HETEROGENEOUS BIG-DATA CLUSTER FOR COMPUTER VISION APPLICATIONS

By

Hazem Abdelmegeed Elsayed Abdelhafez

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2016

HETEROGENEOUS BIG-DATA CLUSTER FOR COMPUTER VISION APPLICATIONS

By
Hazem Abdelmegeed Elsayed Abdelhafez

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Prof. Dr. Hossam Ali Hassan Fahmy

………………………………..

Professor
Electronics and Communications
Engineering Department
Faculty of Engineering, Cairo University

Prof. Dr. Ameen Mohamed Nassar

………………………………..

Professor
Electronics and Communications
Engineering Department
Faculty of Engineering, Cairo University

Dr. Mohamed Mohamed Rehan

……………………………………

Chief Technical Officer
AvidBeam

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2016

HETEROGENEOUS BIG-DATA CLUSTER FOR COMPUTER VISION APPLICATIONS

By
Hazem Abdelmegeed Elsayed Abdelhafez

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Approved by the
Examining Committee

____
Prof. Dr. Hossam Ali Hassan Fahmy, Thesis Main Advisor

____
Prof. Dr. Ameen Mohamed Nassar, Member

____
Dr. Mohamed Mohamed Rehan, Member, Chief Technical Officer, AvidBeam

____
Prof. Dr Elsayed Eissa Abdo Hemayed, Internal Examiner

____
Prof. Dr. Khaled Mostafa Elsayed, External Examiner, Faculty of Computers and Information, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2016

Engineer's Name: Hazem Abdelmegeed Elsayed Abdelhafez
Date of Birth: 26/09/1990
Nationality: Egyptian
E-mail: hazem.abdelmegeed@gmail.com
Phone: +2 01001128853
Address: 4, Street 100, Maadi, Cairo, Egypt
Registration Date: 01/03/2014
Awarding Date: / /2016
Degree: Master of Science
Department: Electronics and Communications Engineering

Supervisors:

Prof. Dr. Hossam Ali Hassan Fahmy
Prof. Dr. Ameen Mohamed Nassar
Dr. Mohamed Mohamed Rehan

Examiners:

Prof. Dr. Hossam Ali Hassan Fahmy
Prof. Dr. Ameen Mohamed Nassar
Dr. Mohamed Mohamed Rehan, Chief Technology Officer, AvidBeam
Prof. Dr. Elsayed Eissa Abdo Hemayed
Prof. Dr. Khaled Mostafa Elsayed, Faculty of Computers and Information, Cairo University

Title of Thesis:

HETEROGENEOUS BIG-DATA CLUSTER FOR COMPUTER VISION APPLICATIONS

Key Words:
Big-data; Parallel Computing; Graphical Processing Units; Computer Vision; Heterogeneous Computing

Summary:

A token-based scheduler is developed to enable efficient utilization of graphics processing unit in big-data clusters specifically for computer vision applications. The scheduler addresses the racing conditions that occur on the graphics processing unit due to simultaneous access by the parallel instances of the running application. The presented scheduler enables the porting of computer vision applications to big-data cluster with heterogeneous computing capabilities where multi-core central processing units exist alongside graphical processing unit.

Acknowledgments

I would like to thank my mother and my family for their support and understanding during the time working on the MScs, without their support and love I wouldn't have crossed this far in my academic life and career. I would like to thank Dr. Hossam and Dr. Rehan for their endless support, patience, sincerity and dedication during my journey in this thesis. I would like also to thank all my colleagues at Avidbeam, Mohamed Fouad, Ahmed Sobhi and Ahmed Abdelsamad for helping me prepare the hardware setup I used in my thesis. Also I would like to thank Menna Ghoneim, Dina Helal and Rana Morsi for helping me understand computer vision algorithms better and for reviewing my thesis to make sure it is in perfect shape. I would like also to thank the management at Avidbeam, Dr. Hani El-Gebaly and Hossam Samy for providing all the help I needed and supporting me during my MScs journey. Finally, I am also thankful for the free, Open-Source community that provided freely most of the tools I used in this work and we gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

Dedication

Dedicated to my mother and my family ...

Table of Contents

Acknowledgments i

Dedication ii

Table of Contents iii

List of Tables v

List of Figures vi

List of Abbreviations ix

Abstract x

1 INTRODUCTION 1
1.1 Motivation and Overview 1
1.2 Problem Statement 4
1.3 Objective and Challenges 4
1.4 Thesis Organization 5

2 BIG-DATA TECHNOLOGY 6
2.1 Introduction 6
2.2 Hadoop 6
2.2.1 Background and Overview 7
2.2.1.1 Hadoop Distributed File System 7
2.2.1.2 Hadoop Map-Reduce Framework 9
2.2.1.3 Yet Another Resource Negotiator 9
2.2.2 Operation 10
2.3 Spark 12
2.3.1 Background and Overview 12
2.3.2 Operation 13
2.4 Storm 15
2.4.1 Background and Overview 15
2.4.2 Operation 17
2.5 Big-data Framework for Computer Vision 19
2.6 Conclusion 20

3 GRAPHICS PROCESSING UNIT 21
3.1 History and Background 21
3.2 Architecture and Operation 22
3.3 GPU Programming Frameworks 27
3.4 GPUs Role in Computer Vision Applications 30
3.5 Conclusion 33

4 LITERATURE REVIEW **34**
4.1 Introduction 34
4.2 Map-reduce Related Work 34
4.2.1 CUDA Framework 34
4.2.2 Other Frameworks 40
4.3 Generic Heterogeneous Computing with GPUs 40
4.4 Conclusion 43

5 DESIGN AND IMPLEMENTATION **44**
5.1 Introduction 44
5.2 Computer Vision Processing on Big-data Frameworks 44
5.3 Computer Vision Processing on Big-data Frameworks with GPU Support 48
5.3.1 Priority-based Static Scheduler 48
5.3.2 GPU Utilization-based Scheduler 50
5.3.3 Single Allocation Scheduler 52
5.3.4 Token-based Scheduler 54
5.4 Conclusion 55

6 EXPERIMENTAL RESULTS **57**
6.1 Introduction 57
6.1.1 Hardware Setup 57
6.1.2 Software Setup 58
6.1.3 Evaluation Procedure 61
6.1.3.1 GPU Scheduler for OpenCL framework performance evaluation 62
6.1.3.2 GPU Scheduler for CUDA framework performance evaluation 65
6.1.3.3 Token-based Scheduler Versus Nvidia MPS for CUDA Framework 68
6.2 Conclusion 70

7 CONCLUSION AND FUTURE WORK **71**
7.1 Conclusion 71
7.2 Future Work 72

References **73**

List of Tables

6.1 Master and slave nodes hardware specs 57
6.2 OpenCL framework performance metrics 65
6.3 Frame processing latency enhancement for OpenCL framework 65
6.4 CUDA framework performance metrics 67
6.5 Frame processing latency enhancement for CUDA framework 68
6.6 MPS performance metrics 70
6.7 MPS latency enhancement 70

List of Figures

1.1 The Digital Universe: 50-fold growth from the beginning of 2010 to the end of 2020 [6] . . . 2

2.1 HDFS architecture [17] . . . 8
2.2 Data management in HDFS [17] . . . 8
2.3 YARN architecture [20] . . . 10
2.4 Hadoop word-count example . . . 11
2.5 Spark software stack . . . 12
2.6 Spark cluster overview [25] . . . 14
2.7 Spark streaming workflow [27] . . . 15
2.8 Storm cluster overview . . . 16
2.9 Storm parallelism components . . . 17
2.10 Storm single layer topology . . . 18
2.11 Storm multiple layers topology . . . 18

3.1 Graphics rendering tasks migration from CPU to GPU [33] . . . 21
3.2 GeForce3 programmable architecture [34] . . . 23
3.3 Unified GPU architecture [33] . . . 24
3.4 Stream processor architecure [33] . . . 25
3.5 Nvidia Multi-process Service architecture [39] . . . 26
3.6 CUDA thread hierarchy [33] . . . 28
3.7 OpenCL architecture . . . 29
3.8 OpenCV performance comparison [33] . . . 32
3.9 CUDA speedup versus CPU [33] . . . 32
3.10 OpenCL speedup versus CPU [33] . . . 33

4.1 GPMR multi-stage map-reduce pipeline [53] . . . 36
4.2 MGMR workflow on multi-GPU system [55] . . . 38
4.3 OCLScheduler client-server model [65] . . . 41
4.4 Hyperflow architecture [66] . . . 42

5.1 Map-reduce workflow in computer vision applications . . . 46
5.2 CPU based Storm topology for computer vision processing . . . 47
5.3 GPU priority-based scheduler . . . 49
5.4 GPU utilization based scheduler workflow . . . 50
5.5 Storm topology with utilization-based GPU scheduler . . . 51
5.6 Single allocation GPU scheduler algorithm . . . 53
5.7 Storm topology with single allocation GPU scheduler . . . 54
5.8 Token-based GPU scheduler algorithm . . . 56

6.1 Evaluation test-bed setup . . . 58
6.2 Example of rectangular features . . . 59
6.3 Mutli-stage cascade classifier . . . 60
6.4 CPU-only Storm topology . . . 60

6.5 Hybrid CPU-GPU Storm topology 61
6.6 OpenCL performance evaluation 63
6.7 CUDA performance evaluation 66
6.8 MPS performance evaluation against token-based scheduler for CUDA framework 69

List of Abbreviations

AdaBoost	Adaptive Boost
AM	Application Master
APIs	Application Programming Interfaces
ASF	Apache Software Foundation
AVX	Advanced Vector Extension
BSD	Berkeley Software Distribution
Cgroups	Linux Control Groups
CUDA	Compute Unified Device Architecture
CUDASA	Computer Unified Device and Systems Architecture
CUs	Compute Units
DAGs	Directed Acyclic Graph
DSL	Domain-Specific Language
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
FPS	Frames per Second
GFS	Google File System
GPGPU	General Purpose Computing on GPU
IGP	Integrated Graphics Processor
IPP	Intel Performance Primitives
JAR	Java Archive
JSON	JavaScript Object Notation
JVMs	Java Virtual Machines
LPR	License Plate Recognition
MPS	Nvidia's Multi-process Service
ms	milliseconds
NM	Node Manager
OCR	Optical Character Recognition
OpenACC	Open Accelerators
OpenCL	Open Computing Language
OpenCV	Open Computer Vision
OpenGL	Open Graphics Library
PCI	Peripheral Component Interconnect
PEs	Processing Elements
PMGMR	Pipelined Mutli-GPU Map-Reduce
RDD	Resilient Distributed Dataset
RM	Resource Manager
SDK	Software Development Kit

SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SSE	Streaming SIMD Extensions
SSSE3	Supplemental Streaming SIMD Extensions
TOM	Task-oriented Modules
VPE	Virtual Processing Elements
YARN	Yet Another Resource Negotiator

Abstract

Big-data technology in recent years has become increasingly utilized to process huge amount of data in a timely manner. The growth in the amount of visual data - videos and images - generated nowadays raises the need for porting computer vision applications to big-data frameworks in order to increase the processing throughput of these applications.

On the other hand, developers and the scientific community have already ported many computer vision algorithms to the Graphics Processing Unit (GPU) that successfully accelerates these algorithms thanks to its data-parallel architecture.

Combining big-data with GPUs to scale computer vision applications both horizontally and vertically yields a promising architecture for processing the huge amounts of visual data and to fulfill the urging need to mine these data for underlying patterns and information. Unfortunately, the number of GPUs available on a typical processing node in a big-data cluster is limited and most of the time there is only one GPU on such nodes. Therefore, multiple instances of the same computer vision application running on top of any big-data framework leads to competition between these instances on the scarce GPU resource.

In this thesis, we address the challenge of combining GPUs with big-data technology in order to accelerate the processing of computer vision applications. We introduce a GPU scheduler that is responsible for assigning the GPU to multiple instances of the computer vision application efficiently with minimal competition and best performance compared to using either the GPU or the Central Processing Unit (CPU) solely. In order to achieve this we propose a token based scheduler that guarantees that no competition occurs on the GPU. The evaluation shows increased processing throughput up to 2.3x compared to CPU-only big-data processing with 24 cores, 2.1x compared to CPU-GPU big-data processing and up to 32x compared to the sequential processing on a single CPU core.

Chapter 1: Introduction

1.1 Motivation and Overview

Recently, the amount of available visual data has increased drastically. According to Cisco Visual Networking Index [1] video data represented 64 % of internet traffic in 2014 with expected growth by more than 10 % in 2019 to reach 80 to 90 % of all IP traffic. This is due to the massive amount of videos uploaded and shared by individuals and enterprises on the internet. It is expected that by 2019 there will be approximately one million minutes of video data crossing the internet every second [1].

In addition to public video content, there is another domain of expanding video content, namely video surveillance. Most enterprises and businesses currently adopt video surveillance solutions and products for security and business monitoring purposes. There are three key drivers for the ongoing adoption of video surveillance solutions, these drivers are: business expansion, security and safety, and operational efficiency [2]. This growth in the adoption of video surveillance solutions is accompanied by an increasing importance of video analytics by nearly 74 % [2]. Video analytics is a term that describes the processing of video data to extract relevant information, for example: Face recognition, License Plate Recognition, Object Tracking, Motion Detection ... etc.

The advances in machine learning, pattern recognition and statistical analysis has significantly helped in advancing data mining science. Data mining is simply the science of extracting important information hidden in huge amounts of data records that are generated at a very large scale in different domains and stored in databases [3, 4]. These domains include Engineering, Biology, Astronomy, Finance and others. The extraction of this information helps decision makers in these fields to make better decisions. Rao [5] and the authors discuss the importance of analyzing medical images such as: MRI scans, X-Rays ... etc, in order to identify and early diagnose tumors and malignant formations in the human body. This helps the doctors to come up with better treatment plans for the patients and save people's lives. Data mining techniques coupled with high performance computing clusters can help scientists and analysts build predictive models or extract unknown information from existing data sets in a timely manner.

In this context, the processing of video data using computer vision algorithms such as: License Plate Recognition, Face Recognition ...etc, produces structured information. For example, in a license plate recognition application, a car that appears in a video stream

and is represented as binary data encoded in any video format, will be transformed to a data record of (license-plate-number, timestamp). This kind of transformation opens up the data mining paradigm for computer vision applications and enables the extraction of deeper insights about the world that is monitored by cameras starting from highways surveillance, drones, satellites, passing by GoPro and mobile cameras, and ending with enterprises and residential buildings surveillance systems.

It is estimated that from 2005 to 2020 the amount of digital data generated world wide will grow from 130 exabytes to 40,000 exabytes (1 exabytes = 1 milion terabytes) and therefore the need for more capable processing paradigms will continue to grow [6]. Figure 1.1 illusrtates the growth in the amount of digital data from 2009 and the expected trend up to 2020. Big-data is a term describing the amount of data that the traditional computing and storage digital systems cannot accommodate [7]. It cannot be defined as a standard number of gigabytes or even terabytes. It rather refers to any amount of data that needs non traditional computing and storage systems. Non traditional in this context means consolidated hardware farms, or high performance computing clusters. These clusters are capable of storing and analyzing huge amounts of data concurrently, thus increasing the overall processing throughput (Rate of data items processed per specific time unit).

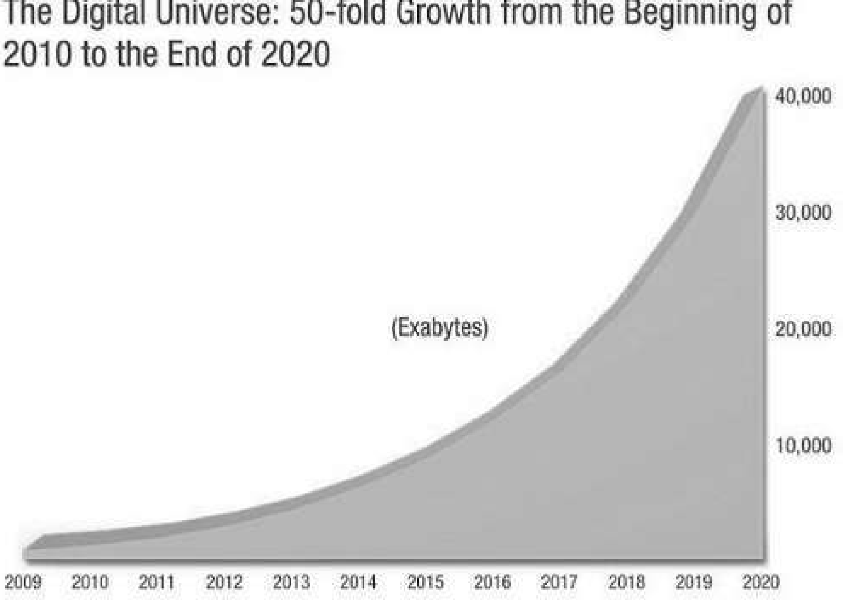

Figure 1.1: The Digital Universe: 50-fold growth from the beginning of 2010 to the end of 2020 [6]

A typical 1080p video has a frame size of 1920 X 1080 which means it has a total of

2,073,600 pixels, and to represent colors in each pixel a combination of red, green and blue is used where each color ratio is described as a value between 0 and 255, therefore each color needs at least one byte to be represented. Hence, a frame size is at least 6075 kilobytes. In a 1080p a typical frame rate is 30 frames per second, thus 1 second in a 1080p video is nearly 178 megabytes. In a typical enterprise building or retail mall there is at least 200 cameras feeding video data 24/7, which means that daily we have about 2933 terabytes of video data residing on digital storage devices. Most computer vision algorithms depends on applying the same instruction of transformation on each pixel of the frame, thus computer vision applications lie under the category of Single Instruction Multiple Data (SIMD) applications.

Nowadays, most computer vision libraries and applications rely on hardware accelerators to leverage the SIMD capabilities of the computational resources. In 1999, Intel introduced Streaming SIMD Extensions (SSE) which is an SIMD instruction set extension to the x86 architecture. Subsequent extensions such as SSE2, SSSE3, SSE4 and Advanced Vector Extension (AVX) were introduced later in modern Intel CPU families. The main target of these extensions is to enable the processing of multiple floating point data at the same time by using single instruction. In other words, these extensions increase the parallelism and enhance the throughput of SIMD operations.

In the same context, an important hardware accelerator that is used to for speeding up computer vision applications is the GPU. A GPU consists of multiple processors, typically more than 128 processors, to perform multiple instructions concurrently. Traditionally, these processors were limited to a certain number of instructions specifically the instructions related to graphics processing. Also, GPU cores exhibit a slower frequency when compared to CPUs. However General Purpose Graphics Processing Unit has become increasingly used to perform operations that were only performed by CPUs. Typically on a big-data node there are multiple CPUs with multiple cores per CPU and a single GPU device. Hence, the racing conditions were observed when processing computer vision applications on big-data clusters with a single GPU per node.

The fusion between computer vision and big-data technologies is an emerging trend in the last couple of years. There is a huge consensus that video big-data applications will produce important information that shall greatly serve humanity in many different fields.

1.2 Problem Statement

The main problem we are addressing in this thesis is the racing and competition between multiple threads or processes in big-data frameworks on the GPU device specifically for computer vision applications. We approach this problem by prohibiting any racing among multiple big-data application instances on the GPU device. In big-data frameworks a typical application workflow starts by dividing the input data into multiple pieces then multiple threads or processes are spawned to analyze these pieces concurrently. On a big-data cluster node there are multiple CPU cores that execute the threads therefore there is no racing on the CPU cores as long as the number of threads is smaller than or equal to the CPU cores. However, on each node typically there is one GPU. This GPU is considered a bottleneck when multiple processes or threads try to utilize it concurrently.

1.3 Objective and Challenges

The main objective of this thesis is to enhance big-data processing on a heterogeneous big-data cluster that incorporates multi-core CPUs and GPUs with specific focus on computer vision algorithms due to its computational complexity.

We faced some challenges in order to reach this objective. We can summarize these challenges as follows:

1. Enabling computer vision algorithms processing on heterogeneous big-data clusters: In order to process visual data on big-data clusters, one should think of how to re-group the parallelized data and aggregate the results correctly. Also, the selection of the data structure and big-data framework to be used varies based on the requirements at hand such as real-timeliness.
2. Enhance processor sharing among multiple instances in a heterogeneous cluster: In big-data frameworks the workload on the multiple processes or threads analyzing the data can vary according to several factors. One of the main challenges in processing visual data on big-data clusters is how to distribute the workload fairly amongst the CPU cores.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 1 is the thesis introduction. Chapter 2 presents a detailed background about state-of-art big-data technologies. Chapter 3 discusses the GPU from both hardware and software perspectives. Chapter 4 discusses previous work and provides literature review on enabling GPUs in big-data clusters. Chapter 5, presents the system design and implementation of the work described in this thesis and Chapter 6 shows the results of the experiments done using the proposed system. Finally, Chapter 7 concludes the work done in this thesis and proposes possible ideas for future work.

Chapter 2: Big-data Technology

2.1 Introduction

Nowadays, many frameworks and software libraries exist to help engineers and scientists process big-data in a short time, these technologies depend on concurrent and parallel processing models. The architecture of most of these frameworks is a cluster of several computers or nodes connected through a network together. A typical cluster consists of at least one master node multiple slave nodes. A master node is responsible of launching user applications and managing the distribution of the workload among the slave nodes; It also manages the resources allocation for the application besides the control of the application life-cycle from its start to its end. The data is distributed on different machines and then processed by multiple instances or threads of the user defined processing application. In the following sections three of the most common big-data processing frameworks are explained in details. These frameworks are: Apache Hadoop [8], Apache Spark [9] and Apache Storm [10], the three are incubated by Apache Software Foundation (ASF) .

2.2 Hadoop

The official description of Hadoop as stated on the official website Apache Hadoop is: ”The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.”[8].

Hadoop originates from Apache Nutch [11] which is a web search engine and a component of Apache Lucene [12]. All of them are created by Doug Cutting who named Hadoop after his son’s stuffed yellow elephant name [13]. There are many corporates currently powered by Hadoop such as: Amazon, Alibaba, Adobe, Yahoo, Facebook and Google [14].

2.2.1 Background and Overview

Hadoop is a parallel programming framework for processing big-data by distributing data on several computing nodes and dividing the compute load among these nodes. Hadoop is primarily targeting batch processing of the subject data. In batch processing model, big-data is divided into several batches and a Hadoop application initializes, starts execution then terminates for each batch until the whole input data is processed. Hadoop consists of three main modules and a set of common utilities that support these modules. The three modules are, Hadoop Distributed File System (HDFS) [16], Map-reduce framework [18] and Yet Another Resource Negotiator (YARN) [19].

2.2.1.1 Hadoop Distributed File System

HDFS is a scalable, fault-tolerant and resilient network based file system designed to be deployed on commodity servers. It was inspired from Google File System (GFS) [15] which is a distributed, highly-scalable and fault tolerant file system developed by Google engineers to process data-intensive software applications. Hadoop separates the metadata and the actual data on two different types of nodes. The metadata is stored in a node called Namenode and the data itself is stored on a node named Datanode. The architecture of HDFS follows a master/slave paradigm where a master node represents the Namenode and slaves represents Datanodes [16].

Namenode is responsible of managing HDFS; It provides namespace for HDFS clients. Clients can then send requests to read, write and other file operations to the Namenode which in turn maps these requests to Datanode that serves the client request. In HDFS, data is split into several blocks, the split size and number of blocks are configured by HDFS client.

A Datanode is responsible of managing the underlying hard drives in order to store the blocks and serve the read and write requests from HDFS client. The Application Programming Interfaces (APIs) for HDFS provide UNIX file system operations and were extended over time to provide better performance [16]. The over all architecture of HDFS is depicted in Figure 2.1.

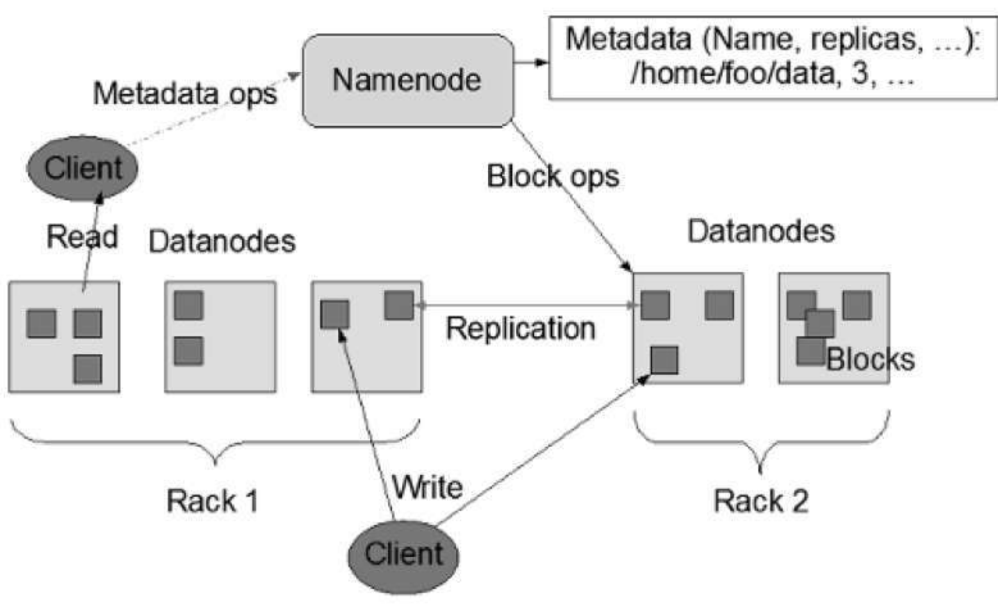

Figure 2.1: HDFS architecture [17]

In HDFS, data is split into several blocks of configurable size then stored on Datanodes. These blocks are replicated across Hadoop cluster Datanodes based on a replication factor set by the system administrator. The data splitting and block replication is depicted in Figure 2.2

Block Replication

Namenode (Filename, numReplicas, block-ids, ...)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...

