Hindawi
Mathematical Problems in Engineering
Volume 2020, Article ID 4839876, 14 pages
https://doi.org/10.1155/2020/4839876

Research Article

Efficient Processing of Image Processing Applications on CPU/GPU

Najia Naz,[1] Abdul Haseeb Malik,[1] Abu Bakar Khurshid,[1] Furqan Aziz,[2] Bader Alouffi,[3] M. Irfan Uddin ,[4] and Ahmed AlGhamdi[5]

[1]*Department of Computer Science, University of Peshawar, Peshawar, Pakistan*
[2]*Department of Computer Science, Institute of Management Sciences, Peshawar, Pakistan*
[3]*Department of Computer Science, College of Computers and Information Technology, Taif University, Taif 21944, Saudi Arabia*
[4]*Institute of Computing, Kohat University of Science and Technology, Kohat, Pakistan*
[5]*Department of Computer Engineering, College of Computers and Information Technology, Taif University, Taif 21944, Saudi Arabia*

Correspondence should be addressed to M. Irfan Uddin; irfanuddin@kust.edu.pk

Received 16 July 2020; Accepted 27 September 2020; Published 10 October 2020

Academic Editor: Jia-Bao Liu

Heterogeneous systems have gained popularity due to the rapid growth in data and the need for processing this big data to extract useful information. In recent years, many healthcare applications have been developed which use machine learning algorithms to perform tasks such as image classification, object detection, image segmentation, and instance segmentation. The increasing amount of big visual data requires images to be processed efficiently. It is common that we use heterogeneous systems for such type of applications, as processing a huge number of images on a single PC may take months of computation. In heterogeneous systems, data are distributed on different nodes in the system. However, heterogeneous systems do not distribute images based on the computing capabilities of different types of processors in the node; therefore, a slow processor may take much longer to process an image compared to a faster processor. This imbalanced workload distribution observed in heterogeneous systems for image processing applications is the main cause of inefficient execution. In this paper, an efficient workload distribution mechanism for image processing applications is introduced. The proposed approach consists of two phases. In the first phase, image data are divided into an ideal split size and distributed amongst nodes, and in the second phase, image data are further distributed between CPU and GPU according to their computation speeds. Java bindings for OpenCL are used to configure both the CPU and GPU to execute the program. The results have demonstrated that the proposed workload distribution policy efficiently distributes the images in a heterogeneous system for image processing applications and achieves 50% improvements compared to the current state-of-the-art programming frameworks.

1. Introduction

GPUs (graphical processing units) are becoming popular to exploit data-level parallelism [1] in embarrassingly parallel applications [2] because of the SIMD (single instruction multiple data) [3, 4] architecture. However, task-level parallelism [5–7] is better exploited in general-purpose processors, because of MIMD (multiple instruction multiple data) [8] architecture. More recently, general-purpose processors are combined with GPUs and provide general-purpose computation accelerated with GPU (commonly referred as GPGPU) [9]. A heterogeneous cluster has many CPUs and GPUs and can exploit both task-level and data-level parallelism in applications.

The amount of data generated in the form of images and videos is enormous because of the fact that many surveillance cameras, smart phones, and many other devices that capture images/videos are installed/used everywhere. These devices are constantly recording scenes and can be processed for different types of computer vision, image processing,

machine learning, and data science tasks. Images inherently have data-level parallelism [1, 10] (i.e., individual images can be processed in embarrassingly parallel fashion), because all pixels can be processed independently and therefore GPUs are commonly used for image processing applications. A heterogeneous cluster (i.e., a cluster containing CPUs and GPUs) can exploit both task-level (across multiple images) and data-level parallelism in images. GPUs enable heterogeneous clusters to accelerate the processing intensive operations of images in big data and machine learning applications.

Data processing applications are commonly processed in a cloud environment using the MapReduce [11] parallel processing model for efficient execution. The scheduler of the MapReduce influences the performance in different applications when utilized in a heterogeneous cluster. The dynamic nature of the cluster and the computing workload affect the execution time of the application. Data locality is an essential part to reduce the total application execution time and hence improve the overall throughput.

In current technology, it is very challenging to provide a mechanism that can efficiently utilize the computing resources in a heterogeneous cluster [12]. Traditionally, applications that use both CPUs and GPUs are created using low level programming languages. Limited support is available to programmers to efficiently exploit parallelism in these clusters. A heterogeneous cluster normally has nodes of different computational capabilities [13]. For instance, a node may have 4 CPUs and 1 GPU, and another node may have 16 CPUs and 8 GPUs. Therefore, static distribution of workload amongst these nodes without taking into account their processing capabilities is not justified [14] as one node will be overloaded, and the other node will remain idle for most of the time [15]. Load balancing can be done dynamically, but it requires extra overheads at runtime [16].

In this paper, we present a new technique for the efficient distribution of images in a heterogeneous cluster. The goal is to maximize the utilization of the processing resources [17–19] (i.e., both CPUs and GPUs) and throughput. We provide a programming framework that ensures efficient workload distribution amongst the nodes by dividing the data into equal size splits and then distribute the split data between CPU and GPU cores based on their computational capabilities [20]. The aim is to achieve maximum resource utilization, gain high performance by dividing the data into equal size splits, allocate the data locally to the computing units, and minimize data migration to GPUs [21].

The rest of the paper is organized as follows. Related studies are given in Section 2. In Section 3, we provide a background to existing work that use CPU and GPU integration to efficiently solve different problems. We also provide a brief overview of the Hadoop programming framework, as this framework is used to test the ideas presented in this paper. We highlight limitations in existing state-of-the-art frameworks and explain the problems due to which efficient utilization of computing resources is not possible. The details of the proposed framework are given in Section 4, and demonstration of experiments is given in Section 5. We conclude the paper in Section 6.

2. Related Work

In [22], image processing for face detection and tracking is performed using CPU and GPU integration in Hadoop framework and has improved the performance by 25%. In another research [23], GPU-based Hadoop framework is used for evaluation of Canny Edge Detection algorithm using the default scheduler of Hadoop for workload distribution and has demonstrated two times performance improvement than HIPI-based image processing [24]. In [25, 26], face detection in video frames is performed by using CUDA-based Hadoop framework. It is observed that actual data processing takes 55% of the processor time, and the remaining 45% is wasted as idle or busy in performing other management activities.

Another framework named SEIP (system for efficient image processing) [27] ensured high performance for image processing application by applying in-node pipeline framework. However, during the processing, it is observed that the number of load/store to the GPU is equal to the number of images, which results into overhead and performance loss in case of processing a large number of small images, and also there is no policy of workload distribution between CPU and GPU. In [28], an integrated framework based on Hadoop and GPU has been developed for processing massive amount of satellite images. In order to achieve application performance, image data are split into parts and then each part is allocated to processing units, but there is no support for efficient workload distribution between CPU and GPU.

An energy efficient runtime mapping and thread partitioning approach has been developed for distribution of concurrent OpenCL application between CPU and GPU cores and has demonstrated a 32% increase in the system performance [29]. The feature extraction algorithm SIFT [30] has been developed in OpenCL that distributes workload on CPU and GPU. It is demonstrated that features were extracted with more than 30 FPS (frames per second) on full HD images, and an average speed up of 2.69 was achieved. Another OpenCL-based framework single node vertically scaled system is developed in [31] where multiple GPUs are combined together and treated as a single computing device. It automatically distributes OpenCL kernel written for a single GPU into multiple CUDA kernels at runtime that is executed on multiple (eight) GPUs. The experiment was performed by combining 8 GPUs in a single node environment, and the performance speedup of 7.1x was achieved as compared to the performance of single GPU. An average overhead of 0.48% is reported.

In [32], an algorithm is proposed that improves the efficiency of Hadoop clusters. The experiments demonstrated that if a process is defined that can handle different use-case scenarios, the overall cost of computing can be reduced and get benefits from distributed system for fast executions. A reinforcement learning-based MapReduce scheduler is proposed in [33] for heterogeneous environment. The system observes the state of task execution and suggests speculative execution of slow tasks to other free nodes in the cluster for faster execution. The proposed approach does not need any prior knowledge of the

environment and adapts itself to the heterogeneous environment. The experiments demonstrate that over a few runs, the system can better map the tasks to the available resources in a heterogeneous cluster and hence improve the overall performance of the system. The workload partition and task granularity for a given application based on machine learning techniques are given in [34]. The machine learning model can train a predictive model off-line, and then the trained model can predict the data partition and task granularity for any program at runtime. The experiments demonstrate a 1.6× average speedup using a single MIC. Other studies that involve the improvement in parallel computation are given in [35–38].

