how a histogram works and why it is useful. A histogram can keep count not only of color intensities, but of whatever image features that we want to measure (i.e. gradients, directions).

Let's identify some parts of the histogram:

1. **Dims:** The number of parameters you want to collect data of. In above example, dims = 1 because we are only counting the intensity values of each pixel (in a grey scale image).

2. **Bins:** It is the number of subdivisions in each dim. In above example, no of bins = 16

3. **Range:** The limits for the values to be measured. In this case: Range = [0,255]

Histogram for counting two features in the image would be a 3D plot (in which x and y would be binx and biny for each feature and z would be the number of counts for each combination of (binx; biny). The same would apply for more features.

3.2 Introduction to Feature Point tracking

The problem of computing the motion in an image is known as finding the optical flow of the image or feature tracking. A feature, or a point of interest, is a point or a set of points where an algorithm can look and follow the motion through frames. There are several ways to select the features: based on brightness and colors or based on corners and edges detection.

To track features there are essentially two important steps. The first one is to decide which features to track, which is called by function the ***_KLTSelectGoodFeatures()***, and the second one is the tracking in itself. There are a variety of well- under stood techniques for doing so, but the Kanade- Lucas- Tomasi method stands out for its simplicity and lack of assumptions about the under lying image.
We are going to change the order in the explanation, and we start explaining how the tracking works using KLT Tracking, and afterwards how the features are selected, this is the main part of this thesis which we have done using *FAST DETECTION ALGORITHM* instead of KLT Detector we explain it in the section 3.3.1.

The simplest algorithm for point feature tracking between two frames of video is as follows:

- ✓ Choose a small window; say 5 pixels on a side, around a pixel of interest in Image A. This pixel of interest will be called pixel x.
- ✓ Let this pixel **x** in image A moves somewhere in image B at pixel **y**
- ✓ Finding this new position is called **Feature Tracking.**
- ✓ **y** is the pixel of B that is the most "similar" to **x**, in a constraint neighborhood
- ✓ Translation between **x** and **y** = Optical Flow

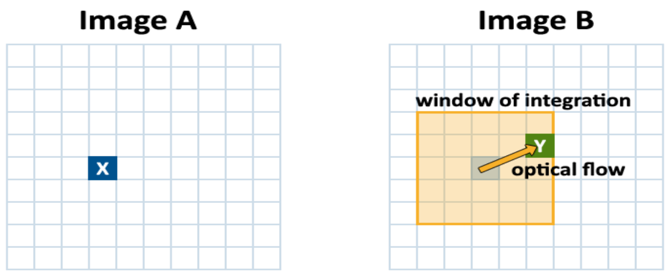


Figure 3.7*: *Point of interest x in Image A moved to y in Image B

- ✓ For each pixel near x in Image B, call it pixel y, and perform the following:
    - ➢ Subtract the value of each pixel in the 5 by 5 region around pixel x from each pixel in the 5 by 5 region around pixel y. Square the result of the difference, and sum these 25 values to produce a 'dissimilarity' for this choice of pixel y.

- ✓ The pixel y in image B with the smallest dissimilarity is considered to be the new location of pixel x in image A.

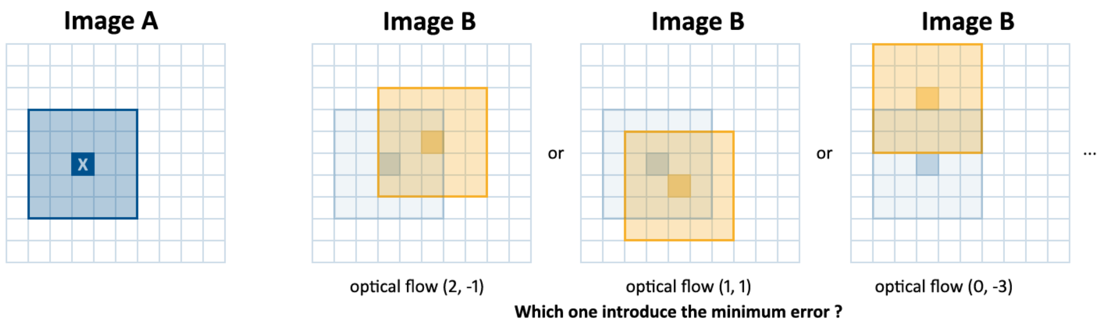


Figure 3.8: *Tracking of interest point using Window of size 5X 5*

3.2.1 Kanade-Lucas-Tomasi Feature Tracking

The following derivation summarizes the iterative step of the Kanade-Lucas-Tomasi algorithm [12]. Consider two images, I and J; here we want to track a feature of known location $x' = [x, y]^T$ in image I to image J, finding its displacement $d = [d_x, d_y]^T$. We have window W over which dissimilarity ε between the new and old feature as:

$$\varepsilon = \iint_W [J(x') - I(x' - d)]^2 dx' \qquad \text{---------- } (3.4)$$

For making this relation symmetric substitute-: $x' = x + \frac{d}{2}$

$$\varepsilon = \iint_W \left[J\left(x + \frac{d}{2} \right) - I\left(x - \frac{d}{2} \right) \right]^2 dx \qquad \text{--------- } (3.5)$$

To get value of d such that it minimizes we get following expression:

$$\frac{\partial \varepsilon}{\partial d} = 0 = 2 \iint_W \left[J\left(x + \frac{d}{2} \right) - I\left(x - \frac{d}{2} \right) \right] \left[\frac{\partial J\left(x + \frac{d}{2} \right)}{\partial d} - \frac{\partial I\left(x - \frac{d}{2} \right)}{\partial d} \right] dx \qquad \text{-- } (3.6)$$

In order to solve for d, Taylor series expansion is used by neglecting second order or higher derivatives:

$$J\left(x+\frac{d}{2}\right)\approx J(x)+\frac{d_x}{2}\frac{\partial J}{\partial x}(x)+\frac{d_y}{2}\frac{\partial J}{\partial y}(x)$$ ---------- *(3.7)*

And, $$I\left(x-\frac{d}{2}\right)\approx I(x)-\frac{d_x}{2}\frac{\partial I}{\partial x}(x)-\frac{d_y}{2}\frac{\partial I}{\partial y}(x)$$ ---------- *(3.8)*

Equation (3.6) can be approximated as:

$$\frac{\partial \varepsilon}{\partial d}\approx\iint_W\left[J(x)-I(x)+\frac{1}{2}g^T(x)d\right]g(x)dx=0$$ ---------- *(3.9)*

Where

$$g=\begin{bmatrix}\frac{\partial}{\partial x}(I+J)\\ \frac{\partial}{\partial y}(I+J)\end{bmatrix}$$ ---------- *(3.10)*

So above terms can we rearranged as follows:

$$\iint_W\left[J(x)-I(x)+\frac{1}{2}g^T(x)d\right]g(x)dx=0$$

$$\iint_W[J(x)-I(x)]g(x)dx=-\iint_W\frac{1}{2}g^T(x)dg(x)dx$$

$$\iint_W[J(x)-I(x)]g(x)dx=-\frac{1}{2}\left[\iint_W g^T(x)g(x)dx\right]d$$ -- *(3.11)*

Thus, the expression can be simplified to 2x2 matrix equation,

$$Zd=e\,,$$ ---------- *(3.12)*

Where Z is a 2x2 matrix,

$$Z = \iint_{W} \left[g(x) g^{T}(x) \right] dx \qquad \text{---------- } (3.13)$$

And *e* is a 2x1 vector,

$$e = 2 \iint_{W} \left[I(x) - J(x) \right] g(x) dx \qquad \text{---------- } (3.14)$$

Equation (3.12) allows solving for the approximate displacement of a feature, given its starting location and the two images. Since in our case we are implementing this algorithm with *C/C++ Programming* we consider it with a discrete image composed of pixels, then above definitions for Z and e are computed with a *summation* over the window rather than an integral. The x and y image derivatives are approximated by convolving the images with a ***Sobel*** operator.

Since the above computation for displacement is only an approximate method, it is useful to repeat the procedure for much iteration. If the displacement does not converge towards zero after several iterations, the feature is considered *lost*. For features displaced by a large amount, the approximation also breaks down because the Taylor series approximation becomes less accurate. To handle such a case, it is best to perform several iterations on versions of the images re-sampled to a coarser resolution, followed by several iterations on the full-resolution images.

A final consideration for Kanade-Lucas-Tomasi algorithm is the choice of initial features. It is wasteful to track all pixels of the first image to the second image. A more useful approach is to track only those pixels which represent sharp and well-defined features [13].