Datanodes

1 2 | 2 | 1 4 | 2 5
5 3 | 4 | 3 5 | 4

Figure 2.2: Data management in HDFS [17]

2.2.1.2 Hadoop Map-Reduce Framework

Hadoop map-reduce is a software framework for processing large data sets in parallel by splitting input data into independent splits that can be processed concurrently on multiple processing nodes. map-reduce framework was initially developed by engineers at Google to process massive amounts of data. In map-reduce applications, the developer or the client writes a map function that processes a set of key-value pairs and emits intermediate key-value pairs that are further aggregated from all maps and then processed by reduce function [18].

In this context, Hadoop map-reduce exposes a set of classes and APIs to Hadoop user in order to write applications that process big-data following the map-reduce paradigm; It mandates specific formats for the input and output data, for example: FileInputFormat, SequenceFileInputFormat and TextInputFormat. A more detailed explanation of how map-reduce works in Hadoop with example is given in section 2.2.2.

2.2.1.3 Yet Another Resource Negotiator

YARN, is a resource management library for Hadoop map-reduce applications. In the early versions of Hadoop - versions 1.x - the resource management and application management functionalities were part of Hadoop map-reduce component. However, starting from Hadoop versions 2.x and onward, the resource and application management were handed off to what is called NextGen map-reduce framework or YARN. The main idea was to decouple the resource and application management from the map-reduce programming model [19].

YARN was also adopted by other big-data frameworks and libraries like Apache Spark, Giraph, Tez and others. In [19], the requirements and motivation behind developing YARN are discussed in details, some of these requirements are in fact shortcomings in the old Hadoop map-reduce framework as a resource, application and task manager, and some of them are related to new requirements such as support for flexible programming models not restricted to map-reduce model. YARN architecture consists of three components as depicted in Figure 2.3, these components are:

1. Resource Manager (RM) : RM is a per cluster daemon running on the master node. It manages the cluster resources and assign containers to the applications running on the cluster. These containers are a logical representation of the computational

resources ”Processor Cores and Memory” on a specific slave node in the cluster.

2. Node Manager (NM) : NM is a daemon running on each slave node in the cluster. It is responsible of reporting the available resources on its node to the RM. It also starts the containers when the application starts execution and kills them on application termination, besides reporting any failure in the containers. Hadoop user can specify the computational resources available on slave node by editing a configuration file read by YARN daemons on start-up.

3. Application Master (AM) : AM is the head of the application, it manages the application life cycle from its submission till its termination. The AM negotiates the application required resources with the RM, and after being allocated the containers by the RM, the client application can start.

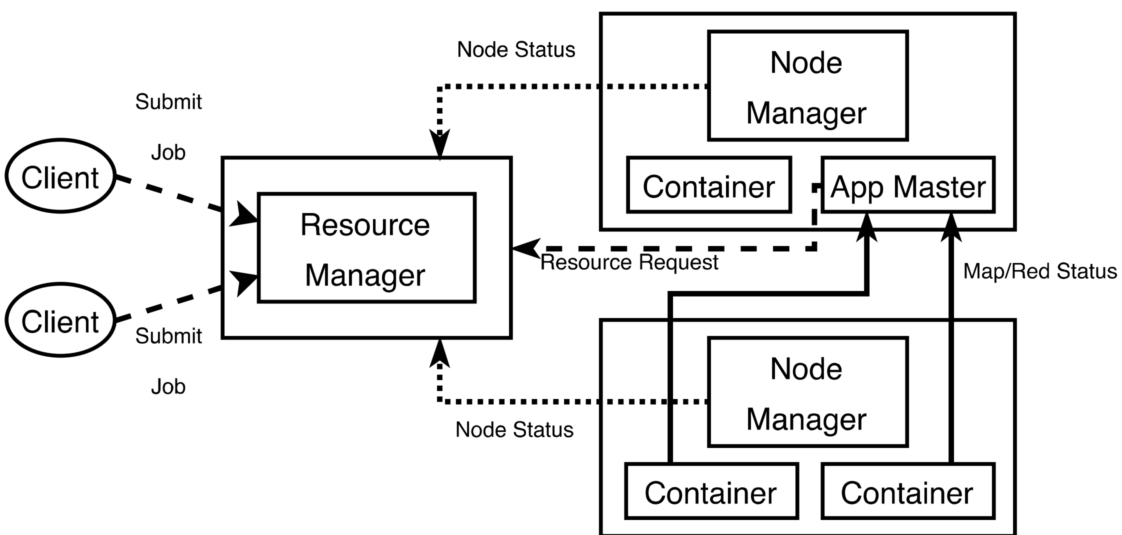

Figure 2.3: YARN architecture [20]

2.2.2 Operation

A map-reduce application typically consists of a map function, reduce function, input folder and output folder on HDFS. A hello world application on Hadoop is a word count example, where the developer writes a map function that counts the occurrences of a certain word in a file and a reduce function that collects the results from each map function and then aggregates the word count for the duplicate words in multiple files. The developer specifies an input folder that contains multiple text files and an output folder for the

map-reduce application in order to save the final result there. An illustration of the word count application is given in Figure 2.4.

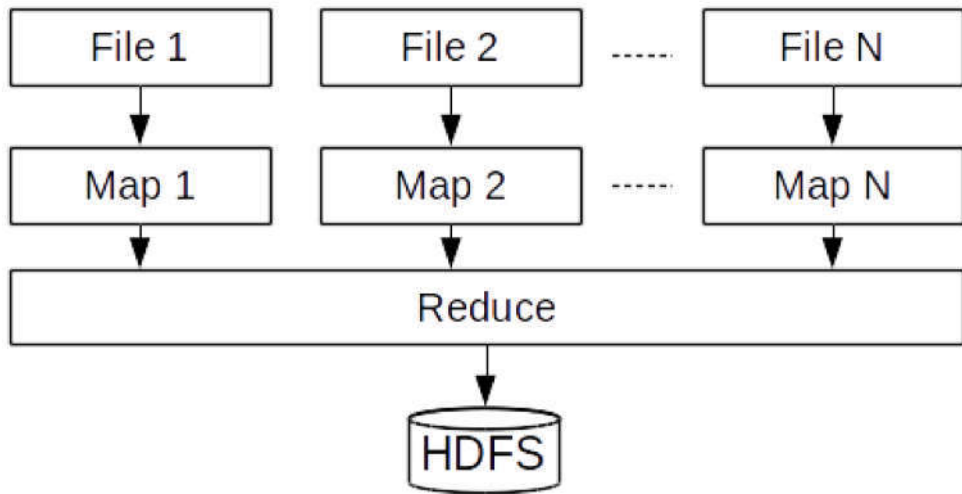

Figure 2.4: Hadoop word-count example

In Hadoop, whenever an input directory or file is being specified for Hadoop application, the application accesses HDFS and loads the input data files based on their format. The next step is splitting the input data according to the configured block size option that controls the minimal building block for the files stored on HDFS. For example, if we have a text file size of 1 megabytes and block size of 128 kilobytes, then this file will be split and stored on HDFS as 8 blocks. Hadoop APIs are provided officially in Java programming language, however, there are other third party libraries for Python, C++ and other languages. Moreover, developers can write map-reduce applications in any language and interface with Hadoop framework through Hadoop streaming feature [21] which uses system's standard input and standard output interfaces to communicate with non-Java applications.

Hadoop follows a batch processing model where Hadoop user specifies a batch of input data to be processed at once. Thus, Hadoop cannot process a stream of incoming data through network interface or any other data source.

2.3 Spark

Apache Spark is an open source, fault-tolerant, distributed cluster computing framework. It was originally developed at the University of California, Berkeley's AMPLab in 2009. In 2013, Spark code was donated to Apache Software Foundation. Currently, Spark is one of the top active projects in big-data technologies. Its support for flexible programming models other than map-reduce has made it one of the most widely used big-data technologies. Spark exposes a set of APIs in order to facilitate developing big-data applications for streaming data such as: Stock data, Social Media activities, Embedded Sensors ...etc [22] [9] [23].

2.3.1 Background and Overview

Spark applications rely on fault-tolerant distributed data structure called Resilient Distributed Dataset (RDD) . RDDs are a collection of objects that can be persisted in memory or in hard disks of the distributed nodes in the cluster, thus whenever a partition is lost from the RDD, Spark can reconstruct this partition from the rest of the nodes. These RDDs represent the input, output and intermediate data throughout Spark application lifecycle.

Spark programming model is more flexible than Hadoop as it is not limited to map-reduce framework [23]. Besides map-reduce programming model support, Spark also provides other APIs and methods that developers can use to create their own algorithms with greater flexibility than Hadoop [24]. There are also other libraries used with Spark such as: Mlib which is a library used to develop machine learning applications, GraphX a library for graph processing applications, Spark Streaming a sub-module of Spark that enables streaming applications to run on top of Spark framework. Figure 2.5, illustrates the building blocks of Spark software stack.

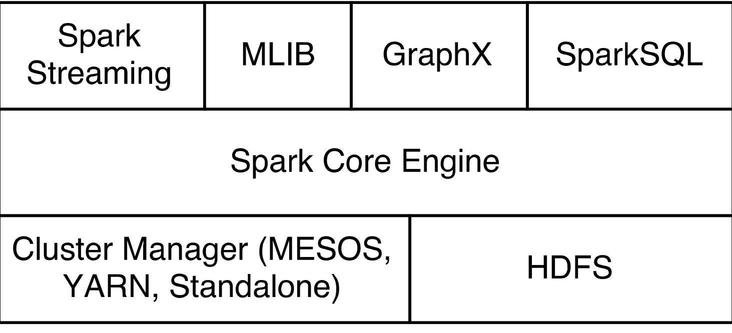

Figure 2.5: Spark software stack

Spark architecture follows a master/slave paradigm same as Hadoop does. Spark applications rely on two main components:

1. Driver Program: It is the program that contains the main method of the user application which creates SparkContext object. SparkContext object manages the Spark application which consists of a group of parallel processes running on the cluster nodes. Driver Program in this context is very similar to Application Master in Hadoop. A Driver Program runs on a master node and the application processes run on worker nodes which are equivalent to slave nodes in Hadoop.

2. Cluster Manager: Spark has its own standalone cluster manager that manages the allocation of computation resources to the application. It supports YARN and Apache MESOS cluster managers too, thus Hadoop and Spark applications can coexist on the same cluster using YARN as the cluster resource manager. In addition to YARN support, Spark provides APIs for accessing complete features of HDFS [23]. Hence, Spark can use HDFS as the data storage service for the input data and output results of Spark applications. Spark can read any data format supported by HDFS.

2.3.2 Operation

In Spark, an application initiates a SparkContext object in the driver program. Next, the SparkContext object communicates with a cluster manager to acquire executors for the application. Executors are Java Virtual Machines (JVMs) that are created on application launching and keep alive till the end of the application lifetime. Whenever the executors are acquired, SparkContext starts assigning tasks to these executors which process these tasks in parallel threads. Each small set of tasks are a Stage, a group of stages construct a Job. A Job is a parallel computation unit created whenever the user application performs a computation action on the input data. Figure 2.6 illustrates the cluster components for Spark and the building blocks for Spark applications.

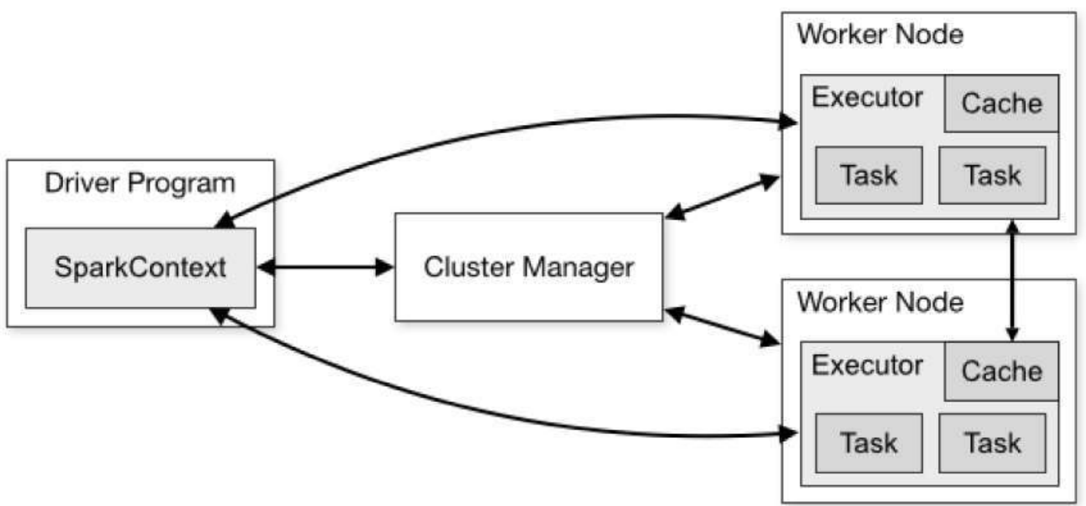

Figure 2.6: Spark cluster overview [25]

An RDD in Spark is the main unit for any processing carried on by Spark application. RDDs can be constructed by any of the following methods:

1. External Datasets: By reading a file from HDFS, any HDFS file format such as TextFileInputFormat, SequenceFileInputFormat ...etc. is supported by Spark applications with ready-to-use APIs.

2. Transformation: A transformation is an operation performed on an RDD and returns a new RDD that can be used in further transformations or actions.

3. Parallelized Collections: An RDD can be created by parallelizing an Array of items in the application driver program. The output collection can then be processed in parallel across different nodes. For example:

```
data = [1, 2, 3, 4, 5]
newData = sparkContext.parallelize(data)
```

In this example, the ”data” array is transformed into a ”newData” RDD which can be processed in parallel according to the user application logic.

Spark application basically is a series of operations performed on an input RDD. The type of operations supported on an RDD are: Transformations and Actions. A Transformation such as ”map()” takes an RDD object and apply user logic on it then returns a new RDD that is a transformation of the original RDD. An Action such as

”reduce()” aggregates the results from transformation steps and returns the final results to the driver program. Spark APIs provide several transformations and actions that the developer can utilize to create a complex algorithm to process the input data. Spark core APIs support different programming languages such as: Scala, Java, Python and R.

In Spark streaming applications, Spark exposes a set of APIs in order to process a stream of RDDs or ”Discretized Stream” [26] received from a data source and then stores the output results in an output sink. Spark streaming process a micro-batch of RDDs as depicted in Figure 2.7. Spark streaming support several input sources such as: Kafka, Flume, Amazon S3 and Kinesis, HDFS and Twitter. It supports output sinks such as: Databases, Dashboards, HDFS ...etc.

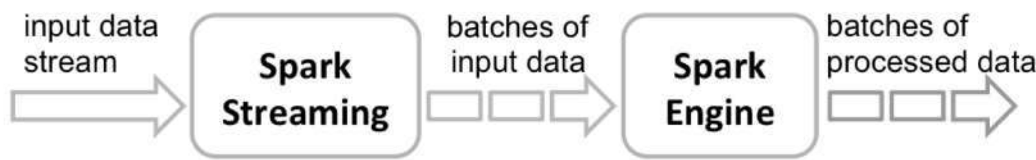

Figure 2.7: Spark streaming workflow [27]

2.4 Storm

Apache Storm is a framework for real-time stream processing of live messages from various data sources. Storm was originally developed by Nathan Marz and Backtype then it was moved to Twitter with the acquisition of BackType by Twitter. Storm has been incubated by Apache Software Foundation since September 2013. Storm applications are defined as topologies which are Directed Acyclic Graphs (DAG) equivalent to Hadoop applications. In these topologies there are two main elements: Spout and Bolt. Spout represents data source in Storm topology and Bolt represents a processing element [28] [29].

2.4.1 Background and Overview

Storm follows a master/slave paradigm where on a master node the management services of Storm framework runs and handles the interaction between the user and the framework.

Storm services are: Nimbus, Supervisor, UI, and Logviewer. UI is a web application service that views the current status of Storm topologies and cluster overview. Logviewer is a service that fetches the logs created by the workers of Storm topologies and serve them to the user whenever they are requested through the UI [30].

Nimbus is a service responsible of managing Storm topologies and Storm client connects to Storm cluster through it. It also schedules the components "Spouts and Bolts" of Storm topologies across the cluster nodes. It send start/stop signals to supervisors in order to start workers which are JVMs that contain the executors of Storm topologies. An executor represents either a Spout or a Bolt. Nimbus also distributes the applications' Java Archive (JAR) and dependencies across the worker nodes.

Supervisor is a service responsible of running the workers scheduled by Nimbus on the slave nodes. It monitors the components and restarts any component that fails. In order to allow the Supervisors running on slave nodes to detect Nimbus service for the cluster, Storm services use Zookeeper which is a service orchestration and discovery library used by many big-data frameworks. Figure 2.8 shows a typical Storm cluster and the mapping of Storm services on the clusters' nodes [31] [10].

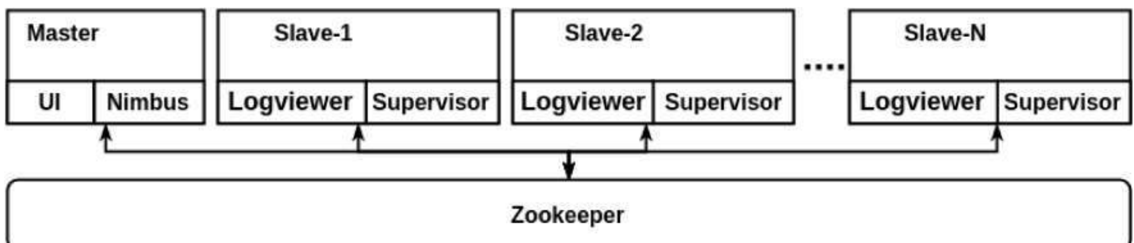

Figure 2.8: Storm cluster overview

Until version 0.10.0 of Storm, there was no official support for cluster managers such as Mesos or YARN. However, it is planned that in the next releases Storm will have its own cluster manager by adding a resource aware scheduler to the Supervisor configuration and enforcing the usage of the computation resources in terms of CPUs and Memory using Linux Control Groups (cgroups) library which is a library of software used in Linux operating systems to control the resource utilization of Linux processes. In version 0.10.0 and preceding versions, storm defined workers as JVMs that run executors. Each JVM has a port to communicate with Storm cluster through it. Inside each worker, Storm can launch at least one executor. Each executor is a thread spawned by the worker and it run a task related to a Spout or a Bolt. Ideally, each executor should execute only one task for either a Spout or Bolt. Figure 2.9 illustrates the different parallelism levels achievable in Storm.

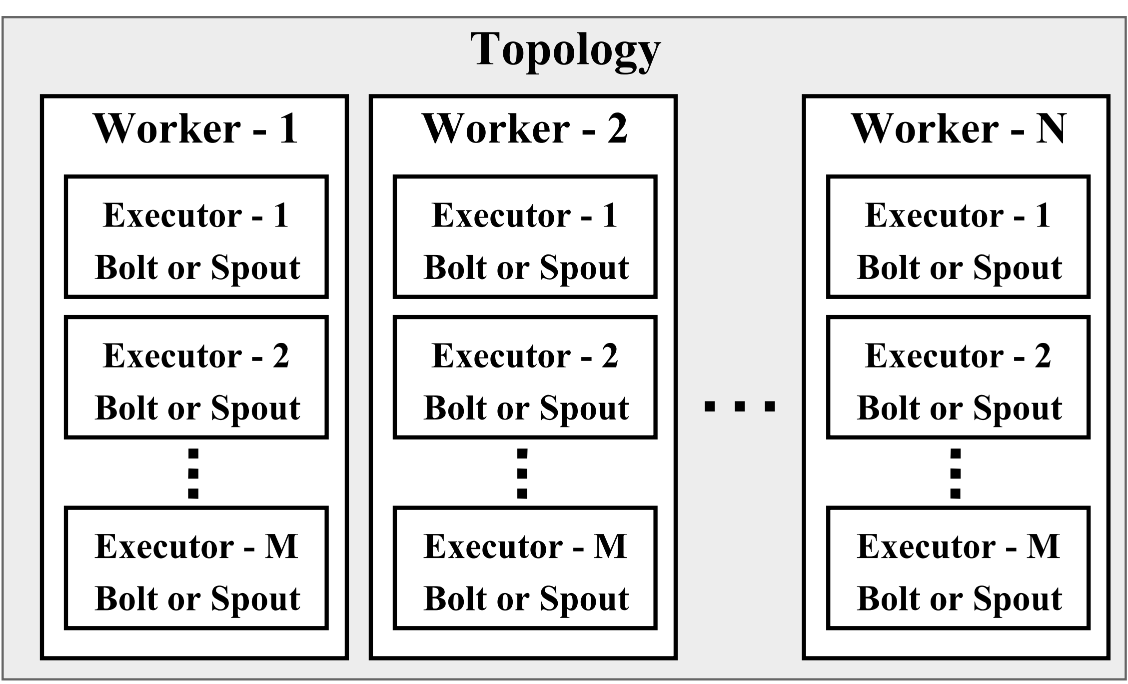

Figure 2.9: Storm parallelism components

Storm applications are defined in terms of topologies which are equivalent to Hadoop applications. A topology is a graph of elements connected to each other according to the user's application design where each element can have multiple input elements and output elements connected to it. The elements that comprise a topology in Storm are: Spouts which are the main data source used in Storm topologies. Spouts read data from external sources such as reading data from files, receiving messages over the network ...etc and then emits messages to bolts Processing elements in Storm topologies. Spouts are defined mainly in Java language however, Storm supports wide range of programming languages through Domain-Specific Language (DSL) feature. Bolts are the processing elements in Storm like maps in Hadoop. However, bolts are very flexible compared to Hadoop maps. User can define bolts to do anything without being restricted to certain programming model like map-reduce.

2.4.2 Operation

Storm APIs are provided in Java programming language, however, developers can write their applications in other language by utilizing the Multi-lang protocol which is a special protocol implemented in Storm that uses Standard Input and Standard Output to communicate with processes that implement the Bolts or Spouts logic. The communication is done

by sending and receiving plain text JavaScript Object Notation (JSON) messages.

Storm users can define multiple layers topology where the output from first stage bolts are emitted to a second layer of bolts to do further processing, therefore Storm users can divide a complex task into smaller ones and allocate more executors to certain tasks than others thus increasing the processing throughput. Also, Storm users define multiple Spouts to read from external data sources in order to balance the load between Spouts and avoid overloading a single Spout in the topology. Figure 2.10 and Figure 2.11 illustrates the single-source single layer topology, multiple-sources multiple layers topology respectively.

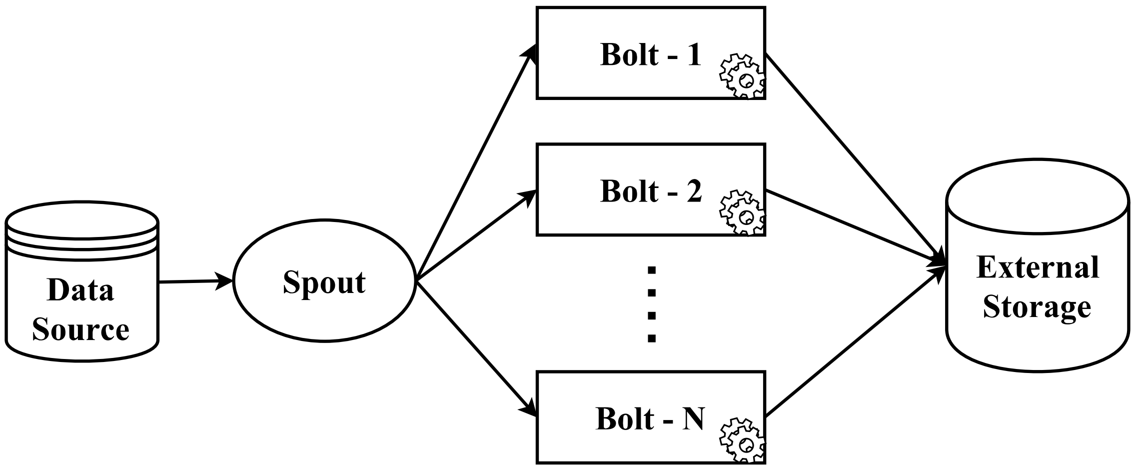

Figure 2.10: Storm single layer topology

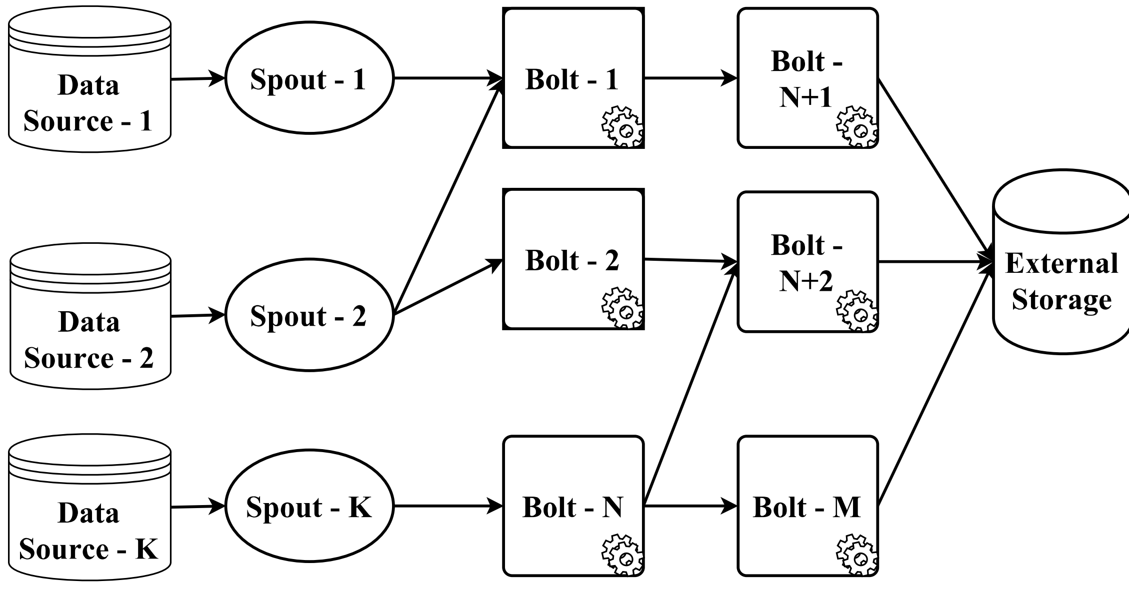

Figure 2.11: Storm multiple layers topology

The messages are emitted from spouts to the next layer of bolts according to key-mapping specified by the developer. Also, if there are multiple executors for the same bolt, Storm makes sure that the messages are distributed evenly among those bolts to guarantee that no Bolt will suffer from over-load.