In all techniques presented in this section, the objective is to improve the performance of execution in a heterogeneous environment. The current state-of-the-art techniques demonstrate that imbalanced distribution of tasks without considering the underlying computational capabilities results into inefficient execution of the applications. The proposed technique in this paper considers the underlying computational capability of the processor and then assigns tasks. This results into performance improvement as demonstrated in Section 5. To the best of our knowledge, no previous studies have addressed the problem in the same perspective as undertaken by this research.

3. Background

In this section, existing frameworks are explained along with the limitations.

3.1. Programming Frameworks. Different programming frameworks such as Hadoop [39], FastFlow [40], OpenMP [41], pthreads [42], OpenCL [43], DirectCompute [44], OpenGL [45], MapReduce [46], and Spark [47] have been developed for the efficient data processing in a heterogeneous environment. Hadoop is becoming very popular because it can efficiently process structured [48], semi-structured [49], and unstructured [50] data. Different image processing applications such as face and motion detection [51], face tracking [52], extracting text from video frames in an online lecture video [53], video processing for surveillance [54], and content-based image retrieval (CBIR) [55] have demonstrated that Hadoop can be efficiently used for image-based applications.

CUDA (compute unified device architecture) [56] is the most commonly used programming language for GPUs developed by Nvidia. CUDA integrated with Hadoop enhances the application throughput by using the distributed computing capability of Hadoop and parallel processing capability of GPU [57]. Mars framework [58], which has been used for processing of web documents (searches and logs), was the first framework which combined GPU with Hadoop. Some other popular frameworks that integrate GPUs with Hadoop are MAPCG [59], StreamMR [60], and GPMR [61]. However, these frameworks are developed for some specific projects and do not improve the performance of image processing applications in Hadoop. Hadoop Image processing interface (HIPI) has been developed that can efficiently process a massive amount of small sized images but has no support of GPU [24].

A heterogeneous Hadooop cluster with nodes equipped with GPUs is shown in Figure 1. Hadoop is one of the famous and easy to use platforms which is a loosely coupled architecture and provides a distributed environment. Hadoop consists of the Hadoop distributed file system (HDFS) [62] and MapReduce [46] programming model. HDFS is an open-source implementation of the Google File System (GFS). It stores the data on different data nodes in a cluster. HDFS depends on the mechanism of the master-slave architecture. The access permission and data service to the slave nodes are provided by *master* node also known as *name* node, while the slaves, known as *data* nodes, are used as storage for the HDFS. Large files are handled efficiently by dividing them into chunks and then distributed amongst multiple data nodes. On each node, to process their local copies, a *map* processing job is located. The function of *name* node is only to keep record of the metadata and log information, while Hadoop API is used for the transfer of data to and from HDFS. MapReduce is an enhanced approach which provides an abstraction for data synchronization, load balancing, and dynamically allocation of tasks to different computing units in a reliable manner.

3.2. Limitations in Existing Systems. Some issues that lead to imbalanced workload distribution in a heterogeneous environment are discussed below.

3.2.1. Data Locality. Data locality means that the mapper and data are located on the same node. If data and mapper are on the same node, than it is easy for a mapper to efficiently map the data for computation, but if the data are on a different node than the mapper, then the mapper have to load data from different node over the network to be distributed. Suppose there are 50 mappers which try to copy data from other data nodes simultaneously. This situation leads to high network congestion which is not desirable because the overall performance of the application is affected. The situation where mapper and data are on the same node is shown in Figure 2(a), and the situation where mapper and data are on different nodes is shown in Figure 2(b). For efficient processing of applications, a programming framework should be able to ensure data locality. When the data stored in HDFS (Hadoop distributed file system) is distributed amongst the nodes, data locality needs to be handled very carefully. The data are divided into splits, and each split is provided to the data node in the cluster for processing. The MapReduce job is executed to map splits to individual mapper that will process the assigned split. That means that moving the computation closer to the data is better than moving the data closer to the computation. Hence, good data locality means good application performance.

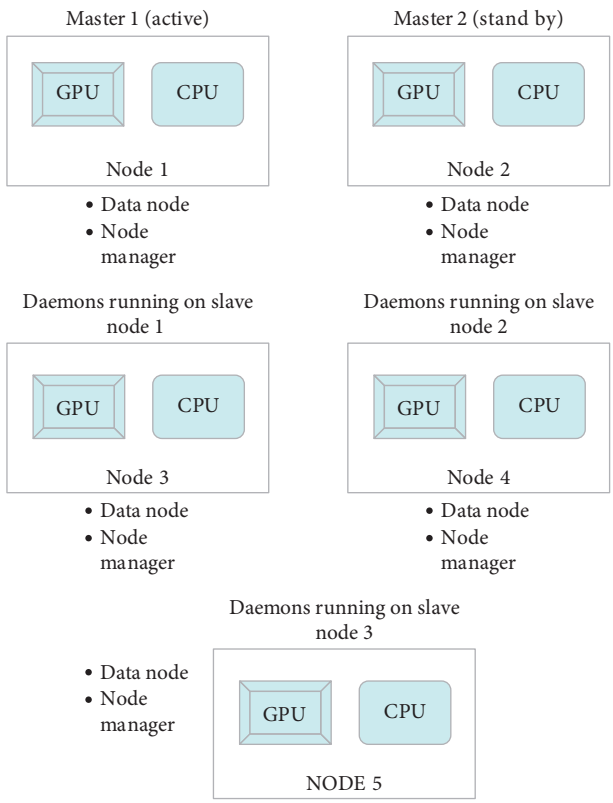

Figure 1: Heterogeneous Hadoop cluster with nodes equipped with GPUs.

3.2.2. Split Size. For efficient execution of programs in heterogeneous cluster, the programming framework should be able to distribute the data evenly into splits and then distribute amongst the available nodes. One of the main characteristics of MapReduce is to divide the whole data into chunks/input splits according to the block size of HDFS. As by default, Hadoop block size is 64 MB, and the issue of data locality arises when the input split size is larger than the block size. For better performance, input split size should be equal to or less than the block size of HDFS.

Block size and *split size* are not the same terms, as the block size is the physical chunk of data stored in disk, whereas input split size is the logical chunk of data with pointers for start and end locations in a block. When the split size is more or very small then the default block size, then uneven distribution of data happens, which leads to issues in data locality and memory wastage.

In Figure 3, the scenario of uneven split size is highlighted. Suppose the block size is 64 MB and each split size is 50 MB. The first split will easily fit in block 1, but the second split starts after the first split ending point and will not fully fit in the block 1, so the remaining part of the second split will be partially stored in block 1 and partially stored in block 2. When the mapper is assigned to block 1, it reads the first split, it will not read the second split data as it is not fully fitted in block 1 and cannot generate any final result of the second split data. According to [39, 63, 64], as quoted from the book "The logical records that *FileInputFormats* define do not usually fit neatly into HDFS blocks. For example, a *TextInputFormat*'s logical records are lines, which will cross HDFS boundaries more often than not. This has no bearing on the functioning of your program—lines are not missed or broken, for example—but it's worth knowing about, as it does mean that data-local maps (that is, maps that are running on the same host as their input data) will perform some remote reads. The slight overhead this causes is not normally significant." If a record/file span across the HDFS boundaries of two nodes, then one of the nodes will perform some remote reads to fetch the missing piece. And it will read the data and generate final results but with the overhead of communication between the two nodes. Hence, the communication overhead arises because the data are not evenly distributed according to the block size, and most of the time, mapper waits for other mappers to generate the result and then to synchronize with each other for final result. This problem can be solved by arranging the whole data into ideal input split size. By dividing the data into equal and suitable split size that is less than or equal to the default block size, the mapper of each block will read its data easily and will not wait for other mapper to send data of the split that is partially stored in different blocks.