For *Selecting Good Features to Track* a function ***_KLTSelectGoodFeatures()*** is called which uses The Eigen values of Z to give us an indication of how successful the tracking will be for a given feature. Large Eigen values indicate a feature that is more well-defined than the image noise and can thus be tracked reliably. Thus, when choosing features to track, we sort the pixels in descending order of their minimum Eigen value and pick the first N from the list, where N is the number of features we wish to track.

Above said method for *Selecting Good Features to Track* is a typical approach by *KLT Detector*. Here we have replaced this with *FAST DETECTION* whose description is given in the section 3.3.1.

A simple example of this algorithm in operation is shown in *Figure 3.9*

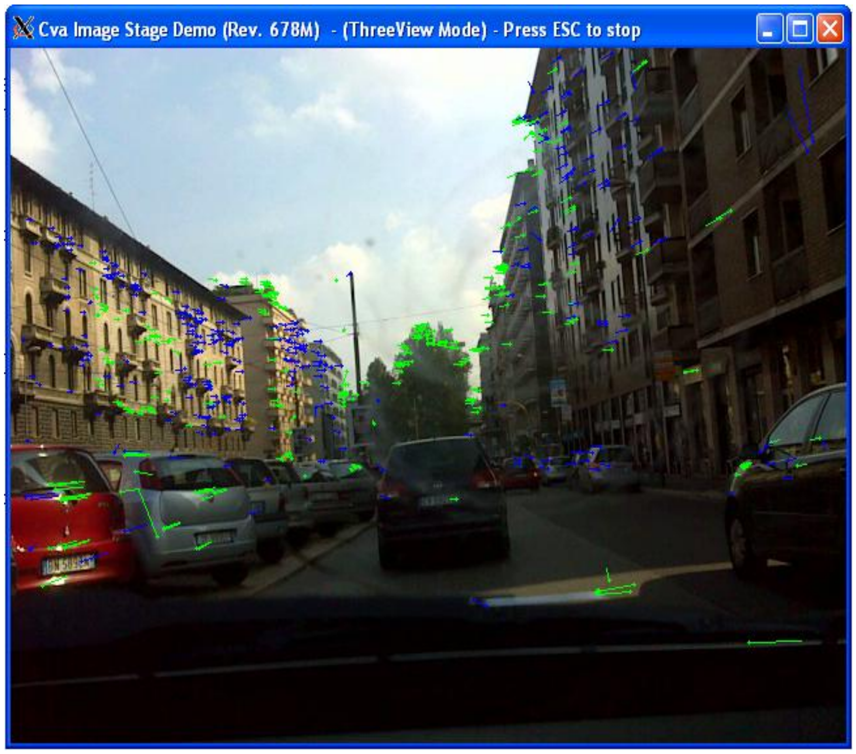

Cva Image Stage Demo (Rev. 678M) - (ThreeView Mode) - Press ESC to stop

Figure 3.9: *An example of the Kanade-Lucas-Tomasi point tracking algorithm in operation. Features being tracked are shown as a green dot, with the inter-frame displacement shown as a green line. The length of the line is exaggerated to show motion.*

3.2.2 Difficulties with KLT Tracking Algorithm

Although this algorithm would give us a new position and velocity for the feature represented by pixel A, it would suffer from several flaws.

- ✓ It would be slow, requiring about a hundreds of computation for each iteration and potentially hundreds of iterations depending on how far we want to search or on the image size.
- ✓ The algorithm would only give us the position and velocity of the feature to the nearest whole pixel.

The Kanade- Lucas-Tomasi algorithm alleviates these problems by using the image's gradients to predict the new location of the feature, iterating until the new location is converged upon.

As we have discussed above that a small window centered on the desired pixel is selected for track the motion of the pixel in successive frames, the size of the window have some effects as follows:

- ✓ Accurate tracking can be done with a small window (small integration area)
    - ➢ But difficult to take large motions or strong difference into account
- ✓ Robustness can be addressed with a large window
    - ➢ But smoothing effect too strong for accuracy

Pyramidal approach can be used to found a balance between accuracy and robustness, between large and small window. So PKLT (Pyramidal Kanade – Lucas - Tomasi) algorithm is used instead of KLT.

3.2.3 Pyramidal Kanade - Lucas - Tomasi (PKLT) Algorithm

- ✓ An image **pyramid** is a collection of images - all arising from a single original image - that are successively down sampled until some desired stopping point is reached.
    - ➢ There is Gaussian pyramid is used to down sample images and get different level of pyramids.

- ✓ In this algorithm different level of details are used for each image
    - ➢ **Low detail levels:** useful for robustness, large area of original image covered by window
    - ➢ **High detail levels:** bring back accuracy with small window relatively to image size

- ✓ Cascade of filtered images
    - ➢ Level 0: original image
    - ➢ In practice, 2 to 4 level of details can be used, **In our case level=2 is used**
    - ➢ Level m: last level

- ✓ Simple example of a Image Pyramid shown in **Figure 3.10**

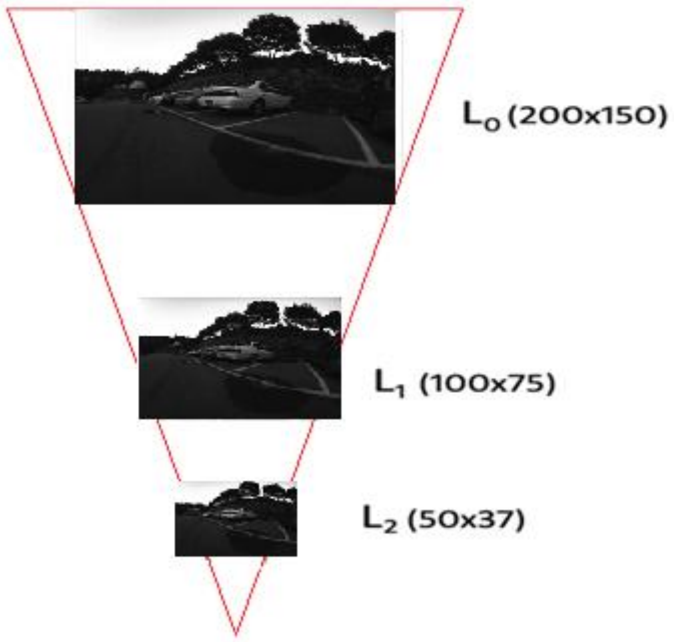


Figure 3.10: *Image Pyramid*

In PKLT algorithm original image is converted into pyramids of Level *L*. If we define $u^L = [u_x^L \quad u_y^L]$ as the coordinates of point u on a first image A [2]. Following the equation can be used to define this coordinate as:

$$u^L = \frac{u}{L} \qquad \text{---------- } (3.15)$$

Thus, the algorithm works as it follows:
1. The optical flow is computed at the deepest pyramid level L_m, using the classical Lucas Kanade optical flow algorithm.
2. The result is propagated to the upper level L_{m-1} in a form of an initial guess for the pixel displacement.
3. The optical flow is computed for the pyramid level L_{m-1}.
4. The same procedure until we reach the highest pyramidal level.

3.3 Feature Detection

A Feature refers to a small point of interest with variation in two dimensions. These points may arise from geometric discontinuities. A feature can be called as a corner. There are many corner detection algorithms like:

- ✓ The Moravec corner detection algorithm [14]
- ✓ The Harris & Stephens / Plessey / Shi-Tomasi corner detection algorithm [15]
- ✓ The SUSAN corner detector [16]

- ✓ KLT feature detector [17]
- ✓ AST based feature detectors [18]

Here we will not go into the details of each algorithm, but we will consider only AST based feature detectors i.e. FAST detector and how it is useful for our work.

3.3.1 FAST: Features from Accelerated Segment Test

The FAST (Features from Accelerated Segment Test) algorithm is a interest point identification algorithm based on the work of Rosten and Drummond [19]. An interest point in an image is a pixel which has a well-defined position and can be robustly detected. Interest points have high local information content and they should be ideally repeatable between different images. Interest point detection has applications in image matching, object recognition, tracking etc.

This is a corner detection algorithm based on segment test method. The Segment test algorithm works as follows:

- ✓ The Segment test criterion operates by considering a circle (This is a Bresenham circle) of 16 pixels around the corner candidate P.
- ✓ Let intensity of the point P is I_p.
- ✓ There is a threshold intensity value say, T is set. (It may be 20% of pixel P).
- ✓ The pixel P is said to be a corner if there exists a set of N contiguous pixels in the circle which are all brighter than the intensity of the candidate pixel I_p plus a threshold T, or all darker than $I_p – T$.
  These conditions can be written as:

  Condition 1: A set of N contiguous pixels S,
  $X \in S, I_x > I_p + T$

  Where I_x =Intensity of pixel X

  Condition 2: A set of N contiguous pixels S,
  $X \in S, I_x < I_p - T$

The value of N was originally [20] chosen to be twelve because it gives a high-speed test which can be used to exclude a very large number of non-corners.