2.5 Big-data Framework for Computer Vision

Hadoop, Spark and Storm are very powerful frameworks, the choice of any of them can vary based on the requirements of the application. It's worth mentioning that none of these tools was built for computer vision applications, thus each of them has its own cons and pros when considering computer vision as a prospective application domain. For example, Hadoop can be used for processing computer vision applications with high scalability and stability as it is very mature and well established amongst big-data frameworks. However, it has some drawbacks detailed as follows:

1. Slowest map limitation: In Hadoop jobs, the processing time is limited by the latency of the slowest CPU on which any of the concurrent maps is running. For example, if we have 8 maps running in a given job, then the reducer will not be able to emit the final results unless it completely received all the output results from the running maps. Therefore, even if seven maps finished processing, the reducer will wait for the eighth to finish which affects the overall processing time of Hadoop applications.

2. Pre-processing overhead: There is a relatively significant time required to prepare video data to be ingested by Hadoop jobs. The overhead comes from the time required to upload video data to HDFS. Job creation latency and overhead, maps and reducers in Hadoop are JVMs that are created and destroyed on job start and end. The creation and destruction of the JVMs is repeated with each job, hence the overhead is periodically repeated and not just once at the start-up which impacts the throughput. This limitation makes processing of individual frames in Hadoop inefficient as the processing time of a single frame in each map task is less than map launching time.

3. Batch processing model: Hadoop is highly scalable and reliable however, it supports only batch processing model thus we cannot have a long running jobs in which maps are created and persisted in memory to process streams of video received through

the network. This limitation makes Hadoop suitable only for processing batched of video data recordings.

In Spark, although it supports In-memory processing, most computer vision libraries don't support the storage APIs of HDFS or In-memory storage exposed by Spark framework. The major speed up from In-memory processing in Spark is for iterative algorithms that require several iterations over the same data such as machine learning applications and not computer vision applications, therefore, no gain is achieved from the In-memory feature of Spark when it comes to computer vision applications. However, Spark framework supports long running tasks which means we can create a job and it keeps persisting the JVMs of the executors constituting this job in memory, thus saving the job creation overhead that existed in Hadoop.

Spark streaming is a layer built on top of Spark core engine. It abstracts processing of streaming data on top of Spark engine. It can suit the nature of video streams to some extent with some limitations. In Spark streaming, the workflow groups the input data or records in micro-batches that are processed sequentially and not concurrently. The concurrency is applied in the processing of the micro-batch records. Therefore, computer vision applications that have relatively large processing latency and stringent real-time requirements won't be satisfied on Spark streaming.

2.6 Conclusion

In this chapter we presented an overview of different big-data frameworks and the typical usage of each framework. Big-data technologies are gaining a lot of momentum in different engineering and scientific fields. They have proven themselves as capable of processing huge amounts of data in an acceptable time, thus delivering more insight from these data. Hadoop, Spark and Storm are the most famous big-data frameworks for processing plethora of data types and structures. However, each of them is suited and optimized for specific processing model. Hadoop is optimized for batch processing model, Spark is optimized for In-memory batch and micro-batch streaming models, Storm is optimized for real-time streaming model and micro-batch processing thus we choose it in this thesis as the big-data framework for processing computer vision applications.

Chapter 3: Graphics Processing Unit

3.1 History and Background

In late 80's home computers which are sub-category of personal computers became popular and spread more and more in the technology markets. These computers had the same hardware capabilities except when it comes to video processing capabilities. In this era, the popularity and successfulness of a home computer over another was determined by the video capabilities of such a computer. This linking between the popularity of a computer and its video capabilities was mainly because of the spread of video games that were the main factor affecting the consumer decision when deciding which computer to buy [32].

Graphics rendering first took place by the CPU. The CPU reads binary data from video buffers then sends signals to displays to draw objects. However, with time, these graphics became more complicated and required expensive memory buffers in order to store the bits that represent these objects. In order to enable more complex visual rendering and to save the memory space, graphics processing was handled by independent hardware circuits known as video controllers or graphics chips. As illustrated in Figure 3.1 more graphics rendering tasks migrated from CPUs to GPUs with time.

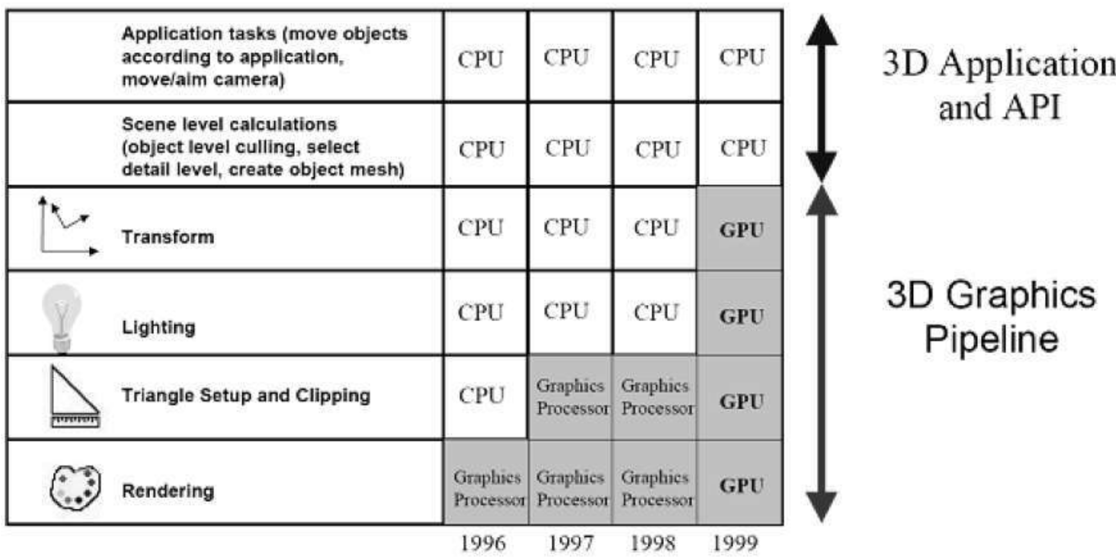

Figure 3.1: Graphics rendering tasks migration from CPU to GPU [33]

A GPU is a dedicated hardware circuitry that is responsible of 2D and 3D visual objects rendering on external displays connected to computer devices. GPUs can coexist on the same chip with the CPU in what is known as Integrated Graphics Processor (IGP) and share the system memory with the CPU too or on an external pluggable device that can be connected to the computer device via the Peripheral Component Interconnect (PCI) . GPUs also exist in smarphones, laptops and supercomputers. The term GPU has become common when Nvidia Corporation announced its GPU "GeForce 256" as "The world's first GPU" [33].

3.2 Architecture and Operation

The traditional architecture of GPUs relied on a graphics pipeline of fixed-function units. These units had specific functions to perform such as: Vertex Transformation and Lighting, Rasterization, Pixel Shading ... etc [33].

In order to allow programmers to access the graphics pipeline and to ease the developement of graphics applications, standards such as Open Graphics Library (OpenGL) and Direct3D were introduced. OpenGL was created by Silicon Graphics Inc. in 1989 as a set of APIs that are abstract function definitions which exposes the graphics pipeline functionality to the programmers. The implementation of those functions can be either software or hardware implementation and it is language independent [34]. The fixed-function graphics pipeline was nearly adopted by the major GPU manufacturers around the globe such as: Nvidia, ATI and Matrox corporations. The fixed pipeline architecture decreased the cost of the GPUs but it restricted the GPU processing power to only specific graphics processing functions.

The continuous need for more powerful graphics cards that are capable of rendering 2D and 3D graphics specially those of video games, has pushed the manufacturers of graphics cards to double the transistor count on GPUs to increase the processing power of the GPUs. According to [34], the number of transistors on Nvidia's GeForce 6 is double the amount of transistors on Intel's Pentium 4 processor and the rate of increase in number of transistors on GPU chips has surpassed Moore's law limitations.

The evolution of GPUs hardware and the extensive features added over and over by the standard groups and technology corporates in libraries such as OpenGL and Direct3D, have raised the need for programmable graphics pipeline to leverage the growing computational power of GPUs in non-3D graphics applications.

In 2001, Nvidia introduced the first programmable GPU ”Geforce3”, in this card instead of sending the data to it in order to render the graphics with no intervention from the programmer as was the case for the traditional GPUs, the programmer now sends the data with a vertex ”shader” program that specifies what type of operation to be done on the accompanying data [34]. The architecture of ”GeForce3” is shown in Figure 3.2. Of course, the level of programmability in GeForce3 was limited but it paved the way for more liberation in the programmabiliy of GPUs.

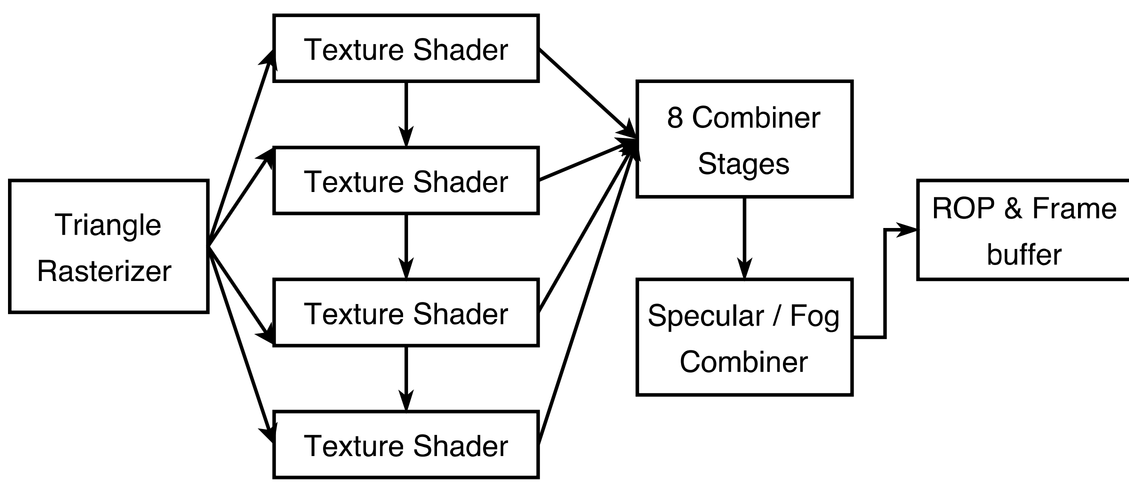

Figure 3.2: GeForce3 programmable architecture [34]

In 2002, Nvidia and ATI released their first fully programmable GPUs GeForce FX and Radeon 9700. In these cards, there were separate hardware units for processing vertex shaders and pixel shaders [33]. By 2003 and 2004, programmers started to look into wider range of applications for GPUs other than graphics applications. The emerging of full floating point units, high bandwidth memory, texture engines, vertex engines and other powerfull hardware components in the GPUs made GPUs with their graphics pipeline appealing and efficient for non-graphics applications specially data-parallel applications that require applying the same operations on multiple input data concurrently [33].

In 2006, Nvidia introduced its GPU GeForce 8800 with a unified architecture as shown in Figure 3.3. Nvidia designed Geforce 8800 to be the first GPU to have a unified programmable shaders. These shaders are referred to as Streaming Multiprocessor (SM) which are capable of performing several graphics operations such as: Vertex, Pixel and Geometry computations [33]. The operation of Stream Processors depends on kernels and Streams. A Stream is the input data to the Stream Processor, a Kernel is the program that will process one or more Streams.

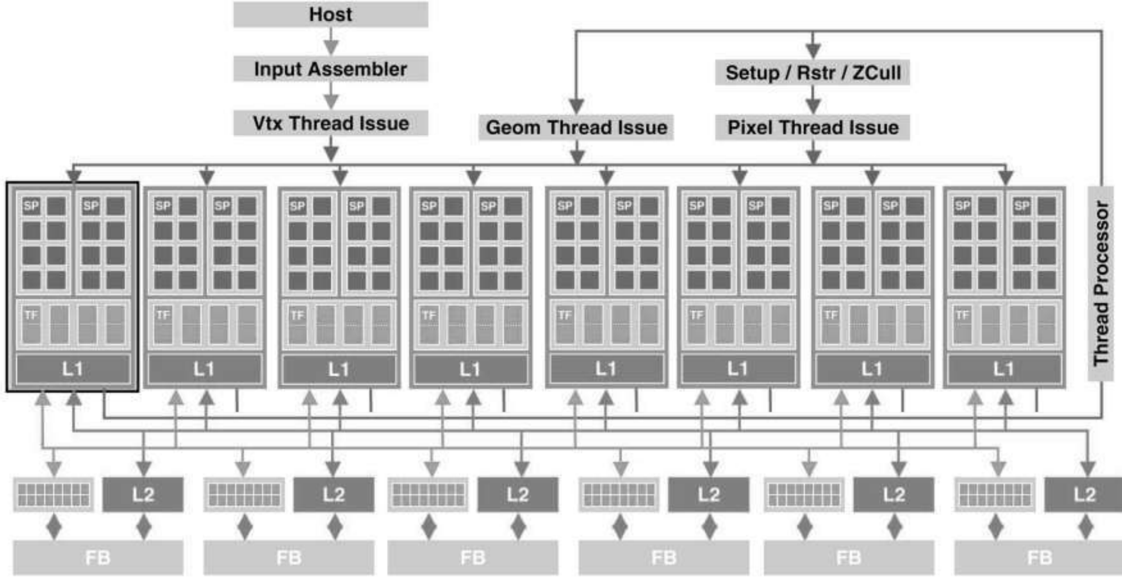

Figure 3.3: Unified GPU architecture [33]

Stream Processors enable three levels of parallelism. First, Instruction Level Parallelism where each Stream Processor can perform different instruction on the same Stream. Second, Data Level Parallelism where multiple Stream Processors can perform the same instruction on multiple Streams concurrently. Third, Task Level Parallelism, where a task from a Kernel can be divided between multiple Stream Processors to increase the throughput [35]. The Stream Processor design in Fermi architecture is shown in Figure 3.4.

The unified architecture has abstracted the hardware components of the GPU to only software level . Along with the unified architecture, GeForce 8800 supported C programming language thus facilitating the programmability of the GPU as now the programmers don't have to learn specific language for the GPU or be aware of the Graphics APIs as before. It also supported Nvidia's Compute Unified Device Architecture (CUDA) technology which will be discussed in details later in this chapter. GeForce 8800 had 128 SMs with clock speed of 1.35 Ghz [36].

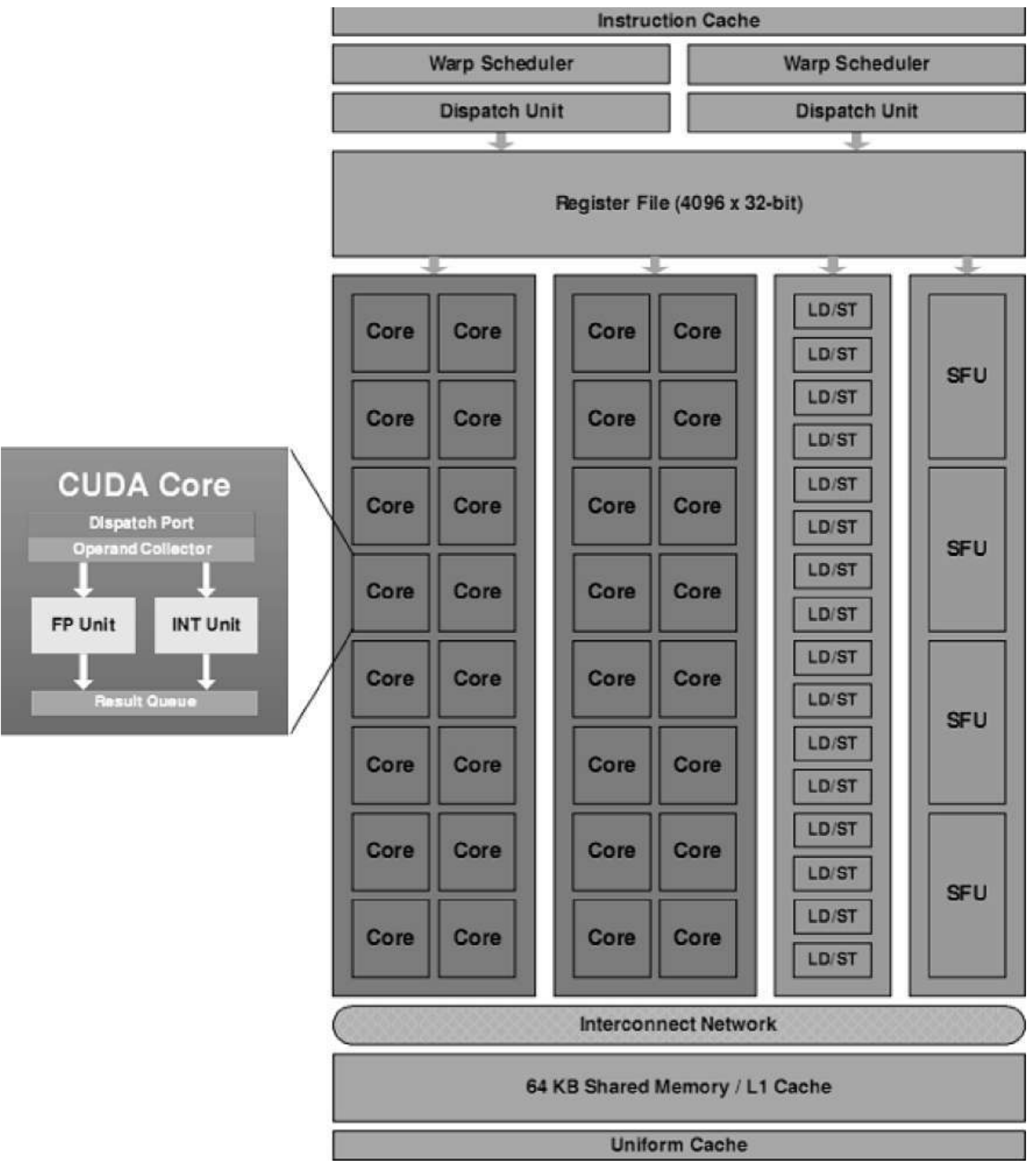

Figure 3.4: Stream processor architecure [33]

The growing interest in the programmability of GPU cores has continued with the introduction of Nvidia's Fermi architecture in 2010. Fermi architecture was the first GPU architecture designed specifically for General Purpose Computing on GPU (GPGPU) . The Fermi architecture included a lot of improvements over its successors. It incorporated 512 CUDA core organized as 16 SM with 32 CUDA core per SM. A CUDA core mainly consists of Floating Point Unit and Arithemtic Logic Unit; It is also capable of executing Floating point or Integer instruction per clock cycle per thread [37]. GTX 480 was the first GPU designed with Fermi architecture. It had 15 SMs with 32 CUDA cores in each SM.

In recent years, Nvidia started to facilitate the utilization of its advanced GPGPUs for big-data analytics. Nvidia's Kepler architecture is an advanced GPU architecture released after the Fermi architecture. It tackles many issues in Fermi architecture, specifically the hardware single kernel queues which yielded false serialization of tasks spawned from different CPU cores. In other words, in Fermi architecture, parallel kernel calls from multiple CPUs are serialized at the GPU end as if they are dependent kernel calls thus the term false serialization [38].

In Kepler architecture a new feature called Hyper-Q is introduced. It enables simultaneous connections from multiple CPU cores to the GPU with no false serialization. Kepler architecture enables up to 32 hardware managed connections to the GPU compared to single connection in Fermi architecture. In order to optimize simultaneous kernel calls to the GPU and achieve the best utilization of the GPUs that support Hyper-Q feature, Nvidia introduced a software service called Multi-Process Service "MPS". This service manages the simultaneous connections from applications accessing the GPU. It acts as a layer between the GPU driver and applications [38]. The MPS architecture is shown in Figure 3.5.

Althouh MPS enhances the performance of parallel applications on GPUs, it has two limitations, first, it works with CUDA applications only so no OpenCL support. Second, it has a limitation in the number of simultaneous applications which is 16, thus whenever the number of applications exceeds 16, the MPS encounters resource allocation errors [39].

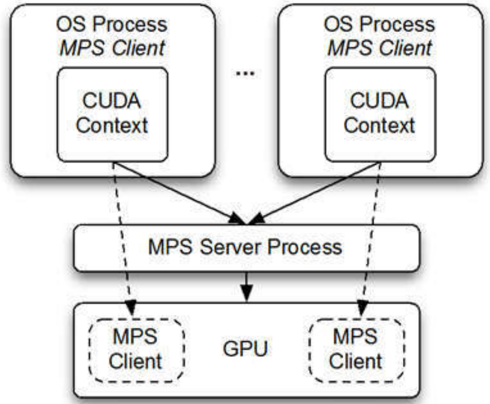

Figure 3.5: Nvidia Multi-process Service architecture [39]

3.3 GPU Programming Frameworks

In the early days before the introduction of GPGPUs, scientific communities and programmers interested in leveraging the processing power of GPUs had to learn Graphics APIs and languages such as OpenGL, they also had to map their calculations and algorithms to Graphics operations and polygon calculations. This restriction limited the wide adoption of GPUs for general purpose computing for a long time.

In 2003, Ian Buck with a group of researchers created Brook which was the first language to extend C programming language and exposing GPUs as a general purpose parallel-computing devices. Brook programs weren't only written in a high-level language but also they were faster than existing codes written in other languages [40].

After Brook was released, Nvidia invited Ian Buck to continue his work on a high-level efficient language for its fast and unified hardware architecture which led to the announcement of CUDA technology in 2006 by Nvidia to be the first hardware and software technology that enables general computing on GPUs in the world.

CUDA is a hardware and software architecture for parallel programming on Nvidia's GPUs. It allows developers to write GPU applications in different languages such as: C, C++, Fortran and DirectCompute, then they can run these applications on Nvidias' GPUs to make use of the fast and unified CUDA architecture [37].

In CUDA programs the developer calls parallel Kernels to process Streams. These Kernels execute across parallel threads. These threads are typically organized either manually by the programmer or automatically by CUDA compiler into thread blocks. Thread blocks are a group of threads working concurrently on the input data set or Streams, these threads can cooperate by using "Barrier Synchronization" and Shared Memory [41].

In each thread block, a thread has a unique ID, a group of thread blocks form a grid and within a grid of thread blocks, each thread block has unique ID. Kernels are executed on grids of thread blocks which read and write input and output data to and from global shared memory. Beside the unique ID, each thread within a thread block has an independent per-thread private memory and program counter. The global shared memory and synchronization are used to share the results and maintain sanity of the running parallel algorithm across multiple threads [37]. Figure 3.6 shows the thread hierarchy in CUDA architecture.

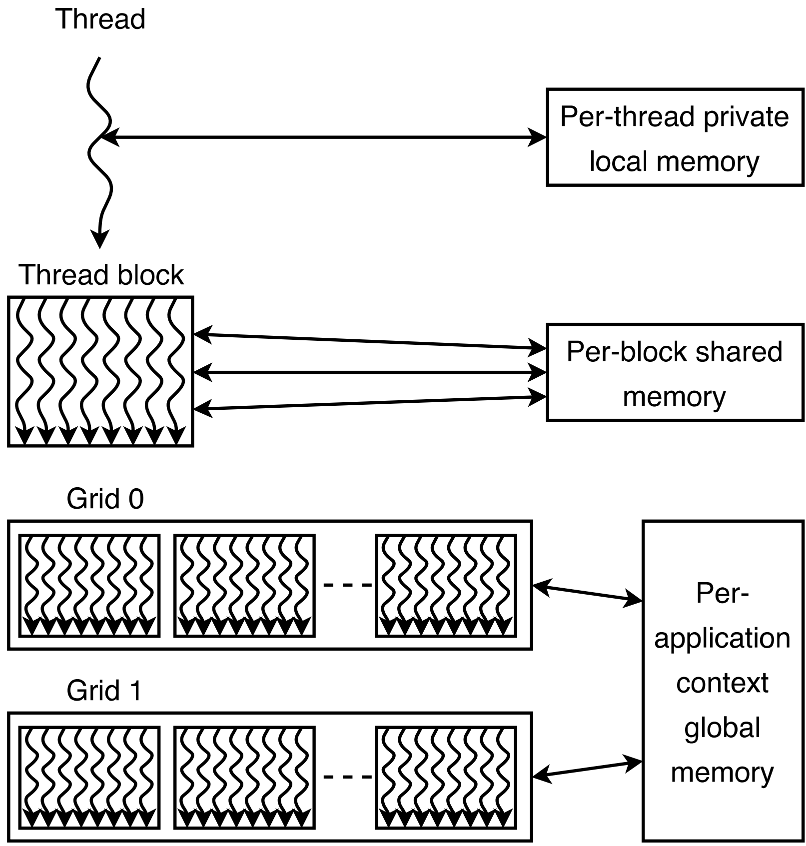

Figure 3.6: CUDA thread hierarchy [33]

From a hardware execution perspective, a GPU executes Kernel grids where a Stream Processor executes thread blocks. CUDA cores within the Stream Processor executes the concurrent threads. Typically, Stream Processor groups 32 threads in a single unit called a warp. Warps greatly improve performance of CUDA applications and programmers are usually advised to use warps concept although it is not an obligatory path while writing CUDA applications [37] [41].