3.2.3. Data Migration and Inefficient Resource Utilization. Data migration is the process of transferring data from one node or processor to another node or processor as shown in Figure 4. Data migration between systems is usually performed programmatically to achieve better performance, but in heterogeneous systems, where a node contains CPU and GPU, the GPU being the faster computing processor will complete its task quickly and will fetch the data from CPU, a slow computing processor. Due to this data migration, the scheduler will always be busy in managing the tasks scheduling, the GPU will be idle, and hence performance in applications is affected.

Above are some of the problems that need to be tackled while using heterogeneous systems, so that application performance can be increased. In this paper, we will integrate CUDA with Hadoop to increase the performance in processing images in heterogeneous clusters. We will integrate the Hadoop platform that is used for distributed processing on clusters with libraries that allow code to be executed on GPUs.

4. Efficient Workload Distribution Based on Processor Speed

The issues of data locality, input split size, data migration, and inefficient resource utilization discussed in Section 3.2 lead to imbalanced workload distribution in heterogeneous environment, which results into inefficient execution of applications. The proposed framework will distribute the data in a balanced form amongst the nodes according to their computing capabilities, as shown in Figure 5. The distribution of data in the framework consists of two phases. In Phase I, the data are distributed amongst the nodes and in Phase II the workload is equally distributed between CPU

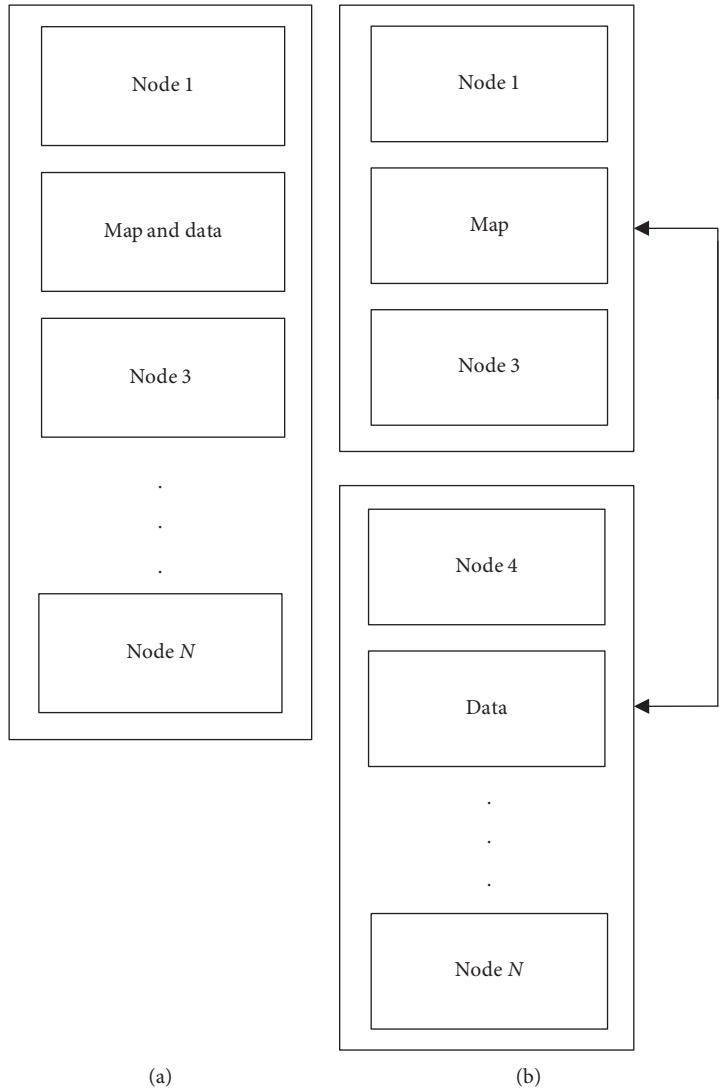

Figure 2: (a) Data and Mapper on the same node; (b) Data and Mapper on different nodes.

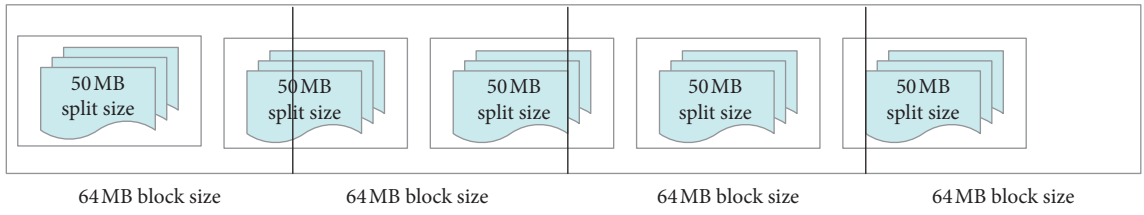

Figure 3: Mismatch between split size and block size.

and GPU based on their computing capability. Both phases are explained below in detail.

4.1. Phase I: Data Distribution amongst Nodes. In Hadoop, workload is organized in splits which are then distributed amongst nodes in the cluster to be processed. However, this workload is not evenly distributed and hence processors with lower processing capabilities are overwhelmed. In the proposed framework, the input is in the form of images and is distributed evenly amongst cluster nodes in order to utilize computation and memory resources efficiently. We have developed a novel distribution policy for the even distribution of same size images in splits, where a split contains one or more images. We are focusing on same sized images only, in order to avoid communication overhead when images of different sizes are loaded. With images of different sizes, the ratio calculations explained in Section 4.2 are useless. This idea is mainly inspired from *arrays*, where continuous blocks of the same size are gathered together,

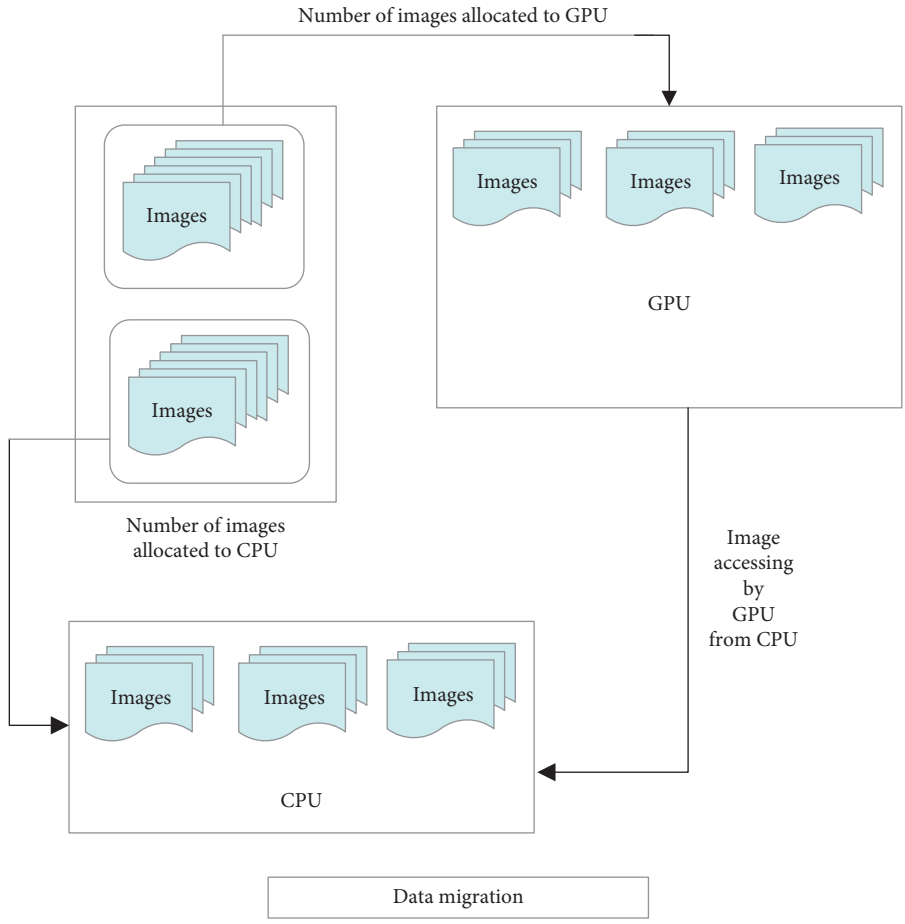

Figure 4: Data migration between CPU and GPU.

providing simplicity and performance. The distribution policy does not allow a single image to be distributed to multiple splits, because of data locality issue. The ideal split size is set according to the size of the images, so that an image does not exceed the boundary of that split. Multiple images evenly grouped together in a split and ready to be distributed amongst nodes is shown in Figure 6, and the ideal split size is calculated as per the default block size (i.e., 64 MB in Hadoop) as shown in Figure 7.