3.3.2 High Speed Test or Accelerated Segment Test

To make the algorithm fast it performs following steps:

- ✓ It takes 4 pixels of the circle as test pixels, namely pixel 1, 9, 5 and 13.

- ✓ Firstly it takes pixels 1 and 9. If both of these are within [I_p - T, I_p + T], then P cannot be a corner. If P can still be a corner, pixels 5 and 13 are examined.

- ✓ At least three of these four pixels should satisfy the above two conditions so that the interest point will exist.

- ✓ The full segment test criterion (above procedure) can then be applied to the remaining candidates by examining all pixels in the circle.

The following *figure 3.11* explains the algorithm:

The green square is the pixel under consideration. The red squares which come on the circumference of the circle are the pixels which are compared with that of green pixel (P), the intensities are compared and then decide whether the pixel is the potential pixel or not.

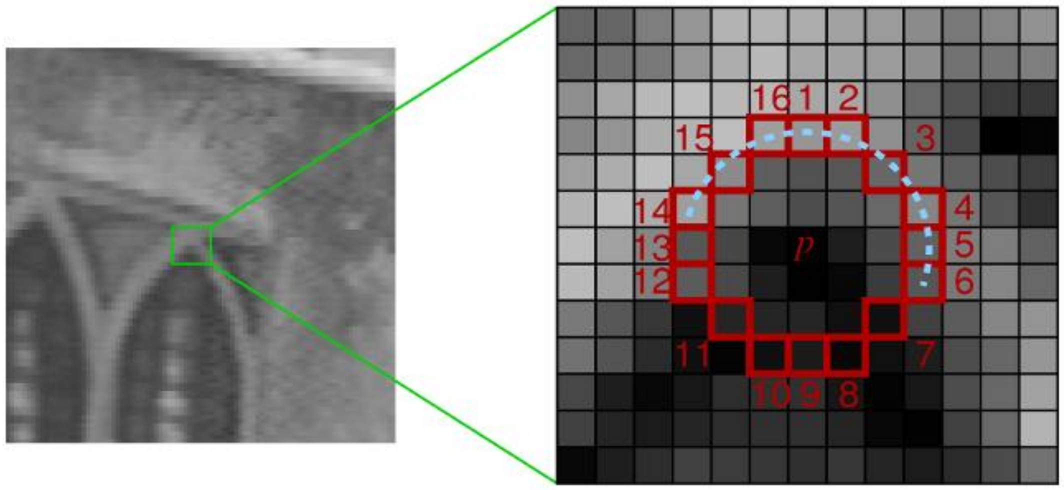


Figure 3.11: *Corner detection using FAST*

This detector in itself exhibits high performance, but there is a problem with this approach is multiple features are detected adjacent to one another. This can be removed by Non Maximal Suppression method.

3.3.3 Non Maximal Suppression for removing adjacent corners [21]

Detection of multiple interest points adjacent to one another is can be dealt with by applying non maximal suppression after detecting the interest points.

The algorithm is described below:
1. Compute a score function V for each of the detected points. The ***score function*** is defined as: **"***The sum of the absolute difference between the pixels in the contiguous arc and the centre pixel***"**.

2. Consider two adjacent interest points, compare their V values.
3. Discard the one with the lower V value.

The entire process can be summarized mathematically as follows:

$$V = \max \begin{cases} \sum(\text{Pixel values} - \text{P}) & ; \quad \text{if}(\text{value} - \text{P}) > T \\ \sum(\text{P} - \text{Pixel values}) & ; \quad \text{if}(\text{P} - \text{value}) > T \end{cases} \text{--------} \ (3.16)$$

Where, P is the centre pixel, T is the threshold for detection and pixel values correspond to the N contiguous pixels in the circle.

The score function can be defined in alternate ways as "*The key point here is to define a heuristic function which can compare two adjacent detected corners and eliminate the comparatively insignificant one*".

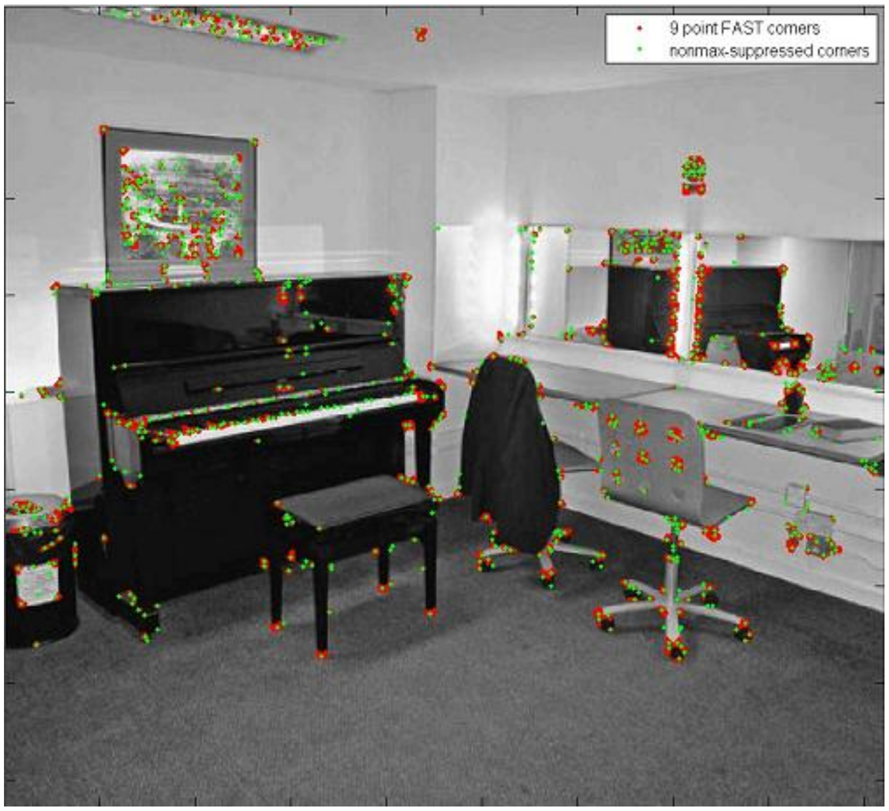


Figure 3.12: *An image with interest points detected. The green dots show the Non-maximally suppressed corners.*

3.3.4 WHY FAST?

1. **Repeatability:** The Repeatability is computed as the number of corners per frame is varied. For example if every pixel is detected as a corner, then the repeatability is 100%.
   As the number of corners per frame is increased, all of the other detectors, at some point, suffer from decreasing repeatability. This effect is least pronounced with the FAST9 detector [19]. Here 9 represent the no of contiguous pixels to be tested.

2. **Numbers of Frames Processed Per Second**: as the name suggest, The numbers of frames processed per second in detection using FAST is far better than that of other detectors, which is the biggest advantage of this algorithm Therefore, we switched to FAST rather than using KLT in our project will be discussed in next section of our thesis.

Chapter-4

METHODOLOGY USED FOR OPTIMIZATION

To perform Optimization (Acceleration) of said algorithms first of all we need some hardware and software specification. Such specifications are listed below as.

4.1 Development Platform

4.1.1 Hardware Used

CPU Used	:	Intel(R) Xeon(R) CPU E5504 @ 2.00GHz
GPU Used	:	NVIDIA Quadro NVS 295
GPU Used	:	NVIDIA Quadro FX 4600

4.1.2 Software Used

Table 4.1: *Software used*

Operating System	:	Ubuntu Linux version 2.6.38-8-generic
Compiler	:	Gcc 4.5.2
Debugging Tool	:	Gdb 7.2
Profiling Tool	:	Valgrind 3.6.1, gprof, Graphviz2.28.0
NVIDIA OpenCL SDK	:	OpenCL Library
Eclipse IDE	:	Helios
Scripting Tool	:	python 2.7.1

4.2 Environment Setup

The environment setup involved primarily of installing the following:

- ✓ NVIDIA driver toolkit
- ✓ NVIDIA CUDA SDK (which includes OpenCL library)
- ✓ OpenCV 2.2.0
- ✓ Eclipse IDE for source development

Further for profiling the software we need:

- ✓ Valgrind
- ✓ Graphviz

4.3 Specifications

4.3.1 CPU Specifications

Code is implemented on an Intel Xeon Processor having 8 CPU cores, below specification are for 0th core, for other cores of the processor specification are exactly same.