CUDA Software Development Kit (SDK) exposes a set of APIs to the developers to facilitate programming against CUDA capable GPUs. CUDA technology also enables other frameworks such as Open Accelerators (OpenACC) and Open Computing Language (OpenCL) to access GPU resources. It is supported in all Nvidia GPUs from G8X series onward. First CUDA SDK was released in 2007 with support for Linux and Windows operating systems, Mac OS support was added in version 2. It is also a proprietary technology owned by Nvidia and it is not supported by any GPUs except Nvidia GPUs which is a major drawback in CUDA technology thus we will discuss another rival to CUDA technology which is OpenCL.

OpenCL [42] is an open standard specified by several industry groups and corporates,

It is maintained by a non-profit organization called Khronos group. It enables writing unified code that can run on a group of any OpenCL capable device such as: CPU, GPU, Digital Signal Processing (DSP) , Field Programmable Gate Array (FPGA) and others. In other words OpenCL enables heterogeneous computing and unifies different hardware computing devices under the same framework. It includes APIs, libraries, Languages and run-time software and hardware support, thus OpenCL is a complete framework rather than just a portable programming language for heterogeneous devices [42].

OpenCL programming language is a C-like language based on C99 standard specification with additional extensions to enable data-parallel applications. It provides a run-time compiler and exposes a set of APIs in C and C++ to access the underlying hardware resources such as memory. Other third party APIs and bindings exist for languages such as: Python [43], Java and .NET [44]. OpenCL abstracts any compatible hardware device such as GPU, CPU or any other device as a Compute Device. A Compute Device typically consists of several Compute Units (CUs) which incorporates several Processing Elements (PEs) .

An OpenCL application consists of functions called Kernels which execute in parallel on some or all of the PEs available on a Compute Device. The division of Compute Device into CUs and PEs is left to the hardware manufacturer of the Compute Device. OpenCl architecture is depicted in Figure 3.7. This hardware abstraction and vendor independent architecture made OpenCL highly portable and compatible framework for heterogeneous compute clusters.

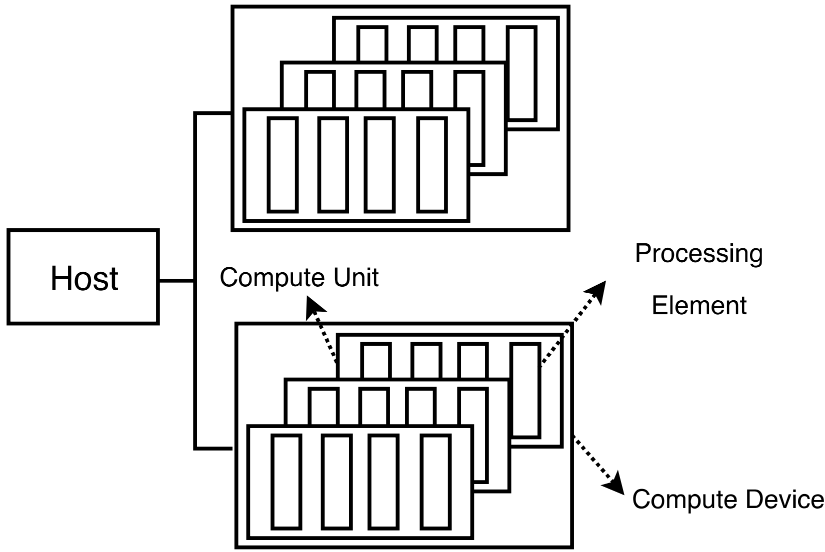

Figure 3.7: OpenCL architecture

In this context one can think of CUDA and OpenCL as two frameworks for programming against GPUs, although OpenCL supports wider range of devices. In this work our focus is on GPUs as a hardware accelerator. As mentioned earlier CUDA is propreitry technology owned by Nvidia, OpenCL is open standard which means that OpenCL is more portable and widely supported on nearly most of the GPUs avaiable in the market. However, this portabiliyty comes with a price. OpenCL performance is slower than CUDA [45], as indicated in a performance study in [46], OpenCL end-to-end run-time is 16% to 67% slower than CUDA for the same test case.

3.4 GPUs Role in Computer Vision Applications

Computer vision is a scientific field that studies acquiring, processing and analyzing visual data represented as 2D images in order to give us a better understanding and constituting better description of the real-world. It can also represent the objects in these visual data in specific models or symbols for further data mining and analysis.

Computer vision exists in a vast amount of devices and is considered a crucial part of many applications such as: Medical imaging, Remote Sensing, Multimedia and others. The nature of visual data either as discrete images or video records and streams combined with the complexity of the algorithms and real-time requirements of processing these data impose technical challenges in computer vision research and development [47].

Most computer vision applications such as: Face Detection, License Plate Recognition, Object Detection ... etc, are comprised of basic computer vision algorithms such as: Template Matching, Feature Detector, Color Conversions and others. With the increasing resolution and quality of videos and images nowadays, these algorithms suffer relatively large latency when processed on CPUs only, sometimes the delay can hinder real-time processing of incoming video streams.

During seventies, many started to think of replicating the instruction execution units to process multiple pixels concurrently, thus new terms emerged such as: SIMD and MIMD. Nowadays, multi-core CPUs incorporate hardware accelerators suited to the SIMD and MIMD architectures. For example, the SSE designed by Intel as an instruction set extension to x86 architecture and introduced with Pentium 3 processors. SSE was expanded later with SSE2, SSE3, Supplemental Streaming SIMD Extensions 3 (SSSE3) , SSE4, AVX and AVX2 [48] [49]. These instructions are well suited for a type of applications where the same instructions are to be executed on multiple data such as pixels

in images.

The continuous increase in GPUs processing power and their adequacy for data-parallel applications made them another superior candidate for accelerating computer vision applications beside the CPU Vector Extensions. The scientific community realized that GPUs can increase the throughput of most of computer vision algorithms that require applying the same operation on multiple pixels of an image. Leveraging the multiple cores on a GPU can meet the real-time requirements and achieve acceptable throughput for computer vision applications. Many computer vision algorithms such as: Optical flow, Feature detection and tracking, have been ported to GPU frameworks such as CUDA and OpenCL.

The reported speed up after porting to GPUs is in the range of five to ten when compared to multiple threads implementation on multiple cores CPUs . The use of GPUs doesn't only increase the throughput, it also decreases the cost and is more energy-efficient for most of the computer vision algorithms [50].

Nowadays, there are many computer vision libraries that the programmers can use to develope computer vision applications. One of the most popular and widely used libraries is Open Computer Vision (OpenCV) . OpenCV was an Intel Research project launched in 1999. Currently, it is released under Berkeley Software Distribution (BSD) license as a free open source library for both academic and commercial use. It is basically written in optimized C/C++, but it has other language bindings and wrappers such as: Python, Java and Clojure.

OpenCV can handle multi-core processors and it can utilize Intel Performance Primitives (IPP) whenever they are installed on the target machine in order to enhance the performance. Figure 3.8 shows a performance comparison between OpenCV with and without IPP and two other computer vision libraries VXL and LTI; The left axis indicates the running time of the algorithm which shows that OpenCV with IPP enabled surpass the other two libraries and OpenCV without IPP [51].

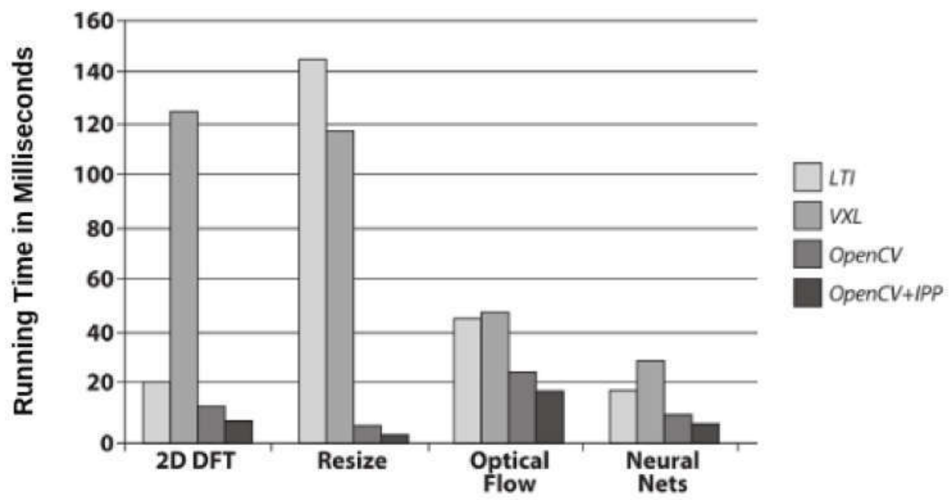

Figure 3.8: OpenCV performance comparison [33]

The increasing interest in GPUs and leveraging the processing power of GPUs in accelerating computer vision applications made OpenCV contributors and developers adopt GPU platforms in OpenCV. In OpenCV library, many algorithms have two or three versions, one is a CPU implementation and the other two are OpenCL and CUDA implementations of the algorithm. The performance gain from using GPUs in OpenCV to speed up some algorithms' processing throughput is depicted in Figure 3.9 and Figure 3.10 for CUDA and OpenCL platforms respectively.

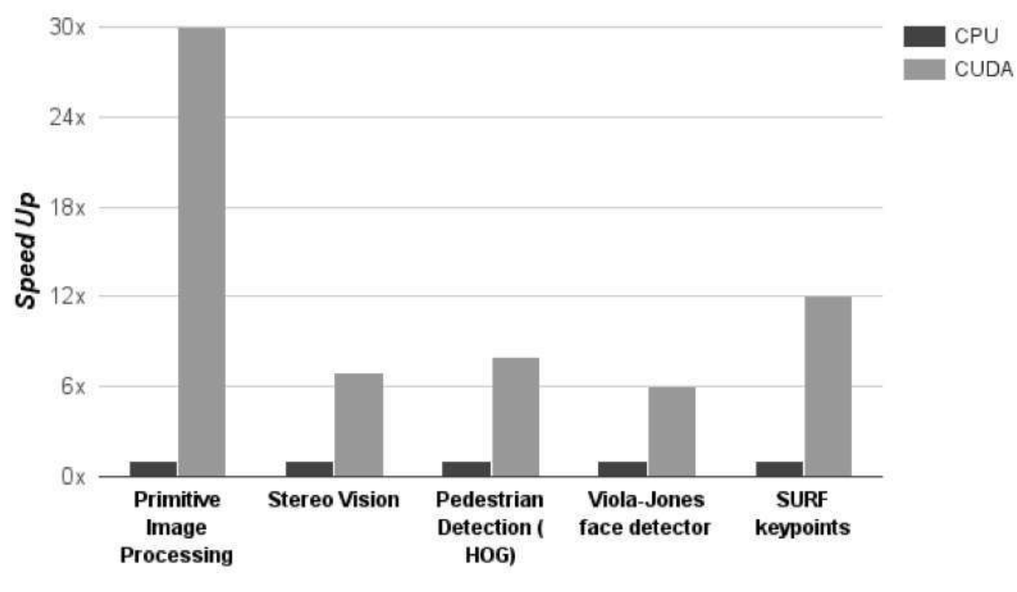

Figure 3.9: CUDA speedup versus CPU [33]

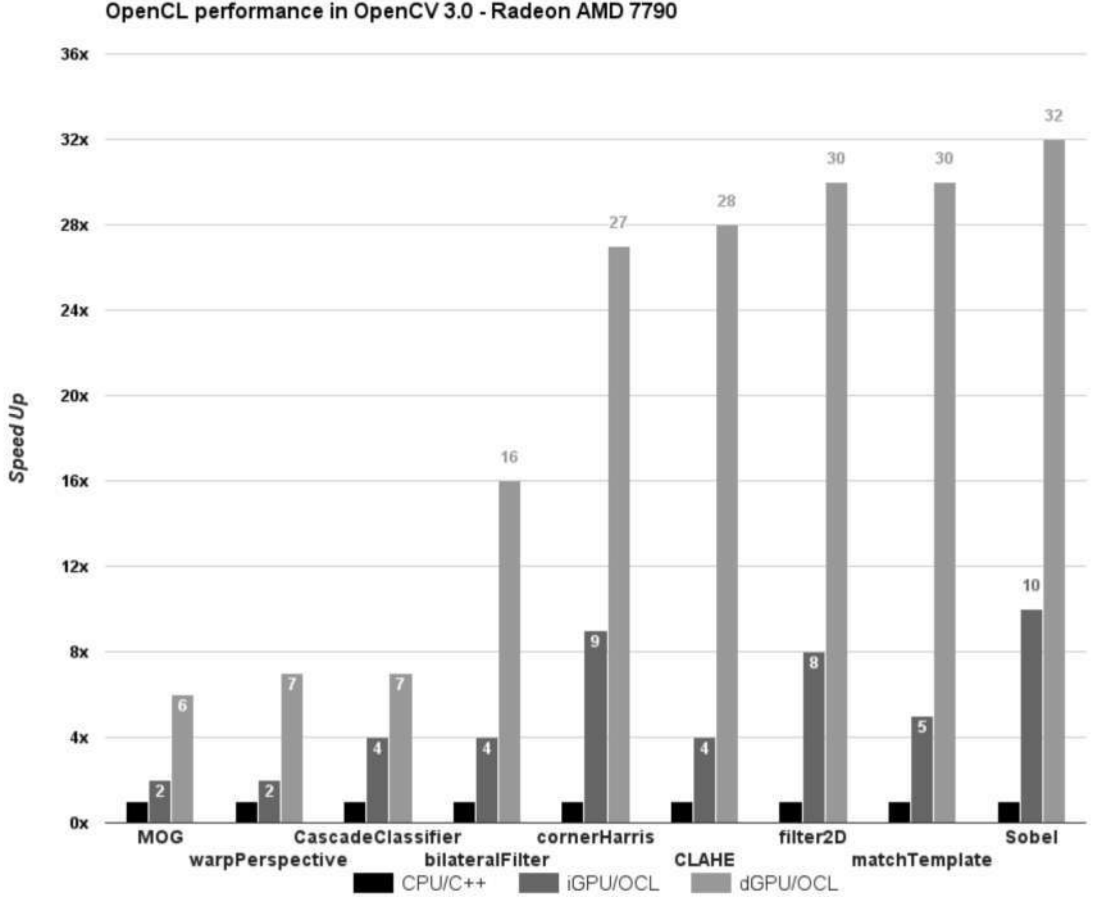

Figure 3.10: OpenCL speedup versus CPU [33]

3.5 Conclusion

In the past few years, GPUs have witnessed a great advancement in their computational power thanks to the hundreds and thousands of cores available on them. GPUs are very suitable for massively data-parallel applications making them a superior candidate for accelerating computer vision applications that rely on performing the same operation on multiple data objects. CUDA and OpenCL are two of the most popular GPU frameworks. Many libraries exist to facilitate and provide ready-to-use methods and algorithms for computer vision applications. OpenCV is one of the most popular computer vision libraries used nowadays. Several OpenCV algorithms have GPU implementations for both OpenCL and CUDA frameworks.

Chapter 4: Literature Review

4.1 Introduction

In this chapter we discuss several research efforts related to enabling GPUs in different big-data frameworks. We also highlight the main limitations in these efforts and the difference between our goals and what they achieved in terms of high-level features and performance. We divided this chapter by relevance of the topics in each work.

4.2 Map-reduce Related Work

4.2.1 CUDA Framework

In 2008, a group of researchers introduced Mars [52] as the first framework for enabling GPUs utilization in map-reduce frameworks. Motivated by the large processing power of GPUs Mars was developed based on Nvidia G80 to hide the complexity of GPU programming in the well known map-reduce framework. Mars was tested against Phoenix - the state of art map-reduce framework in the time Mars was developed - and it achieves around 1.5-16x speed up gain for six web applications: String match, Inverted Index, Similarity score, Matrix multiplication, Page view count and Page view rank. These applications are used in typical web tasks such as: Web document searching, Web document processing and Web logs analysis. The value of the speedup depends on whether the task is computationally intensive or not. The gain for computationally intensive tasks is the highest.

Although Mars presents significant performance improvement in map-reduce framework and hide GPU programming difficulties behind a set of APIs similar to the CPU counter part, it has some cons when applied to computer vision applications on big-data frameworks. First, it lakes generality, as it is only limited to map-reduce frameworks and specific APIs. Second, It is limited to CUDA framework as nothing was mentioned about OpenCL which is widely adopted by the technology community for several reasons as discussed earlier in chapter three.

Finally, Mars assign one data chunk per GPU thread and the performance gains reported by the authors [52] are for web analytics tasks on mostly text data and logs; If

one considered the case where not only one thread but multiple threads or even all the threads on the GPU are utilized by one computer vision task, then Mars - according to its implementation details - cannot guarantee that no competition will occur on the GPU by multiple parallel computer vision tasks that typically consume a lot of GPU resources.

In 2011, GPMR [53] was introduced as a more extend version of Mars. It enables utilizing multiple GPUs in map-reduce frameworks as opposed to single GPU support as introduced in Mars. The authors of GPMR address two main issues in Mars. The first is what would happen if the data item processed by a map task in a GPU thread exceeds the size of the in-core memory. The second is how Mars would address the scalability amongst multiple nodes where the output of any map could be sent to one or multiple reducers across the cluster, thus the sorting and partitioning of the maps output should be optimized to guarantee minimal network communication overhead. GPMR library also tackles generic challenges in multi-GPU support in map-reduce frameworks. These challenges can be summarized as:

1. GPU to GPU communication over the network.
2. Lack of out-of-Core support and virtual memory in which the applications process data that is relatively large to fit completely in the memory.
3. Map-reduce high-level programming model by default abstracts the computational resources thus limiting potential optimization.

The authors developed GPMR library using C++ and CUDA primarily, the map-reduce framework is partitioned into multiple stages each gets executed in a pipeline fashion as illustrated in Figure 4.1, where the CPU executes the solid-line boxes and GPU executes the dashed-line boxes.

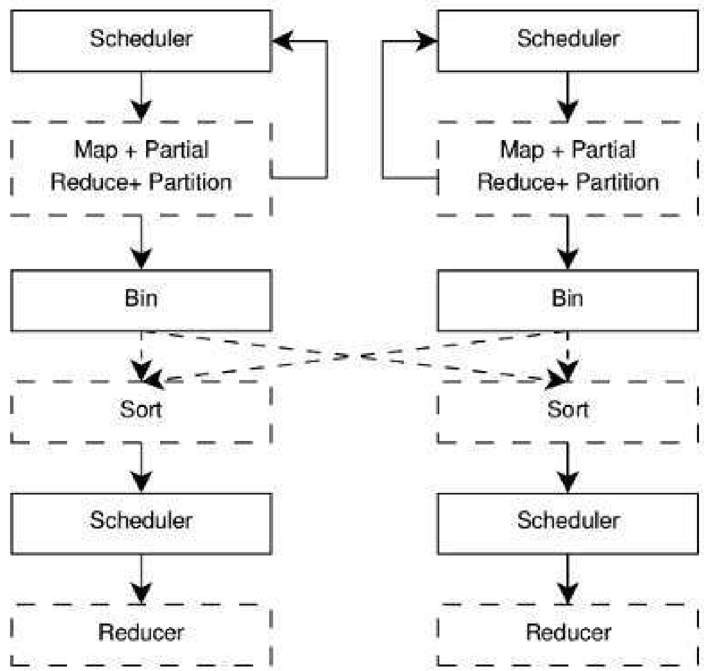

Figure 4.1: GPMR multi-stage map-reduce pipeline [53]

In order to benchmark the GPMR library performance, the authors created a setup of 64 GPU cluster and tested GPMR against the following applications:

1. Matrix multiplication

2. Sparse Integer Occurrence (SIO)

3. Word Occurrence

4. K-means clustering

5. Linear regression

The results included in the paper shows that enhancements made by utilizing the GPUs in map-reduce applications have led to better performance and scalability compared to Mars and Phoenix. As shown in the results section, GPMR achieves around 2.6x to 37x gain with one GPU and 10x to 129x gain with four GPUs when compared to Mars; Comparing GPMR to Phoenix shows 1.3x to 162x with one GPU and 2.3x to 559x with 4 GPUs.

GPMR has achieved significant improvement over Mars and Phoenix, however it is limited to CUDA and strict map-reduce programming model. It didn't address computer vision intensive tasks that consume nearly all the computational power of the GPU, thus it

didn't provide a solution for the competition on the GPU when multiple concurrent heavy tasks are utilizing the GPU from within a big-data framework.

There are still some challenges as concluded by the authors that need to be tackled; Challenges such as: Scalability issues due to growing communication between nodes, Various tasks require different cluster configurations in order to obtain best results and speed-ups. The limited memory in GPUs represent a challenge that can sometimes be tackled by specifying the appropriate chunk size to work on, and finally the lack of network communication capabilities in GPUs.

MapCG [54] is a unified framework for writing parallel programs that execute on both CPUs and GPUs efficiently. The main contributions of MapCG are how to express and synchronize parallel programming frameworks, and how to manage different memory hierarchies automatically without intervention from the developer to reduce the development complexity and cost. The average speedup of MapCG over past map-reduce implementations is around 1.6 - 2.5x for eight common applications. It is only limited to CUDA programming model and doesn't support OpenCL. It also uses either the CPU or the GPU but cannot use both at the same time and it cannot handle multi-GPU clusters.

In [55], the authors introduced a map-reduce framework with multi-GPU support called MGMR. The challenges tackled in this work is:

1. Utilize more than one GPU to accelerate map-reduce applications.
2. Address the fact that GPUs has no virtual memory, thus it cannot handle big-data by nature.
3. Replace the serial operations performed on the GPU with parallel ones to enhance throughput of parallel map-reduce applications running on the GPU.

MGMR is mainly developed in C++, it targets CUDA framework and Nvidia's Fermi architecture. MGMR balances the computational load amongst multi-GPUs installed on the system. The workflow of MGMR is elaborated in Figure 4.2.

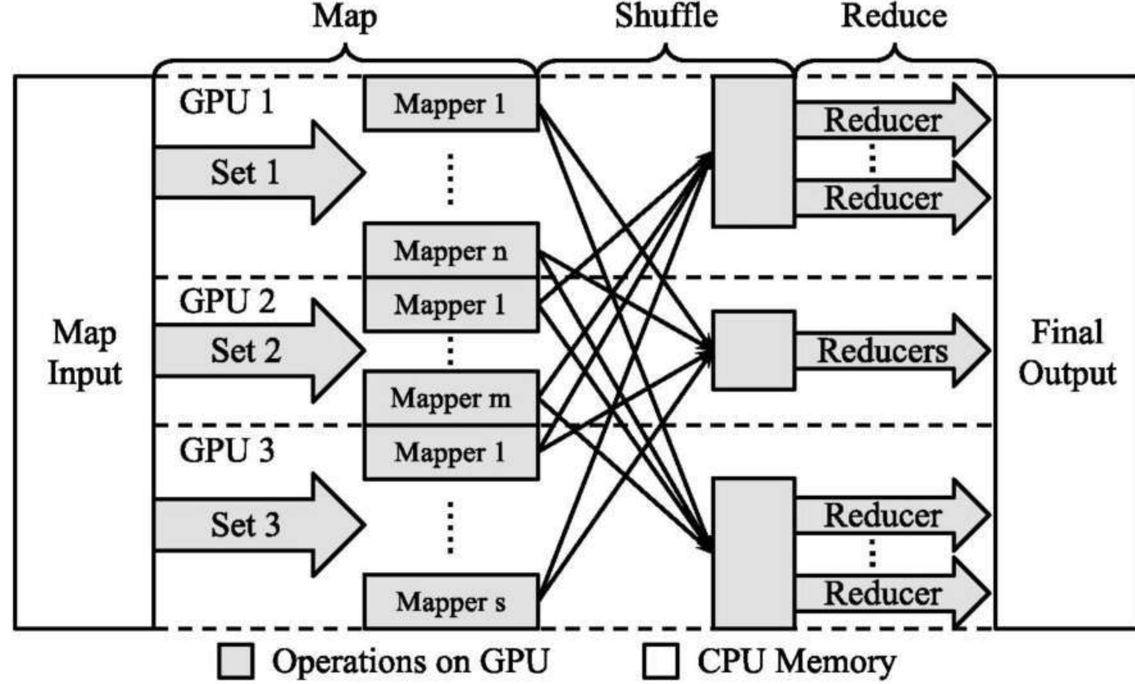

Figure 4.2: MGMR workflow on multi-GPU system [55]

MGMR can handle input data that exceeds the collective memory size of the GPUs installed on the system through multi-round processing scheme. In this scheme the input data is divided and distributed to the GPUs according to the max available memory. Each round process part of the input data until the whole input data is processed by the GPUs.

There is a CPU memory pool used in order to handle any data that exceeds the GPUs memory during the lifecycle of the map-reduce application. In case the whole input data fit into the GPUs memory, then a single round is applied and GPUs global memory is used for inter-mediate data. During the shuffling stage where the intermediate data is sent from the maps to the reducers, any communication between the GPUs is performed through GPUDirect feature in Nvidia Fermi architecture. This feature enables the GPU to directly access memory of another GPU on the host computer without passing by host's CPU or memory.

MGMR is evaluated with two applications: K-means clustering and Unique Phrase Pattern. In K-means, MGMR with double GPUs achieved 91.7 and 1.7 times speed-up against CPU and single-GPU versions respectively. In unique Phrase Pattern, the double-GPU achieved 12.6 and 1.5 times speed-up against CPU and single-GPU versions respectively.

The work presented in MGMR has overcome some challenges in utilizing multiple

GPUs for map-reduce applications, however it is limited to CUDA framework and specific devices from Nividia that supports Fermi architecture. In addition to this, in case there is no GPU or the installed GPU is not strong computationally, then the algorithms will fail to work properly because it doesn't provide alternative path to work on the CPU.