To avoid the problem of uneven splits, i.e., when an image is distributed in multiple splits, the split size is set very carefully based on the image size. We first measure the size of one image and then select an input split size where multiple images of that size will be placed. Let I be the ideal input split size which need to be computed, d be the default split size in Hadoop, s be the size of one of the input images, n_o be the number of images that can be fully accommodated by the default input split, T_i be the total number of images in a dataset, and S_n be the total number of splits in which the data are divided equally. n_o is calculated by dividing d on s, ignoring the fractional part by taking the floor. I is computed by multiplying the image size s with n_o, and S_n is computed by dividing T_i on n_o, as shown in equation (1). This equation calculates an ideal input split size, and no image can occur across two input splits. For instance, we have an input image of size 4.2 MB (s), and 64 MB is the default input size (d), and then the ideal input split size $I = 15 \times 4.2 = 63$ MB ($n_0 = \lfloor 64/4.2 \rfloor = 15$). To calculate the number of splits, the data should be arranged in $S_n = 90/15 = 6$, where 90 is the total number of input images. So, we will have a total of 6 input splits each of size 63 MB to store 90 input images.

$$n_0 = \left(\frac{d}{s}\right),$$
$$I = n_0 \times s, \tag{1}$$
$$S_n = \frac{T_i}{n_0}.$$

4.2. Phase II: Distribution of Workload on CPU and GPU within a Node. In a heterogeneous cluster, every node is equipped with a GPU, which is much faster than the CPU. Therefore, for efficient execution of applications, it is important that tasks are distributed based on the computing capabilities of processors. The proposed workload distribution scheme for heterogeneous Hadoop cluster is shown in Figure 8. A split is a container of a group of images of the same sizes. For every split, the map function is invoked, which takes the input $< key,\ value >$ pair, where the key contains log file of images in a split and the value contains

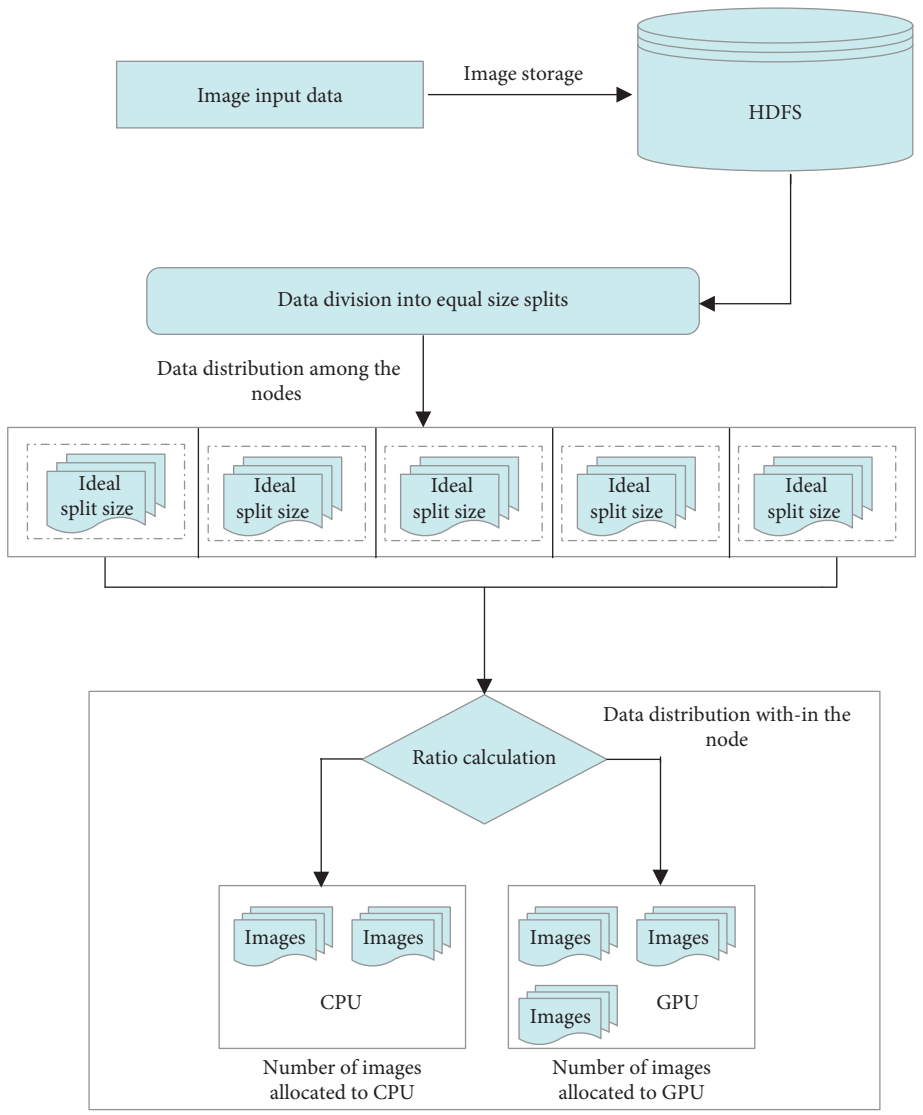

Figure 5: Proposed framework for workload distribution.

contents of images (bytes). The map function processes the split and reads each image in the split. In the proposed algorithm, the map function takes the split and checks the ratio for all the available images in that split so that images can be assigned to CPU and GPU according to their computing capability. Before assigning images to the CPU and GPU, a sample image is executed on both the CPU and GPU to find out the execution time of both the processors on a specified algorithm. From this execution time, the processing power of each processor is identified, and a ratio is calculated, demonstrating the number of images assigned to CPU and GPU.

In order to compute the computing capability of processors in a node, we initially assign a raw image to both CPU and GPU to find the execution time of both processors. Let c be the execution time in CPU and g be the execution time in GPU to execute an image processing algorithm on a raw input image. We take the ratio of $\max(g, c)$ and $\min(g, c)$. The device with larger execution time (i.e., slower processor) is assigned a value 1, and the device with smaller execution time (i.e. faster processor) is assigned an integer value of the execution time of slow processor divided by execution time of fast processor. When the processor fetches images, we assign nP_x number of images to the slower processor and nP_y images to the faster processor, as shown in the following equation:

$$\begin{aligned} x &= 1, \\ y &= \left(\frac{\max(g,c)}{\min(g,c)}\right), \\ nP_x &= \left(\frac{x}{x+y}\right) \times n_0, \\ nP_y &= \left(\frac{y}{x+y}\right) \times n_0. \end{aligned} \tag{2}$$

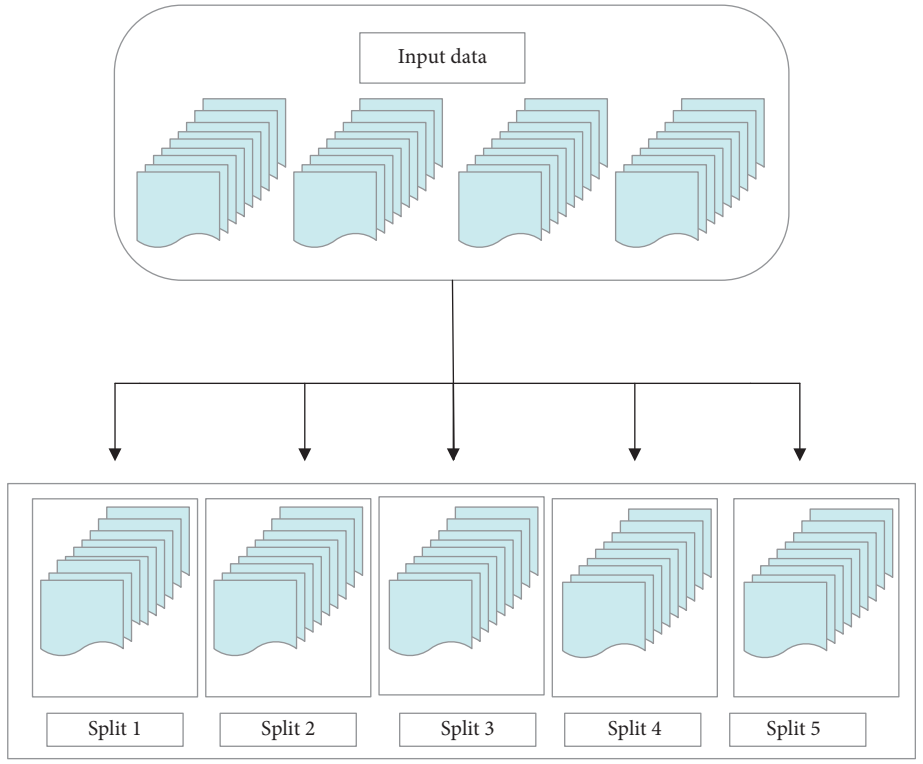

Figure 6: Grouping of input data to be partitioned in splits.