Table 4.2: *CPU Specifications*

Processor	:	0
Vendor_id	:	Genuine Intel
CPU family	:	6
model	:	26
model name	:	Intel Xeon CPU E5504 @ 2.00GHz
stepping	:	5
CPU MHz	:	1995.207
cache size	:	4096 KB
physical id	:	1
siblings	:	4
core id	:	0
CPU cores	:	4

4.3.2 GPU Specifications

GPU that has been used for the development of the project is NVIDIA Graphics Processor named (Quadro NVS 295)

Table 4.3: *OpenCL Software information*

CL_PLATFORM_NAME	:	NVIDIA CUDA
CL_PLATFORM_VERSION	:	OpenCL 1.0 CUDA 4.0.1
OpenCL SDK Revision	:	5985201

Table 4.4: *OpenCL Device information*

CL_DEVICE_NAME	:	Quadro NVS 295
CL_DEVICE_VENDOR	:	NVIDIA Corporation
CL_DEVICE_TYPE	:	CL_DEVICE_TYPE_GPU
CL_DRIVER_VERSION	:	275.28
CL_DEVICE_MAX_COMPUTE_UNITS	:	1
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	:	3
CL_DEVICE_MAX_WORK_ITEM_SIZES	**:**	**512 / 512 / 64**
CL_DEVICE_MAX_WORK_GROUP_SIZE	**:**	**512**
CL_DEVICE_MAX_CLOCK_FREQUENCY	**:**	**1300 MHz**
CL_DEVICE_ADDRESS_BITS	:	32
CL_DEVICE_MAX_MEM_ALLOC_SIZE	:	128 MB
CL_DEVICE_GLOBAL_MEM_SIZE	**:**	**255 MB**
CL_DEVICE_ERROR_CORRECTION_SUPPORT	:	no
CL_DEVICE_LOCAL_MEM_TYPE	:	local
CL_DEVICE_LOCAL_MEM_SIZE	**:**	**16 KB**
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	:	64 KB
CL_DEVICE_IMAGE_SUPPORT	:	1
CL_DEVICE_MAX_READ_IMAGE_ARGS	:	128
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	:	8
CL_DEVICE_COMPUTE_CAPABILITY_NV	:	1.1
NUMBER OF MULTIPROCESSORS	:	1
NUMBER OF CUDA CORES	**:**	**8**
CL_DEVICE_REGISTERS_PER_BLOCK_NV	:	8192
CL_DEVICE_WARP_SIZE_NV	:	32
CL_DEVICE_KERNEL_EXEC_TIMEOUT_NV	:	CL_TRUE
CL_DEVICE_INTEGRATED_MEMORY_NV	:	CL_FALSE
CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t>	:	CHAR 1, SHORT 1, INT 1, LONG 1, FLOAT 1, DOUBLE 0

After selecting the appropriate Platform for our work, first we have to find the parts of the algorithm that to be optimized. Parts of the algorithms which are most time consuming are selected for optimization or to port on GPU to take advantage of the GPGPU computing.

4.4 Constraints in Optimization

For optimization of the code using OpenCL there are certain constraints which we have to follow as:

- ✓ We can’t switch the sequence of instructions in code
    - ➢ numerical exactness is the primary concern
- ✓ Software to be implemented as a linkable library:
    - ➢ Modularity has to be preserved,
    - ➢ Each routine must serve as its stand alone service.
- ✓ We can’t port the entire code on GPU
    - ➢ contains chunks of sequential code which might slow down drastically on GPU
- ✓ We can’t depend on GPU local memory
    - ➢ OpenCL is a heterogeneous platform includes embedded GPUs which lack local memory
- ✓ Comparison with highly optimized code
    - ➢ O3 compilation: This enables more than 60 optimization options [22] gcc compiler.

4.5 Optimization Strategy

- ✓ For optimization we need those functions in the code that are consuming larger computational time and to that part we apply optimizing strategies, we achieve this by the timing analysis of the code with help of **Profiling**.

- ✓ Profiling code is a useful way to find frequently called routines in the application. In a lot of cases, it is possible to make applications run significantly faster, just by analyzing where the slowdowns are occurring and optimizing that code.

- ✓ Testing of implementation is done on a GPU that has 8 cores only to ensure the robustness to hardware changes.

4.6 Profiling Procedure

For generating a profile data of the code we use both *gprof* as well as *call grind* tools.

4.6.1 Profiling with gprof

- ✓ First we compiled the source code with –pg.
- ✓ When we execute the code, we get gmon.out that contains the information about the profiling data.
- ✓ Then we need the gprof tool to read the gmon.out file using following command line: (gprof executable file gmon.out)
- ✓ Profiling output using gprof is in tabular form, in which most time consuming function is listed first then other functions are tabulated in decreasing order of their time consumption. This tabular output is called as Call Graph.

4.6.2 Profiling with Valgrind [23]

- ✓ Valgrind provides a profiling tool called Callgrind.
- ✓ It gives the Call graph & number of cycles spent in each function with number of times each function is called whereas gprof gave call graph + time spent in each function without the CPU cycles.
- ✓ To generate a profile, execute the following command:

```
valgrind –tool = callgrind Execution_command
```

- ✓ The result will be stored in a callgrind.out.XXX file where XXX will be the process identifier.
- ✓ Another tool we have used, to visualize profiling data is the gprof2dot.py python script [24]. It can be used to visualize several different formats: "This is a Python script to convert the output from prof, gprof, oprofile, Shark, and python profilers into a dot graph.
- ✓ We also use the graphviz [25] (reads dot format only) to get the image representation in order to view the callgraph. This is done with command line:

```
python gprof2dot.py -f callgrind callgrind.out.xxx | ./dot -Tpng -o filename.png
```

By the use of Valgrind and callgrind we get a graphical representation of the profiling data, so it makes easy to find the most time consuming routine easily.