In [56], the authors propose a Pipelined Multi-GPU Map-Reduce (PMGMR) system as an upgraded version of MGMR. This version handles larger data sets that don't fit on GPUs memory nor on CPU memory. PMGMR employs new features incorporated in advanced GPUs such as streams and Hyper-Q. The performance gain of PMGMR compared to MGMR is nearly 2.5x. PMGMR targets CUDA platform on Nvidia Fermi and Kepler architectures. Although PMGMR is faster and more scalable than MGMR, it still has nearly the same limitations in terms of supporting CUDA software framework, Nvidia Fermi and Kepler hardware architectures only. Also, if GPUs are occupied the processing don't continue on an alternative CPU path.

In [57], the authors present an integration between CUDA platform and Hadoop map-reduce framework. Their approach was to run map tasks on GPU enviroment and not CPU in order to accelerate the throughput and utilize the GPU multi-core copabailities. They modified how Hadoop map-reduce splits the input data in order to be compatible with the input format of CUDA framework. However, their modified data splitting method is not suitable for all CUDA applications, their framework is only applicable to CUDA platform, hence it is limited to Nvidia GPUs.

In [58] the researchers address the exploitation of both the GPUs and CPUs available on the machines in Hadoop clusters. The contribution of the authors in enabling GPUs as computing resource in map-reduce framework can be summarized in the following points:

1. They created a directive based programming model on top of Hadoop streaming.
2. They created a compile to translate the written map-reduce application into CUDA compliant application.

They integrated their work with Hadoop framework as follows:

1. They modified the Task Tracker in Hadoop to be able to schedule tasks on CPUs and GPUs. The priority is given to the GPU slots.
2. The authors also created GPU drivers as an intermediate layer between GPU and Task Tracker to facilitate the communication between both of them.

3. The authors created HDFS driver which enable GPU access to HDFS.

The performance tests carried on shows that exploiting one GPU per node can enhance performance up to 2.78x.

In [59] a GPU-based map-reduce framework is designed and evaluated against Mars and MapCG. The performance evaluation shows 12.4x and 4.1x speed up over Mars and MapCG. However, the framework depends on the GPU and cannot provide alternative path for CPU implementations.

4.2.2 Other Frameworks

HIPI [60] is a tool developed to facilitate image processing on Hadoop map-reduce framework. It provides an input format for images called HIBI Image Bundle which takes care of distributing images inside of map-reduce framework without any complexities to the user in terms of encoding/decoding or distributing the images across multiple tasks. The work presented in HIPI is concerned with large scale distributed image processing on CPU or homogeneous clusters only with no GPU support.

In [61] a framework for accelerating map-reduce frameworks by utilizing an on-chip GPU is proposed; It supports a single GPU and single node. In [62] a system for large scale processing of satellite images based on homogeneous map-reduce cluster is proposed with significant improvements in throughput. In [63], a map-reduce based system for processing of medical images is proposed, it achieved better performance when compared to the sequential analysis of these images but the system didn't incorporate GPUs. In [64] a proof-of-concept for processing on heterogeneous cluster of computing resources using OpenMP is presented and achieved up to 2x performance gain when compared to the CPU-only or GPU-only processing.

4.3 Generic Heterogeneous Computing with GPUs

In [65], the authors of this paper present a new scheduler named OCLSched which schedules computation tasks on accelerators - OpenCL specifically - for heterogeneous computing. OCLSched adopts a server-client model to enable access to accelerators by multiple users as depicted in Figure 4.3. OpenCL was chosen over other models as it is

open-source and portable. The work presented in this work didn't incorporate CUDA framework.

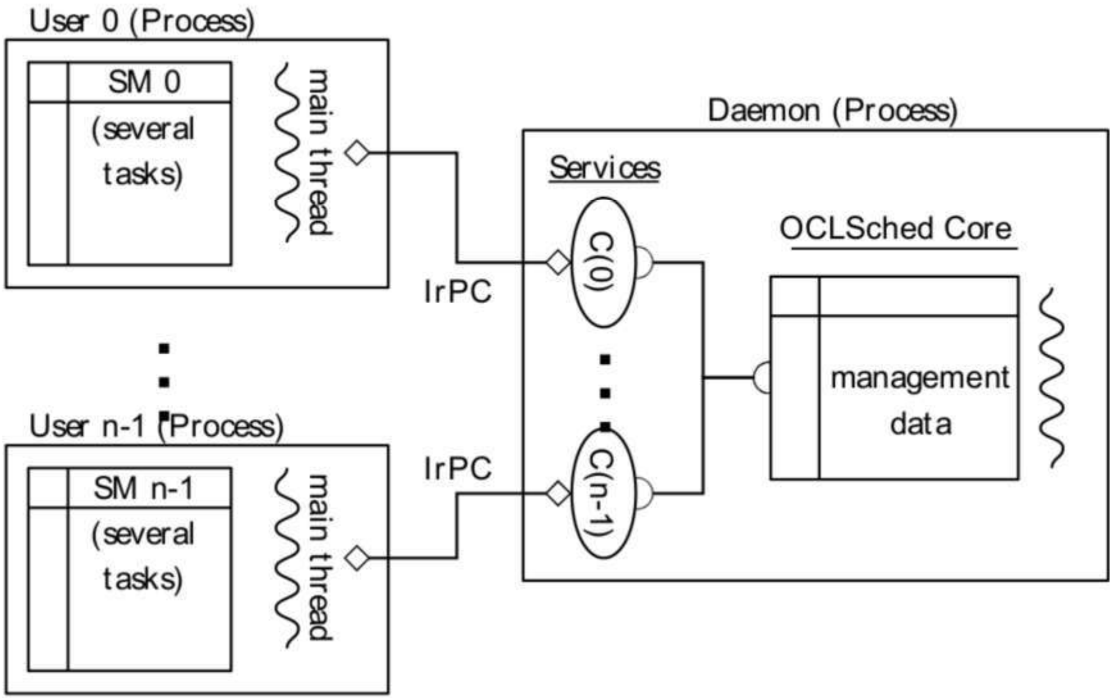

Figure 4.3: OCLScheduler client-server model [65]

In [66], the authors introduce HyperFlow which is a pipelined architecture that makes use of the various processing elements in heterogeneous systems in order to process data and achieve better utilization of the available resources. In their work and in order to facilitate the achievement of HyperFlow architecture, the authors defined Task-Oriented Modules (TOM) which is an abstract computational module that has many implementations for each processing element.

They also defined Virtual Processing Elements (VPE) as an abstraction layer for the actual physical resources. The TOMs enable the construction of computational pipelines that can run on different virtual processing elements. HyperFlow handles the scheduling of TOMs on the VPEs thus eliminating huge development effort from developer side. HyperFlow architecture is illustrated in Figure 4.4.

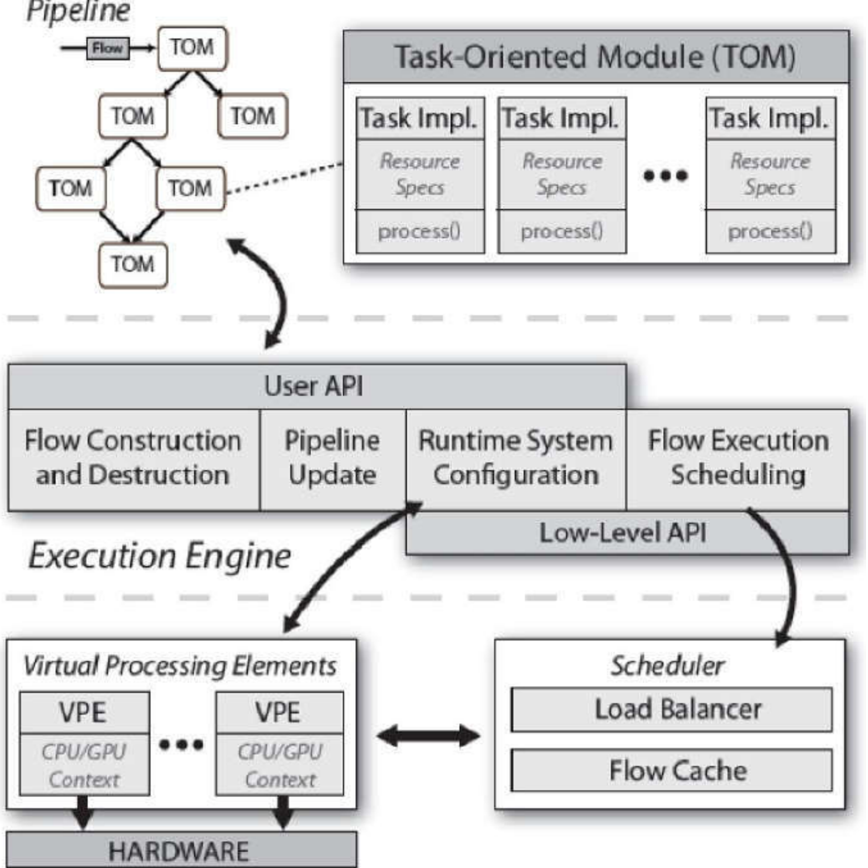

Figure 4.4: Hyperflow architecture [66]

HyperFlow workflow is very similar to the topology concept in Apache Storm. It provides by default two types of VPEs, CPU VPE and CUDA VPE. The user can create a custom VPE for any different type of computing resource. HyperFlow is suited for various types of applications and supports map-reduce frameworks. The major limitation in the mentioned implementation of HyperFlow is the lack of multi-node cluster support, although the authors included this feature in their future work.

In [67], CUDA framework is extended to Compute Unified Device and Systems Architecture (CUDASA) . CUDASA extends CUDA framework to multi-GPU and multi-node clusters. Hence, the parallelism level has been extended from GPU device level to multi-node level. However, CUDASA is only limited to CUDA and Nvidia GPUs. It also has some limitations in scalability due to task and data communication between the different cluster nodes.

In [68], the scalability of Hadoop map-reduce and ready-to-use algorithms in OpenCV library are combined together to provide large scale image processing on the cloud. The

results obtained in the experiments show great throughput compared to sequential processing of the images. However, the work didn't approach the GPU utilization challenges in big-data frameworks like Hadoop.

4.4 Conclusion

In this chapter we review many of the state-of-art efforts in incorporating GPUs in big-data frameworks specially map-reduce framework. The different architectures proposed in these research efforts tackle several challenges in enabling GPUs utilization in big-data clusters. However, each architecture tackles specific challenges and use cases. Most of the proposed solutions target accelerating traditional big-data applications like text and log analysis thus they obtain orders of magnitude performance gain. Few of these solutions target computer vision applications which are computationally intensive. We can summarize the following limitations in the reviewed work:

1. Flexibility: Nearly most of the proposed solutions are limited to map-reduce framework and ignoring more flexible programming models introduced in modern big-data frameworks such as Spark and Storm.
2. Development Efforts: Some of these solutions are very low-level and require comprehensive understanding of GPU programming frameworks such as OpenCL and CUDA, hence they ignore ready-to-use implementations present in libraries such as OpenCV.
3. Portability: The generality of the proposed solutions is limited as some of these solutions are either hardware limited by supporting specific Nvidia GPUs or software limited by supporting either OpenCL or CUDA frameworks.
4. Scalability: The cluster size in some of the proposed solutions is limited to only one node and some solutions support only one GPU per cluster node.
5. Real-timeliness: Most of the architectures proposed don't support real-time computer vision processing.

In the next chapter we will present a scalable, fault-tolerant, flexible, generic big-data cluster for both real-time and offline computer vision applications that supports multi-GPUs per cluster node and utilizes ready-to-use computer vision algorithms provided by open source libraries.

Chapter 5: Design and Implementation

5.1 Introduction

In this chapter we introduce a novel approach in processing real-time video feeds or files using big-data frameworks. We will also illustrate the main difference between using GPU and not using it from architectural point of view. Moreover, we will illustrate the major challenges of running computer vision applications on big-data clusters and how we tackled these challenges in our architecture. First we will start with the homogeneous approach for most big-data frameworks which relies on CPUs only or what is known as homogeneous clusters. Then we will discuss how we enabled GPU utilization in big-data clusters specifically for computer vision applications.

5.2 Computer Vision Processing on Big-data Frameworks

In this section we elaborate the main idea of processing computer vision applications on big-data frameworks. Most of the computer vision applications existing nowadays depend on processing individual images captured from video records, cameras' live feeds or discrete images. The processing involves several steps as illustrated in Chapter 2. Many of these steps incorporate sophisticated algorithms that are applied on the input images. These algorithms yield another form of data that keeps flowing in the application logic until the final transformation is applied and then we get a meaningful and structured representation of the objects in the image.

In order to port the computer vision algorithms to big-data frameworks, we must consider the parallel nature of big-data applications. In other words, most applications that run on big-data frameworks can be parallelized without dependencies between the data elements that are being processed, hence an algorithm relies on the sequence of the fed data may not be suitable or will not certainly leverage the horizontal scalability inherited by default when it is run on big-data frameworks.

In computer vision applications, many algorithms depend on the sequence of images in order to produce correct results. An example of such algorithms is: Optical Flow which determines the moving objects in a sequence of frames or video based on the information from the current and previous images, therefore, a motion detection application that relies

on Optical Flow algorithm may not be the best fit for big-data frameworks.

On the other hand, there is a category of computer vision applications that doesn't necessarily impose inter-data dependency but they require special modifications in order to be ported to big-data technology without degrading the accuracy. For example, in an object detection application, if the video frames are processed independently and given that the video has a frame rate of 30 frames per second, then an object that appeared for a second in the video will be detected 30 times or maybe less depending on the accuracy of the algorithm itself. Since the processing of each frame is independent of others, then falsely an object detection algorithm based on a big-data framework will yield redundant results.

In order to increase the throughput of a computer vision algorithm, some developers tend to discard a group of frames each second of the video, thus reducing network bandwidth in case the video frames are sent over the network and at the same time reducing the latency of processing 1 sec of video to (N x FrameProcessingLatency), where N is the number of frames out of M which is the original frame rate and M is greater than N, instead of (M x FrameProcessingLatency). This leads to a degradation in the accuracy of the algorithm and missing potentially important information in the discarded frames.

Although discarding frames may seem to be a good approach for porting computer vision applications to big-data world because it limits the redundancy in the output of processing video frames individually, one cannot guarantee that the dropped frames will surely remove the redundant data.

For example, consider a case where we have 5 seconds of video recordings being processed, these recordings are generated with 30 frames per second rate. We choose to drop 29 out of each 30 frames; It could happen that an object appears in the 5 seconds continuously, therefore it would be stored as 5 different objects which is a false result.

In order to maintain the sanity of the application, one should have a way of tracking the detected objects in the subsequent frames in order to combine the results or reduce them so that no redundancy occurs. In most of the big-data frameworks it isn't guaranteed that the concurrent running instances of the applications can communicate and exchange the data between each other, therefore we have to tackle the redundancy of the data after the processing of the frames and not during the processing itself.

In order to address the redundancy issue, we considered the map-reduce programming model. In this model we process the video frames in parallel across different threads

and then apply a reduce algorithm on the output of these threads to combine and reduce the results. In Figure 5.1 we illustrate the main idea behind processing computer vision applications on big-data frameworks.

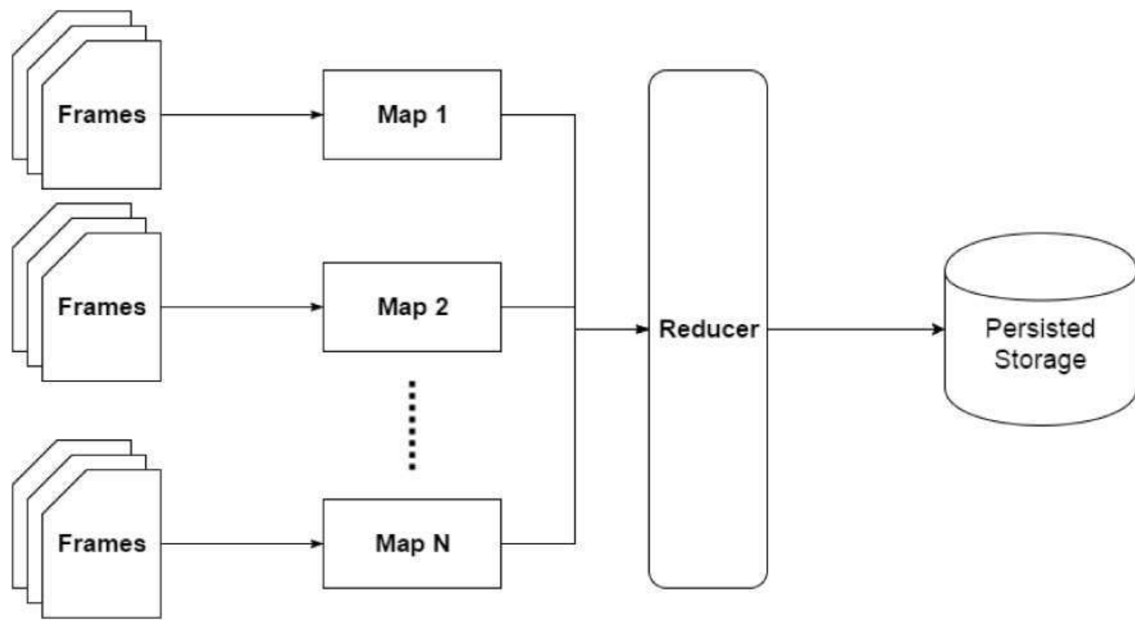

Figure 5.1: Map-reduce workflow in computer vision applications

In the map-reduce architecture depicted in Figure 5.1 the Maps are the elements executing the developer's computer vision algorithm on CPU threads only with no GPU support. It is the developer responsibility to implement the Reducer algorithm to eliminate the redundant data as explained earlier in this chapter. The reducer algorithm is not compute intensive compared to the map algorithm. This is because reducer processes structured text data emitted from the Maps. For example, in License Plate Recognition algorithm, when the Maps emit license plates as text extracted from the input frames the reducer then operates on these text license plates. This is why the work in this thesis is concerned with enabling GPU for the Map stage not the reducer. Hence, any algorithm proposed in this thesis is not concerned with the Reducer performance.

In Storm, an application is described in terms of topology as illustrated in Chapter 2. The nature of Storm topologies enables it to be the best fit for real-time video processing. In order to process visual data on Storm cluster, we define a topology where a spout is responsible of reading video files or images then it sends discrete frames to the bolts in a round-robin fashion. This way Storm enables processing discrete videos frames and not video chunks as in Hadoop and Spark, besides, Storm topologies are persisted in memory and are not destroyed except if the user killed the topology explicitly. Storm also

guarantees fault-tolerance thus each frame is processed at-least-once, which means no data loss.

In Figure 5.2, we introduce an architecture for processing computer vision applications on Storm cluster. In this architecture, each bolt can act as a different computer vision algorithm or multiple bolts can act as the same computer vision algorithm to increase the parallelism level and enhance the overall throughput. For example, a video file on local disk will be read by a spout and then send frames to a specific bolt based on the developer configuration. In this architecture all of the bolt processing is performed by x86 CPU and its vector extensions only with no GPU at all.

A typical big-data cluster consists of multiple processing nodes, each node contains multiple CPUs. A CPU has multiple cores that can handle multiple application threads at the same time. The presence of multiple cores doesn't create a racing condition when multiple computer vision applications run at the same time as long as there are sufficient cores. The main problem when GPU is enabled for these parallel instances is that the computational resources of the GPU cannot handle many computer vision applications running at the same time. The main reason for this is the competition on the GPU cores between these instances.

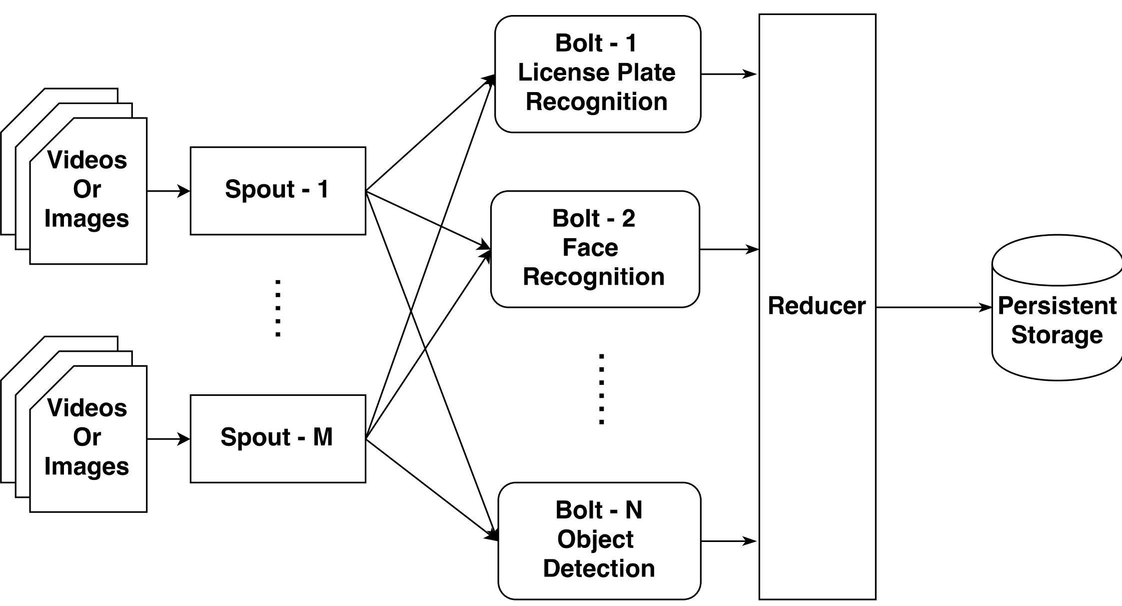

Figure 5.2: CPU based Storm topology for computer vision processing

GPUs are known to have a lot of cores, hundreds and maybe thousands, and that no competition might occur when multiple instances of the algorithm are run on the same

GPU. This is true for applications that doesn't incorporate massive amount of data to be processed in parallel, as discussed earlier in Chapter 4 a word count application can leverage the computational power of the GPU without causing resource competition. However, when it comes to computer vision applications, a single instance of an Object detection algorithm and based on the video resolution, classifier size ... etc, can occupy more than 95 % of the processing power of the GPU because of the multiple pixels that are processed simultaneously.

In Chapter 6, we evaluate the performance of the current implementation of Storm topology for processing computer vision application on CPUs only compared to the case when GPU is enabled for all the running bolts. The results prove that a competition between the bolts on the GPU leads to a drop in the throughput of the system by increasing the average processing latency of the video frames.

In the next section we introduce the procedures and steps carried on to enable hybrid CPU-GPU execution in Storm topology to increase the throughput of frame processing.

5.3 Computer Vision Processing on Big-data Frameworks with GPU Support

In order to enable GPUs in big-data for processing computer vision applications and increase the throughput, several procedures and trials have been carried out to implement a scheduler for GPU tasks. Several schedulers have been investigated until we reached the best one in terms of performance. These schedulers and their underlying algorithms are detailed as follows:

5.3.1 Priority-based Static Scheduler

In this scheduler, the computer vision algorithm is divided into several stages, each stage has a priority index which indicates the scheduling priority of this stage. A stage with higher priority will run on the GPU and other stages with lower priority will run on the CPU. We used License Plate Recognition (LPR) application to study this scheduler implementation. The high level workflow of an LPR algorithm is that a video frame is passed through a detector which detects the presence of license plates in the frames based on a previously trained classifier. The detected plates in the frame are represented as

rectangular co-ordinates which are used as input to the next stage, the segmentation stage. This stage segments the characters in the license plate rectangle before finally sending them to the Optical Character Recognition (OCR) stage which recognizes the characters in the segmented parts of the rectangle and transforms these segments to text characters.

In the License Plate Recognition algorithm, the first stage which is the Detector is more compute demanding. Therefore, the detector will get higher priority index than the segmentation and the OCR stages. This GPU scheduling technique schedules the Detector to run on an entry bolt and then this bolt emit the output of the Detector to another layer of bolts that implement the remaining stages of the LPR algorithm . This scheduling technique released the computational load on the GPU by scheduling less compute demanding stages on the CPU and leaving the GPU for the other stages. The Storm topology with this scheduler is depicted in Figure 5.3.

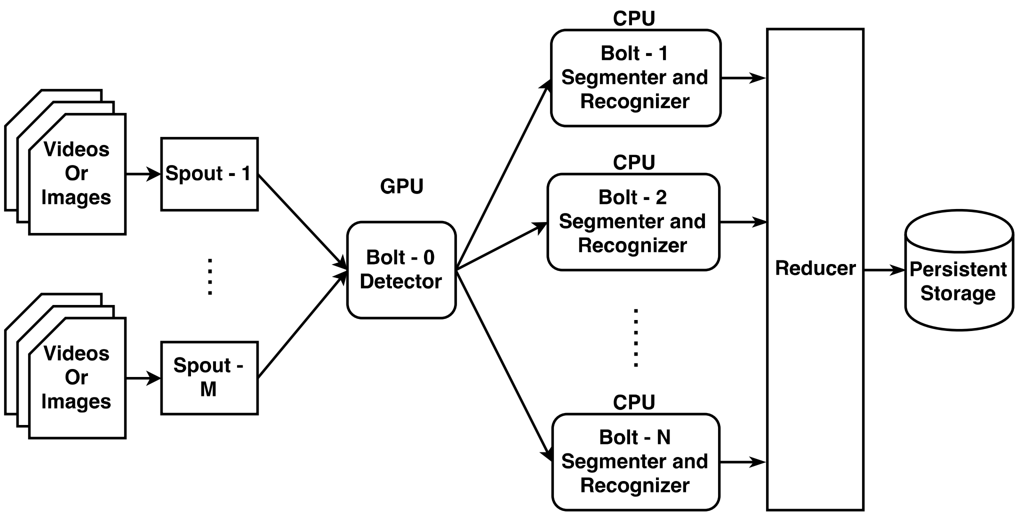

Figure 5.3: GPU priority-based scheduler

This scheduler has two main limitations, first, the priority index is statically assigned before job start, therefore, the developer has to know which stage is more demanding than the others and assign it highest priority. The second limitation is that given a stage that has low performance gain when run on a GPU compared to CPU, then the overall performance will be limited by this performance gain value. In the license plate recognition case, we run a Storm topology where the detector is run on the GPU its processing throughput is doubled nearly 5 times compared to the CPU version. However, the delay in the segmentation and OCR stages when run on the CPU degrades the overall performance gain.