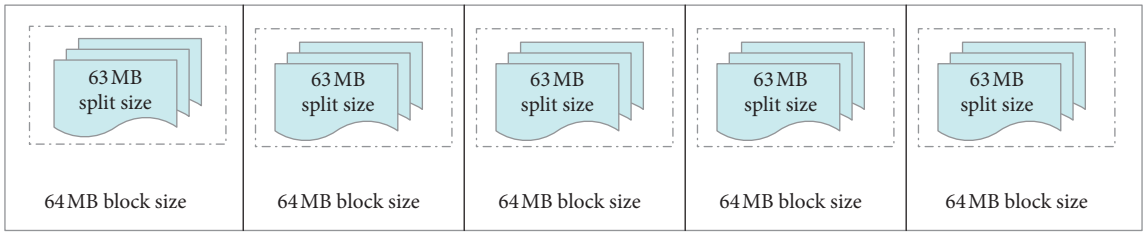

Figure 7: Image data partitioning into ideal split size.

Suppose n_0 is 15 (as explained above), GPU execution time (g) for a raw image is 25 ms, and CPU execution (c) is 160 ms. In this example, $x = 1$ and $y = 160/25 = 6$, which means that CPU and GPU are assigned images in the ratio of 1 : 6. Therefore, $nP_x = (1/(1+6)) \times 15 = 2$ and $nP_y = (6/(6+1)) \times 15 = 13$, i.e., in a single fetch, 2 images are assigned to CPU and 13 images are assigned to GPU. This novel distribution of workload ensures that images are distributed based on the computing capability of the processor, which can speedup the execution of images in heterogeneous nodes. The flow chart of the proposed framework is shown in Figure 9.

5. Evaluation

In Section 4, the proposed framework is introduced, to efficiently handle the imbalanced workload distribution in a heterogeneous Hadoop cluster for image processing applications, and the implementation of such policy is evaluated by using the commodity computer systems accelerated with GPU. As the heterogeneous systems increase the performance of applications by processing a massive amount of data in parallel, in the future, these heterogeneous systems will be commonly used and provide efficient execution of programs, by adopting policies for efficient workload distribution. In this section, the evaluation of the proposed programming framework is discussed. The detail of the dataset used for the evaluation is given in Table 1. The test environment includes a master node having a corei5 processor with speed 2.5 GHz, 8 GB RAM, and 4 processing cores. Two worker nodes are used each having a CPU, corei3, 1.8 GHz, 4 GB RAM, and 4 cores and a GPU NVIDIA 802 M, 64 cores, and 2 GB RAM.

5.1. Processing Images of Different Sizes. The Figure 10(a) shows the average execution time calculated in milliseconds for edge detection algorithm on four images of different sizes on a CPU and GPU, respectively. This experiment demonstrates that, on CPU, execution time of the application increases when the size of the image is increased. However, on GPU, the increase in execution time when processing an

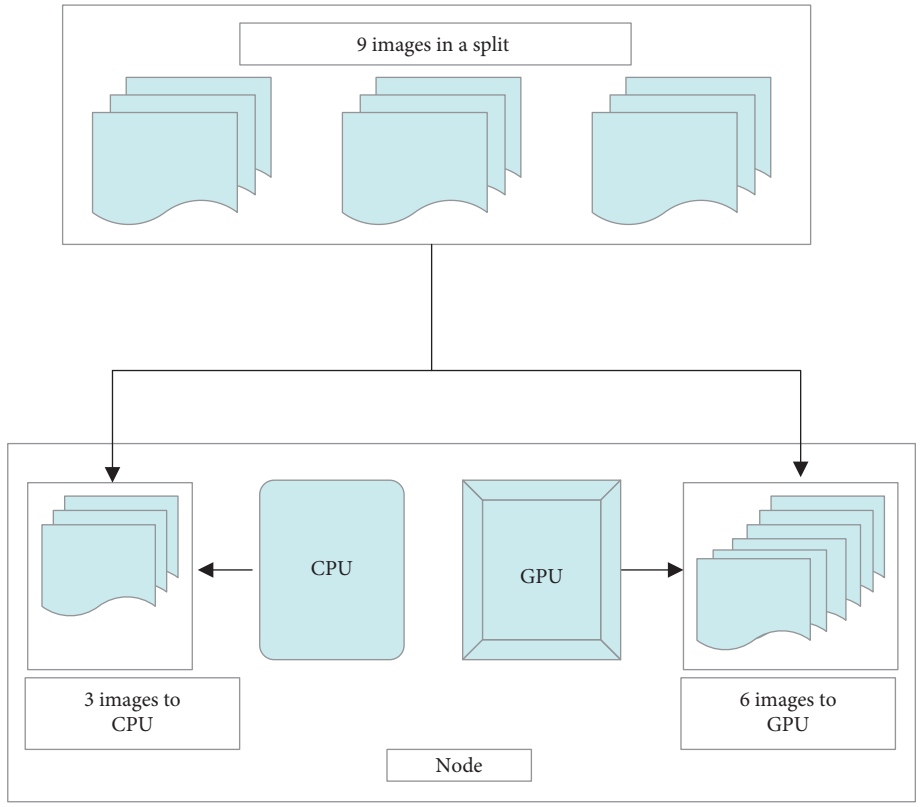

Figure 8: Distribution of workload within a node between CPU and GPU according to their computing capabilities.

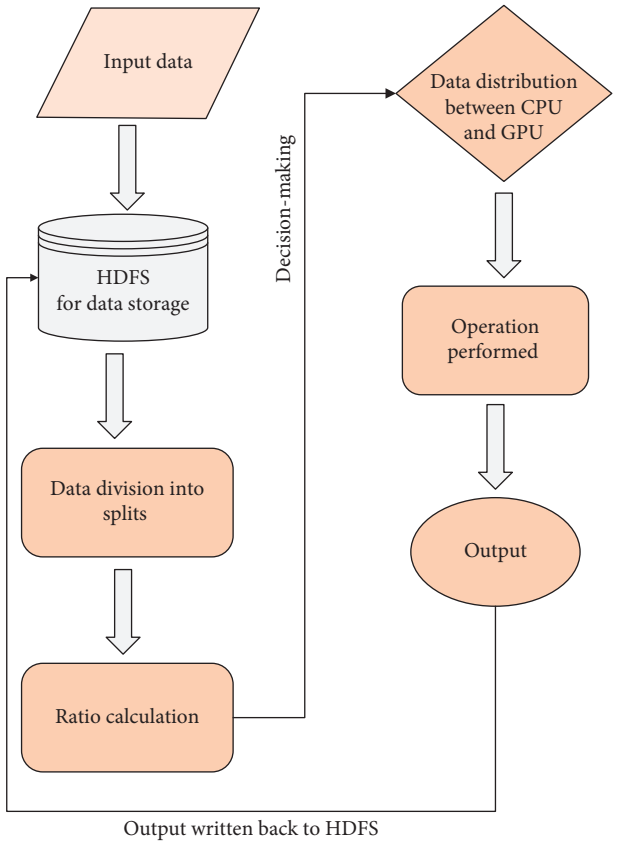

Figure 9: Basic flowchart of the proposed framework.

application on a very large image is not significant. This analysis demonstrates that GPU is a better choice for processing larger images.