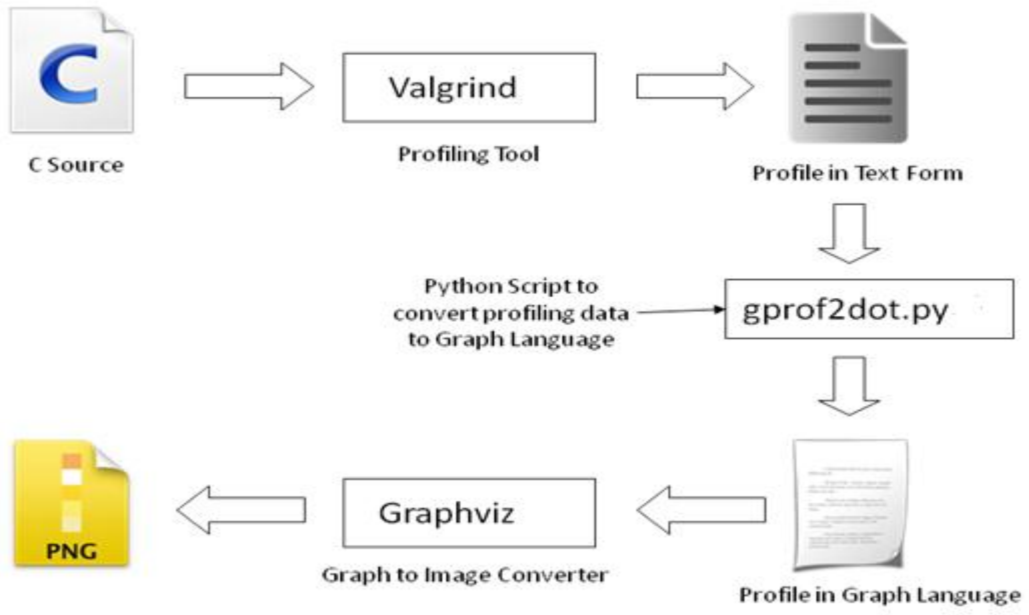


Figure4.1: *Profiling Procedure*

4.7 Profiling Output of KFE Algorithm

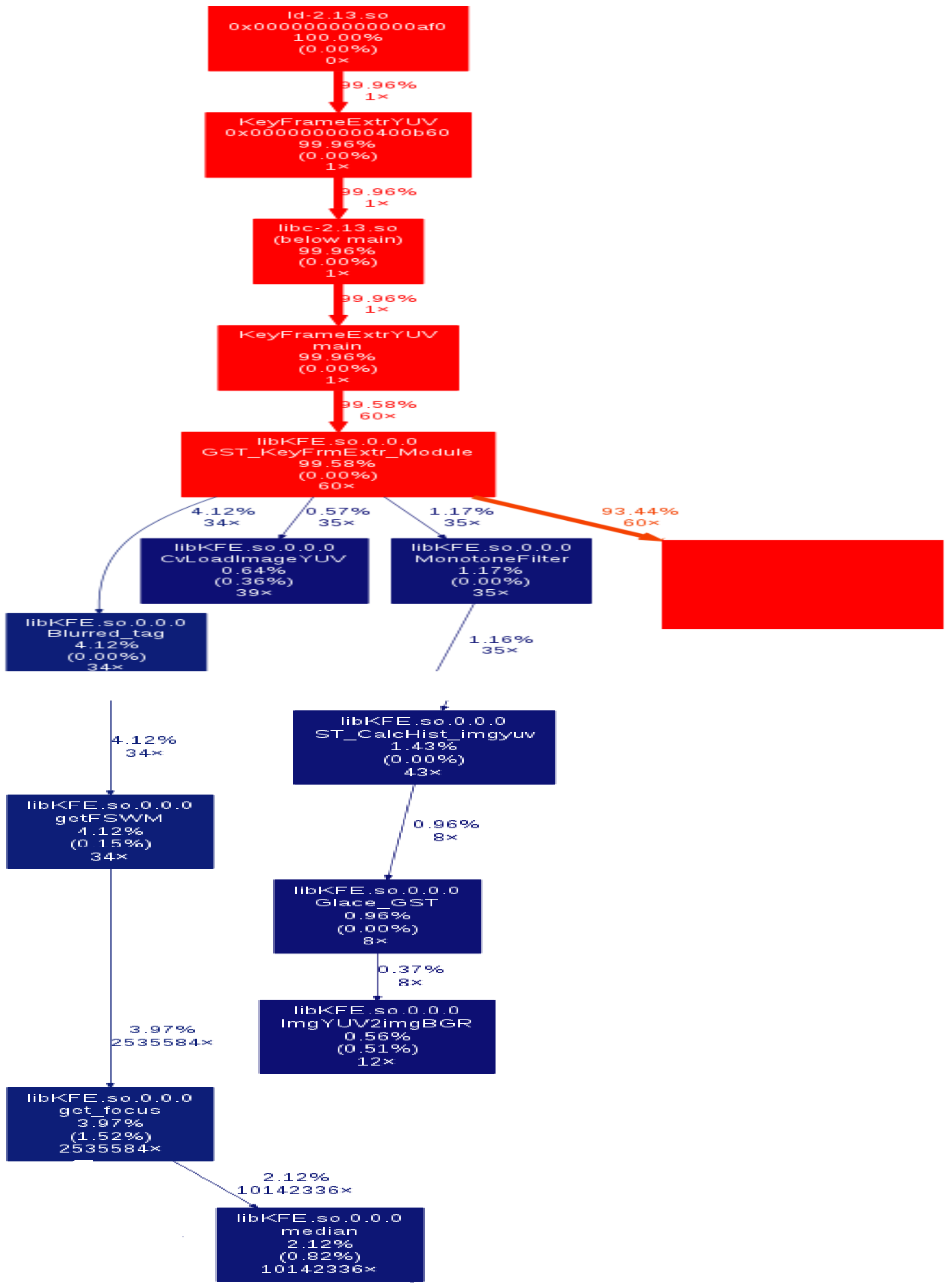


Figure 4.2 *Profiling Output of KFE*

4.8 Acceleration of KFE

- ✓ To accelerate the KFE algorithm firstly we have profiled the code using above procedure.
- ✓ After profiling, approx 97% of the execution time of the program was found to spent in the _KeyFrmExtr_Module as shown in *figure 4.2.*
- ✓ In this module most computation intensive functions are:
    - ➢ Blurred_tag
        - getFSWM
            - get_focus
                - median
    - ➢ MonotoneFilter
        - _CalcHist_imgyuv
            - _CalcCumulHist

Here application of routines FSWM, focus, median and Calchistogram are already discussed in chapter3. So here we will look at how these functions can we optimized using OpenCL.

4.8.1 Optimization of get_focus routine

For optimizing get_focus routine we wrote a kernel (example shown if *figure 2.17*) which is ported on GPU to get required speedup and we have followed following steps:

- ✓ Whenever a routine is called the following process is be repeated
    - ➢ GPU is initialized (only for the first call)
    - ➢ Input data is transferred to the GPU
    - ➢ Kernel is executed for corresponding number of threads
    - ➢ Results are read back into RAM

  This procedure is same as given in *figure 2.16*
- ✓ In the function, get-focus there is a for-loop to get focus value of each pixel in the image by using the median filtering.
- ✓ We removed that for-loop in the GPU kernel and now numbers of threads running are equivalent to the number of pixels in the image i.e. each pixel is executed on each single thread.
- ✓ All threads are independent; read inputs & write output to global memory; caching to local memory avoided for heterogeneity.

4.8.2 Optimization of getFSWM routine

After the previous step we get the focus value of each pixel in the image that is stored on GPUs global memory. Now we have to add all these value to get Sharpness values of complete image. If we done it on sequentially or using CPU, it takes time. So, we tried different methods for the summation in the second kernel in order to maximize the speedup. There are two methods for this addition.

a. Parallel Reduction **b. Vectorization**

1. Parallel Reduction

Parallel Reduction is the technique which reduces a set of numbers to a single value, e.g. find sum of all elements in an array.

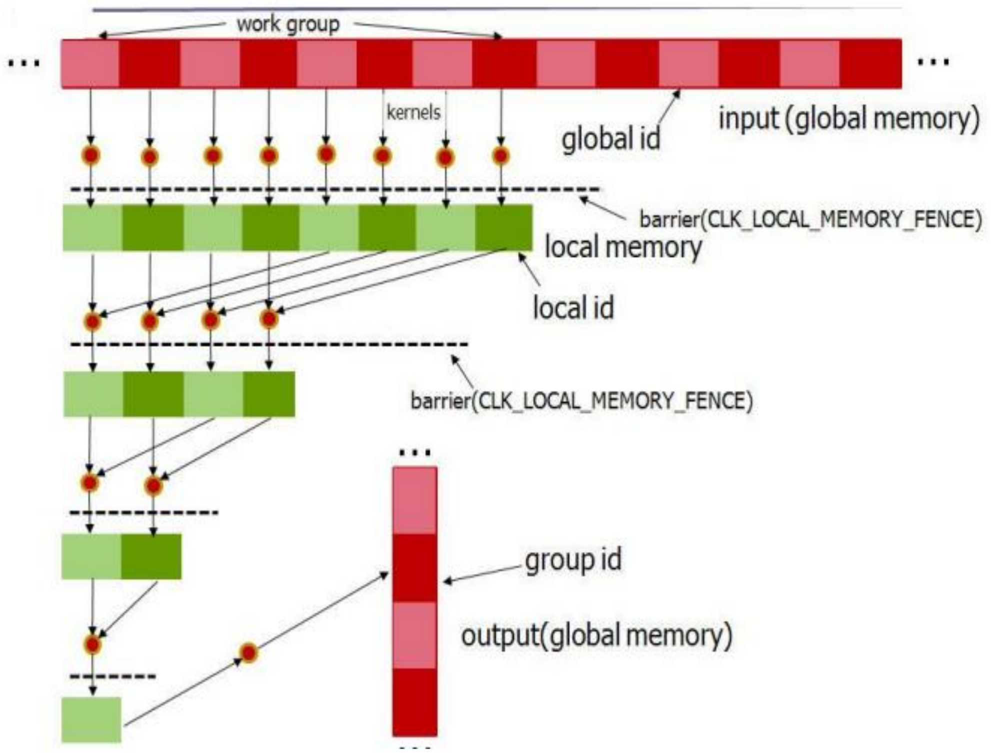


***Figure 4.3** parallel reduction techniques*

- ✓ As shown in figure initially we have input array (of focus values) that is output of first Kernel saved on global memory of GPU.
- ✓ This array is divided into no of workgroups. Firstly each element (work-item) in workgroup is saved on the local memory of GPU.
- ✓ Then half no of elements in the workgroup is added to other half no of elements as shown in *figure 4.3*
- ✓ Above divide and add procedure is carried on until we get final sum.
- ✓ This sum is saved on global memory of GPU at workgroup id corresponding to particular workgroup.
- ✓ Above procedure is executed for each workgroup in parallel and corresponding sum is saved at their workgroup id.
- ✓ The final value can we get by adding all of the values saved on workgroup ids.

- ✓ Because in this method local memory is used that reduces the timing for data read and write on GPU it is a faster process with utilizing parallel execution for each workgroup.

As we have seen that the parallel reduction technique is giving faster results, but we can't use this method because it has certain disadvantages as:

- ✓ It uses local memory for speedup process which is not possible for us. As mentioned (section 4.4), there is certain constraints, so we can't use local memory. We have to use Global memory which is slower than local memory.
- ✓ There must be a proper synchronization at each level of the reduction so that all the parallel threads can complete its work before going to next level of reduction as shown in *figure 4.3*. In standard format Local memory fencing (barrier) is used for this synchronization. But again due to constraint to use this code on Embedded Platform, we can't use local memory fencing. So we have used the Global memory fencing {barrier (CLK_LOCAL_MEM_FENCE)} which also reduces the speed of the reduction process.