5.3.2 GPU Utilization-based Scheduler

In this scheduler, a software service is created to measure the utilization of the GPU in terms of memory and cores utilization e.g. 1024 MegaBytes memory and 65 % cores. The application developer specifies the threshold utilization values for the GPU cores and memory before run-time statically, the scheduler decides to grant access to the GPU for the entity requesting access if the current measured utilization values are below the configured threshold. This scheduler workflow for a single bolt in the topology is illustrated in Figure 5.4.

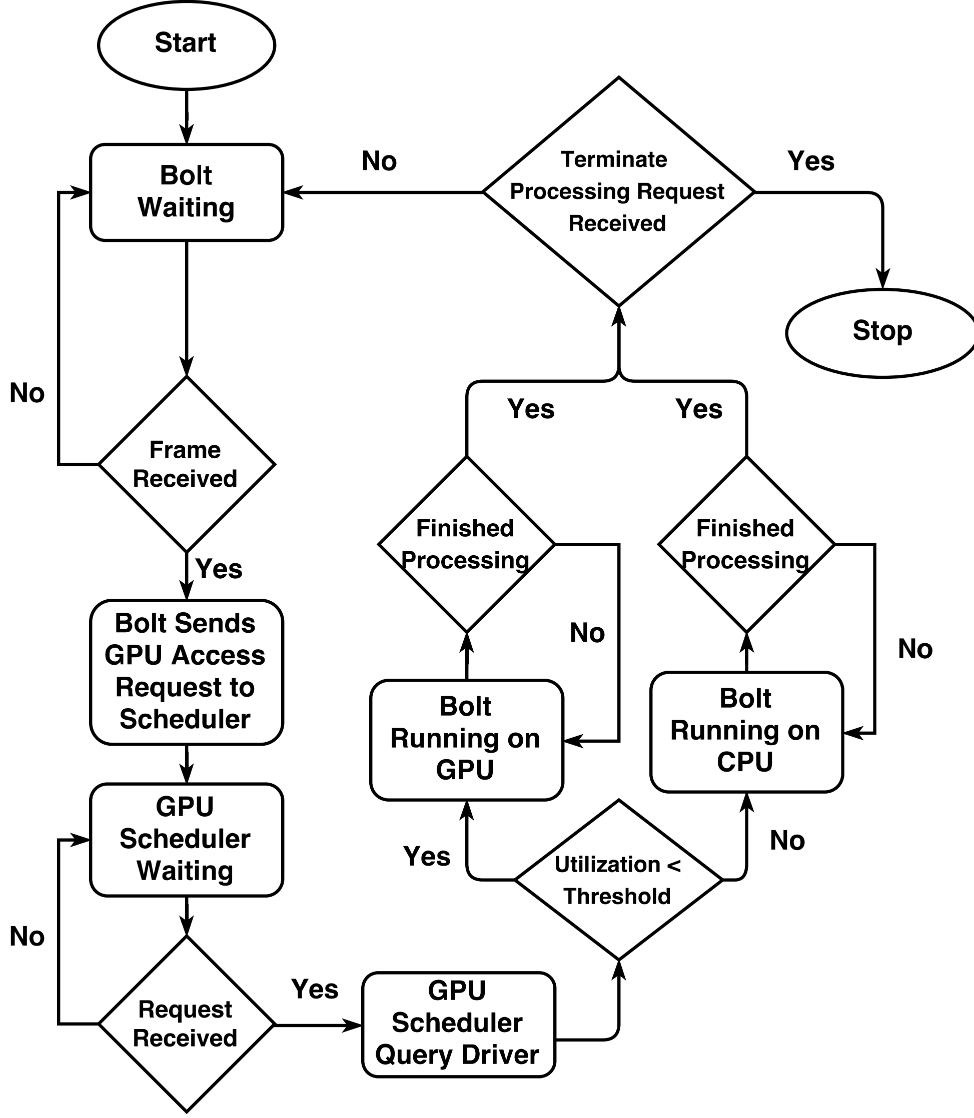

Figure 5.4: GPU utilization based scheduler workflow

The utilization based scheduler guarantees that no resource allocation issue could occur in the GPU due to multiple bolts utilizing it at the same time. The new Storm topology is shown in Figure 5.5. However, this scheduler requires special support from the GPU driver in order to facilitate the probing of the GPU cores utilization and memory usage. In other words, not all GPUs support queries about their current utilization values. Besides, after several experiments, the utilization values of the GPU cores are not always the best criteria for scheduling tasks on the GPU. For instance, a bolt can utilize more than 95 % of the GPU cores therefore the utilization based scheduler may not run any other bolt on the GPU. However, this decision sometimes degrades the performance.

To further illustrate this point, consider that we have a bolt that runs an algorithm with processing delays of 2 milliseconds (ms) and 10 ms when run on the GPU and the CPU respectively. This algorithm occupies 95 % of the GPU cores when run on the GPU. The penalty of running another bolt of the same algorithm on the GPU is that the processing latency of the GPU doubles to be 4 ms. In this situation a utilization based scheduler will decide not to run the second bolt on the GPU which means that the second bolt will run with 10 ms delay on the CPU instead of 4 ms on the GPU.

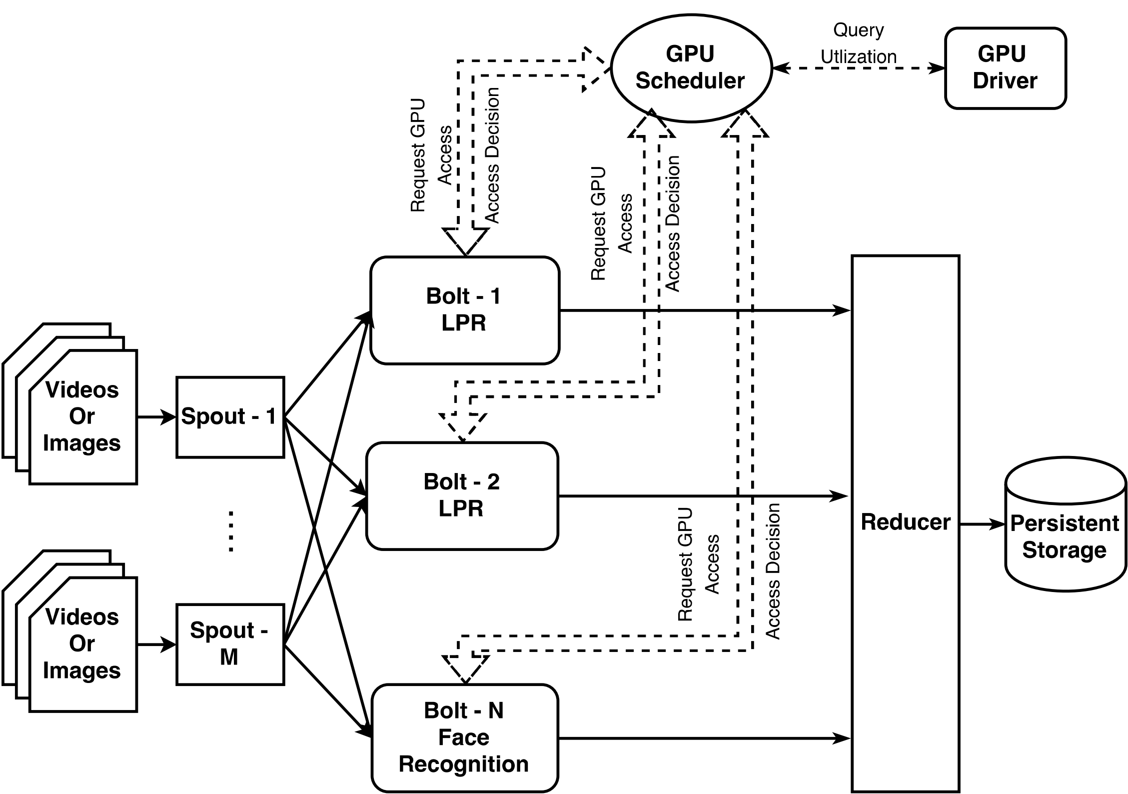

Figure 5.5: Storm topology with utilization-based GPU scheduler

5.3.3 Single Allocation Scheduler

In this scheduler, a software service is created to manage GPU access. A bolt sends request to the GPU service to access the GPU device, then the GPU service checks the current GPU occupancy and if not occupied it will grant the bolt access to the GPU else it will deny access. The bolt uses the reply from the GPU service to decide whether it will use the GPU or the CPU for processing the current video frame or image. The allocation process is thread safe so that no multiple allocations occur at the same time falsely. The flowchart of this algorithm for a single bolt is depicted in Figure 5.6. This algorithm has several advantages illustrated as follows:

1. Generic: This scheduler can be used with any GPU as there is no restriction imposed by the device driver APIs as the case with the Utilization based scheduler.
2. No competition: In this scheduler, only one bolt at a time can access the GPU, therefore, there is no competition on the GPU from other bolts.
3. Simple implementation: There is no complex calculation required to implement this scheduler. The scheduler tracks the available GPUs on a node and grant access to a certain GPU if and only if it isn't currently occupied.

A Storm topology with this scheduler is shown in Figure 5.7. However, this scheduling technique, has a major drawback which is GPU under-utilization. In some cases, the computer vision algorithm running in a bolt doesn't fully utilize the GPU computational resources. At any given time instance, the scheduler can assign the GPU to any bolt however, the GPU utilization could be less than 50 % when this bolt uses the GPU. At this exact moment any other bolt cannot access the GPU which leads to under-utilization of its computational resources.

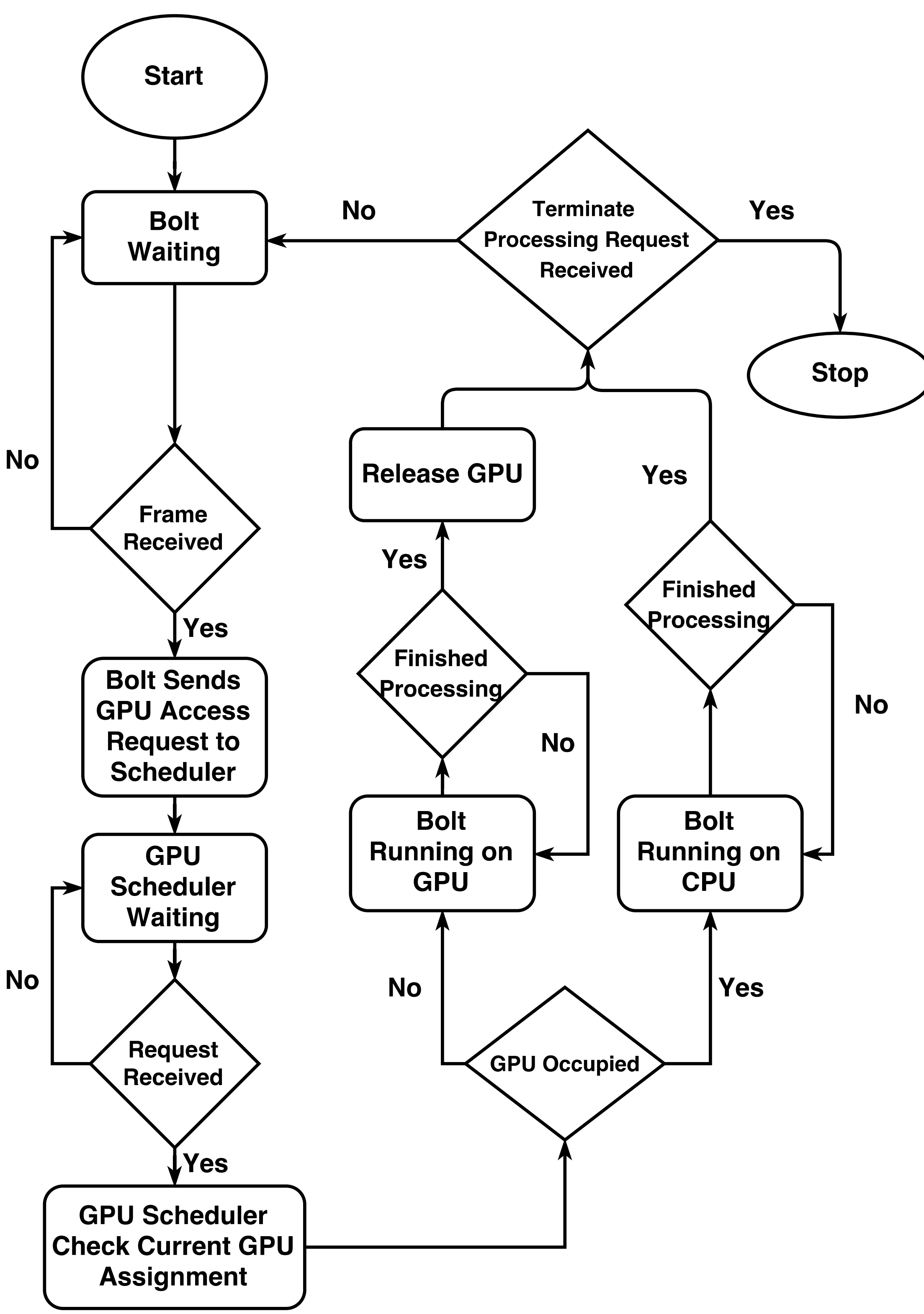

Figure 5.6: Single allocation GPU scheduler algorithm

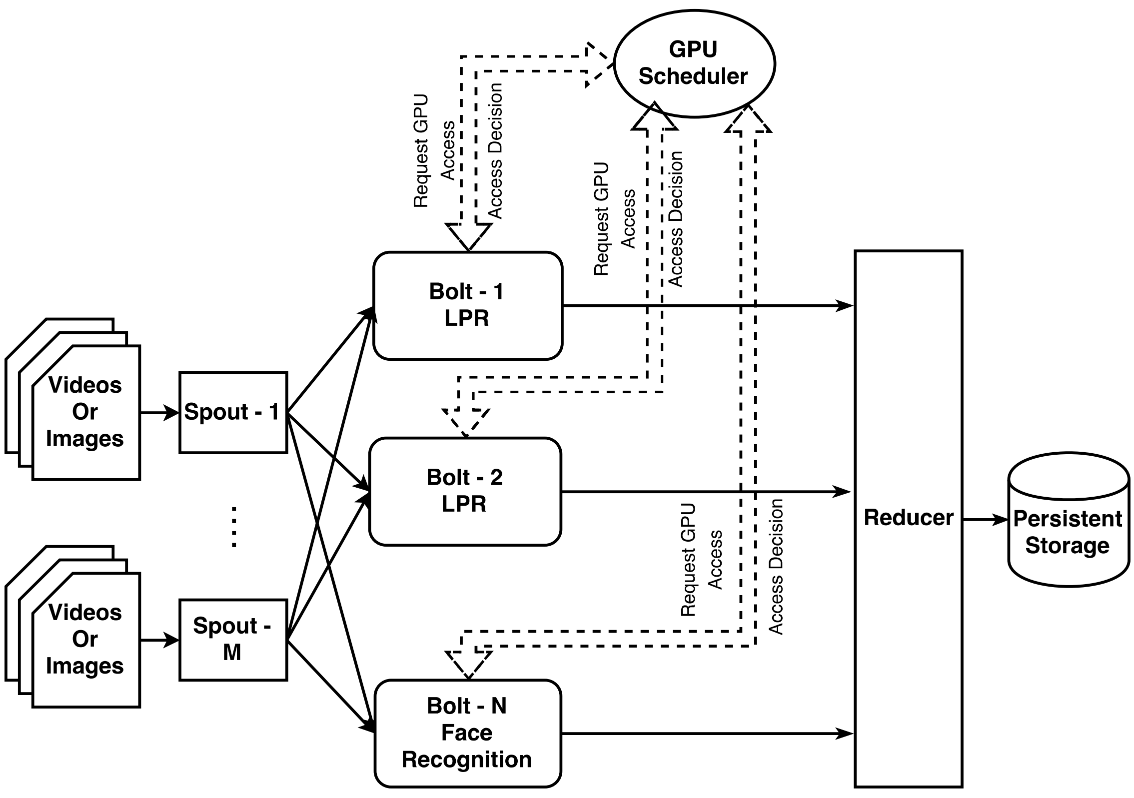

Figure 5.7: Storm topology with single allocation GPU scheduler

5.3.4 Token-based Scheduler

In the single allocation scheduler there is a drawback which is under-utilization of the GPU in case the computer vision algorithm is not utilizing all the cores on the GPU efficiently. The utilization of all the GPU cores is determined by several factors. The most dominant factors are the algorithm complexity compared to the available GPU cores and the input data size compared to the GPU memory.

In this context the GPU scheduling is very similar to the well known medium access problem in computer networks field, where a group of machines need to access the transmission medium in order to send messages to each other. However, if any machine send a message at any time without any co-ordination amongst each others, then a contention of the medium might occur because of the interference between the different messages sent at the same time. Hence, the medium cannot be utilized by any node.

In order to prevent this contention, a token based algorithm is used. In this algorithm, a token representing the medium access right for any node that holds it, is circulated fairly between the nodes sharing the medium. So, at any given time instance, only one node

can access the medium and sends its messages. Of course fairness is guaranteed between these nodes by defining message size and counts constraints to prevent any node from exclusively utilizing the medium all the time.

The medium access problem is very similar to the GPU access problem in big-data frameworks, where multiple instances of the algorithm are trying to access the GPU simultaneously which leads to competition on the available GPU resources ”Memory and Cores”. This competition increases the average frame processing time as shown in Chapter 6.

We introduce a modified token based scheduler that guarantees no competition on the available GPU resource and high utilization of the GPU based on a static configuration defining the capacity of the GPU device. In this scheduler, a GPU service is configured with a certain number of tokens where each token represents a GPU access right to the entity that posses it.

In Storm topology, each bolt can request a token to access the GPU from the scheduler service. Whenever a token is assigned, the scheduler service decrements the number of available tokens by one. It tracks the remaining tokens available for the GPU and whenever a bolt finishes using the GPU, it sends a release-token request to the scheduler which increments the available tokens by one. The algorithm of this scheduler for a single bolt is depicted in Figure 5.8. Storm topology for token-based scheduler is the same as single allocation scheduler shown in Figure 5.7, hence, one can think of a single allocation scheduler as a special case of the token based scheduler with number of tokens equals one.

5.4 Conclusion

In this chapter we proposed several algorithms to address the challenges of using GPUs in big-data frameworks. The main challenge is to eliminate the competition between multiple processes trying to utilize the GPU at the same time. The framework we chose for validating the schedulers implemented in this work is Storm. Storm has minimal latency and overhead compared to other frameworks such as Hadoop and Spark. However, the GPU schedulers proposed in this work are totally compatible with any parallel programming framework for computer vision applications. However, we recommend that if the developer writes his own CUDA or OpenCL code without using libraries like OpenCV, that he will make sure that the GPU memory allocations are not beyond the GPU's memory to avoid out-of-memory exceptions. In the next chapter we evaluate the

performance of the Exclusive-allocation and Token-based schedulers showing the gain from using GPUs big-data frameworks for computer vision applications.

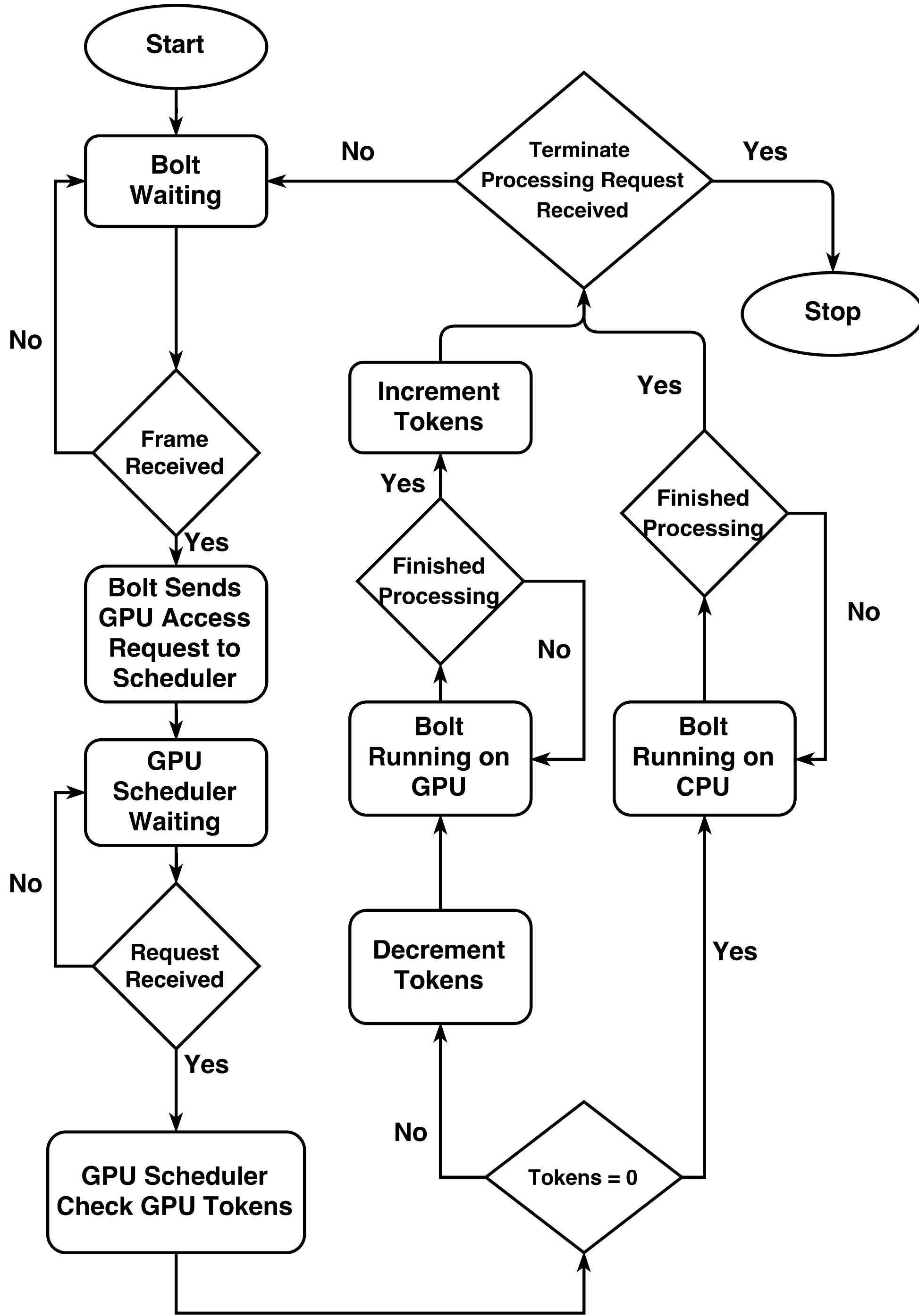

Figure 5.8: Token-based GPU scheduler algorithm

Chapter 6: Experimental Results

6.1 Introduction

In this chapter we evaluate the performance of GPU enabled Storm architecture compared to the case where there was no GPU support. Also we compare the performance of our accelerated architecture compared to Nvidia's Multi-Process Service (MPS) .

6.1.1 Hardware Setup

The evaluation test-bed is based on Apache Storm framework version 0.10.0. We setup a cluster consisting of a master node and a single slave node. The master node is a Desktop PC and the slave is a Dell server model PowerEdge R730 the specs of both nodes are detailed in Table 6.1. The GPU we used on the slave node is Nvidia's Tesla K40 with 2880 CUDA cores and 12 Giga-Bytes global memory.

Table 6.1: Master and slave nodes hardware specs

	Master	Slave
Processor	4x Intel core-i7	2x Intel Xeon
Cores per Processor	2	12
Intel Hyper-Threading	Enabled	Enabled
Memory	16 GB	128 GB
Network	1port - 1 Gigabits Network card	4 ports - 1 Gigabits Network card
Storage	1 TB SATA hard disk	1 TB SAS hard disk

The network connection between the two machines is one Giga-bit connection. In order to make sure that there is no processing power allocated to any software or hardware installed on the slave node, the display was detached from the PowerEdge machine, no secondary applications were running during the evaluation. The hardware setup is depicted in Figure 6.1

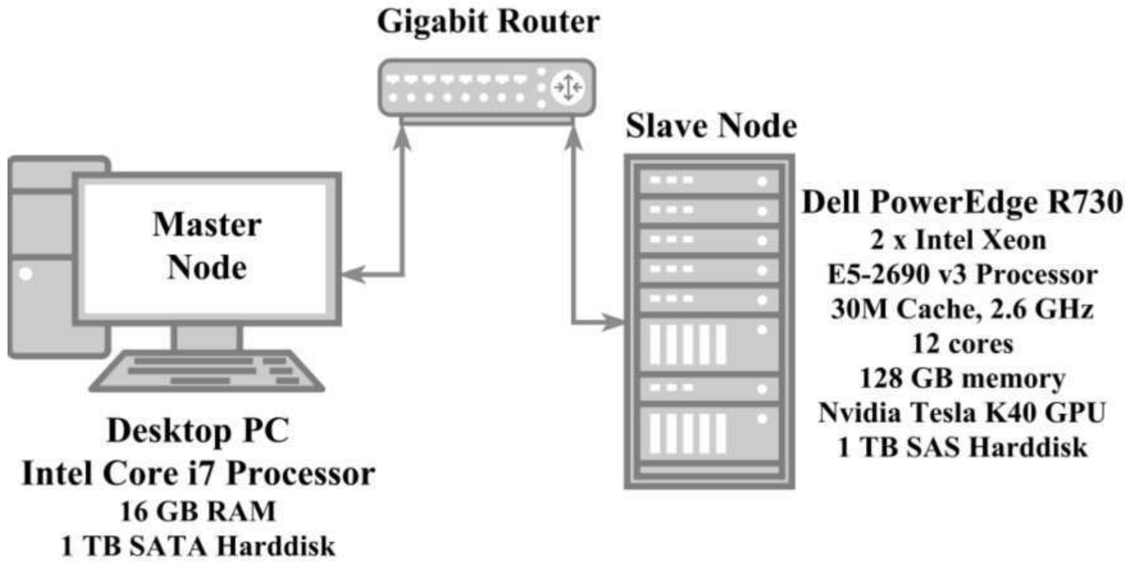

Figure 6.1: Evaluation test-bed setup

6.1.2 Software Setup

The computer vision library used in these tests is OpenCV version 3.0.0. The operating system is Ubuntu desktop x64 version 14.04.04 with Kernel version 3.13.0-68-generic. The computer vision algorithm used in these tests is object detection based on cascade detector. The algorithm of this detector was first introduced in [69] and is known as Viola-Jones algorithm. It consists of several steps which build a classifier that decides whether an image contains a specific object or not. In order to achieve this goal, certain features are extracted from the training data. These training data is a set of images where some of them contain the object to be detected and the remaining images don't contain the object. In machine learning terms, these images represent positive and negative training set.