5.2. Processing Different Number of Images. In Figure 10(b), the performance of the application in processing different number of images on a GPU is shown. *Y*-axis shows execution time in millisecond and *x*-axis shows the number of images of different sizes. Four images with different resolution sizes are grouped together as 1 image, 2 images, 3 images, and 4 images. For instance, when the application processes a single image of size 1024×768, the execution time on GPU is 42 milliseconds. But, in processing four images of the same size, the GPU takes 51 milliseconds. Similarly, a single image of size 2560×1440 takes 69 ms, but processing four images of the same size takes 87 ms. This increase in execution time is mainly because of the communication overhead, as each image is migrated from CPU memory to GPU memory and then the result is written back to the CPU memory.

5.3. Performance Comparison. We compare the performance of our approach, i.e., modified split sizes and optimized workload distribution based on the computation capabilities of processors in a node of heterogeneous system with existing state-of-the-art techniques such as HIPI, HIPI executed on GPU, and Hadoop executed on GPU, as shown in Figure 10(c). The proposed framework is processing

Table 1: Information of images in the dataset.

S. no.	Image resolution	Total size (MB)	Total images	Individual image size (MB)
1	1024 × 768	216	90	2.4
2	1600 × 900	387	90	4.3
3	1920 × 1080	558	90	6.2
4	2560 × 1440	999	90	11.1

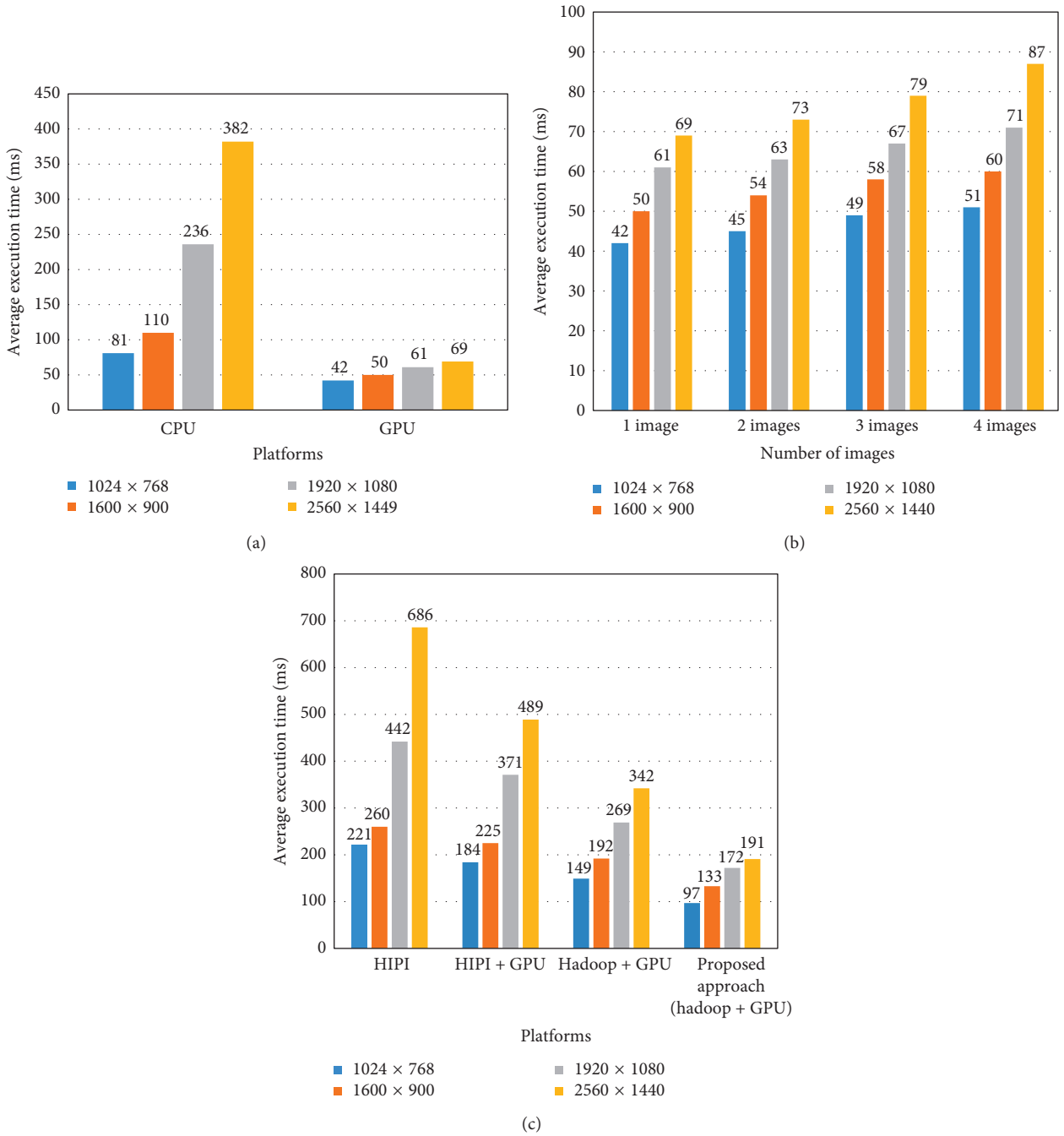

Figure 10: The execution time of CPU and GPU, different number of images, and average performance comparison with existing platforms. (a) Average execution time in CPU and GPU with different number of images for edge detection algorithm. (b) Average execution time of application in processing different number of images on a GPU. (c) Average execution time (milliseconds) for proposed framework (Hadoop + GPU) vs existing platforms.

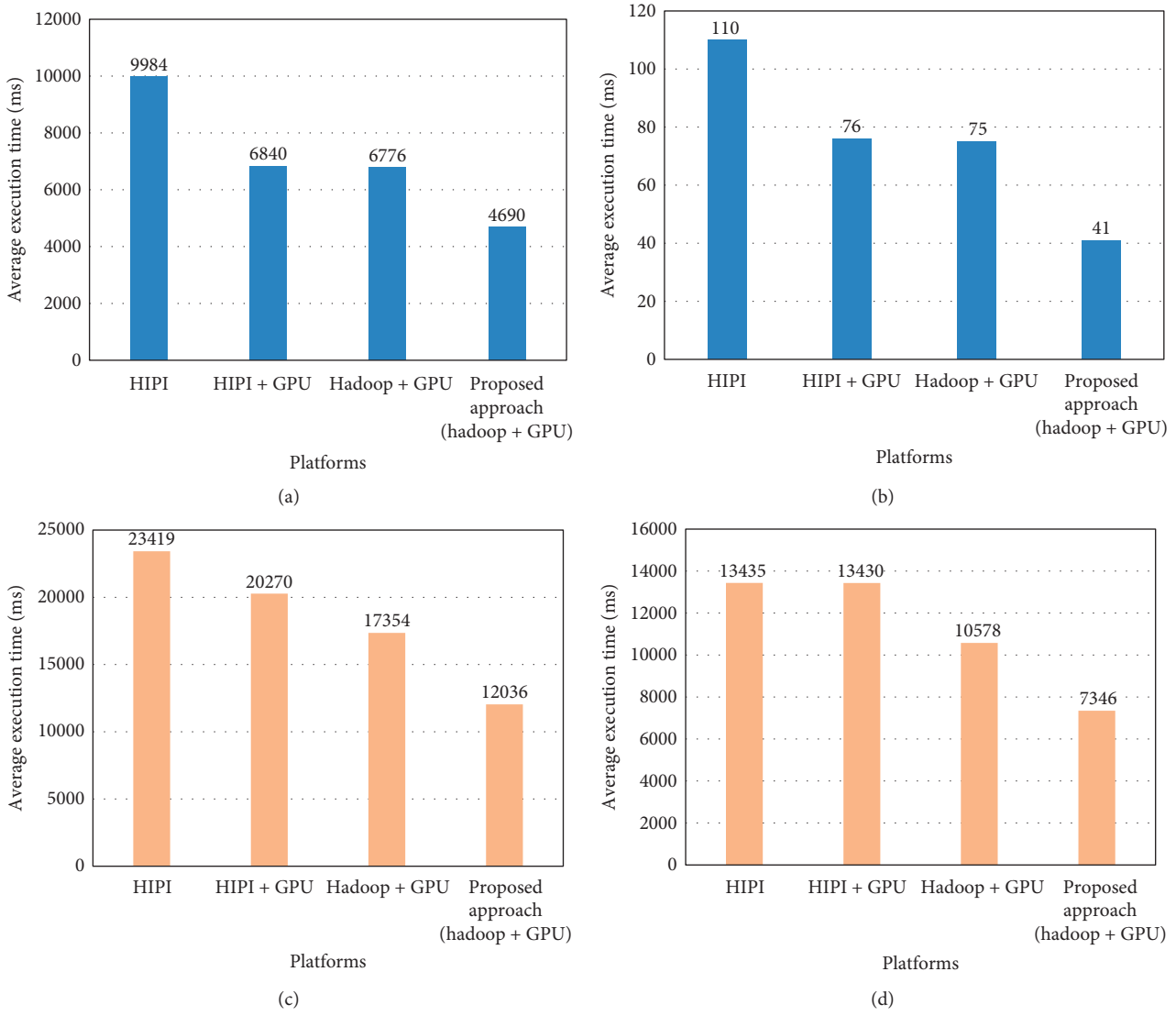

Figure 11: (a) Execution time of a split on a node. (b) Execution time calculated for each split. (c) Execution time of image dataset processed by edge detection application. (d) Execution time of a single image from split to processor.