As we can't use local memory as well as local memory fencing parallel reduction with Global memory use can't give required speedup for getFSWM. So here we have used the Vectorization method of OpenCL.

2. Vectorization

Vectorization is the process of combining some scalar data types together as vector data types for computation in the kernel which speedup the execution of the kernel program. Built-in vector data types are supported by the OpenCL implementation. The vector data type is defined with the type name i.e. char, uchar, short, ushort, int, uint, float, long, ulong followed by a literal value n that defines the number of elements in the vector. Supported values of n are 2, 3, 4, 8, and 16 for all vector data types [7].

Example1: char2, char4, float4, float8 etc.

Example2:

```
        float4  v, u;
        float   f;

                v = u + f;
        This will be equivalent to

        v.x = u.x + f;
        v.y = u.y + f;
        v.z = u.z + f;
        v.w = u.w + f;
```

And

```
         float4  v, u, w;
```

```
w = v + u;
```

Will be equivalent to

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
w.w = v.w + u.w;
```

To utilize vector data types in the kernel Vector Data Load and Store Functions are used. These allow us to read and write vector types from a pointer to memory. Functions used for this are:

vload***n***	Read vectors from a pointer to memory.
vstore***n***	Write a vector to a pointer to memory.

- ✓ We took the advantage of the vector load (vload8) and store (vstore8) capabilities of the GPU.
- ✓ In the second kernel for getting complete sharpness value of image we are reading the eight focus values together and summing them to one value in one GPU cycle. Similarly, reading another eight values in one GPU cycle. Then we add up these two values in one cycle.
- ✓ In this way, we achieved the summation of sixteen values in 3 cycles instead of 16 cycles, hence reducing computational time.

By using Vectorization we achieved good results than Parallel reduction.

4.8.3 Optimization of _CalcCummlHist routine

- ✓ One kernel thread is executed for 240 pixel(= Image height). Each thread calculates the partial histogram of 240 pixel values.
- ✓ OpenCL datatype float8 has been used
  - ➢ To take advantage of the vector load and store capabilities of the GPU. Thus, 8 values are loaded and divided in parallel to calculate histogram.
- ✓ Then partial histograms of each thread are further accumulated at the CPU to get complete Histogram.

- ✓ Although we didn’t get the expected speedup because the partial histograms are inherently dependent on each other.

4.9Acceleration of KLT

- ✓ To accelerate the KLT algorithm firstly we have profiled the code using above procedure given in section 4.6.2.

✓ Profiling output is shown in *figure 4.4*.

✓ As we can see from the call graph, the majority of the computing time is taken up by the two functions **KLTTrackFeatures()** and **KLTSelectGoodFeatures()** **(** which is called by **KLTReplaceLostFeatures())** and **_dodetection().**

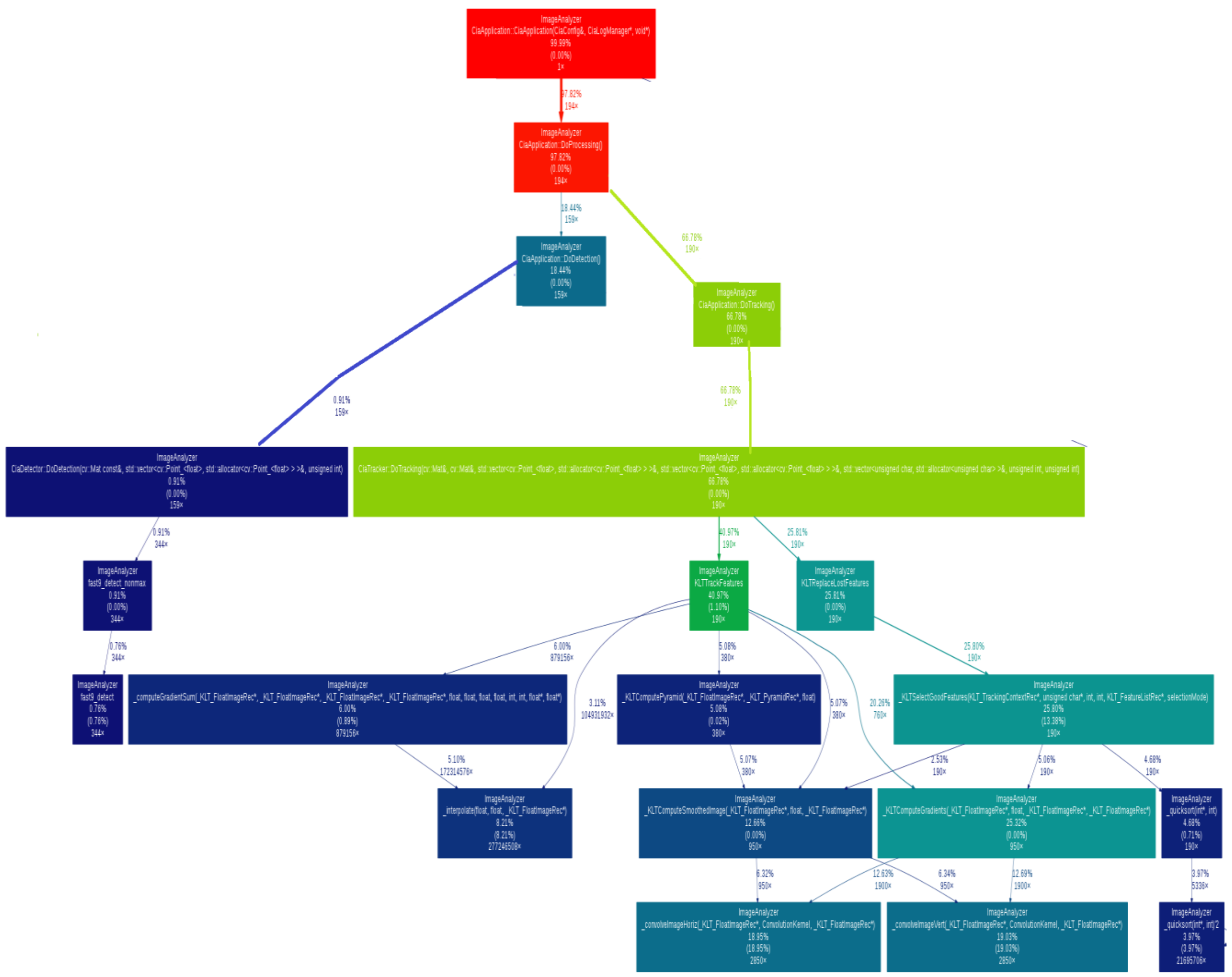

ImageAnalyzer CiaApplication::CiaApplication(CiaConfig&, CiaLogManager*, void*) 99.99% (0.00%) 1×
97.82% 194×
ImageAnalyzer CiaApplication::DoProcessing() 97.82% (0.00%) 194×
18.44% 159×
66.78% 190×
ImageAnalyzer CiaApplication::DoDetection() 18.44% (0.00%) 159×
ImageAnalyzer CiaApplication::DoTracking() 66.78% (0.00%) 190×
0.91% 159×
66.78% 190×
ImageAnalyzer CiaDetector::DoDetection(cv::Mat const&, std::vector<cv::Point_<float>, std::allocator<cv::Point_<float> > >&, unsigned int) 0.91% (0.00%) 159×
ImageAnalyzer CiaTracker::DoTracking(cv::Mat&, cv::Mat&, std::vector<cv::Point_<float>, std::allocator<cv::Point_<float> > >&, std::vector<cv::Point_<float>, std::allocator<cv::Point_<float> > >&, std::vector<unsigned char, std::allocator<unsigned char> >&, unsigned int, unsigned int) 66.78% (0.00%) 190×
0.91% 344×
40.97% 190×
25.81% 190×
ImageAnalyzer fast9_detect_nonmax 0.91% (0.00%) 344×
ImageAnalyzer KLTTrackFeatures 40.97% (1.10%) 190×
ImageAnalyzer KLTReplaceLostFeatures 25.81% (0.00%) 190×
0.76% 344×
6.00% 879156×
5.08% 380×
25.80% 190×
ImageAnalyzer fast9_detect 0.76% (0.76%) 344×
ImageAnalyzer _computeGradientSum(_KLT_FloatImageRec*, _KLT_FloatImageRec*, _KLT_FloatImageRec*, _KLT_FloatImageRec*, float, float, float, float, int, int, float*, float*) 6.00% (0.89%) 879156×
3.11% 104931932×
ImageAnalyzer _KLTComputePyramid(_KLT_FloatImageRec*, _KLT_PyramidRec*, float) 5.08% (0.02%) 380×
5.07% 380×
20.26% 760×
ImageAnalyzer _KLTSelectGoodFeatures(KLT_TrackingContextRec*, unsigned char*, int, int, KLT_FeatureListRec*, selectionMode) 25.80% (13.38%) 190×
5.10% 172314576×
5.07% 380×
2.53% 190×
5.06% 190×
4.68% 190×
ImageAnalyzer _interpolate(float, float, _KLT_FloatImageRec*) 8.21% (8.21%) 277246508×
ImageAnalyzer _KLTComputeSmoothedImage(_KLT_FloatImageRec*, float, _KLT_FloatImageRec*) 12.66% (0.00%) 950×
ImageAnalyzer _KLTComputeGradients(_KLT_FloatImageRec*, float, _KLT_FloatImageRec*, _KLT_FloatImageRec*) 25.32% (0.00%) 950×
ImageAnalyzer _quicksort(int*, int) 4.68% (0.71%) 190×
6.32% 950×
12.63% 1900×
6.34% 950×
12.69% 1900×
3.97% 5336×
ImageAnalyzer _convolveImageHoriz(_KLT_FloatImageRec*, ConvolutionKernel, _KLT_FloatImageRec*) 18.95% (18.95%) 2850×
ImageAnalyzer _convolveImageVert(_KLT_FloatImageRec*, ConvolutionKernel, _KLT_FloatImageRec*) 19.03% (19.03%) 2850×
ImageAnalyzer _quicksort(int*, int)'2 3.97% (3.97%) 21695706×

Figure 4.4: *Profiling output of KLT*

✓ During tracking sometimes there is a loss of some features due to a large residue, drifting out of bounds or because the computation fails for some reason etc. If it is desired to always maintain a certain number of features, then the lost features can be replaced by calling **KLTReplaceLostFeatures()**.This function calls the function KLTSelectGoodFeatures() to find the lost features in the image and place them accordingly in the feature list. Suppose, if *k* features have been lost, the *k* best features will replace them.

- ✓ Basically KLTSelectGoodFeature() function is a corner detector which is detecting new corners in the image so it can replace the lost one.
- ✓ This function uses the KLT detection algorithm for new corner detection algorithm.
- ✓ As seen by the profiling graph KLTSelectGoodFeature() is consuming the 25% of total tracking time, so it is not advisable to use such algorithm in Real time application which is consuming such time in only replacement of the lost feature.
- ✓ As we have discussed about the detection process in last chapter, we know that there is a solution for this problem by using the FAST algorithm.
- ✓ FAST algorithm can reduce the time needed for detection process very efficiently.

For reduce the time taken for replacing lost features, we have replaced the KLTSelectGoodFeature() function with the **FAST9()** routine and made some changes the code required for it. Here 9 is represents the no of contiguous pixels on the **Bresenham circle.** This gives us very good results.

Another problem related to this algorithm is no of feature lost are more than the reference code (OpenCV version of KLT, that requires extra time for computation of new feature (lost features) by calling detection routine which is not desired.

- ✓ By understanding the code thoroughly we got that there can be several factors for these feature loss. These factors are as follows:

    1. **KLT_SMALL_DET:** indicates that the feature has been lost due to the 2 by 2 gradient matrix (as discussed in chapter 3 equation no 3.13)having a small determinant.
    2. **KLT_MAX_ITERATIONS:** means that the feature has been lost because the number of iterations exceeded the maximum allowable (as shown in example of KLT in chapter 3).
    3. **KLT_OOB:** means that the feature has been lost because it was out of bounds (i.e., it was too close to the image border).
    4. **KLT_LARGE_RESIDUE:** means that the feature has been lost because the residue between the two feature windows was too large.

Here the tracking formula, Equation (3.12), is used to compute the displacement for many iterations of the algorithm. The origin of the starting image is then shifted by this displacement so that the tracking equation can be reapplied. Once the displacement converges near zero, tracking is complete (as shown in figure 3.8). If it does not converge in a few iterations, the algorithm fails.

The KLT algorithm has several numerical parameters that were chosen for this implementation. The window size was selected to be 7 pixels by 7 pixels. This parameter is the size of the region over which the summations in Equation (3.12) are evaluated. Tracking iterations are carried out until a single iteration has a displacement less than 0.1 pixels (min displacement). If 10 iterations (Max Iterations) are completed without a displacement less than 0.1 pixels, the feature is discarded.

After a feature is tracked, its residue is computed to determine if the image patch roughly matches the original feature. Residue is the absolute sum of the intensity difference between a feature x in Image I and the displaced feature in the Image J. Formula for residue calculation is

$$\mathrm{E} = \iint_{W} \left| J(x) - I(x-d) \right| dx \qquad \text{---------- } (3.17)$$

If the result divided by the area of the image is greater than 10.0 (max_residue), the feature is discarded.

In our case features are lost mostly because of failure of two conditions

✓ Max Iterations	=10	(in reference code)
✓ Max Residue	=10	(in reference code)

By debugging the code and checking the output of the algorithm for different condition we analyzed that if we increase the both above conditions up to certain values than there is a significant improvement in performance of KLT tracking.

So we have changed the values as:

Max Iterations	**=20**
Max Residue	**=30**

By selecting these values we have reduces the no of feature loss up to certain level so there is less no of calls for detection routine, results in reduction of time for tracking and detection as well.

One more problem is with this code is no of calls for convolution function. Convolution function, which is necessary both to smooth the image and to compute its gradients, is called 15 x times per frame during tracking, which consumes lot of time. So another task is to reduce the no of calls for this convolution function.

- ✓ As we have replaced the KLTSelectGoodFeature() with FAST9() algorithm there is a reduction in calls for convolution per frame. Because KLTSelectGoodFeature() function is using the convolution function many time for replace lost features.

- ✓ Convolution is unnecessary when an image sequence is being processed, because each image is processed more than once. For example, the features are tracked between frames 0 and 1, then between frames 1 and 2, then between frames 2 and 3, etc During each iteration the second image, after being processed, can be stored and recalled the next time as the first image.

- ✓ The tracking context has a flag called **sequentialMode** which, when set to TRUE, causes KLTTrackFeatures() to store the gradients of the second image, along with its smoothed version, into the tracking context. When KLTTrackFeatures() is called, it ignores its second parameter and replaces it with the previously stored image (except for the first time the function is called, in which case it must use both images). The computation is identical, but the speed is improved and no of calls for convolution function is reduced significantly.

Chapter-5

RESULTS ACHIEVED

5.1 Results for KFE

Table5.1: *KFE Results with 8 core GPU*

Kernel name	CPU cycles	GPU cycles	Total CPU Time for kernel in reference code(uSec)	Total GPU Kernel Execution time (usec)	Speed up CPU/GPU (8 core GPU)
Gpu_getFSWM	8.0025 Mega	4.3340 Mega	4000.1	3334.384	**1.85**
Gpu_getFSWM_sum	1.2756 Mega	0.3868 Mega	637.8	297.568	**3.30**
Gpu_hist_calc	0.8621 Mega	1.207 Mega	431.001	929.024	**0.71**

- ✓ Here speedup is calculated as the ratio of CPU cycle to GPU cycle.

Actual Speedup = CPU cycle / GPU cycle

- ✓ Where **GPU cycle** = GPU clock frequency * Total GPU Kernel Execution time
- ✓ And **CPU cycle** = CPU clock frequency * Total CPU Time for kernel in reference code

Note: Above results are obtained with following specifications:

Input video:

Format	:	YUV
Width	:	240
Height	:	320
FPS	:	15
No of frames	:	60

GPU	:	Quadro 295 (with 8 CUDA cores)
GPU clock frequency	:	1.3 GHz
CPU clock frequency	:	2.0 GHz

Table5.2: *KFE Results with 96 core GPU*

Kernel name	CPU cycles	GPU cycles	Total CPU Time for kernel in reference code(uSec)	Total GPU Kernel Execution time (usec)	Speed up CPU/GPU (96 core GPU)
Gpu_getFSWM	8.0025 Mega	0.6493 Mega	4000.1	541.184	**12.38**
Gpu_getFSWM_sum	1.2756 Mega	0.07257 Mega	637.8	60.480	**17.57**
Gpu_hist_calc	0.8621 Mega	0.92521 Mega	431.001	772.768	**0.94**

Note: Above results are obtained with following specifications:

GPU	:	Quadro Fx4600 (with 96 CUDA cores)
GPU clock frequency	:	1.2 GHz
CPU clock frequency	:	2.0 GHz

5.2 Results for KLT Tracking

As discussed in previous chapter we have replaced KLTSelectGoodFeature() routine with FAST9() routine to replace lost feature in frames , so we got following results.

- ✓ Time for tracking of all the frames in the video with KLTSelectGoodFeature() = **127.53 Sec.** in reference GPU code.
- ✓ Time for tracking of all the frames in the video with FAST9() = **94.43 Sec.**
- ✓ Thus we get the speedup of **1.35** get as compared to the reference GPU code.
- ✓ We get this speedup because there is a reduction in no of calls for convolution function by **3 x** time so total no of calls to convolution function is reduced to **12 x** from 15 x per frame.
- ✓ By the use of Sequential Mode Flag we have used the stored gradients of the second image, along with its smoothed version, so there is significant reduction in the tracking process as well as reduction in no of calls for convolution function.

- ✓ Now the total tracking time is **= 60.67 Sec**. with speedup of **2.10** with respect to the GPU reference code.

- ✓ Total no of convolution function calls reduced to **6 x** times than 15 x per frame

Note: Above results are obtained with following specifications:

Input video:

Format	:	YUV
Width	:	480
Height	:	640
FPS	:	12
No of frames	:	194
GPU	:	Quadro 295 (with 8 CUDA cores)

- ✓ Total no of time detection is called in the for complete video in the reference GPU code **= 159**
- ✓ By changing the conditions for maximum iteration (=20) and maximum residue (=30) we have reduced no of lost features per frame so there is no need for detection of features in most of the frames thus total no of time detection is called reduced to **55** , which is a good result compared to reference code.