The Viola-Jones algorithm was originally developed for detecting faces and later it was extended for generic objects. Hence, the features used to train the classifier were suitable for detecting specific characteristics of human face, like the shading difference between the eyes and the area between the eyes. Example of these features are depicted in Figure 6.2 where the features are single values obtained by summing pixels in the white region of the rectangle and subtracting them from the sum of pixels in the black region.

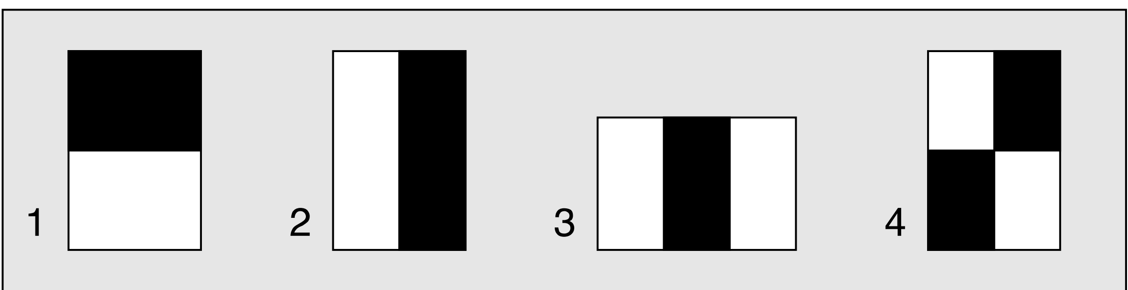

Figure 6.2: Example of rectangular features

Using features instead of pixels to detect objects requires less computational power. However, in order to extract every feature from an image, the algorithm starts with small window of size 24x24 pixels of the image and can scale to the maximum image size based on a scaling factor. Hence, the amount of features to extract from an image is huge, for example a 24x24 window can yield 180,000 features [69].

In order to reduce the computational complexity of calculating the sum of pixels for each feature, the concept of integral image was introduced where the sum of pixels at one point is a function of previously calculated points above and to the left of this point.

In this context, some features aren't relevant for some regions of the image thus a selection process is performed to select the best features in terms of detecting the faces in the images. For example, in Figure 6.2, feature number three is adequate for detecting the region of eyes in human faces but it is irrelevant to the forehead area. Hence the authors of the Viola-Jones algorithm used the Adaptive Boost (AdaBoost) learning technique where a strong classifier is constructed from several weak classifiers. Each weak classifier cannot detect faces solely, but combined with other classifiers they achieve high detection rates.

Moreover, applying all the features selected by the AdaBoost algorithm is time consuming and computationally exhausting, therefore the authors introduced the Cascade Classifier concept. In this concept the features are grouped into stages, each window passes through each stage and if it passed then it continues through the rest of the stages. The window that passes all the stages has a face within it, this flow is depicted in Figure 6.3.

The Viola-Jones algorithm is a main pillar in many object detection computer vision applications whether this object is a license plate, human face, car or generic object, as long as the classifier is trained to detect the desired object, the algorithm will detect it.

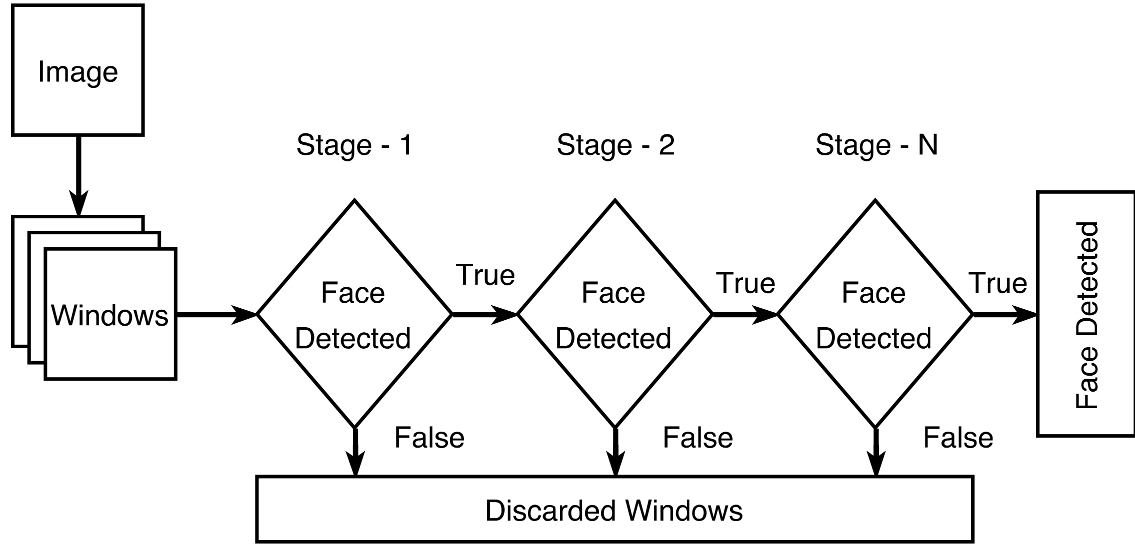

Figure 6.3: Mutli-stage cascade classifier

The Nvidia driver used in this test is the latest Nvidia Ubuntu drivers version 352.79 that support Tesla K40 device. CUDA 7.5 run-time and development tool kit were installed too. On the master node, Zookeeper, Storm Nimbus and Storm UI are installed. On the slave node, Storm supervisor and log viewer are installed. The GPU scheduler can be run on the master or slave node. It reads a user configuration about the available GPUs in the cluster and the tokens available on each GPU. In our setup the scheduler runs on the master node and communicates with bolts through the network.

In our evaluation we are interested in bolts (maps) performance with hybrid CPU-GPU and not the reducer, therefore, in these tests the reducer phase is discarded from the setup of Storm topology. The CPU-only based topology used in this evaluation is shown in Figure 6.4 and the hybrid CPU-GPU topology is shown in Figure 6.5.

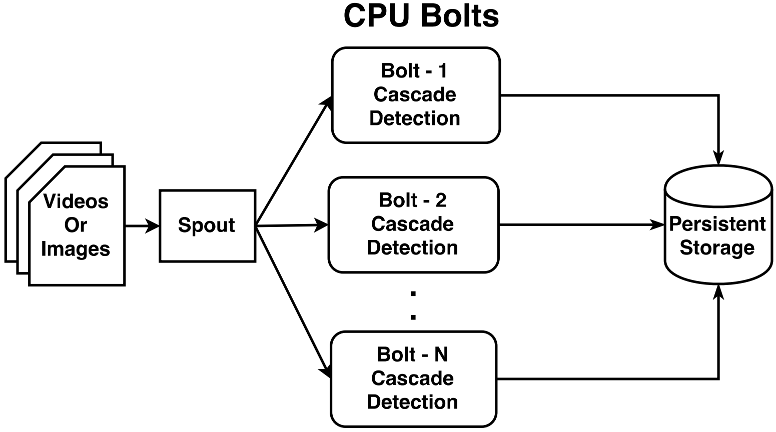

Figure 6.4: CPU-only Storm topology

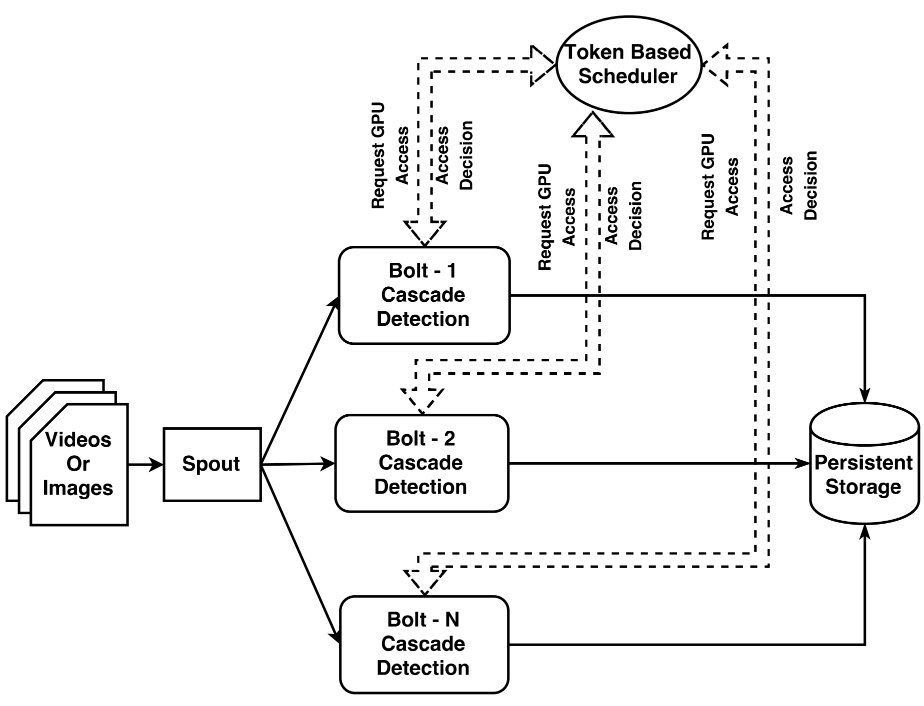

Figure 6.5: Hybrid CPU-GPU Storm topology

6.1.3 Evaluation Procedure

In OpenCV and as indicated in Chapter 3, there are multiple implementations for most of the computer vision algorithms. In our evaluation we used the CPU, OpenCL and CUDA implementations of the cascade detector. The application starts with loading the trained classifier in memory, then start preparing the input frame by converting it to grey scale and finally invoking the detectMultiScale method on the transformed frame which returns the pixel co-ordinates of the rectangles that include the objects to be detected.

One of the challenges we faced was that the CPU implementation of the detectMultiScale method in OpenCV automatically uses the GPU if OpenCV was built with OpenCL enabled. In order to enforce the computation mode and override OpenCV implementation, we added another argument to the detectMultiScale method. This argument is called ”GPUEnabled”, its a boolean variable with default false value. This variable is used inside the detectMutliScale method to disable OpenCL functionality when set to false and enable it when set to true. This modification is generic and can be applied to all OpenCV algorithms that have CPU and GPU implementations. According to the scheduling decision made by the GPU scheduler, the value of the GPUEnabled variable is set either to true or false.

The video file used in this evaluation has 1080p "1920x1080" resolution, 55 minutes and 4 seconds duration, mpeg encoding, avi extension, and 30 frames per second. The procedure of the testing is to run the detection algorithm on Storm topology in order to process the input video file where each frame processing delay is reported in Storm logs. The number of processed frames is considered our main performance metric and referred to as Frames Per Second (FPS) . Each run on Storm either with or without GPU corresponds to a number of parallel bolts running concurrently, so the first run is for 1 bolt the second run is four bolts then eight and increasing by step size of four bolts until we reach 24 bolts which corresponds to the total number of cores available on the slave machine.

To illustrate how we calculated the enhancement, let's assume that we have two cases A and B. We want to know how much enhancement we got from using B instead of A. The enhancement in this case is referred to as E and calculated as follows:

$$E = \frac{T_A - T_B}{T_A} \tag{6.1}$$

where T_A and T_B are the average frame latency in case A and case B respectively. The overall enhancement is calculated as the average of enhancement between case A and B for different number of concurrent bolts.

The total video processing time is calculated as follows:

$$Frames = VideoDuration * FramesPerSec \tag{6.2}$$

$$ProcessingTime = \frac{T * Frames}{N} \tag{6.3}$$

where $Frames$ is the total number of frames in the video, $FramesPerSec$ is the frame rate represented as frames per second, $ProcessingTime$ is the total video processing time in seconds and T is the average frame latency. N represents the parallelism level of the job or number of concurrent bolts.

6.1.3.1 GPU Scheduler for OpenCL framework performance evaluation

Earlier in this thesis we explained how that enabling GPU access for all bolts can lead to competition on the GPU resources thus degrading the performance and therefore we

had to disable GPU for all bolts and rely on CPU only. To validate the competition and quantify its effect on the overall system throughput, we enabled GPU access for all bolts as a part of the evaluation.

In this section we evaluate the performance of the cascade detection algorithm on Storm topology with GPU scheduler compared to the traditional approaches where GPU is enabled and also when it is disabled for all bolts. The GPU framework evaluated in this section is OpenCL and the same evaluation steps are followed as explained earlier in this chapter.

We evaluated the single allocation scheduler, two tokens and we wanted to further test the capacity of the GPU with the cascade detection algorithm so we extended the tokens to four and collected results which are shown in Figure 6.6.

As shown in the obtained results, the competition between the concurrent bolts when the GPU is enabled degrades the performance significantly and decreases the FPS rate compared to the CPU-only case. In the case of using the CPU only we notice that the performance is nearly constant and this is due to the availability of 24 cores on the slave machine, thus limited competition between the bolts when run on the CPU only. In case the number of cores was less than the number of concurrent bolts and given that a bolt can consume up to once core, then we might face racing on CPU but this is out of the scope of this thesis.

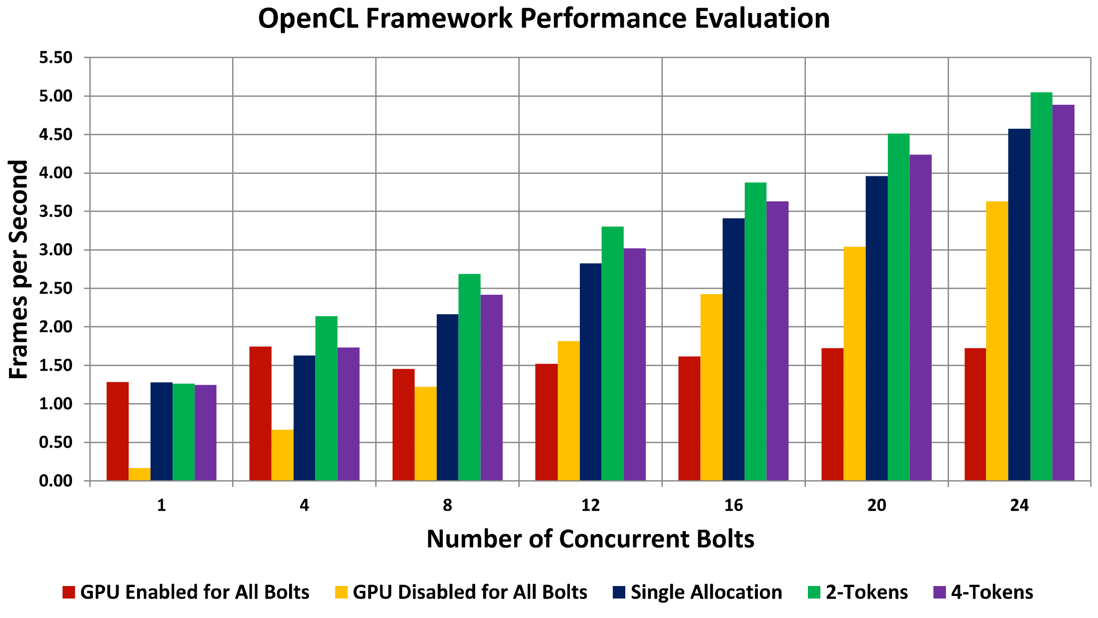

Figure 6.6: OpenCL performance evaluation

The Single allocation scheduler yielded a better performance than the CPU-only and GPU-only cases however, the two-tokens based scheduling achieves better results and this is due to the relatively large computational power of the Tesla K40 device with 2880 CUDA cores and 12 GB memory. This GPU is able to accommodate two bolts running OpenCL version of the algorithm.

The results also shows a degradation compared to the two-tokens case when we extend the number of bolts to four. This means that with four bolts accessing the GPU, a competition between them starts to impact the performance. Therefore, the best capacity for the GPU for this specific test case is two bolts at a time.

The performance metrics for OpenCL framework for the different scheduling options is illustrated in Table 6.2. The enhancement in the average frame latency is tabulated in Table 6.3 and it shows that compared with the case where the GPU is enabled for all bolts, the enhancement increases with increasing number of bolts. However, compared with the case where the GPU is disabled for all bolts, the enhancement decreases. This aligns perfectly with what we stated before about the severe racing that impacts the performance when the concurrency level increases on the GPU. On the other hand, since the CPU already has 24 cores, therefore, the FPS is nearly constant up to 24 bolts when the GPU is disabled. Hence, the effect of the GPU enhancement is divided between the increasing number of bolts.

The collected results, shows that with two-tokens based scheduling we can achieve up to 2x performance gain when compared to traditional CPU Storm topologies and up to 1.7x when compared to enabling the GPU for all the bolts. In the next section we will evaluate CUDA framework and show that the token-based scheduler achieves similar performance gain. However, the optimal number of tokens for CUDA case is different than OpenCL as will be illustrated in the next section.

Table 6.2: OpenCL framework performance metrics

Number of Concurrent Bolts	GPU Enabled for All Bolts		GPU Disabled for All Bolts		Single Allocation		2-Tokens		4-Tokens	
	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes
1	0.78	1287.87	6.01	9933.27	0.78	1292.74	0.79	1307.09	0.80	1324.02
4	2.29	947.73	6.02	2486.01	2.46	1015.81	1.87	772.43	2.31	953.45
8	5.51	1138.73	6.54	1350.41	3.70	763.15	2.98	614.42	3.31	683.39
12	7.90	1087.60	6.61	910.20	4.25	585.57	3.63	500.29	3.98	547.29
16	9.89	1021.25	6.59	680.87	4.90	506.15	4.13	426.61	4.41	455.00
20	11.61	958.61	6.58	543.52	5.05	417.31	4.43	366.11	4.72	389.78
24	13.92	958.33	6.61	455.11	5.24	360.81	4.75	327.12	4.91	337.94

Table 6.3: Frame processing latency enhancement for OpenCL framework

Number of Concurrent Bolts	Enhancement Versus GPU Enabled for All Bolts			Enhancement Versus GPU Disabled for All Bolts		
	Single Allocation	2-Tokens	4-Tokens	Single Allocation	2-Tokens	4-Tokens
1	-0.4%	-1.5%	-2.8%	87.0%	86.8%	86.7%
4	-7.2%	18.5%	-0.6%	59.1%	68.9%	61.6%
8	33.0%	46.0%	40.0%	43.5%	54.5%	49.4%
12	46.2%	54.0%	49.7%	35.7%	45.0%	39.9%
16	50.4%	58.2%	55.4%	25.7%	37.3%	33.2%
20	56.5%	61.8%	59.3%	23.2%	32.6%	28.3%
24	62.4%	65.9%	64.7%	20.7%	28.1%	25.7%
Average	34%	**43%**	38%	42%	**50%**	46%

6.1.3.2 GPU Scheduler for CUDA framework performance evaluation

In this section we evaluate the performance of the cascade detection algorithm on Storm topology with GPU scheduler based on CUDA framework. The same evaluation steps are

followed as explained earlier in this chapter. The results for the proposed token-based scheduling technique for different number of tokens compared to the CPU-only, GPU-only and two tokens OpenCL cases are illustrated in Figure 6.7.

The obtained results show that the single allocation scheduler yielded a better performance than the two tokens and four tokens case. It also shows that CUDA single allocation yields a better performance than OpenCL two tokens by nearly 5.5 %. This finding is consistent with what we discussed earlier in Chapter 3 about the performance differences between CUDA and OpenCL frameworks. In other words, CUDA framework is more optimized for Nvidia GPUs and it uses most of the available computational capacity more than OpenCL. Hence, two tokens for CUDA suffers more resource competition than OpenCL two tokens case.

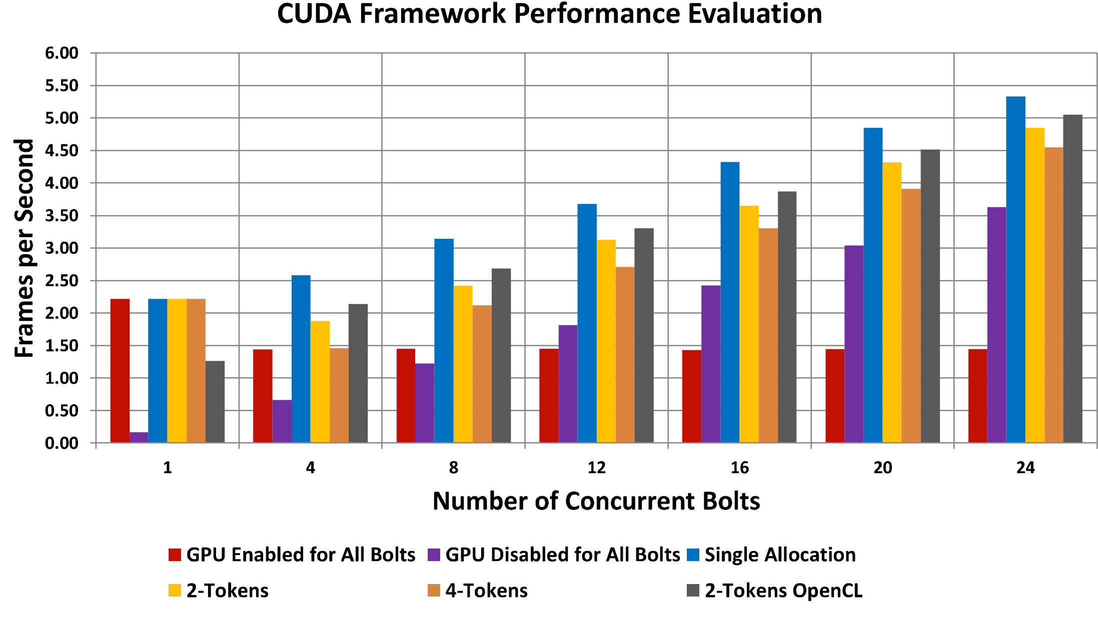

Figure 6.7: CUDA performance evaluation

This claim can be supported by comparing the average frame latency when only one bolt is running and using the GPU in both CUDA and OpenCL. One can clearly see that CUDA achieve better performance - nearly 42.3 % - than OpenCL which means that it utilizes the available resources on the GPU more than OpenCL for this specific implementation of cascade classifier in OpenCV library. As a result of that, the competition between bolts using GPU in CUDA framework is more than OpenCL. However, one should take into consideration that OpenCL is a more portable framework, therefore the scheduling techniques discussed in this thesis are widely applicable to other GPUs thanks to OpenCLs' specifications.

The performance metrics of the token based scheduler for different number of tokens for CUDA framework are shown in Table 6.4. The enhancement in the average frame latency is tabulated in Table 6.5.

As illustrated in the tables, Single allocation scheduler in CUDA can achieve up to 2.27x performance gain compared to CPU-only Storm topologies and up to 2.12x when compared to enabling GPU for all bolts. In the next section we will evaluate the token-based scheduler for CUDA compared to Nvidia's MPS software.

Table 6.4: CUDA framework performance metrics

Number of Concurrent Bolts	GPU Enabled for All Bolts		GPU Disabled for All Bolts		Single Allocation		2-Tokens		4-Tokens	
	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes
1	0.45	744.12	6.01	9933.27	0.45	744.21	0.45	744.30	0.45	744.39
4	2.78	1149.32	6.02	2486.01	1.55	640.20	2.13	879.30	2.74	1130.33
8	5.52	1139.84	6.54	1350.41	2.55	525.62	3.31	682.96	3.77	778.76
12	8.29	1140.66	6.61	910.20	3.26	448.77	3.84	529.01	4.43	610.28
16	11.19	1154.96	6.59	680.87	3.70	382.16	4.38	452.21	4.84	499.85
20	13.85	1143.85	6.58	543.52	4.12	340.66	4.63	382.67	5.11	422.06
24	16.61	1143.57	6.61	455.11	4.50	**309.72**	4.95	340.80	5.28	363.28

Table 6.5: Frame processing latency enhancement for CUDA framework

Number of Concurrent Bolts	Enhancement Versus GPU Enabled for All Bolts			Enhancement Versus GPU Disabled for All Bolts		
	Single Allocation	2-Tokens	4-Tokens	Single Allocation	2-Tokens	4-Tokens
1	0.0%	0.0%	0.0%	92.5%	92.5%	92.5%
4	44.3%	23.5%	1.7%	74.2%	64.6%	54.5%
8	53.9%	40.1%	31.7%	61.1%	49.4%	42.3%
12	60.7%	53.6%	46.5%	50.7%	41.9%	33.0%
16	66.9%	60.8%	56.7%	43.9%	33.6%	26.6%
20	70.2%	66.5%	63.1%	37.3%	29.6%	22.3%
24	72.9%	70.2%	68.2%	31.9%	25.1%	20.2%
Average	**53%**	45%	38%	**56%**	48%	42%

6.1.3.3 Token-based Scheduler Versus Nvidia MPS for CUDA Framework

In the previous two sections we showed how that our token-based scheduler enhances the performance of computer vision applications on big-data framework such as Storm. In this section will also show that the token-based scheduler outperforms Nvidia's MPS when the number of concurrent connections increases which is typical situation in big-data clusters.