images faster than the other state-of-the-art frameworks. For instance, processing 90 images of resolutions 2560×1440 each of size 11.1 MB takes 686 ms in HIPI, but only 191 ms in the proposed framework, i.e., a speedup of 3.5, is achieved. This speedup is possible mainly because of the efficient workload distribution and the efficient arrangement of images in input splits which results into the efficient utilization of resources.

5.4. Effect of Assigning Different Loads to CPU and GPU. In the proposed approach, we have developed a novel technique to compute the ratio of images to be assigned to CPU and GPU in a heterogeneous node. The performance of this calculation is shown in Figure 11(a) along with comparison with other state-of-the-art techniques. It is observed that application processing the dataset of different images of different sizes executes efficiently on the proposed framework. For instance, the speedup in the proposed approach is 2.12x compared to HIPI. Hence, it is proved that, by efficient load balancing techniques in heterogeneous systems, we can increase the performance by two times.

5.5. Execution Time of Each Split. Input data are divided in different input splits and distributed amongst nodes, and from each split, images are accessed and processed. The average execution time taken by images of sizes (1024×768, 1600×900, 1920×1080, and 2560×1440) in an input split is shown in Figure 11(b) for all the four platforms while processing edge detection application. It is demonstrated that the proposed framework processes the images more than two and half times faster than HIPI.

5.6. Execution Time in Processing the Whole Dataset. In Figure 11(c), an average of total execution time for the whole dataset shown in Table 1 is calculated for image processing

application performing edge detection operation. The total execution time includes partitioning of image data into splits, distribution amongst the nodes, and on each node, further distribution between CPU and GPU, and after processing, the result is written back to the HDFS. From the figure, it is clearly shown that, by adopting the proposed approach (Hadoop + GPU) where data is divided into ideal split size and then distributed to the processors according to their computing capabilities, the total execution time calculated in milliseconds is two times less than the other existing platforms.

5.7. Minimization of the Communication Overhead by Ideal Split Size Selection. The communication overhead of images during processing is shown in Figure 11(d). The access time of an image in this figure includes the time from split to the assigned processor. The execution time taken by a single image in recorded. From the experiment, it is observed that, by arranging the data local to the computing processor, the images can be easily accessed and processed. As HIPI has an overhead of data compression and decompression; therefore, the results of both HIPI and HIPI + GPU are relatively same and high. By using Hadoop + GPU, the image access time is less compared to the HIPI and HIPI + GPU, but by overcoming the data locality issue in the proposed framework (Hadoop + GPU), the image access time is almost two times reduced.

6. Conclusion and Future Work

Distributed systems provide parallel frameworks where massive amount of data can be efficiently processed. Hadoop is one of those frameworks that provides large amount of data storage and effective computational capability to handle massive amount of data in a parallel and distributed manner by using the cluster of commodity computer systems. These clusters also contain GPUs, as they are specialized to handle SIMD efficiently. For image processing applications or applications involving the process of big data in different domains such as healthcare, heterogeneous systems are commonly used and have shown improvement over single processor and distributed systems. However, imbalanced workload distribution between the processor causes data locality and inefficient workload distribution between slow and fast processors can affect the performance of applications. To deal with this imbalanced workload distribution in a heterogeneous cluster, this paper has proposed a novel technique of dividing data into ideal input splits so that an image is included in one split and does not exceed the boundary of that split. This paper also introduces a technique of distributing data as per the computing power of the processors in the node. This distribution maximizes the utilization of available resources and tackles the issue of data migration between the fast and slow computing processors. The results have demonstrated that the proposed framework achieves almost two times improvement compared to the current state-of-the-art programming frameworks. The proposed framework provides an efficient mechanism to compute an ideal split size that is suitable for fixed size images but does not provide support for partitioning variable size images into splits. In the future, we will investigate techniques that can compute ideal split size for variable sized images. In the future, we will investigate techniques that can compute ideal split size for variable sized images which will enable us to process images from sources. The proposed model can be extended for big data applications which have inherent data-level parallelism in the form of arrays, matrices, images, or tables.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] L. Baumstark and L. Wills, "Exposing data-level parallelism in sequential image processing algorithms," in *Proceedings of the Ninth Working Conference on Reverse Engineering*, pp. 245–254, New York, NY, USA, 2002.

[2] B. Vinter, "Embarrassingly parallel applications on a java cluster," in *Proceedings of the 8th International Conference on High-Performance Computing and Networking, HPCN Europe 2000*, Springer-Verlag, London, UK, pp. 614–617, 2000.

[3] B. Furht, *SIMD (Single Instruction Multiple Data Processing)*, Springer US, Boston, MA, USA, 2008.

[4] I. Uddin, "Multiple levels of abstractions in the simulation of microthreaded many-core architectures," *Open Journal of Modelling and Simulation*, vol. 3, 2015.

[5] M. Girkar and C. D. Polychronopoulos, "Extracting task-level parallelism," *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 600–634, 1995.

[6] I. Uddin, "High-level simulation of concurrency operations in microthreaded many-core architectures," *GSTF Journal on Computing (JoC)*, vol. 4, p. 21, 2015.

[7] I. Uddin, "One-IPC high-level simulation of microthreaded many-core architectures," *International Journal of High Performance Computing Applications*, vol. 4, 2015.

[8] R. Riesen and A. B. Maccabe, *MIMD (Multiple Instruction, Multiple Data) Machines*, Springer US, Boston, MA, USA, 2011.

[9] L. Hu, X. Che, and S.-Q. Zheng, "A closer look at GPGPU," *ACM Computing Surveys*, vol. 48, 2016.

[10] M. Fatima, "Reliable and energy efficient mac mechanism for patient monitoring in hospitals," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 10, 2018.

[11] J. V. Bibal Benifa, "Performance improvement of Mapreduce for heterogeneous clusters based on efficient locality and replica aware scheduling (ELRAS) strategy," *Wireless Personal Communications*, vol. 95, no. 3, pp. 2709–2733, 2017.

[12] N. Elgendy and A. Elragal, "Big data analytics: a literature review paper," in *Advances in Data Mining, Applications and Theoretical Aspects*, P. Perner, Ed., pp. 214–227, Springer International Publishing, Berlin, Germany, 2014.

[13] A. Khan, M. A. Gul, M. I. Uddin et al., "Summarizing online movie reviews: a machine learning approach to big data analytics," *Scientific Programming*, vol. 2020, pp. 1–14, 2020.

[14] F. Aziz, H. Gul, I. Muhammad, and I. Uddin, "Link prediction using node information on local paths," *Physica A: Statistical Mechanics and its Applications*, vol. 2020, Article ID 124980, 2020.

[15] J. Zhu, H. Jiang, J. Li, E. Hardesty, K.-C. Li, and Z. Li, "Embedding GPU computations in hadoop," *International Journal of Networked and Distributed Computing*, vol. 2, pp. 211–220, 2017.

[16] W. Chen, S. Xu, H. Jiang et al., "GPU computations on hadoop clusters for massive data processing," in *Proceedings of the 3rd International Conference on Intelligent Technologies and Engineering Systems (ICITES2014)*, J. Juang, Ed., Springer International Publishing, Berlin, Germany, pp. 515–521, 2016.