Chapter-6

CONCLUSION AND FUTURE SCOPE OF WORK

6.1 Conclusion

As we get the results listed in the table5.1 shows that significant speedup for both kernels Gpu_getFSWM and Gpu_getFSWM_sum achieved**,** by executing it on NVIDIA Quadro NVS 295 with 8 cores.

This speed-up is improved with the increase in number of cores. As shown in table 5.2 with use of NVIDIA GPU Fx4600 (have 96 cores) speedup is very good for both the functions as compared to CPU as well as 8 cores GPU. Further the speed-up shall increase with size of input data, more data implies more SIMD processing to be exploited.

Most of the time wasted in transfer of input data from RAM to GPU, and reading the results back. The communication time is much high on the NVIDIA Quadro NVS 295. This is a fairly low-end GPU used with the intention to specially stress test the kernel. On high-end GPUs this time would decrease further because of better bandwidth.

Moreover, the KFE algorithm targeted in this work uses the histogram calculation which is inherently sequential, thus much speed-up can not be expected. But we can see from the results for both GPUs that speedup is approaching nearly equal to 1. So we can expect that with further increase in no of core of the GPU will results in the improved speedup for histogram calculation too.

From the results we got for KFE algorithm we can conclude that with use of GPGPU computing through OpenCL offers good performance improvement most of the time, and can we improve further with improvement of the GPU specifications.

From the results of the KLT tracking we find that use of FAST algorithm for the detection purpose can improve the performance of the algorithm, timing of the tracking as well as reduce the no of calling for undesired convolution function.

There is a significant improvement in the performance of KLT tracking with the change in conditions for max iteration and max residue. Thus reduction in no of feature lost in the consecutive frames.

6.2 Future Scope of Work

With the rising importance of GPU computing, GPU hardware and software are changing at a remarkable pace. In the upcoming years, we expect to see several changes to allow more flexibility and performance from future GPU computing systems. AMD and NVIDIA introduced support for double-precision floating-point hardware. The addition of double-precision support removes one of the major obstacles for the adoption of the GPU in many scientific computing applications.

In KFE algorithm, after extracting the key frame we have to retrieve the desired information from these frames. The method used for the information retrieval is called face detection module. So basically KFE is a algorithm used for detection of human motion in the sequence of video. Human motion is detected by face detection in the key frame. Here by profiling of the code we get that face detection module is the highly time consuming routine of KFE algorithm. In the future face detection module (FD_Module) can be optimized with taking all the advantages of the GPGPU computing.

In KLT there is large number of memory transfers from Host->GPU and from GPU->Host per frame. This is happening because the entire image is written to the GPU and the filtered image read out each time there is a filtering operation using the two convolution functions. Filtering operations are done for image smoothing and gradient calculation at each pyramid level. This could be optimized, for example, by calculating the entire pyramid in one launch of the kernel for smoothing and another for gradient computation. This needs to be done in future time for getting further optimization of KLT algorithm.

References

[1] G. Ciocca1 and R. Schettini, "An Innovative Algorithm for Key Frame Extraction in Video Summarization", *Journal Real Time Image Processing*, pg no 69-88, 2006.

[2] Jean-Yves Bouguet, “Pyramidal Implementation of the Lucas Kanade Feature Tracker – Description of the algorithm”, *Intel Corporation – Microprocessor Research Labs*, 2009.

[3] C. Tomasi and T. Kanade, “Detection and tracking of point features,” *Tech. Rep. CMU- CS-91-132, Carnegie Mellon University*, 1991.

[4]“https://computing.llnl.gov/tutorials/parallel_comp/#Abstract”, Accessed June, 2012.

[5] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips “Graphics Processing Units powerful, programmable, and highly parallel are increasingly targeting general-purpose computing applications.” *Proceedings of the IEEE* Vol. 96, No. 5, May 2008.

[6] Kristen Boydstun, “Introduction OpenCL”, *TAPIR, California Institute of Technology*, August 9, 2011.

[7] “http://khronos.org/registry/cl/specs/opencl-1.0.pdf”, Accessed June, 2012.

[8] M Born, E Wolf,”Principles of optics”, *Pergamon Press*, 435-42.1980.

[9] Maria Rudnaya, and Robert Ochshorn “Sharpness Functions for Computational Aesthetics and Image Sublimation” *IAENG International Journal of Computer Science, 38:4, IJCS_38_4_05* (*Advance online publication*: 12 November 2011).

[10]V. Aslantas, R. Kurban “A comparison of different focus measures for use in fusion of multi-focus noisy images” *The 4th International Conference on Information Technology (ICIT 2009), Amman, Jordan*, 2009.

[11] “http://docs.opencv.org/opencv_tutorials.pdf”, Accessed June, 2012.

[12] S. Birchfield, ‘Derivation of Kanade-Lucas-Tomasi tracking equation.”, “http://robotics.stanford.edu/~birch/klt/derivation.ps”, 1997. Accessed June, 2012.

[13] J. Shi and C. Tomasi. "Good Features to Track." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. pp. 593–600, June 1994.

[14]H. Moravec, "Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover". *Tech Report CMU-RI-TR-3 Carnegie-Mellon University, Robotics Institute,* “http://www.ri.cmu.edu/pubs/pub_22.html”, Accessed June, 2012.

[15]C. Harris and M. Stephens, "A combined corner and edge detector", *Proceedings of the 4th Alvey Vision Conference.* pp. 147–151. “http://www.bmva.org/bmvc/1988/avc-88-023.pdf”, Accessed June, 2012.

[16]S. M. Smith and J. M. Brady, "SUSAN - a new approach to low level image processing". *International Journal of Computer Vision pg no. 45–78*, May 1997.

[17]J. Shi and C. Tomasi, "Good Features to Track,". *9th IEEE Conference on Computer Vision and Pattern Recognition. Springer*. http://citeseer.ist.psu.edu/shi94good.html. June 1994. Accessed June, 2012.

[18] M. Trajkovic and M. Hedley. "Fast corner detection". *Image and Vision Computing* pg no 75–87, 1998.

[19] Edward Rosten, Reid Porter, and Tom Drummond, “Faster and Better: A Machine Learning Approach t o Corner Detection” *IEEE TRANSAC TIONS ON PATTERN ANALYSI S AND MACHINE INTELLIGENCE*, VOL. 32, NO. 1, pg no.105, Jan, 2010.

[20] E. Rosten and T. Drummond, “Fusing points and lines for high performance tracking,” *in 10th IEEE International Conference on ComputerVision*, vol. 2, pp. 1508–1515, 2005.

[21] http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/AV1011/AV1FeaturefromAcceleratedSegmentTest.pdf, Accessed June, 2012.

[22]“http://sepwww.stanford.edu/public/docs/sep70/rub/paper_html/node2.html”, Accessed June, 2012.

[23] “http://valgrind.org/docs/manual/QuickStart.html”, Accessed June, 2012.

[24] “http://code.google.com/p/jrfonseca/wiki/Gprof2Dot”, Accessed June, 2012.

[25] “http://www.graphviz.org/Documentation.php”, Accessed June, 2012.