We followed the same steps mentioned earlier in this chapter for evaluating MPS. However, MPS doesn't support more than 16 simultaneous connections and it doesn't support OpenCL framework. Hence, our charts compare single token CUDA scheduler against Nvidia MPS for up to 16 concurrent bolt. In Figure 6.8, we show that the average FPS in our token-based scheduler is better than MPS.

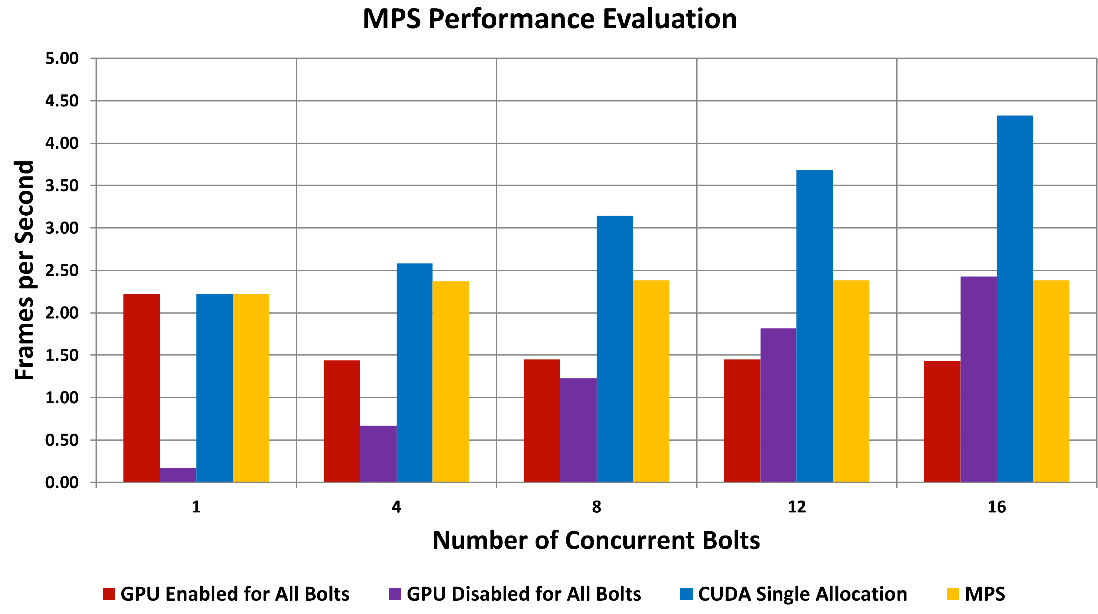

Figure 6.8: MPS performance evaluation against token-based scheduler for CUDA framework

The performance metrics of MPS for CUDA framework are shown in Table 6.6. The enhancement in the average frame latency for MPS is tabulated in Table 6.7. As shown in the results, the performance of our token-based scheduler outperforms MPS when the number of bolts equals to or exceeds four in the case of single-token CUDA scheduler. The latency enhancement for MPS vs GPU and CPU shows great enhancement however, compared to our token based scheduler, MPS shows negative enhancement which means that it worsened the latency instead of improving it when compared to the token based scheduler. Also, as we mentioned earlier MPS is only limited to 16 concurrent connections and in typical big-data clusters, the nodes have powerful processors with multiple cores which means that there might be more than 16 concurrent instances.

On the other hand, the purpose of MPS is to have a better utilization of the GPU when a single connection doesn't fully utilize it, so MPS could greatly enhance performance when the running tasks from a single connection are light weight tasks that doesn't consume a lot of the computational resources. In our case, it is well known that the detection algorithm is compute intensive especially with large resolution videos and classifiers.

Table 6.6: MPS performance metrics

Number of Concurrent Bolts	GPU Enabled for All Bolts		GPU Disabled for All Bolts		MPS	
	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes	Average Frame Latency	Video Processing Time - Minutes
1	0.45	744.12	6.01	9933.27	0.45	744.09
4	2.78	1149.32	6.02	2486.01	1.69	697.29
8	5.52	1139.84	6.54	1350.41	3.36	693.01
12	8.29	1140.66	6.61	910.20	5.04	694.00
16	11.19	1154.96	6.59	680.87	6.72	**694.09**

Table 6.7: MPS latency enhancement

Number of Concurrent Bolts	MPS Versus GPU Enabled for All Bolts	MPS Versus GPU Disabled for All Bolts	MPS Versus CUDA Single Allocation Scheduler
1	0.0%	92.5%	0.0%
4	39.3%	72.0%	-8.9%
8	39.2%	48.7%	-31.8%
12	39.2%	23.8%	-54.6%
16	39.9%	-1.9%	-81.6%
Average	32%	47%	**-35%**

6.2 Conclusion

In this chapter we evaluated the performance gain of enabling GPUs in big-data clusters for computer vision applications. We used Storm as the big-data framework for collecting the results and evaluating the scheduler and cascade detection algorithm. Our proposed scheduling algorithm can be used with any big-data framework as it is GPU and framework agnostic. We proposed several scheduling techniques, but we chose token based scheduling because of its pros and cons compared to the other schedulers proposed. The only drawback for token based scheduler is the static assignment of the GPU capacity in terms of tokens.

Chapter 7: Conclusion and Future Work

7.1 Conclusion

In this work we discussed several big-data frameworks such as Hadoop, Spark and Storm and the limitations of each one of them when it comes to processing visual data. We also illustrated how applying a reduce stage can eliminate redundant data generated due to the parallel processing of frames that could have the same data.

We also discussed the challenges of enabling GPUs utilization in big-data clusters for processing computer vision algorithms. These challenges are mainly due to racing and competition between the multiple instances of the big-data frameworks to access the GPU.

In order to eliminate this racing and competition, we implemented several GPU scheduling techniques that guarantee best performance compared to traditional implementations where the GPU is disabled for the whole cluster which leads to under utilization of a valuable resource as the GPU, or the GPU is enabled for the whole cluster which leads to racing and competition on the GPU thus degrading the overall performance.

We developed a token based scheduler which guarantees that no competition occurs on the GPU and at the same time prevents under-utilization of the GPU. The evaluation of our proposed token based scheduling scheme shows significant improvement up to 2.3x compared to the CPU-only and non-optimized GPU implementations for OpenCL and CUDA frameworks. The token-based scheduler is GPU independent and uses ready-to-use OpenCV library implementations hence it saves a lot of development effort and time for big-data engineers and developers who want to incorporate computer vision in big-data frameworks.

The token-based GPU scheduler introduced in this work can be used with multiple nodes by specifying the computer node and whether it has a GPU or not in the configuration file that includes the number of tokens for each GPU in the cluster. In addition to multiple nodes, the scheduler assigns the tokens with GPU id, therefore the bolts can work on different GPUs installed on the same node.

The work done in this thesis was tested several times on the available hardware at the time of testing. GPUs and big-data servers are expensive hardware and thus we didn't have

a variety of resources to test our token-based scheduler on different machines. However, the token-based scheduler is designed to be generic enough and hardware agnostic to be applicable on different sets of big-data clusters and with different GPUs.

The results we present in this thesis illustrates that for real-time video feed processing it is recommended to use big-data framework like Storm where the computer vision algorithm has no strict inter-frame dependency from processing perspective. Also, we recommend using token-based scheduler for large scale processing where the number of processes running on the processing node is more than 16 process and when the flexibility of choosing either OpenCL or CUDA is desired without specific GPU driver support. MPS is adequate for small number of threads or processes and for less computationally intensive computer vision algorithms based on CUDA framework.

7.2 Future Work

As a prospective extension for the work presented in this thesis, an automatic-adaptive mechanism for setting the best values for the number of tokens in our scheduler is to be developed and tested. The adaptive scheduler will start with initial value for the number of tokens based. This initial number is calculated based on an equation that takes the video resolution, processing complexity of the algorithm, the number of GPU cores and total GPU memory as inputs and outputs an integer number of tokens. The GPU service will record the average FPS for the CPU bolts and the average FPS for the GPU bolts at run-time, then using these FPS values, it can estimate the next FPS rate then decides to increment or decrement the number of tokens to achieve the best FPS. This way, our token based scheduler will be able to set the number of tokens without intervention from the developer during run-time.

References

[1] Cisco, "The Zettabyte Era: Trends and Analysis," tech. rep., Cisco Corporation, 2015.

[2] SeaGate, "Video Surveillance Trends Report," tech. rep., SeaGate Corporation, 2015.

[3] K. Cios, W. Pedrycz, and R. Swiniarski, *Data Mining Methods for Knowledge Discovery*. The Springer International Series in Engineering and Computer Science, Springer US, 2012.

[4] R. Grossman, C. Kamath, P. Kegelmeyer, V. Kumar, and R. Namburu, *Data Mining for Scientific and Engineering Applications*. Massive Computing, Springer US, 2013.

[5] R. B. Rao, G. Fung, B. Krishnapuram, J. Bi, M. Dundar, V. Raykar, S. Yu, S. Krishnan, X. Zhou, A. Krishnan, *et al.*, "Mining Medical Images," in *Proceedings of the third workshop on data mining case studies and practice prize, fifteenth annual SIGKDD international conference on knowledge discovery and data mining*, 2009.

[6] J. Gantz and D. Reinsel, "The Digital Universe In 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East," tech. rep., IDC, 2012.

[7] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The Next Frontier for Innovation, Competition, and Productivity," tech. rep., McKinsey Global Institute, 2011.

[8] Apache Software Foundation, "Apache Hadoop." Retrieved on May 20, 2016 from `http://hadoop.apache.org/`.

[9] Apache Software Foundation, "Spark Overview," 2016. Retrieved on May 28, 2016 from `http://spark.apache.org/docs/latest/index.html`.

[10] Apache Software Foundation, "Apache Storm," 2016. Retrieved on June 3, 2016 from `http://storm.apache.org/`.

[11] Apache Software Foundation, "Apache Nutch." Retrieved on May 20, 2016 from http://nutch.apache.org/.

[12] Apache Software Foundation, "Apache Lucene." Retrieved on May 20, 2016 from https://lucene.apache.org/core/.

[13] T. White, *Hadoop: The Definitive Guide*. Oreilly and Associate Series, O'Reilly, 2012.

[14] Apache Software Foundation, "Hadoop Wiki: Powered by Apache Hadoop," 2016. Retrieved on May 20, 2016 from http://wiki.apache.org/hadoop/PoweredBy.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *ACM SIGOPS operating systems review*, vol. 37, pp. 29–43, ACM, 2003.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[17] Apache Software Foundation, "HDFS Design," 2016. Retrieved on May 20, 2016 from http://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[19] V. K. Vavilapalli, A. C. Murthy, *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.

[20] Apache Software Foundation, "Apache Hadoop NextGen MapReduce (YARN)," 2016. Retrieved on May 21, 2016 from https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html.

[21] P. Gupta, P. Kumar, and G. Gopal, "Sentiment Analysis on Hadoop with Hadoop Streaming," *International Journal of Computer Applications*, vol. 121, no. 11, 2015.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *HotCloud*, vol. 10, pp. 10–10, 2010.

[23] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, 2015.

[24] S. Gopalani and R. Arora, "Comparing Apache Spark and Map Reduce with Performance Analysis using K-Means," *International Journal of Computer Applications*, vol. 113, no. 1, 2015.

[25] Apache Software Foundation, "Spark Cluster Overview," 2016. Retrieved on May 28, 2016 from `http://spark.apache.org/docs/latest/cluster-overview.html`.

[26] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pp. 10–10, USENIX Association, 2012.

[27] Apache Software Foundation, "Spark Streaming," 2016. Retrieved on May 28, 2016 from `http://spark.apache.org/docs/latest/streaming-programming-guide.html`.

[28] R. Ranjan, "Streaming Big Data Processing in Datacenter Clouds," *IEEE Cloud Computing*, vol. 1, no. 1, pp. 78–83, 2014.

[29] A. Jain and A. Nalya, *Learning Storm.* Packt Publishing, 2014.

[30] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@Twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 147–156, ACM, 2014.

[31] D. H. Im, C. H. Cho, and I. Jung, "Detecting a Large Number of Objects in Real-time Using Apache Storm," in *2014 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 836–838, Oct 2014.

[32] G. Ziegler, *GPU Data Structures for Graphics and Vision.* PhD thesis, Ziegler, 2011.

[33] S. Scott, "The Evolution of GPU Accelerated Computing," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*, (Salt Lake City, UT), pp. 1636 – 1672, IEEE, 2012.

[34] T. S. Crow, *Evolution of the Graphical Processing Unit.* PhD thesis, University of Nevada Reno, 2004.

[35] M. T. Do Dinh, "GPUs-Graphics Processing Units," *Vertiefungsseminar Architektur von Prozessoren*, 2008.

[36] N. GeForce, "8800 GPU Architecture Overview," *Technical Brief TB-02787-001 v0*, vol. 9, 2006.

[37] P. N. Glaskowsky, "Nvidia Fermi: The First Complete GPU Computing Architecture," 2009. White Paper. Retrieved on August 20, 2016 from http://www.nvidia.com/content/pdf/fermi_white_papers/p.glaskowsky_nvidia's_fermi-the_first_complete_gpu_architecture.pdf.

[38] "Nvidia's Next Generation CUDATM Compute Architecture: Kepler TM GK110," tech. rep., Nvidia Corporation, 2012. Retreived on August 21, 2016 from https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[39] Nvidia, "Multi-process Service," 2015. Technical Brief. Retrieved on August 21, 2016 from https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[40] Nvidia, "Nvidia's CUDA," 2016. Retrieved on July 3, 2016 from http://www.nvidia.com/object/cuda_home_new.html.

[41] W.-M. Hwu, C. Rodrigues, S. Ryoo, and J. Stratton, "Compute Unified Device Architecture Application Suitability," *Computing in Science & Engineering*, vol. 11, no. 3, pp. 16–26, 2009.

[42] A. Munshi, "The OpenCL Specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, pp. 1–314, IEEE, 2009.

[43] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[44] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1.0.* Newnes, 2012.

[45] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards A Performance-portable Solution for Multi-platform GPU Programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[46] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *arXiv preprint arXiv:1005.2581*, 2010.

[47] A. Merigot and A. Petrosino, "Parallel Processing for Image and Video Processing: Issues and Challenges," *Parallel Computing*, vol. 34, no. 12, pp. 694–699, 2008.

[48] M. Hassaballah, S. Omran, and Y. B. Mahdy, "A Review of SIMD Multimedia Extensions and Their Usage in Scientific and Engineering Applications," *The Computer Journal*, vol. 51, no. 6, pp. 630–649, 2008.

[49] R. B. Lee, "Multimedia Extensions for General-purpose Processors," in *Signal Processing Systems, 1997. SIPS 97-Design and Implementation., 1997 IEEE Workshop on*, pp. 9–23, IEEE, 1997.

[50] H. Fassold, "Computer Vision on the GPU–Tools, Algorithms and Frameworks," in *IEEE 20th Jubilee International Conference on Intelligent Engineering Systems 2016 (INES 2016)*, pp. 9–23, IEEE, 2016.

[51] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[52] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 260–269, ACM, 2008.

[53] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU Clusters," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, (Washington, DC, USA), pp. 1068–1079, IEEE Computer Society, 2011.

[54] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: Writing Parallel Program Portable between CPU and GPU," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 217–226, ACM, 2010.

[55] Y. Chen, Z. Qiao, H. Jiang, K.-C. Li, and W. W. Ro, *MGMR: Multi-GPU Based MapReduce*, pp. 433–442. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[56] Y. Chen, Z. Qiao, S. Davis, H. Jiang, and K.-C. Li, *Pipelined Multi-GPU MapReduce for Big-Data Processing*, pp. 231–246. Heidelberg: Springer International Publishing, 2013.

[57] Z. Wang, P. Lv, and C. Zheng, "CUDA on Hadoop: A Mixed Computing Framework for Massive Data Processing," in *Foundations and Practical Applications of Cognitive Systems and Information Processing*, pp. 253–260, Springer, 2014.

[58] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "HeteroDoop: A MapReduce Programming System for Accelerator Clusters," in *Proceedings of the 24th International*

Symposium on High-Performance Parallel and Distributed Computing, HPDC ’15, (New York, NY, USA), pp. 235–246, ACM, 2015.

[59] C. Basaran and K.-D. Kang, “Grex: An efficient MapReduce Framework for Graphics Processing Units,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 4, pp. 522–533, 2013.

[60] C. Sweeney, L. Liu, S. Arietta, and J. Lawrence, “HIPI: A Hadoop Image Processing Interface for Image-based MapReduce Tasks,” 2011.

[61] L. Chen, X. Huo, and G. Agrawal, “Accelerating MapReduce on a Coupled CPU-GPU Architecture,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 25, IEEE Computer Society Press, 2012.

[62] S. D. Sarade, N. B. Ghule, S. P. Disale, and S. S. R., “Large Scale Satellite Image Processing Using Hadoop Distributed System,” *International Journal of Advanced Research in Computer Engineering and Technology*, vol. 3, 3 2014.

[63] D. Markonis, R. Schaer, I. Eggel, H. Müller, and A. Depeursinge, “Using MapReduce for Large-scale Medical Image Analysis,” *CoRR*, vol. abs/1510.06937, 2015.

[64] T. R. Scogland, B. Rountree, W.-c. Feng, and B. R. de Supinski, “Heterogeneous Task Scheduling for Accelerated OpenMP,” in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 144–155, IEEE, 2012.

[65] A. Tarakji, D. Hebbeker, and L. Georgiev, “Efficient Resource Scheduling for Big Data Processing on Accelerator-based Heterogeneous Systems,” *Computer*, vol. 3, no. 2, p. 125, 2015.

[66] H. T. Vo, D. K. Osmari, J. Comba, P. Lindstrom, and C. T. Silva, “HyperFlow: A Heterogeneous Dataflow Architecture,” in *EGPGV*, pp. 1–10, 2012.

[67] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl, “Cudasa: Compute unified device and systems architecture.,” in *EGPGV*, pp. 49–56, 2008.

[68] Y. Yan and L. Huang, “Large-scale Image Processing Research Cloud,” in *Proceedings of 5th International Conference on Cloud Computing, GRIDs, and Virtualization (Cloud Computing 2014)*, pp. 88–93, 2014.

[69] P. Viola and M. Jones, “Rapid Object Detection Using a Boosted Cascade of Simple Features,” in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, pp. I–511, IEEE, 2001.

الملخص

لقد أصبحت تكنولوجيا البيانات الكبيرة في السنوات الأخيرة تستخدم بشكل متزايد لمعالجة كميات هائلة من البيانات في الوقت المناسب. إن النمو في إنتاج البيانات المرئية – أشرطة الفيديو والصور – في هذه الأيام يثير الحاجة لترقية تطبيقات رؤية الحاسب الآلى إلى أطر البيانات كبيرة من أجل زيادة معدلات المعالجة لهذه التطبيقات.

من ناحية أخرى فإن المطورين و المجتمع العلمي قد بدأوا بالفعل فى ترقية العديد من خوارزميات رؤية الحاسب الآلى إلى وحدة معالجة الرسومات التى تسرع بنجاح هذه الخوارزميات بفضل بنيتها المناسبة للتطبيقات التى تحتاج إلى تنفيذ نفس الأوامر بالتوازى على بيانات عديدة.

إن الجمع بين أطر البيانات الكبيرة مع وحدات معالجة الرسومات لتوسيع نطاق تطبيقات رؤية الحاسب الآلى أفقيا و عموديا, تعطي بنية واعدة لمعالجة كميات هائلة من البيانات المرئية وتلبى الحاجة الملحة إلى استخراج الأنماط الكامنة والمعلومات من هذه البيانات.

للأسف، إن عدد وحدات معالجة الرسومات المتاحة على أى حاسب آلى من ضمن كتلة من الحواسيب لمعالجة البيانات الكبيرة تكون محدودة و معظم الوقت تكون هناك وحدة معالجة رسومات واحدة فقط على أى من هذة الحواسيب. لذلك، تشغيل نسخ متعددة من نفس تطبيق رؤية الحاسب على أي إطار بيانات كبيرة يؤدي إلى التنافس بين هذه النسخ على موارد وحدة معالجة الرسومات.

في هذه الأطروحة، نحن نواجه التحدي المتمثل في الجمع بين وحدات معالجة الرسومات مع تكنولوجيا البيانات كبيرة من أجل تسريع تطبيقات رؤية الحاسب الآلى. من أجل تحقيق ذلك نقدم طريقة لجدولة إستخدام وحدة معالجة الرسومات بين نسخ متعددة من تطبيق رؤية الحاسب بكفاءة بحيث تقلص المنافسة بين هذة النسخ على موارد وحدة معالجة الرسومات. إن أسلوب الجدولة الذى نقدمه فى هذة الأطروحة ينتج أفضل أداء مقارنة بإستخدام إما وحدة معالجة الرسومات فقط أو وحدة المعالجة المركزية فقط.

يعتمد أسلوب الجدولة على رمز مميز ينتقل بين النسخ المختلفة للتطبيق ويسمح للنسخة التى تحوز عليه أن تستخدم وحدة معالجة الرسومات ما يضمن عدم وجود منافسة بينهم. التقييم العملى لأسلوب الجدولة أدى إلى أداء أفضل من حيث سرعة معالجة البيانات بقيمة تصل إلى 230% مقارنة مع استخدام وحدة معالجة مركزية تحتوى على 24 نوى و بقيمة تصل إلى 210% مقارنة مع استخدام وحدة المعالجة المركزية مع وحدة معالجة الرسومات. أخيرا مع مقارنة النظام المقدم فى هذة الأطروحة مع الطريقة المتتابعة لمعالجة تطبيقات رؤية الحاسب يكون فرق الأداء هو الضعف 32 مرة من حيث سرعة المعالجة.

مهندس: حازم عبد المجيد السيد عبد الحافظ

تاريخ الميلاد: 1990\9\26

الجنسية: مصرى

تاريخ التسجيل: 2014\3\1

تاريخ المنح: \ \2016

القسم: هندسة الإلكترونيات و الإتصالات الكهربية

الدرجة: ماجستير العلوم

المشرفون:

أ.د. حسام على حسن فهمى

أ.د. آمين محمد نصار

د. محمد محمد ريحان

الممتحنون:

أ.د حسام على حسن فهمى

أ.د. آمين محمد نصار

د. محمد محمد ريحان، المدير التقنى بشركة افيدبيم

أ.د السيد عيسى عبده حميد

أ.د خالد مصطفى السيد، كلية الحاسبات والمعلومات، جامعة القاهرة

عنوان الرسالة:

مجموعة مشغلات غير متجانسة لحسابات البيانات الكبيرة فى تطبيقات رؤية الحاسب

الكلمات الدالة:

البيانات الكبيرة، الحوسبة المتوازية ، وحدة معالجة الرسومات، رؤية الحاسب الآلى ، حوسبة غير متجانسة

ملخص الرسالة:

في هذا العمل قمنا بتطوير نظام قائم على الرموز الخاصة لجدولة إستخدام وحدة معالجة الرسومات في مجموعات البيانات الكبيرة خصيصا لتطبيقات رؤية الحاسب الآلى. نظام الجدولة يواجه تحدى المنافسة بين النسخ المختلفة من تطبيق رؤية الحاسب التى تحاول إستخدام وحدة معالجة الرسومات فى نفس الوقت. هذا النظام يسمح بترقية تطبيقات رؤية الحاسب إلى أطر البيانات الكبيرة مع قدرات حوسبة غير متجانسة حيث توجد وحدات المعالجة المركزية متعددة النوى جنبا إلى جنب مع وحدة المعالجة الرسومية.

مجموعة مشغلات غير متجانسة لحسابات البيانات الكبيرة فى تطبيقات رؤية الحاسب

إعداد
حازم عبد المجيد السيد عبد الحافظ

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستير العلوم
في
هندسة الإلكترونيات و الإتصالات الكهربية

يعتمد من لجنة الممتحنين:

الاستاذ الدكتور: حسام على حسن فهمى المشرف الرئيسى

الاستاذ الدكتور: آمين محمد نصار عضو

الدكتور: محمد محمد ريحان عضو،
المدير التقنى، شركة أفيدبيم

الاستاذ الدكتور: السيد عيسى عبده حميد الممتحن الداخلي

الاستاذ الدكتور: خالد مصطفى السيد الممتحن الخارجي،
كلية الحاسبات و المعلومات, جامعة القاهرة

كليـة الهندسـة - جامعـة القاهـرة
الجيـزة - جمهوريـة مصـر العربيـة
2016

مجموعة مشغلات غير متجانسة لحسابات البيانات الكبيرة فى تطبيقات رؤية الحاسب

إعداد
حازم عبد المجيد السيد عبد الحافظ

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستيرالعلوم
في
هندسة الإلكترونيات و الإتصالات الكهربية

تحت اشراف

د. حسام على حسن فهمى
أستاذ بقسم هندسة الإلكترونيات والإتصالات، كلية الهندسة،
جامعة القاهرة

د. آمين محمد نصار
أستاذ بقسم هندسة الإلكترونيات والإتصالات، كلية الهندسة،
جامعة القاهرة

د. محمد محمد ريحان
المدير التقنى بشركة أفيدبيم

كليـة الهندسـة - جامعـة القاهـرة
الجيـزة - جمهوريـة مصـر العربيـة

2016

مجموعة مشغلات غير متجانسة لحسابات البيانات الكبيرة فى تطبيقات رؤية الحاسب

إعداد

حازم عبد المجيد السيد عبد الحافظ

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستير العلوم
في
هندسة الإلكترونيات والإتصالات الكهربية

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

2016