[17] S. Amin, M. I. Uddin, S. Hassan et al., "Recurrent neural networks with TF-IDF embedding technique for detection and classification in tweets of dengue disease," *IEEE Access*, vol. 8, pp. 131522–131533, 2020.

[18] M. I. Uddin, S. A. A. Shah, and M. A. Al-Khasawneh, "A novel deep convolutional neural network model to monitor people following guidelines to avoid COVID-19," *Journal of Sensors*, vol. 2020, pp. 1–15, 2020.

[19] Z. Ullah, A. Zeb, I. Ullah et al., "Certificateless proxy reencryption scheme (CPRES) based on hyperelliptic curve for access control in content-centric network (CCN)," *Mobile Information Systems*, vol. 2020, pp. 1–13, 2020.

[20] A. Khan, I. Ibrahim, M. I. Uddin et al., "Machine learning approach for answer detection in discussion forums: an application of big data analytics," *Scientific Programming*, vol. 2020, 2020.

[21] F. Aziz, T. Ahmad, A. H. Malik, M. I. Uddin, S. Ahmad, and M. Sharaf, "Reversible data hiding techniques with high message embedding capacity in images," *PLoS One*, vol. 15, no. 1–24, 2020.

[22] B. Sharma, R. Thota, N. Vydyanathan, and A. Kale, "Towards a robust, real-time face processing system using CUDA-enabled GPUS," in *Proceedings of the 2009 International Conference on High Performance Computing (HiPC)*, pp. 368–377, New York, NY, USA, 2009.

[23] H. Patel and K. Panchal, "Incorporating CUDS in hadoop image processing interface for distributed image processing," in *Proceedings of the International Journal of Advance Research and Innovative Ideas*, pp. 93–98, New York, NY, USA, 2016.

[24] C. Sweeney, L. Liu, S. Arietta, and J. Lawrence, "Hipi: a hadoop image processing interface for image-based mapreduce tasks," 2011.

[25] R. Malakar and N. Vydyanathan, "A cuda-enabled hadoop cluster for fast distributed image processing," in *Proceedings of the 2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pp. 1–5, New York, NY, USA, 2013.

[26] M. I. Uddin, N. Zada, F. Aziz et al., "Prediction of future terrorist activities using deep neural networks," *Complexity*, vol. 2020, pp. 1–16, 2020.

[27] T. Liu, Y. Liu, Q. Li et al., "Seip: system for efficient image processing on distributed platform," *Journal of Computer Science and Technology*, vol. 30, pp. 1215–1232, 2015.

[28] M. S. Gowda and V. G. Hulyal, "Parallel image processing from cloud using cuda and hadoop architecture: a novel approach," 2015.

[29] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent OPENCL applications on CPU-GPU MPSOCS," *ACM Transactions on Embedded Computing Systems*, vol. 16, 2017.

[30] W. Burger, M. J. Burge, and M. J. Burge, *Scale-Invariant Feature Transform (SIFT)*, Springer, London, UK, 2016.

[31] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in OPENCL for multiple GPUS," *SIGPLAN Not*, vol. 46, pp. 277–288, 2011.

[32] P. Dadheech, D. Goyal, S. Srivastava, and A. Kumar, "Performance improvement of heterogeneous hadoop clusters using query optimization," *SSRN Electronic Journal*, vol. 46, 2018.

[33] N. S. Naik, A. Negi, and V. Sastry, "Performance improvement of MapReduce framework in heterogeneous context using reinforcement learning," *Procedia Computer Science*, vol. 50, pp. 169–175, 2015.

[34] C. Lai, Y. Chen, X. Shi, M. Huang, and G. Chen, "Performance improvement on heterogeneous platforms: a machine learning based approach," in *Proceedings of the 2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, IEEE, Berlin, Germany, 2018.

[35] P. Czarnul, J. Proficz, and K. Drypczewski, "Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems," *Scientific Programming*, vol. 2020, pp. 1–19, 2020.

[36] A. Rubio-Largo, J. C. Preciado, and L. Iribarne, "Data-driven computational intelligence for scientific programming," *Scientific Programming*, vol. 2019, pp. 1–4, 2019.

[37] A. Kanan, F. Gebali, A. Ibrahim, and K. F. Li, "Low-complexity scalable architectures for parallel computation of similarity measures," *Scientific Programming*, vol. 2019, 2019.

[38] W. Ma, W. Yuan, and X. Hu, "Implementation and optimization of a CFD solver using overlapped meshes on multiple MIC coprocessors," *Scientific Programming*, vol. 2019, pp. 1–12, 2019.

[39] T. White, *Hadoop: The Definitive Guide*, O'Reilly Media, Inc., New York, NY, USA, 1st edition, 2009.

[40] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," *Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing*, vol. 13, 2012.

[41] L. Dagum and R. Menon, "OPENMP: an industry-standard api for shared-memory programming," *Computing in Science & Engineering*, vol. 5, pp. 46–55, 1998.

[42] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[43] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*, Addison-Wesley Professional, Boston, MA, USA, 1st edition, 2011.

[44] J. S. Harbour, D. Calkins, and R. Meuth, *Multi-Core Programming with CUDA and OpenCL*, Course Technology Press, Boston, MA, USA, 1st edition, 2011.

[45] D. Shreiner and T. K. O. A. W. Group, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*, Addison-Wesley Professional, New York, NY, USA, 7th edition, 2009.

[46] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107–113, 2008.

[47] M. Zaharia, R. S. Xin, P. Wendell et al., "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, pp. 56–65, 2016.

[48] R. V. Guha, D. Brickley, and S. MacBeth, "Schema.org: evolution of structured data on the web," *Queue*, vol. 13, p. 10, 2015.

[49] D. Suciu, *Information Organization and Databases*, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[50] J. P. Isson, *Unstructured Data Analytics: How to Improve Customer Acquisition, Customer Retention, and Fraud Detection and Prevention*, Wiley Publishing, New York, NY, USA, 1st edition, 2018.

[51] H. Tan and L. Chen, "An approach for fast and parallel video processing on Apache hadoop clusters," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, pp. 1–6, New York, NY, USA, 2014.

[52] C. Ryu, D. Lee, M. Jang, C. Kim, and E. Seo, "Extensible video processing framework in Apache hadoop," in *Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, pp. 305–310, New York, NY, USA, 2013.

[53] M. Husain, A. K. Sabarad, H. Hebballi, S. M. Nagaralli, and S. Shetty, "Counting occurrences of textual words in lecture video frames using Apache hadoop framework," in *Proceedings of the 2015 IEEE International Advance Computing Conference (IACC)*, pp. 1144–1147, London, UK, 2015.

[54] H. Tan and L. Chen, "An approach for fast and parallel video processing on Apache hadoop clusters," in *Proceedings of the 2014 IEEE International Conference on Multimedia and Expo (ICME)*, pp. 1–6, London, UK, 2014.

[55] U. S. N. Raju, S. George, V. S. Praneeth, R. Deo, and P. Jain, "Content based image retrieval on hadoop framework," in *Proceedings of the 2015 IEEE International Congress on Big Data*, pp. 661–664, New Jearsey, NJ, USA, 2015.

[56] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 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*, F. Sun, D. Hu, and H. Liu, Eds., pp. 253–260, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[58] 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*, ACM, New York, NY, USA, pp. 260–269, 2008.

[59] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MAPCG: Writing Parallel Program Portable between CPU and GPU," in *Proceedings of the 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 217–226, New York, NY, USA, 2010.

[60] M. Elteir, H. Lin, W. Feng, and T. Scogland, "Streammr: an optimized mapreduce framework for amd GPUS," in *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pp. 364–371, New York, NY, USA, 2011.

[61] J. A. Stuart and J. D. Owens, "Multi-GPU mapreduce on GPU clusters," in *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium*, pp. 1068–1079, Berlin, Germany, 2011.

[62] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10*, IEEE Computer Society, Washington, DC, USA, pp. 1–10, 2010.

[63] I. Uddin, A. Baig, and A. A. Minhas, "A controlled environment model for dealing with smart phone addiction," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 9, 2018.

[64] S. A. A. Shah, I. Uddin, F. Aziz, S. Ahmad, M. A. Al-Khasawneh, and M. Sharaf, "An enhanced deep neural network for predicting workplace absenteeism," *Complexity*, vol. 2020, pp. 1–12, 2020.